ТРЕТЬЕ ИЗДАНИЕ

ПРОГРАММИРОВАНИЕ
И ВНУТРЕННЕЕ УСТРОЙСТВО

ИГРОВОЙ ДВИЖОК

Джейсон Грегори

Game Engine Architecture

Third Edition

Jason Gregory

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

AN A K PETERS BOOK

ТРЕТЬЕ ИЗДАНИЕ

Джейсон Грегори

ИГРОВОЙ ДВИЖОК

ПРОГРАММИРОВАНИЕ И ВНУТРЕННЕЕ УСТРОЙСТВО

ПИТЕР

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.973.23-018.2
УДК 004.9
Г79

Грегори Джейсон

Г79 Игровой движок. Программирование и внутреннее устройство. Третье издание. — СПб.: Питер, 2021. — 1136 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1134-3

Книга Джейсона Грегори не случайно является бестселлером. Двадцать лет работы автора над первоклассными играми в Midway, Electronic Arts и Naughty Dog позволяют поделиться знаниями о теории и практике разработки ПО для игрового движка. Игровое программирование — сложная и огромная тема, охватывающая множество вопросов.

Граница между игровым движком и игрой размыта. В этой книге основное внимание уделено движку, основным низкоуровневым системам, системам разрешения коллизий, симуляции физики, анимации персонажей, аудио, а также базовому слою геймплея, включающему объектную модель игры, редактор мира, системы событий и скриптинга.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.23-018.2
УДК 004.9

ISBN 978-1138035454 англ.
ISBN 978-5-4461-1134-3

Краткое содержание

Предисловие 26

Новое в третьем издании 28

Благодарности 30

Часть I. Основы

Глава 1. Введение 34

Глава 2. Полезные инструменты 100

Глава 3. Основы разработки игрового ПО 135

Глава 4. Параллелизм и параллельное программирование 226

Глава 5. 3D-математика для игр 373

Часть II. Низкоуровневые системы движка

Глава 6. Системы поддержки движка 428

Глава 7. Ресурсы и файловая система 486

Глава 8. Игровой цикл и симуляция в реальном времени 526

Глава 9. Устройства HID 559

Глава 10. Инструменты для отладки и разработки 586

Часть III. Графика, движение и звук

Глава 11. Движок рендеринга 614

Глава 12. Системы анимации 716

Глава 13. Столкновения и динамика твердого тела 807

Глава 14. Звук 896

Часть IV. Игровой процесс

Глава 15. Введение в системы игрового процесса 992

Глава 16. Системы организации игрового процесса на этапе выполнения 1014

Часть V. Подведение итогов

Глава 17. Хотите сказать, что это еще не все? 1126

Список литературы 1130

Оглавление

Предисловие 26
От издательства 27

Новое в третьем издании 28

Благодарности 30

Часть I. Основы

Глава 1. Введение 34
1.1. Структура типичной игровой команды 35
1.1.1. Разработчики 36
1.1.2. Специалисты творческих профессий 36
1.1.3. Геймдизайнеры 37
1.1.4. Продюсеры 38
1.1.5. Другой персонал 38
1.1.6. Издатели и студии 38
1.2. Что такое игра 39
1.2.1. Видеоигры как мягкая симуляция реального времени 39
1.3. Что такое игровой движок 41
1.4. Различия движков для разных жанров 43
1.4.1. Шутеры от первого лица 44
1.4.2. Платформеры и другие игры от третьего лица 46
1.4.3. Файтинги 47

1.4.4. Гонки ... 49
1.4.5. Игры-стратегии ... 51
1.4.6. Многопользовательские онлайн-игры ... 52
1.4.7. Контент, созданный игроком ... 54
1.4.8. Виртуальная, дополненная и смешанная реальность ... 56
1.4.9. Другие жанры ... 60
1.5. Обзор игровых движков ... 60
1.5.1. Семейство движков Quake ... 60
1.5.2. Движок Unreal ... 61
1.5.3. Source — движок Half-Life ... 62
1.5.4. DICE Frostbite ... 62
1.5.5. RAGE ... 62
1.5.6. CRYENGINE ... 63
1.5.7. Sony PhyreEngine ... 63
1.5.8. Microsoft XNA Game Studio ... 63
1.5.9. Unity ... 64
1.5.10. Другие коммерческие игровые движки ... 64
1.5.11. Внутренние движки, находящиеся в частной собственности ... 65
1.5.12. Движки с открытым исходным кодом ... 65
1.5.13. Игровые движки 2D для непрограммистов ... 66
1.6. Архитектура среды выполнения движка ... 67
1.6.1. Целевое аппаратное обеспечение ... 67
1.6.2. Драйверы устройств ... 67
1.6.3. Операционная система ... 70
1.6.4. Сторонние SDK и промежуточное ПО ... 70
1.6.5. Уровень независимости от платформы ... 73
1.6.6. Основные системы ... 74
1.6.7. Управление ресурсами ... 75
1.6.8. Движок рендеринга ... 75
1.6.9. Инструменты профилирования и отладки ... 79
1.6.10. Столкновения и физика ... 81
1.6.11. Анимация ... 82
1.6.12. Устройства ввода ... 83
1.6.13. Аудио ... 83
1.6.14. Многопользовательская онлайн-игра (игра по сети) ... 84

1.6.15. Система основного геймплея ... 85
1.6.16. Специфические для игры подсистемы ... 88
1.7. Инструменты и конвейер ресурсов ... 88
1.7.1. Инструмент создания контента ... 92
1.7.2. Конвейер подготовки ресурсов ... 92
1.7.3. Редактор мира ... 95
1.7.4. База данных ресурсов ... 96
1.7.5. Некоторые подходы к архитектуре инструментов ... 97

Глава 2. Полезные инструменты ... 100
2.1. Контроль версий ... 100
2.1.1. Зачем использовать контроль версий ... 101
2.1.2. Распространенные системы контроля версий ... 101
2.1.3. Обзор Subversion и TortoiseSVN ... 102
2.1.4. Настройка репозитория с кодом ... 103
2.1.5. Установка TortoiseSVN ... 103
2.1.6. Версии файлов, обновление и коммиты ... 105
2.1.7. Многократная синхронизация, ветвление и слияние ... 107
2.1.8. Удаление файлов ... 109
2.2. Компиляторы, компоновщики и IDE ... 109
2.2.1. Исходные файлы, заголовки и единицы компиляции ... 110
2.2.2. Библиотеки, исполняемые файлы и динамически компонуемые библиотеки ... 110
2.2.3. Проекты и решения ... 111
2.2.4. Конфигурации сборки ... 112
2.2.5. Отладка кода ... 122
2.3. Инструменты профилирования ... 130
2.3.1. Список профилировщиков ... 132
2.4. Утечка и нарушение целостности памяти ... 132
2.5. Другие инструменты ... 133

Глава 3. Основы разработки игрового ПО ... 135
3.1. Обзор C++ и лучшие практики ... 135
3.1.1. Краткий обзор ООП ... 136
3.1.2. Стандартизация языка C++ ... 142
3.1.3. Стандарты программирования: для чего и сколько ... 147

3.2. Поиск и обработка ошибок 148
3.2.1. Типы ошибок 148
3.2.2. Обработка ошибок 149
3.2.3. Реализация обнаружения ошибок и их обработки 151
3.3. Данные, код и схема памяти 159
3.3.1. Числовые представления 159
3.3.2. Примитивные типы данных 165
3.3.3. Килобайт или кибибайт 170
3.3.4. Объявления, определения и компоновка 171
3.3.5. Структура памяти программы C/C++ 177
3.3.6. Переменные-члены 182
3.3.7. Расположение объектов в памяти 184
3.4. Основы аппаратного обеспечения компьютеров 191
3.4.1. Обучение на более простых компьютерах прошлых лет 191
3.4.2. Архитектура компьютера 192
3.4.3. Центральный процессор 193
3.4.4. Частота 197
3.4.5. Память 198
3.4.6. Шины 199
3.4.7. Машинный и ассемблерный языки 201
3.5. Архитектура памяти 207
3.5.1. Сопоставление памяти 207
3.5.2. Виртуальная память 209
3.5.3. Архитектуры памяти для уменьшения задержки 213
3.5.4. Иерархии кэш-памяти 215
3.5.5. Неоднородный доступ к памяти 222

Глава 4. Параллелизм и конкурентное программирование 226
4.1. Определение конкурентности и параллелизма 227
4.1.1. Конкурентное выполнение 227
4.1.2. Параллелизм 228
4.1.3. Параллелизм данных и задач 229
4.1.4. Таксономия Флинна 230
4.1.5. Ортогональность конкурентных вычислений и параллелизма 233
4.1.6. Вопросы, рассматриваемые в главе 233

4.2. Неявный параллелизм ... 233
4.2.1. Конвейерная обработка ... 234
4.2.2. Задержка и пропускная способность ... 236
4.2.3. Глубина конвейера ... 236
4.2.4. Потеря скорости конвейера ... 237
4.2.5. Зависимости данных ... 237
4.2.6. Зависимости ветвления ... 239
4.2.7. Суперскалярные процессоры ... 243
4.2.8. Очень длинные командные слова ... 245
4.3. Явный параллелизм ... 247
4.3.1. Гиперпоточность ... 247
4.3.2. Многоядерные процессоры ... 248
4.3.3. Симметричная и асимметричная многопроцессорная обработка ... 250
4.3.4. Распределенные вычисления ... 251
4.4. Основы операционной системы ... 252
4.4.1. Ядро ... 252
4.4.2. Прерывания ... 254
4.4.3. Вызовы ядра ... 255
4.4.4. Вытесняющая многозадачность ... 256
4.4.5. Процессы ... 257
4.4.6. Потоки ... 262
4.4.7. Фиберы ... 271
4.4.8. Потоки пользовательского уровня и корутины ... 273
4.4.9. Что еще почитать о процессах и потоках ... 276
4.5. Введение в параллельное программирование ... 277
4.5.1. Зачем писать параллельное программное обеспечение ... 277
4.5.2. Модели параллельного программирования ... 278
4.5.3. Состояние гонки ... 279
4.5.4. Критические операции и атомарность ... 282
4.6. Примитивы синхронизации потоков ... 287
4.6.1. Мьютексы ... 288
4.6.2. Критически важные секции ... 290
4.6.3. Переменные условия ... 291
4.6.4. Семафоры ... 295
4.6.5. События Windows ... 299

4.7. Проблемы с параллелизмом на основе блокировки 300
4.7.1. Взаимная блокировка 300
4.7.2. Динамическая взаимная блокировка 302
4.7.3. Ресурсное голодание 303
4.7.4. Инверсия приоритетов 303
4.7.5. Обедающие философы 304
4.8. Несколько лучших практик параллелизма 305
4.8.1. Правила глобального порядка 305
4.8.2. Алгоритмы на основе транзакций 306
4.8.3. Минимизация раздоров 307
4.8.4. Безопасность потоков 307
4.9. Параллелизм без блокировок 308
4.9.1. Причины ошибок в гонке данных 310
4.9.2. Реализация атомарности 311
4.9.3. Барьеры 319
4.9.4. Семантика упорядочения памяти 322
4.9.5. Атомарные переменные 332
4.9.6. Параллельность в интерпретируемых языках программирования 335
4.9.7. Спин-блокировки 336
4.9.8. Транзакции без блокировок 344
4.9.9. Связанный список без блокировки 345
4.9.10. Дополнительная литература по программированию без блокировки 346
4.10. SIMD/векторная обработка 347
4.10.1. Набор инструкций SSE и его регистры 348
4.10.2. Использование SSE для векторизации цикла 352
4.10.3. Векторизованное скалярное произведение 353
4.10.4. Векторно-матричное умножение с помощью SSE 358
4.10.5. Матрица-матричное умножение с SSE 358
4.10.6. Обобщенная векторизация 359
4.10.7. Предикация векторов 360
4.11. Введение в программирование GPGPU 363
4.11.1. Параллельные вычисления данных 364
4.11.2. Вычислительные ядра 365

4.11.3. Выполнение ядра 367
4.11.4. Потоки GPU и группы потоков 368
4.11.5. Дополнительная литература 372

Глава 5. 3D-математика для игр 373
5.1. Решение 3D-задач в 2D 373
5.2. Точки и векторы 374
5.2.1. Точки и декартовы координаты 374
5.2.2. Левосторонние и правосторонние системы координат 375
5.2.3. Векторы 376
5.2.4. Векторные операции 377
5.2.5. Линейная интерполяция точек и векторов 388
5.3. Матрицы 388
5.3.1. Умножение матриц 389
5.3.2. Представление точек и векторов в виде матриц 390
5.3.3. Единичная матрица 391
5.3.4. Инвертирование матриц 391
5.3.5. Транспонирование 391
5.3.6. Однородная система координат 392
5.3.7. Матрицы базовых преобразований 394
5.3.8. Матрицы 4×3 397
5.3.9. Координатное пространство 397
5.3.10. Переход к новому базису 401
5.3.11. Преобразование векторов нормали 404
5.3.12. Хранение матриц в памяти 404
5.4. Кватернионы 406
5.4.1. Единичные кватернионы как трехмерные вращения 407
5.4.2. Операции с кватернионами 408
5.4.3. Вращение векторов через кватернионы 409
5.4.4. Кватернионно-матричная эквивалентность 410
5.4.5. Линейная интерполяция вращения 412
5.5. Сравнение представлений вращения 414
5.5.1. Углы Эйлера 415
5.5.2. Матрицы 3×3 415
5.5.3. Ось + угол 416

5.5.4. Кватернионы 416
5.5.5. Преобразования SRT 416
5.5.6. Двойные кватернионы 417
5.5.7. Вращения и степени свободы 418
5.6. Другие полезные математические объекты 419
5.6.1. Прямые, лучи и отрезки 419
5.6.2. Сферы 420
5.6.3. Плоскости 420
5.6.4. Параллельные осям ограничивающие параллелепипеды 422
5.6.5. Ориентированные ограничивающие параллелепипеды 422
5.6.6. Усеченная пирамида 423
5.6.7. Выпуклые многогранные области 424
5.7. Генерация случайных чисел 424
5.7.1. Линейные конгруэнтные генераторы 424
5.7.2. Вихрь Мерсенна 425
5.7.3. Мать всего, Xorshift и KISS99 425
5.7.4. PCG 426

Часть II. Низкоуровневые системы движка

Глава 6. Системы поддержки движка 428
6.1. Подсистема запуска и остановки 428
6.1.1. Порядок статической инициализации C++ (или его отсутствие) 428
6.1.2. Простой работающий подход 431
6.1.3. Некоторые примеры реальных движков 433
6.2. Управление памятью 436
6.2.1. Оптимизация динамического распределения памяти 436
6.2.2. Фрагментация памяти 446
6.3. Контейнеры 450
6.3.1. Операции с контейнерами 452
6.3.2. Итераторы 452
6.3.3. Алгоритмическая сложность 454
6.3.4. Создание пользовательских контейнерных классов 455
6.3.5. Динамические массивы и выделение памяти фрагментами 459
6.3.6. Словари и хеш-таблицы 460

6.4. Строки ... 464
6.4.1. Проблема со строками ... 464
6.4.2. Классы строк ... 465
6.4.3. Уникальные идентификаторы ... 466
6.4.4. Локализация ... 469
6.5. Конфигурация движка ... 479
6.5.1. Параметры загрузки и сохранения ... 479
6.5.2. Параметры для каждого пользователя ... 480
6.5.3. Управление конфигурацией в некоторых реальных движках ... 481

Глава 7. Ресурсы и файловая система ... 486
7.1. Файловая система ... 487
7.1.1. Имена файлов и пути к ним ... 487
7.1.2. Базовый файловый ввод/вывод ... 491
7.1.3. Асинхронный файловый ввод/вывод ... 493
7.2. Менеджер ресурсов ... 497
7.2.1. Автономное управление ресурсами и цепочка инструментов ... 498
7.2.2. Управление ресурсами среды выполнения ... 507

Глава 8. Игровой цикл и симуляция в реальном времени ... 526
8.1. Цикл рендеринга ... 526
8.2. Цикл игры ... 527
8.2.1. Простой пример: пинг-понг ... 527
8.3. Архитектурные стили цикла игры ... 529
8.3.1. Конвейер сообщений Windows ... 529
8.3.2. Фреймворки на основе обратных вызовов ... 530
8.3.3. Обновление на основе событий ... 531
8.4. Абстрактные временные шкалы ... 532
8.4.1. Реальное время ... 532
8.4.2. Игровое время ... 532
8.4.3. Локальное и глобальное время ... 533
8.5. Измерение времени и работа с ним ... 534
8.5.1. Частота смены кадров и время ... 534
8.5.2. От частоты кадров к скорости ... 535
8.5.3. Измерение реального времени с помощью таймера высокого разрешения ... 539

8.5.4. Единицы измерения времени и переменные часов 540
8.5.5. Работа с точками останова 543
8.6. Многопроцессорные игровые циклы 544
8.6.1. Разложение задания 544
8.6.2. Один поток на подсистему 545
8.6.3. Разбиение/сборка 546
8.6.4. Система заданий 549

Глава 9. Устройства HID 559
9.1. Виды HID-устройств 559
9.2. Взаимодействие с HID-устройствами 561
9.2.1. Опрашивание 561
9.2.2. Прерывания 562
9.2.3. Беспроводные устройства 562
9.3. Виды элементов управления 562
9.3.1. Цифровые кнопки 562
9.3.2. Аналоговые оси и кнопки 564
9.3.3. Относительные оси 565
9.3.4. Акселерометр 565
9.3.5. Положение в трехмерном пространстве при использовании Wiimote или DualShock 566
9.3.6. Камеры 566
9.4. Виды вывода 569
9.4.1. Вибрация 569
9.4.2. Силовая обратная связь 569
9.4.3. Звук 569
9.4.4. Другие разновидности ввода и вывода 569
9.5. Системы игрового движка для работы с HID-устройствами 570
9.5.1. Типичные требования 570
9.5.2. Мертвые зоны 571
9.5.3. Фильтрация аналоговых сигналов 571
9.5.4. Обнаружение событий ввода 573
9.5.5. Управление несколькими HID-устройствами для нескольких игроков 580
9.5.6. Межплатформенные HID-системы 580
9.5.7. Переназначение элементов управления 582

9.5.8. Контекстный ввод 583
9.5.9. Отключение элементов управления 584
9.6. HID-устройства на практике 585

Глава 10. Инструменты для отладки и разработки 586
10.1. Журналирование и трассировка 586
10.1.1. Форматированный вывод с помощью функции OutputDebugString() 587
10.1.2. Уровень детализации 588
10.1.3. Каналы 589
10.1.4. Копирование вывода в файл 590
10.1.5. Отчеты о сбоях 590
10.2. Средства отладочной отрисовки 591
10.2.1. API для отладочной отрисовки 593
10.3. Внутриигровые меню 597
10.4. Внутриигровая консоль 598
10.5. Отладочные камеры и остановка игры 599
10.6. Читы 600
10.7. Снимки экрана и запись видео 601
10.8. Внутриигровое профилирование 602
10.8.1. Иерархическое профилирование 603
10.8.2. Экспорт в Excel 608
10.9. Внутриигровые показатели использования памяти и обнаружение утечек 609

Часть III. Графика, движение и звук

Глава 11. Движок рендеринга 614
11.1. Основы растеризации треугольников с буферизацией глубины 614
11.1.1. Описание сцены 616
11.1.2. Описание визуальных свойств поверхности 626
11.1.3. Основы освещения 640
11.1.4. Виртуальная камера 649
11.2. Конвейер рендеринга 660
11.2.1. Общая схема конвейера рендеринга 661
11.2.2. Инструментальный этап 663
11.2.3. Этап подготовки ресурсов 665

11.2.4. Конвейер графического процессора 666
11.2.5. Программируемые шейдеры 671
11.2.6. Сглаживание 677
11.2.7. Этап приложения 680
11.3. Продвинутые методы освещения и глобальное освещение 691
11.3.1. Освещение на основе изображения 691
11.3.2. Освещение с расширенным динамическим диапазоном 695
11.3.3. Глобальное освещение 696
11.3.4. Отложенный рендеринг 703
11.3.5. Физически корректное затенение 704
11.4. Визуальные эффекты и наложения 705
11.4.1. Эффекты частиц 705
11.4.2. Декали 706
11.4.3. Эффекты окружения 707
11.4.4. Наложения 710
11.4.5. Гамма-коррекция 712
11.4.6. Полноэкранные постэффекты 714
11.5. Дополнительная литература 714

Глава 12. Системы анимации 716
12.1. Разновидности анимации персонажей 716
12.1.1. Келевая анимация 717
12.1.2. Жесткая иерархическая анимация 717
12.1.3. Повершинная анимация и морфинг-мишени 718
12.1.4. Скиновая анимация 720
12.1.5. Методы анимации как методы сжатия данных 721
12.2. Скелеты 722
12.2.1. Скелетная иерархия 723
12.2.2. Представление скелета в памяти 723
12.3. Позы и положения 724
12.3.1. Поза привязки 725
12.3.2. Локальные позы 725
12.3.3. Глобальные позы 728
12.4. Клипы 729
12.4.1. Локальная временная шкала 730
12.4.2. Глобальная временная шкала 734

12.4.3. Сравнение глобального и локального таймеров ... 736
12.4.4. Простой формат анимационных данных ... 739
12.4.5. Непрерывные канальные функции ... 740
12.4.6. Метаканалы ... 741
12.4.7. Отношения между мешами, скелетами и клипами ... 742
12.5. Скининг и генерация палитры матриц ... 744
12.5.1. Сведения о скининге для отдельной вершины ... 744
12.5.2. Математический аспект скининга ... 745
12.6. Слияние анимации ... 749
12.6.1. Линейная интерполяция ... 749
12.6.2. Способы применения слияния методом LERP ... 751
12.6.3. Сложные слияния методом LERP ... 756
12.6.4. Частичное скелетное слияние ... 760
12.6.5. Аддитивное слияние ... 761
12.6.6. Способы применения аддитивного слияния ... 764
12.7. Постобработка ... 766
12.7.1. Процедурная анимация ... 766
12.7.2. Инверсная кинематика ... 767
12.7.3. Тряпичные куклы ... 769
12.8. Методы сжатия ... 769
12.8.1. Пропуск каналов ... 769
12.8.2. Квантование ... 770
12.8.3. Частота дискретизации и пропуск семплов ... 774
12.8.4. Сжатие на основе кривых ... 774
12.8.5. Сжатие с использованием вейвлетов ... 775
12.8.6. Выборочные загрузка и потоковая передача ... 775
12.9. Конвейер анимации ... 775
12.10. Конечные автоматы действий ... 778
12.10.1. Подход с плоским средним взвешенным ... 779
12.10.2. Деревья слияния ... 783
12.10.3. Параметры состояния и дерева слияния ... 787
12.10.4. Переходы ... 792
12.10.5. Управляющие параметры ... 795
12.11. Ограничения ... 797
12.11.1. Крепления ... 797
12.11.2. Выравнивание объектов ... 798

12.11.3. Инверсная кинематика движений захвата рукой 802
12.11.4. Извлечение движений и инверсная кинематика ног 803
12.11.5. Другие виды ограничений 805

Глава 13. Столкновения и динамика твердого тела 807
13.1. Нужна ли в вашей игре физика? 808
13.1.1. Что можно делать с системой симуляции физики 808
13.1.2. Делает ли физика игру интересной? 809
13.1.3. Влияние физики на игру 810
13.2. Промежуточный слой для столкновений/физики 813
13.2.1. ODE 813
13.2.2. Bullet 813
13.2.3. TrueAxis 814
13.2.4. PhysX 814
13.2.5. Havok 814
13.2.6. Physics Abstraction Layer (PAL) 814
13.2.7. Digital Molecular Matter 815
13.3. Система обнаружения столкновений 815
13.3.1. Элементы, которые могут сталкиваться между собой 816
13.3.2. Мир столкновений/физики 817
13.3.3. Концепции геометрических форм 819
13.3.4. Примитивы столкновения 820
13.3.5. Проверки на столкновение и аналитическая геометрия 825
13.3.6. Оптимизация производительности 834
13.3.7. Запросы о столкновениях 836
13.3.8. Фильтрация столкновений 840
13.4. Динамика твердого тела 842
13.4.1. Некоторые основы 843
13.4.2. Линейная динамика 845
13.4.3. Решение уравнений движения 847
13.4.4. Численное интегрирование 849
13.4.5. Угловая динамика в двухмерном пространстве 853
13.4.6. Угловая динамика в трехмерном пространстве 857
13.4.7. Реакция на столкновение 862
13.4.8. Ограничения 869
13.4.9. Управление движениями твердых тел 873
13.4.10. Отдельный шаг симуляции столкновений/физики 875

13.5. Интеграция физического движка в игру 877
13.5.1. Связывание игровых объектов и твердых тел 877
13.5.2. Обновление симуляции 882
13.5.3. Пример использования систем столкновений и физики в игре 885
13.6. Расширенные физические возможности 894

Глава 14. Звук 896
14.1. Физика звука 897
14.1.1. Свойства звуковых волн 897
14.1.2. Субъективная громкость звука и децибел 899
14.1.3. Распространение звуковой волны 901
14.1.4. Восприятие положения в пространстве 907
14.2. Математика звука 908
14.2.1. Сигналы 908
14.2.2. Преобразование сигналов 909
14.2.3. Линейные стационарные системы 910
14.2.4. Импульсная характеристика систем ЛСС 911
14.2.5. Частотная область и преобразование Фурье 917
14.3. Аудиотехнологии 925
14.3.1. Аналоговые аудиотехнологии 925
14.3.2. Цифровые аудиотехнологии 930
14.4. Рендеринг звука в 3D 937
14.4.1. Краткий обзор процесса рендеринга 3D-звука 938
14.4.2. Моделирование мира звука 938
14.4.3. Затухание в зависимости от расстояния 939
14.4.4. Панорамирование 941
14.4.5. Распространение, реверберация и акустика 947
14.4.6. Эффект Доплера 954
14.5. Архитектура звукового движка 955
14.5.1. Конвейер обработки звуков 956
14.5.2. Концепции и терминология 959
14.5.3. Голосовая шина 960
14.5.4. Главный микшер 962
14.5.5. Главная шина вывода 964
14.5.6. Реализация шины 965
14.5.7. Управление ресурсами 967

14.5.8. Микширование в игре 969
14.5.9. Обзор звуковых движков 973
14.6. Звуковые возможности, характерные для разных видов игр 976
14.6.1. Поддержка разделенного экрана 977
14.6.2. Система диалогов 977
14.6.3. Музыка 989

Часть IV. Игровой процесс

Глава 15. Введение в системы игрового процесса 992
15.1. Структура игрового мира 993
15.1.1. Элементы мира 993
15.1.2. Области игрового мира 995
15.1.3. Высокоуровневый игровой поток 997
15.2. Реализация динамических элементов: игровые объекты 997
15.2.1. Объектные модели игры 999
15.2.2. Игровые объекты на этапах проектирования и выполнения 1000
15.3. Игровые движки на основе данных 1001
15.4. Редактор игрового мира 1002
15.4.1. Типичные возможности редактора игрового мира 1004
15.4.2. Интегрированные средства управления ресурсами 1011

Глава 16. Системы организации игрового процесса на этапе выполнения 1014
16.1. Компоненты системы организации игрового процесса 1014
16.2. Архитектуры объектной модели времени выполнения 1017
16.2.1. Архитектуры объектного стиля 1018
16.2.2. Архитектуры на основе свойств 1031
16.3. Форматы областей игрового мира 1035
16.3.1. Двоичные образы объектов 1036
16.3.2. Описание сериализованных игровых объектов 1037
16.3.3. Спаунеры и схемы типов 1038
16.4. Загрузка и потоковая передача игровых миров 1042
16.4.1. Простая загрузка уровней 1043
16.4.2. Шаг в направлении бесшовной загрузки: шлюзы 1044
16.4.3. Потоковая загрузка игрового мира 1045
16.4.4. Управление памятью для создания объектов 1047
16.4.5. Сохраненные игры 1049

16.5. Ссылки на объекты и запросы к игровому миру ... 1051
16.5.1. Указатели ... 1051
16.5.2. Умные указатели ... 1052
16.5.3. Дескрипторы ... 1054
16.5.4. Запросы игровых объектов ... 1056
16.6. Обновление игровых объектов в реальном времени ... 1058
16.6.1. Простой подход (который не работает) ... 1059
16.6.2. Влияние пакетных обновлений на производительность ... 1061
16.6.3. Взаимные зависимости объектов и подсистем ... 1064
16.7. Применение конкурентности к обновлению игровых объектов ... 1071
16.7.1. Конкурентные подсистемы движка ... 1072
16.7.2. Асинхронный подход к проектированию ... 1073
16.7.3. Зависимости заданий и степень параллелизма ... 1075
16.7.4. Распараллеливание самой объектной модели игры ... 1078
16.8. События и передача сообщений ... 1083
16.8.1. Проблема со статически типизированным связыванием функций ... 1083
16.8.2. Инкапсуляция события в виде объекта ... 1085
16.8.3. Типы событий ... 1086
16.8.4. Аргументы событий ... 1087
16.8.5. Обработчики событий ... 1089
16.8.6. Распаковка аргументов событий ... 1090
16.8.7. Цепочки обязанностей ... 1090
16.8.8. Подписка на события ... 1092
16.8.9. Использовать ли очередь ... 1093
16.8.10. Некоторые проблемы с незамедлительной отправкой событий ... 1098
16.8.11. Системы передачи событий/сообщений на основе данных ... 1099
16.9. Скрипты ... 1102
16.9.1. Для выполнения кода и описания данных ... 1103
16.9.2. Характеристики языков программирования ... 1103
16.9.3. Некоторые распространенные игровые скриптовые языки ... 1105
16.9.4. Архитектуры для скриптования ... 1109
16.9.5. Возможности игровых скриптовых языков на этапе выполнения ... 1111
16.10. Высокоуровневый игровой поток ... 1123

Часть V. Подведение итогов

Глава 17. Хотите сказать, что это еще не все? ... 1126
17.1. Некоторые подсистемы движка, которые мы не рассмотрели ... 1126
17.1.1. Видеоплеер ... 1126
17.1.2. Сетевые возможности для многопользовательских игр ... 1127
17.2. Системы игрового процесса ... 1127
17.2.1. Игровая механика ... 1127
17.2.2. Камеры ... 1127
17.2.3. Искусственный интеллект ... 1128
17.2.4. Другие системы игрового процесса ... 1129

Список литературы ... 1130

Посвящается Трине, Эвану и Куинн Грегори.
Памяти наших героев — Джойса Остеруса,
Кеннета и Эрики Грегори.

Предисловие

Я рад приветствовать вас на страницах книги «Игровой движок. Программирование и внутреннее устройство». Эта книга нацелена на всестороннее изучение главных компонентов типичного коммерческого игрового движка. Игровое программирование — обширная тема, поэтому мы должны обсудить множество вопросов. Тем не менее я уверен, что вы обнаружите: глубина нашего обсуждения достаточна для того, чтобы получить четкое понимание как теории, так и общих практик, применяемых во всех рассматриваемых дисциплинах разработки. Я хочу сказать, что данная книга — только начало захватывающего путешествия, возможно, длиною в жизнь. Богатая информация доступна во всех аспектах игровых технологий, и текст книги закладывает основы знаний и служит отправной точкой для дальнейшего изучения.

Мы с вами сфокусируем внимание на технологиях игровых движков и их архитектуре. Это означает, что мы раскроем теорию, лежащую в основе подсистем, из которых состоит коммерческий игровой движок, структуры данных, алгоритмы и интерфейсы программного обеспечения, которые часто используются для их реализации. А также узнаем, как эти подсистемы функционируют совместно внутри игрового движка в целом. Граница между игровым движком и игрой довольно сильно размыта. Мы сосредоточимся непосредственно на движке, а также на основных низкоуровневых системах, системах разрешения коллизий, симуляции физики, анимации персонажа, аудио, а также глубже изучим то, что я называю слоем основы геймплея. Данный слой включает объектную модель игры, редактор мира, системы событий и скриптинга. Мы также коснемся некоторых аспектов программирования геймплея, включая механику игры, камеры и искусственный интеллект (ИИ). Однако обсуждение этих вопросов будет ограничено в основном способами взаимодействия интерфейсов систем геймплея с движком.

Книга предназначена для изучения курса промежуточного игрового программирования в высших учебных заведениях. Она также будет полезна разработчикам — любителям программного обеспечения, людям, увлекающимся программированием, игровым разработчикам-самоучкам, а также всем занятым в игровой индустрии. Начинающие разработчики с помощью этого издания лучше поймут игровую математику, архитектуру движка и игровые технологии. Специалисты, сосредоточившиеся на одной конкретной специальности, могут извлечь пользу, получив более обширную картину, прочитав всю книгу.

Чтобы лучше разобраться в материале, желательно иметь понятие об основах объектно-ориентированного программирования (ООП), а также минимальный опыт программирования на языке C++. Игровая индустрия использует широкий спектр языков программирования, однако большинство наиболее мощных игровых 3D-движков по-прежнему пишут на C++. Мы рассмотрим основные постулаты ООП в главе 3, и во время прочтения книги вы, без сомнения, найдете несколько новых хитростей по C++, но твердую основу языка C++ лучше всего получить из книг [36], [37] и [46]. Если ваши познания в C++ несколько устарели, то, чтобы освежить их перед изучением данного текста, рекомендую обратиться к этим или похожим книгам. Если у вас нет опыта работы с C++, попробуйте прочесть по крайней мере первые главы [46] и/или поработайте с несколькими онлайн-руководствами по C++, прежде чем приступить к данному изданию.

Лучшим способом изучения компьютерного программирования любого вида является непосредственное написание кода. Я настоятельно советую во время работы с книгой выбрать несколько областей, которые представляют для вас конкретный интерес, и вернуться к ее прочтению, реализовав несколько собственных проектов в этих областях. Например, если вас занимает анимация персонажа, можете начать с установки OGRE (объектно-ориентированного графического движка с открытым исходным кодом, написанного на C++) и изучить демонстрационный пример — анимации заскиненной модели. Затем попытайтесь реализовать техники смешения анимации, описанные здесь, используя OGRE. Далее можете создать простой, управляемый с помощью геймпада анимированный персонаж, который будет бежать по пустому пространству. Когда у вас заработает что-то относительно простое, усложните задачу! Перейдите к другой области игровых технологий. Изучайте и повторяйте. Не так уж важно, какими проектами вы займетесь, когда будете упражняться в игровом программировании, — главное, не просто читать текст.

Игровые технологии — постоянно меняющаяся и развивающаяся область, ее невозможно целиком охватить на страницах книги. Поэтому на веб-сайте издания (www.gameenginebook.com) периодически будут появляться обновления, ссылки на дополнительные ресурсы, исправления опечаток, примеры кода и идеи проектов. Вы также можете подписаться на меня в Twitter: @jqgregory.

От издательства

Некоторые иллюстрации мы не имели права использовать в русском издании книги, а некоторые лучше воспринимаются в цветном варианте или максимальном увеличении. Поэтому мы дополнили книгу QR-кодами.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Новое в третьем издании

Вычислительные аппаратные средства, которые лежат в основе современных игровых консолей, мобильных устройств и персональных компьютеров, интенсивно используют *параллелизм*. В их процессорах и графических картах несколько функциональных блоков работают одновременно, применяя подход «разделяй и властвуй» для увеличения скорости вычислений. В то время как параллельные вычислительные средства могут ускорить работу традиционных однопоточных программ, программисты должны писать *конкурентное программное обеспечение*, чтобы извлечь максимум пользы из аппаратного параллелизма, который повсеместно используется в современных вычислительных платформах.

В предыдущих изданиях книги «Игровой движок. Программирование и внутреннее устройство» темы параллелизма и конкурентности затрагивались в контексте проектирования игровых движков. Однако указанным темам не уделялось то внимание, которого они заслуживают. В третьем издании книги эта проблема исправлена — добавлена новая глава, посвященная конкурентности и параллелизму. Главы 8 и 16 дополнены — в них включено детальное обсуждение того, как техники параллельного программирования обычно применяются к игровой подсистеме и обновляется игровая объектная модель, а также как система заданий общего назначения помогает полностью использовать все мощи параллельной разработки в игровом движке.

Я уже упоминал, что хороший разработчик игр должен иметь серьезные познания в C++ (в дополнение к широкому спектру других полезных языков, регулярно используемых в игровой индустрии). В моем видении, знание программистом высокоуровневых языков должно основываться на твердом понимании систем программного обеспечения и аппаратных устройств. В данном издании я расширил главу 3, включив в нее трактовку основ компьютерного аппаратного обеспечения, ассемблера и ядра операционной системы.

В этой книге также дополнены различные темы, раскрытые в предыдущих изданиях. Добавлено обсуждение локальной и глобальной оптимизации компилятора. Более подробно рассмотрены различные стандарты языка C++. Расширен раздел о кэшировании памяти и когерентности кэша. Оптимизирована глава, посвященная анимации. А также, как и во втором издании, исправлены опечатки, на которые обратили внимание вы, мои преданные читатели. Спасибо вам! Надеюсь,

вы увидите, что найденные вами ошибки исправлены. (Хотя, несомненно, вместо них появились новые, и, найдя их, не стесняйтесь сообщать об этом мне, чтобы я мог исправить их в четвертом издании книги!)

Конечно, как я говорил ранее, поле программирования игровых движков неописуемо огромно. Невозможно раскрыть все темы в одной книге. По сути, главная цель данного издания — стать инструментом построения фундамента знаний и отправной точкой для дальнейшего изучения. Я надеюсь, вы найдете его полезным в своем путешествии по захватывающей многоаспектной среде архитектуры игровых движков.

Благодарности

Ни одна книга не создается в вакууме, и эта не исключение. Она — и ее третье издание, которое вы сейчас держите в руках, — была бы невозможна без моей семьи, друзей и коллег по игровой индустрии. И я хотел бы горячо поблагодарить каждого, кто помог мне осуществить этот проект.

Конечно, больше всего в проект вовлечена семья автора. Поэтому я хотел бы начать с того, чтобы *в третий раз* сказать спасибо своей жене Трине. Она была для меня опорой во время написания оригинальной книги и оказала столь же ощутимую поддержку и неоценимую помощь при работе над вторым и третьим изданиями. Когда я сидел за клавиатурой, Трина была рядом, заботясь о двух наших мальчиках, Эване (сейчас ему 15) и Квинне (12 лет), день за днем и ночь за ночью, даже забывая о своих собственных планах, делала мою работу по дому, как и свою (чаще, чем я хотел бы признать), и всегда говорила мне добрые слова поддержки, когда они были нужны мне больше всего.

Я хотел бы особо поблагодарить редакторов первого издания Мэтта Уайтинга и Джеффа Лендера. Их проницательные, предметные и своевременные советы всегда были правильными, а их обширный опыт в игровой индустрии помог мне поверить в то, что информация, представленная на этих страницах, настолько точна и актуальна, насколько это возможно для человека. С Мэттом и Джеффом приятно работать, и я горжусь тем, что в данном проекте сотрудничал с такими признанными профессионалами. Я также хочу поблагодарить Джеффа за то, что связал меня с Элис Питерс и помог мне успешно начать этот проект. Мэтт, спасибо и тебе за то, что вновь вернулся к книге и дал мне ценные советы по поводу новой главы о конкурентности.

Множество коллег в Naughty Dog также внесли свой вклад в данную книгу, обеспечив обратную связь или помогая мне со структурой и содержанием одной из глав. Я хотел бы поблагодарить Маршалла Робина и Карлоса Гонсалеса-Окоа за то, что направляли и опекали меня во время написания главы о рендеринге, а также Пола-Кристиана Ингстада за великолепные проницательные замечания относительно содержания этой главы. Спасибо Кристиану Гирлингу за отзывы о различных разделах книги, включая главу об анимации и новую главу о параллелизме и конкурентности. Отдельно я хотел бы поблагодарить Джонатана Ланье, экстраординарного старшего разработчика аудио в Naughty Dog, за то, что предоставил мне большой объем необработанной информации, которую вы найдете

в главе об аудио, всегда был готов пообщаться со мной, когда у меня появлялись вопросы, и обеспечил лазерно точную и неоценимую обратную связь после прочтения рукописи. Я хотел бы поблагодарить одного из самых молодых членов команды разработчиков в Naughty Dog Карима Омара за понимание и очень полезные отзывы о новой главе о конкурентности. Хочу сказать спасибо и всей команде разработчиков Naughty Dog за создание тех невероятных систем игрового движка, которые я выделяю в данной книге.

Кроме того, выражаю благодарность Киту Шефферу из Electronic Arts за то, что предоставил мне большую часть необработанного контента, связанного с влиянием физики на игру (рассмотрен в разделе 13.1). Я хотел бы также горячо поблагодарить Кристофа Балестра (сопрезидента Naughty Dog в первые десять лет моей работы в этой компании), Пола Кита (был ведущим разработчиком франшизы *Medal of Honor*, когда я трудился в Electronic Arts) и Стива Рэнка (ведущего разработчика проекта *Hydro Thunder* в Midway San Diego) за то, что были моими наставниками и руководителями в эти годы. Они не принимали непосредственного участия в написании книги, однако действительно помогли мне стать тем разработчиком, которым я являюсь сейчас, и их влияние так или иначе отражено фактически на каждой странице.

Эта книга выросла из заметок, которые я разработал для курса под названием *ITP-485: Programming Game Engines*. Я четыре года вел его под покровительством Information Technology Program в Университете Южной Калифорнии. Хотел бы поблагодарить доктора Энтони Боркеза, декана факультета ITP, за то, что нанял меня для разработки учебного плана курса ITP-485.

Вся моя семья и друзья также заслуживают благодарности — отчасти за поддержку, отчасти за то, что развлекали мою жену и наших двух мальчиков в то время, когда я работал. Я бы хотел поблагодарить свояченицу и шурина, Трэйси Ли и Дага Провинса, кузена жены Мэтта Гленна и всех наших потрясающих друзей, включая Ким и Дрю Кларк, Шерилин и Джима Критцер, Энни и Майкла Шерер, Ким и Майка Уорнер и Кендру и Энди Уолтер. Когда я был подростком, мой отец Кеннет Грегори написал книгу *Extraordinary Stock Profits*, об инвестировании в фондовый рынок. Этим он вдохновил меня на написание данной книги. За это и многое другое я вечно ему благодарен. Я также хотел бы поблагодарить свою мать Эрику Грегори за ее настойчивость, благодаря которой я начал этот проект, и за то, что, когда я был ребенком, она проводила со мной долгие часы, помогая мне постичь писательское мастерство. Только ей я обязан своими писательскими навыками, своим отношением к труду и странным чувством юмора!

Я хотел бы поблагодарить Элис Питерс и Кевина Джексона-Мэда, а также весь персонал издательства A K Peters за их огромную поддержку при публикации первого издания этой книги. С тех пор компания A K Peters была приобретена издательством CRC Press, основным подразделением по издательству книг по науке и технике Taylor & Francis Group. Я бы хотел пожелать Элис и Клаусу Питерс всего наилучшего в дальнейшей работе. Я также хочу поблагодарить Рика Адамса, Дженифер Арингер, Джессику Вега и Синтию Кливек из Taylor & Francis за их терпеливую поддержку и помощь в процессе создания второго и третьего изданий

«Игрового движка», Джонатана Пеннела за работу над обложкой второго издания, Скотта Шемблина за оформление третьей обложки и Брайана Хэгера (www.bionic3d.com) за то, что любезно разрешил мне использовать его красивую 3D-модель ракетного двигателя Space X Merlin на обложке третьего издания.

Я польщен тем, что первое и второе издания книги переведены на японский, китайский и корейский языки! Хочу выразить искреннюю благодарность Казухисе Минато и его команде в Namco Bandai Games за то, что взялись за грандиозную задачу перевода на японский язык и отлично поработали с обоими изданиями. Я также хотел бы поблагодарить людей из Softbank Creative, Inc. за публикацию японской версии книги. И выразить огромную благодарность Мило Йипу за тяжелую работу над китайским переводом проекта. Я искренне признателен издательству Electronics Industry за публикацию китайского издания, а также Acorn Publishing Company и Hongreung Science Publishing Co. за публикацию корейского перевода первого и второго изданий соответственно.

Многие читатели пишут мне письма и сообщают об ошибках в первом и втором изданиях. За это я хотел бы искренне поблагодарить всех, кто вносит свой вклад в улучшение книги. Хочу сказать отдельное спасибо Мило Йипу, Джо Конли и Захари Тернеру за то, что вышли за пределы своих должностных обязанностей. Все трое прислали мне объемные документы с перечнем множества опечаток и невероятно ценными и проницательными предложениями. Я изо всех сил старался учесть все это в третьей редакции — пожалуйста, продолжайте в том же духе!

Джейсон Грегори, апрель 2018 года

Часть I
Основы

1 Введение

Когда у меня в 1979 году появилась первая игровая консоль — крутая система Intellivision от компании Mattel, термина «игровой движок» не существовало. Тогда аркадные и видеоигры, по мнению большинства взрослых, были не более чем игрушками, и запускающее их программное обеспечение разрабатывалось специально для конкретной игры и оборудования, на котором они запускались. Сегодня игры — это господствующая многомиллиардная индустрия, конкурирующая с Голливудом по масштабу и популярности. И программное обеспечение, которое лежит в основе этих теперь уже повсеместно распространенных трехмерных миров, называется *игровыми движками*. К ним относятся Unreal Engine 4 (компания Epic Games), Source (компания Valve), CRYENGINE® 3 (компания Crytek), Frostbite™ (компания DICE (Electronic Arts)), а также Unity. Эти игровые движки стали полнофункциональными средствами разработки программного обеспечения многократного использования, которые могут лицензироваться и применяться для разработки практически любой игры.

В то время как игровые движки значительно различаются в деталях архитектуры и реализации, благодаря как движкам с публичной лицензией, так и их коммерческим аналогам появились узнаваемые паттерны. Фактически все игровые движки содержат схожие наборы основных компонентов, включая движок рендеринга, движок коллизий и физики, объектную модель игрового мира, системы анимации, аудио, искусственного интеллекта и т. д. В каждом из этих компонентов также начинает появляться относительно небольшое количество вариантов дизайна, постепенно становящихся стандартными.

Существует огромное множество книг, которые детально описывают отдельные подсистемы игровых движков, например трехмерную графику. Другие издания освещают приемы работы и дают ценные советы во всем разнообразии областей игровых технологий. Однако я не смог найти такого, которое представляет читателю полную картину всей гаммы компонентов, составляющих современный игровой движок. Цель этой книги и состоит в том, чтобы провести читателя по обширному сложному пейзажу архитектуры игрового движка.

Прочитав эту книгу, вы узнаете:

- как спроектированы реальные мощные игровые движки;
- как организована работа команды разработки игр в реальном мире;

- какие главные подсистемы и паттерны проектирования повторяются вновь и вновь практически в каждом игровом движке;
- каковы типичные требования к любой ключевой подсистеме;
- какие подсистемы не зависят от жанра и игры, а какие разрабатываются явно для определенных жанра и игры;
- где движок обычно заканчивается и начинается собственно игра.

Вы получите непосредственное представление о внутренней работе некоторых популярных игровых движков, таких как Quake, Unreal и Unity. Также мы изучим известные пакеты промежуточного программного обеспечения, такие как библиотека физики Havok Physics, движок рендеринга OGRE и Granny 3D от компании Rad Game Tools — набор инструментов управления анимацией и геометрией. Мы исследуем множество коммерческих игровых движков, с которыми я имел удовольствие работать, включая движок Naughty Dog, разработанный для игровых серий *Uncharted* и *The Last of Us*.

Прежде чем начать, рассмотрим некоторые методы и инструменты крупномасштабной разработки программного обеспечения в контексте игровых движков, а именно:

- разницу между логической и физической архитектурой программного обеспечения;
- управление конфигурациями, контроль версий и системы сборки;
- советы и приемы работы с одной из наиболее часто используемых сред разработки для C и C++ — Microsoft Visual Studio.

Я полагаю, что вы хорошо знаете C++ (язык, который выбирает большинство современных игровых программистов) и понимаете основные принципы проектирования программного обеспечения. Кроме того, я считаю, что вы имеете представление о линейной алгебре, трехмерной векторной и матричной математике и тригонометрии (речь о некоторых ключевых понятиях пойдет в главе 5). В идеале вы должны иметь начальное представление о фундаментальных понятиях программирования систем реального времени и событийно-ориентированного программирования. Но если это не так, не переживайте — я кратко рассмотрю эти темы и укажу вам правильное направление, если вы почувствуете, что должны усовершенствовать свои навыки, прежде чем мы начнем углубленное изучение.

1.1. Структура типичной игровой команды

Прежде чем погрузиться в изучение структуры типичного игрового движка, кратко рассмотрим структуру типичной команды разработки игры. Игровые студии обычно формируются из специалистов пяти основных направлений: разработчиков, художников, геймдизайнеров, продюсеров и другого управленческого персонала и персонала поддержки (маркетинговая, правовая, информационно-техническая и технологическая поддержка, администрация и т. д.). Каждая должность может иметь различные специализации. Мы коротко поговорим о каждой из них далее.

1.1.1. Разработчики

Разработчики проектируют и создают программное обеспечение, заставляющее игру и инструменты работать. Они зачастую делятся на две группы: *разработчики среды выполнения* (работают над движком и игрой как таковой) и *разработчики инструментов* (создают офлайновые инструменты, позволяющие остальным разработчикам действовать более эффективно). Члены обеих групп имеют различные специальности. Некоторые из них сосредотачиваются на одной определенной системе движка, например рендеринге, искусственном интеллекте, аудио или коллизиях и физике, некоторые — на программировании геймплея и скриптинге. А другие предпочитают работать на системном уровне и не слишком вникают в то, как на самом деле работает игра. Некоторые разработчики — универсалы, мастера на все руки, которые могут переключиться на работу над любой проблемой, требующей решения во время разработки.

Старших разработчиков иногда просят взять на себя техническую руководящую роль. Ведущие разработчики обычно продолжают проектировать и писать код, а также помогают управлять работой команды, принимают решения относительно общего технического направления проекта и иногда даже напрямую руководят другими сотрудниками.

В некоторых компаниях также есть один или несколько технических директоров, работа которых сводится к ведению одного или нескольких проектов на высоком уровне, информированию команд о потенциально возможных технических проблемах, предстоящих событиях в индустрии, новых технологиях и т. д. Самая высокая должность, связанная с программированием в игровой студии, — CTO (если она вообще имеется). Роль CTO — быть своего рода техническим директором для всей студии и выполнять ключевую управляющую роль в компаниии.

1.1.2. Специалисты творческих профессий

В игровой индустрии говорят: «Контент — это главное». Художники создают весь визуальный и аудиоконтент игры, и качество их работы может буквально сделать или уничтожить ее. Для работы требуются художники и дизайнеры всех направлений.

- *Создатели концепт-артов* создают скетчи и зарисовки, позволяющие команде увидеть итог разработки игры. Они начинают раньше других — на стадии разработки концепции, но обычно обеспечивают визуальную направленность проекта на протяжении всего жизненного цикла. Как правило, скриншоты, сделанные в процессе игры, имеют странное сходство с произведениями концептуального искусства.
- *Разработчики 3D-моделей (3D-моделлеры)* создают трехмерную геометрию для всего в виртуальном мире игры. Сюда обычно входят разработчики моделей переднего и заднего плана. Те, кто работает над передним планом, создают объекты, персонажей, транспорт, оружие — все то, что наполняет игровой мир. Разработчики моделей заднего плана создают геометрию статичного фона — растительность, здания, мосты и т. д.

- *Художники по текстурам* создают двухмерные изображения, называемые текстурами, которые накладываются на поверхности 3D-моделей для обеспечения детализации и реализма.
- *Специалисты по освещению* создают в игровом мире источники света, как статичные, так и динамичные, работают с цветом, интенсивностью и направлением освещения, чтобы обеспечить максимальную художественность и эмоциональное воздействие каждой сцены.
- *Аниматоры* придают движущимся персонажам и объектам игры динамичность. Они буквально выступают актерами в производстве игры, как это происходит в кинопроизводстве с компьютерной графикой. Однако игровой аниматор должен иметь уникальный набор навыков, чтобы выполнить анимацию, которая будет безупречно согласовываться с технологическими особенностями игрового движка.
- *Актеры захвата движения* обычно используются для обеспечения грубого набора данных о движении, которые затем обрабатывают аниматоры, прежде чем интегрировать их в игру.
- *Звукорежиссеры* работают вместе с разработчиками, чтобы создать и смешать звуковые эффекты и музыку в игре.
- *Актеры озвучки* наделяют персонажей голосами в большинстве игр.
- Во многих играх есть один или несколько *композиторов*, сочиняющих оригинальное звуковое сопровождение для игры.

Как и разработчиков, старших художников часто призывают стать тимлидерами. Некоторые игровые команды имеют одного (или нескольких) арт-директора, который является главным художником, управляющим всей игрой и обеспечивающим упорядоченность деятельности всех членов группы.

1.1.3. Геймдизайнеры

Работа геймдизайнера заключается в разработке интерактивной составляющей пользовательского взаимодействия с игрой, называемой *геймплеем*. Различные дизайнеры работают на разных уровнях детализации. Некоторые (как правило, старшие) геймдизайнеры действуют на макроуровне, определяя сюжет игры, общую последовательность глав или уровней, а также основные цели и задачи игрока. Другие дизайнеры работают над определенными уровнями или географическими областями внутри виртуального мира игры, создавая слои геометрии статичного фона, определяя, когда и откуда появятся враги, размещая такие припасы, как оружие и восстанавливающие здоровье аптечки, разрабатывая элементы внутриигровых загадок и т. д. В то же время другие дизайнеры работают на более высоком техническом уровне, тесно взаимодействуя с разработчиками геймплея, и/или пишут код (как правило, на высокоуровневом языке скриптинга). Многие геймдизайнеры в прошлом являлись разработчиками, решившими играть более активную роль в определении того, как игра будет выглядеть.

Некоторые игровые команды нанимают одного или нескольких игровых писателей. Работа этого специалиста может варьироваться от сотрудничества со старшими геймдизайнерами в целях конструирования сюжета всей игры до написания отдельных линий диалогов.

Некоторые старшие дизайнеры играют роль менеджеров. Во многих игровых командах есть геймдиректор, работа которого заключается в наблюдении за всеми аспектами геймдизайна, помощи в управлении сроками и обеспечении того, чтобы деятельность всех дизайнеров была упорядочена на протяжении всей работы над продуктом. Старшие дизайнеры иногда вырастают в продюсеров.

1.1.4. Продюсеры

Роль продюсеров в различных студиях определяется по-разному. В одних игровых компаниях их работа заключается в управлении сроками и человеческими ресурсами. В других продюсер служит старшим геймдизайнером. В третьих он является связующим звеном между командой разработчиков и подразделениями компании (финансовыми, правовыми, маркетинговыми и т. д.). В некоторых небольших студиях вообще нет продюсеров. Например, в студии Naughty Dog буквально каждый, включая обоих сопрезидентов, играет ведущую роль в создании игры, управление командами и деловые обязанности поделены между старшими специалистами.

1.1.5. Другой персонал

Группу людей, непосредственно создающих игру, как правило, поддерживает ключевая команда технического персонала. В нее входят исполнительное руководство студии, маркетинговый отдел (или команда, которая кооперируется с внешней маркетинговой группой), административный штат и IT-отдел, работа которого состоит в том, чтобы купить, установить и настроить аппаратное и программное обеспечение и оказывать техническую поддержку сотрудникам.

1.1.6. Издатели и студии

Маркетингом, выпуском и распространением игры обычно занимается *издатель*, а не сама студия. Издатель — это, как правило, крупная корпорация, такая как Electronic Arts, THQ, Vivendi, Sony, Nintendo и др. Многие игровые студии не аффилированы с каким-то определенным издателем. Они продают производимые игры любому издателю, предложившему более выгодные условия. Другие студии работают только с определенным издателем, либо заключив с ним долгосрочный контракт, либо являясь его дочерней компанией. Например, игровые студии THQ управляются независимо от издателя THQ, но они принадлежат ему и полностью контролируются им. Издатель Electronic Arts имеет более тесные отношения со студиями и напрямую управляет ими. Разработчики первого эшелона — это игровые студии, которыми владеют непосредственно производители консолей (Sony,

Nintendo и Microsoft). Например, Naughty Dog — разработчик первого эшелона Sony. Эти студии создают игры эксклюзивно для игровых консолей, выпускаемых родительской компанией.

1.2. Что такое игра

Все мы имеем вполне четкое интуитивное представление о том, что такое игра. Общий термин «игра» охватывает настольные игры, такие как шахматы и «Монополия», карточные игры, например покер и блек-джек, игры в казино, такие как рулетка и автоматы, военные игры, компьютерные игры, различные виды детских игр, и этот список можно продолжать очень долго. Научный термин «теория игр» используется, когда в рамках четко установленного набора игровых правил требуется определить стратегию и тактику поведения нескольких участников для получения максимальной выгоды. Когда термин «игра» применяется в сфере консольных и компьютерных развлечений, он обычно вызывает в памяти изображения трехмерного виртуального мира с гуманоидом, зверем или транспортом в качестве главного персонажа, управляемого игроком. (У более старшего поколения он может вызывать воспоминания о двухмерной классике, такой как *Pong*, *Pac-Man* или *Donkey Kong*.) В прекрасной книге Рэфа Костера «Разработка игр и теория развлечений» игра определена как интерактивный опыт игрока, реализованный в виде усложняющейся последовательности шаблонных действий, которые он или она осваивает и в конечном счете выполняет [30]. Костер утверждает, что действия освоения и выполнения в игре являются главными в том, что мы называем «развлечение», так же как анекдот становится забавным в тот момент, когда мы улавливаем его смысл.

В данной книге мы сконцентрируемся на наборе игр, включающем двух- и трехмерные виртуальные миры с небольшим количеством игроков — от 1 до 16. К большей части того, что мы изучим, можно также отнести игры, написанные на HTML5/JavaScript в Интернете, чистые головоломки, такие как «Тетрис», или массовые многопользовательские онлайн-игры (MMOG). Но основное внимание будет нацелено на игровые движки, с помощью которых создают шутеры от первого лица, экшен-игры от третьего лица, гонки, файтинговые игры и т. п.

1.2.1. Видеоигры как мягкая симуляция реального времени

Большинство двух- и трехмерных видеоигр являются примерами того, что разработчики назвали бы *мягкой компьютерной симуляцией на основе интерактивного взаимодействия агентов в реальном времени*. Давайте разберем эту фразу по частям, чтобы лучше понять, что все это значит.

В большинстве видеоигр некоторая часть реального или воображаемого мира *моделируется* математически, поэтому им можно управлять с помощью компьютера. Модель является приближением и упрощением реальности (даже если это *воображаемая* реальность), потому что совершенно непрактично переносить в нее

каждую деталь вплоть до атомов или кварков. Следовательно, математическая модель является симуляцией реального или воображаемого игрового мира. Приближение и упрощение — два самых мощных инструмента разработчика игр. При умелом использовании даже очень упрощенная модель иногда почти неотличима от реальности — и даже немного веселее.

Симуляция *на основе агентов* — это такая, в которой взаимодействует ряд различных сущностей, известных как агенты. Под это описание подходит большинство трехмерных компьютерных игр, где агентами являются автомобили, герои, огненные шары, точки силы и т. д. Учитывая агентную природу множества игр, неудивительно, что большая их часть в настоящее время реализована на объектно-ориентированном или по крайней мере близком к таковому языке программирования.

Все интерактивные видеоигры являются *временными симуляторами*. Это означает, что модель виртуального игрового мира *динамична* — состояние игрового мира со временем меняется по мере развития событий и истории игры. Видеоигра также должна реагировать на непредсказуемые входные данные от игрока (игроков) — *интерактивные временные симуляции*. Наконец, большинство видеоигр представляют свои истории и реагируют на действия игроков в реальном времени, превращая их в интерактивные симуляции в реальном времени. Исключением может быть категория пошаговых игр, таких как компьютеризированные шахматы или пошаговые стратегии. Но даже эти типы игр обычно предоставляют пользователю некоторую форму графического пользовательского интерфейса в среде выполнения. Таким образом, для целей этой книги мы будем предполагать, что все видеоигры имеют как минимум *некоторые* ограничения, связанные с работой в реальном времени.

В основе любой системы реального времени лежит концепция *предельного значения (deadline)*. Наглядным примером является требование, чтобы экран обновлялся не менее 24 раз в секунду, чтобы создать иллюзию движения. (Большинство игр обновляют экран с частотой 30 или 60 кадров в секунду, потому что это кратно частоте обновления монитора NTSC.) Конечно, в видеоиграх много и других предельных значений. Симуляция физики может требовать обновления 120 раз в секунду, чтобы оставаться стабильной. Системе искусственного интеллекта персонажа может понадобиться «думать» по крайней мере раз в секунду, чтобы персонаж не показался глупым. Аудиобиблиотека может вызываться не реже одного раза в 1/60 секунды, чтобы аудиобуферы были заполнены и не было слышимого заикания.

«Мягкая» система реального времени — это система, в которой несоблюдение сроков не является катастрофическим. Следовательно, все видеоигры — это *мягкие системы реального времени*: если требование к частоте кадров не соблюдается, то с игроком, как правило, ничего не случается. Сравните это с *жесткой системой реального времени*, в которой пропущенный срок может означать серьезную травму или даже смерть оператора. Система авионики на вертолете или система управления стержнем на атомной электростанции — это примеры жестких систем реального времени.

Математические модели бывают *аналитическими* или *численными*. Например, аналитическая (замкнутая) математическая модель твердого тела, падающего

с постоянным ускорением под действием силы тяжести, обычно записывается следующим образом:

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0. \tag{1.1}$$

Аналитическая модель может быть вычислена для любых значений ее независимых переменных, таких как время t в уравнении (1.1), только с учетом начальных условий v_0 и y_0 и постоянной g. Когда такие модели можно создать, пользоваться ими очень удобно. Однако многие математические задачи не имеют решения в аналитическом виде. А в видеоиграх, где ввод пользователя непредсказуем, мы не можем полагаться на аналитическое моделирование всей игры.

Численная модель движения того же твердого тела под действием силы тяжести может быть выражена следующим образом:

$$y(t + \Delta t) = F\left(y(t), \dot{y}(t), \ddot{y}(t)\ldots\right). \tag{1.2}$$

Итак, высота твердого тела в некоторый следующий момент времени ($t + \Delta t$) может быть найдена как функция высоты и ее производной от времени первого, второго и, возможно, более высокого порядка в текущий момент времени t (1.2). Численное моделирование обычно выполняется путем многократных вычислений, чтобы определить состояние системы на каждом дискретном временно́м шаге. Игры работают так же. Основной игровой цикл запускается многократно, и во время каждой итерации различные игровые системы, такие как искусственный интеллект, игровая логика, физическая симуляция и др., получают возможность рассчитать или обновить свое состояние для следующего дискретного временно́го шага. Затем результаты визуализируются путем отображения графики, воспроизведения звука и, вероятно, создания других выходных данных, таких как отдача на джойстике.

1.3. Что такое игровой движок

Термин «игровой движок» появился в середине 1990-х годов в отношении игр-шутеров от первого лица (first-person shooter, FPS), таких как безумно популярная игра *Doom* компании id Software. *Doom* был спроектирован с достаточно четким разделением между основными программными компонентами (такими как система рендеринга трехмерной графики, система обнаружения коллизий или аудиосистема) и графическими ресурсами, игровыми мирами и правилами игры, которые вместе составляли игровой опыт. Ценность этого разделения стала очевидной, когда разработчики начали лицензировать игры и переделывать их в новые продукты, создавая новые изображения, макеты миров, оружие, персонажей, транспортные средства и правила игры при минимальных изменениях в самом движке. Это ознаменовало рождение сообщества моддеров — групп отдельных игроков и небольших независимых студий, которые создавали новые игры, модифицируя

уже существующие с помощью бесплатных наборов инструментов, предоставляемых разработчиками оригинала.

К концу 1990-х годов некоторые игры, такие как *Quake III Arena* и *Unreal*, разрабатывались с учетом возможности повторного использования кода и моддинга. Движки стали делать настраиваемыми с помощью языков сценариев, таких как Quake C компании id Software, а лицензирование движков стало хорошим вторичным источником дохода для их разработчиков. Сегодня разработчики игр могут лицензировать игровой движок и повторно задействовать значительную часть его ключевых программных компонентов для создания игр. Хотя эта практика по-прежнему требует значительных инвестиций в разработку программного обеспечения на заказ, она гораздо более экономична, чем разработка всех основных компонентов движка с нуля.

Граница между игрой и ее движком часто размыта. В некоторых случаях она довольно четкая, в других разработчики даже не пытаются ее провести. В одной игре код рендеринга может «точно знать», как рисовать орка. В другой игре движок рендеринга способен предоставлять универсальный материал и средства затенения и «оркность» может быть полностью определена в данных. Ни одна студия не проводит абсолютно четкого разделения между игрой и движком, и это понятно, учитывая, что определения этих двух компонентов часто меняются по мере разработки дизайна игры.

Вероятно, *архитектура, управляемая данными*, — это то, что отличает игровой движок от программного обеспечения, которое представляет собой игру, но не является движком. Когда игра содержит жестко запрограммированную логику или правила игры либо задействует специальный код для отображения определенных типов игровых объектов, становится трудно или невозможно повторно использовать это программное обеспечение для создания другой игры. Вероятно, нам следует зарезервировать термин «игровой движок» для программного обеспечения, которое расширяемо и может применяться в качестве основы для множества различных игр без значительных модификаций.

Понятно, что такое определение не подразумевает разделения на черное и белое. Мы можем представить шкалу многократного использования, на которой находится каждый движок. На рис. 1.1 показано расположение некоторых известных игр/движков на этой шкале.

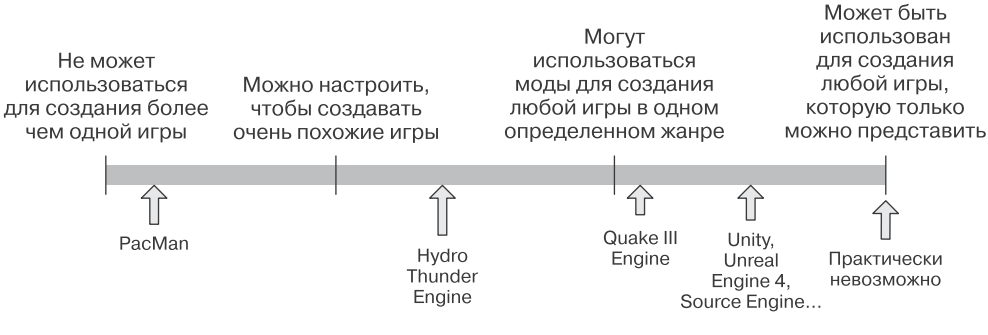


Рис. 1.1. Градация игровых движков по возможности их повторного применения

Можно подумать, что игровой движок должен быть чем-то похожим на Apple QuickTime или Microsoft Windows Media Player: являться универсальным программным обеспечением, способным воспроизводить практически *любой* игровой контент, который только можно себе представить. Однако этот идеальный вариант еще не достигнут и, вероятно, никогда не будет достигнут. Большинство игровых движков разработаны и тщательно настроены для запуска конкретной игры на конкретной аппаратной платформе. И даже самые универсальные кроссплатформенные движки действительно подходят только для создания игр в одном определенном жанре, таких как шутеры от первого лица или гонки. Можно с уверенностью сказать, что чем более универсален игровой движок или компонент промежуточного программного обеспечения, тем менее он оптимален для запуска конкретной игры на конкретной платформе.

Так происходит потому, что разработка любого эффективного компонента программного обеспечения неизменно влечет за собой компромиссы, основанные на предположениях о том, как будет использоваться программное обеспечение, и/или о целевом оборудовании, на котором оно будет работать. Например, движок рендеринга, созданный для работы с закрытыми помещениями, вероятно, не очень подойдет для рендеринга больших открытых пространств. Движок для помещений может задействовать систему порталов или дерево с двоичным разбиением пространства (binary space partitioning, BSP), чтобы гарантировать, что предметы не будут пересекаться стенами или объектами, находящимися ближе к камере. А движок для открытых пространств может применять менее точный механизм наложения или вообще не использовать его. Но он, вероятно, активно задействует техники уровня детализации (level-of-detail, LOD), чтобы гарантировать, что удаленные объекты отображаются минимальным количеством треугольников, а для объектов, которые находятся близко к камере, используется сеть треугольников, реализующая большее разрешение.

Появление более быстрого компьютерного оборудования и специализированных видеокарт наряду со все более эффективными алгоритмами рендеринга и структурами данных постепенно стирает различия между графическими движками для разных жанров. Например, теперь можно использовать движок для шутера от первого лица для создания стратегической игры. Однако компромисс между обобщением и оптимизацией все еще существует. Игру всегда реально сделать более впечатляющей, настроив движок под требования и ограничения конкретной игровой и/или аппаратной платформы.

1.4. Различия движков для разных жанров

Игровые движки, как правило, специфичны для конкретных жанров. Движок, разработанный для файтинга на боксерском ринге, будет сильно отличаться от движка для многопользовательской онлайн-игры (MMOG), или шутера от первого лица, или стратегии в реальном времени (real-time strategy, RTS). Тем не менее они имеют

и много общего — все 3D-игры, независимо от жанра, требуют некоторой формы низкоуровневого пользовательского ввода с джойстика, клавиатуры и/или мыши, определенной формы рендеринга трехмерного меша, некоторой формы отображения информации на экране (heads-up display, HUD), включая отображение текста различными шрифтами, мощную аудиосистему, и этот список можно продолжать. Так, например, несмотря на то что в свое время Unreal Engine был разработан для шутеров от первого лица, позже он успешно использовался для создания игр в ряде других жанров, в том числе в популярной франшизе шутеров от третьего лица *Gears of War* компании Epic Games, экшен-приключении *Batman: Arkham*, представленном Rocksteady Studios, известной игре-файтинге *Tekken 7* компании Bandai Namco Studios и первых трех ролевых шутерах от третьего лица серии *Mass Effect* компании BioWare.

Поговорим о некоторых наиболее распространенных игровых жанрах и рассмотрим примеры технологических требований, характерных для каждого из них.

1.4.1. Шутеры от первого лица

К жанру шутера от первого лица относятся такие игры, как *Quake*, *Unreal Tournament*, *Half-Life*, *Battlefield*, *Destiny*, *Titanfall* и *Overwatch* (рис. 1.2). Исторически они предусматривали относительно медленное пешее передвижение в потенциально большом, но в основном коридорном мире. Тем не менее в современных FPS действие происходит в самых разных виртуальных средах, включая обширные открытые и закрытые помещения. Современная механика перемещения в FPS может включать в себя передвижение пешком, на наземных транспортных средствах, как рельсовых, так и нет, судах на воздушной подушке, лодках и самолетах. Полный обзор этого жанра вы найдете на страницах en.wikipedia.org/wiki/First-person_shooter и https://ru.wikipedia.org/wiki/Шутер_от_первого_лица.

Игры от первого лица, как правило, одни из наиболее технологически сложных в создании. С ними конкурируют только шутеры от третьего лица, игры-платформеры и многопользовательские игры. Это вызвано тем, что FPS стремятся дать игрокам иллюзию погружения в детальный, гиперреалистичный мир. Неудивительно, что многие крупные технологические инновации игровой индустрии возникли из игр этого жанра.

Отличительными чертами движков FPS являются:

- эффективный рендеринг больших трехмерных виртуальных миров;
- отзывчивая механика управления камерой/прицеливания;
- высококачественная анимация виртуального оружия игрока;
- широкий ассортимент мощного оружия в руках;
- нестрогая модель движения персонажа и коллизий, которая часто придает этим играм ощущение плавания;

- высококачественная анимация и искусственный интеллект для неигровых персонажей (non-player characters, NPC) — врагов и союзников игрока;
- небольшие возможности многопользовательской онлайн-игры (обычно с поддержкой от 10 до 100 одновременных игроков), а также особый игровой режим «смертельный бой» (death match).

Рис. 1.2. Игра Overwatch от компании Blizzard Entertainment (Xbox One, PlayStation 4, Windows)

Технология рендеринга, используемая в шутерах от первого лица, почти всегда предельно оптимизирована и тщательно настроена на конкретный тип визуализируемой среды. Например, движки игр в закрытых помещениях типа «обход подземелий» часто используют деревья двоичного разбиения пространства или портальные системы рендеринга. А в FPS-играх на открытой местности применяются другие виды оптимизации, такие как отключение рендеринга объектов, невидимых в данный момент, или автономное разделение игрового мира на секторы с ручным или автоматическим определением того, какие целевые секторы видны из каждого исходного сектора.

Конечно, погружение игрока в гиперреалистичный игровой мир требует гораздо большего, чем просто оптимизированная высококачественная графическая технология. Анимация персонажей, аудио и музыка, физика твердых тел, игровая кинематика и множество других технологий в FPS должны быть самыми передовыми. Так что требования, предъявляемые к этому жанру, одни из самых строгих и обширных в индустрии игр.

1.4.2. Платформеры и другие игры от третьего лица

«Платформер» — это термин, применяемый к экшен-играм от третьего лица, где прыжок с платформы на платформу является основной механикой игрового процесса. Типичные игры эпохи 2D — *Space Panic*, *Donkey Kong*, *Pitfall!* и *Super Mario Brothers*. Эра 3D породила такие платформеры, как *Super Mario 64*, *Crash Bandicoot*, *Rayman 2*, *Sonic the Hedgehog*, серии *Jak* и *Daxter* (рис. 1.3), серии *Ratchet & Clank* и *Super Mario Galaxy*. Подробнее прочитать об этом жанре можно на страницах en.wikipedia.org/wiki/Platformer и https://ru.wikipedia.org/wiki/Платформер.

Рис. 1.3. Игра Jak II компании Naughty Dog (скриншот с сайта https://www.cinemapichollu.com/)

С точки зрения технологических требований платформеры, как правило, могут быть объединены с шутерами от третьего лица и экшен/приключенческими играми от третьего лица, такими как *Just Cause 2*, *Gears of War 4* (рис. 1.4), сериями *Uncharted*, *Resident Evil*, *The Last of Us*, *Red Dead Redemption 2* и многими другими.

Рис. 1.4. Игра Gears of War 4 компании The Coalition (Xbox One)

Игры от третьего лица имеют много общего с шутерами от первого лица, но в них гораздо больше внимания уделяется способностям и передвижению главного персонажа. Кроме того, для аватара игрока требуются высококачественная анимация всего тела, в отличие от несколько менее обременительных требований

к анимации «плавающих рук» в типичной игре FPS. Здесь важно отметить, что почти все шутеры от первого лица имеют многопользовательский онлайн-компонент, поэтому в дополнение к оружию здесь должен быть представлен аватар игрока во весь рост. Тем не менее в FPS реалистичность этих аватаров обычно несопоставима с реалистичностью неигровых персонажей, ее также нельзя сравнивать с точностью аватара игрока в игре от третьего лица.

В платформере главный герой часто похож на героя мультфильма и не особенно реалистичен, зачастую даже выполнен в невысоком разрешении. А в шутерах от третьего лица человеческий персонаж часто очень реалистичен. В обоих случаях персонаж игрока обычно имеет очень богатый набор действий и анимаций.

Некоторые из технологий, специально предназначенных для использования в играх в этом жанре, включают в себя:

- движущиеся платформы, лестницы, канаты, решетки и другие интересные способы передвижения;
- похожие на головоломки элементы окружающей среды;
- следующую за персонажем камеру, которая остается сфокусированной на нем. Ее вращение обычно контролирует игрок с помощью правой ручки джойстика (на консоли) или мыши (на ПК), (обратите внимание: существует ряд популярных шутеров от третьего лица для ПК, а вот платформеры предназначены почти исключительно для консолей);
- сложную систему контроля положения камеры, из-за которой точка обзора никогда не «утопает» в геометрии фона или в динамических объектах на переднем плане.

1.4.3. Файтинги

Файтинги — это драки для двух игроков, в которых персонажи, похожие на людей, бьют друг друга на некоем подобии ринга. Жанр характеризуется такими играми, как *Soul Calibur* и *Tekken 3* (рис. 1.5). Страницы «Википедии» en.wikipedia.org/wiki/Fighting_game и https://ru.wikipedia.org/wiki/Файтинг содержат обзор этого жанра.

Традиционно игры в жанре файтинг ориентированы:

- на богатый набор боевых анимаций;
- точное обнаружение попадания;
- систему пользовательского ввода, способную обнаруживать сложные комбинации нажатия кнопок;
- довольно статичные фоны, за исключением анимированных зрителей.

Поскольку трехмерный мир в этих играх невелик, а камера постоянно сосредоточена на действии, исторически сложилось так, что здесь практически не было

необходимости в разделении мира или отключении рендеринга невидимых объектов. Никто также не ожидает использования в них продвинутых трехмерных моделей распространения звука.

Рис. 1.5. Игра Tekken 3 компании Namco (PlayStation)

В современных файтингах, таких как EA *Fight Night Round 4* и *Injustice 2* компании NetherRealm Studios (рис. 1.6), технологические возможности улучшены благодаря следующим функциям:

- изображению персонажей с высоким разрешением;
- реалистичным оттенкам кожи с эффектами подповерхностного рассеяния и пота;
- фотореалистичному освещению и эффектам частиц;
- высококачественной анимации персонажей;
- реалистичному моделированию движения волос персонажей и ткани их одежды.

Важно отметить, что в некоторых файтингах, таких как *Heavenly Sword* компании The Ninja Theory и *For Honor* компании Ubisoft Montreal, действие происходит в масштабном виртуальном мире, а не на ограниченной арене. Многие считают их отдельным жанром, иногда называемым *дракой* (brawler). Этот вид файтинга может иметь технические требования, более сходные с требованиями шутера от третьего лица или стратегической игры.

Рис. 1.6. Игра Injustice 2 компании NetherRealm Studios (PlayStation 4, Xbox One, Android, iOS, Microsoft Windows)

1.4.4. Гонки

Гоночный жанр охватывает все игры, основной задачей которых является управление автомобилем или другим транспортным средством на каком-либо треке. В жанре много подкатегорий. Гоночные игры, ориентированные на симуляции (симы), нацелены на то, чтобы обеспечить максимально реалистичное вождение (например, *Gran Turismo*). Аркадные гонщики предпочитают увлекательные развлечения, а не реалистичность (например, *San Francisco Rush*, *Cruis'n USA*, *Hydro Thunder*). Один поджанр исследует субкультуру уличных гонок с навороченными переделанными машинами (например, *Need for Speed*, *Juiced*). Гонки на картингах — это подкатегория, в которой популярные персонажи из платформерных игр или персонажи анимационных фильмов превращаются в водителей смешных автомобилей (например, *Mario Kart Jak X*, *Freaky Flyers*). Гоночные игры не всегда должны включать соревнование на время. Например, в некоторых гоночных играх есть режимы, в которых игроки стреляют друг в друга, собирают добычу или участвуют в различных других состязаниях на время или без него. Больше об этом жанре вы прочитаете в «Википедии» (en.wikipedia.org/wiki/Racing_game и https://ru.wikipedia.org/wiki/Гоночная_игра).

Гоночная игра часто линейна и очень похожа на старые FPS-игры. Тем не менее скорость движения в ней, как правило, намного выше, чем в FPS. Поэтому больше внимания уделяется очень длинным прямым или кольцевым трекам, иногда с альтернативными маршрутами и секретными путями, где можно

срезать дорогу. В гоночных играх графические детали обычно сосредоточены на транспортных средствах, трассе и ближайшем окружении. В качестве примера на рис. 1.7 показан снимок экрана с последней версией известной серии гоночных игр *Gran Turismo*, *Gran Turismo Sport*, разработанной Polyphony Digital и опубликованной Sony Interactive Entertainment. Тем не менее в картинге много внимания уделяют анимации персонажей, управляющих транспортными средствами.

Рис. 1.7. Игра Gran Turismo Sport компании Polyphony Digital (PlayStation 4)

Вот некоторые технологические свойства типичной гоночной игры.

- При рендеринге отдаленных фоновых элементов используются различные хитрости, например двухмерные фигурки для деревьев, холмов и гор.
- Трек часто разбивается на относительно простые двухмерные области, называемые секторами. Эти структуры данных используются для оптимизации визуализации и определения видимости, чтобы помочь искусственному интеллекту находить путь для неуправляемых транспортных средств, а также решить многие другие технические проблемы.
- Камера обычно следует за автомобилем игрока. Иногда она находится внутри кабины и показывает вид от первого лица.
- Когда трек включает туннели и другие узкие пространства, разработчикам приходится прилагать значительные усилия, чтобы камера не сталкивалась с геометрией фона.

1.4.5. Игры-стратегии

Жанр современной стратегической игры был определен *Dune II: The Building of a Dynasty* (1992). Другие игры в этом жанре — *Warcraft*, *Command & Conquer*, *Age of Empires* и *Starcraft*. Здесь игрок стратегически размещает боевые единицы из своего арсенала на большом игровом поле и пытается сокрушить своего противника. Игровой мир обычно отображается под углом обзора сбоку сверху вниз. Часто делается различие между пошаговыми стратегиями и стратегиями в реальном времени. Больше информации об этом жанре — по адресам en.wikipedia.org/wiki/Strategy_video_game и https://ru.wikipedia.org/wiki/ Компьютерная_стратегическая_игра.

Игроку в стратегию обычно не дают возможности существенно изменить угол обзора, чтобы он мог сохранить большой охват территории. Это ограничение позволяет разработчикам задействовать различные хитрости оптимизации в движке рендеринга стратегической игры.

В старых играх этого жанра использовалась сеточная (клеточная) конструкция мира, а для упрощения рендеринга применялась ортографическая проекция. Например, на рис. 1.8 показан снимок экрана из классической стратегической игры *Age of Empires*.

Рис. 1.8. Игра Age of Empires компании Ensemble Studios (Windows)

В современных стратегических играх иногда используется перспективная проекция и создается настоящий трехмерный мир, хотя по-прежнему может применяться сетчатая система разметки для обеспечения правильного выравнивания юнитов и фоновых элементов, таких как здания. Известный пример, *Total War: Warhammer 2*, показан на рис. 1.9.

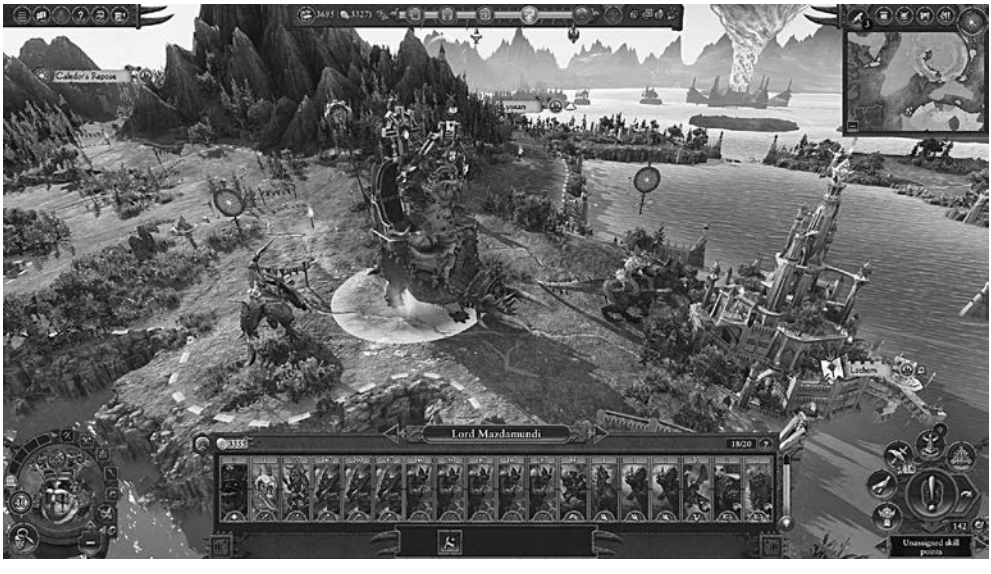


Рис. 1.9. Игра Total War: Warhammer 2 компании Creative Assembly (Windows)

Некоторые распространенные практики в стратегических играх таковы.

- Каждый юнит имеет относительно низкое разрешение, так что игра способна одновременно поддерживать большое их количество на экране.
- Поле местности — это обычно холст, на котором разрабатывается и происходит игра.
- Игроку часто разрешается строить новые структуры на местности в дополнение к развертыванию своих сил.
- Взаимодействие с пользователем обычно реализуется с помощью щелчков кнопкой мыши для выбора юнитов и области, а также меню или панелей инструментов, содержащих команды, оборудование, типы юнитов, типы зданий и т. п.

1.4.6. Многопользовательские онлайн-игры

Жанр массовой многопользовательской онлайн-игры (MMOG, или просто MMO) представлен такими играми, как *Guild Wars 2* (AreaNet/NCsoft), *EverQuest* (989 Stu-

dios/SOE), *World of Warcraft* (Blizzard) и *Star Wars Galaxies* (SOE/Lucas Arts), и это только некоторые. MMO определяется как любая игра, которая поддерживает огромное количество одновременно играющих пользователей (от тысяч до сотен тысяч). Обычно все они действуют в одном очень большом постоянном виртуальном мире (то есть в мире, внутреннее состояние которого сохраняется в течение очень длительных периодов времени, намного превышающих время нахождения в нем любого игрока). В остальном игровой процесс MMO часто похож на развитие событий в небольших многопользовательских аналогах. Подкатегории этого жанра: ролевые игры (MMORPG), стратегии в реальном времени (MMORTS) и шутеры от первого лица (MMOFPS). Больше об этом жанре читайте в «Википедии» (en.wikipedia.org/wiki/MMOG и https://ru.wikipedia.org/wiki/Массовая_многопользовательская_онлайн-игра). На рис. 1.10 показан снимок экрана из чрезвычайно популярной MMORPG *World of Warcraft*.

Рис. 1.10. Игра World of Warcraft компании Blizzard Entertainment (Windows, MacOS)

В основе всех MMOG лежит очень мощный набор серверов. Они поддерживают состояние игрового мира, управляют пользователями, входящими в игру и выходящими из нее, предоставляют услуги межпользовательского чата или передачи

голоса по IP (VoIP) и многое другое. Почти все MMOG требуют, чтобы пользователи оплачивали регулярную подписку для доступа к игре, поэтому они могут предлагать микротранзакции в игровом мире или вне игры. Так что, возможно, самая важная роль центрального сервера заключается в обработке счетов и микротранзакций, которые служат основным источником дохода разработчика игры.

Качество графики у MMO почти всегда ниже, чем у ее немногопользовательских собратьев из-за огромных размеров мира и чрезвычайно большого числа пользователей, поддерживаемых этими играми.

На рис. 1.11 показан экран из последней созданной Bungie FPS-игры *Destiny 2*. Она названа MMOFPS, потому что включает в себя некоторые аспекты жанра MMO. Тем не менее Bungie предпочитает называть ее игрой общего мира, потому что, в отличие от традиционной MMO, в которой игрок может видеть буквально любого другого игрока на конкретном сервере и контактировать с ним, *Destiny* обеспечивает подбор матчей на лету. Это позволяет игроку взаимодействовать только с игроками, которые с ним сопоставлены сервером. Данная система поиска матчей была значительно улучшена для *Destiny 2*. Кроме того, в отличие от традиционной MMO, графика в *Destiny 2* не уступает по качеству графике шутеров от первого и третьего лица.

Рис. 1.11. Игра Destiny 2 компании Bungie (снимок экрана с сайта https://greatgamer.ru) (Xbox One, PlayStation 4, ПК)

Здесь необходимо отметить, что благодаря игре *Player Unknown's Battlegrounds* (PUBG) недавно стал популярен поджанр, известный как *королевская битва (Battle Royale)*. Этот тип игры стирает грань между обычными многопользовательскими шутерами и массовыми многопользовательскими онлайн-играми, потому что в них обычно соревнуются около 100 игроков друг против друга в онлайн-мире. Игровой процесс построен на выживании игроков, побеждает последний оставшийся в живых.

1.4.7. Контент, созданный игроком

Рис. 1.12. Игра LittleBigPlanet™ 2 компании Media Molecule (снимок экрана с сайта https://www.igromania.ru) (PlayStation 3)

По мере развития социальных сетей игры становятся все более совместными по своей природе. Новая тенденция в разработке связана с авторским контентом. Например, *LittleBigPlanet™* компании Media Molecule, *LittleBigPlanet™ 2* (рис. 1.12) и *LittleBigPlanet™ 3: The Journey Home* — технически платформеры с головоломками, но их наиболее заметная и уникальная особенность заключается в том, что они позволяют игрокам создавать собственные игровые миры, публиковать их и делиться ими. Последняя версия Media Molecule в этом увлекательном жанре — *Dreams* для PlayStation 4 (рис. 1.13).

Пожалуй, сейчас самой популярной игрой в жанре созданного игроком контента является *Minecraft* (рис. 1.14). Гениальность этой игры заключается в ее простоте: игровые миры *Minecraft* построены из простых нанесенных на карту кубических элементов с текстурами низкого разрешения, призванными имитировать различные материалы. Блоки могут быть сплошными или содержать различные предметы, такие как факелы, наковальни, таблички, заборы и стеклянные панели. Игровой мир населен одним или несколькими персонажами игроков, животными, например курами и свиньями, и различными мобами — хорошими парнями, такими как сельские жители, плохими парнями, такими как зомби, и вездесущими *криперами*, которые подкрадываются к ничего не подозревающим игрокам и взрываются (лишь предупреждая игрока шипением горящего фитиля).

Рис. 1.13. Игра Dreams компании Media Molecule (снимок экрана с сайта https://worthplaying.com) (PlayStation 4)

Рис. 1.14. Игра Minecraft разработчика Markus «Notch» Persson/Mojang AB (Windows, MacOS, Xbox 360, PlayStation 3, PlayStation Vita, iOS)

Игроки могут создать случайный мир в *Minecraft*, а затем копаться в сгенерированной местности, делая туннели и пещеры. А также строить собственные структуры, начиная с создания простого ландшафта и заканчивая сложными зданиями и механизмами. Вероятно, наиболее гениальный ход в *Minecraft* — добавление красного камня. Этот материал служит «проводкой», позволяющей

игрокам устанавливать схемы, управляющие поршнями, бункерами, шахтными тележками и другими динамическими элементами в игре. В результате игроки могут создавать практически все, что способны представить, а затем делиться своими мирами с друзьями, публикуя свой сервер и приглашая их играть онлайн.

1.4.8. Виртуальная, дополненная и смешанная реальность

Виртуальная, дополненная и смешанная реальность — это захватывающие новые технологии, стремящиеся погрузить зрителя в трехмерный мир, который либо полностью генерируется компьютером, либо дополнен компьютерными изображениями. Эти технологии широко применяются за пределами игровой индустрии, а также стали жизнеспособной основой для широкого спектра игрового контента.

Виртуальная реальность

Виртуальная реальность (virtual reality, VR) может быть определена как многоаспектная мультимедийная или компьютерная реальность, имитирующая присутствие пользователя в среде, которая является местом в реальном или воображаемом мире. Созданная компьютером виртуальная реальность (computer-generated (CG) VR) является подмножеством этой технологии, в ней виртуальный мир генерируется исключительно с помощью компьютерной графики. Пользователь может увидеть эту виртуальную среду, надевая гарнитуру, такую как HTC Vive, Oculus Rift, Sony PlayStation VR, Samsung Gear VR или Google Daydream View. Гарнитура отображает контент прямо перед глазами пользователя, при этом система отслеживает ее перемещение в реальном мире, так что движения виртуальной камеры идеально соответствуют движениям человека, надевшего гарнитуру. Пользователь обычно держит устройства, которые позволяют системе отслеживать движения обеих рук. Это дает возможность игроку взаимодействовать с виртуальным миром. Например, объекты можно толкать, поднимать или бросать.

Дополненная и смешанная реальность

Термины *«дополненная реальность»* (augmented reality, AR) и *«смешанная реальность»* (mixed reality, MR) часто путают или используют взаимозаменяемо. Обе технологии представляют пользователю реальный мир с компьютерной графикой. В обеих устройство просмотра, такое как смартфон, планшет или технически доработанная пара очков, выводит статичное изображение реального мира или изменяемое в реальном времени, а поверх него накладывается компьютерная графика. В системах AR и MR в реальном времени акселерометры в устройстве просмотра позволяют движениям виртуальной камеры отслеживать движения устройства, создавая иллюзию, что оно представляет собой просто окно, через которое мы ви-

дим реальный мир, а следовательно, позволяя наложенной компьютерной графике казаться реальной.

Некоторые различают эти две технологии, используя термин «дополненная реальность» для описания технологий, в которых компьютерная графика накладывается на изображение реального мира, но не привязывается к нему. Термин же «смешанная реальность» чаще применяется к использованию компьютерной графики для визуализации воображаемых объектов, которые привязаны к реальному миру и кажутся существующими в нем. Однако это различие ни в коем случае не является общепринятым.

Вот несколько примеров технологии AR в действии.

- В армии США солдаты получают сведения о необходимых тактических действиях с помощью системы, получившей название «Тактическая дополненная реальность» (TAR). Она накладывает на изображение, видимое солдатом, мини-карту и отметки объектов, как на экране игры (www.youtube.com/watch?v=x8p19j8C6VI).
- В 2015 году Disney продемонстрировал несколько классных технологий AR — они отображают трехмерный анимационный персонаж поверх листа реальной бумаги, на которой 2D-версия персонажа раскрашена карандашом (www.youtube.com/watch?v=SWzurBQ81CM).
- PepsiCo разыграла лондонцев, применив технологию AR. Люди, сидевшие на автобусной остановке, видели изображения крадущегося тигра, падения метеорита или инопланетянина, хватавшего щупальцами прохожих на улице (www.youtube.com/watch?v=Go9rf9GmYpM).

И вот несколько примеров MR.

- Начиная с Android 8.1, приложение камеры на Pixel 1 и Pixel 2 поддерживает *AR Stickers* — забавную функцию, которая позволяет пользователям размещать анимированные 3D-объекты и персонажей на видео и фотографиях.
- *HoloLens* компании Microsoft — еще один пример смешанной реальности. Он накладывает привязанную к миру графику на реальное видеоизображение и может использоваться для широкого спектра целей, включая обучение, инженерию, здравоохранение и развлечения.

VR/AR/MR-игры

Игровая индустрия в настоящее время экспериментирует с технологиями VR и AR/MR и пытается найти свое место в этих новых медиа. Некоторые традиционные 3D-игры были перенесены в виртуальную реальность, что дало очень интересные впечатления. Но, возможно, в дальнейшем появятся более захватывающие, совершенно новые игровые жанры, предлагающие игровой опыт, который нельзя получить без VR или AR/MR.

Например, *Job Simulator* компании Owlchemy Labs погружает пользователя в виртуальный музей, которым управляют роботы, и просит его выполнить стран-

ные примеры различных реальных заданий, используя игровую механику, которая просто не будет работать на платформе без VR. Другая игра Owlchemy, *Vacation Simulator*, использует те же причудливое чувство юмора и художественный стиль в мире, где роботы из *Job Simulator* приглашают игрока расслабиться и выполнять различные задания. На рис. 1.15 показан снимок экрана из другой новаторской (и несколько пугающей!) игры для HTC Vive под названием *Accounting* от создателей *Rick & Morty* и *The Stanley Parable*.

Рис. 1.15. Игра Accounting компании Squanchtendo и Crows (HTC Vive)

Игровые движки виртуальной реальности

Игровые движки VR технологически во многом похожи на движки шутеров от первого лица, и действительно, многие движки с поддержкой FPS, такие как Unity и Unreal Engine, поддерживают VR «из коробки». Тем не менее VR-игры существенно отличаются от FPS-игр.

- *Стереоскопический рендеринг*. VR-игра должна показывать одну и ту же сцену дважды, отдельно для каждого глаза. Это удваивает количество графических примитивов, которые должны быть визуализированы, хотя другие аспекты графического конвейера, такие как определение области видимости, могут выполняться только один раз за кадр, поскольку глаза находятся довольно

близко друг к другу. Таким образом, рендеринг VR-игры не такой затратный, как рендеринг в многопользовательском режиме с разделением экрана, но принцип рендеринга каждого кадра дважды с двух немного разных виртуальных камер одинаков.

- *Очень высокая частота кадров.* Исследования показали, что VR с частотой менее 90 кадров в секунду способна вызвать дезориентацию, тошноту и другие негативные последствия для пользователя. Это означает, что VR-системы должны не только визуализировать сцену дважды за кадр, но и делать это со скоростью 90+ FPS. Вот почему для VR-игр и приложений, как правило, требуются мощные процессор или графическая карта.
- *Проблемы с навигацией.* В игре FPS игрок может просто ходить по игровому миру с помощью джойстика или клавиш WASD. В виртуальной игре пользователь способен физически гулять по игровому миру, немного передвигаясь в реальном, но безопасная физическая игровая зона обычно довольно мала (как небольшая ванная или туалет). «Перелеты» с места на место могут вызвать тошноту, поэтому в большинстве игр выбран механизм телепортации «укажи и щелкни», чтобы перемещать виртуального игрока/камеру на большие расстояния. Были также разработаны различные устройства реального мира, которые позволяют пользователю виртуальной реальности переступать ногами на месте, чтобы перемещаться в виртуальном мире.

Конечно, VR несколько компенсирует эти ограничения, предоставляя новые парадигмы взаимодействия с пользователем, которые невозможны в традиционных видеоиграх, например:

- пользователи могут подходить к предметам в реальном мире, подбирать и бросать их в виртуальном;
- игрок способен увернуться от атаки в виртуальном мире, уклонившись физически в реальном;
- появляются новые возможности пользовательского интерфейса, такие как прикрепление плавающих меню к виртуальным рукам или просмотр очков игры, написанных на доске в виртуальном мире;
- игрок может даже взять пару очков виртуальной реальности, надеть их и перенестись во вложенный мир виртуальной реальности. Этот эффект лучше всего назвать «VR-переход».

Игры на основе местоположения

Такие игры, как *Pokémon Go*, не накладывают графику на изображение реального мира и не создают захватывающий виртуальный мир. Тем не менее сгенерированный компьютером мир *Pokémon Go* реагирует на движения телефона или планшета пользователя, подобно видео с 360-градусным обзором. И игра знает о вашем

местонахождении в реальном мире, побуждая вас искать покемона в близлежащих парках, торговых центрах и ресторанах. Такую игру нельзя назвать полноценной AR/MR, но она не относится и к категории VR. Можно описать ее как форму развлечения на основе местоположения, хотя некоторые люди используют для таких игр термин AR.

1.4.9. Другие жанры

Конечно, есть много других игровых жанров, которые мы не будем здесь подробно описывать. Вот некоторые примеры:

- спорт с поджанрами для каждого основного вида (футбол, бейсбол, гольф и т. д.);
- ролевые игры (РПГ);
- симуляторы бога, такие как *Populous* и *Black & White*;
- социальные симуляторы, такие как *SimCity* или *The Sims*;
- головоломки, например «Тетрис»;
- конверсии неэлектронных игр — шахмат, карточных игр, го и т. п.;
- сетевые игры, подобные предлагаемым на сайте Electronic Arts Pogo.

И этот список можно продолжать.

Мы видели, что у каждого игрового жанра есть особые технологические требования. Это объясняет, почему игровые движки традиционно немного различаются от жанра к жанру. Тем не менее есть и значительное совпадение в технологиях для разных жанров, особенно в контексте одной аппаратной платформы. С появлением все более и более мощного оборудования различия между жанрами, возникшие из-за проблем с оптимизацией, постепенно исчезают. Поэтому повторное использование одной и той же технологии в разных жанрах и даже на разных аппаратных платформах становится более вероятным.

1.5. Обзор игровых движков

1.5.1. Семейство движков Quake

Самым первым 3D-шутером от первого лица принято считать *Castle Wolfenstein 3D* (1992). Написанная id Software в Техасе для ПК, эта игра вывела игровую индустрию в новое захватывающее русло. Компания id Software развила ее в *Doom*, *Quake*, *Quake II* и *Quake III*. Все эти движки очень похожи по архитектуре, и я буду называть их семейством движков Quake. Технология Quake использована для создания многих других игр и даже других движков. Например, родословная *Medal of Honor* для ПК-платформ выглядит примерно так:

- *Quake III* (id Software);
- *Sin* (Ritual);
- *F.A.K.K. 2* (Ritual);
- *Medal of Honor: Allied Assault* (2015 и Dreamworks Interactive);
- *Medal of Honor: Pacific Assault* (Electronic Arts, Лос-Анджелес).

Многие другие игры, основанные на технологии Quake, разрабатываются на разных студиях по тому же принципу. Да и движок Source компании Valve, на котором создана серия игр *Half-Life*, также корнями уходит в технологию Quake.

Исходный код *Quake* и *Quake II* находится в свободном доступе, а оригинальные движки Quake довольно хорошо спроектированы и чисты (хотя, конечно, немного устарели и полностью написаны на C). Эти исходники служат отличным примером того, как создаются игровые движки промышленного масштаба. Полный исходный код для *Quake* и *Quake II* доступен по адресу github.com/id-Software/Quake-2.

Если у вас есть игры Quake и/или Quake II, вы можете собрать код с помощью Microsoft Visual Studio и запустить игру под отладчиком, используя реальные игровые ресурсы с диска. Это может быть невероятно поучительно. У вас получится установить точки останова, запустить игру и затем проанализировать, как на самом деле работает движок, двигаясь по коду пошагово. Я настоятельно рекомендую загрузить один или оба этих движка и таким образом проанализировать исходный код.

1.5.2. Движок Unreal

Epic Games, Inc. в 1998 году вышла на сцену FPS с легендарной игрой *Unreal*. С тех пор Unreal Engine стал основным конкурентом технологии Quake в пространстве FPS. Unreal Engine 2 (UE2) является основой для *Unreal Tournament 2004* (UT2004) и использовался для бесчисленных модов, университетских проектов и коммерческих игр. Создание Unreal Engine 4 (UE4) — эволюционный шаг. Игра может похвастаться одними из лучших подборок инструментов и богатейшими наборами функций в отрасли, включая удобный и мощный графический пользовательский интерфейс для создания шейдеров, а также графический пользовательский интерфейс для программирования игровой логики, называемый *Blueprints* (ранее известный как Kismet).

Unreal Engine стал известен благодаря своим многочисленным функциям и универсальным, простым в применении инструментам. Unreal Engine неидеален, и большинство разработчиков модифицируют его различными способами, чтобы оптимально запустить свою игру на конкретной аппаратной платформе. Тем не менее Unreal — невероятно мощный инструмент для создания прототипов

и коммерческая платформа для разработки игр, и его можно использовать для создания практически любой трехмерной игры от первого или третьего лица (не говоря уже об играх в других жанрах). С помощью UE4 были разработаны многие увлекательные игры всех жанров, включая *Rime* компании Tequila Works, *Genesis: Alpha One* компании Radiation Blue, *Way Out* компании Hazelight Studios и *Crackdown 3* компании Microsoft Studios.

Сообщество разработчиков Unreal (Unreal Developer Network (UDN)) предоставляет богатый набор документации и другой информации обо всех выпущенных версиях Unreal Engine (см. udn.epicgames.com/Main/WebHome.html). Часть документов можно использовать свободно. Однако доступ к полной документации для последней версии Unreal Engine, как правило, ограничен лицензиатами движка. Существует множество других полезных веб-сайтов и вики, посвященных Unreal Engine. Одним из популярных является www.beyondunreal.com.

К счастью, компания Epic теперь предлагает полный доступ к Unreal Engine 4, исходному коду и всему остальному за небольшую ежемесячную абонентскую плату плюс небольшую прибыль от вашей игры, если она выйдет. Это делает UE4 оптимальным выбором для небольших независимых игровых студий.

1.5.3. Source — движок Half-Life

Source — игровой движок, на котором построены *Half-Life 2* и ее продолжения *HL2: Episode One* и *HL2: Episode Two*, *Team Fortress 2* и *Portal* (поставляются вместе под названием *The Orange Box*). Source — это высококачественный движок, конкурирующий с Unreal Engine 4 в областях графических возможностей и набора инструментов.

1.5.4. DICE Frostbite

Движок Frostbite был создан усилиями DICE при разработке игрового движка для *Battlefield Bad Company* в 2006 году. С тех пор он стал самым распространенным движком в Electronic Arts (EA) и используется во многих ее основных франшизах, включая *Mass Effect*, *Battlefield*, *Need for Speed*, *Dragon Age* и *Star Wars Battlefront II*. Frostbite может похвастаться мощным инструментом создания ресурсов под названием FrostEd, мощным конвейером инструментов Backend Services и мощной средой выполнения. Это запатентованный движок, поэтому он, к сожалению, недоступен разработчикам вне EA.

1.5.5. RAGE

RAGE (Rockstar Advanced Game Engine) — это движок, который управляет безумно популярным *Grand Theft Auto V*. Разработанный RAGE Technology Group — подразделением Rockstar Games студии Rockstar San Diego, RAGE использовался внутренними студиями Rockstar Games для разработки игр для PlayStation 4,

Xbox One, PlayStation 3, Xbox 360, Wii, Windows и MacOS. Другие игры, созданные на этом движке, — *Grand Theft Auto IV*, *Red Dead Redemption* и *Max Payne 3*.

1.5.6. CRYENGINE

Crytek разработала свой мощный игровой движок, известный как CRYENGINE, для технической демонстрации для NVIDIA. Когда потенциал технологии был признан, Crytek превратила демо в полноценную игру и на свет появилась *Far Cry*. С тех пор на основе CRYENGINE было создано много игр, включая *Crysis*, *Codename Kingdoms*, *Ryse: Son of Rome* и *Everyone's Gone to the Rapture*. За прошедшие годы движок превратился в то, что сейчас является последним продуктом Crytek, — CRYENGINE V. Эта мощная платформа для разработки игр предлагает широкий набор инструментов для создания ресурсов и многофункциональную среду выполнения с высококачественной графикой. CRYENGINE можно использовать при создании игр для большинства платформ, включая Xbox One, Xbox 360, PlayStation 4, PlayStation 3, Wii U, Linux, iOS и Android.

1.5.7. Sony PhyreEngine

Стремясь сделать разработку игр для платформы Sony PlayStation 3 более доступной, на конференции разработчиков игр (Game Developer's Conference, GDC) в 2008 году Sony представила PhyreEngine. С 2013 года он превратился в мощный полнофункциональный игровой движок, поддерживающий впечатляющий набор функций, включая улучшенное освещение и отложенный рендеринг. Многие студии использовали его для создания более 90 опубликованных игр, в том числе такие хиты, как *flOw*, *Flower*, *Journey* и *Unravel* компании Coldwood Interactive. Теперь PhyreEngine поддерживает платформы Sony PlayStation 4, PlayStation 3, PlayStation 2, PlayStation Vita и PSP. PhyreEngine предоставляет разработчикам доступ к процессору CELL (PS3) с его высокопараллельной архитектурой, к расширенным вычислительным возможностям PS4, а также к новому модернизированному редактору мира и другим мощным инструментам для разработки игр. Он доступен бесплатно любому лицензированному разработчику Sony в составе PlayStation SDK.

1.5.8. Microsoft XNA Game Studio

Microsoft XNA Game Studio — это простая в использовании высокодоступная платформа для разработки игр, основанная на языке C# и Common Language Runtime (CLR) и направленная на то, чтобы побудить игроков создавать собственные игры и делиться ими с сообществом онлайн-игр так же, как YouTube поощряет создание и распространение любительского видео.

Хорошо это или плохо, но Microsoft официально отказалась от XNA в 2014 году. Тем не менее разработчики могут перенести свои игры XNA на iOS, Android,

Mac OS X, Linux и Windows 8 Metro через реализацию XNA с открытым исходным кодом под названием MonoGame. Для получения дополнительной информации см. www.windowscentral.com/xna-dead-long-live-xna.

1.5.9. Unity

Unity — это мощная кросс-платформенная среда разработки игр и движок, поддерживающий широкий спектр платформ. Используя Unity, разработчики могут разворачивать свои игры на мобильных платформах (например, Apple iOS, Google Android), консолях (Microsoft Xbox 360 и Xbox One, Sony PlayStation 3 и PlayStation 4, Nintendo Wii, Wii U), переносимых игровых платформах (например, Playstation Vita, Nintendo Switch), настольных компьютерах (Microsoft Windows, Apple Macintosh и Linux), приставках (например, Android TV и tvOS) и системах виртуальной реальности (VR) (например, Oculus Rift, Steam VR, Gear VR).

Основной целью Unity является простота разработки и развертывания кроссплатформенных игр. Таким образом, Unity предоставляет простой в использовании интегрированный редактор, в котором вы можете управлять ресурсами и сущностями, составляющими игровой мир, и быстро тестировать игру в действии прямо из редактора или непосредственно на своем целевом аппаратном обеспечении. Unity также предоставляет мощный набор инструментов для анализа и оптимизации вашей игры для каждой целевой платформы, всеобъемлющий конвейер подготовки ресурсов и возможность гибкого управления качеством и производительностью для каждой платформы. Unity поддерживает скрипты на JavaScript, C# или Boo, имеет мощную систему анимации, поддерживающую ретаргетинг анимации (способность воспроизводить для персонажа анимацию, которая была написана для совершенно другого персонажа), и поддержку сетевых многопользовательских игр.

Unity был использован для создания большого количества игр, в том числе *Deus Ex: The Fall* компании N-Fusion/Eidos Montreal, *Hollow Knight* компании Team Cherry и игры в ретростиле *Cuphead* компании StudioMDHR. Короткометражный фильм «Адам», удостоенный награды Webby, отрисовывался в режиме реального времени с применением Unity.

1.5.10. Другие коммерческие игровые движки

Есть много других коммерческих игровых движков. Хотя инди-разработчики могут не располагать бюджетом на покупку движка, многие программные продукты имеют отличную онлайн-документацию и/или справочник вики, которые способны стать отличным источником информации об игровых движках и разработке игры в целом. Например, обратите внимание на движок Tombstone (tombstoneengine.com) компании Terathon Software, движок LeadWerks (www.leadwerks.com) и HeroEngine компании Idea Fabrik, PLC (www.heroengine.com).

1.5.11. Внутренние движки, находящиеся в частной собственности

Многие компании создают и поддерживают собственные игровые движки. Electronic Arts построила многие из своих RTS-игр на собственном движке под названием Sage, разработанном в Westwood Studios. *Crash Bandicoot* франшизы Naughty Dog и *Jak and Daxter* базируются на фирменном движке, специально разработанном для PlayStation и PlayStation 2. Для серии *Uncharted* компания Naughty Dog разработала совершенно новый движок, адаптированный для оборудования PlayStation 3. Он развивался и в конечном итоге использовался для создания серии игр Naughty Dog *The Last of Us* для PlayStation 3 и PlayStation 4, а также ее последних выпусков *Uncharted 4: A Thief's End* и *Uncharted: The Lost Legacy*. И конечно же, большинство коммерчески лицензированных игровых движков, таких как Quake, Source, Unreal и CRYENGINE, начинались как внутренние движки.

1.5.12. Движки с открытым исходным кодом

Трехмерные игровые движки с открытым исходным кодом — это движки, созданные любителями и профессиональными разработчиками игр, которые можно скачать бесплатно. Термин «открытый исходный код» обычно подразумевает, что этот код находится в свободном доступе и используется частично открытая модель разработки, то есть почти любой может внести свои доработки. Лицензирование, если оно вообще существует, часто протекает в рамках открытых лицензий Gnu Public License (GPL) или Lesser Gnu Public License (LGPL). Первая позволяет любому пользователю свободно применять код, если его код также находится в свободном доступе, вторая позволяет делать это даже в частных коммерческих приложениях. Для проектов с открытым исходным кодом доступно много других бесплатных и полусвободных схем лицензирования.

В Сети можно скачать огромное количество движков с открытым исходным кодом. Некоторые довольно хороши, некоторые посредственны, а некоторые просто ужасны! Список игровых движков, находящийся в Интернете по адресу en.wikipedia.org/wiki/List_of_game_engines, даст вам представление о том, как их много. (Список на сайте www.worldofleveldesign.com/categories/level_design_tutorials/recommended-game-engines.php немного удобнее.) Оба этих списка включают как движки с открытым исходным кодом, так и коммерческие игровые движки.

OGRE — это хорошо спроектированный, простой в освоении и использовании движок для 3D-рендеринга. Он может похвастаться полнофункциональным 3D-рендерингом, включающим улучшенное освещение и тени, хорошую систему скелетной анимации персонажей, систему двухмерного наложения для экранных дисплеев и графических пользовательских интерфейсов, а также систему постобработки для полноэкранных эффектов, таких как bloom. OGRE,

по признанию его авторов, не является полноценным игровым движком, но он предоставляет многие базовые компоненты, необходимые практически любому игровому движку.

Некоторые другие известные движки с открытым исходным кодом перечислены далее.

- Panda3D — это движок на основе скриптов. Основным интерфейсом является пользовательский язык сценариев Python. Он предназначен для удобного и быстрого создания прототипов, 3D-игр и виртуальных миров.
- Yake — игровой движок, построенный на основе OGRE.
- Crystal Space — игровой движок с расширяемой модульной архитектурой.
- Torque и Irrlicht также известные игровые движки с открытым исходным кодом.
- Движок Lumberyard технически не является проектом с открытым исходным кодом, однако он предоставляет исходный код своим разработчикам. Это бесплатный кросс-платформенный движок, разработанный Amazon и основанный на архитектуре CRYENGINE.

1.5.13. Игровые движки 2D для непрограммистов

Двухмерные игры стали невероятно популярными благодаря браузерным и мобильным играм на таких платформах, как Apple iPhone/iPad и Google Android. Появился ряд популярных наборов инструментов для разработки игр/мультимедиа, позволяющих небольшим игровым студиям и независимым разработчикам создавать 2D-игры для этих платформ. Такие наборы инструментов просты в использовании, позволяют пользователям создавать игры с помощью графического интерфейса и не требуют знания языка программирования. Посмотрите на YouTube видео, показывающее, какие игры можно создать с применением этих наборов инструментов, — www.youtube.com/watch?v=3Zq1yo0lxOU.

- *Multimedia Fusion 2* (www.clickteam.com/website/world — это набор инструментов для создания 2D-игр/мультимедиа, разработанный Clickteam. Fusion используют профессионалы отрасли для создания игр, заставок и других мультимедийных приложений. Fusion и его более простой аналог The Games Factory 2 применяют в образовательных лагерях, таких как PlanetBravo (www.planetbravo.com), чтобы научить детей разработке игр и программированию/логике. Fusion поддерживает платформы iOS, Android, Flash и Java.
- *Game Salad Creator* (gamesalad.com/creator) — еще один графический инструментарий для создания игр/мультимедиа, предназначенный для непрограммистов и во многом похожий на Fusion.
- *Scratch* (scratch.mit.edu) — это авторский инструментарий и язык графического программирования, который можно использовать для создания интерактивных демонстраций и простых игр. Это отличное средство для начинающих, позволяющее узнать о концепциях программирования, таких как условные

выражения, циклы и программирование на основе событий. Scratch разработан в 2003 году группой Lifelong Kindergarten, которую возглавлял Митчел Резник из MIT Media Lab.

1.6. Архитектура среды выполнения движка

Игровой движок обычно состоит из набора инструментов и среды выполнения. Сначала мы изучим архитектуру среды выполнения, а в следующем разделе перейдем к архитектуре инструментов.

На рис. 1.16 показаны основные компоненты среды выполнения, которые составляют типичный движок 3D-игр. Да, их *много*! И здесь далеко не все инструменты. Игровые движки — это, безусловно, большие программные системы.

Как и все программные системы, игровые движки имеют многоуровневое построение. Обычно верхние уровни зависят от нижних, но не наоборот. Зависимость нижнего уровня от более высокого мы называем *цикличной зависимостью*. Таких зависимостей стоит избегать в любой программной системе, потому что они создают нежелательные связи между компонентами, делают программное обеспечение нестабильным и препятствуют повторному использованию кода. Это особенно важно для крупномасштабной системы, такой как игровой движок.

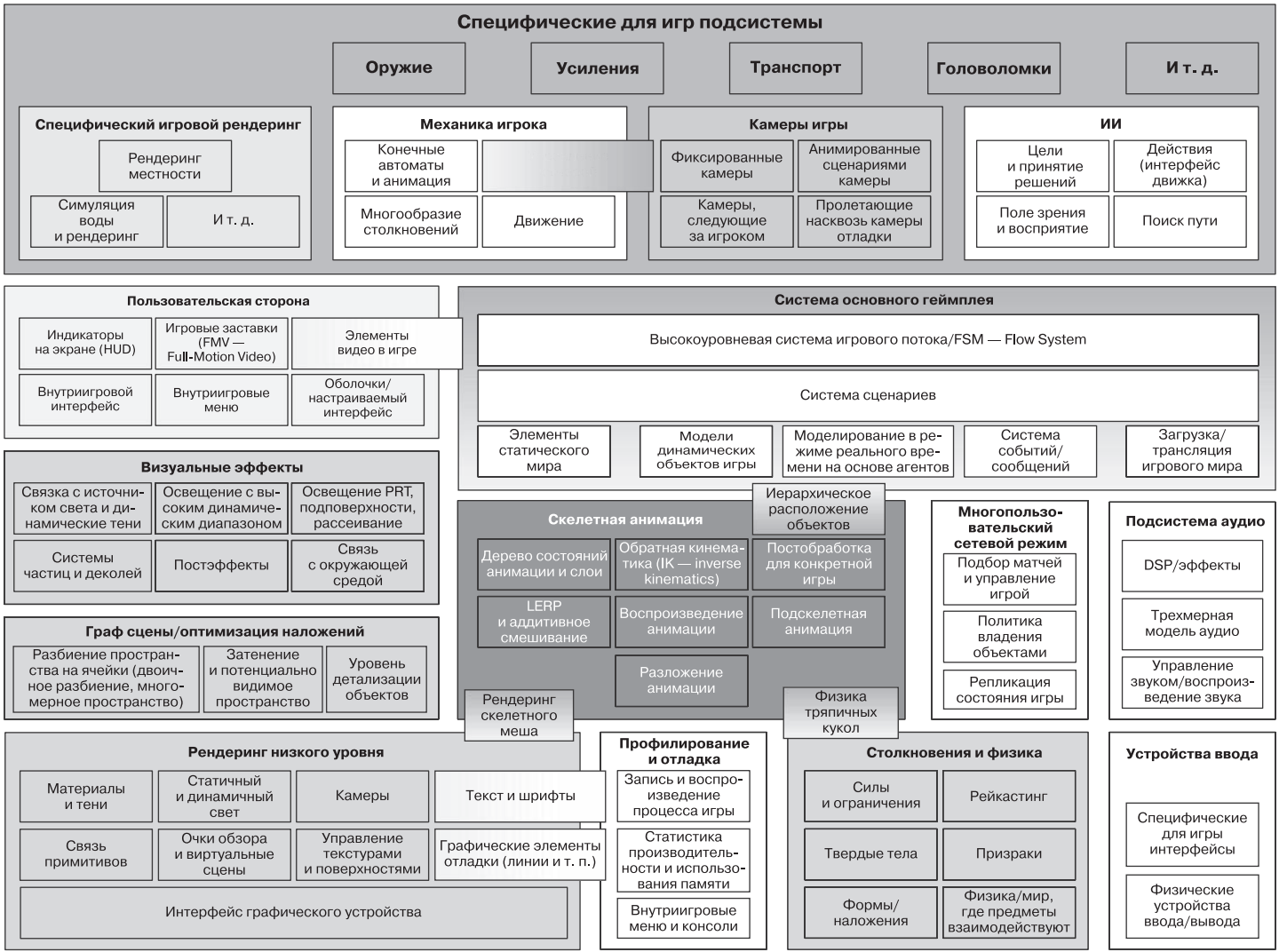
Далее следует краткий обзор компонентов, показанных на рис. 1.16. Дальнейший материал книги будет посвящен гораздо более глубокому изучению каждого из этих компонентов и способов их обычной интеграции в единое функциональное целое.

1.6.1. Целевое аппаратное обеспечение

Целевое аппаратное обеспечение представляет собой компьютерную систему или консоль, на которой будет запускаться игра. К типичным платформам относятся ПК на базе Microsoft Windows, Linux и MacOS, мобильные платформы, такие как Apple iPhone и iPad, смартфоны и планшеты на Android, Sony PlayStation Vita и Amazon Kindle Fire (среди прочих), игровые приставки, такие как Microsoft Xbox, Xbox 360 и Xbox One, Sony PlayStation, PlayStation 2, PlayStation 3 и PlayStation 4, а также Nintendo DS, GameCube, Wii, Wii U и Switch. Большинство тем в этой книге не зависит от платформы, но мы также коснемся некоторых моментов, связанных с разработкой для ПК или консоли, когда эти различия будут важны.

1.6.2. Драйверы устройств

Драйверы устройств — это низкоуровневые программные компоненты, предоставляемые операционной системой или поставщиком оборудования. Драйверы управляют аппаратными ресурсами и ограждают операционную систему и верхние уровни ядра от деталей реализации связи с множеством доступных вариантов аппаратных устройств.


Специфические для игр подсистемы
Оружие
Усиления
Транспорт
Головоломки
И т. д.
Специфический игровой рендеринг
Рендеринг местности
Симуляция воды и рендеринг
И т. д.
Механика игрока
Конечные автоматы и анимация
Многообразие столкновений
Движение
Камеры игры
Фиксированные камеры
Анимированные сценариями камеры
Камеры, следующие за игроком
Пролетающие насквозь камеры отладки
ИИ
Цели и принятие решений
Действия (интерфейс движка)
Поле зрения и восприятие
Поиск пути
Пользовательская сторона
Индикаторы на экране (HUD)
Игровые заставки (FMV — Full-Motion Video)
Элементы видео в игре
Внутриигровой интерфейс
Внутриигровые меню
Оболочки/настраиваемый интерфейс
Система основного геймплея
Высокоуровневая система игрового потока/FSM — Flow System
Система сценариев
Элементы статического мира
Модели динамических объектов игры
Моделирование в режиме реального времени на основе агентов
Система событий/сообщений
Загрузка/трансляция игрового мира
Визуальные эффекты
Связка с источником света и динамические тени
Освещение с высоким динамическим диапазоном
Освещение PRT, подповерхности, рассеивание
Системы частиц и деколей
Постэффекты
Связь с окружающей средой
Скелетная анимация
Иерархическое расположение объектов
Дерево состояний анимации и слои
Обратная кинематика (IK — inverse kinematics)
Постобработка для конкретной игры
LERP и аддитивное смешивание
Воспроизведение анимации
Подскелетная анимация
Разложение анимации
Рендеринг скелетного меша
Физика тряпичных кукол
Многопользовательский сетевой режим
Подбор матчей и управление игрой
Политика владения объектами
Репликация состояния игры
Подсистема аудио
DSP/эффекты
Трехмерная модель аудио
Управление звуком/воспроизведение звука
Граф сцены/оптимизация наложений
Разбиение пространства на ячейки (двоичное разбиение, многомерное пространство)
Затенение и потенциально видимое пространство
Уровень детализации объектов
Рендеринг низкого уровня
Материалы и тени
Статичный и динамичный свет
Камеры
Текст и шрифты
Связь примитивов
Очки обзора и виртуальные сцены
Управление текстурами и поверхностями
Графические элементы отладки (линии и т. п.)
Интерфейс графического устройства
Профилирование и отладка
Запись и воспроизведение процесса игры
Статистика производительности и использования памяти
Внутриигровые меню и консоли
Столкновения и физика
Силы и ограничения
Рейкастинг
Твердые тела
Призраки
Формы/наложения
Физика/мир, где предметы взаимодействуют
Устройства ввода
Специфические для игры интерфейсы
Физические устройства ввода/вывода

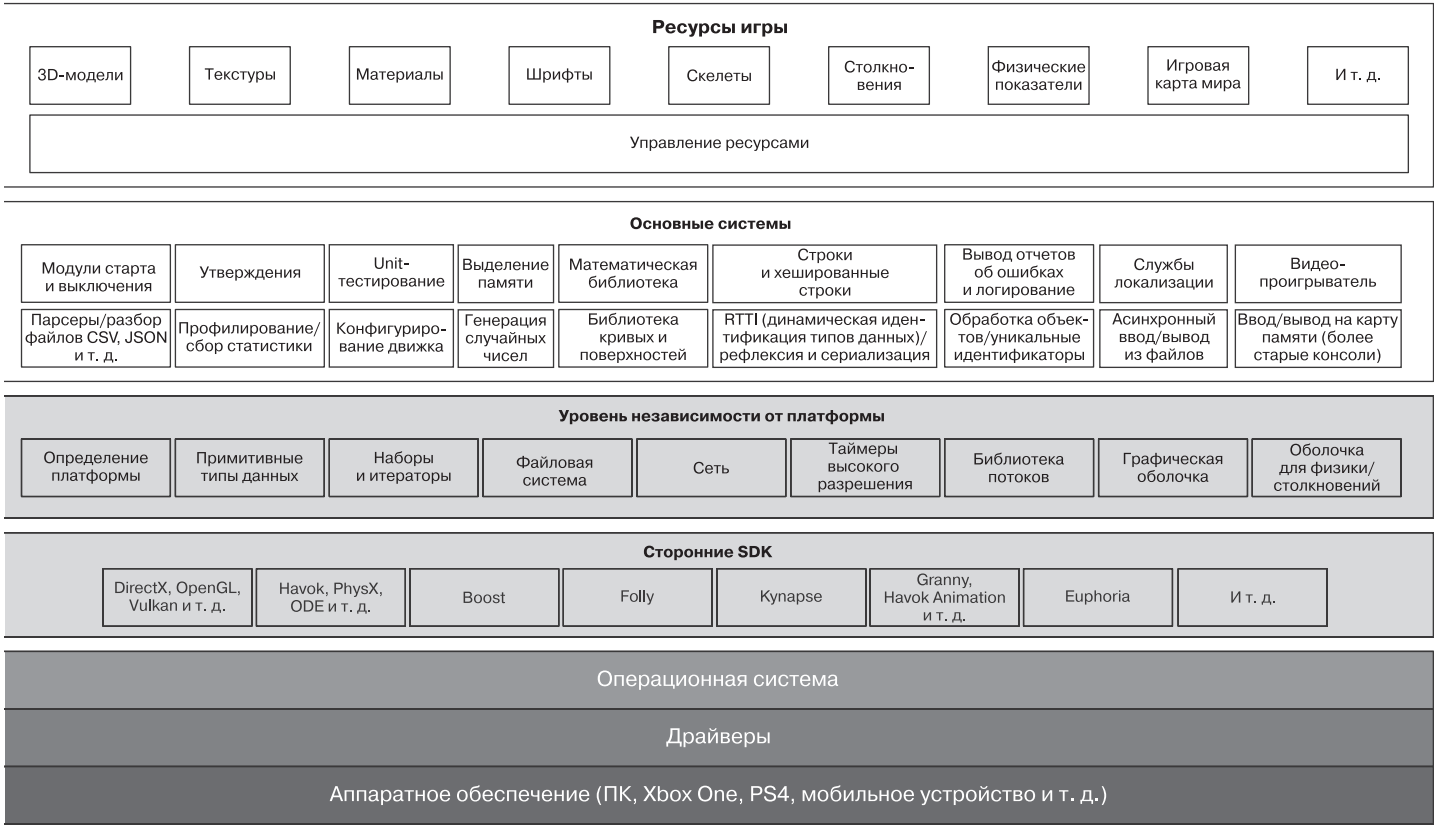


Рис. 1.16. Архитектура среды выполнения движка

1.6.3. Операционная система

На ПК операционная система (ОС) работает постоянно. Она обеспечивает выполнение на компьютере нескольких программ, одной из которых является ваша игра. Операционные системы, такие как Microsoft Windows, для совместного использования оборудования несколькими запущенными программами задействуют подход разделения по времени, известный как вытесняющая многозадачность. Это означает, что компьютерная игра никогда не должна предполагать, что она полностью контролирует оборудование, — она должна искать компромиссы с другими программами в системе.

На ранних консолях операционная система, если она вообще существовала, была просто тонким слоем библиотек, скомпилированных непосредственно с исполняемым файлом игры. На этих системах игра владела всей машиной, пока была запущена. На современных консолях это не так. Операционная система на Xbox 360, PlayStation 3, Xbox One и PlayStation 4 может прерывать выполнение игры или захватывать определенные системные ресурсы, чтобы отобразить онлайн-сообщения или позволить игроку приостановить игру и, например, вызвать пользовательский интерфейс PS4 XMB или панель управления Xbox One. На PS4 и Xbox One ОС постоянно выполняет фоновые задачи, такие как запись видео вашего прохождения, если вы решите поделиться им с помощью кнопки Поделиться на PS4, или загрузка игр, патчей и DLC, чтобы вы весело проводили время, играя во время ожидания. Таким образом, разрыв между разработкой для консоли и для ПК постепенно сокращается (неизвестно, к лучшему или к худшему).

1.6.4. Сторонние SDK и промежуточное ПО

Большинство игровых движков используют сторонние наборы для разработки программного обеспечения (software development kits, SDK) и промежуточное программное обеспечение (рис. 1.17). Функциональный интерфейс, или интерфейс на основе классов, предоставляемый SDK, часто называют интерфейсом прикладного программирования (application programming interface, API). Рассмотрим несколько примеров.

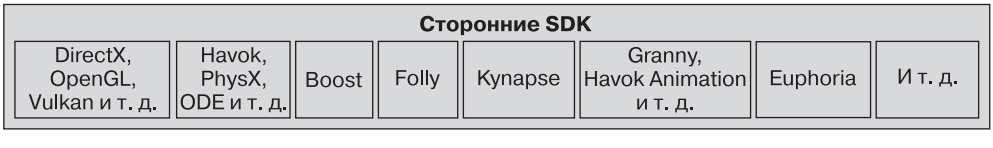


Рис. 1.17. Набор сторонних SDK

Структуры данных и алгоритмы

Как и любые программные системы, игры сильно зависят от структур данных для их хранения и алгоритмов для обработки. Вот несколько примеров сторонних библиотек, которые предоставляют такие виды услуг.

- *Boost* — мощная библиотека структур данных и алгоритмов, разработанная в стиле стандартной библиотеки C++ и ее предшественницы — стандартной библиотеки шаблонов (standard template library, STL). (Онлайн-документация для Boost — это отличное место для изучения программирования в целом!)
- *Folly* — библиотека, используемая в Facebook, назначением которой является расширение стандартной библиотеки C++ и Boost с помощью всевозможных полезных средств с акцентом на максимальном повышении производительности кода.
- *Loki* — мощная универсальная библиотека шаблонов для программирования, которая непременно взорвет ваш мозг!

Стандартная библиотека C++ и STL. Стандартная библиотека C++ также предоставляет многие возможности, которые имеются у сторонних библиотек, таких как Boost. Подмножество стандартной библиотеки, которая реализует классы универсальных контейнеров, такие как `std::vector` и `std::list`, часто называют стандартной библиотекой шаблонов, хотя технически это немного неправильно: STL была написана Александром Степановым и Дэвидом Массером за несколько дней до стандартизации языка C++. Большая часть функциональности этой библиотеки включена в стандартную библиотеку C++. Когда в этой книге используется термин STL, он обычно находится в контексте подмножества стандартной библиотеки C++, которая предоставляет общие классы контейнеров, а не исходный STL.

Графика

Большинство игровых движков построены на основе библиотеки аппаратного интерфейса, например:

- *Glide* — это 3D-графический SDK для старых видеокарт Voodoo. Этот SDK был популярен до эпохи аппаратного преобразования и освещения (Hardware T&L — hardware transform and lighting), которая началась с DirectX 7;
- *OpenGL* — широко используемый переносимый SDK для трехмерной графики;
- *DirectX* — трехмерный графический SDK компании Microsoft и основной конкурент OpenGL;
- *libgcm* — это низкоуровневый прямой интерфейс к графическому оборудованию RSX PlayStation 3, предоставленный Sony в качестве более эффективной альтернативы OpenGL;
- *Edge* — мощный высокоэффективный движок рендеринга и анимации, разработанный Naughty Dog и Sony для PlayStation 3 и используемый рядом сторонних и независимых игровых студий;
- *Vulkan* — это низкоуровневая библиотека, созданная группой Khronos™, которая позволяет программистам игр отправлять порции рендеринга и вычислительные задания GPGPU непосредственно в графический процессор в виде списков команд и обеспечивает детальный контроль над памятью и другими ресурсами, которые распределяются между процессором и графическим процессором. (Подробнее о программировании GPGPU см. в разделе 4.11.)

Столкновения и физика

Обнаружение столкновений и динамика твердого тела (в сообществе разработчиков игр ее называют просто физикой) обеспечиваются следующими хорошо известными SDK:

- *Havok* — это популярный промышленный движок для физики и столкновений;
- *PhysX* — еще один популярный механизм промышленной физики и столкновений, который можно бесплатно загрузить с NVIDIA;
- *Open Dynamics Engine* (ODE) — это хорошо известный пакет физики/столкновений с открытым исходным кодом.

Анимация персонажей

Существует ряд коммерческих анимационных пакетов, включая следующие (но не ограничиваясь ими).

- *Granny*. Популярный инструментарий Granny компании Rad Game Tools включает надежные компоненты экспорта 3D-моделей и анимации для всех основных пакетов 3D-моделирования и анимации, таких как Maya, 3D Studio MAX и т. д., библиотеку среды выполнения для чтения и манипулирования экспортированными моделями и данными анимации, а также мощную систему анимации во время выполнения. На мой взгляд, у Granny SDK самый лучший и самый логичный API анимации из тех, которые я когда-либо видел, среди как коммерческих, так и частных. Особенно он мне нравится обработкой времени.
- *Havok Animation*. Граница между физикой и анимацией становится все более размытой по мере того, как повышается реалистичность персонажей. Компания, выпускающая популярный SDK для физики Havok, решила создать бесплатный SDK для анимации, что сделает разрыв между физикой и анимацией меньше, чем когда-либо.
- *OrbisAnim*. Библиотека OrbisAnim, созданная для PS4 компанией SN Systems совместно с ICE и игровыми командами в Naughty Dog, группой Tools and Technology Sony Interactive Entertainment и группой Advanced Technology Sony в Европе, включает в себя мощный и эффективный механизм анимации и эффективный движок обработки геометрии для рендеринга.

Биомеханические модели персонажей

Endorphin и *Euphoria* — это анимационные пакеты, которые моделируют движение персонажа с использованием продвинутых биомеханических моделей реалистичного движения человека.

Как упоминалось ранее, грань между анимацией персонажа и физикой начинает стираться. Пакеты, такие как Havok Animation, пытаются объединить физику и анимацию традиционным способом, при этом человек-аниматор обеспечивает большую часть движения с помощью такого инструмента, как Maya, а физика

усиливает это движение во время выполнения. Но фирма Natural Motion Ltd. выпустила продукт, который пытается переопределить способы управления движением персонажа в играх и других видах цифровых медиа.

Первый продукт, Endorphin, представляет собой плагин для Maya, который позволяет аниматорам запускать полное биомеханическое моделирование персонажей и экспортировать полученную анимацию, как будто она была создана вручную. Биомеханическая модель учитывает центр тяжести, распределение веса персонажа и детальные знания о том, как реальный человек балансирует и движется под действием силы тяжести и других сил.

Второй продукт, Euphoria, представляет собой версию Endorphin в среде выполнения, предназначенную для создания физически и биомеханически точного движения персонажа во время его выполнения под воздействием непредсказуемых сил.

1.6.5. Уровень независимости от платформы

Большинство игровых движков должны работать более чем на одной аппаратной платформе. Например, компании Electronic Arts и ActivisionBlizzard Inc. всегда разрабатывают игры для самых разных платформ, чтобы охватить максимально возможную долю рынка. Как правило, единственными игровыми студиями, которые не ориентированы как минимум на две разные платформы для каждой игры, являются студии разработки эксклюзивных игр, такие как Sony Naughty Dog и Insomniac. Поэтому большинство игровых движков спроектированы с уровнем независимости от платформы (рис. 1.18). Он располагается поверх уровня аппаратного обеспечения, драйверов, операционной системы и другого стороннего программного обеспечения и экранирует остальную часть движка от большинства тонкостей базовой платформы, оборачивая определенные функции интерфейса в пользовательские функции, над которыми вы, как разработчик игры, будете иметь контроль на каждой целевой платформе.

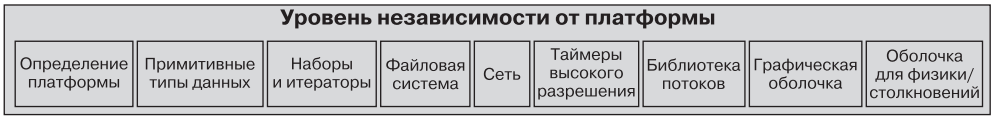


Рис. 1.18. Составляющие уровня независимости от платформы

Есть две основные причины оборачивать функции как часть уровня независимости от платформы вашего игрового движка. Во-первых, некоторые API, например предоставленные операционной системой, или даже определенные функции в старых стандартных библиотеках, таких как стандартная библиотека C, значительно различаются для разных платформ. Оборачивание этих функций обеспечивает остальной части движка единый API для всех целевых платформ. Во-вторых, даже при использовании полностью кросс-платформенной библиотеки, такой как Havok, может потребоваться применить уровень изоляции, чтобы в будущем, например при переходе на другую библиотеку физики или коллизий, иметь меньше проблем.

1.6.6. Основные системы

Каждый игровой движок, да и любое большое и сложное программное приложение на C++, требует целой кучи полезных утилит. Мы будем классифицировать их как основные системы. Типичный слой основных систем показан на рис. 1.19. Вот несколько примеров функций, которые он обычно предоставляет.

- *Утверждения* (assertions) — строки кода, которые вставляют, чтобы отловить логические ошибки и нарушения изначальных предположений программиста. Проверки утверждений обычно удаляются из окончательной сборки игры, которая идет в релиз (утверждения рассматриваются в главе 3).
- *Управление памятью.* Практически каждый игровой движок реализует собственную систему распределения пользовательской памяти для обеспечения высокоскоростного выделения памяти и ее освобождения, а также ограничения негативных последствий ее фрагментации (см. главу 6).
- *Математическая библиотека.* Игры по своей природе используют много математических вычислений. Таким образом, каждый игровой движок применяет минимум одну, а то и несколько математических библиотек. Они предоставляют функции для векторной и матричной математики, кватернионы вращений, тригонометрию, геометрические операции с прямыми, лучами, сферами, трехмерными фигурами и т. д., позволяют работать с кривыми, выполнять численное интегрирование, решать системы уравнений и содержат любые другие функции, которые могут понадобиться программистам.
- *Пользовательские структуры данных и алгоритмы.* Даже если разработчики движка приняли решение полностью положиться на сторонние пакеты, такие как Boost или Folly, то набор инструментов для управления основными структурами данных (связанными списками, динамическими массивами, бинарными деревьями, хеш-таблицами и т. д.) и алгоритмы (поиска, сортировки и т. д.) обычно все равно нужны. Их часто программируют вручную, чтобы свести

Рис. 1.19. Основные системы движка

к минимуму или полностью устранить динамическое выделение памяти и обеспечить оптимальную производительность для целевой платформы.

Подробное обсуждение наиболее распространенных основных систем движка представлено в части II.

1.6.7. Управление ресурсами

Присутствующий в каждом игровом движке в той или иной форме менеджер ресурсов предоставляет универсальный интерфейс (или набор интерфейсов) для доступа к любым типам игровых ресурсов и другим входным данным движка. В некоторых движках этот процесс является централизованным и последовательным (таковы, например, пакеты Unreal, класс OGRE `ResourceManager`). Другие движки используют иной подход, часто оставляя разработчику прямой доступ к необработанным файлам на диске или сжатым архивам, таким как PAK-файлы в Quake. Типичный уровень управления ресурсами изображен на рис. 1.20.

Рис. 1.20. Управление ресурсами

1.6.8. Движок рендеринга

Движок рендеринга — это один из самых больших и сложных компонентов любого игрового движка. Он может быть спроектирован разными способами. Не существует единого общепринятого способа, хотя, как мы увидим, большинство современных движков рендеринга спроектированы в соответствии с некоторыми фундаментальными принципами, в значительной степени определяемыми устройствами для создания трехмерной графики, от которых они зависят.

Один из распространенных эффективных подходов к проектированию рендеринга состоит в том, чтобы использовать многоуровневую архитектуру, подобную представленной далее.

Низкоуровневый рендеринг

Низкоуровневый рендеринг (рис. 1.21) охватывает все средства движка для предварительной обработки. На этом уровне все ориентировано на отображение набора геометрических примитивов максимально быстро, без особого учета того, какие части сцены видны. Этот компонент разбит на различные подкомпоненты, которые мы обсудим в дальнейшем.

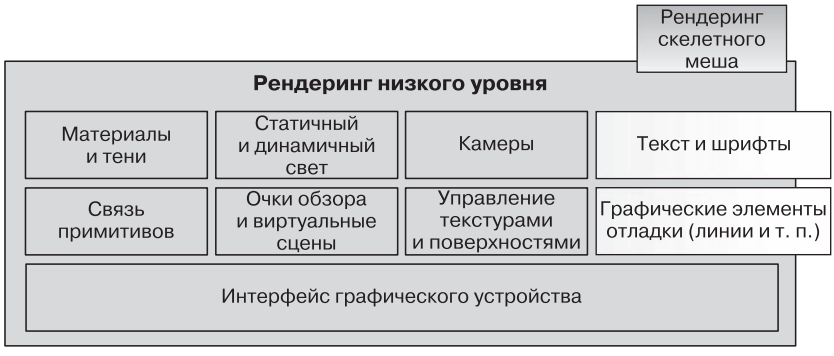


Рис. 1.21. Средства движка для низкоуровневого рендеринга

Интерфейс графического устройства. Графические SDK, такие как DirectX, OpenGL или Vulkan, требуют написания довольно большого количества кода только для перечисления доступных графических устройств, их инициализации, настройки поверхностей рендеринга (обратный буфер, буфер трафарета и т. п.) и т. д. Обычно это обрабатывается компонентом, который я назову интерфейсом графического устройства (хотя в каждом движке используется собственная терминология).

Игровому движку для ПК также необходим код для интеграции компонента рендеринга с циклом сообщений Windows. Вы обычно пишете конвейер сообщений, который обслуживает сообщения Windows, находящиеся в режиме ожидания, или, если их нет, повторяет цикл рендеринга так быстро, как только может. Это связывает цикл опроса ввода с клавиатуры игры с циклом рендеринга для обновления экрана. Такое соединение нежелательно, но, приложив усилия, зависимость реально минимизировать. Позже мы рассмотрим эту тему подробнее.

Другие компоненты рендеринга. Другие компоненты в низкоуровневом рендеринге взаимодействуют для сбора представлений о *геометрических примитивах*, иногда называемых *пакетами рендеринга*, таких как меши, списки линий, списки точек, частицы, участки ландшафта, текстовые строки и все, что вам нужно рисовать и рендерить как можно быстрее.

Низкоуровневое средство визуализации обычно обеспечивает абстракцию области видимости с помощью матрицы связи камеры с миром и параметров трехмерной проекции, таких как поле зрения и расположение ближней и дальней плоскостей отсечения. Низкоуровневый рендеринг также управляет состоянием графического оборудования и шейдеров игры через *систему материалов* и *систему динамического освещения*. Каждый представленный примитив связан с материалом и зависит от *n* динамических источников света. Материал описывает используемые примитивом текстуры, а также то, какие настройки состояния устройства следует применять и какой вершинный и пиксельный шейдеры использовать при рендеринге примитива. Источники света определяют, как динамические расчеты освещения

будут применяться к примитиву. Свет и тени — сложная тема. Мы обсудим ее основы в главе 11 (эта тема подробно рассмотрена во многих превосходных книгах по компьютерной графике, в том числе [2], [16] и [49]).

Граф сцены/оптимизация наложений

Низкоуровневое средство рендеринга рисует всю представленную ему геометрию, не обращая особого внимания на то, является ли она действительно видимой (кроме отбраковки и обрезки задней поверхности в усеченной камере). Компонент более высокого уровня обычно необходим для того, чтобы ограничить число примитивов, представляемых для рендеринга, на основании некоторой формы определения видимости. Этот слой показан на рис. 1.22.

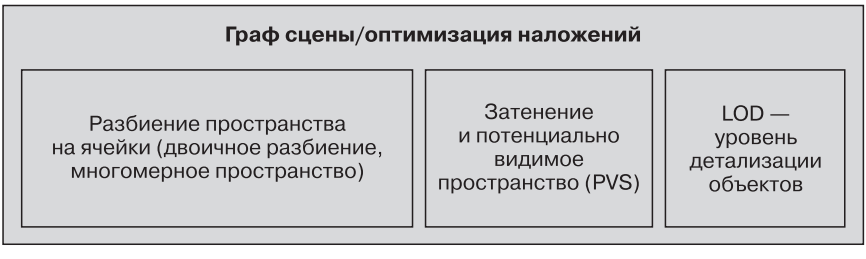


Рис. 1.22. Типичный граф сцены/слой пространственного разбиения для оптимизации наложений

В очень маленьких игровых мирах, вероятно, требуется всего лишь обрезка по пирамиде видимости (то есть удаление объектов, которые не может захватить камера). В больших игровых мирах для повышения эффективности рендеринга можно использовать более продвинутую структуру данных *пространственного разбиения*, чтобы очень быстро определять потенциально видимый набор (potentially visible set, PVS) объектов. Пространственные разбиения способны принимать различные формы, включая двоичное дерево разбиения пространства, дерево квадрантов, дерево октанов, дерево kd или иерархию сфер. Пространственное разбиение иногда называют графом сцены, хотя технически последний представляет собой особый вид структуры данных и не является пространственным разбиением. В этом слое механизма рендеринга также могут применяться порталы или методы обнаружения наложений.

В идеале низкоуровневый рендеринг должен быть полностью независимым от типа используемого пространственного разделения и графа сцены. Это позволяет различным командам разработки повторно задействовать код примитивного представления, но разработать систему определения PVS, соответствующую потребностям игры, над которой работает конкретная команда. Архитектура движка рендеринга с открытым исходным кодом OGRE (www.ogre3d.org) — отличный пример этого принципа в действии. OGRE предоставляет архитектуру графа сцены plug-and-play. Разработчики игр могут выбрать один из множества заранее реализованных графов сцены или создать собственный.

Визуальные эффекты

Современные игровые движки поддерживают широкий спектр визуальных эффектов (рис. 1.23), включая:

- систему частиц (для дыма, огня, брызг воды и т. п.);
- систему деколей (для пулевых отверстий, отпечатков стоп и т. д.);
- зависимость от освещения или окружающей среды;
- динамические тени;
- полноэкранные постэффекты, применяемые после рендеринга 3D-сцены во внеэкранный буфер.

Рис. 1.23. Примеры визуальных эффектов

Вот некоторые примеры полноэкранных постэффектов:

- сопоставление тонов и цвета в расширенном динамическом диапазоне (HDR);
- полноэкранное сглаживание (full scene anti-aliasing, FSAA);
- эффекты коррекции и смещения цвета, включая устранение засветки, увеличение и уменьшение насыщенности и т. д.

Обычно игровой движок имеет отдельный компонент *системы эффектов*, который управляет рендерингом частиц, надписей и других визуальных эффектов по отдельности. Системы частиц и деколей почти всегда являются отдельными компонентами механизма рендеринга и действуют как входные данные для низкоуровневого рендера. В то же время зависимости от освещения, окружающей среды и тени обычно обрабатываются внутри самого движка рендеринга. Полноэкранные постэффекты реализованы либо как неотъемлемая часть системы рендеринга, либо как отдельный компонент, который работает с выходными буферами этой системы.

Пользовательская графика

В большинстве игр используется 2D-графика, накладываемая на 3D-сцену для различных целей. Сюда относятся:

- игровые *индикаторы на экране* (HUD);
- внутриигровые меню, консоль и/или другие средства разработки, которые могут поставляться или не поставляться с конечным продуктом;

- *графический интерфейс пользователя* (GUI), позволяющий игроку управлять инвентарем своего персонажа, настраивать юниты для битвы или выполнять другие сложные игровые задачи.

Этот слой показан на рис. 1.24. Подобная двухмерная графика обычно реализуется рисованием текстурированных квадов (quad) (пар треугольников) с помощью ортогональной проекции. Они также могут быть отрисованы как полноценное 3D, но как билборды, чтобы всегда находиться перед камерой.

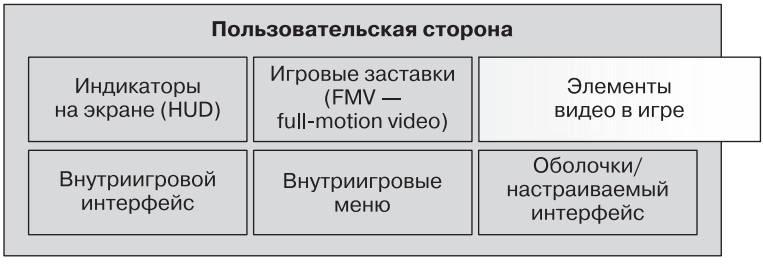


Рис. 1.24. Графика на стороне пользователя

В этот слой мы включили систему *игровых роликов* (full-motion video, FMV). Она отвечает за воспроизведение записанных ранее полноэкранных фильмов (либо отрисованных с помощью системы рендеринга игрового движка или другого пакета рендеринга).

Связанная система — это система игрового кинематографа (in-game cinematics, IGC). Данный компонент обычно позволяет создавать кинематографические эпизоды в самой игре в полноценном 3D. Например, когда игрок гуляет по городу, разговор между двумя ключевыми персонажами может быть реализован как внутриигровой кинематографический эпизод. IGC могут включать или не включать персонажа (ей) игрока. Они могут быть сделаны как преднамеренное отступление, во время которого игрок не контролирует происходящее, или же так тонко интегрированы в сюжет, что он может не понять, что воспроизводится IGC. Разработчики некоторых игр, например Naughty Dog в *Uncharted 4: A Thief's End*, перестали использовать предварительно отрендеренные фильмы и отрисовывают все кинематографические моменты в игре в режиме реального времени.

1.6.9. Инструменты профилирования и отладки

Игры представляют собой системы реального времени, поэтому разработчикам часто приходится профилировать производительность своих игр, чтобы оптимизировать ее. А поскольку ресурсов памяти, как правило, недостаточно, они активно используют инструменты анализа памяти. Слой профилирования и отладки (рис. 1.25) включает в себя эти инструменты и внутриигровые средства отладки, такие как отладка графики, система меню или консоль в игре, а также позволяет записывать и воспроизводить игровой процесс в целях тестирования и отладки.

Существует множество хороших инструментов для профилирования программного обеспечения общего назначения, в том числе:

- *VTune* компании Intel;
- IBM *Quantify and Purify* (часть набора инструментов *PurifyPlus*);
- *Insure++* компании Parasoft;
- *Valgrind* Джулиана Сьюарда и команды разработчиков Valgrind.

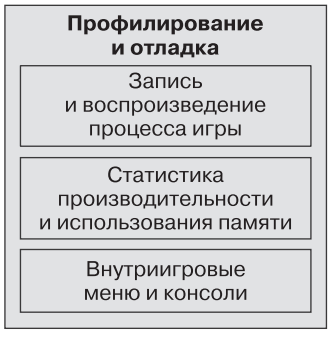


Рис. 1.25. Инструменты профилирования и отладки

Большинство игровых движков включают также набор пользовательских инструментов для профилирования и отладки, например одну или несколько следующих позиций:

- механизм для ручной правки кода, позволяющий синхронизировать определенные части кода;
- средство для отображения статистики профилирования на экране во время игры;
- средство для выгрузки статистики производительности в текстовый файл или электронную таблицу Excel;
- средство для определения того, сколько памяти используют движок в целом и каждая его подсистема в отдельности, и отображения этих сведений на экране;
- возможность сделать дамп используемой памяти, отметку о пиковой точке и показать статистику утечек по окончании игры и/или во время нее;
- инструменты, позволяющие вставить операторы печати для отладки по всему коду, а также возможность включать или отключать различные категории вывода отладочной информации и контролировать уровень многословности выходных данных;
- возможность записывать игровые события, а затем воспроизводить их. Это сложно сделать правильно, но если удастся, то вы получите очень полезный инструмент для поиска и исправления ошибок.

PlayStation 4 предоставляет мощную функцию дампа ядра, чтобы помочь программистам в отладке сбоев. PlayStation 4 всегда записывает последние 15 секунд игры на видео, чтобы игроки могли поделиться своим опытом с помощью кнопки Поделиться на контроллере. Из-за этого средство дампа ядра PS4 автоматически предоставляет программистам не только полный стек вызовов того, что программа делала в случае сбоя, но и снимок экрана с моментом сбоя и 15-секундную видеозапись, показывающую, что происходило незадолго до крушения игры. Дампы ядра могут автоматически загружаться на серверы разработчиков игр при каждом сбое игры даже после ее релиза. Появление этих средств — настоящая революция в области анализа и устранения сбоев.

1.6.10. Столкновения и физика

Обнаружение столкновений важно для каждой игры. Без этого объекты проникали бы друг в друга и было бы невозможно взаимодействовать с ними в виртуальном мире ни одним из разумных способов. В некоторых играх есть также реалистичная или полуреалистичная динамическая симуляция. В игровой индустрии ее называют системой физики, хотя в действительности более уместен термин «динамика твердого тела», потому что мы обычно касаемся только движения (кинематики) твердых тел и сил и моментов (динамики), вызывающих это движение. Этот слой показан на рис. 1.26.

Рис. 1.26. Подсистема столкновений и физики

Столкновение и физика обычно довольно тесно связаны. Это вызвано тем, что при обнаружении столкновений они почти всегда разрешаются как часть логики интеграции физики и соблюдения ограничений. Сейчас очень немногие игровые компании пишут собственные движки столкновений/физики. Вместо этого обычно в движок интегрируют сторонний SDK.

- *Havok* — современный золотой стандарт отрасли. Он многофункционален и хорош по всем параметрам.
- *PhysX* компании NVIDIA — еще один отличный движок для обработки столкновений и динамики. Он был интегрирован в Unreal Engine 4, а также доступен бесплатно как самостоятельный продукт для разработки игр для ПК. Изначально PhysX был разработан как интерфейс к физическому ускорителю Ageia. Теперь этот SDK принадлежит и распространяется NVIDIA, и компания адаптировала его для работы на своих последних графических процессорах.

Доступны также физические движки с открытым исходным кодом. Возможно, самый известный из них — Open Dynamics Engine (ODE) (для получения дополнительной информации см. www.ode.org). I-Collide, V-Collide и RAPID — популярные

некоммерческие механизмы обнаружения столкновений. Все они разработаны в Университете Северной Каролины (UNC) (больше информации — на www.cs.unc.edu/~geom/I_COLLIDE/index.html и www.cs.unc.edu/~geom/V_COLLIDE/index.html).

1.6.11. Анимация

Любая игра, в которой есть органические или полуорганические персонажи (люди, животные, мультипликационные персонажи или даже роботы), нуждается в системе анимации. В играх используются пять основных типов:

- анимация спрайтов/текстур;
- анимация иерархии твердого тела;
- скелетная анимация;
- вершинная анимация;
- морфируемая целевая анимация.

Скелетная анимация позволяет создавать детализированный трехмерный каркас персонажа, используя относительно простую систему костей. По мере движения костей вершины трехмерного каркаса перемещаются вместе с ними. Хотя в некоторых движках применяется морфируемая и вершинная анимация, скелетная является наиболее распространенным методом анимации в современных играх, поэтому в книге мы сосредоточимся именно на ней. Типичная система скелетной анимации показана на рис. 1.27.

Рис. 1.27. Подсистема скелетной анимации

На рис. 1.16 вы можете заметить, что компонент рендеринга скелетного каркаса заполняет разрыв между рендерингом и системой анимации. Здесь наблюдается тесное сотрудничество, но интерфейс очень хорошо определен. Система анимации создает положение каждой кости в скелете, а затем эти положения передаются в механизм рендеринга в виде палитры матриц. Средство рендеринга преобразует каждую вершину с помощью матрицы или матриц в палитре, чтобы сгенерировать окончательное смешанное положение вершины. Этот процесс известен как *скиннинг* (skinning), то есть создание кожи.

При использовании *тряпичных кукол* возникает тесная связь между анимацией и физикой. Тряпичная кукла — это мягкий (часто мертвый) анимационный персонаж, движение тела которого моделируется физической системой. Система физики определяет положение и ориентацию различных частей тела, рассматривая их как связанную систему твердых тел. Система анимации вычисляет палитру матриц, которая необходима движку рендеринга для отрисовки персонажа на экране.

1.6.12. Устройства ввода

Каждая игра должна обрабатывать входные данные от игрока, полученные от различных *устройств ввода* (human interface devices, HID):

- клавиатуры и мыши;
- джойстика;
- других специализированных игровых контроллеров, таких как рули, удочки, танцевальные площадки, Wiimote и т. д.

Мы иногда называем этот компонент компонентом *ввода/вывода игрока*, потому что также можем обеспечивать *вывод* через HID, например принудительную обратную связь/вибрацию на джойстике или звук, воспроизводимый Wiimote. Типичный уровень HID показан на рис. 1.28.

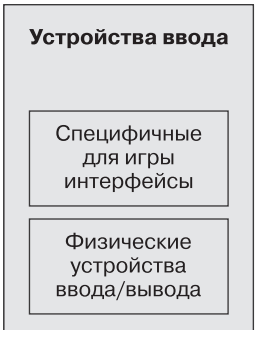


Рис. 1.28. Система ввода/вывода игрока, также известная как уровень устройств ввода

Компонент движка HID часто спроектирован так, чтобы отделить низкоуровневые детали игровых контроллеров на конкретной аппаратной платформе от элементов управления высокого уровня. Он перенаправляет необработанные данные, поступающие от аппаратных устройств, создавая мертвую зону вокруг центральной точки каждого джойстика, разделяет нажатия кнопок, обнаруживает события нажатия кнопок и их отключения, интерпретирует и сглаживает входные данные акселерометра (например, от контроллера PlayStation Dualshock) и делает многое другое. Он часто предоставляет механизм, позволяющий игроку настроить управление — связь между кнопками и логическими игровыми функциями. Иногда он также включает в себя систему обнаружения синхронного нажатия нескольких кнопок одновременно, комбинаций нажатий (кнопки нажимаются последовательно в течение определенного периода времени) и жестов (последовательности входных сигналов от кнопок, джойстиков, акселерометров и т. д.).

1.6.13. Аудио

Для любого игрового движка звук так же важен, как и графика. К сожалению, аудио часто уделяют меньше внимания, чем подсистемам рендеринга, физики, анимации, искусственного интеллекта и геймплею. Наглядный пример: программисты часто

пишут код, когда колонки выключены! (Я знал немало программистов, у которых вообще *не было* динамиков или наушников.) Тем не менее ни одна великолепная игра не обходится без потрясающего звукового движка. Уровень компонента звука проиллюстрирован на рис. 1.29.

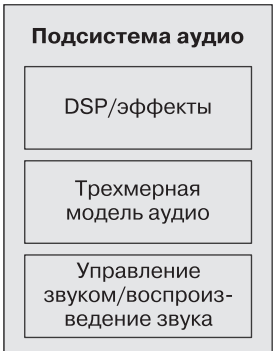


Рис. 1.29. Уровень подсистемы аудио

Аудиодвижки сильно различаются по сложности. Аудиодвижок Quake довольно прост, и команды разработки обычно дополняют его своими функциями или полностью заменяют собственным решением. Unreal Engine 4 предоставляет довольно надежный движок 3D-рендеринга звука (о нем подробно рассказывается в источнике [45]), хотя набор его функций ограничен. И многие команды разработчиков, вероятно, захотят дополнить и настроить его, чтобы он обеспечивал больше функций, специфичных для игры. Для платформ DirectX (ПК, Xbox 360, Xbox One) Microsoft предоставляет отличный аудиодвижок в среде выполнения под названием XAudio2. Electronic Arts разработала усовершенствованный мощный звуковой движок, который называется SoundR!OT. Совместно с первоклассными студиями, такими как Naughty Dog, Sony Interactive Entertainment (SIE), предоставила мощный 3D-движок Scream, который использовался во многих играх для PS3 и PS4, включая *Uncharted 4: A Thief's End* и *The Last of Us: Remastered*. Но даже если игровая команда использует существующий звуковой движок, каждая игра требует разработки специального программного обеспечения, совместной работы, тонкой настройки и внимания к деталям для получения высококачественного звука в конечном продукте.

1.6.14. Многопользовательская онлайн-игра (игра по сети)

Многие игры позволяют нескольким игрокам играть в одном виртуальном мире. Многопользовательские игры представлены как минимум в четырех основных вариантах.

- *Одноэкранные*. Два или более устройства управления (джойстики, клавиатуры, мыши и т. п.) подключены к одному игровому автомату, компьютеру или консоли. В одном виртуальном мире обитают несколько персонажей, а одна камера удерживает всех персонажей в кадре одновременно. Примеры этого стиля игры — *Smash Brothers*, *Lego Star Wars* и *Gauntlet*.
- *С разделенным экраном*. Персонажи такой многопользовательской игры живут в одном виртуальном мире с несколькими HID, прикрепленными к одному устройству, но каждый со своей камерой, а экран разделен на секции, так что каждый игрок может видеть своего персонажа.

- *Сетевые*. Несколько компьютеров или консолей объединены в сеть, причем за каждым из устройств находится один из игроков.
- *Многопользовательские онлайн-игры*. Сотни тысяч пользователей могут одновременно играть в гигантском, постоянно существующем виртуальном мире, который поддерживается мощным кластером центральных серверов.

Многопользовательский сетевой режим представлен на рис. 1.30.

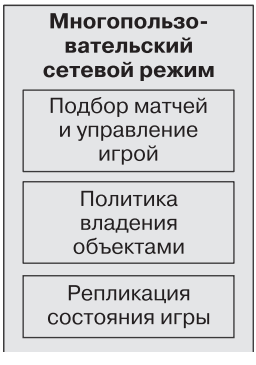


Рис. 1.30. Подсистема многопользовательского сетевого режима

Многопользовательские игры во многом схожи с однопользовательскими. Однако поддержка нескольких игроков может сильно повлиять на архитектуру некоторых компонентов игрового движка. Это воздействует на объектную модель игрового мира, рендеринг, систему ввода пользователя, систему управления игроком и систему анимации. Встраивание многопользовательских функций в уже существующий однопользовательский движок вполне реально, хотя может оказаться очень сложной задачей. Тем не менее многие игровые команды успешно это проделывали. Но они говорят, что если вам доступна такая роскошь, то лучше проектировать многопользовательские функции с первого дня разработки.

Интересно, что обратное преобразование — из многопользовательской игры в однопользовательскую — задача весьма тривиальная. Многие игровые движки обрабатывают однопользовательский режим как частный случай многопользовательского, где только один игрок. Движок Quake хорошо известен своим режимом «клиент поверх сервера», в котором одно приложение, работающее на одном ПК, выступает в роли клиента и сервера в однопользовательском режиме.

1.6.15. Система основного геймплея

Термин *«геймплей»* (игровой процесс) относится к действию, которое происходит в игре, к правилам, управляющим виртуальным миром, в котором происходит игра, к способностям персонажа (-ей) игрока (известны также как *механика игрока*) и других персонажей и объектов в мире, а также к целям и задачам игрока (-ов). Игровой процесс обычно реализуется либо на языке программирования, на котором написана остальная часть движка, либо на языке сценариев высокого уровня, иногда на обоих сразу. Чтобы преодолеть разрыв между кодом игрового процесса и низкоуровневыми системами движка, которые мы обсуждали ранее, большинство игровых движков вводят уровень, который я назову *уровнем основы игрового процесса* (из-за отсутствия стандартизированного названия). Он предоставляет набор основных средств, на которых можно удобно реализовать логику для конкретной игры (рис. 1.31).

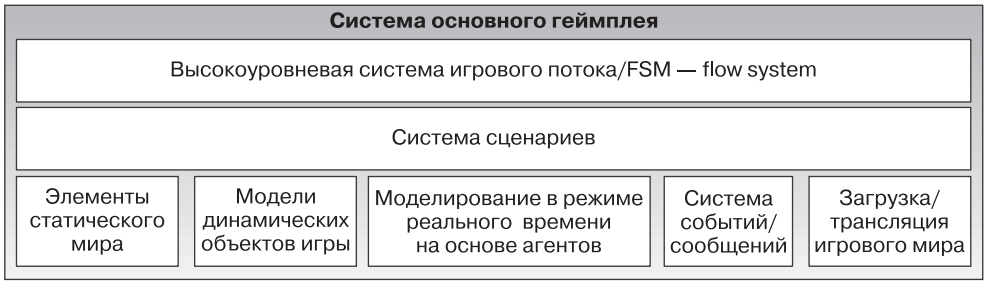


Рис. 1.31. Средства системы основного геймплея

Игровые миры и модели объектов

Уровень основы игрового процесса вводит понятие игрового мира, содержащего как статические, так и динамические элементы. Содержимое мира обычно моделируется объектно-ориентированным способом (часто, но не всегда с использованием объектно-ориентированного языка программирования). В этой книге набор типов объектов, составляющих игру, называется *моделью игровых объектов*. Она обеспечивает моделирование в реальном времени разнородных объектов в виртуальном игровом мире.

Основные типы игровых объектов:

- статическая геометрия фона — здания, дороги, рельеф местности (часто это особый случай) и т. п.;
- динамические твердые тела, такие как камни, банки с газировкой, стулья и т. д.;
- игровые персонажи;
- неигровые персонажи (NPC);
- оружие;
- снаряды;
- транспортные средства;
- источники света, которые могут присутствовать в динамической сцене во время выполнения или использоваться только для статического освещения в автономном режиме;
- камеры.

И этот список можно продолжать.

Модель игрового мира тесно связана с *программной объектной моделью*, которая способна охватывать весь движок. Термин «программная объектная модель» относится к набору функций языка, его политик и соглашений, используемых для реализации части объектно-ориентированной программы. В данном контексте программная объектная модель дает ответы на следующие вопросы.

- Ваш движок написан в объектно-ориентированном стиле?
- Какой язык используется: C, C++, Java, OCaml?
- Как будут организованы статическая иерархия классов, единая огромная иерархия, несколько слабее связанных компонентов?
- Вы будете использовать шаблоны и разработку на основе определенной политики (policy-based design) или классический полиморфизм?
- Как будут организованы ссылки между объектами, старые добрые прямые указатели, умные указатели, обработчики?
- Как будут идентифицироваться объекты: только по адресу в памяти, по имени, по глобальному уникальному идентификатору (GUID)?
- Как будет управляться время жизни объектов?
- Как изменяются состояния игровых объектов с течением времени?

Мы глубже исследуем программные объектные модели и игровые модели объектов в разделе 16.2.

Система событий

Игровые объекты обязаны общаться друг с другом. Это достигается разными способами. Например, объект, отправляющий сообщение, может непосредственно вызвать функцию объекта-получателя. Архитектура на основе событий, на которой обычно основан любой пользовательский интерфейс, также является распространенным вариантом организации взаимодействия объектов. В управляемой событиями системе отправитель создает небольшую структуру данных, называемую *событием* или *сообщением* и содержащую тип сообщения и любые данные в качестве аргумента, которые следует отправить. Событие передается объекту-получателю путем вызова его функции — *обработчика события*. Также события могут быть сохранены в очереди для обработки в будущем.

Система сценариев

Многие игровые движки используют язык сценариев, чтобы упростить и ускорить разработку геймплея и контента для конкретной игры. Без языка сценариев вы должны перекомпилировать игру каждый раз, когда вносятся изменения в логику или структуры данных, используемых в движке. Но когда язык сценариев интегрирован в движок, изменения в игровой логике и данных могут быть выполнены путем изменения и перезагрузки кода сценария. Некоторые движки позволяют перезагружать скрипт, пока игра продолжает работать. Другие требуют, чтобы она была закрыта для перекомпиляции скрипта. Но в любом случае это намного быстрее, чем если бы вам пришлось перекомпилировать исполняемый файл игры при каждом изменении.

Основа искусственного интеллекта

Традиционно искусственный интеллект попадает в область, специфичную для игр, — обычно его не считали частью игрового движка как такового. Совсем недавно, однако, игровые компании обнаружили закономерности, которые отмечаются почти в каждой системе ИИ, и их постепенно начинают относить к компетенции самого движка.

Например, компания под названием Kynogon разработала промежуточный пакет SDK под названием Kynapse, который предоставляет бóльшую часть низкоуровневой технологии, необходимой для создания коммерчески жизнеспособного игрового ИИ. Эта технология была приобретена компанией Autodesk и заменена полностью переработанным пакетом промежуточного программного обеспечения ИИ под названием Gameware Navigation, разработанным той же командой инженеров, которая изобрела Kynapse. Этот SDK обеспечивает низкоуровневые строительные блоки ИИ, такие как генерация навигационной сети, поиск путей, уклонение от статических и динамических объектов, выявление уязвимостей в игровом пространстве (например, открытого окна, возле которого может быть засада) и хорошо определенный интерфейс между ИИ и анимацией.

1.6.16. Специфические для игры подсистемы

Программисты и разработчики геймплея объединяют свои силы не только для создания базового уровня игрового процесса и других низкоуровневых компонентов движка, но и для реализации функций самой игры. Системы геймплея обычно многочисленны, сильно различаются и специфичны для определенной игры. Как показано на рис. 1.32, эти системы включают механику персонажа игрока, различные системы игровых камер, искусственный интеллект для управления неигровыми персонажами, системы оружия, транспортные средства и т. п., но не ограничиваются ими. Если бы можно было провести четкую грань между движком и игрой, она лежала бы между игровыми подсистемами и слоем основы игрового процесса. На практике эта линия никогда не бывает совершенно четкой. По крайней мере некоторые специфические для игры элементы неизменно просачиваются в слой основы игрового процесса, а иногда даже распространяются на ядро движка.

1.7. Инструменты и конвейер ресурсов

Любой игровой движок должен содержать большое количество данных в виде игровых ресурсов, файлов конфигурации, скриптов и т. д. На рис. 1.33 приведены некоторые типы игровых ресурсов, обычно встречающихся в современных игровых движках. Закрашенные стрелки показывают, как данные передаются от инструментов, применяемых для создания исходных ресурсов, игровому движку. Двойные стрелки обозначают, как различные типы ресурсов связаны с другими ресурсами или используют их.

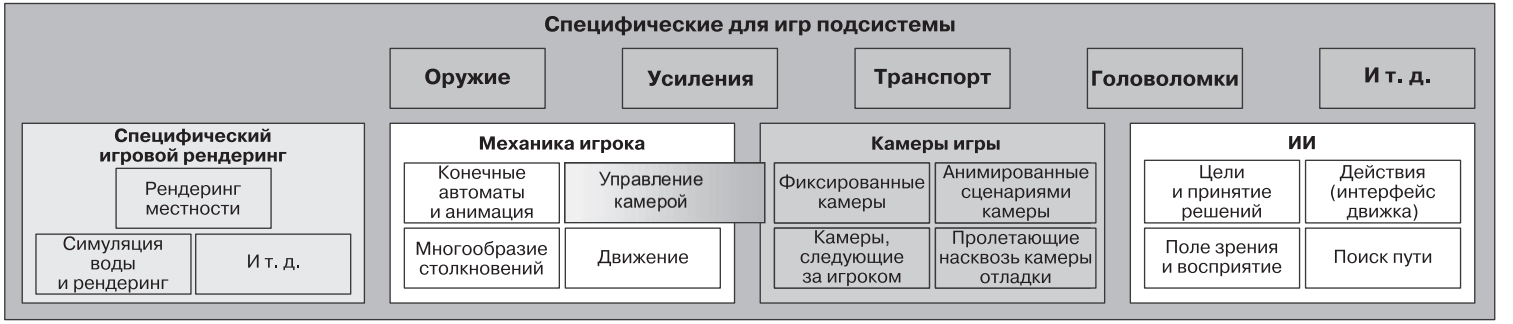
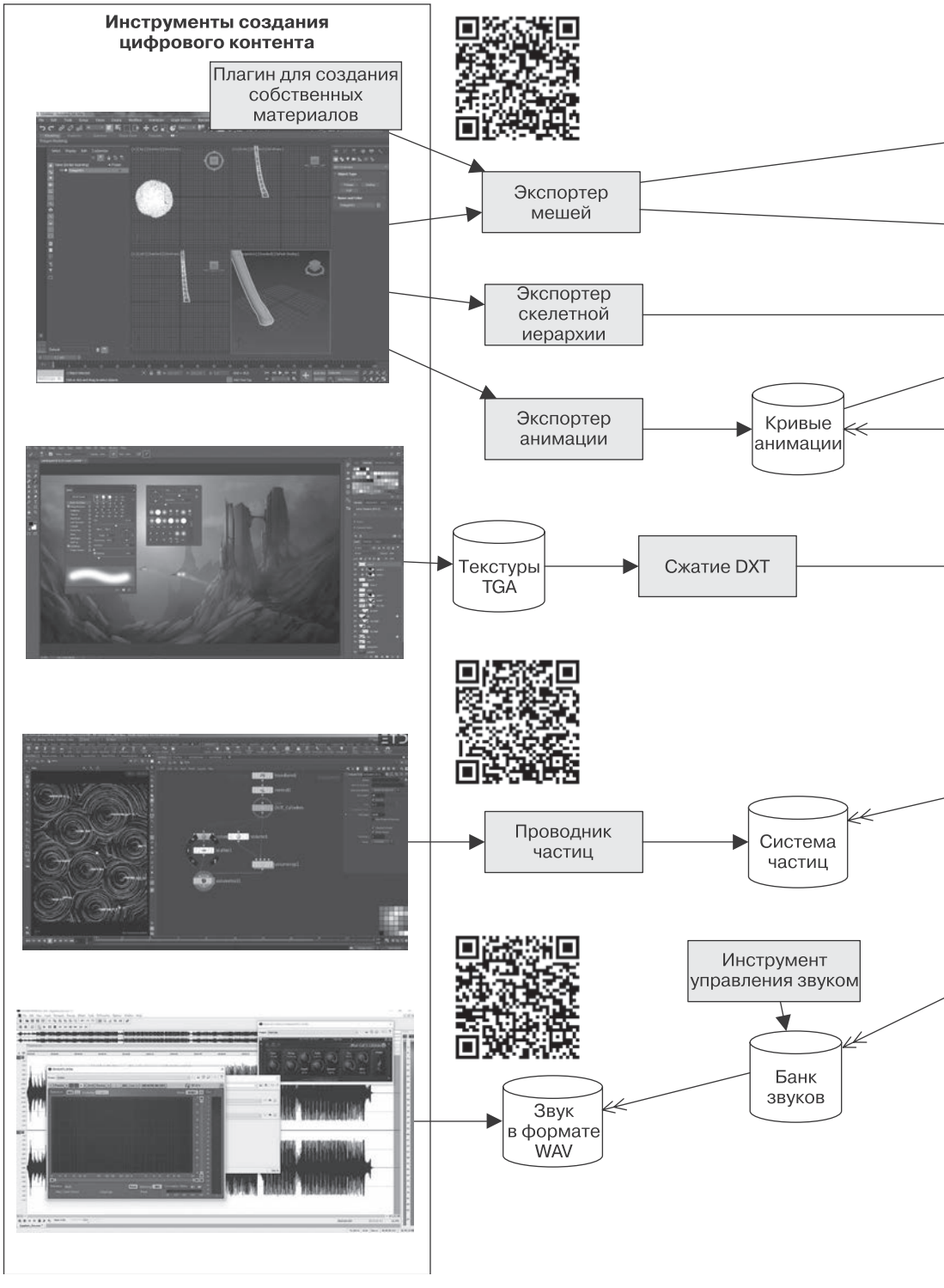


Рис. 1.32. Специфические подсистемы


Инструменты создания цифрового контента
Плагин для создания собственных материалов
Экспортер мешей
Экспортер скелетной иерархии
Экспортер анимации
Кривые анимации
Текстуры TGA
Сжатие DXT
Проводник частиц
Система частиц
Инструмент управления звуком
Банк звуков
Звук в формате WAV

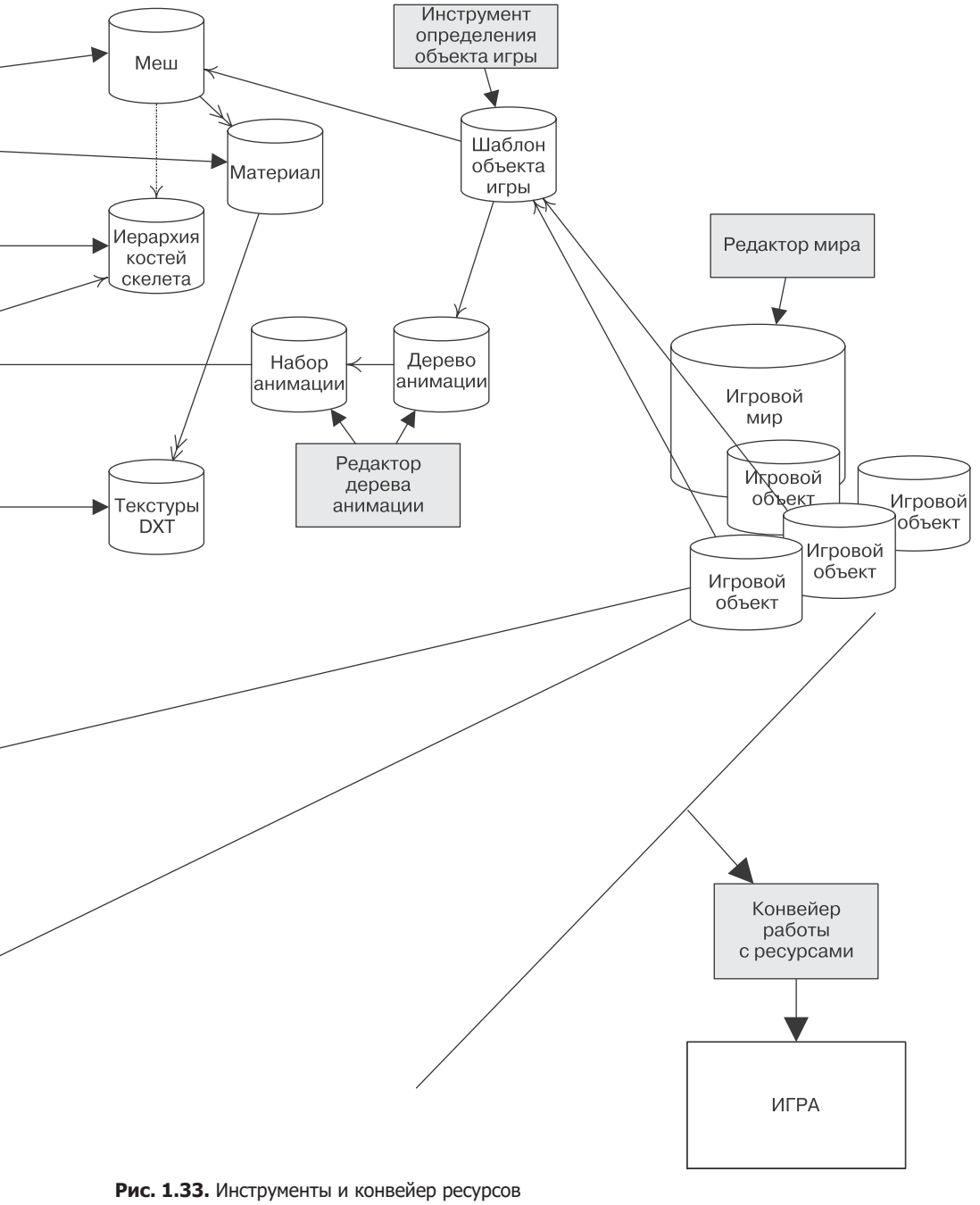


Рис. 1.33. Инструменты и конвейер ресурсов

1.7.1. Инструмент создания контента

Игры по своей природе — это мультимедийные приложения. Входные данные игрового движка представлены в самых разных формах, от параметров трехмерного каркаса до растровых изображений текстуры, данных анимации и аудиофайлов. Все эти исходные данные должны быть созданы и обработаны художниками. Инструменты, которые используют художники, называются *приложениями для создания цифрового контента* (digital content creation, DCC).

Приложение DCC обычно предназначено для создания данных определенного типа, хотя некоторые инструменты могут формировать несколько типов данных. Например, Maya компании Autodesk и ZBrush компании 3ds Max и Pixologic широко используются для создания как трехмерных сеток, так и данных анимации. Adobe Photoshop и его аналоги предназначены для разработки и редактирования растровых изображений (текстур). SoundForge — популярный инструмент для создания аудиоклипов. Некоторые типы игровых данных нельзя получить с помощью готового приложения DCC. Например, большинство игровых движков предоставляют собственный редактор для разметки игровых миров. Но некоторые движки используют для этого готовые инструменты. Я видел, как игровые команды в качестве инструмента разметки мира применяют 3ds Max или Maya с дополнительными плагинами или без них, чтобы облегчить работу пользователю. Спросите большинство разработчиков игр, и они скажут, что все еще помнят времена, когда размещали участки ландшафта по высоте с помощью простого растрового редактора или вводили разметку мира непосредственно в текстовый файл вручную. Инструменты не должны быть красивыми — игровые команды будут использовать все доступные средства, чтобы добиться результата. Тем не менее инструменты должны быть относительно простыми в применении и обязательно надежными, если игровая команда хочет разработать качественный продукт в срок.

1.7.2. Конвейер подготовки ресурсов

Форматы данных, используемые приложениями для создания цифрового контента (DCC), редко подходят для непосредственного применения в игре. На это есть две причины.

- Модель данных, хранящаяся в приложении DCC, обычно намного сложнее, чем требуется игровому движку. Например, Maya хранит направленный ациклический граф (directed acyclic graph, DAG) узлов сцены со сложной сетью взаимосвязей. Она также сохраняет историю всех правок, внесенных в файл, и представляет положение, ориентацию и масштаб каждого объекта в сцене в виде полной иерархии трехмерных преобразований, разделенных на компоненты перемещения, поворота, масштабирования и сдвига. Игровому движку обычно требуется лишь небольшая часть этой информации для визуализации модели в игре.

- Файл приложения DCC часто слишком медленный, чтобы считывать его во время выполнения, а в некоторых случаях он имеет приватный формат, находящийся в частной собственности.

Поэтому данные, создаваемые приложением DCC, обычно экспортируются в более доступный стандартизированный формат или другой формат файла, который уже используется в игре.

После того как данные были экспортированы из приложения DCC, часто требуется обработать их перед отправкой на игровой движок. И если игровая студия распространяет свою игру на более чем одну платформу, промежуточные файлы, вероятно, обрабатываются по-разному для каждой целевой платформы. Например, данные для трехмерного меша могут быть экспортированы в промежуточный формат — двоичный, XML или JSON. Затем он может быть обработан, чтобы объединить меши, которые используют один и тот же материал, или разделить меши, которые слишком велики для обработки движком. Затем данные меши можно упорядочить и упаковать в образ памяти, подходящий для загрузки на конкретную аппаратную платформу.

Конвейер от приложения DCC к игровому движку иногда называют *конвейером подготовки ресурсов* (asset conditioning pipeline, ACP). Каждый игровой движок реализует его в той или иной форме.

3D-модель/меши

Видимая геометрия, которую вы наблюдаете в игре, обычно состоит из треугольных сеток. Некоторые старые игры используют также объемную геометрию, известную как *браши* (brushes). Далее кратко поговорим о типах геометрических данных. Подробное обсуждение методов, используемых для описания и визуализации трехмерной геометрии, вы найдете в главе 11.

3D-модели (меши). Меш — это сложная форма, состоящая из треугольников и вершин. Визуализируемая геометрия может быть построена также из четырехугольников или секций пространства более высокого порядка. Но на современном графическом оборудовании, которое почти исключительно ориентировано на рендеринг растрированных треугольников, все фигуры перед рендерингом должны быть представлены треугольниками.

К мешу обычно применяются один или несколько *материалов*, определяющих визуальные свойства поверхности (цвет, отражающая способность, выпуклость, текстура диффузии и т. д.). В этой книге я буду использовать термин «меш» для обозначения одной визуализируемой фигуры, а «модель» — для составного объекта, который может содержать несколько сеток, данные анимации и другие метаданные для применения в игре.

Меши обычно создаются в программе для трехмерного моделирования, такой как 3ds Max, Maya или SoftImage. Популярный мощный инструмент компании Pixologic под названием ZBrush позволяет создавать меши сверхвысокого разрешения интуитивно понятным способом, а затем преобразовывать их в модель

с более низким разрешением с картами нормалей для приближения деталей высокой частоты.

Модули для экспорта следует писать для извлечения данных из инструмента создания цифрового контента (DCC) (Maya, Max и т. д.) и сохранять на диске в формате, который может быть обработан движком. Приложения DCC предоставляют множество стандартных или полустандартных форматов экспорта, хотя для разработки игр идеально не подходит ни один из них, за исключением, возможно, COLLADA. Поэтому игровые команды часто создают собственные форматы файлов и собственные модули экспорта, чтобы работать с ними.

Геометрия брашей. Геометрия брашей (brush geometry) — это набор выпуклых фигур, каждая из которых определяется несколькими плоскостями. Браши обычно создаются и редактируются непосредственно в редакторе игрового мира. По сути, это подход старой школы к созданию визуализируемой геометрии, но он все еще используется в некоторых движках.

Плюсы:

- быстро и легко создать;
- доступны для разработчиков игр — часто используются, чтобы набросать блоки игрового уровня для создания прототипов;
- могут служить как для обработки столкновений, так и в качестве визуализируемой геометрии.

Минусы:

- низкое разрешение;
- непросто создавать сложные формы;
- не могут поддерживать сочлененные объекты или анимированных персонажей.

Данные скелетной анимации

Скелетный меш — это особый меш, который привязан к иерархии костей и служит для создания шарнирной анимации. Иногда его называют *кожей*, потому что он формирует кожу, которая окружает невидимый основной скелет. Каждая вершина скелетного меша содержит список индексов, указывающих, к какому суставу (-ам) в скелете он привязан. В вершину обычно входит также набор весов суставов, определяющий степень влияния каждого из них на вершину.

Для рендеринга скелетного меша игровому движку требуется три вида данных:

- сам меш;
- иерархия скелета (объединенные имена, родительско-дочерние отношения и базовая поза, в которой он находился, когда был связан с мешем);
- один или несколько анимационных клипов, в которых указывается, как суставы должны двигаться во времени.

Меш и скелет часто экспортируются из приложения DCC в виде одного файла данных. Однако если несколько мешей привязаны к одному скелету, лучше экспортировать скелет как отдельный файл. Анимации обычно экспортируются отдельно, что позволяет загружать в память только те из них, которые используются в данный момент. Однако некоторые игровые движки позволяют экспортировать банк анимаций в виде одного файла, а некоторые даже объединяют меш, скелет и анимации в один монолитный файл.

Неоптимизированная скелетная анимация определяется потоком из матриц размером 3×4, которые берутся с частотой не менее 30 кадров в секунду для каждого из соединений скелета, а их может быть 500 или более для реалистичного человеческого персонажа. Таким образом, данные анимации по своей природе требуют большого объема памяти. Поэтому они почти всегда хранятся в сильно сжатом формате. Схемы сжатия варьируются от движка к движку, а некоторые находятся в частной собственности. Не существует единого стандартизированного формата для данных анимации игры.

Аудиоданные

Аудиоклипы, как правило, экспортируются из Sound Forge или другого инструмента создания аудио различных форматов и с различной скоростью выборки данных. Аудиофайлы могут быть в моно, стерео, 5.1, 7.1 или других многоканальных конфигурациях. Распространены Wave-файлы (`.wav`), часто используются и другие форматы, например файлы PlayStation ADPCM (`.vag`). Аудиоклипы часто организованы в хранилища для упорядочения, обеспечения легкой загрузки в движок и потоковой передачи.

Данные систем частиц

Современные игры используют сложные эффекты частиц. Их разрабатывают художники, которые специализируются на визуальных эффектах. Сторонние инструменты, такие как Houdini, позволяют создавать эффекты, качество которых сравнимо с качеством кино, однако большинство игровых движков не способны отобразить весь спектр эффектов, которые можно выполнить с помощью Houdini. По этой причине многие игровые компании разрабатывают специальный инструмент для редактирования эффектов частиц, который предоставляет только те эффекты, которые фактически поддерживает движок. Этот инструмент позволяет художнику увидеть эффект точно таким, каким он будет отображаться в игре.

1.7.3. Редактор мира

Игровой мир — это то, где все в игровом движке объединяется. Насколько мне известно, коммерчески доступных игровых редакторов мира (то есть игрового мира, эквивалентного Maya или Max) не существует. Тем не менее ряд коммерчески доступных игровых движков предоставляют несколько хороших редакторов мира.

- Какой-то вариант игрового редактора *Radiant* используется большинством игровых движков на основе технологии Quake.
- Движок *Half-Life 2 Source* предоставляет редактор мира под названием *Hammer*.
- *UnrealEd* — редактор мира компании Unreal Engine. Этот мощный инструмент также служит в качестве диспетчера ресурсов для всех типов данных, которые может использовать движок.
- *Sandbox* — мировой редактор в CRYENGINE.

Написать хороший редактор мира сложно, но это чрезвычайно важная часть любого хорошего игрового движка.

1.7.4. База данных ресурсов

Игровые движки работают с широким спектром типов ресурсов, от визуализируемой геометрии до материалов и текстур, анимационных данных и аудио. Эти ресурсы частично определяются необработанными данными, созданными художниками с помощью таких инструментов, как Maya, Photoshop или SoundForge. Каждый ресурс несет в себе также большое количество метаданных. Например, когда аниматор создает анимационный клип в Maya, метаданные предоставляют конвейеру формирования ресурсов и в конечном счете игровому движку следующую информацию:

- уникальный идентификатор, который идентифицирует анимационный клип в среде выполнения;
- имя и путь к каталогу исходного файла Maya (`.ma` или `.mb`);
- диапазон кадров — на каком кадре анимация начинается и заканчивается;
- является анимация цикличной или нет;
- выбранные аниматором техники сжатия и уровень (некоторые ресурсы можно сильно сжать без заметного ухудшения их качества, в то время как другие требуют меньшего сжатия или вообще не сжимаются — только так они будут выглядеть правильно в игре).

Каждому игровому движку требуется какая-то база данных для управления всеми метаданными, связанными с ресурсами игры. Эта база данных может быть реализована с использованием частной реляционной базы данных, такой как MySQL или Oracle, или в виде набора текстовых файлов, управляемых системой контроля версий, например Subversion, Perforce или Git. В книге я буду называть эти метаданные *базой данных ресурсов*.

Независимо от того, в каком формате хранится и как управляется база данных ресурсов, должен быть предусмотрен некоторый интерфейс, позволяющий пользователям создавать и редактировать данные. В Naughty Dog для этой цели написали пользовательский графический интерфейс на C#, названный Builder. Дополнительная информация о Builder и некоторых других пользовательских интерфейсах базы данных ресурсов дается в главе 7.

1.7.5. Некоторые подходы к архитектуре инструментов

Набор инструментов игрового движка можно спроектировать любым способом. Некоторые инструменты могут быть автономными частями программного обеспечения (рис. 1.34), или построенными поверх некоторых нижних уровней, используемых средой выполнения движка (рис. 1.35), или встроенными в саму игру.

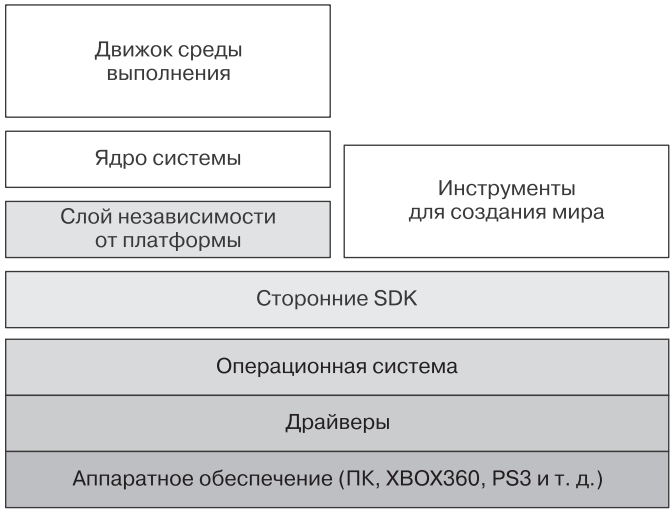


Рис. 1.34. Архитектура автономных инструментов

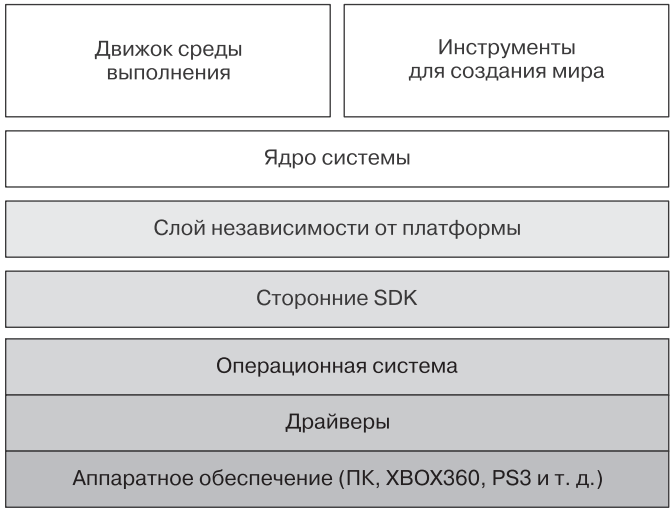


Рис. 1.35. Инструменты, построенные на основе среды разработки, объединенной с игрой

Например, игры на основе движков Quake и Unreal имеют встроенную игровую консоль, которая позволяет разработчикам и создателям модов вводить команды отладки и настройки во время выполнения игры. Наконец, веб-интерфейсы становятся все более популярными для определенных видов инструментов.

В качестве интересного примера: редактор Unreal World и менеджер ресурсов UnrealEd встроены прямо в движок игры, в среду выполнения. Чтобы запустить редактор, вы запускаете игру с аргументом командной строки `editor`. Этот уникальный архитектурный стиль изображен на рис. 1.36.

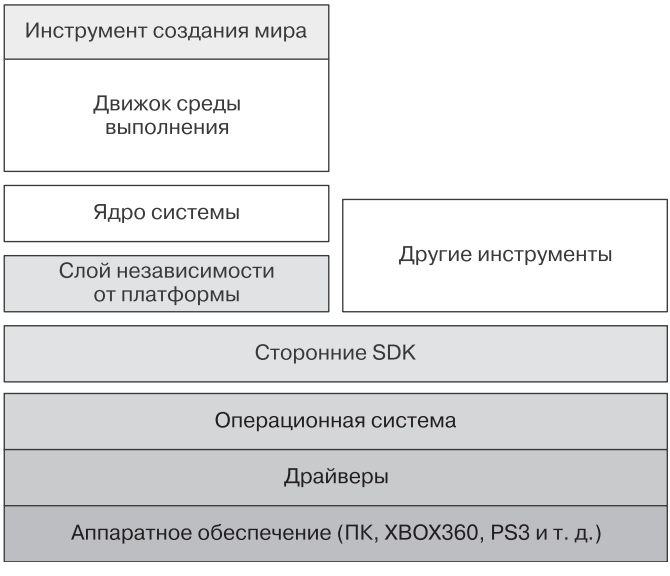


Рис. 1.36. Архитектура инструментов Unreal Engine

Он позволяет инструментам получать полный доступ ко всему спектру структур данных, используемых движком, и избегать общей проблемы, связанной с необходимостью иметь два представления каждой структуры данных: для среды выполнения и для инструментов. Это также означает, что запуск игры из редактора очень быстр, потому что игра фактически уже запущена. Живое редактирование в игре — функция, которую обычно очень сложно реализовать, — может быть относительно легко разработана, когда редактор является частью игры. Однако подобный дизайн с встроенным редактором имеет свои проблемы. Например, когда движок выходит из строя, инструменты также становятся непригодными для использования. Следовательно, тесная связь между движком и инструментами создания ресурсов может замедлить процесс разработки.

Веб-интерфейсы пользователя

Пользовательские веб-интерфейсы быстро становятся нормой для определенных видов инструментов разработки игр. В Naughty Dog мы задействуем несколько веб-интерфейсов. Инструмент локализации Naughty Dog служит интерфейсом

к нашей базе данных локализации. *Tasker* — это веб-интерфейс, используемый всеми сотрудниками Naughty Dog для формулирования задач по разработке игр, их планирования и управления ими, а также общения и совместных действий в процессе производства. Веб-интерфейс, известный как *Connector*, также служит окном к различным потокам отладочной информации, которые генерируются игровым движком во время выполнения. Игра разбрасывает отладочный текст по различным именованным каналам, каждый из которых связан с отдельной системой движка (анимация, рендеринг, ИИ, звук и т. д.). Эти потоки данных собираются в облегченной базе данных Redis. Интерфейс Connect на основе браузера обеспечивает пользователям удобный способ просматривать и фильтровать эту информацию.

Веб-интерфейсы предоставляют ряд преимуществ по сравнению с автономными приложениями с графическим интерфейсом. С одной стороны, веб-приложения, как правило, легче и быстрее разрабатывать и поддерживать, чем автономное приложение, написанное на языке Java, C# или C++. Веб-приложения не требуют специальной установки — пользователю нужен всего лишь совместимый веб-браузер. Пользователям не требуется устанавливать обновления веб-интерфейса — следует только обновить страницу или перезапустить браузер, чтобы получить обновление. Веб-интерфейсы вынуждают нас разрабатывать инструменты, используя архитектуру «клиент — сервер». Это позволяет распространять их среди широкой аудитории. Например, инструмент локализации Naughty Dog доступен сторонним партнерам по всему миру, которые предоставляют нам услуги перевода. Конечно, автономные инструменты по-прежнему применяются, особенно когда требуются специализированные графические интерфейсы, такие как трехмерная визуализация. Но если вашему инструменту нужно только предоставить пользователю редактируемые формы и табличные данные, стоит выбрать веб-инструмент.

2 Полезные инструменты

Прежде чем начать путешествие по захватывающему миру архитектуры игровых движков, важно вооружиться несколькими основными инструментами. В главах 2 и 3 речь пойдет о концепциях и методах разработки программного обеспечения, которые нам понадобятся. В главе 2 поговорим об инструментах, которые большинство профессионалов используют при разработке игр. Затем в главе 3 мы завершим подготовку, рассмотрев некоторые ключевые темы в области объектно-ориентированного программирования, шаблонов проектирования игр и масштабного программирования на C++.

Разработка игр — одна из самых требовательных и обширных областей создания программного обеспечения, поэтому, поверьте, следует хорошо подготовиться, если мы хотим безопасно перемещаться по иногда коварной местности, которую будем исследовать. Для некоторых читателей материал этой и следующей глав будет очень знакомым. Однако я призываю вас не пропускать их полностью. Надеюсь, вы почерпнете из них пару новых идей.

2.1. Контроль версий

Система *контроля версий* — это инструмент, который позволяет нескольким пользователям совместно работать над группой файлов. Он сохраняет историю каждого файла, так что можно отследить изменения и при необходимости откатиться к предыдущей версии. Это дает возможность нескольким пользователям изменять файлы — даже один и тот же файл — одновременно, не мешая друг другу делать свою работу. Контроль версий получил свое название благодаря способности отслеживать историю версий файлов. Его иногда называют *контролем исходного кода*, потому что он в основном используется программистами для управления исходным кодом. Однако контроль версий может применяться и для других типов файлов. Системы контроля версий, как правило, лучше всего справляются с управлением текстовыми файлами по причинам, которые мы рассмотрим в дальнейшем. Однако многие игровые студии задействуют единую систему контроля версий для управления как файлами исходного кода (текстовыми), так и игровыми ресурсами, такими как текстуры, 3D-сетки, анимация и аудиофайлы (обычно двоичные).

2.1.1. Зачем использовать контроль версий

Контроль версий необходим, когда программное обеспечение разрабатывает команда из нескольких инженеров. Контроль версий:

- предоставляет центральный репозиторий, в котором разработчики могут делиться исходным кодом;
- хранит историю изменений, внесенных в каждый исходный файл;
- предоставляет механизмы, позволяющие маркировать конкретные версии кодовой базы и впоследствии извлекать их;
- позволяет разветвить версии кода от основной линии разработки. Эта функция часто используется для создания демонстраций или внесения исправлений в более старые версии программного обеспечения.

Система управления исходным кодом может быть полезна даже для проекта с одним разработчиком. Многопользовательские функции в этом случае не понадобятся, но другие возможности, такие как ведение истории изменений, тегирование версий, создание веток для демонстраций и исправлений, отслеживание ошибок и пр., по-прежнему будут очень полезными.

2.1.2. Распространенные системы контроля версий

Вот наиболее распространенные системы контроля версий, с которыми вы, вероятно, столкнетесь, будучи разработчиком игр.

- *SCCS* и *RCS*. Source Code Control System (SCCS) и Revision Control System (RCS) — две самые старые системы контроля версий. Обе используют интерфейс командной строки. Они распространены в основном на платформах UNIX.
- *CVS*. Concurrent Version System (CVS) — это высокопроизводительная система управления исходным кодом профессионального уровня на основе командной строки, изначально построенная на основе RCS (теперь реализована как самостоятельный инструмент). CVS распространена в системах UNIX, доступна и на других платформах разработки, таких как Microsoft Windows. Имеет открытый исходный код и распространяется по лицензии GNU General Public License (GPL). CVSNT (также известная как WinCVS) — это реализация для Windows, которая основана на CVS и совместима с ней.
- *Subversion*. Система контроля версий с открытым исходным кодом, призванная заменить и улучшить CVS. Поскольку исходный код открытый и, следовательно, бесплатный, это отличный выбор для индивидуальных проектов, студенческих разработок и небольших студий.
- *Git*. Система контроля версий с открытым исходным кодом, которая использовалась во многих легендарных проектах, включая ядро Linux. В модели разработки Git программист вносит изменения в файлы и фиксирует перемены в ветке. Затем он может быстро и легко объединить свои изменения в любую

другую ветку кода, потому что Git знает, как перемотать последовательность изменений и повторно применить их к новой базовой ревизии. Этот процесс в терминах Git называется *rebasing*. В итоге мы получаем систему контроля версий, которая высокоэффективно и быстро работает с несколькими ветвями кода. Git — распределенная система контроля версий. Отдельные программисты способны работать локально большую часть времени, а также легко свести свои изменения в общую кодовую базу. Git очень легко использовать в проекте, над которым работает один человек, тем более что в этом случае не нужно беспокоиться о настройке сервера. Более подробную информацию о Git можно найти по адресу git-scm.com.

- *Perforce*. Система контроля версий профессионального уровня с текстовыми и графическими интерфейсами. Она известна прежде всего своей концепцией списка изменений (changelist). Любое исправление в репозитории однозначно определяется пронумерованным значением changelist, включающим в себя изменения файлов (одного или нескольких тысяч). Поэтому вносимые изменения применяются или для всех файлов из списка, или ни для одного. Perforce используется многими игровыми компаниями, включая Naughty Dog и Electronic Arts.
- *NxN Alienbrain*. Alienbrain — это мощная многофункциональная система контроля версий, разработанная специально для игровой индустрии. Претендует на широкую известность из-за того, что поддерживает очень большие базы данных, содержащие как текстовые файлы с исходным кодом, так и бинарные игровые ресурсы. Имеет настраиваемый пользовательский интерфейс, который можно приспособить для решения задач, стоящих перед конкретными специалистами — художниками, продюсерами или программистами.
- *ClearCase*. Rational ClearCase — система управления исходным кодом профессионального уровня, предназначенная для очень крупных программных проектов. Это мощный инструмент, который использует уникальный пользовательский интерфейс, расширяющий функциональность проводника Windows. Я не видел, чтобы ClearCase широко применялся в игровой индустрии, возможно, потому, что это одна из наиболее дорогих систем контроля версий.
- *Microsoft Visual SourceSafe*. SourceSafe — это легкий пакет управления исходным кодом, который успешно используется в некоторых игровых проектах.

2.1.3. Обзор Subversion и TortoiseSVN

Я решил выделить систему Subversion в этой книге по нескольким причинам. Во-первых, она бесплатна, что всегда приятно. По моему опыту, работает она хорошо и надежно. Центральный репозиторий Subversion довольно прост в настройке, и, как мы увидим, уже есть несколько бесплатных общедоступных серверов с репозиториями, которыми вы можете воспользоваться, если не хотите создавать их самостоятельно. Существует также ряд хороших клиентов Subversion для Windows и Mac, например свободно доступный TortoiseSVN для Windows. Таким образом,

хотя Subversion может оказаться не лучшим выбором для крупного коммерческого проекта (я предпочитаю Perforce или Git для этой цели), считаю, что она идеально подходит для небольших личных и образовательных проектов. Давайте посмотрим, как настроить и использовать Subversion на платформе разработки для Microsoft Windows PC. А также поговорим об основных концепциях, которые можно применить практически к любой системе контроля версий.

Subversion, как и большинство других систем контроля версий, использует архитектуру «клиент — сервер». Сервер управляет центральным репозиторием, в котором хранится иерархия каталогов с полной историей изменений. Клиенты подключаются к серверу и совершают операции, такие как проверка последней версии дерева директорий, внесение изменений в один или несколько файлов, пометка ревизий, ветвление репозитория и т. д. Мы не будем обсуждать здесь настройку серверной части: предполагается, что у вас уже есть настроенный сервер, вместо этого сосредоточимся на настройке и использовании клиента. Вы можете узнать, как настроить сервер Subversion, прочитав главу 6 источника [43]. Однако вам, вероятно, не придется делать это, потому что всегда можно найти бесплатные серверы Subversion. Например, HelixTeamHub предоставляет хостинг кода Subversion по адресу info.perforce.com/try-Perforce-helix-teamhub-free.html, бесплатный для проектов с пятью пользователями или менее и объемом до 1 Гбайт. Beanstalk — еще один хороший хостинг, но с взиманием чисто номинальной ежемесячной платы за используемые ресурсы.

2.1.4. Настройка репозитория с кодом

Самый простой способ начать работу с Subversion — это посетить веб-сайт HelixTeamHub или аналогичного SVN-хостинга и настроить хранилище Subversion. Создайте аккаунт — и вы уже на низком старте. Большинство хостинговых сайтов дают простые инструкции.

Создав репозиторий, вы обычно можете администрировать его прямо на вебсайте хостинг-сервиса. Можете добавлять и удалять пользователей, управлять параметрами и выполнять множество сложных задач. Но что вам действительно нужно сделать — это настроить клиент Subversion и начать использовать свой репозиторий.

2.1.5. Установка TortoiseSVN

TortoiseSVN — популярный клиент для Subversion. Он расширяет функциональность Microsoft Windows Explorer с помощью удобного меню, вызываемого правой кнопкой мыши, и наложенных значков, чтобы показать состояние ваших файлов и папок с управлением версий.

Чтобы установить TortoiseSVN, посетите tortoisesvn.tigris.org. Скачайте последнюю версию со страницы загрузки. При желании можете также загрузить языковой пакет для русского языка. Установите его, дважды щелкнув на файле MSI, который загрузили, и следуйте инструкциям мастера установки.

После этого можете перейти в любую папку в Windows Explorer и щелкнуть правой кнопкой мыши — теперь должны быть видны расширения меню TortoiseSVN. Чтобы подключиться к существующему репозиторию (например, созданному на HelixTeamHub), создайте папку на локальном жестком диске, затем щелкните на ней правой кнопкой мыши и выберите пункт SVN Checkout (Извлечь SVN). Откроется диалоговое окно, показанное на рис. 2.1. В поле URL of repository (URL в хранилище) введите URL-адрес хранилища. Если вы используете HelixTeamHub, это будет https://helixteamhub.cloud/mr3/projects/myprojectname/repositories/subversion/myrepository, где myprojectname — название проекта, которое вы дали ему при создании, а myrepository — имя вашего репозитория SVN.

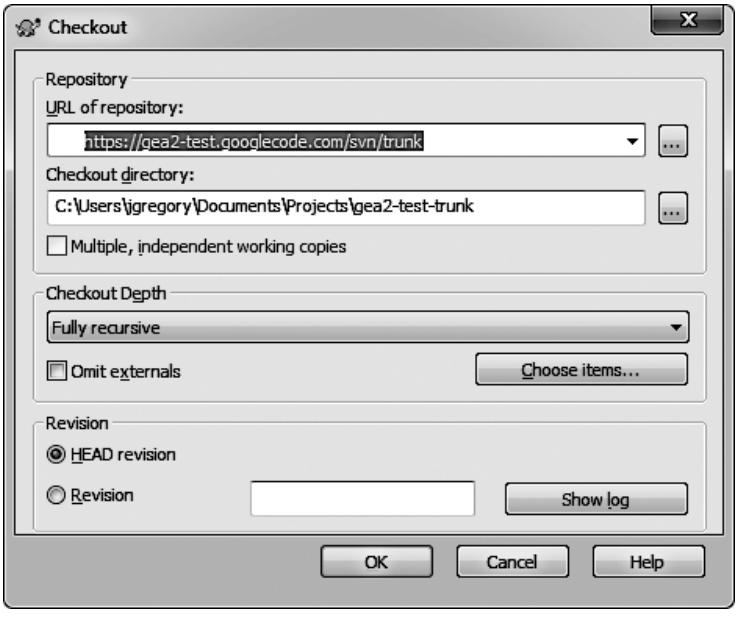


Рис. 2.1. TortoiseSVN, начальное диалоговое окно

Теперь вы должны увидеть диалоговое окно, приведенное на рис. 2.2. Введите свое имя пользователя и пароль. Установив в этом окне флажок Save authentication (Сохранить аутентификацию), вы сможете использовать свой репозиторий, не входя повторно в систему. Выбирайте эту опцию только в том случае, если работаете на личном компьютере, а не на том, который используется несколькими пользователями.

После того как вы аутентифицировались по имени пользователя, TortoiseSVN загрузит (check out) все содержимое вашего репозитория на ваш локальный диск. Если вы только что создали репозиторий, это будет... ничего! Папка, которую вы создали, все еще будет пустой. Но теперь она подключена к вашему хранилищу Subversion на HelixTeamHub (или где там находится ваш сервер). Если вы обновите окно Windows Explorer (нажатием F5), то должны увидеть на папке неболь-

шую зелено-белую галочку. Она указывает, что папка подключена к хранилищу Subversion через TortoiseSVN и что локальная копия хранилища обновлена.

Рис. 2.2. Окно аутентификации пользователя TortoiseSVN

2.1.6. Версии файлов, обновление и коммиты

Как мы уже видели, одна из ключевых целей любой системы управления версиями, такой как Subversion, состоит в том, чтобы позволить нескольким программистам работать на единой базе программного кода, поддерживая центральный репозиторий (мастер-версию) всего исходного кода на сервере. Сервер поддерживает историю версий для каждого файла (рис. 2.3). Эта функция имеет решающее значение для крупномасштабной разработки программного обеспечения большой командой. Например, если кто-то делает ошибку и загружает код, который нарушает сборку, вы можете легко откатиться, чтобы отменить эти изменения (и проверить журнал, чтобы узнать, кто в этом виноват!). Вы также можете сделать снимок кода в том виде, в котором он существовал в любой момент, что позволяет вам работать, демонстрировать или исправлять предыдущие версии программного обеспечения.

Каждый программист загружает локальную копию кода на свой компьютер. В случае применения TortoiseSVN вы получаете первоначальную рабочую копию, извлекая репозиторий, как описано ранее. Периодически вы должны обновлять локальную копию, чтобы отразить любые изменения, которые, возможно, вносили другие программисты. Для этого щелкните правой кнопкой мыши на папке и выберите пункт SVN Update (Обновить SVN) в контекстном меню.

Вы можете работать со своей локальной копией кода, не влияя на других программистов в команде (рис. 2.4). Когда вы готовы поделиться своими изменениями с остальными, вы фиксируете (*коммитите* — commit) эти изменения в центральном репозитории (также это действие известно как *отправка* (submit) или *регистрация* (checking in)). Для этого щелкните правой кнопкой мыши на папке и выберите пункт SVN Commit (Фиксировать SVN) в контекстном меню. Вы увидите окно с просьбой подтвердить изменения (рис. 2.5).

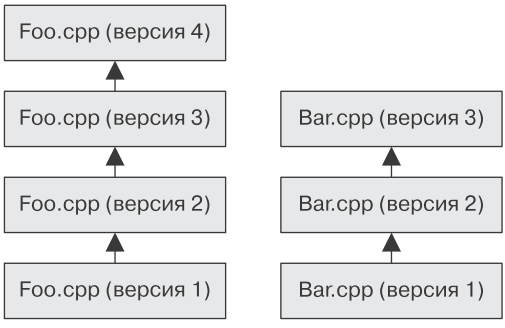


Рис. 2.3. История версий файлов

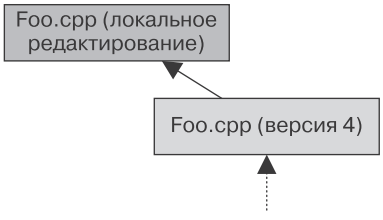


Рис. 2.4. Редактирование локальной копии файла, отслеживаемого системой контроля версий

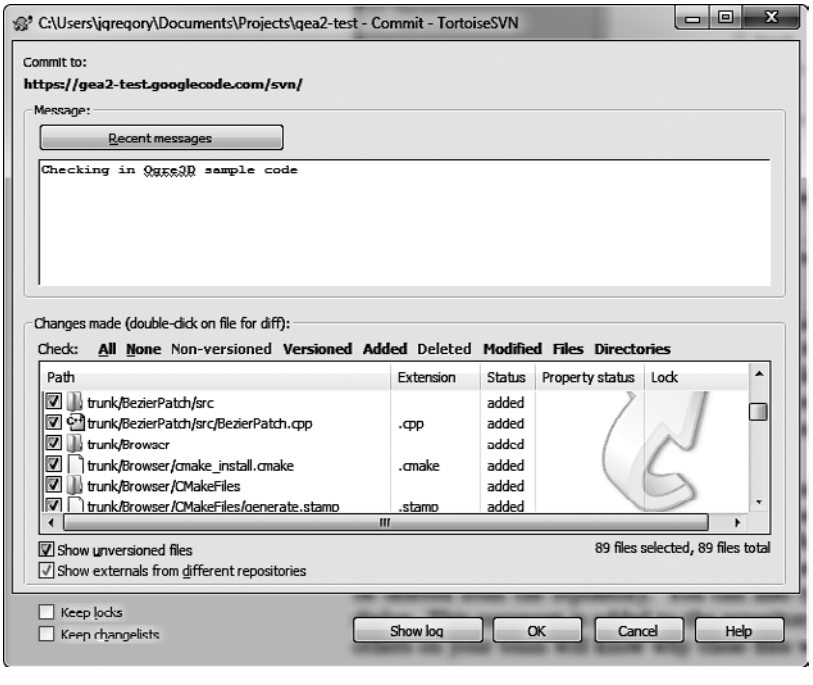


Рис. 2.5. TortoiseSVN, окно коммита

Во время операции коммита Subversion создает *diff* между вашей локальной версией каждого файла и его последней версией в хранилище. Термин *diff* означает разницу (difference), он обычно создается построчным сравнением двух версий файла. Вы можете дважды щелкнуть на любом файле в окне коммита TortoiseSVN (см. рис. 2.5), чтобы увидеть различия между своей версией и последней версией на сервере (то есть внесенные вами изменения). Измененные файлы (то есть любые файлы, имеющие *diff*) фиксируются. Этим вы заменяете последнюю версию в ре-

позитории своей локальной версией и добавляете новую запись в историю версий файла. Любые неизмененные файлы (то есть те, локальная копия которых идентична последней версии в хранилище) по умолчанию игнорируются при коммите. Пример операции коммита показан на рис. 2.6.

Если вы создали какие-либо новые файлы перед коммитом, они будут перечислены как неверсионные в окне коммита. Можете установить рядом с ними флажки, чтобы добавить их в репозиторий. Любые файлы, которые вы удалили локально, также станут отображаться как отсутствующие: если вы отметите их флажками, они будут удалены из центрального хранилища. Можно также ввести в окне коммита комментарий. Он будет добавлен в журнал истории репозитория, чтобы вы и другие члены команды знали, какие файлы были изменены и почему это сделано.

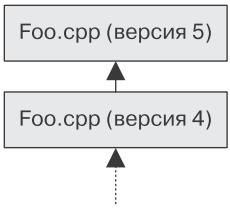


Рис. 2.6. Внесение локальных правок в репозитории

2.1.7. Многократная синхронизация, ветвление и слияние

Некоторые системы контроля версий требуют эксклюзивного редактирования. Это означает, что вы должны сначала указать свои намерения изменить файл, отметив и заблокировав его. Отмеченные файлы доступны для записи только на вашем локальном диске и не могут быть изменены кем-либо еще. Все остальные файлы репозитория на локальном диске доступны только для чтения. Когда редактирование файла закончено, вы должны убрать отметку и тем самым снять блокировку и зафиксировать изменения в репозитории, чтобы они были видны всем. Процесс эксклюзивной блокировки файлов для редактирования гарантирует, что никакие два человека не смогут редактировать один и тот же файл одновременно.

Subversion, CVS, Perforce и многие другие высококачественные системы контроля версий допускают *многократное извлечение*, то есть возможность редактировать файл, пока кто-то другой также редактирует его. Пользователь, который внес изменения первым, сохраняет их как обычно, с изменением номера версии файла в репозитории. Любые последующие коммиты другими пользователями требуют, чтобы программист объединил свои изменения с изменениями, внесенными программистом (-ами) ранее.

Поскольку в одном и том же файле было сделано более одного набора изменений *(diff)*, система контроля версий должна выполнить их *слияние*, чтобы получить окончательную версию файла. Это часто не имеет большого значения, и на самом деле многие конфликты могут быть разрешены автоматически системой контроля версий. Например, если вы изменили функцию `f()`, а другой программист изменил функцию `g()`, то затронутыми оказались разные строки файла. В этом случае слияние между изменениями обычно происходит автоматически без каких-либо конфликтов. Однако если вы оба вносили изменения в одну и ту же функцию `f()`, то второму программисту, чтобы зафиксировать свои изменения, потребуется выполнить *трехстороннее слияние* (рис. 2.7).

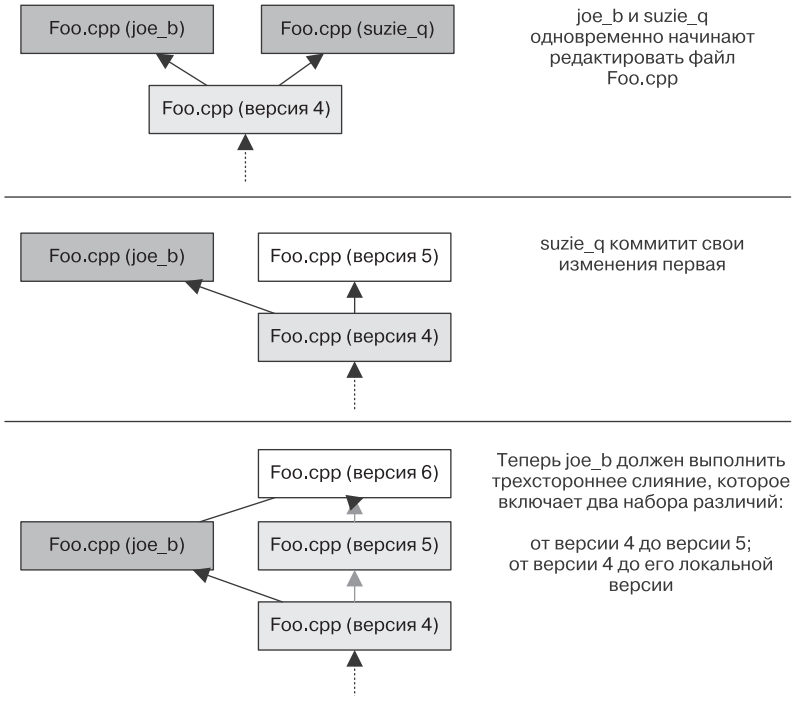


Рис. 2.7. Трехстороннее слияние из-за локальных правок двух пользователей

Для проведения трехстороннего слияния сервер контроля версий должен быть достаточно умным, чтобы отслеживать, какая версия каждого файла есть на вашем локальном диске. Таким образом, когда вы объединяете файлы, система будет знать, какая версия является базовой (общим предком, таким как версия 4 на рис. 2.7).

Subversion допускает многократное извлечение и фактически не требует, чтобы вы извлекали файлы явно. Вы просто начинаете редактировать файлы локально — все они всегда доступны для записи на вашем локальном диске. (Кстати, это одна из причин, по которой, по моему мнению, Subversion плохо масштабируется для больших проектов. Чтобы определить, какие файлы вы изменили, Subversion должна выполнить поиск по всему дереву исходных файлов, что может замедлить работу, если их много. Системы контроля версий типа Perforce, которые явно отслеживают, какие файлы вы изменили, обычно лучше подходят для работы с большими объемами кода. Но в небольших проектах подход Subversion работает просто отлично.)

Когда вы выполняете операцию коммита, щелкая правой кнопкой мыши на любой папке и выбирая пункт SVN Commit (Фиксировать в SVN) в контекстном меню, вам может быть предложено объединить ваши изменения с изменениями, внесенными кем-то еще. Но если никто не изменил файл с момента последнего обновления вашей локальной копии, то ваши изменения будут зафиксированы

и от вас не потребуется никаких действий. Это очень удобная функция, но она может оказаться опасной. Рекомендуется тщательно проверять свои коммиты, чтобы убедиться, что вы не фиксируете файлы, которые не намеревались изменять. Когда TortoiseSVN отображает диалоговое окно Commit Files (Файлы, вошедшие в коммит), дважды щелкните на каждом файле, чтобы увидеть изменения, сделанные вами до нажатия кнопки OK.

2.1.8. Удаление файлов

Когда файл удаляется из хранилища, он на самом деле не исчезает. Файл все еще существует в репозитории, просто его последняя версия помечается как удаленная, чтобы пользователи больше не видели файл в локальных деревьях каталогов. Вы по-прежнему можете получать доступ к предыдущим версиям удаленного файла, щелкнув правой кнопкой мыши на папке, в которой он содержался, и выбрав пункт Show log (Журнал) в меню TortoiseSVN.

Вы можете восстановить удаленный файл, обновив локальный каталог до версии, непосредственно предшествующей версии, в которой он был помечен как удаленный. А затем просто закоммитить файл снова. Это действие заменяет последнюю удаленную версию файла версией, которая была сделана непосредственно перед удалением, и фактически восстанавливает его.

2.2. Компиляторы, компоновщики и IDE

Компилируемые языки, такие как C++, требуют *компилятора* и *компоновщика* для преобразования исходного кода в исполняемую программу. Существует много компиляторов/компоновщиков, доступных для C++, но для платформы Microsoft Windows наиболее часто используемым пакетом, вероятно, является Microsoft Visual Studio. Полнофункциональные версии Professional и Enterprise этого продукта можно приобрести через Интернет в магазине Microsoft, а упрощенную версию Visual Studio, Community Edition (ранее известную как Visual Studio Express), — бесплатно скачать по адресу www.visualstudio.com/downloads. Документация по Visual Studio и стандартным библиотекам C и C++ доступна онлайн на сайте Microsoft Developer Network (MSDN) (msdn.microsoft.com/en-us).

Visual Studio — это больше, чем просто компилятор и компоновщик. Это *интегрированная среда разработки* (integrated development environment, IDE), включающая в себя удобный и полнофункциональный *текстовый редактор* для исходного кода и мощный *отладчик* исходного и машинного уровня. В этой книге основное внимание уделяется платформе Windows, поэтому мы рассмотрим Visual Studio более подробно. Многое из того, что вы узнаете в дальнейшем, будет применимо к другим компиляторам, компоновщикам и отладчикам, таким как LLVM/Clang, gcc/gdb и компилятор Intel C/C++. Поэтому, даже если вы не планируете когда-либо использовать Visual Studio, советую внимательно прочитать этот раздел. Здесь вы найдете всевозможные полезные советы по применению компиляторов, компоновщиков и отладчиков в целом.

2.2.1. Исходные файлы, заголовки и единицы компиляции

Программа, написанная на C++, состоит из *исходных файлов* (source). Обычно они имеют расширение `.c`, `.cc`, `.cxx` или `.cpp` и содержат основную часть исходного кода вашей программы. Исходные файлы известны также как *единицы компиляции*, потому что компилятор переводит один исходный файл за раз из C++ в машинный код.

Специальный вид исходного файла, известный как *заголовочный файл* (header), часто используется для того, чтобы единицы компиляции обменивались между собой информацией, такой как объявление типов и прототипов функций. Файлы заголовков не видны компилятору. Вместо этого *препроцессор* C++ заменяет каждое утверждение `#include` содержимым соответствующего заголовочного файла перед отправкой единицы компиляции в компилятор. Это тонкое, но очень важное различие. Заголовочные файлы существуют как отдельные файлы с точки зрения программиста, но благодаря расширению для обработки заголовочных файлов препроцессора все, что видит компилятор, — это единицы компиляции.

2.2.2. Библиотеки, исполняемые файлы и динамически компонуемые библиотеки

Когда файл компилируется, полученный машинный код помещается в объектный файл (с расширением `.obj` в Windows или `.o` в операционных системах на основе UNIX). Машинный код в объектном файле:

- *перемещаемый* — это означает, что адреса памяти, в которых находится код, еще не определены;
- *несвязанный* — это означает, что любые внешние ссылки на функции и глобальные данные, которые определены вне единицы компиляции, еще не были связаны.

Объектные файлы могут быть собраны в группы, называемые *библиотеками*. Библиотека — это просто архив, очень похожий на файл ZIP или TAR, содержащий ноль или более объектных файлов. Библиотеки существуют просто для удобства, позволяя собирать большое количество объектных файлов в один простой в использовании файл.

Объектные файлы и библиотеки *связываются (линкуются)* в исполняемый файл компоновщиком. Исполняемый файл содержит полностью разрешенный машинный код, который может быть загружен и запущен операционной системой. Работа компоновщика заключается в следующем:

- рассчитать окончательные относительные адреса всего машинного кода, который будет отображаться в памяти при запуске программы;
- убедиться, что все внешние ссылки на функции и глобальные данные, определенные в каждой единице компиляции (объектном файле), правильно разрешены.

Важно помнить, что машинный код в исполняемом файле все еще можно перемещать, а это означает, что адреса всех инструкций и данных в файле по-прежнему относятся к относительному, а не к абсолютному базовому адресу. Окончательный абсолютный базовый адрес программы неизвестен до тех пор, пока она не будет загружена в память непосредственно перед запуском.

Динамически подключаемая библиотека (dynamic link library, DLL) — это особая библиотека, которая представляет собой гибрид между обычной статической библиотекой и исполняемым файлом. DLL действует как библиотека, поскольку содержит функции, которые могут быть вызваны любым количеством различных исполняемых файлов. В то же время DLL действует и как исполняемый файл, потому что она может загружаться операционной системой независимо и содержит код запуска и завершения работы, работающий так же, как функция `main()` в исполняемом файле C++.

Исполняемые файлы, использующие DLL, содержат *частично связанный* машинный код. Большинство ссылок на функции и данные полностью разрешены в окончательном исполняемом файле, но любые ссылки на внешние функции или данные, которые существуют в DLL, остаются несвязанными. Когда исполняемый файл запускается, операционная система разрешает адреса всех несвязанных функций, находя соответствующие библиотеки DLL, загружая их в память, если они еще не загружены, и исправляя необходимые адреса в памяти. Динамически подключаемые библиотеки — очень полезная функция операционной системы, поскольку отдельные библиотеки DLL могут быть обновлены без изменения исполняемых файлов, которые их используют.

2.2.3. Проекты и решения

Теперь, когда мы понимаем, в чем разница между библиотеками, исполняемыми файлами и динамически подключаемыми библиотеками, давайте посмотрим, как их создавать. В Visual Studio *проект* представляет собой набор *исходных файлов*, которые при компиляции создают библиотеку, исполняемый файл или DLL. Во всех версиях Visual Studio, начиная с VS 2013, проекты хранятся в файлах проектов с расширением `.vcxproj`. Эти файлы содержат иерархию виртуальных папок и пути ко всем элементам в проекте. Они также содержат параметры сборки. Файлы представлены в формате XML, поэтому человеку довольно легко читать их и даже редактировать вручную, если это необходимо.

Все версии Visual Studio, начиная с 7-й (Visual Studio 2003), используют *файлы решений* (файлы с расширением `.sln`) в качестве средства хранения и управления набором проектов. Они содержат информацию о среде и состоянии проекта в текстовом виде в собственном формате и хранят параметры, специфические для проекта. Решение — это набор зависимых и/или независимых проектов, предназначенных для создания одной или нескольких библиотек, исполняемых файлов и/или библиотек DLL.

Просматривать проект, решение и связанные с ними элементы, а также управлять ими можно с помощью *обозревателя решений*. В графическом пользовательском

интерфейсе Visual Studio *обозреватель решений* (Solution Explorer) обычно представлен в виде дерева, расположенного вдоль правой или левой стороны главного окна (рис. 2.8).

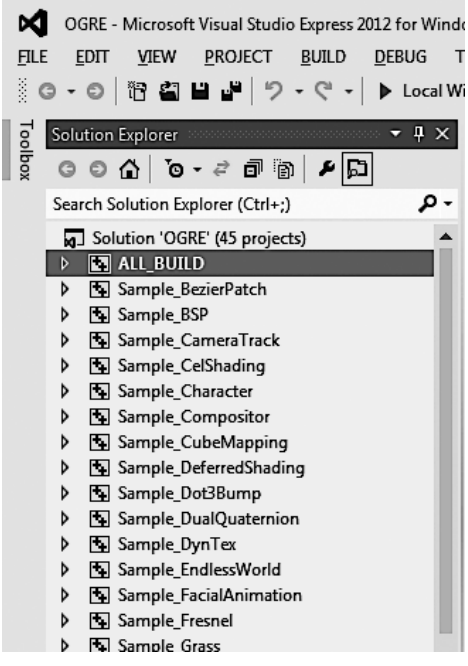


Рис. 2.8. Окно обозревателя решений Visual Studio

Обозреватель решений имеет древовидную структуру. Само решение лежит в корне дерева, где проекты являются его непосредственными потомками. Исходные файлы и заголовки отображаются как дочерние элементы проекта. Проект может содержать любое количество пользовательских папок, вложенных на любую глубину. Папки предназначены только для организационных целей и не имеют ничего общего со структурой папок, в соответствии с которой файлы упорядочены на диске. Тем не менее при настройке папок проекта обычно имитируют структуру папок на диске.

2.2.4. Конфигурации сборки

Препроцессор, компилятор и компоновщик C/C++ предлагают широкий выбор опций для управления тем, как будет собран ваш код. Эти параметры обычно указываются в командной строке при запуске компилятора. Например, типичная команда для создания одного модуля компиляции с помощью компилятора Microsoft может выглядеть так:

```
> cl /c foo.cpp /Fo foo.obj /Wall /Od /Zi
```

В данном примере компилятору/компоновщику предписывается скомпилировать, но не связывать (`/c`) файл `foo.cpp`, вывести результат в объектный файл `foo.obj` (`/Fo foo.obj`), отобразить все предупреждения (`/Wall`), отключить всю оптимизацию (`/Od`) и сгенерировать отладочную информацию (`/Zi`). Эквивалентная командная строка для LLVM/Clang будет выглядеть примерно так:

```
> clang --std=c++14 foo.cpp -o foo.o --Wall -O0 -g
```

Современные компиляторы предоставляют так много опций, что указывать их все явно при каждой сборке было бы непрактично и чревато ошибками. Поэтому применяются готовые наборы параметров проекта, определяющие, как ваш компилятор будет создавать проект. Такой набор параметров называется *конфигурацией сборки* (build configuration). Это просто набор параметров препроцессора, компилятора и компоновщика, связанных с конкретным проектом в вашем решении. Вы можете определить любое количество конфигураций сборки, назвать их как угодно и настроить параметры препроцессора, компилятора и компоновщика по-разному в каждой конфигурации. По умолчанию одни и те же параметры применяются ко всем единицам компиляции в проекте, хотя можно переопределить его глобальные параметры для отдельных единиц компиляции. (Рекомендую по возможности избегать этого, потому что становится трудно определить, какие файлы `.cpp` имеют пользовательские настройки, а какие — нет.)

Когда вы создаете новый проект/решение в Visual Studio, по умолчанию он формирует две конфигурации сборки: конфигурацию выпуска (Release) и конфигурацию отладки (Debug). Релизная сборка предназначена для окончательной версии программного обеспечения, которую вы будете поставлять (продавать) клиентам, а отладочная — для разработки. Отладочная сборка выполняется медленнее, чем неотладочная, но дает программисту бесценную информацию для разработки и отладки программы.

Профессиональные разработчики программного обеспечения часто устанавливают более двух конфигураций сборки для своего программного обеспечения. Чтобы понять, почему так делается, нужно разобраться, как работает локальная (во время компиляции) и глобальная (во время компоновки) оптимизация (мы обсудим их позже). А пока отбросим немного неясный термин «релизная сборка» и остановимся на отладочной сборке (означает, что локальная и глобальная оптимизация отключена) и сборке без отладки (означает, что локальная и/или глобальная оптимизация включена).

Общие параметры сборки

Здесь перечислены некоторые из наиболее распространенных параметров, которыми вы можете управлять с помощью конфигураций сборки для проекта игрового движка.

Настройки препроцессора. При подготовке программного кода к компиляции препроцессор C++ добавляет содержимое произвольных файлов (`#include`), а также определяет и заменяет макросы (`#define`). Чрезвычайно мощная особенность

всех современных препроцессоров C++ — возможность определять макросы препроцессора с помощью параметров командной строки (и, следовательно, с помощью конфигураций сборки). Макросы, определенные таким образом, действуют так, как если бы они были записаны в исходный код с применением инструкции `#define`. В большинстве компиляторов для этого используется параметр командной строки `-D` или `/D`, причем может быть использовано любое количество этих директив.

Эта особенность позволяет передавать различные параметры сборки для вашего кода, не изменяя сам исходный код. В качестве распространенного примера: символ `_DEBUG` всегда определяет отладочную сборку, в то время как сборки без отладочной информации характеризуются символом `NDEBUG`. Исходный код проверяет эти флаги и так узнает, собирается он в режиме отладки или без нее. Это действие известно как условная компиляция (`#ifdef`). Например:

```
void f()
{
#ifdef _DEBUG
    printf("Calling function f()\n");
#endif
    // ...
}
```

Компилятор может свободно вводить в ваш код волшебные макросы препроцессора, основываясь на знании среды компиляции и целевой платформы. Например, макрос `_cplusplus` определяется большинством компиляторов C/C++ при компиляции файла C++. Это позволяет писать код, который автоматически адаптируется к компиляции для C или C++.

Еще один пример: каждый компилятор идентифицирует себя в исходном коде через волшебный макрос препроцессора. При компиляции кода в компиляторе Microsoft определяется макрос `_MSC_VER`, при компиляции под компилятором GNU (gcc) вместо этого определяется макрос `__GNUC__`. И так далее для других компиляторов. Целевая платформа, на которой будет выполняться код, также идентифицируется с помощью макросов. Например, посредством `#ifdef` задается условие для компиляции на различных версиях ОС и архитектурах: `#ifdef _WIN32`, `#ifdef _WIN64`, `#ifdef __linux__`. Эти ключевые функции обеспечивают написание кроссплатформенного кода, потому что позволяют вашему коду знать, какой компилятор его компилирует и на какой целевой платформе он будет запускаться.

Настройки компилятора. Один из наиболее распространенных параметров компилятора определяет, должен ли компилятор включать *отладочную информацию* в создаваемые им объектные файлы. Эта информация используется отладчиками для просмотра вашего кода, отображения значений переменных и т. д. Отладочная информация увеличивает размер исполняемых файлов на диске и позволяет хакерам выполнить обратный инжиниринг вашего кода, поэтому ее всегда удаляют из окончательной версии исполняемого файла. Однако во время разработки отладочная информация неоценима и всегда должна включаться в сборки.

Компилятору также можно указать, следует ли расширять *встроенные функции*. Когда расширение встроенных функций отключено, каждая такая функция

появляется в памяти только один раз по определенному адресу. Это значительно упрощает задачу трассировки кода в отладчике, но, очевидно, за счет уменьшения скорости выполнения, обычно достигаемой в результате использования встроенных функций.

Расширение встроенных функций — всего лишь один пример преобразований кода, известных как *оптимизация*. Агрессивность, с которой компилятор пытается оптимизировать ваш код, и виды оптимизации, которые он использует, можно контролировать с помощью параметров компилятора. Оптимизатор имеет тенденцию переупорядочивать инструкции в коде, и это приводит к тому, что переменные полностью удаляются из кода или перемещаются, а регистры ЦП могут быть повторно применены для новых целей в той же функции. Оптимизированный код обычно сбивает с толку большинство отладчиков, заставляя их по-разному «лгать» вам и затрудняя или делая невозможным понимание того, что происходит на самом деле. Так что все оптимизации обычно отключают в отладочной сборке. Это позволяет тщательно проверить каждую переменную и каждую строку кода в том виде, в котором она была создана программистом.

Но, конечно, такой код будет работать намного медленнее, чем его полностью оптимизированный аналог.

Настройки компоновщика. Компоновщик также предоставляет ряд опций. Вы можете контролировать тип выходного файла — исполняемый файл или DLL. А еще указать внешние библиотеки, которые должны быть связаны с исполняемым файлом, и путь до папок, в которых стоит их искать. Обычная практика — связь с отладочными библиотеками при сборке исполняемого файла отладки и с оптимизированными библиотеками при сборке в режиме без отладки.

Параметры компоновщика также управляют такими вещами, как размер стека, предпочтительный базовый адрес вашей программы в памяти, тип компьютера, на котором будет выполняться код (для оптимизации под конкретное устройство), включена ли глобальная (во время компоновки) оптимизация, и некоторыми другими мелочами, которыми мы здесь не будем заниматься.

Локальная и глобальная оптимизация

Оптимизирующий компилятор — это тот, который может автоматически оптимизировать генерируемый машинный код. Все современные компиляторы C/C++ оптимизирующие. К ним относятся Microsoft Visual Studio, gcc, LLVM/Clang и Intel C/C++.

Две основные формы оптимизации:

- локальная;
- глобальная.

Впрочем, существуют и другие виды, например peep-hole optimization, которая позволяет оптимизатору выполнять оптимизацию для платформы или процессора. Локальная оптимизация работает только на небольших кусках кода, известных как *базовые блоки*. Если грубо, то базовый блок — это последовательность инструкций

на языке ассемблера, в которой нет ветвления. Локальная оптимизация включает в себя:

- алгебраическое упрощение;
- *снижение стоимости операций* (например, преобразование `x/2` в `x >> 1`, потому что оператор побитового сдвига «дешевле» и, следовательно, быстрее, чем оператор целочисленного деления);
- вставку кода;
- *свертывание констант* (распознавание выражений, остающихся постоянными во время компиляции, и замена таких выражений их известными значениями);
- *подстановку значений констант* (замену всех экземпляров переменной, значение которой оказывается постоянным, литеральной константой);
- *развертывание цикла* (например, замену цикла, который всегда повторяется ровно четыре раза, четырьмя копиями кода этого цикла, чтобы исключить условное ветвление);
- *удаление мертвого кода* (удаление кода, который не имеет никакого эффекта, например, удаление выражения присваивания `x = 5`, если сразу за ним следует другое присваивание `x`, например `x = y + 1;`);
- *переупорядочение команд* (для минимизации остановок конвейера ЦП).

Глобальная оптимизация работает за пределами базовых блоков кода — она учитывает весь поток управления программой. Пример такого рода оптимизации — *удаление общего подвыражения* (common subexpression elimination, CSE). Глобальная оптимизация в идеале работает без учета границ единиц компиляции и, следовательно, выполняеется компоновщиком, а не компилятором. Точнее, оптимизация, выполненная компоновщиком, известна как *оптимизация во время компоновки* (link-time optimization, LTO).

Некоторые современные компиляторы, такие как LLVM/Clang, поддерживают *профильную оптимизацию* (profile-guided optimization, PGO). Как следует из названия, они используют информацию профилирования, полученную в ходе предыдущих запусков вашего программного обеспечения, для итеративной идентификации и оптимизации наиболее критических для производительности частей кода. Оптимизация PGO и LTO может привести к впечатляющему повышению производительности, но обходится дорого. Оптимизация LTO значительно увеличивает время, необходимое для компоновки исполняемого файла. А для оптимизации PGO, итеративной по своей природе, требуется запуск программного обеспечения (с помощью команды QA или набора автоматических тестов), чтобы генерировать информацию профилирования, которая способствует дальнейшей оптимизации.

Большинство компиляторов предлагают различные опции, контролирующие, насколько агрессивными будут их усилия по оптимизации. Можно отключить оптимизацию (полезно для отладочных сборок, где отладка важнее производительности) или понизить ее уровень до некоторого заранее определенного максимума. Например, все оптимизации для gcc, Clang и Visual C++ варьируются от –O0 (что

означает, что оптимизации не выполняются) до –O3 (все оптимизации включены). Отдельные оптимизации можно включать и выключать с помощью других параметров командной строки.

Типичные параметры сборки

Проекты игр часто имеют более двух конфигураций сборки. Вот несколько распространенных конфигураций, которые я видел в разработке игр.

- *Debug* (отладочная). Отладочная сборка — это очень медленная версия вашей программы с отключением всех оптимизаций и встроенных функций, содержащая полную отладочную информацию. Эта сборка используется при тестировании нового кода, а также для устранения всех, кроме самых тривиальных, проблем, возникающих в процессе разработки.
- *Development* (для разработки). Сборка разработки (dev build) — это более быстрая версия вашей программы, в которую включены большая часть или все локальные оптимизации, но все еще с отладочной информацией и утверждениями (assertions). (Подробнее об утверждениях см. в главе 3.) Это позволяет увидеть, как игра работает на скорости, соответствующей параметру конечного продукта, но все же дает возможность отследить ошибки.
- *Ship* (релиз). Конфигурация релиза предназначена для построения готовой игры, которую вы поставляете своим клиентам. Иногда его называют окончательной или дисковой сборкой. В отличие от сборки разработки из релизной сборки вся отладочная информация извлечена, большинство или все утверждения не включаются компилятором, везде запускается оптимизация, включая глобальную (LTO и PGO). Релизная сборка очень сложна в отладке, но это самая быстрая и экономная из всех типов сборок.

Гибридные сборки. Гибридная сборка — это конфигурация сборки, в которой большинство модулей компиляции построены в режиме разработки, но небольшая их часть собрана в режиме отладки. Это позволяет заниматься отладкой сегмента кода, на котором в настоящее время сфокусировано внимание, в то время как остальная часть кода продолжает работать почти на полной скорости.

С помощью текстовой системы сборки, такой как `make`, довольно легко настроить гибридную сборку, которая позволяет пользователям задавать режим отладки для каждого отдельного файла. В двух словах: мы определяем переменную `make`, которая называется как-то наподобие `$HYBRID_SOURCES`, в ней перечислены имена всех модулей компиляции (файлы `.cpp`), которые должны быть скомпилированы в режиме отладки для гибридной сборки. Устанавливаем правила сборки для компиляции как отладочной, так и разрабатываемой версии каждого модуля компиляции и организуем размещение полученных объектных файлов (`.obj`/`.o`) в двух разных папках — одной для отладки и одной для разработки. Правило окончательной компоновки настроено для связи с отладочными версиями объектных файлов, перечисленных в `$HYBRID_SOURCES`, и с неотладочными версиями всех других объектных файлов. Если мы верно настроили его, правила зависимостей `make` позаботятся обо всем остальном.

К сожалению, это не так просто сделать в Visual Studio, потому что его конфигурации сборки предназначены для применения на уровне проекта, а не на уровне компиляции. Суть проблемы в том, что мы не можем легко определить список модулей компиляции, которые хотим собрать в режиме отладки. Одно из решений этой проблемы — написание сценария (на Python или другом подходящем языке), который автоматически генерирует файлы Visual Studio `.vcxproj` с учетом списка файлов `.cpp`, которые вы хотите встроить в режиме отладки в гибридной конфигурации. Еще одна альтернатива, которая работает, если ваш исходный код уже организован в библиотеки, — настроить конфигурацию гибридной сборки на уровне решения, которая выбирает между отладочными и разрабатываемыми сборками для каждого проекта (и, следовательно, для библиотек). Это не так гибко, как управление на основе единиц компиляции, но работает довольно хорошо, если библиотеки достаточно детализированы.

Конфигурации сборки и тестируемость. Чем больше конфигураций сборки поддерживает ваш проект, тем сложнее становится тестирование. Хотя различия между конфигурациями могут быть незначительными, существует определенная вероятность того, что в одной из них возникла критическая ошибка, а в других — нет. Поэтому все конфигурации сборки должны быть проверены одинаково тщательно. Большинство игровых студий формально не тестируют свои отладочные сборки, потому что конфигурация отладки предназначена в основном для внутреннего использования во время первоначальной разработки функции и для отладки ошибок, обнаруженных в одной из прочих конфигураций. Однако если ваши тестеры тратят большую часть своего времени на тестирование конфигурации сборки для разработки, вы не можете просто сделать релизную сборку для своей игры накануне Gold Master-тестирования и ожидать, что она будет иметь профиль ошибок, идентичный профилю сборки для разработки. То есть команда тестировщиков должна одинаково тестировать и сборку для разработки, и релизные сборки на протяжении альфа- и бета-тестирования, чтобы убедиться, что в финальной сборке не будет никаких неприятных сюрпризов. С точки зрения тестируемости есть очевидное преимущество в том, чтобы свести число конфигураций сборки к минимуму, и фактически именно по этой причине у некоторых студий нет отдельной релизной сборки — они просто поставляют сборку разработки после того, как тщательно протестируют ее (но удалив информацию об отладке).

Руководство по настройке проекта

Если щелкнуть правой кнопкой мыши на любом проекте в обозревателе решений и выбрать в контекстном меню пункт Properties (Свойства), откроется диалоговое окно Property Pages (Страница свойств проекта). Дерево слева показывает различные категории настроек. Чаще всего мы будем использовать четыре:

- Configuration Properties/General (Свойства конфигурации/Общие);
- Configuration Properties/Debugging (Свойства конфигурации/Отладка);
- Configuration Properties/C++ (Свойства конфигурации/C++);
- Configuration Properties/Linker (Свойства конфигурации/Компоновщик).

Конфигурация в выпадающем комбинированном меню. Обратите внимание на раскрывающееся меню Configuration (Конфигурация) в верхнем левом углу окна. Все свойства, отображаемые на этих страницах свойств, применяются отдельно для каждой конфигурации сборки. Если вы задаете свойство для конфигурации отладки, это не обязательно означает, что такой же параметр существует в других конфигурациях.

Если вы щелкнете на поле с раскрывающимся списком, чтобы увидеть его весь, то обнаружите, что можете выбрать одну или несколько конфигураций и даже All configurations (Все конфигурации). Старайтесь выполнить бóльшую часть редактирования конфигурации сборки, выбрав All configurations (Все конфигурации). Так вам не придется повторять одно и то же несколько раз для каждой конфигурации, и вы не настроите случайно параметры одной из конфигураций неправильно. Однако имейте в виду, что некоторые параметры должны различаться для разных конфигураций отладки и разработки. Например, настройки встраивания функции и оптимизации кода должны быть различными в отладочных сборках и сборках для разработки.

Страница общих свойств. На странице свойств General (Общие) (рис. 2.9) наиболее полезны следующие поля.

- Output Directory (Целевой каталог). Определяет место, куда будут помещены конечные продукты сборки — исполняемый файл, библиотеки или DLL, которые в конечном итоге выдаст компилятор/компоновщик.

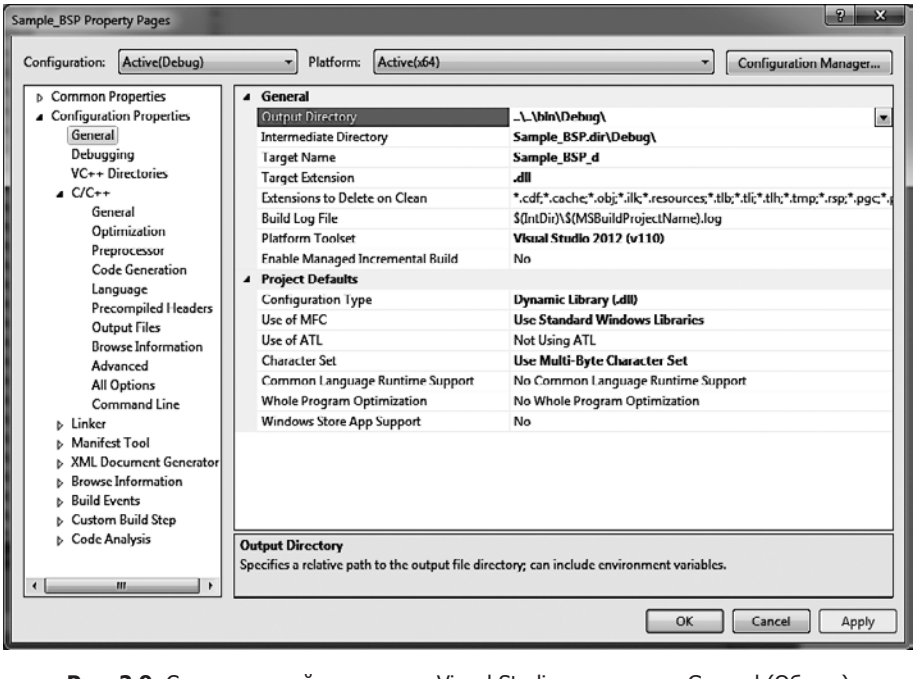


Рис. 2.9. Страницы свойств проекта Visual Studio — страница General (Общие)

- Intermediate Directory (Промежуточный каталог). Определяет, куда помещаются промежуточные файлы, в первую очередь объектные (с расширением `.obj`), во время сборки. Промежуточные файлы никогда не поставляются с окончательной программой — они нужны только в процессе сборки исполняемого файла, библиотеки или DLL. Так что стоит размещать промежуточные файлы в другом каталоге, не там, где находятся конечные (файлы `.exe`, `.lib` или `.dll`).

Обратите внимание на то, что Visual Studio предоставляет средство макросов, которое используется для указания каталогов и других параметров в диалоговом окне Project Property Pages (Страница свойств проекта). Макрос, по сути, является именованной переменной, которая содержит глобальное значение и на которую можно ссылаться в настройках конфигурации проекта.

Макрос вызывают написанием его имени, заключенного в скобки и начинающегося со знака доллара, например `$(ConfigurationName)`. Некоторые часто используемые макросы перечислены далее.

- `$(TargetFileName)`. Имя конечного исполняемого файла, библиотеки или DLL-файла, создаваемого этим проектом.
- `$(TargetPath)`. Полный путь к папке, содержащей конечный исполняемый файл, библиотеку или DLL.
- `$(ConfigurationName)`. Имя конфигурации сборки, которая по умолчанию в Visual Studio будет называться Debug или Release (как уже говорилось, реальный игровой проект, вероятно, будет иметь несколько конфигураций — Debug, Hybrid, Development и Ship).
- `$(OutDir)`. Значение поля Output Directory (Целевой каталог) указывается в этом диалоговом окне.
- `$(IntDir)`. Значение поля Intermediate Directory (Промежуточный каталог) указывается в этом окне.
- `$(VCInstallDir)`. Каталог, в котором в настоящее время установлена стандартная библиотека Visual Studio C++.

Преимущество использования макросов вместо жесткого задания конфигурации в настройках заключается в том, что изменение значения глобального макроса автоматически повлияет на все параметры конфигурации, в которых он применен. Кроме того, некоторые макросы, например `$(ConfigurationName)`, автоматически изменяют свои значения в зависимости от конфигурации сборки, поэтому могут обеспечить единые настройки для всех конфигураций.

Чтобы просмотреть полный список доступных макросов, щелкните в поле Output Directory (Целевой каталог) или Intermediate Directory (Промежуточный каталог) на странице свойств General (Общие), нажмите маленькую стрелку справа от текстового поля и выберите пункт Edit, а затем в появившемся диалоговом окне нажмите кнопку Macros (Макрос).

Страница свойств отладки. На странице свойств Debugging (Отладка) указываются имя и расположение исполняемого файла для отладки. Здесь также можно указать аргументы командной строки, которые должны быть переданы программе при запуске. Мы обсудим отладку программы более подробно в дальнейшем.

Страница свойств C/C++. Страница свойств C/C++ управляет настройками языка во время компиляции, которые определяют то, как исходные файлы будут скомпилированы в объектные файлы (расширение `.obj`). Параметры на этой странице не влияют на то, как объектные файлы связываются в конечный исполняемый файл или DLL.

Рекомендуется изучить различные вкладки на странице свойств C/C++, чтобы узнать, какие настройки доступны. Вот некоторые из наиболее часто используемых настроек.

- General Property Page/Additional Include Directories (Страница общих свойств/Дополнительные каталоги). В этом поле перечислены каталоги на диске, в которых будет выполняться поиск заголовочных файлов `#included`.

ВАЖНО

Всегда лучше указывать эти каталоги, используя относительные пути и/или макросы Visual Studio, такие как $(OutDir) или $(IntDir). Таким образом, если вы переместите дерево сборки в другое место на диске или на другой компьютер с другой корневой папкой, все будет работать правильно.

- General Property Page/Debug Information Format (Страница общих свойств/Формат отладочной информации). Это поле определяет, генерируется ли отладочная информация и в каком формате. Обычно конфигурации отладки и разработки включают в себя информацию об отладке, чтобы вы могли отслеживать ошибки во время разработки игры. Во время релизной сборки всю эту информацию нужно удалить, чтобы не допустить взлома.
- Preprocessor Property Page/Preprocessor Definitions (Страница свойств препроцессора/Определения препроцессора). Это очень удобное поле содержит любое количество ключей препроцессора C/C++, которые должны быть определены в коде при его компиляции.

Страница свойства компоновщика. На странице свойств Linker (Компоновщик) перечислены свойства, влияющие на то, как файлы объектного кода будут связаны в исполняемый файл или DLL. Вновь предлагаю изучить различные вкладки. Далее приведены некоторые часто используемые настройки.

- General Property Page/Output File (Страница общих свойств/Целевой файл). Этот параметр указывает имя и местоположение конечного продукта сборки — обычно исполняемого файла или DLL.
- General Property Page/Additional Library Directories (Страница общих свойств/Дополнительные каталоги библиотек). Как и в поле настроек C/C++ при компиляции

Additional Include Directories (Дополнительные каталоги), в этом поле перечислены несколько каталогов (или ни одного), в которых будет происходить поиск библиотек и объектных файлов, связанных с конечным исполняемым файлом.

- Input Property Page/Additional Dependencies (Страница свойств ввода/Дополнительные зависимости). В этом поле перечислены внешние библиотеки, которые вы хотите связать в исполняемый файл или DLL. Например, здесь будут перечислены библиотеки OGRE, если вы создаете приложение с поддержкой OGRE.

Обратите внимание на то, что Visual Studio применяет различные «волшебные заклинания» для указания библиотек, которые должны быть связаны в исполняемый файл. Например, можно использовать в исходном коде специальную инструкцию `#pragma`, чтобы указать компоновщику на автоматическое связывание с конкретной библиотекой. По этой причине вы можете не найти все библиотеки, которые действительно компонуете, в поле Additional Dependencies (Дополнительные зависимости). (Фактически именно поэтому они называются *дополнительными* зависимостями.) Возможно, вы заметили, например, что приложения DirectX не перечисляют все библиотеки DirectX в своем поле Additional Dependencies (Дополнительные зависимости). И теперь знаете почему.

2.2.5. Отладка кода

Один из самых важных навыков, который способен освоить любой программист, — эффективная отладка кода. В данном подразделе приведены полезные советы и небольшие хитрости этого процесса. Одни можно применить к любому отладчику, а другие — только к Microsoft Visual Studio. Однако обычно эквивалент функций отладки для Visual Studio имеется и в других отладчиках, поэтому подраздел должен оказаться полезным, даже если вы не используете Visual Studio для отладки кода.

Стартап-проект

Решение Visual Studio может содержать более одного проекта. Одни из них создают исполняемые файлы, другие — библиотеки или DLL-файлы. Возможно иметь в рамках одного решения более одного проекта, который создает исполняемый файл. Visual Studio предоставляет параметр Start-Up Project. Это проект, который считается текущим для целей отладки. Как правило, программист будет отлаживать один проект за раз, устанавливая один стартап-проект. Но можно отлаживать и несколько проектов одновременно (более подробная информация размещена по адресу msdn.microsoft.com/en-us/library/0s590bew(v=vs.100).aspx).

Стартап-проект выделен жирным шрифтом в обозревателе решений. По умолчанию нажатие клавиши F5 приведет к запуску файла `.exe`, сформированного этим проектом, если он создает исполняемый файл. (Технически нажатие F5 запускает любую команду, которую вы введете в поле Command на странице свойств отладки, поэтому она не ограничивается запуском файла `.exe`, созданного вашим проектом.)

Точки останова

Добавление *точек останова* — основной метод при отладке кода. Точка останова приказывает программе останавливаться на определенной строке исходного кода, чтобы вы могли проверить, что происходит.

В Visual Studio выберите строку и нажмите клавишу F9, чтобы установить там точку останова. Когда вы запустите свою программу и строка кода, содержащая точку останова, должна быть выполнена, отладчик остановит программу. Мы говорим, что точка останова была достигнута. Маленькая стрелка покажет вам, на какой именно строке находится программный счетчик ЦП после остановки (рис. 2.10).

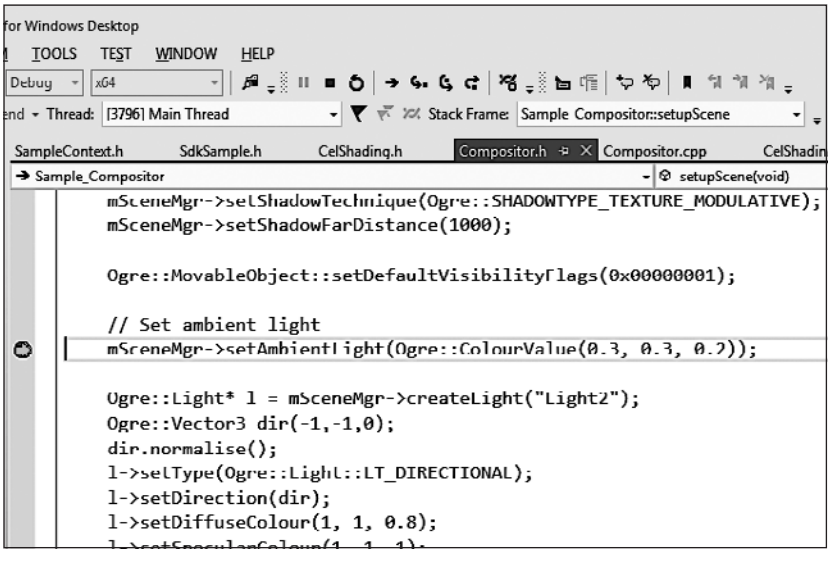


Рис. 2.10. Установка точки останова в Visual Studio

Пошаговое выполнение кода

После достижения точки останова вы можете пошагово выполнить код, нажимая клавишу F10. Желтая стрелка счетчика программ перемещается, показывая строки по мере выполнения. Нажав F11, вы попадете *в* вызов функции (то есть следующая строка кода, которую вы увидите, будет первой строкой вызываемой функции), в то время как нажатие F10 позволит перешагнуть *через* этот вызов функции (то есть отладчик вызовет функцию на полной скорости и остановится на следующей строке после вызова).

Стек вызовов

Окно стека вызовов (рис. 2.11) показывает стек функций, которые были вызваны в любой момент во время выполнения кода. (Более подробно о стеке программы написано в главе 3.) Чтобы отобразить стек вызовов (если он еще не виден),

перейдите в меню Debug (Отладка) в строке главного меню, выберите Windows (Окна), а затем Call Stack (Стек вызовов). Это окно можно быстро открыть, нажав сочетание клавиш Ctrl+Alt+C.

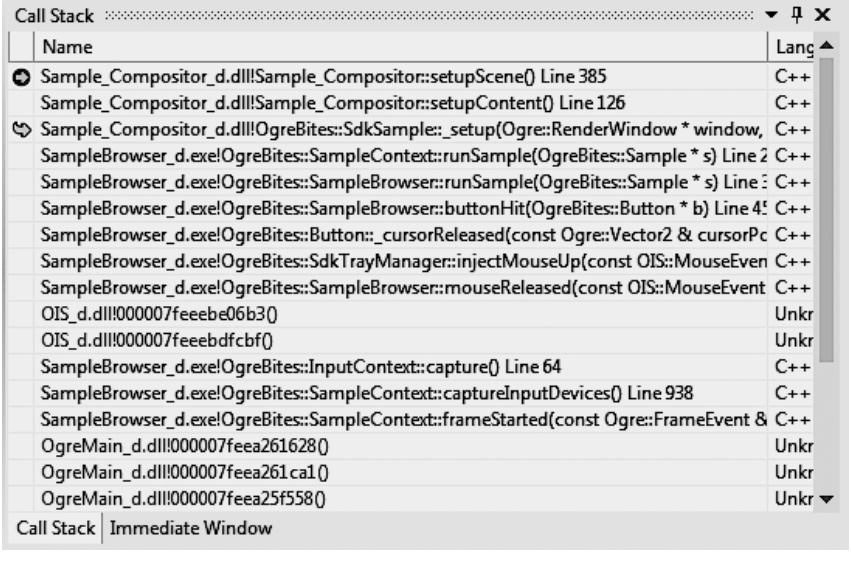


Рис. 2.11. Окно стека вызовов

Как только достигнута точка останова или программа приостановлена вручную, вы можете перемещаться вверх и вниз по стеку вызовов, дважды щелкнув на записи в окне Call Stack (Стек вызовов). Это очень полезно для проверки цепочки вызовов функций, которые были сделаны между вызовом `main()` и текущей строкой кода. Например, можете отследить основную причину ошибки в родительской функции, которая проявилась в глубоко вложенной дочерней функции.

Окно просмотра

По мере того как вы шагаете по своему коду и перемещаетесь вверх и вниз по стеку вызовов, вам, возможно, захочется проверить значения переменных в своей программе. Для этого используется окно просмотра. Чтобы открыть его, перейдите в меню Debug (Отладка) в строке главного меню, выберите Windows (Окна), затем Watch (Контрольные значения) и один из пунктов от Watch 1 до Watch 4 (или нажмите сочетание клавиш Ctrl+Alt+W и номер от 1 до 4). (Visual Studio позволяет открывать до четырех окон просмотра одновременно.) Когда окно просмотра открыто, можете вводить в него имена переменных или перетаскивать выражения прямо из исходного кода.

Как показано на рис. 2.12, переменные с простыми типами данных приведены вместе со значениями, стоящими непосредственно справа от имен. Сложные типы данных отображаются в виде небольших древовидных представлений, которые

можно легко развернуть, чтобы увидеть любую вложенную структуру. Базовый класс класса всегда отображается как первый дочерний элемент экземпляра производного класса. Это позволяет вам проверять члены данных не только самого класса, но и его базового класса (классов).

Рис. 2.12. Окно просмотра Visual Studio

Вы можете ввести в окно просмотра практически любое допустимое *выражение* C/C++, и Visual Studio вычислит его и попытается отобразить полученное значение. Например, введите `5 + 3`, и Visual Studio отобразит `8`. Можно преобразовывать переменные из одного типа в другой, используя синтаксис C или C++. Например, в результате ввода в окно просмотра `(float)intVar1/(float)intVar2` отобразится отношение двух целочисленных переменных в виде значения с плавающей точкой.

Вы даже можете *вызывать функции в своей программе* из окна просмотра. Visual Studio автоматически вычисляет выражения, введенные в окно (окна) просмотра, поэтому, если в нем вызвана функция, она будет вызываться каждый раз, когда при пошаговом проходе программы достигнута точка останова. Это позволяет задействовать функциональные возможности программы, чтобы сэкономить время при попытке интерпретировать данные, которые вы проверяете в отладчике. Допустим, игровой движок предоставляет функцию `quatToAngleDeg()`, которая преобразует кватернион в угол поворота в градусах. Вы можете вызвать ее в окне просмотра, чтобы легко проверить угол поворота любого кватерниона в отладчике.

Также можете использовать различные суффиксы для выражений в окне просмотра, чтобы изменить способ отображения данных в Visual Studio (рис. 2.13).

- Суффикс `,d` заставляет значения отображаться в виде десятичной записи.
- Суффикс `,x` заставляет значения отображаться в шестнадцатеричном формате.
- Суффикс `,n` (где `n` — любое положительное целое число) заставляет Visual Studio обрабатывать значение как массив с `n` элементами. Это позволяет расширять массив данных, на которые ссылается указатель.
- Вы также можете написать простые выражения в квадратных скобках, которые вычисляют значение `n` для суффикса `,n`. Например, наберите что-то вроде:

  ```
  my_array, [my_array_count]
  ```

  чтобы попросить отладчик показать `my_array_count` элементов массива с именем `my_array`.

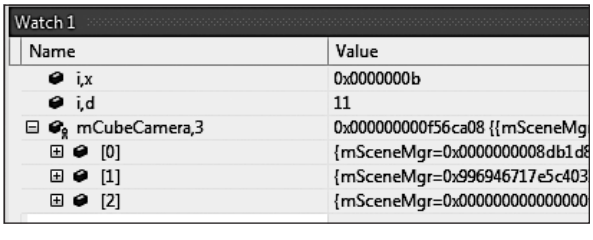


Рис. 2.13. Суффиксы в окне просмотра Visual Studio

Будьте осторожны при развертывании очень больших структур данных в окне просмотра, потому что это способно замедлить работу отладчика до такой степени, что его невозможно будет использовать.

Точки останова данных

Обычные точки останова срабатывают, когда счетчик программ ЦП достигает определенной машинной инструкции или строки кода. Еще одна невероятно полезная особенность современных отладчиков — возможность установить точку останова, которая срабатывает всякий раз, когда *происходит запись* в конкретный адрес памяти (то есть изменяется значение). Такие точки называются *точками останова данных*, потому что они инициируются изменениями данных, а иногда *аппаратными точками останова*, потому что реализуются через специальную функцию аппаратного обеспечения ЦП, а именно возможность вызывать прерывание, когда записывается предопределенный адрес памяти.

Вот как обычно используются точки останова данных. Допустим, вы отслеживаете ошибку, которая проявляется в виде нулевого (`0.0f`) значения, загадочно возникающего внутри переменной — члена определенного объекта с именем `m_angle`, который всегда *должен* содержать ненулевой угол. Вы понятия не имеете, какая функция может записать этот ноль в переменную, но знаете адрес переменной. (Можно набрать в окне просмотра `&object.m_angle`, чтобы найти ее адрес.) Чтобы

отследить виновника, установите точку останова данных по адресу `object.m_angle`, а затем просто запустите программу. Когда значение изменится, отладчик остановится автоматически. Затем проверьте стек вызовов, чтобы поймать функцию-нарушителя с поличным.

Чтобы установить точку останова данных в Visual Studio, выполните следующие действия.

- Откройте окно Breakpoints (Точки останова) из меню Debug ▸ Windows (Отладка ▸ Окна) (рис. 2.14).

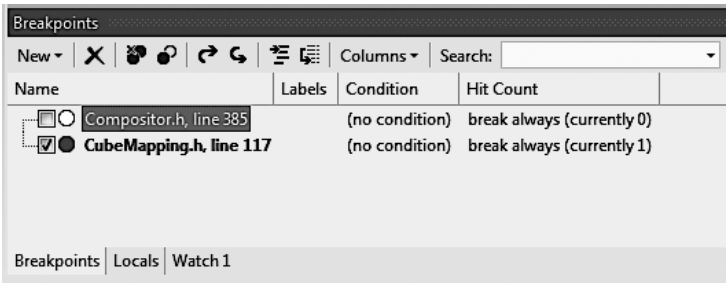


Рис. 2.14. Окно точек останова Visual Studio

- Нажмите кнопку New (Создать) в левом верхнем углу окна.
- Выберите пункт New Data Breakpoint (Точка останова данных).
- Введите адрес или выражение, задающее адрес, например `&myVariable` (рис. 2.15).

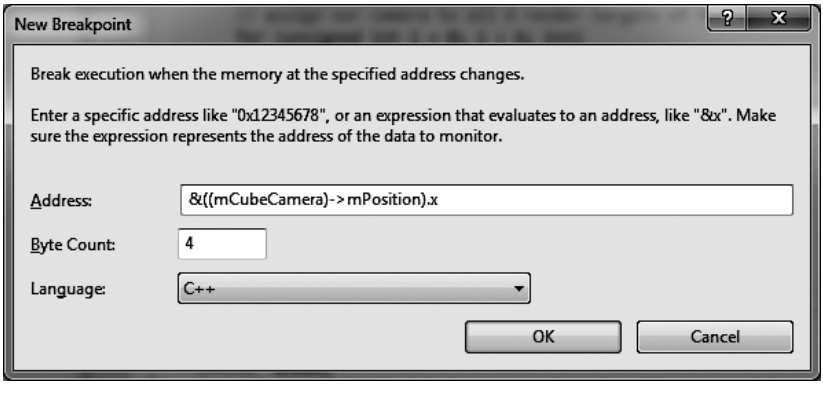


Рис. 2.15. Определение точки останова данных

Условные точки останова

В окне Breakpoints (Точки останова) вы заметите, что можете устанавливать условия и счетчики посещений для точек останова любого типа — точек останова данных или обычных точек останова на строках кода.

Условная точка останова заставляет отладчик оценивать выражение C/C++, которое вы предоставляете, каждый раз при достижении точки останова. Если выражение истинно, отладчик останавливает программу и дает вам возможность увидеть, что происходит. Если выражение ложно, точка останова игнорируется и программа продолжает действовать. Это очень полезно для установки точек останова, которые срабатывают только тогда, когда функция вызывается для определенного экземпляра класса. Например, у вас есть игровой уровень с 20 танками на экране и вы хотите остановить программу, когда работает третий танк, чей адрес памяти вы знаете — `0x12345678`. Установив выражение условия точки останова на что-то вроде `(uintptr_t)this == 0x12345678`, можно ограничить точку останова только экземпляром класса, чей адрес памяти (указатель) `0x12345678`.

Задание *счетчика* срабатывания для точки останова приводит к тому, что отладчик уменьшает счетчик каждый раз при достижении точки останова и фактически останавливает программу только тогда, когда тот достигает нуля. Это действительно полезно в ситуациях, когда точка останова находится внутри цикла и нужно проверить, что происходит во время 376-й итерации цикла (например, в 376-м элементе массива). Вы не можете просто сидеть и нажимать клавишу `F5` 375 раз! Но можете позволить функции счетчика посещений в Visual Studio сделать это за вас.

Здесь следует предостеречь читателя: условные точки останова заставляют отладчик оценивать условное выражение каждый раз, когда достигается точка останова, поэтому они способны снизить производительность отладчика и игры.

Сборки, оптимизированные для отладки

Ранее я упоминал, что отладка ошибок для сборок разработки или релиза может быть очень сложной прежде всего из-за того, как компилятор оптимизирует код. В идеале каждый программист предпочел бы выполнить всю отладку в отладочной сборке. Однако зачастую это нереально. Иногда ошибка возникает настолько редко, что вы будете использовать любую возможность ее устранения, даже если она возникла в неотладочной сборке на чужой машине. Другие ошибки появляются только в ваших неотладочных сборках, но они как по волшебству исчезают, когда вы запускаете отладочную сборку. Эти страшные *ошибки, не связанные только с отладкой*, иногда вызваны неинициализированными переменными, потому что переменные и динамически распределяемые блоки памяти часто устанавливаются в ноль в режиме отладки, но остаются «с мусором» в сборке без отладки. К другим распространенным причинам ошибок только в неотладочных сборках относятся: случайный пропуск кода в сборке без отладки (например, когда важный код ошибочно помещен в оператор проверки `assertion`), наличие структур данных, размер которых различается в сборках для отладки и разработки, ошибки, которые вызываются только путем встраивания или введенных компилятором оптимизаций, и в редких случаях это ошибки в самом оптимизаторе компилятора, приводящие к тому, что он выдает неверный код в полностью оптимизированной сборке.

Ясно, что каждому программисту необходимо уметь отслеживать ошибки не только в отладочной сборке, как это не неприятно. Лучший способ упростить

отладку оптимизированного кода — практиковаться и расширять свои навыки в этой области, когда есть такая возможность. Вот несколько полезных советов.

- *Научитесь читать и проходить команды ассемблера в отладчике.* В неотладочной сборке отладчику часто бывает трудно отследить, какая строка исходного кода выполняется в данный момент. Благодаря переупорядочению инструкций вы часто будете видеть, как счетчик программ скачкообразно перемещается внутри функции при просмотре в режиме исходного кода. Тем не менее все снова нормализуется, когда вы работаете с кодом в режиме дизассемблирования, то есть выполняете инструкции на языке ассемблера по отдельности. Каждый программист на C/C++ должен быть хотя бы немного знаком с архитектурой и языком ассемблера своих целевых процессоров. Таким образом, даже если отладчик будет сбит с толку, вы — не будете. (Ознакомиться с языком ассемблера можно в главе 3.)
- *Используйте регистры для вывода значений или адресов переменных.* Иногда отладчик не может отобразить значение переменной или содержимое объекта во внеотладочной сборке. Но если счетчик программы не слишком далеко ушел от первоначального использования переменной, вполне вероятно, что ее адрес или значение все еще сохраняются в одном из регистров ЦП. Если вы сможете проследить инструкции ассемблера до того места, где переменная была впервые загружена в регистр, то зачастую у вас получится узнать ее значение или адрес, проверив этот регистр. Используйте окно регистров или введите имя регистра в окне просмотра, чтобы просмотреть его содержимое.
- *Проверяйте переменные и содержимое объекта по адресу.* Зная адрес переменной или структуры данных, вы обычно можете увидеть то, что в ней содержится, приведя адрес к соответствующему типу в окне просмотра. Например, если известно, что экземпляр класса `Foo` находится по адресу `0x1378A0C0`, можно ввести `(Foo*)0x1378A0C0` в окне просмотра, и отладчик будет интерпретировать этот адрес памяти, как если бы он был указателем на объект `Foo`.
- *Используйте статические и глобальные переменные.* Даже в оптимизированной сборке отладчик обычно способен проверять глобальные и статические переменные. Если вы не можете вывести адрес переменной или объекта, то попробуйте использовать статическую или глобальную переменную, которая будет содержать адрес, прямо или косвенно. Например, если требуется найти адрес внутреннего объекта в системе физики, то можно обнаружить, что он на самом деле хранится как переменная — член глобального объекта `PhysicsWorld`.
- *Изменяйте код.* Если вы можете воспроизвести ошибку относительно легко только во внеотладочной сборке, попробуйте изменить исходный код, чтобы упростить поиск проблемы. Добавьте операторы печати так, чтобы видеть, что происходит. Введите глобальную переменную, чтобы упростить исследование проблемной переменной или объекта в отладчике. Добавьте код, чтобы обнаружить проблемы состояния или изолировать конкретный экземпляр класса.

2.3. Инструменты профилирования

Игры обычно представляют собой высокопроизводительные программы, выполняющиеся в реальном времени. Таким образом, программисты игровых движков всегда ищут пути ускорения кода. В этом разделе мы исследуем некоторые инструменты, которые можем задействовать для измерения производительности программного обеспечения. (Об использовании профилирования данных для того, чтобы оптимизировать программы, подробнее поговорим в главе 4.)

Существует хорошо известное, хотя и довольно ненаучное правило — *закон Парето* (см. https://ru.wikipedia.org/wiki/Закон_Парето). Оно также известно как *правило 80/20*, потому что утверждает, что во многих ситуациях 20 % усилий дают 80 % результата. В компьютерных науках этот принцип применяется как для устранения ошибок (80 % предполагаемых ошибок в части программного обеспечения могут быть устранены исправлением только 20 % кода), так и для оптимизации программного обеспечения, где, как правило, за 80 % (или больше) времени, затраченного на работу любой программы, выполняются только 20 % (или меньше) строк кода. Другими словами, если вы оптимизируете 20 % кода, то потенциально можете реализовать 80 % прироста скорости выполнения, которые вы когда-либо реализуете вообще.

А как узнать, *какие* 20 % кода следует оптимизировать? Для этого нужен *профилировщик*. Профилировщик — это инструмент, который измеряет время выполнения кода. Он способен сказать, сколько времени затрачивается на каждую функцию. Так что вы можете направить усилия по оптимизации только на те функции, которые требуют львиной доли времени выполнения.

Некоторые профилировщики также сообщают, сколько *раз* вызывается каждая функция. Это важное значение, которое стоит учитывать. Функция может занимать время по двум причинам:

- она выполняется долго;
- она вызывается часто.

Например, только функция, которая запускает алгоритм A* для вычисления оптимальных путей в игровом мире, может вызываться несколько раз в каждом кадре.

Еще больше информации можно получить, если использовать правильный профилировщик. Некоторые из них строят граф вызовов, что означает: вы можете видеть, какие функции (они называются *родительскими*) вызвали любую данную функцию и какие функции вызвала она (известны как *дочерние функции*, или *потомки*). Вы даже можете увидеть процент времени работы функции, который был потрачен на вызов каждого из ее потомков, и процент от общего времени выполнения, который рассчитывается для каждой отдельной функции.

Профилировщики делятся на две большие категории.

- *Статистические профилировщики*. Спроектированы так, чтобы быть ненавязчивыми, что означает: целевой код работает почти с одинаковой скоростью

независимо от того, включено профилирование или нет. Эти профилировщики периодически делают выборки из регистра счетчика программы ЦП и определяют, какая функция выполняется в данный момент. Количество выборок, взятых в каждой функции, дает приблизительный процент от общего времени работы, которое съедает эта функция. Intel *VTune* является золотым стандартом в статистических профилировщиках для машин Windows, использующих процессоры Intel Pentium, а сейчас он доступен и для Linux. Более подробно: software.intel.com/en-us/intel-vtune-amplifier-xe.

- *Инструментальные профилировщики*. Нацелены на предоставление максимально точных и полных временных данных, но целевая программа обычно замедляется из-за сканирования, поскольку выполняется в режиме реального времени при включенном профилировании. Эти профилировщики работают, предварительно обрабатывая ваш исполняемый файл и вставляя специальные прологи и эпилоги в каждую функцию. Код пролога и эпилога вызывает библиотеку профилирования, которая, в свою очередь, проверяет стек вызовов программы и записывает все детали, включая информацию о том, какая родительская функция вызывала соответствующую функцию и сколько раз был вызван дочерний элемент. Данный вид профилировщика может быть настроен даже для мониторинга каждой строки кода в исходной программе, что позволяет ему сообщать, сколько времени выполняется каждая строка. Результаты поразительно точны и полны, но включение профилирования способно сделать игру практически неработающей. Профилировщик Rational Quantify компании IBM, доступный в составе набора инструментов Rational Purify Plus, — отличный инструментальный профилировщик. Чтобы подробнее ознакомиться с профилированием с помощью Quantify, обратитесь к www.ibm.com/developerworks/rational/library/957.html.

Компания Microsoft также выпустила профилировщик, который представляет собой гибрид двух подходов. Он называется LOP, что означает «профилировщик с низкими издержками» (low-overhead profiler). LOP использует статистический подход, периодически выбирая состояние процессора, так что он слабо влияет на скорость выполнения программы. Однако для каждой выборки он анализирует стек вызовов, определяя тем самым цепочку родительских функций, которая привела к каждой выборке. Это позволяет LOP предоставлять информацию, обычно недоступную статистическому профилировщику, такую как распределение вызовов по родительским функциям.

Razor CPU компании SN Systems предпочтителен для измерения скорости игр, работающих на процессоре PS4 для PlayStation 4. Он поддерживает как статический, так и инструментальные методы профилирования. (Для получения более подробной информации см. www.snsystems.com/tech-blog/2014/02/14/function-level-profiling/.) Его аналог Razor GPU предоставляет средства профилирования и отладки для шейдеров и вычислительных заданий на GPU PS4.

2.3.1. Список профилировщиков

Существует множество инструментов для профилирования. На странице en.wikipedia.org/wiki/List_of_performance_analysis_tool представлен довольно полный их список.

2.4. Утечка и нарушение целостности памяти

Две другие проблемы, с которыми сталкиваются программисты на C и C++, — это утечка памяти и нарушение ее целостности. Утечка памяти происходит, когда она выделяется, но не освобождается. Это приводит к потере памяти и в итоге способно вызвать ошибку Fatal: out of memory. Повреждение памяти происходит, когда программа непреднамеренно записывает данные в неправильную область памяти, перезаписывая важные данные, которые там находились, и в то же время не может обновить место памяти, где эти данные *должны* были быть записаны. Вина за обе эти проблемы лежит непосредственно на языковой функции, известной как *указатель*.

Указатель — мощный инструмент. При правильном использовании он способен быть агентом добра, но его легко превратить в агента зла. Если указатель указывает на освобожденную память или ему случайно присвоено ненулевое целочисленное значение или значение с плавающей точкой, он становится опасным инструментом, способным разрушить память, поскольку записанные через него данные могут оказаться буквально где угодно. Аналогично, когда указатели используются для отслеживания распределенной памяти, слишком легко забыть освободить память, когда она больше не нужна. Это приводит к ее утечке.

Очевидно, что лучшие практики программирования — один из способов избежать проблем с памятью, связанных с указателями. И конечно, можно написать надежный код, который, по сути, никогда не теряет память и не вызывает нарушение ее целостности. Тем не менее наличие инструмента, который поможет обнаружить потенциальное повреждение памяти и проблемы утечки, безусловно, не повредит. К счастью, существует много таких инструментов.

Мой личный фаворит — Rational Purify компании IBM, который входит в состав набора инструментов Purify Plus. Purify встраивается в код перед его запуском, чтобы подключиться ко всем разыменованиям указателей и всем операциям выделения памяти и ее освобождения, выполняемых кодом. Запуская код в Purify, вы получаете оперативный отчет о проблемах, реальных и потенциальных, которые в нем встретились. И когда программа завершается, получаете подробный отчет об утечке памяти. Каждая проблема связывается с тем местом в коде, где она возникла, так что отслеживание и устранение подобных ошибок — довольно легкий процесс. Вы найдете более подробную информацию о Purify по адресу www-306.ibm.com/software/awdtools/purify.

Два других популярных инструмента — Insure++ компании Parasoft и Valgrind Джулиана Сьюарда и группы разработчиков Valgrind. Они предоставляют как средства отладки памяти, так и средства профилирования.

2.5. Другие инструменты

В арсенале программиста игр есть ряд других часто используемых инструментов. Я не буду подробно останавливаться на них, но следующий список даст вам представление о том, какие существуют, и подскажет, где их искать, если вы хотите узнать больше.

- *Инструменты сравнения файлов*. Инструмент сравнения, или *инструмент diff*, — это программа, которая сравнивает две версии текстового файла и определяет, что между ними изменилось. (Более подробную информацию об этих инструментах вы найдете на страницах en.wikipedia.org/wiki/Diff и https://ru.wikipedia.org/wiki/Diff.) Различия обычно рассчитываются построчно, хотя современные diff-средства могут показывать и диапазон измененных символов на измененной строке. Большинство систем контроля версий поставляются с инструментом сравнения. Некоторым программистам нравится определенный инструмент сравнения, и они настраивают свое программное обеспечение контроля версий на использование инструмента по своему выбору. Популярные инструменты — ExamDiff (www.prestosoft.com/edp_examdiff.asp), AraxisMerge (www.araxis.com), WinDiff (доступен опционально для большинства версий Windows, а также на многих независимых веб-сайтах) и пакет GNU diff tools (www.gnu.org/software/diffutils/diffutils.html).
- *Трехсторонние инструменты слияния*. Когда два человека редактируют один и тот же файл, генерируются два независимых набора различий. Инструмент, который может объединить их в окончательную версию файла, содержащую изменения, внесенные обоими участниками, называется инструментом трехстороннего слияния. Название «трехсторонний» обусловлено тем, что задействованы три версии файла: оригинал, версия пользователя A и версия пользователя B. (По адресу en.wikipedia.org/wiki/3-way_merge#Three-way_merge описаны технологии двухстороннего и трехстороннего слияния.) Многие инструменты слияния поставляются со связанным инструментом сравнения. Популярные инструменты слияния — AraxisMerge (www.araxis.com) и WinMerge (winmerge.org). Perforce также поставляется с отличным инструментом трехстороннего слияния (www.perforce.com/perforce/products/merge.html).
- *Шестнадцатеричные редакторы*. Шестнадцатеричный редактор (hex-редактор) — это программа, используемая для проверки и изменения содержимого двоичных файлов. Данные обычно отображаются в виде целых чисел в шестнадцатеричном формате, отсюда и название. Большинство хороших шестнадцатеричных

редакторов могут выводить данные в виде целых чисел от 1 до 16 байтов каждый, в 32- и 64-битном формате с плавающей запятой и в виде текста ASCII. Hex-редакторы особенно полезны при отслеживании проблем с бинарными файловыми форматами или обратном инжиниринге неизвестного бинарного формата. Разработчикам игровых движков довольно часто приходится решать эти задачи. Существует не менее миллиона шестнадцатеричных редакторов. Мне повезло с HexEdit компании Expert Commercial Software (www.expertcomsoft.com/index.html), но вы можете выбрать что-то иное.

Как программист игрового движка, вы, несомненно, найдете другие инструменты, которые облегчат вашу жизнь, но я надеюсь, что в этой главе рассмотрены основные инструменты для ежедневной работы.

3 Основы разработки игрового ПО

В этой главе обсудим базовые знания, необходимые любому профессиональному программисту. Мы исследуем основные системы счисления и представления, компоненты и архитектуру типичного компьютера и его ЦП, машинный и ассемблерный языки и язык программирования C++. Рассмотрим также некоторые ключевые концепции объектно-ориентированного программирования (ООП), а затем углубимся в сложные темы, которые имеют неоценимое значение при разработке программного обеспечения, особенно при создании игр. Как и сведения из главы 2, часть этих материалов уже могжет быть знакома некоторым читателям. Тем не менее я настоятельно рекомендую всем хотя бы прочесть эту главу, чтобы мы отправились в путь с одинаковым набором инструментов и объемом знаний.

3.1. Обзор C++ и лучшие практики

Поскольку C++ является, пожалуй, наиболее часто используемым в игровой индустрии языком, в этой книге сосредоточимся в первую очередь на нем. Однако большинство понятий, которые мы рассмотрим, одинаково хорошо применимы к любому объектно-ориентированному языку программирования. Конечно, в игровой индустрии используется множество других языков: императивные, такие как C, объектно-ориентированные, например C# и Java, скриптовые, такие как Python, Lua и Perl, функциональные, такие как Lisp, Scheme и F#, и этот список можно продолжить. Я настоятельно рекомендую каждому программисту выучить как минимум два языка высокого уровня (чем больше, тем лучше), а также хотя бы немного разобраться в программировании на *языке ассемблера*. Каждый новый язык, который вы изучаете, расширяет ваш кругозор и позволяет более глубоко и профессионально мыслить о программировании в целом. Теперь давайте обратим внимание на концепции ООП в целом и C++ в частности.

3.1.1. Краткий обзор ООП

Многое из того, что мы обсудим в этой книге, предполагает, что у вас сложилось четкое понимание принципов объектно-ориентированного проектирования. Если вам надо освежить свои знания, то следующий раздел поможет сделать приятный и быстрый обзор. Если вы не знаете, о чем в нем пойдет речь, то я настоятельно рекомендую прочитать одну или две книги по ООП (например, [7]) и, в частности, по C++ (например, [46] и [36]), прежде чем продолжить.

Классы и объекты

Класс — это набор атрибутов (данных) и поведений (кода), которые вместе образуют полезное, значимое целое. Класс — это *спецификация*, описывающая, как должны создаваться отдельные экземпляры класса, известные как *объекты*. Например, ваш питомец Ровер является экземпляром класса `dog`. Таким образом, между классом и его экземплярами существует отношение «один ко многим».

Инкапсуляция

Инкапсуляция означает, что объект предоставляет лишь ограниченный интерфейс общения с внешним миром, а внутреннее состояние объекта и детали реализации остаются скрытыми. Инкапсуляция упрощает жизнь пользователю класса, потому что ему нужно понимать только ограниченный интерфейс класса, а не потенциально сложные детали его реализации. Это позволяет программисту, написавшему класс, гарантировать, что его экземпляры всегда находятся в логически непротиворечивом состоянии.

Наследование

Наследование позволяет определять новые классы как *расширения* ранее существовавших. Новый класс изменяет или расширяет данные, интерфейс и/или поведение существующего класса. Если класс `Child` расширяет класс `Parent`, мы говорим, что `Child` *наследует* от него или является *производным* от `Parent`. В этих отношениях класс `Parent` известен как *базовый класс* или *суперкласс*, а класс `Child` является *производным классом*, или *подклассом*. Очевидно, что наследование определяет иерархические (древовидные) отношения между классами.

Наследование создает отношения «являться» между классами. Например, круг *является* разновидностью фигуры. Если бы мы писали приложение для 2D-рисования, вероятно, имело бы смысл выводить класс `Circle` (Круг) из базового класса `Shape` (Фигура).

Мы можем рисовать диаграммы иерархий классов, используя соглашения, определенные унифицированным языком моделирования (Unified Modeling Language, UML). В этих обозначениях прямоугольник представляет класс, а стрелка с полым треугольным наконечником — наследование. Стрелка наследования идет от дочернего класса к родительскому. На рис. 3.1 приведен пример простой иерархии классов, представленной в виде *статической диаграммы классов* UML.

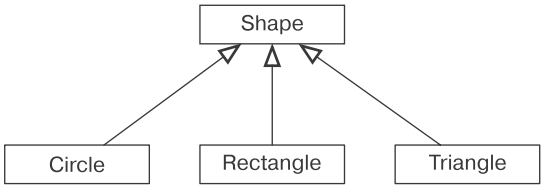


Рис. 3.1. UML-диаграмма статического класса, изображающая простую иерархию классов

Множественное наследование. Некоторые языки поддерживают *множественное наследование* (multiple inheritance, MI), что означает: класс может иметь более одного родительского класса. Теоретически MI способно быть довольно элегантным, но на практике такой дизайн обычно вызывает большую путаницу и много технических трудностей (см. en.wikipedia.org/wiki/Multiple_inheritance и https://ru.wikipedia.org/wiki/Множественное_наследование). Это связано с тем, что множественное наследование превращает простое *дерево* классов в потенциально сложный *граф*. Граф классов может иметь все виды проблем, которые никогда не мешают простому дереву, например проблему *«смертельного бриллианта»* или ромбовидного наследования (https://ru.wikipedia.org/wiki/Ромбовидное_наследование), в котором производный класс заканчивается двумя копиями базового класса прародителя (рис. 3.2).

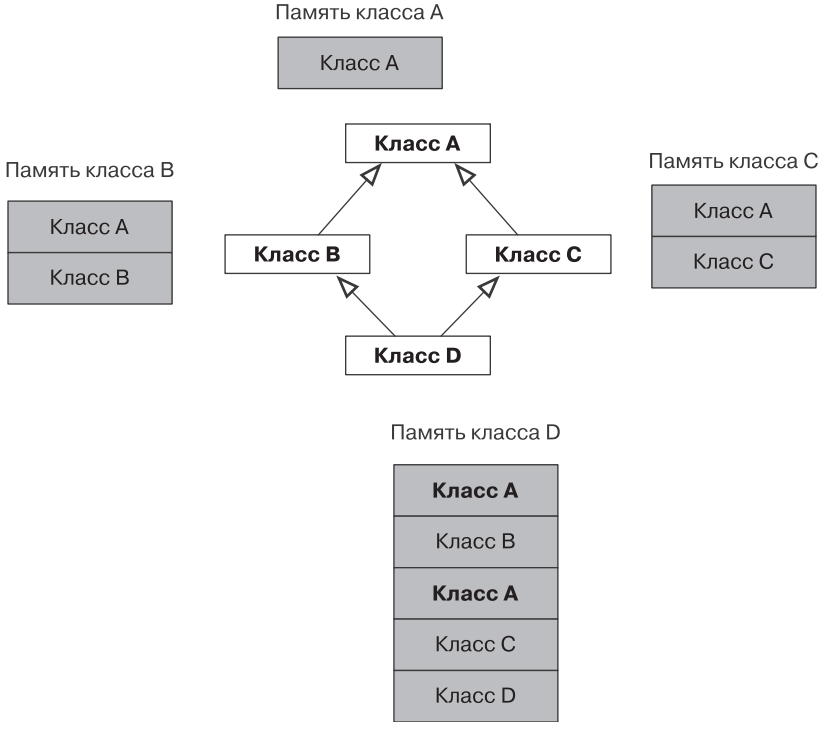


Рис. 3.2. «Смертельный бриллиант» в иерархии множественного наследования

(В C ++ *виртуальное наследование* позволяет избежать этого удвоения данных.) Множественное наследование также усложняет приведение типов, поскольку фактический адрес указателя может меняться в зависимости от того, к какому базовому классу он ведет. Это происходит из-за наличия нескольких *виртуальных таблиц (vtable)* указателей внутри объекта.

Большинство разработчиков программного обеспечения на C++ избегают множественного наследования или разрешают его в ограниченной форме. Общее правило заключается в том, чтобы разрешить множественное наследование только для простых классов без родителей или иерархию со строго единым наследованием. Такие классы иногда называют *смешанными*, потому что они могут использоваться для введения новой функциональности в произвольных точках в дереве классов. На рис. 3.3 приведен искусственный пример смешанного класса.

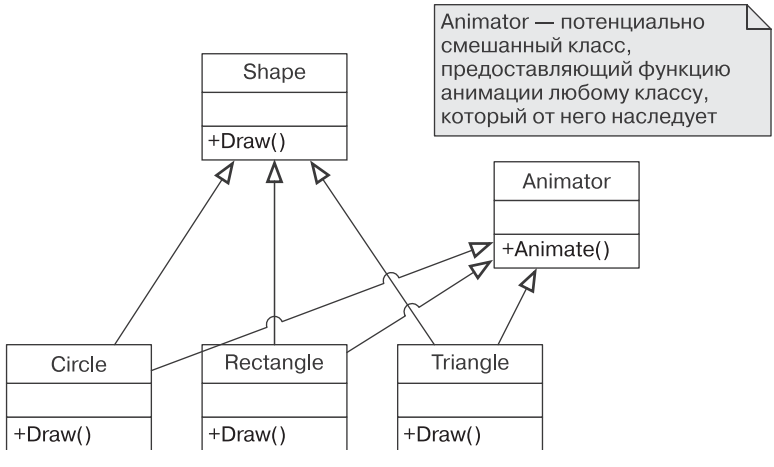


Рис. 3.3. Пример смешанного класса

Полиморфизм

Полиморфизм — это языковая особенность, которая позволяет манипулировать коллекцией объектов разных типов через единый *общий интерфейс*. Общий интерфейс делает набор разнородных объектов одинаковыми с точки зрения кода, использующего интерфейс.

Например, программе 2D-рисования может быть предоставлен список различных фигур для воспроизведения на экране. Один из способов создать эту неоднородную коллекцию фигур — использовать оператор `switch` для выполнения различных команд рисования для каждого отдельного типа фигуры:

```
void drawShapes(std::list<Shape*>& shapes)
{
    std::list<Shape*>::iterator pShape = shapes.begin();
    std::list<Shape*>::iterator pEnd = shapes.end();
```

```
    for ( ; pShape != pEnd; pShape++)
    {
        switch (pShape->mType)
        {
        case CIRCLE:
            // нарисовать фигуру в виде круга
            break;

        case RECTANGLE:
            // нарисовать фигуру в виде прямоугольника
            break;

        case TRIANGLE:
            // нарисовать фигуру в виде треугольника
            break;

        //...
        }
    }
}
```

Проблема этого подхода заключается в том, что функция `drawShapes()` должна знать обо всех видах фигур, которые можно нарисовать. Это хорошо в простом примере, но по мере расширения и усложнения кода может стать затруднительно добавлять новые типы фигур в систему. Всякий раз, когда добавляется новый тип фигуры, нужно найти каждое место в коде, в котором необходимо знание набора типов фигур, например, оператор `switch`, и добавить фрагмент для обработки нового типа.

Решение проблемы состоит в том, чтобы изолировать большую часть кода от любых типов объектов, с которыми он способен иметь дело. Для этого мы можем определить классы для каждого из типов фигур, которые хотим обрабатывать. Все эти классы наследуются от общего базового класса `Shape`. *Виртуальная функция* — основной механизм полиморфизма языка C++ — будет определена как `Draw()`, и каждый отдельный класс фигур будет реализовывать эту функцию по-своему. Не зная, для какого конкретного типа фигуры она была вызвана, функция рисования теперь может просто вызывать функцию `Draw()` для каждой фигуры по очереди:

```
struct Shape
{
    virtual void Draw() = 0; // чистая виртуальная функция
    virtual ~Shape() { }     // обеспечивает виртуальность деструктора
};

struct Circle : public Shape
{
    virtual void Draw()
    {
        // рисуем фигуру в виде круга
    }
};
```

```
struct Rectangle : public Shape
{
    virtual void Draw()
    {
        // рисуем фигуру в виде прямоугольника
    }
};

struct Triangle : public Shape
{
    virtual void Draw()
    {
        // рисуем фигуру в виде треугольника
    }
};

void drawShapes(std::list<Shape*>& shapes)
{
    std::list<Shape*>::iterator pShape = shapes.begin();
    std::list<Shape*>::iterator pEnd = shapes.end();

    for ( ; pShape != pEnd; pShape++)
    {
        pShape->Draw(); // вызов виртуальной функции
    }
}
```

Композиция и агрегация

Композиция — это практика использования *группы взаимодействующих* объектов для выполнения задачи высокого уровня. Композиция создает отношения «имеет» или «использует» между классами. (Отношение «имеет» называется *композицией*, а «использует» — *агрегацией*.) Например, у космического корабля есть двигатель, который, в свою очередь, имеет топливный бак. Композиция/агрегация обычно приводит к тому, что отдельные классы становятся более простыми и более сфокусированными. Неопытные ООП-программисты часто слишком полагаются на наследование и склонны недостаточно задействовать агрегацию и композицию.

В качестве примера представим, что мы разрабатываем графический интерфейс пользователя для игры. У нас есть класс `Window`, который представляет любой прямоугольный элемент GUI. Есть также класс `Rectangle`, который инкапсулирует математическую концепцию прямоугольника. Наивный программист может получить класс `Window` из класса `Rectangle` (используя отношение is-a). Но в более гибком и хорошо инкапсулированном дизайне класс `Window` будет *ссылаться* на прямоугольник или *содержать* его, используя отношение «имеет» или «использует». Это делает оба класса более простыми и сфокусированными и позволяет легче тестировать, отлаживать и повторно использовать их.

Паттерны разработки

Когда одна и та же проблема возникает вновь и вновь и многие программисты используют очень похожее решение, говорят, что возник *паттерн проектирования*. В ООП ряд общих шаблонов был определен и описан различными авторами. Наиболее известной книгой на эту тему, вероятно, является книга «банды четырех» [19].

Вот несколько распространенных шаблонов проектирования общего назначения.

- *«Одиночка»*, или *«Синглтон»* (Singleton). Гарантирует, что конкретный класс имеет только один экземпляр), и обеспечивает глобальную точку доступа к нему.
- *«Итератор»* (Iterator). Обеспечивает эффективный способ доступа к отдельным элементам коллекции, не раскрывая основную информацию о конкретной реализации. Итератор знает подробности реализации коллекции, так что ее пользователям это не нужно.
- *«Абстрактная фабрика»* (Abstract factory). Абстрактная фабрика предоставляет интерфейс для создания семейств связанных или зависимых классов без указания их конкретных классов.

Игровая индустрия имеет собственный набор паттернов проектирования для решения проблем в каждой конкретной области от рендеринга до обработки столкновений, от анимации до звука. В некотором смысле эта книга посвящена шаблонам проектирования высокого уровня, распространенным в архитектуре современного игрового 3D-движка.

Дворники и RAII. В качестве очень полезного примера шаблона проектирования кратко рассмотрим шаблон «Получение ресурсов есть инициализация» (Resource acquisition is initialization, RAII). В этом паттерне получение и освобождение ресурса, такого как файл, блок динамически выделяемой памяти или блокировка мьютекса, связаны с конструктором и деструктором класса соответственно. Это не позволяет программистам случайно забыть освободить ресурс — вы просто создаете локальный экземпляр класса, чтобы получить ресурс, и позволяете ему выйти из области видимости, чтобы освободить его автоматически. В Naughty Dog мы называем такие классы *дворниками*, потому что они «убирают» за вами.

Например, всякий раз, когда нужно выделить память для определенного типа распределителя памяти, мы *вталкиваем* этот распределитель в глобальный *стек распределителей* и всегда должны помнить о том, что, когда заканчиваем, нам необходимо *вытолкнуть* распределитель из стека. Чтобы сделать этот процесс более удобным и менее склонным к ошибкам, мы используем механизм *выделения с дворником*. Конструктор этого крошечного класса помещает распределитель в стек, а его деструктор выталкивает его:

```
class AllocJanitor
{
public:
    explicit AllocJanitor(mem::Context context)
```

```
    {
        mem::PushAllocator(context);
    }
    ~AllocJanitor()
    {
        mem::PopAllocator();
    }
};
```

Чтобы использовать класс-дворник, мы создаем его локальный экземпляр. Когда этот экземпляр выходит из области видимости, распределитель будет выталкиваться автоматически:

```
void f()
{
    // делаем что-то...

    // выделить временные буферы из однокадрового распределителя
    {
        AllocJanitor janitor(mem::Context::kSingleFrame);

        U8* pByteBuffer = new U8[SIZE];
        float* pFloatBuffer = new float[SIZE];

        // использовать буферы ...

        // (Примечание: не нужно освобождать память,
        // потому что мы применили однокадровый распределитель)
    }   // Дворник выталкивает распределитель из стека,
        // когда тот пропадает из области видимости

    // делаем что-то еще...
}
```

Обратитесь к сайту en.cppreference.com/w/cpp/language/raii для получения дополнительной информации об очень полезном паттерне RAII.

3.1.2. Стандартизация языка C++

С момента своего создания в 1979 году язык C++ постоянно развивался. Бьярн Страуструп, его изобретатель, назвал язык *C with Classes* («Си с классами»), но в 1983-м он был переименован в C++. Международная организация по стандартизации (ISO, www.iso.org) стандартизировала язык в 1998 году — эта версия сегодня известна как C++98. С тех пор ISO периодически публикует обновленные стандарты для языка C++, призванные сделать его более мощным, простым в использовании и менее двусмысленным. Эти цели достигаются путем уточнения семантики и правил языка, добавления новых, более мощных языковых функций, а также исключения или удаления тех его аспектов, которые вызывают проблемы или оказались непопулярными.

На момент публикации этой книги последний вариант стандарта языка программирования на C++ был C++17, он опубликован 31 июля 2017 года. Следующая итерация стандарта, C++2a, находилась в стадии разработки. Далее в хронологическом порядке приведены различные версии стандарта C++.

- *C++98* — первый официальный стандарт C++, установленный ISO в 1998 году.
- *C++03* — представлен в 2003 году для решения различных проблем, выявленных в стандарте C++98.
- *C++11* (на протяжении разработки известный как *C++0x*) — одобрен ISO 12 августа 2011 года. C++11 добавил в язык много новых мощных функций, в том числе:
  - типобезопасный литерал `nullptr` для замены подверженного ошибкам макроса `NULL`, унаследованного от языка «Си»;
  - ключевые слова `auto` и `decltype` для типов интерфейса;
  - синтаксис *trailing return type*, который позволяет использовать `decltype` входных аргументов функции для описания типа ее возвращаемого значения;
  - ключевые слова `override` и `final` для улучшения выразительности при определении и переопределении виртуальных функций;
  - функции по умолчанию (`default`) и удаленные функции (`delete`), которые позволяют программисту явно задавать применение реализации функции, созданной компилятором, или, наоборот, предотвращать ее определение и вызов;
  - делегирование конструкторов — способность одного конструктора вызывать другой в том же классе;
  - строго типизированные перечисления;
  - ключевое слово `constexpr` для определения значений констант во время компиляции путем вычисления выражений во время компиляции;
  - унифицированный синтаксис инициализации, который расширяет исходные инициализаторы POD на основе скобок, чтобы охватить также не-POD-типы;
  - поддержку лямбда-функций и захвата переменных (замыкания);
  - введение rvalue-ссылок и семантики перемещения для более эффективной обработки временных объектов;
  - стандартизированные спецификаторы атрибута для замены специфических для компилятора спецификаторов, таких как `__attribute__((...))` и `__declspec()`.

  C++11 также предоставил улучшенную и расширенную стандартную библиотеку, включающую поддержку многопоточности (параллельное программирование),

улучшенные возможности интеллектуальных указателей и расширенный набор общих алгоритмов.

- *C++14* — одобрен ISO 18 августа и выпущен 15 декабря 2014 года. Его дополнения и улучшения в C++11 включают в себя:
  - удаление типа возвращаемого значения, который во многих ситуациях позволяет объявлять типы возвращаемых функций с помощью простого ключевого слова `auto`, не нуждаясь в подробном завершающем выражении `decltype`, требующемся в C++11;
  - общие лямбда-выражения, которые могут действовать как шаблонная функция, используя `auto` для объявления своих входных аргументов;
  - способность инициализировать «захваченные» переменные в лямбда-выражениях;
  - двоичные литералы с `0b` в начале (например, `0b10110110`);
  - поддержку разделителей цифр в числовых литералах для улучшения читаемости (например, `1'000'000` вместо `1000000`);
  - шаблоны переменных, которые позволяют использовать синтаксис шаблонов при объявлении переменных;
  - ослабление некоторых ограничений на `constexpr`, в том числе возможность применения `if`, `switch` и циклов внутри константных выражений.
- *C++17* — опубликован ISO 31 июля 2017 года. Он расширяет и совершенствует C++14 во многих отношениях. В частности, изменения таковы:
  - удален ряд устаревших и/или опасных функций языка, включая триграфы, ключевое слово `register` и класс интеллектуальных указателей `auto_ptr`;
  - гарантирован *пропуск* копии, отсутствует ненужное копирование объекта;
  - спецификации исключений теперь являются частью системы типов. Это означает, что `void f() noexcept(true);` и `void f() noexpcept(false);` — разные типы;
  - добавлены два новых вида литералов — символьные литералы UTF-8 (например, `u8'x'`) и литералы с плавающей точкой с шестнадцатеричным основанием и десятичным показателем (например, `0xC.68p+2`);
  - введены *структурированные привязки*, что позволяет распаковывать значения из коллекции в отдельные переменные (например, `auto [a, b] = func_that_returns_a_pair();`), — синтаксис, поразительно похожий на возвращаемые множественные значения из функции через кортеж в Python;
  - добавлены полезные стандартизированные атрибуты, в том числе `[[fallthrough]]`, который позволяет явно задокументировать тот факт, что пропущенный оператор `break` в конструкции `switch` является преднамеренным, тем самым подавляя предупреждение, которое в противном случае было бы сгенерировано.

Дополнительные ресурсы

Существует множество отличных онлайн-ресурсов и книг, в которых подробно описываются функции C++11, C++14 и C++17, поэтому мы не будем здесь их рассматривать. Вот несколько полезных ссылок.

- Сайт en.cppreference.com/w/cpp содержит превосходное справочное руководство по C++, включающее выноски, такие как «начиная с C++11» или «до C++17», чтобы указать, когда определенные функции языка были добавлены в стандарт или удалены из него соответственно.
- Информацию об усилиях ISO в области стандартизации можно найти по адресу isocpp.org/std.
- Хорошее описание основных новых функций C++11 представлено на следующих сайтах:
  - www.codeproject.com/Articles/570638/Ten-Cplusplus-Features-Every-Cplusplus-Developer;
  - blog.smartbear.com/development/the-biggest-changes-in-c11-and-Why-You-You-Care-Care.
- На thbecker.net/articles/auto_and_decltype/section_01.html хорошо описаны `auto` и `decltype`.
- На www.drdobbs.com/cpp/the-c14-standard-what-you-need-to-know/240169034 изложены изменения C++14 по отношению к C++11.
- На isocpp.org/files/papers/p0636r0.html представлен полный список отличий стандарта C++17 от C++14.

Какие языковые функции использовать

Когда вы читаете о новых интересных функциях, добавляемых в C++, возникает соблазн задействовать все эти функции в своих движке или игре. Однако то, что функция существует, не означает, что ваша команда должна немедленно начать ее применять.

Мы в Naughty Dog придерживаемся консервативного подхода к внедрению новых языковых функций в свой код. Как часть стандартов программирования студии есть список возможностей языка C++, одобренных для использования в нашем коде в среде выполнения, и несколько более либеральный список возможностей языка, разрешенных в коде автономных инструментов. Для такого осторожного подхода есть несколько причин, которые я изложу далее.

Отсутствие полной поддержки. Компилятор может не полностью поддерживать передовые функции. Например, LLVM/Clang, компилятор C++, используемый на Sony Playstation 4, в настоящее время поддерживает весь стандарт C++11 в версиях 3.3 и последующих и весь C++14 в версиях 3.4 и новее. Но его поддержка C++17 распространяется на версии Clang с 3.5 по 4.0, и в настоящее время он не поддерживает черновой вариант стандарта C++2a. Кроме того, Clang компилирует код в режиме C++98 по умолчанию — поддержка некоторых более продвинутых стандартов принимается как расширения, но для обеспечения полной

поддержки необходимо передать конкретные аргументы командной строки компилятору. Для получения подробной информации см. clang.llvm.org/cxx_status.html.

Стоимость переключения между стандартами. Существует ненулевая плата за переключение кодовой базы с одного стандарта на другой. Поэтому игровой студии важно выбрать для поддержки наиболее продвинутый стандарт C++ и придерживаться его в течение разумного периода (например, во время существования одного проекта). Мы в Naughty Dog приняли стандарт C++11 сравнительно недавно и разрешили его использовать только в той ветви кода, в которой разрабатывается *The Last of Us Part II*. Код в ветви, используемый для *Uncharted 4: A Thief's End* и *Uncharted: Lost Legacy*, изначально был написан с помощью стандарта C++98, и мы решили, что довольно незначительные преимущества применения функций C++11 в этой кодовой базе не перевешивают стоимость и риски, связанные с переходом.

Вознаграждение за риск. Не все возможности языка C++ одинаковы. Некоторые функции полезны и в целом универсально приемлемы, например `nullptr`. Другие могут иметь как преимущества, так и недостатки. А некоторые языковые функции вообще можно счесть неприемлемыми для использования в коде движка в среде выполнения.

В качестве примера языковой функции, имеющей как преимущества, так и недостатки, рассмотрим новую интерпретацию ключевого слова `auto` в C++11. Оно, безусловно, делает переменные и функции более удобными для записи. Но программисты Naughty Dog признали, что чрезмерное применение `auto` может запутать код. Представьте, что вы пытаетесь прочитать файл `.cpp`, написанный кем-то другим, в котором практически все переменные, аргументы функции и возвращаемые значения объявляются с помощью `auto`. Это все равно что читать язык без типов, такой как Python или Lisp. Одним из преимуществ строго типизированного языка, такого как C++, является то, что программист способен быстро и легко определять типы всех переменных. Поэтому мы решили принять простое правило: `auto` можно задействовать только при объявлении итераторов, в ситуациях, когда другие подходы не работают (например, в определениях шаблонов), или в особых случаях, когда оно значительно улучшает ясность кода, удобочитаемость и удобство сопровождения. Во всех прочих случаях мы требуем применения явных объявлений типов.

В качестве примера языковой функции, которая может считаться неподходящей для использования в коммерческом продукте, таком как игра, *рассмотрим шаблонное метапрограммирование*. Библиотека Loki Андрея Александреску [3] активно задействует шаблонное метапрограммирование, чтобы выполнять довольно интересные и удивительные вещи. Однако полученный код труден для чтения, иногда непереносим, и его очень нелегко понять программистам. Ведущие специалисты Naughty Dog считают, что любой программист должен быть в состоянии быстро подключиться и отладить ошибку даже в коде, с которым он может быть плохо знаком. Таким образом, в Naughty Dog запрещено сложное шаблонное метапрограммирование в коде движка в среде выполнения, за исключением конкретных случаев, когда выгоды, если подсчитать, перевешивают затраты.

Помните: когда у вас есть молоток, все может выглядеть как гвоздь. Не поддавайтесь искушению использовать возможности языка только потому, что они есть, или потому, что они новы. Благодаря разумному и тщательно продуманному подходу можно создать стабильную кодовую базу, которая будет настолько проста, насколько это возможно, для понимания, анализа, отладки и поддержки.

3.1.3. Стандарты программирования: для чего и сколько

Обсуждение программистами соглашений по разработке часто приводит к жарким «религиозным» дебатам. Я не хочу разжигать здесь подобные дебаты, но пойду настолько далеко, что предположу: следовать хотя бы минимальному набору стандартов программирования — хорошая идея. Стандарты разработки существуют по двум основным причинам.

1. Одни стандарты делают код более читабельным, понятным и проще поддерживаемым.
2. Другие соглашения помогают запретить программистам выстрелить себе в ногу. Например, стандарт разработки может побудить программиста применять только меньшее, лучше тестируемое и менее подверженное ошибкам подмножество всего языка. Язык C++ изобилует возможностями, которыми легко злоупотребить, поэтому такой стандарт разработки особенно важен при использовании C++.

На мой взгляд, наиболее важным из того, чего нужно достичь в соглашениях по разработке, является следующее.

- *Тщательно продумывайте интерфейсы.* Сохраняйте свои интерфейсы (файлы `.h`) чистыми, простыми, легкими для понимания и хорошо прокомментированными.
- *Помните, что хорошие имена упрощают понимание и помогают избежать путаницы.* Старайтесь давать интуитивно понятные имена, которые напрямую соответствуют назначению рассматриваемых класса, функции или переменной. Не жалейте времени, придумывая хорошее имя. Избегайте схемы именования, которая требует, чтобы программисты применяли справочную таблицу для расшифровки смысла вашего кода. Помните, что языки программирования высокого уровня, такие как C++, предназначены для чтения *людьми*. (Если вы не согласны с этим, спросите себя, почему вы не пишете все программное обеспечение непосредственно на машинном языке.)
- *Не загромождайте глобальное пространство имен.* Используйте пространства имен C++ или общий префикс имен так, чтобы введенные вами символы не вступали в противоречие с символами в других библиотеках. (Но будьте осторожны, не злоупотребляйте пространствами имен и не вкладывайте их слишком глубоко.) Называйте `#defined`-символы с особой осторожностью: помните, что макросы препроцессора C++ являются просто текстовыми

подстановками, поэтому они пересекают все границы области видимости C/C++ и пространства имен.

- *Придерживайтесь лучших практик C++*. Книги серии *Effective C++* Скотта Мейерса [36], [37], его же *Effective STL* [38], а также *Large-Scale C++ Software Design* Джона Лакоса [31] содержат отличные рекомендации, которые помогут вам избежать неприятностей.
- *Будьте последовательны*. Правило, которое я пытаюсь использовать, таково: если вы пишете основную часть кода с нуля, можете придумать любое соглашение, которое вам нравится, но строго придерживайтесь его. При редактировании существующего кода стремитесь выполнять соглашения, которые были установлены ранее.
- *Делайте ошибки выделяющимися*. Джоэл Спольски написал отличную статью о соглашениях по разработке, которую можно найти по адресу www.joelonsoftware.com/article/Wrong.html. Он предполагает, что самый чистый код — это не обязательно код, который выглядит аккуратно на поверхностном уровне, а скорее код, который написан так, чтобы облегчить поиск общих ошибок программирования. Статьи Джоэла всегда веселые и познавательные, и я очень рекомендую их почитать.

3.2. Поиск и обработка ошибок

Есть несколько способов найти и обработать ошибки в игровом движке. Программисту, который создает игры, важно разбираться в различных механизмах, видеть их плюсы и минусы и понимать, когда использовать каждый из них.

3.2.1. Типы ошибок

В любом программном проекте существует два основных типа ошибок: *ошибки пользователя* и *ошибки программиста*. Ошибка пользователя возникает, когда пользователь программы делает что-то неправильное, например вводит неверные данные, пытается открыть несуществующий файл и т. п. Ошибка программиста — это результат ошибки в коде. Хотя она может быть вызвана и тем, что сделал пользователь. Суть ошибки программиста заключается в том, что этой проблемы можно было избежать, если бы он не допустил ошибки, и пользователь ожидает, что программа *должна* корректно обработать ситуацию.

Конечно, суть определения «пользователь» меняется в зависимости от контекста. В контексте игрового проекта пользовательские ошибки можно условно разделить на две категории: ошибки, вызванные человеком, играющим в игру, и ошибки, вызванные людьми, которые создают игру в среде разработки. Важно отслеживать, на какой тип пользователя влияет конкретная ошибка, и обрабатывать ее соответствующим образом.

На самом деле есть и третий тип пользователей — другие программисты вашей команды. (И если вы пишете часть промежуточного игрового ПО, такого как Havok или OpenGL, к третьей категории будут относиться другие программисты по всему миру, которые используют вашу библиотеку.) Именно здесь грань между *ошибками пользователя* и *ошибками программиста* сильно размывается. Представьте, что программист A пишет функцию `f()`, а программист B пытается ее вызвать. Если B вызывает `f()` с недопустимыми аргументами, например с нулевым указателем или индексом массива вне диапазона, то это может быть воспринято программистом A как ошибка пользователя, но с точки зрения B это будет ошибка программиста. (Конечно, можно также утверждать, что программист A должен был предвидеть передачу недопустимых аргументов и обработать их изящно, так что проблема на самом деле является ошибкой программиста со стороны A.) Следует помнить главное: различие между пользователем и программистом условно и зависит от контекста — это редко контраст между черным и белым.

3.2.2. Обработка ошибок

Требования к обработке ошибок двух типов существенно различаются между собой. *Ошибки пользователя* лучше всего обрабатывать как можно более изящно, отображая некоторую полезную информацию для пользователя, а затем позволяя ему продолжать работать — в случае игры продолжать играть. Ошибки программиста *не* должны обрабатываться с гибкой политикой «информируй и продолжай». Вместо этого лучше остановить программу и предоставить подробную информацию об отладке на низком уровне, чтобы программист мог быстро выявить и устранить проблему. В идеальном мире все ошибки программиста будут обнаружены и исправлены до того, как программное обеспечение станет общедоступным.

Обработка ошибок игрока

Когда пользователь — это человек, играющий в вашу игру, ошибки, очевидно, должны обрабатываться в контексте игрового процесса. Например, если игрок пытается перезарядить оружие, когда боеприпасы недоступны, звуковой сигнал и/или анимация могут указать ему на эту проблему, не выводя его из игры.

Обработка ошибок разработчика

Когда пользователь — это кто-то, кто делает игру, например художник, аниматор или игровой дизайнер, ошибки могут быть вызваны каким-то недействительным ресурсом. Например, анимация может быть связана с неправильным скелетом, текстура — иметь неправильный размер, аудиофайл мог быть выбран с неподдерживаемой частотой дискретизации. Для устранения таких *ошибок разработчиков* существует два конкурирующих подхода.

С одной стороны, важно не допускать, чтобы плохие игровые ресурсы сохранялись слишком долго. Игра, как правило, содержит буквально тысячи ресурсов,

и проблемный может быть потерян. В этом случае возникает риск того, что плохой ресурс доживет до финального релиза игры. Если довести эту точку зрения до абсурда, то лучший способ справиться с плохими игровыми ресурсами — предотвратить запуск всей игры при возникновении даже одного проблемного. Это, безусловно, сильный стимул для разработчика, создавшего недействительный актив, чтобы немедленно удалить или исправить его.

С другой стороны, разработка игр — это беспорядочный итеративный процесс и генерация совершенных ресурсов с первого раза действительно редка. В соответствии с этим подходом игровой движок должен быть устойчивым практически к любой проблеме, которую можно себе представить, чтобы работа могла продолжаться даже перед лицом совершенно неверных данных игровых активов. Но это тоже неидеальный вариант, потому что игровой движок стал бы раздутым из-за кода перехвата и обработки ошибок, который не понадобится после того, как разработка оптимизируется и игра будет выпущена в свет. К тому же слишком высокой остается вероятность выхода игры с плохими активами.

По моему опыту, лучший подход — найти золотую середину между этими двумя крайностями. Когда возникает ошибка разработчика, я хотел бы, *чтобы она была очевидной*, но при этом позволяла команде продолжить работу даже при наличии проблемы. Чрезвычайно дорого мешать другим разработчикам в команде трудиться только потому, что один разработчик попытался добавить в игру недопустимый ресурс. Игровая студия хорошо платит своим сотрудникам, и, когда несколько членов команды находятся в простое, затраты умножаются на количество людей, которые не работают. Конечно, мы должны обрабатывать ошибки таким образом, только когда это целесообразно, не затрачивая чрезмерного количества времени разработчиков и не раздувая код.

В качестве примера предположим, что невозможно загрузить конкретный меш. На мой взгляд, лучше всего в игровом мире в тех местах, где должен был располагаться этот меш, нарисовать большую красную рамку, допустим, с текстовой строкой, которая гласит: «Меш *такой-то* не удалось загрузить». Это лучше, чем печатать в журнале ошибок сообщение, которое легко пропустить. И *гораздо* лучше, чем просто вылететь из игры, потому что тогда никто не сможет работать до того, как ссылка на меш будет исправлена на корректную. Конечно, при возникновении особенно вопиющих проблем можно просто выплюнуть сообщение об ошибке и вылететь. Для всех видов проблем не существует «серебряной пули», и умение выбирать, какой тип обработки ошибок следует применять в данной ситуации, придет с опытом.

Обработка ошибок программиста

Лучший способ обнаружить и обработать ошибки программиста, также называемые *багами* (bugs), — чаще вставлять код проверки на ошибки в исходный код и останавливать программу при неудачном завершении проверки. Такой механизм известен как *система утверждений*, мы подробно рассмотрим утверждения далее. Конечно, как уже говорилось, для одного программиста ошибка пользователя явля-

ется ошибкой другого программиста, следовательно, утверждения — это не всегда правильный способ обработки каждой ошибки программиста. Верный выбор между утверждением и более изящной техникой обработки ошибок — это навык, который развивается со временем.

3.2.3. Реализация обнаружения ошибок и их обработки

Мы рассмотрели некоторые философские подходы к обработке ошибок. Теперь обратим внимание на выбор, который мы делаем как программисты, когда дело доходит до реализации обнаружения ошибок и обработки кода.

Коды возврата ошибок

Распространенным подходом к обработке ошибок является возвращение некоторого кода ошибки из функции, в которой проблема обнаружена впервые. Это может быть логическое значение, указывающее на успех или неудачу, или невозможное значение, выходящее за пределы диапазона обычно возвращаемых результатов. Например, функция, которая возвращает положительное целое число или значение с плавающей точкой, может вернуть отрицательное значение, чтобы показать, что произошла ошибка. Еще лучше, если функция будет спроектирована так, чтобы вместо логического или невозможного значения возвращать значение перечисления, указывающее на успех или неудачу. Это четко отделяет код ошибки от выходных данных функции, и, если что-то пошло не так, можно определить точный характер проблемы (например, `enum Error {kSuccess, kAssetNotFound, kInvalidRange, ...}`).

Вызывающая функция должна перехватывать коды возврата ошибок и действовать определенным образом. Это может быть немедленная обработка ошибки или обход проблемы, завершение собственного выполнения и затем передача кода ошибки той функции, которая *ее* вызвала.

Исключения

Коды возврата ошибок — это простой и надежный способ связи и реагирования на ошибки. Однако они имеют свои недостатки. Вероятно, самая большая проблема с кодами возврата ошибок состоит в том, что функция, которая определяет ошибку, может быть совершенно не связана с функцией, способной решить проблему. В наихудшем сценарии функция с глубиной 40 вызовов в стеке вызовов может обнаружить проблему, которую способны обработать только игровой цикл верхнего уровня или функция `main()`. В этом сценарии каждая из 40 функций в стеке вызовов должна быть написана так, чтобы иметь возможность передавать соответствующий код ошибки вплоть до функции обработки ошибок верхнего уровня.

Один из способов решения этой проблемы — исключение. *Обработка исключений* — очень мощная функция C++. Это позволяет функции, которая обнаруживает

проблему, сообщать об ошибке остальной части кода, ничего не зная о том, какая функция может обработать ошибку. Когда выдается исключение, информация об ошибке помещается в объект данных по выбору программиста, известный как *объект исключения*. Затем стек вызовов автоматически разматывается в поисках вызывающей функции, чей вызов обернут в блок `try-catch`. Если найден блок `try-catch`, объект исключения сопоставляется со всеми возможными случаями `catch` и, если есть совпадение, выполняется соответствующий блок кода `catch`. Деструкторы любых автоматических переменных вызываются по мере необходимости в процессе разматывания стека.

Возможность отделить обнаружение ошибок от их обработки таким чистым способом, безусловно, привлекательна, а обработка исключений — отличный выбор для некоторых программных проектов. Однако обработка исключений добавляет некоторые накладные расходы в программу. Кадр стека любой функции, содержащей блок `try-catch`, должен быть дополнен, и в нем следует хранить дополнительную информацию, необходимую для процесса разматывания стека. Кроме того, если даже только одна функция в вашей программе (или библиотека, с которой связана программа) применяет обработку исключений, то вся программа должна использовать обработку исключений — компилятор не знает, какие функции могут находиться над вами в стеке вызовов, когда происходит исключение.

Однако можно создать песочницу для библиотеки или библиотек, которые используют обработку исключений, чтобы не писать весь игровой движок с включенными исключениями. Чтобы сделать это, вы должны обернуть все вызовы API в рассматриваемых библиотеках в функции, реализованные в модуле компиляции, для которого включена обработка исключений. Каждая из этих функций будет перехватывать все возможные исключения в блоке `try-catch` и преобразовывать их в коды возврата ошибок. Поэтому любой код, связанный с вашей библиотекой-оберткой, может безопасно отключить обработку исключений.

Возможно, более важным, чем вопрос накладных расходов, является то, что исключение в некоторых отношениях не лучше, чем простая инструкция `goto`. Джоэл Спольски из Microsoft и Fog Creek Software утверждает, что исключения на самом деле *хуже*, чем `goto`, потому что их нелегко увидеть в исходном коде. Функция, которая не генерирует и не перехватывает исключения, тем не менее может быть вовлечена в процесс раскручивания стека, если оказывается зажатой между такими функциями в стеке вызовов. И процесс разматывания сам по себе несовершенен: состояние программного обеспечения легко может остаться недопустимым, если программист не примет во внимание все способы создания исключения и не обработает их соответствующим образом. Это может затруднить написание надежного программного обеспечения. Когда есть возможность получить исключение, почти каждая функция в коде должна быть готовой к тому, что все ее локальные объекты будут уничтожены.

Еще одна проблема, связанная с обработкой исключений, — это стоимость. Хотя в теории современные структуры обработки исключений не создают дополнительных накладных расходов ко времени выполнения при отсутствии ошибок,

на практике это работает не всегда. Например, код, который компилятор добавляет в функции для раскрутки стека вызовов при возникновении исключения, имеет тенденцию увеличивать размер кода в целом. Это может ухудшить производительность I-кэша или привести к тому, что компилятор решит не включать функцию, которую стоило бы включить.

Очевидно, есть несколько довольно веских аргументов для полного *отключения* обработки исключений в игровом движке. Этот подход используется в Naughty Dog, а также в большинстве проектов, над которыми я работал в Electronic Arts и Midway. Будучи директором движка в Insomniac Games, Майк Актон неоднократно возражал против использования обработки исключений во время выполнения в коде игры. JPL и NASA также запрещают обработку исключений в своем критически важном программном обеспечении, вероятно, по тем же причинам, по которым мы склонны избегать его в игровой индустрии. Ваше мнение, безусловно, может быть иным. Не существует идеального инструмента и единственно верного способа что-либо сделать. При разумном применении исключения *способны* облегчить написание кода и работу с ним, просто будьте осторожны!

В Сети есть много интересных статей на эту тему. Вот хорошая серия, которая охватывает большинство ключевых вопросов и рассматривает разные точки зрения:

- www.joelonsoftware.com/items/2003/10/13.html;
- www.nedbatchelder.com/text/exceptions-vs-status.html;
- www.joelonsoftware.com/items/2003/10/15.html.

Исключения и RAII. Шаблон «Получение ресурсов есть инициализация» (RAII) часто используется в сочетании с *обработкой исключений*: конструктор пытается получить нужный ресурс и выдает исключение, если это ему не удается. Так сделано для того, чтобы избежать необходимости проверок `if` для состояния объекта после того, как он был создан: если конструктор возвращается без исключения, мы точно знаем, что ресурс успешно получен.

Тем не менее шаблон RAII может задействоваться даже без исключений. Все, что требуется, — это небольшая дисциплинированность, для того чтобы проверить состояние каждого нового объекта ресурса при его создании. После этого можно использовать все остальные преимущества RAII. (Исключения также могут быть заменены проверкой утверждений, сигнализирующих о каком-то сбое получения ресурсов.)

Утверждения

Утверждение — это строка кода, которая проверяет выражение. Если выражение истинно, ничего не происходит. Но если оно оценивается как ложное, программа останавливается, печатается сообщение об ошибке и по возможности вызывается отладчик.

Утверждения проверяют предположения программиста, действуют подобно *минам* для ошибок. Они проверяют код при его написании, чтобы убедиться, что

он функционирует должным образом. А также гарантируют, что исходные предположения будут оставаться истинными в течение длительного времени, даже если код вокруг них будет постоянно меняться и развиваться. Например, если программист изменяет код, который раньше работал, но случайно нарушает его первоначальные предположения, он нарывается на мину. Та немедленно информирует программиста о проблеме и позволяет ему исправить ситуацию с минимальными усилиями. Без утверждений ошибки имеют тенденцию прятаться и проявляться позже такими способами, которые трудно отследить, и процесс этот занимает много времени. А с утверждениями, встроенными в код, ошибки проявляются в момент возникновения, что обычно лучше всего для устранения проблемы, так как изменения кода, вызвавшие проблему, еще свежи в памяти программиста. Стив Магуайр подробно рассматривает утверждения в своей книге *Writing Solid Code* [35], которую обязательно нужно прочесть.

Затратами ресурсов на проверку утверждений обычно можно пожертвовать во время разработки, но перед выпуском игры удаление утверждений способно вернуть критически необходимую часть производительности. По этой причине утверждения, как правило, реализуются таким образом, чтобы можно было их исключить из исполняемого файла в неотладочных конфигурациях сборки. Реализующий это макрос `assert()` в C определяется заголовочным файлом `<assert.h>` стандартной библиотеки, в C++ это заголовочный файл `<cassert>`.

Определение `assert()` стандартной библиотеки приводит к появлению утверждения в отладочных сборках (сборках с определенным ключом препроцессора `DEBUG`) и его отсутствию в сборках без отладки (сборках с определенным ключом препроцессора `NDEBUG`). В игровом движке вам может потребоваться более детальный контроль над тем, какие конфигурации сборки сохраняют утверждения, а какие удаляют их. Например, ваша игра поддерживает не только конфигурацию сборки, отладки и разработки — у вас также может быть готовая сборка с включенной глобальной оптимизацией и, вероятно, даже сборка PGO для пользования инструментами оптимизации на основе профилей (см. подраздел 2.2.4). Или же вы захотите определить различные варианты утверждений — одни всегда сохраняются даже в релизной версии игры, а другие исключаются даже из нефинальной версии. По этим причинам давайте посмотрим, как можно реализовать собственный макрос `ASSERT()` с помощью препроцессора C/C++.

Реализация утверждений. Утверждения обычно реализуются с помощью комбинации макроса `#defined`, который вычисляется как предложение `if/else`, функции, вызываемой при сбое утверждения (если выражение оценивается как `false`), и небольшого количества кода сборки, который останавливает программу и вызывает отладчик, если он подключен. Вот типичная реализация:

```
#if ASSERTIONS_ENABLED

// определяем некоторую функцию ассемблера, которая вызывает отладчик, —
// он будет различным для каждого целевого процессора
#define debugBreak() asm { int 3 }
```

```
// проверяем, ложно ли выражение
#define ASSERT(expr) \
    if (expr) { } \
    else \
    { \
        reportAssertionFailure(#expr, \
                __FILE__, __LINE__); \
        debugBreak(); \
    }

#else

#define ASSERT(expr)    // ничего не вычисляем

#endif
```

Разберем это определение, чтобы увидеть, как оно работает.

- Внешний `#if/#else/#endif` используется для удаления утверждений из кода. Когда `ASSERTIONS_ENABLED` не равен нулю, макрос `ASSERT()` определяется полностью и все проверки утверждений в коде будут включены в программу. Но когда утверждения отключены, `ASSERT(expr)` ничего не вычисляет и все его экземпляры в коде эффективно удаляются.
- Макрос `debugBreak()` выполняет любые инструкции на языке ассемблера, необходимые для остановки программы и запуска отладчика (если он подключен). Он различается от процессора к процессору, но обычно это единственная инструкция ассемблера.
- Сам макрос `ASSERT()` определяется с помощью полного оператора `if/else`, в отличие от одиночного `if`. Это сделано для того, чтобы макрос мог использоваться в любом контексте, даже в *других* операторах `if/else`, без скобок.

Вот пример того, что произойдет, если функция `ASSERT()` будет определена с применением одиночного выражения `if`:

```
// ПРЕДУПРЕЖДЕНИЕ: НЕХОРОШАЯ ИДЕЯ!
#define ASSERT(expr)  if (!(expr)) debugBreak()

void f()
{
    if (a < 5)
        ASSERT(a >= 0);
    else
        doSomething(a);
}
```

Это распространяется на следующий *неправильный* код:

```
void f()
{
    if (a < 5)
```

```
        if (!(a >= 0))
            debugBreak();
        else // Упс! Связано с неверным if()!
            doSomething(a);
}#
```

- Инструкция `else` макроса `ASSERT()` делает две вещи: отображает какое-то сообщение для программиста, указывающее, что пошло не так, а затем запускает отладчик. Обратите внимание на использование `#expr` в качестве первого аргумента функции вывода сообщений. Оператор препроцессора (`#`) заставляет выражение `expr` превращаться в строку, что позволяет печатать его как часть сообщения об ошибке утверждения.
- Обратите внимание также на применение `__FILE__` и `__LINE__`. Эти макросы, определенные компилятором, чудесным образом содержат имя файла `.cpp` и номер строки кода, в которой они появляются. Передавая их в функцию вывода сообщения об ошибке, мы можем вывести точное место возникновения проблемы.

Я настоятельно рекомендую использовать в вашем коде утверждения. Однако важно знать их стоимость с точки зрения производительности. Возможно, вы захотите определить два вида макросов утверждений: обычный макрос `ASSERT()`, который можно оставить активным во всех сборках, так что ошибки будет легко обнаружить, даже если вы работаете не в режиме отладки, и, например, макрос `SLOW_ASSERT()`, который будет активизироваться только в отладочных сборках. Последний макрос можно применять в тех местах, где стоимость проверки утверждений слишком высока, чтобы разрешить включать их в релизные сборки. Очевидно, что `SLOW_ASSERT()` не так полезен, потому что исключается из той версии игры, в которую каждый день играют ваши тестеры. Но по крайней мере эти утверждения активны, когда программисты отлаживают свой код.

Крайне важно правильно использовать утверждения. Их следует применять для обнаружения ошибок в самой программе, а не для выявления ошибок пользователя. Кроме того, утверждения всегда должны приводить к остановке всей игры, когда проверка заканчивается неудачей. Не стоит разрешать тестировщикам, художникам, дизайнерам и другим не инженерам пропускать утверждения. (Это немного похоже на то, как мальчик постоянно кричал: «Волк!» Если утверждения можно проигнорировать, они перестают иметь какое-либо значение и становятся неэффективными.) Другими словами, утверждения следует использовать только для выявления фатальных ошибок. Если нет ничего критического в том, чтобы продолжить после ошибки утверждения, то, вероятно, лучше уведомить пользователя о ней другим способом, например сообщением на экране или какой-нибудь уродливой ярко-оранжевой трехмерной графикой.

Утверждения во время компиляции. Одна из слабостей утверждений, как мы уже обсуждали, заключается в том, что закодированные в них условия проверяются только во время выполнения. Мы должны запустить программу, *и* рассматриваемый код должен фактически *выполняться*, чтобы можно было проверить условие утверждения.

Иногда условие, которое мы проверяем в утверждении, включает информацию, известную уже во время компиляции. Допустим, мы определяем структуру, которая по какой-то причине должна иметь размер точно 128 байт. Мы хотим добавить утверждение, чтобы компилятор выдал сообщение об ошибке, если другой программист (или вы сами в будущем) решит изменить размер структуры. Другими словами, мы хотели бы написать что-то вроде следующего:

```
struct NeedsToBe128Bytes
{
    U32    m_a;
    F32    m_b;
    // и т. п.
};

// К сожалению, это не работает...
ASSERT(sizeof(NeedsToBe128Bytes) == 128);
```

Проблема заключается в том, что макрос `ASSERT()` (или `assert()`) должен быть исполнен во время выполнения и невозможно даже поместить исполняемый код в глобальную область видимости файла `.cpp` вне определения функции. Решением этой проблемы является проверка утверждения во время компиляции, также известная как *статическое утверждение*.

Начиная с C++11, стандартная библиотека определяет макрос с именем `static_assert()`. Таким образом, можно переписать приведенный ранее пример следующим образом:

```
struct NeedsToBe128Bytes
{
    U32    m_a;
    F32    m_b;
    // и т. п.
};

static_assert(sizeof(NeedsToBe128Bytes) == 128,
    "wrong size");
```

Если вы не используете C++11, то всегда можете развернуть собственный макрос `STATIC_ASSERT()`. Его можно реализовать различными способами, но основная идея всегда одна и та же: макрос помещает в ваш код объявление о том, что допустимо в области видимости файла, оценивает требуемое выражение во время компиляции, а не во время выполнения и выдает ошибку компиляции тогда и только тогда, когда выражение ложно. Некоторые методы определения `STATIC_ASSERT()` основаны на деталях, относящихся к компилятору, и вот один из довольно переносимых способов его определения:

```
#define _ASSERT_GLUE(a, b)  a ## b
#define ASSERT_GLUE(a, b)   _ASSERT_GLUE(a, b)

#define STATIC_ASSERT(expr) \
    enum \
```

```
    { \
        ASSERT_GLUE(g_assert_fail_, __LINE__) \
            = 1 / (int)(!!(expr)) \
    }

STATIC_ASSERT(sizeof(int) == 4);   // проверка должна пройти удачно
STATIC_ASSERT(sizeof(float) == 1); // проверка не должна пройти
```

Это работает путем определения анонимного перечисления, содержащего один перечислитель. Имя перечислителя делается уникальным (в пределах модуля компиляции) склеиванием фиксированного префикса, такого как `g_assert_fail_`, с уникальным суффиксом — в данном случае номером строки, для которой вызывается макрос `STATIC_ASSERT()`. Значение перечислителя установлено в `1 / (!! (expr))`. Двойное отрицание `!!` гарантирует, что `expr` имеет логическое значение. Затем оно приводится к целому числу, получая либо 1, либо 0, в зависимости от того, является выражение истинным или ложным соответственно. Если выражение истинно, перечислителю будет присвоено значение `1/1`, то есть равное единице. Но если выражение ложно, мы будем просить компилятор установить для перечислителя значение `1/0`, что недопустимо и вызовет ошибку компиляции.

Когда макрос `STATIC_ASSERT()`, как определено ранее, не проходит проверку, Visual Studio 2015 выдает сообщение об ошибке во время компиляции, например:

```
1>test.cpp(48): error C2131: expression did not evaluate to
                             a constant
1>  test.cpp(48): note: failure was caused by an undefined
                        arithmetic operation
```

Вот еще один способ определить `STATIC_ASSERT()` с помощью специализации шаблона. В этом примере сначала проверим, используем мы C++11 или более позднюю версию. Если верно последнее, применим реализацию стандартной библиотеки `static_assert()` для максимальной переносимости. В противном случае вернемся к нашей пользовательской реализации:

```
#ifdef __cplusplus
    #if __cplusplus >= 201103L
        #define STATIC_ASSERT(expr) \
            static_assert(expr, \
                          "static assert failed:" \
                          #expr)
    #else
        // объявляем шаблон, но определяем только
        // случай истинности (через специализацию)
        template<bool> class TStaticAssert;
        template<> class TStaticAssert<true> {};

        #define STATIC_ASSERT(expr) \
            enum \
            { \
                ASSERT_GLUE(g_assert_fail_, __LINE__) \
                    = sizeof(TStaticAssert<!!(expr)>) \
```

```
            }
    #endif
#endif
```

Эта реализация, задействующая специализацию шаблонов, может быть предпочтительнее предыдущей, использующей деление на ноль, поскольку в Visual Studio 2015 появляется немного более полезное сообщение об ошибке:

```
1>test.cpp(48): error C2027: use of undefined type
                             'TStaticAssert<false>'
1>test.cpp(48): note: see declaration of
                        'TStaticAssert<false>'
```

Однако каждый компилятор обрабатывает отчеты об ошибках по-своему, поэтому у вас оно может быть иным. Чтобы ознакомиться с другими идеями реализации для утверждений во время компиляции, смотрите сайт www.pixelbeat.org/programming/gcc/static_assert.html.

3.3. Данные, код и схема памяти

3.3.1. Числовые представления

Цифры лежат в основе всего, что мы делаем при разработке игрового движка и программного обеспечения в целом. Каждый инженер-программист должен понимать, как числа представлены и хранятся на компьютере. В этом подразделе приведены основы, которые понадобятся вам в дальнейшем.

Основания систем счисления

Для людей естественна *десятичная система*, известная также как *десятичная запись*. В ней используются десять цифр от 0 до 9, каждая из которых справа налево представляет себя умноженную на следующую степень 10. Например, $7803 = (7 \cdot 10^3) + (8 \cdot 10^2) + (0 \cdot 10^1) + (3 \cdot 10^0) = 7000 + 800 + 0 + 3$.

В информатике математические величины, такие как целые и действительные числа, должны храниться в памяти компьютера. И, как мы знаем, компьютеры хранят числа в *двоичном* формате, то есть используя только две цифры, 0 и 1. Мы называем это *представлением с основанием 2*, потому что каждая цифра справа налево представляет собой себя, умноженную на следующую степень двойки. Информатики иногда используют префикс 0b для представления двоичных чисел. Например, двоичное число 0b1101 эквивалентно десятичному числу 13, поскольку $0b1101 = (1 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) = 8 + 4 + 0 + 1 = 13$.

Еще одно распространенное представление, популярное в компьютерных кругах, — это *шестнадцатеричные* числа, или числа *с основанием 16*. В этих обозначениях используются 10 цифр от 0 до 9 и шесть букв от A до F, которые заменяют десятичные значения от 10 до 15 соответственно. В языках программирования C и C++ для обозначения шестнадцатеричных чисел используется префикс 0x.

Эта запись популярна, потому что компьютеры обычно хранят данные группами по 8 бит, известными как *байты*, и поскольку одна шестнадцатеричная цифра представляет собой точно 4 бита, *пара* шестнадцатеричных цифр представляет байт. Например, значение 0xFF = 0b11111111 = 255 является наибольшим числом, которое может быть сохранено в 8 битах (1 байте). Каждая цифра в шестнадцатеричном числе, справа налево, представляет собой себя, умноженную на 16 в степени, равной количеству стоящих справа цифр. Например, 0xB052 = $(11 \cdot 16^3) + (0 \cdot 16^2) + (5 \cdot 16^1) + (2 \cdot 16^0) = (11 \cdot 4096) + (0 \cdot 256) + (5 \cdot 16) + (2 \cdot 1) = 45\,138$.

Знаковые и беззнаковые целые числа

В информатике мы используем целые числа как со знаком, так и без знака. Конечно, термин «беззнаковое целое число» на самом деле немного неправильный — в математике *целые положительные числа* или *натуральные числа* варьируются от 0 (или 1) до положительной бесконечности, в то время как собственно *целые числа* (integers) в широком смысле варьируются от $+\infty$ до $-\infty$. Тем не менее в этой книге мы будем использовать язык компьютерных наук и придерживаться терминов «знаковое целое число» и «беззнаковое целое число».

Большинство современных персональных компьютеров и игровых приставок легче всего работают с целыми числами длиной 32 или 64 бита (хотя 8- и 16-битные целые числа также широко используются в программировании игр). Чтобы представить 32-разрядное беззнаковое целое число, мы просто кодируем значение в двоичной записи (см. ранее). Диапазон возможных значений для 32-битного беззнакового целого числа составляет от 0x00000000 (0) до 0xFFFFFFFF (4 294 967 295).

Чтобы представить *знаковое* целое число в тех же 32 битах, нам нужен способ различения положительных и отрицательных значений. Один простой подход, называемый *кодированием знака и величины* или *прямым кодированием*, резервирует старший бит (первый) в качестве *бита знака*. Когда этот бит равен нулю, значение числа является положительным, а когда единице — отрицательным. Для записи значения числа остается 31 бит, что сокращает диапазон абсолютных величин вдвое, но при этом добавляется столько же отрицательных значений для каждой величины, включая ноль.

Большинство микропроцессоров используют немного более эффективную технику для кодирования отрицательных целых чисел, называемую *дополнительным кодом (кодированием)*. Эта запись имеет только одно представление для нулевого значения, в отличие от двух представлений, доступных с простым знаковым битом при прямом кодировании (положительный ноль и отрицательный ноль). В дополнительном кодировании число −1 кодируется одними единицами и в 32-битной двоичной записи имеет значение 0xFFFFFFFF, а следующие отрицательные числа отсчитываются от него. Любое значение с установленным в 1 старшим битом считается отрицательным. Таким образом, значения от 0x00000000 (0) до 0x7FFFFFFF (2 147 483 647) представляют собой положительные целые числа, а значения от 0x80000000 (−2 147 483 648) до 0xFFFFFFFF (−1) — отрицательные целые числа.

Запись числа с фиксированной точкой

Тип `integer` отлично подходит для представления целых чисел, но для представления дробных и иррациональных чисел нужен другой формат, выражающий концепцию десятичной точки.

Одним из ранних подходов, использованных учеными-информатиками, были обозначения с фиксированной точкой. В этой записи произвольным образом выбирается количество битов для представления целой части числа, а остальные биты представляют дробную часть. При перемещении слева направо (то есть от старшего значащего бита к младшему) биты целого значения представляют убывающие степени двойки (...16, 8, 4, 2, 1), а дробные биты — убывающие *обратные* степени двойки (1/2, 1/4, 1/8, 1/16...). Например, запишем число –173,25 в 32-битном формате записи с фиксированной точкой с одним знаковым битом, 16 битами для целой и 15 битами для дробной части. Сначала мы преобразуем знак, целую и дробную части в их двоичные эквиваленты по отдельности (минус = 0b1, 173 = 0b0000000010101101 и 0,25 = 1/4 = 0b010000000000000). Затем упакуем эти значения в одно 32-разрядное целое число. Окончательный результат — 0x8056A000. Сказанное иллюстрирует рис. 3.4.

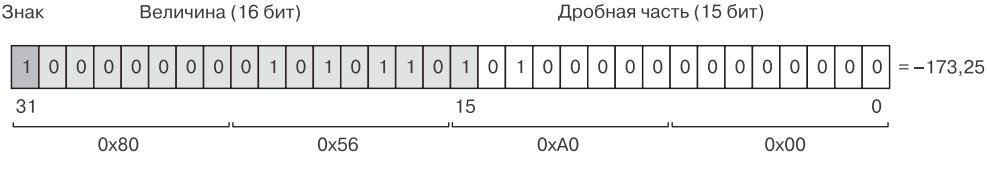


Рис. 3.4. Представление числа с фиксированной точкой с 16-битной целой и 15-битной дробной частью

Недостаток представления дробных чисел методом записи числа с фиксированной точкой заключается в ограничении как диапазона величин, которые могут быть представлены, так и степени точности дробной части. Рассмотрим 32-битное значение с фиксированной точкой с 16 битами для целой части, 15 битами для дробной и со знаковым битом. Этот формат может представлять только величины до ±65 535, что не особенно много. Чтобы преодолеть это ограничение, используется представление с *плавающей точкой*.

Запись с плавающей точкой

В формате записи числа с плавающей точкой положение десятичного знака произвольно и задается с помощью показателя степени. Число с плавающей точкой разбито на три части: мантиссу, которая содержит соответствующие цифры числа по обе стороны от десятичной точки, экспоненту (порядок или показатель степени), определяющую, где в этой строке цифр находится десятичная точка, и бит знака, который указывает, является значение положительным или отрицательным. Есть разные способы расположить эти три компонента в памяти, но наиболее распространенным стандартом является IEEE-754. Он определяет, что число

в 32-разрядном формате записи выглядит следующим образом: старший бит — знак числа, за ним восемь знаков, определяющих значение экспоненты (порядка), а оставшиеся 23 бита отводятся для записи мантиссы.

Значение v, представленное знаковым битом s, показателем степени e и мантиссой m, равно $v = s \cdot 2^{(e-127)} \cdot (1 + m)$.

Знаковый бит s имеет значение +1 или −1. Значение показателя e смещено на 127, чтобы можно было легко представить как положительные, так и отрицательные значения, не вводя дополнительный знак для показателя. Мантисса начинается с неявной 1, которая фактически не хранится в памяти, а остальные биты интерпретируются как обратные степени двойки. Таким образом, записанное значение равно $1 + m$, где m — дробное значение, хранящееся в мантиссе.

Например, битовая комбинация, показанная на рис. 3.5, представляет число 0,156 25, потому что $s = 0$ (указывает положительное число), e = 0b01111100 = 124 и m = 0b0100... = $0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1/4$. Таким образом:

$$v = s \cdot 2^{(e-127)} \cdot (1 + m) = (+1) \cdot 2^{(124-127)} \cdot (1 + 1/4) = 2^{-3} \cdot 5/4 = 1/8 \cdot 5/4 =$$
$$= 0{,}125 \cdot 1{,}25 = 0{,}156\ 25.$$

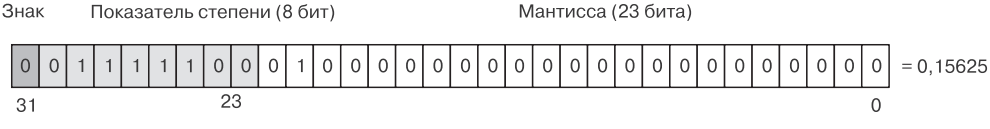


Рис. 3.5. Тридцатидвухбитный формат с плавающей точкой согласно IEEE-754

Компромисс между величиной и точностью. *Абсолютная точность* числа с плавающей точкой увеличивается с уменьшением *величины* и наоборот. Это связано с тем, что в мантиссе имеется фиксированное количество битов и они должны быть каким-то образом разделены между целой и дробной частями числа. Чем больший процент битов используется для представления величины, тем меньшая их часть доступна для обеспечения точности дробной части. Для описания физики этой концепции обычно применяется термин *«значащие цифры»* (en.wikipedia.org/wiki/Significant_digits).

Чтобы понять компромисс между величиной и точностью, посмотрим на максимально возможное число с плавающей точкой `FLT_MAX` ≈ $3{,}403 \cdot 10^{38}$, чье представление в 32-битном формате IEEE с плавающей запятой будет записано как 0x7F7FFFFF. Давайте разберемся, как оно получается.

- Наибольшее абсолютное значение, которое мы можем представить с помощью 23-битной мантиссы, — это 0x00FFFFFF в шестнадцатеричном формате, или 24 последовательные единицы в двоичном представлении — 23 единицы в мантиссе плюс неявная ведущая единица.
- Показатель 255 имеет особый смысл в стандарте IEEE-754 — он используется для таких значений, как «не число» (NaN — not-a-number) и бесконечность,

поэтому его нельзя применять для обычных чисел. Так что для записи максимального восьмибитного показателя используется число 254, в котором после вычитания неявного смещения 127 получается 127.

Таким образом, значение `FLT_MAX` составляет $0x00FFFFFF \cdot 2^{127} =$ $= xFFFFFF0000000000000000000000000000$. Другими словами, 24 двоичные единицы были сдвинуты на 127 битовых позиций, оставляя 127 – 23 = 104 двоичных нуля (или 104 / 4 = 26 шестнадцатеричных нулей) после наименьшей значащей цифры мантиссы. Эти конечные нули не соответствуют никаким действительным битам в 32-битном значении с плавающей точкой — они просто появляются из воздуха из-за показателя степени. Если бы мы вычли из `FLT_MAX` небольшое число (где «небольшое» означает любое число, состоящее из менее чем 26 шестнадцатеричных цифр), результатом все равно было бы `FLT_MAX`, потому что этих 26 наименьших значимых шестнадцатеричных цифр на самом деле не существует!

Противоположный эффект наблюдается для значений с плавающей точкой, намного меньших чем единица. В этом случае показатель степени велик, но отрицателен, а значащие цифры смещены в противоположном направлении. Мы обмениваем способность представлять большие величины на высокую точность. Таким образом, мы всегда имеем одинаковое количество *значащих цифр* (на самом деле *значащих битов*) в числах с плавающей запятой, и показатель степени может использоваться для сдвига этих значащих битов в более высокие или более низкие диапазоны величины.

Субнормальные значения. Еще одна тонкость, на которую следует обратить внимание, заключается в том, что существует конечный разрыв между нулем и наименьшим ненулевым значением, которое мы можем представить с помощью записи с плавающей точкой (как описывалось до сих пор). Наименьшая ненулевая положительная величина, которую мы можем представить, составляет `FLT_MIN` $= 2^{-126} \approx 1{,}175 \cdot 10^{-38}$ и имеет двоичное представление 0x00800000 (то есть экспонента равна 0x01, или –126 после вычитания смещения, а мантисса — все нули, кроме неявной ведущей единицы). Следующее наименьшее допустимое значение равно нулю, поэтому между положительным и отрицательным значениями `-FLT_MIN` и `+ FLT_MIN` существует конечный разрыв. Этим подчеркивается то, что действительное число *квантуется* при использовании представления с плавающей точкой. (Обратите внимание на то, что стандартная библиотека C++ представляет `FLT_MIN` более многословно — `std::numeric_limits<float>::min()`. В книге для краткости будем применять обозначение `FLT_MIN`.)

Разрыв вокруг 0 можно заполнить, используя расширение представления с плавающей точкой, известное как *денормализованные* (или *субнормальные*) *значения*. С этим расширением любое значение с плавающей точкой со значением показателя, равным 0 (–127 после смещения), интерпретируется как субнормальное число. Показатель степени обрабатывается так, как если бы он был равен 1, а не 0, а неявная ведущая 1, которая обычно находится перед битами мантиссы, заменяется на 0. Это приводит к заполнению разрыва между `-FLT_MIN` и `+ FLT_MIN` линейной последовательностью равномерно распределенных субнормальных значений.

Положительное субнормальное значение с плавающей точкой, которое ближе всего к нулю, представлено константой `FLT_TRUE_MIN`.

Преимущество использования субнормальных значений заключается в том, что они обеспечивают бóльшую точность вблизи 0. Например, это гарантирует, что следующие два выражения эквивалентны даже для значений `a` и `b`, очень близких к `FLT_MIN`:

```
if (a == b) { ... }
if (a - b == 0.0f) { ... }
```

Без субнормальных значений выражение `a – b` может быть приравнено к нулю, даже если `a != b`.

Машинный эпсилон. *Машинный эпсилон* для конкретного представления с плавающей точкой определяется как наименьшее значение ε, которое удовлетворяет уравнению $1 + \varepsilon \neq 1$. Для числа с плавающей точкой IEEE-754 с 23-битной точностью значение ε равно 2^{-23}, что составляет приблизительно $1{,}192 \cdot 10^{-7}$. Наиболее значимая цифра ε *попадает прямо* в конец диапазона значащих цифр в значении 1,0, так что добавление любого значения, меньшего чем ε, к 1,0 не дает никакого эффекта. Другими словами, любые новые биты, добавляющие значение меньше ε, будут обрезаны, когда мы попытаемся поместить сумму в мантиссу, состоящую всего из 23 бит.

Единица на последнем месте (units in the last place, ULP). Рассмотрим два числа с плавающей точкой, идентичных во всех отношениях, за исключением значения младшего разряда в их мантиссах. Говорят, что эти два значения различаются *на одну единицу на последнем месте* (1 ULP). Фактическое значение 1 ULP изменяется в зависимости от показателя степени. Например, число в записи с плавающей запятой `1.0f` имеет несмещенный показатель степени 0 и мантиссу, в которой все биты равны 0, за исключением неявной начальной 1. При этом показателе степени 1 ULP равна машинному эпсилону (2^{-23}). Если мы увеличим показатель степени на 1, получив значение `2,0f`, значение 1 ULP станет равным двум значениям машинного эпсилона. И если показатель степени равен 2, что дает значение `4,0f`, значение 1 ULP равно четырем значениям машинного эпсилона. В общем, если несмещенный показатель значения с плавающей точкой равен x, то $1\ \text{ULP} = 2^x \cdot \varepsilon$.

Концепция единиц в последнем разряде иллюстрирует то, что точность числа с плавающей запятой зависит от его степени и полезна для количественной оценки погрешности, присущей любому вычислению с плавающей точкой. Это также может быть полезно для нахождения значения с плавающей точкой, которое является следующим наибольшим представимым значением относительно известного значения или, наоборот, следующего наименьшего представимого значения относительно данного значения. Что, в свою очередь, может быть полезно для преобразования сравнения «больше или равно» в сравнение «больше». Математически условие $a \geq b$ эквивалентно условию $a + 1\ \text{ULP} > b$. Мы задействуем этот маленький трюк в движке Naughty Dog, чтобы упростить логику в системе диалогов героев. В ней простые сравнения могут быть использованы при выборе для персонажей различных строк диалога, которые они произнесут. Вместо того чтобы поддерживать все

возможные операторы сравнения, мы поддерживаем только проверки «больше» и «меньше» и обрабатываем «больше или равно» или «меньше или равно», добавляя 1 ULP к сравниваемому значению или вычитая 1 ULP из него.

Влияние точности чисел с плавающей точкой на программное обеспечение. Концепции ограниченной точности и машинного эпсилона действительно влияют на игровое программное обеспечение. Допустим, мы используем переменную с плавающей точкой для отслеживания абсолютного игрового времени в секундах. Как долго мы сможем играть в игру, прежде чем переменная времени станет настолько большой, что добавление к ней 1/30 секунды больше не изменит ее значения? Ответ: 12,14 дня, или 2^{20} секунды. Это дольше, чем будут играть в большинство игр, поэтому мы, вероятно, можем избежать использования 32-битной переменной для таймера, измеряемого в секундах в игре. Очевидно, что важно понимать ограничения формата с плавающей точкой, чтобы можно было предсказать потенциальные проблемы и предпринять шаги, которые помогут при необходимости избежать их.

«Битовые» хитрости со значениями с плавающей точкой IEEE. В [9, раздел 2.1] приведено несколько полезных «битовых» трюков с переменными с плавающей точкой IEEE, которые помогут сделать некоторые вычисления с плавающей точкой молниеносными.

3.3.2. Примитивные типы данных

В C и C++ используют несколько основных типов данных. Стандарты C и C++ предоставляют рекомендации относительно размеров и знаков этих типов данных, но каждый компилятор может определять типы немного по-разному, чтобы обеспечить максимальную производительность на целевом оборудовании.

- *Символьный* (`char`). Значение `char` (символ) обычно составляет 8 бит и почти всегда достаточно велико для того, чтобы содержать символы ASCII или UTF-8 (см. главу 6). Некоторые компиляторы определяют, что `char` должен иметь знак, в то время как другие используют символы без знака по умолчанию.
- *Целочисленный* (`int`, `short`, `long`). Предполагается, что `int` содержит целочисленное значение со знаком, которое по размеру является наиболее эффективным для целевой платформы. Оно обычно 32-битное для 32-битной архитектуры CPU, такой как Pentium 4 или Xeon, и 64-битное в 64-битной архитектуре, такой как Intel Core i7, хотя размер `int` зависит и от других факторов, таких как параметры компилятора и целевая операционная система. Обычно `short` меньше, чем `int`, и на многих машинах составляет 16 бит. Тип `long` равен `int` или больше него и может иметь размер 32 или 64 бита или даже больше, опять же в зависимости от архитектуры процессора, параметров компилятора и целевой ОС.
- *С плавающей запятой* (`float`, `double`). В большинстве современных компиляторов `float` — это 32-разрядное значение IEEE-754 с плавающей точкой. `Double` — это значение с плавающей точкой IEEE-754 с двойной точностью, то есть 64-разрядное.

- *Логический* (`bool`). `Bool` является значением «истина/ложь». Размер `bool` широко варьируется в зависимости от различных компиляторов и аппаратных архитектур. Он никогда не реализуется как один бит, но некоторые компиляторы определяют его как 8 бит, а другие используют полные 32 бита.

Переносимые типы данных. Встроенные основные типы данных в C и C ++ разработаны, чтобы быть переносимыми и, следовательно, неспецифичными. Однако во многих начинаниях по разработке программного обеспечения, включая программирование игрового движка, важно точно знать, сколько памяти занимает конкретная переменная.

До появления C++11 программистам приходилось полагаться на непереносимые типы размеров, предоставляемые компилятором. Например, в компиляторе Visual Studio C/C++ были определены следующие расширенные ключевые слова для объявления переменных с явным количеством битов: `__int8`, `__int16`, `__int32` и `__int64`. Большинство других компиляторов имеют свои собственные размерные типы данных с похожей семантикой, но немного иным синтаксисом.

Из-за этих различий между компиляторами большинство игровых движков достигали переносимости исходного кода, определяя собственные типы нестандартных размеров. Например, в Naughty Dog мы используем следующие размеры:

- `F32` — это 32-разрядное значение IEEE-754 с плавающей точкой;
- `U8`, `I8`, `U16`, `I16`, `U32`, `I32`, `U64` и `I64` — беззнаковые (U) и со знаком (I) 8-, 16-, 32- и 64-разрядные целые числа соответственно;
- `U32F` и `I32F` — «быстрые» беззнаковые и знаковые с 32-битными значениями соответственно. Каждый из этих типов данных содержит значение не менее 32 бит, но оно может быть больше, если это приведет к ускорению работы программы на целевом ЦП.

<cstdint>. Стандартная библиотека C++11 представляет набор целочисленных типов стандартизированного размера. Они объявлены в заголовочном файле `<cstdint>` и включают в себя знаковые типы `std::int8_t`, `std::int16_t`, `std::int32_t` и `std::int64_t` и беззнаковые типы `std::uint8_t`, `std::uint16_t`, `std::uint32_t` и `std::uint64_t` наряду с «быстрыми» вариантами (такими как типы `I32F` и `U32F`, которые мы определили в Naughty Dog). Эти типы освобождают программиста от необходимости оборачивать специфичные для компилятора типы для достижения переносимости. Полный список размеров этих типов можно найти по адресу en.cppreference.com/w/cpp/types/integer.

Примитивные типы данных OGRE. OGRE определяет несколько собственных размерных типов. `Ogre::uint8`, `Ogre::uint16` и `Ogre::uint32` являются основными целочисленными типами без знака.

`Ogre::Real` определяет действительное значение с плавающей точкой. Обычно он имеет размер 32 бита (эквивалентно числу с плавающей точкой), но его можно переопределить глобально, чтобы он имел размер 64 бита (как `double`), определив макрос препроцессора `OGRE_DOUBLE_PRECISION` в значение 1. Эта способность из-

менять значение `Ogre::Real` обычно используется только в том случае, если в игре есть особые требования к точности математических вычислений, что редко, но встречается. Графические карты (GPU) всегда выполняют математические вычисления с 32- или 16-разрядными числами с плавающей точкой, CPU/FPU также обычно быстрее, когда работают с одинарной точностью, а векторные инструкции SIMD имеют дело с 128-разрядными регистрами, которые содержат по четыре 32-разрядные переменные с плавающей точкой. Следовательно, большинство игр применяют вычисления с плавающей точкой с одинарной точностью.

Типы данных `Ogre::uchar`, `Ogre::ushort`, `Ogre::uint` и `Ogre::ulong` — это просто сокращенные обозначения для `unsigned char`, `unsigned short` и `unsigned long` соответственно из C/C++. Как таковые они не более или менее полезны, чем собственные аналоги из C/C++.

Типы `Ogre::Radian` и `Ogre::Degree` особенно интересны. Они являются обертками вокруг простого значения `Ogre::Real`. Основная роль этих типов заключается в том, чтобы разрешить документирование единиц угла жестко закодированных литеральных констант и обеспечить автоматическое преобразование между двумя системами единиц. Кроме того, тип `Ogre::Angle` представляет угол в действующих единицах измерения по умолчанию. Программист может определить, какие значения, в радианах или градусах, будут использованы по умолчанию при первом запуске приложения OGRE.

Что удивительно, OGRE не предоставляет ряд стандартных типов данных, обычно используемых в других игровых движках. Например, он не определяет 8-, 16- или 64-битные целочисленные типы со знаком. Если вы пишете игровой движок поверх OGRE, возможно, в какой-то момент придется определить эти типы вручную.

Многобайтовые значения и порядковый номер

Значения, размер которых превышает 8 бит (1 байт), называются *многобайтовыми величинами*. Они обычны для любого программного проекта, в котором используются целые числа и значения с плавающей точкой размером 16 бит или больше. Например, целочисленное значение 4660 = 0x1234 представлено двумя байтами, 0x12 и 0x34. Мы называем 0x12 старшим байтом, а 0x34 — младшим. В 32-битном значении, таком как 0xABCD1234, старший байт равен 0xAB, а младший — 0x34. Те же понятия применимы и к 64-разрядным целым числам, а также к 32- и 64-разрядным значениям с плавающей точкой.

Многобайтовые целые числа можно сохранить в памяти одним из двух способов, и микропроцессоры различаются по выбору способа хранения (рис. 3.6).

- *Little-endian* (от младшего к старшему). Если микропроцессор хранит младший значащий байт многобайтового значения по более низкому адресу памяти, чем самый старший байт, мы говорим, что процессор имеет *обратный порядок байтов*. На машине с обратным порядком байтов число 0xABCD1234 будет храниться в памяти, используя последовательные байты 0x34, 0x12, 0xCD, 0xAB.

- *Big-endian* (от старшего к младшему). Если микропроцессор хранит старший значащий байт многобайтового значения по более низкому адресу памяти, чем самый младший байт, мы говорим, что это процессор *с прямым порядком байтов*. На машине с прямым порядком байтов число 0xABCD1234 будет храниться в памяти, используя последовательные байты 0xAB, 0xCD, 0x12, 0x34.

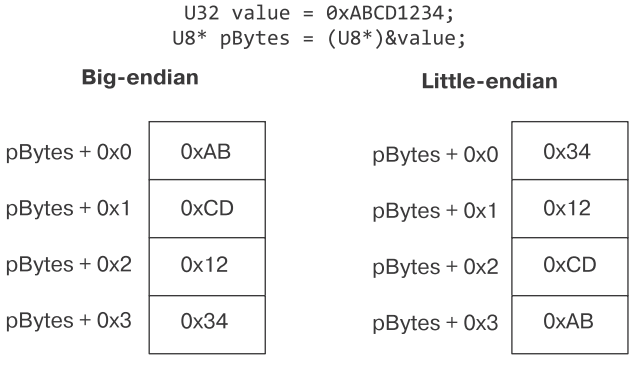


Рис. 3.6. Представления с прямым и обратным порядком байтов значения 0xABCD1234

Большинству программистов не нужно сильно беспокоиться о порядке байтов. Тем не менее, когда вы работаете над игрой, всегда стоит помнить о данной проблеме. Это связано с тем, что игры обычно *разрабатываются* на компьютере под управлением Windows или Linux, на котором установлен процессор Intel Pentium с прямым порядком байтов, но работают на консоли Wii, Xbox 360 или PlayStation 3 — во всех трех используется вариант процессора PowerPC, который может быть сконфигурирован для любого порядка байтов, но по умолчанию — обратного. Теперь представьте, что происходит, когда вы генерируете файл данных для использования игровым движком на процессоре Intel, а затем пытаетесь загрузить его в движок, работающий на процессоре PowerPC. Любое многобайтовое значение, которое вы записали в этот файл данных, будет сохранено в формате с прямым порядком байтов. Но когда игровой движок читает файл, он ожидает, что все его данные будут в формате с обратным порядком байтов. Что получится в итоге? Вы напишете 0xABCD1234, но считаете 0x3412CDAB, и это явно не то, чего вы хотели!

Есть как минимум два решения этой проблемы.

1. Вы можете записать все файлы данных в виде текста и сохранить все многобайтовые числа в виде последовательности десятичных или шестнадцатеричных цифр, один символ (один байт) на цифру. Это работает, но крайне неэффективно с точки зрения использования дискового пространства.
2. Вы можете использовать свои инструменты изменения порядка байтов данных перед записью их в двоичный файл данных. По сути, это гарантирует, что файл

данных использует порядок целевого микропроцессора (игровой консоли), даже если инструменты работают на машине, которая применяет противоположный порядок.

Целочисленный метод изменения порядка. Перестановка целых чисел по порядку не слишком сложна. Вы просто начинаете с самого старшего байта значения и меняете его местами с младшим байтом. И продолжаете этот процесс, пока не достигнете середины значения. Например, 0xA7891023 станет 0x231089A7.

Единственная сложная часть — это определить, *какие байты* поменять местами. Допустим, вы записываете содержимое структуры C или класса C++ из памяти в файл. Чтобы правильно заменить эти данные, вам нужно отслеживать расположение и размеры каждого элемента данных в структуре и менять местами каждый из них соответственно его размеру. Например, структура:

```
struct Example
{
    U32  m_a;
    U16  m_b;
    U32  m_c;
};
```

может быть записана в файл данных следующим образом:

```
void writeExampleStruct(Example& ex, Stream& stream)
{
    stream.writeU32(swapU32(ex.m_a));
    stream.writeU16(swapU16(ex.m_b));
    stream.writeU32(swapU32(ex.m_c));
}
```

И функции изменения порядка могли быть определены вот так:

```
inline U16 swapU16(U16 value)
{
    return ((value & 0x00FF) << 8)
        | ((value & 0xFF00) >> 8);
}

inline U32 swapU32(U32 value)
{
    return ((value & 0x000000FF) << 24)
        | ((value & 0x0000FF00) << 8)
        | ((value & 0x00FF0000) >> 8)
        | ((value & 0xFF000000) >> 24);
}
```

Вы не можете просто выполнить приведение объекта `Example` к массиву байтов и слепо поменять местами байты, используя одну общую функцию. Нужно знать, байты *каких элементов данных* следует поменять местами и *каков их размер*, и каждый элемент данных должен изменять порядок индивидуально.

Некоторые компиляторы предоставляют встроенные макросы с обратным порядком байтов, освобождая вас от необходимости писать собственные. Например, gcc предлагает семейство макросов `__builtin_bswapXX()` для выполнения 16-, 32- и 64-битных порядковых перестановок. Однако такие специфичные для компилятора средства, конечно, неуниверсальны.

Изменение порядка для значений с плавающей точкой. Как вы уже видели, значение с плавающей точкой IEEE-754 имеет детализированную внутреннюю структуру, включающую в себя несколько битов для мантиссы, несколько битов для показателя степени и знаковый бит. Однако вы можете поменять их местами, как если бы они были целыми числами, потому что байты — это байты. Просто можете рассматривать битовый паттерн значения `float` как будто `std::int32_t`, выполнить операцию перестановки с обратным порядком байтов, а затем заново интерпретировать результат как число с плавающей точкой.

Вы можете переинтерпретировать числа с плавающей точкой как целые числа, используя оператор `reinterpret_cast` в C++ для указателя на `float`, а затем разыменовать указатель приведения типа. Данная процедура известна как *каламбур типизации*. Но каламбур способен привести к ошибкам оптимизации, если включены строгие псевдонимы. (Эта проблема превосходно описана по адресу www.cocoawithlove.com/2008/04/using-pointers-to-recast-in-c-is-bad.html.) Есть одна альтернатива, которая гарантирует, что игра будет работать на разных платформах, — использовать объединение, как показано далее:

```
union U32F32
{
    U32   m_asU32;
    F32   m_asF32;
};

inline F32 swapF32(F32 value)
{
    U32F32 u;
    u.m_asF32 = value;

    // Изменение порядка для значений с плавающей точкой
    u.m_asU32 = swapU32(u.m_asU32);

    return u.m_asF32;
}
```

3.3.3. Килобайт или кибибайт

Вы, вероятно, применяли метрические единицы (СИ), такие как килобайты (Кбайт) и мегабайты (Мбайт), чтобы описать объемы памяти. Но использовать эти единицы для описания количества памяти, которое измеряется в степенях двойки, не совсем правильно. Когда программист говорит о килобайтах, это обычно означает 1024 байта. Но единицы СИ определяют префикс «кило» для обозначения 10^3, или 1000, а не 1024.

Чтобы устранить эту неоднозначность, Международная электротехническая комиссия (МЭК) в 1998 году установила новый набор префиксов, подобных определяемым СИ, для использования в компьютерной сфере. Эти префиксы определяются в виде степеней двойки, а не степеней 10, так что компьютерные инженеры могут точно и удобно оперировать величинами, которые являются степенями двойки. В новой системе вместо килобайта (1000 байт) мы употребляем кибибайт (1024 байта (КиБ)), а вместо мегабайта (1 000 000 байт) — мебибайт ($1024 \cdot 1024 = 1\,048\,576$ байт (МиБ)). В табл. 3.1 приведены размеры, префиксы и названия наиболее часто используемых единиц измерения байтов в СИ и МЭК. В этой книге будем применять единицы измерения МЭК.

Таблица 3.1. Сравнение метрических (СИ) единиц и единиц МЭК для описания количества байтов

СИ			**МЭК**		
Значение	**Единица**	**Название**	**Значение**	**Единица**	**Название**
1000	Кбайт	Килобайт	1024	КиБ	Кибибайт
1000^2	Мбайт	Мегабайт	1024^2	МиБ	Мебибайт
1000^3	Гбайт	Гигабайт	1024^3	ГиБ	Гибибайт
1000^4	Тбайт	Терабайт	1024^4	ТиБ	Тебибайт
1000^5	Пбайт	Петабайт	1024^5	ПиБ	Пебибайт
1000^6	Эбайт	Эксабайт	1024^6	ЭиБ	Эксбибайт
1000^7	Збайт	Зеттабайт	1024^7	ЗиБ	Зебибайт
1000^8	Ибайт	Йоттабайт	1024^8	ЙиБ	Йобибайт

3.3.4. Объявления, определения и компоновка

Единицы компиляции снова

Как мы видели в главе 2, программа на C или C++ состоит из *единиц компиляции*. Компилятор берет один файл `.cpp` за раз и генерирует для него выходной файл, называемый объектным файлом (`.o` или `.obj`). Файл `.cpp` — это наименьшая единица, используемая компилятором, отсюда и название «единица компиляции». Объектный файл содержит не только скомпилированный машинный код для всех функций, определенных в файле `.cpp`, но и все его глобальные и статические переменные. Кроме того, объектный файл может содержать *неразрешенные ссылки* на функции и глобальные переменные, определенные в *других* файлах `.cpp`.

Компилятор работает только с одним модулем компиляции единовременно, поэтому всякий раз, когда он встречает ссылку на внешнюю глобальную переменную или функцию, он должен верить и предполагать, что рассматриваемая сущность действительно есть (рис. 3.7). Работа компоновщика состоит в том, чтобы объединить все объектные файлы в конечный исполняемый образ. При этом компоновщик считывает все объектные файлы и пытается разрешить все неразрешенные перекрестные ссылки между ними. Если процесс проходит успешно, генерируется исполняемый образ, содержащий все функции, глобальные переменные

и статические переменные, причем все перекрестные ссылки в единицах компиляции разрешаются должным образом (рис. 3.8).

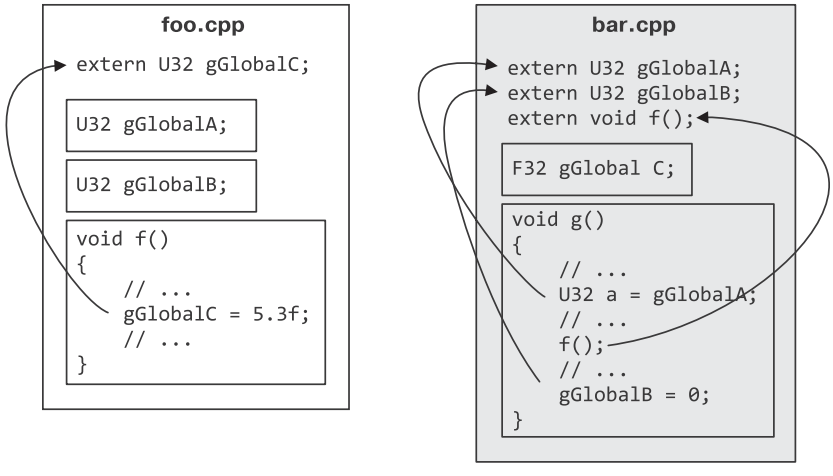


Рис. 3.7. Неразрешенные внешние ссылки в двух единицах компиляции

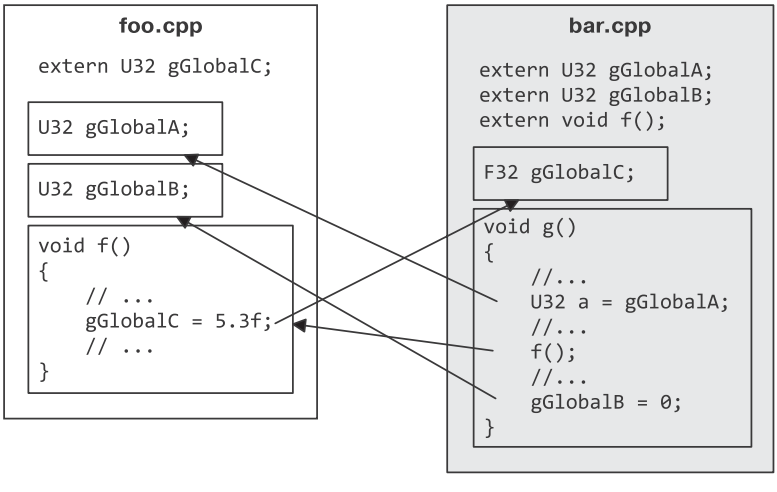


Рис. 3.8. Полностью разрешенные внешние ссылки после успешной компоновки

Основная задача компоновщика состоит в том, чтобы разрешить внешние ссылки, и при этом он может генерировать только два вида ошибок.

1. Цель внешней (`extern`) ссылки может не быть найдена, в этом случае компоновщик генерирует ошибку «неразрешенный символ».
2. Компоновщик смог найти более одной переменной или функции с одним и тем же именем, в этом случае он генерирует ошибку «множественно определенный символ».

Эти две ситуации показаны на рис. 3.9.

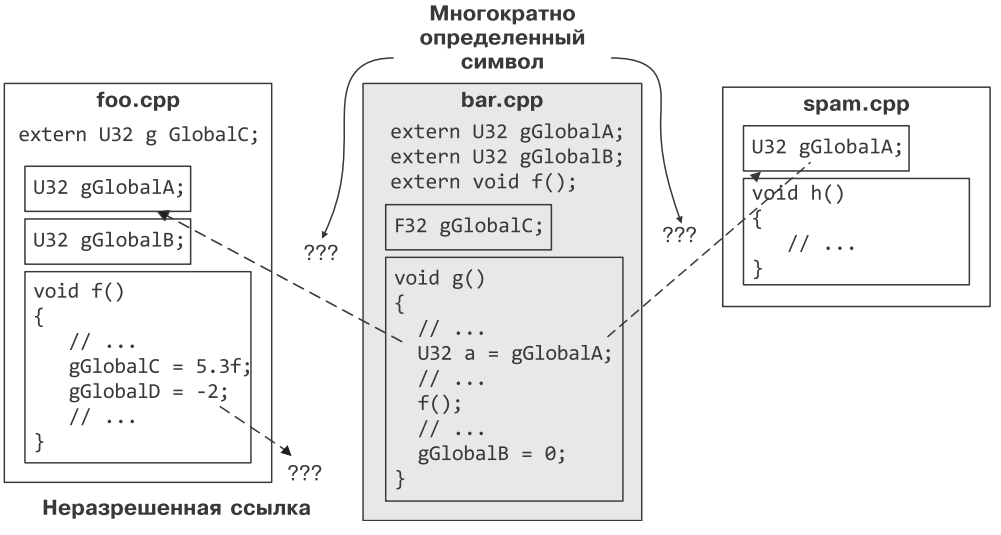


Рис. 3.9. Две наиболее распространенные ошибки компоновщика

Объявление и определения

В языках C и C++ переменные и функции должны быть *объявлены* и *определены* до того, как их можно будет использовать. Важно понимать разницу между объявлением и определением в C и C++.

- *Объявление* — это описание объекта данных или функции. Оно предоставляет компилятору *имя* объекта и его *тип данных* или *сигнатуру функции*, то есть тип возвращаемого значения и тип (-ы) аргумента (-ов).
- *Определение*, напротив, описывает уникальную область памяти в программе. Эта память может содержать переменную, экземпляр структуры или класса либо машинный код функции.

Другими словами, объявление — это *ссылка на сущность*, а определение — *сама сущность*. Определение всегда является объявлением, но не всегда верно обратное — можно написать чистую декларацию на C и C++, которая не является определением.

Функции определяются написанием тела функции, заключенного в фигурные скобки, сразу после сигнатуры:

```
foo.cpp
// определение функции max()
int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

```
// определение функции min()
int min(int a, int b)
{
    return (a <= b) ? a : b;
}
```

Чистое объявление может быть предоставлено для функции, чтобы ее можно было использовать в других единицах компиляции (или позже в той же). Это делается написанием сигнатуры функции, за которой следует точка с запятой с необязательным префиксом `extern`:

foo.h
```
extern int max(int a, int b); // объявление функции

int min(int a, int b); // такое же объявление (extern –
                       // необязательный/подразумевается)
```

Переменные и экземпляры классов и структур определяются записью типа данных, за которым стоят имя переменной или экземпляра и необязательный спецификатор массива в квадратных скобках:

foo.cpp
```
// Все это определения переменных:
U32 gGlobalInteger = 5;
F32 gGlobalFloatArray[16];
MyClass gGlobalInstance;
```

Глобальная переменная, определенная в одной единице компиляции, может быть дополнительно объявлена для использования в других единицах компиляции с помощью ключевого слова `extern`:

foo.h
```
// Это все чистые объявления:
extern U32 gGlobalInteger;
extern F32 gGlobalFloatArray[16];
extern MyClass gGlobalInstance;
```

Множественность объявлений и определений. Неудивительно, что любые объект данных или функция в программе на C/C++ могут иметь несколько идентичных *объявлений*, но только одно *определение*. Если два или более идентичных определения существуют в одном модуле компиляции, компилятор заметит, что несколько объектов носят одно и то же имя, и выдаст ошибку. Если два или более идентичных определения существуют в *разных* единицах компиляции, компилятор не сможет определить проблему, потому что он одновременно работает только с одной единицей компиляции. Но в этом случае компоновщик выдаст ошибку «множественно определенный символ» при попытке разрешить перекрестные ссылки.

Определения в заголовочных файлах и встраивание. Обычно опасно помещать *определения* в заголовочные файлы. Причина этого довольно очевидна: если заголовочный файл, содержащий *определение*, включен (`#included`) в более чем один

файл .cpp, это верный способ сгенерировать ошибку компоновщика с многократно определенным символом.

Определения встроенных функций — исключение из этого правила, поскольку каждый вызов встроенной функции приводит к появлению новой копии машинного кода этой функции, встроенной непосредственно в вызывающую функцию. Фактически определения встроенных функций *следует* помещать в заголовочные файлы, если они должны использоваться более чем в одной единице компиляции. Обратите внимание на то, что *недостаточно* пометить объявление функции ключевым словом inline в файле .h, а затем поместить тело этой функции в файл .cpp. Компилятор должен иметь возможность видеть тело функции, чтобы встроить его, например:

```
foo.h
// Это определение функции будет вставлено правильно
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}

// Это объявление не может быть встроено, потому что
// компилятор не может видеть тело функции
inline int min(int a, int b);

foo.cpp
// Тело функции min() эффективно скрыто от
// компилятора, так как она встроена ТОЛЬКО в foo.cpp
int min(int a, int b)
{
    return (a <= b) ? a : b;
}
```

Ключевое слово inline — это просто подсказка компилятору. Он анализирует затраты/выгоды для каждой встроенной функции, взвешивая размер ее кода в сравнении с потенциальными преимуществами производительности при ее вставке и принимает окончательное решение относительно того, будет функция встроенной или нет. Некоторые компиляторы предоставляют синтаксис, такой как __forceinline, что позволяет программисту напрямую обходить функции анализа затрат/выгод и контролировать встраиваемые функции.

Шаблоны и заголовочные файлы. Определение шаблонного класса или функции должно быть видимым для компилятора во всех единицах компиляции, в которых оно используется. Таким образом, если вы хотите, чтобы шаблон мог применяться более чем в одной единице компиляции, он должен быть помещен в заголовочный файл, как и определения встроенных функций. Поэтому объявление и определение шаблона неразделимы: вы не можете объявить шаблонные функции или классы в заголовке, но скрыть их определения в файле .cpp, потому что это сделает данные определения невидимыми в любом другом файле .cpp, который содержит этот заголовок.

Компоновка

Каждое определение в C и C++ имеет свойство, известное как *компоновка (связь)*. Определение с *внешней связью* видимо, и на него могут ссылаться единицы компиляции, отличные от той, в которой оно появляется. Определение с *внутренней связью* можно увидеть только внутри единицы компиляции, в которой оно появляется, и, следовательно, на него не могут ссылаться другие единицы компиляции. Мы называем это свойство *компоновкой*, потому что оно определяет, разрешено ли компоновщику использовать перекрестную ссылку на рассматриваемый объект. Таким образом, в некотором смысле компоновка — это эквивалент ключевых слов `public:` и `private:` в определениях классов C++.

По умолчанию определения имеют внешнюю компоновку. Ключевое слово `static` используется для изменения компоновки определения на внутреннюю. Обратите внимание на то, что два или более идентичных определения `static` в двух или более разных файлах `.cpp` рассматриваются компоновщиком как *разные сущности* (как если бы им были даны разные имена), поэтому они *не* будут генерировать ошибку «множественно определенный символ». Вот несколько примеров:

foo.cpp
```
// Эта переменная может использоваться другими
// файлами .cpp (внешняя компоновка)
U32 gExternalVariable;

// Эта переменная может применяться только в foo.cpp
// (внутренняя компоновка)
static U32 gInternalVariable;

// Эта функция может быть вызвана из других файлов
// .cpp (внешняя компоновка)
void externalFunction()
{
    // ...
}

// Эта функция может быть вызвана только из foo.cpp
// (внутренняя компоновка)
static void internalFunction()
{
    // ...
}
```

bar.cpp
```
// Это объявление предоставляет доступ к переменной в foo.cpp
extern U32 gExternalVariable;

// Нет ошибки
// Эта gInternalVariable отличается от определенной в foo.cpp
// Мы могли бы назвать ее gInternalVariableForBarCpp –
// эффект был бы таким же
static U32 gInternalVariable;
```

```
// Нет ошибки
// Эта функция отличается от версии в foo.cpp
// Она работает так, как если бы мы назвали ее internalFunctionForBarCpp()
static void internalFunction()
{
    // ...
}

// ОШИБКА – многократно определенный символ!
void externalFunction()
{
    // ...
}
```

В техническом смысле *объявления* вообще не имеют свойства компоновки, потому что не выделяют никакого хранилища в исполняемом образе. Поэтому нет никаких сомнений относительно того, следует ли разрешить компоновщику перекрестную ссылку на это хранилище. Объявление — это просто ссылка на объект, определенный в другом месте. Однако иногда удобно говорить об объявлениях как об имеющих внутреннюю связь, поскольку декларация применяется только к той единице компиляции, в которой она появляется. Если мы позволим себе немного смягчить терминологию, то объявления *всегда* будут иметь внутреннюю связь — невозможно будет дать перекрестную ссылку на одно объявление в нескольких файлах `.cpp`. (Если мы поместим объявление в заголовочный файл, то несколько файлов `.cpp` могут увидеть его, но в действительности каждый из них получает отдельную копию объявления, которая имеет внутреннюю связь внутри этого модуля компиляции.)

Это приводит нас к реальной причине того, что определения встроенных функций разрешены в заголовочных файлах: так произошло, потому что встроенные функции по умолчанию имеют *внутреннюю связь*, как если бы они были объявлены `static`. Если несколько файлов `.cpp` заголовка включают (`#include`) заголовочный файл, содержащий определение встроенной функции, каждая единица компиляции получает свою копию тела этой функции и ошибки «множественно определенный символ» не генерируются. Компоновщик видит каждую копию как отдельную сущность.

3.3.5. Структура памяти программы C/C++

Программа, написанная на C или C++, хранит свои данные в разных местах памяти. Чтобы понять, как распределено хранилище и как работают различные типы переменных C/C++, необходимо понять структуру памяти программы на C/C++.

Исполняемый образ

Когда разрабатывается программа на C/C++, компоновщик создает *исполняемый файл*. Большинство UNIX-подобных операционных систем, включая многие игровые приставки, задействует популярный формат исполняемых файлов, называемый форматом *исполняемых и компонуемых файлов* (executable and linking format, ELF).

Соответственно, исполняемые файлы в этих системах имеют расширение `.elf`. Исполняемый формат Windows похож на формат ELF, исполняемые файлы под Windows имеют расширение `.exe`. Независимо от формата исполняемый файл всегда содержит частичный *образ* программы, который будет загружен в память при его запуске. Я говорю, что образ частичный, потому что программа обычно выделяет память во время выполнения в дополнение к памяти, размещенной в ее исполняемом образе.

Исполняемый образ делится на непрерывные блоки, называемые *сегментами* или *секциями*. Каждая операционная система располагает их по-своему, и расстановка может также немного различаться от одного исполняемого файла к другому в той же операционной системе. Образ обычно состоит как минимум из следующих четырех сегментов.

1. *Текстовый сегмент*. Иногда называемый *сегментом кода*, этот блок содержит исполняемый машинный код для всех функций, определенных программой.
2. *Сегмент данных*. Содержит все *инициализированные* глобальные и статические переменные. Память, необходимая для каждой глобальной переменной, распределяется точно так же, как она будет отображаться при запуске программы, и все правильные нужные значения заполняются. Поэтому, когда исполняемый файл загружается в память, инициализированные глобальные и статические переменные готовы к работе.
3. *Сегмент BSS*. BSS — это устаревшее сокращение, которое означает «блок, начинающийся с символа» (block started by symbol). Этот сегмент содержит все определенные, но *не инициализированные* программой глобальные и статические переменные. Языки C и C++ явно определяют начальное значение любой неинициализированной глобальной или статической переменной равным нулю. Но вместо того, чтобы хранить потенциально очень большой блок нулей в разделе BSS, компоновщик просто хранит *счетчик* того, сколько нулевых байтов требуется для учета всех неинициализированных глобальных и статических переменных в сегменте. Когда исполняемый файл загружается в память, операционная система резервирует запрошенное количество байтов для раздела BSS и заполняет его нулями до вызова точки входа в программу, например `main()` или `WinMain()`.
4. *Сегмент данных только для чтения*. Иногда называемый сегментом *rodata*, он содержит любые глобальные данные только для чтения (постоянные), определенные программой. Так, в этом сегменте находятся все константы с плавающей точкой (например, `const float kPi = 3.141592f;`) и все экземпляры глобальных объектов, которые были объявлены с помощью ключевого слова `const` (например, `const Foo gReadOnlyFoo;`). Обратите внимание на то, что целочисленные константы (например, `const int kMaxMonsters = 255;`) часто используются компилятором в качестве *именованных констант*, то есть вставляются непосредственно в машинный код, где бы они ни использовались. Такие константы занимают место в текстовом сегменте, но их нет в сегменте данных только для чтения.

Глобальные переменные (переменные, определенные *в области видимости файла* вне любой функции или объявления класса) хранятся либо в данных, либо в сегментах BSS в зависимости от того, были ли они инициализированы. Следующая глобальная переменная будет сохранена в сегменте данных, потому что она инициализирована:

```
F32 gInitializedGlobal = -2.0f;
```

И следующая глобальная переменная будет распределена и инициализирована равной нулю операционной системой на основе спецификаций, приведенных в сегменте BSS, поскольку она не инициализирована программистом:

```
F32 gUninitializedGlobal;
```

Мы видели, что ключевое слово `static` можно использовать для определения *внутренней компоновки* глобальной переменной или функции, что означает: они будут скрыты от других единиц компиляции. Ключевое слово `static` применяется также для объявления глобальной переменной *внутри функции*. Функционально-статическая переменная *лексически ограничена* функцией, в которой объявлена (то есть имя переменной можно увидеть только внутри функции). Она инициализируется при первом вызове функции, а не до вызова `main()`, как в случае статических переменных в области видимости файла. Но с точки зрения расположения памяти в исполняемом образе функционально-статическая переменная действует идентично файловой статической глобальной переменной — она сохраняется в сегменте данных или BSS в зависимости от того, была ли она инициализирована:

```
void readHitchhikersGuide(U32 book)
{
    static U32 sBooksInTheTrilogy = 5; // Сегмент данных
    static U32 sBooksRead;             // Сегмент BSS
    // ...
}
```

Стек программы

Когда исполняемая программа загружается в память и запускается, операционная система резервирует область памяти для *стека программы*. Всякий раз, когда вызывается функция, область стековой памяти помещается в стек — мы называем этот блок памяти *кадром стека*. Если функция `a()` вызывает другую функцию `b()`, новый кадр стека для `b()` помещается над кадром `a()`. Когда `b()` возвращает управление, ее кадр стека выталкивается и выполнение продолжается там, где была остановлена `a()`.

Кадр стека хранит три вида данных.

1. *Адрес возврата* вызывающей функции, так что, когда вызываемая функция возвращает управление, выполнение программы продолжится в вызывающей функции.

2. Содержимое всех соответствующих *регистров ЦП*. Это позволяет новой функции использовать регистры любым удобным для нее способом, не опасаясь перезаписи данных, необходимых для вызывающей функции. После возврата в вызывающую функцию состояние регистров восстанавливается, так что выполнение вызывающей функции может возобновиться. Возвращаемое значение вызываемой функции, если таковое имеется, обычно оставляется в определенном регистре, чтобы вызывающая функция могла получить его, но остальные регистры восстанавливаются до первоначальных значений.
3. Все *локальные переменные*, объявленные функцией (известны также как *автоматические переменные*). Это позволяет каждому отдельному вызову функции сохранять собственную копию каждой локальной переменной, даже когда функция вызывает сама себя рекурсивно. (На практике некоторые локальные переменные фактически распределяются по регистрам ЦП, а не хранятся в кадре стека, но по большей части они работают так, как если бы размещались в кадре стека функции.)

Вталкивание и выталкивание кадров стека обычно осуществляется установкой значения одного регистра в ЦП, известного как указатель стека. На рис. 3.10 показано, что происходит, когда выполняются приведенные далее функции:

```
void c()
{
    U32 localC1;
    // ...
}

F32 b()
{
    F32 localB1;
    I32 localB2;

    // ...

    c();

    // ...

    return localB1;
}
void a()
{
    U32 aLocalsA1[5];

    // ...

    F32 localA2 = b();

    // ...
}
```

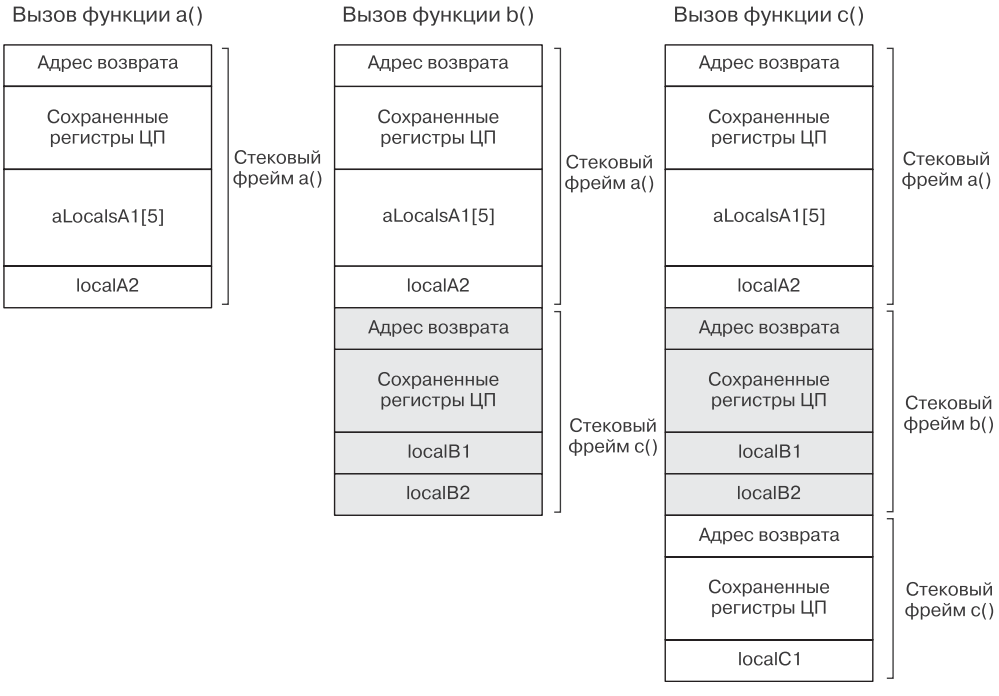


Рис. 3.10. Кадры стека

Когда функция, содержащая автоматические переменные, возвращается, ее стековый фрейм прекращает существование и все автоматические переменные в функции должны обрабатываться так, как будто их больше не существует. Технически память, занятая этими переменными, все еще находится в оставленном кадре стека, но она, скорее всего, будет перезаписана, как только будет вызвана другая функция. Распространенная ошибка заключается в возвращении адреса локальной переменной, например:

```
U32* getMeaningOfLife()
{
    U32 anInteger = 42;
    return &anInteger;
}
```

Это *может* сработать, только если вы немедленно используете возвращенный указатель, не вызывая перед этим никаких других функций. Но чаще всего такой код вызовет критическую ошибку, иногда таким образом, что ее трудно найти и отладить.

Динамическое размещение на куче

До сих пор мы видели, что данные программы могут храниться как глобальные или статические переменные либо как локальные переменные. Глобальные и статические переменные размещаются в исполняемом образе, как определено сегментами

данных и BSS исполняемого файла. Локальные размещаются в программном стеке. Оба этих типа хранилищ определены *статически*, что означает: размер и расположение памяти известны в момент, когда программа компилируется и компонуется. Однако требования к памяти программы часто не полностью известны на стадии компиляции. Программа обычно должна динамически выделять *дополнительную* память.

Чтобы обеспечить динамическое распределение, операционная система поддерживает блок памяти для каждого запущенного процесса, из которого память может быть выделена путем вызова `malloc()` (или специфичной для ОС функции, такой как `HeapAlloc()` в Windows) и позже освобождена для повторного использования процессом в будущем с помощью вызова `free()` (или специфичной для ОС функции, такой как `HeapFree()`). Этот блок памяти известен как *куча памяти* или *свободное хранилище*. Распределяя память динамически, мы иногда говорим, что эта память находится *в куче*.

В C++ глобальные операторы `new` и `delete` используются для выделения и освобождения памяти в куче. Однако будьте осторожны — отдельные классы могут переопределять эти операторы для распределения памяти пользовательскими способами, и даже *глобальные* операторы `new` и `delete` могут быть переопределены, поэтому вы просто не можете быть уверены, что `new` всегда выделит память в куче.

Мы подробнее обсудим динамическое распределение памяти в главе 7. Дополнительные сведения о нем вы найдете по адресам en.wikipedia.org/wiki/Dynamic_memory_allocation и https://ru.wikipedia.org/wiki/Динамическое_распределение_памяти.

3.3.6. Переменные-члены

Структуры `structs` C и классы C++ позволяют группировать переменные в логические единицы. Важно помнить, что *объявление* `class` или `structure` не выделяет памяти. Это просто описание макета данных — форма для печенья, которую можно использовать для последующего выпекания *экземпляров* данной структуры или класса, например:

```
struct Foo // объявление структуры
{
    U32 mUnsignedValue;
    F32 mFloatValue;
    bool mBooleanValue;
};
```

Как только структура или класс объявлены, память для них может быть распределена любым из способов выделения памяти для примитивного типа данных, например:

- как для автоматической переменной в стеке программы:

```
void someFunction()
{
    Foo localFoo;
    // ...
}
```

- как для глобальной, статической, определенной в файле или функции:

```
Foo gFoo;
static Foo sFoo;

void someFunction()
{
    static Foo sLocalFoo;
    // ...
}
```

- как для динамически выделяемой в куче. В этом случае память для указателя или ссылочной переменной, используемой для хранения адреса данных, может быть выделена как для автоматических, глобальных, статических переменных или даже динамически:

```
Foo* gpFoo = nullptr; // global pointer to a Foo

void someFunction()
{
    // выделяем память в куче для экземпляра Foo
    gpFoo = new Foo;

    // ...

    // выделяем память для другой Foo, назначить локальный указатель
    Foo* pAnotherFoo = new Foo;

    // ...

    // выделяем память для POINTER (указателя) на Foo на куче
    Foo** ppFoo = new Foo*;
     (*ppFoo) = pAnotherFoo;
}
```

Статические члены класса

Как мы уже видели, ключевое слово `static` имеет много значений в зависимости от контекста.

- Когда применяется в области видимости файла, `static` означает «ограничить видимость этой переменной или функции, чтобы ее можно было увидеть только внутри данного файла `.cpp`».
- При использовании в области действия функции `static` означает «эта переменная является глобальной, а не автоматической, но ее можно увидеть только внутри этой функции».
- При задействовании внутри структуры или класса `static` означает «эта переменная не является обычной переменной-членом, а действует как глобальная переменная».

Обратите внимание: когда `static` используется внутри объявления класса, оно не управляет видимостью переменной, как происходит при использовании

в области видимости файла, — наоборот, оно разграничивает обычные переменные — члены каждого экземпляра и переменные класса, которые действуют как глобальные. *Видимость* статической переменной класса определяется применением ключевых слов `public:`, `protected:` или `private:` в объявлении класса. Статические переменные класса автоматически включаются в пространство имен класса или структуры, в которой они объявлены. Таким образом, имя класса или структуры должно использоваться для устранения неоднозначности имени переменной всякий раз, когда она используется вне этого класса или структуры (например, `Foo::sVarName`).

Как и объявление `extern` для обычной глобальной переменной, объявление статической переменной класса внутри класса не выделяет память. Память для статической переменной класса должна быть определена в файле `.cpp`, например:

foo.h
```
class Foo
{
public:
    static F32 sClassStatic;  // не происходит выделения памяти!
};
```

foo.cpp
```
F32 Foo::sClassStatic = -1.0f; // выделить память и инициализировать
```

3.3.7. Расположение объектов в памяти

Полезно иметь возможность визуализировать структуру памяти ваших классов и структур. Обычно это довольно просто — можно просто нарисовать прямоугольник для структуры или класса с горизонтальными линиями, разделяющими элементы данных. Пример такой схемы для `struct Foo`, приведенной далее, показан на рис. 3.11.

```
struct Foo
{
    U32   mUnsignedValue;
    F32   mFloatValue;
    I32   mSignedValue;
};
```

Размеры элементов данных важны и должны быть представлены в схемах. Это легко сделать, используя ширину каждого элемента данных для обозначения его размера в битах. То есть 32-разрядное целое число должно примерно в четыре раза превышать ширину 8-битного целого числа (рис. 3.12):

```
struct Bar
{
    U32   mUnsignedValue;
    F32   mFloatValue;
    Bool  mBooleanValue; // На схеме показан размер 8 битов
};
```

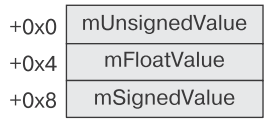


Рис. 3.11. Расположение в памяти простой структуры

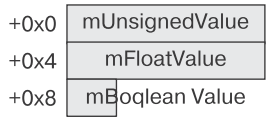


Рис. 3.12. Структура памяти с указанием ширины для задания размеров элементов

Выравнивание и упаковка

Когда мы начнем целенаправленно думать о расположении структур и классов в памяти, мы, вероятно, станем задаваться вопросом, что происходит, когда небольшие элементы данных перемежаются более крупными элементами, например:

```
struct InefficientPacking
{
    U32   mU1; // 32 бита
    F32   mF2; // 32 бита
    U8    mB3; // 8 бит
    I32   mI4; // 32 бита
    bool  mB5; // 8 бит
    char* mP6; // 32 бита
};
```

Вы можете думать, что компилятор просто упаковывает элементы данных в память настолько плотно, насколько возможно. Но на практике так происходит не всегда. Обычно компилятор оставляет дыры в структуре (рис. 3.13). (Некоторым компиляторам может быть предложено не оставлять такие дыры с помощью директивы препроцессора, такой как `#pragma pack`, или параметров командной строки. Но поведение по умолчанию заключается в разнесении элементов, показанном на рис. 3.13.)

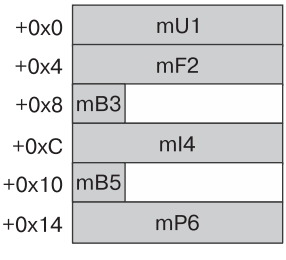


Рис. 3.13. Неэффективная структура упаковки из-за неравномерных размеров элементов данных

Почему компилятор оставляет эти дыры? Причина заключается в том, что каждый тип данных имеет естественное *выравнивание*, которое необходимо соблюдать, чтобы процессор мог эффективно считывать память и записывать в нее. *Выравнивание* объекта данных относится к тому, кратен ли его *адрес в памяти* его *размеру* (который обычно является степенью двойки).

- Объект с 1-байтовым выравниванием может находиться по любому адресу памяти.
- Объект с 2-байтовым выравниванием может располагаться только по четным адресам, то есть адресам, наименее значимым полубайтом которых является 0x0, 0x2, 0x4, 0x8, 0xA, 0xC или 0xE.

- Объект с 4-байтовым выравниванием может находиться только по адресам, кратным 4, то есть по адресам, наименее значимым полубайтом которых является 0x0, 0x4, 0x8 или 0xC.
- Выровненный 16-байтовый объект размещается только по адресам, кратным 16, то есть по адресам, наименее значимым полубайтом которых является 0x0.

Выравнивание важно, потому что многие современные процессоры фактически могут читать и записывать только правильно выровненные блоки данных. Например, если программа запрашивает чтение 32-разрядного (4-байтового) целого числа по адресу 0x6A341174, контроллер памяти успешно загрузит данные, поскольку адрес выровнен по 4 байта (в данном случае его наименее значимый полубайт 0x4). Но если сделан запрос на загрузку 32-разрядного целого числа с адреса 0x6A341173, контроллер памяти должен прочитать два 4-байтовых блока: один в 0x6A341170 и один в 0x6A341174. Затем он должен замаскировать и соединить две части 32-разрядного целого с помощью логического ИЛИ и поместить результат в регистр назначения на ЦП (рис. 3.14).

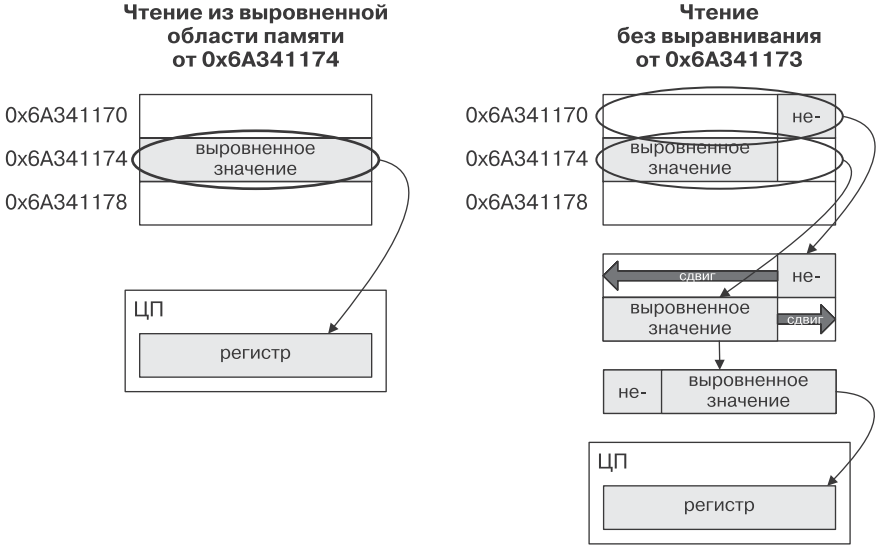


Рис. 3.14. Чтение 32-разрядного целого числа из выровненной и невыровненной областей

Некоторые микропроцессоры даже не пытаются сделать ничего подобного. Если вы запросите чтение или запись невыровненных данных, то можете получить мусор. Или программа аварийно завершится! (PlayStation 2 — яркий пример такой нетерпимости к невыровненным данным.)

Различные типы данных предъявляют разные требования к выравниванию. Хорошее практическое правило: тип данных должен быть выровнен по границе, равной ширине типа данных в байтах. Например, 32-битные значения обычно

имеют требование 4-байтового выравнивания, 16-битные должны быть выровнены по 2 байта, а 8-битные могут быть сохранены по любому адресу (выровнены по 1 байту). В процессорах, которые поддерживают векторную математику SIMD, каждый из векторных регистров содержит четыре 32-разрядных числа с плавающей точкой — всего 128 бит, или 16 байт. И как вы можете догадаться, для вектора с четырьмя переменными типа `float` SIMD обычно требуется 16-байтовое выравнивание.

Это возвращает нас к дырам в неэффективной упаковке `struct Inefficient Packing`, которые показаны на рис. 3.13. Когда в структуре или классе более мелкие типы данных, такие как 8-битные `bools`, перемежаются более крупными типами, такими как 32-битные целые числа или числа с плавающей запятой, компилятор вводит отступы (дыры), чтобы гарантировать, что все будет правильно выровнено. Следует немного подумать о выравнивании и упаковке при объявлении структур данных. Просто переставив элементы `struct InefficientPacking` из приведенного ранее примера, мы можем сохранить часть потерянного пространства, как показано далее и на рис. 3.15:

```
struct MoreEfficientPacking
{
    U32   mU1; // 32 бита (4-байтовое выравнивание)
    F32   mF2; // 32 бита (4-байтовое выравнивание)
    I32   mI4; // 32 бита (4-байтовое выравнивание)
    char* mP6; // 32 бита (4-байтовое выравнивание)
    U8    mB3; // 8 бит   (1-байтовое выравнивание)
    bool  mB5; // 8 бит   (1-байтовое выравнивание)
};
```

На рис. 3.15 показано, что размер структуры в целом теперь составляет 20, а не 18 байт, как можно было бы ожидать, потому что она была дополнена двумя байтами в конце. Это делает компилятор, чтобы обеспечить правильное выравнивание структуры в *контексте массива*. То есть если массив этих структур определен и первый его элемент выровнен, то заполнение в конце гарантирует, что *все последующие элементы* тоже будут выровнены правильно.

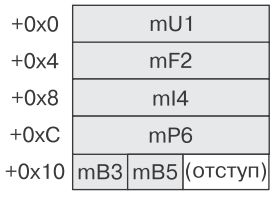


Рис. 3.15. Более эффективная упаковка, достигнутая группировкой небольших элементов

Выравнивание структуры в целом равно наибольшему среди ее членов по требованию выравнивания. В приведенном ранее примере наибольшее выравнивание элемента составляет 4 байта, поэтому структура в целом должна быть выровнена по 4 байтам. Многим нравится добавлять явное заполнение в конец структур, чтобы сделать это пространство видимым и явным, как тут:

```
struct BestPacking
{
    U32   mU1;     // 32 бита (4-байтовое выравнивание)
    F32   mF2;     // 32 бита (4-байтовое выравнивание)
```

```
    I32   mI4;     // 32 бита (4-байтовое выравнивание)
    char* mP6;     // 32 бита (4-байтовое выравнивание)
    U8    mB3;     // 8 бит   (1-байтовое выравнивание)
    bool  mB5;     // 8 бит   (1-байтовое выравнивание)
    U8    _pad[2]; // явное заполнение
};
```

Структура памяти классов C++

Две вещи делают классы C++ немного отличными от структур C с точки зрения размещения в памяти: *наследование* и *виртуальные функции*.

Когда класс B наследуется от класса A, члены данных B просто сразу же появляются после A в памяти (рис. 3.16). Каждый новый производный класс просто привязывает свои элементы данных в конце, хотя требования выравнивания могут порождать заполнение между классами. (Множественное наследование делает некоторые глупые вещи, такие как включение нескольких копий одного базового класса в структуру памяти производного класса. Мы не будем подробно останавливаться на этом, потому что программисты игр обычно избегают множественного наследования.)

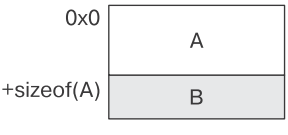


Рис. 3.16. Влияние наследования на расположение класса в памяти

Если класс содержит или наследует одну или несколько *виртуальных функций*, то в структуру памяти класса помещают дополнительные 4 байта (или 8 байт, если целевое оборудование использует 64-разрядные адреса), обычно в самом начале класса. Эти 4 или 8 байт вместе называются *указателем виртуальной таблицы*, или *vpointer*, потому что они содержат указатель на структуру данных, известную как *таблица виртуальных функций* или *vtable*. Vtable для определенного класса содержит указатели на все виртуальные функции, которые он объявляет или наследует. Каждый конкретный класс имеет собственную виртуальную таблицу, и каждый экземпляр этого класса имеет указатель на нее, хранящийся в его vpointer.

Таблица виртуальных функций лежит в основе полиморфизма, потому что она позволяет писать код, не знающий конкретных классов, с которыми имеет дело. Возвращаясь к вездесущему примеру базового класса `Shape` с производными классами `Circle`, `Rectangle` и `Triangle`, давайте представим, что `Shape` определяет виртуальную функцию `Draw()`. Все производные классы переопределяют ее, предоставляя различные реализации с именами `Circle::Draw()`, `Rectangle::Draw()` и `Triangle::Draw()`. Виртуальная таблица для любого класса, производного от `Shape`, будет содержать запись для функции `Draw()`, но эта запись станет указывать на различные реализации функции, в зависимости от конкретного класса. В виртуальной таблице `Circle` будет указатель на `Circle::Draw()`, в `Rectangle` — на `Rectangle::Draw()`, а в `Triangle` — на `Triangle::Draw()`. Учитывая произвольный указатель на `Shape (Shape * pShape)`, код может просто разыменовать указатель `vtable`, найти запись функции `Draw()` в `vtable` и вызвать ее. Результатом будет вы-

зов `Circle::Draw()`, когда `pShape` указывает на экземпляр `Circle`, `Rectangle::Draw()`, когда `pShape` указывает на `Rectangle`, и `Triangle::Draw()`, когда `pShape` указывает на `Triangle`.

Эти идеи иллюстрируются приводимым далее фрагментом кода. Обратите внимание на то, что базовый класс `Shape` определяет две виртуальные функции, `SetId()` и `Draw()`, последняя из которых объявлена чисто виртуальной. (Это означает, что `Shape` не обеспечивает реализацию по умолчанию функции `Draw()` и производные классы должны переопределить ее, если хотят быть инстанцируемыми.) Класс `Circle` наследует от `Shape`, добавляет некоторые элементы данных и функции для управления своим центром и радиусом и переопределяет функцию `Draw()` (рис. 3.17).

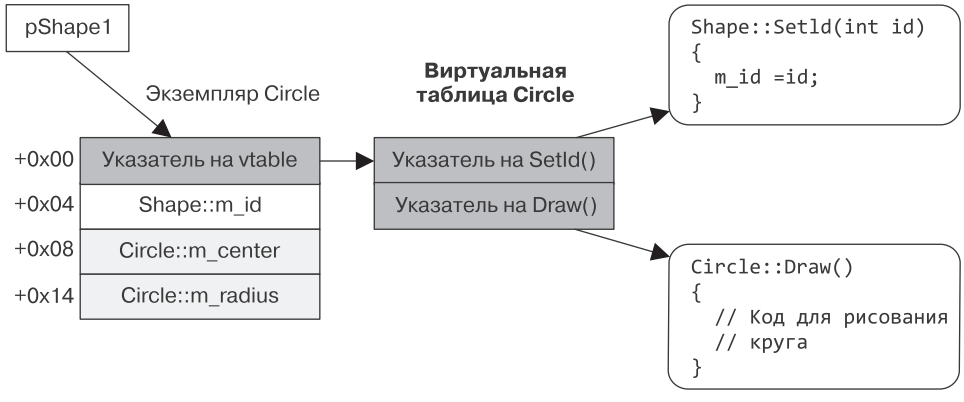


Рис. 3.17. pShape1 указывает на экземпляр класса Circle

Класс `Triangle` также происходит от `Shape`. Он добавляет массив объектов `Vector3` для хранения своих трех вершин и добавляет некоторые функции для получения и установки отдельных вершин. Класс `Triangle` переопределяет `Draw()`, как и следовало ожидать, и в иллюстративных целях также переопределяет `SetId()`. Образ памяти, сгенерированный классом `Triangle`, показан на рис. 3.18.

```
class Shape
{
public:
    virtual void  SetId(int id) { m_id = id; }
    int           GetId() const { return m_id; }

    virtual void  Draw() = 0; // чисто виртуальный – не реализован

private:
    int  m_id;
};

class Circle : public Shape
{
```

```
public:
    void     SetCenter(const Vector3& c) { m_center=c; }
    Vector3  GetCenter() const { return m_center; }

    void     SetRadius(float r) { m_radius = r; }
    float    GetRadius() const { return m_radius; }

    virtual void Draw()
    {
        // код для рисования круга
    }

private:
    Vector3  m_center;
    float    m_radius;
};

class Triangle : public Shape
{
public:
    void    SetVertex(int i, const Vector3& v);
    Vector3 GetVertex(int i) const { return m_vtx[i]; }

    virtual void Draw()
    {
        // код для рисования треугольника
    }

    virtual void SetId(int id)
    {
        // вызов реализации базового класса
        Shape::SetId(id);

        // делаем дополнительную работу, специфичную для треугольников...
    }

private:
    Vector3  m_vtx[3];
};

// -----------------------------

void main(int, char**)
{
    Shape* pShape1 = new Circle;
    Shape* pShape2 = new Triangle;

    pShape1->Draw();
    pShape2->Draw();

    // ...
}
```

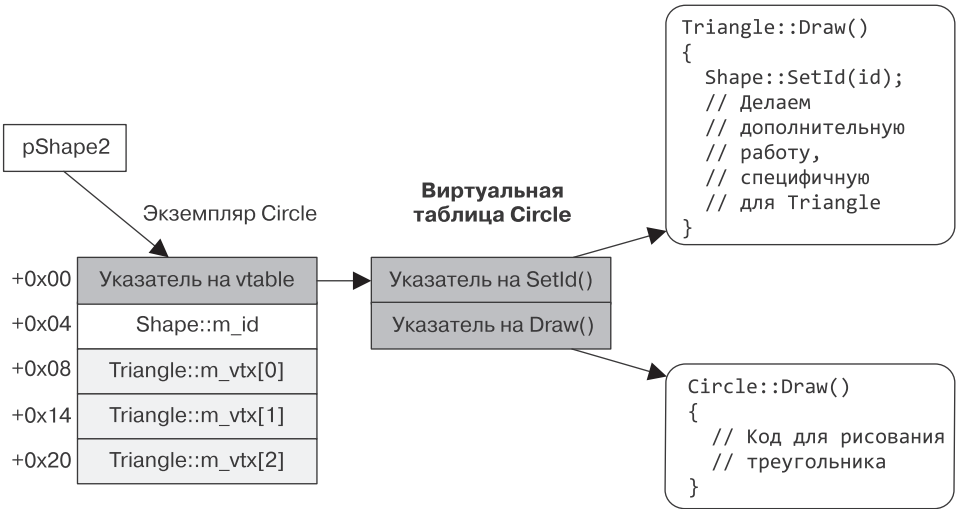


Рис. 3.18. pShape1 указывает на экземпляр класса Triangle

3.4. Основы аппаратного обеспечения компьютеров

Программирование на высокоуровневом языке, таком как C++, C# или Python, — эффективный способ создания программного обеспечения. Но чем более высокого уровня ваш язык, тем больше он ограждает вас от необходимости понимать базовые детали аппаратного обеспечения, на котором работает код. Чтобы стать действительно опытным программистом, важно разбираться в архитектуре целевого оборудования. Эти знания могут помочь вам оптимизировать код. Важно это и для параллельного программирования — все программисты должны понимать параллелизм, если они надеются в полной мере использовать возможности возрастающей степени параллелизма в современном вычислительном оборудовании.

3.4.1. Обучение на более простых компьютерах прошлых лет

На следующих страницах мы обсудим конструкцию простого универсального процессора, а не будем углубляться в особенности конкретного устройства. Тем не менее некоторые читатели могут счесть полезным расширить теоретическое обсуждение и почитать о специфике работы реального процессора. Я сам узнал, как работают компьютеры, когда был подростком, программируя свои компьютеры Apple II и Commodore 64. Обе эти машины имели простой процессор, известный как 6502, который был разработан и изготовлен MOS Technology Inc. Я также узнал

кое-что из того, чему научился, читая об общем предке всей линейки процессоров Intel x86, 8086 (и его «двоюродном брате» 8088) и работая с ним. Оба процессора отлично подходят для учебных целей благодаря своей простоте. Это особенно верно для 6502, который является самым простым процессором, с каким я когда-либо работал. Как только вы поймете, как работает 6502 и/или 8086, разобраться в современных процессорах станет намного проще.

Приведу несколько отличных ресурсов, посвященных деталям архитектуры и программирования для 6502 и 8086.

- В главах 1 и 2 книги Гэри Б. Литтла *Inside the Apple IIe* [33] представлен большой обзор программ для 6502 на языке ассемблера. Книга доступна в Интернете по адресу www.apple2scans.net/files/InsidetheIIe.pdf.
- flint.cs.yale.edu/cs421/papers/x86-asm/asm.html — хороший обзор набора команд для архитектуры x86.
- Вы должны обязательно прочитать «Черную книгу» по программированию Майкла Абраша [1], где найдете множество полезной информации о программировании для процессора 8086, а также невероятные советы по оптимизации программного обеспечения и графическому программированию, которые использовались, когда игры только зарождались.

3.4.2. Архитектура компьютера

Простейший компьютер состоит из *центрального процессора* и банка *памяти*, расположенных на печатной плате, называемой *материнской платой*, соединенных друг с другом через одну или несколько *шин* и сообщающихся с внешними периферийными устройствами с помощью набора *входных/выходных портов* и/или *слотов расширения*. Этот базовый проект упоминается как архитектура фон Неймана, потому что впервые был описан в 1945 году математиком и физиком Джоном фон Нейманом и его коллегами во время работы над секретным проектом ENIAC. Простая архитектура последовательного компьютера показана на рис. 3.19.

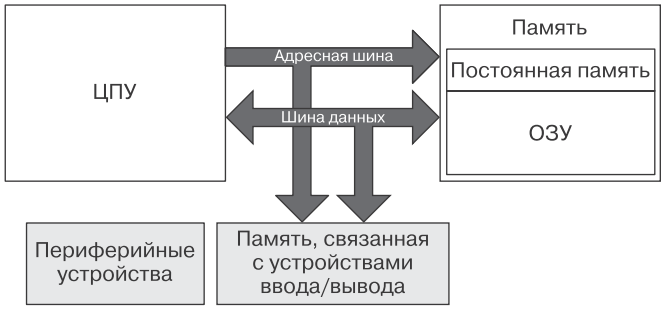


Рис. 3.19. Простая архитектура последовательного компьютера фон Неймана

3.4.3. Центральный процессор

Процессор — это мозг компьютера. Он состоит из следующих компонентов:

- *арифметико-логического устройства* (АЛУ/ALU) для выполнения арифметических действий с целыми числами и побитового сдвига;
- *модуля операций с плавающей точкой* (floating-point unit, FPU) для реализации арифметики с числами с плавающей точкой (обычно с использованием стандартного представления IEEE 754);
- *модуля векторной обработки* (vector processing unit, VPU), имеющегося практически у всех современных процессоров, который способен выполнять операции с плавающей точкой и целочисленные операции над несколькими элементами данных параллельно;
- *контроллера памяти* (memory controller, MC) или *блока управления памятью* (memory management unit, MMU) для взаимодействия с устройствами памяти на чипе и вне его;
- *набора регистров*, которые служат временным хранилищем во время расчетов (помимо прочего);
- *блока управления* (control unit, CU) для декодирования и отправки инструкций на машинном языке другим компонентам на чипе и маршрутизации данных между ними.

Все эти компоненты приводятся в действие периодическим сигналом прямоугольной формы, известным как такт. Тактовая частота определяет частоту, с которой ЦП выполняет, например, инструкции или арифметические операции. Типичные компоненты последовательного процессора показаны на рис. 3.20.

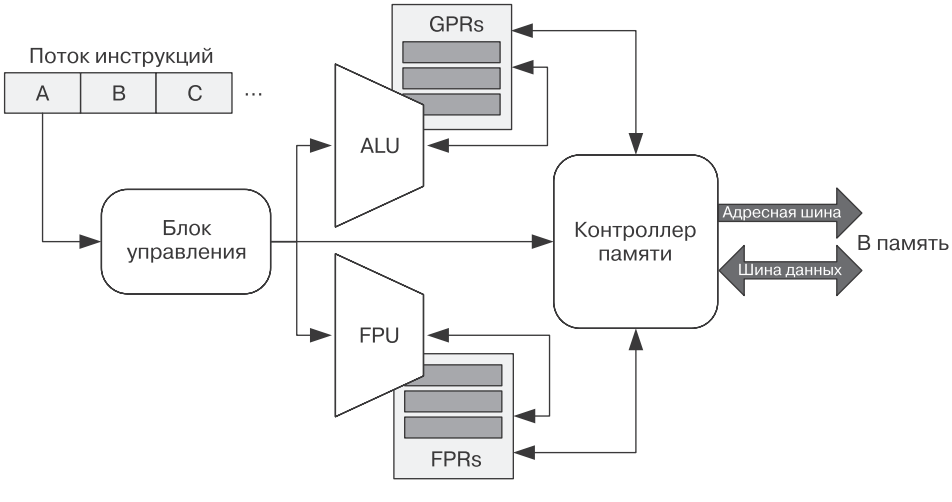


Рис. 3.20. Типичные компоненты последовательного процессора

Арифметико-логическое устройство

Арифметико-логическое устройство выполняет унарные и бинарные арифметические операции, такие как отрицание, сложение, вычитание, умножение и деление, а также логические операции, такие как И (AND), ИЛИ (OR), исключающее ИЛИ (XOR или EOR), побитовое дополнение и сдвиг. В некоторых конструкциях ЦПУ ALU физически разбивается на арифметический блок (AU) и логический блок (LU).

ALU обычно выполняет только целочисленные операции. Расчеты с плавающей точкой требуют совсем других схем и обычно производятся физически отдельным модулем операций с плавающей точкой (FPU). Ранние процессоры, такие как Intel 8088/8086, не имели встроенного процессора. Если требовалась поддержка математических операций с плавающей точкой, их следовало дополнить отдельным чипом сопроцессора FPU, таким как Intel 8087. В более поздних версиях ЦП FPU обычно включался в его основную матрицу.

Блок векторной обработки

Блок векторной обработки (VPU) работает как комбинация ALU/FPU в том смысле, что обычно может выполнять как целочисленные вычисления, так и вычисления с плавающей точкой. VPU отличает способность применять арифметические операторы к *векторам* входных данных (каждый из которых обычно имеет от 2 до 16 значений с плавающей точкой или до 64 целочисленных значений различной ширины), а не к скалярным входным даннымм. Векторная обработка известна также как *одна команда для множественных данных* (single instruction multiple data, SIMD), потому что один арифметический оператор, например умножение, применяется к нескольким парам входов одновременно (более подробно обсудим это в разделе 4.10).

Современные ЦП фактически не содержат FPU как таковой. Вместо этого все вычисления с плавающей точкой, даже включающие скалярные значения с плавающей точкой, выполняются в VPU. Исключение FPU уменьшает количество транзисторов на кристалле ЦП, позволяя использовать эти транзисторы для реализации больших кэшей, более сложной логики выполнения не по порядку и т. д. И, как мы увидим в подразделе 4.10.6, оптимизирующие компиляторы обычно преобразуют математические вычисления, выполняемые для переменных с плавающей точкой, в векторизованный код, который в любом случае использует VPU.

Регистры

Чтобы максимально увеличить производительность, ALU или FPU обычно могут выполнять вычисления только с данными, которые существуют в специальных высокоскоростных ячейках памяти, называемых *регистрами*. Регистры обычно физически отделены от основной памяти компьютера, расположены на кристалле и в непосредственной близости от компонентов, которые к ним обращаются. Они реализуются с использованием быстрой, дорогостоящей, многопортовой статической оперативной памяти (static random access memory, SRAM). (Больше

информации о технологиях памяти — в подразделе 3.4.5.) Набор регистров в ЦП называется *регистровым файлом*.

Поскольку регистры не являются частью основной памяти[1], они обычно не имеют адресов, но имеют имена. Они могут быть такими же простыми, как R0, R1, R2 и т. д., хотя в ранних процессорах обычно использовались буквы или короткие мнемоники в качестве имен регистров. Например, у Intel 8088/8086 было четыре 16-разрядных регистра общего назначения с именами AX, BX, CX и DX. A 6502 от MOS Technology, Inc. выполнял все свои арифметические операции, используя регистр, известный как *накопитель*[2] (A), и задействовал два вспомогательных регистра, называемых X и Y, для других операций, таких как индексация в массиве.

Некоторые из регистров ЦП предназначены для общих вычислений. Они называются *регистрами общего назначения* (general-purpose registers, GPR). Каждый процессор содержит также ряд *регистров специального назначения* (special-purpose registers, SPR). К ним относятся:

- *указатель инструкции* (instruction pointer, IP);
- *указатель стека* (stack pointer, SP);
- *базовый указатель* (base pointer, BP);
- *регистр состояния*.

Указатель инструкции. Содержит адрес выполняемой в настоящее время инструкции в программе на машинном языке (подробнее о машинном языке говорится далее).

Указатель стека. Ранее мы видели, что *стек вызовов* программы служит и основным механизмом, с помощью которого функции вызывают друг друга, и средством выделения памяти для локальных переменных. *Указатель стека* содержит адрес вершины стека вызовов программы. Стек может увеличиваться или уменьшаться в смысле адресов памяти, но сейчас предположим, что он растет вниз. В этом случае элемент данных может быть *помещен* в стек путем вычитания размера элемента из значения указателя стека, а затем записи элемента по новому адресу, на который указывает SP. Аналогично элемент можно *извлечь* из стека, считав его с адреса, на который указывает SP, и затем добавив его размер к SP.

Базовый указатель. Содержит базовый адрес *кадра стека* текущей функции в стеке вызовов. Многие локальные переменные функции размещаются в ее стековом фрейме, хотя оптимизатор может назначать другие значения регистру на время выполнения функции. Перемещенные в стеке переменные занимают диапазон адресов памяти с уникальным смещением от базового указателя. Такую переменную

---

[1] Некоторые ранние компьютеры использовали основную оперативную память для реализации регистров. Например, 32 регистра в IBM 7030 Stretch (первый суперкомпьютер IBM на основе транзисторов) были наложены на первые 32 адреса в основной оперативной памяти. В некоторых ранних разработках ALU один из его входов поступал из регистра, а другой — из основной оперативной памяти. Эти конструкции были практичными, потому что в то время задержки доступа к ОЗУ были невелики по сравнению с общей производительностью ЦП.

[2] Термин «накопитель» возник из-за того, что ранние ALU обрабатывали по одному биту за раз и поэтому накапливали ответ, маскируя и сдвигая отдельные биты в регистр результата.

можно разместить в памяти простым вычитанием ее уникального смещения из адреса, хранящегося в BP (при условии, что стек увеличивается).

Регистр состояния. Специальный регистр, известный как *регистр состояния*, *условный регистр* или *флаг*, содержит биты, которые отражают результаты последней операции ALU. Например, если результат вычитания равен 0, *бит нулевого флага* (обычно называемый `Z`) устанавливается в регистре состояния в 1, в противном случае — очищается. Аналогично, если операция сложения привела к переполнению, означающему, что двоичная 1 должна быть перенесена в следующее слово сложения, состоящего из нескольких слов, устанавливается *бит флага переноса* (часто называемый `C`), в противном случае он очищается.

Флаги в регистре состояния могут применяться для управления потоком программы посредством *условного ветвления* или для выполнения последующих вычислений, таких как добавление бита переноса к следующему байту при сложении многобайтовых значений.

Форматы регистров. Важно понимать, что FPU и VPU чаще всего работают с собственными частными наборами регистров, а не используют целочисленные регистры ALU общего назначения. Одной из причин этого является скорость — чем ближе регистры к вычислительному устройству, которое их задействует, тем меньше времени требуется для доступа к содержащимся в них данным. Другая причина заключается в том, что регистры FPU и VPU обычно *шире*, чем *регистры общего назначения* (GPR) ALU.

Например, 32-разрядный ЦП имеет GPR шириной 32 бита каждый, но FPU может работать с 64-разрядными числами с плавающей точкой двойной точности или даже 80-разрядными расширенными значениями двойной точности, что означает: его регистры должны быть шириной 64 или 80 бит соответственно. Аналогично каждый регистр, используемый VPU, должен содержать *вектор* входных данных, что означает: эти регистры должны быть намного шире, чем типичный GPR. Например, векторный процессор Intel с набором инструкций SSE2 (Streaming SIMD Extensions 2) может быть сконфигурирован для выполнения вычислений для векторов, содержащих по четыре значения с плавающей точкой одинарной точности (32-битной) каждое или по два значения с плавающей точкой двойной точности (64-битной) каждое. Следовательно, каждый векторный регистр SSE2 имеет ширину 128 бит.

Физическое разделение регистров между ALU и FPU является одной из причин того, что преобразования между `int` и `float` оказывались очень дорогими в те времена, когда FPU были обычным явлением. Мало того, что битовая комбинация каждого значения должна была преобразовываться туда и обратно между его целочисленным форматом дополнительного кода и его представлением с плавающей запятой IEEE 754, данные также должны были физически передаваться между целочисленным регистром общего назначения и регистром FPU. Однако современные процессоры не содержат FPU как такового — вся математика с плавающей точкой выполняется модулем векторной обработки. VPU может обрабатывать как целочисленные операции, так и операции с плавающей точкой, и преобразования между этими типами намного дешевле даже при перемещении данных из целочисленного

GPU в векторный регистр или наоборот. Тем не менее стоит избегать преобразования данных между форматами `int` и `float`, где только возможно, потому что даже низкозатратное преобразование все еще дороже, чем отсутствие преобразования.

Блок управления

Если процессор является мозгом компьютера, то *блок управления* (CU) — это мозг процессора. Его задача — управлять потоком данных внутри ЦП, а также работой всех других компонентов ЦП.

CU запускает программу, читая поток инструкций *машинного языка*, декодируя каждую инструкцию, разбивая ее на свои коды операций (опкоды) и операнды, а затем отправляя рабочие запросы и/или данные маршрутизации в ALU, FPU, VPU, регистры и/или контроллер памяти в соответствии с кодом операции текущей инструкции. В конвейерных и суперскалярных процессорах CU также содержит сложные схемы для обработки прогнозирования ветвлений и планирования команд для выполнения не по порядку. Рассмотрим работу CU подробнее в подразделе 3.4.7.

3.4.4. Частота

Каждая цифровая электронная схема по сути является конечным автоматом. Для того чтобы он мог изменять состояния, цепь должна приводиться в действие соответствующим цифровым сигналом. Таким типичным сигналом может являться *изменение напряжения* в цепи с 0 до 3,3 В или наоборот.

Изменения состояния в ЦП обычно приводятся в действие периодическим прямоугольным сигналом, известным как *системные часы*. Каждый нарастающий или падающий фронт этого сигнала известен как *тактовый цикл*, и ЦП может выполнить по крайней мере одну примитивную операцию в каждом цикле. Следовательно, для процессора время представляется *квантованным*[1].

Скорость, с которой процессор может выполнять свои операции, определяется тактовой *частотой* системных часов. Тактовые частоты значительно увеличились за последние несколько десятилетий. Первые процессоры, разработанные в 1970-х годах, такие как 6502 MOS Technology и Intel 8086/8088, имели тактовые частоты в диапазоне 1–2 МГц (миллионы циклов в секунду). Современные процессоры, такие как Intel Core i7, работают в диапазоне 2–4 ГГц (*миллиарды* циклов в секунду).

Важно понимать, что для выполнения одной инструкции ЦП требуется не один такт. Не все инструкции созданы равными — одни очень простые, другие более сложные. Некоторые инструкции реализованы как комбинация более простых микроопераций (μ-ops), поэтому для их выполнения требуется намного больше циклов, чем для выполнения их простых аналогов.

Кроме того, хотя ранние процессоры действительно могли выполнять некоторые инструкции за один такт, современные конвейерные процессоры разбивают

[1] Сравните это с аналоговой электронной схемой, где время рассматривается как непрерывное. Например, генератор сигналов старой школы может генерировать синусоидальную волну, которая плавно изменяется, скажем, в пределах от –5 до 5 В.

даже самые простые инструкции на несколько *этапов*. Выполнение каждого этапа в конвейерном процессоре занимает один такт, это означает, что минимальная *задержка выполнения команд* в процессоре с *N*-ступенчатым конвейером составляет *N* тактов. Простой конвейерный ЦП может обрабатывать команды со *скоростью* одна инструкция за такт, потому что новая команда подается в конвейер каждый такт. Но если бы вы проследили выполнение одной конкретной инструкции в конвейере, потребовалось бы *N* циклов, чтобы пройти по нему от начала до конца. Обсудим конвейерные процессоры подробнее в разделе 4.2.

Тактовая частота и вычислительная мощность

Вычислительную мощность процессора или компьютера можно определить различными способами. Одним из распространенных показателей является *пропускная способность* машины — количество операций, которые она способна выполнить в течение заданного интервала времени. Пропускная способность выражается в миллионах команд в секунду (millions of instructions per second, MIPS) или операций с плавающей точкой в секунду (floating-point operations per second, FLOPS).

Поскольку инструкции или операции с плавающей точкой, как правило, не завершаются ровно за один цикл, а разным командам для выполнения требуется различное число циклов, показатели MIPS или FLOPS ЦП являются усредненными. Таким образом, вы не можете просто посмотреть на тактовую частоту процессора и определить его вычислительную мощность в MIPS или FLOPS. Например, последовательный процессор, работающий на частоте 3 ГГц, в котором для одного умножения с плавающей точкой требуется в среднем шесть циклов, может теоретически достичь 0,5 GFLOPS. Но многие факторы, включая конвейерную обработку, суперскалярные схемы, векторную обработку, многоядерные процессоры и другие формы параллелизма, способствуют запутыванию взаимосвязи между тактовой частотой и пропускной способностью. Таким образом, единственный способ определить истинную вычислительную мощность процессора или компьютера — это измерить ее, обычно с помощью стандартизированных тестов производительности.

3.4.5. Память

Память в компьютере действует как набор почтовых ящиков в почтовом отделении, причем каждый ящик, или ячейка, обычно содержит один *байт* данных (восьмибитное значение)[1]. Каждая однобайтовая ячейка памяти идентифицируется по своему *адресу*, для этого применяется простая схема нумерации от 0 до $N - 1$, где N — размер адресуемой памяти в байтах.

[1] На самом деле ранние компьютеры часто обращались к памяти в единицах слов, размер которых превышал 8 бит. Например, IBM 701 (производство 1952 года) адресовал память в единицах 36-битных слов, а PDP-1 (1959 года) мог обращаться к 18-битным словам памяти, их было до 4096. Восьмибитные байты стали популярны с приходом Intel 8008 в 1972 году. Семь бит требуется для кодирования как строчных, так и прописных букв английского алфавита. Расширение до 8 бит позволило поддерживать широкий спектр специальных символов.

Память поставляется в двух основных вариантах:

- *постоянное запоминающее устройство* (ПЗУ) (read-only memory, ROM);
- перезаписываемая память для чтения/записи, известная по историческим причинам[1] как память с произвольным доступом (random access memory, RAM) — *оперативная память*.

Модули ПЗУ сохраняют данные, даже если на них не подается питание. Некоторые типы ПЗУ могут быть запрограммированы только один раз. Другие, известные как *электронно стираемое программируемое ПЗУ* (ЭСППЗУ, или EEPROM), могут перепрограммироваться снова и снова. (Флеш-накопители — один из примеров памяти ЭСППЗУ.)

RAM может быть разделена на *статическую* RAM (SRAM) и *динамическую* RAM (DRAM). Как статическая, так и динамическая оперативная память сохраняют данные до тех пор, пока к ним подается питание. Но в отличие от статической динамическую RAM необходимо периодически обновлять, считывая, а затем перезаписывая информацию, чтобы предотвратить их исчезновение. Это связано с тем, что ячейки памяти DRAM построены на МОП-конденсаторах (CMOS), которые постепенно теряют заряд, и чтение таких ячеек памяти разрушительно для содержащихся в них данных.

RAM можно классифицировать и по другим характеристикам, таким как:

- является ли она *многопортовой*. Это означает, что к ней одновременно могут обращаться более одного компонента ЦП;
- работает она путем синхронизации с тактами (SDRAM) или асинхронно;
- поддерживает ли она доступ с двойной скоростью передачи данных (double data rate, DDR). Это означает, что к RAM можно обращаться (выполнять запись или чтение) как по нарастающему, так и по ниспадающему фронту тактового сигнала.

3.4.6. Шины

Данные передаются между процессором и памятью через соединения, известные как *шины*. Шина — это просто набор параллельных цифровых проводов, называемых *линиями*, каждый из которых может представлять один бит данных. Когда линия несет сигнал с напряжением[2], она представляет двоичную единицу, а когда

---

[1] Память с произвольным доступом была названа так, потому что в более ранних технологиях памяти для хранения данных использовались циклы задержки, то есть считывать их можно только в том порядке, в котором они были записаны. Технология RAM улучшила эту ситуацию, позволив получать доступ к данным в произвольном порядке.

[2] Ранее транзисторно-транзисторные логические устройства (transistor-transistor logic, TTL) работали при напряжении питания 5 В, поэтому пятивольтный сигнал представлял бы двоичную единицу. Большинство современных цифровых электронных устройств используют дополнительную металлоксидно-полупроводниковую логику (CMOS), которая может работать при более низком напряжении питания, обычно 1,2–3,3 В.

напряжение на нее не подается (0 В) — двоичный ноль. Пакет из *n* однобитных линий, расположенных параллельно, может передавать *n*-битное число (то есть любое число в диапазоне от 0 до $2^n - 1$).

Типичный компьютер содержит две шины: *адресную шину* и *шину данных*. Процессор запрашивает данные из памяти, передавая адрес необходимой ячейки контроллеру памяти через адресную шину. Контроллер памяти отвечает, предоставляя биты элемента данных, хранящиеся в запрашиваемой (-ых) ячейке (-ах), на шину данных, где их может видеть процессор. Аналогичным образом ЦПУ записывает элемент данных в память, передавая адрес назначения по шине адреса и помещая битовую комбинацию элемента данных для записи на шину данных. Контроллер памяти отвечает записью данных в соответствующие ячейки памяти. Следует отметить, что адресная шина и шина данных иногда реализуются в виде двух физически отдельных наборов проводов, а иногда — в виде одного набора проводов, которые мультиплексируются между функциями шины данных и адресной шины на разных этапах цикла доступа к памяти.

Ширина шины

Ширина адресной шины, измеряемая в битах, определяет диапазон адресов, к которым может обращаться ЦП (то есть размер *адресуемой памяти* в машине). Например, компьютер с 16-разрядной адресной шиной может получить доступ к памяти объемом не более 64 Кбайт, используя адреса в диапазоне от 0x0000 до 0xFFFF. Компьютер с 32-разрядной адресной шиной может получить доступ к памяти объемом до 4 Гбайт, используя адреса в диапазоне от 0x00000000 до 0xFFFFFFFF. И машина с 64-битной адресной шиной может получить доступ к потрясающим 16 ЭиБ памяти. Это $2^{64} = 16 \cdot 1024^6 \approx 1{,}8 \cdot 10^{19}$ байт!

Ширина *шины данных* определяет, сколько данных может быть передано между регистрами ЦП и памятью одновременно. (Шина данных обычно имеет ту же ширину, что и регистры общего назначения в ЦП, хотя так бывает не всегда.) Так, 8-битная шина данных означает, что данные могут передаваться всего по одному байту за раз — загрузка 16-битного значения из памяти потребовала бы двух отдельных циклов доступа к памяти: по одному для извлечения младшего и старшего значащих байтов. На другом конце спектра находится 64-битная шина данных, которая может передавать данные между памятью и 64-битным регистром за один цикл обращения к памяти.

Можно получать доступ к элементам данных, которые меньше ширины шины данных машины, но обычно это более затратно, чем доступ к элементам, ширина которых соответствует ширине шины данных. Например, при считывании 16-разрядного значения на 64-разрядном компьютере из памяти по-прежнему должны считываться полные 64-разрядные данные. Желаемое 16-битное поле затем должно быть замаскировано и, возможно, сдвинуто на нужное место в регистре назначения. Это одна из причин того, почему язык C не определяет `int` по размеру как конкретное число битов — оно было специально определено, чтобы соответствовать естественному размеру слова на целевой машине, в попытке сделать исходный

код лучше переносимым. (По иронии судьбы эта политика фактически привела к тому, что исходный код часто *хуже* переносим из-за неявных предположений о размере `int`.)

Слова

Термин «слово» часто используется для описания многобайтового значения. Однако количество байтов, составляющих слово, не является универсально определенным. Это в некоторой степени зависит от контекста.

Иногда термин «слово» относится к наименьшему многобайтовому значению, а именно к 16 битам, или 2 байтам. В этом контексте двойное слово составит 32 бита (4 байта), а четверное слово — 64 бита (8 байт). Именно так термин «слово» используется в Windows API.

В то же время термин «слово» применяется и для обозначения естественного размера элементов данных на конкретной машине. Например, машина с 32-разрядными регистрами и 32-разрядной шиной данных наиболее естественно работает с 32-разрядными (4-байтовыми) значениями. Программисты и специалисты по аппаратному обеспечению иногда говорят, что такая машина имеет *размер слова* 32 бита. Здесь вы должны понимать, что подразумевается всякий раз, когда термин «слово» используется для обозначения размера элемента данных.

N-битные компьютеры

Возможно, вы встречали термин «*n*-битный компьютер». Обычно это означает машину с *n*-битной шиной данных и/или регистрами. Но этот термин немного неоднозначен, поскольку он может относиться также к компьютеру, адресная шина которого имеет ширину *n* бит. Кроме того, на некоторых процессорах ширины шины данных и регистра не совпадают. Например, 8088 имел 16-битные регистры и 16-битную адресную шину, но у него была только 8-битная шина данных. Следовательно, внутри он действовал как 16-битный компьютер, но 8-битная шина данных заставляла его вести себя как 8-битный компьютер с точки зрения доступа к памяти. Опять же помните о контексте, когда говорите об *n*-битной машине.

3.4.7. Машинный и ассемблерный языки

Что касается процессора, то программа — это не что иное, как последовательный поток относительно простых *инструкций*. Каждая инструкция приказывает блоку управления (CU) и в конечном итоге другим компонентам в ЦП, таким как контроллер памяти, ALU, FPU или VPU, выполнять конкретную операцию. Инструкция может перемещать данные внутри компьютера либо самого ЦП или каким-то образом преобразовывать их, например выполняя арифметическую или логическую операцию над данными. Обычно инструкции в программе выполняются последовательно, хотя некоторые из них могут изменить последовательный поток управления, перепрыгнув на новое место в общем потоке инструкций программы.

Архитектура набора инструкций

Конструкция ЦП сильно варьируется от производителя к производителю. Набор всех команд, поддерживаемых данным ЦП, наряду с другими деталями его конструкции, такими как режимы адресации и формат команд в памяти, называется *архитектурой набора инструкций* (instruction set architecture, ISA). (Не следует путать с *двоичным интерфейсом приложения* языка программирования, или ABI, который определяет протоколы более высокого уровня, такие как соглашения о вызовах.) Мы не будем пытаться охватить здесь детали ISA какого-либо одного процессора, но следующие категории команд являются общими почти для всех ISA.

- *Move*. Эти инструкции перемещают данные между регистрами или между памятью и регистром. Некоторые ISA разбивают инструкцию `move` на отдельные инструкции `load` и `store`.
- *Арифметические операции*. К ним относятся не только сложение, вычитание, умножение и деление, но и другие операции, такие как унарное отрицание, инверсия, извлечение квадратного корня и т. д.
- *Битовые операторы*. К ним относятся И (AND), ИЛИ (OR), исключающее ИЛИ (XOR или EOR) и НЕ (NOT).
- *Операторы сдвига/поворота*. Эти инструкции позволяют сдвигать биты в слове данных влево или вправо, влияя или не влияя на бит переноса в регистре состояния, а также циклически перемещать их, когда значение последнего бита по направлению сдвига копируется в первый бит.
- *Сравнение*. Эти инструкции позволяют сравнивать два значения, чтобы определить, является одно меньше или больше другого или равно ему. В большинстве процессоров инструкции сравнения используют ALU для вычитания двух входных значений, тем самым устанавливая соответствующие биты в регистре состояния, а результат вычитания просто отбрасывая.
- *Прыжок и ветвление*. Позволяет изменять поток выполнения программы, сохраняя новый адрес в указателе инструкций. Его можно выполнить либо безусловно (этот процесс называется командой перехода (`jump`)), либо условно на основе состояния различных флагов в регистре состояния (в этом случае часто говорят «ветвь» (`branch`)). Например, инструкция `branch if zero` изменяет содержимое регистра IP тогда и только тогда, когда в регистре состояния установлен бит `Z`.
- *Push* и *pop*. Большинство процессоров предоставляют специальные инструкции для помещения содержимого регистра в программный стек и выталкивания текущего значения из верхней части стека в регистр.
- *Вызов функции* и *возврат*. Некоторые ISA предоставляют явные инструкции для вызова функции (также называемой процедурой или подпрограммой) и возврата из нее. Семантика вызова и возврата функции может быть обеспечена также комбинацией команд `push`, `pop` и `jump`.

- *Прерывания*. Инструкция прерывания посылает цифровой сигнал в ЦПУ, который заставляет его временно перейти к предварительно установленной подпрограмме обработки прерывания, зачастую не являющейся частью выполняемой программы. Прерывания используются для уведомления операционной системы или пользовательской программы о таких событиях, как ввод, который становится доступным на периферийном устройстве. Прерывания могут запускаться также пользовательскими программами для вызова в подпрограммы ядра операционной системы. (Подробнее обсудим прерывания в подразделе 4.4.2.)
- *Другие типы инструкций*. Большинство ISA поддерживают различные типы команд, которые не попадают ни в одну из перечисленных категорий. Например, инструкция `no-op`, часто называемая `NOP`, — это команда, которая не оказывает никакого влияния, лишь вводит небольшую задержку. Инструкции `NOP` также занимают память, а на некоторых ISA используются для правильного выравнивания последующих инструкций в памяти.

Мы не можем перечислить здесь все типы команд, но, если интересно, вы всегда сможете прочитать документацию ISA для реального процессора, такого как Intel x86 (она доступна по адресу intel.ly/2woVFQ8).

Машинный язык

Компьютеры способны работать только с числами. Таким образом, каждая инструкция в потоке команд программы должна быть закодирована численно. Когда программа кодируется таким образом, мы говорим, что она написана на *машинном языке* (machine language, ML). Конечно, машинный язык — это не один язык, а действительно множество языков, по одному для каждого процессора/ISA.

Любая инструкция на машинном языке состоит из трех основных частей:

- *кода операции*, который сообщает процессору, какую операцию выполнить (сложение, вычитание, перемещение, прыжок и т. д.);
- нуля или более *операндов*, которые определяют входы и/или выходы инструкции;
- некоторого *поля опций*, определяющего такие вещи, как *режим адресации* инструкции и, возможно, другие флаги.

Операнды бывают разных типов. Некоторые инструкции могут принимать в качестве операндов закодированные как числовые идентификаторы имена одного или нескольких регистров. Другие могут ожидать литеральное значение, например «загрузить значение 5 в регистр R2» или «перейти к адресу 0x0102ED5C». Способ, которым операнды инструкции интерпретируются и используются ЦП, известен как *режим адресации* инструкции. (Мы обсудим режимы адресации более подробно далее.)

Код операции и операнды (если есть) инструкции ML упакованы в непрерывную последовательность битов, называемую *словом команды*. Гипотетический

процессор может кодировать свои инструкции (рис. 3.21), возможно, с первым байтом, содержащим код операции, режим адресации и флаги различных опций, за которыми следует некоторое количество байтов для операндов. Каждая архитектура по-разному определяет ширину слова инструкции, то есть количество битов, занимаемых последней. В некоторых ISA все инструкции занимают фиксированное количество битов, это типично для компьютеров *с сокращенным набором команд* (RISC). В других ISA различные типы команд могут быть закодированы в слова различного размера, это часто встречается в компьютерах со *сложным набором команд* (CISC).

Рис. 3.21. Две гипотетические схемы кодирования команд машинного языка. *Вверху*: в схеме кодирования с переменной шириной разные инструкции могут занимать разное количество байтов в памяти. *Внизу*: при кодировании инструкций с фиксированной шириной для каждой из них в потоке команд используется одинаковое количество байтов

В некоторых микроконтроллерах слова инструкций могут составлять всего 4 бита или иметь размер в несколько байтов. Часто они кратны 32 или 64 битам, потому что это соответствует ширине регистров ЦП и/или шины данных. В конструкциях ЦП с *очень длинным словом инструкции* (very long instruction word, VLIW) параллелизм достигается кодированием нескольких операций в одно очень широкое слово для их параллельного выполнения. Таким образом, инструкции в VLIW ISA могут иметь ширину более 100 байт.

Пример кодирования инструкций машинного языка в Intel x86 ISA приведен по адресу aturing.umcs.maine.edu/~meadow/courses/cos335/Asm07-MachineLanguage.pdf.

Язык ассемблера

Написание программы непосредственно на машинном языке утомительно и чревато ошибками. Чтобы облегчить жизнь программистам, была разработана простая текстовая версия машинного языка, называемая *языком ассемблера*. На ассемблере каждая инструкция в ISA данного ЦП получает *мнемонику* — короткое английское слово или аббревиатуру, которые гораздо легче запомнить, чем соответствующий

числовой код операции. Операнды инструкции также могут быть указаны удобным способом: можно ссылаться на регистры по имени (например, R0 или EAX), а адреса памяти записывать в шестнадцатеричном формате или назначать им символические имена, очень похожие на глобальные переменные языков более высокого уровня. Места в программе на ассемблере могут быть помечены удобочитаемыми метками, и инструкции перехода/ветвления ссылаются на эти метки, а не на необработанные адреса памяти.

Программа на языке ассемблера состоит из последовательности инструкций, каждая из которых включает в себя мнемонику и ноль или более операндов, перечисленных в текстовом файле с одной инструкцией на строку. Инструмент, известный как *ассемблер*, считывает исходный файл программы и преобразует его в числовое представление ML, понятное ЦП. Например, следующий фрагмент кода C:

```
if (a > b)
    return a + b;
else
    return 0;
```

может быть реализован на ассемблере и будет выглядеть примерно так:

```
; if (a > b)
cmp   eax, ebx   ; сравнение значений содержимого регистров
jle   ReturnZero ; переход, если меньше или равно

; return a + b;
add   eax, ebx   ; сложение и сохранение результата в EAX
ret              ; получение значения EAX и выход из программы

ReturnZero:
    ; else return 0;
    xor   eax, eax   ; установка значения EAX равным 0
    ret              ; получение значения EAX и выход из программы
```

Давайте разберем ее. Инструкция `cmp` сравнивает значения в регистрах EAX и EBX (которые, как мы предполагаем, содержат значения `a` и `b` из фрагмента кода C). Затем инструкция `jle` переходит к метке `ReturnZero`, но это происходит *тогда и только тогда*, когда значение в EAX меньше или равно EBX. Иначе она пропускается.

Если EAX больше, чем EBX (`a > b`), переходим к инструкции `add`, вычисляющей `a + b` и сохраняющей результат обратно в EAX, который, как мы предполагаем, служит возвращаемым значением. Мы выполняем команду `ret`, и управление возвращается вызывающей функции.

Если EAX меньше или равен EBX (`a <= b`), происходит переход в ветвь и мы продолжаем выполнение сразу после метки `ReturnZero`. Используем здесь небольшую хитрость, чтобы установить EAX в ноль, используя XOR с самим собой. Затем выполняем команду `ret`, чтобы вернуть этот ноль вызывающей стороне.

Дополнительные сведения о программировании на языке ассемблера Intel вы найдете по адресам www.cs.virginia.edu/~evans/cs216/guides/x86.html и http://av-assembler.ru/instructions/.

Режимы адресации

На первый взгляд простая инструкция типа `move`, которая передает данные между регистрами и памятью, имеет много разных вариантов. Перемещаем ли мы значение из одного регистра в другой? Загружаем ли мы литеральное значение, например 5, в регистр? Загружаем ли значение из памяти в регистр? Или храним значение в регистре в памяти? Все эти варианты называются *режимами адресации*. Мы не будем здесь рассматривать все возможные режимы адресации, но следующий список даст вам представление о режимах адресации, которые вы встретите на реальном процессоре.

- *Адресация регистра*. Этот режим позволяет передавать значения из одного регистра в другой. В этом случае операнды инструкции указывают, какие регистры участвуют в операции.
- *Непосредственная адресация*. Этот режим позволяет загружать литеральное или непосредственное значение в регистр. В этом случае операндами являются целевой регистр и непосредственное значение для загрузки.
- *Прямая адресация*. Этот режим позволяет перемещать данные в память или из нее. В этом случае операнды определяют направление движения (в память или из памяти) и необходимый адрес памяти.
- *Непрямая адресация регистров*. В этом режиме адрес целевой памяти берется из регистра, а не кодируется в виде литерального значения в операндах инструкции. Именно так реализуются разыменования указателей в таких языках, как C и C++. Значение указателя (адрес) загружается в регистр, а затем непрямая для регистра инструкция `move` используется для разыменования указателя и либо загрузки значения, на которое он указывает, в целевой регистр, либо сохранения значения в исходном регистре в этот адрес памяти.
- *Относительная адресация*. В данном режиме адрес целевой памяти указывается в качестве операнда, а значение, хранящееся в указанном регистре, используется в качестве смещения от этого адреса целевой памяти. Так реализован доступ к индексированным массивам на языке C или C ++.
- *Другие режимы адресации*. Существует множество других вариаций и комбинаций, некоторые общие практически для всех процессоров, а некоторые — специфичные для определенных процессоров.

Что еще почитать про язык ассемблера

В предыдущих разделах мы лишь немного опробовали язык ассемблера. Простое для понимания описание программирования на ассемблере x86 можно найти по адресу flint.cs.yale.edu/cs421/papers/x86-asm/asm.html. Подробнее о соглашениях о вызове и ABI — по адресу en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames.

3.5. Архитектура памяти

В простой компьютерной архитектуре фон Неймана память рассматривается как единый однородный блок, каждая из частей которого одинаково доступна для процессора. На самом деле память компьютера почти никогда не строится таким упрощенным образом. Конечно, регистры ЦП являются формой памяти, но обычно на них ссылаются по имени в программе на языке ассемблера, а не адресуются, как в обычных ПЗУ или ОЗУ. Более того, даже обычная память чаще всего разделяется на блоки с разными характеристиками и разными целями. Это выполняется по ряду причин, включая управление затратами и оптимизацию общей производительности системы. В этом разделе мы рассмотрим некоторые типы архитектуры памяти, которые обычно встречаются в современных персональных компьютерах и игровых консолях, а также поговорим о том, почему они спроектированы именно такими.

3.5.1. Сопоставление памяти

N-битная адресная шина предоставляет процессору доступ к *адресному пространству*, теоретически размером 2^n байт. Отдельное запоминающее устройство (ПЗУ или ОЗУ) всегда рассматривается как непрерывный блок ячеек памяти. Таким образом, адресное пространство компьютера обычно делится на непрерывные сегменты. Один или несколько этих сегментов соответствуют модулям памяти ПЗУ, другие — модулям ОЗУ. Например, в Apple II 16-разрядные адреса в диапазоне от 0xC100 до 0xFFFF были назначены микросхемам ПЗУ, содержащим прошивку компьютера, а адреса от 0x0000 до 0xBFFF — ОЗУ. Всякий раз, когда физическому устройству памяти назначается диапазон адресов в адресном пространстве компьютера, мы говорим, что диапазон адресов был *сопоставлен* с устройством памяти.

Конечно, на компьютере не обязательно установлено столько памяти, сколько теоретически может быть адресовано его адресной шиной; 64-битная адресная шина может получить доступ к 16 ЭиБ памяти, поэтому мы вряд ли когда-нибудь до предела заполним такое адресное пространство! (Имея 160 ТиБ, даже прототип суперкомпьютера HP под названием The Machine не слишком близок к этому объему физической памяти.) Поэтому некоторые сегменты адресного пространства компьютера обычно не назначаются.

Сопоставление памяти с вводом/выводом

Не все диапазоны адресов должны отображаться на устройства памяти — диапазон адресов может быть сопоставлен также с другими *периферийными устройствами*, такими как джойстик или сетевая карта. Этот подход называется *отображением в память ввода-вывода*, поскольку ЦП способен выполнять операции ввода-вывода на периферийном устройстве путем чтения или записи в адреса, как если бы это была обычная ОЗУ. При этом специальная схема обнаруживает, что ЦП выполняет чтение или запись в диапазон адресов, которые были сопоставлены с устройством без памяти, и преобразует запрос на чтение или запись в операцию ввода-вывода, направленную на соответствующее устройство. Приведу конкретный пример: Apple II

отобразил устройства ввода-вывода в диапазон адресов от 0xC000 до 0xC0FF. Тем самым программам было разрешено управлять оперативной памятью с переключением банков, считывать напряжение на выводах разъема игрового контроллера на материнской плате и управлять им, а также выполнять другие операции ввода-вывода, просто читая и записывая данные в названном диапазоне адресного пространства.

В качестве альтернативы ЦП может связываться с устройствами без памяти через специальные регистры, известные как *порты*. В этом случае всякий раз, когда ЦП запрашивает чтение или запись данных в регистр портов, аппаратное обеспечение преобразует запрос в операцию ввода-вывода на целевом устройстве. Этот подход называется *«связанный с портами ввод/вывод»*. В линейке микроконтроллеров Arduino привязка ввода/вывода к порту предоставляет программе возможность напрямую обращаться к цифровым входам и выходам определенных выводов микросхемы.

Видеопамять

Устройства отображения на основе растра обычно считывают аппаратный диапазон адресов физической памяти, чтобы определить яркость/цвет каждого пиксела на экране. Аналогично ранние дисплеи на основе символов определяли бы, какой символьный глиф отображать в конкретном месте на экране, читая коды ASCII из блока памяти. Диапазон адресов памяти, назначенных для использования видеоконтроллером, называется *видеопамятью* (VRAM).

В ранних компьютерах, таких как Apple II и первые ПК IBM, видеопамять соответствовала микросхемам памяти на материнской плате, а адреса памяти в VRAM могли считываться и записываться процессором точно так же, как и любая другая область памяти. Это относится и к игровым консолям, таким как PlayStation 4 и Xbox One, где CPU и GPU совместно используют доступ к одному большому блоку объединенной памяти.

В персональных компьютерах графический процессор часто живет на отдельной плате, подключенной к разъему расширения на материнской плате. Видеопамять обычно расположена на видеокарте, чтобы графический процессор мог получить к ней доступ как можно быстрее. Протокол шины, такой как PCI, AGP или PCI Express (PCIe), используется для передачи данных назад и вперед между основным RAM и VRAM через шину слота расширения. Такое физическое разделение между основной RAM и VRAM может стать существенным узким местом производительности и является одним из основных факторов, влияющих на сложность механизмов рендеринга и графических API, таких как OpenGL и DirectX 11.

Пример из практики: область памяти Apple II

Чтобы проиллюстрировать концепции сопоставления памяти, рассмотрим простой пример из реальной жизни. У Apple II была 16-битная адресная шина, что означало: его адресное пространство составляет всего 64 КиБ. Оно было сопоставлено с ПЗУ и RAM, отображаемыми в память устройств ввода-вывода, а также с областями видеопамяти следующим образом:

```
0xC100 - 0xFFFF  ROM (Firmware)
0xC000 - 0xC0FF  Memory-Mapped I/O
0x6000 - 0xBFFF  General-purpose RAM
0x4000 - 0x5FFF  High-res video RAM (page 2)
0x2000 - 0x3FFF  High-res video RAM (page 1)
0x0C00 - 0x1FFF  General-purpose RAM
0x0800 - 0x0BFF  Text/lo-res video RAM (page 2)
0x0400 - 0x07FF  Text/lo-res video RAM (page 1)
0x0200 - 0x03FF  General-purpose and reserved RAM
0x0100 - 0x01FF  Program stack
0x0000 - 0x00FF  Zero page (mostly reserved for DOS)
```

Отмечу, что адреса в карте памяти Apple II напрямую соответствовали чипам памяти на материнской плате. В современных операционных системах программы работают с *виртуальными адресами*, а не с физическими. Виртуальную память мы исследуем в следующем подразделе.

3.5.2. Виртуальная память

Большинство современных процессоров и операционных систем поддерживают функцию переназначения памяти, известную как *система виртуальной памяти*. В этих системах адреса памяти, используемые программой, не отображаются непосредственно на модули памяти, установленные на компьютере. Вместо этого всякий раз, когда программа читает или записывает адрес, он сначала *перераспределяется* ЦП через справочную таблицу, которую поддерживает ОС. Переназначенный адрес может в конечном итоге ссылаться на фактическую ячейку в памяти (с совершенно другим числовым адресом). Это способно привести к обращению к блоку данных на диске. Или окажется, что он вообще не сопоставлен с каким-либо физическим хранилищем. В системе виртуальной памяти адреса, используемые программами, называются *виртуальными адресами*, а битовые комбинации, которые фактически передаются по шине адреса контроллером памяти для доступа к модулю RAM или ROM, — *физическими адресами*.

Виртуальная память — это мощная концепция. Она позволяет программам использовать больше памяти, чем фактически установлено в компьютере, поскольку данные при переполнении могут перемещаться из физической оперативной памяти на диск. Виртуальная память повышает стабильность и безопасность операционной системы, поскольку каждая программа имеет собственный закрытый объем памяти и не вмешивается в блоки памяти, выделенные другим программам или самой операционной системе. Подробнее о том, как операционная система управляет пространствами виртуальной памяти запущенных программ, поговорим в подразделе 4.4.5.

Страницы виртуальной памяти

Чтобы понять, как происходит переназначение, нужно представить, что все адресуемое пространство памяти (это ячейки размером 2^n, если шина адреса имеет ширину n бит) концептуально разделено на непрерывные куски одинакового размера, называемые страницами. Размеры страниц различаются в разных операционных системах, но всегда кратны степени двойки — типичный размер страницы

составляет 4 или 8 КиБ. Если предположить, что размер страницы 4 КиБ, 32-разрядное адресное пространство будет разделено на 1 048 576 отдельных страниц, пронумерованных от 0x0 до 0xFFFFF (табл. 3.2).

Таблица 3.2. Разделение 32-битного адресного пространства на страницы объемом по 4 КиБ

От адреса	До адреса	Индекс страницы
0x00000000	0x00000FFF	Page 0x0
0x00001000	0x00001FFF	Page 0x1
0x00002000	0x00002FFF	Page 0x2
...		
0x7FFF2000	0x7FFF2FFF	Page 0x7FFF2
0x7FFF3000	0x7FFF3FFF	Page 0x7FFF3
...		
0xFFFFE000	0xFFFFEFFF	Page 0xFFFFE
0xFFFFF000	0xFFFFFFFF	Page 0xFFFFF

Сопоставление виртуальных и физических адресов всегда выполняется на уровне детализации страниц. Одно такое гипотетическое сопоставление показано на рис. 3.22.

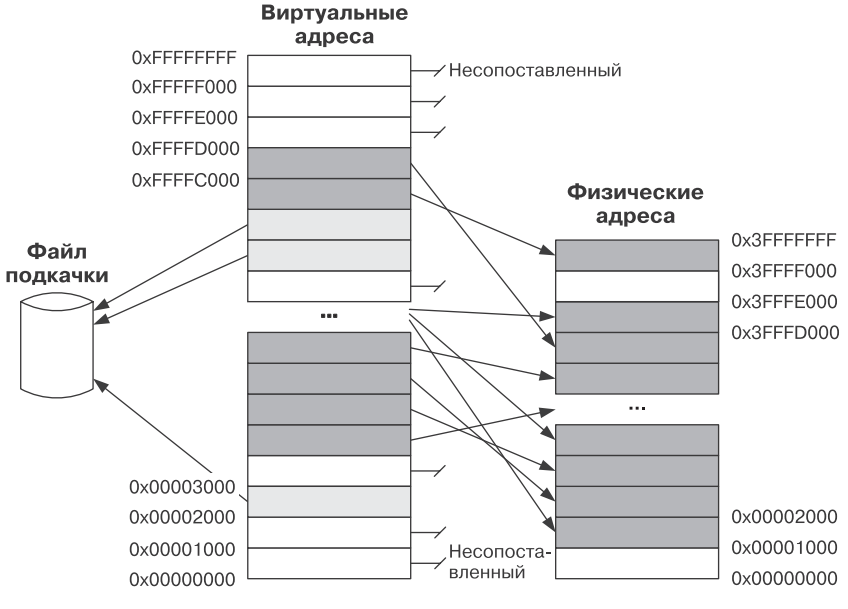


Рис. 3.22. Диапазоны адресов размера страницы в пространстве виртуальной памяти либо сопоставляются со страницами физической памяти или файлом подкачки на диске, либо не сопоставляются

Перевод виртуальных адресов в физические

Всякий раз, когда ЦП обнаруживает операцию чтения или записи в память, адрес разделяется на две части: *индекс страницы* и *смещение* на этой странице (измеряется в байтах). Для страницы размером 4 КиБ смещение составляет 12 младших бит адреса, а индекс страницы получается из оставшихся верхних 20 бит, которые затем маскируются и сдвигаются вправо на 12 бит. Например, виртуальный адрес 0x1A7C6310 соответствует смещению 0x310 и индексу страницы 0x1A7C6.

Индекс страницы затем ищется *модулем управления памятью* (memory management unit, MMU) ЦП в *таблице страниц*, которая отображает индексы виртуальных страниц на физические. (Таблица страниц хранится в ОЗУ и управляется операционной системой.) Если рассматриваемая страница отображается на страницу физической памяти, индекс виртуальной страницы преобразуется в соответствующий индекс физической страницы: биты ее индекса смещаются влево, к ним применяется логическое ИЛИ с битами исходного смещения страницы — и вуаля! Мы получили физический адрес. Итак, продолжая рассматривать пример: если виртуальная страница 0x1A7C6 отображается на физическую страницу 0x73BB9, то преобразованный физический адрес в конечном итоге будет 0x73BB9310. Это адрес, который фактически будет передаваться по адресной шине. Работу MMU иллюстрирует рис. 3.23.

Если таблица страниц указывает, что страница не сопоставлена с физической RAM (потому что она либо никогда не выделялась, либо с тех пор была выгружена в файл на диске), MMU вызывает *прерывание*, которое сообщает операционной системе, что запрос памяти не может быть выполнен. Это называется *ошибкой страницы*. (Подробнее о прерываниях поговорим в подразделе 4.4.2.)

Обработка ошибок страниц

При попытке доступа к нераспределенным страницам памяти ОС обычно реагирует на ошибку, вылетая из программы и генерируя дамп ядра. При доступе же к страницам, которые были выгружены на диск в файл подкачки (swap), ОС временно приостанавливает работающую в данный момент программу, считывает страницу из файла подкачки в физическую страницу ОЗУ, а затем, как обычно, преобразует виртуальный адрес в физический. Наконец, она возвращает управление приостановленной программе. С точки зрения программы эта операция является полностью бесшовной — она никогда не знает, была ли страница уже в памяти, или ее нужно было выгружать с диска.

Обычно страницы выгружаются на диск только тогда, когда нагрузка на систему памяти высока, а физических страниц не хватает. Операционная система пытается выгрузить только наиболее редко используемые страницы памяти, чтобы программы не метались между страницами памяти в ОЗУ и на диске.

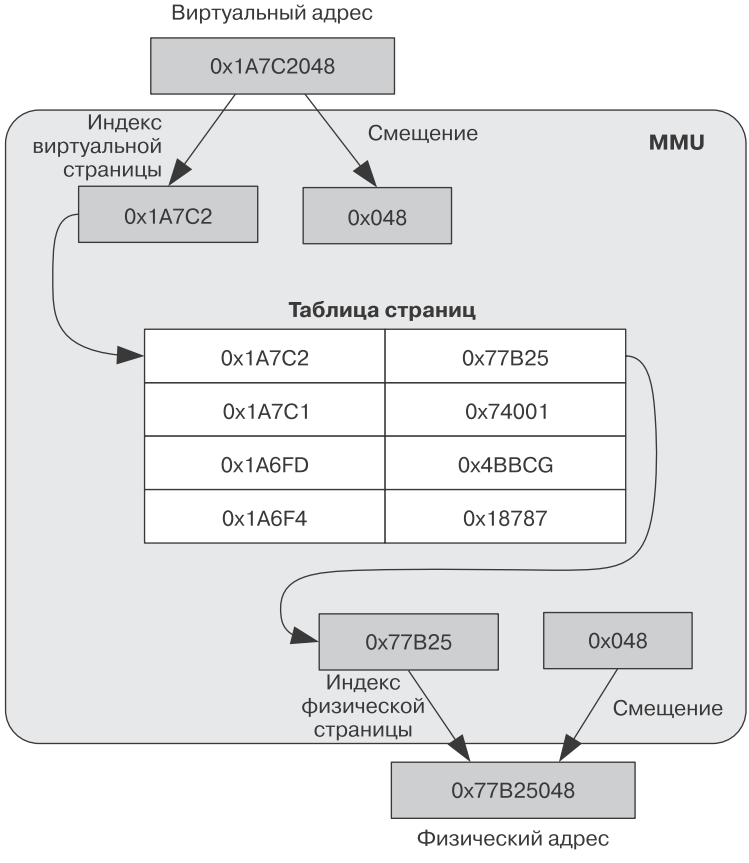


Рис. 3.23. MMU перехватывает операцию чтения из памяти и разбивает виртуальный адрес на индекс виртуальной страницы и смещение. Индекс виртуальной страницы преобразуется в индекс физической страницы с помощью таблицы страниц, а физический адрес составляется из индекса физической страницы и исходного смещения. Наконец, инструкция выполняется с использованием переназначенного физического адреса

Буфер ассоциативной трансляции

Поскольку размеры страниц обычно невелики по отношению к общему объему адресуемой памяти (обычно 4 или 8 КиБ), таблица страниц может стать очень большой. Поиск физических адресов в таблице страниц занял бы много времени, если бы всю таблицу приходилось сканировать для каждого обращения к памяти, которое делает программа.

Для ускорения доступа применяется механизм кэширования, основанный на предположении, что среднестатистическая программа будет стремиться повторно использовать адреса на относительно небольшом количестве страниц, а не читать и писать случайным образом по всему диапазону адресов. Небольшая таблица, известная как *буфер ассоциативной трансляции* (translation lookaside buffer, TLB),

поддерживается в MMU на кристалле ЦП, в котором кэшируются сопоставления последних использованных физических и виртуальных адресов. Поскольку этот буфер расположен в непосредственной близости от MMU, доступ к нему осуществляется очень быстро.

TLB действует во многом как *иерархия кэша памяти* общего назначения, за исключением того, что он используется только для кэширования записей таблицы страниц. В подразделе 3.5.4 описано, как работают иерархии кэша.

Что еще почитать о виртуальной памяти

На сайтах www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/virtual.html и gabrieletolomei.wordpress.com/miscellanea/operating-systems/virtual-memory-paging-and-swap приведены детали реализации виртуальной памяти.

Также рекомендую к прочтению всем программистам статью Ульриха Дреппера *What Every Programmer Should Know About Memory* («Что каждый программист должен знать о памяти»): www.akkadia.org/drepper/cpumemory.pdf.

3.5.3. Архитектуры памяти для уменьшения задержки

Скорость, с которой можно получить доступ к данным на запоминающем устройстве, — важная характеристика. Мы часто говорим о *задержке доступа к памяти*, которая определяется как промежуток времени между моментом, когда ЦП запрашивает данные из системы памяти, и моментом, когда эти данные фактически принимаются ЦП. Задержка доступа к памяти в первую очередь зависит от трех факторов:

- технологии, используемой для реализации отдельных ячеек памяти;
- количества портов чтения и/или записи, поддерживаемых памятью;
- физического расстояния между этими ячейками памяти и ядром процессора, которое их использует.

Задержка доступа к данным у *статического* ОЗУ (SRAM) обычно намного ниже, чем у *динамического* ОЗУ (DRAM). Высокое быстродействие у SRAM достигается применением более сложной конструкции, которая требует использования большего количества транзисторов на бит памяти, чем в DRAM. Это делает SRAM более дорогой, чем DRAM, с точки зрения как финансовых затрат на бит, так и места, занимаемого на кристалле.

Самая простая ячейка памяти имеет один порт, поэтому в любой момент способна выполнить только одну операцию чтения или записи. *Многопортовая* RAM позволяет одновременно выполнять несколько операций чтения и/или записи, тем самым уменьшая задержку, вызванную конфликтом, когда несколько ядер или несколько компонентов в одном ядре пытаются одновременно получить доступ к памяти. Ожидаемо многопортовая RAM требует больше транзисторов на бит, чем однопортовая конструкция, следовательно, стоит дороже и использует больше ресурсов на кристалле, чем однопортовая память.

Физическое расстояние между процессором и областью RAM также значимо при определении задержки доступа к этой памяти. Это связано с тем, что электронные сигналы внутри компьютера распространяются с конечной скоростью. Электронные сигналы состоят из электромагнитных волн и, следовательно, распространяются со скоростью, близкой[1] к скорости света. Дополнительные задержки вводятся различными коммутирующими и логическими схемами, с которыми сигнал доступа к памяти встречается на своем пути через систему. Таким образом, чем ближе ячейка памяти к ядру ЦП, которое ее использует, тем меньше будет задержка доступа.

Разрыв памяти

На заре вычислительной техники задержка доступа к памяти и задержка выполнения команд находились примерно на одном уровне. Например, в Intel 8086 инструкции на основе регистров могли выполняться за 2–4 цикла, доступ к основной памяти также занимал примерно четыре цикла. Однако за последние несколько десятилетий как сами тактовые частоты, так и эффективная пропускная способность процессоров увеличились намного сильнее, чем скорость доступа к памяти. Сегодня задержка доступа к основной памяти чрезвычайно высока по сравнению со временем выполнения одной инструкции. В то время как инструкции на основе регистров для Intel Core i7 по-прежнему занимают от одного до десяти циклов, доступ к основной ОЗУ может потребовать около 500 циклов! Это постоянно увеличивающееся несоответствие между скоростями процессора и задержками доступа к памяти часто называют *разрывом памяти*. Тенденция постоянного увеличения разрыва памяти со временем показана на рис. 3.24.

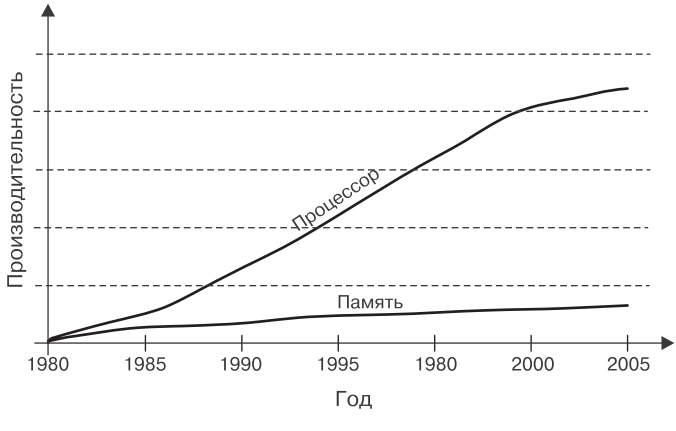


Рис. 3.24. Постоянно увеличивающаяся разница между производительностью процессора и производительностью памяти называется разрывом памяти [23]

[1] Скорость, с которой электронный сигнал проходит в среде передачи, такой как медный провод или оптоволокно, всегда несколько ниже скорости света в вакууме. Каждый соединительный материал имеет собственный коэффициент скорости (velocity factor, VF), варьирующийся от менее чем 50 до 99 % скорости света в вакууме.

Программисты и разработчики оборудования вместе создали широкий спектр методов для решения проблем, вызванных высокой задержкой доступа к памяти. Эти методы обычно фокусируются на одном или нескольких таких вариантах, как:

- *уменьшение* средней задержки памяти за счет размещения небольших, но быстрых областей памяти ближе к ядру ЦП, чтобы ускорить доступ к наиболее часто используемым данным;
- *скрытие* задержки доступа к памяти путем нагрузки ЦП другой полезной работой в ожидании завершения операции с памятью;
- *сведение к минимуму* доступа к основной памяти путем организации данных программы как можно более эффективным способом в отношении работы, которая должна быть выполнена с этими данными.

В этом подразделе мы более подробно рассмотрим архитектуры памяти, способные уменьшить среднюю задержку. Два других метода — скрытие задержки и минимизацию доступа к памяти посредством продуманной компоновки данных — обсудим при изучении параллельного проектирования аппаратных средств и методов параллельного программирования в главе 4.

Регистровые файлы

Регистровый файл ЦП является, пожалуй, самым ярким примером архитектуры памяти, разработанной для минимизации задержки доступа. Он обычно реализуется с использованием многопортовой статической памяти (SRAM), как правило, с выделенными портами для операций чтения и записи, что позволяет выполнять эти операции параллельно, а не последовательно. Более того, регистровый файл обычно расположен рядом с использующим его ALU. Кроме того, ALU обращается к этим регистрам практически напрямую, тогда как доступ к основной RAM обычно проходит через систему трансляции виртуальных адресов, иерархию кэша памяти и протокол согласованности кэша (см. подраздел 3.5.4), адресную шину, шину данных и, возможно, логику коммутации. Эти факты объясняют, почему доступ к регистрам можно получить так быстро и почему стоимость такой памяти довольно высока по сравнению RAM общего назначения. Более высокая стоимость оправданна, потому что регистры являются наиболее часто используемой памятью на любом компьютере, а также потому, что общий размер файла регистров очень мал по сравнению с размером основной оперативной памяти.

3.5.4. Иерархии кэш-памяти

Иерархия кэш-памяти является одним из основных механизмов смягчения воздействия высоких задержек доступа к памяти на современных персональных компьютерах и игровых приставках. В иерархии кэша небольшая, но быстрая область оперативной памяти, называемая кэшем *первого уровня* (L1), расположена очень близко к ядру процессора (на том же кристалле). Его задержка доступа почти такая же низкая, как и у регистрового файла ЦП, потому что он находится так

близко к ядру. Некоторые системы предоставляют больший, но несколько более медленный кэш *второго уровня* (L2), расположенный дальше от ядра (обычно также на кристалле) и часто совместно используемый двумя или более ядрами в многоядерном процессоре. Некоторые машины даже имеют большую, но более удаленную кэш-память L3 или L4. Эти кэши работают согласованно, чтобы автоматически сохранять копии наиболее часто используемых данных, так что доступ к очень большой, но очень медленной области основной оперативной памяти, расположенной на системной плате, сводится к минимуму.

Система кэширования повышает производительность доступа к памяти, сохраняя в кэше *локальные копии* тех фрагментов данных, к которым программа чаще всего обращается. Если данные, запрашиваемые ЦП, уже находятся в кэше, они могут быть переданы ЦП очень быстро — примерно за десятки циклов. Это называется *попаданием в кэш (cache hit)*. Если данных еще нет в кэше, они должны быть извлечены в него из основной оперативной памяти. Это называется *промахом кэша (cache miss)*. Чтение данных из основного RAM может занимать сотни циклов, поэтому стоимость промаха кэша очень высока.

Строки кэша

В кэшировании памяти используется то, что шаблоны доступа к памяти в реальном программном обеспечении, как правило, используют два вида *локальности ссылок*.

1. *Пространственная локальность*. Если программа обращается к адресу памяти *N*, вполне вероятно, что соседние адреса, такие как $N + 1$, $N + 2$ и т. д., также будут скоро использоваться. Последовательная итерация по данным, хранящимся в массиве, — это пример схемы доступа к памяти с высокой степенью пространственной локальности.
2. *Временна́я локальность*. Если программа обращается к адресу памяти *N*, вполне вероятно, что этот же адрес снова будет применен в ближайшем будущем. Считывание данных из переменной или структуры данных, выполнение преобразования для них, а затем запись обновленного результата в ту же переменную или структуру данных — это пример схемы доступа к памяти с высокой степенью временно́й локальности.

Чтобы воспользоваться преимуществами локальности ссылок, системы кэширования памяти перемещают данные в кэш в виде непрерывных блоков, называемых *строками кэша*, а не кэшируют элементы данных по отдельности.

Например, предположим, что программа обращается к членам данных экземпляра класса или структуры. Когда читается первый элемент, контроллеру памяти могут потребоваться сотни циклов, чтобы добраться до основной оперативной памяти и извлечь из нее данные. Однако контроллер кэша не просто читает один этот элемент — он фактически читает больший непрерывный блок оперативной памяти в кэш. Таким образом, последующие операции чтения других элементов данных экземпляра, вероятно, станут дешевыми обращениями в кэш.

Сопоставление строк кэша с адресами основной оперативной памяти

Существует простое соответствие «один ко многим» между адресами памяти в кэше и в основной оперативной памяти. Мы можем рассматривать адресное пространство кэша как отображаемое в основное адресное пространство RAM в виде повторяющегося шаблона, начиная с адреса 0 в основной RAM и продолжая до тех пор, пока все основные адреса не будут охвачены кэшем.

В качестве примера примем, что кэш имеет размер 32 Кбайт, а каждая его строка занимает 128 байт. Поэтому кэш может содержать 256 строк кэша ($256 \cdot 128 = 32\,768$ байт = 32 КиБ). Далее предположим, что объем оперативной памяти составляет 256 Мбайт. Таким образом, основная оперативная память в 8192 раза больше кэша, потому что $(256 \cdot 1024) / 32 = 8192$. Это означает, что нужно наложить адресное пространство кэша на адресное пространство основной RAM 8192 раза, чтобы охватить все возможные области физической памяти. Или, другими словами, одна строка в кэше сопоставляется с 8192 отдельными фрагментами основной оперативной памяти размером со строку.

Если у нас есть любой адрес в основной памяти, мы можем найти его адрес в кэше, взяв основной адрес памяти по модулю размера кэша. Таким образом, для кэша емкостью 32 Кбайт и основной оперативной памяти 256 Мбайт адреса кэша от 0x0000 до 0x7FFF (то есть 32 КиБ) сопоставляются с адресами основной памяти от 0x0000 до 0x7FFF. Но этот диапазон адресов кэша также отображается на адреса основной оперативной памяти от 0x8000 до 0xFFFF, от 0x10000 до 0x17FFF, от 0x18000 до 0x1FFFF и т. д. вплоть до последнего основного блока памяти, расположенного по адресам от 0xFFF8000 до 0xFFFFFFF. Сопоставление между основной и кэш-памятью иллюстрирует рис. 3.25.

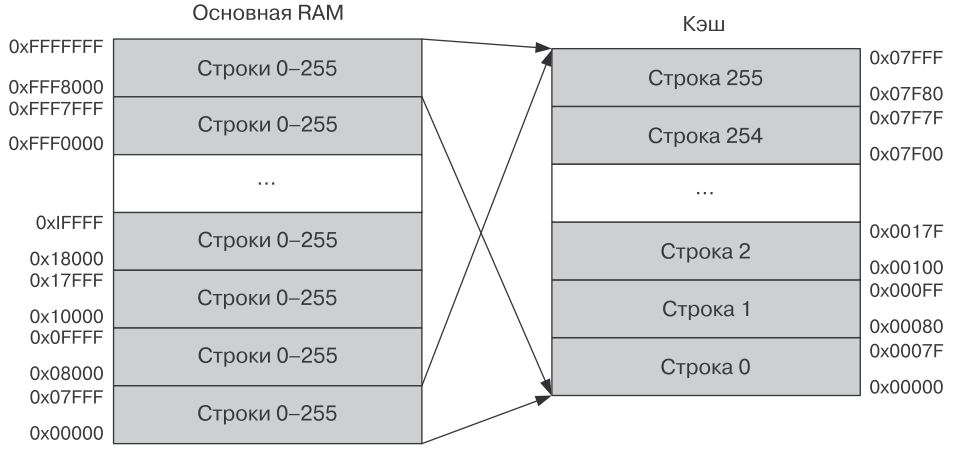


Рис. 3.25. Прямое сопоставление между адресами основной памяти и строками кэша

Адресация кэша

Рассмотрим, что происходит, когда процессор читает одиночный байт из памяти. Сначала адрес желаемого байта в основном ОЗУ преобразуется в адрес в кэше. Затем контроллер кэша проверяет, существует ли в нем строка, содержащая этот байт. Случай совпадения называется *попаданием в кэш*, при этом байт читается из кэша, а не из основной оперативной памяти. Обратный случай называется *промахом кэша*, и порция данных размером в строку будет считана из основного ОЗУ и загружена в кэш, так что в дальнейшем соседние адреса будут читаться быстро.

Кэш может работать только с адресами памяти, которые *выровнены* по кратному размеру строки кэша (выравнивание памяти обсуждалось в подразделе 3.3.7). Другими словами, к нему можно обратиться только в строках, а не в байтах. Следовательно, нужно преобразовать адрес байта в *индекс строки кэша*.

Рассмотрим кэш размером 2^M байт, содержащий строки размером 2^n. Эти n наименее значимых бит адреса основной RAM представляют *смещение* байта в строке кэша. Мы отбрасываем их для преобразования из единиц байтов в единицы строк, то есть делим адрес на размер строки кэша, равный 2^n. Наконец, мы разбиваем полученный адрес на две части: наименее значимые биты $(M - n)$ становятся *индексом строки кэша*, а все оставшиеся биты говорят, из *какого* блока (размера кэша) в основной RAM поступила строка кэша. Индекс блока известен как *тег*.

В случае промаха кэша контроллер кэша загружает кусок данных размером со строку из основного ОЗУ в соответствующую строку в кэше. Кэш содержит также небольшую таблицу тегов, по одному для каждой строки кэша. Это позволяет системе кэширования отслеживать, из *какого* основного блока ОЗУ поступила каждая строка в кэше. Это необходимо из-за отношения «многие к одному» между адресами памяти в кэше и в основной оперативной памяти. На рис. 3.26 показано, как кэш связывает тег с каждой активной строкой в нем.

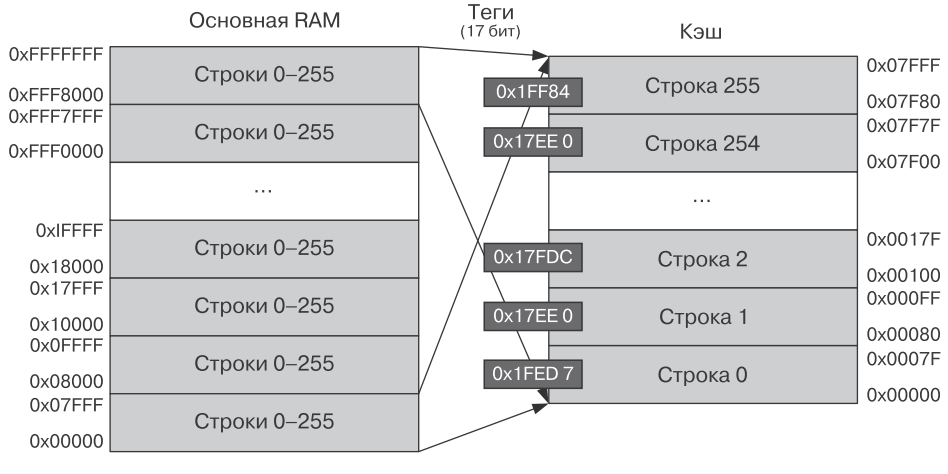


Рис. 3.26. С каждой строкой в кэше связан тег, указывающий, из какого блока размером с кэш основной оперативной памяти пришла соответствующая строка

Возвратимся к примеру чтения байта из оперативной памяти. Целиком последовательность событий выглядит следующим образом. Процессор вызывает операцию чтения. Основной адрес RAM преобразуется в смещение, индекс строки и тег. С помощью индекса строки в кэше выполняется проверка, для того чтобы найти соответствующий тег. Если тег в кэше соответствует запрошенному тегу, это попадание в кэш. В этом случае индекс строки используется для извлечения фрагмента данных размера строки из кэша, а смещение — для определения местоположения нужного байта в строке. Если теги не совпадают, значит, произошел промах кэша. В этом случае соответствующий кусок основной оперативной памяти считывается в кэш, а соответствующий тег сохраняется в таблице тегов кэша. Последующее чтение соседних адресов (тех, которые находятся в одной и той же строке кэша) приведет к гораздо более быстрым попаданиям в кэш.

Ассоциативность кэша и политика замены

Простое сопоставление строк кэша и адресов основной оперативной памяти, описанное ранее, известно как кэш *с прямым отображением*. Это означает, что каждый адрес в основной памяти соответствует только *одной* строке в кэше. При использовании в качестве примера кэш-памяти объемом 32 Кбайт со 128-байтовыми строками адрес основной памяти 0x203 отображается на четвертую строку кэша (потому что 0x203 — это 515, а 515 / 128 = 4). Однако в нашем примере есть 8192 уникальных блока размером со строку кэша основной оперативной памяти, которые отображаются на четвертую строку кэша. В частности, четвертая строка кэша соответствует адресам основной оперативной памяти от 0x200 до 0x27F, от 0x8200 до 0x827F и от 0x10200 до 0x1027f, а также *другим* 8189 диапазонам адресов размером, равным строке кэша.

При возникновении ошибки кэша ЦП должен загрузить соответствующую строку кэша из основной памяти в кэш. Если строка в кэше не содержит действительных данных, мы просто копируем данные в него и на этом заканчиваем. Но если она уже содержит данные из другого блока основной памяти, мы должны перезаписать их. Это называется *удалением* строки кэша.

Проблема с кэшем с прямым отображением состоит в том, что его использование может привести к патологическим случаям. Например, два несвязанных блока основной памяти могут циклично выселять друг друга. Мы сможем добиться лучшей производительности, если каждый адрес основной памяти будет отображаться в две и более отдельные строки кэша. В *двухканальном ассоциативном кэше* каждый основной адрес RAM отображается в две строки кэша (рис. 3.27). Очевидно, что ассоциативный кэш с четырьмя каналами работает даже лучше, чем двухканальный, а кэш с 8 или 16 каналами может превзойти четырехканальный и т. д.

Когда у нас есть более одного пути кэширования, контроллер кэша сталкивается с дилеммой: при промахе кэша какой из путей следует выселить, а какие оставить в кэше? Ответ на этот вопрос различается в зависимости от конструкции ЦП и известен как *политика замены*. Популярна политика *«не используется в последнее время»* (not most-recently used, NMRU). В этой схеме отслеживается последний

использованный путь, а выселение всегда влияет на путь или пути, которые не применяются в последнее время. Другие политики: *«первым пришел — первым ушел»* (first in first out, FIFO), которая является единственным вариантом в кэше с прямым отображением, *«давно неиспользуемый»* (least-recently used, LRU), *«наименее часто используемый»* (least-frequently used, LFU) и *«псевдослучайный»*. Для получения дополнительной информации о политиках замены кэша см. ece752.ece.wisc.edu/lect11-cache-replacement.pdf.

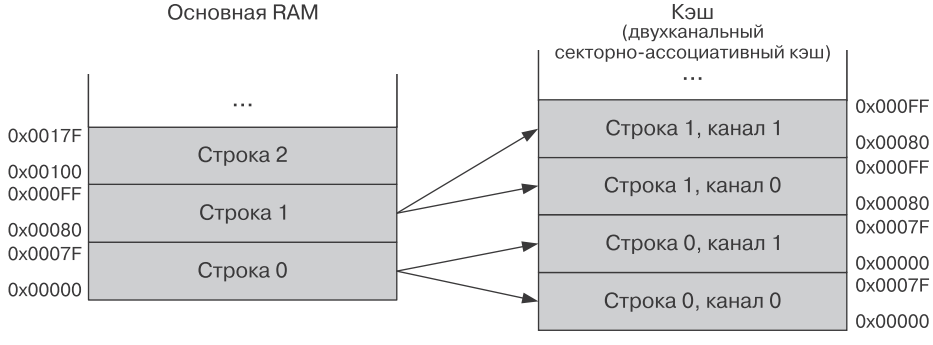


Рис. 3.27. Двухканальный секторно-ассоциативный кэш

Многоуровневые кэши

Частота попаданий — это мера того, как часто программа находит нужные данные в кэше, противоположность высокой стоимости пропуска кэша. Чем выше рейтинг попаданий, тем лучше будет работать программа при прочих равных условиях. Существует фундаментальный компромисс между задержкой кэша и частотой попаданий. Чем больше кэш, тем выше частота попаданий, но более крупные кэши не могут быть расположены ближе к процессору, поэтому они, как правило, медленнее, чем маленькие.

Большинство игровых приставок используют минимум два *уровня* кэша. Сначала процессор пытается найти данные в кэше *уровня 1* (L1) — небольшом и с очень низкой задержкой доступа. Если нужных данных там нет, он обращается в кэш *уровня 2* (L2), который работает с большей задержкой. Только если данные не обнаруживаются в кэше L2, происходит более затратное обращение к основной памяти. Поскольку задержка основного ОЗУ может быть слишком высокой по сравнению с тактовой частотой процессора, некоторые ПК даже включают кэш-память *уровня 3* (L3) и *уровня 4* (L4).

Кэш инструкций и кэш данных

При написании высокопроизводительного кода для игрового движка или любой другой системы, критической с точки зрения производительности, важно понимать, что и данные, и код кэшируются. *Кэш инструкций* (I-кэш, часто обозначаемый I$) используется для предварительной загрузки исполняемого машинного кода

перед его выполнением, тогда как *кэш данных* (D-кэш, или D$) применяется для ускорения операций чтения и записи, выполняемых этим машинных кодом. В кэше уровня 1 (L1) эти два кэша всегда физически различны, поскольку нежелательно, чтобы чтение инструкции вызывало удаление актуальных данных из кэша или наоборот. Поэтому при оптимизации кода мы должны учитывать производительность как D-, так и I-кэша (хотя, как мы увидим, оптимизация одного из них имеет положительный эффект для другого). Кэши более высокого уровня (L2, L3, L4), как правило, не делают различия между кодом и данными, поскольку их больший размер смягчает проблемы, вызванные данными, замещающими код, или кодом, замещающим данные.

Политика записи

Мы еще не говорили о том, что происходит, когда процессор *записывает* данные в оперативную память. То, как контроллер кэша обрабатывает запись, называется его *политикой записи*. Самый простой вид кэша — *сквозной кэш*, в этой относительно простой архитектуре кэша все записи в него немедленно отражаются в основной оперативной памяти. В конструкции кэша *с обратной записью* (или *с обратной копией*) данные сначала записываются в кэш, и строка кэша выгружается в основную память только при определенных обстоятельствах, например, когда необходимо удалить строку кэша с данными, чтобы считать новую строку из основной RAM, или когда программа явно запрашивает сброс.

Целостность данных в кэш-памяти: MESI, MOESI и MESIF

Когда несколько ядер ЦП одновременно используют одно хранилище основной памяти, все усложняется. Для каждого ядра характерно наличие собственного кэша L1, но несколько ядер могут совместно задействовать кэш L2 и основную оперативную память. На рис. 3.28 показана двухуровневая архитектура кэша с двумя ядрами процессора, совместно использующими одно хранилище основной памяти и кэш-память уровня 2.

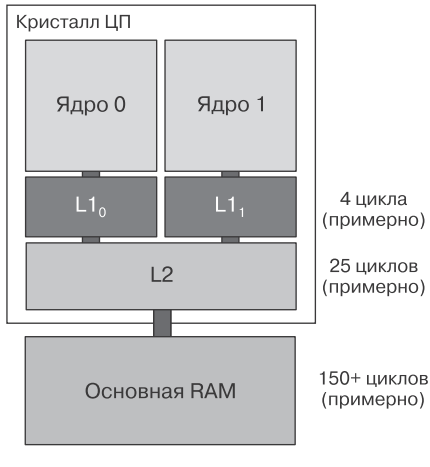


Рис. 3.28. Кэш уровней 1 и 2

При наличии нескольких ядер для системы важно поддерживать *когерентность кэша*. Это означает, что данные в кэшах, относящиеся к нескольким ядрам, должны соответствовать друг другу и содержимому основной RAM. Когерентность не должна поддерживаться в каждый момент времени, значимо лишь то, чтобы работающая программа никогда не смогла *определить*, что содержимое кэшей не синхронизировано.

За поддержание корректности данных между всеми кэшами в системе отвечают протоколы поддержки когерентности. Они описывают состояния, в одном из которых находится каждая строка кэша. Три наиболее распространенных протокола известны как MESI (modified, exclusive, shared, invalid — «модифицировано, владелец, разделяемое, неактуальное»), MOESI (modified, owned, exclusive, shared, invalid — «модифицировано, владелец, эксклюзивное, разделяемое, неактуальное») и MESIF (modified, exclusive, shared, invalid and forward — «модифицировано, эксклюзивное, разделяемое, неактуальное и направленное»). Мы обсудим протокол MESI более подробно, когда будем говорить о многоядерных вычислительных архитектурах в главе 4.

Предотвращение промахов кэша

Очевидно, что нельзя полностью избежать промахов кэш-памяти, поскольку в итоге данные должны перемещаться в оперативную память и из нее. Хитрость написания высокопроизводительного программного обеспечения при наличии иерархии кэш-памяти состоит в том, чтобы упорядочить данные в RAM и спроектировать алгоритмы обработки данных таким образом, чтобы происходило минимальное количество пропусков кэша.

Лучший способ избежать промахов D-кэша — организовать данные в виде *непрерывных* блоков как можно *меньшего* размера и затем обращаться к ним *последовательно*. Когда данные непрерывны (то есть вы не слишком много прыгаете по памяти), один промах кэша будет загружать максимальное количество релевантных данных за раз. Когда данные невелики, более вероятно, что они поместятся в одной строке кэша или по крайней мере в минимальном количестве строк. При последовательном доступе к данным вы избегаете многократного удаления и перезагрузки строк кэша.

Избегание пропусков I-кэша соответствует тому же основному принципу, что и предотвращение пропусков D-кэша. Однако реализация требует другого подхода. Самое простое, что нужно сделать, — как можно сильнее уменьшить высокопроизводительные циклы с точки зрения размера кода и избегать вызова функций внутри вложенных циклов. Если вы решили вызывать функции, постарайтесь сохранить их код небольшим. Это помогает гарантировать, что все тело цикла, включая все вызываемые функции, будет оставаться в I-кэше все время работы цикла.

Используйте встроенные функции разумно. Встраивание небольшой функции может значительно повысить производительность. Тем не менее слишком большое встраивание раздувает размер кода, в результате чего критически важная его часть может не поместиться в кэш.

3.5.5. Неоднородный доступ к памяти

При проектировании многопроцессорной игровой консоли или персонального компьютера системные архитекторы должны выбирать между двумя принципиально разными архитектурами памяти — *однородным доступом к памяти* (uniform

memory access, UMA) и *неоднородным доступом к памяти* (nonuniform memory access, NUMA).

В конструкции UMA компьютер содержит один большой блок основной оперативной памяти, который виден всем ядрам процессора в системе. Физическое адресное пространство выглядит одинаково для каждого ядра, и все они могут выполнять чтение и запись в любой области основной памяти. Архитектура UMA обычно использует иерархию кэша для уменьшения последствий проблем задержки доступа к памяти.

Однако проблема с архитектурой UMA состоит в том, что ядра часто борются за доступ к основной оперативной памяти и любым совместно используемым кэшам. Например, PS4 содержит восемь ядер, сгруппированных в два кластера. Каждое ядро имеет собственный кэш L1, но каждый кластер из четырех ядер разделяет один кэш L2, а все ядра разделяют основную оперативную память. Таким образом, ядра часто конкурируют друг с другом за доступ к кэш-памяти L2 и основной оперативной памяти.

Один из способов решения проблем, связанных с конфликтами, заключается в использовании структуры *с неоднородным доступом к памяти*. В системе NUMA каждое ядро снабжено относительно небольшим блоком высокоскоростной выделенной оперативной памяти, называемой *локальным хранилищем*. Как и кэш L1, локальное хранилище обычно находится на том же кристалле, что и само ядро, и доступно только этому ядру. Но, в отличие от доступа к кэшу L1, доступ к локальному хранилищу является явным. Локальное хранилище может быть сопоставлено с частью адресного пространства ядра, а основная RAM — с другим диапазоном адресов. Как альтернатива, некоторые ядра способны видеть *только* физические адреса в своем локальном хранилище и полагаться на *контроллер прямого доступа к памяти* (direct memory access controller, DMAC) для передачи данных между локальным хранилищем и основной оперативной памятью.

Локальные хранилища SPU на PS3

PlayStation 3 — классический пример архитектуры NUMA. Консоль PS3 содержит одно основное ядро, основанное на архитектуре IBM POWER и называемое PPU (POWER processing unit), восемь[1] ядер, известных как SPU (synergistic processing units), и графический процессор NVIDIA RSX (graphics processing unit, GPU). PPU имеет эксклюзивный доступ к 256 МиБ RAM основной системы (с кэш-памятью L1 и L2), графический процессор получает эксклюзивный доступ к 256 МиБ видеопамяти RAM (VRAM), а каждый SPU имеет собственное частное локальное хранилище объемом 256 КиБ.

Физические адресные пространства основной RAM, видеопамяти и локальных хранилищ SPU полностью изолированы друг от друга. Это означает, что, например,

[1] Только шесть SPU доступны для использования в игровых приложениях — один SPU зарезервирован для операционной системы, а применение последнего запрещено — он нужен в случае неизбежного заводского брака.

PPU не может напрямую обращаться к памяти в VRAM или любом из локальных хранилищ SPU и любой SPU не может напрямую обращаться к основной оперативной памяти, видеопамяти RAM или любому из локальных хранилищ других SPU — только к своему собственному локальному хранилищу. Архитектура памяти PS3 показана на рис. 3.29.

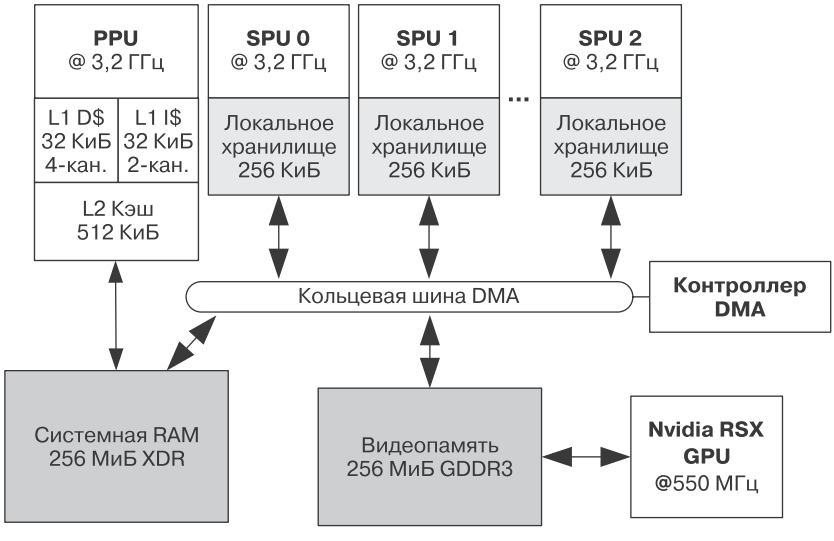


Рис. 3.29. Упрощенное представление сотовой широкополосной архитектуры PS3

Дополнительная память PS2

Оглядываясь еще дальше назад, на PlayStation 2, мы можем узнать о другой архитектуре памяти, которая была разработана для улучшения общей производительности системы. Основной ЦП PS2 с названием *Emotion Engine* (EE) имеет специальную область памяти объемом 16 КиБ, называемую *scratchpad RAM* (SPR), в дополнение к кэшу L1 (16 КиБ) для I-кэша и L1 (8 КиБ) для данных (D-кэш). PS2 содержит также векторные сопроцессоры VU0 и VU1, каждый с собственными I- и D-кэшами L1, и графический процессор Graphics Synthesizer (GS), подключенный к блоку видеопамяти RAM объемом 4 МиБ. Архитектура памяти PS2 показана на рис. 3.30.

Scratchpad RAM находится на кристалле ЦП, поэтому имеет такую же низкую задержку, что и кэш-память L1. Но, в отличие от кэша L1, она отображается в памяти, поэтому программист представляет, что он работает с диапазоном обычных адресов основной памяти. Scratchpad на PS2 сама по себе не кэширована. Это означает, что чтение с нее и запись на нее осуществляются напрямую, при этом кэш L1 процессора совершенно не задействуется.

Основным преимуществом scratchpad (можно перевести как «блокнот для черновых записей») является не низкая задержка доступа, а то, что процессор

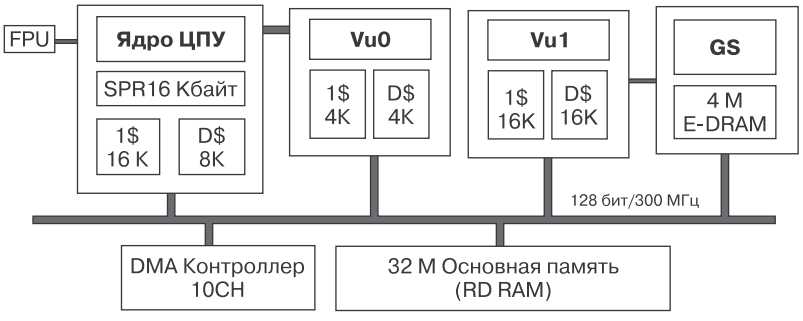


Рис. 3.30. Упрощенный вид архитектуры памяти PS2, иллюстрирующий сверхоперативную память на 16 КиБ (SPR), кэш L1 основного ЦП (EE) и два векторных блока (VU0 и VU1), блок видеопамяти объемом 4 МиБ, доступный для графического синтезатора, блок на 32 МиБ оперативной памяти, контроллер DMA и системные шины

способен получить доступ к ее памяти, не используя системные шины. Таким образом, чтение и запись этой области памяти выполняются, когда адрес системы и шина данных используются для других целей. Например, игра может установить цепочку запросов DMA для передачи данных между основным RAM и двумя модулями векторной обработки (VU) в PS2. Пока эти запросы DMA обрабатываются DMAC и/или когда VU заняты выполнением вычислений (оба они будут интенсивно использовать системные шины), EE может выполнять вычисления на данных, которые находятся в SPR, без вмешательства в DMA или работу VU. Перемещение данных в scratchpad и из нее можно выполнить с помощью обычных инструкций перемещения памяти (или `memcpy()` в C/C++) или через запросы DMA. Таким образом, дополнительная SPR-память PS2 дает программисту большую гибкость и мощь для увеличения пропускной способности игрового движка до максимума.

4 Параллелизм и конкурентное программирование

За последние четыре десятилетия производительность компьютеров, обычно измеряемая миллионами команд в секунду (MIPS) или количеством операций с плавающей точкой в секунду (FLOPS), увеличивается с поразительно быстрой и стабильной скоростью. В конце 1970-х годов сопроцессор Intel 8087 мог набирать только около 50 kFLOPS ($5 \cdot 10^4$ FLOPS), тогда как примерно в то же время суперкомпьютер Cray-1 размером с большой холодильник мог работать со скоростью примерно 160 MFLOPS ($1{,}6 \cdot 10^8$ FLOPS). Сегодня процессоры в игровых приставках, таких как PlayStation 4 или Xbox One, производят около 100 GFLOPS (10^{11} FLOPS) операций. А самый быстрый в настоящее время[1] суперкомпьютер — Sunway TaihuLight в Китае — по оценке LINPACK, имеет 93 PFLOPS (peta-FLOPS, или ошеломляющие $9{,}3 \cdot 10^{16}$ операций с плавающей точкой в секунду). То есть производительность улучшилась за этот период на семь порядков для персональных компьютеров и восемь порядков — для суперкомпьютеров.

Такому быстрому повышению производительности способствовали многие факторы. В самом начале переход от электронных ламп к твердотельным транзисторам позволил сильно уменьшить размеры вычислительного оборудования. Количество транзисторов, которые можно разместить на одной микросхеме, резко возросло, поскольку были разработаны новые типы транзисторов и цифровой логики, новые материалы подложки и передовые производственные процессы. Эти достижения способствовали также уменьшению энергопотребления и резкому увеличению тактовой частоты процессора. А начиная с 1990-х годов производители компьютерного оборудования все чаще обращаются к *параллелизму* как средству повышения вычислительной мощности.

Написать программное обеспечение, которое *правильно* и *эффективно* работает на аппаратных средствах параллельных вычислений, значительно сложнее, чем программы для последовательных компьютеров прошлых лет. Это требует

---

[1] На конец 2020 года самым мощным считается разработанный в японском Институте физико-химических исследований суперкомпьютер Fugaku с пиковой вычислительной мощностью 415 петафлопс и потенциально максимальной 513 петафлопс (https://www.top500.org/system/179807/). — *Примеч. пер.*

глубокого понимания того, как на самом деле работает аппаратное обеспечение. К тому же для максимального использования преимуществ многоядерных процессоров современных вычислительных платформ требуется особый подход к проектированию программного обеспечения, называемый *параллельным программированием*. В параллельной программной системе несколько потоков управления взаимодействуют для решения общей проблемы. Эти потоки должны быть тщательно скоординированы. Многие методы, которые хороши в последовательных программах, не будут работать в параллельных. Поэтому современным программистам (во всех отраслях, включая игры) важно хорошо разбираться в аппаратных средствах параллельных вычислений и методах параллельного программирования.

4.1. Определение конкурентного выполнения и параллелизма

4.1.1. Конкурентное выполнение

Конкурентное (concurrent) программное обеспечение использует для решения задачи *несколько потоков управления*. Это могут быть несколько *потоков*, запущенных в контексте одного процесса, или несколько взаимодействующих процессов, запущенных на одном компьютере либо на нескольких машинах. Множество потоков управления могут быть реализованы также при использовании других методов, таких как *фиберы* или *корутины*.

Основным фактором, который отличает *параллельное программирование* от *последовательного*, является чтение и/или запись *общих данных*. Если у нас есть два и более потока управления, каждый из которых работает с полностью независимым блоком данных, то технически это не является параллельным выполнением — это просто одновременные вычисления (рис. 4.1).

Главная задача конкурентного программирования заключается в том, как координировать несколько считывателей и/или несколько записывающих потоков в общий файл данных таким образом, чтобы обеспечить предсказуемые правильные результаты. В основе этой проблемы лежит особая разновидность *состояния гонки*, известная как *гонка данных*, в которой два и более потока управления конкурируют за то, чтобы первыми считать, модифицировать и записать кусок общих данных. Суть проблемы параллелизма заключается в выявлении и устранении гонки данных. Два примера конкурентных вычислений показаны на рис. 4.2.

Мы будем исследовать методы, которые используют программисты, чтобы избежать гонки данных и, таким образом, написать надежные параллельные программы, начиная с раздела 4.5. Но прежде посмотрим, как параллельное компьютерное оборудование может обеспечить эффективную платформу для запуска параллельного программного обеспечения, а также повысить скорость выполнения последовательных программ.

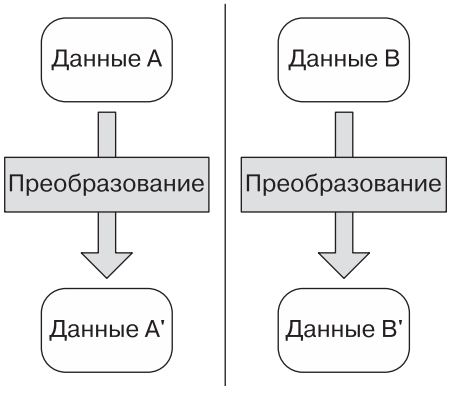


Рис. 4.1. Два потока управления, работающие с независимыми данными, не считаются одновременными, поскольку они не подвержены гонке данных

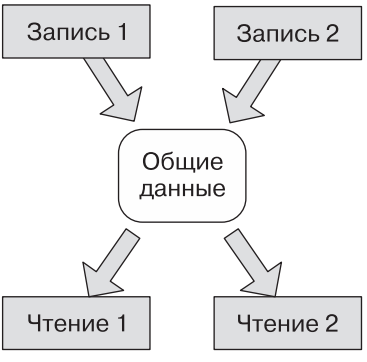


Рис. 4.2. Два потока управления, читающие из общего файла данных и/или записывающие в общий файл данных, являются примерами конкурентного выполнения

4.1.2. Параллелизм

В компьютерной инженерии термин *«параллелизм»* относится к любой ситуации, в которой два и более различных аппаратных компонента работают одновременно. Другими словами, *параллельное* компьютерное оборудование может выполнять более одной задачи одновременно, в отличие от *последовательного*, способного выполнять только одну.

До 1989 года все вычислительные устройства потребительского уровня были исключительно машинами с последовательным выполнением задач. В качестве примера можно привести процессор 6502 компании MOS Technology использовавшийся в персональных компьютерах Apple II и Commodore 64, процессоры Intel 8086, 80286 и 80386, на котором были построены ранние компьютеры IBM и их клоны.

Сегодня оборудование для параллельных вычислений распространено повсеместно. Один из очевидных примеров аппаратного параллелизма — многоядерный процессор, такой как Intel Core™ i7 или AMD Ryzen™ 7. Но параллелизм может быть использован в различных видах отдельных устройств. Например, один ЦП может содержать несколько ALU и, следовательно, выполнять несколько независимых вычислений параллельно. Кластер компьютеров, работающих совместно для решения общей проблемы, также является примером аппаратного параллелизма.

Неявный и явный параллелизм

Одним из способов классификации различных форм параллелизма в проектировании компьютерного оборудования является рассмотрение *цели* параллелизма. Другими словами, какую проблему решает параллелизм в данном проекте?

Размышляя в этом направлении, мы можем грубо разделить параллелизм на две категории:

- неявный;
- явный.

Под *неявным параллелизмом* подразумевается использование параллельных аппаратных компонентов внутри ЦП с целью повышения производительности *отдельного* потока команд. Этот метод известен как *параллелизм на уровне команд* (instruction level parallelism, ILP), потому что ЦП выполняет команды из одного потока инструкций, но каждая команда выполняется с помощью параллелизма аппаратных средств. Примеры неявного параллелизма включают в себя:

- реализацию конвейерной архитектуры;
- суперскалярные архитектуры;
- архитектуры очень длинных командных слов (very long instruction word, VLIW).

Мы рассмотрим неявный параллелизм в разделе 4.2. Графические процессоры также широко используют неявный параллелизм (подробнее поговорим о проектировании и программировании графических процессоров в разделе 4.11).

Явный параллелизм подразумевает использование одинаковых аппаратных компонентов в ЦП, компьютере или компьютерной системе с целью одновременного запуска *более одного потока инструкций*. Другими словами, явно параллельное оборудование специально предназначено для выполнения *параллельного* программного обеспечения, и делает это значительно эффективнее, чем это было бы возможно на последовательной вычислительной платформе. Наиболее распространенные примеры явного параллелизма:

- многопоточные процессоры;
- многоядерные процессоры;
- многопроцессорные компьютеры;
- кластеры компьютеров;
- распределенные вычисления;
- облачные вычисления.

Мы будем исследовать явные параллельные архитектуры в разделе 4.3.

4.1.3. Параллелизм данных и задач

Другой способ понять параллелизм состоит в том, чтобы разделить его на две широкие категории в зависимости от вида работы, выполняемой параллельно.

- *Параллелизм задач*. Так называется параллельное выполнение нескольких разнородных операций. Расчет анимации на одном ядре при проверке столкновений на другом будет примером такой формы параллелизма.

- *Параллелизм данных.* В этом случае выполнение одной операции осуществляется параллельно над несколькими элементами данных. Вычисление 1000 матриц скинов путем одновременного выполнения 250 матричных вычислений на каждом из четырех ядер было бы примером параллелизма данных.

Большинство реальных программ с параллельными вычислениями в разной степени используют параллелизм задач и параллелизм данных.

4.1.4. Таксономия Флинна

Еще один способ классификации различных степеней параллелизма, с которыми мы сталкиваемся в вычислительном оборудовании, заключается в использовании *таксономии Флинна*. Предложенный Майклом Дж. Флинном из Стэнфордского университета в 1966 году подход представляет параллелизм в двухмерном пространстве. На одной оси откладывается количество параллельных потоков управления, которое Флинн называет количеством команд, выполняющихся параллельно в любой данный момент. На другой оси — количество различных потоков данных, с которыми работает каждая инструкция в программе. Таким образом, пространство делится на четыре сектора.

- *Одиночный поток команд, одиночный поток данных* (single instruction, single data, SISD) — один поток команд, работающий с одним потоком данных.
- *Множество потоков команд, множество потоков данных* (multiple instruction, multiple data, MIMD) — несколько командных потоков, работающих с несколькими независимыми потоками данных.
- *Одиночный поток команд, множество потоков данных* (single instruction, multiple data, SIMD) — один поток команд, работающий с несколькими потоками данных, то есть выполняющий одну и ту же последовательность операций с несколькими независимыми потоками данных одновременно.
- *Множество потоков команд, одиночный поток данных* (multiple instruction, single data, MISD) — несколько потоков команд работают в одном потоке данных. (MISD редко используется в играх, широко применяется, в частности, для обеспечения отказоустойчивости за счет избыточности.)

Одиночный и множественный потоки данных

Важно понимать, что поток данных — это не просто массив чисел. Большинство арифметических операторов являются бинарными — они принимают на вход два значения, а выдают одно. Применительно к двоичной арифметике термин «одиночные данные» относится к одной паре входов с одним выходом. В качестве примера посмотрим, как две двоичные арифметические операции, умножение (ab) и деление (c / d), могут выполняться в каждой из четырех категорий Флинна.

- В архитектуре SISD один ALU сначала выполняет умножение, а затем деление (рис. 4.3).

- В архитектуре MIMD два ALU выполняют операции параллельно, работая с двумя независимыми потоками команд (рис. 4.4).

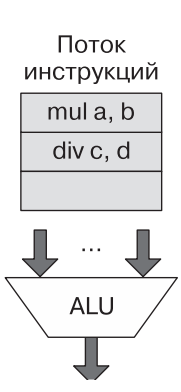


Рис. 4.3. Пример SISD. Один ALU выполняет сначала умножение, затем деление

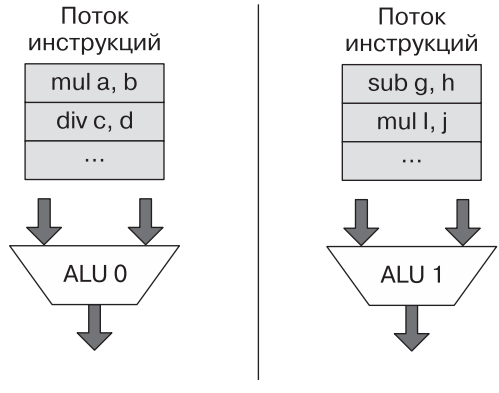


Рис. 4.4. Пример MIMD. Два ALU выполняют операции параллельно

- Классификация MIMD применяется также к случаю, в котором один ALU обрабатывает два независимых потока команд с помощью *квантования времени* (рис. 4.5).

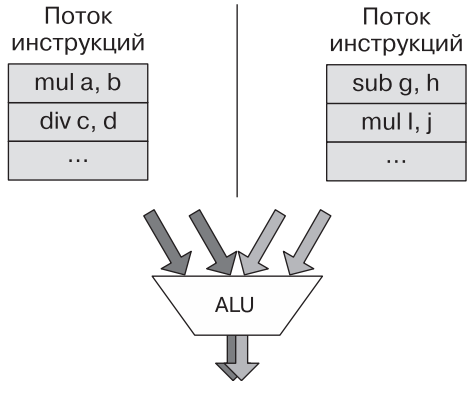


Рис. 4.5. Пример MIMD с временным разделением. Один ALU выполняет операции с двумя независимыми потоками команд, работая в каждый промежуток времени только с одним из них и чередуя их выполнение

- В архитектуре SIMD один широкий ALU, известный как модуль векторной обработки (VPU), сначала выполняет умножение, а затем деление, но каждая команда работает с двумя входными векторами, содержащими по четыре элемента, и производит на выходе один вектор также из четырех элементов (рис. 4.6).

- В архитектуре MISD два ALU обрабатывают один и тот же поток команд (сначала умножение, затем деление) и в идеале дают идентичные результаты. Эта архитектура (рис. 4.7) в первую очередь полезна для реализации отказоустойчивости за счет избыточности. ALU 1 выступает в качестве горячего резерва для ALU 0 и наоборот. Это означает: если один из ALU испытывает сбой, система может беспрепятственно переключаться на другой.

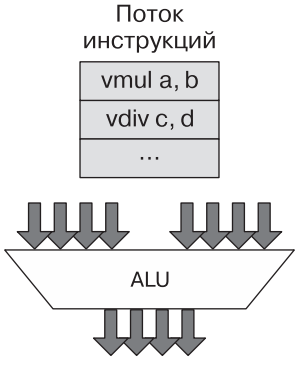


Рис. 4.6. Пример SIMD. VPU сначала выполняет умножение, а затем деление, но каждая команда работает с парой четырехэлементных входных векторов и производит четыре элемента выходного вектора

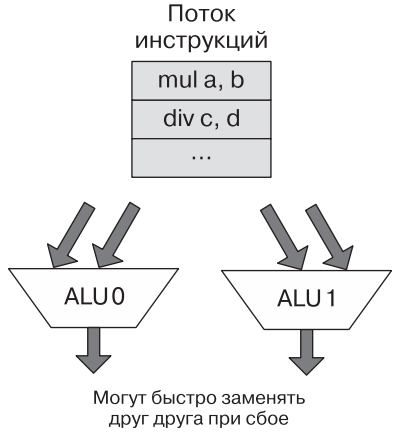


Рис. 4.7. Пример MISD. Два ALU обрабатывают один и тот же поток команд (сначала умножение, затем деление) и в идеале дают идентичные результаты

Параллелизм графического процессора: SIMT

В последние годы к таксономии Флинна был добавлен пятый пункт классификации для учета проектирования графических процессоров (GPU). *Архитектура SIMT (одиночный поток команд, множество потоков выполнения)* по сути является гибридом между SIMD и MIMD и используется главным образом при разработке графических процессоров. Она сочетает метод SIMD (одиночный поток команд, работающий с несколькими потоками данных одновременно) с многопоточностью (выполнение процессором нескольких потоков команд с разделением их по времени).

Термин SIMT был придуман NVIDIA, но его можно использовать для описания архитектуры любого графического процессора. Для обозначения конструкции SIMT, то есть графического процессора, состоящего из довольно большого количества легких SIMD-ядер, часто используется термин *«многоядерный»*, тогда как термин *«мультиядерный»* относится к конструкциям MIMD, то есть ЦП с меньшим количеством тяжеловесных ядер общего назначения. Подробнее рассмотрим дизайн SIMT, используемый графическими процессорами, в разделе 4.11.

4.1.5. Ортогональность конкурентных вычислений и параллелизма

Здесь следует подчеркнуть, что параллельное программное обеспечение не требует параллельного аппаратного обеспечения, а параллельное оборудование предназначено *не только* для запуска параллельного программного обеспечения. Например, конкурентная многопоточная программа может работать на одном последовательном ядре ЦП с помощью вытесняющей многозадачности (см. подраздел 4.4.4). Аналогично параллелизм на уровне команд предназначен для повышения производительности одного потока и, следовательно, полезен как для параллельного, так и для последовательного программного обеспечения. Таким образом, хотя они тесно связаны, конкурентность и параллелизм являются действительно ортогональными понятиями.

Пока в системе задействованы *несколько записывающих* и/или *несколько считывающих* объектов общего объекта данных, у нас есть параллельная система. Параллелизм может быть достигнут с помощью вытесняющей многозадачности (на последовательном или параллельном оборудовании) или истинного параллелизма (в котором каждый поток выполняется на отдельном ядре) — методы, которые мы изучим в этой главе, применимы в любом случае.

4.1.6. Вопросы, рассматриваемые в главе

В следующих разделах мы сначала обратим внимание на неявный параллелизм и на то, как лучше всего оптимизировать программное обеспечение, чтобы использовать его в своих интересах. Далее рассмотрим наиболее распространенные формы явного параллелизма. Затем поговорим о различных методах параллельного программирования, применяемых для явного использования параллельных вычислительных платформ. А завершим обсуждение параллельного программирования векторной обработкой SIMD и ее применением к методам проектирования графических процессоров и программирования GPU общего назначения (general-purpose GPU programming, GPGPU).

4.2. Неявный параллелизм

Ранее мы говорили, что *неявный параллелизм* — это использование аппаратных средств параллельного вычисления для повышения скорости выполнения одного потока. Производители процессоров начали задействовать неявный параллелизм в своих продуктах в конце 1980-х годов, пытаясь ускорить работу *существующего кода* на каждом новом поколении процессоров в рамках данной линейки продуктов.

Существует несколько способов применения параллелизма для повышения производительности одного потока ЦП. Наиболее распространенными являются

конвейерные архитектуры ЦП, *суперскалярные* и архитектуры *очень длинных командных слов* (VLIW). Начнем с рассмотрения работы конвейерного процессора, а затем разберем два других варианта неявного параллелизма.

4.2.1. Конвейерная обработка

Для того чтобы ЦП обработал одну инструкцию на машинном языке, он должен выполнить ряд отдельных *этапов*. Конструкции ЦП немного различаются — в одних ЦП используется большое количество отдельных этапов, в других — меньшее количество этапов общего назначения. Но каждый процессор так или иначе реализует следующие основные этапы:

- *запрос* — выполняемая инструкция считывается из ячейки памяти, на которую указывает регистр указателя инструкций (IP);
- *декодирование* — слово команды разбивается на код операции, режим адресации и дополнительные операнды;
- *выполнение* — на основании кода операции выбирается соответствующий функциональный блок в CPU (ALU, FPU, контроллере памяти и т. д.). Инструкция отправляется выбранному компоненту для обработки вместе с любыми соответствующими данными операндов. Затем функциональный блок выполняет свою работу;
- *доступ к памяти* — если инструкция включает чтение или запись в память, на этом этапе контроллер памяти выполняет соответствующую операцию;
- *обратная запись в регистр* — функциональный блок, выполняющий инструкцию (ALU, FPU и т. д.), записывает полученные результаты обратно в регистр назначения.

На рис. 4.8 прослеживается путь двух команд, A и B, через пять фаз выполнения последовательного ЦП. Вы сразу заметите, что на диаграмме много пустого пространства: в то время как выполняется один этап инструкции, все остальные этапы не задействуются.

Тактовый цикл	Запрос	Декодирование	Выполнение	Доступ к памяти	Обратная запись
0	A				
1		A			
2			A		
3				A	
4					A
5	B				
6		B			

Рис. 4.8. В процессоре без конвейера этапы команд бездействуют большую часть времени

Каждый из этапов выполнения инструкции фактически обрабатывается различными аппаратными средствами внутри ЦПУ (рис. 4.9). Блок управления (control unit, CU) и контроллер памяти обрабатывают стадию запроса инструкции. Затем другая схема внутри CU выполняет этап декодирования. ALU, FPU или VPU осуществляют львиную долю этапа выполнения. Этап доступа к памяти реализуется контроллером памяти. И наконец, стадия обратной записи в основном включает регистры. Такое разделение труда между различными цепями в ЦП является ключом к повышению эффективности процессора: нам просто нужно на всех этапах загружать работой имеющееся аппаратное обеспечение.

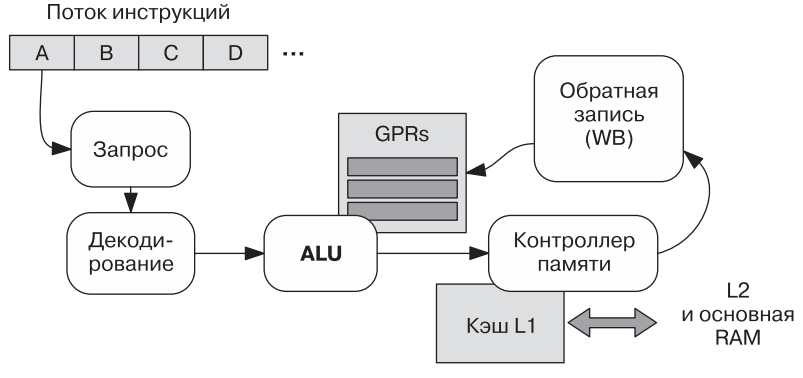


Рис. 4.9. Компоненты конвейерного скалярного процессора

Такое решение известно как *конвейерная обработка*. Вместо того чтобы ждать, пока каждая инструкция завершит все пять этапов, прежде чем начинать выполнять следующую, мы приступаем к выполнению новой инструкции *на каждом* такте. Поэтому несколько инструкций находятся «в полете» одновременно (рис. 4.10).

Тактовый цикл	Запрос	Декодирование	Выполнение	Доступ к памяти	Обратная запись
0	A				
1	B	A			
2	C	B	A		
3	D	C	B	A	
4	E	D	C	B	A
5	F	E	D	C	B
6	G	F	E	D	C

Рис. 4.10. Идеальный поток инструкций через конвейерный процессор

Конвейер немного похож на стирку. Если вам нужно постирать много вещей, было бы неэффективно ждать, пока каждая порция белья будет выстирана и высушена, перед стиркой следующей партии: пока стиральная машина занята, сушилка

будет скучать без дела и наоборот. Гораздо лучше постоянно держать обе машины включенными, начиная стирку второй порции белья, как только первая будет загружена в сушилку, и т. д.

Конвейер — это форма параллелизма, известная *как параллелизм на уровне команд* (instruction-level parallelism, ILP). По большей части ILP должен быть прозрачным для программиста. В идеале при заданной тактовой частоте программа, которая правильно работает на скалярном ЦП, должна иметь возможность корректно, но быстрее работать на конвейерном ЦП, если, конечно, два процессора поддерживают одинаковую архитектуру набора команд (ISA). Теоретически процессор с конвейерной обработкой, который реализует N этапов обработки, может выполнять программу в N раз быстрее, чем его последовательный аналог. Однако, как мы рассмотрим в подразделе 4.2.5, конвейерная обработка не всегда протекает так, как ожидается, из-за различных типов *зависимостей* между инструкциями в потоке команд. Программисты, заинтересованные в написании высокопроизводительного кода, не могут оставаться в стороне от ILP. Мы должны принять и понять это и иногда изменять свои код и/или данные так, чтобы получить максимальную отдачу от конвейерного процессора.

4.2.2. Задержка и пропускная способность

Задержка конвейера — это количество времени, необходимое для полной обработки одной инструкции. Это просто сумма задержек всех этапов конвейера. Обозначая задержки с помощью переменной времени T для конвейера с N этапами, мы можем написать:

$$T = \sum_{i=0}^{N-1} T_j. \tag{4.1}$$

Пропускная способность, или *полоса пропускания* конвейера, показывает, сколько инструкций он может обработать за единицу времени. Пропускная способность конвейера определяется задержкой его самого медленного этапа, так же как прочность цепи определяется прочностью ее самого слабого звена. Пропускная способность может рассматриваться как частота f, измеренная в инструкциях в секунду. Это можно записать следующим образом:

$$f = \frac{1}{\max\left(\tau_j\right)}. \tag{4.2}$$

4.2.3. Глубина конвейера

Мы сказали, что все этапы в ЦП потенциально могут иметь различную задержку T_i и этап с самой большой задержкой определяет пропускную способность всего процессора. На каждом такте, ожидая завершения самого медленного этапа, более быстрые бездействуют. В идеале мы бы хотели, чтобы все этапы в процессоре имели примерно одинаковую задержку.

Это может быть достигнуто увеличением общего количества этапов в конвейере: если одна стадия занимает намного больше времени, чем другие, она может быть разбита на две или несколько более коротких стадий в попытке сделать все задержки примерно равными. Однако мы не можем просто разделять этапы до бесконечности: чем больше этапов, тем выше общая задержка команд. Это увеличивает потери скорости конвейера (см. подраздел 4.2.4). Поэтому производители процессоров пытаются найти баланс между увеличением пропускной способности через более глубокие конвейеры и контролем над общей задержкой инструкций. В результате длина реальных конвейеров ЦП варьируется от 4 этапов минимум до 30 этапов максимум.

4.2.4. Потеря скорости конвейера

Иногда процессор на определенном такте не может выдать новую инструкцию. Это называется *потерей скорости конвейера (stall)*. В таком тактовом цикле первая ступень конвейера простаивает. В следующем тактовом цикле будет простаивать второй этап и т. д. Поэтому остановка может рассматриваться как «пузырь» времени простоя, распространяющийся по конвейеру со скоростью один этап за такт. Эти «пузыри» иногда называют *слотами задержки*.

4.2.5. Зависимости данных

Задержки также могут быть вызваны зависимостями между инструкциями в выполняемом потоке команд. Например, рассмотрим такую последовательность инструкций:

```
mov  ebx,5       ;; загрузить значение 5 в регистр EBX
imul eax,10      ;; умножить содержимое EAX на 10
                 ;; (результат сохраняется в EAX)
add  eax,7       ;; прибавить 7 к EAX (результат сохраняется в EAX)
```

В идеале мы хотели бы выдать инструкции `mov`, `imul` и `add` для трех последовательных тактов, чтобы конвейер был максимально загружен. Но в этом примере результаты инструкции `imul` используются следующей за ней инструкцией `add`, поэтому ЦП должен дождаться, пока `imul` полностью пройдет через конвейер, прежде чем начинать `add`. Если конвейер содержит пять этапов, это означает, что четыре цикла будут потрачены впустую (рис. 4.11). Такого рода зависимости между инструкциями называются *зависимостями данных*.

На самом деле существует три вида зависимостей между инструкциями, которые могут вызвать потерю скорости конвейера:

- зависимости данных;
- зависимости управления, известные также как зависимости ветвления;
- структурные, или ресурсные, зависимости.

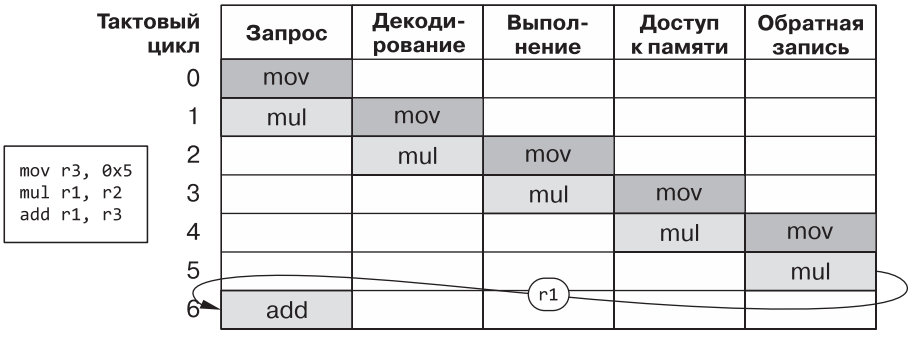


Рис. 4.11. Зависимость данных между инструкциями вызывает остановку конвейера

Сначала мы обсудим, как избежать зависимостей данных, затем рассмотрим зависимости ветвей и то, как смягчить их влияние. Наконец, представим суперскалярные архитектуры ЦП и обсудим, как они могут привести к структурным зависимостям в конвейерном ЦП.

Изменение порядка инструкций

Чтобы смягчить воздействие зависимости данных, нужно найти другие инструкции для процессора, которые будут выполняться, пока он ожидает выполнения зависимой инструкции. Часто это можно сделать, *переупорядочив* инструкции в программе, но при этом стараясь не изменять *поведение* программы в процессе. Для любой данной пары взаимозависимых инструкций мы хотим найти некие соседние инструкции, которые *не* зависят от них, и переместить их вверх или вниз, чтобы они в конечном итоге выполнялись *между* зависимой парой команд, заполняя «пузырь» полезной работой.

Конечно, переупорядочение инструкций может сделать вручную программист, который не против погружения в программирование на ассемблере. К счастью, часто это не является необходимостью: современные оптимизирующие компиляторы очень хорошо умеют автоматически переупорядочивать инструкции, чтобы уменьшить или устранить влияние зависимостей данных.

Как программисты, мы не должны слепо доверять компилятору и позволять ему целиком оптимизировать наш код — при написании высокопроизводительного кода всегда полезно взглянуть на дизассемблирование и убедиться, что компилятор выполнил разумную работу. Однако мы должны помнить и правило 80/20 (см. раздел 2.3) и тратить время только на оптимизацию 20 % кода или менее, что все равно заметно влияет на общую производительность.

Внеочередное исполнение

Компиляторы и программисты — не единственные, кто способен переупорядочить последовательность инструкций машинного языка для предотвращения простоев. Многие из современных процессоров поддерживают функцию, известную как *вы-*

полнение вне очереди, которая позволяет им динамически определять зависимости данных между инструкциями и автоматически разрешать их.

Чтобы реализовать это, ЦП просматривает поток команд и анализирует использование регистра команд, чтобы обнаружить зависимости между ними. Когда зависимость найдена, в окне предварительного просмотра выполняется поиск другой инструкции, которая не зависит ни от одной из выполняемых в настоящее время. Если такая найдена, она выполняется (вне очереди!), чтобы поддерживать занятость конвейера. Детали того, как происходит внеочередное выполнение, выходят за рамки данной книги. Достаточно сказать, что, как программисты, мы не можем полагаться на то, что ЦП будет выполнять инструкции в том же порядке, в котором мы (или компилятор) их написали.

Оптимизатор компилятора и логика внеочередного выполнения CPU заботятся о том, чтобы поведение программы не изменилось в результате переупорядочения команд. Однако, как мы увидим в подразделе 4.9.3, оптимизация компилятора и выполнение не по порядку могут вызвать ошибки в параллельной программе (то есть в программе, состоящей из нескольких потоков, которые совместно используют данные). Это одна из многих причин, почему параллельное программирование требует большего внимания, чем последовательное.

4.2.6. Зависимости ветвления

Что происходит, когда конвейерный ЦП встречает команду условного перехода, например оператор `if` или условное выражение в конце цикла `for` или `while`? Чтобы ответить на этот вопрос, рассмотрим следующий код C/C++:

```
int SafeIntegerDivide(int a, int b, int defaultVal)
{
    return (b != 0) ? a / b : defaultVal;
}
```

Если дизассемблировать эту функцию, то результат для процессора Intel x86 может выглядеть примерно так:

```
    ; преамбула функции опущена для ясности...

    ; сначала поместите значение по умолчанию в регистр возврата
    mov  eax,dword ptr [defaultVal]

    mov  esi,dword ptr [b] ; проверяем (b != 0)
    cmp  esi,0
    jz   SkipDivision

    mov  eax, dword ptr[a] ; делитель (a) должен быть в EDX: EAX
    cdq                    ; ... расширенный EDX
    idiv esi               ; арендованное пространство в EAX

SkipDivision:
    ; заключительная часть функции опущена для ясности...
    ret                    ; EAX — возвращаемое значение
```

Зависимость здесь находится между инструкциями `cmp` (сравнение) и `jz` (переход, если равен нулю). Процессор не может выполнить условный переход, пока не узнает результаты сравнения. Это называется *зависимостью ветвления*, известной также как *зависимость управления*. На рис. 4.12 изображена зависимость перехода.

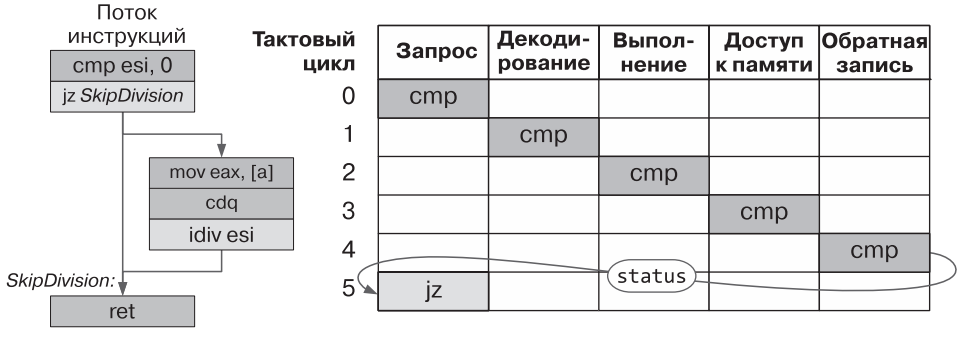


Рис. 4.12. Зависимость между инструкцией сравнения и инструкцией условного перехода называется зависимостью перехода

Спекулятивное выполнение

Один из способов, с помощью которого ЦПУ справляются с зависимостями ветвлений, — это *спекулятивное выполнение*, или *прогнозирование ветвлений*. Всякий раз, когда встречается ветвление, процессор пытается *угадать*, какая ветвь будет выполнена. Он продолжает выполнять инструкции из выбранной ветви в надежде, что его предположение было верным. Конечно, процессор не будет знать наверняка, правильно ли он угадал, пока зависимая инструкция не дойдет до конца конвейера. Если предположение неверное, оказывается, что процессор выполнил инструкции, которые не должны были выполняться вообще. Таким образом, конвейер должен быть *очищен* и перезапущен на первой инструкции правильной ветви. Это называется *накладными расходами ветвления*.

Самое простое предположение, которое может сделать процессор, — то, что ветви никогда не будут выполнены. Процессор продолжает последовательно выполнять инструкции и перемещает указатель инструкций в новое место только тогда, когда его предположение оказалось неверным. Этот подход дружествен I-кэшу в том смысле, что процессор всегда предпочитает ветвь, инструкции которой, скорее всего, находятся в кэше.

Другой, чуть более продвинутый подход к предсказанию ветвлений состоит в том, чтобы предполагать, что всегда выполняются обратные ветви, а прямые — никогда. Обратная ветвь — это разновидность ветвления, которая встречается в конце цикла `while` или `for`, поэтому такие ветви оказываются предпочтительными чаще, чем прямые.

Большинство высококачественных ЦП включают аппаратное *предсказание ветвлений*, которое может значительно улучшить качество статических предположений. Предиктор ветвления может отслеживать результаты инструкции ветвления в течение нескольких итераций цикла и обнаружить шаблоны, которые помогут ему точнее угадывать на последующих итерациях.

Программистам PS3 постоянно приходилось сталкиваться с плохой производительностью «ветвистого» кода, потому что предикторы ветвления в процессоре Cell были, честно говоря, ужасными. Но процессор AMD Jaguar, установленный в PS4 и Xbox One, оснащен высокоразвитым аппаратным обеспечением для прогнозирования ветвлений, поэтому разработчики игр могут облегченно вздохнуть при написании кода для PS4.

Предикация

Еще один способ смягчить влияние зависимостей ветвления — полностью избежать ветвления. Снова рассмотрим функцию `SafeIntegerDivide()`, немного изменив ее, чтобы она работала со значениями с плавающей точкой вместо целых чисел:

```
float SafeFloatDivide(float a, float b, float d)
{
    return (b != 0.0f) ? a / b : d;
}
```

Эта простая функция вычисляет один из двух ответов в зависимости от результатов условного теста `b! = 0`. Вместо использования оператора условного перехода для возврата одного из этих двух ответов мы можем организовать условный тест так, чтобы генерировалась *битовая маска*, состоящая только из нулей (0x0U), если условие ложно, и только из единиц (0xFFFFFFFFU), если истинно. Затем можем выполнить обе ветви, генерируя два альтернативных ответа. Наконец, применим маску для получения окончательного ответа, который вернем из функции.

Следующий псевдокод иллюстрирует идею предикации. (Обратите внимание на то, что он не будет работать как есть. В частности, вы не сможете замаскировать число с плавающей точкой с беззнаковым `int` и получить результат с плавающей точкой — потребуется применить объединение, чтобы интерпретировать битовые шаблоны чисел с плавающей точкой, как если бы они были целыми числами без знака при использовании маски.)

```
int SafeFloatDivide_pred(float a, float b, float d)
{
    // преобразовать логическое значение (b! = 0.0f) в условие 1U или 0U
    const unsigned condition = (unsigned)(b != 0.0f);

    // конвертировать 1U -> 0xFFFFFFFFU
    // конвертировать 0U -> 0x00000000U
    const unsigned mask = 0U - condition;
```

```
    // вычислить частное (будет QNaN, если b == 0.0f)
    const float q = a / b;

    // выбираем частное, когда маска равна единице, или значение
    // по умолчанию d, когда маска – все нули (Примечание: это не
    // будет работать, как написано, – нужно использовать объединение (union)
    // для интерпретации значения с плавающей точкой как беззнакового
    // для маскировки)
    const float result = (q & mask) | (d & ~mask);
    return result;
}
```

Рассмотрим, как это работает.

- Проверка `b! = 0.0f` дает в результате логическое значение `bool`. Мы преобразуем его в целое число без знака, просто приводя его. Это дает либо `1U` (соответствует `true`), либо `0U` (соответствует `false`).
- Конвертируем беззнаковый результат в битовую маску, вычитая ее из `0U`. Ноль минус ноль — по-прежнему ноль, а ноль минус один равно –1, или `0xFFFFFFFU` в 32-разрядной целочисленной арифметике без знака.
- Далее вычислим частное. Мы запускаем этот код независимо от результата ненулевого теста, тем самым обходя стороной любые проблемы с зависимостями ветвления.
- Теперь у нас есть два готовых ответа — частное `q` и значение по умолчанию `d`. Мы хотим применить маску, чтобы выбрать одно из них. Для этого нужно *интерпретировать* битовые комбинации с плавающей точкой `q` и `d`, *как если бы* они были целыми числами без знака. Самый распространенный способ сделать это в C/C++ — задействовать объединение, содержащее два члена, один из которых интерпретирует 32-разрядное значение как число с плавающей точкой, а другой — как беззнаковое целое.
- Маска используется следующим образом: мы поразрядно применяем логическое И к частному `q` и маске, производя битовый шаблон, который соответствует `q`, если маска — одни единицы, или нулю, если нули. Далее поразрядно применяем логическое И к значению по умолчанию `d` и дополнению маски, получая все нули, если маска состоит из одних единиц, или битовую комбинацию `d`, если маска — нули. Наконец, поразрядно применяем логическое ИЛИ для двух значений вместе, эффективно выбирая либо значение `q`, либо значение `d`.

Применение маски для выбора одного из двух возможных значений, таких как эти, называется предикацией, потому что мы выполняем оба пути кода (тот, который возвращает `a/b`, и тот, который возвращает `d`), но каждый путь кода основан на результатах теста `(a! = 0)` через маску. Поскольку в ходе этой процедуры мы выбираем одно из двух возможных значений, ее часто называют также операцией *выбора*.

Попытка избежать ветвления может показаться излишней. И так может случиться — ее полезность будет зависеть от относительной стоимости ветвления по сравнению с предопределенной альтернативой на вашем целевом оборудовании. Предикация действительно необходима, когда ISA процессора предоставляют специальные инструкции на машинном языке для выполнения операции выбора. Например, PowerPC ISA предлагает целочисленную операцию выбора `isel`, операцию выбора с плавающей точкой `fsel` и даже SIMD-команду выбора вектора `vecsel`, и их применение может улучшить производительность на платформах на основе PowerPC, таких как PS3.

Важно понимать, что предикация работает только тогда, когда обе ветви могут быть выполнены *безопасно*. Операция деления на ноль с плавающей точкой генерирует «тихое не число» (QNaN), а вот целочисленное деление на ноль генерирует исключение, которое, если оно не отловлено, приведет к сбою в игре. Вот почему мы преобразовали этот пример в вычисление с плавающей точкой, прежде чем задействовать предикацию.

4.2.7. Суперскалярные процессоры

Конвейерный процессор, который мы описали в подразделе 4.2.1, называется *скалярным* процессором. Это означает, что он может начать выполнять не более одной инструкции за такт. Да, в любой момент несколько команд находятся «в полете», но только одна новая команда отправляется по конвейеру в каждый такт.

По сути, параллелизм заключается в одновременном использовании нескольких аппаратных компонентов. Таким образом, один из способов удвоить пропускную способность ЦП (по крайней мере теоретически!) состоял бы в *дублировании* большинства компонентов в микросхеме таким образом, чтобы в каждом такте можно было запускать две инструкции. Это называется *суперскалярной* архитектурой.

В суперскалярном ЦП два или более экземпляра схемы, которая управляет каждым этапом конвейера[1], присутствуют на кристалле. Процессор все еще берет инструкции из одного потока команд, но вместо выдачи одной инструкции, на которую указывает IP во время каждого тактового цикла, берутся следующие две команды и отправляются на обработку на каждом такте. На рис. 4.13 показаны аппаратные компоненты, находящиеся в двухстороннем суперскалярном процессоре, а на рис. 4.14 прослеживается путь 14 инструкций от A до N, когда они проходят через два параллельных конвейера этого процессора.

[1] Технически конвейерная и суперскалярная конструкции — это две независимые формы параллелизма. Конвейерный процессор не обязательно должен быть суперскалярным. Аналогично суперскалярный процессор не обязан поддерживать конвейер, хотя большинство из них это делают.

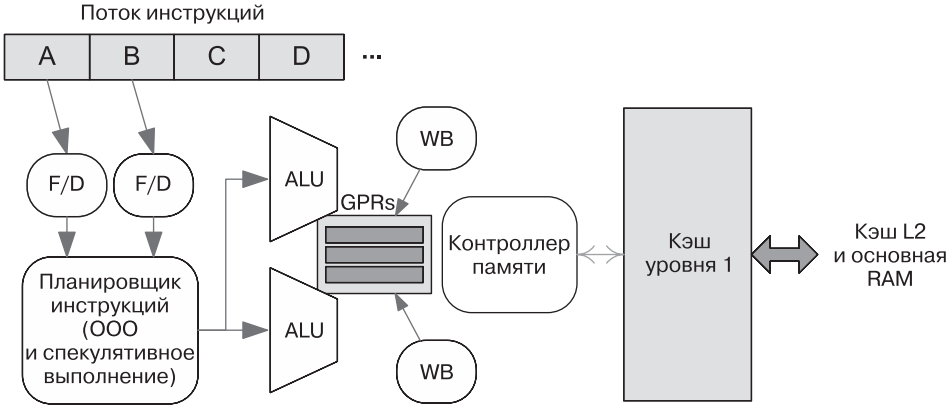


Рис. 4.13. Конвейерный суперскалярный ЦП содержит несколько исполнительных компонентов (ALU, FPU и/или VPU), управляемых одним планировщиком команд, который обычно поддерживает неупорядоченное и спекулятивное выполнение

Тактовый цикл	F_0	F_1	D_0	D_1	E_0	E_1	M_0	M_1	W_0	W_1
0	A	B								
1	C	D	A	B						
2	E	F	C	D	A	B				
3	G	H	E	F	C	D	A	B		
4	I	J	G	H	E	F	C	D	A	B
5	K	L	I	J	G	H	E	F	C	D
6	M	N	K	L	I	J	G	H	E	F

Рис. 4.14. Идеальный случай выполнения 14 команд от A до N на суперскалярном конвейерном процессоре в течение семи тактов

Сложности суперскалярной архитектуры

Реализация суперскалярного ЦП не так проста, как копирование и вставка двух одинаковых ядер ЦП на кристалл. Хотя разумно представить суперскалярный ЦП в виде двух параллельных конвейеров команд, эти два конвейера питаются из одного потока команд. Поэтому требуется некоторая логика управления на входе этих параллельных конвейеров. Как и на процессоре, который поддерживает внеочередное выполнение, логика управления суперскалярного процессора просматривает поток команд в попытке определить *зависимости* между инструкциями, а затем выдает команды не по порядку, чтобы сгладить последствия.

Вдобавок к зависимостям данных и ветвления суперскалярный ЦП подвержен третьему виду зависимости, известной как *зависимость от ресурса*. Этот вид зависимости появляется, когда двум или более последовательным инструкциям требуется один и тот же функциональный блок внутри процессора. Например, представим, что у нас суперскалярный процессор с двумя целочисленными ALU, но только одним FPU. Такой процессор способен выполнить две целочисленные арифметические инструкции на каждом такте. Но если в потоке команд встречаются две арифметические команды с плавающей точкой, они не могут быть выполнены в одном и том же тактовом цикле, поскольку ресурс, который нужен второй команде (FPU), уже будет использоваться первой. Таким образом, логика управления, которая управляет диспетчеризацией команд на суперскалярном процессоре, даже более сложна, чем логика на скалярном процессоре, который поддерживает выполнение не по порядку.

Суперскаляр и RISC

Суперскалярный ЦП с двумя потоками требует примерно в два раза большей площади на кристалле микросхемы, чем сопоставимая скалярная конструкция ЦП. Поэтому для уменьшения количества транзисторов большинство суперскалярных ЦП выполняют в виде процессоров *с сокращенным набором команд* (reduced instruction set CPU, RISC). ISA процессора RISC предоставляет сравнительно небольшой набор инструкций, каждая из которых имеет четко определенную цель. Более сложные операции выполняются созданием последовательностей этих более простых инструкций. В отличие от RISC, ISA компьютера *со сложным набором инструкций* (complex instruction set computer, CISC) предлагает гораздо более широкий набор инструкций, каждая из которых способна выполнять более сложные операции.

4.2.8. Очень длинные командные слова

В подразделе 4.2.7 мы увидели, что суперскалярный ЦП содержит очень сложную логику диспетчеризации. Она занимает ценное пространство на кристалле процессора. Кроме того, центральные процессоры способны подсмотреть сравнительно небольшое количество команд при анализе зависимостей и поиске возможностей для внеочередной и/или суперскалярной диспетчеризации инструкций. Это ограничивает эффективность динамической оптимизации, которую может выполнять ЦП.

Несколько более простой способ реализовать параллелизм на уровне команд состоит в том, чтобы спроектировать ЦП, который имеет на кристалле несколько вычислительных элементов (ALU, FPU, VPU), но задачу отправки инструкций этим вычислительным элементам отдает на откуп программисту и/или компилятору. Таким образом, вся сложная логика *диспетчеризации* команд может

быть устранена и те транзисторы, которые предназначены для ее реализации, могут быть переданы вычислительным элементам или использованы для увеличения кэша. Дополнительным преимуществом является то, что программисты и компиляторы изначально могут лучше оптимизировать распределение команд в своих программах, чем ЦП, потому что они способны выбирать инструкции для отправки из гораздо более широкого окна (как правило, набора всех инструкций функции).

Чтобы позволить программистам и/или компиляторам отправлять команды множеству вычислительных элементов в каждом тактовом цикле, слово инструкции расширяется таким образом, чтобы содержать два и более слота, каждый из которых соответствует вычислительному элементу на чипе. Например, если наш гипотетический процессор содержал два целочисленных ALU и два FPU, программист или компилятор должны были бы иметь возможность кодировать до двух целочисленных и двух операций с плавающей точкой в каждом слове инструкции. Мы называем это *очень длинным командным словом* (VLIW). Архитектура VLIW показана на рис. 4.15.

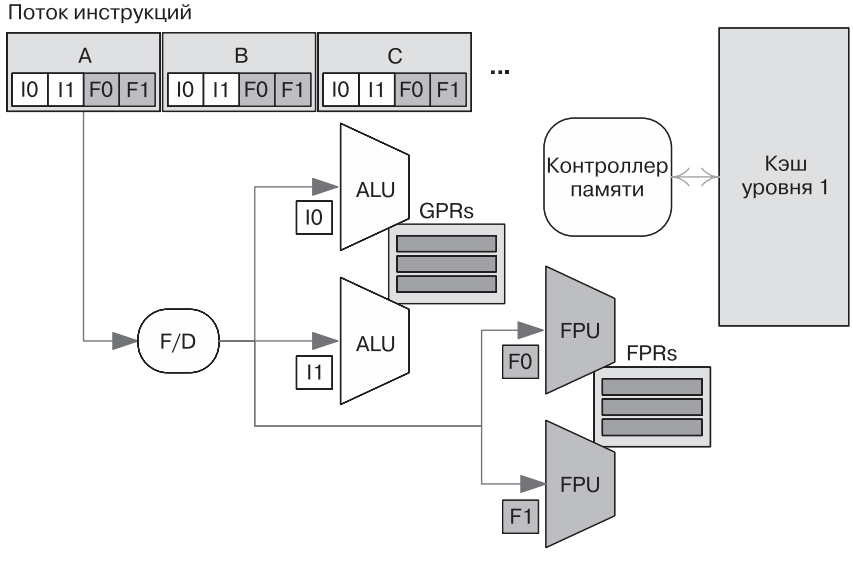


Рис. 4.15. Конвейерная архитектура процессора VLIW, состоящая из двух целочисленных ALU и двух FPU для вычислений с плавающей точкой. Каждое очень длинное командное слово состоит из двух целочисленных операций и двух операций с плавающей точкой, которые отправляются в соответствующие функциональные блоки. Обратите внимание на отсутствие сложной логики планирования команд, которая присутствует в суперскалярном процессоре

Конкретный пример архитектуры VLIW мы найдем в PlayStation 2: он содержит два сопроцессора, называемых *векторными модулями* (VU0 и VU1), каждый

из которых способен отправлять две инструкции за такт. Каждое командное слово состоит из двух слотов, называемых *низким* и *высоким*. Зачастую сложно эффективно заполнить оба слота при ручном кодировании на языке ассемблера, хотя разработаны инструменты, которые помогают программистам преобразовать программу с одной инструкцией за такт в эффективный формат с двумя инструкциями за такт.

Есть компромиссы между суперскалярным и VLIW-подходом. Поскольку в нем отсутствует сложное планирование, логическое выполнение и логика предсказания ветвления суперскалярного ЦП, то процессор VLIW намного проще и поэтому может потенциально более интенсивно использовать параллелизм, чем его суперскалярные аналоги. Однако бывает очень сложно преобразовать последовательную программу в форму, которая в полной мере задействует преимущества параллелизма в VLIW. Это усложняет работу программиста и/или компилятора. Тем не менее ряд достижений помогли преодолеть некоторые из этих ограничений, в частности, появились конструкции VLIW переменной ширины (например, см. https://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2834).

4.3. Явный параллелизм

Явный параллелизм предназначен для повышения эффективности работы параллельного программного обеспечения. Следовательно, все явно параллельные аппаратные конструкции позволяют параллельно обрабатывать более одного потока команд. Далее мы перечислим несколько типичных явных параллельных конструкций, начиная с *гиперпоточности* на одном конце спектра до *облачных вычислений* — на другом, от мелких к крупным.

4.3.1. Гиперпоточность

Как мы видели в подразделе 4.2.5, некоторые конвейерные процессоры способны выполнять команды *не по порядку*, чтобы уменьшить задержки конвейера. Обычно конвейерный процессор выполняет инструкции в порядке, определенном программой, но иногда следующая инструкция в потоке команд не может быть выполнена из-за зависимости от инструкции «в полете». Это создает *интервал задержки*, в который теоретически может быть выполнена другая инструкция. Процессор с внеочередной логикой может смотреть в поток инструкций и выбирать команду для выполнения не по порядку во время этого интервала задержки.

При наличии только одного потока команд возможности ЦП несколько ограничены при выборе команды для отправки в интервал задержки. Но что, если процессор может выбирать инструкции одновременно из двух отдельных потоков команд? Этот принцип стоит за ядром процессора с *гиперпоточностью* (hyperthreading, HT).

Технически HT-ядро состоит из двух файлов регистров и двух модулей декодирования инструкций, но с одним внутренним интерфейсом для выполнения инструкций и одним общим кэшем L1. Такая конструкция позволяет HT-ядру запускать два независимых потока, требуя меньше транзисторов, чем двухъядерный процессор, благодаря общей внутренней части и кэш-памяти L1. Конечно, совместное использование аппаратных компонентов снижает пропускную способность команд по сравнению с присущей двухъядерному ЦП, поскольку потоки конкурируют за общие ресурсы. На рис. 4.16 показаны ключевые компоненты типичной конструкции многопоточного процессора.

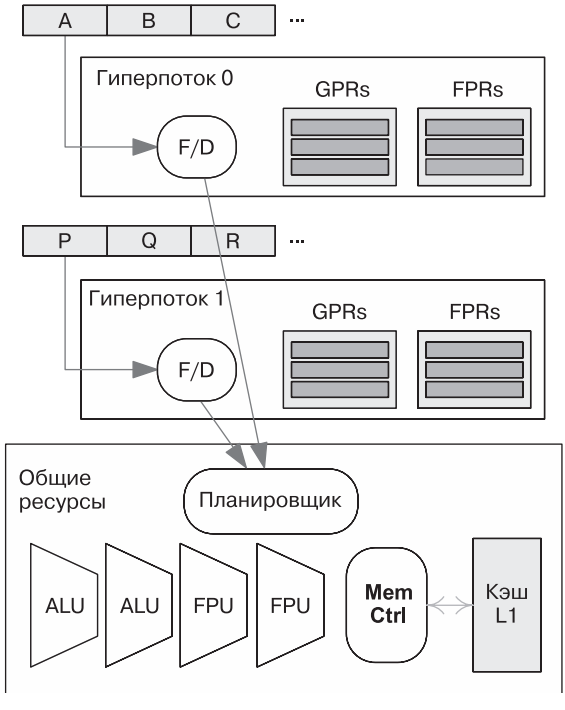


Рис. 4.16. Многопоточный ЦП, имеющий два внешних интерфейса (каждый из которых состоит из блока выборки/декодирования и файла регистра), но с одним внутренним интерфейсом, содержащим ALU, FPU, контроллер памяти, кэш-память L1 и планировщик внеочередных команд. Планировщик выдает инструкции от обоих внешних потоков к общим внутренним компонентам

4.3.2. Многоядерные процессоры

Ядро ЦП может быть определено как автономное устройство, способное выполнять инструкции по меньшей мере из одного потока команд. Поэтому каждую архитектуру процессора, которую мы рассматривали до сих пор, можно квали-

фицировать как ядро. Процессор, имеющий более одного ядра, мы называем *многоядерным*.

Конкретная архитектура ядра может быть любой из тех, что мы рассматривали до сих пор, — ядро способно использовать простой последовательный дизайн, конвейерный дизайн, суперскалярную архитектуру, дизайн VLIW или быть ядром с гиперпоточностью. Рисунок 4.17 иллюстрирует простой пример конструкции многоядерного процессора.

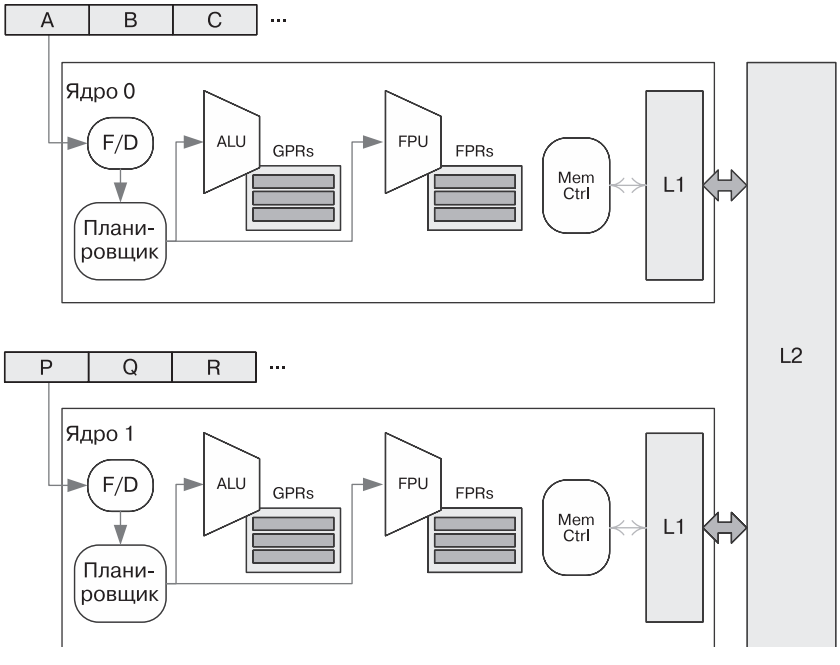


Рис. 4.17. Простой многоядерный процессор

Игровые приставки PlayStation 4 и Xbox One обладают многоядерными процессорами, каждый из которых содержит блок ускоренной обработки (accelerated processing unit, APU), состоящий из двух четырехъядерных модулей AMD Jaguar, интегрированных в один кристалл с графическим процессором, контроллером памяти и видеокодеком. (Из восьми ядер семь доступны для игровых приложений. Однако примерно половина полосы пропускания на седьмом ядре зарезервирована для использования операционной системой.) Xbox One X также содержит восьмиядерный APU, но его ядра основаны на запатентованной технологии, разработанной в сотрудничестве с AMD, а не на микроархитектуре Jaguar, как у его предшественника. На рис. 4.18 показана структурная схема аппаратной архитектуры PS4, а на рис. 4.19 — структурная схема аппаратной архитектуры Xbox One.

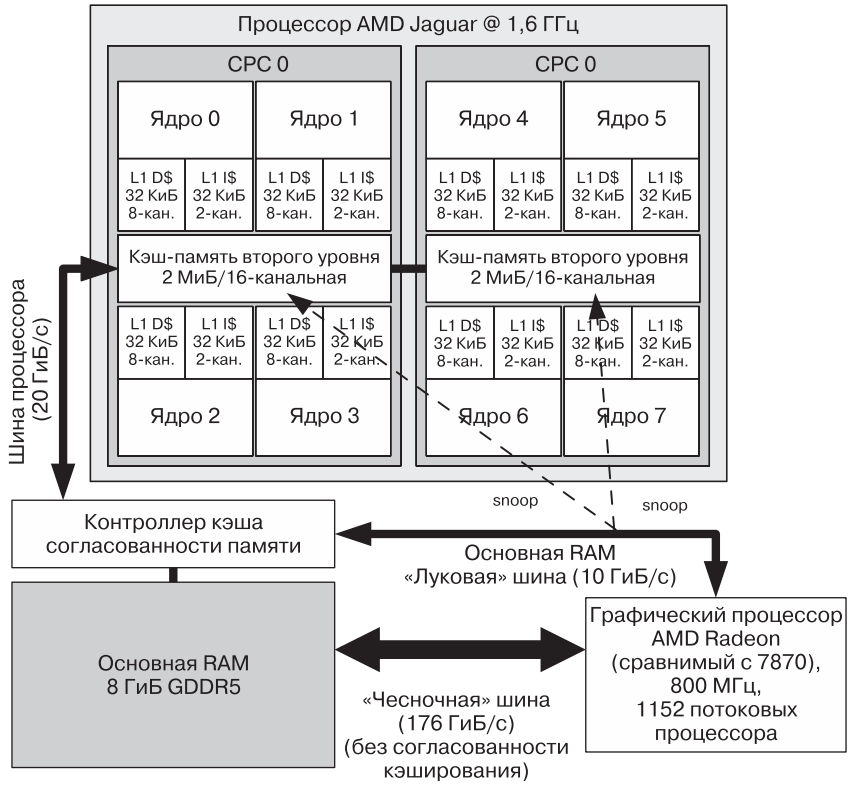


Рис. 4.18. Упрощенное представление архитектуры PS4

4.3.3. Симметричная и асимметричная многопроцессорная обработка

Симметрия платформы параллельных вычислений определяется тем, как операционные системы обрабатывают ядра процессора. При *симметричной многопроцессорной* обработке (symmetric multiprocessing, SMP) доступные ядра ЦП в машине (любая комбинация гиперпоточности, многоядерных ЦП или нескольких ЦП на одной материнской плате) являются однородными с точки зрения архитектуры и ISA и одинаково обрабатываются операционной системой. Любой поток может быть запланирован для выполнения на любом ядре. (Обратите внимание: в таких системах возможно указать *привязку* к потоку, что делает более вероятным или даже гарантированным его исполнение на конкретном ядре.)

PlayStation 4 и Xbox One являются примерами SMP. Обе эти консоли содержат восемь ядер, из которых семь доступны для использования программистом, и приложение может свободно запускать потоки на любом из них.

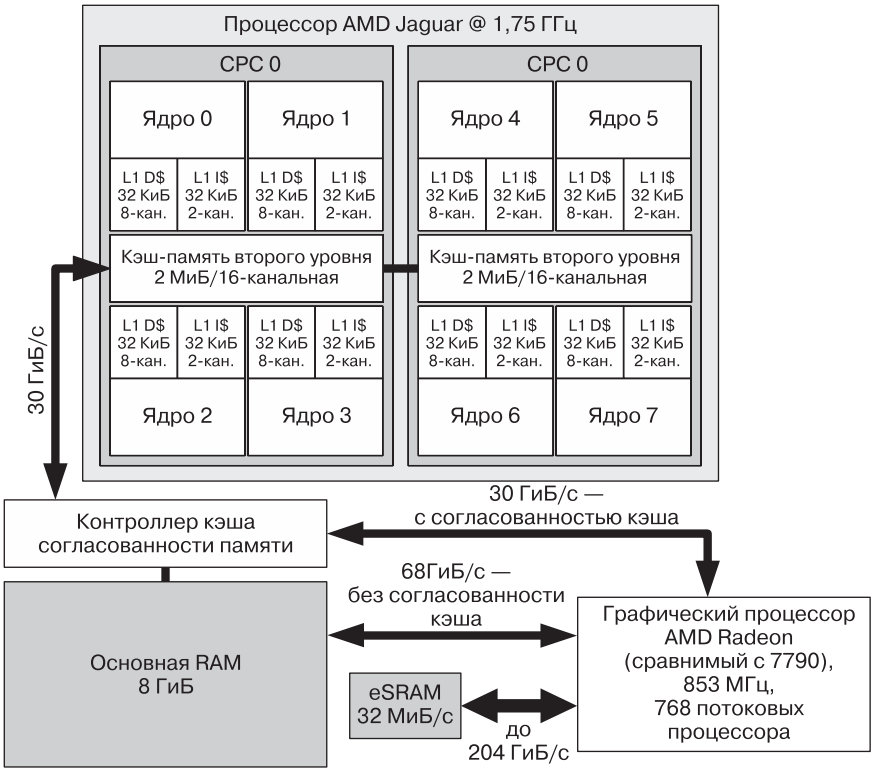


Рис. 4.19. Упрощенный вид архитектуры Xbox One

При *асимметричной многопроцессорной* обработке (asymmetric multiprocessing, AMP) ядра ЦП не обязательно являются однородными, и операционная система не обрабатывает их одинаково. В AMP одно главное ядро ЦП обычно работает под управлением операционной системы, а другие ядра рассматриваются как ведомые, рабочие нагрузки на них распределяются главным ядром.

Архитектура Cell (cell broadband engine, CBE), используемая в PlayStation 3, является примером AMP: в ней задействуется основной ЦП, основанный на ISA PowerPC (POWER processing unit, PPU), а также восемь сопроцессоров, называемых также синергетическими блоками обработки (synergystic processing units, SPU), которые основаны на совершенно другом ISA. (Подробнее об аппаратной архитектуре PS3 говорится в подразделе 3.5.5.)

4.3.4. Распределенные вычисления

Еще один способ достижения параллелизма в вычислениях — использование нескольких автономных компьютеров, работающих совместно. Этот метод называется

распределенными вычислениями в самом общем смысле. Существуют различные способы создания распределенной вычислительной системы, в том числе:

- кластеры компьютеров;
- распределенные вычисления;
- облачные вычисления.

В этой книге мы сосредоточимся исключительно на параллелизме внутри одного компьютера. Узнать больше о распределенных вычислениях вы можете, выполнив поиск упомянутых терминов в Интернете.

4.4. Основы операционной системы

Теперь, хорошо разобравшись в основах параллельного компьютерного оборудования, рассмотрим сервисы, предоставляемые операционной системой, которые делают возможным параллельное программирование.

4.4.1. Ядро

Современные операционные системы решают широкий диапазон самых разных задач от обработки событий клавиатуры и мыши или планирования программ при *вытесняющей многозадачности* до управления очередью принтера или сетевым стеком. Главная часть операционной системы, которая выполняет все основные операции и операции самого нижнего уровня, называется *ядром* (kernel). Остальная часть операционной системы и все пользовательские программы построены на сервисах, предоставляемых ядром. Эта архитектура показана на рис. 4.20.

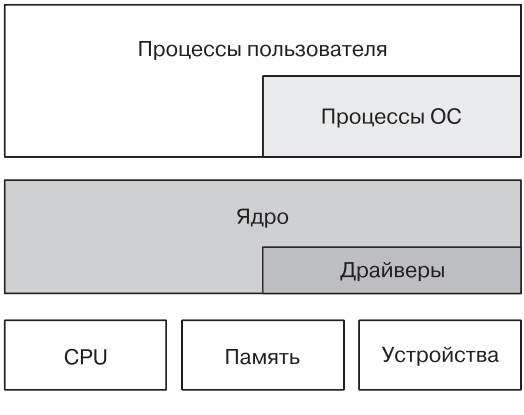


Рис. 4.20. Драйверы ядра и устройств располагаются непосредственно над аппаратными устройствами и работают в привилегированном режиме. Остальное программное обеспечение операционной системы и все пользовательские программы реализованы поверх уровня ядра и драйверов и работают в несколько ограниченном пользовательском режиме

Режим ядра и режим пользователя

Ядро и его устройства работают в специальном режиме, называемом *защищенным режимом*, *привилегированным режимом* или *режимом ядра*, а все прочие программы в системе, включая части операционной системы, которые не являются частью ядра, работают в *пользовательском режиме*.

Как следует из названия, программное обеспечение, работающее в привилегированном режиме, имеет полный доступ ко всему аппаратному обеспечению компьютера, в то время как программное обеспечение пользовательского режима *ограничено* различными способами обеспечения стабильности компьютерной системы в целом. Программное обеспечение, работающее в пользовательском режиме, может получить доступ к низкоуровневым службам, только сделав специальный *вызов ядра* — запрос к ядру выполнить низкоуровневую операцию от имени пользовательской программы. Это гарантирует, что программа не может случайно или намеренно дестабилизировать систему.

На практике операционные системы могут реализовывать несколько *защитных колец*. Ядро работает в кольце 0, самом надежном, которое имеет все возможные привилегии в системе. Драйверы устройств могут работать в кольце 1, доверенные программы с разрешениями ввода-вывода — в кольце 2, а все прочие «ненадежные» пользовательские программы — в кольце 3. Но это только один пример — количество колец варьируется от процессора к процессору и от ОС к ОС, как и распределение подсистем по кольцам. Концепция защитного кольца показана на рис. 4.21.

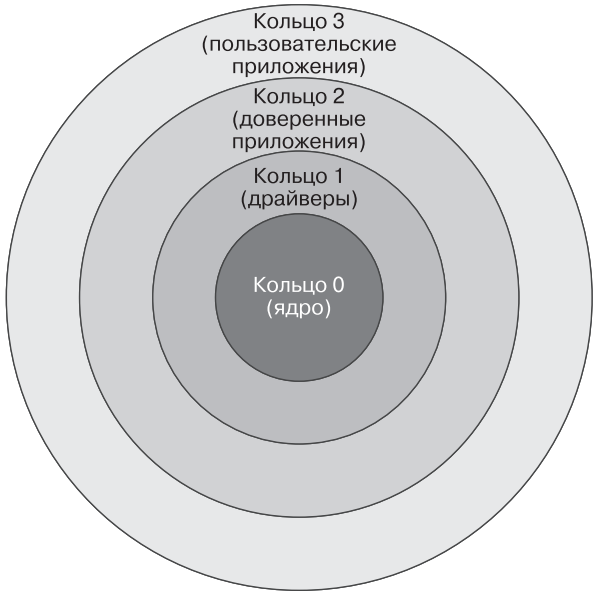


Рис. 4.21. Пример защитных колец процессора, показывающий четыре кольца. Ядро работает в кольце 0, драйверы устройств — в кольце 1, доверенные программы с разрешениями ввода/вывода — в кольце 2, а все остальные пользовательские программы — в кольце 3

Привилегии режима ядра

Программное обеспечение режима ядра (кольцо 0) имеет доступ ко всем инструкциям на машинном языке, определенным ISA ЦП. Сюда входит мощный набор инструкций, называемых *привилегированными* инструкциями. Они могут позволить изменять определенные регистры, которые обычно запрещено изменять (например, для управления отображением виртуальной памяти или маскирования и снятия маскировки прерываний). Или могут предоставить доступ к определенным областям памяти или разрешить выполнить другие, обычно недопустимые операции. Примеры привилегированных инструкций для процессора Intel x86 включают `wrmsr` (запись в особый регистр модели) и `cli` (очистку флага прерываний). То, что использовать эти мощные инструкции может только доверенное программное обеспечение, такое как ядро, повышает стабильность и безопасность системы.

С помощью привилегированных инструкций машинного языка в ядре реализованы меры безопасности. Например, ядро обычно блокирует определенные страницы виртуальной памяти, чтобы в них не могла ничего записать пользовательская программа. Как программное обеспечение ядра, так и все его внутренние данные хранятся на защищенных страницах памяти. Это гарантирует, что пользовательская программа не перезапишет память ядра и тем самым не приведет к краху всей системы.

4.4.2. Прерывания

Прерывание — это сигнал, отправляемый в ЦПУ, чтобы уведомить его о важном низкоуровневом событии, таком как нажатие клавиши на клавиатуре, сигнал от периферийного устройства или истечение таймера. Когда такое событие происходит, возникает *запрос прерывания* (interrupt request, IRQ). Если операционная система желает ответить на событие, она приостанавливает (прерывает) любую выполняемую обработку и вызывает специальный тип функции, называемый *процедурой обработки прерывания* (interrupt service routine, ISR). Функция ISR выполняет какую-то операцию в ответ на событие (в идеале делает это как можно быстрее), а затем управление возвращается обратно программе, которая выполнялась до возникновения прерывания.

Существует два вида прерываний: *аппаратные* и *программные*. Аппаратное прерывание запрашивается подачей ненулевого напряжения на один из контактов ЦП. Аппаратные прерывания могут вызываться такими устройствами, как клавиатура или мышь, либо периодическим таймером на материнской плате или внутри самого ЦП. Поскольку аппаратное прерывание вызывается внешним устройством, оно может произойти в любое время, даже в середине выполнения инструкции CPU. Также может существовать небольшая задержка между моментами, когда аппаратное прерывание физически вызвано и когда процессор находится в подходящем состоянии для его обработки.

Программное прерывание вызывается программным обеспечением, а не напряжением на выводе ЦП. Оно производит тот же базовый эффект, что и аппаратное

прерывание, в том смысле, что вызывает прерывание работы ЦП и сервисную процедуру. Программное прерывание может быть вызвано явным образом выполнением инструкции машинного языка «прерывание». Или в ответ на ошибочное состояние, обнаруженное ЦП во время работы программного обеспечения, — это называется *ловушкой* или иногда *исключением* (хотя последнее не следует путать с обработкой исключений на уровне языка). Например, если ALU необходимо выполнить команду деления на ноль, будет вызвано программное прерывание. Операционная система обычно обрабатывает такие прерывания, прекращая работу программы и создавая файл *дампа ядра*. Однако отладчик, подключенный к программе, может перехватить данное прерывание и вместо этого заставить программу открыть отладчик для проверки.

4.4.3. Вызовы ядра

Чтобы пользовательское программное обеспечение выполнило привилегированную операцию, такую как отображение или удаление страниц физической памяти в системе виртуальной памяти или доступ к необработанному сетевому сокету, пользовательская программа должна сделать запрос к ядру. Ядро отвечает, выполняя операцию безопасным способом от имени пользовательской программы. Такой запрос называется *вызовом ядра* или *системным вызовом*.

В большинстве систем вызов ядра выполняется с помощью программного прерывания[1]. В случае системного вызова, инициируемого прерыванием, пользовательская программа помещает любые входные аргументы в определенное место (либо в памяти, либо в регистрах), а затем вызывает инструкцию «программное прерывание» с целочисленным аргументом, который указывает, какая операция ядра запрашивается. Это приводит к тому, что ЦП переводится в режим с повышенными привилегиями, сохраняет состояние вызывающей программы и затем вызывает соответствующую процедуру обработки прерывания ядра. Предполагая, что ядро разрешает выполнение запроса, оно производит запрошенную операцию (в привилегированном режиме), а затем управление возвращается вызывающей стороне (сперва восстанавливая ее состояние выполнения). Такой переход программы из пользовательского режима в режим ядра является примером *переключения контекста*. (Более подробно о переключении контекста говорится в подразделе 4.4.6.)

В большинстве современных операционных систем пользовательская программа не выполняет программное прерывание или инструкцию системного вызова вручную, например, с помощью встроенного ассемблерного кода. Это было бы грязно и чревато ошибками. Вместо этого пользовательская программа вызывает функцию *API ядра*, которая, в свою очередь, сортирует аргументы и запрашивает программное прерывание. Вот почему системные вызовы выглядят как обычные вызовы функций с точки зрения пользовательской программы.

---

[1] В некоторых системах для вызова ядра используется специальный вариант инструкции вызова call. Например, на процессорах MIPS эта инструкция называется syscall.

4.4.4. Вытесняющая многозадачность

Самые ранние мини-компьютеры и персональные компьютеры могли выполнять одну программу за раз. По сути, они были *последовательными* компьютерами, способными считывать программу из одного потока инструкций и выполнять одну инструкцию из этого потока за раз. *Дисковые операционные системы* (disk operating systems, DOS) в те времена были не более чем драйверами устройств, позволяющими программам взаимодействовать с магнитными лентами, дискетами и жесткими дисками. Весь компьютер выполнял одну-единственную программу. На рис. 4.22 показано несколько полноэкранных программ, работающих на компьютере Apple II.

Рис. 4.22. Три полноэкранные программы, работающие на компьютере Apple II. Программы на Apple II всегда были полноэкранными, потому что единовременно можно было запускать только одну программу. *Слева направо*: Copy II Plus, текстовый процессор AppleWorks и The Locksmith

Поскольку операционные системы и компьютерное оборудование стали более продвинутыми, теперь возможно запускать более одной программы на последовательном компьютере. В распределенных системах мейнфреймов технология, известная как *многозадачный режим*, позволяла одной программе работать, в то время как другая ожидала, что периферийное устройство удовлетворит ее запрос. В классических MacOS и версиях Windows до Windows NT и Windows 95 исполь-

зовалась технология, известная как *совместная многозадачность*: на компьютере одновременно выполнялась бы только одна программа, но каждая программа регулярно *освобождала* ЦП, чтобы другая могла быть запущена. Таким образом, каждая программа заканчивалась периодическим вытеснением процессорного времени. Технически этот метод известен как *мультиплексирование с временным разделением* (time division multiplexing, TDM) или *временная многопоточность* (temporal multithreading, TMT). Неформально он называется *квантованием времени*.

Совместная многозадачность страдала от одной большой проблемы: квантование времени требовало сотрудничества всех программ в системе. Одна мошенническая программа могла занимать все время процессора, если периодически не уступала его другим программам. Операционные системы PDP-6 Monitor и Multics решили эту проблему, внедрив прием, известный как *вытесняющая многозадачность*. Впоследствии эта технология была принята операционной системой UNIX и всеми ее вариантами, а также более поздними версиями Mac OS и Windows.

В вытесняющей многозадачности программы по-прежнему совместно используют ЦП с помощью квантования времени. Однако планирование программ контролируется операционной системой, а не взаимодействием между самими программами. В результате каждая программа регулярно и гарантированно получает временной интервал на процессоре. Временной интервал, в течение которого одной конкретной программе разрешено запускаться на процессоре, иногда называют *квантом* программы. Для реализации вытесняющей многозадачности операционная система реагирует на регулярно синхронизируемое аппаратное прерывание, чтобы периодически *переключать контекст* между различными программами, работающими в системе. Мы более подробно рассмотрим, как работает переключение контекста, в подразделе 4.4.6.

Здесь следует отметить, что вытесняющая многозадачность используется даже на многоядерных машинах, поскольку обычно число потоков больше, чем число ядер. Например, если бы у нас было 100 потоков и только четыре ядра ЦП, то ядро задействовало бы вытесняющую многозадачность для разделения времени между 25 потоками на каждом ядре.

4.4.5. Процессы

Процесс — это способ, применяемый операционной системой для управления запущенным экземпляром программы, содержащейся в *исполняемом файле* (`.exe` в Windows, `.elf` в Linux). Процесс существует только во время работы программы — когда экземпляр программы завершает выполнение (в том числе принудительно) или падает, ОС уничтожает процесс, связанный с ним. В один момент времени в системе может быть запущено несколько процессов, в том числе несколько экземпляров *одной и той же* программы. Программисты взаимодействуют с процессами через API, предоставляемый операционной системой. Детали этого

API различаются в разных ОС, но ключевые концепции почти одинаковы для всех. Исчерпывающее обсуждение API процессов какой-то одной операционной системы выходит за рамки книги, а для иллюстрации концепций мы сосредоточимся прежде всего на API UNIX-подобных операционных систем, таких как Linux, BSD и MacOS. Отдельно остановимся на моментах, где операционные системы Windows или игровых приставок значительно отличаются от основных идей UNIX-подобных API процессов.

Структура процесса

«Под капотом» процесса содержатся:

- *идентификатор процесса* (PID), который однозначно идентифицирует процесс в операционной системе;
- набор *разрешений*, например то, какой пользователь владеет каждым процессом и к какой группе пользователей он принадлежит;
- ссылка на *родительский процесс*, если таковой имеется;
- *пространство виртуальной памяти*, содержащее представление физической памяти процесса (дополнительная информация — в следующем пункте);
- значения всех определенных *переменных окружения*;
- набор всех *дескрипторов открытых файлов*, используемых процессом;
- текущий *рабочий каталог* для процесса;
- ресурсы для управления *синхронизацией* и *связью* между процессами в системе, такими как очереди сообщений, каналы и семафоры;
- один или несколько *потоков*.

Поток инкапсулирует запущенный экземпляр отдельной последовательности выполнения инструкций машинного языка. По умолчанию процесс содержит один поток. Но, как будет сказано в подразделе 4.4.6, в процессе может быть создано более одного потока, что позволяет *одновременно* выполнять более одного потока инструкций. Ядро планирует запуск всех потоков в системе (из всех запущенных в настоящее время процессов) на доступных ядрах. При этом используется вытесняющая многозадачность для разделения времени между потоками, когда их больше, чем ядер.

Здесь следует подчеркнуть, что *основной единицей выполнения программы* в операционной системе являются потоки, а не процессы. Процесс просто обеспечивает *среду*, в которой могут выполняться потоки, включая карту виртуальной памяти и набор *ресурсов*, используемые и разделяемые всеми потоками в этом процессе. Всякий раз, когда поток планируется запустить на ядре, его процесс активизируется и его ресурсы и среда становятся доступными для применения данным потоком. Поэтому, когда мы говорим, что *поток* работает на ядре, помните: он всегда делает это в *контексте* ровно одного процесса.

Отображение виртуальной памяти процесса

Как вы, надеюсь, помните, программа обычно не работает напрямую с адресами физической памяти[1]. Скорее, она обращается к памяти в виде *виртуальных адресов*, а ЦП и операционная система взаимодействуют, чтобы преобразовать эти виртуальные адреса в физические.Было сказано, что переназначение виртуальных адресов физическими происходит в терминах смежных блоков адресов, называемых *страницами*, и что *таблица страниц* используется ОС для отображения индексов виртуальных страниц на индексы физических страниц.

Каждый процесс имеет собственную таблицу виртуальных страниц. Это означает, что у него есть собственное *представление* в памяти. Это один из основных способов, с помощью которого операционная система обеспечивает безопасную и стабильную среду выполнения. Два процесса не могут повредить память друг друга, потому что физические страницы, принадлежащие одному из них, просто не отображаются в адресном пространстве другого (если они явно не разделяют страницы). Кроме того, страницы, принадлежащие ядру, защищены как от непреднамеренного, так и от преднамеренного повреждения пользовательским процессом, поскольку сопоставлены со специальным диапазоном адресов, известным как *пространство ядра*, к которому может обращаться только код, работающий в режиме ядра.

Таблица виртуальных страниц процесса эффективно определяет его *карту памяти*. Карта памяти обычно содержит:

- разделы кода, данных и BSS, считанные из исполняемого файла программы;
- любые общие библиотеки (DLL, PRX), применяемые программой;
- *стек вызовов* для каждого потока;
- область памяти, называемую *кучей*, для динамического выделения памяти;
- некоторые страницы памяти, используемые *совместно* с другими процессами;
- диапазон адресов *пространства ядра*, недоступных для процесса, но становящихся доступными всякий раз, когда выполняется вызов ядра.

Разделы кода, данных и BSS. Когда программа запускается впервые, ядро создает внутри себя процесс и присваивает ему уникальный PID. После чего устанавливает виртуальную карту страниц для процесса, другими словами, создает виртуальное адресное пространство процесса. Затем оно распределяет физические страницы по мере необходимости и отображает их в виртуальное адресное пространство, добавляя записи в таблицу страниц процесса.

Ядро считывает исполняемый файл (код, данные и разделы BSS) в память, выделяя виртуальные страницы и загружая в них данные. Это позволяет программному коду и глобальным данным быть видимыми в виртуальном адресном пространстве

[1] Пользовательские программы всегда работают, ориентируясь на адреса виртуальной памяти, но ядро может работать напрямую с физическими адресами.

процесса. Машинный код в исполняемом файле фактически *перемещаем*, что означает: его адреса указываются как относительные смещения, а не как абсолютные адреса памяти. Эти относительные адреса *фиксируются* операционной системой, то есть преобразуются обратно в реальные (виртуальные) адреса до запуска программы. (Подробнее о формате исполняемого файла см. в подразделе 3.3.5.)

Стек вызовов. Каждый работающий поток нуждается в стеке вызовов (см. подраздел 3.3.5). Когда процесс запускается впервые, ядро создает для него один поток по умолчанию. Ядро выделяет страницы физической памяти для стека вызовов этого потока и отображает их в виртуальное адресное пространство процесса, чтобы поток мог видеть стек. Значения указателя стека (SP) и базового указателя (BP) инициализируются так, чтобы указывать на дно пустого стека. Наконец, поток начинает выполняться с точки входа в программу. (В C/C++ это обычно `main()` или `WinMain()` для Windows.)

Куча. Процессы могут динамически распределять память через `malloc()` и `free()` в C или глобальные `new` и `delete` в C++. Данные запросы поступают из области памяти, называемой *кучей*. Физические страницы памяти выделяются ядром по требованию для выполнения запросов динамического выделения. Эти страницы отображаются в виртуальном адресном пространстве процесса как выделенная память, а страницы, содержимое которых полностью освобождено, больше не отображаются и возвращаются в систему.

Общие библиотеки. Все нетривиальные программы зависят от внешних библиотек. Библиотека может быть *статически* связана с программой, что означает: *копия* кода библиотеки помещается в исполняемый файл. Большинство операционных систем поддерживают также концепцию *общих библиотек*. В этом случае программа содержит только *ссылки* на функции API библиотеки, а не копию ее машинного кода. Совместно используемые библиотеки называются *динамически подключаемыми* (DLL) в Windows. В PlayStation 4 ОС поддерживает своего рода динамически связанную библиотеку, называемую PRX. (Интересно, что название PRX происходит из PlayStation 3, где оно обозначалось как перемещаемый исполняемый файл PPU (relocatable executable) по отношению к основному процессору в PS3, который назывался PPU.)

Общие библиотеки обычно работают следующим образом. Когда у процесса в первый раз возникает необходимость в общей библиотеке, ОС загружает эту библиотеку в физическую память и отображает ее представление в виртуальном адресном пространстве процесса. Адреса функций и глобальных переменных, предоставляемых совместно используемой библиотекой, вставляются в машинный код программы, что позволяет ей вызывать их так, как если бы они были статически связаны с исполняемым файлом.

Преимущество разделяемых библиотек становится очевидным только при запуске второго процесса, использующего ту же общую библиотеку. Вместо загрузки *копии* кода библиотеки и глобальных переменных уже загруженные физические страницы отображаются в виртуальное адресное пространство нового процесса. Это экономит память и ускоряет запуск всех процессов, кроме первого, задействующего данную общую библиотеку.

Совместно используемые библиотеки имеют и другие преимущества. Например, общая библиотека может быть обновлена, скажем, для исправления некоторых ошибок, и теоретически все программы, использующие ее, сразу же получат выгоду (не потребуются повторная компоновка и распространение среди пользователей). Тем не менее на практике обновление общих библиотек может невольно вызвать проблемы совместимости программ, которые их применяют. Это приводит к распространению различных версий каждой совместно используемой библиотеки в системе — данную ситуацию разработчики Windows ласково называют адом DLL. Чтобы обойти эти проблемы, Windows перешла к системе манифестов, которые помогают гарантировать совместимость между общими библиотеками и программами, которые их задействуют.

Страницы ядра. В большинстве операционных систем адресное пространство процесса фактически разделено на два больших смежных блока — пространство пользователя и пространство ядра. Например, в 32-разрядной Windows пользовательское пространство соответствует диапазону адресов от 0x0 до 0x7FFFFFFF (нижние 2 ГиБ адресного пространства), а пространство ядра — адресам 0x80000000–0xFFFFFFFF (верхние 2 ГиБ). В 64-битной Windows пользовательское пространство соответствует диапазону адресов 8 ТиБ от 0x0 до 0x7FF'FFFFFFFF, а гигантский диапазон 248 ТиБ от 0xFFFF0800'00000000 до 0xFFFFFFFF'FFFFFFFF зарезервирован для использования ядром (хотя он задействуется не весь).

Пространство пользователя отображается с помощью таблицы виртуальных страниц, уникальной для каждого процесса. Однако в пространстве ядра применяется отдельная таблица виртуальных страниц, которую используют все процессы. Это делается для того, чтобы все процессы в системе имели согласованное представление о внутренних данных ядра.

Обычно пользовательские процессы не имеют доступа к страницам ядра — если они попытаются его получить, произойдет сбой страницы и, соответственно, сбой программы. Но когда пользовательский процесс выполняет системный вызов, выполняется *переключение контекста* в ядро (см. подраздел 4.4.6). Это переводит ЦП в привилегированный режим, позволяя ядру получать доступ к диапазонам адресов пространства ядра, а также к виртуальным страницам текущего процесса. Ядро запускает свой код в привилегированном режиме, обновляет свои внутренние структуры данных по мере необходимости и, наконец, переводит ЦП обратно в пользовательский режим и возвращает управление пользовательской программе. Для получения дополнительной информации о том, как сопоставление памяти пространства пользователя и ядра работает в Windows, найдите тему «Виртуальные адресные пространства» на hdocs.microsoft.com.

Интересно (и немного пугающе!) отметить, что недавно обнаруженные эксплойты Meltdown и Spectre используют внеочередную и спекулятивную логику выполнения ЦП соответственно, чтобы обманным путем заставить его обращаться к данным, расположенным на страницах памяти, которые обычно защищены от пользовательских процессов. Подробнее об этих уязвимостях и о том, как операционные системы защищаются от них, см. на hmeltdownattack.com.

Пример карты памяти процесса. На рис. 4.23 изображена карта памяти процесса, какой она может быть для 32-битной Windows. Все виртуальные страницы процесса отображаются в пользовательское пространство — нижние 2 ГиБ адресного пространства. Сегменты кода, данных и BSS исполняемых файлов отображаются в низшие адреса памяти, затем в более высоком диапазоне следует куча, за которой — любые страницы общей памяти. Стек вызовов отображается в верхний конец адресного пространства пользователя. Наконец, страницы ядра операционной системы отображаются в верхние 2 ГиБ адресного пространства.

Фактические адреса каждой из областей памяти непредсказуемы. Так происходит отчасти потому, что сегменты каждой программы имеют разные размеры и, следовательно, будут отображаться в разных диапазонах адресов. Кроме того, числовые значения адресов фактически изменяются между запусками одной и той же исполняемой программы благодаря функции безопасности, известной как *рандомизация расположения адресного пространства* (address space layout randomization, ASLR).

Рис. 4.23. Карта памяти процесса, как она может выглядеть для 32-битной Windows

4.4.6. Потоки

Поток инкапсулирует запущенный экземпляр отдельного потока инструкций машинного языка. Каждый поток процесса состоит:

- из *идентификатора потока* (TID), уникального в своем процессе, но не обязательно уникального во всей операционной системе;
- *стека вызовов* потока — непрерывного блока памяти, содержащего кадры стека всех выполняемых в данный момент функций;
- значений всех *регистров*[1] специального и общего назначения, включая указатель инструкций (IP), который ссылается на текущую инструкцию в потоке, базовый указатель (BP) и указатель стека (SP), которые определяют кадр стека текущей функции;
- блока памяти общего назначения, связанного с каждым потоком и известного как *локальное хранилище потока* (thread local storage, TLS).

[1] Технически контекст выполнения потока включает только значения регистров, которые видны в пользовательском режиме. Он исключает значения некоторых регистров привилегированного режима.

По умолчанию процесс содержит один *основной поток* и, следовательно, выполняет один поток команд. Этот поток начинает выполняться в точке входа программы — обычно это функция `main()`. Однако все современные операционные системы способны выполнять более одного *параллельного* потока инструкций в контексте одного процесса.

Вы можете рассматривать поток как *основную единицу выполнения* в операционной системе. Поток предоставляет минимальные ресурсы, необходимые для выполнения потока инструкций, — стек вызовов и набор регистров. Процесс просто обеспечивает *окружение*, в котором выполняется один или несколько потоков. Это продемонстрировано на рис. 4.24.

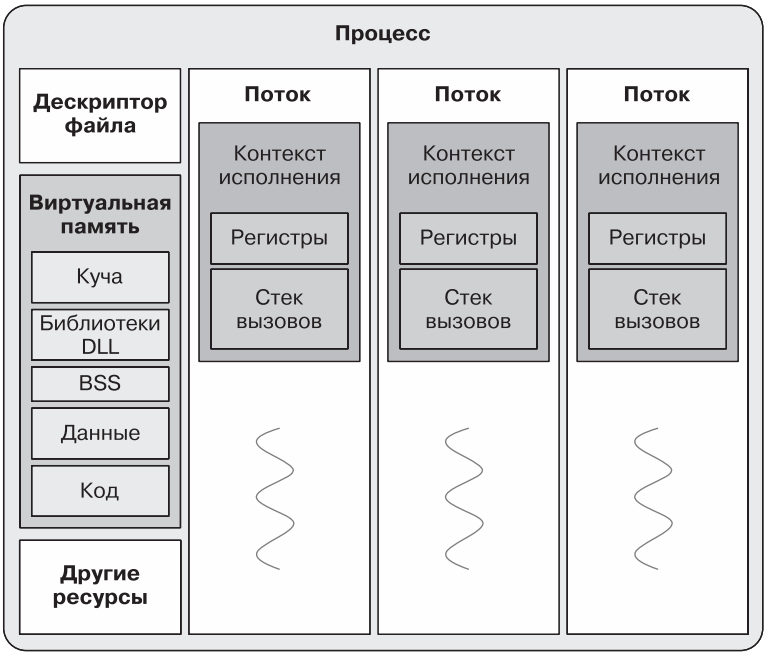


Рис. 4.24. Процесс инкапсулирует ресурсы, необходимые для запуска одного или нескольких потоков. Каждый поток инкапсулирует контекст выполнения, состоящий из содержимого регистров ЦП и стека вызовов

Библиотеки потоков

Все операционные системы, поддерживающие многопоточность, предоставляют набор системных вызовов для создания потоков и управления ими. Доступны также несколько портативных библиотек потоков, среди которых наиболее известны библиотека стандартных потоков IEEE POSIX 1003.1c (pthread) и стандартные библиотеки потоков C11 и C++11. SDK Sony PlayStation 4 предоставляет набор функций потоков с префиксом `sce`, которые в значительной степени отображаются непосредственно в API потоков POSIX.

API потоков различаются деталями реализации, но все они обеспечивают выполнение следующих основных операций.

- *Создать* — функция или конструктор класса, который порождает новый поток.
- *Завершить* — функция, которая завершает вызывающий поток.
- *Запрос на выход* — функция, которая позволяет одному потоку запросить завершение другого потока.
- *Спать* — функция, которая переводит действующий поток в спящий режим на определенный промежуток времени.
- *Уступить* — функция, которая возвращает остаток временного интервала потока, чтобы другие потоки могли получить шанс на запуск.
- *Объединить* — функция, которая переводит вызывающий поток в спящий режим до тех пор, пока не завершится другой поток или группа потоков.

Создание и уничтожение потоков

Когда запускается исполняемый файл, процесс, созданный ОС для его инкапсуляции, автоматически содержит один поток, выполнение которого начинается в *точке входа* программы — в C/C++ это специальная функция `main()`. Этот «основной поток», если это необходимо, может создавать новые потоки вызовом функций, специфичных для операционной системы, таких как `pthread_create()` (потоки POSIX) и `CreateThread()` (Windows), или созданием экземпляра класса потока, такого как `std::thread` (C ++11). Новый поток начинает выполнение с функции точки входа, адрес которой предоставляется вызывающей стороной.

После создания поток будет продолжать существовать, пока не завершит работу. Выполнение потока может быть прекращено несколькими способами.

- Он может закончиться естественно, *вернувшись* из функции точки входа. (Если это основной поток, то возврат из `main()` завершает не только поток, но и весь процесс.)
- Он может вызвать функцию, такую как `pthread_exit()`, чтобы *явно* прекратить выполнение, *прежде чем* вернуться из своей функции точки входа.
- Он может быть *убит* извне другим потоком. В этом случае внешний поток делает *запрос* на уничтожение рассматриваемого потока, но он может не сразу ответить на запрос или полностью его проигнорировать. Возможность такого завершения по внешнему запросу определяется при создании потока.
- Его можно принудительно убить, потому что его процесс завершен. (Процесс завершается, когда основной поток возвращается из функции точки входа `main()`, когда любой поток вызывает функцию `exit()` для явного уничтожения процесса или внешний субъект убивает процесс.)

Объединение потоков

Обычно один поток порождает один или несколько дочерних потоков, выполняет некую полезную работу самостоятельно и затем ждет продолжения выполнения дочерних потоков и их работы. Предположим, что основной поток хочет выполнить 1000 вычислений и программа работает на четырехъядерном компьютере. Наиболее эффективным подходом было бы разделить работу на четыре части одинакового размера и создать четыре потока для параллельной обработки. Предположим, что, как только вычисления завершатся, основной поток должен вычислить контрольную сумму результатов. Полученный код может выглядеть примерно так:

```
ComputationResult g_aResult[1000];

void Compute(void* arg)
{
    uintptr_t startIndex = (uintptr_t)arg;
    uintptr_t endIndex = startIndex + 250;
    for (uintptr_t i = startIndex; i < endIndex; ++i)
    {
        g_aResult[i] = ComputeOneResult(...);
    }
}

void main()
{
    pthread_t  tid[4];

    for (int i = 0; i < 4; ++i)
    {
        const uintptr_t startIndex = i * 250;
        pthread_create(&tid[i], nullptr,
                       Compute, (void*)startIndex);
    }

    // возможно, делаем другую полезную работу...

    // ждем завершения вычислений
    for (int i = 0; i < 4; ++i)
    {
        pthread_join(&tid[i], nullptr);
    }

    // все потоки завершены, поэтому можем посчитать нашу контрольную сумму
    unsigned checksum = Sha1(g_aResult,
        1000*sizeof(ComputationResult));

    // ...
}
```

Опрос, блокировка и освобождение

Обычно поток работает до тех пор, пока не завершится. Но иногда работающему потоку нужно ждать какого-то будущего события. Например, потоку может потребоваться дождаться завершения трудоемкой операции или доступности какого-либо ресурса. В такой ситуации есть три варианта поведения потока.

1. Поток может *опрашивать*.
2. Он может *заблокироваться*.
3. Он может *уступить* выполнение другому потоку на время ожидания.

Опрос. Опрос заключается в том, что поток находится в цикле, ожидая, что условие станет истинным. Это очень похоже на то, как дети на заднем сиденье в поездке постоянно спрашивают: «Мы уже приехали? Мы уже приехали?» Пример:

```
// ждем, пока условие не станет истинным
while (!CheckCondition())
{
    // бьем баклуши
}

// теперь условие верно, и мы можем продолжить...
```

Этот подход иногда называют *вращающимся* или *циклическим ожиданием*.

Блокировка. Если мы предполагаем, что поток довольно долго будет ожидать выполнения какого-то условия, это не самый хороший вариант. В идеале хотелось бы перевести поток в спящий режим, чтобы он не тратил ресурсы ЦП, и положиться на то, что ядро разбудит его, когда условие станет истинным. Это называется *блокировкой* потока.

Поток блокируется с помощью специального вида вызова операционной системы, известного как *блокирующая функция*. Если в момент вызова блокирующей функции условие уже выполнено, она фактически не будет блокировать — просто сразу вернет управление. Но если это не так, ядро переводит поток в спящий режим и добавляет поток и условие, которое он ожидает, в таблицу. Позже, когда условие становится истинным, ядро использует эту внутреннюю таблицу для идентификации и пробуждения любых потоков, ожидающих данного условия.

В ОС существуют различные виды блокирующих функций. Вот несколько примеров.

- *Открытие файла*. Большинство функций, открывающих файл, таких как `fopen()`, блокируют вызывающий поток до тех пор, пока файл не будет фактически открыт (что может занять сотни или даже тысячи циклов). Некоторые функции, такие как `open()` в Linux, предлагают неблокирующую опцию (`O_NONBLOCK`) для поддержки асинхронного файлового ввода-вывода.
- *Явный сон*. Некоторые функции явно переводят вызывающий поток в спящий режим на определенный промежуток времени. Варианты включают `usleep()` (Linux), `Sleep()` (Windows), `std::this_thread::sleep_until()` (стандартная библиотека C++11) и `pthread_sleep()` (потоки POSIX).

- *Объединение с другим потоком.* Функция `pthread_join()` блокирует вызывающий поток до тех пор, пока не будет завершен поток, завершение которого ожидается.
- *Ожидание блокировки мьютекса.* Такие функции, как `pthread_mutex_wait()`, пытаются получить монопольную блокировку ресурса через объект операционной системы, известный как *мьютекс* (см. раздел 4.6). Если ни один прочий поток не удерживает ресурс, функция предоставляет блокировку вызывающему потоку и возвращает управление немедленно. В противном случае вызывающий поток переводится в спящий режим до тех пор, пока не будет получена блокировка ресурса.

Вызовы операционной системы — не единственные функции, которые способны блокировать. Любая функция пользовательского пространства, которая в конечном итоге вызывает блокирующую функцию ОС, считается блокирующей. Такую функцию стоит документировать, чтобы программисты, которые ее используют, знали, что она может блокировать.

Освобождение. Этот метод — что-то среднее между опросом и блокировкой. Поток опрашивает условие в цикле, но на каждой итерации освобождает остаток своего временного интервала, вызывая `pthread_yield()` (POSIX), `Sleep(0)` или `SwitchToThread()` (Windows) либо делая эквивалентный системный вызов. Пример:

```
// ждем, пока условие не станет истинным
while (!CheckCondition())
{
    // бьем баклуши
    pthread_yield(nullptr);
}

// теперь условие верно, и мы можем продолжить...
```

Такой подход способен уменьшать количество потерянных циклов и улучшать потребление энергии по сравнению с чистым циклом ожидания занятости.

Освобождение CPU по-прежнему требует вызова ядра и, следовательно, является довольно дорогостоящим. Некоторые процессоры предоставляют легкую инструкцию паузы. (Например, на Intel x86 ISA с SSE2 такую инструкцию выполняет встроенная функция `_mm_pause()`.) Этот вид инструкций уменьшает энергопотребление цикла ожидания занятости, просто указывая ожидать, пока конвейер команд ЦП не опустеет перед тем, как продолжить выполнение:

```
// ждем, пока условие не станет истинным
while (!CheckCondition())
{
    // только Intel SSE2:
    // уменьшаем энергопотребление, делая паузу примерно на 40 циклов
    _mm_pause();
}

// теперь условие верно, и мы можем продолжить...
```

На software.intel.com/en-us/comment/1134767 и software.intel.com/en-us/forums/topic/309231 приводится углубленное обсуждение того, как и зачем использовать инструкцию паузы в цикле ожидания.

Переключение контекста

Каждый поток, поддерживаемый ядром, существует в одном из трех состояний[1]:

- *выполнение (running)* — поток активно работает на ядре;
- *готов к выполнению (runnable)* — поток в состоянии работать, но ожидает получения кванта времени на ядре;
- *заблокирован (blocked)* — поток спит, ожидая, что какое-то условие станет истинным.

Переключение контекста происходит всякий раз, когда ядро вызывает переход потока из одного из перечисленных состояний в другое.

Переключение контекста всегда происходит в привилегированном режиме работы процессора в ответ на аппаратное прерывание, которое управляет вытесняющей многозадачностью (то есть переходами между «выполняется» и «готов к выполнению»), в ответ на явный блокирующий вызов ядра работающим потоком (то есть переходом из состояния «выполняется» или «готов к выполнению» в «заблокирован») или в ответ на выполнение условия ожидания, таким образом пробуждая спящий поток, то есть переводя его из «заблокирован» в «готов к выполнению»). Конечный автомат (машина состояний) потоков ядра показан на рис. 4.25.

Когда поток находится в состоянии выполнения, он активно задействует ядро процессора. Регистры ядра содержат информацию, относящуюся к выполнению потока, такую как указатель его инструкции (IP), указатели стека и базовый указатель (SP и BP) и содержимое различных регистров общего назначения (GPR). Поток также поддерживает стек вызовов, в котором хранятся локальные переменные и адреса возврата для выполняемой в данный момент функции, и весь стек функций, которые в конечном итоге вызвали ее. Вся эта информация известна как *контекст выполнения* потока.

Всякий раз, когда поток переходит из состояния «выполняется» в состояние «готов к выполнению» или «заблокирован», содержимое регистров ЦП сохраняется в блоке памяти, который был зарезервирован для потока ядром. Позже, когда поток из «готов к выполнению» возвращается в состояние «выполняется», ядро снова заполняет регистры процессора сохраненным содержимым регистров этого потока.

Здесь следует отметить, что стек вызовов потока не нужно сохранять или восстанавливать явно во время переключения контекста. Это происходит потому, что стек вызовов каждого потока уже находится в отдельной области в карте виртуальной памяти процесса. Акт сохранения и восстановления содержимого регистров

[1] Некоторые операционные системы используют дополнительные состояния, но это детали реализации, которые мы можем игнорировать.

ЦП включает в себя сохранение и восстановление указателя стека и базового указателя (SP и BP) и, таким образом, эффективно сохраняет и восстанавливает стек вызовов потока «бесплатно».

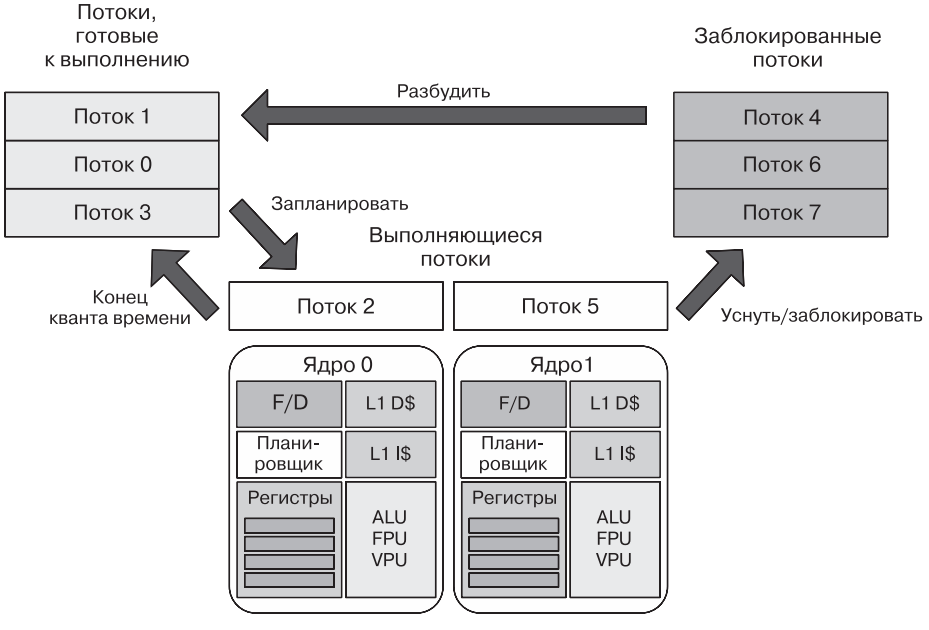


Рис. 4.25. Каждый поток может находиться в одном из трех состояний: «выполняется», «готов к выполнению», «заблокирован»

Если во время переключения контекста входящий поток находится в процессе, отличном от процесса исходящего потока, ядру необходимо также сохранить состояние карты виртуальной памяти исходящего процесса и настроить карту виртуальной памяти нового процесса. Из подраздела 3.5.2 вы помните, что карта виртуальной памяти определяется таблицей виртуальных страниц. Поэтому сохранение и восстановление карты виртуальной памяти включает сохранение и восстановление указателя на эту таблицу страниц, которая обычно поддерживается в специальном привилегированном регистре ЦП. Буфер ассоциативной трансляции (translation lookaside buffer, TLB) необходимо очищать всякий раз, когда происходит переключение контекста между процессами. Эти дополнительные шаги делают переключение контекста между процессами более дорогостоящим, чем переключение контекста между потоками в рамках одного процесса.

Приоритет потоков и привязка

По большей части ядро планирует работу потоков на доступных ядрах в машине. У программистов есть два пути влияния на то, в каком порядке запускаются потоки: *приоритет* и *привязка*.

Приоритет потока контролирует его запуск относительно других потоков, готовых к запуску в системе. Потоки с более высоким приоритетом обычно имеют преимущество над потоками с более низким приоритетом. Различные операционные системы предлагают разное количество уровней приоритета. Например, потоки Windows могут принадлежать одному из шести классов приоритета, а в каждом классе — семь уровней приоритета. Эти два значения дают в общей сложности 42 базовых приоритета, которые используются при планировании потоков.

Простейшее правило планирования потоков таково: пока существует хотя бы один готовый к запуску поток с более высоким приоритетом, запуск потоков с более низким приоритетом не планируется. Суть этого подхода заключается в том, что большинство потоков в системе будут создаваться с определенным уровнем приоритета по умолчанию и, следовательно, совместно использовать ресурсы обработки. Но время от времени поток с более высоким приоритетом может войти в состояние «готов к выполнению». Когда это происходит, он запускается настолько рано, насколько возможно, завершается через довольно короткий промежуток времени и тем самым возвращает управление всем потокам с более низким приоритетом.

Такой простой алгоритм планирования на основе приоритетов может привести к возникновению ситуации, когда небольшое количество потоков с высоким приоритетом работает непрерывно, тем самым предотвращая запуск любых потоков с более низким приоритетом. Эта ситуация известна как *ресурсный голод*. Некоторые операционные системы пытаются смягчить пагубные последствия голодания введением в простое правило планирования исключений, целью которых является выделение какого-то времени ЦП для голодающих потоков с более низким приоритетом.

Еще один способ, с помощью которого программисты могут управлять планированием потоков, — *привязка* потоков. Этот параметр требует, чтобы либо ядро заблокировало поток для определенного ядра, либо при планировании поток по крайней мере *предпочел* одно или несколько ядер другим.

Локальное хранилище потока

Уже говорилось, что все потоки внутри процесса совместно используют его ресурсы, включая пространство виртуальной памяти. Из этого правила есть одно исключение — каждому потоку предоставляется отдельный блок памяти, называемый *локальным хранилищем потока* (thread local storage, TLS). Это позволяет потокам отслеживать данные, которые не должны передаваться другим процессам. Например, каждый поток может поддерживать частный распределитель памяти. Мы можем думать о блоке памяти TLS как о части контекста выполнения потока.

На практике блоки памяти TLS обычно видны всем потокам в процессе. Как правило, они не защищены так, как страницы виртуальной памяти операционной системы. Вместо этого ОС обеспечивает каждому потоку собственный блок TLS, все они отображаются в виртуальное адресное пространство процесса с разными числовыми адресами и им предоставляется системный вызов, который позволяет любому потоку получить адрес его частного блока TLS.

Отладка потоков

Сейчас все хорошие отладчики предоставляют инструменты для отладки многопоточных приложений. В Microsoft Visual Studio *окно потоков (threads)* является центральным инструментом для этой цели. Всякий раз, когда вы входите в отладчик, в этом окне перечисляются все потоки, существующие в настоящее время в приложении. Двойной щелчок на потоке делает его контекст выполнения активным в отладчике. После активации контекста потока вы можете перемещаться вверх и вниз по его стеку вызовов в окне стека вызовов (call stack) и просматривать локальные переменные в области действия каждой функции в окне Watch. Это работает, даже если поток находится в состоянии «готов к запуску» или «заблокирован». Окно Threads Visual Studio показано на рис. 4.26.

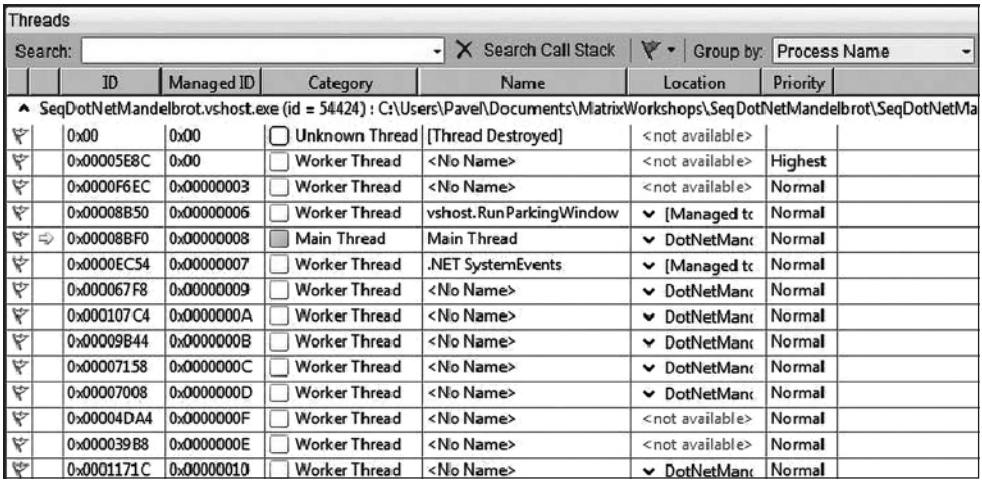


Рис. 4.26. Окно Threads в Visual Studio является основным интерфейсом для отладки многопоточных программ

4.4.7. Фиберы

В вытесняющей многозадачности планирование потоков обрабатывается ядром автоматически. Зачастую это удобно, но иногда программисты считают необходимым контролировать планирование рабочих нагрузок в своих программах. Например, при реализации *системы заданий* для игрового движка (обсуждается в подразделе 8.6.4) мы могли бы разрешить заданиям явно уступать ресурсы ЦП другим заданиям, не беспокоясь о том, что в момент выполнения они будут выдергивать ковер из-под ног друг у друга. Иными словами, иногда мы хотим использовать *совместную*, а не вытесняющую многозадачность.

Некоторые операционные системы предоставляют именно такой кооперативный механизм многозадачности, называемый *фибером* (fiber). Фибер очень похож на поток в том смысле, что представляет собой запущенный экземпляр потока

инструкций машинного языка. Фибер имеет стек вызовов и состояние регистра (контекст выполнения), как и поток. Но между ними есть большая разница: фиберы никогда не распределяются напрямую ядром. Вместо этого они работают в контексте потока и планируются совместно друг с другом.

В этом подразделе мы поговорим конкретно о фиберах Windows. Другие операционные системы, такие как Sony PlayStation 4 SDK, предоставляют очень похожие фибер-API.

Создание и уничтожение фиберов

Как можно преобразовать процесс на основе потока в процесс на основе фибера? Каждый процесс при первом запуске начинается с одного потока, следовательно, процессы основаны на потоках по умолчанию. Когда поток вызывает функцию `ConvertThreadToFiber()`, новый фибер создается в контексте вызывающего потока. Это запускает процесс создания и выполнения большего количества фиберов. Другие фиберы создаются путем вызова `CreateFiber()` и передачи ей адреса функции, которая будет служить точкой входа. Любой работающий фибер может совместно запланировать запуск другого фибера в своем потоке, вызвав `SwitchToFiber()`. Когда фибер больше не нужен, его можно уничтожить, вызвав `DeleteFiber()`.

Состояния фиберов

Фибер может находиться в одном из двух состояний: активном или неактивном. Когда фибер в активном состоянии, он назначается потоку и выполняется от его имени. А когда в неактивном — находится в стороне, не потребляя ресурсы какого-либо потока, и просто ожидает активации. Операционная система Windows называет активный фибер *выбранным* для данного потока.

Активный фибер может деактивировать себя и сделать активным другой фибер, вызвав `SwitchToFiber()`. Это единственный способ переключения фиберов из активного состояния в неактивное и наоборот.

Выполняется или нет активный фибер на ядре ЦП, определяется состоянием окружающего потока. Когда поток активного фибера находится в состоянии выполнения, инструкции на машинном языке этого фибера выполняются на ядре. Когда поток активного фибера находится в состоянии «готов к выполнению» или «заблокирован», его инструкции, конечно, не могут быть выполнены, потому что весь поток находится в стороне: ожидает либо планирования на ядре, либо выполнения условия.

Важно понимать, что фиберы сами по себе не имеют заблокированного состояния, как потоки. Другими словами, невозможно перевести фибер в состояние ожидания. Можно усыпить только создавший его поток. Из-за этого ограничения всякий раз, когда фиберу нужно ждать, пока условие станет истинным, он либо ожидает занятости, либо вызывает `SwitchToFiber()`, чтобы на время ожидания передать управление другому фиберу. Выполнение блокирующего вызова ОС из

фибера, как правило, довольно грубая ошибка. Это приведет к тому, что поток, оборачивающий фибер, перейдет в спящий режим, что не позволит этому фиберу делать что-либо, включая вызов других фиберов для совместной работы.

Миграция фиберов

Фибер может мигрировать из потока в поток, но только в неактивном состоянии. В качестве примера рассмотрим фибер F, работающий в контексте потока A. Фибер F вызывает `SwitchToFiber(G)` для активации внутри потока A фибера G. Это переводит фибер F в неактивное состояние (то есть он больше не связан ни с каким потоком). Теперь предположим, что поток с именем B выполняет фибер H. Если фибер H вызывает `SwitchToFiber(F)`, значит, фибер F эффективно мигрировал из потока A в поток B.

Отладка при использовании фиберов

Поскольку фиберы предоставляются ОС, инструменты отладки и профилирования должны иметь возможность видеть их так, как они могут видеть потоки. Например, при отладке на PS4 с помощью подключаемого модуля отладчика Visual Studio SN Systems для Clang фиберы автоматически отображаются в окне потоков, как если бы они были потоками. Вы можете дважды щелкнуть на фибере, чтобы активировать его в окнах Watch и Call Stack, а затем пройти вверх и вниз по стеку вызовов, как обычно делаете с потоком.

Если вы планируете использовать фиберы в игровом движке, рекомендуется проверить возможности вашего отладчика на целевой платформе, прежде чем тратить много времени и усилий на проектирование на основе фиберов. Если ваш отладчик и/или целевая платформа не предоставляют хороших инструментов для отладки фиберов, это может все испортить.

Дополнительная литература о фиберах

Больше о фиберах Windows можно прочитать здесь: msdn.microsoft.com/ru-ru/library/windows/desktop/ms682661(v=vs.85).aspx.

4.4.8. Потоки пользовательского уровня и корутины

Как потоки, так и фиберы могут быть довольно тяжелыми, потому что эти функции обеспечиваются ядром. Это означает, что большинство функций, которые вы вызываете для управления потоками или фиберами, подразумевают переключение контекста в пространство ядра, а это недешевая операция. Но есть более легкие альтернативы потокам и фиберам. Такие механизмы позволяют программистам писать код с точки зрения нескольких независимых потоков управления, каждый с собственным контекстом выполнения, не требуя дорого платить за выполнение

вызовов ядра. В совокупности эти средства называются *потоками уровня пользователя*.

Потоки уровня пользователя полностью реализованы в пространстве пользователя. Ядро ничего не знает о них. Каждый поток пользовательского уровня представлен обычной структурой данных, которая отслеживает идентификатор потока, возможно понятное человеку имя и информацию контекста выполнения (содержимое регистров ЦП и стек вызовов). Библиотека потоков пользовательского уровня предоставляет функции API для создания и уничтожения потоков и переключения контекста между ними. Каждый поток пользовательского уровня выполняется в контексте реального потока или фибера, предоставленного операционной системой.

Хитрость при реализации библиотеки потоков пользовательского уровня заключается в том, чтобы выяснить, как реализовать переключение контекста. Если вы поразмыслите над этим, то поймете, что переключение контекста в основном сводится к обмену содержимым регистров процессора. В конце концов, регистры содержат всю информацию, необходимую для описания контекста выполнения потока, включая указатель инструкций и стек вызовов. Поэтому, написав хитрый код на ассемблере, можно реализовать переключение контекста. А когда у вас есть переключение контекста, остальная часть пользовательской библиотеки потоков становится не чем иным, как управлением данными.

Потоки пользовательского уровня не очень хорошо поддерживаются в C и C++, но некоторые переносимые и непереносимые решения существуют. POSIX предоставил набор функций для управления контекстами выполнения легких потоков с помощью своего заголовочного файла `ucontext.h` (https://ru.wikipedia.org/wiki/Setcontext), но с тех пор этот API устарел. Библиотека C++ Boost предоставляет переносимую библиотеку потоков пользовательского уровня. (Ищите по слову context на www.boost.org, где находится документация по этой библиотеке.)

Корутины

Корутины (coroutines), или сопрограммы, — это особый тип потока пользовательского уровня, который может оказаться очень полезным для написания изначально асинхронных программ, таких как веб-серверы и игры! Корутина — это обобщение понятия подпрограммы. В то время как подпрограмма может выйти, только вернув управление вызывающей стороне, корутина также может выйти, *уступив место* другой корутине. Когда корутина отходит в сторону, ее контекст выполнения сохраняется в памяти. В следующий раз, когда *вызывается* корутина, которая прежде уступила ресурсы другой корутине, она продолжается с того места, где остановилась.

Подпрограммы вызывают друг друга иерархически. Подпрограмма A вызывает B, которая вызывает C, которая возвращает управление B, а та возвращает его A. Корутины же вызывают друг друга симметрично. Корутина A может

уступить B, которая может уступать, A до бесконечности. Этот образец вызова туда и обратно не создает бесконечно углубляющийся стек вызовов, потому что каждая корутина поддерживает собственный частный контекст выполнения (стек вызовов и содержимое регистров). Таким образом, переход от корутины A к корутине B действует скорее как переключение контекста между потоками, а не как вызов функции. Но, поскольку корутины реализованы с потоками *пользовательского уровня*, эти переключатели контекста очень эффективны.

Вот пример псевдокода системы, в которой одна корутина непрерывно создает данные, используемые другой корутиной:

```
Queue g_queue;

coroutine void Produce()
{
    while (true)
    {
        while (!g_queue.IsFull())
        {
            CreateItemAndAddToQueue(g_queue);
        }
        YieldToCoroutine(Consume);

        // продолжается отсюда при следующем входе...
    }
}

coroutine void Consume()
{
    while (true)
    {
        while (!g_queue.IsEmpty())
        {
            ConsumeItemFromQueue(g_queue);
        }
        YieldToCoroutine(Produce);

        // продолжается отсюда при следующем входе...
    }
}
```

Корутины чаще всего создаются в языках высокого уровня, таких как Ruby, Lua и Go от Google. Можно использовать корутины и в C или C++. Библиотека C++ Boost обеспечивает надежную реализацию корутин, но Boost требует, чтобы вы компилировали и компоновали довольно большую кодовую базу. Если хочется чего-то более простого, можете попробовать создать собственную библиотеку корутин. Запись в блоге Malte Skarupke (probblydance.com/2013/02/20/handmade-coroutines-for-windows/) демонстрирует, что это не такая обременительная задача, как вы могли бы сначала подумать.

Потоки ядра и пользовательские потоки

Термин *«поток ядра»* имеет два значения, и это может стать основным источником путаницы, когда вы читаете о многопоточности. Давайте демистифицируем термин. Его определения приводятся далее.

1. В Linux поток ядра — это особый вид потока, созданного для внутреннего использования самим ядром, который работает только тогда, когда процессор находится в привилегированном режиме. Также ядро создает потоки для применения процессами (через API, такие как pthread, или `std::thread` в C++11). Эти потоки выполняются в пространстве пользователя в контексте процесса. В этом смысле любой поток, который выполняется в привилегированном режиме, является потоком ядра, а любой поток, выполняемый в пользовательском режиме (в контексте однопоточного или многопоточного процесса), — пользовательским потоком.
2. Термином «поток ядра» может обозначаться любой поток, *известный* ядру, чье выполнение *запланировано*. В соответствии с этим определением, поток ядра может выполняться либо в пространстве ядра, либо в пространстве пользователя, а термин «пользовательский поток» применяется только к потоку управления, который управляется программой из пространства пользователя *без* участия ядра, например, в качестве корутины.

Применяя определение 2, фиберы стирают грань между потоком ядра и потоком пользователя. С одной стороны, ядро знает о фиберах и поддерживает для каждого из них отдельный стек вызовов. С другой стороны, фибер не планируется ядром — он может работать только тогда, когда другие фибер или поток явно передают ему управление через вызов такой функции, как `SwitchToFiber()`.

4.4.9. Что еще почитать о процессах и потоках

В предыдущих подразделах мы рассмотрели основы процессов, потоков и фиберов (на самом деле лишь коснулись темы). Для получения дополнительной информации посетите некоторые из следующих сайтов.

- Для ознакомления с потоками — www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html.
- Полные документы по API pthread доступны онлайн — просто поищите «документация pthread».
- Документацию по API потоков Windows можно найти в разделе «Функции процессов и потоков» по адресу msdn.microsoft.com.
- Для получения дополнительной информации о планировании потоков найдите в Сети Полное руководство по планированию процессов Linux Никиты Ишкова.

- Для того чтобы познакомиться с реализацией корутин на Go (там они называются горутинами), посмотрите презентацию Роба Пайка: www.youtube.com/watch?v=f6kdp27TYZs.

4.5. Введение в параллельное программирование

Существует широкий спектр явно параллельного вычислительного оборудования, но как его использовать программистам? Ответ кроется в *параллельных методах программирования*. В параллельном программном обеспечении рабочая нагрузка разбивается на два и более потока управления, которые могут выполняться полунезависимо. Как мы видели в разделе 4.1, для того чтобы система квалифицировалась как параллельная, она должна включать несколько процессов чтения и/или несколько процессов записи *общих* данных.

Роб Пайк, выдающийся инженер Google Inc., специализирующийся на распределенных и параллельных системах и языках программирования, определяет параллельность как «композицию независимо выполняемых вычислений». Это подчеркивает, что множественные потоки управления в параллельной системе обычно работают полунезависимо, но их вычисления строятся на *совместном* использовании данных и *синхронизации* операций различными способами.

Параллелизм может принимать несколько форм. Вот некоторые примеры:

- запущенная под Linux или Windows цепочка команд, такая как `cat render.cpp | grep "light"`;
- один процесс, состоящий из нескольких потоков, которые совместно используют пространство виртуальной памяти и работают с общим набором данных;
- группа потоков, состоящая из тысяч потоков, работающих на графическом процессоре и взаимодействующих для визуализации сцены;
- многопользовательская видеоигра, разделяющая общее игровое состояние между клиентами, действующими на нескольких ПК или игровых приставках.

4.5.1. Зачем писать параллельное программное обеспечение

Параллельные программы иногда пишутся лишь потому, что модель из нескольких полунезависимых потоков управления лучше решает проблему, чем единая схема потока управления. Параллельный дизайн может быть выбран и для наилучшего использования многоядерной вычислительной платформы, даже если рассматриваемую задачу можно более эффективно решить с помощью последовательного проектирования.

4.5.2. Модели параллельного программирования

Для того чтобы различные потоки в параллельной программе могли взаимодействовать, им необходимо *обмениваться данными* и *синхронизировать* свои действия. Другими словами, им нужно общаться. Существует два основных способа взаимодействия параллельных потоков.

- *Обмен сообщениями.* В этом режиме общения параллельные потоки передают сообщения друг другу, чтобы обмениваться данными и синхронизировать свои действия. Сообщения могут передаваться по сети, между процессами с использованием канала или с помощью очереди сообщений в памяти, доступной и отправителю, и получателю. Этот подход применим как для потоков, работающих на одном компьютере (в рамках либо одного процесса, либо нескольких процессов), так и для потоков в процессах, работающих на физически различных компьютерах, например объединенных в кластер или сеть компьютеров, распределенных по всему миру.
- *Общая память.* В этом режиме связи двум или более потокам предоставляется доступ к одному и тому же блоку физической памяти, поэтому они могут работать непосредственно с любыми объектами данных, находящимися в этой области памяти. Прямой доступ к общей памяти работает, только если все потоки выполняются на одном компьютере с областью физической памяти, которую видят все ядра ЦП. Потоки внутри одного процесса всегда совместно задействуют виртуальное адресное пространство, поэтому могут использовать память «бесплатно». Также потоки в разных процессах могут совместно задействовать память, отображая определенные страницы физической памяти во все виртуальные адресные пространства процессов.

Интересно отметить, что *иллюзия* общей памяти между физически отдельными компьютерами может быть реализована поверх системы передачи сообщений — этот метод известен как *распределенная общая память*. Аналогично механизм передачи сообщений может быть реализован поверх архитектуры общей памяти путем реализации очереди сообщений, которая находится в пуле общей памяти.

У каждого подхода есть свои плюсы и минусы. Совместное использование физической памяти — наиболее эффективный способ совместного применения большого объема данных, поскольку их не нужно копировать для передачи между потоками. В то же время, как мы увидим в подразделе 4.5.3 и разделе 4.7, *совместное* использование ресурсов любого типа (памяти или других) сопряжено со множеством проблем синхронизации, о которых, как правило, трудно рассуждать и которые очень сложно учесть так, чтобы гарантировать правильность работы программы. Архитектура передачи сообщений способна уменьшить влияние подобных проблем, но не устраняет их полностью.

В этой книге мы сконцентрируемся в первую очередь на параллельности на основе общей памяти. Делаем так по двум причинам. Во-первых, это тот тип па-

раллелизма, с которым вы, скорее всего, столкнетесь как игровой программист, поскольку игровые движки обычно реализуются как однопроцессные многопоточные программы. (Сетевые многопользовательские игры являются явным исключением из этого правила, поскольку они интенсивно задействуют передачу сообщений.) Во-вторых, параллелизм совместно используемой памяти — более сложная для понимания тема. Как только вы поймете параллелизм в среде совместно используемой памяти, методы передачи сообщений должны стать относительно простыми для освоения.

4.5.3. Состояние гонки

Состояние гонки определяется как любая ситуация, в которой поведение программы зависит от времени. Другими словами, при наличии состояния гонки поведение программы может меняться, когда изменяется относительная последовательность событий, происходящих в системе, из-за изменчивости продолжительности времени, затрачиваемого различными потоками управления на выполнение своих задач.

Критическая гонка

Иногда состояние гонки безобидно: поведение программы может несколько изменяться в зависимости от времени, но негативных последствий не будет. Но в состоянии *критической гонки* программа может начать некорректно вести себя.

Ошибки, вызванные критическими гонками, часто кажутся странными или даже невозможными программистам, не имеющим опыта работы с ними. Вот некоторые примеры:

- случайные или кажущиеся случайными ошибки или сбои;
- неверные результаты;
- структуры данных, которые попадают в неверное состояние;
- ошибки, которые как по волшебству исчезают при переключении на отладочную сборку;
- ошибки, которые появляются на некоторое время, затем исчезают на несколько дней и снова возникают (обычно в ночь перед релизом!);
- ошибки, которые исчезают при добавлении в программу журналирования (или метода отладки `printf()`) в попытке обнаружить источник проблемы.

Программисты часто называют такие ошибки Heisenbugs — ошибками синхронизации в многопоточном приложении.

Гонки данных

Гонка данных — это критическое состояние гонки, при котором два и более потока управления взаимодействуют друг с другом при чтении и/или записи блока

общих данных, что приводит к повреждению данных. Гонки данных — это основная проблема параллельного программирования. Написание параллельных программ всегда сводится к устранению гонок данных либо тщательным контролем доступа к совместно используемым данным, либо заменой общих данных их частными независимыми копиями, что превращает проблему параллелизма в проблему последовательного программирования.

Для лучшего понимания причин возникновения гонки данных рассмотрим следующий простой фрагмент кода C/C++:

```
int g_count = 0;

inline void IncrementCount()
{
    ++g_count;
}
```

Если вы скомпилируете этот код для процессора Intel x86 и затем пропустите через дизассемблер, результат будет выглядеть примерно так:

```
mov   eax,[g_count]   ; считать g_count в регистр EAX
inc   eax             ; увеличить значение
mov   [g_count],eax   ; записать EAX обратно в g_count
```

Это пример операции *чтения — изменения — записи* (read — modify — write, RMW).

Теперь представьте, что два потока, A и B, должны будут одновременно вызвать функцию `IncrementCount()` (либо параллельно, либо через вытесняющую многопоточность). Если нормальной работе каждый поток вызывал функцию ровно один раз, мы ожидаем, что конечное значение `g_count` будет равно 2, потому что либо поток A увеличивает значение `g_count`, а затем поток B увеличивает его, либо наоборот (табл. 4.1).

Таблица 4.1. Пример правильной работы простой двухпоточной параллельной программы

Поток A		**Поток B**		**Значение g_count**
Действие	**EAX**	**Действие**	**EAX**	
	?		?	0
Чтение	0		?	0
Увеличение	1		?	0
Запись	1		?	1
	1	Чтение	1	1
	1	Увеличение	2	1
	1	Запись	2	2

Здесь поток A считывает содержимое общей переменной, увеличивает значение и записывает результаты обратно в общую переменную. Позже поток B выполняет те же шаги. Окончательное значение общей переменной равно 2, как и ожидалось.

Далее рассмотрим случай, когда два потока работают на одноядерном компьютере с вытесняющей многозадачностью. Допустим, поток A запускается первым. Как только он заканчивает выполнение первой инструкции `mov`, происходит переключение контекста на поток B. Вместо потока A, выполняющего свою инструкцию `inc`, поток B запускает свою первую инструкцию `mov`. Через некоторое время квант времени потока B истекает и контекст ядра переключается обратно на поток A, который продолжается там, где остановился, и выполняет инструкцию `inc`. Происходящее иллюстрирует табл. 4.2. (Подсказка: это нехорошо! Конечное значение `g_count` больше не равно 2, как должно быть.)

Таблица 4.2. Пример состояния гонки

Поток A		**Поток B**		**Значение g_count**
Действие	**EAX**	**Действие**	**EAX**	
	?		?	0
Чтение	0		?	0
	0	Чтение	0	0
	0	Увеличение	1	0
	0	Запись	1	1
Увеличение	1		1	1
Запись	1		1	1

Поток A читает значение общей переменной, но затем вытесняется потоком B, который также читает то же самое значение. К тому времени, когда оба потока увеличили значение и записали свои результаты обратно в общую переменную, глобальная переменная содержит неверное значение 1 вместо ожидаемого 2.

Если мы запустим два потока на параллельном оборудовании, может возникнуть похожая ошибка, хотя и по иной причине. Как и в случае с одноядерным процессором, нам может повезти: две операции чтения — изменения — записи не перекроются и результат окажется правильным. Однако если они перекрываются, либо смещены относительно друг друга, либо находятся в идеальной синхронизации, оба потока могут в конечном итоге загрузить одно и то же значение `g_count` в свои регистры EAX. Оба будут увеличивать значение и записывать его в память. Один поток перезапишет результаты другого, но это на самом деле не имеет значения: поскольку они оба загрузили одно и то же начальное значение, окончательное значение `g_count` окажется неверным, как было в одноядерном сценарии. Три сценария гонки данных (вытеснение, перекрытие смещений и идеальная синхронизация) показаны на рис. 4.27.

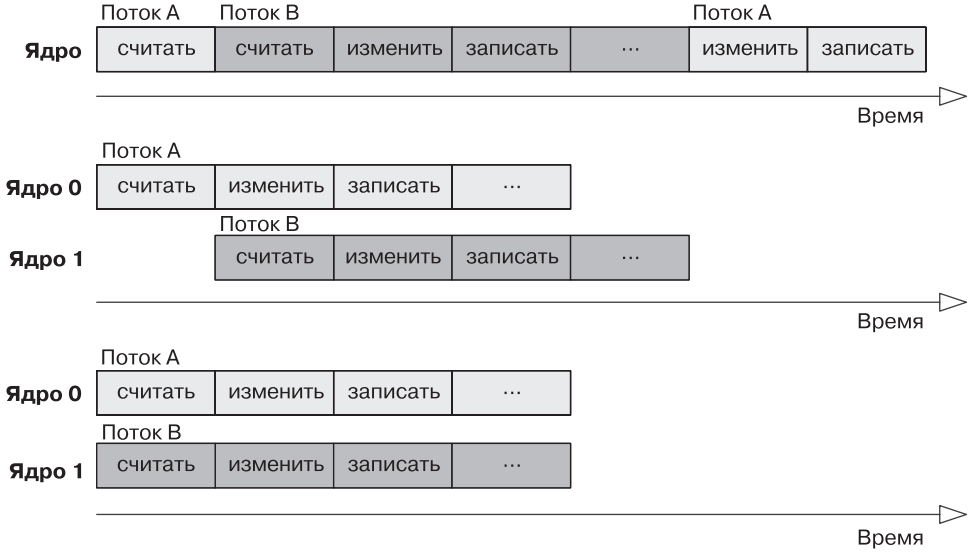


Рис. 4.27. Три вида гонки данных в операции чтения — изменения — записи: *вверху* — два потока состязаются на одном ядре процессора; *в центре* — два потока накладываются на двух отдельных ядрах и смещаются на одну инструкцию; *внизу* — два потока перекрываются в идеальной синхронизации на двух ядрах

4.5.4. Критические операции и атомарность

Всякий раз, когда одна операция прерывается другой, возникает вероятность ошибки гонки данных. Однако не все такие прерывания действительно вызывают ошибки. Например, если поток выполняет операцию с фрагментом данных, который может быть виден только этому одному потоку, гонка данных невозможна. Такая операция без последствий может быть прервана в любой момент любой другой операцией.

Аналогично, если операция над одним объектом данных прерывается операцией над другим объектом, эти две операции не могут взаимодействовать друг с другом[1] и, следовательно, ошибка гонки данных невозможна. Такие ошибки возникают только тогда, когда операция над объектом прерывается другой операцией *над тем же* объектом. Таким образом, мы можем говорить только о гонках данных по отношению к определенному общему объекту данных.

Давайте используем термин *«критическая операция»* для обозначения любой операции, которая может считывать или изменять один конкретный общий объект. Чтобы гарантировать, что общий объект не подвержен ошибкам гонки данных, мы должны гарантировать, что ни одна из его критических операций не сможет пре-

[1] Это верно только в том случае, если два объекта находятся в разных строках кэша.

рвать другую. Когда критическая операция таким образом становится бесперебойной, она называется *атомарной операцией*. Или можно сказать, что такая операция обладает свойством *атомарности*.

Вызов и ответ

Когда мы только учимся программировать, нам обычно говорят, что *время*, необходимое для выполнения каждого шага в алгоритме, не влияет на правильность алгоритма — значение имеет лишь то, чтобы шаги выполнялись в правильном *порядке*. Эта простая модель хороша для последовательных (однопоточных) программ. Но при наличии нескольких потоков невозможно определить порядок операций, когда каждая из них имеет конечную продолжительность (рис. 4.28).

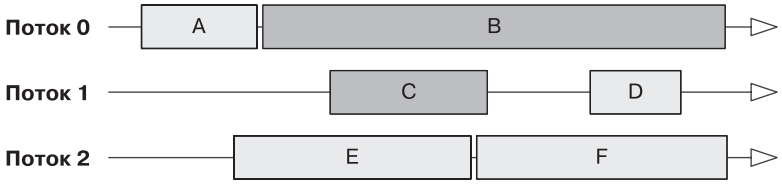


Рис. 4.28. Поскольку каждый шаг в алгоритме занимает определенное время, трудно ответить на вопросы об относительном порядке шагов в многопоточной программе, например, происходит операция B до или после операции C

В параллельной системе единственный способ определить *порядок* — ограничиться *мгновенными событиями*. Для любой пары мгновенных событий существует только три возможности: событие A происходит до события B, событие A происходит после события B или два события происходят одновременно. (Абсолютно одновременные события редки, но они возможны в многоядерном компьютере, в котором некоторые или все ядра совместно используют синхронизированные часы.)

Любая операция конечной продолжительности может быть разбита на два мгновенных события — ее *вызов* (момент, когда операция начинается) и ее *ответ* (момент, когда она считается завершенной). Любой фрагмент кода, который включает в себя критическую операцию с каким-либо общим объектом данных, можно разделить на три секции, причем мгновенные события вызова и ответа обозначают границы между ними. Обратите внимание на то, что здесь говорится о порядке, в котором события происходят так, как они записаны в исходном коде, — это называется *порядком программы*.

- *Раздел преамбулы* — весь код, который появляется перед вызовом критической операции, в порядке программы.
- *Критически важный раздел* — код, содержащий саму критическую операцию.
- *Раздел постамбулы* — весь код, который появляется после ответа критической операции, в порядке программы.

Разделение блока кода на три секции иллюстрируется на рис. 4.29 и 4.30.

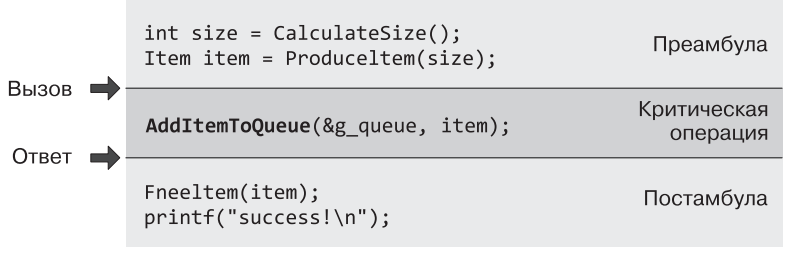


Рис. 4.29. Любой фрагмент кода, который включает в себя критическую операцию, можно разбить на три раздела. Критическая операция ограничена сверху ее вызовом, а снизу — ответом

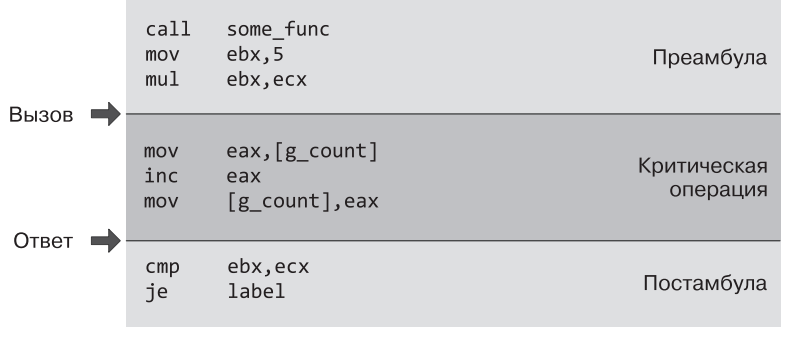


Рис. 4.30. Пример на ассемблере фрагмента кода, разделенного на три раздела, ограниченные вызовом и ответом критической операции

Определение атомарности

Как мы видели ранее, ошибка гонки данных может возникнуть, когда критическая операция прерывается другой критической операцией над общим объектом. Это может случиться:

- когда один поток вытесняет другой на одном ядре;
- когда две и более критические операции перекрываются между несколькими ядрами.

Рассматривая эту ситуацию с точки зрения вызова и ответа, мы можем более точно сформулировать общее понятие прерывания. *Прерывание* происходит всякий раз, когда *между* вызовом и ответом одной операции происходит вызов и/или ответ другой операции. Но как уже говорилось, не все прерывания приводят к ошибкам гонки данных. Гонка данных может повлиять на критическую операцию с определенным общим объектом только в том случае, если ее вызов и ответ прерваны другой критической операцией над тем же объектом. Следо-

вательно, можно определить *атомарность* критической операции следующим образом.

> Критическая операция выполняется *атомарно*, если между ее вызовом и ответом не вызывается другая критическая операция над тем же объектом.

Здесь следует подчеркнуть, что совершенно нормально, если критическая операция прерывается другими некритическими операциями или критическими операциями, воздействующими на другие, несвязанные объекты данных. Только когда две критические операции над одним и тем же объектом прерывают друг друга, возникает ошибка гонки данных. Рисунок 4.31 иллюстрирует различные случаи — один, в котором критическая операция успешно выполняется атомарно, и три, в которых она не выполняется.

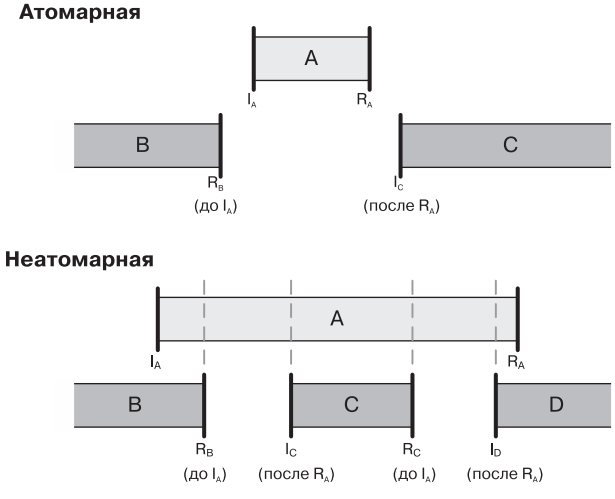


Рис. 4.31. *Вверху*: можно сказать, что критическая операция A выполнялась атомарно, потому что она не была прервана никакими другими вызовами или ответами от критических операций над тем же объектом. *Внизу*: три сценария, в которых критическая операция A выполняется неатомарно, потому что она была прервана вызовом и/или ответом другой критической операции над тем же объектом

Можно гарантировать, что критическая операция будет выполнена атомарно, если с точки зрения других потоков в системе мы сможем заставить ее произойти *мгновенно*. Другими словами, должно казаться, что вызов и ответ операции являются одновременными или что критическая операция имеет нулевую продолжительность. Таким образом, не допускается возможность вызова другой критической операции или ответа, «пробирающегося» между вызовом и ответом рассматриваемой операции.

Делаем операцию атомарной

Как можно преобразовать критическую операцию в атомарную? Самый простой и надежный способ сделать это — использовать специальный объект, называемый

мьютексом. Мьютекс — объект, предоставляемый операционной системой, который действует как замок, так как он может быть заблокирован и разблокирован потоком. При наличии двух критических операций над определенным общим объектом данных мы защищаем вызов каждой операции с помощью получения мьютекса и освобождаем его при ответе каждой из них. Поскольку ОС гарантирует, что мьютекс может быть получен только одним потоком за раз, мы можем быть уверены, что вызов или ответ одной операции никогда не произойдет между вызовом и ответом другой. С точки зрения глобального упорядочения событий в параллельной системе критическая операция, защищенная блокировкой мьютекса, представляется мгновенной.

Мьютексы являются частью набора инструментов параллелизма, предоставляемых операционной системой и известных как *примитивы синхронизации потоков*. Мы рассмотрим их в разделе 4.6.

Атомарность как сериализация

Рассмотрим группу потоков, которые пытаются выполнить одну операцию над общим объектом данных. Без атомарности эти операции могут происходить одновременно или начать непредсказуемо пересекаться между собой со временем (рис. 4.32).

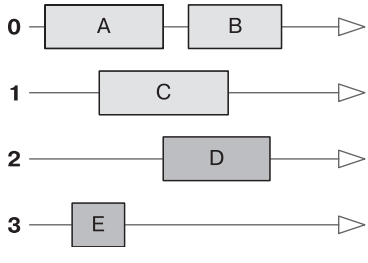


Рис. 4.32. Без атомарности операции, выполняемые несколькими потоками, могут со временем перекрываться непредсказуемым образом

Однако выполнение операции атомарно гарантирует, что только один поток будет выполнять ее в любой момент времени. Это создает эффект *сериализации* операций: то, что раньше было беспорядочным перекрытием операций, превращается в упорядоченную последовательность атомарных операций. Преобразование операций в атомарные не позволяет контролировать, каким будет их порядок, мы можем сказать наверняка лишь то, что операции будут выполняться в *некоторой* последовательности (рис. 4.33).

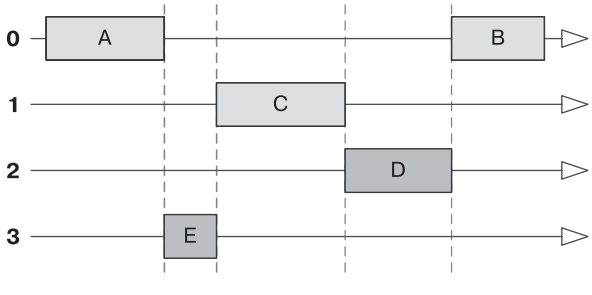


Рис. 4.33. Оборачивая каждую критическую операцию в пару блокировки/разблокировки мьютекса, мы заставляем операции выполняться последовательно

Модели согласованности, ориентированные на данные

Концепции атомарности и сериализации операций в параллельной системе являются частью более широкой темы, известной как *модели согласованности, ориентированные на данные*. Модель согласованности — *соглашение* между хранилищем данных, таким как общий объект данных в параллельной системе или база данных в распределенной системе, и набором потоков, которые совместно используют такое хранилище. Это облегчает рассуждения о поведении хранилища данных: поскольку потоки придерживаются правил контракта, программист может быть уверен, что хранилище данных станет вести себя согласованно и предсказуемо и находящиеся в нем данные не будут повреждены.

Хранилище данных, обеспечивающее гарантию атомарности, можно назвать *линейным*. Тема согласованности, ориентированной на данные, находится за пределами нашей книги, но вы можете прочитать об этом в Интернете. Вот несколько хороших мест для начала:

- поищите термины «модель согласованности» и «линеаризуемость» в «Википедии»;
- www.cse.buffalo.edu/~stevko/courses/cse486/spring13/lectures/26-consistency2.pdf;
- www.cs.cmu.edu/~srini/15-446/S09/lectures/10-consistency.pdf.

4.6. Примитивы синхронизации потоков

Каждая операционная система, которая поддерживает параллелизм, имеет набор инструментов, известных как *примитивы синхронизации потоков*. Эти инструменты предоставляют две услуги для программистов, пишущих параллельные программы.

1. Способность распределять ресурсы между потоками, делая критические операции *атомарными*.
2. Возможность *синхронизировать* работу двух или более потоков:
   - позволяя потоку *переходить в спящий режим*, пока он ожидает, когда ресурс станет доступным или один либо несколько других потоков выполнят свою задачу;
   - позволяя запущенному потоку *отправлять уведомления* одному или нескольким спящим потокам, тем самым пробуждая их.

Здесь следует отметить: хотя эти примитивы синхронизации потоков надежны и относительно просты в использовании, они, как правило, довольно дороги. Это потому, что данные инструменты предоставляются ядром. Поэтому для взаимодействия с любым из них требуется вызов ядра, который подразумевает переключение контекста в защищенный режим. Такие переключатели контекста могут стоить более 1000 тактов. Из-за высокой стоимости некоторые программисты, работающие

с параллельными программами, предпочитают реализовывать собственные инструменты атомарности и синхронизации или используют программирование без блокировок, чтобы повысить эффективность параллельного программного обеспечения. Тем не менее хорошо разбираться в примитивах синхронизации важно для любого программиста.

4.6.1. Мьютексы

Мьютекс — это объект операционной системы, который позволяет выполнять критические операции атомарно. Мьютекс может находиться в одном из двух состояний: *«разблокирован»* или *«заблокирован»*. (Эти состояния иногда называют *свободным* и *занятым* или *«сигнальным»* (signaled) и *«несигнальным»* (nonsignaled) соответственно.)

Наиболее важным свойством мьютекса является то, что он гарантирует: только один поток когда-либо будет удерживать блокировку в любой момент времени. Итак, если мы обернем все критические операции для определенного общего объекта данных в блокировку мьютекса, они станут атомарными относительно друг друга. Другими словами, операции становятся *взаимоисключающими*. Отсюда и название «мьютекс» — сокращение от «взаимное исключение» (mutual exclusion).

Мьютекс представлен либо обычным объектом C++, либо дескриптором «непрозрачного» объекта ядра. Его API обычно состоит из следующих функций:

- `create()` или `init()` — вызов функции или конструктор класса, который создает мьютекс;
- `destroy()` — вызов функции или деструктор, который уничтожает мьютекс;
- `lock()` или `acquire()` — блокирующая функция, которая блокирует мьютекс от лица вызывающего потока, но переводит его в спящий режим, если в данный момент блокировка удерживается другим потоком;
- `try_lock()` или `try_acquire()` — неблокирующая функция, которая пытается заблокировать мьютекс, но немедленно возвращается, если сделать это невозможно;
- `unlock()` или `release()` — неблокирующая функция, снимающая блокировку с мьютекса. В большинстве операционных систем только поток, заблокировавший мьютекс, может разблокировать его.

Когда мьютекс заблокирован потоком в системе, мы говорим, что он находится в *сигнальном* состоянии. Когда поток снимает блокировку, состояние мьютекса становится *несигнальным*. Если один или несколько других потоков спят (заблокированы), ожидая мьютекса, то такая сигнализация заставляет ядро выбрать один из ожидающих потоков и разбудить его. В некоторых операционных системах поток может явно ожидать, пока объект ядра, такой как мьютекс, не освободится. В Windows для этой цели служат вызовы ОС `WaitForSingleObject()` и `WaitForMultipleObjects()`.

POSIX

Теперь, когда мы понимаем, как работают мьютексы, рассмотрим несколько примеров. Библиотека потоков POSIX предоставляет объекты мьютекса ядра через функциональный интерфейс в стиле C. Вот как мы могли бы использовать его, чтобы превратить пример общего счетчика из раздела «Гонки данных» в атомарную операцию:

```
#include <pthread.h>

int g_count = 0;
pthread_mutex_t g_mutex;
inline void IncrementCount()
{
    pthread_mutex_lock(&g_mutex);
    ++g_count;
    pthread_mutex_unlock(&g_mutex);
}
```

Обратите внимание на то, что для ясности и краткости мы пропустили код, обычно выполняемый основным потоком, вызывающим `pthread_mutex_init()` для инициализации мьютекса перед порождением потоков, которые будут его использовать, и `pthread_mutex_destroy( )` — для уничтожения мьютекса, как только все остальные потоки завершатся.

Стандартная библиотека языка C++

Начиная с C++11 стандартная библиотека C++ предоставляет доступ к мьютексам ядра через класс `std::mutex`. Вот как можно использовать его для атомарного увеличения счетчика:

```
#include <mutex>

int g_count = 0;
std::mutex g_mutex;

inline void IncrementCount()
{
    g_mutex.lock();
    ++g_count;
    g_mutex.unlock();
}
```

Конструктор и деструктор класса `std::mutex` обрабатывают инициализацию и уничтожение базового объекта мьютекса ядра, что делает его немного проще в использовании, чем `pthread_mutex_t`.

Windows

В Windows мьютекс представлен скрытым объектом ядра и на него ссылаются через дескриптор. Мьютекс заблокирован и ожидает освобождения. Это реализуется

универсальной функцией `WaitForSingleObject()`. Разблокировка мьютекса осуществляется вызовом `ReleaseMutex()`. Переписав рассматриваемый пример с использованием мьютексов Windows и снова опуская детали создания и уничтожения объекта мьютекса, мы получаем следующий код:

```
#include <windows.h>

int g_count = 0;
HANDLE g_hMutex;

inline void IncrementCount()
{
    if (WaitForSingleObject(g_hMutex, INFINITE)
        == WAIT_OBJECT_0)
    {
        ++g_count;
        ReleaseMutex(g_hMutex);
    }
    else
    {
        // Учимся обрабатывать неудачи...
    }
}
```

4.6.2. Критически важные секции

В большинстве операционных систем мьютекс может использоваться несколькими процессами. Как таковая это структура данных, которая управляется внутри ядра. Это означает, что все операции, выполняемые с мьютексом, включают вызов ядра и, следовательно, переключение контекста в защищенный режим на ЦП. Это делает мьютексы довольно дорогими, даже когда никакие другие потоки не борются за блокировку.

Некоторые операционные системы предоставляют менее дорогие альтернативы мьютексу. Например, у Microsoft Windows есть механизм блокировки, известный как *критическая секция*. Его терминология и API выглядят немного иначе, чем относящиеся к мьютексу, но критическая секция в Windows — это просто недорогой мьютекс.

API критической секции выглядит следующим образом.

1. `InitializeCriticalSection()`. Создает объект критической секции.
2. `DeleteCriticalSection()`. Уничтожает инициализированный объект критической секции.
3. `EnterCriticalSection()`. Блокирующая функция, которая блокирует критическую секцию от лица вызывающего потока, но переводит ее в спящий режим (см. подраздел 4.4.6), если в данный момент блокировка удерживается другим потоком.

4. `TryEnterCriticalSection()`. Неблокирующая функция, которая пытается заблокировать критическую секцию, но немедленно возвращается, если блокировка невозможна.
5. `LeaveCriticalSection()`. Неблокирующая функция, снимающая блокировку с объекта критической секции.

Вот как мы атомарно реализовали счетчик, используя API критической секции Windows:

```
#include <windows.h>

int g_count = 0;
CRITICAL_SECTION g_critsec;

inline void IncrementCount()
{
    EnterCriticalSection(&g_critsec);
    ++g_count;
    LeaveCriticalSection(&g_critsec);
}
```

Как и прежде, мы пропустили некоторые детали. Основной поток обычно инициализирует критическую секцию перед порождением потоков, которые ее используют, и, конечно, очищает ее после завершения всех потоков.

Как достигается низкая стоимость критической секции? Когда поток сначала пытается войти (заблокировать) в критическую секцию, которая уже заблокирована другим потоком, для ожидания применяется недорогая *спин-блокировка*, пока другой поток не покинет (разблокирует) эту критическую секцию. Спин-блокировка не требует переключения контекста в ядре, что делает ее на несколько тысяч тактов дешевле, чем мьютекс. Только если поток занят и ждет слишком долго, поток переводится в спящий режим, как было бы с обычным мьютексом. Этот менее дорогой подход работает, потому что, в отличие от мьютекса, критическая секция не может быть передана за границы процесса. Мы обсудим спин-блокировки более подробно в подразделе 4.9.7.

Некоторые другие операционные системы также предоставляют дешевые варианты мьютекса. Например, Linux поддерживает фьютексы (futex — fast userspace mutex), которые действуют как критическая секция в Windows. Их использование выходит за рамки книги, но вы можете прочитать о фьютексах на www.akkadia.org/drepper/futex.pdf.

4.6.3. Переменные условия

В параллельном программировании часто нужно посылать сигналы между потоками, чтобы синхронизировать их действия. Одним из примеров этого является повсеместная проблема производителя — потребителя, которую мы представили в подразделе 4.4.8. Есть два потока: поток-*производитель* вычисляет или иным образом генерирует некоторые данные, которые считываются и используются

потоком-*потребителем*. Очевидно, что потребитель не может применять данные, пока производитель не создал их. Следовательно, поток-производитель нуждается в способе уведомить потребителя о том, что данные готовы.

Мы могли бы рассмотреть возможность использования глобальной булевой переменной в качестве механизма сигнализации. Следующий фрагмент кода иллюстрирует идею, задействуя потоки POSIX. (Некоторые детали были опущены для ясности.)

```
Queue           g_queue;
pthread_mutex_t g_mutex;
bool            g_ready = false;

void* ProducerThread(void*)
{
    // постоянно производим данные...
    while (true)
    {
        pthread_mutex_lock(&g_mutex);

        // заполняем очередь данными
        ProduceDataInto(&g_queue);

        g_ready = true;
        pthread_mutex_unlock(&g_mutex);

        // освободить остаток моего времени,
        // чтобы потребитель мог запуститься
        pthread_yield();
    }
    return nullptr;
}

void* ConsumerThread(void*)
{
    // постоянно производим данные...
    while (true)
    {
        // ждем, пока данные будут готовы
        while (true)
        {
            // читаем значение в локальную переменную,
            // проверяем блокировку мьютекса
            pthread_mutex_lock(&g_mutex);
            const bool ready = g_ready;
            pthread_mutex_unlock(&g_mutex);

            if (ready)
                break;
            }
```

```
            // потребляем данные
            pthread_mutex_lock(&g_mutex);
            ConsumeDataFrom(&g_queue);
            g_ready = false;
            pthread_mutex_unlock(&g_mutex);

            // освободить остаток моего времени,
            // чтобы производитель мог запуститься
            pthread_yield();
    }
    return nullptr;
}
```

Помимо того, что этот пример несколько надуманный, есть одна большая проблема: поток-потребитель крутится в узком цикле, опрашивая значение `g_ready`. Как мы обсуждали в подразделе 4.4.6, это впустую тратит ценные циклы центрального процессора.

В идеале нужен способ заблокировать поток-потребитель (перевести его в спящий режим), пока производитель выполняет свою работу, а затем разбудить его, когда данные будут готовы для использования. Это может быть достигнуто применением нового вида объекта ядра, называемого *условной переменной* (condition variable, CV).

Условная переменная — на самом деле не переменная, которая хранит условие. Скорее, это очередь ожидающих (спящих) потоков в сочетании с механизмом, который позволяет работающему потоку пробуждать спящие потоки по своему выбору. (Возможно, лучше было бы назвать ее очередью ожидания.) Операции сна и пробуждения выполняются атомарно с помощью мьютекса, предоставляемого программой, и небольшого участия ядра.

API для условной переменной выглядит примерно так.

1. `create()` или `init()`. Вызов функции или конструктор класса, который создает условную переменную.
2. `destroy()`. Вызов функции или деструктора, уничтожающих условную переменную.
3. `wait()`. Функция блокировки, которая переводит вызывающий поток в спящий режим.
4. `notify()`. Неблокирующая функция, пробуждающая все спящие в данный момент потоки в ожидании условной переменной.

Перепишем пример «производитель — потребитель», используя условную переменную:

```
Queue           g_queue;
pthread_mutex_t g_mutex;
bool            g_ready = false;
pthread_cond_t  g_cv;
```

```
void* ProducerThreadCV(void*)
{
    // постоянно производим данные...
    while (true)
    {
        pthread_mutex_lock(&g_mutex);

        // заполняем очередь данными
        ProduceDataInto(&g_queue);

        // уведомить и разбудить поток-потребитель
        g_ready = true;
        pthread_cond_signal(&g_cv);
        pthread_mutex_unlock(&g_mutex);
    }
    return nullptr;
}

void* ConsumerThreadCV(void*)
{
    // постоянно потребляем данные...
    while (true)
    {
        // ожидаем готовности данных
        pthread_mutex_lock(&g_mutex);
        while (!g_ready)
        {
            // засыпаем, пока не получим уведомление...
            // мьютекс будет освобожден для нас ядром
            pthread_cond_wait(&g_cv, &g_mutex);

            // когда он просыпается, ядро проверяет,
            // что этот поток снова удерживает мьютекс
        }

        // потребляем данные
        ConsumeDataFrom(&g_queue);
        g_ready = false;
        pthread_mutex_unlock(&g_mutex);
    }
    return nullptr;
}
```

Поток-потребитель вызывает `pthread_cond_wait()`, чтобы перейти в спящий режим до того момента, когда значение `g_ready` станет истинным. Производитель некоторое время работает, производя свои данные. Когда они готовы, производитель устанавливает для глобального флага `g_ready` значение `true`, а затем пробуждает спящего потребителя, вызывая `pthread_cond_signal()`. Затем последний потребляет данные. В этом примере между потребителем и производителем происходит своеобразный пинг-понг с бесконечными переходами от одного потока к другому.

Вы, вероятно, заметили, что поток потребителя блокирует свой мьютекс перед тем, как войти в цикл `while`, который проверяет флаг `g_ready`. Ожидая переменную условия, он, по-видимому, засыпает, *удерживая блокировку мьютекса*! Обычно это было бы фатальным: если поток переходит в спящий режим, удерживая блокировку, это почти наверняка приведет к *тупиковой ситуации* (см. подраздел 4.7.1). Однако это не проблема при использовании условной переменной. Так происходит потому, что ядро немного ослабляет хватку, *разблокируя мьютекс* после того, как поток был благополучно отправлен в спящий режим. Позже, когда спящий поток возвращается в исходное состояние, ядро выполняет несколько более легких действий, чтобы гарантировать, что блокировка снова будет удерживаться только что пробужденным потоком.

Возможно, вы заметили еще одну странность: поток-потребитель по-прежнему использует цикл `while` для проверки значения `g_ready`, хотя применяет и условную переменную, чтобы дождаться, пока флаг не примет значения `true`. Причина, по которой этот цикл необходим, заключается в том, что ядро иногда может внезапно пробуждать потоки. В результате, когда возвращается вызов `pthread_cond_wait()`, значение `g_ready` может еще не быть истинным. Таким образом, мы должны продолжать опрос до тех пор, пока условие действительно не станет истинным.

4.6.4. Семафоры

Подобно тому как мьютекс действует как атомарный логический флаг, *семафор* действует как атомарный счетчик, значение которого никогда не должно опускаться ниже нуля. Мы можем рассматривать семафор как особый вид мьютекса, который позволяет *нескольким потокам* получать его одновременно.

Семафор можно использовать, чтобы позволить группе потоков совместно задействовать ограниченный набор ресурсов. Предположим, что мы реализуем систему рендеринга, которая позволяет отображать текст и 2D-изображения в закадровых буферах для рисования индикаторов на экране (heads-up display, HUD) и игровых меню. Далее предположим, что из-за ограничений памяти можем выделить только четыре буфера. Семафор можно применить, чтобы гарантировать, что в каждый данный момент в эти выделенные буферы разрешено рендерить не более четырех потоков.

API семафора обычно состоит из следующих функций.

1. `init()` — инициализирует объект семафора и устанавливает его счетчик в указанное начальное значение.
2. `destroy()` — уничтожает объект семафора.
3. `take()` или `wait()` — если значение счетчика, инкапсулированное данным семафором, больше нуля, эта функция уменьшает его на единицу и немедленно возвращает значение. Если значение счетчика уже равно нулю, эта функция блокирует вызвавший ее поток (переводит его в спящий режим) до тех пор, пока счетчик семафора не превысит ноль.

4. `give()`, `post()` или `signal()` — увеличивает значение инкапсулированного счетчика на единицу, тем самым открывая слот для другого потока, давая ему возможность выполнить функцию `take()`. Если поток в данный момент спит, ожидая семафора, то после вызова функции `give()` этот поток проснется от вызова функции `take()` или `wait()`[1].

Таким образом, для реализации пула ресурсов, к которому могут одновременно обращаться до *N* потоков, мы просто создадим семафор и инициализируем его счетчик значением *N*. Поток получает доступ к пулу ресурсов, вызывая `take()`, и освобождает его, когда заканчивает выполнение, с помощью метода `give()`.

Мы говорим, что семафор *отправляет сигнал* всякий раз, когда его счетчик больше нуля, и *не сигнализирует*, когда счетчик равен нулю. Вот почему функции, которые принимают и освобождают семафор, называются `wait()` и `signal()` соответственно в некоторых API. Если семафор не сигнализирует, когда поток вызывает какую-то функцию, то этот поток будет *ждать* сигнального состояния семафора.

Мьютекс в сравнении с двоичным семафором

Семафор, начальное значение которого равно 1, называется *двоичным семафором*. Можно подумать, что двоичный семафор идентичен мьютексу. Конечно, оба объекта синхронизации разрешают только один поток за раз. Однако они не эквивалентны и обычно используются для совершенно разных целей.

Основное различие между мьютексом и двоичным семафором заключается в том, что мьютекс может быть разблокирован только тем потоком, который его заблокировал. В то же время счетчик семафора может быть увеличен одним потоком и позже уменьшен другим. Это подразумевает, что двоичный семафор может быть разблокирован не тем потоком, который его заблокировал. Иначе говоря, двоичный семафор может быть переведен в *сигнальное состояние* потоком, отличным от того, который его перевел в *ожидание*. Это, казалось бы, тонкое различие между мьютексами и двоичными семафорами приводит к очень разным вариантам использования двух типов объектов синхронизации. Мьютекс применяется для того, чтобы сделать операцию *атомарной*, а двоичный семафор — обычно для *отправки сигнала* из одного потока в другой.

Вернемся к примеру «производитель — потребитель», в котором первый должен уведомить второго о том, что производимые им данные готовы к потреблению. Механизм уведомления может быть реализован с помощью двух бинарных семафоров, один из которых позволяет производителю разбудить потребителя, а другой — потребителю разбудить производителя. Мы можем рассматривать эти семафоры как представление количества используемых и свободных элементов в буфере, который совместно задействуется двумя потоками, хотя в этом простом примере буфер может содержать только один элемент. Поэтому мы назовем сема-

[1] Читая о семафорах, вы можете обнаружить, что некоторые авторы используют имена функций p() и v() вместо wait() и signal(). Буквы взяты из голландских имен этих двух операций.

форы `g_semUsed` и `g_semFree` соответственно. Вот как будет выглядеть код с применением семафоров POSIX:

```
Queue g_queue;
sem_t g_semUsed; // инициализируется в 0
sem_t g_semFree; // инициализируется в 1

void* ProducerThreadSem(void*)
{
    // продолжаем производить...
    while (true)
    {
        // произвести значение (можно сделать неатомарно,
        // потому что это локальные данные)
        Item item = ProduceItem();

        // уменьшаем свободный счетчик
        // (подождите, пока есть место)
        sem_wait(&g_semFree);

        AddItemToQueue(&g_queue, item);

        // увеличиваем используемый счетчик
        // (уведомить потребителя, что есть данные)
        sem_post(&g_semUsed);
    }
    return nullptr;
}

void* ConsumerThreadSem(void*)
{
    // продолжаем потреблять...
    while (true)
    {
        // уменьшаем использованный счетчик
        // ждем, пока данные будут готовы
        sem_wait(&g_semUsed);

        Item item = RemoveItemFromQueue(&g_queue);

        // увеличиваем свободный счетчик
        // (уведомить производителя о наличии места)
        sem_post(&g_semFree);

        // потребляем элемент (можно сделать неатомарно,
        // потому что это локальные данные)
        ConsumeItem(item);
    }
    return nullptr;
}
```

Реализация семафора

Оказывается, можно реализовать семафор в терминах мьютекса, условной переменной и целого числа. В этом смысле семафор — конструкция более высокого уровня, чем мьютекс или условная переменная. Вот как выглядит реализация:

```
class Semaphore
{
private:
    int             m_count;
    pthread_mutex_t m_mutex;
    pthread_cond_t  m_cv;

public:
    explicit Semaphore(int initialCount)
    {
        m_count = initialCount;
        pthread_mutex_init(&m_mutex, nullptr);
        pthread_cond_init(&m_cv, nullptr);
    }

    void Take()
    {
        pthread_mutex_lock(&m_mutex);
        // переводим поток в спящий режим,
        // пока счетчик равен нулю
        while (m_count == 0)
            pthread_cond_wait(&m_cv, &m_mutex);
        --m_count;
        pthread_mutex_unlock(&m_mutex);
    }

    void Give()
    {
        pthread_mutex_lock(&m_mutex);
        ++m_count;
        // если счетчик был равен нулю до
        // увеличения, разбудить ожидающий поток,
        if (m_count == 1)
            pthread_cond_signal(&m_cv);
        pthread_mutex_unlock(&m_mutex);
    }

    // псевдонимы для других часто используемых имен функций
    void Wait()   { Take(); }
    void Post()   { Give(); }
    void Signal() { Give(); }
    void Down()   { Take(); }
    void Up()     { Give(); }
    void P()      { Take(); } // По-голландски proberen = test
    void V()      { Give(); } // По-голландски verhogen = increment
};
```

4.6.5. События Windows

Windows предоставляет механизм, называемый *объектом события*, который по выполняемой функции похож на переменную условия, но намного проще в применении. Как только объект события создан, поток может перейти в спящий режим, вызвав `WaitForSingleObject()`, затем этот поток может быть вызван другим потоком с помощью вызова `SetEvent()`. Переписанный пример «производитель — потребитель» с использованием объектов событий имеет очень простую реализацию:

```
#include <windows.h>

Queue g_queue;
Handle g_hUsed; // инициализируется как ложь (несигнальное состояние)
Handle g_hFree; // инициализируется как истина (сигнальное состояние)

void* ProducerThreadEv(void*)
{
    // продолжаем производить...
    while (true)
    {
        // произвести значение (можно сделать неатомарно,
        // потому что это локальные данные)
        Item item = ProduceItem();

        // (ждем, пока есть место)
        WaitForSingleObject(&g_hFree);

        AddItemToQueue(&g_queue, item);

        // уведомить потребителя, что есть данные
        SetEvent(&g_hUsed);
    }
    return nullptr;
}

void* ConsumerThreadEv(void*)
{
    // продолжаем потреблять...
    while (true)
    {
        // ждем, пока данные будут готовы
        WaitForSingleObject(&g_hUsed);

        Item item = RemoveItemFromQueue(&g_queue);

        // уведомить производителя о наличии места
        SetEvent(&g_hFree);

        // потребляем элемент (можно сделать неатомарно,
        // потому что это локальные данные)
        ConsumeItem(item);
```

```
    }
    return nullptr;
}

void MainThread()
{
    // создаем событие в несигнальном состоянии
    g_hUsed = CreateEvent(nullptr, false,
                          false, nullptr);
    g_hFree = CreateEvent(nullptr, false,
                          true, nullptr);

    // порождаем наши потоки
    CreateThread(nullptr, 0x2000, ConsumerThreadEv,
                 0, 0, nullptr);
    CreateThread(nullptr, 0x2000, ProducerThreadEv,
                 0, 0, nullptr);
    // ...
}
```

4.7. Проблемы с параллелизмом на основе блокировки

В подразделе 4.5.3 мы узнали, что *гонки данных* могут привести к некорректному поведению программы в параллельной системе. Мы увидели, что решение этой проблемы состоит в том, чтобы сделать операции с *общими* объектами данных *атомарными*. Один из способов достижения атомарности состоит в том, чтобы обернуть эти операции в *блокировки*, которые часто реализуются с использованием предоставляемых операционной системой примитивов синхронизации потоков, таких как мьютексы.

Однако атомарность — это только часть параллелизма. Существуют и другие проблемы, которые способны мешать работе параллельной системы, даже если все операции с общими данными тщательно защищены блокировками. В следующих разделах кратко рассмотрим наиболее распространенные.

4.7.1. Взаимная блокировка

Взаимная блокировка (deadlock) — это ситуация, в которой ни один поток в системе не может начать выполняться, что приводит к зависанию. При возникновении взаимной блокировки все потоки находятся в состояний «Блокирован», ожидая, когда какой-либо ресурс станет доступным. Но, поскольку ни один из потоков не находится в состоянии «Готов к выполнению», ни один из ожидаемых ресурсов никогда не станет доступным, поэтому зависает вся программа.

Для возникновения подобной ситуации нужны как минимум два потока и два ресурса. Например, поток 1 удерживает ресурс A, но ожидает ресурс B, а в это же

время поток 2 удерживает ресурс B, но ожидает ресурс A. Иллюстрируется она следующим фрагментом кода:

```
void Thread1()
{
    g_mutexA.lock(); // держит блокировку ресурса A
    g_mutexB.lock(); // спит в ожидании ресурса B
    // ...
}

void Thread2()
{
    g_mutexB.lock(); // держит блокировку ресурса B
    g_mutexA.lock(); // спит в ожидании ресурса A
    // ...
}
```

Конечно, могут возникать и более сложные виды взаимоблокировок, включающие в себя больше потоков и больше ресурсов. Но ключевым фактором, который определяет любую тупиковую ситуацию, является *циклическая* зависимость между потоками и их ресурсами.

Чтобы проанализировать систему на предмет возможности взаимной блокировки, мы можем нарисовать график потоков, ресурсов и зависимостей между ними (рис. 4.34). На этом графике квадратами представлены потоки, кругами — ресурсы (точнее, блокировки мьютекса, которые их защищают). Сплошные стрелки соединяют ресурсы с потоками, в данный момент удерживающими на них блокировки. Пунктирные стрелки соединяют потоки с ресурсами, которые они ждут. Если мы захотим, то можем для простоты исключить из графа узлы ресурсов, оставив только пунктирные линии, соединяющие потоки, которые ожидают другие потоки. Если такой граф зависимостей содержит цикл, возникает взаимная блокировка.

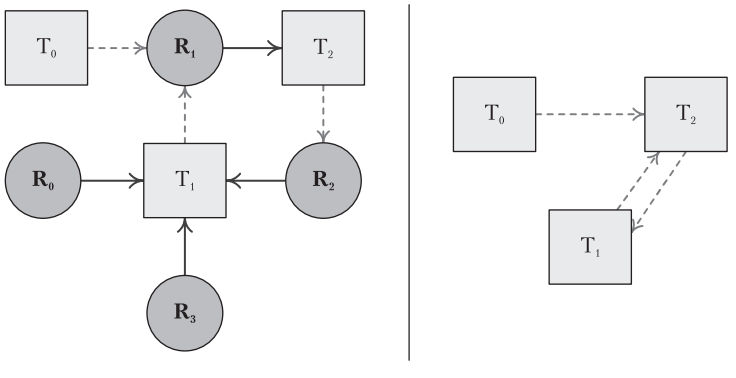


Рис. 4.34. *Слева*: темные стрелки показывают ресурсы, в настоящее время удерживаемые потоками. Светлые пунктирные стрелки показывают потоки, ожидающие, когда ресурсы станут доступными. *Справа*: мы можем исключить ресурсы и просто нарисовать зависимости (светлые пунктирные стрелки) между потоками. Цикл в таком графике зависимостей потока указывает на взаимную блокировку

На самом деле цикла в графе зависимостей недостаточно для возникновения взаимной блокировки. Строго говоря, для этого существуют четыре необходимых и достаточных условия, известных как *условия Коффмана*.

1. *Эксклюзивность мьютекса*. Одному потоку может быть предоставлен эксклюзивный доступ к одному ресурсу через блокировку мьютекса.
2. *Удержание и ожидание*. Поток должен удерживать одну блокировку, когда он засыпает, ожидая другой блокировки.
3. *Нет приоритета блокировки*. Никто, даже ядро, не может насильственно снять блокировку, удерживаемую спящим потоком.
4. *Циклическое ожидание*. В графе зависимостей потоков должен существовать цикл.

Гарантированное предотвращение возникновения взаимной блокировки всегда сводится к обеспечению невыполнения одного или нескольких условий Коффмана. Поскольку нарушение условий 1 и 3 будет обманом, решения обычно ориентированы на то, чтобы избежать условий 2 и 4.

Удержания и ожидания можно избежать, уменьшив количество блокировок. В нашем простом примере, если бы ресурсы A и B были защищены одной блокировкой L, тупиковая ситуация не могла бы возникнуть. Либо поток 1 получит блокировку и монопольный доступ к обоим ресурсам, пока поток 2 ожидает, либо поток 2 получит блокировку, пока поток 1 ожидает.

Условия циклического ожидания можно избежать, установив глобальный порядок для всех блокировок в системе. Если бы в примере с двумя потоками мы смогли обеспечить выполнения условия, что ресурс A всегда блокируется до блокировки ресурса B, то можно было бы избежать взаимной блокировки. Это будет работать, потому что один поток всегда получит блокировку на ресурсе A, прежде чем попытается вызвать любые другие блокировки. Это эффективно переводит все остальные конкурирующие потоки в режим ожидания, тем самым гарантируя, что попытка заблокировать ресурс B всегда будет успешной.

4.7.2. Динамическая взаимная блокировка

Другое решение проблемы взаимной блокировки состоит в том, чтобы потоки *пытались* захватывать блокировки, не переходя в спящий режим (с помощью функции, такой как `pthread_mutex_trylock()`). Если получить блокировку невозможно, поток в течение короткого времени спит, а затем *повторяет попытку* блокировки.

Когда потоки используют явную стратегию, такую как повторные попытки избежать или устранить взаимные блокировки через определенное время, может возникнуть новая проблема: потоки могут тратить все свое время, просто пытаясь устранить взаимную блокировку, а не выполняя реальную работу. Это известно как *динамическая взаимная блокировка* (livelock).

В качестве простого примера динамической блокировки снова рассмотрим потоки 1 и 2, конкурирующие за ресурсы A и B. Всякий раз, когда поток не мо-

жет получить блокировку ресурса, он снимает любые блокировки, которые уже удерживает, и ждет фиксированное время, прежде чем предпринять очередную попытку. Если у обоих потоков одинаковое время ожидания, может сложиться так, что одна и та же вырожденная ситуация будет повторяться снова и снова. Потоки застряли навсегда, пытаясь разрешить конфликт, и ни у одного из них нет шанса выполнить свою настоящую работу. Динамическая взаимная блокировка сродни патовой ситуации в шахматах.

Ее можно избежать, используя алгоритм асимметричного разрешения взаимной блокировки. Например, мы могли бы установить, что только один поток, выбранный случайно или по приоритету, обнаружив взаимную блокировку, может предпринимать действия для ее устранения.

4.7.3. Ресурсное голодание

Под *голоданием* понимается любая ситуация, когда один или несколько потоков не могут получить какое-то время для выполнения на процессоре. Голодание может наблюдаться, если один или несколько потоков с более высоким приоритетом не могут освободить процессор, что препятствует работе потоков с более низким приоритетом. Динамическая взаимная блокировка — одна из разновидностей голодания, при которой алгоритм разрешения взаимоблокировок фактически лишает все потоки их способности выполнять реальную работу.

Голодания при использовании приоритетов потоков обычно стремятся избежать, обеспечивая не очень продолжительную работу высокоприоритетных потоков. В идеале многопоточная программа должна состоять из пула потоков с низким приоритетом, среди которых обычно справедливо распределяются ресурсы ЦП системы. Время от времени запускается поток с более высоким приоритетом, быстро делает свое дело и завершает выполнение, возвращая ресурсы ЦП потокам с низким приоритетом.

4.7.4. Инверсия приоритетов

Блокировки мьютекса могут привести к ситуации, известной как инверсия приоритетов, когда поток с низким приоритетом действует так, как если бы он был потоком с самым высоким приоритетом в системе. Рассмотрим два потока, L и H, с низким и высоким приоритетом соответственно. Поток L блокирует мьютекс и затем вытесняется H. Если H попытается выполнить эту же блокировку, он будет переведен в спящий режим, потому что L уже удерживает блокировку. Это позволяет L продолжать работать даже притом, что он имеет более низкий приоритет, чем H, в нарушение принципа, по которому потоки с более низким приоритетом не должны выполняться, пока поток с более высоким приоритетом готов к выполнению.

Инверсия приоритета может произойти, если поток M со средним приоритетом вытесняет поток L, в то время как последний удерживает блокировку, необходимую для H. Во время работы потока M поток L переходит в спящий режим, предотвращая снятие им блокировки с нужного ресурса. Когда H запускается, он не может

захватить блокировку — он засыпает, и теперь приоритет M эффективно инвертирован с приоритетом потока H.

Последствия инверсии приоритетов могут быть незначительными. Например, если поток с более низким приоритетом немедленно освобождает блокировку, длительность инверсии приоритета может быть короткой и она способна остаться незамеченной или иметь несерьезные последствия. Однако в экстремальных случаях инверсия приоритетов может вызвать взаимную блокировку или другие системные сбои. К примеру, из-за инверсии приоритетов высокоприоритетный поток может не успеть выполнить критически важную операцию в отведенный временной интервал.

Решение проблемы инверсии приоритетов включает в себя следующие приемы.

- Избегайте блокировок, которые могут быть использованы как низко-, так и высокоприоритетными потоками. (Такое решение зачастую неосуществимо.)
- Присвойте очень высокий приоритет мьютексу. Любой поток, который получает мьютекс, имеет приоритет, временно повышенный до приоритета мьютекса, таким образом он гарантированно не будет прерван, пока удерживает блокировку.
- Задайте случайное повышение приоритета. При этом потоки, которые готовы к выполнению, случайным образом повышаются в приоритете. Этот подход используется в модели планировщика Windows.

4.7.5. Обедающие философы

Знаменитая *загадка обедающих философов* — прекрасная иллюстрация проблемы взаимной блокировки, динамической блокировки и голодания. Она описывает следующую ситуацию: пять философов сидят за круглым столом, перед каждым стоит тарелка спагетти. Между каждыми двумя философами лежит одна палочка для еды. Философы хотят чередовать размышления (это они могут делать без палочек) и прием пищи (задача, для которой нужны две палочки). Проблема состоит в том, чтобы определить модель поведения философов, которая гарантирует, что все они могут чередовать размышления и прием пищи, не испытывая взаимной блокировки, динамической блокировки или голодания. (Очевидно, что философы представляют собой потоки, а палочки для еды — мьютексные блокировки.)

Вы можете прочитать об этой задаче в Интернете, поэтому мы не будем долго обсуждать ее здесь. Тем не менее полезно рассмотреть несколько наиболее распространенных решений.

- *Глобальный порядок.* Цикл зависимости может сформироваться, если философ всегда будет сначала брать палочку для еды слева от себя (или всегда справа). Одним из решений этой проблемы является присвоение каждой палочке уникального глобального индекса. Когда философ хочет есть, он всегда берет палочку с самым низким индексом. Это предотвращает циклическую зависимость и, следовательно, взаимоблокировку.

- *Центральный арбитр*. В этом решении центральный арбитр (официант) либо дает философу две палочки для еды, либо не дает ни одной. Это позволяет избежать проблемы удержания и ожидания, гарантируя, что ни один философ не попадет в ситуацию, в которой он держит только одну палочку для еды, тем самым предотвращая взаимную блокировку.
- *Чанди-Мисра*. В этом решении палочки для еды помечаются как грязные или чистые и философы посылают друг другу сообщения, запрашивая палочки. Вы можете узнать больше об этом решении, выполнив поиск в Интернете по термину chandy-misra.
- *N – 1 философов*. Для таблицы с *N* философами и *N* палочками можно использовать целочисленный семафор, чтобы ограничить число философов, которым разрешено брать палочки в любой момент, до *N* – 1. Это решает проблемы взаимоблокировки и динамической блокировки, потому что даже в вырожденных ситуациях по крайней мере одному философу удастся получить две палочки для еды. Это, однако, позволяет одному из философов голодать, если не введен дополнительный критерий справедливости.

4.8. Несколько лучших практик параллелизма

Решения проблемы обедающих философов, которые мы рассмотрели в предыдущем разделе, намекают на некоторые общие принципы и правила, которые мы можем применить практически к любой проблеме параллельного программирования. Кратко рассмотрим некоторые из них.

4.8.1. Правила глобального порядка

В параллельной программе *порядок*, в котором происходят события, не определяется порядком инструкций в программе, как происходит в однопоточной программе. Если в параллельной системе требуется упорядочение, оно должно быть установлено *глобально* для всех потоков.

Это одна из причин, по которой двусвязный список не является параллельной структурой данных. Двусвязный список разработан таким образом, чтобы обеспечить быструю ($O(N)$) вставку и удаление элементов в любом месте списка. В основе этой схемы лежит предположение, что *программный порядок* эквивалентен *порядку данных*: когда упорядоченная последовательность операций выполняется над списком, результирующий список будет содержать соответственно упорядоченные элементы. Например, дан связанный список, который содержит упорядоченные элементы {*A*, *B*, *C*}. Если мы выполним следующие операции:

- вставить *D* перед *C*;
- вставить *E* перед *C*,

то предполагается, что результирующий список будет содержать упорядоченные элементы {*A*, *B*, *D*, *E*, *C*}.

Для однопоточной программы это предположение верно. Но в многопоточной системе программный порядок не диктует порядок данных. Если один поток вставляет *D* перед *C*, а другой — *E* перед *C*, мы получим условие гонки, которое может привести к любому из следующих результатов:

- {*A*, *B*, *D*, *E*, *C*};
- {*A*, *B*, *E*, *D*, *C*};
- {*A*, *B*, поврежденные данные}.

Поврежденный список может возникнуть, если операции структуры данных не защищены должным образом критическими секциями. Например, next-указатели *D* и *E* могут в итоге указывать на *C*.

Правило глобального порядка — единственное жизнеспособное решение этой проблемы. Сначала мы должны спросить себя, почему порядок элементов в списке имеет значение, если он вообще имеет значение. Если порядок неважен, мы можем использовать односвязный список и всегда добавлять элементы — операцию, которая может быть надежно реализована либо с помощью блокировок, либо без них. И если требуется глобальное упорядочение, следует определить устойчивый и детерминированный критерий упорядочения, не зависящий от порядка, в котором происходят события программы. Например, можно отсортировать список по алфавиту, приоритету или другому полезному критерию. Ответы на эти вопросы, в свою очередь, определяют, какую структуру данных мы хотим использовать. Попытка применить двусвязный список параллельно (то есть предоставить нескольким потокам изменяемый доступ к нему) — все равно что пытаться вставить квадратный колышек в круглое отверстие.

4.8.2. Алгоритмы на основе транзакций

В решении с помощью центрального арбитра задачи обедающих философов арбитр, или официант, раздает палочки для еды парами: философ либо получает все необходимые ему ресурсы (две палочки для еды), либо не получает ни одной. Это известно как *транзакция*.

Транзакция может быть более точно определена как неделимый набор ресурсов и/или операций. Потоки в параллельной системе отправляют запросы транзакций некоему центральному арбитру. Транзакция либо целиком завершается успешно, либо целиком терпит неудачу, поскольку транзакция какого-либо другого потока активно обрабатывается при поступлении запроса. Если транзакция завершается неудачно, ее поток продолжает повторно отправлять запрос транзакции, пока она не завершится успешно (возможно, ожидая короткое время между повторными попытками).

Алгоритмы на основе транзакций распространены в параллельных и распределенных системах программирования. И как мы увидим в разделе 4.9, концепция транзакции лежит в основе большинства структур данных и алгоритмов *без блокировок*.

4.8.3. Минимизация раздоров

Наиболее эффективная параллельная система — такая, в которой все потоки работают так, что им не требуется ожидать блокировки. Конечно, эта идеальная ситуация никогда не может быть достигнута, но разработчики параллельных систем пытаются минимизировать количество конфликтов между потоками.

В качестве примера рассмотрим группу потоков, которые создают данные и хранят их в центральном хранилище. Каждый раз, когда один из потоков пытается сохранить свои данные в хранилище, он конкурирует со всеми другими потоками за этот общий ресурс. Простое решение, которое иногда может сработать, — предоставить каждому потоку собственное хранилище. Теперь потоки могут генерировать данные независимо друг от друга, без конфликтов. Когда все потоки готовы к выводу, центральный поток может сопоставить результаты.

Аналогом этого подхода в задаче обедающих философов было бы дать каждому философу две палочки для еды с самого начала. Это, конечно, уничтожит в ней параллелизм — без общих ресурсов нет параллелизма. В реальных параллельных системах мы не можем удалить *все* совместное использование ресурсов, но, безусловно, можем найти способы *минимизировать* его, чтобы свести к минимуму конфликт блокировки.

4.8.4. Безопасность потоков

Вообще говоря, класс или функциональный API называется *потоково-безопасным*, когда его функции могут безопасно вызываться любым потоком в многопоточном процессе. Для любой функции безопасность потока обычно достигается входом в критическую секцию в верхней части тела функции, выполнением некоторой работы и последующим выходом из критической секции непосредственно перед возвратом. Класс, все функции которого являются потоково-ориентированными, иногда называют *монитором*. (Термин «монитор» используется также для обозначения класса, который задействует условные переменные, чтобы разрешить клиентам спать, ожидая, пока их защищенные ресурсы станут доступными.)

Потоковая безопасность — это удобная функция для класса или интерфейса. Но она вызывает дополнительные расходы, которые иногда не нужны. Например, эти издержки будут бесполезными, если интерфейс задействуется в однопоточной программе или используется исключительно одним потоком в многопоточной программе.

Потоковая безопасность также может стать проблемой, когда интерфейсную функцию необходимо вызывать повторно. Например, если класс предоставляет потоково-ориентированные функции `A()` и `B()`, они не могут вызывать друг друга, поскольку каждая из них независимо входит в критическую секцию и покидает ее. Одно из решений этой проблемы — применение реентерабельных блокировок (см. подраздел 4.9.7). Второе — реализовать небезопасные версии функций в интерфейсе, а затем обернуть каждую из них в потоково-безопасный вариант, который просто входит в критическую секцию, вызывает небезопасную функцию

и затем покидает критическую секцию. Таким образом, мы можем вызывать небезопасные функции внутри системы, но внешний интерфейс системы остается потоково-безопасным.

На мой взгляд, не стоит пытаться использовать параллельное программирование, просто создавая классы и API, которые на 100 % потоково-ориентированы. Это приводит к появлению ненужных тяжелых интерфейсов и побуждает программистов игнорировать то, что они работают в параллельной среде. Вместо этого следует признать наличие параллелизма в своем программном обеспечении и принять его, а также стремиться разрабатывать структуры данных и алгоритмы, которые явно обращаются к параллелизму. Целью должно стать создание программной системы, которая минимизирует конфликты, зависимости между потоками и применение блокировок. Создание безблокировочной системы — это большая работа, трудно сделать все правильно (см. раздел 4.9). Но *стремиться* к свободе от блокировок (то есть минимизировать их применение) гораздо лучше, чем чрезмерно использовать их в тщетной попытке создать «водонепроницаемые» интерфейсы, которые позволяют программистам не обращать внимания на параллелизм.

4.9. Параллелизм без блокировок

До сих пор все решения проблемы состояний гонки в параллельной системе вращались вокруг использования *мьютекс-блокировок*, позволяющих сделать критические операции атомарными, и, возможно, применения *условных переменных* и способности ядра *переводить потоки в спящий режим*, чтобы они не тратили ценные циклы ЦП во время ожидания. В процессе обсуждения мы упомянули, что есть еще один способ избежать условий гонки, который потенциально может быть более эффективным. Этот подход известен как *параллелизм без блокировок (lock-free)*.

Вопреки распространенному мнению, термин «без блокировок» не относится к устранению мьютекс-блокировок, хотя это, безусловно, является частью подхода. Фактически «без блокировки» относится к практике предотвращения перехода потоков в спящий режим в ожидании доступности ресурса. Другими словами, в программировании без блокировок мы никогда не позволяем потоку *блокироваться*. Так что, возможно, термин «свободный от блокировок» (blocking-free) лучше описывал бы ситуацию.

Программирование без блокировок — это лишь один метод из набора *неблокирующих* методов параллельного программирования. Когда поток заблокирован, он перестает выполнять вычисления. Цель всех этих методов — гарантировать *прогресс* всех потоков в системе и системы в целом. Мы можем организовать эти методы в следующие категории, перечисленные в порядке возрастания степени гарантированности прогресса, которую они обеспечивают.

- *Блокировка.* Алгоритм блокировки — это алгоритм, в котором поток можно перевести в спящий режим в ожидании доступности общих ресурсов. При исполь-

зовании блокирующего алгоритма возможны взаимоблокировка, динамическая блокировка, голодание и инверсия приоритетов.

- *Свобода от препятствий.* Алгоритм свободен от препятствий, если мы можем гарантировать, что один поток всегда завершит свою работу за ограниченное число шагов, если все другие потоки в системе внезапно приостановятся. Говорят, что если продолжает работать единственный поток, происходит *одиночное выполнение*, и алгоритм без препятствий иногда называют *соло-завершением*, потому что одиночный поток в итоге завершается, в то время как все другие потоки приостановлены. Ни один алгоритм, использующий мьютекс- или спин-блокировку, не может быть беспрепятственным, так как, если какой-либо поток должен быть приостановлен, пока он удерживает блокировку, одиночный поток может застрять навсегда в ее ожидании.
- *Свобода от блокировок.* Формальное определение свободы от блокировки состоит в том, что при любом бесконечно долгом запуске программы будет выполнено бесконечное количество операций. Интуитивно понятно, что алгоритм без блокировок гарантирует, что какой-то поток в системе всегда сможет добиться прогресса. Другими словами, если один поток произвольно приостановлен, все остальные могут все еще выполняться. Это опять-таки исключает использование мьютексных или спиновых блокировок, поскольку то, что поток, удерживающий блокировку, приостановлен, может привести к блокировке других потоков. Алгоритм, свободный от блокировки, обычно основан на транзакциях: транзакция может завершиться неудачей, если другой поток прервет ее, в этом случае она откатывается и повторяется до тех пор, пока не будет выполнена успешно. Такой подход позволяет избежать ситуации взаимоблокировки, но он может позволить некоторым потокам голодать. Другими словами, некоторые потоки могут застрять в цикле неудачных транзакций и повторных попыток их реализации на неопределенный срок, в то время как транзакции других потоков всегда выполняются успешно.
- *Свобода от ожидания.* Алгоритм без ожидания гарантирует свободу от блокировки и невозможность голодания. Другими словами, все потоки могут прогрессировать, и ни одному из них не разрешается голодать бесконечно.

Термин «программирование без блокировок» иногда свободно используется для обозначения любого алгоритма, который избегает блокировок, но формально правильное название алгоритмов без препятствий, блокировок и ожидания в целом — это *«неблокирующий алгоритм»*.

Тема неблокирующих алгоритмов обширна и остается открытой областью исследований. Полное ее обсуждение потребовало бы целой книги. Далее мы познакомимся с некоторыми основными принципами программирования без блокировок. Начнем с изучения истинных причин ошибок при гонке данных. Затем посмотрим, как мьютексы фактически реализуются «под капотом», и узнаем, как реализовать собственные недорогие *спин-блокировки*. Наконец, представим реализацию простого связанного списка без блокировок. Этого должно быть достаточно, чтобы

дать вам представление о том, как обычно выглядят структуры данных и алгоритмы без блокировок, и обеспечить надежную отправную точку для дальнейшего чтения и экспериментов с методами без блокировок.

4.9.1. Причины ошибок в гонке данных

Как мы видели в подразделе 4.5.3, ошибка гонки данных может возникать, когда критическая операция прерывается другой критической операцией над теми же данными. Как выясняется, есть и еще два случая, когда могут возникать ошибки гонки данных, и если мы собираемся реализовать собственные спин-блокировки или написать алгоритмы без блокировок, нужно разобраться в них. Ошибки гонки данных могут быть вызваны в параллельной программе:

- *прерыванием* одной критической операции другой;
- оптимизацией *переупорядочения команд*, выполняемой компилятором и процессором;
- в результате аппаратно-ориентированной *семантики упорядочения памяти*.

Рассмотрим их подробнее.

- Потоки все время прерывают друг друга в результате вытесняющей многозадачности и/или запуска их на нескольких ядрах. Тем не менее, когда *критическая* операция прерывается другой критической операцией над *общим* объектом, может возникнуть ошибка гонки данных.
- Оптимизирующие компиляторы часто переупорядочивают инструкции в попытке минимизировать задержки конвейера. Аналогично логика выполнения не по порядку в ЦП может привести к выполнению команд в порядке, отличном от программного. Переупорядочение инструкций гарантированно не изменит наблюдаемое поведение однопоточной программы. Но оно *может* изменить способ *взаимодействия* двух или более потоков при совместном использовании данных, тем самым внося ошибки в параллельную программу.
- Благодаря агрессивной оптимизации в контроллере памяти компьютера *эффекты* инструкции чтения или записи могут иногда *задерживаться* относительно других операций чтения и/или записи в системе. Как и в случае с оптимизацией компилятора и выполнением не по порядку (out-of-order, OOO), эти оптимизации контроллера памяти не изменяют наблюдаемое поведение однопоточной программы. Тем не менее они *могут* изменить порядок критической пары операций чтения и/или записи в параллельной системе, тем самым предотвращая потоки от совместного использования данных предсказуемым способом. В этой книге мы будем ссылаться на такие ошибки параллелизма, как *ошибки упорядочения памяти*.

Чтобы гарантировать, что критическая операция свободна от гонок данных и, следовательно, является атомарной, мы должны гарантировать, что не может произойти ничто из перечисленного ранее.

4.9.2. Реализация атомарности

Вначале давайте рассмотрим проблему превращения критической операции в атомарную, то есть непрерываемую. Раньше мы махали руками и говорили, что, обернув критические операции в пару «блокировка/разблокировка *мьютекса*», мы можем как по волшебству превратить их в атомарные операции. Но как на самом деле работает мьютекс?

Атомарность путем отключения прерываний

Чтобы другие потоки не прерывали нашу работу, мы могли бы попытаться отключить прерывания непосредственно перед выполнением операции, убедившись, что их снова включили после завершения операции. Это, безусловно, помешало бы ядру переключиться в контекст другого потока в середине атомарной операции. Но этот подход работает только на одноядерной машине с использованием вытесняющей многозадачности.

Прерывания отключаются путем выполнения инструкции на машинном языке (например, `cli` (clear interruptenable bit) — «очистить бит прерывания» в архитектуре Intel x86). Но этот вид инструкции влияет только на ядро, которое ее выполнило. Другие ядра будут продолжать выполнять свои потоки (со все еще включенными прерываниями и, следовательно, с вытесняющей многопоточностью), которые все еще могут прерывать нашу работу. Таким образом, этот подход имеет ограниченную применимость в реальном мире.

Атомарные инструкции

Термин «атомарный» предлагает подход с разбиением операции на все более мелкие кусочки, пока не будет получена неделимая гранула. И тут возникает вопрос: существуют ли *естественно* атомарные инструкции машинного языка? Другими словами, гарантирует ли процессор, что некоторые инструкции являются непрерываемыми?

Ответ на эти вопросы — «да», но с некоторыми оговорками. Скорее всего, есть определенные инструкции машинного языка, которые *никогда* не могут выполняться атомарно. Другие инструкции являются атомарными, но только в ходе работы с определенными видами данных. Некоторые процессоры допускают *принудительное* выполнение практически любой инструкции атомарно путем указания префикса для инструкции на языке ассемблера. (Один из примеров — префикс `lock` в Intel x86 ISA.)

Это хорошая новость для разработчиков параллельного ПО. Фактически именно существование *атомарных инструкций* позволяет нам реализовывать инструменты атомарности, такие как мьютексы и спин-блокировки, которые, в свою очередь, дают возможность делать атомарными операции большего масштаба.

Разные процессоры и ISA предоставляют разные наборы атомарных инструкций, управляемых по разным правилам. Но мы можем свести все атомарные инструкции в две категории:

- атомарные чтение и запись;
- атомарные инструкции чтения — изменения — записи (read — modify — write, RMW).

Атомарные чтение и запись

Работая с большинством процессоров, мы можем быть уверены, что чтение или запись 32-разрядного целого числа с выравниванием 4 байта будет атомарной операцией. При этом любой процессор отличается от прочих, поэтому важно всегда обращаться к документации ISA своего процессора, прежде чем полагаться на атомарность какой-либо конкретной инструкции.

Некоторые процессоры также поддерживают атомарные операции чтения и записи для более мелких или больших объектов, таких как одиночные байты или 64-разрядные целые числа, опять-таки предполагая, что они выровнены по размеру, кратному их собственному. Это верно, потому что на большинстве процессоров чтение и запись выровненного целого числа, ширина которого в битах равна ширине регистра (или иногда ширине строки кэша) или меньше его, может выполняться за один *цикл доступа к памяти*. Поскольку процессор работает в режиме блокировки с дискретными часами, цикл памяти не может быть прерван даже другим ядром. В результате чтение или запись фактически являются атомарными.

Смещенные чтение и запись обычно не имеют свойства атомарности. Это связано с тем, что для чтения или записи смещенного объекта процессор обычно выполняет два выровненных доступа к памяти. Таким образом, чтение или запись могут быть прерваны, и мы не можем предполагать, что они будут атомарными. (В подразделе 3.3.7 приведена более подробная информация о выравнивании.)

Атомарные чтение — изменение — запись

Атомарных чтения и записи недостаточно для реализации атомарных операций в общем смысле. Чтобы реализовать механизм блокировки, такой как мьютекс, мы должны иметь возможность читать содержимое переменной из памяти, выполнять с ней некоторые операции, а затем записывать результаты обратно в память, и все это без прерывания.

Все современные процессоры поддерживают параллелизм, предоставляя как минимум одну атомарную инструкцию чтения — изменения — записи (RMW). В следующих разделах мы рассмотрим несколько видов атомарных инструкций RMW. У каждой из них есть свои плюсы и минусы. Все они могут использоваться для реализации мьютекса или спин-блокировки.

Тестирование и установка

Самая простая инструкция RMW известна как *тестирование и установка* (test-and-set, TAS). Инструкция TAS фактически не проверяет и не устанавливает значение. Скорее, она атомарно устанавливает логическую переменную в 1 (`true`) и возвращает свое *предыдущее* значение, чтобы можно было его проверить:

```
// псевдокод для инструкции test-and-set
bool TAS(bool* pLock)
{
    // атомарно...
    const bool old = *pLock;
    *pLock = true;
    return old;
}
```

Команду `test-and-set` можно задействовать для реализации простого вида блокировки, называемой *спин-блокировкой*. Вот некоторый псевдокод, демонстрирующий идею. В этом примере мы использовали гипотетический *встроенный компилятор*, называемый `_tas()`, для передачи инструкции машинного языка TAS в наш код. Разные компиляторы будут предоставлять разные встроенные функции для этой инструкции, если целевой процессор ее поддерживает. Например, в Visual Studio встроенная функция TAS называется `_interlockedbittestandset()`.

```
void SpinLockTAS(bool* pLock)
{
    while (_tas(pLock) == true)
    {
        // кто-то другой заблокировал – занято – ждем...
        PAUSE();
    }

    // когда мы попадаем сюда, мы знаем, что успешно сохранили
    // значение true в *pLock и что оно ранее содержало false,
    // так что больше никто не блокирует – конец
}
```

Здесь макрос `PAUSE()` указывает на использование встроенной функции компилятора, такой как `_mm_pause()` (Intel SSE2), для снижения энергопотребления во время цикла ожидания занятости. В подразделе 4.4.6 приведена подробная информация о том, почему рекомендуется применять инструкцию паузы в цикле ожидания, где это возможно.

Здесь важно подчеркнуть, что приведенный ранее пример предназначен только для иллюстративных целей. Он не на 100 % правилен, потому что в нем недостает надлежащей защиты памяти. Мы представим полный рабочий пример спин-блокировки в подразделе 4.9.7.

Обмен

Некоторые ISA, такие как Intel x86, предлагают инструкцию атомарного *обмена*. Эта инструкция меняет местами содержимое двух регистров или регистра и ячейки памяти. На архитектуре x86 она атомарна по умолчанию при обмене регистра с памятью, то есть действует так, как если бы инструкции предшествовал префикс `lock`.

Вот как реализовать спин-блокировку с помощью инструкции атомарного *обмена*. В этом примере мы используем встроенную функцию компилятора Visual Studio `_InterlockedExchange()` для передачи инструкции Intel x86 `xchg` в наш код. (Обратите внимание на то, что без надлежащей защиты памяти этот пример неполон и не будет работать надежно. Полная реализация приведена в подразделе 4.9.7.)

```
void SpinLockXCHG(bool* pLock)
{
    bool old = true;
    while (true)
    {
        // выдает инструкцию xchg для восьмибитных слов
        _InterlockedExchange8(old, pLock);
        if (!old)
        {
            // если мы получим false в качестве возврата,
            // значит, блокировка прошла успешно
            break;
        }
        PAUSE();
    }
}
```

В Microsoft Visual Studio все взаимоблокирующие функции с начальным подчеркиванием у названия являются *встроенными в компилятор* — они просто выдают соответствующую инструкцию на ассемблере прямо в код. Пакет Windows SDK предоставляет набор функций с аналогичными именами *без* начального подчеркивания — эти функции, где это возможно, реализуются в терминах встроенных функций, но они намного дороже, поскольку требуют вызова ядра.

Сравнить и поменять местами

Некоторые процессоры предоставляют инструкцию, известную как *сравнение и замена* (compare-and-swap, CAS). Она проверяет существующее значение в ячейке памяти и атомарно заменяет его новым значением тогда и только тогда, когда существующее значение соответствует ожидаемому, предоставленному программистом. Инструкция возвращает `true`, если операция прошла успешно, что означает: в ячейке памяти содержалось ожидаемое значение. И возвращает `false`, если операция завершилась неудачно, поскольку содержимое ячейки не соответствует ожидаемому.

Инструкция CAS может работать со значениями, превышающими логическое значение. Варианты обычно предоставляются по крайней мере для 32- и 64-битных целых чисел, хотя могут поддерживаться и меньшие размеры слов.

Поведение инструкции CAS иллюстрируется следующим псевдокодом:

```
// Псевдокод для инструкции сравнения и замены
bool CAS(int* pValue, int expectedValue, int newValue)
{
    // атомарно...
    if (*pValue == expectedValue)
    {
        *pValue = newValue;
        return true;
    }
    return false;
}
```

Для реализации любой атомарной операции чтения — изменения — записи с использованием CAS обычно применяется следующая стратегия.

1. Считать старое значение переменной, которую мы пытаемся обновить.
2. Изменить значение любым удобным способом.
3. При записи результата задействовать инструкцию CAS вместо обычной записи.
4. Повторять, пока CAS не завершится успехом.

Инструкция CAS позволяет обнаруживать гонку данных, сравнивая значение, которое фактически находится в ячейке памяти во время записи, со значением *до* вызова операции чтения — изменения — записи. В отсутствие гонки инструкция CAS действует как инструкция записи. Но если значение в памяти изменяется между чтением и записью, мы знаем, что какой-то другой поток опередил нас. В этом случае мы отступаем и пытаемся снова.

Реализация спин-блокировки с помощью сравнения и замены выглядела бы примерно так, как показано далее, опять же с использованием гипотетической встроенной функции компилятора `_cas()` для генерации инструкции CAS. (В этом примере вновь пропущена защита памяти, которая потребуется для надежной работы на всех устройствах, — см. подраздел 4.9.7, где рассказывается о полнофункциональной спин-блокировке.)

```
void SpinLockCAS(int* pValue)
{
    const int kLockValue = -1; // 0xFFFFFFFF
    while (!_cas(pValue, 0, kLockValue))
    {
        // должно быть заблокировано кем-то еще – попробовать снова
        PAUSE();
    }
}
```

А вот так мы реализовали бы атомарный инкремент, используя CAS:

```
void AtomicIncrementCAS(int* pValue)
{
    while (true)
    {
        const int oldValue = *pValue; // атомарное чтение
        const int newValue = oldValue + 1;
        if (_cas(pValue, oldValue, newValue))
        {
            break; // Успех!
        }
        PAUSE();
    }
}
```

На Intel x86 ISA инструкция CAS называется `cmpxchg`, и ее можно генерировать с помощью встроенной функции компилятора Visual Studio `_InterlockedCompareExchange()`.

Проблема ABA

Следует отметить, что инструкция CAS страдает от неспособности обнаружить один конкретный тип гонки данных. Рассмотрим атомарную операцию RMW, в которой чтение видит значение A. Прежде чем мы сможем выполнить инструкцию CAS, другой поток вытесняет нас или работает на другом ядре и записывает два значения в то место, которое мы пытаемся атомарно обновить: сначала значение B, а затем снова значение A. Когда инструкция CAS наконец-то будет выполнена, она не сможет определить разницу между первым прочитанным A и A, записанным другим потоком. Следовательно, она будет думать, что никакой гонки данных не возникло, хотя на самом деле это не так. Данная ситуация известна как проблема ABA.

Загрузка с пометкой/условная запись

Некоторые процессоры разбивают операцию сравнения и обмена на *пару* инструкций, известных как *загрузка с пометкой* и *условная запись* (load linked/store conditional, LL/SC). *Загрузка с пометкой* читает значение ячейки памяти атомарно и сохраняет адрес в специальном регистре ЦП, известном как *регистр метки*. Инструкция *условной записи* записывает значение в заданный адрес, но только в том случае, если он соответствует содержимому регистра метки. Она возвращает истину, если запись прошла успешно, или ложь, если она не удалась.

Любая операция записи на шину (включая условную запись) очищает регистр метки в ноль. Это означает, что пара команд LL/SC способна обнаруживать гонки данных, потому что, если будет сделана какая-либо запись между командами LL и SC, SC потерпит неудачу.

Пара команд LL/SC используется почти так же, как обычное чтение в паре с инструкцией CAS. В частности, атомарная операция чтения — изменения — записи будет реализована с применением следующей стратегии.

1. Считать старое значение переменной с помощью инструкции LL.
2. Изменить значение любым удобным способом.
3. Записать результат с помощью инструкции SC.
4. Повторять, пока SC не закончится успехом.

Вот как мы реализовали бы атомарный инкремент с помощью LL/SC (используя гипотетические встроенные функции компилятора `_ll()` и `_sc()` для создания инструкций LL и SC соответственно):

```
void AtomicIncrementLLSC(int* pValue)
{
    while (true)
    {
        const int oldValue = _ll(*pValue);
        const int newValue = oldValue + 1;
        if (_sc(pValue, newValue))
        {
            break; // Успех!
        }
        PAUSE();
    }
}
```

Поскольку регистр метки очищается любой записью шины, инструкция SC может внезапно завершиться ошибкой. Но это не повлияет на правильность атомарного RMW, реализованного с помощью LL/SC, — это просто означает, что в цикле ожидания может быть выполнено несколько больше итераций, чем мы предполагали.

Преимущества LL/SC над CAS

Пара команд LL/SC имеет два явных преимущества перед инструкцией CAS.

Во-первых, поскольку инструкция SC не выполняется *всякий* раз, когда выполняется какая-либо запись на шине, для пары LL/SC не характерна проблема ABA.

Во-вторых, пара инструкций LL/SC более дружественна к конвейеру, чем инструкция CAS. Простейший конвейер — это пять этапов: получение, декодирование, выполнение, доступ к памяти и обратная запись в регистр. Но инструкция CAS требует двух циклов доступа к памяти: одного для чтения ячейки памяти, чтобы можно было сравнить ее значение с ожидаемым, а другого — для записи результата, если сравнение прошло удачно. Это означает, что конвейер, который поддерживает CAS, должен включать дополнительный этап доступа к памяти, который бóльшую часть времени не используется. В то же время для команд LL и SC требуется только один цикл доступа к памяти, поэтому они более естественно

вписываются в конвейер с одним этапом доступа к памяти. Сравнение CAS и LL/SC с точки зрения конвейеризации показано на рис. 4.35.

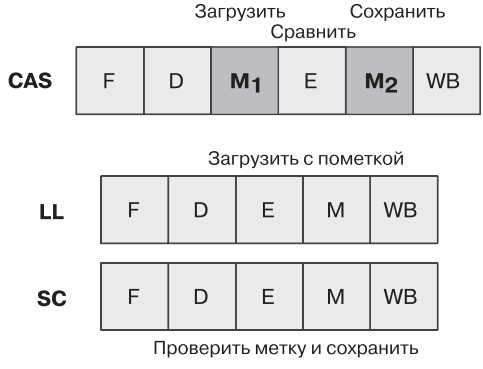


Рис. 4.35. Инструкция сравнения и обмена (CAS) считывает ячейку памяти, выполняет сравнение и затем условно записывает в эту же ячейку. Следовательно, она требует двух этапов доступа к памяти, что делает ее более сложной для реализации в простой конвейерной архитектуре ЦП, чем пара команд загрузки с меткой/условной записью (LL/SC), каждая из которых требует только одного этапа доступа к памяти

Сильная и слабая операции сравнения и обмена

Стандартная библиотека C++11 предоставляет переносимые функции для выполнения атомарных операций сравнения — обмена. Они могут быть реализованы инструкцией CAS на некотором целевом оборудовании или парой инструкций LL/SC на другом аппаратном обеспечении. Из-за вероятности ложных сбоев инструкции условной записи C++11 предоставляет два варианта сравнения — обмена: сильный и слабый. Сильная операция сравнения — обмена скрывает сбои проверки от программиста, а слабая — нет. Поищите в Интернете статью *Strong Compare and Exchange* («Сильные сравнение и обмен» Лоуренса Кроула, описывающую сильную и слабую функции сравнения — обмена) в C++11.

Относительная сила атомарных инструкций RMW

Интересно отметить, что инструкция TAS слабее инструкций CAS и LL/SC с точки зрения достижения *согласованности* между несколькими потоками в параллельной системе. Консенсус в этом контексте относится к соглашению между потоками о значении общей переменной, даже если в некоторых потоках в системе произошел сбой.

Поскольку инструкция TAS работает только с логическим значением, она может решить проблему, известную как *проблема консенсуса без ожидания*, для двух параллельных потоков. Инструкция CAS работает с 32-битным значением, поэтому она может решить эту проблему для любого количества потоков.

Тема консенсуса без ожидания выходит за рамки этой книги, она представляет интерес в основном при создании отказоустойчивых систем. Если вы хотите улучшить отказоустойчивость, можете узнать больше о проблеме консенсуса, выполнив поиск по словам «консенсус (информатика)» в «Википедии».

4.9.3. Барьеры

Прерывания не единственная причина ошибок, возникающих при гонке данных. Компиляторы и центральные процессоры также склонны вносить незначительные ошибки в параллельные программы посредством оптимизации *переупорядочением команд*, которую они выполняют, как описано в подразделе 4.2.5.

Основное правило оптимизации компиляторов и выполнения не по порядку состоит в том, что их оптимизация не должна оказывать видимого влияния на поведение *отдельного потока*. Однако ни компилятор, ни логика управления ЦП не знают, какие еще потоки могут выполняться в системе или что они могут делать. В результате этого кардинального правила недостаточно, чтобы предотвратить ошибки параллельных программ, возникающие в результате переупорядочения команд.

Примитивы синхронизации потоков, предоставляемые операционной системой (мьютексы и др.), тщательно обрабатывают, чтобы избежать ошибок параллелизма, которые могут быть вызваны оптимизацией в результате переупорядочения команд. Теперь, когда мы исследовали, как реализованы мьютексы, посмотрим, как вручную не допустить этих проблем.

Как переупорядочение инструкций вызывает ошибки параллелизма

В качестве примера проблем, связанных с переупорядочением команд в параллельном программном обеспечении, вновь рассмотрим проблему «производитель — потребитель» из подраздела 4.6.3. Мы упростили пример и удалили мьютексы, чтобы выявить ошибки, которые могут быть вызваны переупорядочением команд:

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread()
{
    // получить некоторые данные
    g_data = 42;

    // информируем потребителя
    g_ready = 1;
}

void ConsumerThread()
{
    // ждем, пока данные не будут готовы
    while (!g_ready)
        PAUSE();

    // потребляем данные
    ASSERT(g_data == 42);
}
```

На процессоре, в котором выровненные чтение и запись 32-разрядных целых чисел являются атомарными, этот пример фактически не требует мьютексов. Тем не менее ничто не мешает компилятору или логике внеочередного выполнения CPU переупорядочить запись 1 в `g_ready` так, чтобы это происходило *до* записи `42` в `g_data`. Аналогично в теории компилятор может переупорядочить проверку, что `g_data` равно `42`, так, что это произойдет *до* цикла `while`. Таким образом, даже если все операции чтения и записи являются атомарными, этот код может работать ненадежно.

Переупорядочение инструкций действительно происходит на уровне ассемблера, поэтому оно может быть намного более тонким, чем переупорядочение операторов в программе C/C++. Например, следующий код C/C++:

```
A = B + 1;
B = 0;
```

будет производить такой код ассемблера Intel x86:

```
mov  eax,[B]
add  eax,1
mov  [A],eax
mov  [B],0
```

Компилятор может легко переупорядочить инструкции следующим образом, не производя заметного эффекта в однопоточном исполнении:

```
mov eax,[B]
mov [B],0 ;; Записать Б перед А!
add eax,1
mov [A],eax
```

Если бы второй поток ожидал, что `B` станет равным нулю, прежде чем прочитать значение `A`, он перестал бы работать правильно в случае применения этой оптимизации компилятора.

Джефф Прешинг сделал в блоге отличную запись на эту тему, доступную по адресу preshing.com/20120625/memory-ordering-at-compile-time/. (Вот откуда взялся приведенный ранее пример на языке ассемблера.) Я настоятельно рекомендую прочесть посты Джеффа по параллельному программированию.

Волатильность в C/C++, и почему она нам не помогает

Как мы можем помешать компилятору переупорядочить критически важные последовательности операций чтения и записи? В C и C++ квалификатор типа `volatile` гарантирует, что последовательное чтение или запись переменной не могут быть оптимизированы компилятором, так что это кажется многообещающей идей. Тем не менее по ряду причин она не работает.

Квалификатор `volatile` в C/C++ действительно был создан для обеспечения надежных действий обработчиков ввода-вывода с отображением в память. Таким образом, единственная гарантия, которую он предоставляет, заключается в том, что содержимое переменной, помеченной как `volatile`, не будет кэшироваться в ре-

гистре — значение переменной станет считываться непосредственно из памяти каждый раз, когда к ней обращаются. Некоторые компиляторы гарантируют, что инструкции не будут переупорядочены при чтении или записи изменяемой переменной, но не все они это делают, а некоторые предоставляют такую гарантию только при определенных целевых процессорах или только при передаче определенной команды компилятору. Стандарты C и C++ не требуют такого поведения, поэтому мы не можем полагаться на него при написании переносимого кода. (На странице msdn.microsoft.com/en-us/magazine/dn973015.aspx эта тема обсуждается более глубоко.)

Более того, ключевое слово `volatile` в C/C++ не делает ничего, чтобы помешать внеочередной логике выполнения CPU изменить порядок инструкций во время выполнения. И это также не может помочь избежать проблем, связанных с согласованностью кэша. Таким образом, по крайней мере в C и C++ ключевое слово `volatile` не поможет написать надежное параллельное программное обеспечение[1].

Барьеры компилятора

Один из надежных способов помешать компилятору переупорядочить инструкции чтения и записи внутри границ критических операций — явно дать ему указание не делать так. Для этого следует вставить в код специальную псевдоинструкцию, известную как *барьер компилятора*.

Различные компиляторы обозначают барьеры с помощью разного синтаксиса. С GCC барьер компилятора может быть вставлен через некоторый встроенный синтаксис ассемблера, как показано далее (с Microsoft Visual C++ тот же эффект достигается встроенной функцией `_ReadWrite Barrier()` компилятора):

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread()
{
    // получить некоторые данные
    g_data = 42;

    // Уважаемый компилятор, пожалуйста, переупорядочивая
    // инструкции, не пересекайте этот барьер!
    asm volatile("" ::: "memory")

    // информируем потребителя
    g_ready = 1;
}

void ConsumerThread()
{
```

[1] В некоторых языках, включая Java и C#, квалификатор типа volatile действительно гарантирует атомарность и может использоваться для реализации параллельных структур данных и алгоритмов. В подразделе 4.9.6 больше информации по этой теме.

```
    // ждем, пока данные не будут готовы
    while (!g_ready)
        PAUSE();

    // Уважаемый компилятор, пожалуйста, переупорядочивая
    // инструкции, не пересекайте этот барьер!
    asm volatile("" ::: "memory")

    // потребляем данные
    ASSERT(g_data == 42);
}
```

Есть и другие способы помешать компилятору переупорядочивать инструкции. Например, большинство вызовов функций служат неявным барьером для компилятора. Это имеет смысл, потому что компилятор ничего не знает[1] о побочных эффектах вызова функции. Таким образом, он не может предполагать, что состояние памяти будет одинаковым до и после вызова, что означает: большинство оптимизаций недопустимы при вызове функции. Некоторые оптимизирующие компиляторы делают исключение из этого правила для встроенных функций.

К сожалению, барьеры компилятора не препятствуют тому, чтобы внеочередная логика исполнения CPU изменяла порядок инструкций во время выполнения. Некоторые ISA предоставляют специальную инструкцию для этой цели, например инструкцию `isync` в PowerPC. В подразделе 4.9.4 мы узнаем о наборе инструкций машинного языка, известных как *ограничения памяти*, которые служат барьерами переупорядочения команд как для компилятора, так и для ЦП. И что более важно, они также предотвращают ошибки *переупорядочения памяти*. Так что атомарные инструкции и барьеры — это все, что нам действительно нужно для написания надежных мьютексов, а также спин-блокировок и других алгоритмов *без блокировок*.

4.9.4. Семантика упорядочения памяти

В подразделе 4.9.1 говорилось: вдобавок к тому, что компилятор или ЦП фактически переупорядочивают инструкции машинного языка в наших программах, можно *эффективно переупорядочивать* команды чтения и записи в параллельной системе. В частности, в многоядерной машине с многоуровневым кэшем памяти два ядра или более могут иногда *не соглашаться* с очевидной последовательностью команд чтения и записи, даже когда инструкции *фактически* выполнялись в том порядке, в котором мы хотели. Очевидно, что такие разногласия могут вызвать незначительные ошибки в параллельном программном обеспечении.

Эти загадочные и неприятные проблемы могут возникнуть только на многоядерном оборудовании с многоуровневым кэшем. Одно ядро ЦП всегда видит результаты выполнения собственных инструкций чтения и записи в порядке их

[1] Функциональные вызовы служат неявными барьерами, когда компилятор не может увидеть определение функции, — например, когда функция определена в отдельной единице компиляции. Оптимизация во время линковки (LTO) может вызвать ошибки параллелизма, позволяя оптимизатору компилятора видеть определения функций, которые он иначе не мог бы видеть, и тем самым эффективно устраняя неявные барьеры.

выполнения, разногласия могут возникнуть только при наличии двух или более ядер. Более того, разные процессоры по-разному ведут себя при упорядочении памяти, что означает: эти странные эффекты могут различаться от компьютера к компьютеру при запуске одной и той же исходной программы.

К счастью, еще не все потеряно. Каждым процессором управляет строгий набор правил, известный как *семантика упорядочения памяти*. Эти правила предоставляют различные гарантии распределения операций чтения и записи между ядрами, и они также дают программистам инструменты, необходимые для обеспечения определенного порядка, когда семантики по умолчанию недостаточно.

Одни процессоры предлагают только слабые гарантии по умолчанию, другие — более сильные гарантии и, следовательно, требуют меньшего вмешательства программиста. Итак, если мы сможем понять, как преодолеть проблемы с упорядочением памяти на процессоре с помощью самой слабой семантики упорядочения памяти, то можем быть уверены, что эти методы будут работать и на процессорах с более строгой семантикой по умолчанию.

Кэширование памяти

Чтобы понять, как могут происходить загадочные эффекты переупорядочения памяти, нужно более внимательно посмотреть, как работает многоуровневый кэш памяти.

В подразделе 3.5.4 мы подробно рассмотрели, как кэш-память избегает очень высокой задержки доступа к основной оперативной памяти, сохраняя часто используемые данные в кэше. Это означает, что до тех пор, пока объект данных присутствует в кэше, ЦП всегда будет пытаться работать с этой кэшированной копией, а не обращаться к самому объекту.

Кратко рассмотрим, как это работает, взглянув на следующую простую (и полностью придуманную) функцию:

```
constexpr int      COUNT = 16;
alignas(64) float  g_values[COUNT];
float              g_sum = 0.0f;

void CalculateSum()
{
    g_sum = 0.0f;
    for (int i = 0; i < COUNT; ++i)
    {
        g_sum += g_values[i];
    }
}
```

Первое утверждение устанавливает `g_sum` в ноль. Предполагая, что содержимое `g_sum` еще не в кэше L1, строка кэша, содержащая его, на этом этапе будет считана в L1. Аналогично на первой итерации цикла строка кэша, содержащая все элементы массива `g_values`, будет загружена в L1. (Все они должны поместиться, если предположить, что строки кэша имеют ширину не менее 64 байт, потому что мы выровняли массив по 64-байтовой границе с указателем `alignas` C++11.)

Последующие итерации будут считывать элементы из копии `g_values`, которая находится в кэше L1, а не из основной оперативной памяти.

Во время каждой итерации `g_sum` обновляется. Компилятор может оптимизировать это действие, сохраняя сумму в регистре до конца цикла. Но независимо от того, выполняется ли эта оптимизация, мы знаем, что переменная `g_sum` в какой-то момент во время выполнения данной функции будет записана в память. Когда это произойдет, ЦП снова будет записывать в *копию* `g_sum`, которая существует в кэше L1, а не напрямую в основное ОЗУ.

В конце концов основная копия `g_sum` должна быть обновлена. Аппаратная кэш-память делает это автоматически, инициируя операцию *обратной записи*, которая копирует строку кэша из L1 обратно в основное ОЗУ. Но обычно обратная запись происходит не сразу — она откладывается до тех пор, пока измененная переменная не будет считана снова[1].

Многоядерные протоколы когерентности кэша

В многоядерной машине кэширование памяти становится намного сложнее. Простую двухъядерную машину, в которой каждое ядро имеет собственный кэш L1, а два ядра совместно используют кэш L2 и большой объем основной оперативной памяти, иллюстрирует рис. 4.36. Чтобы максимально упростить последующее обсуждение, проигнорируем кэш L2 и будем считать его примерно эквивалентным основной оперативной памяти.

Предположим, что упрощенный пример «производитель — потребитель», приведенный в подразделе 4.9.3, работает на этом двухъядерном компьютере. Поток производителя работает на ядре 1, а поток потребителя — на ядре 2. Также предположим, что для целей этого обсуждения ни одна из инструкций в любом потоке не была переупорядочена:

```
int32_t g_data = 0;
int32_t g_ready = 0;

void ProducerThread() // запуск на ядре 1
{
    g_data = 42;
    // предполагаем, что команды не переупорядочиваются в этой строке
    g_ready = 1;
}

void ConsumerThread() // запуск на ядре 2
{
    while (!g_ready)
        PAUSE();
    // предполагаем, что в этой строке нет перестановки команд
    ASSERT(g_data == 42);
}
```

[1] Определенная аппаратная кэш-память позволяет делать немедленную запись в основное ОЗУ. Сейчас мы можем безопасно игнорировать эту возможность.

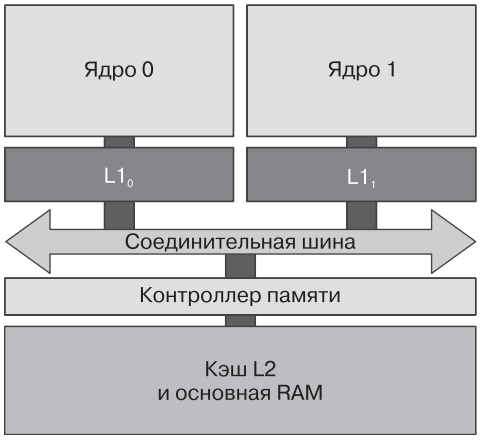


Рис. 4.36. Двухъядерный компьютер с локальной кэш-памятью L1 для каждого ядра, подключенный к контроллеру памяти через соединительную шину (ICB). Контроллер памяти реализует протокол когерентности кэша, такой как MESI, чтобы гарантировать, что оба ядра имеют согласованное представление о содержимом памяти в области когерентности кэша ЦП

Теперь рассмотрим, что происходит, когда производитель (работающий на ядре 1) пишет в `g_ready`. Для обеспечения эффективности эта запись приводит к обновлению кэша L1 в ядре 1, но некоторое время не запускает обратную запись в основное ОЗУ. Это означает, что в течение конечного промежутка времени после записи наиболее актуальное значение `g_ready` не существует нигде, кроме как в кэше L1 в ядре 1.

Допустим, потребитель (работающий на ядре 2) пытается прочитать `g_ready` через некоторое время после того, как производитель установил его в ядро 1. Как и ядро 1, ядро 2 предпочитает читать из кэша, когда это возможно, чтобы избежать высоких расходов на чтение основной оперативной памяти. Локальный кэш L1 в ядре 2 не содержит копию `g_ready`, а кэш L1 в ядре 1 — содержит. Поэтому в идеале ядро 2 хотело бы запросить у ядра 1 его копию, потому что это все равно было бы намного дешевле, чем получение данных из основной оперативной памяти. Также в данном случае это обеспечило бы явное преимущество, заключающееся в возврате наиболее актуального значения.

Протокол когерентности кэша — это механизм связи, который позволяет ядрам таким образом обмениваться данными между своими локальными кэшами L1. Большинство процессоров используют протокол MESI или MOESI.

Протокол MESI

По протоколу MESI каждая строка кэша может находиться в одном из четырех состояний.

- *«Изменена»* (modified). Эта строка кэша была изменена (в нее была произведена запись) локально.

- *«Исключительная»* (exclusive). Основной блок оперативной памяти, соответствующий данной строке кэша, существует только в кэше L1 этого ядра — ни у одного другого ядра его нет.
- *«Разделяемая»* (shared). Основной блок оперативной памяти, соответствующий этой строке кэша, существует в более чем одном ядре L1-кэша, и все ядра имеют его идентичную копию.
- *«Недействительная»* (invalid). Эта строка кэша больше не содержит допустимых данных — при следующем чтении необходимо получить строку либо из кэша L1 другого ядра, либо из основной оперативной памяти.

Протокол MOESI добавляет еще одно состояние — *«владеет»* (owned), которое позволяет ядрам обмениваться измененными данными, не записывая их обратно в основную оперативную память. Ради простоты мы сосредоточимся на MESI.

По протоколу MESI все кэши L1 всех ядер подключаются через специальную шину, называемую *соединительной шиной* (interconnect bus, ICB). В совокупности кэши L1, любые кэши более высокого уровня и основная RAM образуют так называемое *пространство когерентности кэша*. Протокол гарантирует, что все ядра имеют согласованное представление о данных в этой области.

Мы можем разобраться, как работает конечный автомат MESI, вернувшись к нашему примеру.

- Предположим, что ядро 1 (производитель) по какой-то причине сначала пытается прочитать текущее значение `g_ready`. Предполагая, что эта переменная еще не существует в кэше L1 какого-либо ядра, строка, содержащая ее, загружается в кэш L1 ядра 1. Строка кэша переводится в состояние «исключительная» (exclusive), это означает, что ни одно другое ядро не имеет этой строки.
- Теперь предположим, что ядро 2 (потребитель) пытается прочитать `g_ready`. Сообщение чтения отправляется через ICB. Ядро 1 содержит эту строку кэша, поэтому отвечает копией данных. В этот момент строка кэша переводится в состояние «разделяемая» (shared) на обоих ядрах, что указывает на то, что оба имеют одинаковую копию строки.
- Затем производитель на ядре 1 записывает 1 в `g_ready`. Это обновляет значение в кэше L1 в ядре 1 и переводит его копию строки в состояние «изменена» (modified). Сообщение о недействительности отправляется через ICB, в результате чего копия строки в ядре 2 переводится в состояние «недействительная» (invalid). Это указывает на то, что копия в ядре 2 строки, содержащей `g_ready`, больше не актуальна.
- В следующий раз, когда ядро 2 (потребитель) попытается прочитать `g_ready`, то обнаружит, что его локально кэшированная копия недействительна. Оно отправит сообщение *Read* через ICB и получит новую модифицированную строку из кэша L1 ядра 1. Это приведет к тому, что строки кэша обоих ядер снова будут переведены в состояние «разделяемая» (shared). Также это запускает *обратную запись* строки в основную оперативную память.

Полное обсуждение протокола MESI выходит за рамки книги, но этот пример должен дать вам хорошее представление о том, как он работает, позволяя нескольким ядрам обмениваться данными между их кэшами L1 при минимизации доступа к основной памяти.

Что с MESI может пойти не так

Если исходить из обсуждения протокола MESI в предыдущем разделе, то кажется, что проблема обмена данными между кэшами L1 в многоядерной машине была решена абсолютно надежным «водонепроницаемым» способом. Как же тогда могут произойти ошибки упорядочения памяти, на которые мы намекали?

На этот вопрос можно ответить одним словом: оптимизация. На большинстве аппаратных средств протокол MESI оптимизирован для минимизации задержки. Это означает, что некоторые операции не выполняются сразу, как только сообщения принимаются через ICB. Вместо этого для экономии времени они откладываются. Как и оптимизация компилятора и оптимизация выполнения ЦП, оптимизация MESI тщательно продумана, чтобы ее невозможно было обнаружить одним потоком. Но, как и следовало ожидать, параллельные программы страдают от этой условности.

Например, производитель, работающий на ядре 1, записывает `42` в `g_data`, а затем немедленно записывает 1 в `g_ready`. При определенных обстоятельствах оптимизация в протоколе MESI может привести к тому, что новое значение `g_ready` станет видимым для других ядер в области когерентности кэша *до того*, как станет видимым обновленное значение `g_data`. Это может произойти, например, если в ядре 1 уже есть строка кэша `g_ready` в локальном кэше L1, но еще нет строки `g_data`. Это означает, что потребитель (работает в ядре 2) потенциально может увидеть значение 1 для `g_ready`, прежде чем увидит значение `42` в `g_data`, что приведет к ошибке гонки данных.

Все это можно резюмировать следующим образом.

> Оптимизация в протоколе когерентности кэша может заставить другие ядра *видеть* две операции чтения и/или записи в порядке, противоположном тому, в котором они выполнялись на самом деле.

Ограждения памяти

Когда протокол когерентности кэша изменяет видимый порядок двух инструкций, мы говорим, что первая инструкция (в порядке программы) *опередила* вторую. Существует четыре способа, которыми одна инструкция может опередить другую.

1. Чтение может опередить другое чтение.
2. Чтение может опередить запись.
3. Запись может опередить другую запись.
4. Запись может опередить чтение.

Чтобы предотвратить влияние на память инструкций чтения или записи, передающих другие операции чтения и/или записи, современные процессоры предоставляют специальные инструкции машинного языка, известные как *ограждения* или *барьеры памяти*.

Теоретически процессор может предоставить индивидуальные инструкции ограничения, чтобы предотвратить каждый из четырех случаев. Например, ограждение *ReadRead* будет препятствовать тому, чтобы операции чтения опережали другие операции чтения, но не предотвратит ни один из прочих случаев. Кроме того, инструкция ограждения может быть однонаправленной или двунаправленной. Одностороннее ограждение гарантирует, что все операции чтения или записи, предшествующие ему в программе, будут выполнены до него, но не наоборот. Двунаправленное ограждение, в отличие от однонаправленного, предотвратит утечку результатов через ограждение в любом направлении. Таким образом, теоретически мы могли бы представить процессор, который обеспечивает 12 различных команд ограждения — двунаправленные, прямые и обратные варианты каждого из четырех основных типов ограждений.

К счастью, реальные процессоры обычно не предоставляют все 12 видов ограждений. Обычно ISA определяет несколько ограждающих инструкций, которые служат комбинациями теоретических типов ограждений.

Самый сильный вид ограждения называется *полным ограждением*. Оно гарантирует, что все операции чтения и записи, происходящие в программе до ограждения, никогда не будут появляться после него, а также что все операции чтения и записи, выполняемые после него, никогда не будут выполнены до. Другими словами, полное ограждение — это двусторонний барьер, который влияет как на чтение, так и на запись.

Реализация в оборудовании полного ограждения — очень дорогое удовольствие. Разработчики ЦП не любят заставлять программистов использовать дорогостоящие конструкции, когда того же самого можно добиться при помощи более дешевых. Поэтому большинство ЦП предоставляют разнообразные менее дорогие инструкции ограничения, которые обеспечивают более слабые гарантии, чем те, которые дает полное ограничение.

Все инструкции ограничения имеют два очень полезных побочных эффекта.

- Они служат барьерами компилятора.
- Они препятствуют внеочередному выполнению инструкций процессором через границы ограждения.

Это означает, что использование ограждения для предотвращения ошибок упорядочения памяти, вызванных протоколом когерентности кэша ЦП, предотвращает также переупорядочение команд. Так что атомарные инструкции и барьеры памяти — это все, что нам действительно нужно для написания надежных мьютексов, а также спин-блокировок и других алгоритмов *без блокировок*.

Семантика захвата и освобождения

Независимо от того, как выглядят конкретные инструкции ограничения в рамках конкретной ISA, мы можем рассуждать об их эффектах, думая о семантике, которую они предоставляют, иными словами, о гарантиях, которые эти инструкции обеспечивают относительно поведения операций чтения и записи в системе.

Семантика упорядочения памяти — это свойства инструкций чтения или записи, а не самих ограждений. Барьеры просто предоставляют программистам возможность гарантировать, что инструкция чтения или записи имеет определенную семантику упорядочения памяти. На самом деле есть только три семантики упорядочения памяти, о которых нужно помнить.

- *Семантика освобождения*. Гарантирует, что *запись* в общую память никогда не может опередить никакие другие операции чтения или записи, которые *предшествуют* ей в программе. Применение этой семантики к разделяемой записи мы называем *освобождением записи*. Семантика действует только в *прямом* направлении — она ничего не говорит о предотвращении появления операций с памятью, которые находятся после освобождения записи, но происходят перед ним.
- *Семантика захвата*. Гарантирует, что *чтение* из общей памяти никогда не опередят никакие другие операции чтения или записи, которые идут *после* него в программе. Применение этой семантики к разделяемому чтению мы называем *захватом чтения*. Семантика работает только в *обратном* направлении — она не делает ничего, чтобы предотвратить операции памяти, которые находятся в программе до чтения, но выполняются после него.
- *Полная семантика ограждения* (двунаправленная). Гарантирует, что все операции с памятью происходят в программе внутри границы, созданной инструкцией ограждения в коде. Никакие операции чтения или записи, которые находятся до нее в программе, не могут произойти после ограждения, и аналогично никакие операции чтения или записи, которые находятся после, не могут быть выполнены до ограждения.

Отдельные команды ограничения, отдаваемые каким-либо конкретным ISA, обычно предоставляют по меньшей мере одну из этих трех семантик упорядочения памяти. Детали того, как каждая инструкция ограждения на самом деле обеспечивает семантические гарантии, зависят от процессора, и по большей части программистам они неважны. Пока мы можем выразить концепцию освобождения записи, захвата чтения и полного ограничения в исходном коде, мы имеем возможность писать надежную спин-блокировку или создавать другие алгоритмы без блокировки.

Когда использовать семантику получения и освобождения

Освобождение записи чаще всего применяется в сценарии производителя, в котором поток выполняет две последовательные записи, например, в `g_data`, а затем

в g_ready, и мы должны убедиться, что все другие потоки увидят эти записи в соответствующем порядке. Мы можем навязать этот порядок, сделав вторую из инструкций записью освобождения. Чтобы реализовать это, инструкцию ограждения, которая предоставляет *семантику освобождения*, помещаем *перед* инструкцией освобождающей записи. Формально, когда ядро выполняет инструкцию ограничения с семантикой освобождения, оно ожидает, пока все предыдущие записи не будут полностью зафиксированы в памяти в области когерентности кэша, прежде чем выполнять вторую запись (освобождающую).

Захват чтения обычно используется в сценарии потребителя, в котором поток выполняет два последовательных чтения, причем второе является условным для первого (например, только если чтение g_data после чтения флага g_ready возвращает true). Мы применяем этот порядок, следя за тем, чтобы первое чтение было захватом чтения. Чтобы реализовать это, инструкцию ограждения, которая обеспечивает *семантику захвата*, помещаем *после* инструкции захвата чтения. Формально, когда ядро выполняет команду ограждения с семантикой захвата, оно ожидает, пока все записи с других ядер не будут полностью загружены в область когерентности кэша, прежде чем продолжить выполнение второго чтения. Это гарантирует, что второе чтение никогда не будет происходить до операции захвата чтения.

Вернемся к примеру с производителем — потребителем, полностью без блокировок использующему ограждения захвата и освобождения для наложения нужной семантики упорядочения памяти:

```
int32_t g_data = 0;
int32_t g_ready = 0;

void ProducerThread() // запуск на ядре 1
{
    g_data = 42;
    // превращаем запись в g_ready в запись-освобождение,
    // поместив ограждение перед ним
    RELEASE_FENCE();
    g_ready = 1;
}

void ConsumerThread() // запуск на ядре 2
{
    // преобразуем чтение g_ready в чтение-захват
    // путем установки ограждения захвата после него
    while (!g_ready)
        PAUSE();
    ACQUIRE_FENCE();

    // теперь можем безопасно читать g_data...
    ASSERT(g_data == 42);
}
```

Превосходное подробное описание того, почему требуются ограждения захвата и освобождения для использования протокола когерентности кэша MESI, вы найдете на www.swedishcoding.com/2017/11/10/multi-core-programming-and-cache-coherency/.

Модели памяти процессора

В подразделе 4.9.4 мы упоминали, что некоторые процессоры по умолчанию обеспечивают более строгую семантику порядка памяти, чем другие. На процессоре с сильной семантикой памяти инструкции чтения и/или записи по умолчанию ведут себя как определенный тип ограничения, при этом программисту не нужно явно указывать инструкцию ограничения. Например, DEC Alpha имеет общеизвестно слабую семантику по умолчанию, что требует тщательности практически в любой ситуации. На другом конце спектра процессор Intel x86, который по умолчанию имеет довольно сильную семантику упорядочения памяти. Хорошее описание слабого и сильного упорядочения памяти есть на preshing.com/20120930/weak-vs-strong-memory-models/.

Инструкции ограждения на реальных процессорах

Теперь, когда мы понимаем теорию семантики упорядочения памяти, очень кратко рассмотрим инструкции ограничения памяти на некоторых реальных процессорах.

В Intel x86 ISA определены три команды ограничения: `sfence` предоставляет семантику освобождения, `lfence` обеспечивает семантику захвата, а `mfence` выступает в качестве полного ограничения. Некоторым инструкциям x86 может предшествовать модификатор `lock`, чтобы заставить их вести себя атомарно и обеспечить ограничение памяти до выполнения инструкции. ISA x86 строго упорядочен по умолчанию. Это означает, что ограждения фактически не требуются во многих случаях, когда они бывают нужны в процессорах со слабой семантикой упорядочения по умолчанию. Но в некоторых ситуациях инструкции ограничения *требуются*. Для примера посмотрите пост *Who ordered memory fences on an x86?*, написанный Бартошем Милевски (bit.ly/2HuXpfo).

Набор инструкций PowerPC довольно слабо упорядочен, поэтому для обеспечения правильной семантики обычно требуются явные инструкции ограничения. PowerPC отличает чтение и запись в память от чтения и записи на устройства ввода-вывода и, следовательно, предлагает множество инструкций ограничения, которые различаются главным образом тем, как обрабатывают память по сравнению с вводом-выводом. Полное ограждение на PowerPC обеспечивается инструкцией `sync`, но есть также облегченное ограждение `lwsync`, ограждение для операций ввода-вывода `eieio` (обеспечивает выполнение ввода-вывода по порядку) и даже чистая команда переупорядочения барьера `isync`, которая не обеспечивает никакой семантики упорядочения памяти. Больше почитать об инструкциях ограничения PowerPC можно здесь: www.ibm.com/developerworks/systems/article/powerpc.html.

ARM ISA предоставляет чистый барьер переупорядочения команд `isb`, команды полного ограничения памяти `dmb` и `dsb`, одностороннюю инструкцию чтения — захвата `ldar` и одностороннюю инструкцию записи — освобождения `stlr`. Интересно, что семантика захвата и освобождения ISA находится в самих инструкциях чтения и записи, а не в виде отдельных команд ограничения. Для получения дополнительной информации см. веб-страницу infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CJAIAJFI.html.

4.9.5. Атомарные переменные

Непосредственное использование атомарных инструкций и ограждений памяти может быть утомительно и чревато ошибками, не говоря уже о том, что оно полностью непереносимо. К счастью, в C++11 шаблон класса `std: atomic<T>` позволяет преобразовать практически любой тип данных в *атомарную переменную*. (Специализированный класс `std::atomic_flag` инкапсулирует атомарную логическую переменную.) В дополнение к атомарности семейство шаблонов `std::atomic` по умолчанию предоставляет своим переменным семантику упорядочения памяти «полное ограждение» (хотя при желании может быть указана более слабая семантика). Это позволяет писать код без блокировок, не беспокоясь ни об одной из трех причин ошибок в гонке данных.

Используя эти возможности, пример с производителем — потребителем можно реализовать следующим образом:

```
std::atomic<float> g_data;
std::atomic_flag g_ready = false;

void ProducerThread()
{
    // получить некоторые данные
    g_data = 42;

    // информируем потребителя
    g_ready = true;
}

void ConsumerThread()
{
    // ждем, пока данные будут готовы
    while (!g_ready)
        PAUSE();

    // потребляем данные
    ASSERT(g_data == 42);
}
```

Обратите внимание на то, что этот код выглядит почти идентично ошибочному коду, который мы впервые представили, когда говорили об условиях гонки в подразделе 4.5.3. Просто поместив наши переменные в `std::atomic`, мы преобразовали параллельную программу, склонную к ошибкам гонки данных, в программу, в которой нет гонок.

Внутри реализации `std::atomic<T>` и `std::atomic_flag`, конечно, сложны. Стандартная библиотека C++ должна быть переносимой, поэтому в ее реализации используются любые атомарные и барьерные инструкции машинного языка, доступные на целевой платформе. Более того, шаблон `std::atomic<T>` может обернуть любой мыслимый тип, но, конечно, процессоры не предоставляют элементарных инструкций для манипулирования объектами данных произвольного размера.

Таким образом, шаблону `std::atomic<T>` необходимо указать размер: когда шаблон применяется к 32- или 64-битному типу, он может быть реализован без блокировок, напрямую с помощью инструкций атомарного машинного языка. Но когда он применяется к более крупным типам, необходимо использовать блокировки мьютекса для обеспечения правильного атомарного поведения. (Можете вызвать `is_lock_free()` для любой атомарной переменной, чтобы узнать, действительно ли ее реализация не блокируется на целевом оборудовании.)

Упорядочение памяти C++

По умолчанию атомарные переменные C++ используют полные барьеры памяти, чтобы гарантировать, что они будут работать правильно во всех возможных случаях. Однако можно ослабить эти гарантии, передав семантику порядка памяти (необязательный аргумент типа `std::memory_order`) функциям, которые манипулируют атомарными переменными. Документация относительно `std::memory_order` довольно запутанная, поэтому рассмотрим ее. Вот возможные настройки порядка памяти и их значение.

- *«Ослабленный»* (relaxed). Атомарная операция выполняется с ослабленной семантикой порядка памяти и гарантирует только атомарность. Никакие барьер или ограждение не используются.
- *«Потребление»* (consume). Чтение, выполняемое с применением семантики потребления, гарантирует, что никакие другие чтение или запись *в том же потоке* не могут быть переупорядочены и выполнены перед этим чтением. Другими словами, эта семантика служит только для предотвращения переупорядочения команд в *оптимизации* и *внеочередном выполнении* — она ничего не делает для обеспечения конкретной семантики упорядочения памяти в области когерентности кэша. Обычно она реализуется с барьером переупорядочения команд, таким как команда PowerPC `isync`.
- *«Освобождение»* (release). Запись, выполняемая с помощью семантики *освобождения*, гарантирует, что никакие другие операции чтения или записи в этом потоке не могут быть переупорядочены и выполнены после нее. Запись гарантированно будет видна другим потокам, читающим тот же адрес. Для этого используется *ограничение освобождения* в области когерентности кэша ЦП.
- *«Захват»* (acquire). Чтение, выполняемое с помощью семантики *захвата*, гарантирует применение семантики, а также то, что запись по тому же адресу другими потоками будет видна этому потоку. Это делается через ограничение *захвата* в области когерентности кэша ЦП.
- *«Захват»/«Освобождение»*. Эта семантика (по умолчанию) самая безопасная, потому что применяет *полное* ограничение памяти.

Важно понимать, что использование спецификатора упорядочения памяти не гарантирует, что конкретная семантика будет применяться на каждой платформе. Оно лишь гарантирует, что семантика будет *по крайней мере* такой сильной — на некотором

целевом оборудовании может быть применена более сильная семантика. Например, на Intel x86 ослабленный порядок памяти невозможен, потому что семантика упорядочения памяти процессора по умолчанию является довольно сильной. На процессоре Intel любой запрос на ослабленную операцию чтения в конечном итоге приобретет семантику *захвата*.

Использование этих спецификаторов порядка в памяти требует перехода от перегруженных операторов присваивания и приведения `std::atomic` к явным вызовам `store()` и `load()`. Вот еще раз наш пример с производителем — потребителем, на этот раз с применением спецификаторов `std::memory_order` для обеспечения ограждения освобождения и захвата:

```
std::atomic<float> g_data;
std::atomic<bool> g_ready = false;

void ProducerThread()
{
    // получить некоторые данные
    g_data.store(42, std::memory_order_relaxed);

    // информируем потребителя
    g_ready.store(true, std::memory_order_release);
}

void ConsumerThread()
{
    // ждем, пока данные не будут готовы
    while (!g_ready.load(std::memory_order_acquire))
        PAUSE();

    // потребляем данные
    ASSERT(g_data.load(std::memory_order_relaxed) == 42);
}
```

Важно помнить правило 80/20 при использовании семантики упорядоченной памяти, подобной данной. В этой семантике легко ошибиться, поэтому вам, вероятно, следует задействовать упорядочение памяти не по умолчанию с помощью `std::atomic`, только когда вы можете доказать с помощью профилирования, что повышение производительности действительно необходимо и что изменение кода с применением явной семантики упорядочения памяти в самом деле дает ожидаемые результаты!

Полное обсуждение того, как применять семантику упорядочения памяти в C++11, выходит за рамки книги, но вы можете узнать об этом больше, найдя в Интернете статью Майкла Чайновета *Implementing Scalable AtomicLocks for Multi-Core Intel® EM64T and IA32 Architectures*. Еще есть обсуждение на форуме, где также предлагаются интересные идеи и демонстрируется, насколько сложным может стать этот вид программирования: groups.google.com/forum/#!topic/boost-developers-archive/Qlrat5ASrnM.

4.9.6. Параллельность в интерпретируемых языках программирования

До сих пор мы обсуждали параллелизм только в контексте компилируемых языков, таких как C и C++, и языка ассемблера. Эти языки компилируются или собираются в необработанный машинный код, который выполняется непосредственно процессором. Сами атомарные операции и блокировки должны быть реализованы посредством специальных инструкций машинного языка, которые обеспечивают атомарные операции и кэшируют когерентные барьеры памяти при помощи ядра (оно гарантирует, что потоки переводятся в спящий режим и соответственно активируются) и компилятора (он учитывает ограждающие инструкции при оптимизации кода).

Немного иная ситуация для интерпретируемых языков программирования, таких как Java и C#. Программы, написанные на них, выполняются в контексте *виртуальной машины* (VM): Java-программы работают внутри виртуальной машины Java (JVM), а программы на C# — в контексте Common Language Runtime (CLR). Виртуальная машина — это, по сути, программная эмуляция процессора, считывающая инструкции в байт-коде и выполняющая их. VM также действует как ядро эмулируемой операционной системы: она предоставляет собственное понятие «потоков», состоящих из байт-кодов инструкций, и берет все детали управления этими потоками на себя. Поскольку работа виртуальной машины полностью реализована в программном обеспечении, интерпретируемый язык, такой как Java или C#, может предоставить мощные средства параллелизма, не настолько ограниченные аппаратным обеспечением, как в компилируемом языке, таком как C или C++.

Одним из примеров этого принципа в действии является квалификатор `volatile`. В разделе 4.6 мы говорили, что в C/C++ переменная типа `volatile` не является атомарной. Однако в Java и C# спецификатор типа `volatile` гарантирует атомарность. Операции с переменными данного типа в этих языках не могут быть оптимизированы, а также прерваны другим потоком. Кроме того, все чтения переменной типа `volatile` в Java и C# эффективно выполняются непосредственно из основной памяти, а не из кэша. Аналогично все записи эффективно выполняются в основную оперативную память, а не в кэш. Все это может быть гарантировано отчасти потому, что виртуальная машина полностью контролирует выполнение байт-кодов команд потоков, составляющих каждое приложение.

Полное обсуждение возможностей атомарности и синхронизации потоков, предоставляемых интерпретируемыми языками, такими как C# и Java, выходит далеко за рамки этой книги. Но теперь, когда вы хорошо понимаете принципы атомарности на самых низких уровнях, разобраться в возможностях языков более высокого уровня должно быть несложно. Начните со следующего:

- C# — на docs.microsoft.com наберите в строке поиска «параллельная обработка и параллелизм в .NET Framework»;
- Java — docs.oracle.com/javase/tutorial/essential/concurrency/.

4.9.7. Спин-блокировки

При обсуждении атомарных инструкций машинного языка в подразделе 4.9.2 я представил фрагменты кода, иллюстрирующие, как можно реализовать спин-блокировку с использованием каждой из этих инструкций. Но из-за переупорядочения команд и семантики упорядочения памяти эти примеры не были на 100 % правильными. В данном подразделе я приведу реальный пример спин-блокировки, а затем мы рассмотрим несколько полезных вариаций.

Базовая спин-блокировка

Спин-блокировка может быть реализована с помощью `std::atomic_flag`, либо обернута в класс C++, либо доступна через простой функциональный API. Спин-блокировка создается с помощью инструкции TAS, которая атомарно устанавливает флаг в значение `true`, и выполняется в цикле `while`, пока TAS не завершится успехом. Она снимается атомарной записью значения `false` во флаг.

При получении спин-блокировки важно использовать семантику упорядочения памяти для *чтения — захвата*, чтобы прочитать текущее содержимое блокировки как часть операции TAS. Это ограждение защищает от редкого сценария, в котором блокировка считается снятой, хотя на самом деле какой-то другой поток еще не *полностью* вышел из своего критического раздела. В C++11 это может быть достигнуто передачей `std::memory_order_acquire` в вызов функции `test_and_set()`. На чистом языке ассемблера мы разместили бы инструкцию ограждения после инструкции TAS.

При снятии спин-блокировки важно применять также семантику *записи — освобождения*, чтобы гарантировать, что все записи, выполненные *после* вызова `Unlock()`, не отслеживаются другими потоками, как происходило *до* снятия блокировки.

Вот полная реализация спин-блокировки на основе TAS с использованием правильной и минимальной семантики упорядочения памяти:

```
class SpinLock
{
    std::atomic_flag m_atomic;

public:
    SpinLock() : m_atomic(false) { }

    bool TryAcquire()
    {
        // используем ограничения захвата, чтобы гарантировать, что все
        // последующие операции чтения этого потока будут валидными
        bool alreadyLocked = m_atomic.test_and_set(
                                std::memory_order_acquire);

        return !alreadyLocked;
    }
```

```
    void Acquire()
    {
        // повторять до успешного выполнения
        while (!TryAcquire())
        {
            // уменьшает энергопотребление на процессорах Intel
            // (можно заменить на std::this_thread::yield()
            // на платформах, которые не поддерживают паузу процессора,
            // если конкуренция потоков ожидается высокой)
            PAUSE();
        }
    }

    void Release()
    {
        // использовать семантику освобождения, гарантируя, что все предыдущие
        // операции записи были полностью зафиксированы до снятия блокировки
        m_atomic.clear(std::memory_order_release);
    }
};
```

Блокировка области

Часто освободить мьютекс или спин-блокировку неудобно, к тому же это чревато ошибками, особенно когда функция, использующая блокировку, имеет несколько возможных возвратов. В C++ мы можем задействовать простой класс-оболочку, называемый *блокировкой области*, чтобы гарантировать, что блокировка автоматически снимается при выходе из определенной области. Она работает, просто получая блокировку в конструкторе и освобождая ее в деструкторе:

```
template<class LOCK>
class ScopedLock
{
    typedef LOCK lock_t;
    lock_t* m_pLock;

public:
    explicit ScopedLock(lock_t& lock) : m_pLock(&lock)
    {
        m_pLock->Acquire();
    }

    ~ScopedLock()
    {
        m_pLock->Release();
    }
};
```

Класс блокировки области действия можно использовать с любой спин-блокировкой или мьютексом, который имеет соответствующий интерфейс, то есть

с любым классом блокировки, который поддерживает функции `Acquire()` и `Release()`. Вот как это работает:

```
SpinLock g_lock;

int ThreadSafeFunction()
{
    // блокировка области действия работает как сборщик
    // мусора, потому что убирает за нами!
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // выполнить некоторую работу...

    if (SomethingWentWrong())
    {
        // здесь будет снята блокировка
        return -1;
    }

    // так что еще немного работы...

    // здесь также будет снята блокировка
    return 0;
}
```

Реентерабельные блокировки

«Ванильная» спин-блокировка приведет к взаимоблокировке потока, если он когда-либо попытается захватить блокировку, которую уже удерживает. Это может происходить всякий раз, когда две или более *потоково-ориентированные* функции пытаются взаимно вызвать друг друга из одного и того же потока. Например, даны две функции:

```
SpinLock g_lock;

void A()
{
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // выполнить некоторую работу...
}

void B()
{
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // выполнить некоторую работу...

    // сделать вызов A(), удерживая блокировку
    A(); // тупик!

    // так что еще немного работы...
}
```

Мы ослабим ограничение повторного входа, если сможем организовать для класса спин-блокировки кэширование идентификатора потока, который его заблокировал. Таким образом, блокировка может знать разницу между потоком, пытающимся захватить собственную блокировку (которую мы хотим разрешить), и потоком, пытающимся захватить блокировку, которая уже удерживается другим потоком (что должно перевести вызывающую сторону в режим ожидания). Чтобы убедиться, что вызовы `Acquire()` и `Release()` выполняются в соответствующих парах, мы также должны добавить счетчик в класс. Вот функциональная реализация, использующая соответствующее ограждение памяти:

```
class ReentrantLock32
{
    std::atomic<std::size_t>  m_atomic;
    std::int32_t              m_refCount;

public:
    ReentrantLock32() : m_atomic(0), m_refCount(0) { }

    void Acquire()
    {
        std::hash<std::thread::id> hasher;
s        td::size_t tid = hasher(std::this_thread::get_id());

        // если этот поток еще не удерживает блокировку...
        if (m_atomic.load(std::memory_order_relaxed) != tid)
        {
            // ...выполнять, пока мы ее не освободим
            std::size_t unlockValue = 0;
            while (!m_atomic.compare_exchange_weak(
                unlockValue,
                tid,
                std::memory_order_relaxed, // ограждение ниже!
                std::memory_order_relaxed))
            {
                unlockValue = 0;
                PAUSE();
            }

        }
        // увеличить счетчик ссылок, чтобы можно было убедиться, что
        // методы Acquire() и Release() вызываются попарно
        ++m_refCount;
        // используем ограничения захвата, чтобы гарантировать, что все
        // последующие операции чтения этого потока будут валидными
        std::atomic_thread_fence(std::memory_order_acquire);
    }

    void Release()
    {
        // использовать семантику освобождения, чтобы гарантировать, что все
        // предыдущие операции записи были полностью зафиксированы перед
        // разблокировкой
        std::atomic_thread_fence(std::memory_order_release);
```

```
        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());
        std::size_t actual = m_atomic.load(std::memory_order_relaxed);
        assert(actual == tid);

        --m_refCount;
        if (m_refCount == 0)
        {
            // снять блокировку, которая безопасна, поскольку мы владеем ею
            m_atomic.store(0, std::memory_order_relaxed);
        }
    }

    bool TryAcquire()
    {
        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());

        bool acquired = false;

        if (m_atomic.load(std::memory_order_relaxed) == tid)
        {
            acquired = true;
        }
        else
        {
            std::size_t unlockValue = 0;
            acquired = m_atomic.compare_exchange_strong(
                unlockValue,
                tid,
                std::memory_order_relaxed, // Ограждение ниже!
                std::memory_order_relaxed);
        }
        if (acquired)
        {
            ++m_refCount;
        std::atomic_thread_fence(
            std::memory_order_acquire);
        }
        return acquired;
    }
};
```

Блокировки чтения — записи

В системе, в которой несколько потоков могут читать и записывать общий объект данных, мы можем контролировать доступ к объекту с помощью мьютекса или спин-блокировки. Однако нескольким потокам следует разрешить только одновременное *чтение* общего объекта. Лишь когда общий объект *изменяется*, мы должны обеспечить взаимную исключительность. Здесь хотелось бы применить своего рода блокировку, которая любому количеству читателей позволяет одновременно удер-

живать ее. Всякий раз, когда поток-писатель пытается получить блокировку, он должен дождаться завершения работы всех читателей, а затем получить блокировку в специальном эксклюзивном режиме, который препятствует получению доступа другими читателями или писателям до тех пор, пока он не завершит свое изменение общего объекта. Этот вид блокировки называется *блокировкой чтения — записи* (а также *блокировкой общего доступа* или *принудительной блокировкой*).

Мы можем реализовать блокировку чтения — записи способом, аналогичным тому, каким выполняли реентерабельную блокировку. Однако вместо того, чтобы хранить идентификатор потока в атомарной переменной, мы будем хранить счетчик ссылок, указывающий, сколько читателей в настоящее время удерживают блокировку. Каждый раз, когда читатель получает блокировку, количество увеличивается, когда снимает блокировку — уменьшается.

А как мы можем обеспечить эксклюзивный режим блокировки для писателя? Для этого нужно лишь зарезервировать одно (очень высокое) значение счетчика ссылок и использовать его для обозначения того, что писатель в настоящее время удерживает блокировку. Если счетчик ссылок представляет собой 32-разрядное целое число без знака, значение 0xFFFFFFFFU вполне может подойти в качестве зарезервированного значения. Или, еще проще, мы можем зарезервировать старший значащий бит. Это означает, что число ссылок от 0 до 0x7FFFFFFFU представляет блокировки читателей, а зарезервированное значение 0x80000000U — блокировку записи (без других допустимых значений).

Блокировка чтения — записи страдает от проблемы голодания: писатель, который слишком долго удерживает блокировку, может заставить всех читателей голодать, а большое число читателей может заставить голодать писателей. *Последовательная блокировка* — это одна из возможных альтернатив, которая решает проблему голодания (подробнее см. en.wikipedia.org/wiki/Seqlock). Посетите lwn.net/Articles/262464, там представлено описание еще одной интересной техники блокировки, используемой в ядре Linux, при которой поддерживаются несколько одновременных программ чтения и записи, называемых *read-copy-update* (RCU).

Предлагаю вам создать программу чтения — записи самостоятельно в качестве упражнения! Но если вы хотите сравнить заметки, можете найти полнофункциональную реализацию на сайте этой книги (www.gameenginebook.com).

Утверждения о необязательности блокировки

Независимо от того, как вы используете блокировки, они стоят дорого. Мьютексы дороги даже при отсутствии параллелизма. В сценарии с низким уровнем конкуренции спин-блокировки относительно дешевы, но все же не равны нулю в любой части программного обеспечения.

Часто бывает, что программист априори знает, что блокировка не требуется. Например, в игровом движке каждая итерация игрового цикла обычно выполняется на разных этапах. Если доступ к общей структуре данных получают единственный поток в начале кадра и другой единственный поток позже в кадре, тогда блокировка фактически не нужна. Да, теоретически эти два потока могут

перекрываться, и если так произойдет, блокировка определенно понадобится. Но на практике мы, учитывая, как построен игровой цикл, можем точно знать, что такое никогда не произойдет.

В такой ситуации у нас есть несколько вариантов. Мы можем поставить блокировки на всякий случай. Так что, если кто-то изменит порядок, в котором все происходит в рамках одного кадра, и заставит эти потоки перекрываться, мы будем застрахованы. Другой вариант — просто игнорировать возможность перекрытия и ничего не блокировать.

Есть третий вариант, который я считаю более привлекательным в таком сценарии: мы можем использовать *утверждение*, что блокировка не требуется. У данного способа два преимущества. Во-первых, это очень дешево, а утверждения можно удалить перед выпуском игры. Во-вторых, это позволяет автоматически обнаружить проблемы, если предположение о перекрытии потоков оказывается неверным или если после рефакторинга кода ситуация меняется. Для этого вида утверждений не существует стандартизированного имени, поэтому в книге будем называть их утверждениями *о необязательности блокировки*.

Итак, как мы можем обнаружить, что блокировка необходима? Одним из способов может быть применение атомарной логической переменной с полным ограждением памяти и использование ее как мьютекса. За исключением того, что вместо фактического получения блокировки мьютекса мы просто утверждали бы, что логическое значение ложно, и затем устанавливали его в значение истины атомарно. Вместо того чтобы снимать блокировку, мы утверждаем, что логический тип равен `true`, а затем атомарно устанавливаем для него значение `false`. Этот подход будет работать, но он окажется таким же дорогим, как и спин-блокировка. Мы можем добиться большего.

Хитрость заключается в том, чтобы понять, что мы заботимся только об *обнаружении* наложений между критическими операциями над общим объектом. И обнаружение не должно быть на 100 % надежным. Вероятность попадания в 90 % — просто отлично. Если две критические операции когда-либо перекрываются, вероятны случаи, когда мы не сможем обнаружить этого. Но если ваша игра каждый день по нескольку раз запускается командой из 100 и более разработчиков и отделом контроля качества, состоящим из 10–20 человек или даже больше, можете быть уверены, что кто-то обнаружит проблему, если она существует.

Таким образом, вместо *атомарного* логического значения мы используем изменяемый логический тип. Как уже говорилось, ключевое слово `volatile` мало помогает в предотвращении ошибок гонки данных. Но оно гарантирует, что чтение и запись логического значения не будут оптимизированы, и этого нам достаточно. Мы получим хороший уровень обнаружения, и тест будет очень дешевым:

```
class UnnecessaryLock
{
    volatile bool     m_locked;

public:
    void Acquire()
```

```
    {
        // утверждаем, что никто не удерживает блокировку
        assert(!m_locked);

        // теперь блокировка (чтобы мы могли обнаружить перекрытие
        // критических операций, если оно происходит)
        m_locked = true;
    }
    void Release()
    {
        // подтверждаем правильное использование (этот Release()
        // вызывается только после Acquire())
        assert(m_locked);

        // разблокировать
        m_locked = false;
    }
};

#if ASSERTIONS_ENABLED
#define BEGIN_ASSERT_LOCK_NOT_NECESSARY(L) (L).Acquire()
#define END_ASSERT_LOCK_NOT_NECESSARY(L)   (L).Release()
#else
#define BEGIN_ASSERT_LOCK_NOT_NECESSARY(L)
#define END_ASSERT_LOCK_NOT_NECESSARY(L)
#endif

// Пример использования:

UnnecessaryLock g_lock;
void EveryCriticalOperation()
{
    BEGIN_ASSERT_LOCK_NOT_NECESSARY(g_lock);

    printf("perform critical op...\n");

    END_ASSERT_LOCK_NOT_NECESSARY(g_lock);
}
```

Мы также можем обернуть блокировки в сборщик (см. подраздел 3.1.1), например, так:

```
class UnnecessaryLockJanitor
{
    UnnecessaryLock* m_pLock;
public:
    explicit
    UnnecessaryLockJanitor(UnnecessaryLock& lock)
        : m_pLock(&lock) { m_pLock->Acquire(); }
    ~UnnecessaryLockJanitor() { m_pLock->Release(); }
};
```

```
#if ASSERTIONS_ENABLED
#define ASSERT_LOCK_NOT_NECESSARY(J,L) \
    UnnecessaryLockJanitor J(L)
#else
#define ASSERT_LOCK_NOT_NECESSARY(J,L)
#endif

// Пример использования:

UnnecessaryLock g_lock;

void EveryCriticalOperation()
{
    ASSERT_LOCK_NOT_NECESSARY(janitor, g_lock);

    printf("perform critical op...\n");
}
```

Мы реализовали данный прием в Naughty Dog, и код успешно обнаружил множество случаев критических операций, перекрывающихся, когда программисты полагали, что они никогда не смогут этого сделать. Так что этот маленький драгоценный камень проверен на практике.

4.9.8. Транзакции без блокировок

Предполагалось, что этот раздел будет посвящен программированию без блокировок, но пока мы все время писали спин-блокировки! Возможно, это нелогично, но написание спин-блокировки является примером программирования без блокировки с точки зрения реализации самой спин-блокировки. Мы также узнали много нового об атомарных инструкциях, барьерах компилятора и ограждениях памяти. Так что провели время с пользой. Однако мы не исследовали принципы программирования без блокировок как таковые, для их изучения будет целесообразно рассмотреть примеры без спин-блокировки. Тема безблокировочных и неблокирующих алгоритмов огромна. Она заслуживает отдельной книги, поэтому я не буду пытаться подробно освещать ее. Но можем по крайней мере понять, как обычно работают подходы без блокировки.

Цель программирования без блокировок состоит в том, чтобы избежать блокировок, которые либо переведут поток в спящий режим, либо вызовут его зацикливание на стадии ожидания внутри спин-блокировки. Чтобы выполнить критическую операцию без блокировки, мы должны рассматривать каждую такую операцию как *транзакцию*, которая может завершиться либо успешно, либо неудачно. Если она не удается, то просто повторяется до тех пор, пока не достигнет успеха.

Чтобы реализовать любую транзакцию, какой бы сложной она ни была, мы выполняем большую часть работы *локально*, то есть используем данные, которые видны только текущему потоку, а не работаем непосредственно с общими данными. Когда все наши утки выстроены в ряд и транзакция готова к фиксации, мы выполняем *одну атомарную инструкцию*, такую как CAS или LL/SC. Если эта

инструкция выполнена успешно — мы удачно опубликовали транзакцию в глобальном масштабе, — она становится постоянной частью общей структуры данных, над которой мы работаем. Но если она терпит неудачу, это означает, что какой-то *другой* поток пытался завершить транзакцию в то же время, что и мы.

Этот подход с ошибкой и повтором работает, потому что всякий раз, когда одному потоку не удается зафиксировать транзакцию, другой поток добивается успеха. В результате один поток в системе всегда продвигается вперед (возможно, не наш). И это определение разработки *без блокировок*.

4.9.9. Связанный список без блокировки

В качестве примера рассмотрим простой список без блокировок с односвязными ссылками. Единственная операция, которую мы будем поддерживать, — это `push_front()`.

Чтобы подготовить транзакцию, выделяем в памяти новый `Node` (Узел) и заполняем его данными. А также устанавливаем указатель `next`, указывающий на тот узел, который в данный момент находится в начале связанного списка. Теперь транзакция готова к атомарной фиксации.

Сама фиксация состоит из вызова `compare_exchange_weak()` на указателе начала списка, который мы объявили атомарным указателем на `Node`. Если вызов выполнен успешно, мы вставили новый узел в начало связанного списка, и все готово. Но если он не удается, следует повторить попытку. Это действие включает в себя повторную инициализацию указателя на то, что теперь потенциально является *новым головным узлом* (предположительно, вставленным другим потоком — возможно, именно поэтому мы потерпели неудачу). Этот двухэтапный процесс показан на рис. 4.37.

В приведенном далее коде вы не увидите явной повторной инициализации указателя на следующий узел. Это потому, что `compare_exchange_weak()` выполняет за нас этап повторной инициализации. (Как удобно!) Вот как будет выглядеть код:

```
template< class T >
class SList
{
    struct Node
    {
        T      m_data;
        Node*  m_pNext;
    };
    std::atomic< Node* > m_head { nullptr };

public:
    void push_front(T data)
    {
        // подготовить транзакцию локально
        auto pNode = new Node();
        pNode->m_data = data;
        pNode->m_pNext = m_head;
```

```
        // совершаем транзакцию атомарно
        // (повторная попытка, пока не получится)
        while (!m_head.compare_exchange_weak(
                        pNode->m_pNext, pNode))
        { }
    }
};
```

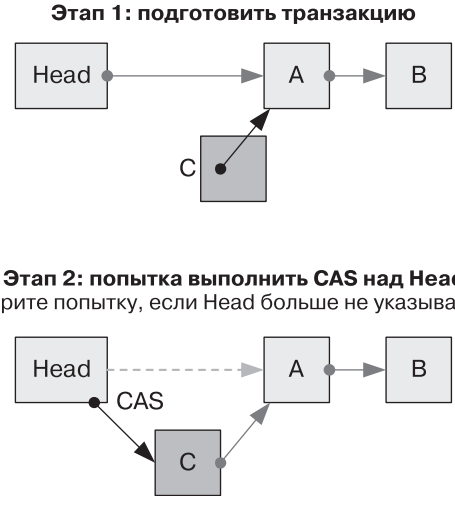


Рис. 4.37. Реализация вставки в начало односвязного списка без блокировок. *Вверху*: транзакция подготавливается путем установки указателя next нового узла, указывающего на текущий заголовок списка. *Внизу*: транзакция фиксируется с помощью атомарной операции CAS, чтобы заменить указатель заголовка указателем на новый узел. Если операция CAS дает сбой, мы возвращаемся к началу и повторяем попытку, пока все не получится

4.9.10. Дополнительная литература по программированию без блокировки

Параллелизм — это обширная и глубокая тема, и в этой главе мы лишь коснулись ее поверхности. Но цель этой книги — просто дать вам базовую информацию, которая послужит отправной точкой для дальнейшего обучения.

- Реализация односвязного списка без блокировок подробно обсуждается в выступлении Херба Саттера на CppCon 2014, откуда был взят приведенный ранее пример. Выступление доступно на YouTube в двух частях:
  - www.youtube.com/watch?v=c1gO9aB9nbs;
  - www.youtube.com/watch?v=CmxkPChOcvw.
- Лекция Джеффа Лэнгдейла из CMU дает отличный обзор: www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf.

- Обязательно ознакомьтесь с презентацией Сами аль-Бахры. Это четкий и очень понятный обзор почти всех тем, относящихся к параллельному программированию: concurrencykit.org/Presentations/lockfree_introduction/#/.
- Отличное рассуждение Майка Актона о параллельном мышлении прочитать обязательно: cellperformance.beyond3d.com/articles/public/concurrency_rabit_hole.pdf.
- Две онлайн-книги являются прекрасными источниками информации о параллельном программировании: greenteapress.com/semaphores/LittleBookOfSemaphores.pdf и www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2011.01.02a.pdf.
- Хорошие статьи о программировании без блокировок и о том, как работают атомарность, барьеры и ограждения, можно найти в блоге Джеффа Прешинга preshing.com/20120612/an-introduction-to-lock-free-programming.
- На этом сайте представлена отличная информация о барьерах памяти в Linux: www.mjmwired.net/kernel/Documentation/memory-barriers.txt#305.

4.10. SIMD/векторная обработка

В подразделе 4.1.4 была введена форма параллелизма, известная как *одиночный поток команд, множественный поток данных* (single instruction multiple data, SIMD). Это относится к тому, что большинство современных микропроцессоров способны выполнять математическую операцию с несколькими элементами данных *параллельно*, используя одну машинную инструкцию. В этом разделе мы подробно рассмотрим методы SIMD и завершим главу кратким обсуждением того, как SIMD и многопоточность объединяются в форму параллелизма, известную как *одиночный поток команд, множественные потоки* (single instruction multiple thread, SIMT), которая лежит в основе всех современных графических процессоров.

Intel впервые представила набор команд MMX[1] в линейке процессоров Pentium в 1994 году. Эти инструкции позволили выполнять вычисления SIMD для восьми 8-разрядных, четырех 16-разрядных или двух 32-разрядных целых чисел, упакованных в специальные 64-разрядные регистры MMX. Вслед за этим Intel выпустила различные версии расширенного набора инструкций, называемых *потоковыми расширениями SIMD*, или SSE (streaming SIMD extensions), первая версия которого появилась в процессоре Pentium III.

Набор команд SSE использует 128-битные регистры, которые могут содержать целочисленные данные или данные с плавающей точкой формата IEEE. Режим SSE, наиболее часто применяемый игровыми механизмами, называется *упакованным 32-разрядным режимом с плавающей точкой*. В этом режиме четыре 32-битных значения с плавающей точкой упаковываются в один 128-битный

[1] Официально MMX — это бессмысленная аббревиатура, зарегистрированная под торговой маркой Intel. Неофициально разработчики считают, что это означает «мультимедийные расширения» (multimedia extensions) или «матричные математические расширения» (matrix math extensions).

регистр. Таким образом, сложение и умножение могут выполняться параллельно на четырех парах чисел с плавающей точкой, принимая два 128-битных регистра в качестве входных данных. С тех пор Intel представила несколько обновлений набора инструкций SSE — SSE2, SSE3, SSSE3 и SSE4. В 2007 году AMD представила собственные варианты — XOP, FMA4 и CVT16.

В 2011 году Intel представила новый, более широкий файл регистров SIMD и сопровождающий его набор инструкций под названием *advanced vector extensions* (AVX). Регистры AVX имеют ширину 256 бит, что позволяет одной команде работать с парами до восьми 32-битных операндов с плавающей точкой параллельно. Набор инструкций AVX2 — это расширение AVX. Некоторые процессоры Intel теперь поддерживают AVX-512 — расширение для AVX, позволяющее упаковывать шестнадцать 32-разрядных операций с плавающей запятой в 512-разрядный регистр.

4.10.1. Набор инструкций SSE и его регистры

Набор инструкций SSE включает в себя широкий спектр операций со многими вариантами работы с элементами данных разного размера в регистрах SSE. Однако мы будем обсуждать относительно небольшое подмножество инструкций, которые имеют дело с упакованными 32-битными данными с плавающей точкой. Эти инструкции обозначены суффиксом `ps`, указывающим, что мы работаем с *упакованными* данными (*p* — packed) и что каждый элемент представляет собой число с плавающей точкой *одинарной* точности (*s* — single-precision). Однако большинство приводимых в дальнейшем примеров интуитивно распространяются на 256- и 512-битные режимы AVX (обзор AVX см. на странице software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions).

Регистры SSE называются XMM*i*, где *i* — целое число от 0 до 15 (например, XMM0, XMM1 и т. д.). В упакованном 32-разрядном режиме с плавающей точкой каждый 128-разрядный регистр XMM*i* содержит четыре 32-разрядных числа с плавающей точкой. В AVX регистры имеют ширину 256 бит и называются YMM*i*, в AVX-512 они имеют ширину 512 бит и называются ZMM*i*.

В этой главе мы часто будем ссылаться на отдельные числа с плавающей точкой в регистре SSE как $[x\ y\ z\ w]$, как если бы выполняли математические векторные или матричные операции в системе координат на бумаге (рис. 4.38). Обычно не имеет значения, что вы называете элементами регистра SSE, если согласны с тем, как *интерпретируете* каждый элемент. Наиболее общий подход заключается в представлении вектора SSE **r** как содержащего элементы $[r_0, r_1, r_2, r_3]$. Это соглашение используется в большинстве документации по SSE, хотя в некоторых документах применяется соглашение $[x\ y\ z\ w]$, так что будьте внимательны!

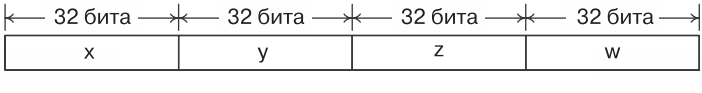


Рис. 4.38. Четыре компонента SSE-регистров в 32-битном режиме с плавающей точкой

Тип данных _m128

Чтобы инструкции SSE могли применять арифметику с упакованными данными с плавающей точкой, эти данные должны находиться в одном из регистров XMM*i*. Конечно, для долгосрочного хранения упакованные данные с плавающей точкой могут храниться в памяти, но требуется перенести их из ОЗУ в регистр SSE, прежде чем задействовать для каких-либо вычислений, а затем передать результаты обратно в ОЗУ.

Для упрощения работы с данными SSE и AVX компиляторы C и C++ предоставляют специальные типы данных, которые являются упакованными массивами чисел с плавающей точкой. Тип `_m128` инкапсулирует упакованный массив из четырех чисел с плавающей точкой для использования с внутренними компонентами SSE. (Типы `_m256` и `_m512` также представляют собой упакованные массивы из 8 и 16 значений с плавающей точкой соответственно для применения внутренними компонентами AVX.)

Тип данных `_m128` и его родственники могут задействоваться для объявления глобальных переменных, автоматических переменных, аргументов функций и возвращаемых типов и даже членов класса и структуры. Объявление автоматических переменных и аргументов функций типа `_m128` часто приводит к тому, что компилятор обрабатывает эти значения как прямые прокси для регистров SSE. Но использование типа `_m128` для объявления глобальных переменных, членов структуры/класса, а иногда и локальных переменных приводит к тому, что данные сохраняются в виде 16-байтового выровненного массива чисел с плавающей точкой в памяти. Применение основанной на памяти переменной `_m128` в вычислениях SSE приведет к тому, что компилятор неявно выдаст инструкции для загрузки данных из памяти в регистр SSE до выполнения вычислений, а также выдаст инструкции для сохранения результатов вычисления обратно в память, которая поддерживает каждую такую переменную. Поэтому рекомендуется провести дизассемблирование, чтобы убедиться, что вы не выполняете ненужных загрузок и сохранений регистров SSE при использовании типа `_m128` (и его родственников AVX).

Выравнивание данных SSE

Всякий раз, когда данные, предназначенные для использования в регистре XMM*i*, сохраняются в памяти, они должны быть выровнены по 16 байт (128 бит). (Аналогично данные, предназначенные для применения с 256-битными регистрами YMM*i* AVX, должны быть выровнены по 32 байта (256 бит), а данные для использования с 512-битными регистрами ZMM*i* должны быть выровнены по 64 байта (512 бит).)

Компилятор обеспечивает автоматическое выравнивание глобальных и локальных переменных типа `_m128`. Он также дополняет структуру и члены класса, так что любые члены типа `_m128` правильно выровнены *относительно* начала объекта, и гарантирует, что выравнивание всей структуры или класса равно выравниванию его членов в худшем случае. Это означает, что объявление экземпляра

глобальной или локальной переменной структуры или класса, включающего по крайней мере один член типа `_m128`, будет автоматически выровнено компилятором на 16 байт.

Однако все *динамически размещенные* экземпляры таких структуры или класса должны быть выровнены вручную. Аналогично любой массив чисел с плавающей точкой, который вы собираетесь использовать с инструкциями SSE, должен быть выровнен правильно. Вы можете убедиться в этом с помощью спецификатора `alignas` в C++11. (Дополнительная информация о выравнивании элементов в памяти есть в подразделе 6.2.1.)

Внутренний механизм SSE-компилятора

Мы могли бы напрямую работать с инструкциями на ассемблере SSE и AVX, возможно, с помощью встроенного синтаксиса компилятора. Однако не только написанный таким образом код непереносим, но и сам процесс не самый приятный! Чтобы упростить жизнь, современные компиляторы предоставляют *встроенные* функции — специальный синтаксис, который выглядит и ведет себя как обычная функция C, но на самом деле встраивает код ассемблера при компиляции. Многие встроенные функции переводятся в одну инструкцию на ассемблере, хотя некоторые являются *макросами*, переводящими в последовательность инструкций.

Чтобы использовать встроенные функции SSE и AVX, ваш файл `.cpp` должен включать `#include <xmmintrin.h>` в Visual Studio или `<x86intrin.h>` при компиляции с Clang или gcc.

Некоторые полезные встроенные функции SSE

Существует множество встроенных функций SSE, но для наших целей достаточно пяти из них.

- `_m128 _mm_set_ps (float w, float z, float y, float x);` — эта внутренняя переменная инициализирует переменную типа `_m128` четырьмя предоставленными значениями с плавающей точкой.
- `_m128 _mm_load_ps (const float * pData);` — это внутренняя загрузка четырех значений с плавающей точкой из массива в стиле C в переменную типа `_m128`. Входной массив должен быть выровнен по 16 байт.
- `void _mm_store_ps (float * pData, m128 v);` — эта встроенная функция хранит содержимое переменной типа `_m128` в массиве в стиле C из четырех чисел с плавающей точкой, которые должны быть выровнены по 16 байт.
- `_m128 _mm_add_ps (m128 a, m128 b);` — эта внутренняя функция складывает четыре пары чисел, содержащихся в переменных `a` и `b` параллельно, и возвращает результат.
- `_m128 _mm_mul_ps (m128 a, m128 b);` — это внутреннее произведение четырех пар чисел, содержащихся в переменных `a` и `b`, выполняется параллельно и возвращает результат.

Возможно, вы заметили, что аргументы `x`, `y`, `z` и `w` передаются в функцию `_mm_set_ps ()` в обратном порядке. Это странное соглашение, вероятно, вытекает из того факта, что процессоры Intel имеют архитектуру *little-endian*. Подобно тому как одно значение с плавающей точкой с битовой комбинацией 0x12345678 будет храниться в памяти в виде байтов 0x78, 0x56, 0x34, 0x12 в порядке увеличения адресов, содержимое регистра SSE хранится в памяти в порядке, противоположном порядку, в котором эти компоненты фактически появляются в регистре. Другими словами, в обратном порядке байтов хранятся не только четыре байта, составляющие каждое число с плавающей точкой в регистре SSE, но и сами эти четыре числа. Все это просто вопрос соглашения об именовании: в регистре SSE действительно нет наиболее или наименее значимых значений с плавающей точкой. Таким образом, мы можем либо рассматривать порядок в памяти как правильный и считать `_mm_set_ps ()` задом наперед, либо рассматривать аргументы `_mm_set_ps ()` как стоящие в правильном порядке и думать о векторе в памяти как расположенном задом наперед. Будем придерживаться прежнего соглашения, поскольку это означает, что мы сможем считывать векторы более естественным для нас образом: однородный вектор **v**, состоящий из членов (v_x, v_y, v_z, v_w), будет сохранен в массиве C/C++ как `float v[] = {vx, vy, vz, vw}`, но передан `_mm_set_ps()` как `w`, `z`, `y`, `x`.

Вот небольшой фрагмент кода, который загружает два четырехэлементных вектора с плавающей точкой, складывает их и печатает результаты:

```
#include <xmmintrin.h>

void TestAddSSE()
{
    alignas(16) float A[4];
    alignas(16) float B[4] = { 2.0f, 4.0f, 6.0f, 8.0f };

    // устанавливаем a = (1, 2, 3, 4) — литеральные значения
    // и загружаем b = (2, 4, 6, 8) из массива чисел с плавающей точкой,
    // просто чтобы проиллюстрировать два способа сделать это
    // (помните, что _mm_set_ps () представлен задом наперед!)
    __m128 a = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f);
    __m128 b = _mm_load_ps(&B[0]);

    // складываем векторы
    __m128 r = _mm_add_ps(a, b);

    // сохраняем 'm128 a' в массив float для печати
    _mm_store_ps(&A[0], a);

    // сохраняем результат в массив float для печати;
    alignas(16) float R[4];
    _mm_store_ps(&R[0], r);

    // проверяем результаты
    printf("a = %.1f %.1f %.1f %.1f\n",
           A[0], A[1], A[2], A[3]);
```

```
    printf("b = %.1f %.1f %.1f %.1f\n",
           B[0], B[1], B[2], B[3]);
    printf("r = %.1f %.1f %.1f %.1f\n",
           R[0], R[1], R[2], R[3]);
}
```

Векторный тип AltiVec

Если вкратце, компилятор GNU C/C++ gcc (используемый, например, для компиляции кода для PS3) обеспечивает работу набора команд PowerPC *AltiVec*, который поддерживает операции SIMD так же, как SSE на процессорах Intel. Стодвадцативосьмибитные векторные типы могут быть объявлены как обычные типы C/C++, но им предшествует ключевое слово `vector`. Например, переменная SIMD, содержащая четыре числа с плавающей точкой, будет объявлена как `vector float`. Компилятор gcc также позволяет записывать литеральные значения SIMD в исходный код. Например, вы можете инициализировать вектор чисел с плавающей точкой следующим значением:

```
vector float v = (vector float)(-1.0f, 2.0f, 0.5f, 1.0f);
```

Соответствующий код Visual Studio более неуклюжий:

```
// используем встроенный компилятор для загрузки литерального значения
// (помните, _mm_set_ps () идет задом наперед!)
__m128 v = _mm_set_ps(1.0f, 0.5f, 2.0f, -1.0f);
```

Я не буду подробно освещать тип AltiVec в этой главе, но, как только вы поймете SSE, работу с ним будет очень легко освоить.

4.10.2. Использование SSE для векторизации цикла

SIMD позволяет ускорить определенные виды вычислений в четыре раза, потому что одна инструкция машинного языка может выполнять арифметическую операцию с четырьмя значениями с плавающей точкой параллельно. Посмотрим, как это можно сделать с помощью встроенных функций SSE.

Сначала рассмотрим простой цикл, который складывает два массива `float` попарно, сохраняя каждый результат в выходной массив:

```
void AddArrays_ref(int count,
                   float* results,
                   const float* dataA,
                   const float* dataB,
{
    for (int i = 0; i < count; ++i)
    {
        results[i] = dataA[i] + dataB[i];
    }
}
```

Мы можем значительно ускорить этот цикл, используя встроенные функции SSE, например:

```
void AddArrays_sse(int count,
                   float* results,
                   const float* dataA,
                   const float* dataB)
{
    // Примечание: вызывающая сторона должна убедиться,
    // что все три массива имеют разный размер, кратный четырем!
    assert(count % 4 == 0);

    for (int i = 0; i < count; i += 4)
    {
        __m128 a = _mm_load_ps(&dataA[i]);
        __m128 b = _mm_load_ps(&dataB[i]);
        __m128 r = _mm_add_ps(a, b);
        _mm_store_ps(&results[i], r);
    }
}
```

В этой версии мы за итерацию цикла проходим по четыре значения за раз. Мы загружаем блоки из четырех чисел в регистры SSE, складываем их параллельно и сохраняем результаты в соответствующем блоке из четырех чисел в массиве результатов. Это называется *векторизацией* цикла. (В этом примере предполагается, что размер трех массивов равен и кратен четырем. Вызывающая сторона отвечает за *заполнение* массивов по мере необходимости одним, двумя или тремя фиктивными значениями каждый, чтобы удовлетворить это требование.)

Векторизация может привести к значительному увеличению скорости. Данный конкретный пример не точно в четыре раза быстрее из-за необходимости загружать значения в группы по четыре и сохранять результаты на каждой итерации. Но когда я измерил скорости этих функций, работающих на очень больших массивах элементов с плавающей точкой, невекторизованному циклу потребовалось примерно в 3,8 раза больше времени, чем векторизованному.

4.10.3. Векторизованное скалярное произведение

Применим векторизацию к более интересной задаче — расчету скалярного произведения. Цель состоит в том, чтобы рассчитать скалярные произведения двух массивов четырехэлементных векторов попарно и сохранить результаты в выходном массиве чисел с плавающей точкой. Вот эталонная реализация без использования SSE. В ней каждый смежный блок из четырех чисел с плавающей точкой содержится во входных массивах `a[]` и `b[]` (оба интерпретируются как один однородный четырехэлементный вектор):

```
void DotArrays_ref(int count,
                   float r[],
                   const float a[],
                   const float b[])
```

```
{
    for (int i = 0; i < count; ++i)
    {
        // обрабатываем каждый блок из четырех чисел с плавающей точкой
        // как одиночный четырехэлементный вектор
        const int j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

Первая попытка (медленная)

Вот первая попытка использования встроенных функций SSE для решения этой задачи:

```
void DotArrays_sse_horizontal(int count,
                              float r[],
                              const float a[],
                              const float b[])
{
    for (int i = 0; i < count; ++i)
    {
        // обрабатываем каждый блок из четырех чисел с плавающей точкой
        // как одиночный четырехэлементный
        const int j = i * 4;

        __m128 va = _mm_load_ps(&a[j]); // ax,ay,az,aw
        __m128 vb = _mm_load_ps(&b[j]); // bx,by,bz,bw

        __m128 v0 = _mm_mul_ps(va, vb);

        // выполняем сложение для всего регистра...
        __m128 v1 = _mm_hadd_ps(v0, v0);
        // (v0w+v0z, v0y+v0x, v0w+v0z, v0y+v0x)
        __m128 vr = _mm_hadd_ps(v1, v1);
        // (v0w+v0z+v0y+v0x, v0w+v0z+v0y+v0x,
        // v0w+v0z+v0y+v0x, v0w+v0z+v0y+v0x)

        _mm_store_ss(&r[i], vr); // extract vr.x as a float
    }
}
```

Эта реализация требует новой инструкции `_mm_hadd_ps()` *(горизонтальное сложение)*. Данная внутренняя функция работает с одним входным регистром (x, y, z, w) и вычисляет две суммы: $s = x + y$ и $t = z + w$. Эти суммы хранятся в четырех слотах регистра назначения как (t, s, t, s). Выполнение операции дважды позволяет вычислить сумму $d = x + y + z + w$. Это называется *сложением через регистр*.

Сложение через регистр — обычно не слишком хорошая идея, потому что это очень медленная операция. Профилирование реализации `DotArrays_sse()` показывает, что на самом деле она занимает немного больше времени, чем эталонная реализация. Использование SSE здесь действительно все замедлило![1]

Применение более эффективного подхода

Ключ к пониманию силы SIMD-параллелизма для скалярного произведения состоит в том, чтобы найти способ избежать необходимости сложения через регистр. Это можно сделать, но сначала следует *транспонировать* входные векторы. Храня их в транспонированном виде, мы можем вычислить скалярное произведение точно так же, как рассчитывали его при использовании чисел с плавающей точкой: умножаем x-компоненты, результат добавляем к произведению y-компонентов, затем добавляем результат сложения к произведению z-компонентов и, наконец, добавляем полученный результат к произведению w-компонентов. Вот как выглядит итог:

```
void DotArrays_sse(int count,
                   float r[],
                   const float a[],
                   const float b[])
{
    for (int i = 0; i < count; i += 4)
    {
        __m128 vaX = _mm_load_ps(&a[(i+0)*4]); // a[0,4,8,12]
        __m128 vaY = _mm_load_ps(&a[(i+1)*4]); // a[1,5,9,13]
        __m128 vaZ = _mm_load_ps(&a[(i+2)*4]); // a[2,6,10,14]
        __m128 vaW = _mm_load_ps(&a[(i+3)*4]); // a[3,7,11,15]

        __m128 vbX = _mm_load_ps(&b[(i+0)*4]); // b[0,4,8,12]
        __m128 vbY = _mm_load_ps(&b[(i+1)*4]); // b[1,5,9,13]
        __m128 vbZ = _mm_load_ps(&b[(i+2)*4]); // b[2,6,10,14]
        __m128 vbW = _mm_load_ps(&b[(i+3)*4]); // b[3,7,11,15]

        __m128 result;
        result = _mm_mul_ps(vaX, vbX);
        result = _mm_add_ps(result, _mm_mul_ps(vaY, vbY));
        result = _mm_add_ps(result, _mm_mul_ps(vaZ, vbZ));
        result = _mm_add_ps(result, _mm_mul_ps(vaW, vbW));

        _mm_store_ps(&r[i], result);
    }
}
```

[1] С SSE4 корпорация Intel представила встроенную функцию _mm_dp_ps() (и соответствующую инструкцию dpps), которая вычисляет скалярное произведение с несколько меньшей задержкой по сравнению с версией, включающей два вызова _mm_hadd_ps(). Но все горизонтальные сложения очень дороги, и их следует избегать везде, где это возможно.

Инструкция madd. Интересно отметить, что умножение, сопровождаемое сложением, является настолько распространенной операцией, что имеет собственное имя — *madd*. Некоторые процессоры предоставляют одну SIMD-инструкцию для выполнения операции *madd*. Например, эту операцию выполняет встроенная функция PowerPC Altiviec `vec_madd()`. Таким образом, в AltiVec тело функции `DotArrays()` может быть немного упрощено:

```
vector float result = vec_mul(vaX, vbX);
result = vec_madd(vaY, vbY, result));
result = vec_madd(vaZ, vbZ, result));
result = vec_madd(vaW, vbW, result));
```

Транспонирование как оно есть

Вышеуказанная реализация *предполагает*, что входные данные уже были транспонированы вызывающей стороной. Другими словами, предполагается, что массив `a[]` содержит компоненты $\{a_0, a_4, a_8, a_{12}, a_1, a_5, a_9, a_{13}...\}$, то же самое для массива `b[]`. Если мы хотим работать с входными данными в том же формате, что и для реализации ссылок, то должны будем выполнить транспонирование внутри функции. Вот так:

```
void DotArrays_sse_transpose(int count,
                             float r[],
                             const float a[],
                             const float b[])
{
    for (int i = 0; i < count; i += 4)
    {
        __m128 vaX = _mm_load_ps(&a[(i+0)*4]); // a[0,1,2,3]
        __m128 vaY = _mm_load_ps(&a[(i+1)*4]); // a[4,5,6,7]
        __m128 vaZ = _mm_load_ps(&a[(i+2)*4]); // a[8,9,10,11]
        __m128 vaW = _mm_load_ps(&a[(i+3)*4]); // a[12,13,14,15]

        __m128 vbX = _mm_load_ps(&b[(i+0)*4]); // b[0,1,2,3]
        __m128 vbY = _mm_load_ps(&b[(i+1)*4]); // b[4,5,6,7]
        __m128 vbZ = _mm_load_ps(&b[(i+2)*4]); // b[8,9,10,11]

        __m128 vbW = _mm_load_ps(&b[(i+3)*4]); // b[12,13,14,15]
        _MM_TRANSPOSE4_PS(vaX, vaY, vaZ, vaW);
                                        // vaX = a[0,4,8,12]
                                        // vaY = a[1,5,9,13]
                                        // ...
        _MM_TRANSPOSE4_PS(vbX, vbY, vbZ, vbW);
                                        // vbX = b[0,4,8,12]
                                        // vbY = b[1,5,9,13]
                                        // ...
        __m128 result;
        result = _mm_mul_ps(vaX, vbX);
        result = _mm_add_ps(result, _mm_mul_ps(vaY, vbY));
        result = _mm_add_ps(result, _mm_mul_ps(vaZ, vbZ));
        result = _mm_add_ps(result, _mm_mul_ps(vaW, vbW));
```

```
        _mm_store_ps(&r[i], result);
    }
}
```

Эти два вызова `_MM_TRANSPOSE()` на самом деле являются вызовами довольно сложного макроса, который использует команды *тасования* для перемещения компонентов четырех входных регистров. К счастью, тасование не особо дорогостоящая операция, поэтому транспонирование векторов при вычислении скалярных произведений не приводит к чрезмерным накладным расходам. Профилирование всех трех реализаций `DotArrays()` показывает, что окончательная версия (та, которая транспонирует векторы по ходу) примерно в 3,5 раза быстрее, чем эталонная реализация.

Тасование и транспонирование

Для любопытных читателей. Вот как выглядит макрос `_MM_TRANSPOSE ()`:

```
#define _MM_TRANSPOSE4_PS(row0, row1, row2, row3)     \
    { __m128 tmp3, tmp2, tmp1, tmp0;                  \
                                                      \
    tmp0 = _mm_shuffle_ps((row0), (row1), 0x44);      \
    tmp2 = _mm_shuffle_ps((row0), (row1), 0xEE);      \
    tmp1 = _mm_shuffle_ps((row2), (row3), 0x44);      \
    tmp3 = _mm_shuffle_ps((row2), (row3), 0xEE);      \
                                                      \
     (row0) = _mm_shuffle_ps(tmp0, tmp1, 0x88);       \
     (row1) = _mm_shuffle_ps(tmp0, tmp1, 0xDD);       \
     (row2) = _mm_shuffle_ps(tmp2, tmp3, 0x88);       \
     (row3) = _mm_shuffle_ps(tmp2, tmp3, 0xDD); }
```

Эти сумасшедшие шестнадцатеричные числа представляют собой упакованные в биты четырехэлементные поля, называемые *масками тасования*. Они сообщают встроенной функции `_mm_shuffle()`, как именно перетасовать компоненты. Эти битовые поля являются распространенным источником путаницы, возможно, из-за соглашений об именах, используемых в большинстве документов. В действительности все довольно просто. Маска тасования состоит из четырех целых чисел, каждое из которых представляет один из компонентов регистра SSE (и, следовательно, может иметь значение от 0 до 3):

```
#define SHUFMASK(p,q,r,s) \
               (p | (q<<2) | (r<<4) | (s<<6))
```

Передача регистров SSE `a` и `b` вместе с маской тасования в `_mm_shuffle_ps()` приводит к тому, что элементы `a` и `b` появляются в выходном регистре `r` следующим образом:

```
__m128 a = ...;
__m128 b = ...;
__m128 r = _mm_shuffle_ps(a, b,
                          SHUFMASK(p,q,r,s));
// r == ( a[p], a[q], b[r], b[s] )
```

4.10.4. Векторно-матричное умножение с помощью SSE

Теперь, когда мы понимаем, как выполнить скалярное произведение, можем умножить вектор из четырех элементов на матрицу 4×4. Для этого нужно просто выполнить четыре скалярных произведения между входным вектором и каждой из четырех строк входной матрицы.

Начнем с определения класса `Mat44`, который инкапсулирует четыре вектора SSE, представляющих четыре строки матрицы. Используем объединение, чтобы легко получать доступ к отдельным элементам матрицы как к числам с плавающей точкой (это работает, потому что экземпляры класса `Mat44` всегда находятся в памяти, а не непосредственно в регистрах SSE):

```
union Mat44
{
    float c[4][4]; // компоненты
    __m128 row[4]; // строки
};
```

Функция умножения вектора и матрицы выглядит следующим образом:

```
__m128 MulVecMat_sse(const __m128& v, const Mat44& M)
{
    // сперва транспонирование v
    __m128 vX = _mm_shuffle_ps(v, v, 0x00); // (vx,vx,vx,vx)
    __m128 vY = _mm_shuffle_ps(v, v, 0x55); // (vy,vy,vy,vy)
    __m128 vZ = _mm_shuffle_ps(v, v, 0xAA); // (vz,vz,vz,vz)
    __m128 vW = _mm_shuffle_ps(v, v, 0xFF); // (vw,vw,vw,vw)

    __m128 r = _mm_mul_ps(vX, M.row[0]);
    r = _mm_add_ps(r, _mm_mul_ps(vY, M.row[1]));
    r = _mm_add_ps(r, _mm_mul_ps(vZ, M.row[2]));
    r = _mm_add_ps(r, _mm_mul_ps(vW, M.row[3]));
    return r;
}
```

Перестановки используются для репликации одного компонента **v** (любого из v_x, v_y, v_z или v_w) по *всем четырем* линиям регистра SSE. Это работает как транспонирование **v** перед выполнением скалярного произведения со строками M, которые уже транспонированы. (Помните, что векторно-матричное умножение обычно включает в себя получение скалярных произведений между входным вектором и *столбцами* матрицы. Здесь мы переносим **v** в четыре регистра SSE, а затем выполняем умножение покомпонентно со строками матрицы).

4.10.5. Матрица-матричное умножение с SSE

Перемножение двух матриц 4×4 с применением SSE является тривиальным, если есть функция для перемножения вектора и матрицы. Вот как выглядит код:

```
void MulMatMat_sse(Mat44& R, const Mat44& A, const Mat44& B)
{
    R.row[0] = MulVecMat_sse(A.row[0], B);
    R.row[1] = MulVecMat_sse(A.row[1], B);
    R.row[2] = MulVecMat_sse(A.row[2], B);
    R.row[3] = MulVecMat_sse(A.row[3], B);
}
```

4.10.6. Обобщенная векторизация

Поскольку регистр SSE содержит четыре значения с плавающей точкой, заманчиво считать его естественным представлением четырехэлементного однородного вектора **v** и думать, что наилучшее использование SSE — выполнение векторных вычислений для 3D. Однако это очень ограниченное представление о параллелизме SIMD.

Большинство пакетных операций, в которых одно вычисление выполняется многократно для большого набора данных, могут быть векторизованы с помощью SIMD-параллелизма. Если задуматься, компоненты регистра SIMD действительно функционируют как параллельные дорожки, на которых может выполняться произвольная обработка. Работа с переменными типа `float` предоставляет нам одну дорожку, а работа с 128-битными (четырехэлементными) переменными SIMD позволяет выполнять те же вычисления параллельно на четырех дорожках. Иными словами, мы можем проводить вычисления по четыре за раз. Работа с 256-битными регистрами AVX дает восемь полос, что позволяет выполнять восемь вычислений за раз. А AVX-512 дает 16 полос, что позволяет производить 16 расчетов одновременно.

Самый простой способ написания *векторизованного* кода — начать с создания его в виде алгоритма с одной полосой (просто используя `float`). Как только он заработает, можно преобразовать его для одновременной работы с N элементами, применяя SIMD-регистры с пропускной способностью N полос. Мы уже видели этот процесс в действии: в подразделе 4.10.3 сначала написали цикл, который выполнял большую партию скалярных произведений по одному, а затем перешли к применению SSE, чтобы выполнять эти вычисления по четыре за раз.

Таким образом, приятным побочным эффектом векторизации кода является то, что, лишь немного изменив код, можно воспользоваться преимуществами более широкого SIMD-оборудования. На машине с поддержкой только SSE вы выполняете четыре операции за итерацию цикла, на машине, которая поддерживает AVX, просто меняете код на восемь операций за итерацию, а в системе AVX-512 можете выполнить 16 операций за итерацию.

Интересно, что большинство оптимизирующих компиляторов могут автоматически векторизовать некоторые виды однолинейных циклов. Фактически при написании приведенных ранее примеров потребовалось некоторое усилие, чтобы заставить компилятор *не* векторизовать однополосный код, чтобы я мог сравнить его производительность с реализацией SIMD! Еще раз напомню: при написании оптимизированного кода полезно взглянуть на код ассемблера — можно обнаружить, что компилятор делает больше (или меньше), чем вы думали!

4.10.7. Предикация векторов

Рассмотрим другой (полностью выдуманный) пример. Он будет отражать идеи обобщенной векторизации, а также послужит иллюстрацией еще одного полезного метода — *предикации векторов*.

Представьте, что нужно взять квадратные корни большого массива чисел с плавающей точкой. Начнем с кода в виде однополосного цикла, например, так:

```
#include <cmath>

void SqrtArray_ref(float* __restrict__ r,
                   const float* __restrict__ a,
                   int count)
{
    for (unsigned i = 0; i < count; ++i)
    {
        if (a[i] >= 0.0f)
            r[i] = std::sqrt(a[i]);
        else
            r[i] = 0.0f;
    }
}
```

Далее преобразуем этот цикл в SSE, выполняя четыре извлечения квадратных корня за раз:

```
#include <xmmintrin.h>

void SqrtArray_sse_broken(float* __restrict__ r,
                          const float* __restrict__ a,
                          int count)
{
    assert(count % 4 == 0);
    __m128 vz = _mm_set1_ps(0.0f); // все нули

    for (int i = 0; i < count; i += 4)
    {
        __m128 va = _mm_load_ps(a + i);

        __m128 vr;
        if (_mm_cmpge_ps(va, vz)) // ???
            vr = _mm_sqrt_ps(va);
        else
            vr = vz;

        _mm_store_ps(r + i, vr);
    }
}
```

Это кажется довольно простым: мы просто перемещаемся по входному массиву по четыре числа с плавающей точкой за раз, загружаем группы по четыре числа с плавающей точкой в регистр SSE, а затем выполняем четыре параллельных извлечения квадратного корня с помощью `_mm_sqrt_ps()`.

Однако в этом цикле есть одна маленькая ошибка. Нам нужно проверить, больше или равны входные значения нулю, потому что квадратный корень из отрицательного числа является мнимым (и, следовательно, будет давать QNaN). Внутренняя функция `_mm_cmpge_ps()` сравнивает значения двух регистров SSE по компонентам, чтобы определить, больше они или равны вектору значений, предоставленных вызывающей стороной. Однако эта функция не возвращает `bool`. Как такое может быть? Мы сравниваем четыре значения с четырьмя другими значениями, поэтому некоторые из них могут пройти тест, а другие его не пройдут. Это означает, что нельзя просто выполнить проверку `if` для результатов `_mm_cmpge_ps()`[1].

Означает ли это, что векторизованная реализация обречена? К счастью, нет. Все встроенные функции сравнения SSE, такие как `_mm_cmpge_ps()`, возвращают четырехкомпонентный результат, сохраняемый в регистре SSE. Но вместо того, чтобы содержать четыре значения с плавающей точкой, результат состоит из четырех 32-битных масок. Каждая маска содержит только двоичные 1 (0xFFFFFFFFU), если соответствующий компонент во входном регистре прошел тест, и только двоичные 0 (0x0U), если не прошел.

Мы можем использовать результат встроенного сравнения SSE, применяя его в качестве битовой маски для *выбора* одного из двух возможных результатов. В нашем примере, когда входное значение проходит тест (больше или равно нулю), мы хотим найти его квадратный корень, когда оно не проходит тест (отрицательно), хотим вернуть значение 0. Этот процесс называется *предикацией*, и поскольку мы применяем его к векторам SIMD, это будет *предикация векторов*.

В подразделе 4.2.6 мы видели, как выполнять предикацию со значениями с плавающей точкой, используя побитовые операторы И, ИЛИ и НЕ. Вот выдержка из этого примера:

```
    // ...

    // выбираем частное, когда маска равна единице, или значение
    // по умолчанию d, когда маска – все нули (Примечание: это не будет
    // работать как написано, – вам нужно использовать объединение (union)
    // для интерпретации значения с плавающей точкой – беззнаковое
    // для маскировки)
    const float result = (q & mask) | (d & ~mask);
    return result;
}
```

[1] Проверка if в эталонной реализации с одной линией также не позволяет компилятору автоматически векторизовать цикл.

Здесь все так же, просто нужно использовать SSE-версии побитовых операторов:

```
#include <xmmintrin.h>

void SqrtArray_sse(float* __restrict__ r,
                   const float* __restrict__ a,
                   int count)
{
    assert(count % 4 == 0);
    __m128 vz = _mm_set1_ps(0.0f);

    for (int i = 0; i < count; i += 4)
    {
        __m128 va = _mm_load_ps(a + i);
        // всегда вычисляем частное, но в результате
        // может возникнуть QNaN на некоторых
        // или на всех линиях
        __m128 vq = _mm_sqrt_ps(va);

        // теперь выбираем между vq и vz в зависимости
        // от того, был вход больше
        // или равен нулю или нет
        __m128 mask = _mm_cmpge_ps(va, zero);

        // (vq & mask) | (vz & ~mask)
        __m128 qmask = _mm_and_ps(mask, vq);
        __m128 znotmask = _mm_andnot_ps(mask, vz);
        __m128 vr = _mm_or_ps(qmask, znotmask);

        _mm_store_ps(r + i, vr);
    }
}
```

Это удобно и распространено для инкапсуляции данной операции предикации вектора в функцию, которая обычно называется *выбором вектора*. AltiVec ISA в PowerPC предоставляет для этой цели встроенную функцию `vec_sel()`. Она работает следующим образом:

```
// псевдокод, показывающий, как работает
// встроенная функция AltiVec vec_sel()
vector float vec_sel(vector float falseVec,
                     vector float trueVec,
                     vector bool mask)
{
    vector float r;
    for (each lane i)
    {
        if (mask[i] == 0)
            r[i] = falseVec[i];
```

```
        else
            r[i] = trueVec[i];
    }
    return r;
}
```

В SSE2 инструкции выбора вектора не было, но, к счастью, она введена в SSE4 — ее порождает встроенная функция `_mm_blendv_ps()`.

Посмотрим, как можно реализовать операцию выбора вектора. Мы можем написать ее так:

```
__m128 _mm_select_ps(const __m128 a,
                     const __m128 b,
                     const __m128 mask)
{
        // (b & mask) | (a & ~mask)
        __m128 bmask = _mm_and_ps(mask, b);
        __m128 anotmask = _mm_andnot_ps(mask, a);
        return _mm_or_ps(bmask, anotmask);
}
```

А если чувствуем себя смелыми, можем сделать то же самое с помощью исключающего ИЛИ:

```
__m128 _mm_select_ps(const __m128 a,
                     const __m128 b,
                     const __m128 mask)
{
    // (((a ^ b) & mask) ^ a)
    __m128 diff = _mm_xor_ps(a, b);
    return _mm_xor_ps(a, _mm_and_ps(mask, diff));
}
```

Посмотрите, сможете ли вы понять, как это работает. Вот две подсказки: оператор исключающего ИЛИ вычисляет *побитовую разницу* между двумя значениями. Два XOR в строке оставляют входное значение без изменений: `((a ^ b) ^ b == a)`.

4.11. Введение в программирование GPGPU

В предыдущем разделе мы говорили, что большинство оптимизирующих компиляторов могут автоматически векторизовать некоторый код, если целевой ЦП включает в себя модуль векторной обработки SIMD и исходный код соответствует определенным требованиям, например не включает сложного ветвления. Векторизация также является одним из столпов программирования *GPU общего назначения* (general-purpose GPU, GPGPU). В этом разделе мы кратко рассмотрим, как GPU отличается от CPU с точки зрения аппаратной архитектуры и как концепции параллелизма и векторизации SIMD позволяют программистам писать *вычислительные шейдеры*, способные обрабатывать большие объемы данных параллельно на графическом процессоре.

4.11.1. Параллельные вычисления данных

Графический процессор — это специализированный сопроцессор, разработанный специально для ускорения вычислений, требующих высокой степени *параллелизма данных*. Ускорение достигается сочетанием параллелизма SIMD (векторизованных ALU) с параллелизмом MIMD (с использованием формы вытесняющей многопоточности). NVIDIA ввела термин *«одиночный поток команд, множественные потоки»* (SIMT), чтобы описать гибридный дизайн SIMD/MIMD, хотя он не уникален для графических процессоров NVIDIA. Особенности проектирования графических процессоров различаются от вендора к вендору и от линейки продуктов к линейке, но все графические процессоры применяют общие принципы параллелизма SIMT в своей архитектуре.

Графические процессоры предназначены специально для выполнения *параллельных вычислений* с очень большими наборами данных. Для того чтобы вычислительная задача хорошо подходила для исполнения на графическом процессоре, вычисления, выполняемые для любого одного элемента набора данных, должны быть *независимыми* от результатов вычислений для других элементов. Иными словами, должна существовать возможность обрабатывать элементы в любом порядке.

Простые примеры векторизации SIMD, которые мы представили, начиная с подраздела 4.10.3, — это примеры вычислений параллельных данных. Вспомните функцию, которая обрабатывает два потенциально очень больших массива входных векторов и создает выходной массив, содержащий скалярные произведения этих векторов:

```
void DotArrays_ref(unsigned count,
                   float r[],
                   const float a[],
                   const float b[])
{
    for (unsigned i = 0; i < count; ++i)
    {
        // обрабатываем каждый блок из четырех чисел с плавающей точкой
        // как одиночный четырехэлементный вектор
        const unsigned j = i * 4;

        r[i] = a[j+0]*b[j+0]   // ax*bx
             + a[j+1]*b[j+1]   // ay*by
             + a[j+2]*b[j+2]   // az*bz
             + a[j+3]*b[j+3];  // aw*bw
    }
}
```

Вычисления, выполняемые на каждой итерации этого цикла, не зависят от вычислений на других итерациях. Это означает, что мы можем выполнять вычисления в любом порядке, который сочтем нужным. Более того, именно это свойство позволяет применять встроенные функции SSE или AVX для векторизации цикла — вместо выполнения вычислений по одному мы можем с помощью параллелизма

SIMD одновременно производить 4, 8 или 16 вычислений, уменьшая тем самым количество итераций в 4, 8 или 16 раз соответственно.

А теперь представьте, что распараллеливание SIMD доходит до крайности. Что, если бы у нас был SIMD VPU с 1024 полосами? В этом случае мы могли бы разделить общее количество итераций на 1024, а когда входные массивы содержат 1024 элемента или меньше, можно было бы выполнить весь цикл за одну итерацию!

Если грубо, это то, что делает GPU. Однако он не использует SIMD, ширина каждого из которых на самом деле 1024 полосы. Ширина *SIMD-блоков* графического процессора обычно 8 или 16 полос, но они обрабатывают рабочие нагрузки партиями по 32 или 64 элемента за раз. Более того, графический процессор содержит *много* SIMD-блоков. Таким образом, большая нагрузка может распределяться между SIMD-блоками параллельно, в результате чего графический процессор способен обрабатывать буквально тысячи элементов данных одновременно.

Параллельные вычисления данных — это именно то, что доктор прописал при применении пиксельного шейдера (также известного как фрагментный шейдер) к миллионам пикселов или вершинного шейдера — к сотням тысяч или даже миллионам вершин трехмерного меша для каждого кадра на скорости 30 или 60 FPS. Но современные графические процессоры предоставляют свои вычислительные возможности программистам, что позволяет нам писать *вычислительные шейдеры* общего назначения. Поскольку вычисления, которые мы хотим выполнить на большом наборе данных, в значительной степени независимы друг от друга, графический процессор, вероятно, может выполнять их более эффективно, чем центральный процессор.

4.11.2. Вычислительные ядра

В подразделе 4.10.3 мы видели, что для векторизации цикла, подобного циклу в функции `DotArrays_ref()`, следует переписать код. Векторизованная версия функции использует встроенные функции SSE или AVX, скалярные типы данных заменяются векторными типами, такими как SSE `_m128` или `vector float` AltiVec, и цикл жестко закодирован для итерации 4, 8 или 16 элементов одновременно.

При написании вычислительного шейдера GPGPU мы придерживаемся другого подхода. Вместо того чтобы жестко кодировать цикл для работы в пакетах фиксированного размера, оставляем цикл как однолинейное вычисление с помощью скалярных типов данных. Затем извлекаем тело цикла в отдельную функцию, известную как *ядро*. Вот так приведенный ранее пример будет выглядеть как ядро:

```
void DotKernel(unsigned i,
               float r[],
               const float a[],
               const float b[])
{
    // обрабатываем каждый блок из четырех чисел с плавающей точкой
    // как одиночный четырехэлементный вектор
    const unsigned j = i * 4;
```

```
    r[i] = a[j+0]*b[j+0] // ax*bx
         + a[j+1]*b[j+1] // ay*by
         + a[j+2]*b[j+2] // az*bz
         + a[j+3]*b[j+3]; // aw*bw
}
void DotArrays_gpgpu1(unsigned count,
                      float r[],
                      const float a[],
                      const float b[])
{
    for (unsigned i = 0; i < count; ++i)
    {
        DotKernel(i, r, a, b);
    }
}
```

Функция `DotKernel()` теперь представлена в форме, которая подходит для преобразования в *вычислительный шейдер*. Он обрабатывает только один элемент входных данных и дает один элемент вывода. Это аналогично тому, как пиксельный/фрагментный шейдер получает один входной пиксельный/фрагментный цвет и преобразует его в один выходной цвет или как вершинный шейдер получает одну входную вершину и создает одну выходную. Графический процессор эффективно выполняет цикл `for`, вызывая функцию ядра один раз для каждого элемента набора данных.

Вычислительные ядра GPGPU обычно пишутся на специальном *языке шейдинга*, который может быть скомпилирован в машинный код, понятный GPU. Языки шейдеров обычно очень близки к C по синтаксису, поэтому преобразование цикла C или C++ в вычислительное ядро GPU обычно не представляет особой сложности. Некоторые примеры языков шейдинга включают HLSL (high-level shader language — высокоуровневый язык шейдеров) DirectX, GLSL OpenGL, языки Cg и CUDA C NVIDIA и OpenCL.

Некоторые языки шейдеров требуют, чтобы вы перемещали код ядра в специальные исходные файлы, отдельно от кода приложения C++. Но OpenCL и CUDA C являются расширениями языка C++. Они позволяют программистам писать вычислительные ядра как обычные функции C/C++ с незначительными синтаксическими изменениями и вызывать эти ядра в GPU относительно простым синтаксисом.

В качестве конкретного примера — функция `DotKernel()`, написанная на CUDA C:

```
__global__ void DotKernel_CUDA(int count,
                               float* r,
                               const float* a,
                               const float* b)
{
    // CUDA предоставляет волшебный индекс потока
    // каждому вызову ядра — он служит
    // индексом цикла i
    size_t i = threadIdx.x;
```

```
    // убедитесь, что индекс действителен
    if (i < count)
    {
        // обрабатываем каждый блок из четырех чисел с плавающей точкой
        // как одиночный четырехэлементный вектор
        const unsigned j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

Вы заметите, что индекс цикла `i` взят из переменной `threadIdx` в самой функции ядра. Индекс потока является «волшебным» вводом, предоставляемым компилятором, во многом так же, как указатель `this` «магически» указывает на текущий экземпляр в функции-члене класса C++. Более подробно об индексе потоков поговорим в следующем подразделе.

4.11.3. Выполнение ядра

Учитывая, что мы написали вычислительное ядро, давайте посмотрим, как выполнить его на GPU. Детали различаются от языка шейдера к языку, но ключевые понятия примерно одинаковы. Например, вот как мы запускаем вычислительное ядро в CUDA C:

```
void DotArrays_gpgpu2(unsigned count,
                      float r[],
                      const float a[],
                      const float b[])
{
    // выделяем управляемые буферы, видимые
    // процессору и графическому контроллеру
    int *cr, *ca, *cb;
    cudaMallocManaged(&cr, count * sizeof(float));
    cudaMallocManaged(&ca, count * sizeof(float) * 4);
    cudaMallocManaged(&cb, count * sizeof(float) * 4);

    // передаем данные в видимую GPU память
    memcpy(ca, a, count * sizeof(float) * 4);
    memcpy(cb, b, count * sizeof(float) * 4);

    // запускаем ядро на GPU
    DotKernel_CUDA <<<1, count>>> (cr, ca, cb, count);

    // ждем окончания работы графического процессора
    cudaDeviceSynchronize();
```

```
    // возвращаем результаты и подчищаем за собой
    memcpy(r, cr, count * sizeof(float));
    cudaFree(cr);
    cudaFree(ca);
    cudaFree(cb);
}
```

Небольшой код настройки требуется для выделения входных и выходных буферов в качестве управляемой памяти, которую видят как CPU, так и GPU, и для копирования в них входных данных. Специфичная для CUDA тройная угловая скобка (`<<<G,N>>>`) затем запускает вычислительное ядро на GPU, отправляя запрос драйверу. Вызов `cudaDeviceSynchronize()` заставляет ЦП ждать, пока графический процессор выполнит свою работу (во многом аналогично тому, как `pthread_join()` заставляет один поток ожидать завершения другого). Наконец, мы освобождаем графические буферы данных.

Давайте внимательнее посмотрим на `<<<G,N>>>`. Второй аргумент `N` в угловых скобках позволяет указать количество *измерений* входных данных. Это соответствует количеству *итераций* цикла, выполнения которых мы хотим от графического процессора. Это может быть одно-, двух- или трехмерная величина, позволяющая обрабатывать одно-, двух- или трехмерные входные массивы. Однако, как и в C/C++, многомерный массив является просто одномерным массивом, который индексируется особым образом. Например, в C/C++ доступ к двухмерному массиву, написанный как `[row] [column]`, действительно эквивалентен доступу к одномерному массиву `[row * numColumns + column]`. Тот же принцип применяется к многомерным буферам графического процессора.

Первый аргумент `G` в угловых скобках указывает драйверу, сколько *групп потоков* (*блоков потоков* в терминологии NVIDIA) использовать при запуске этого вычислительного ядра. Вычислительное задание с одной группой потоков предназначено для выполнения на одном *вычислительном устройстве* на GPU. Вычислительная единица по сути является ядром в графическом процессоре — аппаратным компонентом, способным выполнять поток команд. Передача большего числа для `G` позволяет драйверу разделить рабочую нагрузку между несколькими вычислительными блоками.

4.11.4. Потоки GPU и группы потоков

Аргумент `G` сообщает драйверу GPU, на сколько групп потоков разделить работу. Как и следовало ожидать, группа потоков состоит из некоторого числа *потоков графического процессора*. Но что именно означает термин «поток» в контексте графического процессора?

Каждое ядро GPU компилируется в *поток команд*, состоящий из последовательности инструкций машинного языка GPU, почти так же, как функция C/C++ компилируется в поток инструкций CPU. Таким образом, поток GPU эквивалентен потоку CPU в том смысле, что оба вида представляют собой поток инструкций

машинного языка, которые могут выполняться одним или несколькими ядрами. Однако графический процессор выполняет свои потоки способом, несколько отличающимся от способа, которым это делает процессор. В результате термин «поток» имеет разное значение, когда применяется к вычислительным ядрам на GPU и к программам на процессоре.

Чтобы понять различие в терминологии, кратко рассмотрим архитектуру графического процессора. В подразделе 4.11.1 было сказано, что графический процессор состоит из нескольких *вычислительных блоков*, каждый из которых содержит некоторое количество *SIMD-блоков*. Мы можем представить вычислительный блок как урезанное ядро ЦП: он содержит блок выборки/декодирования инструкций, возможно, некоторые кэши памяти, скалярный ALU и обычно где-то рядом четыре SIMD-блока, которые выполняют почти ту же функцию, что и блок векторной обработки (VPU) в процессоре, где используется SIMD. Эта архитектура показана на рис. 4.39.

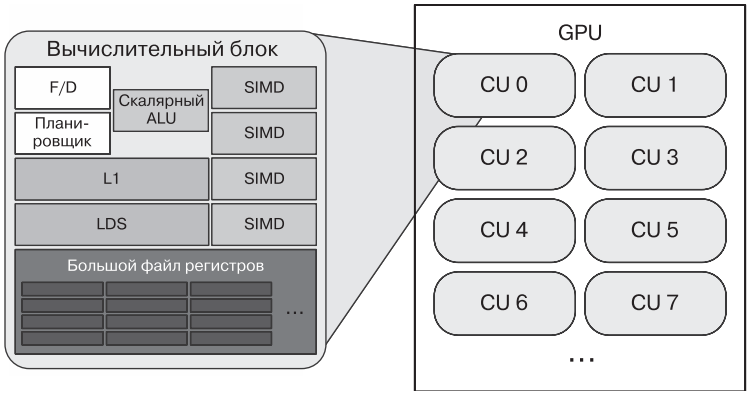


Рис. 4.39. Типичный графический процессор состоит из нескольких вычислительных блоков (CU), каждый из которых действует как урезанное ядро процессора. CU содержит модуль выборки/декодирования команд, кэш-память L1, возможно, блок локального ОЗУ для хранения данных (LDS), скалярный ALU и несколько блоков SIMD для выполнения векторизованного кода

Блоки SIMD в CU имеют разную ширину полосы на разных графических процессорах, но для упрощения предположим, что мы работаем на архитектуре AMD Radeon™ Graphics Core Next (GCN), в которой SIMD имеют ширину 16 полос. ЦП не способен к упреждающему или внеочередному выполнению — он просто читает поток инструкций и выполняет их одну за другой, используя блоки SIMD, чтобы применять каждую инструкцию[1] к 16 элементам входных данных параллельно.

[1] В вычислительных блоках на графическом процессоре есть скалярный ALU, следовательно, он может выполнять некоторые инструкции в режиме «одна линия», работая с одним входным параметром за раз.

Чтобы выполнить вычислительное ядро на ЦП, драйвер сначала подразделяет входные буферы данных на блоки, состоящие из 64 элементов. Для каждого из этих 64-элементных блоков данных вычислительное ядро вызывается на одном ЦП. Этот вызов называется *волновым фронтом* (в NVIDIA — *warp*). При выполнении волнового фронта ЦП выбирает и декодирует инструкции ядра одну за другой. Каждая инструкция применяется к 16 элементам данных на *этапе блокировки* с использованием блока SIMD. Внутри SIMD-блок состоит из *четырехступенчатого* конвейера, поэтому для его выполнения требуется четыре такта. Таким образом, вместо того, чтобы позволить этапам конвейера бездействовать в течение трех тактов из каждых четырех, CU применяет одну и ту же инструкцию еще три раза еще к трем блокам из 16 элементов данных.

Вот почему волновой фронт состоит из 64 элементов данных, хотя SIMD в ЦП имеет ширину всего 16 полос.

Из-за этого несколько специфического способа, которым центральный процессор выполняет поток инструкций, термин *«поток GPU»* фактически относится к одной полосе SIMD. Таким образом, вы можете думать о GPU-потоке как об одной итерации исходного невекторизованного цикла, с которого мы начали, прежде чем преобразовать его тело в вычислительное ядро. В качестве альтернативы можете думать о GPU-потоке как о единственном вызове функции ядра, работающем с одним входным набором данных и создающем один выходной набор данных. То, что GPU фактически запускает несколько потоков GPU параллельно (то есть действительно запускает ядро один раз на волновой фронт, но обрабатывает 64 элемента данных за один раз), является лишь деталью реализации. Избавив программиста от необходимости думать о деталях того, как вычисления векторизованы на любом конкретном графическом процессоре, вычислительные ядра (и графические шейдеры также) можно написать переносимым способом.

От SIMD к SIMT

Термин *«одиночный поток команд, множественные потоки»* (SIMT) был введен, чтобы подчеркнуть тот факт, что графический процессор не просто использует параллелизм SIMD — он также задействует форму вытесняющей многопоточности для временного интервала между волновым фронтом. Кратко рассмотрим, почему так сделано.

Модуль SIMD запускает волновой фронт, применяя каждую инструкцию в шейдерной программе одновременно к 64 элементам данных, по существу, на шаге блокировки. (Мы можем игнорировать то, что для упрощения волновой фронт обрабатывается в подгруппах по 16 элементов.) Однако любая инструкция в программе может в конечном итоге получить доступ к памяти, и это приводит к большой остановке, пока SIMD-модуль ожидает ответа контроллера памяти.

Чтобы заполнить эти большие интервалы задержки, SIMD-блок разделяет временные интервалы между несколькими волновыми фронтами (взятыми из одной шейдерной программы или, возможно, из многих несвязанных шейдерных программ). Всякий раз, когда один волновой фронт останавливается, контекст

блока SIMD переключается на другой волновой фронт, тем самым удерживая блок занятым (пока есть работающие волновые фронты для переключения). Эта стратегия представлена на рис. 4.40.

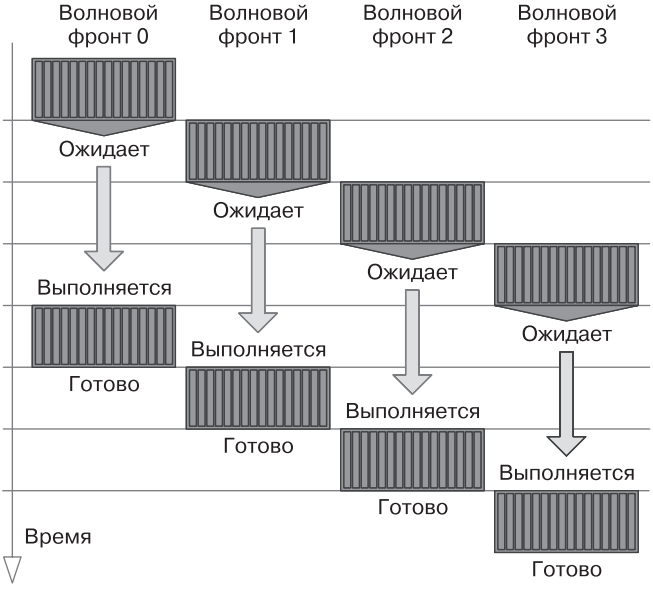


Рис. 4.40. Всякий раз, когда один волновой фронт останавливается, например, из-за задержки доступа к памяти, контекст блока SIMD переключается на другой волновой фронт, чтобы заполнить интервал задержки

Как можно представить, SIMD-блоки в GPU должны выполнять переключение контекста на очень высокой частоте. Во время переключения контекста на ЦП состояние регистров исходящего потока сохраняется в памяти, поэтому они не будут потеряны, а затем состояние регистров входящего потока считывается из памяти в регистры ЦП, чтобы тот мог продолжить выполнение с того места, где остановился. Однако на графическом процессоре требуется слишком много времени для сохранения состояния регистров SIMD каждого волнового фронта при каждом переключении контекста.

Чтобы исключить затраты на сохранение регистров при переключении контекста, каждый блок SIMD содержит очень большой файл регистров. Количество *физических* регистров в нем во много раз превышает количество *логических* регистров, доступных для любого волнового фронта (обычно примерно в десять раз). Это означает, что содержимое логических регистров до десяти волновых фронтов может постоянно поддерживаться в этих физических регистрах. А это, в свою очередь, подразумевает, что переключение контекста может выполняться между волновыми фронтами без сохранения или восстановления каких-либо регистров вообще.

4.11.5. Дополнительная литература

Очевидно, что программирование на GPGPU — огромная тема, и мы только коснулись ее в этой книге. Для получения дополнительной информации о программировании GPGPU и графических шейдеров ознакомьтесь со следующими онлайн-учебниками и ресурсами:

- введение в программирование CUDA — developer.nvidia.com/how-to-cuda-c-cpp;
- учебные ресурсы OpenCL — developer.nvidia.com/opencl;
- руководство по программированию HLSL и справочное руководство — msdn.microsoft.com/en-us/library/bb509561(v=VS.85).aspx;
- введение в язык шейдеров OpenGL — www.khronos.opr/OpenGL/wiki/OpenGL_Shading_Language;
- техническая документация по архитектуре AMD Radeon ™ GCN — www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

5 3D-математика для игр

Игра представляет собой математическую модель виртуального мира, симулируемую в реальном времени на каком-то компьютере. Поэтому математика пронизывает все, что бы мы ни делали в игровой индустрии. Программисты игр применяют практически все разделы математики, от тригонометрии до алгебры, от статистики до арифметики. Но, безусловно, наиболее распространенным видом математики, который вы будете использовать как программист игр, является трехмерная векторная и матричная математика, то есть трехмерная *линейная алгебра*.

Эта область математики очень обширна, поэтому мы не надеемся погрузиться в нее достаточно глубоко в одной главе. Вместо этого я попытаюсь дать обзор математических инструментов, необходимых типичному разработчику игр. По ходу предложу несколько советов и подсказок, которые помогут вам закрепить все довольно запутанные понятия и правила. Я настоятельно рекомендую книгу Эрика Ленгеля [32] по этой теме, где отлично всесторонне освещена трехмерная математика в играх. Глава 3 книги Кристера Эриксона [14] об обнаружении коллизий в реальном времени — также отличный ресурс.

5.1. Решение 3D-задач в 2D

Многие математические операции, о которых мы узнаем в следующей главе, одинаково хорошо работают как в 2D, так и в 3D. Это очень здорово, так как означает, что *иногда* задачу с трехмерным вектором можно решить, представив и нарисовав его в 2D (что значительно проще!). К сожалению, равнозначность между 2D и 3D действует не всегда. Некоторые операции, такие как векторное произведение, определены только в трехмерном пространстве, и при этом в отдельных задачах можно разобраться, только рассмотрев все три измерения. Тем не менее почти никогда не повредит сначала подумать об упрощенной двухмерной версии рассматриваемой задачи. Как только вы найдете решение в двухмерном пространстве, сможете подумать над тем, как эта проблема распространяется на три измерения. В некоторых случаях вы с радостью обнаружите, что результат в 2D работает и в 3D. В других случаях сможете найти систему координат, в которой задача *является* двухмерной. В этой книге мы будем использовать двухмерные диаграммы везде, где между 2D и 3D нет существенных различий.

5.2. Точки и векторы

Большинство современных 3D-игр состоят из трехмерных объектов в виртуальном мире. Игровому движку необходимо отслеживать положение, ориентацию и масштаб всех этих объектов, анимировать их в игровом мире и преобразовывать в экранное пространство для последующего отображения на экране. В играх трехмерные объекты почти всегда состоят из треугольников, вершины которых представлены точками. Итак, прежде чем научиться представлять целые объекты в игровом движке, давайте посмотрим на точку и ее близкого родственника — вектор.

5.2.1. Точки и декартовы координаты

Технически *точка* — это местоположение в n-мерном пространстве. (В играх n обычно равно 2 или 3.) Декартова система координат, безусловно, самая распространенная система координат в играх. Она использует две или три взаимно перпендикулярные оси для определения положения в 2D- или 3D-пространстве. Таким образом, точка **P** представлена парой или тройкой действительных чисел P_x, P_y или P_x, P_y, P_z (рис. 5.1).

Разумеется, декартова система координат не единственный наш выбор. Есть другие часто применяемые системы.

- *Цилиндрические координаты*. В этой системе используются вертикальная ось H — высота, радиальная ось R, исходящая из вертикали, и угол отклонения θ. В цилиндрической системе координат точка **P** представляется тройкой чисел P_h, P_r, P_θ (рис. 5.2).
- *Сферические координаты*. Эта система использует угол наклона ϕ, угол отклонения по оси θ и радиальное измерение r. Таким образом, точки представляются тройкой чисел P_r, P_ϕ, P_θ (рис. 5.3).

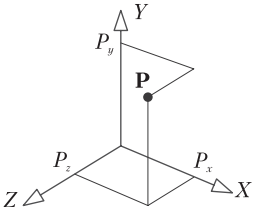


Рис. 5.1. Точка, представленная в декартовой системе координат

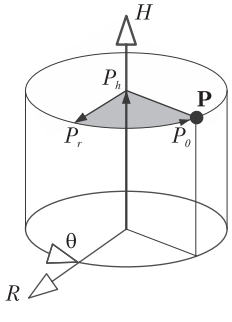


Рис. 5.2. Точка, представленная в цилиндрической системе координат

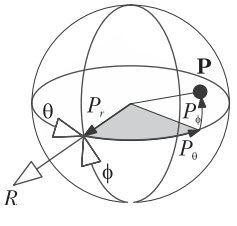


Рис. 5.3. Точка, представленная в сферической системе координат

Декартовы координаты — самая широко используемая система координат в игровом программировании. Однако всегда старайтесь выбирать систему координат, которая наилучшим образом соответствует текущей задаче. Например, в игре *Crank the Weasel* от Midway Home Entertainment главный герой Крэнк бегает по городу в стиле ар-деко, собирая добычу. Я хотел сделать так, чтобы собираемые предметы кружились вокруг тела Крэнка по спирали, постепенно приближаясь к нему, после чего исчезали. Я представил положение добычи в цилиндрической системе координат относительно текущего положения Крэнка. Чтобы реализовать спиральную анимацию, задал предметам постоянную угловую скорость θ, небольшую постоянную линейную скорость, направленную внутрь, вдоль радиальной оси *R*, и совсем небольшую постоянную линейную скорость, направленную вверх, вдоль оси *H*, чтобы добыча постепенно поднималась до уровня карманов брюк Крэнка. Эта чрезвычайно простая анимация выглядела отлично, и ее гораздо проще моделировать с помощью цилиндрических координат, чем было бы при использовании декартовой системы.

5.2.2. Левосторонние и правосторонние системы координат

В трехмерных декартовых координатах есть два варианта расположения трех взаимно перпендикулярных осей: правосторонний и левосторонний. В правосторонней системе координат, если большой палец правой руки принять за положительное направление оси *X*, указательный — за положительное направление оси *Y*, а средний — за положительное направление оси *Z*, образуется правая система координат. Аналогичными пальцами левой руки образуется левая система координат.

Единственная разница между левосторонней и правосторонней системами координат — это направление, в котором указывает одна из трех осей. Например, если ось *Y* направлена вверх, а *X* — вправо, то *Z* направлена к нам (из страницы) в правосторонней системе и от нас (в страницу) — в левосторонней. Левосторонние и правосторонние декартовы системы координат изображены на рис. 5.4.

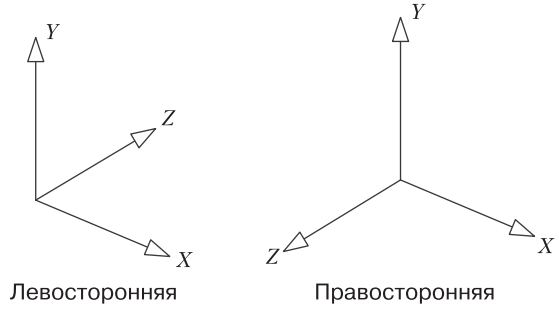


Рис. 5.4. Лево- и правосторонние декартовы системы координат

Конвертировать из левосторонних в правосторонние координаты и наоборот несложно. Мы просто меняем направление любой оси, оставляя две другие как есть. Важно помнить, что математические правила не меняются в левосторонних и правосторонних системах координат, меняется только *интерпретация* чисел — мысленное представление о том, как числа отображаются в трехмерном пространстве. Левосторонние и правосторонние условия отображения применяются только к визуализации, а не к математическим правилам. (На самом деле сторона отображения имеет значение в ходе работы с векторными произведениями в физическом моделировании, потому что векторное произведение не является вектором — это специальный математический объект, известный как *псевдовектор*. Мы обсудим псевдовекторы немного подробнее в подразделе 5.2.4.)

Соотношение между числовым и визуальным представлением мы, как математики и программисты, можем выбирать на свое усмотрение. Можем направить ось Y вверх, Z — вперед, а X — влево или вправо. Или же направить вверх ось Z. Или вместо нее ось X будет указывать вверх или вниз. Самое главное — выбрать соотношение, а затем придерживаться его.

При этом некоторые условия отображения для определенных приложений обычно подходят лучше других. Например, программисты трехмерной графики обычно работают с левосторонней системой координат: ось Y направлена вверх, X — вправо и положительные координаты оси Z — от зрителя, то есть туда, куда указывает виртуальная камера. Когда 3D-графика отображается на 2D-экране с использованием этой системы координат, увеличение координат Z соответствует увеличению глубины в сцене, то есть увеличению расстояния от виртуальной камеры. Как мы увидим в последующих главах, это именно то, что требуется при задействовании z-буферизации для окклюзии глубины.

5.2.3. Векторы

Вектор — это величина, имеющая *длину* и *направление* в n-мерном пространстве. Вектор может быть представлен как *направленный отрезок* между двумя граничными точками, которые называются *началом* и *концом* вектора. Сравните это со *скаляром* (обычным вещественным числом), который имеет размер, но не имеет направления. Обычно скаляры выделены курсивом (например, v), а векторы — жирным шрифтом (например, $\mathbf{v}$).

Трехмерный вектор может быть представлен тремя скалярами: x, y, z — так же, как и точка. Различие между точками и векторами на самом деле довольно тонкое. Технически вектор — это просто смещение *относительно* известной точки. Вектор можно перемещать в любое место в трехмерном пространстве — если его длина и направление не меняются, это один и тот же вектор.

Вектор можно использовать для задания точки при условии, что мы совместим его начало с началом системы координат. Такой вектор называется *радиус-вектором*. Для своих целей мы можем интерпретировать любую тройку скаляров как точку

или как вектор при условии, что будем помнить: *радиус-вектор* ограничен тем, что его начало находится в начале координат. Это означает, что с математической точки зрения точки и векторы рассматриваются слегка по-разному. Можно сказать, что точки *абсолютны*, а векторы *относительны*.

Подавляющее большинство программистов игр пользуются термином «вектор» для обозначения как точек (радиус-векторы), так и векторов линейной алгебры (векторы направления). Большинство трехмерных математических библиотек применяют термин «вектор» таким же образом. В этой книге станем использовать термин «вектор направления» или просто «направление», когда нужно будет обозначить разницу. Всегда четко помните о различии между точками и направлениями, даже если ваша математическая библиотека его не делает. Как мы увидим в подразделе 5.3.6, направления не могут обрабатываться так же, как точки, при их преобразовании в однородные координаты для проведения манипуляций с матрицами 4×4, поэтому путаница между двумя типами векторов может привести к ошибкам в вашем коде.

Декартовы базисные векторы

Зачастую полезно определить три *ортогональных единичных вектора* (то есть взаимно перпендикулярных вектора, каждый из которых имеет длину 1), соответствующих трем основным декартовым осям. Единичный вектор вдоль оси X обычно называется **i**, вдоль оси Y — **j**, а вдоль оси Z — **k**. Векторы **i**, **j** и **k** иногда называют декартовыми *базисными векторами*.

Любые точка или вектор могут быть выражены в виде суммы скаляров (действительных чисел), умноженных на единичные базисные векторы, например:

$$(5, 3, -2) = 5\mathbf{i} + 3\mathbf{j} - 2\mathbf{k}.$$

5.2.4. Векторные операции

Большинство математических операций, которые вы можете выполнить над скалярами, можно применить и к векторам. Есть и некоторые другие операции, которые применяются только к векторам.

Умножение на скаляр

Умножение вектора **a** на скаляр s выполняется умножением отдельных компонентов **a** на s:

$$s\mathbf{a} = (sa_x, sa_y, sa_z).$$

Умножение на скаляр дает эффект масштабирования длины вектора, оставляя его направление неизменным (рис. 5.5). Умножение на -1 изменяет направление вектора — начало становится концом и наоборот.

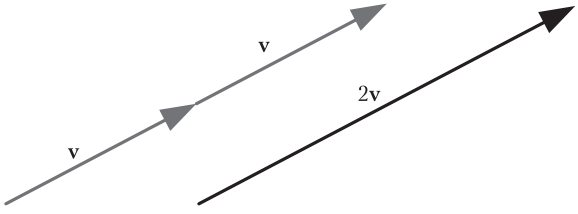


Рис. 5.5. Умножение вектора на скаляр 2

Коэффициент масштабирования может быть различным для каждой оси. Это называется *неоднородным масштабированием* и может быть представлено как *покомпонентное произведение* вектора масштабирования **s** и рассматриваемого вектора, которое обозначим оператором ⊗. Этот особый вид произведения между двумя векторами известен как *произведение Адамара*. Оно редко используется в игровой индустрии — фактически неоднородное масштабирование является его единственным распространенным применением в играх:

$$\mathbf{s} \otimes \mathbf{a} = \left(s_x a_x, s_y a_y, s_z a_z \right). \tag{5.1}$$

Как мы увидим в подразделе 5.3.7, вектор масштабирования **s** — это просто удобный способ представления диагональной матрицы масштабирования 3×3 **S**. Таким образом, уравнение (5.1) можно написать иначе:

$$\mathbf{aS} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = \begin{bmatrix} s_x a_x & s_y a_y & s_z a_z \end{bmatrix}.$$

Подробнее рассмотрим матрицы в разделе 5.3.

Сложение и вычитание

Сложение векторов **a** и **b** определяется как вектор, компоненты которого являются суммами *компонентов* **a** и **b**. Это можно изобразить, расположив конец вектора **a** в начале вектора **b,** тогда сумма будет вектором от начала **a** до конца **b** (рис. 5.6):

$$\mathbf{a} + \mathbf{b} = \left[\left(a_x + b_x \right), \left(a_y + b_y \right), \left(a_z + b_z \right) \right].$$

Вычитание векторов **a** – **b** — это не что иное, как сложение векторов **a** и –**b**, то есть результат умножения **b** на –1, меняющий его направление. Это соответствует вектору, компоненты которого являются разностью между компонентами **a** и **b**:

$$\mathbf{a} - \mathbf{b} = \left[\left(a_x - b_x \right), \left(a_y - b_y \right), \left(a_z - b_z \right) \right].$$

Сложение и вычитание векторов изображено на рис. 5.6.

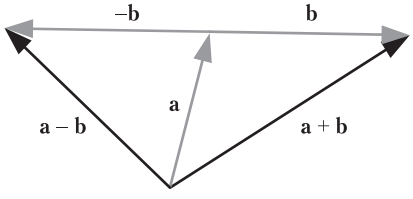


Рис. 5.6. Сложение и вычитание векторов

Сложение и вычитание точек и направлений. Векторы направления можно свободно складывать и вычитать. А вот точки не могут быть добавлены друг к другу — вы можете только добавить вектор направления к точке, результатом чего будет другая точка. Или взять разницу между двумя точками, в результате чего получится вектор направления. Вот эти операции:

- направление + направление = направление;
- направление – направление = направление;
- точка + направление = точка;
- точка – точка = направление;
- точка + точка = *бессмыслица*.

Величина

Размером вектора является скаляр, представляющий длину вектора так, как она была бы измерена в 2D- или 3D-пространстве. Обозначается вертикальными линиями справа и слева от символа вектора, выделенного жирным шрифтом. Для вычисления длины вектора можно использовать теорему Пифагора (рис. 5.7):

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}.$$

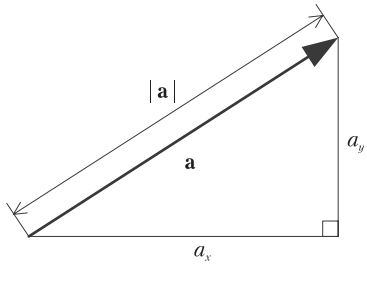


Рис. 5.7. Длина вектора (показана в 2D для упрощения иллюстрации)

Применение векторных операций

Хотите верьте, хотите нет, но мы уже можем решать реальные игровые задачи — с помощью только что изученных векторных операций. При этом можем использовать операции сложения, вычитания, умножения и масштабирования, чтобы генерировать новые данные из имеющихся. Например, если есть вектор текущей позиции персонажа, управляемого ИИ, $\mathbf{P}_1$ и вектор $\mathbf{v}$, представляющий его текущую скорость, мы можем найти его положение в следующем кадре $\mathbf{P}_2$, умножив вектор скорости на интервал времени Δt, а затем добавив его к текущей позиции. Как показано на рис. 5.8, получаем уравнение вектора $\mathbf{P}_2 = \mathbf{P}_1 + \mathbf{v}\Delta t$. (Это *явный метод Эйлера* — он работает, только когда скорость постоянна, но вы поняли суть.)

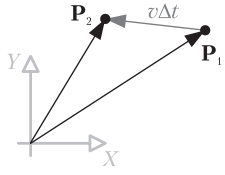


Рис. 5.8. Простое сложение векторов может использоваться для определения позиции персонажа в следующем кадре, учитывая его позицию и скорость в текущем кадре

Или другой пример: у нас есть две сферы и мы хотим знать, пересекаются ли они. Учитывая, что известны центральные точки сфер, $\mathbf{C}_1$ и $\mathbf{C}_2$, можем найти вектор направления между ними, просто определив их разность: $\mathbf{d} = \mathbf{C}_2 - \mathbf{C}_1$. Длина этого вектора $d = |\mathbf{d}|$ определяет, как далеко друг от друга расположены центры сфер. Если это расстояние меньше суммы радиусов сфер, они пересекаются, в противном случае — нет (рис. 5.9).

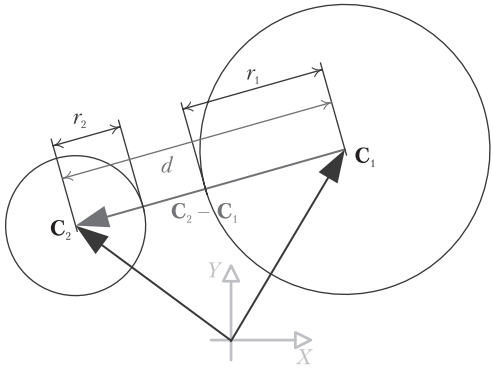


Рис. 5.9. Проверка пересечения двух сфер включает в себя только операции вычитания векторов, масштабирования длин векторов и сравнения чисел с плавающей точкой

На большинстве компьютеров рассчитывать квадратные корни слишком затратно, поэтому разработчики игр всегда должны использовать квадратичную длину тогда, когда это допустимо:

$$|\mathbf{a}|^2 = \left(a_x^2 + a_y^2 + a_z^2\right).$$

Применение квадратичной величины подходит при сравнении относительных длин двух векторов («вектор **a** длиннее, чем вектор **b**?») или величины вектора

с некоторой другой (квадратичной) скалярной величиной. Таким образом, проверяя, пересекаются ли сферы, мы должны вычислить $d^2 = |\mathbf{d}|^2$ и сравнить результат с квадратом суммы радиусов $(r_1 + r_2)^2$, чтобы обеспечить максимальную скорость вычислений. При написании высокопроизводительного программного обеспечения никогда не вычисляйте квадратный корень, когда это не нужно!

Нормализация и единичные векторы

Единичный вектор — это вектор, длина которого равна 1. Единичные векторы очень полезны в трехмерной математике и при программировании игр, почему — увидим в дальнейшем.

Произвольный вектор **v** длиной $v = |\mathbf{v}|$ можно преобразовать в единичный вектор **u**, который имеет то же направление, что и **v**, но единичную длину. Для этого просто умножим **v** на величину, обратную его длине:

$$\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{v}\mathbf{v}.$$

Это называется *нормализацией*.

Векторы нормали

Вектор называется *нормальным* по отношению к поверхности, если он *перпендикулярен* этой поверхности. Векторы нормали очень полезны в играх и компьютерной графике. Например, *плоскость* может быть определена точкой и вектором нормали. А в трехмерной графике нормальные векторы широко используются при расчете освещения для определения ориентации поверхностей по отношению к направлению падающих на них лучей света.

Нормальные векторы обычно (но не обязательно) имеют единичную длину. Старайтесь не путать термины «нормализованный вектор» и «нормальный вектор». Нормализованный вектор — это любой вектор единичной длины. Нормальный вектор — это любой вектор, перпендикулярный поверхности, независимо от длины.

Скалярное произведение и проекция

Векторы можно умножать, но, в отличие от умножения скаляров, существует множество видов умножения векторов. В программировании игр мы чаще всего работаем с двумя видами:

- со *скалярным произведением* (иногда называемым внутренним произведением);
- *векторным произведением* (иногда называемым внешним произведением).

Скалярное произведение двух векторов дает скаляр, что определяется сложением произведений отдельных компонентов двух векторов:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z = d.$$

Скалярное произведение можно записать также как произведение длин двух векторов и косинуса угла между ними:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta.$$

Скалярное произведение является *коммутативным* (то есть порядок двух векторов может быть обратным) и *дистрибутивным* по сложению:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a};$$
$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}.$$

Скалярное произведение совмещается с умножением на скаляр следующим образом:

$$s\mathbf{a} \cdot \mathbf{b} = \mathbf{a} \cdot s\mathbf{b} = s(\mathbf{a} \cdot \mathbf{b}).$$

Проекция вектора. Если **u** — единичный вектор (|**u**| = 1), то скалярное произведение **a** · **u** представляет длину *проекции* вектора **a** на бесконечную линию, определяемую направлением **u** (рис. 5.10). Эта концепция проекции одинаково хорошо работает как в 2D, так и в 3D и очень полезна для решения широкого спектра трехмерных задач.

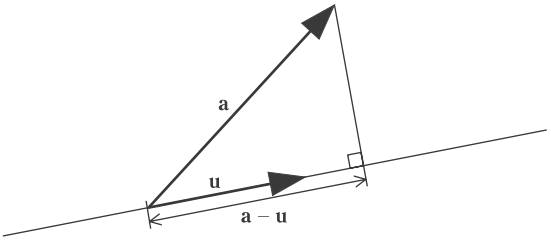


Рис. 5.10. Векторная проекция с использованием скалярного произведения

Величина как скалярное произведение. Квадратичную величину вектора можно найти, взяв скалярное произведение этого вектора на самого себя. Его длина затем легко определяется взятием квадратного корня:

$$|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a};$$
$$|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}}.$$

Это работает, потому что косинус нулевых градусов равен 1, поэтому $|\mathbf{a}||\mathbf{a}|\cos\theta = |\mathbf{a}||\mathbf{a}| = |\mathbf{a}|^2$.

Тесты на основе скалярного произведения. Скалярные произведения отлично подходят для проверки коллинеарности и перпендикулярности двух векторов, а также их направлений: указывают ли они в примерно одинаковом или примерно противоположном направлениях. Для любых двух произвольных векторов **a** и **b** игровые разработчики часто используют следующие тесты (рис. 5.11).

- *Коллинеарны.* **a** · **b** = |**a**||**b**| = *ab*, то есть угол между ними равен 0. Это скалярное произведение равно +1, когда **a** и **b** являются *единичными векторами*.

- *Коллинеарны, но противоположно направлены.* $\mathbf{a} \cdot \mathbf{b} = -ab$, то есть угол между ними составляет 180°. Это скалярное произведение равно –1, когда **a** и **b** являются единичными векторами.
- *Перпендикулярны.* $\mathbf{a} \cdot \mathbf{b} = 0$, то есть угол между ними составляет 90°.
- *Сонаправлены.* $\mathbf{a} \cdot \mathbf{b} > 0$, то есть угол между ними меньше 90°.
- *Противоположно направлены.* $\mathbf{a} \cdot \mathbf{b} < 0$, то есть угол между ними больше 90°.

Другие применения скалярного произведения. Скалярное произведение может быть использовано для самых разнообразных задач в программировании игр. Допустим, мы хотим выяснить, находится враг перед персонажем игрока или позади него. Можем найти вектор из позиции игрока **P** в позицию противника **E** простым вычитанием векторов (**v** = **E** – **P**). Предположим, у нас есть вектор **f**, указывающий в направлении, в котором *смотрит* игрок. (Как мы увидим в подразделе 5.3.10, вектор **f** может быть получен непосредственно из матрицы «модель — мир» игрока.) С помощью скалярного произведения $d = \mathbf{v} \cdot \mathbf{f}$ можно проверить, находится враг впереди или позади игрока: оно будет положительным, когда враг впереди, и отрицательным, когда позади.

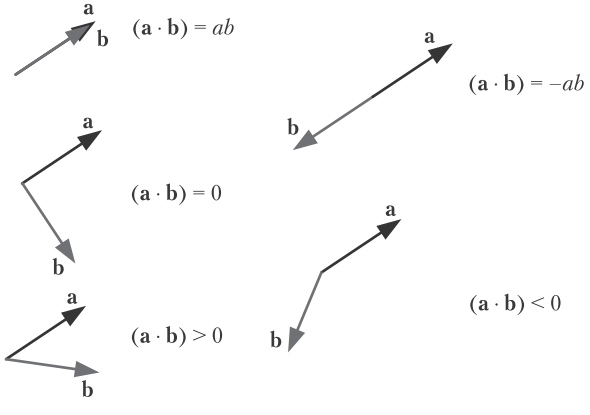


Рис. 5.11. Распространенные тесты на основе скалярного произведения

Скалярное произведение можно использовать также для определения высоты точки над плоскостью или под ней (например, при написании игры о посадке на Луну). Мы можем определить плоскость двумя векторными величинами: точкой **Q**, лежащей в любом месте на плоскости, и единичным вектором **n**, перпендикулярным (то есть нормальным) к плоскости. Чтобы найти высоту *h* точки **P** над плоскостью, сначала вычисляем вектор из любой точки на плоскости (**Q** отлично для этого подойдет) до рассматриваемой точки **P**. Итак, мы имеем **v** = **P** – **Q**. Скалярным произведением вектора **v** и нормального вектора единичной длины **n** является просто проекция **v** на прямую, определенную **n**. Но это именно та высота, которую мы ищем. Таким образом (рис. 5.12):

$$h = \mathbf{v} \cdot \mathbf{n} = (\mathbf{P} - \mathbf{Q}) \cdot \mathbf{n}. \tag{5.2}$$

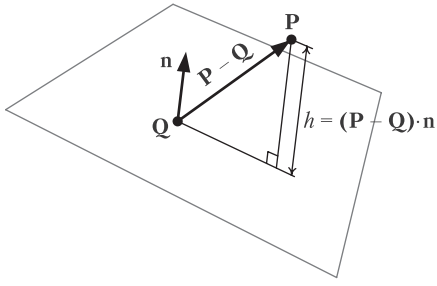


Рис. 5.12. Скалярное произведение может быть использовано для определения высоты точки над плоскостью или под ней

Векторное произведение

Векторное произведение двух векторов, известное также как *внешнее произведение*, дает другой вектор, *перпендикулярный* двум перемножаемым векторам (рис. 5.13). Операция векторного произведения выполняется только в трех измерениях:

$$\mathbf{a}\times\mathbf{b}=\left[\left(a_y b_z-a_z b_y\right),\left(a_z b_x-a_x b_z\right),\left(a_x b_y-a_y b_x\right)\right]=$$
$$=\left(a_y b_z-a_z b_y\right)\mathbf{i}+\left(a_z b_x-a_x b_z\right)\mathbf{j}+\left(a_x b_y-a_y b_x\right)\mathbf{k}.$$

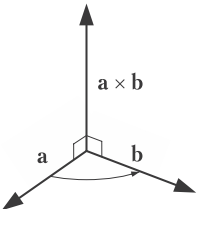


Рис. 5.13. Векторное произведение векторов a и b (правостороннее)

Длина векторного произведения. Длина векторного произведения является произведением длин двух векторов и синуса угла между ними (это похоже на определение скалярного произведения, только косинус заменяется на синус):

$$\left|\mathbf{a}\times\mathbf{b}\right|=\left|\mathbf{a}\right|\left|\mathbf{b}\right|\sin\theta.$$

Векторное произведение $\mathbf{a}\times\mathbf{b}$ равно площади параллелограмма со сторонами **a** и **b** (рис. 5.14). Поскольку треугольник является половиной параллелограмма, то площадь треугольника, вершины которого определяются радиус-векторами $\mathbf{V}_1$, $\mathbf{V}_2$ и $\mathbf{V}_3$, можно получить как половину длины векторного произведения любых двух его сторон:

$$A_{\text{triangle}}=\frac{1}{2}\left|\left(\mathbf{V}_2-\mathbf{V}_1\right)\times\left(\mathbf{V}_3-\mathbf{V}_1\right)\right|.$$

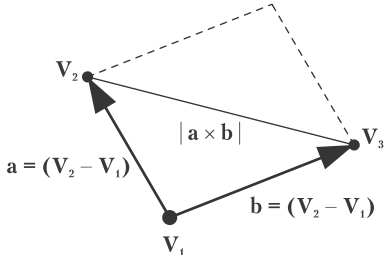


Рис. 5.14. Площадь параллелограмма, выраженная как величина векторного произведения

Направление векторного произведения. При правосторонней системе координат вы можете применить *правило правой руки*, чтобы определить направление векторного произведения. Расположите пальцы так, чтобы они указывали в направлении, в котором вы бы повернули вектор **a**, чтобы переместить его к началу вектора **b**, тогда направлением векторного произведения $(\mathbf{a} \times \mathbf{b})$ будет направление вашего большого пальца.

Обратите внимание на то, что при использовании левосторонней системы координат направление векторного произведения определяется *правилом левой руки*. Это означает, что направление векторного произведения меняется в зависимости от выбора системы координат. Поначалу это может показаться странным, но помните: сторона отображения системы координат не влияет на выполняемые математические вычисления, она лишь меняет наше представление о том, как числа выглядят в трехмерном пространстве. При преобразовании из правосторонней системы в левостороннюю или наоборот числовые представления всех точек и векторов остаются неизменными, но одна ось меняет направление. Поэтому представление всех элементов отражается вдоль этой перевернутой оси. Таким образом, если векторное произведение совместится с осью, которую мы переворачиваем (например, осью *Z*), оно также должно изменить направление. Если бы этого не происходило, то математическое определение векторного произведения пришлось бы изменить так, чтобы его координата *z* векторного произведения оказывалась отрицательной в новой системе координат. Не забивайте себе этим голову. Просто помните: при отображении векторного произведения используйте правило правой руки в правосторонней системе координат и правило левой руки — в левосторонней.

Свойства векторного произведения. Векторное произведение *не является коммутативным* (то есть порядок имеет значение):

$$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}.$$

Тем не менее оно *антикоммутативно*:

$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a}).$$

Векторное произведение является дистрибутивным относительно сложения:

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) + (\mathbf{a} \times \mathbf{c}).$$

И оно совмещается с умножением на скаляр следующим образом:

$$(s\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (s\mathbf{b}) = s(\mathbf{a} \times \mathbf{b}).$$

Декартовы базисные векторы связаны между собой следующим образом:

$$\mathbf{i} \times \mathbf{j} = -(\mathbf{j} \times \mathbf{i}) = \mathbf{k};$$
$$\mathbf{j} \times \mathbf{k} = -(\mathbf{k} \times \mathbf{j}) = \mathbf{i};$$
$$\mathbf{k} \times \mathbf{i} = -(\mathbf{i} \times \mathbf{k}) = \mathbf{j}.$$

Эти три векторных произведения определяют направление *положительных вращений* вокруг декартовых осей. Положительные вращения направлены от *X* к *Y* (вокруг *Z*), от *Y* к *Z* (вокруг *X*) и от *Z* к *X* (вокруг *Y*). Обратите внимание на то, что вращение вокруг оси *Y* направлено в обратном алфавитном порядке, поскольку оно идет от *Z* к *X*, а не от *X* к *Z*. Как мы увидим в дальнейшем, это подсказывает, почему матрица для вращения вокруг оси *Y* выглядит инвертированной по сравнению с матрицами для вращения вокруг осей *X* и *Z*.

Применение векторного произведения. Векторное произведение имеет ряд применений в играх. Одним из его наиболее распространенных использований является нахождение вектора, перпендикулярного двум другим векторам. Как будет показано в подразделе 5.3.10, если мы знаем локальные единичные базисные векторы объекта ($\mathbf{i}_{local}$, $\mathbf{j}_{local}$ и $\mathbf{k}_{local}$), то легко можем найти матрицу, представляющую ориентацию объекта. Предположим, что все, что мы знаем, — это вектор $\mathbf{k}_{local}$ объекта, то есть направление, в котором смотрит объект. Если мы предположим, что объект не вращается вокруг $\mathbf{k}_{local}$, то можем найти $\mathbf{i}_{local}$, взяв векторное произведение между $\mathbf{k}_{local}$ (который уже известен) и направленным вверх в виртуальном мире вектором $\mathbf{j}_{world}$, равным [0 1 0]. Мы это делаем следующим образом: $\mathbf{i}_{local}$ = normalize ($\mathbf{j}_{world} \times \mathbf{k}_{local}$). После чего можем найти $\mathbf{j}_{local}$, просто совместив $\mathbf{i}_{local}$ и $\mathbf{k}_{local}$ следующим образом: $\mathbf{j}_{local} = \mathbf{k}_{local} \times \mathbf{i}_{local}$.

Очень похожая методика может быть использована для нахождения единичного вектора, нормального к поверхности треугольника или другой плоскости. Зная три точки на плоскости, $\mathbf{P}_1$, $\mathbf{P}_2$ и $\mathbf{P}_3$, вектор нормали просто $\mathbf{n}$ = normalize $((\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1))$.

Векторные произведения применяют также в физических симуляциях. Когда сила прилагается к объекту, она вызывает вращательное движение тогда и только тогда, когда действует не на центр. Эта вращающая сила называется крутящим моментом и рассчитывается следующим образом. При заданной силе $\mathbf{F}$ и векторе $\mathbf{r}$ от центра массы до точки, к которой прилагается сила, крутящий момент равен $\mathbf{N} = \mathbf{r} \times \mathbf{F}$.

Псевдовекторы и внешняя алгебра

В подразделе 5.2.2 мы упомянули, что векторное произведение на самом деле дает не вектор — оно создает особый вид математического объекта, известный как псевдовектор. Разница между вектором и псевдовектором довольно тонкая. По сути, вы

вообще не сможете различить их при выполнении преобразований, с которыми мы обычно сталкиваемся в программировании игр, — перемещения, вращения и масштабирования. Только когда вы отражаете систему координат (что происходит при переходе от левосторонней системы координат к правосторонней), особая природа псевдовекторов становится очевидной. При отражении вектор превращается в свое зеркальное отображение, что вполне ожидаемо. Но когда отражается псевдовектор, он превращается в зеркальное отображение, при этом меняя направление.

Положения и все их производные (линейная скорость, ускорение, рывок) представлены истинными векторами, известными также как полярные или контравариантные векторы. Угловые скорости и магнитные поля представлены псевдовекторами, известными также как осевые векторы, ковариантные векторы, бивекторы или 2-векторы. Нормаль поверхности треугольника, которая рассчитывается с помощью векторного произведения, также является псевдовектором.

Интересно отметить, что векторное произведение $\mathbf{A} \times \mathbf{B}$, скалярное тройное произведение $\mathbf{A} \cdot (\mathbf{B} \times \mathbf{C})$ и определитель матрицы взаимосвязаны, и в основе всего лежат псевдовекторы. Математики придумали набор алгебраических правил, называемых внешней алгеброй, или алгеброй Грассмана, которые описывают работу векторов и псевдовекторов и позволяют вычислять площадь параллелограмма (в 2D), объем параллелепипеда (в 3D) и прочего в более высоких измерениях.

Не будет здесь вдаваться в детали, скажу лишь, что основная идея алгебры Грассмана состоит в том, чтобы ввести специальный вид векторного произведения, называемый внешним произведением и обозначенный $\mathbf{A} \wedge \mathbf{B}$. Парное внешнее произведение дает псевдовектор и эквивалентно векторному произведению, которое также представляет площадь параллелограмма, образованного двумя векторами, где знак показывает, происходит вращение от **A** к **B** или наоборот. Выполнение двух внешних произведений подряд, $\mathbf{A} \wedge \mathbf{B} \wedge \mathbf{C}$, эквивалентно скалярному тройному произведению $\mathbf{A} \cdot (\mathbf{B} \times \mathbf{C})$ и дает другой необычный математический объект, известный как псевдоскаляр (также тривектор, или 3-вектор), который можно интерпретировать как объем параллелепипеда, образованного тремя векторами (рис. 5.15). Это распространяется и на более высокие измерения.

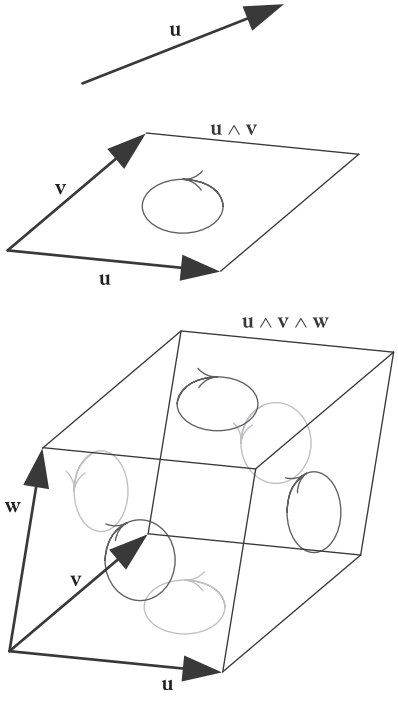


Рис. 5.15. Во внешней алгебре (алгебре Грассмана) одно внешнее произведение дает псевдовектор, или бивектор, а два внешних произведения — псевдоскаляр, или тривектор

Что все это значит для нас, программистов? Не слишком много. Все, что нам действительно нужно помнить, — это то, что некоторые векторы в коде на самом деле являются псевдовекторами и мы можем преобразовать их должным образом, например, при изменении стороны отображения координат. Конечно, вы можете произвести впечатление на своих друзей, рассказывая о внешних алгебрах и внешних произведениях и объясняя, что векторные произведения на самом деле не являются векторами. Благодаря этому вы, может быть, будете круто выглядеть при следующем выходе в свет... а может, и нет.

Для получения дополнительной информации зайдите на en.wikipedia.org/wiki/Pseudovector, http://en.wikipedia.org/wiki/Exterior_algebra, https://ru.wikipedia.org/wiki/Внешняя_алгебра и htwww.terathon.com/gdc12_lengyel.pdf.

5.2.5. Линейная интерполяция точек и векторов

В играх часто нужно найти вектор, который находится посередине двух известных векторов. Например, если мы хотим красиво анимировать перемещение объекта из точки **A** в точку **B** в течение 2 секунд со скоростью 30 кадров в секунду, нужно найти 60 промежуточных позиций между **A** и **B**.

Линейная интерполяция — это простая математическая операция, которая находит промежуточную точку между двумя известными точками. Название этой операции часто сокращается до LERP (linear interpolation). Операция определяется следующим образом, где β варьируется от 0 до 1 включительно:

$$\mathbf{L} = \mathrm{LERP}(\mathbf{A}, \mathbf{B}, \beta) = (1-\beta)\mathbf{A} + \beta\mathbf{B} =$$
$$= \left[(1-\beta)A_x + \beta B_x, (1-\beta)A_y + \beta B_y, (1-\beta)A_z + \beta B_z\right].$$

С геометрической точки зрения **L** = LERP(**A**, **B**, β) — это радиус-вектор точки, которая находится на β процентах пути вдоль отрезка от точки **A** до точки **B** (рис. 5.16). С математической точки зрения функция LERP представляет собой взвешенное среднее двух входных векторов с весами (1 – β) и β. Обратите внимание на то, что веса всегда в сумме образуют 1, что является общим требованием для любого средневзвешенного значения.

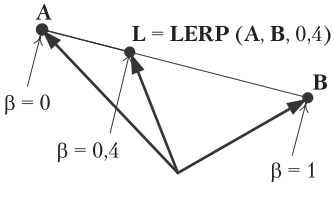


Рис. 5.16. Линейная интерполяция между точками A и B при β = 0,4

5.3. Матрицы

Матрица — это прямоугольный массив $m \times n$ скаляров. Матрицы являются удобным способом представления линейных преобразований, таких как перемещение, вращение и масштабирование.

Матрица **M** обычно записывается в виде сетки скаляров M_{rc}, заключенных в квадратные скобки, где индексы r и c представляют индексы строк и столбцов

соответственно. Например, если **M** является матрицей 3×3, ее можно записать следующим образом:

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}.$$

Мы можем представлять строки и/или столбцы матрицы 3×3 в виде трехмерных векторов. Когда все вектор-строки и вектор-столбцы матрицы 3×3 имеют единичную длину, мы называем ее *специальной ортогональной матрицей*. Также она известна как *изотропная* или *ортонормированная* матрица. Такие матрицы представляют собой чистые вращения.

При определенных ограничениях матрица 4×4 может представлять произвольные трехмерные *преобразования*, включая *перемещение*, *вращение* и *изменение в масштабе*. Это так называемые *матрицы преобразования*, они наиболее полезны нам как игровым инженерам. Преобразования, представленные матрицей, применяются к точке или вектору посредством умножения матриц. Далее мы рассмотрим, как это работает.

Аффинное преобразование — это матрица преобразования 4×4, которая сохраняет параллельность линий и относительные отношения расстояний, но может не сохранять абсолютные длины и углы. Аффинное преобразование — это любая комбинация операций вращения, трансляции, масштабирования и/или сдвига.

5.3.1. Умножение матриц

Произведение **P** двух матриц **A** и **B** записывается как **P** = **AB**. Если **A** и **B** — матрицы преобразования, то произведение **P** — это тоже матрица преобразования, которая выполняет *оба* исходных преобразования. Например, если **A** — это матрица масштабирования, а **B** — вращения, матрица **P** будет *одновременно* масштабировать и вращать точки или векторы, к которым она применяется. Это особенно полезно при программировании игр, потому что можно предварительно рассчитать одну матрицу, которая выполняет целую последовательность преобразований, а затем эффективно применить их все к большому количеству векторов.

Чтобы вычислить произведение матриц, просто берем скалярные произведения между строками $n_A \times m_A$ матрицы **A** и столбцами $n_B \times m_B$ матрицы **B**. Каждое скалярное произведение становится одним компонентом итоговой матрицы **P**. Две матрицы могут быть перемножены, если их внутренние размеры равны, то есть $m_A = n_B$. Например, если **A** и **B** являются матрицами 3×3, то **P** = **AB** может быть выражено следующим образом:

$$\mathbf{P} = \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{\text{row1}} \cdot \mathbf{B}_{\text{col1}} & \mathbf{A}_{\text{row1}} \cdot \mathbf{B}_{\text{col2}} & \mathbf{A}_{\text{row1}} \cdot \mathbf{B}_{\text{col3}} \\ \mathbf{A}_{\text{row2}} \cdot \mathbf{B}_{\text{col1}} & \mathbf{A}_{\text{row2}} \cdot \mathbf{B}_{\text{col2}} & \mathbf{A}_{\text{row2}} \cdot \mathbf{B}_{\text{col3}} \\ \mathbf{A}_{\text{row3}} \cdot \mathbf{B}_{\text{col1}} & \mathbf{A}_{\text{row3}} \cdot \mathbf{B}_{\text{col2}} & \mathbf{A}_{\text{row3}} \cdot \mathbf{B}_{\text{col3}} \end{bmatrix}.$$

Умножение матриц не является коммутативным. Другими словами, порядок, в котором выполняется умножение матриц, имеет значение:

$$\mathbf{AB} \neq \mathbf{BA}.$$

Почему именно это важно, мы увидим в подразделе 5.3.2.

Умножение матриц часто называют *конкатенацией*, поскольку произведение n матриц преобразований представляет собой матрицу, которая присоединяет друг к другу *(concatenate)* или собирает в цепочку исходную последовательность преобразований в порядке умножения матриц.

5.3.2. Представление точек и векторов в виде матриц

Точки и векторы могут быть представлены в виде *матрицы-строки* $(1 \times n)$ или *матрицы-столбца* $(n \times 1)$, где n — размерность пространства, с которым мы работаем (обычно 2 или 3). Например, вектор $\mathbf{v} = (3, 4, -1)$ может быть записан как:

$$\mathbf{v}_1 = \begin{bmatrix} 3 & 4 & -1 \end{bmatrix}$$

или:

$$\mathbf{v}_2 = \begin{bmatrix} 3 \\ 4 \\ -1 \end{bmatrix} = \mathbf{v}_1^{\mathrm{T}}.$$

Здесь верхний индекс «T» представляет собой *транспонирование* матрицы (см. подраздел 5.3.5).

Выбор между вектор-столбцами и вектор-строками произволен, но он влияет на порядок записи умножения матриц. Это происходит потому, что при перемножении двух матриц их внутренние размеры должны быть равны. Поэтому:

- чтобы умножить вектор-строку $1 \times n$ на матрицу $n \times n$, вектор должен появиться *слева* от матрицы $(\mathbf{v}'_{1\times n} = \mathbf{v}_{1\times n}\mathbf{M}_{n\times n})$;
- чтобы умножить матрицу $n \times n$ на вектор-столбец $n \times 1$, вектор должен появиться *справа* от матрицы $(\mathbf{v}'_{n\times 1} = \mathbf{M}_{n\times n}\mathbf{v}_{n\times 1})$.

Если к вектору **v** по очереди применяются матрицы преобразования **A**, **B** и **C**, преобразования читаются *слева направо* при использовании *вектор-строк*, но *справа налево* — при использовании *вектор-столбцов*. Самый простой способ запомнить это — понять, что матрица, *ближайшая* к вектору, применяется первой. Это показано далее:

$$\mathbf{v}' = \Big(\big(\big(\mathbf{v}\mathbf{A}\big)\mathbf{B}\big)\mathbf{C}\Big)$$

(вектор-строки читаются слева направо);

$$\mathbf{v}'^{\mathrm{T}} = \Big(\mathbf{C}^{\mathrm{T}}\Big(\mathbf{B}^{\mathrm{T}}\Big(\mathbf{A}^{\mathrm{T}}\mathbf{v}^{\mathrm{T}}\Big)\Big)\Big)$$

(вектор-столбцы читаются справа налево).

В этой книге будем придерживаться *правила вектор-строк*, потому что порядок преобразований слева направо нам наиболее привычен. Тем не менее не забывайте проверять, какое правило используется вашим игровым движком, а также упоминается в книгах, статьях или на веб-сайтах, которые вы читаете. Обычно это можно определить, посмотрев, где при умножении векторов на матрицы записан вектор: слева (для вектор-строк) или справа (для вектор-столбцов) матрицы. При использовании вектор-столбцов нужно будет *транспонировать* все матрицы, приведенные в этой книге.

5.3.3. Единичная матрица

Единичная матрица — это матрица, которая при умножении на любую другую матрицу дает ту же самую матрицу. Обычно она обозначается **I**. Единичная матрица всегда представляет собой квадратную матрицу с единицами по диагонали и нулями в качестве остальных элементов:

$$\mathbf{I}_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \ \mathbf{AI} = \mathbf{IA} \equiv \mathbf{A}.$$

5.3.4. Инвертирование матриц

Инверсией для матрицы **A** будет другая матрица (обозначенная $\mathbf{A}^{-1}$), которая *отменяет* эффекты матрицы **A**. Так, например, если **A** поворачивает объекты на 37° вокруг оси *Z*, то $\mathbf{A}^{-1}$ будет поворачивать на –37° вокруг оси *Z*. Аналогично если **A** увеличивает объекты в два раза по сравнению с их исходным размером, то $\mathbf{A}^{-1}$ уменьшает объекты до половины размера. Когда матрица умножается на собственную инверсию, результатом *всегда* является единичная матрица, поэтому $\mathbf{A}(\mathbf{A}^{-1}) \equiv (\mathbf{A}^{-1})\mathbf{A} \equiv \mathbf{I}$. Не все матрицы имеют инвертированную матрицу. Однако все *аффинные* преобразования (комбинации чистых вращений, перемещений, масштабирования и сдвигов) обладают инверсией. В случае существования инвертированной матрицы для ее нахождения можно использовать метод Гаусса или LU-разложение.

Поскольку мы часто будем иметь дело с умножением матриц, важно знать, что инверсия последовательности объединенных матриц может быть записана как *обратная конкатенация* инверсий отдельных матриц, например:

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}.$$

5.3.5. Транспонирование

Транспонированная матрица для матрицы **M** обозначается $\mathbf{M}^{\mathrm{T}}$. Она получается отражением элементов исходной матрицы по всей ее диагонали. Другими словами,

строки исходной матрицы становятся столбцами транспонированной матрицы и наоборот:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}.$$

Транспонирование полезно по ряду причин. Инверсия ортонормированной матрицы (чистого вращения) в точности равна ее транспонированию, что хорошо, потому что на транспонирование матрицы затрачивается гораздо меньше ресурсов, чем на нахождение ее инверсии. Транспонирование может быть полезно и при перемещении данных из одной математической библиотеки в другую, потому что некоторые библиотеки используют вектор-столбцы, в то время как другие ожидают вектор-строки. Матрицы, применяемые библиотекой, основанной на вектор-строках, будут *транспонированы* относительно матриц, используемых библиотекой, основанной на вектор-столбцах.

Как и в случае с инверсией, транспонирование последовательности объединенных матриц может быть записано как обратная конкатенация транспонирований отдельных матриц, например:

$$(\mathbf{ABC})^{\mathrm{T}} = \mathbf{C}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}.$$

Это очень пригодится, когда мы будем рассматривать, как применять матрицы преобразования к точкам и векторам.

5.3.6. Однородная система координат

Возможно, из курса школьной алгебры вы помните, что матрица 2×2 может представлять вращение в двух измерениях. Чтобы повернуть вектор **r** на угол ϕ (где вращение в положительную сторону идет против часовой стрелки), мы можем написать:

$$\begin{bmatrix} r'_x & r'_y \end{bmatrix} = \begin{bmatrix} r_x & r_y \end{bmatrix} \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix}.$$

Вероятно, нет ничего удивительного в том, что вращения в трех измерениях могут быть представлены матрицей 3×3. Двухмерный пример, приведенный ранее, — это на самом деле просто вращение в трехмерном пространстве вокруг оси *Z*, поэтому можно написать:

$$\begin{bmatrix} r'_x & r'_y & r'_z \end{bmatrix} \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Естественно, возникает вопрос: можно ли использовать матрицу 3×3 для отображения *вращений*? К сожалению, ответ — нет. Результат трансляции точки **r**

с помощью трансляции **t** требует добавления компонентов **t** к компонентам **r** по отдельности:

$$\mathbf{r}+\mathbf{t}=\begin{bmatrix}(r_x+t_x) & (r_y+t_y) & (r_z+t_z)\end{bmatrix}.$$

Умножение матриц включает в себя умножение и добавление элементов матриц, поэтому идея применения умножения для трансляции кажется многообещающей. Но, к сожалению, нет способа расположить компоненты **t** в матрице 3×3 так, чтобы результат ее умножения на вектор-столбец **r** давал суммы наподобие $r_x + t_x$.

Хорошей новостью является то, что мы *можем* получить такие суммы, если будем использовать матрицу 4×4. Как она будет выглядеть? Что ж, мы знаем, что нам не нужны эффекты вращения, поэтому верхнее поле 3×3 должно содержать единичную матрицу. Если мы расположим компоненты **t** в самом нижнем ряду матрицы и сделаем четвертый элемент вектора **r**, обычно называемый w, равным 1, то скалярное произведение вектора **r** на столбец 1 матрицы даст $(1r_x) + (0r_y) + (0r_z) + + (t_x1)$, а это именно то, что нужно. Если в нижнем правом углу матрицы стоит 1, а в остальной части четвертого столбца — нули, то итоговый вектор также будет иметь 1 в компоненте w. Вот как выглядит окончательная трансляция матрицы 4×4:

$$\mathbf{r}+\mathbf{t}=\begin{bmatrix}r_x & r_y & r_z & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ t_x & t_y & t_z & 1\end{bmatrix}=\begin{bmatrix}(r_x+t_x) & (r_y+t_y) & (r_z+t_z) & 1\end{bmatrix}.$$

Когда таким образом точка или вектор расширяются с трех измерений до четырех, мы говорим, что они записаны в *однородных координатах*. Точка в однородных координатах всегда имеет $w = 1$. Большая часть операций с трехмерными матрицами выполняется игровыми движками с использованием матриц 4×4 с четырехэлементными точками и векторами, записанными в однородных координатах.

Преобразование векторов направления

В математике точки (радиус-векторы) и векторы направления обрабатываются немного по-разному. При преобразовании точки с помощью матрицы перемещение, вращение и масштабирование матрицы применяются к точке. Но при преобразовании направления с помощью матрицы эффекты *перемещения* матрицы игнорируются. Это связано с тем, что векторы направления сами по себе не транслируются, поскольку трансляция вектора направления изменит его длину, а это обычно не то, что нам нужно.

В однородных координатах мы достигаем этого, определяя точки таким образом, чтобы их w-компоненты были равны единице, в то время как векторы направления имели w-компоненты, равные *нулю*. В приведенном далее примере обратите

внимание на то, как компонент $w = 0$ вектора $\mathbf{v}$ умножается на вектор $\mathbf{t}$ в матрице, тем самым устраняя трансляцию в конечном результате:

$$\begin{bmatrix}\mathbf{v} & 0\end{bmatrix}\begin{bmatrix}\mathbf{U} & 0\\ \mathbf{t} & 1\end{bmatrix}=\begin{bmatrix}(\mathbf{vU}+0t) & 0\end{bmatrix}=\begin{bmatrix}\mathbf{vU} & 0\end{bmatrix}.$$

Технически точка в однородных (четырехмерных) координатах может быть преобразована в неоднородные (трехмерные) координаты путем разделения компонентов x, y и z на компонент w:

$$\begin{bmatrix}x & y & z & w\end{bmatrix}\equiv\begin{bmatrix}\frac{x}{w} & \frac{y}{w} & \frac{z}{w}\end{bmatrix}.$$

Это немного объясняет, почему мы устанавливаем w-компонент точки равным единице, а w-компонент вектора — равным нулю. Деление на $w = 1$ не влияет на координаты точки, но деление компонентов чистого вектора направления на $w = 0$ даст бесконечность. Точка на бесконечности в 4D может быть повернута, но не перемещена, потому что, независимо от того, как мы попытаемся ее переместить, она останется на бесконечности. Таким образом, чистый вектор направления в трехмерном пространстве действует как точка на бесконечности в четырехмерном однородном пространстве.

5.3.7. Матрицы базовых преобразований

Любая аффинная матрица преобразования может быть создана простым объединением последовательности из матриц 4×4, представляющих чистые трансляции и вращения, операции чистого масштабирования и/или чистые сдвиги. Эти элементы, из которых получаются базовые преобразования, представлены далее. (Мы не будем обсуждать сдвиг, поскольку он довольно редко используется в играх.)

Обратите внимание на то, что все аффинные матрицы преобразований 4×4 могут быть разбиты на четыре компонента:

$$\mathbf{M}_{\text{affine}}=\begin{bmatrix}\mathbf{U}_{3\times3} & 0_{3\times1}\\ \mathbf{t}_{1\times3} & 1\end{bmatrix},$$

в частности:

- верхнюю матрицу 3×3 $\mathbf{U}$, которая представляет вращение и/или масштабирование;
- вектор трансляции 1×3 $\mathbf{t}$;
- вектор 3×1 из нулей $0 = [0\ 0\ 0]^{\mathrm{T}}$;
- скаляр 1 в нижнем правом углу матрицы.

Когда точка умножается на матрицу, которая была разделена таким образом, результат будет следующим:

$$\begin{bmatrix}\mathbf{r}'_{1\times3} & 1\end{bmatrix}=\begin{bmatrix}\mathbf{r}_{1\times3} & 1\end{bmatrix}\begin{bmatrix}\mathbf{U}_{3\times3} & 0_{3\times1}\\ \mathbf{t}_{1\times3} & 1\end{bmatrix}=\begin{bmatrix}(\mathbf{rU}+\mathbf{t}) & 1\end{bmatrix}.$$

Трансляция

Следующая матрица транслирует точку по вектору **t**:

$$\mathbf{r}+\mathbf{t}=\begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}=\begin{bmatrix} (r_x+t_x) & (r_y+t_y) & (r_z+t_z) & 1 \end{bmatrix} \quad (5.3)$$

или, если записать сокращенно:

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix}\begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{t} & 1 \end{bmatrix}=\begin{bmatrix} (\mathbf{r}+\mathbf{t}) & 1 \end{bmatrix}.$$

Чтобы инвертировать чистую матрицу трансляции, просто инвертируйте вектор **t**, то есть инвертируйте t_x, t_y и t_z.

Вращение

Все чистые матрицы вращения 4×4 имеют вид:

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix}\begin{bmatrix} \mathbf{R} & 0 \\ 0 & 1 \end{bmatrix}=\begin{bmatrix} \mathbf{rR} & 1 \end{bmatrix}.$$

Вектор **t** равен нулю, а верхняя матрица 3×3 **R** содержит косинусы и синусы угла поворота, измеренные в радианах.

Вращение вокруг оси *X* на угол ϕ представляет следующая матрица:

$$\text{rotate}_x(\mathbf{r},\varphi)=\begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (5.4)$$

Вращение вокруг оси *Y* на угол θ представляет следующая матрица (обратите внимание на то, что она *транспонирована* относительно двух других — положительные и отрицательные синусные члены отражены по диагонали):

$$\text{rotate}_y(\mathbf{r},\theta)=\begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (5.5)$$

Вращение вокруг оси *Z* на угол γ представляет следующая матрица:

$$\text{rotate}_z(\mathbf{r},\gamma)=\begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix}\begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (5.6)$$

Вот несколько замечаний по поводу этих матриц.

- Единица в верхней матрице 3×3 всегда находится на оси, вокруг которой происходит вращение, а синусные и косинусные члены — вне оси.
- Положительное вращение направлено от X к Y (вокруг Z), от Y к Z (вокруг X) и от Z к X (вокруг Y). Вращение от Z к X «обвивает» ось, поэтому матрица вращения вокруг оси Y транспонирована относительно двух других. (Используйте правило правой или левой руки, чтобы запомнить.)
- Инверсия чистого вращения — это просто его транспонирование. Это так работает, потому что инвертирование вращения эквивалентно вращению на отрицательный угол. Вы, возможно, помните, что $\cos(-\theta) = \cos(\theta)$, а $\sin(-\theta) = -\sin(\theta)$, поэтому отрицание угла приводит к тому, что два синусных члена меняются местами, в то время как косинусные остаются на месте.

Масштабирование

Следующая матрица масштабирует точку $\mathbf{r}$ в s_x раз вдоль оси X, в s_y раз вдоль оси Y и в s_z раз вдоль оси Z:

$$\mathbf{rS} = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x r_x & s_y r_y & s_z r_z & 1 \end{bmatrix} \tag{5.7}$$

или, если записать сокращенно:

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}_{3\times3} & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{rS}_{3\times3} & 1 \end{bmatrix}.$$

Вот некоторые замечания об этом типе матриц.

- Чтобы инвертировать матрицу масштабирования, просто замените s_x, s_y и s_z их обратными величинами (то есть $1/s_x$, $1/s_y$ и $1/s_z$).
- Равенство коэффициентов масштабирования по всем трем осям ($s_x = s_y = s_z$) называется *равномерным масштабом*. Сферы остаются сферами в равномерном масштабе, тогда как в неравномерном они становятся эллипсоидами. Чтобы сохранить математическую проверку ограничивающих сфер простой и быстрой, многие игровые движки разрешают применять к отображаемым геометрическим и коллизионным примитивам только равномерный масштаб.
- Когда матрица равномерного масштаба $\mathbf{S}_u$ и матрица вращения $\mathbf{R}$ конкатенированы, порядок умножения неважен, то есть $\mathbf{S}_u\mathbf{R} = \mathbf{RS}_u$. Это работает только для *равномерного* масштаба!

5.3.8. Матрицы 4 × 3

Крайний правый столбец аффинной матрицы преобразования 4 × 4 всегда содержит вектор $[0\ 0\ 0\ 1]^T$. Так что разработчики игр часто пропускают четвертый столбец для экономии памяти. Вы часто будете встречать аффинные матрицы преобразования 4 × 3 в математических библиотеках для игр.

5.3.9. Координатное пространство

Мы видели, как применять преобразования к точкам и векторам направления, используя матрицы 4 × 4. Можем распространить этот принцип на твердые объекты, осознав, что такой объект можно рассматривать как бесконечный набор точек. Применение преобразования к твердому объекту похоже на применение этого же преобразования к каждой точке внутри объекта. Например, в компьютерной графике объект обычно представлен сеткой треугольников, каждый из которых имеет три вершины, представленные точками. В этом случае объект может быть преобразован путем применения матрицы преобразований ко всем его вершинам по очереди.

Ранее мы говорили, что точка — это вектор, начало которого зафиксировано на начале некоторой системы координат. Это еще один способ сказать, что точка (радиус-вектор) всегда выражается *относительно* набора осей координат. Тройка чисел, представляющих точку, изменяется численно всякий раз, когда мы выбираем новый набор координатных осей. На рис. 5.17 мы видим точку *P*, представленную двумя разными радиус-векторами: вектор $\mathbf{P}_A$ задает положение *P* относительно осей *A*, а вектор $\mathbf{P}_B$ задает положение этой же точки относительно набора осей *B*.

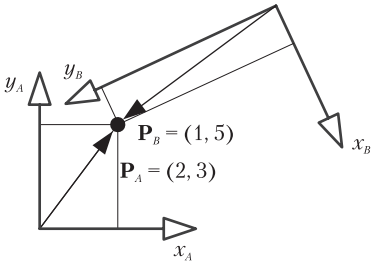


Рис. 5.17. Радиус-векторы для точки P относительно различных осей координат

В физике набор осей координат представляет собой систему отсчета, поэтому мы иногда называем набор осей *системой координат* или просто *системой*. Работающие в игровой индустрии для обозначения набора осей координат используют также термин *«пространство координат»* (или просто *«пространство»*). В следующих подразделах рассмотрим некоторые из наиболее распространенных координатных пространств, применяемых в играх и компьютерной графике.

Пространство модели

Когда треугольная сетка создается с помощью инструмента Maya или 3DStudioMAX, позиции вершин треугольников указываются относительно декартовой системы координат, которую мы называем *пространством модели* (известно также как *пространство объекта* или *локальное пространство*). Центр пространства модели

обычно размещается посередине объекта, например в его центре масс, или на земле между ногами гуманоидного персонажа или животного.

Большинству игровых объектов присуща направленность. Например, у самолета есть нос, киль и крылья, которые соответствуют направлениям вперед, вверх и влево/вправо. Оси пространства модели обычно соответствуют этим естественным направлениям модели, им дают интуитивно понятные имена, чтобы обозначить их направленность (рис. 5.18).

Рис. 5.18. Один из возможных вариантов базисных векторов фронтальной, левой и верхней осей для пространства модели самолета

- *Фронтальная*. Такое название дается оси, которая указывает, в каком направлении перемещается или смотрит объект. В этой книге будем использовать символ **F** для обозначения единичного базисного вектора, сонаправленного с фронтальной осью.
- *Верхняя*. Такое название дается оси, которая направлена к верхней части объекта. Единичный базисный вектор, сонаправленный с этой осью, будет обозначен **U**.
- *Левая* или *правая*. Название «левая» или «правая» дается оси, которая направлена к левой или правой части объекта. Выбор названия зависит от того, какую систему координат задействует ваш игровой движок — левостороннюю или правостороннюю. Единичный базисный вектор, сонаправленный с этой осью, будет обозначен **L** или **R**.

Соотношение обозначений (*фронтальная*, *верхняя*, *левая*) и осей (*X*, *Y*, *Z*) совершенно произвольно. Обычно при работе с правосторонними координатами *фронтальной* называют ось *Z* со знаком «+», *левой* — ось *X* со знаком «+» и *верхней* — ось *Y*

со знаком «+» (или, если речь идет о единичных базисных векторах, **F** = **k**, **L** = **i** и **U** = **j**). Тем не менее часто ось *X* со знаком «+» может называться *фронтальной* и ось *Z* со знаком «+» — *правой* (**F** = **i**, **R** = **k**, **U** = **j**). Я также работал с движками, где ось *Z* расположена вертикально. Самое важное — строго придерживаться одной ориентации координат на протяжении всей работы с движком.

В качестве примера того, как интуитивно понятные названия осей могут уменьшить путаницу, рассмотрим *углы Эйлера* (*тангаж*, *рыскание*, *крен*), которые часто используются для описания ориентации самолета. Невозможно определить углы тангажа, рыскания и крена с помощью базисных векторов (**i**, **j**, **k**), поскольку их ориентация произвольна. Тем не менее мы *можем* определить тангаж, рыскание и крен с помощью базисных векторов (**L**, **U**, **F**), потому что их ориентация четко определена, а именно:

- *тангаж* — это вращение вокруг **L** или **R**;
- *рыскание* — вращение вокруг **U**;
- *крен* — вращение вокруг **F**.

Пространство мира

Пространство мира — это фиксированное координатное пространство, в котором выражены положения, ориентация и масштаб всех объектов в игровом мире. Это координатное пространство связывает все отдельные объекты в единый виртуальный мир.

Расположение центра пространства мира произвольно, часто его размещают вблизи центра игрового пространства, чтобы минимизировать снижение точности плавающей точки, которое может произойти, когда координаты (*x*, *y*, *z*) станут очень большими. Точно так же ориентация осей *X*, *Y* и *Z* может быть произвольной, хотя большинство игровых движков, с которыми я сталкивался, используют в качестве верхней оси *Y* или *Z*. Система координат, в которой ось *Y* направлена вверх, скорее всего, базируется на двухмерной системе координат, рассматриваемой в большинстве учебников математики, где ось *Y* направлена вверх, а ось *X* — вправо. Система координат, в которой вверх направлена ось *Z*, тоже встречается часто, потому что это позволяет ортографическому виду сверху на игровой мир выглядеть как стандартная двухмерная *XY*-проекция.

В качестве примера предположим, что конец левого крыла самолета имеет координаты (5, 0, 0) в пространстве модели. (В игре фронтальные векторы соответствуют положительной части оси *Z* в пространстве модели с осью *Y*, направленной вверх, как показано на рис. 5.18.) Теперь представьте, что самолет смотрит в сторону положительной части оси *X* в пространстве мира, а центр пространства модели находится в случайной точке, например, (–25, 50, 8). Поскольку вектор **F** самолета, соответствующий положительной части оси *Z* в пространстве модели, соответствует положительной части оси *X* в пространстве мира, мы знаем, что самолет был повернут на 90° вокруг оси *Y* пространства мира. Таким образом, если бы самолет находился в центре пространства мира, конец его левого крыла был бы в точке (0, 0, –5) этого пространства. Но, поскольку центр самолета был перенесен

в (–25, 50, 8), итоговое положение конца левого крыла самолета в пространстве мира будет (–25, 50, [8 – 5]) = (–25, 50, 3) (рис. 5.19).

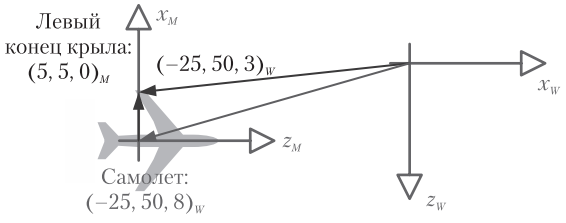


Рис. 5.19. Самолет, конец левого крыла которого имеет координаты (5, 0, 0) в пространстве модели. Если самолет повернуть на 90° вокруг оси пространства мира, а центр пространства модели перенести в координаты (–25, 50, 8) пространства мира, то конец его левого крыла в этих координатах окажется в точке (–25, 50, 3)

Конечно, мы могли бы заполнить небо большим количеством самолетов. В этом случае все концы их левых крыльев будут иметь координаты (5, 0, 0) в пространстве модели. Но в пространстве мира концы их левых крыльев будут иметь самые разнообразные координаты в зависимости от положения и перемещения каждого самолета.

Пространство вида

Пространство вида, также называемое *пространством камеры*, — это координатная рамка, прикрепленная к камере. Начало координат пространства вида находится в фокусе камеры. Здесь также возможна любая ориентация системы координат. Стандартной является левосторонняя система координат с верхней осью *Y*, где ось *Z* указывает в направлении, в котором смотрит камера, что позволяет оси *Z* отображать глубину на экране. Другие движки и API, такие как OpenGL, определяют пространство вида правосторонним, в этом случае камера направлена в сторону отрицательной части оси *Z*, координаты которой отображают отрицательную глубину.

Два возможных определения пространства вида показаны на рис. 5.20.

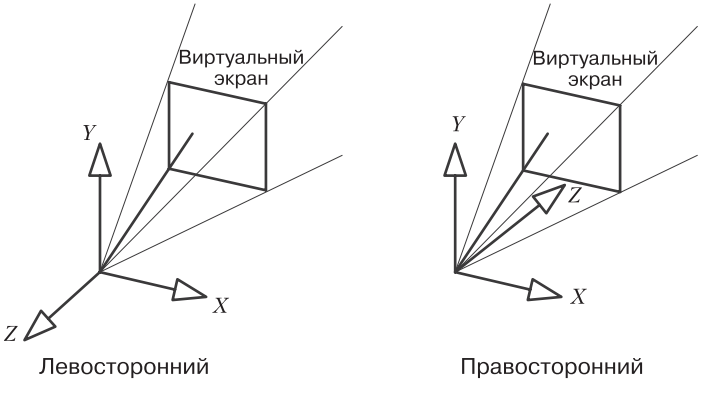


Рис. 5.20. Примеры лево- и правостороннего пространства вида

5.3.10. Переход к новому базису

В играх и компьютерной графике часто полезно преобразовать позицию, положение и масштаб объекта из одной системы координат в другую. Называется эта операция *переходом к новому базису*.

Иерархии координатных пространств

Координатные рамки относительны. То есть если вы хотите определить позицию, положение и масштаб набора осей в трехмерном пространстве, то должны указать эти величины *относительно* другого набора осей, в противном случае числа не будут иметь никакого значения. Это означает, что координатные пространства образуют *иерархию* — каждое из них является *дочерним* по отношению к какому-то другому координатному пространству, а оно, в свою очередь, будет для первого *родительским*. У пространства мира нет родителя — оно находится в корне дерева координатных пространств, а все остальные системы координат указываются относительно него либо как дочерние, либо как имеющие более дальнее родство.

Построение матрицы перехода к новому базису

Матрица, которая преобразует точки и направления из любой дочерней системы координат C в ее родительскую систему координат P, может быть записана как $\mathbf{M}_{C\to P}$ (произносится «си к пи»). Нижний индекс указывает, что эта матрица преобразует точки и направления из дочернего пространства в родительское. Любой вектор положения дочернего пространства $\mathbf{P}_C$ может быть преобразован в вектор положения родительского пространства $\mathbf{P}_P$ следующим образом:

$$\mathbf{P}_P = \mathbf{P}_C\mathbf{M}_{C\to P};\ \mathbf{M}_{C\to P} = \begin{bmatrix} \mathbf{i}_C & 0 \\ \mathbf{j}_C & 0 \\ \mathbf{k}_C & 0 \\ \mathbf{t}_C & 1 \end{bmatrix} = \begin{bmatrix} i_{Cx} & i_{Cy} & i_{Cz} & 0 \\ j_{Cx} & j_{Cy} & j_{Cz} & 0 \\ k_{Cx} & k_{Cy} & k_{Cz} & 0 \\ t_{Cx} & t_{Cy} & t_{Cz} & 1 \end{bmatrix}.$$

В этом уравнении:

- $\mathbf{i}_C$ — единичный базисный вектор, направленный вдоль оси *X* дочернего пространства, выраженный в координатах родительского пространства;
- $\mathbf{j}_C$ — единичный базисный вектор, направленный вдоль оси *Y* дочернего пространства, выраженный в координатах родительского пространства;
- $\mathbf{k}_C$ — единичный базисный вектор, направленный вдоль оси *Z* дочернего пространства, выраженный в координатах родительского пространства;
- $\mathbf{t}_C$ — трансляция дочерней системы координат относительно родительского пространства.

Такой результат не должен быть неожиданным. Вектор $\mathbf{t}_C$ — это просто трансляция осей дочернего пространства относительно родительского, поэтому, если бы

остальная часть матрицы была тождественной, точка (0, 0, 0) в дочернем пространстве ожидаемо стала бы $\mathbf{t}_C$ в родительском. Единичные векторы $\mathbf{i}_C$, $\mathbf{j}_C$ и $\mathbf{k}_C$ образуют верхнее поле матрицы 3×3, которая является чистой матрицей вращения, поскольку эти векторы имеют единичную длину. Это можно увидеть более наглядно, рассмотрев простой пример, такой как ситуация, в которой дочернее пространство поворачивается на угол γ вокруг оси *Z* без перемещения. Вспомним из уравнения (5.6), что матрица для такого вращения имеет вид

$$\text{rotate}_z(\mathbf{r}, \gamma) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Но на рис. 5.21 мы видим, что координаты векторов $\mathbf{i}_C$ и $\mathbf{j}_C$, выраженных в родительском пространстве, равны $\mathbf{i}_C = [\cos\gamma \sin\gamma 0]$ и $\mathbf{j}_C = [-\sin\gamma \cos\gamma 0]$. Когда мы вставим эти векторы в формулу для $\mathbf{M}_{C\to P}$ вместе с $\mathbf{k}_C = [0\ 0\ 1]$, она будет точно соответствовать матрице $\text{rotate}_z(\mathbf{r}, \gamma)$ из уравнения (5.6).

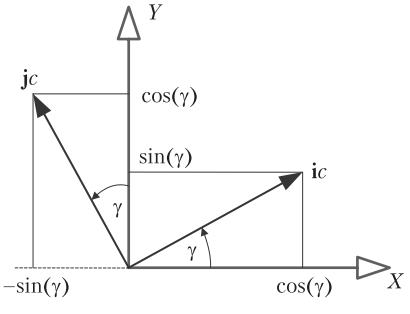


Рис. 5.21. Замена базиса, при которой дочерние оси поворачиваются на угол γ относительно родителя

Масштабирование дочерних осей. Масштабирование дочерней системы координат достигается простым масштабированием соответствующих базисных векторов. Например, если дочернее пространство увеличено в два раза, то базисные векторы $\mathbf{i}_C$, $\mathbf{j}_C$ и $\mathbf{k}_C$ будут иметь длину 2, а не 1.

Извлечение единичных базисных векторов из матрицы

То, что мы можем построить матрицу замены базиса с помощью трансляции и трех декартовых базисных векторов, дает нам еще один мощный инструмент: зная *любую* аффинную матрицу преобразования 4×4, мы можем пойти другим путем и извлечь из нее базисные векторы дочернего пространства $\mathbf{i}_C$, $\mathbf{j}_C$ и $\mathbf{k}_C$, просто изолировав соответствующие строки матрицы (или столбцы, если ваша математическая библиотека использует вектор-столбцы).

Это может быть невероятно полезно. Допустим, дано преобразование «модель — мир» автомобиля в виде аффинной матрицы преобразования 4×4 (очень распространенный пример). На самом деле это просто матрица замены базиса, превращающая точки в пространстве модели в их эквиваленты в пространстве мира. Также предположим, что в игре положительная часть оси Z всегда указывает в направлении, в котором смотрит объект. Итак, чтобы найти единичный вектор, представляющий направление движения автомобиля, можно просто извлечь $\mathbf{k}_C$ непосредственно из матрицы «модель — мир», взяв ее третью строку. Этот вектор уже будет нормализован и готов к использованию.

Преобразование систем координат и векторов

Мы уже говорили, что матрица $\mathbf{M}_{C \to P}$ преобразует точки и направления из дочернего пространства в родительское. Вспомните, что четвертая строка $\mathbf{M}_{C \to P}$ содержит $\mathbf{t}_C$ — трансляцию дочерних координатных осей относительно осей пространства мира. Поэтому другой способ визуализации матрицы $\mathbf{M}_{C \to P}$ состоит в том, чтобы представить, что она принимает родительские оси координат и преобразует их в дочерние. Это обратно тому, что происходит с точками и векторами направления. Другими словами, если матрица преобразует *векторы* из дочернего пространства в родительское, то она преобразует также *оси координат* из родительского пространства в дочернее. Если подумать, все вполне логично: перемещение точки на 20 единиц вправо при фиксированных осях координат равнозначно перемещению координатных осей на 20 единиц влево при фиксированной точке (рис. 5.22).

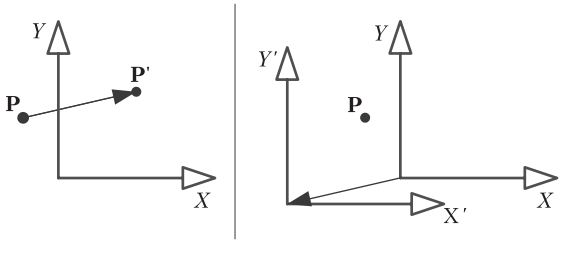


Рис. 5.22. Два способа интерпретации матрицы преобразования. Слева точка движется против фиксированного набора осей. Справа оси движутся в противоположном направлении, в то время как точка остается неподвижной

Конечно, это тоже может сбить с толку. Если рассматривать все с точки зрения координатных осей, то преобразования проходят в одном направлении, но если с позиции точек и векторов — в другом! Как и во многих затруднительных ситуациях в жизни, возможно, лучше всего выбрать единый образ действий и придерживаться его. Например, в этой книге мы придерживаемся следующих положений:

- преобразования применяются к векторам (не осям координат);
- векторы записываются в виде строк (не столбцов).

Вместе эти два условия позволяют слева направо читать последовательности умножения матриц так, чтобы они оставались понятными (то есть в выражении $\mathbf{r}_D = \mathbf{r}_A\mathbf{M}_{A\to B}\mathbf{M}_{B\to C}\mathbf{M}_{C\to D}$ B и C просто сокращают, оставляя только $\mathbf{r}_D = \mathbf{r}_A\mathbf{M}_{A\to D}$). Конечно, если вы начинаете двигать оси координат вместо точек и векторов, вам либо надо читать преобразования справа налево, либо использовать обратное одному из перечисленных условий. Неважно, какие условия вы выберете, если только вам будет легко их запомнить и с ними работать.

Однако надо заметить, что некоторые проблемы легче решать, преобразуя векторы, в то время как с другими легче работать, передвигая оси координат. Как только вы начнете хорошо ориентироваться в трехмерных векторах и матричной математике, то сможете довольно легко применять нужные условия в зависимости от задачи.

5.3.11. Преобразование векторов нормали

Вектор нормали — это особый вид единичного вектора. Он всегда должен оставаться *перпендикулярным* любой поверхности или плоскости, к которой относится. При преобразовании вектора нормали надо соблюдать особую осторожность, чтобы сохранить как его длину, так и свойство перпендикулярности.

В целом, если точку или вектор (не нормали) можно повернуть из пространства A в пространство B с помощью матрицы 3×3 ($\mathbf{M}_{A\to B}$), то вектор нормали $\mathbf{n}$ будет преобразован из пространства A в пространство B *обратным транспонированием* этой матрицы $(\mathbf{M}_{A\to B}^{-1})^{\mathrm{T}}$. Здесь мы не будем доказывать или выводить этот результат (см. хороший пример в [32, раздел 3.5]). Однако заметим, что если матрица $\mathbf{M}_{A\to B}$ содержит только равномерный масштаб и не имеет сдвига, то углы между всеми поверхностями и векторами в пространстве B будут такими же, как и в пространстве A. В таком случае матрица $\mathbf{M}_{A\to B}$ будет работать для любого вектора, в том числе нормального. Но если $\mathbf{M}_{A\to B}$ содержит неравномерный масштаб или сдвиг (то есть она *неортогональна*), то углы между поверхностями и векторами *не сохраняются* при перемещении из пространства A в пространство B. Вектор, который был нормальным к поверхности в пространстве A, не обязательно будет перпендикулярным этой поверхности в пространстве B. Операция обратного транспонирования учитывает это искажение, делая векторы нормали снова перпендикулярными их поверхностям, даже когда преобразование включает неравномерный масштаб или сдвиг. Другой способ рассмотреть это состоит в том, что потребуется обратное транспонирование, поскольку нормаль к поверхности на самом деле является *псевдовектором*, а не обычным вектором (см. подраздел 5.2.4).

5.3.12. Хранение матриц в памяти

В языках C и C++ для хранения матрицы часто используется двухмерный массив. Напомню, что в синтаксисе двухмерного массива C/C++ первый индекс — это

строка, а второй — столбец, при этом индекс столбца меняется быстрее всего при последовательном перемещении по памяти:

```
float m[4][4]; // [row] [col], col меняется быстрее всего

// выравниваем массив, чтобы продемонстрировать упорядоченность
float* pm = &m[0][0];
ASSERT( &pm[0] == &m[0][0] );
ASSERT( &pm[1] == &m[0][1] );
ASSERT( &pm[2] == &m[0][2] );
// и т. п.
```

Есть два варианта сохранения матрицы в двухмерном массиве C/C++:

- векторы ($\mathbf{i}_C$, $\mathbf{j}_C$, $\mathbf{k}_C$, $\mathbf{t}_C$) хранятся *непрерывно* в памяти (то есть каждая строка содержит один вектор);
- векторы хранятся *выстроенными по вертикали* в памяти (то есть каждый столбец содержит один вектор).

Преимущество первого подхода состоит в том, что мы можем обратиться к любому из четырех векторов, просто проиндексировав матрицу и интерпретировав четыре смежных значения, которые найдем как вектор из четырех элементов. У этой схемы есть еще одно преимущество — она точно соответствует матричным уравнениям *векторов строк* (это еще одна причина, почему я в этой книге решил придерживаться условия «вектор — строка»). Второй подход иногда необходим при быстром умножении матрицы на вектор с применением микропроцессора с векторной поддержкой (SIMD), как мы увидим позже в этой главе. В большинстве игровых движков, с которыми я сталкивался, матрицы хранятся с использованием первого подхода с *векторами* в *строках* двухмерного массива C/C++. Это продемонстрировано далее:

```
float M[4][4];

M[0][0]=ix;   M[0][1]=iy;   M[0][2]=iz;   M[0][3]=0.0f;
M[1][0]=jx;   M[1][1]=jy;   M[1][2]=jz;   M[1][3]=0.0f;
M[2][0]=kx;   M[2][1]=ky;   M[2][2]=kz;   M[2][3]=0.0f;
M[3][0]=tx;   M[3][1]=ty;   M[3][2]=tz;   M[3][3]=1.0f;
```

При просмотре в отладчике матрица выглядит так:

```
M[][]
    [0]
         [0] ix
          [1] iy
          [2] iz
          [3] 0.0000
     [1]
          [0] jx
          [1] jy
          [2] jz
          [3] 0.0000
```

```
[2]
     [0] kx
     [1] ky
     [2] kz
     [3] 0.0000
[3]
     [0] tx
     [1] ty
     [2] tz
     [3] 1.0000
```

Простой способ определить, какую схему использует ваш движок, — найти функцию, которая построит матрицу перевода 4×4. (Каждая хорошая трехмерная математическая библиотека предоставляет такую функцию.) Затем можете проверить исходный код, чтобы увидеть, где хранятся элементы вектора **t**. Если у вас нет доступа к исходному коду вашей математической библиотеки (что в игровой индустрии встречается довольно редко), вы всегда можете вызвать функцию с простой и понятной трансляцией, такой как (4, 3, 2), и посмотреть полученную матрицу. Если строка 3 содержит значения `4.0f`, `3.0f`, `2.0f`, `1.0f`, то векторы находятся в строках, в противном случае векторы находятся в столбцах.

5.4. Кватернионы

Мы видели, что матрица 3×3 может представлять произвольное вращение в трех измерениях. Однако по ряду причин матрица не всегда является идеальным представлением вращения.

- Нужно девять значений с плавающей точкой, чтобы представить вращение, что кажется чрезмерным, учитывая, что есть только три степени свободы — наклон, поворот и вращение.
- Вращение вектора требует умножения матрицы на вектор, которое включает в себя три скалярных произведения, или всего девять умножений и шесть сложений. Было бы неплохо найти менее затратное для вычисления представление вращения, если это возможно.
- В играх и компьютерной графике порой очень важно иметь возможность находить вращения, которые представляют собой какой-то процент от расстояния между двумя известными вращениями. Например, если требуется плавно переместить камеру от некоторого начального положения A до конечного положения B в течение нескольких секунд, мы должны быть в состоянии найти множество промежуточных вращений между A и B в ходе движения. Оказывается, это трудно сделать, когда положения A и B выражены матрицами.

К счастью, есть представление с помощью вращения, которое решает все три проблемы. Это математический объект, известный как *кватернион*. Кватернион очень похож на четырехмерный вектор, но ведет себя иначе. Кватернионы обычно записывают, не используя ни курсив, ни жирный шрифт, следующим образом: $\mathrm{q} = [q_x\ q_y\ q_z\ q_w]$.

Кватернионы были разработаны сэром Уильямом Роуэном Гамильтоном в 1843 году как расширение комплексных чисел. (В частности, кватернион может быть интерпретирован как четырехмерное комплексное число с одной действительной осью и тремя мнимыми осями, представленными мнимыми числами *i*, *j* и *k*. Таким образом, кватернион можно записать в сложной форме следующим образом: $q = iq_x + jq_y + kq_z + q_w$.) Кватернионы впервые были использованы для решения задач в области механики. Технически кватернион подчиняется ряду правил, известных как *четырехмерная нормированная ассоциативная алгебра* с делением над вещественными числами. К счастью, нам не нужно глубоко вникать в эти довольно эзотерические алгебраические правила. Достаточно будет знать, что *единичные кватернионы*, то есть все кватернионы, подчиняющиеся ограничению $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$, представляют трехмерные вращения.

Есть много отличных статей, веб-страниц и презентаций по кватернионам, доступных в Сети для дополнительного чтения. Вот один из моих любимых источников: http://graphics.ucsd.edu/courses/cse169_w05/CSE169_04.ppt.

5.4.1. Единичные кватернионы как трехмерные вращения

Единичный кватернион может быть визуализирован как трехмерный вектор плюс четвертая скалярная координата. Векторная часть $\mathbf{q}_V$ представляет собой единицу оси вращения, масштабированную по синусу половины угла вращения. Скалярная часть q_S является косинусом половины угла. Таким образом, единичный кватернион q можно записать:

$$q = \begin{bmatrix} \mathbf{q}_V & q_S \end{bmatrix} = \left[\mathbf{a} \sin\left(\frac{\theta}{2}\right) \quad \cos\left(\frac{\theta}{2}\right) \right],$$

где **a** — единичный вектор, направленный вдоль оси вращения; θ — угол вращения. Направление вращения следует *правилу правой руки*, поэтому, если ваш большой палец соответствует вектору **a**, положительное вращение будет направлено в сторону согнутых пальцев.

Конечно, можно записать q и как простой четырехэлементный вектор:

$$q = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix},$$

где

$$q_x = q_{Vx} = a_x \sin\frac{\theta}{2};\ q_y = q_{Vy} = a_y \sin\frac{\theta}{2};$$

$$q_z = q_{Vz} = a_z \sin\frac{\theta}{2};\ q_w = qS = \cos\frac{\theta}{2}.$$

Единичный кватернион очень похож на представление вращения «ось + угол», то есть вектор из четырех элементов формы [**a** θ]. Однако кватернионы с математической точки зрения удобнее, чем их аналоги в виде «ось + угол», как мы увидим в дальнейшем.

5.4.2. Операции с кватернионами

Над кватернионами можно выполнять знакомые из векторной алгебры операции масштабирования и сложения векторов. Однако следует помнить, что сумма двух единичных кватернионов не представляет трехмерное вращение, поскольку длина такого кватерниона не будет равна 1. В результате вы не увидите никаких сумм кватернионов в игровом движке, если только они не масштабируются с сохранением единичной длины.

Умножение кватернионов

Одной из наиболее важных операций, которые мы будем выполнять над кватернионами, является операция умножения. Для кватернионов p и q, представляющих вращения **P** и **Q** соответственно, произведение обозначает составное вращение, то есть вращение **Q**, за которым следует вращение **P**. Существует довольно много видов умножения кватернионов, но мы ограничимся обсуждением используемого в сочетании с трехмерными вращениями, а именно произведения Грассмана. С помощью этого определения произведение pq вычисляется следующим образом:

$$\mathrm{pq} = \left[\left(p_S \mathbf{q}_V + q_S \mathbf{p}_V + \mathbf{p}_V \times \mathbf{q}_V \right) \quad \left(p_S q_S - \mathbf{p}_V \cdot \mathbf{q}_V \right) \right].$$

Обратите внимание, как произведение Грассмана определяется с точки зрения векторной части, которая заканчивается компонентами x, y и z получившегося кватерниона, и скалярной части, которая заканчивается компонентом w.

Сопряжение и инверсия

Кватернион, *обратный* q, обозначается q^{-1} и определяется как кватернион, который при умножении на оригинал дает скаляр 1, то есть $\mathrm{qq}^{-1} = 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k} + 1$. Кватернион [0 0 0 1] представляет нулевое вращение, что логично, поскольку sin(0) = 0 для первых трех компонентов и cos(0) = 1 для последнего компонента.

Чтобы вычислить обратное значение кватерниона, мы должны сначала найти величину, известную как *сопряжение*. Обычно она обозначается q* и определяется следующим образом:

$$\mathrm{q}^* = \left[-\mathbf{q}_V \quad q_S \right].$$

Другими словами, мы инвертируем векторную часть, а скалярную оставляем без изменений.

Используя найденное сопряжение кватерниона, обратный кватернион q^{-1} вычисляем следующим образом:

$$\mathrm{q}^{-1} = \frac{\mathrm{q}^*}{\left|\mathrm{q}\right|^2}.$$

Наши кватернионы всегда имеют единичную длину, то есть q = 1, поскольку представляют трехмерные вращения. Итак, в данном случае обратный и сопряженный совпадают:

$$q^{-1} = q^{*} = \left[-\mathbf{q}_V \quad q_S\right]$$

при |q| = 1.

Этот очень полезно, потому что таким образом мы всегда можем избежать довольно дорогого деления на квадратную величину при инвертировании кватерниона, если заранее знаем, что кватернион нормализован. Это также означает, что инвертирование кватерниона, как правило, происходит намного быстрее, чем инвертирование матрицы 3 × 3, что можно использовать в некоторых ситуациях при оптимизации своего движка.

Сопряжение и инверсия произведения. Сопряжение произведения кватернионов (pq) равно обратному произведению сопряжений отдельных кватернионов:

$$(pq)^{*} = q^{*}p^{*}.$$

Аналогично инверсия произведения кватернионов равна обратному произведению инверсий отдельных кватернионов:

$$(pq)^{-1} = q^{-1}p^{-1}. \tag{5.8}$$

Это аналогично действию, выполняемому при транспонировании или инвертировании произведений матриц.

5.4.3. Вращение векторов через кватернионы

Как можно применить вращение кватерниона к вектору? Первый шаг — переписать вектор в *форме кватерниона*. Вектор — это сумма, включающая в себя единичные базисные векторы **i**, **j** и **k**. Кватернион — это сумма, включающая не только **i**, **j** и **k**, но и четвертое, скалярное, слагаемое. Поэтому логично, что вектор можно записать в виде кватерниона со скалярным слагаемым q_S, равным нулю. Учитывая вектор **v**, мы можем записать соответствующий кватернион v = [**v** 0] = [v_x v_y v_z 0].

Чтобы повернуть вектор **v** на кватернион q, мы предварительно умножим вектор, записанный в форме кватерниона v, на q, а затем умножим его на *обратный* кватернион q^{-1}. Таким образом, повернутый вектор **v**′ можно найти следующим образом:

$$v' = \text{rotate}(q, \mathbf{v}) = qvq^{-1}.$$

Это эквивалентно использованию сопряжения кватерниона, потому что наши кватернионы всегда единичной длины:

$$v' = \text{rotate}(q, \mathbf{v}) = qvq^{*}. \tag{5.9}$$

Повернутый вектор **v**′ получается простым его извлечением из кватернионной формы v′.

Умножение кватернионов может быть полезно в самых разных ситуациях в реальных играх. Например, мы хотим найти единичный вектор, описывающий направление, в котором летит самолет. Далее предположим, что по условию в нашей игре положительная часть оси *Z* всегда указывает на переднюю часть объекта. Таким образом, фронтальный единичный вектор любого объекта *в пространстве модели* по определению всегда $\mathbf{F}_m = [0\ 0\ 1]$. Чтобы преобразовать этот вектор в пространство мира, можно просто взять кватернион положения самолета q и использовать его с уравнением (5.9), чтобы повернуть вектор в пространстве модели $\mathbf{F}_m$ в его эквивалент в пространстве мира $\mathbf{F}_w$ (конечно, после того, как мы преобразовали эти векторы в форму кватерниона):

$$F_w = qF_mq^{-1} = q\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}q^{-1}.$$

Кватернионная конкатенация

Вращения могут быть *конкатенированы* точно так же, как преобразования на основе матрицы, — путем умножения кватернионов. Например, рассмотрим три различных вращения, представленных кватернионами q_1, q_2 и q_3 с матричными эквивалентами $\mathbf{R}_1$, $\mathbf{R}_2$ и $\mathbf{R}_3$. Сначала мы хотим выполнить вращение 1, затем вращение 2 и, наконец, вращение 3. Составная матрица вращения $\mathbf{R}_{net}$ может быть найдена и применена к вектору **v** следующим образом:

$$\mathbf{R}_{net} = \mathbf{R}_1\mathbf{R}_2\mathbf{R}_3;$$
$$\mathbf{v}' = \mathbf{v}\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3 = \mathbf{v}\mathbf{R}_{net}.$$

Аналогично составной кватернион вращения qn_{et} может быть найден и применен к вектору **v** (в кватернионной форме v) так:

$$q_{net} = q_3q_2q_1;$$
$$v' = q_3q_2q_1 \vee q_1^{-1}q_2^{-1}q_3^{-1} = q_{net} \vee q_{net}^{-1}.$$

Обратите внимание на то, что произведение кватернионов нужно находить в порядке, обратном тому, в котором применяются вращения ($q_3q_2q_1$). Это обусловлено тем, что вращения кватернионов всегда перемножаются с *обеих* сторон вектора: с расположенными слева необращенными кватернионами и с расположенными справа обращенными. Как мы видели в уравнении (5.8), инверсией произведения кватернионов является обратное произведение отдельных инверсий, поэтому неинвертированные кватернионы читаются справа налево, а инвертированные — слева направо.

5.4.4. Кватернионно-матричная эквивалентность

Мы можем свободно преобразовать любое трехмерное вращение между матричным представлением $\mathbf{R}$ 3×3 и кватернионным представлением q. Если мы выполним:

$$q = \begin{bmatrix}\mathbf{q}_V & q_S\end{bmatrix} = \begin{bmatrix}q_{Vx} & q_{Vy} & q_{Vz} & q_S\end{bmatrix} = \begin{bmatrix}x & y & z & w\end{bmatrix},$$

то можем найти **R** следующим образом:

$$\mathbf{R} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2zw & 2xz-2yw \\ 2xy-2zw & 1-2x^2-2z^2 & 2yz+2xw \\ 2xz-2yw & 2yz-2xw & 1-2x^2-2y^2 \end{bmatrix}.$$

Аналогично при заданном **R** можем найти q с помощью приведенного далее кода (где $q[0] = q_{Vx}$, $q[1] = q_{Vy}$, $q[2] = q_{Vx}$ и $q[3] = q_S$). Этот код предполагает, что мы используем вектор-строки в C/C++, то есть строки матрицы соответствуют строкам матрицы **R**, показанной ранее. Код взят из статьи Ника Бобика *Gamasutra* (опубликована 5 июля 1998 года), находящейся здесь: hwww.gamasutra.com/view/feature/3278/rotating_objects_using_ quaternions.php. Обсуждение некоторых еще более быстрых методов преобразования матрицы в кватернион с помощью различных предположений о природе матрицы см. по адресу httwww.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm.

```
void matrixToQuaternion(
            const float R[3][3],
            float q[/*4*/])
{
    float trace = R[0][0] + R[1][1] + R[2][2];

    // проверяем диагональ
    if (trace > 0.0f)
    {
        float s = sqrt(trace + 1.0f);
        q[3] = s * 0.5f;

        float t = 0.5f / s;
        q[0] = (R[2][1] – R[1][2]) * t;
        q[1] = (R[0][2] – R[2][0]) * t;
        q[2] = (R[1][0] – R[0][1]) * t;
    }
    else
    {
        // диагональ отрицательна
        int i = 0;
        if (R[1][1] > R[0][0]) i = 1;
        if (R[2][2] > R[i][i]) i = 2;

        static const int NEXT[3] = {1, 2, 0};
        int j = NEXT[i];
        int k = NEXT[j];

        float s = sqrt((R[i][j]
                        - (R[j][j] + R[k][k]))
                            + 1.0f);

        q[i] = s * 0.5f;
```

```
        float t;
        if (s != 0.0)   t = 0.5f / s;
        else            t = s;

        q[3] = (R[k][j] – R[j][k]) * t;
        q[j] = (R[j][i] + R[i][j]) * t;
        q[k] = (R[k][i] + R[i][k]) * t;
    }
}
```

Давайте на минуту остановимся, чтобы рассмотреть условные обозначения. В этой книге мы пишем кватернионы так: [*x y z w*]. Это отличается от принятой во многих научных работах по кватернионам нормы [*w x y z*] как расширения комплексных чисел. Наши условные обозначения — это попытка совмещения с обычной практикой записи однородных векторов как [*x y z* 1] (с *w* = 1 в конце). Академическая норма возникает из параллели между кватернионами и комплексными числами. Регулярные двухмерные комплексные числа обычно записываются как $c = a + jb$, соответствующие кватернионные обозначения имеют вид $q = w + ix + jy + kz$. Так что будьте внимательны — убедитесь, что знаете, какая норма используется, прежде чем погрузиться с головой в статьи!

5.4.5. Линейная интерполяция вращения

Интерполяция вращения имеет множество применений в анимации, динамике и системах камер игрового движка. С помощью кватернионов вращение можно легко интерполировать так же, как векторы и точки.

Самый простой и наименее затратный в вычислительном плане подход заключается в четырехмерной векторной LERP для кватернионов, которые вы хотите интерполировать. Для кватернионов q_A и q_B, представляющих вращение *A* и *B*, мы можем найти промежуточное вращение q_{LERP}, составляющее β % пути от *A* до *B*, следующим образом:

$$\mathrm{q}_{\text{LERP}} = \text{LERP}\left(\mathrm{q}_A, \mathrm{q}_B, \beta\right) = \frac{(1-\beta)\mathrm{q}_A + \beta\mathrm{q}_B}{\left|(1-\beta)\mathrm{q}_A + \beta\mathrm{q}_B\right|} = \text{normalize}\left(\begin{bmatrix} (1-\beta)q_{Ax} + \beta q_{Bx} \\ (1-\beta)q_{Ay} + \beta q_{By} \\ (1-\beta)q_{Az} + \beta q_{Bz} \\ (1-\beta)q_{Aw} + \beta q_{Bw} \end{bmatrix}^{\mathrm{T}}\right).$$

Обратите внимание на то, что полученный интерполированный кватернион должен был быть перенормирован. Это необходимо, потому что операция LERP не сохраняет длину вектора в целом.

С геометрической точки зрения $q_{\text{LERP}} = \text{LERP}(q_A, q_B, \beta)$ — кватернион, положение которого находится в β % расстояния от положения *A* до положения *B*, как по-

казано (для наглядности в двух измерениях) на рис. 5.23. Математически операция LERP приводит к *средневзвешенному значению* двух кватернионов с весами $(1 - \beta)$ и β (обратите внимание: в сумме эти два веса дают 1).

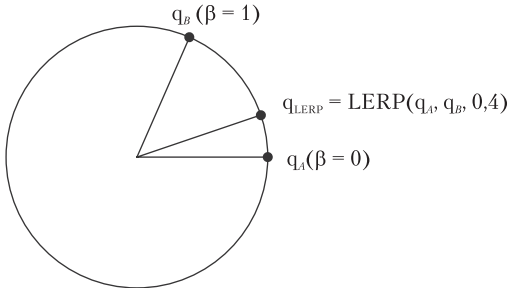


Рис. 5.23. Линейная интерполяция (LERP) между кватернионами q_A и q_B

Сферическая линейная интерполяция

Проблема с операцией LERP состоит в том, что она не учитывает: кватернионы — это на самом деле точки на четырехмерной *гиперсфере*. LERP фактически интерполируется вдоль *хорды* гиперсферы, а не вдоль ее поверхности. Из-за этого анимация вращения не имеет постоянной угловой скорости, притом что параметр β изменяется с постоянной скоростью. Вращение будет медленнее в крайних точках и быстрее — в середине анимации.

Чтобы решить эту проблему, можно использовать вариант операции LERP, известный как *сферическая линейная интерполяция* или SLERP. Операция SLERP задействует синусы и косинусы для интерполяции вдоль *большого круга* четырехмерной гиперсферы, а не вдоль хорды (рис. 5.24). Это приводит к постоянной угловой скорости, когда β изменяется с постоянной частотой.

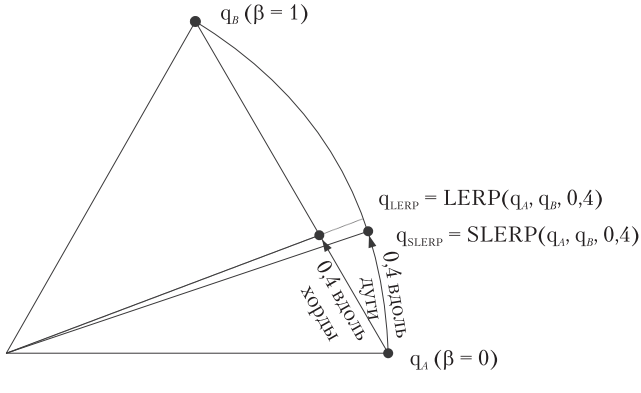


Рис. 5.24. Сферическая линейная интерполяция вдоль большой дуги окружности четырехмерной гиперсферы

Формула SLERP аналогична формуле LERP, но веса $(1 - \beta)$ и β заменяются весами w_p и w_q, включающими синусы угла между двумя кватернионами:

$$\mathrm{SLERP}(\mathrm{p}, \mathrm{q}, \beta) = w_p \mathrm{p} + w_q \mathrm{q},$$

где

$$w_p = \frac{\sin(1-\beta)\theta}{\sin\theta}; \; w_q = \frac{\sin\beta\theta}{\sin\theta}.$$

Косинус угла между любыми двумя кватернионами единичной длины можно найти, взяв их четырехмерное скалярное произведение. Узнав $\cos\theta$, мы можем довольно легко вычислить угол θ и синусы, которые нам нужны:

$$\cos\theta = \mathrm{p} \cdot \mathrm{q} = p_x q_x + p_y q_y + p_z q_z + p_w q_w;$$
$$\theta = \cos^{-1}(\mathrm{p} \cdot \mathrm{q}).$$

SLERP или не SLERP, вот в чем все еще вопрос

До сих пор не решено, использовать ли SLERP в игровом движке. Джонатан Блоу написал отличную статью, в которой утверждается, что SLERP слишком дорогая, а качество LERP не так уж и плохо, поэтому, по его мнению, мы должны знать SLERP, но избегать ее применения в игровых движках (см. http://number-none.com/product/Understanding Slerp, Then Not Using It/index.html).

В то же время некоторые из моих коллег из Naughty Dog обнаружили, что удачная реализация SLERP работает почти так же хорошо, как и LERP. (Например, в SPU для PS3 реализация SLERP, выполненная командой Ice от Naughty Dog, занимает 20 циклов на соединение, в то время как реализация LERP — 16,25 цикла на соединение.) Поэтому я рекомендую вам взвесить все за и против реализации SLERP и LERP, прежде чем принимать решение. Если снижение производительности для SLERP не является неприемлемым, используйте ее, потому что это может обеспечить немного более симпатичную анимацию. Но если ваша SLERP медленная и вы не можете ускорить ее или у вас просто нет времени для этого, то LERP обычно вполне подходит для большинства целей.

5.5. Сравнение представлений вращения

Мы уже видели, что вращения могут быть представлены разными способами. В этом разделе собраны наиболее распространенные представления вращений и выделены их плюсы и минусы. Ни одно представление не идеально для всех ситуаций. Используя приведенную здесь информацию, вы сможете выбрать лучшее представление для конкретного приложения.

5.5.1. Углы Эйлера

Мы кратко рассмотрели *углы Эйлера* в подразделе 5.3.9. Вращение, представленное углами Эйлера, состоит из трех скалярных значений: поворота, наклона и вращения. Эти величины иногда показывают трехмерным вектором $[\theta_Y \theta_P \theta_R]$.

Преимуществами этого представления являются его простота, небольшой размер (три числа с плавающей запятой) и интуитивная понятность — поворот, наклон и вращение легко представить. Также вы можете легко интерполировать простые вращения вокруг одной оси. Например, довольно просто найти промежуточные вращения между двумя различными углами поворота линейной интерполяцией скаляра θ_Y. Однако углы Эйлера не могут быть легко интерполированы, когда вращение происходит вокруг произвольно ориентированной оси.

Кроме того, для углов Эйлера характерно состояние, известное как *складывание рамок*. Это происходит, когда поворот на 90° заставляет одну из трех главных осей свалиться на другую главную ось. Например, если вы поворачиваете на 90° вокруг оси *X*, ось *Y* сваливается на ось *Z*. Это предотвращает любые дальнейшие вращения вокруг исходной оси *Y*, потому что вращения вокруг *Y* и *Z* фактически стали эквивалентными.

Другая проблема с углами Эйлера состоит в том, что порядок, в котором вращения выполняются вокруг каждой оси, имеет значение. Возможен порядок PYR, YPR, RYP и т. д., и все они могут создавать разные составные вращения. Не существует одного стандартного порядка вращения углов Эйлера для всех дисциплин, хотя некоторые дисциплины придерживаются определенных норм. Таким образом, углы вращения $[\theta_Y \theta_P \theta_R]$ не определяют однозначно какое-то конкретное вращение — нужно знать порядок вращения, чтобы правильно интерпретировать эти числа.

И последняя проблема с углами Эйлера состоит в том, что они зависят от отображения осей *X*, *Y* и *Z* на *фронтальное*, *левое/правое* и *верхнее* направления для вращаемого объекта. Например, поворот всегда определяется как вращение вокруг верхней оси, но без дополнительной информации мы не можем сказать, соответствует он вращению вокруг *X*, *Y* или *Z*.

5.5.2. Матрицы 3×3

Матрица 3×3 по ряду причин является удобным и эффективным представлением вращения. Она не страдает от складывания рамок и может уникальным образом представлять произвольные повороты. Вращения могут быть применены к точкам и векторам посредством простого умножения матриц, то есть серии скалярных произведений. Большинство центральных процессоров и все графические процессоры теперь имеют встроенную поддержку аппаратно ускоренных скалярных произведений и умножений матриц. Вращения можно направить обратно, найдя обратную матрицу, что для чистой матрицы вращения — то же самое, что найти транспонированную, то есть довольно простая операция. И матрицы 4×4 предлагают способ

представления произвольных аффинных преобразований — вращений, трансляций и масштабирования — полностью согласованным способом.

Однако матрицы вращения не особенно интуитивно понятны. Большая таблица с числами не сильно помогает представить соответствующее преобразование в трехмерном пространстве. Кроме того, матрицы вращения не так уж легко интерполируются. И наконец, матрица вращения занимает много места (девять чисел с плавающей запятой) по сравнению с углами Эйлера (три числа с плавающей запятой).

5.5.3. Ось + угол

Мы можем представить вращение в виде единичного вектора, определяющего ось вращения и скаляр для угла вращения. Это известно как представление «ось + угол», иногда оно обозначается четырехмерным вектором $[\mathbf{a}\ \theta] = [a_x\ a_y\ a_z\ \theta]$, где **a** — ось вращения, θ — угол в радианах. В правосторонней системе координат направление положительного вращения определяется правилом правой руки, а в левосторонней — правилом левой руки.

Преимущества представления «ось + угол» заключаются в том, что оно довольно интуитивно понятно и компактно. (Требуются только четыре числа с плавающей запятой, в отличие от девяти, необходимых для матрицы 3×3.)

Одно важное ограничение представления «ось + угол» состоит в том, что вращения не могут быть легко интерполированы. Кроме того, вращения в этом формате невозможно применить к точкам и векторам напрямую — сначала нужно преобразовать представление «ось + угол» в матрицу или кватернион.

5.5.4. Кватернионы

Как мы уже видели, кватернион единичной длины может представлять трехмерные вращения способом, аналогичным представлению «ось + угол». Основное различие между этими двумя представлениями состоит в том, что ось вращения кватерниона масштабируется синусом половины угла вращения и вместо сохранения угла в четвертом компоненте вектора мы сохраняем косинус половины угла.

Формулировка кватерниона дает два огромных преимущества по сравнению с представлением «ось + угол». Во-первых, она позволяет объединять вращения и применять их непосредственно к точкам и векторам с помощью умножения кватернионов. Во-вторых, она позволяет легко интерполировать вращения с помощью простых операций LERP или SLERP. Небольшой размер кватерниона (четыре числа с плавающей точкой) также является преимуществом по сравнению с матричной формулировкой.

5.5.5. Преобразования SRT

Сам по себе кватернион может представлять только вращение, тогда как матрица 4×4 может представлять произвольное аффинное преобразование (вращение, трансляцию и масштаб). Когда кватернион объединяется с *вектором трансляции*

и *коэффициентом масштабирования* (либо скаляр для однородного масштабирования, либо вектор для неоднородного), мы получаем подходящую альтернативу матричному представлению аффинных преобразований. Иногда его называют *преобразованием SRT*, потому что оно содержит коэффициент масштабирования (scale), кватернион вращения (rotation) и вектор трансляции (translation) (также иногда его называют SQT, потому что вращение является кватернионом (quaternion)):

$$\mathrm{SRT} = \begin{bmatrix} s & \mathrm{q} & \mathbf{t} \end{bmatrix}$$

(однородная шкала s) или:

$$\mathrm{SRT} = \begin{bmatrix} \mathbf{s} & \mathrm{q} & \mathbf{t} \end{bmatrix}$$

(неоднородный вектор масштабирования $\mathbf{s}$).

Преобразования SRT широко используются в компьютерной анимации из-за их небольшого размера (восемь чисел с плавающей запятой или десять чисел с неоднородной шкалой, в отличие от 12 чисел с плавающей запятой, необходимых для матрицы 4×3) и способности легко интерполироваться. Вектор трансляции и масштабный коэффициент интерполируются через LERP, а кватернион может быть интерполирован либо LERP, либо SLERP.

5.5.6. Двойные кватернионы

Жесткое преобразование, движение «штопор» — это преобразование, включающее в себя вращение и трансляцию. Такие преобразования широко распространены в анимации и робототехнике. Жесткое преобразование может быть представлено с помощью математического объекта, известного как *двойной кватернион*. Представление в виде двойного кватерниона дает ряд преимуществ по сравнению с обычным представлением «вектор — кватернион». Основное преимущество заключается в том, что линейное интерполяционное смешивание может выполняться с постоянной скоростью, кратчайшим путем и без изменений по координатам, аналогично использованию LERP для векторов трансляции и SLERP — для кватернионов вращения (см. подраздел 5.4.5), но в некотором смысле это легко обобщается для смешиваний, включающих в себя три или более преобразования.

Двойной кватернион подобен обычному кватерниону, за исключением того, что его четыре компонента являются *дуальными числами*, а не обычными вещественными числами. Дуальное число может быть записано как сумма недвойственной и двойственной частей следующим образом: $\hat{a} = a + \varepsilon b$. Здесь ε — волшебное число, называемое *дуальной единицей*, определенное таким образом, что $\varepsilon^2 = 0$ (но сама ε не равна нулю). Это аналогично мнимому числу $j = \sqrt{-1}$, которое используется при записи комплексного числа в виде суммы действительной и мнимой частей: $c = a + jb$.

Поскольку каждое дуальное число может быть представлено двумя действительными числами (недуальной и дуальной частями a и b), двойной кватернион может быть представлен вектором из восьми элементов. Его можно представить

также в виде суммы двух обычных кватернионов, в которой второй умножается на дуальную единицу следующим образом: $\hat{q} = q_a + \varepsilon q_b$.

Мы не будем слишком глубоко рассматривать дуальные числа и двойные кватернионы. Есть превосходная статья Ладислава Кавана *Dual Quaternions for Rigid Transformation Blending* («Двойные кватернионы для смешивания с жесткой трансформацией»), в которой описываются теория и практика использования двойных кватернионов для представления жестких преобразований. Она доступна в Интернете: https://bit.ly/2vjD5sz. Обратите внимание на то, что в этой статье дуальное число записывается в виде $\hat{a} = a_0 + \varepsilon a_\varepsilon$, в то время как я раньше применял $a + \varepsilon b$, чтобы подчеркнуть схожесть двойных и комплексных чисел[1].

5.5.7. Вращения и степени свободы

Термин *«степени свободы»* (degrees of freedom, DOF) относится к числу взаимно независимых способов изменения физического состояния объекта (положения и ориентации). Возможно, вы встречали фразу «шесть степеней свободы» в таких областях, как механика, робототехника и аэронавтика. Это говорит о том, что трехмерный объект, движение которого искусственно не ограничено, имеет три степени свободы в перемещении вдоль осей *X*, *Y* и *Z* и три степени свободы — во вращении по осям *X*, *Y* и *Z*, в общей сложности шесть степеней свободы.

Концепция DOF поможет понять, как различные представления вращения могут использовать разное количество параметров с плавающей точкой, но при этом определять вращения только с тремя степенями свободы. Например, для углов Эйлера требуются три числа с плавающей запятой, но представления «ось + угол» и кватернионы используют четыре числа с плавающей запятой, а матрица 3 × 3 занимает девять чисел с плавающей запятой. Как все эти представления могут описывать 3-DOF-вращения (вращения в трех степенях свободы)?

Ответ заключается в *ограничениях*. Все представления трехмерных вращений используют три и более параметра с плавающей точкой, но некоторые представления имеют одно или несколько ограничений на эти параметры. Ограничения означают, что параметры не являются *независимыми*: изменение одного из них требует изменения других, чтобы сохранить действие ограничения (ограничений). Если мы вычтем количество ограничений из числа параметров с плавающей запятой, то получим число степеней свободы. Оно всегда должно быть равно трем для трехмерного вращения:

$$N_{\text{DOF}} = N_{\text{parameters}} - N_{\text{constraints}}. \tag{5.10}$$

Далее показано применение уравнения (5.10) для каждого из представлений вращения, с которыми мы столкнулись в этой книге.

---

[1] Лично я предпочел бы символ a_1, а не a_0, чтобы дуальное число записывалось $\hat{a} = (1)a_1 + (\varepsilon)a_\varepsilon$. Точно так же когда мы строим комплексное число в комплексной плоскости, то воспринимаем реальную единицу как базис-вектор, направленный вдоль действительной оси, а двойственную единицу ε — как базис-вектор, направленный вдоль двойной оси.

- *Углы Эйлера*: 3 параметра – 0 ограничений = 3 DOF.
- *«Ось + угол»*: 4 параметра – 1 ограничение = 3 DOF. Ограничение: ось ограничена единичной длиной.
- *Кватернион*: 4 параметра – 1 ограничение = 3 DOF. Ограничение: кватернион ограничен единичной длиной.
- *3 × 3 матрица*: 9 параметров – 6 ограничений = 3 DOF. Ограничения: все три строки и все три столбца должны иметь единичную длину (если рассматривать их как трехэлементные векторы).

5.6. Другие полезные математические объекты

Как игровые инженеры, мы встретим не только точки, векторы, матрицы и кватернионы, но и множество других математических объектов. В этом разделе кратко рассмотрены наиболее распространенные из них.

5.6.1. Прямые, лучи и отрезки

Бесконечная прямая может быть представлена точкой $\mathbf{P}_0$ и единичным вектором $\mathbf{u}$ в направлении прямой. *Параметрическое представление* прямой прослеживает каждую возможную точку $\mathbf{P}$ вдоль линии от начальной точки $\mathbf{P}_0$ на произвольном расстоянии t вдоль направления единичного вектора $\mathbf{v}$. Бесконечно большой набор точек $\mathbf{P}$ становится *векторной функцией* скалярного параметра t:

$$\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{u}, \quad -\infty < t < \infty. \tag{5.11}$$

Это изображено на рис. 5.25.

Луч — это прямая, которая простирается до бесконечности только в одном направлении. Это легко выразить как $\mathbf{P}(t)$ с ограничением $t \geq 0$ (рис. 5.26).

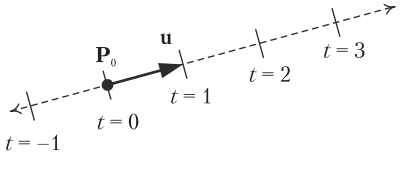


Рис. 5.25. Параметрическое представление прямой

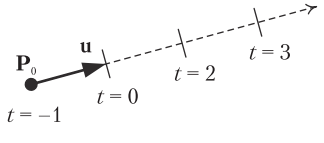


Рис. 5.26. Параметрическое представление луча

Отрезок ограничен на обоих концах, $\mathbf{P}_0$ и $\mathbf{P}_1$. Он также может быть представлен как $\mathbf{P}(t)$ любым из следующих двух способов (где $\mathbf{L} = \mathbf{P}_1 - \mathbf{P}_0$, $L = |\mathbf{L}|$ — длина отрезка, $\mathbf{u} = (1/L)\mathbf{L}$ — единичный вектор в направлении $\mathbf{L}$):

- $\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{u}$, где $0 \leq t \leq L$;
- $\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{L}$, где $0 \leq t \leq 1$.

Последний формат (рис. 5.27) особенно удобен, поскольку параметр t нормализован. Другими словами, t всегда ограничен нулем и единицей, независимо от того, с каким конкретным отрезком мы имеем дело. Это означает, что нам не нужно хранить ограничение L в отдельном параметре с плавающей точкой — оно уже зашифровано в векторе $\mathbf{L} = L\mathbf{u}$, который мы все равно должны хранить.

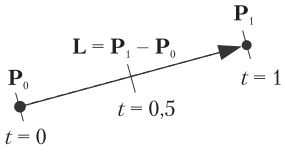


Рис. 5.27. Параметрическое уравнение отрезка с нормированным параметром t

5.6.2. Сферы

Сферы используются везде в программировании игрового движка. Сфера обычно определяется центральной точкой **C** и радиусом r (рис. 5.28). Это очень удобно укладывается в вектор из четырех элементов $[C_x\ C_y\ C_z\ r]$. Как мы видели, когда обсуждали обработку вектора SIMD, есть определенные преимущества в возможности поместить данные в вектор, содержащий четыре 32-битных числа с плавающей точкой (то есть упакованные 128 бит).

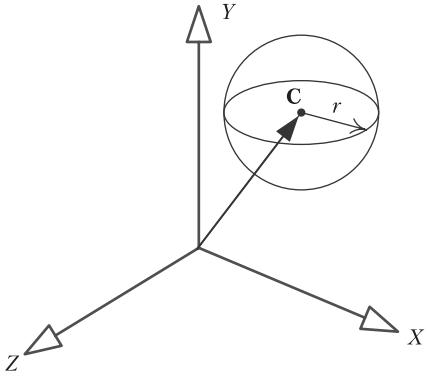


Рис. 5.28. Представление сферы с помощью точки и радиуса

5.6.3. Плоскости

Плоскость — это двухмерная поверхность в трехмерном пространстве. Как вы помните из курса школьной алгебры, уравнение плоскости часто записывается следующим образом:

$$Ax + By + Cz + D = 0.$$

Это уравнение выполняется только для геометрического места точек $\mathbf{P} = [x\ y\ z]$, лежащих на плоскости.

Плоскости могут быть представлены точкой $\mathbf{P}_0$ и единичным вектором **n**, нормальным для плоскости. Эту ситуацию иногда называют *представлением с помощью точки и вектора нормали* (рис. 5.29).

Интересно отметить: когда параметры A, B и C из традиционного уравнения плоскости интерпретируются как трехмерный вектор, этот вектор лежит в направлении нормали плоскости. Если вектор $[A\ B\ C]$ нормализован к единичной длине, то нормализованный вектор $[a\ b\ c] = \mathbf{n}$ и нормализованный параметр $d = D/\sqrt{A^2 + B^2 + C^2}$ — это просто расстояние от плоскости до начала координат. d положительное, если вектор нормали плоскости $\mathbf{n}$ направлен к началу координат (то есть начало координат находится на передней стороне плоскости), и отрицательное, если нормаль указывает от начала координат (то есть начало координат находится позади плоскости).

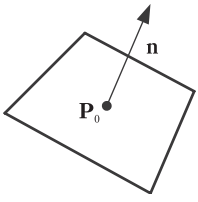


Рис. 5.29. Плоскость, представленная с помощью точки и вектора нормали

Можно взглянуть на это и по-другому: уравнение плоскости и представление с помощью точки и вектора нормали — это два способа написания одного и того же уравнения. Допустим, надо проверить, лежит ли произвольная точка $\mathbf{P} = [x\ y\ z]$ на плоскости. Для этого мы находим расстояние со знаком от точки $\mathbf{P}$ до начала координат по нормали $\mathbf{n} = [a\ b\ c]$, и если оно равно расстоянию со знаком $d = -\mathbf{n} \cdot \mathbf{P}_0$ от плоскости от начала координат, то $\mathbf{P}$ лежит на плоскости. Итак, давайте уравняем их и рассмотрим некоторые термины:

$$\mathbf{n} \cdot \mathbf{P} = \mathbf{n} \cdot \mathbf{P}_0$$

(расстояние со знаком от $\mathbf{P}$ до начала координат) = (расстояние со знаком от $\mathbf{P}_0$ плоскости до начала координат);

$$\mathbf{n} \cdot \mathbf{P} - \mathbf{n} \cdot \mathbf{P}_0 = 0;$$

$$ax + by + cz - \mathbf{n} \cdot \mathbf{P}_0 = 0;$$

$$ax + by + cz + d = 0. \tag{5.12}$$

Уравнение (5.12) верно только тогда, когда точка $\mathbf{P}$ лежит на плоскости. Но что происходит, когда точка $\mathbf{P}$ *не* лежит на плоскости? В этом случае левая часть уравнения плоскости ($ax + by + cz$, что равно $\mathbf{n} \cdot \mathbf{P}$) показывает, насколько далеко точка находится от плоскости. Это выражение вычисляет разницу между расстоянием от $\mathbf{P}$ до начала координат и расстоянием от плоскости до начала координат. Другими словами, левая часть уравнения (5.12) дает длину перпендикуляра h между точкой и плоскостью! Это просто другой способ написать уравнение (5.2):

$$h = \left(\mathbf{P} - \mathbf{P}_0\right) \cdot \mathbf{n};$$

$$h = ax + by + cz + d. \tag{5.13}$$

Плоскость может быть помещена в вектор из четырех элементов, так же как сфера. Чтобы это сделать, заметим, что для однозначного описания плоскости понадобятся только вектор нормали $\mathbf{n} = [a\ b\ c]$ и расстояние от начала координат d. Вектор из четырех элементов $\mathbf{L} = [\mathbf{n}\ d] = [a\ b\ c\ d]$ — это компактный и удобный способ представления и сохранения плоскости в памяти. Обратите внимание: когда

P записывается в однородных координатах с $w = 1$, уравнение $(\mathbf{L} \cdot \mathbf{P}) = 0$ является еще одним способом записи $(\mathbf{n} \cdot \mathbf{P}) = -d$. Эти уравнения выполняются для всех точек **P**, лежащих на плоскости **L**.

Плоскости, определенные в форме вектора из четырех элементов, могут быть легко преобразованы из одного координатного пространства в другое. Имея матрицу $\mathbf{M}_{A\to B}$, которая преобразует точки и векторы (ненормированные) из пространства *A* в пространство *B*, мы уже знаем, что для преобразования *вектора нормали*, такого как вектор **n** плоскости, нужно использовать обратное транспонирование этой матрицы, $(\mathbf{M}_{A\to B}^{-1})^{\mathrm{T}}$. Так что не должно быть большим сюрпризом применение обратного транспонирования матрицы к вектору из четырех элементов плоскости **L**, которое в итоге правильно трансформирует эту плоскость из пространства *A* в пространство *B*. Здесь мы не будем выводить или доказывать этот результат, подробное объяснение того, почему этот маленький трюк работает, приведено в [32, подраздел 4.2.3].

5.6.4. Параллельные осям ограничивающие параллелепипеды

Параллельный осям ограничивающий параллелепипед (axis-aligned bounding boxes (AABB)) — это трехмерный *кубоид*, шесть прямоугольных граней которого выровнены по взаимно ортогональным осям конкретной системы координат. Таким образом, AABB может быть представлен шестиэлементным вектором, содержащим минимальные и максимальные координаты вдоль каждой из трех главных осей, $[x_{\min}, y_{\min}, z_{\min}, x_{\max}, y_{\max}, z_{\max}]$, или две точки, $\mathbf{P}_{\min}$ и $\mathbf{P}_{\max}$.

Это простое представление позволяет особенно удобно и незатратно проверить, находится точка **P** внутри или снаружи любого данного AABB. Мы просто проверяем, выполняются ли все следующие условия:

$$P_x \geq x_{\min};\ P_x \leq x_{\max};$$
$$P_y \geq y_{\min};\ P_y \leq y_{\max};$$
$$P_z \geq z_{\min};\ P_z \leq z_{\max}.$$

Поскольку тесты на пересечение очень быстрые, AABB часто используются для ранней проверки на столкновение: если AABB двух объектов не пересекаются, то нет необходимости проводить более подробный (и более дорогой) тест на столкновение.

5.6.5. Ориентированные ограничивающие параллелепипеды

Ориентированный ограничивающий параллелепипед (oriented bounding boxes, OBB) — это *кубоид*, логически выровненный по отношению к объекту, который он ограничивает. Обычно OBB выравнивается по осям локального пространства

объекта. Следовательно, он действует как AABB в локальном пространстве, хотя не обязательно совпадает с осями пространства мира.

Существуют различные методы для проверки того, находится ли точка в OBB, но один общий подход состоит в том, чтобы преобразовать точку в выровненную систему координат OBB и затем использовать проверку на столкновение с помощью AABB, как описано ранее.

5.6.6. Усеченная пирамида

Как показано на рис. 5.30, *frustum* — это группа из шести плоскостей, которые имеют *форму усеченной пирамиды*. Это обычное явление в 3D-рендеринге, потому что она удобно ограничивает видимую область 3D-мира при визуализации через перспективную проекцию с точки зрения виртуальной камеры. Четыре плоскости ограничивают края пространства экрана, другие две — это ближняя и дальняя отсекающие плоскости (они определяют минимальные и максимальные координаты z для любой видимой точки).

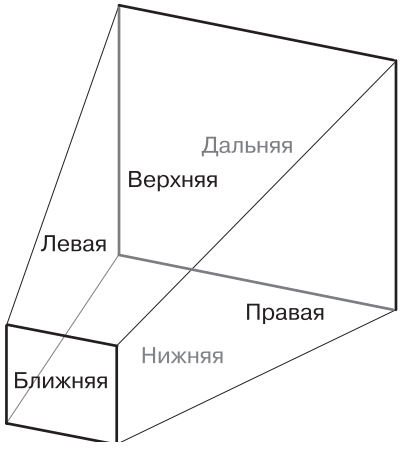


Рис. 5.30. Усеченная пирамида (frustum)

Удачным представлением усеченной пирамиды является массив из шести плоскостей, каждая из которых представлена с помощью точки и вектора нормали (то есть одной точки и одного вектора нормали на плоскость).

Проверка того, находится ли точка внутри усеченной пирамиды, довольно сложна, ее основная идея состоит в том, чтобы использовать скалярные произведения для определения того, находится точка на фронтальной или обратной стороне каждой плоскости. Если она лежит внутри всех шести плоскостей, она находится внутри усеченной пирамиды.

Есть один полезный трюк: при проверке точки пространства мира преобразовать ее, применив к ней перспективную проекцию камеры. Это переносит точку из пространства мира в пространство, известное как *однородное усеченное*

пространство. В нем усеченная пирамида представляет собой просто выровненный по оси кубоид (AABB). Это позволяет выполнять намного более простые тесты на пересечение.

5.6.7. Выпуклые многогранные области

Выпуклая многогранная область определяется произвольным набором плоскостей, у каждой из которых есть нормаль, указывающая внутрь или наружу. Проверка того, находится точка внутри или вне объема, определенного плоскостями, относительно проста. Она похожа на тест усеченной пирамиды, но, возможно, с большим количеством плоскостей. Выпуклые области очень полезны при реализации объектов произвольной формы для триггеров в играх. Многие движки применяют данную технику: например, *кисты*, используемые повсеместно в движке Quake, — это просто объемы, ограниченные плоскостями именно таким образом.

5.7. Генерация случайных чисел

Случайные числа широко задействуются в игровых движках, поэтому нам следует взглянуть на два наиболее распространенных генератора случайных чисел (ГСЧ): линейный конгруэнтный генератор и вихрь Мерсенна. Важно понимать, что ГСЧ на самом деле не генерируют случайные числа — они просто создают сложную, но полностью детерминированную заранее определенную последовательность значений. По этой причине мы называем последовательности, которые они генерируют, *псевдослучайными* и формально должны были бы называть их генераторами псевдослучайных чисел (ГПСЧ). Хороший генератор от плохого отличает то, насколько длинна последовательность чисел до того, как она начинает повторяться (ее *период*), и насколько хорошо последовательности выдерживают различные известные тесты на случайность.

5.7.1. Линейные конгруэнтные генераторы

Линейные конгруэнтные генераторы — это очень быстрый и простой способ генерации последовательности псевдослучайных чисел. В зависимости от платформы этот алгоритм иногда используется в функции `rand()` стандартной библиотеки C. Однако некоторые факторы могут отличаться, поэтому не рассчитывайте, что `rand()` будет основана на каком-либо конкретном алгоритме. Если вы хотите быть полностью уверены в работе ГСЧ, то вам, возможно, стоит реализовать собственный.

Линейный конгруэнтный алгоритм подробно объясняется в книге *Numerical Recipes in C*, поэтому я не буду здесь вдаваться в подробности. Скажу только, что этот ГСЧ не дает особо высококачественных псевдослучайных последовательностей. При одинаковых начальных значениях последовательности всегда будут совершенно одинаковыми. Полученные числа не соответствуют многим критериям,

которые часто желательны, таким как длинный период, младшие и старшие биты, имеющие одинаково длинные периоды, и отсутствие последовательной или пространственной корреляции между сгенерированными значениями.

5.7.2. Вихрь Мерсенна

Алгоритм генерации псевдослучайных чисел вихрь Мерсенна специально разработан для исправления различных проблем линейного конгруэнтного алгоритма. «Википедия» дает следующее описание преимуществ алгоритма.

- Он был создан так, чтобы иметь огромный период $2^{19\,937} - 1$ (создатели алгоритма подтвердили это). В действительности нет особых причин использовать такие большие периоды, так как большинство приложений не требует $2^{19\,937}$ уникальных комбинаций ($2^{19\,937} \approx 4{,}3 \cdot 10^{6001}$).
- Алгоритм имеет очень высокий порядок равномерного распределения. Обратите внимание: это означает, что между последовательными значениями в получаемой последовательности имеется незначительная автокорреляция.
- Он проходит многочисленные тесты на статистическую случайность, в том числе строгие тесты Дихарда.
- Он быстр.

В Интернете доступны различные реализации вихря Мерсенна, в том числе одна особенно крутая, использующая векторные инструкции SIMD для дополнительного ускорения, которая называется SFMT (SIMD-oriented fast Mersenne Twister). SFMT можно загрузить с www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html.

5.7.3. Мать всего, Xorshift и KISS99

В 1994 году Джордж Марсалья, исследователь в области информатики и математики, наиболее известный разработкой набора тестов для измерения качества набора случайных чисел Дихарда (http://www.stat.fsu.edu/pub/diehard), опубликовал алгоритм генерации псевдослучайных чисел, который работает быстрее, чем алгоритм Мерсенна, и гораздо проще в реализации. Ученый утверждал, что он может дать последовательность 32-битных псевдослучайных чисел с периодом 2250. Алгоритм прошел все тесты Дихарда и до сих пор считается одним из лучших генераторов псевдослучайных чисел для высокоскоростных приложений. Марсалья назвал свой алгоритм «мать всех генераторов псевдослучайных чисел» (mother of all pseudorandom number generators), так как считал, что это единственный ГСЧ, который кому-либо когда-либо понадобится.

Позже Марсалья опубликовал еще один генератор, Xorshift, который по качеству набора случайных чисел находится между вихрем Мерсенна и «матерью всех...», но работает немного быстрее, чем последний.

Марсалья также разработал серию генераторов случайных чисел, названную KISS (keep it simple stupid). Алгоритм KISS99 довольно популярен, поскольку

имеет большой период (2123) и проходит все тесты в наборе тестов TestU01 (bit.ly/2r5FmSP).

Вы можете прочитать о Джордже Марсалье на странице en.wikipedia.org/wiki/George_Marsaglia, а о генераторе «мать всех...» — на страницах ftp.forth.org/pub/C/mother.c и www.agner.org/random. Можете скачать статью Джорджа Марсальи о Xorshift в формате PDF по адресу www.jstatsoft.org/v08/i14/paper.

5.7.4. PCG

Другое очень популярное и качественное семейство генераторов псевдослучайных чисел называется PCG. Оно работает путем объединения конгруэнтного генератора для переходов состояний (CG — congruential generator в PCG) с функциями перестановки для генерации его вывода (P — permutation functions в PCG). Вы можете прочитать больше об этом семействе генераторов псевдослучайных чисел на сайте www.pcg-random.org/.

Часть II

Низкоуровневые системы движка

6 Системы поддержки движка

Любому игровому движку требуются низкоуровневые системы поддержки, которые управляют обыденными, но важными задачами, такими как его запуск и остановка, настройка его и функций игры, управление использованием памяти движком, обработка доступа к файловой системе (системам), предоставление доступа к широкому спектру разнородных типов активов, применяемых в игре (меши, текстуры, анимация, аудио и т. д.). Нужны также инструменты отладки для команды разработки. Эта глава будет посвящена системам поддержки самого нижнего уровня, которые есть в большинстве игровых движков. В следующих главах мы рассмотрим более крупные базовые системы, включая управление ресурсами, устройства взаимодействия с пользователем и внутриигровые инструменты отладки.

6.1. Подсистема запуска и остановки

Игровой движок — это сложное программное обеспечение, состоящее из множества взаимодействующих подсистем. Когда движок запускается впервые, каждая подсистема должна быть сконфигурирована и инициализирована в определенном порядке. Взаимозависимости между ними неявно определяют порядок, в котором они должны быть запущены, то есть если подсистема B зависит от подсистемы A, то необходимо запустить A, прежде чем будет инициализирована B. Завершается работа обычно в обратном порядке, поэтому сначала прекращается выполнение B, затем A.

6.1.1. Порядок статической инициализации C++ (или его отсутствие)

Поскольку языком программирования, используемым в большинстве современных игровых движков, является C++, мы должны кратко рассмотреть вопрос о том, можно ли применять собственные семантики запуска и остановки C++ для запуска и остановки подсистем движка. В C++ глобальные и статические объекты создаются перед вызовом точки входа программы (`main()` или `WinMain()` в Windows). Однако эти конструкторы вызываются в совершенно непредсказуемом порядке.

Деструкторы глобальных и статических экземпляров классов вызываются после возврата из `main()` (или `WinMain()`), опять же в непредсказуемом порядке. Ясно, что такое поведение нежелательно для инициализации и остановки подсистем игрового движка или любой другой программной системы, в которой между глобальными объектами имеются взаимозависимости.

Это несколько прискорбно, потому что общий шаблон проектирования для реализации основных подсистем, таких как системы, составляющие игровой движок, состоит в определении *класса-одиночки* (singleton) (часто называемого *менеджером*) для каждой подсистемы. Если бы C++ давал нам больший контроль над порядком, в котором глобальные и статические экземпляры классов создаются и уничтожаются, мы могли бы определять экземпляры одиночек как глобальные, не нуждаясь в динамическом выделении памяти. Например, мы могли бы написать:

```
class RenderManager
{
public:
    RenderManager()
    {
        // запустить менеджер...
    }

    ~RenderManager()
    {
        // остановить менеджер...
    }

    // ...
};

// экземпляр-одиночка
static RenderManager gRenderManager;
```

Увы, если нет возможности напрямую контролировать порядок создания и уничтожения, этот подход не сработает.

Создание по требованию

Есть в C++ один трюк, который мы можем использовать для своих целей. Статическая переменная, объявленная внутри функции, будет создаваться не до вызова `main()`, а скорее при первом вызове этой функции. Поэтому, если одиночка является функционально-статическим, мы можем контролировать порядок построения глобальных синглтонов:

```
class RenderManager
{
public:
    // Получить один-единственный экземпляр
    static RenderManager& get()
```

```
    {
        // Эта статическая переменная функции будет
        // создана при первом вызове данной функции
        static RenderManager sSingleton;
        return sSingleton;
    }

    RenderManager()
    {
        // Запускаем другие менеджеры, от которых мы
        // зависим, но сначала вызываем их функции get()
        VideoManager::get();
        TextureManager::get();

        // Теперь запускаем менеджер рендеринга
        // ...
    }

    ~RenderManager()
    {
        // Останавливаем менеджер...
        // ...
    }
};
```

Вы обнаружите, что многие учебники по разработке ПО предлагают этот способ или вариант, который включает в себя динамическое размещение класса-одиночки, как показано далее:

```
static RenderManager& get()
{
    static RenderManager* gpSingleton = nullptr;
    if (gpSingleton == nullptr)
    {
        gpSingleton = new RenderManager;
    }
    ASSERT(gpSingleton);
    return *gpSingleton;
}
```

К сожалению, это все еще не позволяет нам контролировать порядок уничтожения. Вполне возможно, что C++ выполнит уничтожение одного из менеджеров, от которых зависит `RenderManager`, как часть процедуры завершения работы до вызова деструктора `RenderManager`. Кроме того, трудно точно предсказать, когда будет создан синглтон `RenderManager`, потому что инициализация произойдет при первом вызове `RenderManager::get()` — кто знает, когда это случится? Более того, программисты, использующие этот класс, могут даже не ожидать, что безобидно выглядящая функция `get()` сделает что-то непростое, например выделит и инициализирует тяжеловесный синглтон. Это непредсказуемый и опасный вариант. Поэтому нам предлагается прибегнуть к более прямому подходу, который дает больший контроль.

6.1.2. Простой работающий подход

Предположим, что мы хотим использовать в своих подсистемах менеджеры-синглтоны. В этом случае самый простой силовой подход состоит в том, чтобы определить явные функции запуска и остановки для каждого класса синглтон-менеджера. Эти функции заменяют конструктор и деструктор, и на самом деле мы должны обеспечить, чтобы конструктор и деструктор *абсолютно ничего* не делали. Таким образом, функции запуска и остановки могут быть явно вызваны в необходимом порядке внутри `main()` (или из какого-то всеобъемлющего объекта синглтона, который управляет движком в целом), например:

```
class RenderManager
{
public:
    RenderManager()
    {
        // Ничего не делаем
    }

    ~RenderManager()
    {
        // Ничего не делаем
    }

    void startUp()
    {
        // Запускаем менеджер...
    }

    void shutDown()
    {
        // Останавливаем менеджер...
    }

    // ...
};

class PhysicsManager    { /* То же самое... */ };

class AnimationManager  { /* То же самое... */ };

class MemoryManager     { /* То же самое... */ };

class FileSystemManager { /* То же самое... */ };

// ...

RenderManager           gRenderManager;
PhysicsManager          gPhysicsManager;
AnimationManager        gAnimationManager;
TextureManager          gTextureManager;
```

```
VideoManager            gVideoManager;
MemoryManager           gMemoryManager;
FileSystemManager       gFileSystemManager;
// ...

int main(int argc, const char* argv)
{
    // Запускаем системы движка в нужном порядке
    gMemoryManager.startUp();
    gFileSystemManager.startUp();
    gVideoManager.startUp();
    gTextureManager.startUp();
    gRenderManager.startUp();
    gAnimationManager.startUp();
    gPhysicsManager.startUp();
    // ...

    // Запускаем игру
    gSimulationManager.run();

    // Останавливаем все в обратном порядке
    // ...
    gPhysicsManager.shutDown();
    gAnimationManager.shutDown();
    gRenderManager.shutDown();
    gFileSystemManager.shutDown();
    gMemoryManager.shutDown();

    return 0;
}
```

Есть и более элегантные способы сделать это. Например, мы могли бы попросить каждого менеджера зарегистрироваться в глобальной очереди, а затем пройти эту очередь, чтобы запустить всех менеджеров в правильном порядке. Вы можете определить граф зависимостей «менеджер — менеджер», попросив каждого менеджера явно перечислить других менеджеров, от которых он зависит, а затем написать код для расчета оптимального порядка запуска с учетом их взаимозависимостей. Вы можете использовать подход конструкции по требованию, описанный ранее. По моему опыту, силовой подход всегда побеждает по следующим причинам.

- Он простой, и его легко реализовать.
- Он очевиден. Вы можете сразу увидеть и понять порядок запуска, просто взглянув на код.
- Его легко отлаживать и поддерживать. Если что-то запускается слишком рано или поздно, вы можете просто переместить одну строку кода.

Небольшим недостатком силового метода запуска и остановки является то, что вы можете случайно остановить все в порядке, не строго противоположном порядку запуска. Но я бы не стал переживать по этому поводу. До тех пор пока вы можете успешно запускать и останавливать подсистемы своего движка, вы молодец.

6.1.3. Некоторые примеры реальных движков

Давайте кратко рассмотрим некоторые примеры запуска и остановки движка, взятые из реальных игровых движков.

OGRE

По признанию авторов, OGRE — движок рендеринга, а не игровой движок как таковой. Но по необходимости он предоставляет множество низкоуровневых функций, присутствующих в полноценных игровых движках, включая простой и элегантный механизм запуска и выключения. Все в OGRE контролируется объектом-синглтоном `Ogre::Root`. Он содержит указатели на все прочие подсистемы в OGRE и управляет их созданием и уничтожением. Это позволяет программисту легко запускать OGRE: нужно просто создать экземпляр `Ogre::Root` — и все готово.

Приведу несколько выдержек из исходного кода OGRE, чтобы вы увидели, как он работает:

OgreRoot.h

```
    class _OgreExport Root : public Singleton<Root>
    {
        // <часть кода опущена...>

        // Синглтоны
        LogManager* mLogManager;
        ControllerManager* mControllerManager;
        SceneManagerEnumerator* mSceneManagerEnum;
        SceneManager* mCurrentSceneManager;
        DynLibManager* mDynLibManager;
        ArchiveManager* mArchiveManager;
        MaterialManager* mMaterialManager;
        MeshManager* mMeshManager;
        ParticleSystemManager* mParticleManager;
        SkeletonManager* mSkeletonManager;
        OverlayElementFactory* mPanelFactory;
        OverlayElementFactory* mBorderPanelFactory;
        OverlayElementFactory* mTextAreaFactory;
        OverlayManager* mOverlayManager;
        FontManager* mFontManager;
        ArchiveFactory *mZipArchiveFactory;
        ArchiveFactory *mFileSystemArchiveFactory;
        ResourceGroupManager* mResourceGroupManager;
        ResourceBackgroundQueue* mResourceBackgroundQueue;
        ShadowTextureManager* mShadowTextureManager;

        // и т. д.
    };
```

OgreRoot.cpp

```
    Root::Root(const String& pluginFileName,
               const String& configFileName,
```

```
            const String& logFileName) :
        mLogManager(0),
        mCurrentFrame(0),
        mFrameSmoothingTime(0.0f),
        mNextMovableObjectTypeFlag(1),
        mIsInitialised(false)
    {
        // суперкласс проверит синглтоны
        String msg;

        // Init
        mActiveRenderer = 0;
        mVersion
            = StringConverter::toString(OGRE_VERSION_MAJOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_MINOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_PATCH)
            + OGRE_VERSION_SUFFIX + " "
            + "(" + OGRE_VERSION_NAME + ")";
        mConfigFileName = configFileName;

        // создаем менеджер журналирования и файл
        // журнала по умолчанию, если его еще нет
        if(LogManager::getSingletonPtr() == 0)
        {
            mLogManager = new LogManager();
            mLogManager->createLog(logFileName, true, true);
        }

        // менеджер динамических библиотек
        mDynLibManager = new DynLibManager();
        mArchiveManager = new ArchiveManager();

        // ResourceGroupManager
        mResourceGroupManager = new ResourceGroupManager();

        // ResourceBackgroundQueue
        mResourceBackgroundQueue
            = new ResourceBackgroundQueue();

        // и т. д.
    }
```

OGRE предоставляет шаблонный базовый класс `Ogre::Singleton`, из которого происходят все его классы-синглтоны (менеджеры). Если вы посмотрите на его реализацию, вы увидите, что `Ogre::Singleton` не использует отложенную инициализацию, а вместо этого полагается на `Ogre::Root` для явного создания каждого нового синглтона. Как мы уже говорили выше, это делается для того, чтобы синглтоны создавались и уничтожались в четко определенном порядке.

Uncharted компании Naughty Dog и серия игр The Last of Us

Движок, созданный Naughty Dog, Inc. для серии игр *Uncharted* и *The Last of Us*, использует аналогичную явную технику для запуска своих подсистем. Посмотрев на следующий код, вы заметите, что запуск движка не всегда является простой последовательностью выделения одноэлементных экземпляров. Во время инициализации движка необходимо запустить широкий спектр служб операционной системы, сторонних библиотек и т. д. Кроме того, динамического выделения памяти следует избегать, поэтому многие синглтоны являются статически распределенными объектами (например, `g_fileSystem`, `g_languageMgr` и т. д.). Это не очень элегантно, но работает.

```
Err BigInit()
{
    init_exception_handler();

    U8* pPhysicsHeap = new(kAllocGlobal, kAlign16)
        U8[ALLOCATION_GLOBAL_PHYS_HEAP];
    PhysicsAllocatorInit(pPhysicsHeap,
        ALLOCATION_GLOBAL_PHYS_HEAP);

    g_textDb.Init();
    g_textSubDb.Init();
    g_spuMgr.Init();

    g_drawScript.InitPlatform();

    PlatformUpdate();

    thread_t init_thr;
    thread_create(&init_thr, threadInit, 0, 30,
        64*1024, 0, "Init");

    char masterConfigFileName[256];
    snprintf(masterConfigFileName,
        sizeof(masterConfigFileName),
        MASTER_CFG_PATH);
    {
        Err err = ReadConfigFromFile(
            masterConfigFileName);
        if (err.Failed())
        {
            MsgErr("Config file not found (%s).\n",
                masterConfigFileName);
        }
    }

    memset(&g_discInfo, 0, sizeof(BootDiscInfo));
    int err1 = GetBootDiscInfo(&g_discInfo);
    Msg("GetBootDiscInfo() : 0x%x\n", err1);
```

```
if(err1 == BOOTDISCINFO_RET_OK)
{
    printf("titleId       : [%s]\n",
        g_discInfo.titleId);
    printf("parentalLevel : [%d]\n",
        g_discInfo.parentalLevel);
}

g_fileSystem.Init(g_gameInfo.m_onDisc);

g_languageMgr.Init();
if (g_shouldQuit) return Err::kOK;

// и т. д.
```

6.2. Управление памятью

Как разработчики игр, мы всегда пытаемся заставить код работать быстрее. Производительность любого программного обеспечения определяется не только применяемыми алгоритмами или эффективностью, с которой они кодируются, но и тем, как программа *использует память* (RAM). Память влияет на производительность двумя способами.

- *Динамическое распределение* памяти с помощью функции `malloc()` или глобального оператора C++ `new` — это очень медленная операция. Мы можем улучшить производительность кода либо вообще избегая динамического выделения, либо задействуя специальные распределители памяти, которые значительно снижают затраты на ее выделение.
- На современных процессорах производительность программного обеспечения часто зависит от *паттернов доступа к памяти*. Как мы увидим, данные, расположенные в небольших смежных блоках памяти, могут обрабатываться ЦП гораздо эффективнее, чем если бы эти же данные были распределены по широкому диапазону адресов памяти. Даже самый эффективный алгоритм, разработанный с особой тщательностью, может быть поставлен на колени, если данные, с которыми он работает, не будут эффективно размещены в памяти.

В этом разделе мы узнаем, как оптимизировать использование памяти для этих критериев.

6.2.1. Оптимизация динамического распределения памяти

Динамическое распределение памяти с помощью функций `malloc()` и `free()` или глобальных операторов `new` и `delete` в C++, также называемых *распределением кучи*, обычно очень медленное. Высокую стоимость процесса можно объяснить двумя основными факторами. Во-первых, распределитель кучи является сред-

ством общего назначения, поэтому ему надо явно указывать необходимость обработки выделенного блока любого размера, от 1 байта до 1 Гбайт. Это требует значительных затрат на управление, что делает функции `malloc()` и `free()` медленными. Во-вторых, в большинстве операционных систем вызов `malloc()` или `free()` должен сначала переключиться из режима пользователя в режим ядра, обработать запрос и затем вернуться в программу. Эти переключатели контекста могут быть чрезвычайно дорогими. При разработке игр стоит придерживаться одного правила: свести количество операций выделений на куче к минимуму и никогда не делать этого в цикле.

Конечно, ни один игровой движок не может полностью избежать динамического выделения памяти, поэтому большинство игровых движков реализуют один или несколько пользовательских распределителей. Пользовательский распределитель может иметь лучшие характеристики производительности, чем распределитель кучи операционной системы, по двум причинам. Во-первых, пользовательский распределитель может удовлетворить запросы из предварительно выделенного блока памяти, который сам выделяется с помощью `malloc()` или `new` или объявляется как глобальная переменная. Это позволяет ему работать в пользовательском режиме и полностью избежать затрат на переключение контекста в операционной системе. Во-вторых, делая различные предположения о паттернах применения, пользовательский распределитель может быть гораздо более эффективным, чем распределитель кучи общего назначения.

В следующих разделах мы рассмотрим некоторые распространенные виды пользовательских распределителей. Для получения дополнительной информации по этой теме прочтите превосходный пост в блоге Кристиана Гирлинга: www.swedishcoding.com/2008/08/31/are-we-out-of-memory.

Стековые распределители

Многие игры выделяют память стековым способом. Каждый раз, когда загружается новый игровой уровень, для него выделяется память. После того как уровень был загружен, происходит небольшое или полное динамическое распределение памяти. При завершении уровня все его данные выгружаются, и эта память может быть освобождена. Имеет смысл использовать стековую структуру данных для такого рода распределения памяти.

Распределитель стека очень прост в реализации. Мы просто выделяем большой непрерывный блок памяти, применяя `malloc()` или глобальный `new` или объявляя глобальный массив байтов (в этом случае память эффективно выделяется из сегмента BSS исполняемого файла). Определяем указатель стека, указывающий на его вершину. Все адреса памяти под этим указателем считаются используемыми, а все адреса над ним — свободными. Указатель стека инициализируется самым низким адресом памяти в стеке. Каждый запрос на выделение памяти просто перемещает указатель вверх на запрошенное количество байтов. Последний выделенный блок можно освободить, просто переместив верхний указатель стека вниз на размер блока.

Важно понимать, что с распределителем стека память не может быть освобождена в произвольном порядке. Все освобождения должны быть выполнены в порядке, противоположном тому, в котором они были выделены. Простой способ обеспечить соблюдение этих ограничений — запретить освобождение отдельных блоков. Вместо этого мы можем предоставить функцию, которая откатывает вершину стека обратно к ранее отмеченному месту, освобождая все блоки между ней и точкой отката.

Важно всегда возвращать указатель стека к точке, лежащей на границе между двумя выделенными блоками, потому что в противном случае новые блоки будут перезаписывать хвостовую часть самого верхнего блока. Чтобы убедиться, что все сделано правильно, распределитель стека часто предоставляет функцию, которая возвращает *указатель* на текущую вершину стека. Затем функция отката принимает один из этих указателей в качестве аргумента. Это подробно изображено на рис. 6.1. Интерфейс распределителя стека часто выглядит примерно так.

```
class StackAllocator
{
public:
    // Указатель стека: Представляет текущую вершину
    // стека. Откатиться можно только до указателя,
    // а не до произвольного места в стеке.
    typedef U32 Marker;

    // Создает распределитель стека заданного размера.
    explicit StackAllocator(U32 stackSize_bytes);

    // Выделяет новый блок заданного размера
    // на вершине стека.
    void* alloc(U32 size_bytes);

    // Возвращает указатель к текущей вершине стека.
    Marker getMarker();

    // Откатывает стек к предыдущему указателю.
    void freeToMarker(Marker marker);

    // Очищает весь стек (сворачивает стек обратно в ноль).
    void clear();

private:
    // ...
};
```

Двухсторонние стековые распределители. Один блок памяти может фактически содержать два распределителя стека — один выделяется снизу вверх, другой — сверху вниз. Двухсторонний распределитель стека полезен, потому что он задействует память более эффективно, позволяя балансировать между использованием памяти нижнего и верхнего стеков. В некоторых ситуациях оба стека могут получать примерно одинаковый объем памяти и встречаться в середине блока.

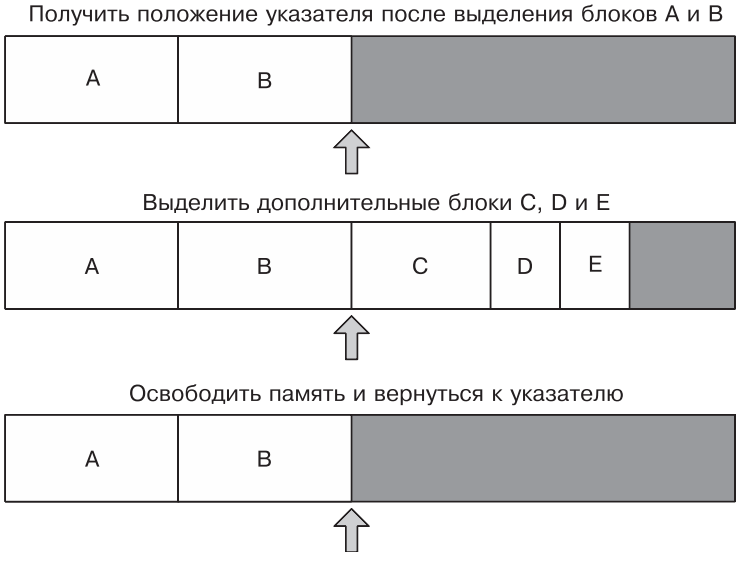


Рис. 6.1. Выделение памяти стека и возврат к указателю

В других ситуациях один из двух стеков может потреблять намного больше памяти, чем другой, но все запросы на выделение могут быть удовлетворены, если общий объем запрошенной памяти не превышает размер блока, совместно используемого двумя стеками (рис. 6.2).

Рис. 6.2. Двухсторонние стековые распределители

В аркадной игре *Hydro Thunder* от Midway вся память выделяется из одного большого блока памяти, управляемого двухсторонним стековым распределителем. Нижний стек используется для загрузки и выгрузки уровней (гоночных треков), а верхний — для временных блоков памяти, которые выделяются и освобождаются на каждом кадре. Эта схема распределения работала очень хорошо и гарантировала, что *Hydro Thunder* никогда не страдал от проблем фрагментации памяти (см. подраздел 6.2.1). Стив Ранк, ведущий инженер *Hydro Thunder*, подробно описывает эту технику распределения в [8, раздел 1.9].

Распределители пула

При программировании игрового движка и разработке программного обеспечения в целом довольно часто выделяют множество маленьких блоков памяти одинакового размера. Например, нам может потребоваться выделить и освободить память

для матрицы, или итератора, или ссылки в связанном списке, или экземпляра визуализируемого меша. Для этого типа шаблона распределения памяти идеальным выбором часто является *распределитель пула*.

Распределитель пула предварительно выделяет большой блок памяти, размер которого кратен размеру создаваемых элементов. Например, пул для матриц 4×4 будет кратен 64 байтам — это 16 элементов на матрицу, умноженные на 4 байта на элемент (при условии, что каждый элемент — это 32-разрядное число с плавающей точкой `float`). Каждый элемент в пуле добавляется в связанный список свободных элементов. Когда пул инициализируется впервые, свободный список содержит все элементы. Всякий раз, когда делается запрос на выделение, мы удаляем следующий свободный элемент из списка свободных и используем его. Когда элемент освобождается, мы просто возвращаем его в список свободных. И выделение, и освобождение являются операциями сложности $O(1)$, поскольку каждая включает в себя только пару манипуляций с указателями, независимо от того, сколько элементов сейчас свободно. (Обозначение $O(1)$ — это пример нотации «большого О». Это означает, что время выделения и освобождения памяти примерно постоянно и не зависит от таких вещей, как количество элементов, находящихся в настоящее время в пуле. См. подраздел 6.3.3, где объясняются обозначения нотации «большого О».)

Связанный список свободных элементов может быть односвязным, что означает: нам нужен один указатель (4 байта на 32-битных машинах или 8 байт на 64-битных) для каждого свободного элемента. Где взять память для всех этих указателей? Конечно, они могут храниться в отдельном предварительно выделенном блоке памяти, занимая `sizeof (void*) * numElementsInPool` байт. Но это слишком расточительно. Ключевым моментом является осознание того, что блоки памяти, находящиеся в свободном списке, по определению свободные. Так почему бы не сохранить каждый указатель на следующий элемент (next) в свободном списке *в самом свободном блоке*? Этот маленький трюк работает до тех пор, пока `elementSize> = sizeof(void *)`. Мы не тратим впустую память, потому что все указатели свободного списка находятся внутри блоков свободной памяти — в памяти, которая в любом случае не использовалась ни для чего!

Если каждый элемент меньше указателя, то мы можем применить индексы элементов пула вместо указателей для реализации связанного списка. Например, если пул содержит 16-разрядные целые числа, то можно использовать 16-разрядные индексы в качестве указателей next в связанном списке. Это работает до тех пор, пока пул содержит не более $2^{16} = 65\,536$ элементов.

Выровненные распределения

Как говорилось в подразделе 3.3.7, ко всем переменным и объектам данных предъявляется требование выравнивания. Восьмиразрядная целочисленная переменная может быть выровнена по любому адресу, но 32-разрядное целочисленное значение или переменная с плавающей точкой должны быть выровнены на 4 байта,

то есть их адрес может заканчиваться только полубайтами 0x0, 0x4, 0x8 или 0xC. Стодвадцативосьмибитное векторное значение SIMD обычно имеет требование выравнивания 16 байт, что означает: его адрес памяти может заканчиваться только полубайтом 0x0. На PS3 блоки памяти, которые должны быть переданы в SPU через контроллер прямого доступа к памяти (direct memory access, DMA), необходимо выровнять на 128 *байт* для обеспечения максимальной пропускной способности DMA, то есть они могут заканчиваться только байтами 0x00 или 0x80.

Все распределители памяти должны быть способны возвращать выровненные блоки памяти. Это довольно легко реализовать. Мы просто выделяем чуть больше памяти, чем было запрошено, немного смещаем адрес блока памяти вверх, чтобы он был правильно выровнен, а затем возвращаем смещенный адрес. Поскольку памяти выделено больше, чем запрошено, возвращаемый блок все равно будет достаточно большим даже при небольшом смещении вверх.

В большинстве реализаций количество выделенных дополнительных байтов равно выравниванию минус один, чтобы получить правильное выравнивание даже в самом худшем случае. Например, если мы хотим выровнять 16-байтовый блок памяти, то худшим случаем будет возвращение невыровненного указателя, заканчивающегося 0x1, поскольку, чтобы привести его к 16-байтовой границе, потребуется сдвиг на 15 байт.

Вот одна из возможных реализаций выровненного распределителя памяти:

```
// Переместить указанный адрес вверх, если/насколько необходимо,
// чтобы убедиться, что он выровнен по заданному количеству байтов.
inline uintptr_t AlignAddress(uintptr_t addr, size_t align)
{
    const size_t mask = align – 1;
    assert((align & mask) == 0); // степень 2
    return (addr + mask) & ~mask;
}

// Переместить указатель вверх, если/насколько необходимо, чтобы
// убедиться, что он выровнен по заданному количеству байтов.
template<typename T>
inline T* AlignPointer(T* ptr, size_t align)
{
    const uintptr_t addr = reinterpret_cast<uintptr_t>(ptr);
    const uintptr_t addrAligned = AlignAddress(addr, align);
    return reinterpret_cast<T*>(addrAligned);
}

// Функция выровненного выделения. ВАЖНО: "выровнять"
// должно быть степенью 2 (обычно 4, 8 или 16).
void* AllocAligned(size_t bytes, size_t align)
{
    // Определяем количество байтов, которое нам понадобится,
    // в наихудшем варианте.
    size_t worstCaseBytes = bytes + align – 1;
```

```
    // Выделяем невыровненный блок памяти.
    U8* pRawMem = new U8[worstCaseBytes];

    // Выравниваем блок.
    return AlignPointer(pRawMem, align);
}
```

Волшебство выравнивания обеспечивается функцией `AlignAddress()`. Вот как это работает: учитывая адрес и желаемое выравнивание L, мы можем выровнять этот адрес по границе L байт, сначала добавив к ней $L - 1$, а затем убрав N младших битов результирующего адреса, где $N = \log_2(L)$. Например, чтобы выровнять любой адрес по 16-байтовой границе, мы сдвигаем его на 15 байт, а затем маскируем $N = \log_2(16) = 4$ младших разряда.

Чтобы удалить эти биты, нужна маска, которую можно применить к адресу, используя побитовый оператор AND. Поскольку L всегда является степенью двойки, $L - 1$ будет маской с двоичными единицами в N младших значащих битах и двоичными нулями во всех остальных местах. Таким образом, нам нужно лишь *инвертировать* эту маску, а затем провести операцию AND с адресом (`addr & ~mask`).

Освобождение выровненных блоков. Позже, когда выровненный блок будет освобожден, нам будет передан смещенный адрес, а не исходный, который мы выделили. Но чтобы освободить память, требуется освободить адрес, который фактически был возвращен методом `new`. Как можно преобразовать выровненный адрес обратно в исходный, невыровненный адрес?

Простой подход — сохранить смещение, то есть разницу между выровненным и исходным адресами, в каком-то месте, где функция освобождения сможет его найти. Напомню, что мы фактически выделяем `align - 1` дополнительный байт в `AllocAligned()`, чтобы обеспечить пространство для выравнивания указателя. Эти дополнительные байты — идеальное место для хранения значения сдвига. Наименьшее смещение, которое мы когда-либо сделаем, составляет 1 байт, так что это минимальное пространство, которое потребуется для хранения смещения. Поэтому, учитывая выровненный указатель `p`, можно просто сохранить сдвиг как однобайтовое значение по адресу `p - 1`.

Здесь есть одно «но»: возможно, необработанный адрес, возвращаемый `new`, уже будет выровнен. В этом случае приведенный ранее код не станет смещать необработанный адрес, то есть не будет никаких дополнительных байтов для хранения смещения. Чтобы преодолеть это, мы выделяем L дополнительных байтов вместо $L - 1$, а затем *всегда* перемещаем необработанный указатель до следующей границы байтов L, даже если он уже выровнен. Теперь максимальный сдвиг будет составлять L байт, а минимальный — 1 байт. Таким образом, всегда будет хотя бы один дополнительный байт, в который мы можем сохранить величину сдвига.

Хранение сдвига в 1 байте работает для выравниваний вплоть до 128. Мы никогда не сдвигаем указатель на 0 байт, поэтому можем заставить эту схему работать до 256-байтового выравнивания, интерпретируя невозможное значение смещения на ноль как на 256-байтовое. (Для большего выравнивания нужно было бы вы-

делить еще больше байтов и еще значительнее сместить указатель вверх, чтобы освободить место для более широкого «заголовка».)

Вот как могут быть реализованы модифицированная функция `AllocAligned()` и соответствующая ей функция `FreeAligned()` (выделение и освобождение выровненных блоков показаны на рис. 6.3):

```
// Функция выровненного выделения памяти. ВАЖНО:
// выравнивание должно быть степенью 2 (обычно 4, 8 или 16).
void* AllocAligned(size_t bytes, size_t align)
{
    // Выделяем выравниванию больше байтов, чем нужно.
    size_t actualBytes = bytes + align;

    // Выделяем невыровненный блок памяти.
    U8* pRawMem = new U8[actualBytes];

    // Выравниваем блок. Если выравнивания не
    // произошло,сдвигаем его на все байты,
    // чтобы всегда было место для хранения смещения.
    U8* pAlignedMem = AlignPointer(pRawMem, align);
    if (pAlignedMem == pRawMem)
        pAlignedMem += align;

    // Определить сдвиг и сохранить его.
    // (Это работает для выравнивания до 256 байт.)
    ptrdiff_t shift = pAlignedMem – pRawMem;
    assert(shift > 0 && shift <= 256);
    pAlignedMem[-1] = static_cast<U8>(shift & 0xFF);

    return pAlignedMem;
}

void FreeAligned(void* pMem)
{
    if (pMem)
    {
        // Преобразовать в указатель U8.
        U8* pAlignedMem = reinterpret_cast<U8*>(pMem);

        // Извлечь сдвиг.
        ptrdiff_t shift = pAlignedMem[-1];
        if (shift == 0)
            shift = 256;

        // Вернуться к фактическому выделенному
        // адресу и удалить массив.
        U8* pRawMem = pAlignedMem – shift;
        delete[] pRawMem;
    }
}
```

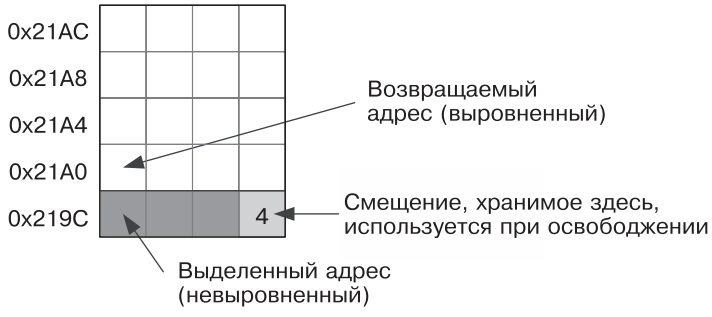


Рис. 6.3. Выравнивание памяти с 16-байтовым требованием выравнивания. Разница между выделенным адресом памяти и скорректированным (выровненным) адресом сохраняется в байте, непосредственно предшествующем скорректированному адресу, так что он может быть получен во время освобождения

Однокадровые и буферизованные распределители памяти

Практически все игровые движки выделяют память по крайней мере для некоторых временных данных во время прохождения игрового цикла. Эти данные либо отбрасываются в конце каждой итерации цикла, либо используются в следующем кадре, а затем отбрасываются. Этот паттерн распределения настолько широко распространен, что многие механизмы поддерживают *однокадровые* распределители и распределители *с двойной буферизацией*.

Однокадровые распределители. Однокадровый распределитель реализуется путем резервирования блока памяти и управления им с помощью простого распределителя описанного ранее стека. В начале каждого кадра указатель на вершину стека очищается до нижней части блока памяти. Распределения, сделанные во время кадра, растут к вершине блока, уничтожаются, и описанные действия повторяются:

```
StackAllocator g_singleFrameAllocator;

// Основной цикл игры
while (true)
{
    // Очистить буфер однокадрового распределителя на каждом кадре.
    g_singleFrameAllocator.clear();

    // ...

    // Выделить память из однокадрового буфера. Нам никогда
    // не нужно освобождать эту память! Просто обязательно
    // убедитесь, что используете эти данные только на этом кадре.
    void* p = g_singleFrameAllocator.alloc(nBytes);

    // ...
}
```

Одним из основных преимуществ однокадрового распределителя является то, что выделенная память никогда не должна освобождаться, — мы полагаемся на то, что распределитель будет очищен в начале каждого следующего кадра. Однокадровые распределители невероятно быстры. Большой минус в том, что использование однокадрового распределителя требует от программиста разумного уровня дисциплины. Вы должны понимать, что блок памяти, выделенный из однокадрового буфера, будет действителен только в текущем кадре. Программисты никогда не должны кэшировать указатель на блок памяти одного кадра за пределами этого кадра!

Распределители с двойной буферизацией. Распределитель с двойной буферизацией позволяет блоку памяти, выделенному в кадре *i*, использоваться в кадре *i* + 1. Для этого создадим два стековых распределителя одинакового размера, а затем устроим между ними пинг-понг на каждом кадре:

```
class DoubleBufferedAllocator
{
    U32            m_curStack;
    StackAllocator m_stack[2];

public:

    void swapBuffers()
    {
        m_curStack = (U32)!m_curStack;
    }

    void clearCurrentBuffer()
    {
        m_stack[m_curStack].clear();
    }

    void* alloc(U32 nBytes)
    {
        return m_stack[m_curStack].alloc(nBytes);
    }

    // ...
};

// ...

DoubleBufferedAllocator g_doubleBufAllocator;

// Основной цикл игры
while (true)
{
    // Очищаем однокадровый распределитель каждый
    // кадр, как и раньше.
    g_singleFrameAllocator.clear();

    // Меняем местами активные и неактивные буферы
    // буферизованного распределителя.
    g_doubleBufAllocator.swapBuffers();
```

```
    // Теперь очищаем новый активный буфер, оставляя
    // буфер последнего кадра нетронутым.
    g_doubleBufAllocator.clearCurrentBuffer();

    // ...

    // Выделить память в текущем буфере, не уничтожая
    // данные последнего кадра.
    // Используйте эти данные только в этом или следующем кадре.
    // Эта память также никогда не нуждается в освобождении.
    void* p = g_doubleBufAllocator.alloc(nBytes);

    // ...
}
```

Этот тип распределителя чрезвычайно полезен для кэширования результатов асинхронной обработки на многоядерной игровой консоли, такой как Xbox 360, Xbox One, PlayStation 3 или PlayStation 4. В кадре i мы можем запустить асинхронное задание на одном из ядер PS4, например передав ему адрес буфера назначения, который был выделен из распределителя с двойной буферизацией. Задание запускается и выдает результаты за некоторое время до конца кадра i, сохраняя их в предоставленном нами буфере. На кадре $i + 1$ буферы меняются местами. Результаты задания теперь находятся в неактивном буфере, поэтому они не будут перезаписаны никаким выделением памяти распределителем с двойной буферизацией, которое может быть сделано на этом кадре. Пока мы используем результаты работы до кадра $i + 2$, данные не будут перезаписаны.

6.2.2. Фрагментация памяти

Другая проблема с динамическим распределением кучи заключается в том, что со временем память может *фрагментироваться*. Когда программа запускается впервые, вся ее куча памяти свободна. Когда блок выделен, непрерывная область кучи памяти соответствующего размера помечается как используемая, а оставшаяся часть кучи свободна. Когда блок освобождается, он соответствующим образом помечается и соседние свободные блоки объединяются в один большой свободный блок. Со временем, когда различные размеры распределяются и освобождаются в случайном порядке, куча памяти начинает выглядеть как лоскутное одеяло из свободных и занятых блоков. Свободные регионы — это своеобразные дыры в ткани использованной памяти. Когда отверстий становится много и/или они довольно малы, мы говорим, что память стала фрагментированной (рис. 6.4).

Проблема с фрагментацией памяти состоит в том, что выделение может завершиться сбоем, даже если свободных байтов достаточно для удовлетворения запроса. Суть проблемы заключается в том, что выделенные блоки памяти всегда должны быть *непрерывными*. Например, чтобы удовлетворить запрос на 128 КиБ, должна существовать свободная область размером 128 КиБ или больше. Если имеется два

пропуска по 64 КиБ, то байтов достаточно, но выделение не выполняется, потому что это не *смежные байты*.

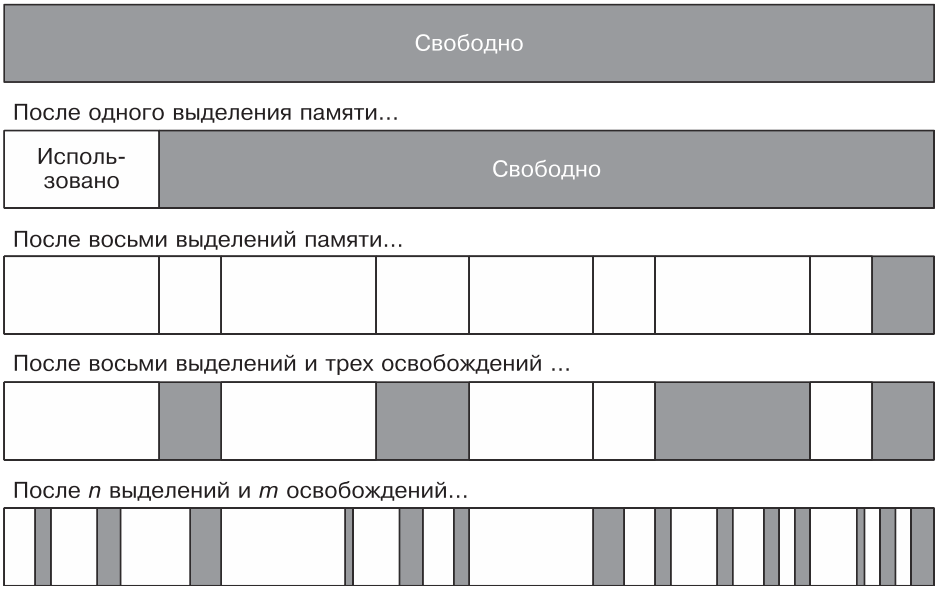


Рис. 6.4. Фрагментация памяти

Фрагментация памяти не является большой проблемой для операционных систем, поддерживающих *виртуальную память*. Система виртуальной памяти отображает отдельные блоки физической памяти, известные как *страницы*, в *виртуальное адресное пространство*, в котором страницы кажутся приложению смежными. Устаревшие страницы могут быть перенесены на жесткий диск, когда не хватает физической памяти, и снова загружены с диска, когда понадобятся. Подробное описание того, как работает виртуальная память: https://ru.wikipedia.org/wiki/Виртуальная_память. Большинство встроенных систем не могут позволить себе реализовать систему виртуальной памяти. Некоторые современные консоли поддерживают ее технически, однако большинство игровых консольных игр все еще не используют виртуальную память из-за наследственных проблем с производительностью.

Предотвращение фрагментации с помощью распределителя стека и распределителя пула

Пагубных последствий фрагментации памяти можно избежать, используя распределители стека и/или пула.

- Распределитель стека невосприимчив к фрагментации, поскольку выделение всегда является смежным и блоки должны освобождаться в порядке, противоположном тому, в котором были распределены (рис. 6.5).

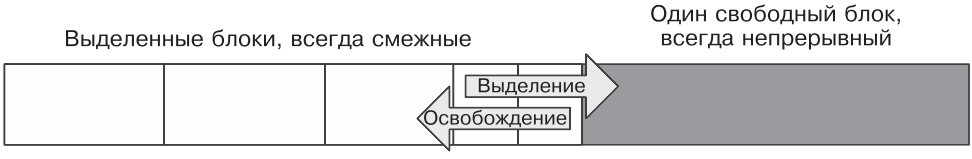


Рис. 6.5. Распределитель стека свободен от проблем фрагментации

- Распределитель пула также свободен от проблем фрагментации. Пулы *становятся* фрагментированными, но фрагментация никогда не вызывает преждевременных состояний нехватки памяти, как происходит в куче общего назначения. Запросы на выделение памяти пула никогда не могут завершиться неудачей из-за отсутствия достаточно большого непрерывного свободного блока, потому что все блоки одинакового размера (рис. 6.6).

Рис. 6.6. Распределитель пула не подвержен фрагментации

Дефрагментация и перемещение

Когда объекты разного размера выделяются и освобождаются в случайном порядке, ни распределитель на основе стека, ни распределитель на основе пула не могут использоваться. В таких случаях можно избежать фрагментации, периодически выполняя *дефрагментацию* кучи. Она предусматривает объединение всех свободных областей в куче путем смещения выделенных блоков с более высоких адресов памяти на более низкие, а дыр соответственно на более высокие. Алгоритм простой: найти первую дыру, затем взять выделенный блок непосредственно над ней и сдвинуть его вниз, к началу дыры. Это приводит к тому, что дыра всплывает, как пузырек, на более высокий адрес памяти. Если этот процесс повторять, в итоге все выделенные блоки займут непрерывную область памяти в нижнем конце адресного пространства кучи, а все дырки сольются в одну большую дыру в верхнем конце кучи (рис. 6.7).

Сдвиг блоков памяти не особенно сложен для реализации. Хитрость заключается в том, что мы перемещаем *выделенные* блоки памяти. Но если где-то есть *указатель* на один из выделенных блоков, то перемещение блока сделает его недействительным.

Решением этой проблемы является исправление любых указателей в сдвинутом блоке памяти, чтобы они указывали на правильный новый адрес после сдвига. Эта процедура известна как *перемещение* указателя. К сожалению, нет универсального способа найти все указатели на конкретную область памяти. Поэтому, если мы собираемся поддерживать дефрагментацию памяти в игровой системе, то либо программисты должны тщательно отслеживать все указатели вручную, чтобы их

можно было перемещать, либо указателям стоит предпочесть что-то более дружелюбное к переносу, например *умные указатели* или *дескрипторы*.

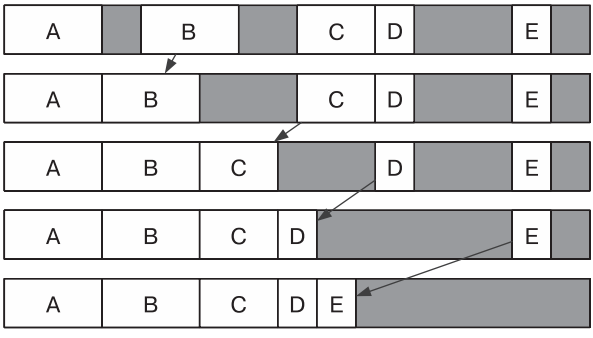


Рис. 6.7. Дефрагментация путем смещения блоков выделенной памяти на более низкие адреса

Умный указатель — это небольшой класс, который содержит указатель и действует как указатель для большинства целей и задач. Но поскольку умный указатель является классом, его можно запрограммировать для правильной обработки перемещения памяти. Один из подходов состоит в том, чтобы все умные указатели могли добавлять себя в глобальный связанный список. Всякий раз, когда блок памяти смещается в куче, можно сканировать связанный список умных указателей и соответствующим образом модифицировать каждый указатель, который указывает на смещенный блок памяти.

Дескриптор обычно внедряется как индекс в неперемещаемую таблицу, которая сама содержит указатели. Когда выделенный блок смещается в памяти, таблица дескрипторов может быть просканирована и все соответствующие указатели найдены и обновлены автоматически. Поскольку дескрипторы — это просто индексы в таблице указателей, их значения никогда не меняются независимо от смещения блоков памяти, поэтому на объекты, которые используют дескрипторы, никогда не влияет перемещение в памяти.

Другая проблема с перемещением возникает, когда некоторые блоки памяти нельзя переместить. Например, если вы работаете со сторонней библиотекой, которая не использует умные указатели или дескрипторы, возможно, любые указатели на ее структуры данных не будут перемещаться. Обычно наилучший способ обойти эту проблему — организовать для рассматриваемой библиотеки выделение ее памяти из специального буфера за пределами области перемещаемой памяти. Другой вариант — просто принять то, что некоторые блоки не могут быть перемещены. Если количество и размер неперемещаемых блоков невелики, система перемещения все равно будет работать довольно хорошо.

Интересно отметить, что все движки Naughty Dog поддерживают дефрагментацию. По возможности мы используем дескрипторы, чтобы избежать необходимости перемещения указателей. Однако в некоторых случаях нельзя обойтись

без обычных указателей. Их тщательно отслеживают и перемещают вручную всякий раз, когда блок памяти перемещается в процессе дефрагментации. Некоторые классы игровых объектов Naughty Dog по разным причинам переместить нельзя. Однако, как упоминалось ранее, это не создает никаких проблем на практике, потому что количество таких объектов всегда очень мало, а их размеры ничтожны по сравнению с общим размером области перемещаемой памяти.

Амортизирующая стоимость дефрагментации. Дефрагментация может выполняться медленно, потому что предусматривает копирование блоков памяти. Однако нам не нужно полностью дефрагментировать кучу за один раз — этот процесс может быть растянут на несколько кадров. Мы можем допустить смещение до N выделенных блоков в каждом кадре для некоторого небольшого значения N, например 8 или 16. Если игра работает со скоростью 30 кадров в секунду, то каждый кадр длится 1/30 секунды (33 миллисекунды). Таким образом, куча обычно может быть полностью дефрагментирована менее чем за одну секунду, что почти не повлияет на частоту кадров игры. Пока выделение и освобождение памяти не протекают быстрее, чем сдвиги дефрагментации, куча все время будет оставаться в основном дефрагментированной.

Этот подход работает только тогда, когда размер каждого блока довольно мал, так что время, необходимое для перемещения одного блока, не превышает времени, отведенного на каждый кадр. Если необходимо переместить очень большие блоки, мы часто можем разбить их на два или более подблока, каждый из которых переместить независимо. В движке Naughty Dog это не проблема, потому что перемещение используется только для динамических игровых объектов, а они никогда не превышают нескольких кибибайтов — обычно даже намного меньше.

6.3. Контейнеры

Разработчики игр задействуют широкий спектр структур данных, называемых *контейнерами* или *коллекциями*. Работа контейнера всегда одинакова — хранить ноль или несколько элементов данных и управлять ими, однако детали этих процессов сильно различаются, и у каждого типа контейнера есть свои плюсы и минусы. Общие типы данных контейнера включают следующие (но не ограничены ими).

- *Массив*. Упорядоченный непрерывный набор элементов, доступ к которым осуществляется по индексу. Длина массива обычно статически определяется во время компиляции. Он может быть многомерным. C и C++ поддерживают массивы изначально (например, `int a[5]`).
- *Динамический массив*. Массив, длина которого может динамически изменяться во время выполнения (например, стандартная библиотека C++ `std::vector`).
- *Односвязный список*. Упорядоченная коллекция элементов, которые не хранятся в памяти непрерывно, а связаны друг с другом с помощью указателей (например, `std::list` стандартной библиотеки C++).

- *Стек*. Контейнер, поддерживающий модель «последним пришел — первым ушел» (last-in-first-out, LIFO) для добавления и удаления элементов, также известный как push/pop (например, `std::stack`).
- *Очередь*. Контейнер, поддерживающий модель «первым пришел — первым ушел» (first-in-first-out, FIFO) для добавления и удаления элементов (например, `std::queue`).
- *Deque* (двухсторонняя очередь). Поддерживает эффективные вставку и удаление на обоих концах массива (например, `std::deque`).
- *Дерево*. Контейнер, в котором элементы сгруппированы иерархически. Каждый элемент (узел) имеет ноль или один родительский элемент и ноль или более дочерних элементов. Дерево является частным случаем DAG (см. далее).
- *Двоичное дерево поиска* (binary search tree, BST). Дерево, в котором каждый узел имеет не более двух потомков, со свойством порядка, позволяющим сортировать узлы по некоторым четко определенным критериям. Существуют различные виды деревьев бинарного поиска: красно-черные деревья, расширяющиеся деревья, деревья AVL и т. д.
- *Двоичная куча*. Бинарное дерево, которое поддерживает себя в отсортированном порядке, во многом как бинарное дерево поиска, по двум правилам: *свойству формы*, которое указывает, что дерево должно быть целиком заполнено, а его последняя строка заполнена слева направо, и *свойству кучи*, которое утверждает, что каждый узел по некоторому определяемому пользователем критерию больше всех его дочерних элементов или равен им.
- *Приоритетная очередь*. Контейнер, который позволяет добавлять элементы в любом порядке, а затем удалять в порядке, определяемом некоторым свойством самих элементов, то есть их *приоритетом*. Очередь приоритетов обычно реализуется в виде кучи (например, `std::priority_queue`), но возможны и другие виды. Очередь с приоритетами немного похожа на список, который постоянно сортируется, за исключением того, что очередь с приоритетами поддерживает только поиск элемента с наивысшим приоритетом и редко реализуется в виде списка изнутри.
- *Словарь*. Таблица пар «ключ — значение». Значение может быть эффективно найдено по соответствующему ключу. Словарь также известен как *карта* или *хеш-таблица*, хотя технически последняя является лишь одной из возможных реализаций словаря (например, `std::map`, `std::hash_map`).
- *Множество*. Контейнер, который гарантирует, что все элементы уникальны по некоторым критериям. Множество действует как словарь, только с ключами, но без значений.
- *Граф*. Набор узлов, соединенных друг с другом однонаправленными или двунаправленными путями в произвольном порядке.
- *Ориентированный ациклический граф* (directed acyclic graph, DAG). Набор узлов с однонаправленными (то есть *направленными*) взаимосвязями без *циклов* (то есть не существует непустого пути, который начинается и заканчивается в одном и том же узле).

6.3.1. Операции с контейнерами

Игровые движки, в которых используются контейнерные классы, неизбежно также применяют различные стандартные алгоритмы. Вот некоторые примеры.

- *Вставка*. Добавить новый элемент в контейнер. Новый элемент может быть помещен в начало или конец списка или в другое место. Или контейнер может не иметь представления о порядке.
- *Удаление*. Удалить элемент из контейнера, для чего может потребоваться операция поиска (см. далее). Однако, если доступен итератор, который ссылается на нужный элемент, может быть более эффективно удалить элемент с помощью итератора.
- *Последовательный доступ (итерация)*. Доступ к каждому элементу контейнера в некотором естественном предопределенном порядке.
- *Произвольный доступ*. Доступ к элементам в контейнере в произвольном порядке.
- *Поиск*. Поиск в контейнере элемента, соответствующего заданному критерию. Существует множество вариантов операции поиска, включая поиск в обратном направлении, поиск нескольких элементов и т. д. Кроме того, разные типы структур данных и разные ситуации требуют разных алгоритмов (см. en.wikipedia.org/wiki/Search_algorithm).
- *Сортировка*. Отсортировать содержимое контейнера в соответствии с некоторыми заданными критериями. Существует много различных алгоритмов сортировки, включая пузырьковую сортировку, сортировку выбором, сортировку вставками, быструю сортировку и т. д. (подробнее см. en.wikipedia.org/wiki/Sorting_algorithm и https://ru.wikipedia.org/wiki/Алгоритм_сортировки).

6.3.2. Итераторы

Итератор — это маленький класс, который знает, как эффективно посещать элементы в контейнере определенного типа. Он действует как индекс массива или указатель — ссылается на один элемент в контейнере за раз, его можно продвигать к следующему элементу, и он предоставляет своего рода механизм для проверки того, были ли посещены все элементы в контейнере. Например, первый из следующих двух фрагментов кода выполняет итерацию по массиву в стиле C с помощью указателя, а второй — по связанному списку с использованием почти идентичного синтаксиса:

```
void processArray(int container[], int numElements)
{
    int* pBegin = &container[0];
    int* pEnd = &container[numElements];

    for (int* p = pBegin; p != pEnd; p++)
    {
```

```
        int element = *p;
        // обработать элемент...
    }
}

void processList(std::list<int>& container)
{
    std::list<int>::iterator pBegin = container.begin();
    std::list<int>::iterator pEnd = container.end();

    for (auto p = pBegin; p != pEnd; ++p)
    {
        int element = *p;
        // обработать элемент...
    }
}
```

Основные преимущества использования итератора по сравнению с прямым доступом к элементам контейнера.

- Прямой доступ нарушил бы инкапсуляцию класса контейнера. Итератор же, как правило, является *дружественным* классом для класса контейнера, поэтому он может эффективно выполнять итерации и не зависит от деталей реализации внешнего мира. (На самом деле большинство хороших контейнерных классов скрывают свои внутренние детали и недоступны без *итератора*.)
- Итератор может упростить процесс итерации. Большинство итераторов действуют как индексы или указатели массива, поэтому можно написать простой цикл, в котором итератор увеличивается и сравнивается с условием выхода из цикла, даже когда базовая структура данных невероятно сложна. Например, итератор может сделать так, чтобы обход дерева в глубину в порядке упорядочения выглядел не более сложным, чем простая итерация по массиву.

Преинкремент и постинкремент

Обратите внимание на то, что в примере `processArray()` мы используем оператор *постинкремента* `p++` в C++, а не оператор *преинкремента* `++p`. Это тонкая, но иногда важная оптимизация. Оператор преинкремента увеличивает содержимое переменной *до того*, как ее значение (теперь измененное) применяется в выражении. Оператор постинкремента увеличивает содержимое переменной *после* того, как она будет задействована. Это означает, что написание `++p` вводит в код *зависимость данных* — процессор должен дождаться завершения операции приращения, прежде чем его значение можно будет задействовать в выражении. На процессоре, где часто используется конвейеризация, это приводит к *простою*. В то же время при наличии `p++` не возникает зависимость от данных. Значение переменной можно применить немедленно, а операция приращения может произойти позже или параллельно с этим. В любом случае конвейер не простаивает.

Конечно, в выражении `update` цикла `for` не должно быть различий между пре- и постинкрементами, так как любой хороший компилятор распознает, что *значение* переменной не используется в `update_expr`. Но в тех случаях, когда значение задействовано, постинкремент предпочтительнее, поскольку он не вызывает остановку в конвейере ЦП.

Может быть целесообразно сделать исключение из этого небольшого практического правила для классов, *перегруженных* операторами приращения, что является обычной практикой в классах *итераторов*. По определению оператор постинкремента должен возвращать неизмененную *копию* объекта, для которого он вызывается. В зависимости от размера и сложности данных — членов класса — дополнительная стоимость копирования итератора может склонить вас к предпочтению преинкремента при использовании таких классов в циклах, для которых важна производительность. (Преинкремент не обязательно лучше, чем постинкремент, в таком простом примере, как функция `processList()`, показанная ранее, но я реализовал ее с преинкрементом, чтобы подчеркнуть разницу.)

6.3.3. Алгоритмическая сложность

Выбор типа контейнера для использования в конкретном приложении зависит от его производительности и характеристик памяти. Для каждого типа контейнера можно определить теоретическую производительность обычных операций, таких как вставка, удаление, поиск и сортировка.

Обычно мы указываем количество времени T, которое означает, сколько будет выполняться функция от числа элементов n в контейнере:

$$T = f(n).$$

Вместо того чтобы пытаться найти точную функцию f, мы заботимся только о поиске общего *порядка* функции. Например, если бы теоретическая функция была любой из следующих:

$$T = 5n^2 + 17;$$

$$T = 102n^2 + 50n + 12;$$

$$T = 1/2n^2,$$

во всех случаях мы упростили бы выражение до ее наиболее общего вида — в данном случае n^2. Чтобы обозначить, что мы указываем лишь порядок функции, а не ее точное значение, используем «O большое» и пишем:

$$T = O(n^2).$$

Порядок функции алгоритма обычно можно определить с помощью проверки псевдокода. Если время выполнения алгоритма не зависит от количества элементов в контейнере, мы говорим, что это $O(1)$ (то есть алгоритм завершает выполнение *за постоянное время*). Если алгоритм выполняет *цикл* над элементами в контейнере и посещает каждый элемент один раз, например, при линейном поиске по не-

отсортированному списку, говорим, что сложность алгоритма $O(n)$. Если вложены два цикла, каждый из которых потенциально посещает каждый узел один раз, то мы говорим, что сложность алгоритма $O(n^2)$. Если используется подход «разделяй и властвуй», как в *бинарном поиске* (где половина списка удаляется на каждом шаге, мы ожидаем, что только элементы $[\log_2(n) + 1]$ будут фактически посещены алгоритмом в худшем случае, и, следовательно, называем его операцией $O(\log n)$. Если алгоритм выполняет подалгоритм n раз, а сложность подалгоритма равна $O(\log n)$, то результирующий алгоритм будет $O(n \log n)$.

Чтобы выбрать подходящий класс контейнеров, мы должны посмотреть на наиболее широко распространенные операции и взять контейнер с наиболее благоприятными для этих операций характеристиками производительности. Самые распространенные порядки сложностей, с которыми вы столкнетесь (от самого быстрого до самого медленного), перечислены здесь: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(nk)$ для $k > 2$.

Мы также должны учитывать структуру памяти и характеристики использования контейнеров. Так, массив (например, `int a[5]` или `std::vector`) хранит свои элементы *непрерывно* в памяти и *не требует служебной памяти* для чего-либо, кроме самих элементов. (Следует учитывать, что *динамический* массив требует небольших фиксированных накладных расходов.) В то же время связанный список (например, `std::list`) оборачивает каждый элемент в структуру данных «ссылка», которая содержит указатель на следующий элемент и, возможно, также указатель на предыдущий элемент — в общей сложности до 16 байт служебных данных на элемент на 64-битной машине. Кроме того, элементы в связанном списке не обязательно должны быть смежными в памяти и часто таковыми не являются. Непрерывный блок памяти обычно гораздо более дружественен к кэшу, чем набор разрозненных блоков памяти. Следовательно, для высокоскоростных алгоритмов массивы обычно лучше, чем связанные списки, с точки зрения производительности кэша (если только узлы связанного списка сами не выделены из небольшого непрерывного блока памяти). Но связанный список лучше для ситуаций, в которых скорость вставки и удаления элементов имеет первостепенное значение.

6.3.4. Создание пользовательских контейнерных классов

Многие игровые движки предоставляют собственные реализации общих структур данных контейнера. Эта практика особенно распространена в движках консольных игр и в играх, предназначенных для мобильных телефонов и КПК. Причины для создания этих классов таковы.

- *Полный контроль.* Вы контролируете требования к памяти структуры данных, используемые алгоритмы, когда и как выделяется память и т. д.
- *Возможности для оптимизации.* Можете оптимизировать свои структуры данных и алгоритмы, чтобы задействовать преимущества аппаратных функций,

специфичных для целевой (-ых) консоли (-ей), на которую (-ые) вы ориентируетесь, или точно настроить эти структуры для конкретного применения в движке.

- *Гибкость*. Вы можете предоставить пользовательские алгоритмы, отсутствующие в стандартной библиотеке C++ или сторонних библиотеках, таких как Boost (например, поиск *n* самых значимых элементов в контейнере, а не только одного наиболее значимого).
- *Устранение внешних зависимостей*. Поскольку вы создали программное обеспечение самостоятельно, то при его поддержке не зависите от какой-либо другой компании или команды. В случае возникновения проблем можно сразу же отладить и исправить их, а не ждать до следующего выпуска библиотеки, что может произойти лишь после того, как вы выпустите игру!
- *Контроль над параллельными структурами данных*. Создавая собственные классы контейнеров, вы полностью контролируете средства, с помощью которых они защищены от одновременного доступа в многопоточной или многоядерной системе. Например, на PS4 Naughty Dog использует легкие мьютексы со спин-блокировкой для большинства параллельных структур данных, поскольку они хорошо работают с нашей системой планирования заданий на основе фиберов (fibers). Сторонняя контейнерная библиотека не могла бы обеспечить нам такую гибкость.

Мы не сможем охватить здесь все возможные структуры данных, но рассмотрим несколько распространенных способов, с помощью которых разработчики игрового движка стремятся справиться с контейнерами.

Строить или не строить

Мы не будем обсуждать здесь детали того, как реализовать все эти типы данных и алгоритмы, — для этого есть множество книг и онлайн-ресурсов. Однако мы *будем* заниматься вопросом, где можно получить реализации тех типов и алгоритмов, которые вам нужны. Как у разработчиков игровых движков, у нас есть три основных варианта.

1. Построить необходимые структуры данных вручную.
2. Использовать контейнеры в стиле STL, предоставляемые стандартной библиотекой C++.
3. Положиться на стороннюю библиотеку, такую как Boost (www.boost.org).

Стандартная библиотека C++ и сторонние библиотеки, например Boost, являются привлекательными вариантами, поскольку они предоставляют богатый и мощный набор классов контейнеров, охватывающих практически все возможные типы структур данных. Кроме того, эти пакеты обеспечивают мощный набор основанных на шаблонах *универсальных алгоритмов* — реализации общих алгоритмов, таких как поиск элемента в контейнере, который может быть применен практиче-

ски к любому типу объекта данных. Однако эти реализации могут не подойти для некоторых видов игровых движков. Давайте посвятим некоторое время изучению плюсов и минусов каждого подхода.

Стандартная библиотека языка C++. Преимущества контейнерных классов в стиле STL в стандартной библиотеке C++.

- Они предлагают богатый набор функций.
- Их реализации надежны и полностью переносимы.

Однако эти контейнерные классы имеют и некоторые недостатки.

- Заголовочные файлы загадочны и трудны для понимания, хотя документация довольно хорошая.
- Контейнерные классы общего назначения часто работают медленнее, чем структура данных, специально созданная для решения конкретной проблемы.
- Универсальный контейнер может потреблять больше памяти, чем собственная структура данных.
- Стандартная библиотека C++ выполняет динамическое распределение памяти, и иногда сложно контролировать ее аппетиты в отношении памяти, что важно для высокопроизводительных игр с ограниченным объемом памяти.
- Система шаблонных распределителей, предоставляемая стандартной библиотекой C++, недостаточно гибка для того, чтобы эти контейнеры могли использоваться с определенными типами распределителей памяти, такими как распределители на основе стека (см. подраздел 6.2.1).

В движке *Medal of Honor: Pacific Assault (MOHPA)* для ПК активно применялось то, что когда-то было известно как стандартная библиотека шаблонов (standard template library, STL). И хотя у *MOHPA* были некоторые проблемы с частотой кадров, команда смогла исключить проблемы с производительностью, вызванные STL (прежде всего путем тщательного ограничения и контроля ее использования). OGRE — популярная объектно-ориентированная библиотека рендеринга, которую мы применяем для некоторых примеров из этой книги, — также интенсивно задействует контейнеры в стиле STL. Однако в Naughty Dog мы запрещаем контейнеры STL в коде игры (хотя разрешаем задействовать их в коде автономных инструментов). Ваш опыт может быть иным: использовать контейнеры в стиле STL, предоставляемые стандартной библиотекой C++ в проекте игрового движка, безусловно, можно, но следует делать это осторожно.

Boost. Работу над проектом Boost начали члены рабочей группы C++ Standards Committee Library, но теперь это проект с открытым исходным кодом, в котором участвует множество людей со всего мира. Его целью является создание библиотек, которые расширяют стандартную библиотеку C++ и работают вместе с ней в сферах как коммерческого, так и некоммерческого использования. Многие из библиотек Boost уже включены в стандартную библиотеку C++, начиная с C++11, а другие компоненты входят в технический отчет по библиотекам Комитета по

стандартам (TR2), который является шагом к тому, чтобы стать частью будущего стандарта C++. Вот краткое изложение преимуществ, которые дает Boost.

- Boost обеспечивает множество полезных возможностей, недоступных в стандартной библиотеке C++.
- В некоторых случаях Boost предоставляет альтернативы для некоторых классов стандартной библиотеки C++, у которых есть известные проблемы реализации.
- Boost отлично справляется с очень сложными задачами, такими как умные указатели. (Имейте в виду, что умные указатели — это сложные звери, они могут быть убийственными для производительности. Дескрипторы зачастую предпочтительнее, подробности см. в разделе 16.5.)
- Документация к библиотекам Boost обычно превосходна. Она не только объясняет, что делает каждая библиотека и как ее использовать, но и в большинстве случаев дает отличное всестороннее описание проектных решений, ограничений и требований, с которыми столкнулись при создании библиотеки. Таким образом, чтение документации Boost — отличный способ узнать о принципах разработки программного обеспечения.

Если вы уже используете стандартную библиотеку C++, то Boost может стать отличным расширением и/или альтернативой многим ее функциям. Однако нужно помнить о следующем.

- Большинство основных классов Boost являются шаблонами, поэтому все, что для них нужно, — это соответствующий набор заголовочных файлов. Тем не менее некоторые библиотеки Boost встроены в довольно большие файлы `.lib` и не могут применяться в небольших проектах.
- В то время как всемирное сообщество Boost предоставляет отличную поддержку, библиотеки Boost все же поставляются без гарантий. Если вы столкнетесь с ошибкой, ваша команда будет вынуждена исправлять ее самостоятельно.
- Библиотеки Boost распространяются по лицензии Boost Software. Внимательно прочитайте информацию о ней (www.boost.org/more/license_info.html), чтобы убедиться, что она подходит для вашего движка.

Folly. Folly — это библиотека с открытым исходным кодом, разработанная Андреем Александреску и инженерами Facebook. Ее назначение — расширить стандартную библиотеку C++ и библиотеку Boost (а не конкурировать с ними), причем упор сделан на простоту использования и разработку высокопроизводительного программного обеспечения. Вы можете прочитать о ней, найдя в Интернете статью *Folly: The Facebook Open Source Library* (www.facebook.com). Сама библиотека в GitHub находится здесь: github.com/facebook/folly.

Loki. Существует сложная ветвь программирования на C++, известная как *шаблонное метапрограммирование*. Ее суть заключается в том, чтобы использовать компилятор для большей части работы, которую в противном случае пришлось бы выполнять с помощью функции шаблона C++ и фактически обманывая компилятор — заставляя его делать то, для чего он изначально не был предназначен.

Это может привести к появлению поразительно мощных и полезных инструментов программирования.

Безусловно, самой известной и, вероятно, самой мощной библиотекой шаблонного метапрограммирования для C++ является Loki, разработанная и написанная Андреем Александреску (его домашняя страница находится по адресу www.erdani.org). Библиотеку можно получить из SourceForge по адресу loki-lib.sourceforge.net.

Библиотека Loki очень мощная, это увлекательный код для изучения, из которого можно многое почерпнуть. Однако у нее есть два больших недостатка: ее код может быть устрашающим для чтения, применения и тем более понимания, некоторые из ее компонентов зависят от использования побочных эффектов поведения компилятора, что требует тщательной настройки при работе с новыми компиляторами. Таким образом, задействовать Loki может оказаться довольно сложно, и она не такая портативная, как ее менее экстремальные аналоги. Loki не для слабых духом. Тем не менее некоторые ее концепции, такие как программирование на основе политик, можно применить к любому проекту C++, даже если вы не используете библиотеку Loki как таковую. Я настоятельно рекомендую всем разработчикам программного обеспечения прочитать новаторскую книгу Андрея Александреску *Modern C++ Design* [3], из которой родилась библиотека Loki.

6.3.5. Динамические массивы и выделение памяти фрагментами

Массивы в стиле C фиксированного размера довольно часто используются в игровом программировании, поскольку не требуют выделения памяти, занимают непрерывные области памяти, а следовательно, удобны для кэширования, и очень эффективно поддерживают многие общие операции, такие как добавление данных и поиск.

Когда размер массива невозможно определить заранее, программисты склонны обращаться либо к *связанным спискам*, либо к *динамическим массивам*. Если мы хотим сохранить характеристики производительности и памяти массивов фиксированной длины, то динамический массив часто является предпочтительной структурой данных.

Самый простой способ реализовать динамический массив — изначально выделить *n*-элементный буфер и *увеличивать* его, только если сделана попытка добавить в него более *n* элементов. Это дает возможность использовать полезные свойства массива фиксированного размера, но не быть ограниченными размером. Увеличение осуществляется выделением нового большего буфера, копированием в него данных из исходного буфера и дальнейшим освобождением последнего. Размер буфера увеличивается в определенном порядке, например, добавляется *n* к его длине или его длина удваивается каждый раз, когда это требуется. Большинство реализаций, с которыми я сталкивался, никогда не сжимают массив, а только увеличивают его (единственное исключение — очистка массива до нулевого размера, что может как освободить буфер, так и не освободить его). Следовательно, размер массива становится своего рода отметкой уровня воды. Именно так работает класс `std::vector`.

Конечно, если вы можете установить верхнюю планку для данных, тогда, вероятно, лучше просто выделить один буфер такого размера при запуске движка. Увеличение динамического массива может оказаться невероятно дорогостоящим из-за перераспределения памяти и затрат на копирование данных. Их влияние зависит от размеров задействованных буферов. Рост может привести также к фрагментации, когда память освобождается от уже не используемых буферов. Таким образом, работая с динамическими массивами, как и с любыми структурами данных, которые выделяют память, следует соблюдать осторожность. Динамические массивы, вероятно, лучше всего задействовать во время разработки, когда вы еще не уверены, какой размер буфера потребуется. Их можно преобразовать в массивы фиксированного размера после того, как будут установлены подходящие бюджеты памяти.

6.3.6. Словари и хеш-таблицы

Словарь — это таблица пар «ключ — значение». Значение в словаре можно быстро найти, зная его ключ. Ключи и значения могут быть данными любого типа. Такая структура данных обычно реализуется в виде либо двоичного дерева поиска, либо хеш-таблицы.

При реализации двоичного дерева пары «ключ — значение» хранятся в узлах двоичного дерева, а оно поддерживается в порядке отсортированных ключей. Поиск значения по ключу включает выполнение бинарного поиска $O(\log n)$.

При реализации хеш-таблицы значения хранятся в таблице фиксированного размера, где каждый слот в таблице представляет один или несколько ключей. Чтобы вставить пару «ключ — значение» в хеш-таблицу, сначала ключ преобразуется в целочисленную форму с помощью процесса *хеширования* (если это еще не целое число). Затем *индекс* для хеш-таблицы вычисляется путем взятия хешированного ключа *по модулю* размера таблицы. Наконец, пара «ключ — значение» сохраняется в слоте, соответствующем этому индексу. Напомню, что оператор *взятия по модулю* (`%` в C/C++) находит остаток от деления целочисленного ключа на размер таблицы. Так что если хеш-таблица имеет пять слотов, то ключ 3 будет храниться в индексе 3 (`3%5 == 3`), а ключ 6 — в индексе 1 (`6%5 == 1`). Поиск пары «ключ — значение» является операцией $O(1)$ при отсутствии коллизий.

Коллизии: открытые и закрытые хеш-таблицы

Иногда два или более ключа занимают один и тот же слот в хеш-таблице. Эта ситуация известна как *коллизия*. Существует два основных способа *разрешения* коллизий, дающих два вида хеш-таблиц.

- *Открытые*. В открытой хеш-таблице (рис. 6.8) коллизии разрешаются простым хранением более чем одной пары «ключ — значение» в каждом индексе, обычно в форме связанного списка. Этот подход прост в реализации и не устанавливает верхней границы для числа пар «ключ — значение», которые могут

быть сохранены. Однако требуется, чтобы память выделялась динамически при добавлении в таблицу новой пары «ключ — значение».

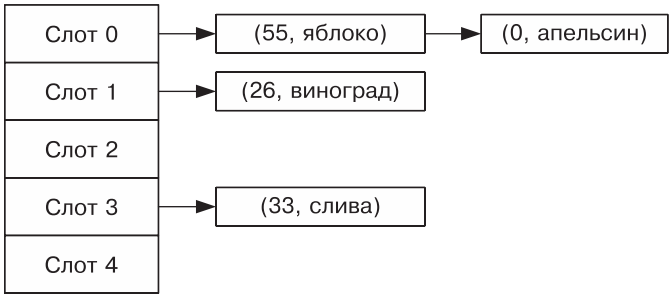


Рис. 6.8. Открытая хеш-таблица

- *Закрытые*. В закрытой хеш-таблице (рис. 6.9) коллизии разрешаются с помощью *последовательных проб*, реализуемых до тех пор, пока не будет найден свободный слот. (Последовательные пробы — это применение четко определенного алгоритма для поиска свободного слота.) Данный подход немного сложнее реализовать, и он накладывает ограничение на количество пар «ключ — значение», которые могут находиться в таблице, поскольку каждый слот может содержать только одну такую пару. Но главное преимущество этого вида хеш-таблицы заключается в том, что она использует фиксированный объем памяти и не требует ее динамического выделения. Так что она часто оказывается хорошим выбором для консольного движка.

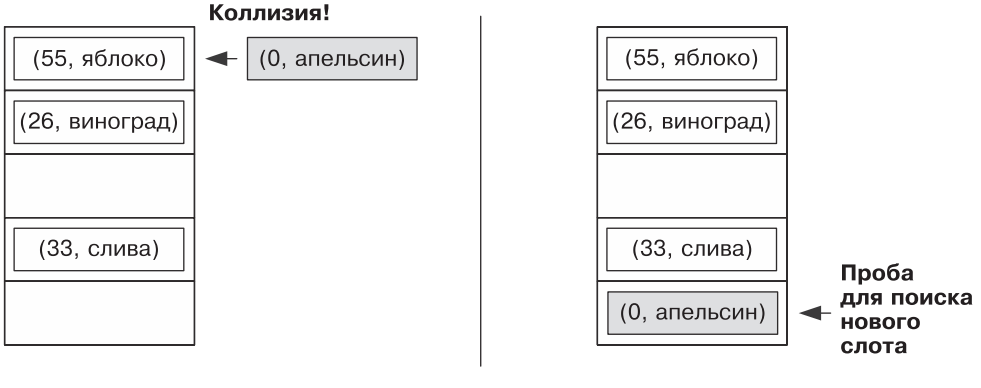


Рис. 6.9. Закрытая хеш-таблица

Может запутать следующее: иногда говорят, что закрытые хеш-таблицы используют *открытую адресацию*, тогда как открытые хеш-таблицы применяют метод адресации, известный как *цепочка*, названный так из-за связанных списков в каждом слоте в таблице.

Хеширование

Хеширование — это процесс преобразования ключа произвольного типа данных в целое число, которое можно использовать по модулю размера таблицы в качестве индекса в ней. Математически, имея ключ k, мы хотим сгенерировать целочисленное хеш-значение h с помощью хеш-функции H, а затем найти индекс i в таблице следующим образом:

$$h = H(k);$$

$$i = h \bmod N,$$

где N — количество слотов в таблице, а символ mod представляет операцию *взятия по модулю*, то есть нахождения остатка от частного h/N.

Если ключи являются уникальными целыми числами, хеш-функция может быть тождественна функции $H(k) = k$. Если ключи — это уникальные 32-разрядные числа с плавающей точкой, хеш-функция может просто интерпретировать битовую комбинацию 32-разрядного числа с плавающей точкой, как если бы это было 32-разрядное целое число:

```
U32 hashFloat(float f)
{
    union
    {
        float m_asFloat;
        U32 m_asU32;
    } u;

    u.m_asFloat = f;
    return u.m_asU32;
}
```

Если ключ является строкой, мы можем использовать *функцию хеширования строки*, которая объединяет коды ASCII или UTF всех символов в строке в одно 32-разрядное целочисленное значение.

Качество хеш-функции $H(k)$ имеет решающее значение для эффективности хеш-таблицы. Хорошая хеш-функция — это функция, которая распределяет множество допустимых ключей равномерно по всей таблице, тем самым сводя к минимуму вероятность коллизии. Также хеш-функция должна быть достаточно быстрой для вычисления и *детерминированной* в том смысле, что должна выдавать одинаковые выходные данные при каждом вызове с идентичным вводом.

Вероятно, строки — это наиболее распространенный тип ключей, с которыми вы сталкиваетесь, поэтому особенно полезно использовать хорошую функцию хеширования строк. В табл. 6.1 приведен ряд известных алгоритмов хеширования, их рейтинги быстродействия (определенные на основе результатов эталонных измерений, преобразованных в рейтинги «Низкий», «Средний» или «Высокий») и оценки по тесту SMHasher (github.com/aappleby/smhasher). Обратите внимание на то, что относительная пропускная способность предназначена только для прибли-

зительного сравнения. На быстродействие хеш-функции влияют многие факторы, включая аппаратное обеспечение, на котором она выполняется, и свойства входных данных. Криптографические функции хеширования преднамеренно сделаны медленными, поскольку их целью является создание хешей, которые очень редко вступают в коллизии с хешами других входных строк и для которых задача определения строки, выдающая заданное значение хеша, чрезвычайно сложна в вычислительном отношении.

Таблица 6.1. Сравнение известных алгоритмов хеширования с точки зрения их относительной пропускной способности и оценок по тесту SMHasher. Обратите внимание на то, что SBox сам по себе не является криптографическим хешем, но выступает одним из компонентов алгоритмов симметричного ключа, используемых в криптографии

Имя	Пропускная способность	Оценка	Криптографическая или нет
xxHash	Высокая	10	Нет
MurmurHash 3a	Высокая	10	Нет
SBox	Средняя	9	Нет ++
Lookup3	Средняя	9	Нет
CityHash64	Средняя	10	Нет
CRC32	Низкая	9	Нет
MD5-32	Низкая	10	Да
SHA1-32	Низкая	10	Да

Для получения дополнительной информации о хеш-функциях прочтите превосходную статью Пола Се по адресу www.azillionmonkeys.com/qed/hash.html.

Реализация закрытой хеш-таблицы

В закрытой хеш-таблице пары «ключ — значение» хранятся непосредственно в таблице, а не в связанном списке в каждой ее записи. Такой подход позволяет программисту заранее определить точный объем памяти, который будет использоваться хеш-таблицей. Проблема возникает, когда мы сталкиваемся с *коллизией* — двумя ключами, которые хотят быть сохраненными в одном слоте таблицы. Для решения этой проблемы задействуем процесс, известный как *последовательные пробы*.

Самый простой подход — использование *линейных проб*. Представьте, что хеш-функция вернула индекс таблицы i, но этот слот уже занят. Мы пытаемся использовать слоты $(i + 1)$, $(i + 2)$ и т. д. до тех пор, пока не будет найден пустой слот (переход к началу таблицы при $i = N$). Другой вариант линейных проб заключается в чередовании поиска вперед и назад $(i + 1)$, $(i – 1)$, $(i + 2)$, $(i – 2)$ и т. д., что позволяет убедиться, что получающиеся индексы по модулю относятся к допустимому диапазону таблицы.

Линейные пробы приводят к тому, что пары «ключ — значение» «слипаются». Чтобы избежать этих кластеров, можно применить алгоритм, известный как

квадратичные пробы. Мы начинаем с индекса занятой таблицы i и используем последовательность проб $ij = (i \pm j^2)$ для $j = 1, 2, 3$. Другими словами, мы пробуем $(i + 1^2)$, $(i - 1^2)$, $(i + 2^2)$, $(i - 2^2)$ и т. д., всегда помня, что результирующий индекс необходимо взять по модулю, чтобы значение попало в допустимый диапазон таблицы.

При закрытом хешировании рекомендуется сделать размер таблицы *простым числом*. Использование основного размера таблицы в сочетании с квадратичными пробами позволяет полнее охватить доступные слоты таблицы при минимальной кластеризации. Хорошее объяснение того, почему в качестве размера хеш-таблиц предпочтительны простые числа, представлено на сайте stackoverflow.com/questions/1145217/why-should-hash-functions-use-a-prime-number-module.

Хеширование Робин Гуда

Хеширование Робин Гуда — еще один метод исследования закрытых хеш-таблиц, который недавно приобрел популярность. Эта схема проб повышает производительность закрытой хеш-таблицы, даже когда она почти заполнена. Хорошее обсуждение того, как работает хеширование Робин Гуда, вы найдете на странице www.sebastiansylvan.com/post/robin-hood-hashing-should-be-your-default-hash-table-implementation/.

6.4. Строки

Строки повсеместно применяются практически в каждом программном проекте, и игровые движки не исключение. На первый взгляд строка может показаться простым фундаментальным типом данных. Но, когда вы начнете использовать строки в своих проектах, вы быстро обнаружите широкий спектр связанных с архитектурой проблем и ограничений, которые необходимо учитывать.

6.4.1. Проблема со строками

Основные вопросы относительно строк таковы: как они должны храниться и как ими следует управлять в программе? В C и C++ строки даже не являются атомарными типами — они реализованы как *массивы символов*. Переменная длина строк означает, что нужно либо жестко ограничивать их размеры, либо динамически распределять строковые буферы.

Другая большая проблема со строками связана с *локализацией* — процессом адаптации программного обеспечения к выпуску на других языках. Она известна также как *интернационализация* или, кратко, I18N. Любая строка, которая выводится для пользователя на английском языке, должна быть переведена на любой из языков, которые вы планируете поддерживать. (Строки, используемые внутри программы, но никогда не отображаемые для пользователя, конечно же, не локализуются.) Это подразумевает не только то, что вы можете представить все глифы символов всех языков, которые планируете поддерживать (с помощью соответствующего набора шрифтов), но и то, что игра сможет обрабатывать различные направления текста. Например, традиционный китайский текст ориентирован вертикально (хотя

современные китайцы и японцы обычно пишут горизонтально и слева направо), а некоторые языки, такие как иврит, читаются справа налево. Игра также должна изящно обрабатывать вероятность того, что переведенная строка будет либо намного длиннее либо намного короче, чем ее английский аналог.

Наконец, важно понимать, что строки используются внутри игрового движка для таких вещей, как имена файлов ресурсов и идентификаторы объектов. Например, когда разработчик выкладывает уровень, очень удобно разрешить ему идентифицировать объекты, имеющиеся на уровне, значимыми именами, такими как PlayerCamera, enemy-tank-01 или ExplosionTrigger.

То, как движок обрабатывает внутренние строки, часто влияет на производительность игры. Это связано с тем, что строки по своей природе дороги для обработки во время выполнения. Сравнение или копирование целых чисел или чисел с плавающей точкой может быть выполнено с помощью простых инструкций машинного языка. В то же время сравнение строк требует $O(n)$ сканирования символьных массивов с помощью функции, подобной `strcmp()`, где n — длина строки. Для копирования строки требуется $O(n)$ копия памяти, не говоря уже о возможности динамического выделения памяти для копии. Однажды во время работы над проектом мы оценивали производительность игры и обнаружили, что `strcmp()` и `strcpy()` — две самые дорогие функции! Устраняя ненужные строковые операции и используя некоторые методы, описанные в этом разделе, мы смогли практически исключить эти функции и значительно увеличить частоту кадров в игре. (Я слышал похожие истории от разработчиков из разных студий.)

6.4.2. Классы строк

Многие программисты C++ предпочитают работать со *строковым классом*, таким как `std::string` стандартной библиотеки C++, а не иметь дело непосредственно с символьными массивами. Такие классы могут сильно упростить работу со строками для программиста и сделать ее более удобной. Однако класс строк может иметь скрытые затраты, которые трудно увидеть, пока вы не проведете профилирование. Например, передача строки в функцию в виде массива символов в стиле C происходит быстро, потому что адрес первого символа обычно передается в аппаратном регистре. В то же время передача строкового объекта может привести к возникновению накладных расходов в одном или нескольких конструкторах копирования, если функция объявлена или используется неправильно. Копирование строк может включать динамическое распределение памяти в результате чего-то, что выглядит как безобидный вызов функции, а в итоге будет стоить тысяч машинных циклов.

Из-за множества проблем со строковыми классами я обычно предпочитаю избегать их в выполняющемся коде игры. Но если вы очень хотите задействовать строковый класс, убедитесь, что выбрали или внедрили тот, который имеет приемлемые характеристики производительности во время выполнения, а также что все программисты, которые его применяют, знают о его стоимости. О своем строковом классе нужно знать следующее. Обрабатывает ли он все строковые буферы только для чтения? Использует ли оптимизацию для *копирования при записи* (см. en.wikipedia.org/wiki/Copy-on-write и https://ru.wikipedia.org/wiki/Копирование_при_записи)?

Он предоставляет конструктор перемещения в C++11? Он владеет памятью, связанной со строкой, или только ссылается на память, которой не владеет? (Для получения дополнительной информации о проблеме владения памятью в строковых классах см. www.boost.org/doc/libs/1_57_0/libs/utility/doc/html/string_ref.html.) Старайтесь передавать строковые объекты по ссылке, а не по значению, так как последнее часто сопряжено с расходами на копирование строк. Профилируйте свой код сразу и часто, чтобы класс строк не стал основным источником уменьшения частоты кадров!

Применение специализированного строкового класса кажется мне оправданным в единственном случае — для хранения путей файловой системы и управления ими. Здесь гипотетический класс `Path` мог бы сделать необработанный массив символов в стиле C намного более ценным. Например, он может предоставлять функции для извлечения имени файла, расширения файла или каталога из пути. Может скрыть различия операционных систем, автоматически конвертируя обратный слеш в стиле Windows в прямую косую черту в стиле UNIX или разделитель пути какой-либо другой операционной системы. Написание класса `Path`, обеспечивающего такую функциональность кросс-платформенным способом, может оказаться очень полезным в контексте игрового движка. (В подразделе 7.1.1 больше информации по этой теме.)

6.4.3. Уникальные идентификаторы

Объекты в любом виртуальном игровом мире должны быть каким-то образом однозначно идентифицированы. Например, в *PacMan* есть игровые объекты с именами `pac_man`, `blinky`, `pinky`, `inky` и `clyde`. Уникальные идентификаторы позволяют разработчикам игр отслеживать множество объектов, формирующих игровые миры, а также обнаруживать эти объекты и управлять ими во время выполнения. *Ресурсы*, из которых создаются игровые объекты — меши, материалы, текстуры, аудиоклипы, анимация и т. д., — также нуждаются в уникальных идентификаторах.

Строки кажутся естественным выбором для таких идентификаторов. Ресурсы часто хранятся в отдельных файлах на диске, поэтому их обычно можно однозначно идентифицировать по файловым путям, которые, конечно, являются строками. А игровые объекты создают разработчики, для которых естественно присваивать объектам понятные строковые имена, а не запоминать их целочисленные индексы либо 64- или 128-битные глобально уникальные идентификаторы (globally unique identifiers, GUID). Однако скорость *сравнения* уникальных идентификаторов имеет первостепенное значение, поэтому `strcmp()` просто не подходит. Нам нужен способ усидеть на двух стульях — способ совместить всю информативность и гибкость строк со скоростью обработки целого числа.

Хешированные идентификаторы строк

Одним из удачных решений является *хеширование* строк. Как мы уже видели, хеш-функция отображает строку в полууникальное целое число. Строковые хеш-коды можно сравнивать, как и любые другие целые числа, поэтому сравнение выполня-

ется быстро. Если мы сохраняем фактические строки в хеш-таблице, то исходная строка всегда может быть восстановлена из хеш-кода. Это полезно для целей отладки и позволяет отображать хешированные строки на экране или в файлах журнала. Программисты игры иногда используют термин *«id (идентификатор) строки»* для обозначения хешированной строки. В движке Unreal вместо этого применяется термин *«имя»* (реализуется классом `FName`).

Как и в любой системе хеширования, здесь возможны *коллизии*, то есть две разные строки могут преобразовываться в один и тот же хеш-код. Однако имея подходящую хеш-функцию, мы можем почти гарантировать, что коллизии не возникнут для всех входных строк, которые мы могли бы использовать в игре. В конце концов, 32-битный хеш-код представляет более 4 млрд возможных значений. Таким образом, если хеш-функция равномерно распределит строки по всему этому громадному диапазону, мы вряд ли столкнемся с коллизией. В Naughty Dog мы начали задействовать вариант алгоритма CRC-32 для хеширования строк и за многие годы разработки *Uncharted* и *The Last of Us* столкнулись лишь с несколькими коллизиями. И когда это случилось, исправить их было просто — слегка изменить одну из строк, например добавить `2` или `b` или использовать совершенно другую, но синонимичную строку. Тем не менее Naughty Dog перешел на 64-битную функцию хеширования для *The Last of Us Part II* и всех будущих игр. Это должно исключить вероятность коллизий хешей, учитывая количество и типичные длины строк, которые мы применяем в любой игре.

Некоторые идеи реализации

Концептуально достаточно просто выполнить хеш-функцию для строк, чтобы сгенерировать идентификаторы строк. Однако на практике важно учитывать, *когда* будет вычисляться хеш. Большинство игровых движков, которые используют строковые идентификаторы, выполняют хеширование во время выполнения. В Naughty Dog мы разрешаем хеширование строк во время выполнения, а также с помощью функции *пользовательских литералов* C++11 преобразуем синтаксис `"any_string"_sid` непосредственно в хешированное целочисленное значение во время компиляции. Это позволяет использовать строковые идентификаторы везде, где может применяться целочисленная константа манифеста, включая константы `case` в конструкции `switch`. (Результат вызова функции, которая генерирует идентификатор строки во время выполнения, — не константа, поэтому ее нельзя использовать в качестве метки `case`.)

Генерирование идентификатора строки из строки иногда называют *интернированием* строки, потому что, помимо хеширования, она обычно добавляется в глобальную таблицу строк. Это позволяет в дальнейшем восстановить исходную строку из хеш-кода. Возможно, вы также захотите, чтобы ваши инструменты могли хешировать строки в идентификаторы строк. Таким образом, когда инструмент генерирует данные для потребления движком, строки уже будут хешированы.

Основная проблема с интернированием строки заключается в том, что это медленная операция. Функция хеширования должна выполняться для строки,

что может дорого стоить, особенно когда интернируется много строк. Кроме того, для строки должна быть выделена память, а сама строка должна быть скопирована в таблицу поиска. Так что если вы не генерируете строковые идентификаторы во время компиляции, лучше всего интернировать каждую строку только один раз и сохранить результат для последующего использования. Например, предпочтительно написать код, подобный следующему, потому что вторая реализация приводит к ненужному повторному интернированию строк каждый раз, когда вызывается функция `f()`:

```
static StringId sid_foo = internString("foo");
static StringId sid_bar = internString("bar");

// ...

void f(StringId id)
{
    if (id == sid_foo)
    {
        }
    else if (id == sid_bar)
    {
        // обрабатываем случай id == "bar"
    }
}
```

Следующий подход менее эффективен:

```
void f(StringId id)
{
    if (id == internString("foo"))
    {
        // обрабатываем случай id == "foo"
    }
    else if (id == internString("bar"))
    {
        // обрабатываем случай id == "bar"
    }
}
```

Вот одна из возможных реализаций `internString()`:

stringid.h

```
    typedef U32 StringId;

    extern StringId internString(const char* str);
```

stringid.cpp

```
    static HashTable<StringId, const char*> gStringIdTable;
```

```
StringId internString(const char* str)
{
    StringId sid = hashCrc32(str);

    HashTable<StringId, const char*>::iterator it
        = gStringIdTable.find(sid);

    if (it == gStringTable.end())
    {
        // Эта строка еще не была добавлена в таблицу. Добавим ее,
        // обязательно скопировав на случай,если память для оригинала
        // была выделена динамически и позже может быть освобождена.
        gStringTable[sid] = strdup(str);
    }

    return sid;
}
```

Другой прием, используемый в Unreal Engine, состоит в том, чтобы заключить идентификатор строки и указатель на соответствующий массив символов в стиле C в крошечный класс. В Unreal Engine этот класс называется `FName`. В Naughty Dog мы делаем то же самое и упаковываем строковые идентификаторы в класс `StringId`. Мы определяем макрос так, чтобы `SID ("any_string")` создавал экземпляр этого класса с его хешированным значением, созданным в нашем самостоятельно определяемом строковом литеральном синтаксисе `"any_string"_sid`.

Использование отладочной памяти для строк. При задействовании идентификаторов строк сами строки хранятся только для того, чтобы их могли применять люди. Когда вы выпускаете игру в продажу, вам почти наверняка не нужны строки — сама игра должна задействовать только идентификаторы. Поэтому рекомендуется хранить таблицу строк в области памяти, которой не будет в релизной версии игры. Например, комплект разработчика PS3 имеет 256 Мбайт памяти для релизных сборок, а также дополнительные 256 Мбайт отладочной памяти, недоступной в продаваемой версии. Если мы храним строки в отладочной памяти, нам не нужно беспокоиться об их влиянии на объем памяти в финальной игре. (Просто нужно быть осторожными, чтобы никогда не писать рабочий код, который зависит от доступных строк!)

6.4.4. Локализация

Локализация игры, да и любого программного проекта, — большое дело. Эту задачу лучше всего решать, планируя ее с первого дня и учитывая на каждом этапе разработки. Однако это происходит не так часто, как хотелось бы. Вот несколько советов, которые должны помочь вам подготовить игровой движок для локализации еще на стадии локализации. Подробное описание локализации программного обеспечения смотрите в источнике [34].

Юникод

Для большинства англоязычных разработчиков программного обеспечения проблема заключается в том, что они с самого рождения (или около того!) учатся воспринимать строки как массивы восьмибитных кодов символов ASCII, то есть символов, соответствующих стандарту ANSI. Строки ANSI отлично работают для языка с простым алфавитом, например английского. Но они просто не подходят для языков со сложными алфавитами, содержащими намного больше символов и иногда совершенно другие глифы, а не 26 букв английского языка. Для устранения ограничений стандарта ANSI была разработана система кодировки Unicode.

Основная идея Unicode — присвоить каждому символу или глифу из каждого языка, широко используемого по всему миру, уникальный шестнадцатеричный код, известный как *точка кода*. При хранении строки символов в памяти мы выбираем определенную *кодировку* — конкретное средство представления кодовых точек Unicode для каждого символа — и, следуя этим правилам, устанавливаем очередность битов в памяти, которая представляет строку. UTF-8 и UTF-16 — две общие кодировки. Вы должны выбрать конкретный стандарт кодирования, который наилучшим образом соответствует вашим потребностям.

Пожалуйста, отложите книгу прямо сейчас и прочитайте статью Джоэла Спольски *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and CharacterSets (No Excuses!)* («Абсолютный минимум для каждого разработчика программного обеспечения, который он определенно точно должен знать об Unicode и наборах символов (без исключений!)»). Она находится здесь: www.joelonsoftware.com/articles/Unicode.html. (Как только вы это сделаете, пожалуйста, возьмите книгу снова!)

UTF-32. Простейшая кодировка Unicode — это UTF-32. Здесь каждая кодовая точка кодируется в 32-битное (4-байтовое) значение. Эта кодировка расходует много памяти по двум причинам: во-первых, большинство строк в западноевропейских языках не используют кодовые точки с наивысшим значением, поэтому на символ обычно затрачивается в среднем 16 бит (2 байта). Во-вторых, самая высокая кодовая точка Unicode — 0x10FFFF, поэтому, даже если бы мы хотели создать строку, задействующую каждый возможный символ Unicode, нам все равно потребовался бы только 21 бит на символ, а не 32.

Тем не менее у UTF-32 есть преимущество — простота. Это кодирование *с фиксированной длиной*, означающее, что каждый символ занимает в памяти одно и то же количество битов (32 бита). Таким образом, мы можем определить длину любой строки UTF-32, разделив ее длину в байтах на четыре.

UTF-8. В схеме кодирования UTF-8 кодовые точки для каждого символа в строке сохраняются с использованием восьмибитной (однобайтовой) детализации, но некоторые из них занимают более 1 байта. Следовательно, число байтов, занимаемых символьной строкой UTF-8, не обязательно равно длине строки в символах. Это называется кодированием *переменной длины*, или *многобайтовым набором*

символов (multibyte character set, MBCS), поскольку каждый символ в строке может занимать один или несколько байтов памяти.

Одно из больших преимуществ кодировки UTF-8 — то, что она обратно совместима с кодировкой ANSI. Это работает, потому что первые 127 кодовых точек Unicode численно соответствуют старым кодам символов ANSI. Это означает, что каждый символ ANSI будет представлен ровно одним байтом в UTF-8, а строка символов ANSI может быть интерпретирована как строка UTF-8 без изменений.

Для представления кодовых точек с более высокими значениями стандарт UTF-8 использует многобайтовые символы. Каждый многобайтовый символ начинается с байта, старший бит которого равен 1, то есть его значение лежит в диапазоне 128... 255. Такие старшие байты никогда не появятся в строке ANSI, поэтому при различении однобайтовых и многобайтовых символов нет никакой неопределенности.

UTF-16. Кодирование UTF-16 применяет несколько более простой, хотя и более дорогой подход. Каждый символ в строке UTF-16 представлен одним или двумя 16-битными значениями. Кодирование UTF-16 известно как широкий набор символов (wide character set, WCS), потому что каждый символ имеет ширину не менее 16 бит вместо 8 бит, используемых обычными символами ANSI и их аналогами UTF-8.

В UTF-16 набор всех возможных кодовых точек Unicode разделен на 17 *плоскостей*, содержащих по 2^{16} кодовых точек. Первая плоскость известна как *базовая многоязычная плоскость* (basic mutilingual plane, BMP). Она содержит наиболее часто используемые кодовые точки в широком диапазоне языков. Таким образом, многие строки UTF-16 могут быть полностью представлены кодовыми точками в первой плоскости. Это означает, что каждый символ в такой строке представлен только *одним* 16-битным значением. Однако если в строке требуется символ из какой-то другой плоскости (они называются *дополнительными*), он представляется двумя последовательными 16-разрядными значениями.

Кодировка UCS-2 (двухбайтовый универсальный набор символов — 2-byte universal character set) — это ограниченное подмножество кодировки UTF-16, использующее *только* базовую многоязычную страницу. Таким образом, она не может представлять символы, чьи кодовые точки Unicode численно превышают 0xFFFF. Это упрощает формат, потому что каждый символ гарантированно занимает ровно 16 бит (2 байта). Другими словами, UCS-2 является кодировкой символов *фиксированной длины*, в то время как в общем случае UTF-8 и UTF-16 — это кодировки *переменной длины*.

Если мы априори знаем, что строка UTF-16 использует только кодовые точки из BMP (или имеем дело со строкой в кодировке UCS-2), то можем определить количество символов в строке, просто разделив число байтов на 2. Конечно, если в строке UTF-16 задействуются дополнительные плоскости, этот простой прием не сработает.

Обратите внимание на то, что кодировка UTF-16 может быть с прямым или обратным порядком байтов (см. подраздел 3.3.2) в зависимости от используемого

порядка целевого процессора. При сохранении текста UTF-16 на диске перед текстовыми данными обычно ставится *метка порядка байтов* (byte order mark, BOM), указывающая, в формате с каким порядком байтов хранятся отдельные 16-битные символы — прямым или обратным. (Конечно, это верно и для строковых данных в кодировке UTF-32.)

Типы char и wchar_t

Стандартная библиотека C/C++ определяет два типа данных для работы со строками символов — `char` и `wchar_t`. Тип `char` предназначен для применения со старыми строками ANSI и с многобайтовыми наборами символов (MBCS), включая UTF-8, но не ограничиваясь ею. Тип `wchar_t` — это широкий символьный тип, предназначенный для представления любой допустимой кодовой точки одним целым числом. Таким образом, размер этого типа зависит от компилятора и системы. Это может быть 8 бит в системе, которая вообще не поддерживает Unicode. Это может быть 16 бит, если для всех широких символов предполагается кодирование UCS-2 или используется многословное кодирование, такое как UTF-16. Или это может быть 32 бита, если UTF-32 выбрана в качестве широкой кодировки символов.

Из-за этой неопределенности в определении `wchar_t`, если нужно написать действительно переносимый код для обработки строк, потребуется определить собственный (-е) тип (-ы) символьных данных и предоставить библиотеку функций для работы с любой кодировкой Unicode, которую нужно поддерживать. Но если вы выбираете конкретные платформу и компилятор, то можете написать код в рамках этой конкретной реализации, потеряв некоторую переносимость.

Преимущества и недостатки использования типа данных `wchar_t` хорошо описаны в следующей статье: icu-project.org/docs/papers/unicode_wchar_t.html.

Unicode под Windows

В Windows тип данных `wchar_t` применяется исключительно для строк Unicode, кодированных в UTF-16, а тип `char` — для строк ANSI и устаревших кодировок строк *кодовой страницы Windows*. Поэтому при чтении документации по Windows API термин Unicode всегда синонимичен выражению «широкий набор символов» (WCS) и кодировке UTF-16. Это немного сбивает с толку, потому что, конечно, строки Unicode, как правило, могут кодироваться в нешироком многобайтовом формате UTF-8.

Windows API определяет три набора функций управления символами/строками: один для строк ANSI (SBCS) однобайтового набора символов, один для строк многобайтового набора символов (MBCS) и один для строк набора широких символов (WCS). По сути, функции ANSI — это строковые функции в стиле C старой школы, на которых все мы выросли. Строковые функции MBCS обрабатывают различные многобайтовые кодировки и в первую очередь предназначены для

работы с устаревшими кодировками кодовых страниц Windows. Функции WCS обрабатывают строки Unicode UTF-16.

Во всем Windows API префикс или суффикс *w*, *wcs* или W указывает на кодировку широкого набора символов (UTF-16), префикс или суффикс *mb* — на многобайтовое кодирование, а префикс или суффикс *a* или *A* либо отсутствие префикса или суффикса — на кодировку ANSI или кодовые страницы Windows. Стандартная библиотека C++ использует аналогичное соглашение: например, `std::string` — это ее строковый класс ANSI, а `std::wstring` — его эквивалент широких символов. К сожалению, имена функций не всегда согласованы на 100 %. Все это немного запутывает программистов, которые не в курсе нюансов. (Но *вы* не один из этих программистов!) В табл. 6.2 приведено несколько примеров.

Таблица 6.2. Варианты некоторых распространенных строковых функций стандартной библиотеки C для использования с ANSI, широкими и многобайтовыми наборами символов

ANSI	WCS	MBCS
strcmp()	wcscmp()	_mbscmp()
strcpy()	wcscpy()	_mbscpy()
strlen()	wcslen()	_mbstrlen()

Windows также предоставляет функции для перевода между символьными строками ANSI, многобайтовыми строками и широкими строками UTF-16. Например, `wcstombs()` преобразует широкую строку UTF-16 в многобайтовую строку в соответствии с текущей активной настройкой *региона*.

Windows API имеет небольшую встроенную в препроцессор функцию, что позволяет вам писать код, который по крайней мере внешне переносим между широким (Unicode) и нешироким (ANSI/MBCS) кодированием строк. Универсальный символьный тип данных `TCHAR` определяется как `typedef` для `char` при сборке приложения в режиме ANSI и определен как `typedef` для `wchar_t` при сборке приложения в Unicode-режиме. Макрос `_T()` используется для преобразования восьмибитного строкового литерала (например, `char * s = "это строка ";`) в широкий строковый литерал (например, `wchar_t * s = L"это строка ";`) при компиляции в режиме Unicode. Аналогичным образом предоставляется набор «поддельных» API-функций, которые автоматически переходят в свой 8- или 16-битный вариант в зависимости от того, работаете вы в режиме Unicode или нет. Эти волшебные независимые от набора символов функции имеют имена без префикса или суффикса либо с префиксом или суффиксом *t*, *tcs* или *T*.

Полную документацию по всем этим функциям можно найти на сайте Microsoft MSDN. Вот ссылка на документацию по `strcmp()` и т. п., откуда вы можете легко перейти к другим связанным функциям работы со строками, используя древовидное меню в левой части страницы или панель поиска: msdn2.microsoft.com/en-us/library/kk6xf663(VS.80).aspx.

Unicode на консолях

Пакет разработки программного обеспечения Xbox 360 (XDK) использует WCS в основном для всех строк — даже для внутренних, таких как пути к файлам. Это, безусловно, неплохой подход к проблеме локализации, он обеспечивает очень согласованную обработку строк в XDK. Однако кодирование UTF-16 немного расточительно для памяти, поэтому разные игровые движки могут задействовать разные соглашения. В Naughty Dog мы задействуем восьмибитные строки символов в движке и обрабатываем иностранные языки с помощью кодировки UTF-8. Выбор кодировки не особенно важен, если, работая над проектом, вы выберете ее как можно раньше и будете придерживаться постоянно.

Другие проблемы локализации

Даже после того, как вы адаптировали свое программное обеспечение для использования символов Unicode, вы все еще сталкиваетесь с множеством проблем локализации. Одна из них заключается в том, что строки не единственное место, где возникают проблемы локализации. Аудиоклипы, включая записанные голоса, должны быть переведены. Текстуры могут содержать английские слова, которые требуют перевода. Многие символы имеют разные значения в разных культурах. Даже что-то столь безобидное, как знак, запрещающий курение, может быть неверно истолковано в другой культуре. Кроме того, некоторые страны по-разному определяют границы возрастного ценза. Например, в Японии в играх с рейтингом Teen (для подростков) не разрешается показывать любое количество крови, тогда как в Северной Америке небольшие красные брызги считаются приемлемыми.

Таблица 6.3. Пример строковой базы данных, используемой для локализации

Id	Английский	Французский
p1score	Player 1 Score	Joueur 1 Score
p2score	Player 2 Score	Joueur 2 Score
p1wins	Player one wins!	Joueur un gagne!
p2wins	Player two wins!	Joueur deux gagne!

Для строк есть и другие детали, о которых нужно побеспокоиться. Вам нужно будет управлять базой данных всех читаемых человеком в игре строк, чтобы обеспечить удобство и надежность перевода. Программное обеспечение должно отображать правильный язык с учетом параметров установки пользователя. Форматирование строк может быть совершенно разным на разных языках — например, в китайском иногда пишут вертикально, а иврит читают справа налево. Длина строк будет сильно различаться от языка к языку. Вам также необходимо решить, поставлять ли один DVD или диск Blu-ray, который содержит все языки, или разные диски для разных территорий.

Наиболее важными компонентами системы локализации будут центральная база данных читаемых человеком строк и внутриигровая система для поиска этих строк по идентификатору. Допустим, вы хотите, чтобы в режиме «один на один» отображался счет каждого игрока с метками Player 1 Score (Счет игрока 1) и Player 2 Score (Счет игрока 2) и текст Player 1 Wins (Игрок 1 выиграл) или Player 2 Wins (Игрок 2 выиграл) в конце раунда. Эти четыре строки будут храниться в базе данных локализации под уникальными идентификаторами, которые понятны вам, как разработчику игры. Таким образом, база данных может использовать идентификаторы `"p1score"`, `"p2score"`, `"p1wins"` и `"p2wins"`. Как только строки игры будут переведены на французский язык, база данных станет выглядеть как табл. 6.3. Дополнительные столбцы можно добавлять для каждого нового языка, поддерживаемого игрой.

Точный формат базы данных зависит от вас. Она может быть простым листом Microsoft Excel, который сохранен в виде файла значений, разделенных запятыми (CSV), и проанализирован игровым движком, или сложной, как полноценная база данных Oracle. Специфика базы данных строк не слишком много значит для игрового движка, поскольку она может считывать идентификаторы строк и соответствующие строки Unicode для любых языков, поддерживаемых игрой. (Однако специфика базы данных может быть *очень* важна с практической точки зрения в зависимости от организационной структуры вашей игровой студии. Небольшая студия с собственными переводчиками может, вероятно, пользоваться таблицей Excel, размещенной на сетевом диске. Но большая студия с филиалами в Великобритании, Европе, Южной Америке и Японии, возможно, посчитает гораздо более удобной распределенную базу данных.)

Во время выполнения вам необходимо предоставить простую функцию, которая возвращает строку Unicode на используемом в данный момент языке, учитывая уникальный идентификатор этой строки. Функция может быть объявлена так:

```
wchar_t getLocalizedString(const char* id);
```

и применена так:

```
void drawScoreHud(const Vector3& score1Pos,
                  const Vector3& score2Pos)
{
    renderer.displayTextOrtho(getLocalizedString("p1score"),
                              score1Pos);

    renderer.displayTextOrtho(getLocalizedString("p2score"),
                              score2Pos);

    // ...
}
```

Конечно, вам понадобится способ установить этот язык глобально. Можно сделать это с помощью настройки конфигурации, которая фиксируется во время установки игры. Или можете позволить пользователям изменять текущий язык на лету через игровое меню. В любом случае настройка несложна для реализации,

это может быть так же просто, как глобальная целочисленная переменная, указывающая индекс графы в таблице строк для чтения (например, первая графа может быть английской, вторая — французской, третья — испанской и т. д.).

После того как инфраструктура создана, программисты должны всегда помнить, что для пользователя нужно отображать *только обработанные локализацией строки*. Они всегда должны использовать идентификатор строки в базе данных и вызывать функцию поиска, чтобы получить соответствующую строку.

Пример из практики: инструмент локализации Naughty Dog

В Naughty Dog применяется база данных локализации, которую мы разработали сами. Бэкенд инструмента локализации состоит из базы данных MySQL, расположенной на сервере, доступном как для разработчиков из Naughty Dog, так и для внешних компаний, с которыми мы работаем над переводом на различные языки текстовых и речевых аудиоклипов, использующихся в наших играх. Внешний интерфейс — это веб-интерфейс, который обращается к базе данных, позволяя пользователям просматривать все текстовые и звуковые ресурсы, редактировать их содержимое, предоставлять переводы для каждого ресурса, искать ресурсы по идентификатору или содержимому и т. д.

В инструменте локализации Naughty Dog любой актив представляет собой либо строку (для использования в меню или HUD), либо речевой аудиоклип с необязательным текстом субтитров (для применения в диалоге в игре или в виде ролика). У каждого актива есть уникальный идентификатор, который имеет вид идентификатора хешированной строки (см. подраздел 6.4.3). Если строка требуется для использования в меню или HUD, мы ищем ее по идентификатору и возвращаем строку Unicode (UTF-8), подходящую для отображения на экране. Если должна быть воспроизведена строка диалога, мы также ищем аудиоклип по идентификатору и задействуем обработчик данных для поиска соответствующих субтитров, если они есть. Субтитры обрабатываются так же, как строки меню или HUD, в том смысле, что они возвращаются API инструмента локализации как строка UTF-8, подходящая для отображения.

На рис. 6.10 показан основной интерфейс инструмента локализации, в данном случае отображаемый в веб-браузере Chrome. Здесь вы видите, что пользователь набрал идентификатор `MENU_NEWGAME`, чтобы найти строку NEW GAME (используется в главном меню игры для запуска новой игры).

На рис. 6.11 показан подробный вид актива `MENU_NEWGAME`. Если пользователь нажимает кнопку Text Translations (Перевод текста) в верхнем левом углу окна данных актива, появляется экран (рис. 6.12), позволяющий пользователю вводить или редактировать различные переводы строки.

На рис. 6.13 приведена другая вкладка главной страницы инструмента локализации, на которой перечислены аудиоречевые активы. Наконец, на рис. 6.14 показан подробный вид актива для речевого набора `BADA_GAM_MIL_ESCAPE_OVERPASS_001` (We missed all the action — «Мы пропустили все действия»), выводящий перевод этой строки диалога на некоторые из поддерживаемых языков.

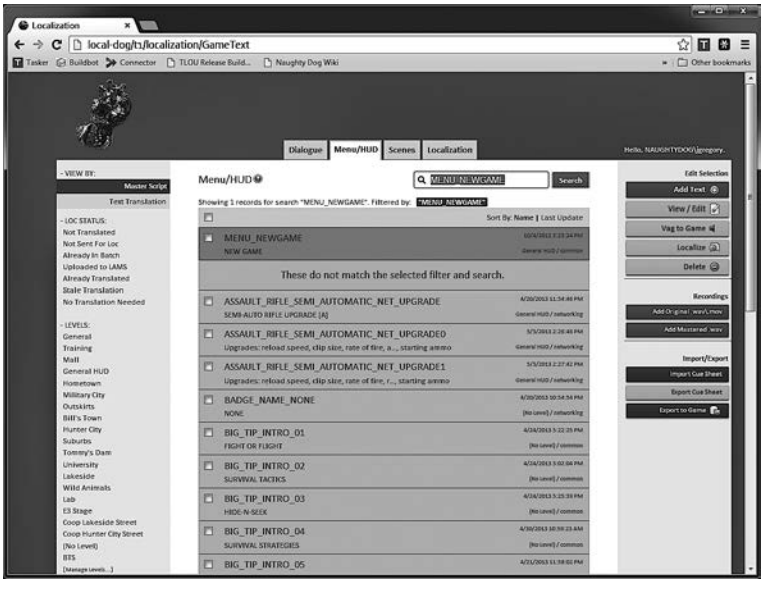


Рис. 6.10. Главное окно инструмента локализации Naughty Dog, показывающее список чистых текстовых ресурсов, применяемых в меню и HUD. Пользователь только что выполнил поиск ресурса под названием MENU_NEWGAME

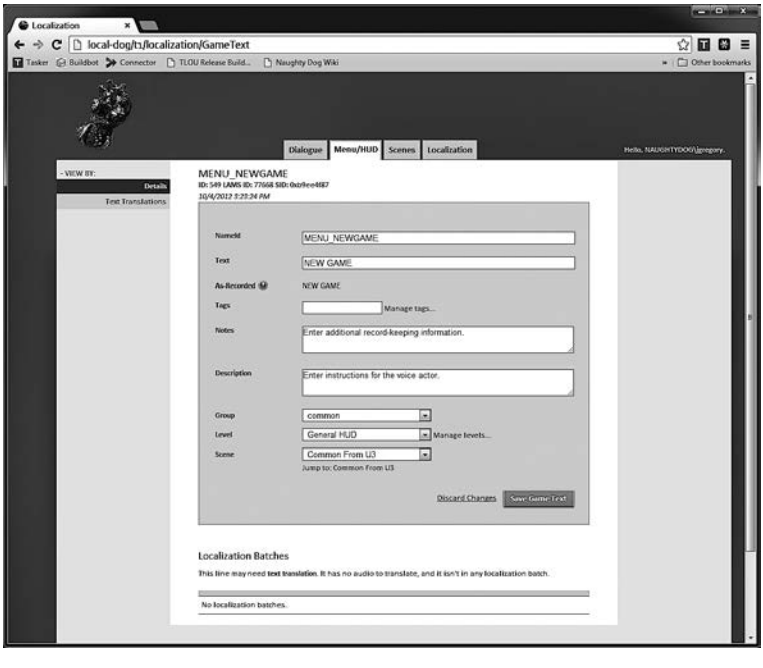


Рис. 6.11. Подробный вид активов, показывающий строку MENU_NEWGAME

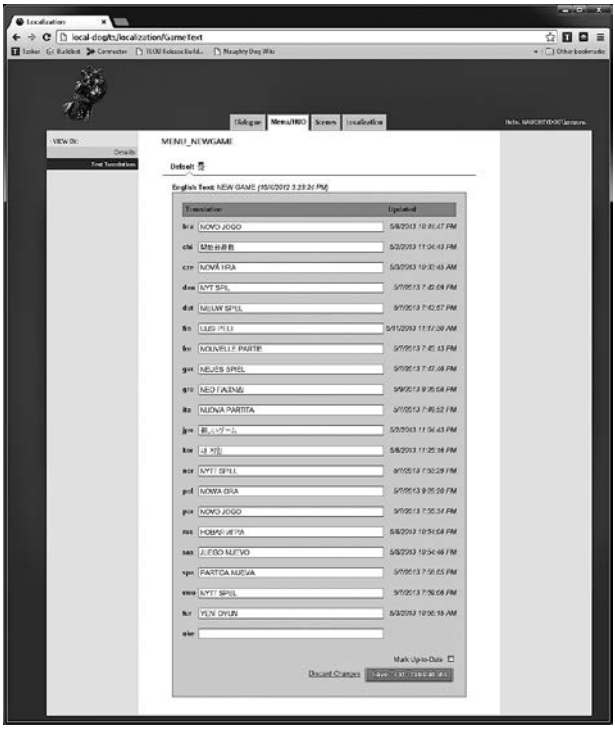


Рис. 6.12. Текстовые переводы строки NEW GAME на все языки, поддерживаемые игрой The Last of Us от Naughty Dog

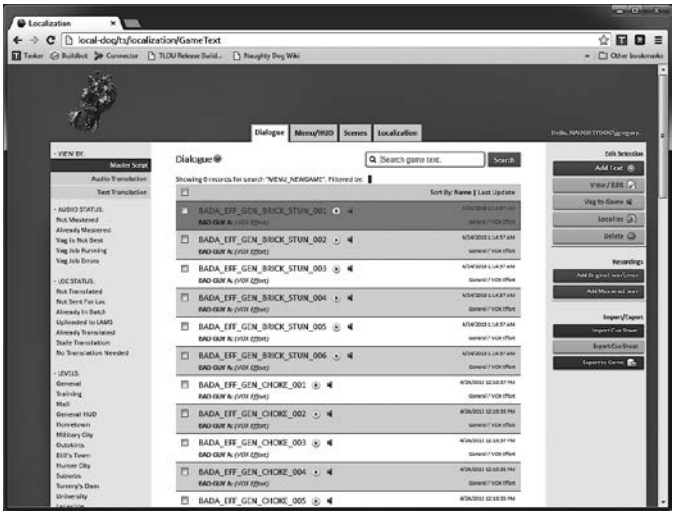


Рис. 6.13. Снова появилось главное окно инструмента локализации Naughty Dog, на этот раз показывающее список аудиоактивов с текстом субтитров

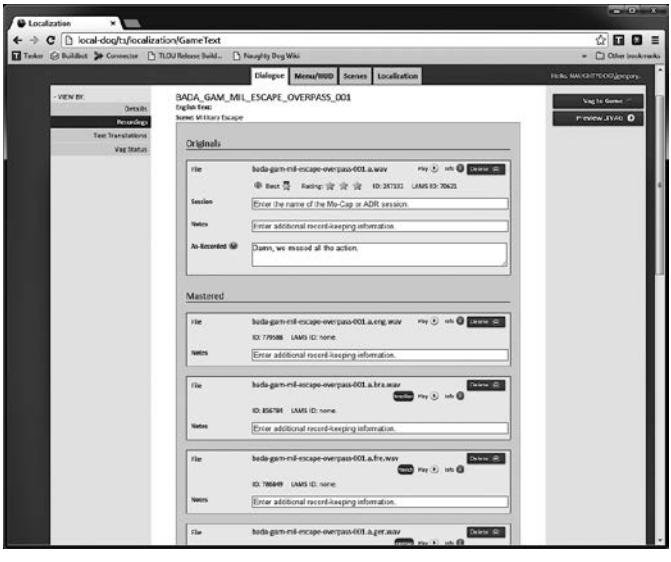


Рис. 6.14. Подробное представление актива, показывающее переводы для речевого актива BADA_GAM_MIL_ESCAPE_OVERPASS_001 (Мы пропустили все действия)

6.5. Конфигурация движка

Игровые движки — сложные «существа», они всегда имеют множество настраиваемых параметров. До некоторых из этих опций игрок может добраться, открыв одно или несколько меню. Например, игра может предоставлять параметры, связанные с качеством графики, громкостью музыкальных и звуковых эффектов или конфигурацией контроллера. Другие параметры создаются только для команды разработчиков игр, и их либо скрывают, либо полностью удаляют из игры перед ее выпуском. Например, максимальная скорость персонажа игрока может быть выставлена как опция, чтобы регулировать ее во время разработки, но ее можно жестко задать перед релизом.

6.5.1. Параметры загрузки и сохранения

Настраиваемая опция может быть реализована просто как глобальная переменная или переменная — член одноэлементного класса. Однако настраиваемые параметры не особенно полезны, если их значения не могут быть настроены и сохранены на жестком диске, карте памяти или другом носителе данных, а затем получены игрой. Существует несколько простых способов загрузить и сохранить параметры конфигурации.

- *Файлы конфигурации*. Безусловно, самый распространенный метод сохранения и загрузки параметров конфигурации — это размещение их в одном или нескольких текстовых файлах. Формат этих файлов сильно различается от движка

к движку, но обычно он очень прост. Например, файлы INI Windows, которые используются средством визуализации OGRE, состоят из плоских списков пар «ключ — значение», сгруппированных в логические секции. Формат JSON — еще один распространенный вид настраиваемых файлов параметров игры. XML — тоже жизнеспособный вариант, хотя большинство разработчиков в настоящее время считают JSON менее многословным и более простым для чтения, чем XML.

- *Сжатые бинарные файлы.* В большинстве современных консолей есть жесткие диски, но старые консоли не могли позволить себе такую роскошь. В результате все игровые приставки, начиная с Super Nintendo Entertainment System (SNES), оснащены фирменными сменными картами памяти, которые позволяют как читать, так и записывать данные. Варианты игры иногда хранятся на этих картах вместе с сохраненными играми. Сжатые двоичные файлы являются предпочтительным форматом для карты памяти, поскольку объем последних часто очень ограничен.
- *Реестр Windows.* Операционная система Microsoft Windows предоставляет глобальную базу данных опций, известную как реестр. Он хранится в виде дерева, где внутренние узлы *(ключи реестра)* действуют как папки файлов, а конечные узлы хранят отдельные опции в виде пар «ключ — значение». При этом я не рекомендую использовать реестр Windows для хранения информации о конфигурации движка. Реестр представляет собой монолитную базу данных, которая может быть легко повреждена, потеряна (при переустановке Windows) или выброшена из синхронизации с файлами в файловой системе. Подробнее о недостатках реестра Windows см. blog.codinghorror.com/was-the-windows-registry-a-good-idea/.
- *Параметры командной строки.* Командная строка может быть учтена при настройке параметров. Движок может предоставить механизм для управления любой опцией в игре через командную строку или показать здесь лишь некоторые ее опции.
- *Переменные среды.* На персональных компьютерах под управлением Windows, Linux или MacOS переменные среды иногда используются для хранения параметров конфигурации.
- *Онлайн-профили пользователей.* С появлением игровых онлайн-сообществ, таких как Xbox Live, каждый пользователь может создать профиль и применять его для сохранения достижений, приобретенных и разблокируемых игровых функций, параметров игры и другой информации. Данные хранятся на центральном сервере, и игрок может получить к ним доступ везде, где есть подключение к Интернету.

6.5.2. Параметры для каждого пользователя

Большинство игровых движков различают глобальные опции и опции для отдельного пользователя. Это необходимо, потому что большинство игр позволяют каждому игроку настраивать игру по своему вкусу. Такая концепция полезна и при разработке игры, поскольку позволяет каждому программисту, художнику и дизайнеру настраивать свою рабочую среду, не затрагивая других членов команды.

Очевидно, необходимо позаботиться о том, чтобы сохранить параметры любого пользователя таким образом, чтобы каждый игрок видел только свои параметры, а не параметры других игроков на одном компьютере или одной консоли. В консольной игре пользователю обычно разрешается сохранять свои успехи вместе с опциями для каждого пользователя, такими как настройки контроллера, в слотах на карте памяти или жестком диске. Эти слоты обычно реализуются в виде файлов на данном носителе.

На компьютере под управлением Windows каждый пользователь имеет в папке C:\Users собственную папку, содержащую Рабочий стол, папку Мои документы, историю посещений интернет-сайтов, временные файлы и т. д. Скрытая подпапка с именем AppData используется для хранения информации о пользователях для каждого приложения. Все приложения создают папку в AppData и могут хранить в ней необходимую им информацию для каждого пользователя.

Игры для Windows иногда хранят данные конфигурации для каждого пользователя в реестре. Он организован в виде дерева, и один из дочерних элементов верхнего уровня корневого узла, называемый HKEY_CURRENT_USER, хранит настройки для любого пользователя, вошедшего в систему. Каждый пользователь имеет собственное поддерево в реестре (оно хранится в поддереве верхнего уровня HKEY_USERS), а HKEY_CURRENT_USER — это просто псевдоним поддерева текущего пользователя. Таким образом, игры и другие приложения могут управлять параметрами конфигурации для каждого пользователя, просто читая и записывая их в ключи в поддереве HKEY_CURRENT_USER.

6.5.3. Управление конфигурацией в некоторых реальных движках

В этом разделе мы кратко рассмотрим, как некоторые реальные игровые движки управляют своими параметрами конфигурации.

Пример: cvars в Quake

Семейство движков Quake использует систему управления конфигурацией, известную как *консольные переменные* или сокращенно *cvars*. Cvar — это просто глобальная переменная с плавающей точкой или строковое значение, которое можно проверить и изменить с игровой консоли Quake. Значения некоторых cvars могут быть сохранены на диск и позже загружены движком.

Во время выполнения cvars хранятся в глобальном связанном списке. Каждая cvar является динамически размещенным экземпляром переменной `struct cvar_t`, которая содержит имя переменной, ее значение в виде строки или числа с плавающей точкой, набора битов флага и указателя на следующую cvar в связанном списке всех cvars. Доступ к cvars осуществляется путем вызова `Cvar_Get()`, который создает переменную, если она еще не существует, и изменяет ее вызовом `Cvar_Set()`. Один из битовых флагов, `CVAR_ARCHIVE`, контролирует, будет ли сохранена cvar в файл конфигурации с именем `config.cfg`. Если этот флаг установлен, значение cvar будет сохраняться в течение нескольких запусков игры.

Пример: OGRE

Движок рендеринга OGRE использует набор текстовых файлов в формате INI Windows для своих параметров конфигурации. По умолчанию параметры хранятся в трех файлах, каждый из которых находится в той же папке, что и исполняемая программа:

- `plugins.cfg` содержит параметры, указывающие, какие дополнительные подключаемые модули движка включены и где их найти на диске;
- `resources.cfg` содержит *путь поиска*, указывающий, где можно найти игровые ресурсы (то есть медиа и пр.);
- `ogre.cfg` содержит богатый набор опций, определяющих, какой рендерер (DirectX или OpenGL) использовать, а также предпочтительный режим видео, размер экрана и т. д.

В стандартной комплектации OGRE не предоставляет механизма для хранения параметров конфигурации для каждого пользователя. Однако исходный код OGRE находится в свободном доступе, поэтому его довольно легко изменить для поиска файлов конфигурации в домашнем каталоге пользователя, а не в папке, содержащей исполняемый файл. Класс `Ogre::ConfigFile` позволяет легко писать код, который также читает и записывает новые файлы конфигурации.

Пример: серии Uncharted и The Last of Us

Движок Naughty Dog использует несколько механизмов конфигурации.

Настройки игрового меню. Движок Naughty Dog поддерживает мощную систему игровых меню, позволяя разработчикам контролировать глобальные параметры конфигурации и вызывать команды. Типы данных настраиваемых параметров должны быть довольно простыми (в основном булевы значения, целочисленные переменные и переменные с плавающей точкой), но это ограничение не помешало разработчикам из Naughty Dog создать сотни полезных опций на основе меню.

Каждый параметр конфигурации реализован в виде глобальной переменной или члена одноэлементной структуры или класса. При создании опции меню, которая управляет параметром, предоставляется адрес переменной, и пункт меню напрямую управляет ее значением. Например, следующая функция создает элемент подменю, содержащий некоторые опции для *рельсовых транспортных средств* Naughty Dog (простые транспортные средства, которые едут на сплайнах, используемых практически во всех играх Naughty Dog, начиная с уровня погони за джипом *Outof the Frying Pan* в *Uncharted: Drake's Fortune* и до серии уровней, где преследовалась колонна грузовиков/джипов в *Uncharted 4*). Он определяет пункты меню, управляющие тремя глобальными переменными: двумя логическими значениями и одним значением с плавающей точкой. Элементы собираются в меню, и возвращается специальный элемент, который вызывает меню при выборе. Предположительно код, вызывающий данную функцию, добавляет этот элемент в родительское меню, которое она создает:

```
DMENU::ItemSubmenu * CreateRailVehicleMenu()
{
    extern bool g_railVehicleDebugDraw2D;
    extern bool g_railVehicleDebugDrawCameraGoals;
    extern float g_railVehicleFlameProbability;

    DMENU::Menu * pMenu
        = new DMENU::Menu("RailVehicle");

    pMenu->PushBackItem(
        new DMENU::ItemBool("Draw 2D Spring Graphs",
            DMENU::ToggleBool,
            &g_railVehicleDebugDraw2D));

    pMenu->PushBackItem(
        new DMENU::ItemBool("Draw Goals (Untracked)",
            DMENU::ToggleBool,
            &g_railVehicleDebugDrawCameraGoals));

    DMENU::ItemFloat * pItemFloat;
    pItemFloat = new DMENU::ItemFloat(
        "FlameProbability",
        DMENU::EditFloat, 5, "%5.2f",
        &g_railVehicleFlameProbability);

    pItemFloat->SetRangeAndStep(0.0f, 1.0f, 0.1f, 0.01f);
    pMenu->PushBackItem(pItemFloat);

    DMENU::ItemSubmenu * pSubmenuItem;
    pSubmenuItem = new DMENU::ItemSubmenu(
        "RailVehicle...", pMenu);

    return pSubmenuItem;
}
```

Значение любой опции можно сохранить, просто пометив ее круглой кнопкой на джойстике Dualshock, когда выбран соответствующий пункт меню. Настройки меню сохраняются в текстовом файле в стиле INI, что позволяет сохраненным глобальным переменным восстанавливать значения при перезапуске игры. Возможность контролировать то, какие опции сохраняются *для каждого пункта меню*, очень полезна, потому что любая несохраненная опция будет иметь заданное программистом значение по умолчанию. Если программист изменит значение по умолчанию, все пользователи будут видеть новое значение, если, конечно, пользователь не сохранил пользовательское значение для этой конкретной опции.

Аргументы командной строки. Движок Naughty Dog сканирует командную строку для поиска предопределенного набора специальных опций. Можно указать имя уровня для загрузки, а также ряд других часто используемых аргументов.

Определения схемы данных. Подавляющее большинство информации о конфигурации движка и игры в движке Naughty Dog, задействуемом для создания

серий *Uncharted* и *The Last of Us*, указывается с использованием подобного Lisp языка, называемого Scheme (схема). Применяя собственный компилятор данных, структуры данных, определенные на языке Scheme, преобразуются в двоичные файлы, которые может загружать движок. Компилятор данных также выдает заголовочные файлы, содержащие объявления структуры C, для каждого типа данных, определенного в Scheme. Эти заголовочные файлы позволяют движку правильно интерпретировать данные, содержащиеся в загруженных двоичных файлах. Двоичные файлы можно даже перекомпилировать и перезагружать на лету, что позволяет разработчикам изменять данные в Scheme и сразу же видеть последствия их изменений, если элементы данных не добавляются и не удаляются, поскольку для этого потребуется перекомпиляция движка.

Следующий пример иллюстрирует создание структуры данных, определяющей свойства анимации. Затем он экспортирует в игру три уникальные анимации. Возможно, раньше вы никогда не читали код схемы, но в этом сравнительно простом примере все должно быть понятно. Одна странность, которую вы заметите, заключается в том, что в символах схемы разрешены дефисы, поэтому `simple-animation` — это один символ (в отличие от C/C++, где `simple-animation` будет частным двух переменных, `simple` и `animation`).

simple-animation.scm

```
;; Определяем новый тип данных с именем simple-animation.
(deftype simple-animation ()
    (
        (name             string)
        (speed            float  :default 1.0)
        (fade-in-seconds  float  :default 0.25)
        (fade-out-seconds float  :default 0.25)
    )
)

;; Теперь определим три экземпляра этой структуры данных...
(define-export anim-walk
    (new simple-animation
        :name "walk"
        :speed 1.0
    )
)

(define-export anim-walk-fast
    (new simple-animation
        :name "walk"
        :speed 2.0
    )
)

(define-export anim-jump
    (new simple-animation
        :name "jump"
```

```
            :fade-in-seconds 0.1
            :fade-out-seconds 0.1
        )
    )
```

Этот код схемы будет генерировать следующий заголовочный файл C/C++:

simple-animation.h

```
    // ПРЕДУПРЕЖДЕНИЕ: этот файл был автоматически
    // сгенерирован из схемы данных. Не редактируйте его вручную.

    struct SimpleAnimation
    {
        const char* m_name;
        float       m_speed;
        float       m_fadeInSeconds;
        float       m_fadeOutSeconds;
    };
```

В игре данные можно прочитать, вызвав функцию `LookupSymbol()`, которая определяется шаблоном возвращаемого типа данных следующим образом:

```
#include "simple-animation.h"
void someFunction()
{
    SimpleAnimation* pWalkAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-walk"));

    SimpleAnimation* pFastWalkAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-walk-fast"));

    SimpleAnimation* pJumpAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-jump"));

    // Использовать данные здесь...
}
```

Эта система обеспечивает программистам большую свободу в определении всех видов данных конфигурации — от простых булевых значений, чисел с плавающей точкой и строковых опций до сложных, вложенных, взаимосвязанных структур данных. Они используются для указания подробных деревьев анимации, физических параметров, механики игрока и т. д.

7 Ресурсы и файловая система

Игры по своей природе мультимедийны. Поэтому игровой движок должен быть способен загружать различные виды мультимедиа: растровые изображения текстур, данные трехмерных сеток, анимацию, аудиоклипы, данные, относящиеся к коллизиям и физике, макеты игрового мира и т. д. — и управлять ими. Более того, поскольку памяти обычно не хватает, игровой движок должен гарантировать, что в любой момент времени в память загружается только одна копия каждого медиафайла. Например, если пять сеток используют одну и ту же текстуру, мы бы хотели, чтобы в памяти была только одна ее копия, а не пять. Большинство игровых движков задействуют какой-нибудь *менеджер ресурсов* (он же *менеджер активов*, он же *менеджер медиафайлов*) для загрузки бесчисленных ресурсов, составляющих современную 3D-игру, и управления ими.

Все менеджеры ресурсов активно работают с файловой системой. На персональном компьютере файловая система предоставляется программисту через библиотеку вызовов операционной системы. Однако игровые движки часто оборачивают собственный API файловой системы в API, специфичный для движка, по двум основным причинам. Во-первых, движок может быть кросс-платформенным, и в этом случае API файловой системы игрового движка способен обезопасить остальную часть программного обеспечения от различий между целевыми аппаратными платформами. Во-вторых, API файловой системы операционной системы может не предоставлять все инструменты, необходимые игровому движку. Например, многие движки поддерживают *потоковую передачу файлов* (возможность загружать данные на лету во время игры), однако большинство операционных систем не предоставляют API файловой системы «из коробки». Консольные игровые движки также должны давать доступ к различным съемным и несъемным носителям, от карт памяти до дополнительных жестких дисков, дисков DVD-ROM или Blu-ray и сетевых файловых систем (например, Xbox Live или PlayStation Network, PSN). Различия между типами носителей также могут быть спрятаны за API файловой системы игрового движка.

В этой главе мы сначала рассмотрим виды API файловой системы, которые применяются в современных 3D-игровых движках. Затем поговорим о том, как работает стандартный менеджер ресурсов.

7.1. Файловая система

API файловой системы игрового движка обычно предназначен для следующих элементов функциональности:

- оперирования путями к файлам и их именами;
- открытия, закрытия, чтения и записи отдельных файлов;
- сканирования содержимого каталога;
- обработки асинхронных запросов IRP (I/O request packet — пакет запроса ввода-вывода) к файлам (для потоковой передачи).

Кратко рассмотрим каждый из них в следующих подразделах.

7.1.1. Имена файлов и пути к ним

Путь — это строка, описывающая расположение файла или папки в иерархии файловой системы. Форматы путей к файлам немного различаются в разных операционных системах, тем не менее пути всегда имеют практически одинаковую структуру. Путь обычно имеет следующую форму:

```
том/каталог1/каталог2/.../каталогN/имя-файла
```

или

```
том/каталог1/каталог2/.../каталог(N – 1)/каталогN
```

Другими словами, он обычно состоит из необязательного *обозначения тома*, за которым идет последовательность *компонентов пути*, отделенных друг от друга специальным разделительным символом, например прямой или обратной косой чертой (`/` или `\`). Каждый компонент обозначает каталог на маршруте от корневого каталога до рассматриваемого файла или каталога. Если путь указывает на местоположение файла, последним компонентом в нем будет имя файла, в противном случае он будет указывать на целевой каталог. Корневой каталог обычно указывается с помощью пути, состоящего из необязательного обозначения тома, за которым следует один разделяющий символ (например, `/` в UNIX или `C:\` в Windows).

Различия между операционными системами

Каждая операционная система вносит небольшие изменения в общую структуру пути. Вот некоторые из ключевых различий между Microsoft DOS, Microsoft Windows, операционной системой семейства UNIX и Apple Macintosh OS.

- UNIX использует косую черту, или слеш (`/`), в качестве разделителя компонентов пути, в то время как в DOS и более ранних версиях Windows для этого применялась обратная косая черта, или обратный слеш (`\`). Последние версии Windows позволяют разделять компоненты пути прямой или обратной косой чертой, хотя некоторые приложения по-прежнему не принимают обычную косую черту.

- В Mac OS 8 и 9 разделяющим символом служит двоеточие (`:`). Mac OS X основана на BSD UNIX, поэтому она поддерживает косую черту UNIX.
- В некоторых файловых системах пути к файлам и их имена чувствительны к регистру (например, UNIX и его варианты), в других — нечувствительны (например, Windows). Это может вызвать проблемы во время разработки в ходе работы с файлами в разных операционных системах или при написании кросс-платформенной игры. (Например, следует считать файл активов `EnemyAnims.json` эквивалентным файлу активов `enemyanims.json` или нет?)
- UNIX и его вариации не поддерживают тома как отдельные иерархии каталогов. Вся файловая система содержится в единой цельной иерархии, а локальные диски, сетевые диски и другие ресурсы при подключении оказываются поддеревьями в основной иерархии. В результате в системах UNIX в пути никогда не указывается том.
- В Microsoft Windows том может быть указан двумя способами. Локальный диск обозначается одной буквой, после которой стоит двоеточие (например, вездесущая `C:`). Удаленный сетевой ресурс может быть смонтирован так, чтобы он выглядел как локальный диск, или на него можно ссылаться через обозначение тома, состоящее из двух обратных косых черт, за которыми указываются имя удаленного компьютера и имя общего каталога или ресурса на нем (например, `\\some-computer\some-share`). Двойная обратная косая черта является примером универсального соглашения об именах (UNC).
- В DOS и более ранних версиях Windows имя файла могло иметь длину до восьми символов с *расширением* из трех символов, которое отделено точкой от основного имени файла. Расширение описывает тип файла, например `.txt` для текстового файла или `.exe` для исполняемого файла. В последних реализациях Windows имена файлов могут содержать любое количество точек (как и в UNIX), но символы после последней точки все еще интерпретируются как расширение файла во многих приложениях, включая Windows Explorer.
- Различные операционные системы запрещают использовать определенные символы в именах файлов и каталогов. Например, в Windows или DOS двоеточие в пути может стоять только после буквы диска определителя тома. Некоторые операционные системы позволяют применять подмножество этих специальных символов в пути, когда он полностью заключен в кавычки или когда запрещенный символ *экранирован* стоящей перед ним обратной косой чертой или другим специальным *экранирующим символом*. Например, в Windows имена файлов и каталогов могут содержать пробелы, но такой путь в определенных контекстах должен быть заключен в двойные кавычки.
- Как UNIX, так и Windows используют концепцию текущего рабочего каталога (*current working directory*, CWD, или *present working directory*, PWD). Рабочий каталог может быть задан из командной оболочки с помощью команды `cd`

(*change directory* — «изменить каталог») в обеих операционных системах, это можно сделать, введя `cd` без аргументов в Windows или выполнив команду `pwd` в UNIX. В UNIX есть только один текущий рабочий каталог. В Windows каждый том имеет собственный текущий рабочий каталог.

- Операционные системы, которые поддерживают несколько томов, как Windows, также имеют концепцию *текущего рабочего тома*. В командной оболочке Windows текущий том можно установить, введя его букву диска и двоеточие, а затем нажав клавишу Enter (например, `C:<Enter>`).
- Консоли часто также используют набор предопределенных префиксов пути для представления нескольких томов. Например, PlayStation 3 применяет префикс `/dev_bdvd/` для обращения к дисководу Blu-ray, а `/dev_hddx/` относится к одному или нескольким жестким дискам (где `x` — индекс устройства). В пакете разработки для PS3 префикс `/app_home/` ведет на определенный пользователем путь на любом компьютере, который задействован в разработке. В процессе разработки игра обычно забирает свои активы из `/app_home/`, а не с Blu-ray или жесткого диска.

Абсолютные и относительные пути

Все пути указываются относительно какой-то позиции в файловой системе. Когда путь указан относительно корневого каталога, мы называем его *абсолютным путем*, а когда относительно *другого* каталога в иерархии файловой системы — *относительным путем*.

В UNIX и Windows абсолютные пути начинаются с разделительного символа / или \, в то время как относительные пути не имеют начального разделителя. В Windows как абсолютные, так и относительные пути могут иметь необязательный определитель тома — если том не указан, предполагается, что путь ссылается на текущий рабочий том.

Следующие пути являются абсолютными.

- Windows:
  - `C:\Windows\System32`;
  - `D:\` (корневой каталог в томе `D:`);
  - `\` (корневой каталог в текущем рабочем томе);
  - `\game\assets\animation\walk.anim` (текущий рабочий том);
  - `\\joe-dell\Shared_Files\Images\foo.jpg` (сетевой путь);
- UNIX:
  - `/usr/local/bin/grep`;
  - `/game/src/audio/effects.cpp`;
  - `/` (корневой каталог).

Следующие пути являются относительными.

- Windows:
  - `System32` (относительно текущего рабочего каталога `\Windows` в текущем томе);
  - `X:animation\walk.anim` (относительно текущего рабочего каталога `\game\assets` в томе `X:`);
- UNIX:
  - `bin/grep` (относительно текущего рабочего каталога `/usr/local`);
  - `src/audio/effects.cpp` (относительно текущего рабочего каталога `/game`).

Пути поиска

Термин *«путь»* не следует путать с термином *«путь поиска»*. *Путь* — это строка, описывающая расположение файла или папки в иерархии файловой системы. *Путь поиска* — это строка, содержащая список путей, который используется при поиске файла и в котором каждый из путей отделен от прочих специальным символом, таким как двоеточие или точка с запятой. Например, когда вы запускаете любую программу из командной строки, операционная система находит исполняемый файл путем просмотра каждого каталога в пути поиска, который содержится в переменной среды оболочки.

Некоторые игровые движки задействуют пути поиска также для поиска файлов ресурсов. Например, механизм рендеринга OGRE задействует путь поиска ресурса, содержащийся в текстовом файле с именем `resources.cfg`. Файл предоставляет простой список каталогов и ZIP-архивов, которые следует просмотреть по порядку при попытке найти актив. Тем не менее поиск активов во время работы является трудоемким процессом. Обычно пути к активам всегда известны априори. Учитывая это, мы можем вообще избежать поиска активов, что предпочтительно.

API путей

Очевидно, что пути гораздо сложнее, чем простые строки. Работая с путями, программисту придется выполнять множество операций, таких как изоляция каталога, имени файла и расширения, канонизация пути, преобразования между абсолютными и относительными путями и т. д. Очень полезно иметь многофункциональный API для решения этих задач.

Microsoft Windows предоставляет API для этой цели. Он реализован как динамическая библиотека `shlwapi.dll` и доступен через заголовочный файл `shlwapi.h`. Полная документация этого API есть в сети разработчиков Microsoft (Microsoft Developer's Network, MSDN) по адресу msdn2.microsoft.com/en-us/library/bb773559(VS.85).aspx.

Конечно, API `shlwapi` доступен только на платформах Win32. Sony предоставляет аналогичные API для использования на PlayStation 3 и PlayStation 4. Но при написании кросс-платформенного игрового движка мы не можем напрямую за-

действовать API конкретных платформ. В любом случае игровому движку могут не понадобиться все функции, предоставляемые таким API, как `shlwapi`. Поэтому игровые движки часто реализуют упрощенный API обработки пути, который отвечает конкретным потребностям движка и работает на каждой операционной системе, на которую движок ориентирован. Такой API может быть реализован в виде тонкой обертки вокруг собственного API на каждой платформе или написан с нуля.

7.1.2. Базовый файловый ввод/вывод

Стандартная библиотека C предоставляет два API для открытия, чтения и записи содержимого файлов — один буферизованный, другой небуферизованный. Каждому API файлового ввода/вывода требуются блоки данных, называемые *буферами*, которые служат источником или местом назначения байтов, проходящих между программой и файлом на диске. Мы называем API файлового ввода/вывода *буферизованным*, если API управляет необходимыми нам входными и выходными буферами данных. При использовании небуферизованного API программист должен сам выделять буферы данных и управлять ими. Подпрограммы буферизованного файлового ввода-вывода стандартной библиотеки C иногда называют API *потокового ввода-вывода*, потому что они предоставляют абстракцию, которая делает файлы диска похожими на потоки байтов.

Функции стандартной библиотеки C для буферизованного и небуферизованного файлового ввода-вывода перечислены в табл. 7.1.

Таблица 7.1. Буферизованные и небуферизованные файловые операции в стандартной библиотеке C

Операция	Буферизованный API	Небуферизованный API
Открыть файл	fopen()	open()
Закрыть файл	fclose()	close()
Чтение из файла	fread()	read()
Запись в файл	fwrite()	write()
Поиск смещения	fseek()	seek()
Возвращение текущего смещения	ftell()	tell()
Чтение одной строки	fgets()	Недоступно
Записать одну строку	fputs()	Недоступно
Чтение форматированной строки	fscanf()	Недоступно
Записать форматированную строку	fprintf()	Недоступно
Запросить статус файла	fstat()	stat()

Функции ввода/вывода этой библиотеки хорошо документированы, поэтому мы не будем давать здесь подробное руководство. Для получения дополнительной информации обратитесь, пожалуйста, к msdn.microsoft.com/en-us/library/c565h7xx.aspx,

чтобы посмотреть реализацию компанией Microsoft буферизованного (потоковый ввод-вывод) API, и к msdn.microsoft.com/en-us/library/40bbyw78.aspx, чтобы увидеть реализацию небуферизованного (низкоуровневый ввод-вывод) API.

В UNIX и его вариантах небуферизованные маршруты ввода-вывода стандартной библиотеки C являются вызовами родной операционной системы. Однако в Microsoft Windows эти подпрограммы — просто обертки вокруг еще более низкоуровневого API. Функция Win32 `CreateFile()` создает или открывает файл для записи или чтения, `ReadFile()` и `WriteFile()` читает или записывает данные соответственно, а `CloseFile()` закрывает дескриптор открытого файла. Преимущество низкоуровневых системных вызовов перед функциями стандартной библиотеки C в том, что первые предоставляют полную информацию из файловой системы. Например, вы можете запрашивать и контролировать атрибуты безопасности файлов при использовании собственного API Windows, чего не можете сделать со стандартной библиотекой C.

Некоторые команды разработчиков считают полезным управление своими собственными буферами. Например, команда *Red Alert 3* в Electronic Arts заметила, что запись данных в файлы журналов приводит к значительному снижению производительности. Систему журналирования изменили так, чтобы она накапливала свои выходные данные в буфере памяти и записывала буфер на диск только после его заполнения. Затем переместили подпрограмму дампа буфера в отдельный поток, чтобы избежать задержки основного цикла игры.

Оборачивать или не оборачивать

Игровой движок может быть написан так, что он будет использовать либо функции ввода-вывода файлов стандартной библиотеки C, либо встроенный API операционной системы. Однако многие игровые движки *оборачивают* API файлового ввода/вывода в библиотеку пользовательских функций ввода/вывода. Оборачивание API ввода-вывода операционной системы имеет минимум три преимущества. Во-первых, программисты движка могут гарантировать одинаковое поведение на всех целевых платформах, даже если собственные библиотеки несовместимы или содержат ошибки на конкретной платформе. Во-вторых, API можно упростить до тех функций, которые действительно необходимы для движка, что сводит к минимуму затраты на обслуживание. В-третьих, может быть предоставлена расширенная функциональность. Например, пользовательская обертка API движка может работать с файлами на жестком диске, DVD-ROM или Blu-ray-диске на консоли, файлами в сети (например, удаленными файлами, которые управляются Xbox Live или PSN), а также файлами на картах памяти или других видах съемных носителей.

Синхронный файловый ввод/вывод

Обе библиотеки файлового ввода-вывода стандартной библиотеки C являются *синхронными*, что означает: программа, выполняющая запрос ввода-вывода, прежде чем продолжить, должна ждать, пока данные не будут полностью перенесены на медиаустройство или с него. Следующий фрагмент кода демонстрирует, как

содержимое целого файла может быть считано в буфер в памяти с помощью функции синхронного ввода-вывода `fread()`. Обратите внимание на то, что функция `syncReadFile()` не возвращает значение, пока все данные не будут считаны в предоставленный буфер:

```
bool syncReadFile(const char* filePath,
                  U8* buffer,
                  size_t bufferSize,
                  size_t& rBytesRead)
{
    FILE* handle = fopen(filePath, "rb");
    if (handle)
    {
        // Блокируем тут, пока все данные не будут прочитаны.
        size_t bytesRead = fread(buffer, 1,
                                 bufferSize, handle);

        int err = ferror(handle); // получаем ошибку, если...

        fclose(handle);

        if (0 == err)
        {
            rBytesRead = bytesRead;
            return true;
        }
    }
    rBytesRead = 0;
    return false;
}

void main(int argc, const char* argv[])
{
    U8 testBuffer[512];
    size_t bytesRead = 0;

    if (syncReadFile("C:\\testfile.bin",
                     testBuffer, sizeof(testBuffer),
                     bytesRead))
    {
        printf("success: read %u bytes\n", bytesRead);
        // здесь можно использовать содержимое буфера...
    }
}
```

7.1.3. Асинхронный файловый ввод/вывод

Потоковая передача данных обозначает процесс загрузки данных в фоновом режиме, в то время как основная программа продолжает выполняться. Многие игры позволяют игроку получить опыт «бесшовной» игры без загрузочных экранов благодаря потоковой загрузке данных для предстоящих уровней с DVD-ROM,

Blu-ray или жесткого диска прямо во время нее. Аудио- и текстурные данные являются, пожалуй, наиболее распространенным типом потоковых данных, но на самом деле передаваться потоком на фоне могут любые данные, в том числе геометрия, макеты уровней и анимационные клипы.

Чтобы поддерживать потоковую передачу данных, мы должны использовать *асинхронную* библиотеку файлового ввода-вывода, то есть такую, которая позволяет программе продолжать работу, пока выполняются ее запросы ввода-вывода. Некоторые операционные системы предоставляют асинхронную библиотеку файлового ввода-вывода «из коробки». Например, общеязыковая среда выполнения Windows (Common Language Runtime, CLR, — виртуальная машина, на которой реализованы языки Visual BASIC, C#, управляемый C++ и J#) обеспечивает такие функции, как `System.IO.BeginRead()` и `System.IO.BeginWrite()`. Асинхронный API, известный как `fios`, доступен для PlayStation 3 и PlayStation 4. Если асинхронная библиотека файлового ввода-вывода недоступна для вашей целевой платформы, можно написать ее самостоятельно. И даже если вам не придется создавать ее с нуля, неплохо будет обернуть системный API для портативности.

Следующий фрагмент кода демонстрирует, как содержимое целого файла может быть считано в буфер в памяти с помощью операции асинхронного чтения. Обратите внимание на то, что функция `asyncReadFile()` возвращает значение сразу же — данных нет в буфере, пока функция обратного вызова `asyncRead-Complete()` не была вызвана библиотекой ввода/вывода:

```
AsyncRequestHandle g_hRequest; // дескриптор асинхронного запроса ввода/вывода
U8 g_asyncBuffer[512];         // буфер ввода

static void asyncReadComplete(AsyncRequestHandle hRequest);

void main(int argc, const char* argv[])
{
    // Примечание: этот вызов asyncOpen() сам по себе может быть
    // асинхронным, но мы это проигнорируем
    // и просто предположим, что это блокирующая функция.

    AsyncFileHandle hFile = asyncOpen(
        "C:\\testfile.bin");

    if (hFile)
    {
        // Эта функция запрашивает чтение ввода/вывода, а затем
        // немедленно делает возврат (неблокирующая).
        g_hRequest = asyncReadFile(
            hFile,                    // дескриптор файла
            g_asyncBuffer,            // буфер ввода
            sizeof(g_asyncBuffer),    // размер буфера
            asyncReadComplete);       // функция обратного вызова
    }
```

```
    // Теперь мы можем двигаться дальше...
    // (Этот цикл имитирует выполнение какой-то работы, пока мы ожидаем
    // завершения чтения ввода-вывода.)

    for (;;)
    {
        OutputDebugString("zzz...\n");
        Sleep(50);
    }
}

// Эта функция будет вызываться после прочтения данных.
static void asyncReadComplete(AsyncRequestHandle hRequest)
{
    if (hRequest == g_hRequest
    && asyncWasSuccessful(hRequest))
    {
        // Данные теперь присутствуют в g_asyncBuffer[]
        // и могут быть использованы. Запрос количества
        // прочитанных байтов:
        size_t bytes = asyncGetBytesReadOrWritten(
                            hRequest);

        char msg[256];
        snprintf(msg, sizeof(msg),
            "async success, read %u bytes\n",
            bytes);
        OutputDebugString(msg);
    }
}
```

Большинство асинхронных библиотек ввода/вывода позволяют основной программе в течение еще некоторого времени после запроса дожидаться завершения операции ввода/вывода. Это бывает полезно в ситуациях, когда может быть выполнен лишь ограниченный объем работы, прежде чем потребуются результаты запроса ввода-вывода. Это показано в следующем фрагменте кода:

```
U8 g_asyncBuffer[512];     // буфер ввода

void main(int argc, const char* argv[])
{
    AsyncRequestHandle hRequest = ASYNC_INVALID_HANDLE;
    AsyncFileHandle hFile = asyncOpen(
                                "C:\\testfile.bin");

    if (hFile)
    {
        // Эта функция запрашивает чтение ввода/вывода, а затем
        // немедленно делает возврат (неблокирующая).
        hRequest = asyncReadFile(
            hFile,                      // дескриптор файла
```

```
            g_asyncBuffer,              // буфер ввода
            sizeof(g_asyncBuffer),      // размер буфера
            nullptr);                   // нет обратного вызова
    }

    // Теперь выполняем ограниченный объем работы...
    for (int i = 0; i < 10; i++)
    {
        OutputDebugString("zzz...\n");
        Sleep(50);
    }

    // Мы не можем ничего делать дальше, пока у нас
    // нет этих данных, так что ждите здесь.
    asyncWait(hRequest);

    if (asyncWasSuccessful(hRequest))
    {
        // Данные теперь присутствуют в g_asyncBuffer[]
        // и могут быть использованы. Запрос количества
        // прочитанных байтов:
        size_t bytes = asyncGetBytesReadOrWritten(
                                            hRequest);
        char msg[256];
        snprintf(msg, sizeof(msg),
            "async success, read %u bytes\n",
            bytes);
        OutputDebugString(msg);
    }
}
```

Некоторые библиотеки асинхронного ввода-вывода позволяют программисту запрашивать оценку того, сколько времени займет выполнение конкретной асинхронной операции. Некоторые API также позволяют устанавливать *сроки истечения* запроса (что фактически задает его приоритет относительно других ожидающих запросов) и определять, что следует предпринять, когда срок действия запроса истекает (например, отменить запрос, уведомить программу и продолжить попытки и т. д.).

Приоритеты

Важно помнить, что файловый ввод-вывод — это система реального времени, которая так же ограничивается сроками, как и остальная часть игры. Поэтому асинхронные операции ввода-вывода часто имеют различные приоритеты. Например, если мы загружаем аудио с жесткого диска или Blu-ray и сразу же воспроизводим его, загрузка следующего буфера с аудиоданными имеет более высокий приоритет, чем, скажем, загрузка текстуры или фрагмента игрового уровня. Асинхронные системы ввода-вывода должны быть способны приостанавливать запросы с более низким приоритетом, чтобы запросы ввода-вывода с более высоким приоритетом могли быть выполнены в установленные сроки.

Как работает асинхронный файловый ввод-вывод

Асинхронный файловый ввод-вывод работает путем обработки запросов ввода-вывода в отдельном потоке. Основной поток вызывает функции, которые просто помещают запросы в очередь и сразу же делают возврат. Тем временем поток ввода-вывода получает запросы из очереди и последовательно обрабатывает их, используя блокирующие процедуры ввода-вывода, такие как `read()` или `fread()`. Когда запрос завершен, выполняется обратный вызов из основного потока, тем самым уведомляя его о том, что операция выполнена. Если основной поток решает дождаться завершения запроса ввода-вывода, это достигается с помощью *семафора*. (Каждый запрос имеет связанный с ним семафор, и основной поток может перевести себя в режим ожидания, пока поток ввода-вывода не сообщит этому семафору о завершении запроса. Подробнее о семафорах см. в подразделе 4.6.4.)

Практически *любую* синхронную операцию, которую вы можете себе представить, можно преобразовать в асинхронную, переместив код в отдельный поток или запустив его на физически отдельном процессоре, например на одном из ядер процессора на PlayStation 4. Более подробное описание приведено в разделе 8.6.

7.2. Менеджер ресурсов

Любая игра построена из самых разных *ресурсов* (иногда их называют *активами* или *медиа*). В качестве примеров можно привести меши, материалы, текстуры, шейдерные программы, анимацию, аудиоклипы, макеты уровней, примитивы столкновений, физические параметры и т. д. Список можно продолжить. Должна быть возможность управлять ресурсами игры с точки зрения как автономных инструментов, используемых для их создания, так и их загрузки, выгрузки и манипулирования ими в среде выполнения. Поэтому у каждого игрового движка есть *менеджер ресурсов*.

Все менеджеры ресурсов состоят из двух разных, но объединенных компонентов. Один компонент управляет цепочкой автономных инструментов, применяемых для создания активов и преобразования их в готовую для использования движком форму. Другой управляет ресурсами в среде выполнения, обеспечивая их загрузку в память до того, как они понадобятся игре, а также выгрузку из памяти, когда они больше не нужны.

В некоторых движках менеджер ресурсов представляет собой четко спроектированную, унифицированную, централизованную подсистему, которая управляет всеми типами ресурсов, используемых в игре. В других движках менеджер ресурсов не существует как отдельная подсистема, а скорее представляет собой разнородную совокупность подсистем, которые, возможно, были написаны разными людьми в разное время в течение долгой, а иногда и яркой истории движка. Но вне зависимости от реализации менеджер ресурсов неизменно берет на себя определенные обязанности и решает четко определенный набор задач. В этом разделе мы рассмотрим функциональные возможности и некоторые детали реализации типичного менеджера ресурсов игрового движка.

7.2.1. Автономное управление ресурсами и цепочка инструментов

Контроль версий для активов

В небольшом игровом проекте активами игры можно управлять, храня свободные файлы на общем сетевом диске со специальной структурой каталогов. Такой вариант не подходит для современной коммерческой 3D-игры, состоящей из огромного количества разнообразных активов. Для такого проекта команде требуется более формализованный способ отслеживания собственных активов и управления ими.

Некоторые команды разработчиков игр используют систему контроля версий исходного кода для управления своими ресурсами. Графические исходные файлы (сцены Maya, файлы Photoshop PSD, файлы Illustrator и т. д.) записываются художниками в Perforce или аналогичном пакете. Такой подход работает довольно хорошо, хотя некоторые команды разработчиков создают собственные инструменты управления активами, чтобы облегчить обучение своих художников работе с ними. Такие инструменты могут быть простыми обертками вокруг коммерческой системы контроля версий или же быть полностью индивидуальными.

Размер данных. Одной из самых больших проблем в контроле версий графических активов является огромное количество данных. В то время как файлы C++ и исходные коды скриптов невелики по сравнению с их влиянием на проект, графические файлы обычно намного больше. Поскольку многие системы контроля версий работают, копируя файлы из центрального репозитория на локальный компьютер пользователя, один только размер файлов активов может сделать эти пакеты почти бесполезными.

Я видел множество разных решений этой проблемы, используемых в разных студиях. Некоторые студии обращаются к коммерческим системам контроля версий, таким как Alienbrain, которые были специально разработаны для обработки очень больших объемов данных. Некоторые команды просто работают с тем, что есть, и позволяют своему инструменту контроля версий копировать активы локально. Такой подход работает, если у вас хватает свободного места на дисках и ширины канала сети, но он может быть очень неэффективным и замедляющим работу команды. Некоторые команды создают сложные системы поверх своего инструмента контроля версий, чтобы гарантировать, что конкретный конечный пользователь получит только локальные копии файлов, которые ему действительно нужны. В этой модели пользователь либо не имеет доступа к остальной части хранилища, либо может получить доступ на общем сетевом диске, когда это необходимо.

В Naughty Dog мы задействуем запатентованный инструмент, который использует символические ссылки UNIX, чтобы практически исключить копирование данных, в то же время предоставляя каждому пользователю возможность полностью просматривать хранилища активов. Пока файл не извлечен для редактирования, это просто символическая ссылка на главный файл на общем сетевом диске. Такая ссылка занимает очень мало места на локальном диске, потому что это не более чем запись каталога. Когда пользователь извлекает файл для редактирования, симво-

лическая ссылка удаляется и ее заменяет локальная копия файла. Когда пользователь завершает редактирование и регистрирует файл, локальная копия становится новой главной копией, ее история изменений обновляется в основной базе данных, а локальный файл снова превращается в символическую ссылку. Такая система работает очень хорошо, но для этого требуется, чтобы команда создала собственную систему контроля версий с нуля. Я не знаю ни об одном коммерческом инструменте, который работает таким образом. Кроме того, символические ссылки являются функцией UNIX — такой инструмент, вероятно, может быть создан с помощью соединений Windows (эквивалент символической ссылки для Windows), но я пока не видел, чтобы кто-нибудь пробовал это сделать.

База данных ресурсов

Как мы подробно рассмотрим в следующем разделе, большинство активов не используются игровым движком в их оригинальном формате. Им нужно пройти через конвейер подготовки активов, задачей которого является их преобразование в двоичный формат, необходимый движку. Для каждого ресурса, проходящего через конвейер подготовки активов, существует некоторое количество *метаданных*, которые описывают, *как* этот ресурс должен обрабатываться. При сжатии растрового изображения текстуры нам нужно знать, какой *тип* сжатия лучше всего подходит для данного изображения, при экспорте анимации — какой диапазон кадров в Maya следует экспортировать. При экспорте сеток персонажей из сцены Maya, содержащей несколько персонажей, нам должно быть известно, какой меш соответствует какому персонажу в игре.

Для управления всеми этими метаданными нужна какая-то база данных. Если мы делаем очень маленькую игру, эту базу данных разработчики могут просто держать в уме. Я как будто слышу: «Помните, для анимации игрока должен быть установлен флаг flip X, но у других персонажей его *не* должно быть... или... черт... там все наоборот?»

Очевидно, что при создании любой игры достойного размера мы просто не можем так полагаться на память разработчиков. Во-первых, один только объем активов довольно быстро становится просто огромным. Обработка отдельных ресурсных файлов вручную также слишком трудоемка, чтобы ее можно было использовать при полномасштабном производстве коммерческих игр. Поэтому каждая профессиональная игровая команда имеет своего рода полуавтоматический конвейер ресурсов, и данные, которые управляют конвейером, хранятся в некоторой *базе данных ресурсов*.

База данных ресурсов выглядит по-разному в разных игровых движках. В одном движке метаданные, описывающие, как должен создаваться ресурс, могут быть встроены в сами исходные активы (например, они могут храниться как так называемые скрытые данные в файле Maya). В другом движке каждый файл исходного ресурса может сопровождаться небольшим текстовым файлом, который описывает, как он должен обрабатываться. В то же время часть движков кодируют свои метаданные создания ресурсов в виде набора файлов XML, возможно обернутых

в какой-то пользовательский графический интерфейс. Некоторые движки используют настоящую реляционную базу данных, такую как Microsoft Access, MySQL или, возможно, даже такую тяжеловесную, как Oracle.

Независимо от формы, база данных ресурсов должна обеспечивать следующие основные функции:

- способность иметь дело с несколькими *типами* ресурсов, в идеале (конечно, не обязательно) будучи последовательной;
- возможность создавать новые ресурсы;
- возможность удалять ресурсы;
- возможность проверять и изменять существующие ресурсы;
- возможность перемещать исходный файл (файлы) ресурса с одного места на диске на другое. (Это очень полезно, потому что художникам и разработчикам игр часто нужно перегруппировать активы, чтобы отразить меняющиеся цели проекта, переосмысление игровых дизайнов, добавление и вырезание игровых функций и т. д.);
- способность ресурса создавать перекрестные ссылки на другие ресурсы (например, материал, используемый мешем, или набор анимаций, необходимый для 17-го уровня). Эти перекрестные ссылки обычно управляют процессами создания ресурса и его загрузки в среде выполнения;
- способность поддерживать *ссылочную целостность* всех перекрестных ссылок в базе данных и делать это с учетом всех распространенных операций, таких как удаление или перемещение ресурсов;
- возможность вести историю изменений, дополненную журналом с записями о том, кто внес каждое изменение и почему.

Также очень полезно, когда база данных ресурсов поддерживает поиск или запросы различных видов. Например, разработчик может захотеть узнать, на каких уровнях используется определенная анимация или на какие текстуры ссылается набор материалов. Или просто пытаться найти ресурс, название которого никак не может вспомнить.

Из приведенного ранее списка должно быть очевидно, что создание крепкой и надежной базы данных ресурсов — задача не из легких. При правильных разработке и реализации база данных ресурсов может в буквальном смысле определить разницу между командой, которая выпускает хитовую игру, и командой, которая топталась на месте в течение 18 месяцев, пока руководство не вынудило ее оставить проект (или еще похуже). Я знаю, что это так, потому что испытал на себе оба варианта.

Успешные проекты базы данных ресурсов

Любая команда разработчиков будет предъявлять разные требования и принимать разные решения в ходе создания базы данных ресурсов. Тем не менее вот несколько проектов, которые хорошо зарекомендовали себя.

Unreal Engine 4. Базой данных ресурсов Unreal управляет их универсальный инструмент UnrealEd. Он отвечает буквально за все: от управления метаданными ресурса до создания актива, макета уровня и многого другого. У UnrealEd есть определенные недостатки, но самое большое его преимущество в том, что он является частью игрового движка. Это позволяет создавать активы, а затем сразу же просматривать их во всей красе — именно так, как они будут отображаться в игре. Игру можно даже запустить в UnrealEd, чтобы визуализировать активы в их естественном окружении и посмотреть, как они работают в игре (и работают ли вообще).

Еще одно большое преимущество UnrealEd — то, что я называю *универсальным магазином*. Универсальный браузер UnrealEd (рис. 7.1) позволяет разработчику получить доступ буквально ко всем ресурсам, которые использует движок. Наличие единого, унифицированного и логично построенного интерфейса для создания всех типов ресурсов и управления ими — большое достижение. Это особенно верно, если учесть, что данные о ресурсах в большинстве других игровых движков разбросаны по бесчисленным несовместимым и часто непонятным инструментам. Возможность легко *найти* любой ресурс в UnrealEd — уже большой плюс.

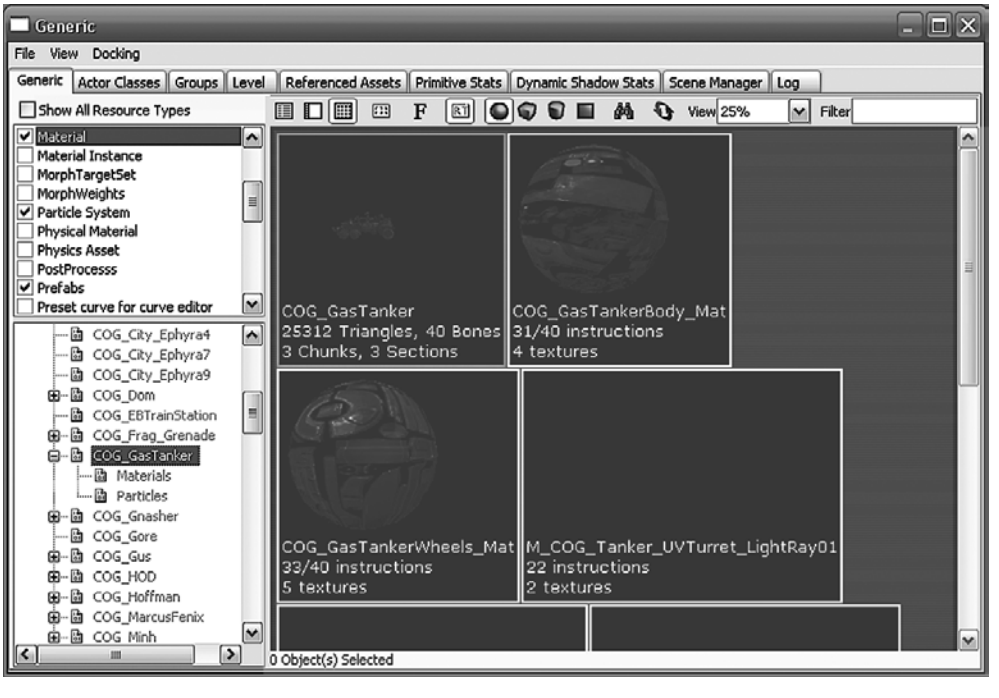


Рис. 7.1. Универсальный браузер UnrealEd

Unreal может быть меньше подвержен риску появления различных ошибок, чем многие другие движки, потому что активы должны быть напрямую импортированы в базу данных ресурсов Unreal. Это позволяет проверять активы на валидность в самом начале производственного процесса. В большинстве игровых движков любые

старые данные могут быть вброшены в базу данных ресурсов, а валидны они или нет, вы узнаете, лишь когда база будет построена, а иногда только когда она будет загружена в игру уже во время работы. Но с Unreal активы могут быть проверены сразу после импорта в UnrealEd. Это означает, что человек, который создал актив, сразу же получает отзыв о том, правильно ли его актив настроен.

Конечно, у подхода Unreal есть и серьезные недостатки. Например, все данные ресурсов хранятся в нескольких больших файлах пакета. Эти файлы — бинарные, поэтому их нелегко объединить с помощью пакета контроля версий, такого как CVS, Subversion или Perforce. Это создает серьезные проблемы в том случае, когда более одного пользователя хотят изменить ресурсы, которые находятся в одном пакете. Даже если пользователи пытаются изменить *различные* ресурсы, одновременно только один пользователь может сохранить изменения в пакете, поэтому другому приходится ждать. Такую проблему можно смягчить, разделив ресурсы на относительно небольшие детализованные пакеты, но полностью устранить ее нельзя.

Ссылочная целостность довольно хороша в UnrealEd, но и здесь есть некоторые проблемы. Когда ресурс переименовывается или перемещается, все ссылки на него автоматически сохраняются с использованием фиктивного объекта, который переназначает старый ресурс на новое имя/местоположение. Плохо в фиктивных переназначающих объектах то, что они остаются, накапливаются и иногда вызывают проблемы, особенно когда ресурс удаляется. В общем, ссылочная целостность Unreal довольно хороша, но не идеальна.

Несмотря на свои проблемы, UnrealEd на сегодняшний день является наиболее удобным, хорошо интегрированным и оптимизированным инструментарием для создания активов, базой данных ресурсов и конвейером подготовки активов из всех, с которыми я когда-либо работал.

Движок Naughty Dog. Naughty Dog хранит свои метаданные ресурсов для игры *Uncharted: Drake's Fortune* в базе данных MySQL. Для управления содержимым базы данных был написан пользовательский графический интерфейс. Он позволил художникам, дизайнерам игр и программистам создавать новые ресурсы, удалять существующие, а также проверять и изменять их. Этот графический интерфейс был важнейшим компонентом системы, поскольку освободил пользователей от необходимости изучать тонкости взаимодействия с реляционной базой данных с помощью SQL.

Исходная база данных MySQL, используемая в *Uncharted*, не умела давать полезную историю изменений базы данных, и у нее не было хорошего способа откатить «плохие» изменения. Также она не поддерживала редактирование одного и того же ресурса несколькими пользователями, и ее было сложно администрировать. С тех пор Naughty Dog отказалась от MySQL в пользу базы данных активов на основе XML, управляемой в Perforce.

Builder — графический интерфейс базы данных ресурсов Naughty Dog — изображен на рис. 7.2. Окно разбито на два основных раздела: слева — древовидное представление данных, показывающее все ресурсы в игре, справа — окно свойств, позволяющее просматривать и редактировать ресурсы, выбранные в дереве. Дерево

ресурсов содержит папки для организационных целей, так что художники и дизайнеры игр могут организовать свои ресурсы так, как считают нужным. В любой папке можно создавать различные типы ресурсов (включая акторы и уровни), а также подресурсы, из которых они состоят (в основном меши, скелеты и анимации), и управлять ими. Анимации также могут быть сгруппированы в псевдопапки, известные как пакеты. Это дает возможность создавать большие группы анимаций, а затем управлять ими как отдельной единицей, что позволяет не тратить время на перемещение отдельных анимаций в дереве.

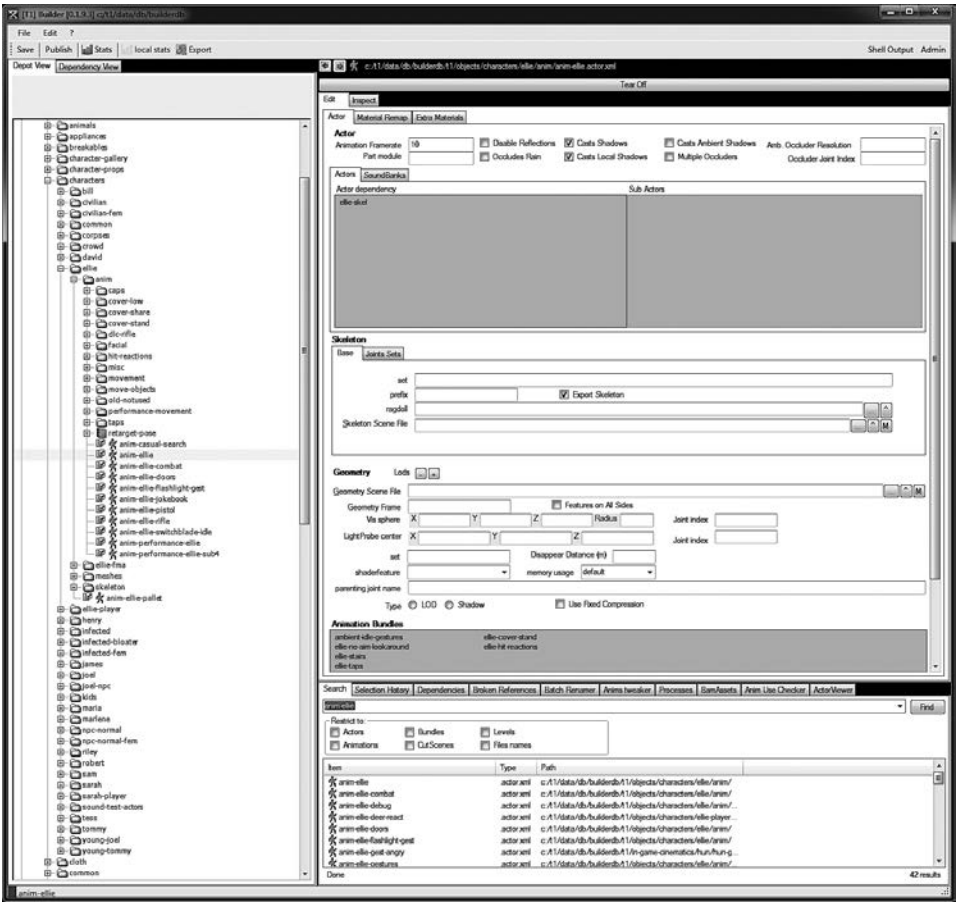


Рис. 7.2. Клиентская сторона GUI для автономной базы данных ресурсов Builder компании Naughty Dog

Конвейер подготовки активов, используемый в сериях *Uncharted* и *The Last of Us*, состоит из набора конвертеров ресурсов, компиляторов и компоновщиков, запускаемых из командной строки. Движок способен работать с самыми разными типами объектов данных, но они упакованы в один из двух видов файлов

ресурсов: акторы и уровни. Актор может содержать скелеты, меши, материалы, текстуры и/или анимацию. Уровень содержит статические фоновые меши, материалы и текстуры, а также информацию о макете уровня. Чтобы создать актор, нужно просто ввести `ba` *название_актора* в командной строке, чтобы создать уровень — ввести `bl` *название_уровня*. Эти инструменты командной строки запрашивают базу данных, чтобы точно определить, *как* построить рассматриваемый актор или уровень. Сюда включается информация о том, как экспортировать активы из инструментов DCC, таких как Maya и Photoshop, как обрабатывать данные и упаковывать их в двоичные `.pak`-файлы, которые могут быть загружены игровым движком. Это намного проще, чем во многих других движках, где художники должны *экспортировать ресурсы вручную* — это трудоемкая, утомительная и чреватая ошибками задача.

Преимущества дизайна конвейера ресурсов, используемого Naughty Dog.

- *Детализованные ресурсы.* Ресурсами можно манипулировать с точки зрения логических объектов в игре — сеток, материалов, скелетов и анимации. Эти типы ресурсов довольно хорошо детализованы, поэтому у разработчиков почти никогда не бывает ситуаций, когда два пользователя пытаются редактировать один и тот же ресурс одновременно.
- *Необходимые функции (и не более того).* Инструмент Builder предоставляет богатый набор функций, которые удовлетворяют потребности команды, но Naughty Dog не тратили впустую ресурсы, создавая функции, которые им не нужны.
- *Очевидное соответствие исходным файлам.* Пользователь может очень быстро определить, какие исходные активы (собственные файлы DCC, такие как файлы Maya `.ma` или файлы Photoshop `.psd`) составляют определенный ресурс.
- *Легко изменяемый способ экспорта и обработки данных DCC.* Просто выберите необходимый ресурс и измените настройки его обработки в графическом интерфейсе базы данных ресурсов.
- *Легко создаваемые активы.* Просто введите `ba` или `bl`, а затем имя ресурса в командной строке. Система зависимостей позаботится обо всем остальном.

Конечно, цепочка инструментов Naughty Dog имеет и некоторые недостатки.

- *Отсутствие инструментов визуализации.* Единственный способ предварительно просмотреть актив — загрузить его в игру или средство просмотра моделей/анимации (что на самом деле является просто специальным режимом самой игры).
- *Инструменты не полностью интегрированы.* Naughty Dog использует один инструмент для разметки уровней, другой — для управления большинством ресурсов в базе данных и третий — для настройки материалов и шейдеров (это не является частью интерфейса базы данных ресурсов). Сборка активов производится в командной строке. Было бы немного удобнее, если бы все эти функции были объединены в одном инструменте. Тем не менее Naughty Dog не планирует ничего делать, потому что выгода, вероятно, не оправдает связанные с этим расходы.

Система управления ресурсами OGRE. OGRE — это движок для рендеринга, а не полноценный игровой движок. Тем не менее он может похвастаться довольно полным и очень хорошо разработанным менеджером ресурсов среды выполнения. Простой и понятный интерфейс используется для загрузки практически любого вида ресурса. И система была разработана с учетом возможности расширения. Любой программист может довольно легко внедрить менеджер ресурсов для совершенно нового вида активов и интегрировать его в структуру ресурсов OGRE.

Одним из недостатков менеджера ресурсов OGRE является то, что это решение, работающее только в среде выполнения. В OGRE отсутствует какая-либо база автономных ресурсов. OGRE предоставляет некоторые конвертеры, способные конвертировать файл Maya в меш, который может применяться OGRE (вместе с материалами, шейдерами, скелетом и дополнительной анимацией). Однако конвертеры должны запускаться вручную из самой Maya. Хуже того, все метаданные, описывающие, как конкретный файл Maya должен быть экспортирован и обработан, приходится вводить пользователю, выполняющему экспорт.

Таким образом, менеджер ресурсов среды выполнения OGRE функционален и хорошо продуман. Но если говорить об инструментах, OGRE извлекла бы большую пользу из не менее мощной и современной базы данных ресурсов и конвейера формирования активов.

Microsoft XNA. XNA — это набор инструментов для разработки игр от Microsoft, предназначенный для ПК и платформ Xbox 360. Несмотря на то что Microsoft прекратила поддержку проекта в 2014 году, он все еще является хорошим ресурсом для изучения игровых движков. Система управления ресурсами XNA уникальна тем, что использует системы управления и создания проектов среды IDE Visual Studio для управления и создания активов в игре. Инструмент для разработки игр XNA, Game Studio Express, — это просто плагин для Visual Studio Express.

Конвейер подготовки активов

В разделе 1.7 мы узнали, что данные о ресурсах обычно формируются с помощью таких современных инструментов создания цифрового контента (DCC), как Maya, ZBrush, Photoshop или Houdini. Однако форматы данных, используемые этими инструментами, обычно не подходят для игрового движка. Поэтому бóльшая часть данных ресурсов передается через *конвейер подготовки активов* (asset conditioning pipeline, ACP) до применения игровым движком. ACP иногда называют *конвейером подготовки ресурсов* (resource conditioning pipeline, RCP) или просто *цепочкой инструментов*.

Каждый конвейер ресурсов начинается с коллекции *исходных активов* в собственных форматах DCC (например, файлы Maya `.ma` или `.mb`, файлы Photoshop `.psd` и т. д.). Эти активы обычно проходят три этапа обработки на пути к игровому движку.

1. *Конвертеры*. Нам нужен какой-то способ преобразования данных из родного формата DCC в формат, которым мы можем оперировать. Обычно это достигается написанием специального плагина для данного DCC. Работа плагина

заключается в экспорте данных в отдельный промежуточный формат файла, который можно передать на более поздние стадии конвейера. Большинство приложений DCC имеют довольно удобный механизм для этой цели. У Maya их целых три: C++ SDK, сценарный язык под названием MEL, а также совсем недавно появившийся интерфейс Python.

В случаях, когда приложение DCC не предоставляет хуки кастомизации, мы всегда можем сохранить данные в одном из собственных форматов инструмента DCC. Если повезет, одним из них будет открытый формат, интуитивно понятный текстовый формат или какой-то другой, который мы сможем исследовать. Предположим, что это так и теперь можно передать файл непосредственно на следующую стадию конвейера.

2. *Компиляторы ресурсов*. Нам часто нужно различными способами «помассировать» необработанные данные, экспортируемые из приложения DCC, чтобы подготовить их для использования игрой. Например, может понадобиться перестроить треугольники меша в полосы, или сжать растровое изображение текстуры, или вычислить длины дуг сегментов сплайна Кэтмулла — Рома. Не все типы ресурсов следует компилировать — некоторые могут быть готовы к применению в игре сразу после экспорта.
3. *Компоновщики ресурсов*. Иногда необходимо несколько файлов ресурсов объединить в один полезный пакет перед загрузкой игровым движком. Это имитирует процесс компоновки объектных файлов скомпилированной программы C++ в исполняемый файл, поэтому данный процесс иногда называют *компоновкой ресурсов*. Например, при создании сложного составного ресурса, такого как 3D-модель, нам может потребоваться объединить данные из нескольких экспортированных файлов сеток, нескольких файлов материалов, файла скелета и нескольких файлов анимации в один ресурс. Не все типы ресурсов нужно компоновать — некоторые активы готовы к использованию в игре после экспорта или компиляции.

Зависимости ресурсов и правила сборки. Подобно компиляции исходных файлов в проекте на C или C++ и затем связыванию их в исполняемый файл, конвейер подготовки активов обрабатывает исходные активы (в форме файлов геометрии и анимации Maya, PSD-файлов Photoshop, необработанных аудиоклипов, текстовых файлов и т. д.), преобразует их в готовую для игры форму и затем связывает в единое целое для использования движком. Как и исходные файлы компьютерной программы, игровые активы часто имеют взаимозависимости. (Например, меш ссылается на один или несколько материалов, которые, в свою очередь, ссылаются на различные текстуры.) Эти взаимозависимости обычно влияют на порядок, в котором активы должны обрабатываться конвейером. (Например, может понадобиться построить скелет персонажа, прежде чем мы сможем обработать любую из его анимаций.) Кроме того, зависимости между активами говорят нам, какие активы необходимо перестроить при изменении конкретного исходного актива.

Зависимости сборки касаются изменений не только в самих активах, но и в форматах данных. Например, если формат файлов, используемых для хранения треугольных сеток, меняется, то все меши во всей игре, возможно, придется пере-

экспортировать и/или перестроить. Некоторые игровые движки задействуют форматы данных, устойчивые к изменениям версий. Например, актив может содержать номер версии, а игровой движок — код, который знает, как загружать устаревшие активы и работать с ними. Недостатком такого подхода является то, что файлы активов и код движка могут стать довольно громоздкими. Если формат данных изменяется достаточно редко, возможно, лучше будет после таких изменений просто взять и повторно обработать все файлы.

Каждый конвейер подготовки активов требует набора правил, которые описывают взаимозависимости между ними, и какого-то инструмента сборки, способного использовать эту информацию, чтобы гарантировать, что надлежащие активы строятся в правильном порядке при изменении исходного ресурса. У некоторых команд разработчиков есть собственная система сборки. Другие применяют уже существующие инструменты, такие как make. Какое бы решение ни было выбрано, разработчикам следует с особой тщательностью относиться к своей системе зависимостей сборок. В противном случае изменения в источниках и активах могут не позволить восстановить соответствующие активы. Результатом могут быть несовместимые игровые активы, которые способны вызвать визуальные аномалии или даже сбои движка. Мне приходилось видеть, как тратятся бесчисленные часы на отслеживание проблем, которых можно было избежать, если бы были правильно определены взаимозависимости активов и внедрена система сборки для их надежного использования.

7.2.2. Управление ресурсами среды выполнения

Теперь посмотрим, как активы загружаются в нашей базе данных ресурсов и выгружаются в движке в среде выполнения, а также как происходит управление ими.

Обязанности менеджера ресурсов среды выполнения

Менеджер ресурсов среды выполнения игрового движка имеет широкий круг обязанностей, связанных с его основной задачей — загрузкой ресурсов в память.

- Менеджер гарантирует, что только *одна копия* каждого уникального ресурса существует в памяти в любой момент времени.
- Управляет *временем жизни* ресурсов.
- *Загружает* необходимые ресурсы и выгружает те, которые больше не нужны.
- Управляет загрузкой *составных ресурсов*. Составной ресурс — это ресурс, состоящий из других ресурсов. Например, 3D-модель — это составной ресурс, который состоит из меша, одного или нескольких материалов, одной или нескольких текстур и, возможно, скелета и нескольких скелетных анимаций.
- Поддерживает *ссылочную целостность*. К ней относятся *внутренняя* ссылочная целостность (перекрестные ссылки в пределах одного ресурса) и *внешняя* ссылочная целостность (перекрестные ссылки между ресурсами). Например, модель ссылается на свой меш и скелет; меш ссылается на свои материалы, которые, в свою очередь, ссылаются на ресурсы текстуры; анимации ссылаются

на скелет, который в конечном итоге связывает их с одной или несколькими моделями. При загрузке составного ресурса менеджер ресурсов должен убедиться, что загружены все необходимые подресурсы, а также корректно исправить все перекрестные ссылки.

- Контролирует *использование памяти* загруженными ресурсами и гарантирует, что ресурсы хранятся в соответствующих местах в памяти.
- Позволяет выполнять *настраиваемую обработку* ресурса любого типа после его загрузки. Этот процесс иногда называют *входом в систему* или *инициализацией загрузки ресурса*.
- Обычно (но не всегда) предоставляет единый *унифицированный интерфейс*, через который можно управлять различными типами ресурсов. В идеале менеджер ресурсов также легко расширяем, чтобы иметь возможность обрабатывать новые типы ресурсов, когда они нужны команде разработчиков игр.
- Управляет *потоковой передачей* (то есть асинхронной загрузкой ресурсов), если движок поддерживает эту функцию.

Файл ресурсов и организация каталогов

В некоторых игровых движках (обычно движках ПК) управление каждым отдельным ресурсом протекает в отдельном свободном файле на диске. Эти файлы обычно содержатся в дереве каталогов, внутренняя организация которых должна быть удобна в первую очередь людям, создающим активы, — движок обычно не заботится о том, где находятся файлы ресурсов в дереве ресурсов. Вот типичное дерево каталогов ресурсов для гипотетической игры под названием *Space Evaders*:

```
SpaceEvaders   Корневой каталог для всей игры.
  Resources    Корневой каталог для всех ресурсов.
    NPC        Модели неигровых персонажей и анимации.
      Pirate   Модели и анимации для пиратов.
      Marine   Модели и анимации для морских пехотинцев.
      …

    Player     Модели игровых персонажей и анимации.

    Weapons    Модели и анимации для оружия.
      Pistol   Модели и анимации для пистолета.
      Riffle   Модели и анимации для винтовки.
      BFG      Модели и анимации для большой пушки.
      …

    Levels     Геометрия фона и макеты уровней.
      Level 1  Ресурсы первого уровня.
      Level 2  Ресурсы второго уровня.
      …

    Objects    Разные 3D-объекты.
      Crate    Вездесущий хрупкий ящик.
      Barrel   Вездесущая взрывающаяся бочка.
```

Другие движки объединяют несколько ресурсов в один файл, такой как ZIP-архив или другой составной файл (возможно, собственного формата). Основным преимуществом этого подхода является сокращение времени загрузки. При загрузке данных из файлов тремя самыми затратными процессами являются *время поиска* (то есть перемещение считывающей головки в нужное место на физическом носителе), время, необходимое для открытия каждого отдельного файла, и время для чтения данных из файла в память. Из них время поиска и время открытия файла могут быть нетривиальными во многих операционных системах. Когда используется один большой файл, все эти затраты сводятся к минимуму. Один файл может быть расположен на диске последовательно, что сокращает время поиска до минимума. И если нужно открыть только один файл, затраты на открытие отдельных файлов ресурсов исключаются.

Твердотельные накопители (SSD) не страдают от проблем времени поиска, которые мешают вращающимся носителям, таким как DVD, диски Blu-ray и жесткие диски (HDD). Однако пока ни одна игровая приставка не использует твердотельный накопитель в качестве основного устройства хранения данных (даже PS4 и Xbox One). Поэтому разработка шаблонов ввода/вывода в вашей игре для минимизации времени поиска, вероятно, станет обязательной задачей в течение некоторого времени.

Менеджер ресурсов движка для рендеринга OGRE позволяет ресурсам существовать в виде свободных файлов на диске или в виде виртуальных файлов в большом ZIP-архиве. Основные преимущества формата ZIP следующие.

- *Это открытый формат.* Библиотеки `zlib` и `zziplib`, применяемые для чтения и записи ZIP-архивов, находятся в свободном доступе. Zlib SDK является полностью бесплатной (см. www.zlib.net), в то время как zziplib SDK подпадает под действие лицензии Lesser Gnu Public (LGPL) (см. zziplib.sourceforge.net).
- *Виртуальные файлы в ZIP-архиве запоминают свои относительные пути.* Это означает, что ZIP-архив выглядит как простая файловая система для большинства целей и задач. Менеджер ресурсов OGRE идентифицирует все ресурсы уникально через строки, которые кажутся путями файловой системы. Однако эти пути иногда идентифицируют виртуальные файлы в ZIP-архиве, а не свободные файлы на диске, и в большинстве случаев программисту не нужно знать, что между ними есть разница.
- *ZIP-архивы могут быть сжаты.* Это сокращает объем дискового пространства, занимаемого ресурсами. Но, что более важно, это также ускоряет время загрузки, так как загружать в память с фиксированного диска необходимо меньшее количество данных. Это особенно полезно при чтении данных с DVD-ROM или диска Blu-ray, поскольку скорость передачи данных на этих устройствах намного ниже, чем на жестком диске. Следовательно, стоимость распаковки данных после их загрузки в память часто более чем компенсируется временем, сэкономленным при загрузке меньшего количества данных с устройства.
- *ZIP-архивы являются модульными.* Ресурсы могут быть сгруппированы в ZIP-файл и управляться как единое целое. Одним из особенно изящных вариантов

применения этой идеи является локализация продукта. Все ресурсы, которые необходимо локализовать (например, аудиоклипы, содержащие диалоги и текстуры, которые, в свою очередь, содержат слова или региональные символы), можно поместить в один ZIP-файл, а затем создать разные версии этого ZIP-файла, по одной для каждого языка или региона. Чтобы запустить игру для определенного региона, движок просто загружает соответствующую версию ZIP-архива.

Unreal Engine использует аналогичный подход, но с несколькими важными отличиями. В Unreal все ресурсы должны находиться в больших составных файлах, известных как *пакеты* (или PAK-файлы). Свободных файлов на диске не должно быть. Формат файла пакета является проприетарным. Редактор игр Unreal Engine, UnrealEd, позволяет разработчикам создавать пакеты и содержащиеся в них ресурсы и управлять ими.

Форматы файлов ресурсов

Каждый тип файла ресурсов потенциально имеет свой формат. Например, файл меша всегда хранится в формате, отличном от формата растрового изображения текстуры. Некоторые виды активов хранятся в стандартизированных, открытых форматах. Например, текстуры обычно хранятся в виде файлов Targa (TGA), Portable Network Graphics (PNG), Tagged Image File Format (TIFF), Joint Photographic Experts Group (JPEG), Windows Bitmap (BMP) или в стандартизированном сжатом формате, таком как семейство форматов DirectX S3 Texture Compression (S3TC, также известное как DXTn или DXTC). Аналогично данные трехмерного меша часто экспортируются из такого инструмента моделирования, как Maya или Lightwave, в стандартизированный формат, такой как OBJ или COLLADA, для использования игровым движком.

Иногда один формат файла может применяться для размещения множества различных типов активов. Например, *Granny* SDK от Rad Game Tools (www.radgametools.com) реализует гибкий формат открытого файла, который можно задействовать для хранения данных трехмерного меша, иерархий скелета и данных скелетной анимации. (На самом деле формат файла Granny может быть легко перенастроен для хранения практически любых возможных данных.)

Многие программисты игровых движков по разным причинам применяют собственные форматы файлов. Такой подход оправдан, если ни один стандартизированный формат не предоставляет всю информацию, необходимую движку. Кроме того, многие игровые движки стараются выполнять как можно бóльшую автономную обработку, чтобы минимизировать время, необходимое для загрузки и обработки данных ресурсов в среде выполнения. Если, скажем, данные должны соответствовать определенному макету в памяти, то может быть выбран простой двоичный формат для того, чтобы данные можно было обработать автономным инструментом, вместо того чтобы пытаться сделать это в среде выполнения после загрузки ресурса.

GUID ресурсов

Каждый ресурс в игре должен иметь *глобальный уникальный идентификатор* (globally unique identifier, GUID). Наиболее распространенным выбором GUID является путь в файловой системе к ресурсу (хранится в виде строки или 32-разрядного хеша). Этот тип GUID интуитивно понятен, потому что каждому ресурсу соответствует физический файл на диске. И он совершенно точно будет уникальным во всей игре, потому что операционная система уже гарантирует, что никакие два файла не будут иметь одинаковые пути.

Однако путь файловой системы вовсе не единственный выбор для GUID ресурса. Некоторые движки используют менее интуитивно понятный тип GUID, такой как 128-битный хеш-код, возможно назначенный инструментом, который обеспечивает уникальность. В других движках применение пути файловой системы в качестве идентификатора ресурса недопустимо. Например, Unreal Engine хранит много ресурсов в одном большом файле, называемом *пакетом*, поэтому путь к файлу пакета не является уникальным идентификатором отдельного ресурса. Для решения этой проблемы файл пакета Unreal организован в виде иерархии папок, содержащей отдельные ресурсы. Unreal дает каждому отдельному ресурсу в пакете уникальное имя, очень похожее на путь файловой системы. Таким образом, в Unreal GUID ресурса формируется путем объединения уникального имени файла пакета и пути необходимого ресурса в пакете. Например, GUID ресурса `Locust_Boomer.PhysicalMaterials.LocustBoomerLeather` в *Gears of War* указывает на материал с именем `LocustBoomerLeather` в каталоге `PhysicalMaterials` файла пакета `Locust_Boomer`.

Реестр ресурсов

Чтобы гарантировать, что в каждый момент времени в память загружается только одна копия отдельного уникального ресурса, большинство менеджеров ресурсов поддерживают некий *реестр* загруженных ресурсов. Самая простая реализация — это *словарь*, то есть набор *пар «ключ — значение»*. Ключи содержат уникальные идентификаторы ресурсов, а значения обычно являются указателями на ресурсы в памяти.

Всякий раз, когда ресурс загружается в память, запись для него добавляется в словарь реестра ресурсов, используя его GUID в качестве ключа. Всякий раз, когда ресурс выгружается, его запись в реестре удаляется. Когда ресурс запрашивается игрой, менеджер ресурсов ищет его по GUID в реестре ресурсов. Если ресурс найден, указатель на него просто возвращается. Если ресурс не найден, можно либо загрузить его автоматически, либо вернуть код ошибки.

На первый взгляд, если ресурс не найден в реестре ресурсов, наиболее естественным решением может показаться автоматическая загрузка запрошенного ресурса. И действительно, некоторые игровые движки делают именно так. Однако у этого подхода есть серьезные проблемы. Загрузка ресурса — медленная операция,

поскольку включает в себя поиск и открытие файла на диске, считывание предположительно большого объема данных в память (с потенциально медленного устройства, такого как дисковод DVD-ROM), а также, возможно, инициализацию данных ресурса после их загрузки. Если запрос поступает во время активного игрового процесса, то необходимость затратить некоторое время на загрузку ресурса может вызвать очень заметное изменение частоты кадров в игре или даже зависание на несколько секунд. По этой причине движки обычно используют один из двух альтернативных подходов.

- Загрузка ресурсов может быть полностью запрещена во время активного игрового процесса. В этой модели все ресурсы для игрового уровня загружаются *в массовом порядке* непосредственно перед игровым процессом, обычно в то время, когда игрок видит экран загрузки или какой-либо индикатор выполнения.
- Загрузка ресурсов может протекать *асинхронно* (то есть данные будут передаваться с помощью *потоковой передачи*). В этой модели, пока игрок занят на уровне *X*, ресурсы для уровня *Y* загружаются в фоновом режиме. Этот подход предпочтителен, потому что дает игроку возможность играть без экранов загрузки. Однако его значительно сложнее реализовать.

Время жизни ресурса

Время жизни ресурса определяется временем между его первой загрузкой в память и моментом, когда его память забирают для других целей. Одной из задач менеджера ресурсов является управление временем жизни ресурса — либо автоматически, либо предоставлением необходимых API-функций игре, чтобы она сама могла управлять временем жизни ресурса.

Каждый ресурс имеет свои требования к времени жизни.

- Некоторые ресурсы должны быть загружены при первом запуске игры и находиться в памяти на протяжении всего игрового процесса. То есть их время жизни практически бесконечно. Их иногда называют *глобальными ресурсами* или *глобальными активами*. Типичные примеры — это меш, материалы, текстуры и базовые анимации персонажа игрока, текстуры и шрифты для индикаторов HUD, ресурсы для всех стандартных видов оружия, используемых в игре. Любой ресурс, который виден или слышен игроку на протяжении всей игры (и не может быть загружен на лету при необходимости), должен рассматриваться как глобальный.
- Время жизни других ресурсов совпадает с временем жизни определенного уровня игры. Они должны находиться в памяти к моменту, когда игрок впервые видит уровень, и могут быть выгружены после того, как игрок его покинул.
- Время жизни некоторых ресурсов может быть меньше длительности уровня, на котором они находятся. Например, анимации и аудиоклипы, составляющие *внутриигровой видеоролик* (мини-фильм, который продвигает сюжет или предоставляет игроку важную информацию), могут быть загружены до того, как игрок его увидит, и выгружены после окончания.

- Некоторые ресурсы, такие как фоновая музыка, звуки окружающего мира или полноэкранные видео, транслируются вживую во время воспроизведения. Трудно определить время жизни такого рода ресурса, потому что каждый байт сохраняется в памяти в течение мельчайшей доли секунды, но весь музыкальный фрагмент звучит так, как будто он прокручивается в течение длительного времени. Такие активы обычно загружаются порциями, размер которых соответствует требованиям используемого оборудования. Например, музыкальная дорожка может быть прочитана фрагментами по 4 Кбайт, потому что таков размер буфера низкоуровневой звуковой системы. В любой момент в памяти присутствуют только два блока — воспроизводимый в данный момент и следующий непосредственно за ним (он загружается в память).

На вопрос о том, когда загружать ресурс, обычно довольно легко ответить, зная, когда игрок впервые видит его. Однако ответить на вопрос, когда выгрузить ресурс и забрать занимаемую им память, не так просто. Проблема заключается в том, что многие ресурсы одновременно используются несколькими уровнями. Мы не хотим выгружать ресурс по завершении уровня *X*, чтобы сразу же снова его загрузить, потому что уровню *Y* он тоже нужен.

Одним из решений этой проблемы является подсчет ссылок ресурсов. Всякий раз, когда требуется загрузить новый игровой уровень, список всех используемых им ресурсов просматривается и счетчик ссылок для каждого ресурса увеличивается на 1 (но они еще не загружены). Затем мы просматриваем ресурсы любых ненужных уровней и уменьшаем их счетчик ссылок на 1. Любой ресурс, чей счетчик ссылок опускается до 0, выгружается. В заключение мы рассматриваем список ресурсов, чей счетчик ссылок просто изменился с 0 на 1, и загружаем эти ресурсы в память.

Например, представьте, что уровень *X* задействует ресурсы A, B и C, а уровень *Y* —ресурсы B, C, D и E (B и C совместно используются обоими уровнями). В табл. 7.2 показано количество ссылок на эти пять ресурсов, когда игрок проходит уровни *X* и *Y*.

Таблица 7.2. Применение ресурсов при загрузке и выгрузке двух уровней

Событие	**A**	**B**	**C**	**D**	**E**
Изначальное состояние	0	0	0	0	0
Увеличение счетчика уровня *X*	1	1	1	0	0
Уровень *X* загружается	(1)	(1)	(1)	0	0
Уровень *X* проигрывается	1	1	1	0	0
Увеличение счетчика уровня *Y*	1	2	2	1	1
Уменьшение счетчика уровня *X*	0	1	1	1	1
Уровень *X* выгружается, уровень *Y* загружается	(0)	1	1	(1)	(1)
Уровень *Y* проигрывается	0	1	1	1	1

В этой таблице количество ссылок выделено серым, чтобы показать, что соответствующий ресурс действительно существует в памяти, а серый фон говорит о том, что ресурс не находится в памяти. Счетчик ссылок в скобках указывает на загрузку и выгрузку соответствующих данных ресурса.

Управление памятью для ресурсов

Управление ресурсами тесно связано с управлением памятью, потому что нам нужно решить, *где* ресурсы должны находиться в памяти после загрузки. Место назначения каждого ресурса не всегда одно и то же. Прежде всего определенные типы ресурсов должны находиться в видеопамяти RAM (для PlayStation 4 — в блоке памяти, который был выбран для доступа через высокоскоростную «чесночную» шину). Типичные примеры — это текстуры, буферы вершин, буферы индексов и код шейдеров. Большинство других ресурсов могут находиться в основном ОЗУ, но различные виды ресурсов могут требовать разные диапазоны адресов. Например, ресурс, который загружен и в течение всей игры остается в памяти (глобальный), может быть размещен в одной ее области, а часто загружаемые и выгружаемые ресурсы попадут в какую-то другую.

Строение подсистемы выделения памяти для игрового движка обычно тесно связано с ее менеджером ресурсов. Иногда мы проектируем менеджер ресурсов так, чтобы он наиболее эффективно использовал имеющиеся типы распределителей памяти, и наоборот, можем проектировать распределители памяти в соответствии с потребностями менеджера ресурсов.

Как мы видели в подразделе 6.2.1, одной из основных проблем, стоящих перед любой системой управления ресурсами, является необходимость избегать фрагментации памяти при загрузке и выгрузке ресурсов. Далее обсудим несколько наиболее распространенных решений этой проблемы.

Распределение ресурсов на основе кучи. Один из подходов заключается в том, чтобы просто игнорировать проблемы фрагментации памяти и для распределения ресурсов использовать распределитель кучи общего назначения (например, реализованный через `malloc()` в C или глобальный оператор `new` в C++). Это лучше всего работает, когда ваша игра предназначена для запуска только на персональных компьютерах, операционные системы которых поддерживают распределение виртуальной памяти. В такой системе физическая память станет фрагментированной, но способность операционной системы отображать несмежные страницы физической ОЗУ в непрерывное пространство виртуальной памяти помогает сгладить некоторые последствия фрагментации.

Если игра запущена на консоли с ограниченным физическим ОЗУ и простым менеджером виртуальной памяти (или вообще без него), то фрагментация будет проблемой. В этом случае одна из альтернатив — периодически дефрагментировать память. О том, как это сделать, говорилось в подразделе 6.2.2.

Распределение ресурсов на основе стека. Распределитель стека не страдает от проблем фрагментации, поскольку память выделяется непрерывно и освобождается в порядке, противоположном тому, в котором была выделена. Распределитель

стека может использоваться для загрузки ресурсов при выполнении следующих двух условий.

- Игра является линейной и построенной вокруг уровней (то есть игрок видит экран загрузки, проходит уровень, затем видит другой экран загрузки и проходит другой уровень).
- Каждый уровень полностью помещается в память.

Предполагая, что эти требования удовлетворяются, мы можем использовать распределитель стека для загрузки ресурсов следующим образом. Когда игра запускается впервые, глобальные ресурсы распределяются первыми. Затем отмечается вершина стека, чтобы позже можно было вернуться к этой позиции. Чтобы загрузить уровень, мы просто размещаем его ресурсы в верхней части стека. Когда уровень завершен, устанавливаем вершину стека обратно на маркер, который поставили ранее, тем самым освобождая все ресурсы уровня одним махом, не нарушая глобальные ресурсы. Этот процесс может повторяться для любого количества уровней и совсем без фрагментации памяти. На рис. 7.3 показано, как это сделать.

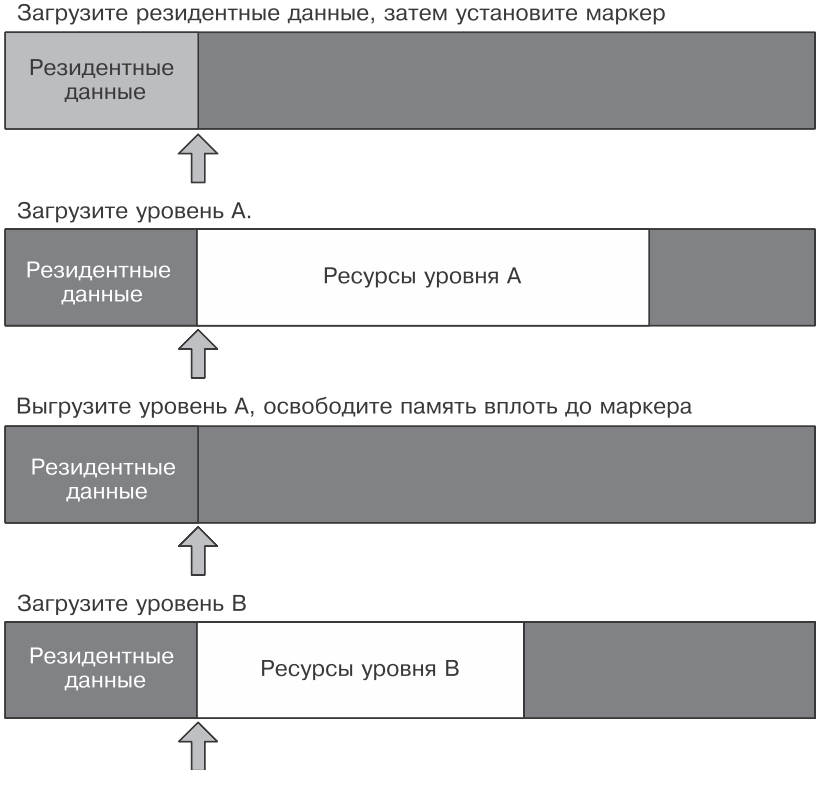


Рис. 7.3. Загрузка ресурсов с использованием стекового распределителя

Двухсторонний распределитель стека может применяться для улучшения этого подхода. Два стека определены в одном большом блоке памяти. Один накапливается внизу области памяти, а другой — вверху. Пока стеки не пересекаются, они могут обмениваться ресурсами памяти естественным образом, что было бы невозможно, если бы каждый из них находился в собственном блоке фиксированного размера.

Midway в игре *Hydro Thunder* использовали двухсторонний распределитель стека. Нижний стек применялся для постоянно загруженных данных, а верхний — для временного выделения памяти, которая освобождалась на каждом кадре. Еще один способ использования двухстороннего распределителя стека — это перебрасывание загрузки уровней. С помощью такого подхода в Bionic Games, Inc. создавали один из своих проектов. Суть заключается в загрузке сжатой версии уровня B в верхний стек, в то время как текущий активный уровень A находится (в несжатом виде) в нижнем стеке. Чтобы переключиться с уровня A на уровень B, освобождаем ресурсы уровня A (очищая нижний стек), а затем распаковываем уровень B из верхнего стека в нижний. Распаковка обычно происходит намного быстрее, чем загрузка данных с диска, поэтому такой подход фактически устраняет необходимость тратить время на загрузку, которая в противном случае могла бы возникнуть у игрока между уровнями.

Распределение ресурсов на основе пула. Другой метод распределения ресурсов, распространенный в игровых движках, поддерживающих потоковую передачу данных, заключается в загрузке данных ресурсов в виде блоков одинакового размера. Поскольку размер порций одинаков, они могут распределяться с помощью *распределителя пула* (см. подраздел 6.2.1). Когда ресурсы позднее выгружаются, блоки могут освобождаться без фрагментации.

Конечно, подход к распределению на основе блоков требует, чтобы все данные о ресурсах располагались таким образом, чтобы их можно было делить на блоки одинакового размера. Мы не можем просто загрузить произвольный файл ресурсов в блоки, потому что он может содержать непрерывную структуру данных, такую как массив, или очень большой `struct` — больше одного блока. Например, если фрагменты, содержащие массив, не расположены последовательно в ОЗУ, непрерывность массива будет нарушена и индексирование массива перестанет работать должным образом. Это означает, что все данные о ресурсах должны быть разработаны с учетом блочности. Следует избегать больших смежных структур данных, предпочитая им структуры данных, которые либо настолько малы, чтобы помещаться в один блок, либо не требуют непрерывного ОЗУ для правильного функционирования (например, связанные списки).

Каждый блок в пуле обычно связан с определенным игровым уровнем. (Один из простых способов сделать это — дать каждому уровню связанный список его блоков.) Это позволяет движку соответствующим образом управлять временем жизни каждого блока, даже когда в памяти одновременно находятся несколько уровней с разными периодами жизни. Например, когда загружен уровень *X*, он может выделять и использовать блоки *N*. Позже уровень *Y* может выделить дополнительные блоки *M*. Когда уровень *X* в конечном итоге выгружается, его блоки *N* возвращаются в свободный пул. Если уровень *Y* все еще загружен, его блоки *M* должны оставаться

в памяти. Связывая каждый блок с определенным уровнем, можно легко и быстро управлять временем жизни блоков (рис. 7.4).

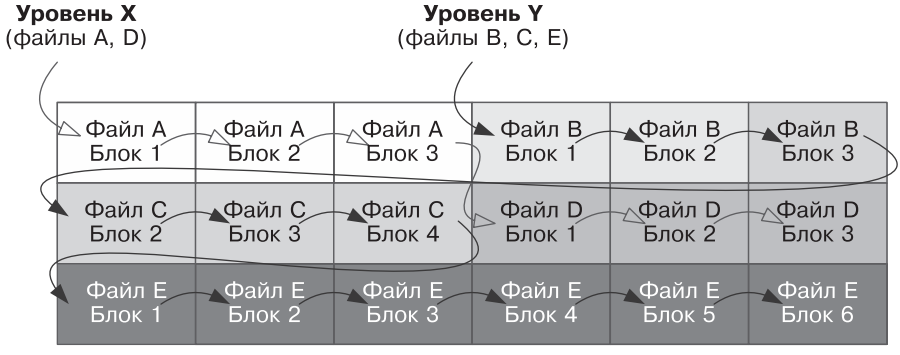


Рис. 7.4. Блочное распределение ресурсов для уровней X и Y

Один большой минус блочной схемы распределения ресурсов — это потраченное впустую пространство. Если размер файла ресурса не кратен размеру блока, последний блок в файле будет использован не полностью (рис. 7.5). Выбор меньшего размера блока может помочь решить эту проблему, но чем меньше блоки, тем более серьезными будут ограничения на размещение данных ресурса. (В качестве яркого примера предположим, что выбран размер фрагмента 1 байт, в таком случае ни одна структура данных не может быть больше 1 байта, что явно невозможно.) Типичный размер фрагмента — порядка нескольких кибибайтов. Например, в Naughty Dog мы применяем объемный распределитель ресурсов как часть системы потоковой передачи ресурсов, и размер блоков 512 Кбайт на PS3 и 1 Мбайт на PS4. Возможно, вы также захотите выбрать размер блока, кратный размеру буфера ввода-вывода операционной системы, чтобы сделать эффективность при загрузке отдельных блоков максимальной.

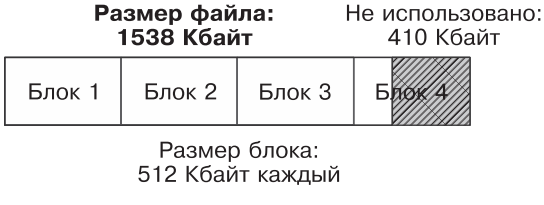


Рис. 7.5. Последний блок файла ресурсов часто используется не полностью

Распределители блоков ресурсов. Один из способов смягчить последствия того, что часть памяти блоков потрачена впустую, состоит в установке специального распределителя памяти, который может задействовать неиспользуемые части блоков. Насколько я знаю, у такого рода распределителя не существует стандартного названия, мы будем называть его *распределителем блоков ресурсов* из-за отсутствия лучшего варианта.

Распределитель блоков ресурсов не особенно сложен в реализации. Нам нужно только поддерживать связанный список всех блоков, которые содержат неиспользуемую память, с указанием расположения и размеров каждого свободного блока. Затем мы можем распределять эти свободные блоки любым удобным способом. Например, мы могли бы управлять связанным списком свободных блоков, задействуя распределитель кучи общего назначения. Или присвоить небольшой распределитель стека каждому свободному блоку и всякий раз, когда поступает запрос на память, сканировать свободные блоки в поисках того, чей стек имеет достаточно свободной оперативной памяти, а затем с помощью этого стека удовлетворить запрос.

К сожалению, в бочке с медом есть довольно неприятная ложка дегтя. Если мы выделим память в неиспользуемых областях блоков ресурсов, что произойдет, когда эти блоки будут освобождены? Мы не можем освободить часть блока — тут действует принцип «все или ничего». Таким образом, любая память, которую мы выделяем в незадействованной части блока ресурса, чудесным образом пропадает, когда этот ресурс выгружается.

Простое решение данной проблемы — применять распределитель свободных блоков только для тех запросов памяти, время жизни которых соответствует времени жизни уровня, с которым связан конкретный блок. Другими словами, мы должны выделять из блоков уровня A память только для данных, связанных исключительно с уровнем A, и выделять из блоков B только ту память, которая используется исключительно уровнем B. Для этого нужно, чтобы распределитель блоков ресурсов управлял блоками каждого уровня отдельно. А это требует, чтобы пользователи распределителя блоков указывали, для какого уровня они выделяются, и, таким образом, удовлетворять запрос можно было бы с помощью правильного связанного списка свободных блоков.

К счастью, большинству игровых движков необходимо динамически распределять память при загрузке ресурсов сверх той памяти, которая нужна для самих файлов ресурсов. Таким образом, распределитель блоков ресурсов может быть полезным способом восстановить память блока, которая иначе была бы потрачена впустую.

Файлы ресурсов, разбитые на секции. Еще одна полезная идея, связанная с блочными файлами ресурсов, — это концепция *файловых секций*. Типичный файл ресурсов может содержать от одной до четырех секций, каждая из которых разделена на один или несколько блоков для выделения пула, как описано ранее. Одна секция может содержать данные, предназначенные для RAM, а другая — данные видеопамяти. Еще в одной секции могут находиться временные данные, необходимые в ходе загрузки ресурса, но отбрасываемые после ее окончания. Четвертая секция может содержать отладочную информацию. Эти отладочные данные могут загружаться при запуске игры в режиме отладки, но не загружаться вообще в финальной сборке игры. Файловая система Granny SDK (www.radgametools.com) является отличным примером простой и гибкой реализации разбиения файлов на секции.

Составные ресурсы и ссылочная целостность

Обычно база данных игровых ресурсов состоит из нескольких *файлов ресурсов*, каждый из которых содержит один или несколько *объектов данных*. Эти объекты данных могут ссылаться друг на друга и зависеть друг от друга произвольным образом. Например, структура данных меша может содержать ссылку на свой материал, который, в свою очередь, содержит список ссылок на текстуры. Обычно перекрестные ссылки подразумевают зависимость (то есть если ресурс A ссылается на ресурс B, то и A, и B должны находиться в памяти, чтобы оба могли функционировать в игре). В общем, база данных ресурсов игры может быть представлена как *ориентированный граф* взаимозависимых объектов данных.

Перекрестные ссылки между объектами данных могут быть *внутренними* (ссылка между двумя объектами в одном файле) или *внешними* (ссылка на объект в другом файле). Это различие важно, потому что внутренние и внешние перекрестные ссылки часто реализуются по-разному. При визуализации базы данных ресурсов игры мы можем нарисовать пунктирные линии, окружающие отдельные файлы ресурсов, чтобы прояснить внутреннее/внешнее различие: любой край графа, который пересекает границу файла в виде пунктирной линии, — это внешняя ссылка, тогда как края, которые не пересекают границы файла, являются внутренними (рис. 7.6).

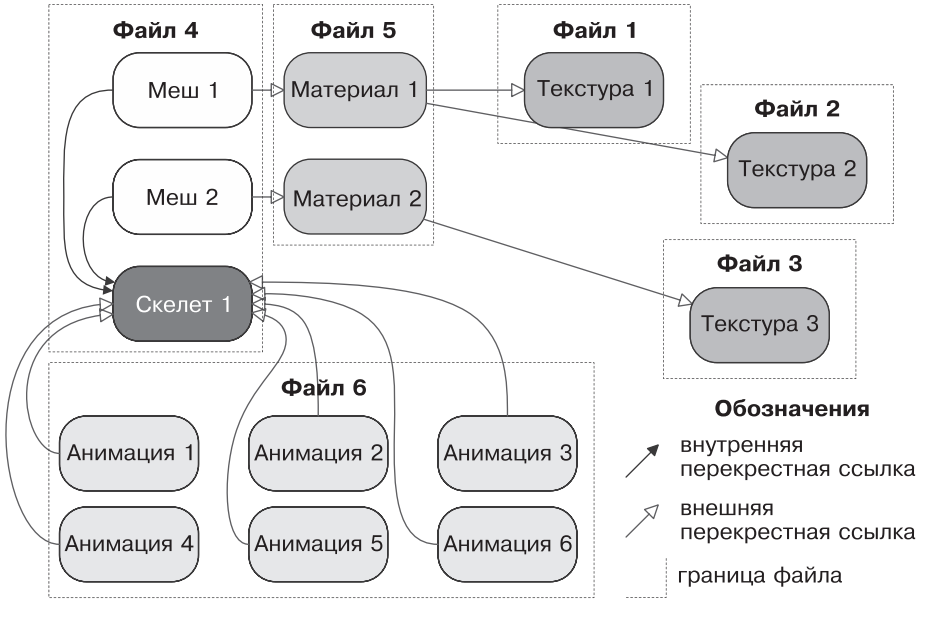


Рис. 7.6. Пример графа зависимости базы данных ресурса

Иногда мы используем термин *«составной ресурс»* для описания самодостаточного кластера взаимозависимых ресурсов. Например, модель — это составной ресурс, состоящий из одной или нескольких *треугольных сеток*, необязательного

скелета и такого же необязательного набора *анимаций*. Каждый меш сопоставлен с *материалом*, а каждый материал относится к одной или нескольким *текстурам*. Чтобы полностью загрузить в память составной ресурс, такой как 3D-модель, нужно загрузить и все его зависимые ресурсы.

Обработка перекрестных ссылок между ресурсами

Одним из наиболее сложных аспектов использования менеджера ресурсов является управление перекрестными ссылками между объектами ресурса и сохранение ссылочной целостности. Чтобы понять, как менеджер ресурсов это выполняет, посмотрим, как перекрестные ссылки представлены в памяти и на диске.

В C++ перекрестная ссылка между двумя объектами данных обычно реализуется через *указатель* или *ссылку*. Например, меш может содержать элемент данных `Material* m_pMaterial` (указатель) или `Material& m_material` (ссылка) для ссылки на свой материал. Однако указатели — это просто адреса памяти, и они теряют свое значение, когда оказываются вырванными из контекста запущенного приложения. Фактически адреса памяти могут изменяться даже при последующих запусках одного и того же приложения. Очевидно, что при сохранении данных в файл на диске мы не можем задействовать указатели для описания межобъектных зависимостей.

GUID в виде перекрестных ссылок. Хорошим подходом является хранение каждой перекрестной ссылки в виде строки или хеш-кода, содержащего уникальный идентификатор объекта, на который ведет ссылка. Это подразумевает, что любой объект ресурса, который может иметь перекрестные ссылки, должен иметь *глобально уникальный идентификатор* (GUID).

Чтобы данный вид перекрестных ссылок работал, менеджер ресурсов среды выполнения поддерживает глобальную таблицу поиска ресурсов. Всякий раз, когда объект ресурса загружается в память, указатель на этот объект сохраняется в таблице с GUID в качестве ключа поиска. После того как все объекты ресурсов были загружены в память, а их записи добавлены в таблицу, мы можем пройтись по всем объектам и преобразовать все их перекрестные ссылки в указатели, найдя адрес каждого объекта, на который ведет ссылка, в глобальной таблице поиска ресурсов с помощью GUID этого объекта.

Таблицы корректировки указателей. Другой подход, который часто используется при хранении объектов данных в двоичном файле, заключается в преобразовании *указателей* в *смещения файлов*. Рассмотрим группу структур C или объектов C++, которые ссылаются друг на друга с помощью указателей. Чтобы сохранить эту группу объектов в двоичном файле, нужно посетить каждый объект один и только один раз в произвольном порядке и последовательно записать образ памяти каждого объекта в файл. Это приводит к сериализации объектов в *непрерывное* изображение в файле, даже если их образы памяти не являются непрерывными в ОЗУ (рис. 7.7).

Поскольку образы памяти объектов теперь располагаются непрерывно внутри файла, мы можем определить *смещение* изображения каждого объекта относительно начала файла. В процессе записи изображения двоичного файла мы находим каждый указатель в каждом объекте данных, преобразуем все указатели в смеще-

ния и сохраняем их в файле вместо указателей. Мы можем просто перезаписать указатели их смещениями, потому что они никогда не требуют больше битов для хранения, чем изначальные указатели. По сути, смещение — это двоичный файл, эквивалентный указателю в памяти. (Помните о различиях между платформой, на которой вы ведете разработку, и целевой платформой. Если записываете образ памяти на 64-разрядной машине Windows, все его указатели будут иметь 64-битную ширину и полученный файл не будет совместим с 32-разрядной консолью.)

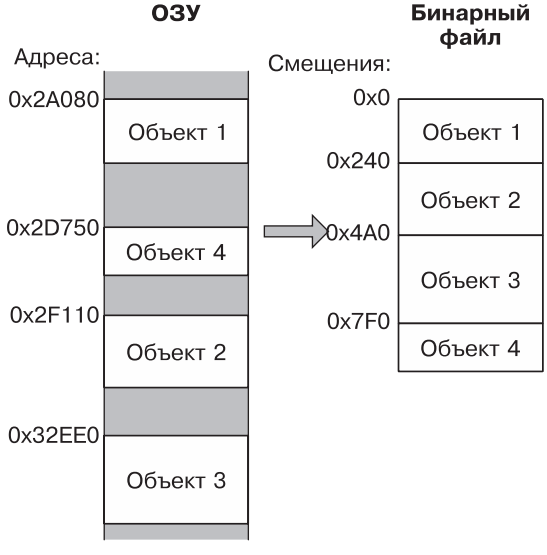


Рис. 7.7. Изображения объектов в памяти становятся непрерывными при сохранении в двоичный файл

Конечно, нам потребуется преобразовать смещения обратно в указатели, когда файл через некоторое время будет загружен в память. Такие преобразования называются *корректировкой указателей*. Когда двоичный образ файла загружен, объекты, содержащиеся в образе, сохраняют свою непрерывную компоновку, поэтому преобразование смещения в указатель не будет проблемой. Мы просто добавляем смещение к адресу образа файла целиком. Это продемонстрировано следующим фрагментом кода и показано на рис. 7.8:

```
U8* ConvertOffsetToPointer(U32 objectOffset,
                           U8* pAddressOfFileImage)
{
    U8* pObject = pAddressOfFileImage + objectOffset;
    return pObject;
}
```

Проблема, с которой мы сталкиваемся, пытаясь преобразовать указатели в смещения и наоборот, заключается в *нахождении* всех указателей, которые требуют преобразования. Эта проблема обычно решается во время записи двоичного файла. Код, который записывает образы объектов данных, знает о записываемых типах

данных и классах, поэтому ему известно расположение всех указателей в каждом объекте. Расположение указателей хранится в простой таблице, известной как *таблица корректировки указателей*. Она записывается в двоичный файл вместе с двоичными образами всех объектов. Позже, когда файл снова загружается в ОЗУ, к таблице можно обратиться, чтобы найти и скорректировать каждый указатель. Сама таблица — это просто список смещений в файле, где каждое смещение представляет один указатель, требующий корректировки (рис. 7.9).

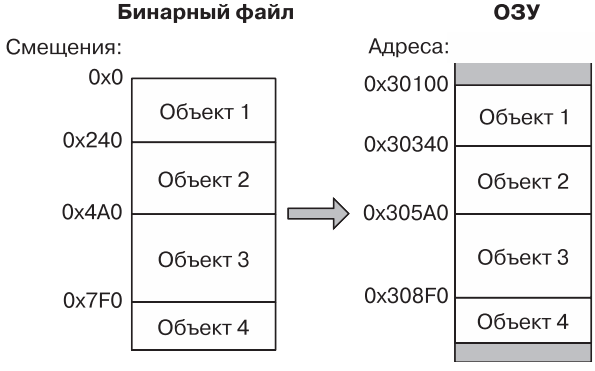


Рис. 7.8. Непрерывный образ файла ресурсов после его загрузки в ОЗУ

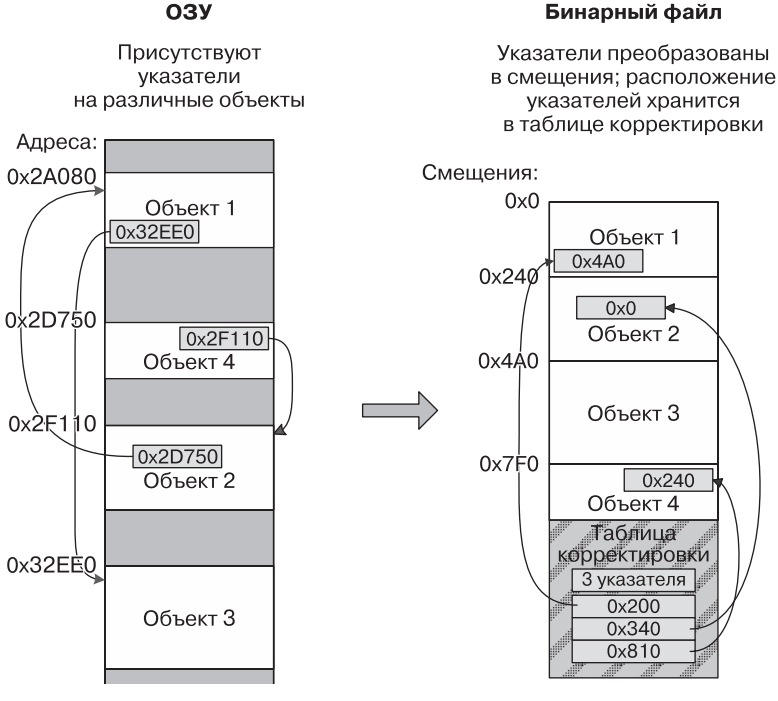


Рис. 7.9. Таблица корректировки указателей

Хранение объектов C++ в виде двоичных образов: конструкторы. Одним из важных шагов, который легко пропустить при загрузке объектов C ++ из двоичного файла, является обеспечение вызова конструкторов объектов. Например, если мы загружаем двоичный образ, содержащий три объекта — экземпляр класса A, экземпляр класса B и экземпляр класса C, — то должны убедиться, что правильный конструктор вызывается для каждого из них.

Есть два стандартных решения этой проблемы. Во-первых, вы можете решить вообще не поддерживать объекты C++ в своих двоичных файлах. Другими словами, ограничьтесь простыми структурами данных (plain old data structures, PODS или POD) — структурами и классами C и C++, которые *не содержат виртуальных функций* и *тривиальных бесполезных конструкторов*. (См. en.wikipedia.org/wiki/Plain_Old_Data_Structures и https://ru.wikipedia.org/wiki/Простая_структура_данных, чтобы лучше понять, что такое PODS.)

Во-вторых, можете сохранить таблицу, содержащую смещения всех не-PODS-объектов в двоичном образе вместе с указаниями, к какому классу относится каждый объект. Затем, как только двоичный образ будет загружен, можете перебирать эту таблицу, просматривать каждый объект и вызывать соответствующий конструктор, используя синтаксис *placement new* (то есть вызывая конструктор в предварительно выделенном блоке памяти). Например, учитывая смещение объекта в двоичном образе, мы можем написать:

```
void* pObject = ConvertOffsetToPointer(objectOffset,
                                       pAddressOfFileImage);
::new(pObject) ClassName; // Синтаксис placement new.
```

где `ClassName` — это класс, экземпляром которого является объект.

Обработка внешних ссылок. Два подхода, описанные ранее, работают очень хорошо, когда применяются к ресурсам, в которых все перекрестные ссылки являются *внутренними*, то есть ссылаются только на объекты в одном файле ресурсов. В этом простом случае вы можете загрузить двоичный образ в память, а затем скорректировать указатели для разрешения всех перекрестных ссылок. Но, когда перекрестные ссылки ведут к другим файлам ресурсов, требуется расширенный подход.

Чтобы успешно составить *внешнюю* перекрестную ссылку, мы должны указать не только смещение или GUID рассматриваемого объекта данных, но и путь к файлу ресурса, в котором находится указанный объект.

Ключом к загрузке многофайлового составного ресурса является загрузка *всех* взаимозависимых файлов. Это можно сделать, загрузив один файл ресурса, а затем просканировав его таблицу перекрестных ссылок и загрузив все еще не загруженные файлы, на которые ведут внешние ссылки. Загружая каждый объект данных в оперативную память, мы можем добавить адрес объекта в главную справочную таблицу. Когда все взаимозависимые файлы загружены и все объекты находятся в оперативной памяти, мы можем сделать последний шаг для корректировки всех указателей, используя главную справочную таблицу для преобразования в реальные адреса GUID или смещения файлов.

Постзагрузочная инициализация

В идеале каждый ресурс должен быть полностью подготовлен автономными инструментами, чтобы его можно было использовать сразу же после загрузки в память. На практике это не всегда возможно. Многие типы ресурсов требуют доработки после загрузки, чтобы быть готовыми к применению движком. В этой книге я буду обозначать термином *«постзагрузочная инициализация»* любую обработку данных ресурса после загрузки. Другие движки могут использовать другую терминологию. (Например, в Naughty Dog мы называем это *входом* в ресурс.) Большинство менеджеров ресурсов реализуют также этап удаления до освобождения памяти ресурса. (В Naughty Dog мы называем это *выходом* из ресурса.)

Постзагрузочная инициализация обычно бывает одного из двух видов.

- В некоторых случаях постзагрузочная инициализация является неизбежным шагом. Например, на ПК вершины и индексы, которые описывают трехмерный меш, загружаются в основную оперативную память, но должны быть переданы в видеопамять перед их рендером. Этого можно достичь только в среде выполнения, создав буфер вершин DirectX или индексный буфер, заблокировав его, скопировав или записав данные в буфер, а затем разблокировав его.
- В других случаях можно избежать обработки, выполняемой во время постзагрузочной инициализации (то есть ее можно перенести в инструменты), но это делается для удобства или из соображений целесообразности. Например, программист может захотеть добавить расчет точной длины дуги в библиотеку сплайнов движка. Вместо того чтобы тратить время на доработку инструментов для генерации данных о длине дуги, он может просто вычислить их в среде выполнения в ходе постзагрузочной инициализации. Позже, когда вычисления станут более совершенными, этот код можно перенести в инструменты, что позволит избежать затрат на вычисления в среде выполнения.

Очевидно, что каждый тип ресурса имеет уникальные требования к инициализации и удалению после загрузки. Таким образом, менеджеры ресурсов обычно разрешают настраивать эти два шага для каждого типа ресурса. В необъектно-ориентированном языке, таком как C, мы можем представить справочную таблицу, которая отображает каждый тип ресурса на пару указателей на функции — один для постзагрузочной инициализации и один для удаления. В объектно-ориентированном языке, таком как C++, все еще проще — можно использовать полиморфизм, чтобы позволить каждому классу обрабатывать постзагрузочную инициализацию и удаление уникальным способом.

В C++ инициализация после загрузки может быть реализована как специальный конструктор, а удаление выполнено в деструкторе класса. Однако существуют некоторые проблемы с применением конструкторов и деструкторов для этой цели. Например, обычно нужно сначала создать все загруженные объекты, затем скорректировать указатель и, наконец, выполнить постзагрузочную инициализацию как отдельный шаг. Таким образом, большинство разработчиков перекладывают постзагрузочную инициализацию и удаление на старые добрые виртуальные функ-

ции. Например, мы могли бы использовать пару логично названных виртуальных функций вроде `Init()` и `Destroy()`.

Постзагрузочная инициализация тесно связана с процессом выделения памяти ресурса, потому что новые данные часто генерируются подпрограммой инициализации. В некоторых случаях данные, создаваемые на этапе постзагрузочной инициализации, *дополняют* данные, загруженные из файла. (Например, если мы рассчитываем длины дуг сегментов сплайновой кривой Кэтмулла — Рома после ее загрузки, то, вероятно, захотим выделить дополнительную память для хранения результатов.) В других случаях данные, сгенерированные в ходе постзагрузочной инициализации, *заменяют* загруженные данные. (Например, мы можем разрешить загрузку данных меша в устаревшем формате, а затем для обеспечения обратной совместимости автоматически преобразовать их в новый формат.) В этом случае загруженные данные, возможно, придется частично или полностью отбросить, после того как постзагрузочный шаг сгенерирует новые данные.

У движка *Hydro Thunder* был простой, но эффективный метод справиться с такой ситуацией. Он разрешал загружать ресурсы одним из двух способов: непосредственно в место, где они до этого находились в памяти, или во временную область памяти. В последнем случае подпрограмма постзагрузочной инициализации отвечала за копирование итоговых данных в конечный пункт назначения, а временная копия ресурса отбрасывалась после завершения постзагрузочной инициализации. Это было очень полезно для загрузки файлов ресурсов, которые содержали как нужные, так и ненужные данные. Нужные данные копировались в конечный пункт назначения в памяти, а ненужные отбрасывались. Например, данные меша в устаревшем формате могут быть загружены во временную память, а затем преобразованы в новый формат с помощью постзагрузочной инициализации, так что не придется тратить память, сохраняя данные старого формата.

8 Игровой цикл и симуляция в реальном времени

Игры представляют собой динамические интерактивные компьютерные симуляции в реальном времени. А значит, *время* играет невероятно важную роль в любой компьютерной игре. В игровом движке есть много разных видов времени: реальное время, игровое время, локальное время анимации, фактические циклы процессора, время, затраченное на выполнение определенной функции, — этот список можно продолжить. Каждая система движка может определять время по-своему и манипулировать им. Мы должны четко понимать все способы применения времени в игре. В этой главе поговорим о том, как работает программное обеспечение для динамического моделирования в реальном времени, и рассмотрим общие способы использования времени в таком моделировании.

8.1. Цикл рендеринга

В графическом пользовательском интерфейсе (graphical user interface, GUI), который можно найти на ПК с операционной системой Windows или Macintosh, большая часть содержимого экрана является статической. Только небольшие детали любого окна активно меняют внешний вид в отдельные моменты времени. Из-за этого графические пользовательские интерфейсы традиционно отображались на экране с помощью метода, известного как *недействительный прямоугольник*, когда перерисовываются лишь небольшие части экрана, содержимое которых фактически изменилось. В старых двухмерных видеоиграх, чтобы минимизировать количество требующих отрисовки пикселов, использовались аналогичные методы.

Компьютерная графика 3D в реальном времени реализована совершенно по-другому. Когда камера перемещается в трехмерной сцене, *все содержимое* экрана или окна постоянно меняется, поэтому концепция недействительных прямоугольников больше не применяется. Вместо этого иллюзия движения и интерактивности создается во многом так же, как это делается в кино, представляя зрителю серию неподвижных изображений, быстро сменяющих друг друга.

Очевидно, что создание такой последовательности неподвижных изображений на экране требует цикла. В приложении рендеринга в реальном времени это ино-

гда называют *циклом рендеринга*. В самом простом виде цикл рендеринга имеет следующую структуру:

```
while (!quit)
{
    // Обновление преобразования камеры на основе
    // интерактивного ввода или следованием по заранее определенному пути.
    updateCamera();

    // Обновляем позиции, направления и любые другие
    // релевантные визуальные состояния любых
    // динамических элементов в сцене.
    updateSceneElements();

    // Рендеринг неподвижного кадра в закадровый
    // буфер, известный как обратный буфер.
    renderScene();

    // Заменяем основной буфер обратным буфером, делая последнее
    // отрисованное изображение видимым на экране. (Или в оконном режиме
    // копируем содержимое обратного буфера в основной буфер.)
    swapBuffers();
}
```

8.2. Цикл игры

Игра состоит из множества взаимодействующих подсистем, включая устройства ввода-вывода, рендеринг, анимацию, обнаружение и разрешение столкновений, дополнительное моделирование динамики твердого тела, многопользовательскую сеть, аудио и т. д. Большинство подсистем игрового движка требуют периодического *обслуживания* во время игры, причем с различной скоростью. Анимация обычно должна обновляться с частотой 30 или 60 Гц синхронно с подсистемой рендеринга. Однако динамическое (физическое) моделирование может потребовать более частых обновлений (например, 120 Гц). Системы более высокого уровня, такие как ИИ, могут нуждаться в обновлении только раз или два в секунду, и их не нужно синхронизировать с циклом рендеринга.

Существует несколько способов периодического обновления подсистем игрового движка. Чуть позже мы рассмотрим некоторые из возможных архитектур. А пока давайте придерживаться самого простого способа обновления подсистем движка — использовать один цикл для обновления всего. Его часто называют *игровым циклом*, потому что это главный цикл, который обслуживает все подсистемы движка.

8.2.1. Простой пример: пинг-понг

Пинг-понг — это известный жанр видеоигр по настольному теннису, который появился в 1958 году в форме аналоговой компьютерной игры под названием *Tennis for Two* («Теннис для двоих»), созданной Уильямом Хигинботамом в Брукхейвенской

национальной лаборатории (играли в нее на осциллографе). Этот жанр лучше всего известен по более поздним воплощениям на цифровых компьютерах — настольному теннису *Magnavox Oddysey* и аркаде *Atari Pong*.

В пинг-понге мяч скачет туда-сюда между двумя подвижными вертикальными ракетками и двумя неподвижными горизонтальными стенками. Игроки-люди контролируют положение ракеток с помощью paddle-контроллера. (Современные реализации позволяют управлять с помощью джойстика, клавиатуры или другого устройства интерфейса.) Если мяч пролетает мимо ракетки, не коснувшись ее, другая команда зарабатывает очко, мяч перемещается в начальное положение и начинается новый цикл игры.

Следующий псевдокод демонстрирует, как может выглядеть цикл игры в пинг-понг:

```
void main() // Пинг-понг
{
    initGame();

    while (true) // игровой цикл
    {
        readHumanInterfaceDevices();

        if (quitButtonPressed())
        {
            break; // выход из игрового цикла
        }

        movePaddles();

        moveBall();

        collideAndBounceBall();

        if (ballImpactedSide(LEFT_PLAYER))
        {
            incremenentScore(RIGHT_PLAYER);
            resetBall();
        }
        else if (ballImpactedSide(RIGHT_PLAYER))
        {
            incrementScore(LEFT_PLAYER);
            resetBall();
        }

        renderPlayfield();
    }
}
```

Очевидно, этот пример несколько надуманный. В оригинальных играх в пинг-понг, конечно же, весь экран не перерисовывался со скоростью 30 кадров в секунду.

Тогда процессоры были настолько медленными, что едва могли провести две линии для ракеток и отрисовать поле для мяча в режиме реального времени. Для отображения движущихся объектов на экране часто использовались специализированные аппаратные средства 2D-спрайтов. Однако нас интересуют только идеи, а не детали реализации оригинального пинг-понга.

Вы видите, что, когда игра запускается впервые, она вызывает `initGame()`, чтобы выполнить любые настройки, которые могут потребоваться графической системе, устройствам ввода-вывода, аудиосистеме и т. д. Затем начинается основной игровой цикл. Утверждение `while(true)` говорит о том, что цикл будет продолжаться вечно, пока не будет прерван изнутри. Первое, что мы делаем внутри цикла, — считываем ввод устройства (устройств) ввода. Мы проверяем, нажал ли кто-то из игроков кнопку Exit (Выход), и если да, то выходим из игры с помощью оператора `break`. Затем ракетки слегка передвигаются вверх или вниз в `movePaddles()` в зависимости от текущего отклонения контроллера, джойстиков или других устройств ввода/вывода. Функция `moveBall()` прибавляет вектор текущей скорости шара к его положению, чтобы найти новое положение в следующем кадре. В `collideAndBounceBall()` эта позиция проверяется на наличие столкновений как с неподвижными горизонтальными стенками, так и с ракетками. Если обнаружены столкновения, положение мяча пересчитывается с учетом отскока. Также отмечаем, коснулся ли мяч левого или правого края экрана. Это означает, что он пролетел мимо одной из ракеток. В этом случае мы увеличиваем счет другого игрока и сбрасываем мяч в начальное положение для следующего раунда. Наконец, `renderPlayfield()` рисует все содержимое экрана.

8.3. Архитектурные стили цикла игры

Игровые циклы могут быть реализованы разными способами, но обычно сводятся к одному или нескольким простым циклам с различными дополнениями. Далее мы рассмотрим некоторые из наиболее распространенных архитектур.

8.3.1. Конвейер сообщений Windows

На платформе Windows в дополнение к обслуживанию различных подсистем самого игрового движка игры должны обслуживать сообщения от операционной системы Windows. Поэтому игры для Windows содержат кусок кода, известный как *конвейер сообщений*. Основная его функция состоит в том, чтобы обслуживать сообщения Windows каждый раз, когда они поступают, а игровой движок — только когда сообщений Windows нет. Конвейер сообщений обычно выглядит примерно так:

```
while (true)
{
    // Обслуживаем все ожидающие сообщения Windows.
    MSG msg;
```

```
    while (PeekMessage(&msg, nullptr, 0, 0) > 0)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // Нет больше сообщений Windows для обработки – запустим одну
    // итерацию нашего реального игрового цикла.
    RunOneIterationOfGameLoop();
}
```

Одним из побочных эффектов реализации игрового цикла, подобного данному, является то, что сообщения Windows имеют приоритет над визуализацией и симуляцией игры. В результате игра будет зависать при изменении размера окна игры на Рабочем столе или его перетаскивании.

8.3.2. Фреймворки на основе обратных вызовов

Большинство подсистем игрового движка и сторонние поддерживающие программные пакеты структурированы как *библиотеки*. Библиотека — это набор функций и/или классов, которые могут быть вызваны любым удобным для разработчика приложения способом. Библиотеки обеспечивают максимальную гибкость для программиста. В то же время иногда с ними трудно работать, потому что разработчик должен понимать, как правильно использовать функции и классы, которые они предоставляют.

Поэтому некоторые игровые движки и пакеты программного обеспечения для игр структурированы как *фреймворки*. Фреймворк — это частично сконструированное приложение — программист завершает его, предоставляя пользовательские реализации отсутствующих функций в рамках фреймворка или переопределяя его поведение по умолчанию. Но специалист практически не контролирует общий поток управления в приложении, поскольку его контролирует фреймворк.

В движке рендеринга или игровом движке на основе фреймворка основной игровой цикл обычно уже написан, но он, как правило, пустой. Разработчик игры может написать функции обратного вызова, чтобы добавить недостающие детали. Механизм рендеринга OGRE является примером библиотеки, которая была обернута во фреймворк. На самом низком уровне OGRE предоставляет функции, которые может вызывать непосредственно программист игрового движка. Тем не менее OGRE также предоставляет структуру, которая знает, как эффективно использовать низкоуровневую библиотеку OGRE. Если программист решает задействовать фреймворк OGRE, он наследует от класса `Ogre::FrameListener` и переопределяет две виртуальные функции: `frameStarted()` и `frameEnded()`. Как можно догадаться, эти функции вызываются соответственно до и после того, как основная 3D-сцена отрисовывается OGRE. Реализация в OGRE внутреннего игрового цикла выглядит примерно так, как в следующем псевдокоде (можете посмотреть реализацию функции `Ogre::Root::renderOneFrame()` в `OgreRoot.cpp`, чтобы получить настоящий исходный код):

```
while (true)
{
    for (each frameListener)
    {
        frameListener.frameStarted();
    }

    renderCurrentScene();

    for (each frameListener)
    {
        frameListener.frameEnded();
    }

    finalizeSceneAndSwapBuffers();
}
```

Реализация слушателя фрейма конкретной игры может выглядеть примерно так:

```
class GameFrameListener : public Ogre::FrameListener
{
public:
    virtual void frameStarted(const FrameEvent& event)
    {
        // Делаем то, что должно произойти до рендеринга 3D-сцены
        // (то есть обслуживаем все подсистемы игрового движка).
        pollJoypad(event);
        updatePlayerControls(event);
        updateDynamicsSimulation(event);
        resolveCollisions(event);
        updateCamera(event);

        // и т. д.
    }

    virtual void frameEnded(const FrameEvent& event)
    {
        // Делаем то, что должно произойти после рендеринга 3D-сцены и т. д.
        drawHud(event);

        // и т. д.
    }
};
```

8.3.3. Обновление на основе событий

Под *событием* в играх понимается любое интересное изменение состояния игры или ее среды. Вот некоторые примеры: игрок-человек, нажимающий кнопку на джойстике, взрыв, вражеский персонаж, замечающий игрока, и этот список можно продолжить. Большинство игровых движков имеют *систему событий*, которая

позволяет различным подсистемам регистрировать интерес к определенным видам событий и реагировать на них, когда они происходят (подробности см. в разделе 16.8). Система событий в игре обычно очень похожа на систему событий/сообщений, лежащую в основе практически всех графических пользовательских интерфейсов, например оконные сообщения Microsoft Windows, система обработки событий в AWT Java или службы, предоставляемые ключевыми словами C# `delegate` и `event`.

Некоторые игровые движки используют свою систему событий для периодического обслуживания части или всех своих подсистем. Чтобы это работало, система событий должна разрешать планирование событий на будущее, то есть ставить их в очередь для последующей доставки. При этом игровой движок может осуществлять периодическое обновление, просто публикуя событие. В обработчике событий код способен выполнить любое периодическое обслуживание, которое требуется. При этом он может запланировать новое событие через 1/30 или 1/60 секунды, тем самым производя периодическое обслуживание столько времени, сколько потребуется.

8.4. Абстрактные временные шкалы

В игровом программировании может быть чрезвычайно полезно мыслить в терминах *абстрактных временных шкал*. Временная шкала — это непрерывная одномерная ось, начало которой ($t = 0$) может находиться в произвольном месте относительно других временных шкал в системе. Временная шкала может быть реализована с помощью простой переменной часов, которая хранит абсолютные значения времени в целочисленном формате или в формате с плавающей точкой.

8.4.1. Реальное время

Мы будем рассуждать об отрезках времени, измеренных непосредственно через регистр таймера высокого разрешения ЦП (см. подраздел 8.5.3), и именно поэтому будем называть такое время *реальной временно́й шкалой*. Ее начало определено так, чтобы совпадать с моментом последнего включения или сброса процессора. Он измеряет время в единицах циклов ЦП (или нескольких их кратных), хотя эти значения времени можно легко преобразовать в единицы секунд, умножив их на частоту таймера высокого разрешения на текущем ЦП.

8.4.2. Игровое время

Нам не нужно ограничиваться работой исключительно в реальном времени. Мы можем определить столько других временных шкал, сколько нужно для решения имеющихся проблем. Например, определить *игровую временную шкалу*, которая технически не зависит от реального времени. В обычных условиях игровое время совпадает с реальным. Если мы хотим приостановить игру, то можем просто временно приостановить обновление игровой шкалы. Если нужно, чтобы игра шла

медленно, можем обновлять игровые часы медленнее, чем часы реального времени. Можно получить всевозможные эффекты масштабированием и деформацией одной временной шкалы относительно любой другой.

Приостановка или замедление игровых часов — также очень полезный инструмент отладки. Чтобы отследить визуальную аномалию, разработчик может приостановить игровое время и тем самым заморозить действие. При этом механизм рендеринга и отладочная перемещаемая камера могут продолжать работать, если ими управляют другие часы (либо *часы реального времени*, либо отдельные *часы камеры*). Это позволяет разработчику развернуть камеру в игровом мире, чтобы осмотреть его под другим углом. Мы даже можем поддерживать пошаговые игровые часы, продвигая игровые часы на один целевой интервал кадра (например, 1/30 секунды) каждый раз, когда нажата кнопка «один шаг» на джойстике или клавиатуре в момент паузы в игре.

При использовании описанного ранее подхода важно понимать, что игровой цикл все еще идет, когда игра приостановлена, — стоят только игровые часы. Пошаговое выполнение игры путем добавления 1/30 секунды к приостановленным игровым часам — это не то же самое, что установка точки останова в основном цикле, а затем нажатие клавиши F5 для запуска одной итерации цикла. Оба типа «одного шага» могут быть полезны для отслеживания различных проблем. Просто нужно помнить различия между этими подходами.

8.4.3. Локальное и глобальное время

Мы можем придумать множество других видов временных шкал. Например, у анимационного клипа или аудиоклипа может быть *локальная временная шкала*, начало которой ($t = 0$) определено так, чтобы совпадать с началом клипа. Локальная временная шкала показывает время таким, каким оно было, когда клип был создан или записан. Когда клип воспроизводится в игре, нам не нужно делать это с первоначальной скоростью. Возможно, мы захотим ускорить анимацию или замедлить аудио. Мы даже можем воспроизвести анимацию от конца к началу, запустив локальные часы в обратном направлении.

Любой из этих эффектов может быть визуализирован как *зависимость* между локальной временной шкалой и глобальной временной шкалой, такой как реальное или игровое время. Чтобы воспроизвести анимационный клип с первоначальной скоростью, мы просто отображаем начало локальной временной шкалы анимации ($t = 0$) на желаемое время начала ($\tau = \tau_{start}$) вдоль глобальной временной шкалы (рис. 8.1).

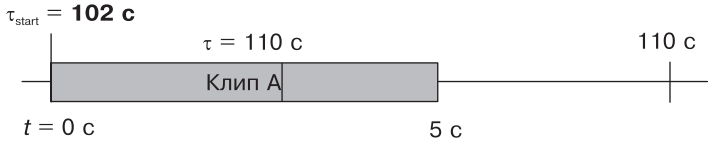


Рис. 8.1. Воспроизведение анимационного клипа можно визуализировать как отображение его локальной временной шкалы на глобальную игровую временную шкалу

Чтобы воспроизвести анимационный клип с половинной скоростью, мы можем представить этот процесс как масштабирование локальной временной шкалы в два раза по сравнению с первоначальным размером до ее отображения на глобальной временной шкале. Для этого вводим коэффициент масштабирования времени или скорость воспроизведения R в дополнение к глобальному времени начала клипа τ_{start} (рис. 8.2). Клип даже можно воспроизвести задом наперед, используя отрицательный коэффициент для шкалы времени $R < 0$ (рис. 8.3).

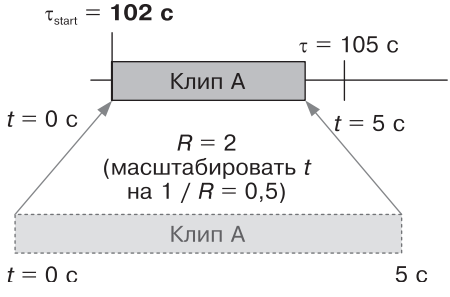


Рис. 8.2. Скорость воспроизведения анимации можно контролировать, масштабируя локальную временную шкалу перед отображением на глобальную временную шкалу

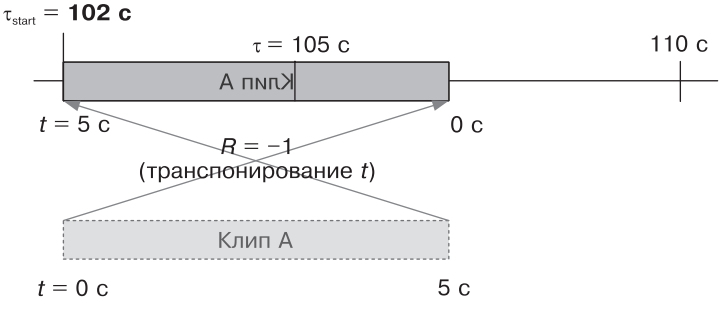


Рис. 8.3. Воспроизведение анимации в обратном направлении похоже на отображение клипа на глобальную временную шкалу при $R = -1$

8.5. Измерение времени и работа с ним

В этом разделе мы рассмотрим некоторые тонкие и не очень различия между разными типами временных шкал и часов и посмотрим, как они реализованы в реальных игровых движках.

8.5.1. Частота смены кадров и время

Частота кадров в игре в реальном времени показывает, насколько быстро последовательность неподвижных трехмерных кадров представляется зрителю. Единица измерения *Герц* (Гц), определяемая как число циклов в секунду, может

использоваться для описания скорости любого периодического процесса. В играх и фильмах частота кадров, как правило, измеряется в *кадрах в секунду* (FPS), так что она аналогична герцам в контексте любых целей и задач. Фильмы традиционно показывают с частотой 24 FPS. Игры в Северной Америке и Японии обычно воспроизводят со скоростью 30 или 60 кадров в секунду, поскольку это унаследованная от цветного телевидения стандарта NTSC частота обновления цветного телевидения NTSC, применяемого в тех регионах. В Европе и большинстве остальных стран мира игры обновляются со скоростью 50 FPS, так как это принятая частота обновления цветного телевизионного сигнала стандарта PAL или SECAM.

Количество времени, которое проходит между кадрами, называется *временем кадра*, *временной дельтой* или *дельтой времени*. Последний термин вполне обычен, поскольку длительность между кадрами часто математически представляется символом Δt. (Чисто технически Δt действительно следует называть *периодом кадра*, так как это параметр, обратный *частоте кадра*: $T = 1/f$. Но разработчики игр почти никогда не используют термин «период» в этом контексте.) Если игра рендерится со скоростью ровно 30 кадров в секунду, тогда ее дельта времени составляет 1/30 секунды, или 33,3 миллисекунды. При скорости 60 кадров в секунду временная дельта в два раза меньше — 1/60 секунды, или 16,6 миллисекунды. Чтобы действительно знать, сколько времени прошло за одну итерацию игрового цикла, нужно измерить его. Рассмотрим, как это сделать.

Здесь следует отметить, что миллисекунды являются обычной единицей измерения времени в играх. Например, мы можем сказать, что анимационная система работает 4 миллисекунды, что означает: она занимает около 12 % кадра (4 / 33,3 = 0,12). Другие распространенные единицы измерения — секунды и машинные циклы. Мы обсудим единицы времени и переменные часов более подробно далее.

8.5.2. От частоты кадров к скорости

Допустим, мы хотим, чтобы космический корабль летел через игровой мир с постоянной скоростью 40 метров в секунду (если бы это была 2D-игра, мы могли бы сказать: 40 *пикселов* в секунду). Простой способ сделать это — умножить скорость корабля v (измеряется в метрах в секунду) на длительность одного кадра Δt (измеряется в секундах), что приведет к изменению положения на $\Delta x = v\Delta t$ (измеряется в метрах на кадр). Эту дельту позиции затем можно добавить к текущей позиции корабля x_1, чтобы найти его позицию в следующем кадре: $x_2 = x_1 + \Delta x = x_1 + v\Delta t$.

На самом деле это простая форма *численного интегрирования*, известная как *явный метод Эйлера* (см. подраздел 13.4.4). Она работает хорошо, пока скорости объектов примерно постоянны. Для обработки переменных скоростей нужно прибегнуть к более сложным методам интегрирования. Но все методы численного интегрирования так или иначе используют прошедшее время кадра Δt. Так что можно с уверенностью сказать, что *воспринимаемые скорости* объектов в игре зависят от длительности кадра Δt. Следовательно, центральной проблемой при разработке игр является определение подходящего значения Δt. Далее мы обсудим различные способы сделать это.

Старые игры, зависимые от скорости работы CPU

Во многих ранних видеоиграх не предпринималось никаких попыток измерить, сколько реального времени прошло во время итерации игрового цикла. Программисты, по существу, полностью игнорировали Δt и вместо этого указывали скорости объектов непосредственно в метрах (или пикселах, или какой-то другой единице расстояния) на *кадр*. Другими словами, они, возможно, невольно указывали скорость объекта в терминах $\Delta x = v\Delta t$, а не в терминах v.

Общий результат реализации этого упрощенного подхода состоял в том, что воспринимаемые скорости объектов в этих играх полностью зависели от частоты кадров, которой игра фактически достигала на определенной части оборудования. Если бы игра такого типа запускалась на компьютере с более быстрым ЦП, чем тот, что на машине, для которой она была изначально написана, она, по-видимому, работала бы в ускоренном режиме. По этой причине я буду называть такие игры *зависимыми от ЦП*.

На некоторых старых компьютерах предусмотрена кнопка Turbo для поддержки игр такого типа. При ее нажатии ПК будет работать с максимальной скоростью и игры, зависящие от процессора, станут работать быстрее. Когда кнопка Turbo не была нажата, ПК имитировал скорость процессора ПК более старого поколения, что позволяло правильно запускать игры, зависящие от скорости процессора, написанные для этих ПК.

Обновление на основе прошедшего времени

Чтобы сделать игры независимыми от процессора, мы должны каким-то образом измерять Δt, а не просто игнорировать его. Делать это довольно просто. Мы считываем значение таймера высокого разрешения ЦП дважды — в начале и в конце кадра. Затем вычитаем одно значение из другого, получая точное значение Δt для только что пройденного кадра. Затем делаем эту дельту доступной всем подсистемам движка, которые в ней нуждаются, либо передавая ее каждой функции, которую мы вызываем из игрового цикла, либо сохраняя в глобальной переменной, либо инкапсулируя в какой-то одноэлементный класс. (Мы опишем таймер высокого разрешения процессора более подробно в подразделе 8.5.3.)

Описанный подход характерен для многих игровых движков. На самом деле мне хочется схитрить и сказать, что его применяют *большинство* движков. Однако у этой техники есть одна большая проблема: мы используем измеренное значение Δt, взятое в течение кадра k, в качестве оценки продолжительности *следующего* кадра ($k + 1$). Иногда это не дает достаточного, точного результата. (Как говорится по поводу инвестиций: «Полученные в прошлом результаты не являются гарантией будущих результатов».) В следующем кадре может произойти что-то, что заставит его занять гораздо больше времени (или намного меньше), чем текущий кадр. Мы называем такое событие *скачком частоты кадров*.

Использование дельты последнего кадра в качестве оценки предстоящего кадра может иметь некоторые очень реальные вредные последствия. Например, если мы

не будем осторожны, это может поставить игру в «вязкий» цикл с плохими значениями времени кадров. Предположим, что физическое моделирование наиболее стабильно при обновлении каждые 33,3 миллисекунды (то есть при 30 Гц). Если мы получим один плохой кадр, скажем, 57 миллисекунд, то можем сделать ошибку *двукратного* перехода системы физики на следующий кадр для покрытия прошедших 57 миллисекунд. Эти два шага выполняются примерно вдвое дольше обычного шага, в результате чего *следующий* кадр будет по крайней мере таким же плохим, как этот, а возможно, и хуже. Это только усугубляет и продлевает проблему.

Использование среднего значения

Не секрет, что игровые циклы склонны иметь хотя бы какую-то межкадровую согласованность. Если камера направлена вниз по коридору, где имеется множество дорогих для рисования объектов в одном кадре, существует большая вероятность того, что камера будет направлена вниз по тому же коридору и в следующем кадре. Значит, один разумный подход заключается в усреднении измерений времени кадра по небольшому количеству кадров и использовании их в качестве оценки Δt следующего кадра. Это позволяет игре адаптироваться к различной частоте кадров, одновременно смягчая эффекты мгновенных всплесков производительности. Чем длиннее интервал усреднения, тем менее чувствительной будет игра к изменяющейся частоте кадров, но пики тоже будут влиять меньше.

Управление частотой кадров

Мы можем избежать неточности использования Δt последнего кадра в качестве оценки его длительности в целом, перевернув проблему с ног на голову. Вместо того чтобы пытаться *угадать*, какой будет длительность следующего кадра, мы можем попытаться *гарантировать*, что длительность каждого кадра будет ровно 33,3 миллисекунды (или 16,6 миллисекунды, если работаем со скоростью 60 кадров в секунду). Для этого измеряем длительность текущего кадра, как и раньше. Если измеренная продолжительность меньше идеального времени кадра, просто переводим основной поток в спящий режим, пока не истечет целевое время кадра. Если измеренная длительность превышает идеальное время кадра, мы должны «смириться с нашими ошибками» и подождать, пока не истечет еще одно целое время кадра. Это называется *управлением частотой кадров*.

Очевидно, что этот подход работает только тогда, когда частота кадров игры довольно близка к целевой частоте кадров в среднем. Если игра постоянно скачет между 30 и 15 FPS из-за множества «медленных» кадров, то ее качество может значительно ухудшиться. Таким образом, все еще является хорошей идеей проектирование всех систем движка таким образом, чтобы они могли работать с произвольной длительностью кадров. Во время разработки вы можете оставить движок в режиме переменной частоты кадров, и все будет работать как положено. Позже, когда игра приблизится к достижению целевой частоты кадров, можно включить управление частотой кадров и начать пользоваться его преимуществами.

Сохранить частоту кадров может быть важно по ряду причин. Некоторые системы движка, такие как числовые интеграторы, используемые в физическом моделировании, работают лучше всего при обновлении с постоянной скоростью. Постоянная частота кадров также выглядит лучше, и, как мы увидим в следующем разделе, ее можно задействовать, чтобы избежать *разрывов*, которые могут возникнуть при обновлении видеобуфера со скоростью, не соответствующей частоте обновления монитора (см. подраздел 8.5.2).

Кроме того, когда длительность кадров постоянна, такие функции, как *запись* и *воспроизведение*, становятся намного надежнее. Как следует из названия, функция записи и воспроизведения позволяет записывать игровой процесс пользователя, а затем точно так же воспроизводить его. Это может стать забавной игровой функцией и ценным инструментом для тестирования и отладки. Например, трудно обнаруживаемую ошибку можно повторить, просто воспроизведя записанную игру, которая ее демонстрирует.

Для реализации записи и воспроизведения мы отмечаем каждое такое событие, которое происходит во время игры, сохраняя его в списке вместе с точной временной меткой. Затем список событий можно воспроизвести с точно такой же синхронизацией, используя те же начальные условия и идентичное начальное случайное начальное число. Теоретически это должно привести к воспроизведению игрового процесса, неотличимого от первоначального прохождения игры. Однако, если частота кадров непостоянна, действия могут происходить не в том порядке. Это может вызвать «дрейф», и довольно скоро ваши ИИ-персонажи станут наступать, когда должны были отступить.

Разрыв изображения и V-Sync

Визуально аномалия, известная как *разрыв изображения*, возникает, когда второй буфер меняется с основным буфером, а экран лишь частично был отрисован видеооборудованием. Когда происходит разрыв, на одной части экрана изображение новое, а на другой — все еще старое. Чтобы избежать разрывов, многие движки рендеринга ждут завершения *вертикального гасящего импульса* монитора, прежде чем менять буферы местами.

Старые мониторы и телевизоры с электронно-лучевой трубкой рисуют содержимое буфера кадров в памяти, возбуждая люминофор на экране с помощью пучка электронов, который сканирует экран слева направо и сверху вниз. На таких дисплеях интервал v-blank — это время, в течение которого электронная пушка гаснет (выключается), перемещаясь в левый верхний угол экрана. Современные жидкокристаллические, плазменные и светодиодные дисплеи не используют электронный луч, и им не требуется время между завершением рисования последней строки одного кадра и началом рисования первой строки следующего. Но интервал v-blank все еще существует — отчасти потому, что стандарты видео были установлены, когда ЭЛТ были нормой, а отчасти из-за необходимости поддерживать старые дисплеи.

Ожидание интервала v-blank называется *v-sync*. На самом деле это просто еще одна форма управления частотой кадров, поскольку она эффективно ограничивает частоту кадров основного игрового цикла, делая ее кратной частоте обновления экрана. Например, на мониторе NTSC, который обновляется с частотой 60 Гц, реальная частота обновления игры эффективно квантуется с кратностью 1/60 секунды. Если между кадрами проходит более 1/60 секунды, мы должны дождаться следующего интервала v-blank, что означает ожидание 2/60 секунды (30 FPS). Пропустив два v-blank, нужно подождать в целом 3/60 секунды (20 FPS) и т. д. Кроме того, будьте осторожны, прежде чем сделать предположение о частоте кадров вашей игры, даже если эта частота синхронизирована с интервалом v-blank: вы должны помнить, что стандарты PAL и SECAM основаны на частоте обновления 50, а не 60 Гц.

8.5.3. Измерение реального времени с помощью таймера высокого разрешения

Мы много говорили об измерении относительно реальных «настенных часов» — времени, которое проходит в каждом кадре. В этом разделе подробно рассмотрим, как такое измерение времени реализуется.

Большинство операционных систем предоставляют функцию для запроса системного времени, такую как функция `time()` стандартной библиотеки C. Однако эти функции не подходят для измерения прошедшего времени в игре в реальном времени, поскольку они не обеспечивают достаточной точности. Например, функция `time()` возвращает целое число, представляющее количество *секунд* с полуночи 1 января 1970 года, поэтому ее разрешение составляет 1 секунду — оно слишком грубое, учитывая, что для выполнения кадра требуются всего десятки миллисекунд.

Все современные ЦП содержат *таймер высокого разрешения*, обычно реализуемый как аппаратный регистр, который подсчитывает количество циклов ЦП (или несколько кратных им), прошедших с момента последнего включения или перезагрузки процессора. Это таймер, который мы должны использовать при измерении истекшего в игре времени, потому что его точность обычно порядка нескольких циклов ЦП. Например, на процессоре Pentium с тактовой частотой 3 ГГц показания таймера высокого разрешения увеличиваются один раз за цикл ЦП, или 3 млрд раз в секунду. Следовательно, его разрешение составляет 1/3 млрд = $3{,}33 \cdot 10^{-10}$ секунды = 0,333 наносекунды. Это более чем удовлетворяет все наши потребности измерения времени в игре.

Различные микропроцессоры и операционные системы предоставляют разные способы запроса таймера высокого разрешения. На Pentium можно использовать специальную инструкцию `rdtsc` (read time-stamp counter — «считать счетчик времени»), хотя Win32 API оборачивает ее в пару функций: `QueryPerformanceCounter()` считывает регистр 64-битного счетчика, а `QueryPerformanceFrequency()` возвращает количество приращений счетчика в секунду для текущего процессора. В архитектуре PowerPC, построеной на таких чипах, как находящиеся в Xbox 360 и PlayStation 3, применяется команда `mftb` (move from time base

register), работающая с двумя 32-разрядными регистрами, тогда как в других архитектурах PowerPC вместо этого берется инструкция `mfspr` (перемещение из регистра специального назначения).

Регистр таймера высокого разрешения ЦП имеет ширину 64 бита на большинстве процессоров, чтобы гарантировать, что он не будет переноситься слишком часто. Максимально возможное значение 64-разрядного целого числа без знака равно 0xFFFFFFFFFFFFFFFF $\approx 1{,}8 \cdot 10^{19}$ тактов. Таким образом, на процессоре Pentium с тактовой частотой 3 ГГц, который обновляет свой таймер высокого разрешения один раз за цикл ЦП, значение регистра будет возвращаться к нулю один раз каждые 195 лет или около того — это определенно не та ситуация, из-за которой нужно сильно переживать. Для сравнения: 32-разрядные целочисленные часы на частоте 3 ГГц будут возвращаться к нулю примерно через 1,4 секунды.

Сдвиг времени в часах высокого разрешения

Помните, что в определенных обстоятельствах даже измерения времени, сделанные с помощью таймера высокого разрешения, могут быть неточными. Например, на некоторых многоядерных процессорах таймеры высокого разрешения независимы для каждого ядра и могут (и будут) расходиться. Сравнив друг с другом абсолютные показания таймера, полученные на разных ядрах, вы можете получить странные результаты — даже отрицательные дельты времени. Обязательно учитывайте эти особенности.

8.5.4. Единицы измерения времени и переменные часов

Всякий раз, когда мы измеряем или указываем продолжительность времени в игре, нам надо принять два решения.

- Какие *единицы измерения* времени следует использовать? Мы хотим хранить время в секундах, или миллисекундах, или машинных циклах... или в какой-то другой единице?
- Какой *тип данных* следует применять для хранения измерений времени? Должны ли мы использовать 64-разрядное целое число, или 32-разрядное целое число, или 32-разрядную переменную с плавающей точкой?

Ответы на эти вопросы зависят от предполагаемой цели измерения. Они вызывают еще два вопроса: насколько точные измерения нам нужны и представления какого диапазона величин мы ожидаем?

64-разрядные целочисленные часы

Мы уже видели, что 64-разрядные целочисленные такты без знака, измеренные в машинных циклах, поддерживают как чрезвычайно высокую точность (длительность одного цикла составляет 0,333 нс на процессоре с тактовой частотой 3 ГГц),

так и широкий диапазон величин (64-битовые часы обнуляются один раз примерно в 195 лет на частоте 3 ГГц). Так что это самое гибкое представление времени, если вы, конечно, можете позволить себе 64-битное хранилище.

32-разрядные целочисленные часы

При измерении относительно коротких длительностей с высокой точностью можно обратиться к 32-разрядным целочисленным часам, измеряемым в машинных циклах. Например, чтобы профилировать производительность блока кода, мы могли бы написать что-то вроде следующего:

```
// Получаем снимок времени.
U64 begin_ticks = readHiResTimer();

// Это блок кода, производительность которого мы хотим измерять.
doSomething();
doSomethingElse();
nowReallyDoSomething();

// Измеряем продолжительность.
U64 end_ticks = readHiResTimer();
U32 dt_ticks = static_cast<U32>(end_ticks - begin_ticks);

// Теперь используем или кэшируем значение dt_ticks...
```

Обратите внимание на то, что мы все еще храним необработанные измерения времени в 64-битных целочисленных переменных. Только дельта времени хранится в 32-битной переменной. Это позволяет избежать потенциальных проблем с переносом на 32 бита. Например, если мы станем сокращать отдельные измерения времени до 32 битов каждый раз перед их вычитанием, то при `begin_ticks = 0x12345678FFFFFFB7` и `end_ticks = 0x1234567900000039` мы получим отрицательную дельту времени.

32-битные часы с плавающей точкой

Другим распространенным подходом является хранение относительно небольших временных разниц, измеряемых в секундах, в формате с плавающей точкой. Чтобы сделать это, мы просто умножаем длительность, измеренную в циклах ЦП, на тактовую частоту ЦП, то есть на циклы в секунду, например:

```
// Начнем с предположения, что имеем идеальное время кадра
// (30 кадров в секунду)
F32 dt_seconds = 1.0f / 30.0f;

// Заполнить pump, считывая текущее время.
U64 begin_ticks = readHiResTimer();

while (true) // главный игровой цикл
{
    runOneIterationOfGameLoop(dt_seconds);
```

```
    // Считаем текущее время еще раз и вычислим дельту.
    U64 end_ticks = readHiResTimer();

    // Проверим единицы: секунды = тики/(тики/секунды)
    dt_seconds = (F32)(end_ticks - begin_ticks)
              / (F32)getHiResTimerFrequency();

    // Используем end_ticks в качестве нового begin_ticks
    // для следующего кадра.
    begin_ticks = end_ticks;
}
```

Еще раз обратите внимание на то, что следует быть осторожными и необходимо сначала вычесть два 64-битных измерения времени, прежде чем преобразовать их в формат с плавающей точкой. Это гарантирует, что мы не сохраняем слишком большую величину в 32-битной переменной с плавающей точкой.

Ограничения значений с плавающей точкой

Напомню, что в 32-битном плавающем элементе IEEE 23 бита мантиссы динамически распределяются между целой и дробной частями значения с помощью показателя степени (см. подраздел 3.3.1). Малые величины требуют всего нескольких битов, оставляя много битов точности для дроби. Но как только значение переменной часов становится слишком большим, его целая часть съедает больше битов, оставляя меньше битов для дроби. В конце концов даже самые незначительные биты всей части становятся неявными нулями. Это означает, что мы должны быть осторожны при хранении длинных значений в переменной с плавающей точкой. Если будем отслеживать количество времени, прошедшее с момента запуска игры, часы с плавающей точкой в итоге станут неточными настолько, что с ними невозможно будет работать.

Часы с плавающей точкой обычно используются только для хранения относительно коротких временных дельт, измеряющих не более нескольких минут, а чаще всего один кадр или меньше. Если в игре применяются абсолютные часы с плавающей точкой, необходимо периодически сбрасывать их на ноль, чтобы избежать накопления больших величин.

Другие единицы измерения времени

Некоторые игровые движки позволяют указывать временные значения в определенных для игры целочисленных единицах (то есть нет требования формата с плавающей точкой), но они должны быть достаточно точными, чтобы быть полезными для широкого диапазона приложений внутри движка, и при этом достаточно большими, чтобы 32-разрядные часы не сбрасывались слишком часто. Один из распространенных вариантов — единица измерения времени 1/300 секунды. Эта единица работает хорошо, потому что она достаточно точна для многих целей, такие часы обнуляются лишь один раз каждые 165,7 дня и она даже кратна частоте

обновления NTSC и PAL. При 60 FPS длительность кадра будет 5 таких единиц, а при 50 FPS — 6 единиц.

Очевидно, что единица времени 1/300 секунды недостаточно точна для обработки тонких эффектов, таких как масштабирование времени анимации. (Если бы мы попытались замедлить анимацию с начальной скоростью 30 кадров в секунду до менее чем 1/10 от ее обычной скорости, мы бы потеряли точность!) Так что для многих целей все же лучше использовать единицы времени с плавающей точкой или машинные циклы. Но единицу времени 1/300 секунды можно эффективно применять, например, для указания того, сколько времени должно пройти между выстрелами из автоматического оружия, или как долго персонажу, контролируемому ИИ, следует ждать, прежде чем начать патрулирование, или как долго игрок может выжить, стоя в луже кислоты.

8.5.5. Работа с точками останова

Когда ваша игра достигает точки останова, ее цикл останавливается и отладчик перехватывает управление. Но если игра работает на том же компьютере, на котором и отладчик, тогда ЦП продолжит действовать, а часы реального времени все так же станут накапливать циклы. Может пройти большое количество реального времени, пока вы проверяете свой код в точке останова. Когда вы позволите программе продолжиться, это может привести к изменению времени кадра, которое станет равно многим секундам, минутам или даже часам!

Ясно, что, если мы допустим, чтобы такая огромная дельта времени передавалась подсистемам в движке, случится страшное. Если повезет, игра может продолжить функционировать должным образом после того, как переместится вперед на много секунд в одном кадре. В худшем случае она может просто аварийно завершиться.

Чтобы решить эту проблему, можно использовать простой подход. В основном игровом цикле, если мы когда-нибудь получим время кадра, превышающее некоторый заранее определенный верхний предел (например, 1 секунду), мы можем предположить, что только что возобновили выполнение после точки останова, и искусственно установить дельту времени 1/30 или 1/60 секунды (или независимо от целевой частоты кадров). По сути, игра будет заблокирована на один кадр, чтобы избежать резкого скачка измеренной длительности кадра:

```
// Начнем с предположения, что идеальная дельта dt (30 FPS).
F32 dt = 1.0f / 30.0f;

// Заполнить pump, считывая текущее время.
U64 begin_ticks = readHiResTimer();

while (true) // главный игровой цикл
{
    updateSubsystemA(dt);
    updateSubsystemB(dt);
```

```
    // ...
    renderScene();
    swapBuffers();

    // Считаем текущее время еще раз и вычислим предполагаемую дельту
    // следующего кадра.
    U64 end_ticks = readHiResTimer();

    dt = (F32)(end_ticks - begin_ticks)
      / (F32)getHiResTimerFrequency();

    // Если dt слишком велико, мы должны восстановить
    // исполнение после точки останова –
    // заблокировать значение для целевой скорости кадра:
    if (dt > 1.0f)
    {
        dt = 1.0f/30.0f;
    }

    // Используем end_ticks в качестве нового begin_ticks для следующего кадра
    begin_ticks = end_ticks;
}
```

8.6. Многопроцессорные игровые циклы

В главе 4 мы исследовали аппаратные средства параллельных вычислений, которые сейчас широко распространены на пользовательских компьютерах, мобильных устройствах и игровых приставках, и изучили механизм написания *параллельного* программного обеспечения, которое использует преимущества параллельных вычислительных ресурсов. В данном разделе обсудим различные способы применения этих знаний в *игровом цикле* игрового движка.

8.6.1. Разложение задания

Чтобы воспользоваться преимуществами оборудования для параллельных вычислений, нам необходимо *разделить* различные задачи, выполняемые во время всех итераций игрового цикла, на несколько подзадач, каждая из которых может выполняться параллельно. Этот акт разложения превращает наше программное обеспечение из *последовательного* в *параллельное*.

Есть много способов разложить программную систему для обеспечения параллелизма, но, как говорилось в подразделе 4.1.3, мы можем грубо разделить их все на две категории: *параллелизм задач* и *параллелизм данных*.

Параллелизм задач естественным образом подходит для ситуаций, в которых необходимо выполнить множество разных задач, что предпочтительно делать параллельно на нескольких ядрах. Например, мы можем попытаться выполнить

анимирование параллельно с обнаружением столкновений во время каждой итерации игрового цикла или представить примитивы графического процессора для визуализации кадра N параллельно с началом обновления состояния игрового мира для кадра $N + 1$.

Параллелизм данных лучше всего подходит для ситуаций, в которых *одно* и то же вычисление необходимо выполнять повторно для большого количества элементов данных. Графический процессор, вероятно, лучший пример параллелизма данных в действии: он выполняет миллионы вычислений для каждого пиксела и каждой вершины в каждом кадре, распределяя работу по большому количеству ядер обработки, работающих параллельно. Однако, как мы увидим в следующих разделах, параллелизм данных встречается не только на графическом процессоре — в течение игрового цикла процессор выполняет множество задач, которые также могут выиграть от параллелизма данных.

В следующих разделах мы рассмотрим ряд способов разделить работу, выполняемую в игровом цикле, часть которых реализуют параллелизм задач, другие полагаются на параллелизм данных. Поговорим о плюсах и минусах каждого подхода, а затем посмотрим, как универсальная система заданий может стать полезным инструментом для преобразования буквально любой рабочей нагрузки в параллельную операцию, которая может использовать преимущества аппаратного параллелизма.

8.6.2. Один поток на подсистему

Простой способ разложить игровой цикл для параллелизма — назначить отдельные подсистемы движка для работы в отдельных потоках. Например, движок рендеринга, симуляция столкновений и физики, конвейер анимации и звуковой движок могут быть отнесены каждый к собственному потоку. Главный поток будет контролировать и синхронизировать операции потоков вторичных подсистем, а также продолжать обрабатывать львиную долю высокоуровневой логики игры (основной игровой цикл). На аппаратной платформе с несколькими физическими процессорами такая архитектура позволит подсистемам с многопоточным механизмом работать параллельно друг другу и основному игровому циклу. Это простой пример *параллелизма задач* (рис. 8.4).

Существует ряд проблем, связанных с простым подходом, предусматривающим назначение каждой подсистемы движка своему потоку. Во-первых, количество подсистем движка, вероятно, не будет соответствовать количеству ядер на игровой платформе. В результате, скорее всего, потоков будет больше, чем ядер, и некоторым подсистемам потребуется совместно использовать ядро с помощью квантования времени.

Другая проблема состоит в том, что все подсистемы движка требуют различного количества вычислений на каждом кадре. Это означает, что в то время, как некоторые потоки и соответствующие им ядра ЦП интенсивно используются в каждом кадре, другие могут бездействовать на большей части кадра.

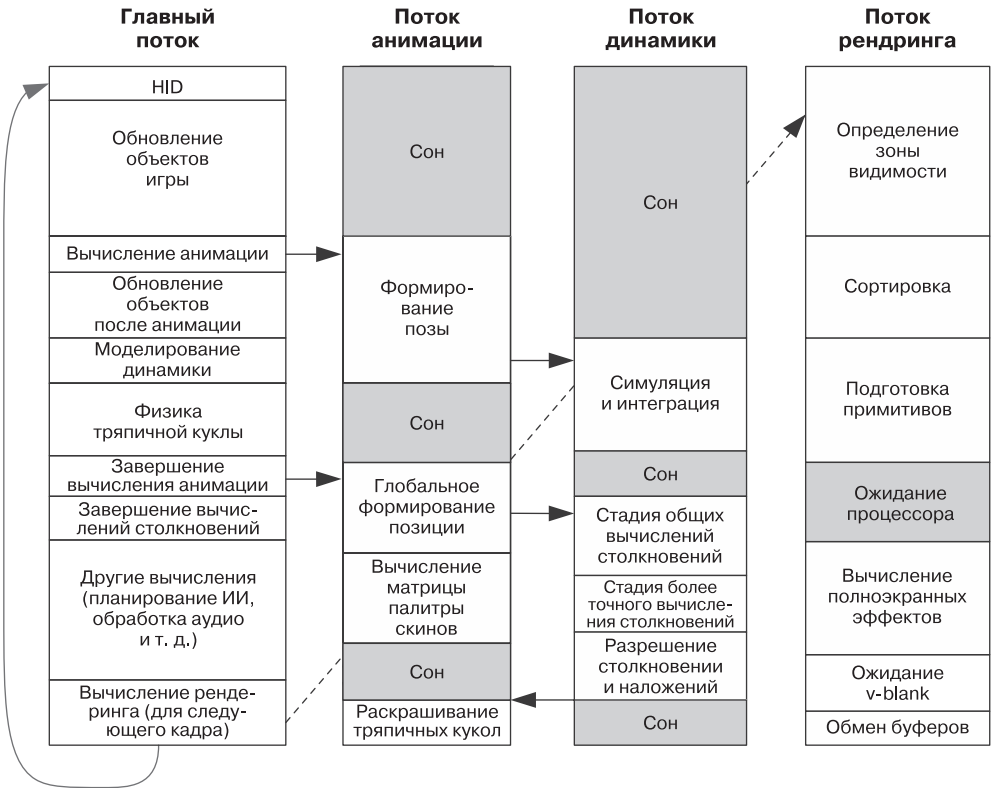


Рис. 8.4. По одному потоку для большинства подсистем движка

Еще одна проблема заключается в том, что некоторые подсистемы движка зависят от данных, предоставленных другими. Например, подсистемы рендеринга и аудио не могут начать выполнять работу для кадра *N*, пока системы анимации, столкновения и физики не завершат свои вычисления для этого кадра. Мы не можем запустить две подсистемы параллельно, если они зависят друг от друга.

Из-за этих проблем попытка назначить каждой подсистеме движка собственный поток оказывается непрактичной архитектурой. Мы должны стремиться к лучшему.

8.6.3. Разбиение/сборка

Многие из задач, выполняемых за одну итерацию игрового цикла, требуют большого объема данных. Например, нам может потребоваться обработать много запросов на кастование чар, смешать в одну анимацию множество мелких жестов или вычислить матрицы мирового пространства для каждого объекта в большой интерактивной сцене. Один из вариантов использовать преимущества аппаратных средств параллельных вычислений для выполнения подобных задач — подход «разделяй и властвуй». Вместо того чтобы пытаться обрабатывать 9000 выпущенных

лучей по одному на одном ядре процессора, мы можем разделить работу, скажем, на шесть партий по 1500 лучей, а затем выполнить по одной партии в каждом из шести[1] процессорных ядер на PS4 или Xbox One. Этот подход является формой *параллелизма данных*.

В терминологии распределенных систем это называется подходом *разбиения/сборки*, потому что единица работы делится на более мелкие субъединицы, распределяется по нескольким ядрам обработки для выполнения (разбиение) и, как только будут выполнены все вычисления, результаты объединяются или финализируются нужным способом (сборка).

Разбиение/сборка в игровом цикле

В контексте игрового цикла одна или несколько операций разбиения/сборки могут выполняться в разное время в течение одной итерации игрового цикла основным потоком игрового цикла (рис. 8.5).

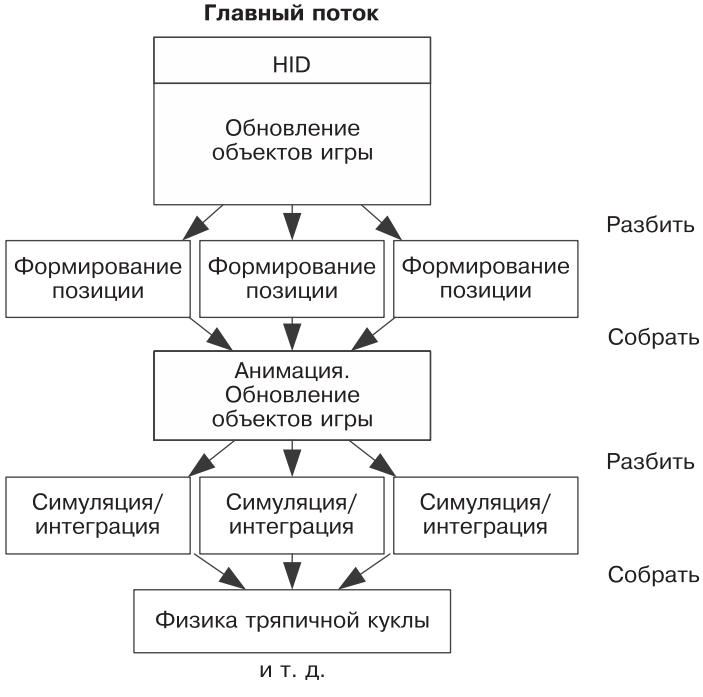


Рис. 8.5. Разбиение/сборка используется для распараллеливания выбранных частей игрового цикла, интенсивно задействующих процессор

[1] И PS4, и Xbox One позволяют разработчикам получить доступ к некоторым вычислительным мощностям седьмого ядра. Но они не могут задействовать всю его пропускную способность, поскольку ядро используется также операционной системой. Восьмое ядро в этих восьмиядерных машинах полностью недоступно. Так сделано из-за того, что при изготовлении любого процессора могут появиться неисправные ядра.

При наличии набора данных, содержащего N элементов данных, требующих обработки, главный поток разделит работу на m пакетов, каждый из которых содержит примерно N / m элементов. (Значение m, вероятно, будет определяться на основе количества доступных в системе ядер, хотя это может быть не так, если, например, мы хотим оставить некоторые ядра свободными для другой работы.) Главный поток затем породит m рабочих потоков и предоставит каждому начальный индекс элемента и количество, которое необходимо обработать, позволяя ему обрабатывать назначенное подмножество данных. Каждый рабочий поток может обновлять элементы данных сразу или (как правило, так лучше) выводить выходные данные в отдельный предварительно выделенный буфер (по одному на рабочий поток).

Успешно распределив рабочую нагрузку, главный поток сможет свободно выполнять другую полезную работу, ожидая, пока рабочие потоки завершат свои задачи.

В какой-то момент позже в кадре главный поток соберет результаты, дождавшись завершения работы всех остальных потоков и, возможно, используя функцию, такую как `pthread_join()`. Если все рабочие потоки завершены, эта функция вернется немедленно, но если какие-то из них все еще действуют, вызов приведет к тому, что главный поток перейдет в спящий режим.

Как только этап сбора будет завершен, мастер-поток может объединить результаты любым доступным способом. Например, следующим шагом после вычисления общей анимации может быть вычисление матриц скинов — этот шаг будет запущен только после того, как все потоки анимации завершат свою работу. Мы представили очень похожий пример в подразделе 4.4.6, когда рассматривали создание и объединение потоков.

SIMD для разбиения/сборки

В разделе 4.10 мы исследовали *векторизацию* циклов как метод использования параллелизма SIMD для повышения производительности работы с большими объемами данных. На самом деле это просто еще одна форма подхода разбиения/сборки, выполняемая на очень тонком уровне детализации. SIMD может применяться вместо разбиения/сборки на основе потоков, но, скорее всего, будет задействован вместе с ним (при этом каждый рабочий поток выполняет свою задачу с помощью векторизации).

Делаем разбиение/сборку более эффективной

Подход разбиения/сборки — это интуитивно понятный способ распределения работы с большими объемами данных между несколькими ядрами. Однако, как сказано ранее, этот метод параллелизма страдает от одной большой проблемы: порождение потоков требует больших затрат. Порождение потока включает в себя вызов ядра, как и объединение основного потока с рабочими. Само ядро выполняет довольно много работы, когда потоки приходят и уходят. Поэтому порождать кучу потоков каждый раз, когда требуется выполнить разбиение/сборку, нецелесообразно.

Мы могли бы снизить стоимость порождения потоков, используя пул предварительно порожденных потоков. Некоторые операционные системы, такие как Windows, предоставляют API для создания пула потоков и управления им (пример можно найти по адресу bit.ly/2H8ChIp). В отсутствие такого API вы всегда можете самостоятельно реализовать простой пул потоков, применяя *условные переменные*, *семафоры*, атомарные логические переменные или какой-то другой механизм для синхронизации действий потоков.

Мы бы хотели, чтобы наш пул потоков мог выполнять широкий спектр операций разбиения/сборки в рамках каждого кадра. Это означает, что мы больше не можем просто порождать набор потоков для каждого разбиения/сборки, чья входная функция выполняет одно конкретное вычисление. Вместо этого каждый поток в пуле должен быть способен выполнять любые операции разбиения/сборки, которые можно было бы выполнить во время любой итерации игрового цикла. Мы можем представить себе использование гигантского оператора `switch` для реализации данного подхода, но эта идея неуклюжая и уродливая, к тому же такой код будет сложно поддерживать. Действительно, нам нужна универсальная система для одновременного выполнения единиц работы через доступные ядра на целевом оборудовании.

8.6.4. Система заданий

Система заданий — это система общего назначения для выполнения произвольных единиц работы, обычно называемых заданиями, на нескольких ядрах. С ее помощью разработчик игры может подразделить каждую итерацию игрового цикла на довольно большое количество независимых заданий и направить их в систему заданий для выполнения. Система заданий поддерживает очередь отправленных заданий и планирует их распределение по доступным ядрам либо отправляя их для выполнения рабочими потоками в пуле потоков, либо другими способами. В некотором смысле система заданий похожа на обычное легковесное ядро операционной системы, за исключением того, что вместо планирования потоков для запуска на доступных ядрах она планирует задания.

Задания могут сколько угодно разбиваться на подзадачи, и в реальном игровом движке многие из них не зависят друг от друга. Как показано на рис. 8.6, эти факты помогают сделать эффективность использования процессора максимальной. Данная архитектура также естественным образом масштабируется до аппаратного обеспечения с любым количеством ядер ЦП.

Типичный интерфейс системы заданий

Типичная система заданий предоставляет простой и удобный API, очень похожий на API библиотеки потоков. Есть функция для порождения задания (эквивалент `pthread_create()`, часто называемая *вбросом задания*), функция, позволяющая одной задаче ожидать завершения одного или нескольких других заданий (эквивалент `pthread_join()`), и, возможно, способ самозавершения задания

досрочно — до возврата из функции точки входа. Система заданий также должна обеспечивать спин-блокировки мьютексов для выполнения критических операций атомарным способом. Это позволит перевести работу в спящий режим и разбудить ее с помощью переменных состояния или подобного механизма.

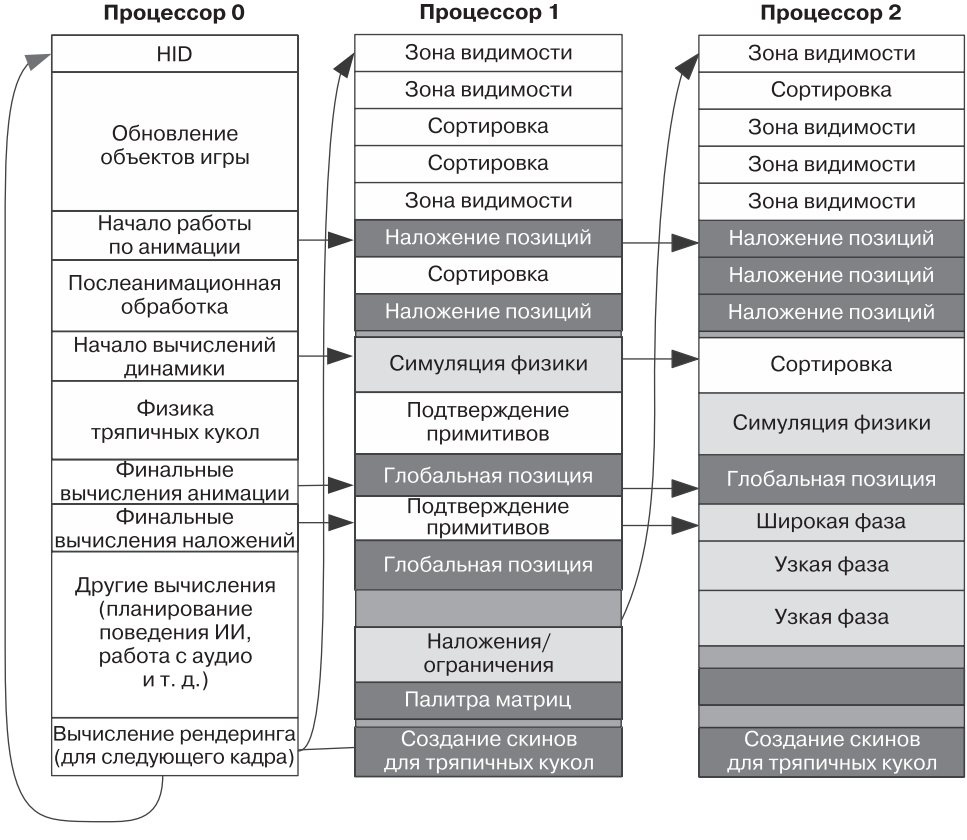


Рис. 8.6. В модели заданий работа разбивается на мелкие фрагменты, которые могут быть получены любым доступным процессором. Это может помочь довести загрузку процессора до максимальной, обеспечивая повышенную гибкость основного игрового цикла

Чтобы начать работу, нужно сообщить системе заданий, *какое* задание необходимо выполнять и *как* это делать. Обычно эта информация передается в функцию `KickJob()` через небольшую структуру данных, которую мы будем называть *объявлением задания*.

Как минимум объявление задания должно содержать указатель на функцию точки входа в задачу. Также важно иметь возможность передавать в эту функцию произвольные входные параметры. Сделать это можно различными способами, но самое простое — предоставить один параметр типа `uintptr_t`, который будет передан функции точки входа, когда задание действительно выполняется. Это позволяет без особых хлопот передавать простую информацию, например одно булево

или целое число, на задачу. Но поскольку указатель может быть безопасно назначен `uintptr_t`, мы можем использовать этот параметр задания также для передачи указателя на произвольную структуру данных, которая сама содержит любые входные параметры, которые могут потребоваться для задания.

Система заданий может предоставлять механизм приоритетов, как и большинство библиотек потоков. В этом случае приоритет может быть включен в объявление задачи.

Вот пример простого объявления задания вместе с простым API системы заданий:

```
namespace job
{
    // сигнатура всех точек входа в задание
    typedef void EntryPoint(uintptr_t param);

    // допустимые приоритеты
    enum class Priority
    {
        LOW, NORMAL, HIGH, CRITICAL
    };

    // счетчик (реализация не показана)
    struct Counter  ... ;
    Counter* AllocCounter();
    void FreeCounter(Counter* pCounter);

    // простое объявление задания
    struct Declaration
    {
        EntryPoint*     m_pEntryPoint;
        uintptr_t       m_param;
        Priority        m_priority;
        Counter*        m_pCounter;
    };

    // начать работу
    void KickJob(const Declaration& decl);
    void KickJobs(int count, const Declaration aDecl[]);

    // ожидание завершения задания (когда счетчик станет равным нулю)
    void WaitForCounter(Counter* pCounter);

    // начать другие задания и ждать их завершения
    void KickJobAndWait(const Declaration& decl);
    void KickJobsAndWait(int count, const Declaration aDecl[]);
}
```

Вы заметили, что здесь есть скрытый тип, называемый `job::Counter`? Счетчики позволяют одному заданию уходить в спящий режим и ждать, пока одно или несколько других заданий не завершатся. Мы обсудим счетчики заданий далее.

Простая система заданий на основе пула потоков

Как уже упоминалось, можно создать систему заданий на основе пула рабочих потоков. Рекомендуется создавать один поток для каждого процессора, присутствующего на целевой машине, и использовать привязку каждого потока к определенному ядру. Каждый рабочий поток находится в бесконечном цикле обработки запросов заданий, которые предоставляют ему другие потоки и/или другие задания. В верхней части этого бесконечного цикла рабочий поток переходит в спящий режим (возможно, через переменную условия), ожидая доступности запроса на выполнение задания. Когда запрос получен, рабочий поток просыпается, вызывает функцию точки входа в задание и передает ему входной параметр из объявления задания. Возвращение функции точки входа означает, что работа завершена, поэтому рабочий поток возвращается к вершине своего бесконечного цикла, чтобы выполнить больше заданий. Если нельзя получить доступ ни к одному из заданий, поток возвращается в режим ожидания другого запроса на выполнение задания.

Вот как может выглядеть рабочий поток изнутри (пожалуйста, имейте в виду, что это не полная реализация — она приведена здесь только для демонстрации):

```
namespace job
{
    void* JobWorkerThread(void*)
    {
        // продолжаем выполнять задания...
        while (true)
        {
            Declaration declCopy;

            // ждем, пока работа станет доступной
            pthread_mutex_lock(&g_mutex);
            while (!g_ready)
            {
                pthread_cond_wait(&g_jobCv, &g_mutex);
            }

            // копируем JobDeclaration локально
            // и снимаем блокировки мьютекса
            declCopy = GetNextJobFromQueue();
            pthread_mutex_unlock(&g_mutex);

            // запускаем задание
            declCopy.m_pEntryPoint(declCopy.m_param);

            // Работа выполнена! Очистить и повторить...
        }
    }
}
```

Ограничения потоковых заданий

Представим, что мы пишем задание, которое обновляет состояние неигрового персонажа, управляемого ИИ. Функция точки входа в задание может выглядеть примерно так:

```
void NpcThinkJob(uintparam_t param)
{
    Npc* pNpc = reinterpret_cast<Npc*>(param);

    pNpc->StartThinking();
    pNpc->DoSomeMoreUpdating();
    // ...

    // теперь кастуем луч, чтобы увидеть, нацелены ли мы
    // на что-нибудь интересное, — это включает в себя
    // запуск другой работы, которую будет выполнять
    // другой код (рабочий поток)
    RayCastHandle hRayCast = CastGunAimRay(pNpc);

    // результаты каста луча не будут
    // готовы до конца этого кадра, так что пока ложимся спать
    WaitForRayCast(hRayCast);

    // сон...

    // Просыпайтесь!

    // теперь стреляем из оружия, но только если луч указывает,
    // что мы нацелены на врага
    pNpc->TryFireWeaponAtTarget(hRayCast);

    // ...
}
```

Эта задача кажется довольно простой: она выполняет некоторые обновления, запускает задачу по кастованию луча (на другом рабочем потоке/ядре), чтобы определить, на какой объект NPC (неигровой персонаж) направляет свое оружие. Затем NPC стреляет, но только в том случае, если луч сообщает, что в его поле каста находится враг.

К сожалению, такая задача не сработает, если мы попытаемся запустить ее в рамках простой системы заданий, которую описали в предыдущем разделе. Проблема заключается в вызове `WaitForRayCast()`. В нашей простой системе заданий на основе пула потоков каждое задание должно *завершаться, если оно начинает выполняться*. Оно не может «уйти спать» в ожидании результатов кастования лучей, позволяя другим заданиям выполняться в рабочем потоке, и «проснуться», когда эти результаты будут готовы.

Данное ограничение возникает потому, что в простой системе каждое выполняемое задание использует тот же стек вызовов, что и рабочий поток, который его

вызвал. Чтобы перевести задание в спящий режим, нужно было бы эффективно *переключать контекст* на другую задачу. Сюда будут входить сохранение стека вызовов и регистров исходящего задания, а затем замена стека вызовов рабочего потока на стек вызовов входящего задания. Нет простого способа сделать это с помощью такого простого метода выполнения заданий.

Задания в виде корутин

Одним из способов преодоления этой проблемы было бы переключение с системы заданий на основе пула потоков на систему, основанную на *корутинах*. Вспомните подраздел 4.4.8, где говорится, что корутина обладает одним важным качеством, которого нет у обычных потоков, — способностью *уступить* другой корутине на середине выполнения и продолжить с того места, где остановилась, когда другая корутина возвратит контроль. Корутины могут уступать друг другу таким образом, потому что реализация фактически меняет местами стеки вызовов исходящих и входящих корутин в потоке, в котором они работают. Таким образом, в отличие от задания, основанного исключительно на потоках, задание на основе корутин может эффективно «уходить спать» и запускать другие задания, ожидая завершения операции, такой как кастование лучей.

Задания как фиберы

Другой способ разрешить заданиям переходить в спящий режим и уступать друг другу — это реализовывать их с помощью *фиберов*, а не потоков. Напомню из подраздела 4.4.7, что ключевое различие между фиберами и потоками заключается в том, что переключение контекста между фиберами всегда *кооперативное*, а не *вытесняющее*. Система на основе фиберов начинается с преобразования одного из ее потоков в фибер. Поток будет продолжать запускать этот фибер до тех пор, пока явно не вызовет `SwitchToFiber()`, чтобы явно *передать* управление другому фиберу. Как и в случае с корутинами, весь стек вызовов фибера сохраняется при каждом переключении контекста на другой фибер. Фиберы могут даже мигрировать из одного потока в другой. Именно поэтому они хорошо подходят для реализации системы заданий. Система заданий Naughty Dog основана на фиберах.

Счетчики заданий

Если мы реализуем систему заданий с корутинами или фиберами, у нас появится возможность перевести задание в спящий режим (сохраняя контекст его выполнения) и в будущем восстановить его (восстановив, соответственно, контекст выполнения). Это, в свою очередь, позволит реализовать функцию *объединения* для системы заданий — функцию, которая заставляет вызывающее задание переходить в спящий режим, ожидая завершения выполнения одного или нескольких других заданий. Такая функция была бы приблизительно эквивалентна `pthread_join()` в библиотеке потоков POSIX или `WaitForSingleObject()` в Windows.

Один из способов реализовать данный подход — связать *дескриптор* с каждым заданием так же, как потоки связаны с дескрипторами в большинстве библиотек

потоков. Тогда ожидание задания будет равносильно вызову функции `job::Join()`, передающей дескриптор задания, завершения которого вы ожидаете.

Недостатком подхода, основанного на дескрипторе, является то, что он плохо масштабируется для ожидания большого количества заданий. Кроме того, чтобы дождаться завершения определенного задания, нам нужно периодически опрашивать всю систему, чтобы проверить состояние всех заданий в ней. Такой опрос будет тратить ценные циклы процессора. По этим причинам API системы заданий, представленный ранее, вводит концепцию *счетчика*, который действует почти как семафор, только наоборот. Всякий раз, когда задание запускается, его можно при желании связать со счетчиком, предоставленным ему через `job::Statement`. Действие запуска задания увеличивает счетчик, а когда задание завершается, счетчик уменьшается. Ожидание завершения нескольких заданий включает в себя простое включение их всех в один счетчик, а затем ожидание того, что этот счетчик достигнет нуля, то есть все задания завершат работу. Ожидание достижения счетчиком нуля намного эффективнее, чем опрос отдельных заданий, потому что проверка может быть выполнена в тот момент, когда счетчик уменьшается. Таким образом, система на основе счетчиков может помочь увеличить производительность. Такие счетчики используются в системе заданий Naughty Dog.

Примитивы синхронизации заданий

Любая параллельная программа требует механизма для выполнения *атомарных операций*, и система заданий не исключение. Как библиотека потоков предоставляет набор *примитивов синхронизации потоков*, таких как мьютексы, условные переменные и семафоры, так и система заданий должна предоставлять коллекцию *примитивов синхронизации заданий*.

Реализация примитивов синхронизации заданий несколько варьируется в зависимости от того, как на самом деле реализована система заданий. Но обычно они *не* просто обертки вокруг примитивов синхронизации потоков ядра. Чтобы понять, почему так, рассмотрим, что делает мьютекс ОС: он переводит поток в режим сна, когда блокировка, которую он пытается получить, уже удерживается другим потоком. Если бы мы внедрили систему заданий как пул потоков, то ожидание мультимедиа в задании привело бы к тому, что в спящий режим перешел бы *весь рабочий поток*, а не только одно задание, которое ожидает блокировки. Очевидно, что это создаст серьезную проблему, потому что никакие задания не смогут выполняться на ядре этого потока, пока он не восстановится. Такая система, скорее всего, будет чревата проблемами взаимных блокировок.

Чтобы решить эту проблему, задания могут использовать *спин-блокировки* вместо мьютексов ОС. Этот подход хорошо работает до тех пор, пока между потоками не так много конфликтов блокировок, иначе никакое задание никогда не будет выполняться, а станет ждать каждой блокировки, пытаясь ее получить. Система работы Naughty Dog задействует спин-блокировки для большинства своих задач.

Однако иногда при выполнении задания могут возникать конфликтные ситуации. Хорошо спроектированная система заданий справляется с этим с помощью

специального механизма мьютекса, который может перевести задание в спящий режим, пока тот ожидает определенного ресурса. Такой мьютекс может начинаться с ожидания занятости, когда блокировка не может быть получена. Если после короткого тайм-аута блокировка все еще недоступна, мьютекс может уступить корутину или фибер другому ожидающему заданию, тем самым переводя ожидающее задание в спящий режим. Подобно тому как ядро отслеживает все спящие потоки, ожидающие мьютекс, наша система заданий должна отслеживать все спящие задания, чтобы разбудить их, когда их мьютексы освободятся.

Визуализация заданий и инструменты профилирования

Как только вы начинаете использовать систему заданий, график запущенных заданий и их зависимостей очень быстро становится большим и сложным. Хорошая идея — предоставить инструменты визуализации и профилирования для любой системы заданий.

Например, система заданий Naughty Dog предлагает способ визуализации, подобный показанному на рис. 8.7. На этом экране вы видите каждое из семи ядер (плюс графический процессор), они перечислены по вертикали вдоль левого края. Время увеличивается слева направо, и каждый логический кадр обозначен вертикальными маркерами. Вдоль шкалы времени каждого ядра тонкие блоки представляют различные задания, выполняемые в течение данного кадра. Под каждым заданием располагаются дополнительные ряды тонких прямоугольников — они представляют стек вызовов задания (какие функции вызывались, и сколько времени потребовалось каждой на выполнение).

Рис. 8.7. Система заданий, используемая в Uncharted: The Lost Legacy и других играх PS4 для Naughty Dog, предлагает инструмент визуализации, который показывает, какие задания выполнялись на каждом ядре в течение определенного кадра. Время увеличивается слева направо. Рабочие места представлены тонкими прямоугольниками вдоль шкалы времени каждого ядра. Функции, вызываемые каждым заданием, отображаются под ним в виде дополнительных тонких прямоугольников

Цвет задания определен в его функции, поэтому пользователи могут быстро найти конкретное задание. Например, мы ищем кастование лучей, потому что оно занимает особенно много времени. Если задания с кастованием лучей окрашены в красный цвет, мы можем визуально отсканировать на экране все красные задания, которые шире, чем хотелось бы. Нажатие на задание приводит к тому, что все другие задания, не относящиеся к тому же типу, отображаются серым цветом. Это позволяет легко увидеть все задания выбранного типа. Кроме того, при нажатии на задание тонкие линии связывают его с заданиями, которые оно порождало, и с заданием, запустившим его. При наведении указателя мыши на задание или на одну из функций в его стеке вызовов

появляется текст с названием задания или функции и временем выполнения в миллисекундах.

Еще одна очень полезная функция системы заданий — то, что я назову *ловушкой профилирования*. Скажем, в большинстве случаев скорость игры 30 FPS, но время от времени она падает до 24 FPS. Мы можем установить ловушку для любого кадра, который занимает более 35 миллисекунд. Затем играем в игру нормально. Как только обнаруживается кадр, занимающий более 35 миллисекунд, игра автоматически приостанавливается системой ловушек и на экран выводится информация профилирования. Затем можно проанализировать задания, которые запускались в этом кадре, чтобы найти виновника (-ов) замедления.

Система заданий Naughty Dog

Система заданий, используемая Naughty Dog в играх *The Last of Us: Remastered*, *Uncharted 4: A Thief's End* и *Uncharted: The Lost Legacy*, в значительной степени соответствует гипотетическим системам заданий, которые мы обсуждали до сих пор. Она основана на фиберах, а не на потоках или корутинах. Она задействует спин-блокировки, а также предоставляет специальный мьютекс заданий, который может перевести задание в спящий режим, пока оно ожидает блокировки. Для реализации операции *объединения* применяются счетчики, а не дескрипторы заданий.

Посмотрим, как работает система заданий на основе фиберов, используемая в движке Naughty Dog. Когда система загружается в первый раз, основной поток преобразуется в фибер, чтобы включить фиберы в процессе в целом. Затем создаются рабочие потоки, по одному на каждое из семи ядер, доступных разработчикам на PS4. Каждый из этих потоков привязан к одному ядру с помощью настроек привязки к процессору, поэтому мы можем считать рабочие потоки и их ядра примерно синонимами (хотя в действительности другие потоки с более высоким приоритетом иногда прерывают рабочие потоки на очень короткие периоды во время кадра). Создание фибера на PS4 происходит медленно, поэтому заранее создается пул фиберов вместе с блоками памяти, которые служат стеком вызовов каждого фибера.

Когда задания запускаются, их объявления помещаются в очередь. Поскольку ядра/рабочие потоки становятся свободными (по мере завершения заданий), новые задания извлекаются из этой очереди и выполняются. Выполняемое задание также может добавить больше заданий в их очередь.

Для выполнения задания из пула фиберов извлекается свободный фибер и рабочий поток выполняет `SwitchToFiber()`, чтобы запустить задание. Когда оно возвращается из функции точки входа или иным образом самостоятельно завершается, итоговым действием задания является выполнение `SwitchToFiber()` к фиберу самой системы заданий. Затем фибер выбирает другую работу из очереди, и процесс повторяется до бесконечности.

Ожидая значение счетчика, задание переводится в спящий режим, а его фибер (контекст исполнения) помещается в список ожидания вместе со счетчиком,

которого оно ожидает. Когда этот счетчик достигает нуля, задание возвращается в рабочее состояние и может продолжиться с того места, на котором остановилось. Спящие и бодрствующие задания снова реализуются путем вызова `SwitchToFiber()` между фибером задания и фибером управления системой заданий в каждом ядре или рабочем потоке.

Чтобы хорошо разобраться в том, как была построена система заданий Naughty Dog и почему это было сделано таким образом, ознакомьтесь с превосходным докладом Кристиана Гирлинга на GDC 2015 под названием «Распараллеливание механизма Naughty Dog», который доступен по адресу bit.ly/2H6v0J4. Слайды Кристиана находятся по адресу bit.ly/2ETr5x9.

9 Устройства HID

Игры — интерактивные компьютерные симуляции, поэтому игроку нужно как-то с ними взаимодействовать. Для этого существуют так называемые *устройства с человеческим интерфейсом* (human interface devices, HID): джойстики, клавиатуры и мыши, трекболы, пульты дистанционного управления (как в Wii) и специальные устройства ввода вроде руля, удочки, танцевального мата или даже электрогитары. В этой главе мы обсудим, как игровые движки обычно считывают, обрабатывают и используют ввод с устройств HID. Вы также увидите, как вывод этих устройств обеспечивает обратную связь с игроком.

9.1. Виды HID-устройств

Игровые HID-устройства чрезвычайно разнообразны. Консоли, такие как Xbox 360 и PS3, поставляются вместе с джойстиками (рис. 9.1, 9.2). Консоль Wii от Nintendo славится своим уникальным игровым контроллером Wii Remote, который часто называют Wiimote (рис. 9.3). Компании Nintendo также удалось создать инновационный гибрид между контроллером и полумобильным игровым устройством — Wii U (рис. 9.4). В играх на ПК обычно используется либо комбинация из клавиатуры и мыши, либо джойстик (компания Microsoft сделала так, что контроллер для Xbox 360 можно подключать и к компьютерам с Windows/DirectX). Аркадные автоматы содержат один или несколько элементов управления, таких как джойстики, различные кнопки, трекболы, рули и т. д. (рис. 9.5). Их устройства ввода обычно предназначены для конкретной игры, хотя производители часто оснащают ими разные модели автоматов.

Рис. 9.1. Стандартные джойстики для консолей Xbox 360 и PlayStation 3

Рис. 9.2. Контроллер DualShock 4 для PlayStation 4

Рис. 9.3. Инновационный контроллер Wii Remote для Nintendo Wii

Рис. 9.4. Контроллер Wii U от Nintendo

Рис. 9.5. Кнопки и джойстики для аркадной игры Mortal Kombat II от Midway

На консольных платформах обычно доступны специализированные устройства ввода и адаптеры в дополнение к стандартным, таким как джойстики. Например, для игр из цикла *Guitar Hero* предусмотрены гитара и барабан; для гоночных игр можно докупить руль; в играх вроде *Dance Revolution* можно использовать специальный танцевальный коврик. Некоторые из этих устройств показаны на рис. 9.6.

Рис. 9.6. К консолям можно подключать множество узкоспециализированных устройств

Nintendo Wiimote на сегодняшний день является одним из самых гибких устройств ввода на рынке. В связи с этим ему часто находят новое применение, вместо того чтобы заменять его другим контроллером. Например, игра *Mario Kart Wii* поставляется с пластиковым адаптером руля, в который можно вставить Wiimote (рис. 9.7).

Рис. 9.7. Адаптер руля для Nintendo Wii

9.2. Взаимодействие с HID-устройствами

Все HID-устройства передают ввод игровому ПО, а некоторые умеют принимать вывод из игры для обеспечения обратной связи с игроком. Для этого у них есть разнообразные средства вывода. Игра считывает и записывает ввод и вывод HID-устройств разными способами в зависимости от конструкции того или иного устройства.

9.2.1. Опрашивание

Ввод некоторых простых устройств, таких как геймпады и старомодные джойстики, считывается путем периодического *опрашивания* оборудования (обычно в каждой итерации игрового цикла). Для этого состояние устройства запрашивается вручную: либо напрямую из его регистров, либо чтением порта ввода/вывода, отображенного в память, либо с помощью высокоуровневого программного интерфейса, который, в свою очередь, обращается к соответствующим регистрам или портам ввода/вывода. Точно так же вывод может быть передан HID-устройству либо за счет записи в специальные регистры или адреса ввода/вывода, отображенные в память, либо с использованием более высокоуровневого API, который делает всю грязную работу за нас.

Хорошим примером простого опрашивающего механизма является API XInput от Microsoft, предназначенный для работы с контроллерами Xbox 360 на одноименной консоли и на платформе Windows. В каждом кадре игра вызывает функцию `XInputGetState()`, которая обращается к оборудованию и/или драйверам, корректно считывает данные и упаковывает их в удобный формат для дальнейшего использования в коде программы. Она возвращает указатель на структуру `XINPUT_STATE`, которая, в свою очередь, содержит экземпляр структуры `XINPUT_GAMEPAD`. Последняя описывает состояние всех элементов управления (кнопок, аналоговых стиков и триггеров), доступных на устройстве.

9.2.2. Прерывания

Некоторые HID-устройства передают данные игровому движку только в случае изменения состояния контроллера. Например, мышь большую часть времени просто лежит на коврике. Ей нет никакого смысла слать компьютеру непрерывный поток данных в периоды бездействия, информация должна передаваться только во время движения или нажатия/отпускания клавиш.

Такого рода устройства обычно взаимодействуют с компьютером посредством *аппаратных прерываний*. Прерывание — это электронный сигнал, генерируемый оборудованием, он заставляет центральный процессор временно приостановить работу главной программы и выполнить небольшой фрагмент кода — так называемую *процедуру обслуживания прерываний* (interrupt service routine, ISR). Прерывания имеют множество способов применения, но в случае с HID-устройствами код ISR обычно считывает состояние устройства, сохраняет его для дальнейшей обработки и затем уступает поток выполнения главной программе. Игровой движок может подобрать эти данные в любое удобное для него время.

9.2.3. Беспроводные устройства

Ввод и вывод Bluetooth-устройств, таких как Wiimote, DualShock 3 и беспроводной контроллер Xbox 360, нельзя считывать или записывать путем обращения к регистрам или портам ввода/вывода, отображенным в память. Вместо этого общение с устройством происходит по протоколу Bluetooth. Программа может запросить ввод (например, состояние кнопок) или послать вывод (вибрацию или аудиопоток). Такое взаимодействие зачастую происходит в отдельном от основного игрового цикла потоке или по крайней мере инкапсулируется с помощью простого интерфейса, с которым можно работать в главном цикле. Поэтому, с точки зрения разработчика игр, состояние Bluetooth-устройства может ничем не отличаться от состояния традиционного опрашиваемого оборудования.

9.3. Виды элементов управления

Игровые HID-устройства могут существенно различаться своей конструкцией и компоновкой, однако большинство элементов управления, которые они предоставляют, можно разделить всего на несколько категорий. Каждая категория будет подробно рассмотрена в дальнейшем.

9.3.1. Цифровые кнопки

Почти у любого HID-устройства есть как минимум пара *цифровых кнопок*. Это кнопки, которые могут находиться в одном из двух состояний — *нажатом* и *ненажатом*. Игровые разработчики часто называют эти состояния *«вниз»* и соответственно *«вверх»*.

С точки зрения электротехники электрическая цепь с переключателем может быть *замкнутой* (когда по ней проходит ток) или *разомкнутой* (когда ток отсутствует, цепь обладает *бесконечным* сопротивлением). То, какому состоянию соответствует *замкнутость*, нажатому или ненажатому, зависит от реализации. Если переключатель *нормально разомкнут*, ненажатая кнопка («вверх») оставляет цепь *разомкнутой*, а ее нажатие приводит к *замыканию*. Если переключатель *нормально замкнут*, все наоборот: нажатие кнопки *размыкает* цепь.

На программном уровне состояние цифровых кнопок (нажатое и ненажатое) в основном представлено одним битом. Как правило, 0 обозначает ненажатую кнопку («вверх»), а 1 — нажатую («вниз»). Но опять же эти значения могут иметь противоположный смысл — все зависит от конкретной микросхемы и решений, принятых авторами драйвера устройства.

Довольно часто состояние всех кнопок на устройстве упаковывают в единое целочисленное значение. Например, в API XInput от Microsoft состояние джойстика Xbox 360 возвращается в виде структуры под названием `XINPUT_GAMEPAD`, которая показана далее:

```
typedef struct _XINPUT_GAMEPAD
{
    WORD  wButtons;
    BYTE  bLeftTrigger;
    BYTE  bRightTrigger;
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

Эта структура содержит беззнаковую целочисленную переменную `wButtons` размером 16 бит (ее еще называют *машинным словом*), которая хранит состояние всех кнопок. Следующие маски определяют, к какому биту в этом значении относится та или иная кнопка (обратите внимание на то, что биты 10 и 11 не используются):

```
#define XINPUT_GAMEPAD_DPAD_UP          0x0001 // бит 0
#define XINPUT_GAMEPAD_DPAD_DOWN        0x0002 // бит 1
#define XINPUT_GAMEPAD_DPAD_LEFT        0x0004 // бит 2
#define XINPUT_GAMEPAD_DPAD_RIGHT       0x0008 // бит 3
#define XINPUT_GAMEPAD_START            0x0010 // бит 4
#define XINPUT_GAMEPAD_BACK             0x0020 // бит 5
#define XINPUT_GAMEPAD_LEFT_THUMB       0x0040 // бит 6
#define XINPUT_GAMEPAD_RIGHT_THUMB      0x0080 // бит 7
#define XINPUT_GAMEPAD_LEFT_SHOULDER    0x0100 // бит 8
#define XINPUT_GAMEPAD_RIGHT_SHOULDER   0x0200 // бит 9
#define XINPUT_GAMEPAD_A                0x1000 // бит 12
#define XINPUT_GAMEPAD_B                0x2000 // бит 13
#define XINPUT_GAMEPAD_X                0x4000 // бит 14
#define XINPUT_GAMEPAD_Y                0x8000 // бит 15
```

Чтобы прочитать состояние отдельной кнопки, нужно на значение `wButtons` наложить подходящую битовую маску (в C/C++ для этого предусмотрен битовый оператор И [&]) и проверить, равен ли результат нулю. Например, следующим образом можно определить, нажата ли кнопка A:

```
bool IsButtonADown(const XINPUT_GAMEPAD& pad)
{
    // Маскируем все биты, кроме 12-го (кнопка A).
    return ((pad.wButtons & XINPUT_GAMEPAD_A) != 0);
}
```

9.3.2. Аналоговые оси и кнопки

Аналоговый ввод может принимать значения в некотором диапазоне (а не только 0 или 1) и часто используется для представления степени нажатия триггера или позиции аналогового джойстика в двух измерениях (который использует два аналоговых входа — для оси *X* и для оси *Y* (рис. 9.8). В связи с таким применением аналоговый ввод иногда называют *аналоговой осью* или просто *осью*.

В некоторых устройствах аналоговыми являются и определенные кнопки, то есть игра может понять, насколько сильно игрок их нажимает. Однако сигналы, генерируемые аналоговыми кнопками, слишком «грязные», что делает их не очень практичными. Игры, в которых удается эффективно использовать подобный ввод, — редкость. В качестве удачного примера можно привести *Metal Gear Solid 2* для PS2. В этой игре данные аналоговых кнопок, зависящие от силы нажатия, применялись в режиме прицеливания: если кнопку X отпускали быстро, происходил выстрел, если медленно — выстрел отменялся. Это может пригодиться в играх, в которых раскрывать врагам свое присутствие следует только в случае крайней необходимости.

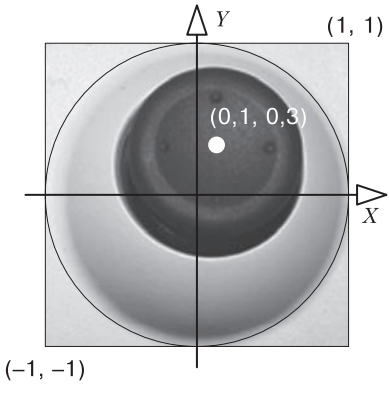


Рис. 9.8. Два аналоговых ввода можно применять для описания отклонения стика относительно осей *X* и *Y*

Строго говоря, аналоговый ввод перестает быть таковым к моменту, когда доходит до игрового движка. Аналоговый входящий сигнал обычно *оцифровывают*, то есть квантуют и представляют на программном уровне в виде целых чисел. Например, если это целое 16-битное значение со знаком, оно будет находиться в диапазоне от –32 768 до 32 767. Иногда аналоговый ввод преобразуют в числа с плавающей запятой, значения которых могут варьироваться в пределах, скажем, от –1 до 1. Но, как мы узнали в подразделе 3.3.1, числа с плавающей запятой — это тоже, в сущности, квантованные цифровые значения.

Если взглянуть на определение `XINPUT_GAMEPAD`, копия которого показана далее, можно увидеть, что в Microsoft решили выполнить отклонения левого и правого сти-

ков в контроллере Xbox 360 с помощью 16-битных целых чисел со знаком (`sThumbLX` и `sThumbLY` — для левого, `sThumbRX` и `sThumbRY` — для правого). Это соответствует значениям в диапазоне от –32 768 (влево или вниз) до 32 767 (вправо или вверх). Но для представления позиции триггеров в Microsoft стали использовать восьмибитные беззнаковые целые числа (`bLeftTrigger` и `bRightTrigger` соответственно). Эти значения ввода находятся в диапазоне от 0 (без нажатия) до 255 (полное нажатие). Разные игровые платформы по-разному представляют свои аналоговые оси.

```
typedef struct _XINPUT_GAMEPAD
{
    WORD  wButtons;

    // 8-битное беззнаковое.
    BYTE  bLeftTrigger;
    BYTE  bRightTrigger;

    // 16-битное со знаком.
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

9.3.3. Относительные оси

Позиции аналоговых кнопок, триггеров и стиков являются *абсолютными* — это означает, что мы точно знаем, где находится нулевое значение. Однако в некоторых устройствах ввод может быть *относительным*. В таких случаях у нас нет четкого положения, в котором значение ввода должно быть равно нулю. Вместо этого ноль указывает на то, что позиция устройства не изменилась, а любые другие значения представляют собой отличие от предыдущей считанной величины. В качестве примеров можно привести колесо прокрутки мыши и трекбол.

9.3.4. Акселерометр

Джойстики DualShock для PlayStation и контроллеры Nintendo Wiimote содержат датчики ускорения (акселерометры). Такие устройства способны определять ускорение по трем главным осям — *X*, *Y* и *Z* (рис. 9.9). Это *относительный* аналоговый ввод, во многом похожий на двухмерные показания мыши. Когда контроллер не ускоряется, данные значения равны нулю, в противном случае они измеряют ускорение в пределах ±3*g* по каждой оси с квантованием результатов в виде трех восьмибитных целых чисел со знаком, по одному для *X*, *Y* и *Z*.

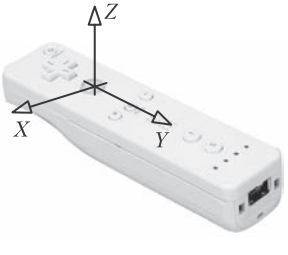


Рис. 9.9. Оси акселерометра в Wiimote

9.3.5. Положение в трехмерном пространстве при использовании Wiimote или DualShock

Некоторые игры для Wii и PS3 задействуют три акселерометра в Wiimote или джойстике DualShock для определения приблизительного положения контроллера в руках игрока. Например, в *Super Mario Galaxy* Марио запрыгивает на большой шар и катит его ногами. Для управления в этом режиме Wiimote следует держать инфракрасным сенсором вверх. Наклоняя контроллер влево, вправо, вперед или назад, мы ускоряем шар в соответствующем направлении.

С помощью этих трех акселерометров можно определять положение Wiimote и джойстика DualShock, так как мы находимся на поверхности Земли, где сила тяжести создает постоянное ускорение $1g$ ($\approx 9{,}8$ м/с^2). Если повернуть контроллер так, чтобы инфракрасный сенсор был направлен в сторону телевизора, и держать идеально ровно, то вертикальное *(z)* ускорение должно быть примерно $-1g$.

Если повернуть контроллер инфракрасным сенсором вверх, можно ожидать нулевое ускорение по оси Z и $+1g$ по оси Y, так как теперь сила тяжести направлена строго вдоль нее. Повернув Wiimote под углом 45°, мы получим ускорение по осям Y и Z, примерно равное $\sin 45^\circ = \cos 45^\circ = 0{,}707g$. Если откалибровать ввод акселерометров, чтобы найти нулевые значения по всем трем осям, можно с легкостью вычислить тангаж, рыскание и крен (см. подраздел 5.3.9), используя операции арксинуса и арккосинуса.

Здесь стоит сделать два замечания. Во-первых, если человек с Wiimote в руках двигается, ввод акселерометров будет включать в себя это ускорение, что сведет на нет наши вычисления. Во-вторых, ось Z в акселерометре уже откалибрована с учетом силы тяжести, а две другие оси — нет. Это означает, ось Z имеет пониженную точность при определении положения. Многие игры для Wii требуют, чтобы пользователь держал Wiimote нестандартным образом (например, кнопками к себе или инфракрасным сенсором вверх). Это максимально повышает точность показаний за счет размещения оси X или Y в направлении силы тяжести и отказа от предварительно откалиброванной оси Z. Подробнее об этом можно почитать, перейдя по ссылке http://druid.caughq.org/presentations/turbo/Wiimote-Hacking.pdf.

9.3.6. Камеры

У Wiimote есть уникальная особенность, которой нет у в любого другого стандартного HID-устройства для консолей, — инфракрасный (ИК) датчик. Это, в сущности, камера низкого разрешения, которая записывает двухмерное инфракрасное изображение в том направлении, куда смотрит контроллер Wiimote. Wii поставляется с сенсорной панелью, которая лежит на телевизоре и имеет два инфракрасных светодиода. На изображении, записанном ИК-камерой, эти светодиоды выглядят как две яркие точки на темном фоне. ПО для обработки изображений в Wiimote анализирует полученную картинку и находит местоположение и размер этих точек (на самом деле оно способно распознавать и передавать информацию сразу о четы-

рех точках). Затем эти данные могут быть прочитаны консолью по беспроводному Bluetooth-соединению.

Положение и наклон прямой, проведенной через две точки, используется для определения поворота Wiimote относительно всех трех осей (при условии, что контроллер направлен в сторону сенсорной панели). Анализируя расстояние между точками, программное обеспечение может определить, насколько близко или далеко находится Wiimote от телевизора. Некоторое ПО учитывает также размер точек (рис. 9.10).

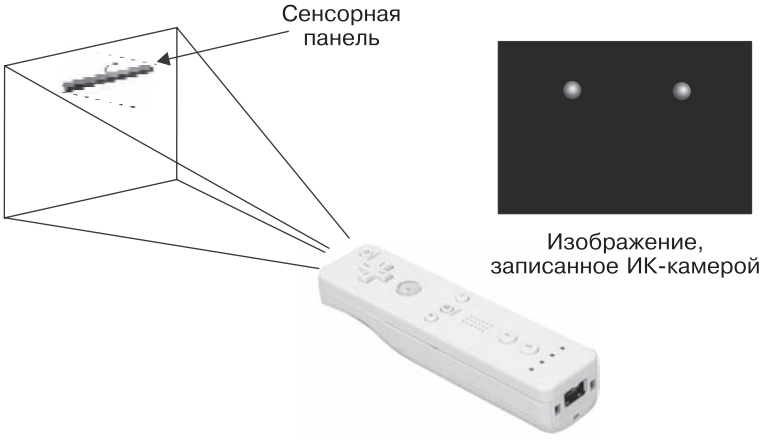


Рис. 9.10. Сенсорная панель Wii имеет два инфракрасных светодиода в виде двух ярких точек на изображении, записанном ИК-камерой Wiimote

Еще одним популярным устройством с камерой является Sony PlayStation Eye для PS3 (рис. 9.11). Это, в сущности, высококачественная цветная камера, которую можно использовать для разного рода задач. Она может выступать и как обычная веб-камера для видеоконференций. Ее также можно применять подобно ИК-камере в Wiimote, определяя положение, наклон и глубину. Игровое сообщество только начало открывать для себя возможности этих продвинутых устройств ввода.

Рис. 9.11. Sony PlayStation Eye для PS3

В PlayStation 4 компания Sony улучшила устройство Eye и переименовала его в PlayStation Camera. В сочетании с контроллерами PlayStation Move (рис. 9.12) или DualShock 4 оно способно распознавать жесты примерно так же, как это делает инновационная система Kinect от Microsoft (рис. 9.13).

Рис. 9.12. PlayStation Camera, контроллер PlayStation Move и джойстик DualShock 4 для PS4

Рис. 9.13. Microsoft Kinect для Xbox 360 *(вверху)* и Xbox One *(внизу)*

9.4. Виды вывода

HID-устройства в основном используются для передачи ввода от игрока к игровому программному обеспечению. Но некоторые из них предоставляют обратную связь с помощью средств вывода.

9.4.1. Вибрация

Джойстики из серии DualShock для PlayStation и контроллеры для Xbox и Xbox 360 поддерживают *вибрацию* (или отдачу). Это позволяет им вибрировать в руках игрока, имитируя турбулентность или столкновения, которые персонаж может испытывать в игре. Вибрации обычно создаются одним или несколькими вибромоторами, каждый из которых вращает немного несбалансированный груз с переменной скоростью. Для создания разных тактильных эффектов, ощущаемых игроком, игра может менять скорость их работы и включать/выключать их, когда это нужно.

9.4.2. Силовая обратная связь

Силовая обратная связь подразумевает, что элементы управления HID-устройства подключены к мотору, который позволяет им слегка сопротивляться воздействию игрока. Эта методика распространена в аркадных гоночных играх, в которых для имитации сложных условий вождения или резких поворотов регулируется жесткость вращения руля. Как и в случае с вибрацией, игровое ПО обычно может включать и выключать моторы, а также управлять силой и направлением их вращения по отношению к рулевому механизму.

9.4.3. Звук

Звуком в игровом движке обычно занимается отдельная подсистема, и некоторые устройства умеют принимать ее вывод. Например, Wiimote имеет маленький динамик низкого качества. У контроллеров Xbox 360, Xbox One и DualShock 4 есть разъем для наушников с микрофоном, которые, как и любое аудиоустройство, подключенное по USB, можно использовать как для вывода (в качестве динамиков), так и для ввода (в качестве микрофона). USB-гарнитуры часто применяются в многопользовательских играх, в которых игроки могут общаться друг с другом с помощью цифрового голосового соединения.

9.4.4. Другие разновидности ввода и вывода

HID-устройства, безусловно, могут поддерживать много других разновидностей ввода и вывода. На некоторых старых консолях, таких как Sega Dreamcast, на джойстике находились разъемы для карт памяти. У контроллера для Xbox 360, джойстиков Sixaxis и DualShock 3, а также у Wiimote есть четыре светодиода,

которые могут загораться по запросу игрового ПО. Игры могут управлять цветом световой полосы в DualShock 4. Ну и, конечно, специализированные устройства вроде музыкальных инструментов, танцевальных матов и прочего имеют собственные разновидности ввода и вывода.

Индустрия устройств для взаимодействия с пользователями активно развивается. Сейчас одной из самых интересных областей являются интерфейсы управления на основе жестов и мыслей. Можно не сомневаться, что в ближайшие годы производители консолей и HID-устройств продолжат радовать нас новинками.

9.5. Системы игрового движка для работы с HID-устройствами

Большинство игровых движков не используют сырой ввод напрямую из HID-устройств. Обычно эти данные обрабатываются разными способами, чтобы игра была плавной, приятной и предсказуемой. Кроме того, в большинстве движков между HID и игрой существует минимум один промежуточный слой, позволяющий абстрагировать ввод тем или иным образом. Например, с помощью таблицы привязки кнопок мы можем превращать обычные нажатия в логические игровые действия. Это даст возможность игроку переназначать функции кнопок согласно собственным предпочтениям. В этом разделе я перечислю несколько типичных требований к подсистеме для работы с HID-устройствами и затем мы подробно рассмотрим каждое из них.

9.5.1. Типичные требования

HID-система игрового движка обычно имеет некоторые или все из перечисленных далее особенностей:

- мертвые зоны;
- фильтрация аналоговых сигналов;
- распознавание событий (например, нажатие кнопки «вверх» и «вниз»);
- распознавание *последовательных* нажатий или комбинаций нажатий нескольких кнопок, известных как *аккорды*;
- распознавание жестов;
- управление несколькими HID-устройствами в многопользовательском режиме;
- поддержка HID на разных платформах;
- переназначение элементов управления;
- контекстный ввод;
- возможность временно отключать некоторые элементы управления.

9.5.2. Мертвые зоны

Джойстики, триггеры и любые другие аналоговые элементы управления генерируют ввод в диапазоне, который определяется заранее заданными минимальным и максимальным значениями (обозначим их I_{min} и I_{max}). Мы ожидаем, что элемент, к которому прикоснулись, вернет нам стабильное и четкое нейтральное значение, которое мы обозначим I_0. Нейтральная величина, как правило, численно равна нулю и находится либо строго посередине между I_{min} и I_{max} (если речь идет о центрированных, двунаправленных элементах, таких как стики), либо совпадает с I_{min} (если это однонаправленный элемент вроде триггера).

К сожалению, HID-устройства являются аналоговыми по своей природе, поэтому напряжение, которое они создают, содержит много шума, а тот ввод, который мы на самом деле получаем, может немного отклоняться от I_0. Чтобы решить эту проблему, вокруг I_0 чаще всего создают небольшую *мертвую зону*. Для стика ее можно определить как $[I_0 - \delta, I_0 + \delta]$, а для триггера — как $[I_0, I_0 + \delta]$. Любой ввод, находящийся в пределах этого диапазона, просто приравнивается к I_0. Мертвая зона должна быть достаточно широкой, чтобы охватывать наиболее шумный ввод, генерируемый нейтральным элементом управления, но достаточно узкой, чтобы HID-устройство казалось игроку отзывчивым.

9.5.3. Фильтрация аналоговых сигналов

Шум в сигнале создает проблемы даже тогда, когда элементы управления находятся вне мертвой зоны. Иногда из-за этого движения, управляемые HID-устройством, становятся дергаными и неестественными. В связи с этим многие игры *фильтруют* сырой ввод, полученный из контроллера. Шум, как правило, имеет более высокую частоту по сравнению с сигналом, который создает игрок. Следовательно, в качестве решения сырые данные можно сначала пропустить через *низкочастотный фильтр* и только потом использовать в игре.

Дискретный низкочастотный фильтр первого порядка можно реализовать объединением текущего нефильтрованного ввода с фильтрованным вводом из предыдущего кадра. Если обозначить последовательность нефильтрованных и фильтрованных входящих значений в виде функций от времени $u(t)$ и соответственно $f(t)$, где t — это время, можно записать следующее:

$$f(t) = (1 - a)f(t - \Delta t) + au(t), \tag{9.1}$$

где параметр a определяется продолжительностью кадра Δt и константой фильтрации RC (которая является произведением сопротивления и емкости в традиционной аналоговой RC-цепи для низкочастотной фильтрации):

$$a = \frac{\Delta t}{RC + \Delta t}. \tag{9.2}$$

Далее показано, как это можно легко реализовать на C или C++:

```
F32 lowPassFilter(F32 unfilteredInput,
                  F32 lastFramesFilteredInput,
                  F32 rc, F32 dt)
{
    F32 a = dt / (rc + dt);

    return (1 - a) * lastFramesFilteredInput
        + a * unfilteredInput;
}
```

Здесь предполагается, что вызывающий код хранит отфильтрованный ввод из предыдущего кадра и может использовать его в текущем. Подробнее об этом — по адресу ru.wikipedia.org/wiki/Фильтр_нижних_частот.

Еще один способ фильтрации ввода, полученного из HID-устройства, заключается в вычислении простого скользящего среднего. Например, если нужно усреднить входящие данные в интервале длиной 3/30 секунды (три кадра), мы можем просто хранить сырые значения в трехэлементном кольцевом буфере. Таким образом, отфильтрованный ввод в произвольный момент времени будет представлять собой сумму значений в этом массиве, разделенную на 3. При создании такого фильтра можно уделить внимание нескольким малозначительным деталям. Например, нужно как следует обработать ввод в первых двух кадрах, когда трехэлементный массив еще не заполнен корректными данными. Но сама реализация этого подхода не должна вызвать особых затруднений. Код, представленный далее, — это один из вариантов корректного использования скользящего среднего из N элементов:

```
template< typename TYPE, int SIZE >
class MovingAverage
{
    TYPE m_samples[SIZE];
    TYPE m_sum;
    U32 m_curSample;
    U32 m_sampleCount;

public:
    MovingAverage() :
        m_sum(static_cast<TYPE>(0)),
        m_curSample(0),
    mm_sampleCount(0)
    {
    }

    void addSample(TYPE data)
    {
        if (m_sampleCount == SIZE)
        {
            m_sum -= m_samples[m_curSample];
        }
        else
```

```
        {
            m_sampleCount++;
        }

        m_samples[m_curSample] = data;
        m_sum += data;
        m_curSample++;

        if (m_curSample >= SIZE)
        {
            m_curSample = 0;
        }
    }

    F32 getCurrentAverage() const
    {
        if (m_sampleCount != 0)
        {
            return static_cast<F32>(m_sum)
                / static_cast<F32>(m_sampleCount);
        }
        return 0.0f;
    }
};
```

9.5.4. Обнаружение событий ввода

Низкоуровневый HID-интерфейс обычно предоставляет текущее состояние различных элементов управления устройства. Однако игры зачастую заинтересованы в обнаружении *событий*, таких как изменение состояния, а не в анализе текущих значений в каждом кадре. Наверное, самыми распространенными, но далеко не единственными HID-событиями являются нажатие (button-down) и отпускание (button-up) кнопки.

Нажатие и отпускание кнопки

Давайте пока что будем считать, что нажатой и ненажатой кнопке соответствуют биты ввода 1 и 0 соответственно. Сравнение значений этих битов в предыдущем и текущем кадрах — самый простой способ обнаружения изменений в состоянии кнопки. Если значения различаются, мы знаем, что произошло событие. Тип события (button-up или button-down) можно определить по текущему состоянию кнопки.

Для обнаружения событий button-down и button-up можно использовать простые битовые операторы. Учитывая, что 32-битное машинное слово `buttonStates` может содержать текущее состояние битов для 32 кнопок, следует сгенерировать еще два 32-битных значения: одно для событий button-down (назовем его `buttonDowns`), а другое — для событий button-up (`buttonUps`). В обоих случаях, если в текущем кадре произошло событие, бит соответствующей кнопки будет равен `1`,

в противном случае его значением будет 0. Чтобы это реализовать, нам также понадобится состояние кнопок в предыдущем кадре prevButtonStates.

Оператор «исключающее ИЛИ» (XOR) возвращает 0, если оба входящих значения совпадают, и 1, если различаются. Если применить XOR к предыдущему и текущему состояниям, единицы будут только у тех кнопок, состояние которых изменилось между предыдущим и текущим кадрами. Чтобы понять, какое событие произошло, button-up или button-down, нужно проверить текущее состояние каждой кнопки. Любая кнопка, которая в данный момент является нажатой, генерирует событие button-down (и наоборот для button-up). В представленном далее коде эти способы применяются для генерации двух машинных слов с событиями кнопок:

```
class ButtonState
{
    U32 m_buttonStates;      // состояние кнопок в текущем кадре
    U32 m_prevButtonStates;  // состояние в предыдущем кадре
    U32 m_buttonDowns;       // 1 - кнопка нажата в этом кадре
    U32 m_buttonUps;         // 1 - кнопка отпущена в этом кадре

    void DetectButtonUpDownEvents()
    {
        // Если m_buttonStates и m_prevButtonStates
        // корректные, генерируем m_buttonDowns
        // и m_buttonUps.

        // Сначала используем XOR, чтобы определить,
        // какие биты изменились.
        U32 buttonChanges = m_buttonStates
                          ^ m_prevButtonStates;

        // Теперь используем И, чтобы оставить только
        // НАЖАТЫЕ биты.
        m_buttonDowns = buttonChanges & m_buttonStates;

        // Используем И-НЕ, чтобы оставить только
        // НЕНАЖАТЫЕ биты.
        m_buttonUps = buttonChanges & (~m_buttonStates);
    }

    // ...
};
```

Аккорды

Аккорд — это одновременное нажатие группы кнопок, которое вызывает уникальное поведение в игре. Вот несколько примеров.

- Во время загрузки *Super Mario Galaxy* игрок должен одновременно нажать на Wiimote кнопки A и B, чтобы начать новую игру.
- Одновременное нажатие кнопок 1 и 2 переключает Wiimote в режим обнаружения Bluetooth-устройств (вне зависимости от текущей игры).

- Прием захвата во многих файтингах инициируется комбинацией двух кнопок.
- Если во время игры в *Uncharted* нажать и удерживать в DualShock 3 оба триггера, персонаж игрока получает возможность свободно летать по игровому миру с выключением столкновений (простите, но эта возможность доступна только для разработчиков и отсутствует в публичной версии игры). Подобные трюки существуют во многих играх для упрощения разработки (естественно, их можно вызывать не только с помощью аккордов). В движке Quake этот режим называется *no-clip* («без обрезки»), поскольку область столкновения персонажа не *обрезается* до размеров допустимой области, предусмотренной для игры.

Сам принцип распознавания аккордов довольно прост: мы отслеживаем состояние двух или больше кнопок и выполняем нужную операцию, только когда *все* они нажаты.

Однако здесь необходимо учитывать некоторые тонкости. К примеру, если одна или несколько кнопок аккорда имеют другие функции, мы должны проследить за тем, чтобы они не были выполнены *вместе* с аккордом. Для этого при обработке отдельных нажатий нужно проверять, нажаты ли другие кнопки аккорда.

Еще одним неприятным моментом является то, что люди неидеальны и часто нажимают одну или несколько кнопок аккорда немного раньше остальных. Поэтому код распознавания аккордов должен учитывать вероятность того, что одна или несколько отдельных кнопок будут нажаты в кадре i, а остальные — в кадре $i + 1$ или даже несколькими кадрами позже. С этим можно бороться по-разному.

- Кнопочный ввод можно спроектировать так, чтобы аккорд всегда выполнял функции отдельных кнопок *плюс* дополнительное действие. Например, если нажатие L1 вызывает выстрел основного оружия, а L2 — бросок гранаты, мы можем сделать так, чтобы аккорд L1 + L2 приводил к выстрелу, бросанию гранаты *и* излучению энергетической волны, которая удваивает урон от этого оружия. Таким образом, с точки зрения игрока поведение всегда будет одним и тем же независимо от того, распознались ли отдельные нажатия еще до аккорда.
- Можно создать задержку между обнаружением отдельного события нажатия и моментом, когда оно засчитывается. Если во время этой задержки (скажем, два или три кадра) обнаружен аккорд, он получает приоритет над отдельными нажатиями. Это дает игроку некоторый запас времени для выполнения аккорда.
- Можно сам аккорд распознать при нажатии кнопок, а связанное с ним действие инициировать только в момент их отпускания.
- Действие отдельной кнопки можно начать выполнять сразу, но сделать так, чтобы аккорд мог его прервать.

Распознавание последовательностей и жестов

Идея введения задержки между собственно нажатием кнопки и тем, когда ее можно считать нажатой, является частным случаем *распознавания жестов*. Жест — это последовательность действий, которые игрок выполняет с помощью HID-устройства на протяжении какого-то времени. Например, если мы разрабатываем игру в жанре

файтинг, может возникнуть необходимость в распознавании *последовательных* нажатий, таких как A — B — A. Это могут быть не только кнопки. В качестве примера можно привести A — B — A — влево — вправо — влево, где последние три действия выглядят как раскачивание из стороны в сторону одного из стиков на контроллере. Обычно последовательности или жесты считаются корректными, только если они выполнены за какое-то ограниченное время. Таким образом, нажатие A — B — A в пределах четверти секунды может быть засчитано, но если растянуть его на целую секунду или две, оно может оказаться недействительным.

Распознавание жестов в основном реализуется за счет хранения краткой истории HID-действий, выполненных игроком. Когда обнаруживается первый элемент жеста, он сохраняется в буфер вместе с временной меткой, сигнализирующей о времени его появления. В дальнейшем сверяются временные метки предыдущего и только что обнаруженного элементов. Если разница находится в пределах допустимого интервала, элемент добавляется в буфер. Если вся последовательность завершается в пределах выделенного времени (то есть если буфер полон), генерируется событие, которое сообщает остальному коду игрового движка о распознавании жеста. Но если во время этого процесса обнаруживается какой-то недопустимый ввод или один из элементов жеста выходит за рамки отведенного интервала, весь буфер сбрасывается и игроку приходится вводить жест сначала.

Чтобы понять, как это работает, рассмотрим три конкретных примера.

Быстрое нажатие кнопок. Во многих играх для выполнения действий требуется быстрое нажатие кнопок. Частота нажатий может превращаться в какой-то игровой показатель, такой как скорость, с которой персонаж бежит или делает что-то другое. Частота обычно используется также для определения действительности жеста: если она опускается ниже какого-то минимального значения, жест считается недействительным.

Чтобы определить частоту нажатия кнопки, можно проверить, когда она в последний раз генерировала событие button-down. Обозначим этот момент T_{last}. Тогда частота f будет обратно пропорциональной временному отрезку между нажатиями $\Delta T = T_{\text{cur}} - T_{\text{last}}$:

$$f = 1 / \Delta T.$$

При обнаружении каждого нового нажатия частота f вычисляется заново. Для реализации минимально допустимой частоты мы сравниваем f с минимальной частотой f_{min} (можно также сравнить ΔT непосредственно с максимальным интервалом $\Delta T_{\text{max}} = 1 / f_{\text{min}}$). Если значение не выходит за пределы этого лимита, мы обновляем T_{last} и считаем, что жест находится в процессе ввода. В противном случае не обновляем T_{last}. Жест будет считаться недействительным, пока не случится еще одна пара событий button-down с достаточно коротким промежутком времени. Это проиллюстрировано в следующем псевдокоде:

```
class ButtonTapDetector
{
    U32 m_buttonMask; // какую кнопку мы наблюдаем (битовая маска)
    F32 m_dtMax;      // максимально допустимое время между нажатиями
```

```
    F32 m_tLast;      // время, прошедшее с момента
                      // последнего события button-down (в секундах)

public:

    // Создаем объект, распознающий быстрые нажатия
    // заданной кнопки (с определенным индексом).
    ButtonTapDetector(U32 buttonId, F32 dtMax) :
        m_buttonMask(1U << buttonId),
        m_dtMax(dtMax),
        m_tLast(CurrentTime() - dtMax) // начинаем с недействительного
                                       // значения
    {
    }

    // Вызываем этот код, чтобы проверить,
    // выполняется ли жест в текущий момент.
    bool IsGestureValid() const
    {
        F32 t = CurrentTime();
        F32 dt = t - m_tLast;
        return (dt < m_dtMax);
    }

    // Вызываем один раз в каждом кадре.
    void Update()
    {
        if (ButtonsJustWentDown(m_buttonMask))
        {
            m_tLast = CurrentTime();
        }
    }
};
```

В приведенном фрагменте кода предполагается, что каждая кнопка имеет уникальный идентификатор. На самом деле это просто индекс в диапазоне от 0 до $N-1$, где N — количество кнопок на HID-устройстве. Мы преобразуем этот идентификатор в битовую маску, сдвигая беззнаковый бит 1 влево на количество позиций, равное индексу кнопки (`1U << buttonId`). Функция `ButtonsJustWentDown()` возвращает ненулевое значение, если *любая* из кнопок, заданных битовой маской, была нажата в текущем кадре. Здесь нас интересует лишь одно событие button-down, но эту же функцию можно использовать для проверки нескольких одновременных нажатий, чем мы и займемся чуть позже.

Многокнопочные последовательности. Представьте, что мы хотим распознать последовательность A — B — A, введенную в пределах одной секунды. Сделаем это следующим образом. Создадим переменную, которая определяет, какая кнопка в последовательности нас сейчас интересует. Если последовательность имеет вид массива с идентификаторами кнопок (например, `aButtons[3] = {A, B, A}`), эта переменная будет играть роль его индекса *i*. Сначала она указывает на первую

кнопку, $i = 0$. Отслеживаем начальное время последовательности T_{start} так же, как делали это в примере с быстрыми нажатиями.

Логика выглядит так: при возникновении каждого события button-down, соответствующего кнопке, которая интересует нас в данный момент, мы сравниваем его временную метку с моментом начала всей последовательности T_{start}. Если оно произошло на подходящем отрезке времени, переходим к следующей кнопке в последовательности. Значение T_{start} обновляется только в случае, если текущая кнопка — первая ($i = 0$). Если обнаруживается событие button-down, которое не соответствует следующей кнопке в последовательности, или если временной интервал получился слишком длинным, мы сбрасываем индекс кнопки i в начало последовательности и присваиваем T_{start} какое-нибудь недействительное значение, например 0. Этот подход проиллюстрирован в следующем листинге:

```
class ButtonSequenceDetector
{
    U32* m_aButtonIds;  // последовательность кнопок, которую мы отслеживаем
    U32  m_buttonCount; // количество кнопок в последовательности
    F32  m_dtMax;       // максимальное время всей последовательности
    U32  m_iButton;     // очередная кнопка последовательности,
                        // которую мы ожидаем
    F32  m_tStart;      // начальное время последовательности (в секундах)

public:

    // Создаем объект, отслеживающий заданную кнопку
    // в последовательности. Когда последовательность успешно
    // распознана, данное событие распространяется так, чтобы
    // остальной код игры мог как следует на него среагировать.

    ButtonSequenceDetector(U32* aButtonIds,
                           U32 buttonCount,
                           F32 dtMax,
                           EventId eventIdToSend) :
        m_aButtonIds(aButtonIds),
        m_buttonCount(buttonCount),
        m_dtMax(dtMax),
        m_eventId(eventIdToSend), // событие, отправляемое по окончании
        m_iButton(0),             // начало последовательности
        m_tStart(0)               // начальное значение может быть любым
    {
    }

    // Вызываем единожды за кадр.
    void Update()
    {
        ASSERT(m_iButton < m_buttonCount);

        // Определяем, какую кнопку мы ожидаем дальше, в виде битовой маски
        // (сдвигаем 1 на количество битов, равное индексу).
```

```
        U32 buttonMask = (1U << m_aButtonId[m_iButton]);

        // Если только что была нажата любая другая кнопка, кроме той,
        // что мы ожидаем, последовательность сбрасывается (для проверки
        // всех других кнопок используем оператор «битовое НЕ»).

        if (ButtonsJustWentDown(~buttonMask))
        {
            m_iButton = 0; // сбрасываем
        }

        // В противном случае, если только что была нажата
        // подходящая кнопка, проверяем dt и обновляем состояние
        // соответствующим образом.

        else if (ButtonsJustWentDown(buttonMask))
        {
            if (m_iButton == 0)
            {
                // Это первая кнопка в последовательности.
                m_tStart = CurrentTime();
                m_iButton++; // переходим к следующей кнопке
            }
            else
            {

                F32 dt = CurrentTime() - m_tStart;

                if (dt < m_dtMax)
                {
                    // Последовательность все еще действительная.

                    m_iButton++; // переходим к следующей кнопке

                    // Завершена ли последовательность?
                    if (m_iButton == m_buttonCount)
                    {
                        BroadcastEvent(m_eventId);
                        m_iButton = 0; // сбрасываем
                    }
                }
                else
                {
                    // Извините, слишком медленно.
                    m_iButton = 0; // сбрасываем
                }
            }
        }
    }
};
```

Вращение аналогового стика. В качестве примера более сложного жеста рассмотрим процесс распознавания вращения левого аналогового стика по часовой стрелке. Это довольно легко реализовать, разделив двухмерный диапазон возможных позиций стика на квадранты (рис. 9.14). При вращении по часовой стрелке стик проходит квадранты в таком порядке: левый верхний, правый верхний, правый нижний и левый нижний. Можно обрабатывать каждый из этих случаев как нажатие кнопки и распознавать полное вращение с помощью слегка видоизмененной версии представленного ранее кода. Оставляем это вам в качестве упражнения. Попробуйте его выполнить!

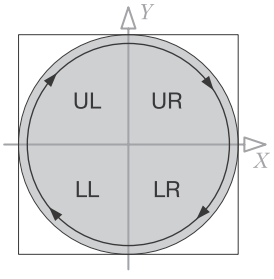


Рис. 9.14. Распознавание круговых движений стика путем разделения его двухмерного диапазона на квадранты

9.5.5. Управление несколькими HID-устройствами для нескольких игроков

Большинство игровых консолей поддерживают подключение как минимум двух HID-устройств для многопользовательских игр. Движок должен следить за тем, какие устройства подключены в данный момент, и направлять ввод каждого из них подходящему игроку. Это означает, что нужно каким-то образом привязывать контроллеры к игрокам. В простейшем случае можно обеспечить прямое соответствие между индексами контроллера и игрока, применяется также более тонкий подход — связывание контроллеров с игроками в момент нажатия кнопки Start.

Даже в однопользовательских играх с одним HID-устройством движок должен уметь справляться с исключительными ситуациями, такими как случайное отключение контроллера или разрядка батареек. При потере соединения с контроллером большинство игр ставят игровой процесс на паузу, выводят сообщение и ждут, когда его снова подключат. Некоторые многопользовательские игры замораживают или временно убирают аватар пропавшего HID-устройства, позволяя другим игрокам продолжить игру. Удаленный/замороженный аватар может быть заново активирован при подключении контроллера.

Если HID-устройство питается от батареек, ответственность за отслеживание их заряда ложится на игру или операционную систему. Обычно игрока предупреждают о заканчивающемся заряде, например отображая на экране ненавязчивое сообщение и/или с помощью определенного звука.

9.5.6. Межплатформенные HID-системы

Многие игровые движки являются межплатформенными. Для обработки HID-ввода они могли бы использовать директивы условной компиляции на всех участках кода, где происходит взаимодействие с HID-устройствами, как показано далее (это решение, несомненно, далеко от идеала, но оно работает):

```
#if TARGET_XBOX360
if (ButtonsJustWentDown(XB360_BUTTONMASK_A))
#elif TARGET_PS3
if (ButtonsJustWentDown(PS3_BUTTONMASK_TRIANGLE))
#elif TARGET_WII
if (ButtonsJustWentDown(WII_BUTTONMASK_A))
#endif
{
    // Какие-то действия…
}
```

Более разумный подход состоит в создании некоего слоя абстрагирования оборудования, который позволит изолировать код игры от деталей аппаратной реализации.

Если повезет, нам удастся нивелировать большинство различий между HID-устройствами на разных платформах за счет рационального выбора абстрактных кнопок и идентификаторов осей. Например, если игра должна быть доступна для Xbox 360 и PS3, этого легко добиться, так как расположение элементов управления (кнопок, осей и триггеров) на их джойстиках почти одинаковое. Различаются лишь идентификаторы, но мы можем довольно легко заменить их универсальными аналогами, которые подходят для обоих видов джойстиков, например:

```
enum AbstractControlIndex
{
    // Кнопки для начала игры и возвращения назад
    AINDEX_START,           // Xbox 360 Start, PS3 Start
    AINDEX_BACK_SELECT,     // Xbox 360 Back, PS3 Select

    // Группа кнопок слева (D-pad)
    AINDEX_LPAD_DOWN,
    AINDEX_LPAD_UP,
    AINDEX_LPAD_LEFT,
    AINDEX_LPAD_RIGHT,

    // Группа из четырех кнопок справа
    AINDEX_RPAD_DOWN,       // Xbox 360 A, PS3 X
    AINDEX_RPAD_UP,         // Xbox 360 Y, PS3 Треугольник
    AINDEX_RPAD_LEFT,       // Xbox 360 X, PS3 Квадрат
    AINDEX_RPAD_RIGHT,      // Xbox 360 B, PS3 Круг

    // Кнопки аналоговых стиков слева и справа
    AINDEX_LSTICK_BUTTON,   // Xbox 360 LThumb, PS3 L3,
                            // Xbox белый
    AINDEX_RSTICK_BUTTON,   // Xbox 360 RThumb, PS3 R3,
                            // Xbox черный

    // Левый и правый бамперы
    AINDEX_LSHOULDER,       // Xbox 360 бампер L, PS3 L1
    AINDEX_RSHOULDER,       // Xbox 360 бампер R, PS3 R1
```

```
    // Оси левого стика
    AINDEX_LSTICK_X,
    AINDEX_LSTICK_Y,

    // Оси правого стика
    AINDEX_RSTICK_X,
    AINDEX_RSTICK_Y,

    // Оси левого и правого триггеров
    AINDEX_LTRIGGER,        // Xbox 360 -Z, PS3 L2
    AINDEX_RTRIGGER,        // Xbox 360 +Z, PS3 R2
};
```

Слой абстракции может переводить реальные идентификаторы элементов управления на поддерживаемой платформе в универсальные индексы. Например, при сохранении состояния кнопок в 32-битное машинное слово мы можем отсортировать биты в нужном порядке, чтобы они соответствовали абстрактным индексам. Точно так же можно перестраивать и аналоговый ввод.

Иногда при связывании физических и абстрактных элементов управления приходится прибегать к некоторым хитростям. Например, в Xbox левый и правый триггеры представлены в виде единой оси: когда нажат левый, получаются отрицательные значения, когда правый — положительные, а когда ни один из них не нажат, ввод равен нулю. Чтобы сделать такое поведение совместимым с PlayStation DualShock, мы можем разделить этот ввод на две отдельные оси, масштабируя значения так, чтобы они находились в одном и том же диапазоне на всех платформах.

Естественно, это не единственный способ работы с вводом/выводом HID-устройств в многоплатформенном движке. К примеру, мы могли бы применить более функциональный подход, назначая абстрактным элементам управления имена, соответствующие их функциям в игре, а не физическому расположению на джойстике. Мы могли бы создать функции более высокого уровня, распознающие абстрактные жесты, с отдельным кодом для каждой платформы. Можно пойти еще дальше и написать разные платформозависимые версии любого кода, который имеет дело с вводом/выводом HID-устройств. Существует множество разных решений, но практически все межплатформенные игровые движки *тем или иным* способом изолируют игру от деталей аппаратной реализации.

9.5.7. Переназначение элементов управления

Многие игры дают игроку определенную свободу выбора в отношении тех функций, которые выполняют разные элементы управления, расположенные на физическом HID-устройстве. Типичным случаем является выбор направления вертикальной оси правого аналогового стика в консольных играх. Одни игроки предпочитают, чтобы при смещении стика вперед камера поднималась вверх, а другим по душе перевернутая схема управления, когда для подъема камеры вверх нужно оттянуть стик на себя, как штурвал самолета. Бывают игры, которые позволяют игроку выбирать между двумя или более готовыми вариантами привязки кнопок. На ПК пользователям иногда дают полный контроль над функциями отдельных клавиш,

кнопок и колесика мыши, а также обеспечивают различные схемы управления двумя осями мыши.

Прежде чем рассматривать детали реализации, процитирую любимую поговорку старого профессора Джея Блека из Университета Уотерлу: «Любую проблему в информатике можно решить с помощью слоя абстракции». Присвоим всем функциям в игре уникальные идентификаторы и сформируем простую таблицу, которая связывает физический или абстрактный индекс каждого элемента управления с логической функцией. Если нужно определить, должна ли быть активирована определенная логическая функция, мы ищем в таблице абстрактный или физический идентификатор соответствующего элемента управления и считываем его состояние. Для изменения привязки можем либо заменить всю таблицу целиком, либо дать возможность пользователю редактировать отдельные ее записи.

Мы упустили несколько подробностей. Например, разные элементы управления генерируют разного рода ввод. Аналоговые оси могут возвращать значения в диапазоне от –32 768 до 32 767, от 0 до 255 или каком-то другом. Состояние всех цифровых кнопок HID-устройства обычно упаковывают в единое машинное слово. Следовательно, мы должны разрешить изменение только тех привязок, которые имеет смысл менять. Например, кнопку нельзя использовать для логической функции, которой нужен диапазон значений (ось). Чтобы решить эту проблему, мы можем нормализовать весь ввод. Скажем, все входящие значения аналоговых осей и кнопок можно свести к диапазону [0, 1]. Это не так полезно, как может показаться на первый взгляд, поскольку одни оси по своей природе являются двунаправленными (стики), а другие — однонаправленными (триггеры). Но мы можем разделить элементы управления на несколько классов и разрешить переназначение только внутри каждого из них. Далее приведен пример разумного набора классов и показан их нормализованный ввод для стандартного консольного джойстика.

- *Цифровые кнопки*. Их состояние упаковывается в 32-битное машинное слово, по одному биту на кнопку.
- *Однонаправленные абсолютные оси*, такие как *триггеры* и *аналоговые кнопки*. Генерируют ввод с плавающей запятой в диапазоне [0, 1].
- *Двунаправленные абсолютные оси*, такие как *аналоговые стики*. Генерируют ввод с плавающей запятой в диапазоне [–1, 1].
- *Относительные оси*, такие как *трекболы*, *оси* и *колесико мыши*. Генерируют ввод с плавающей запятой в диапазоне [–1, 1], где ±1 представляет максимально возможный относительный сдвиг в рамках одного кадра игры, то есть в интервале длиной 1/30 или 1/60 с.

9.5.8. Контекстный ввод

Во многих играх один и тот же элемент управления может иметь разные функции в зависимости от контекста. Простейшим примером является вездесущая кнопка Использовать. Если нажать ее, стоя возле двери, она может открыть эту дверь. Если нажать возле объекта, она может сделать так, что персонаж игрока подберет этот

объект. В качестве еще одного распространенного примера можно привести модальную схему управления. Когда игрок куда-то идет, ввод применяется для передвижения и изменения наклона камеры. Когда игрок за рулем автомобиля, ввод используется для вождения, а управление камерой может быть другим.

Контекстный ввод довольно просто реализовать с помощью конечного автомата. Назначение конкретного элемента управления HID-устройства может зависеть от того, в каком состоянии мы сейчас находимся. Самое сложное — выбор подходящего состояния. Например, в момент нажатия контекстной кнопки Использовать игрок может находиться на равном расстоянии от оружия и аптечки. Какой объект в этом случае будет использован? В некоторых играх для разрешения таких спорных ситуаций предусмотрена система приоритетов. В этом примере оружие может оказаться более важным, чем аптечка. Реализация контекстного ввода не самая сложная задача, но, чтобы решить ее как следует, в любом случае потребуется множество проб и ошибок. Приготовьтесь к большому числу повторений и к фокус-тестированию!

Следует также обратить внимание на концепцию *владения элементами управления*. Некоторые элементы HID-устройства могут относиться к разным частям игры. Например, одни значения предназначены для управления персонажем игрока, другие — для управления камерой, а третьи — для взаимодействия с системой меню игры (пауза и т. д.). В некоторых игровых движках существует концепция логических устройств, состоящих лишь из какого-то подмножества элементов управления физического контроллера. Одно логическое устройство может использоваться для управления персонажем игрока, другое — для управления камерой, а третье — для навигации по меню.

9.5.9. Отключение элементов управления

В большинстве игр время от времени необходимо делать так, чтобы игрок не мог управлять своим персонажем. Например, когда персонаж игрока участвует в кинематографической сцене, имеет смысл временно отключить все элементы управления, а когда он проходит через узкий дверной проем, мы можем временно заблокировать свободное вращение камеры.

Довольно радикальное решение состоит в том, чтобы отключать отдельные элементы управления на самом устройстве, используя битовую маску. Каждый раз при считывании ввода мы проверяем битовую маску и, если в ней установлен соответствующий бит, возвращаем нейтральное или нулевое значение вместо того, которое на самом деле было прочитано. Но при отключении элементов управления следует быть особенно осторожными. Если забыть сбросить битовую маску, игра может войти в состояние, в котором из-за полной потери контроля игроку придется ее перезагружать. Очень важно тщательно проверять логику игры. Также не помешает предусмотреть специальные механизмы, с помощью которых битовая маска очищается в определенные ключевые моменты, такие как смерть и перерождение персонажа игрока.

Отключение HID-ввода делает его недоступным для всех потенциальных клиентов, что может оказаться слишком жестким ограничением. Наверное, лучше предусмотреть логику для отключения отдельных действий игрока или движений камеры в самом коде, который отвечает за управление персонажем и камерой. Таким образом, если камера, к примеру, решит игнорировать наклон правого аналогового стика, подсистемы движка смогут продолжить считывать состояние этого элемента управления в каких-то других целях.

9.6. HID-устройства на практике

Корректная и гладкая работа с HID-устройствами является важным аспектом любой хорошей игры. Сама концепция HID выглядит довольно простой и понятной. Но на практике может существовать много тонких моментов, таких как различия между разными физическими устройствами ввода, корректная реализация низкочастотной фильтрации, безошибочное переключение между схемами привязки элементов управления, достижение правильного ощущения вибрации в джойстике, учет ограничений, налагаемых производителями с помощью списков технических требований, и т. д. Команда разработчиков игры должна быть готова к тому, что для тщательной и полной реализации системы HID придется потратить необычайно много времени и инженерных усилий. Это очень важно, так как данная система лежит в основе самого ценного ресурса вашей игры — механики управления персонажем.

10 Инструменты для отладки и разработки

Разработка игрового ПО — сложный и запутанный процесс, требующий активных математических вычислений и чреватый ошибками. Поэтому неудивительно, что практически любая профессиональная команда разработчиков игр имеет собственный набор инструментов, которые делают процесс создания игры более простым и надежным. В этой главе мы рассмотрим инструменты для разработки и отладки, которые чаще всего можно встретить в игровых движках профессионального уровня.

10.1. Журналирование и трассировка

Помните свою первую программу на Паскале или Бейсике (ладно, если вы намного моложе меня, вполне вероятно, что ваша первая программа была написана на Java или, может быть, Python или Lua)? Как бы то ни было, вы, наверное, помните, как выглядел процесс ее отладки. В те дни вы наверняка имели смутное представление о том, что такое *отладчик*, и использовали вместо него инструкции print для вывода внутреннего состояния своей программы. Программисты на C/C++ называют это *отладкой методом printf* в честь функции `printf()` из стандартной библиотеки C.

Оказывается, такой подход по-прежнему вполне приемлем, даже *если* вы знаете о существовании отладчика. Применение точек останова и окон просмотра для отслеживания определенного вида ошибок может быть проблематичным, особенно в программировании для систем реального времени. Некоторые ошибки происходят только в случае, когда игра работает на полной скорости, другие являются результатом сложной цепочки событий, которая выглядит слишком длинной и запутанной, чтобы разбирать ее пошагово. В таких ситуациях наиболее мощным инструментом отладки может служить инструкция print.

У каждой игровой платформы есть какого-то рода консоль или устройство стандартного вывода в терминал. Вот несколько примеров.

- В консольных приложениях, написанных на C/C++ и запущенных под Linux или Win32, для вывода данных в консоль можно выполнять запись в `stdout` или `stderr`, используя функции `printf()`, `fprintf()` или интерфейс `iostream` из стандартной библиотеки C++.
- К сожалению, если ваша игра написана в виде оконного приложения для Win32, `printf()` и `iostream` работать не будут, так как консоли для вывода попросту не существует. Но если игра запущена в отладчике Visual Studio, вам доступна консоль отладки, на которую можно выводить информацию с помощью функции `OutputDebugString()` из состава Win32.
- Если вы пишете код для PlayStation 3 или PlayStation 4, на вашем компьютере запускается программа под названием Target Manager (или PlayStation Neighborhood в случае с PS4), которая позволяет запускать приложения на приставке. Target Manager включает в себя набор окон с терминальным выводом, в которые игровой движок может записывать свои сообщения.

Таким образом, вывод информации с целью отладки почти всегда сводится к простому добавлению в код вызовов `printf()`. Однако большинство игровых движков на этом не останавливаются. В следующих разделах мы рассмотрим различные средства вывода, которые они предоставляют.

10.1.1. Форматированный вывод с помощью функции OutputDebugString()

Функция `OutputDebugString()` из состава Windows SDK отлично подходит для вывода отладочной информации в окно `Debug Output` в Visual Studio. Но в отличие от `printf()` она не поддерживает *форматирование* и способна выводить лишь обычные строки в виде массивов. В связи с этим большинство игровых движков для Windows заворачивают ее в самописную функцию примерно такого вида:

```
#include <stdio.h>     // для va_list и прочего

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN 1
#endif
#include <windows.h>   // для OutputDebugString()

int VDebugPrintF(const char* format, va_list argList)
{
    const U32 MAX_CHARS = 1024;
    static char s_buffer[MAX_CHARS];

    int charsWritten
     = vsnprintf(s_buffer, MAX_CHARS, format, argList);
```

```
    // Сформировав отформатированную строку, вызываем Win32 API.
    OutputDebugString(s_buffer);

    return charsWritten;
}

int DebugPrintF(const char* format, ...)
{
    va_list argList;
    va_start(argList, format);

    int charsWritten = VDebugPrintF(format, argList);

    va_end(argList);
    return charsWritten;
}
```

Обратите внимание на то, что здесь реализованы две функции: `DebugPrintF()`, принимающая список аргументов переменной длины, заданный с помощью многоточия (…), и `VDebugPrintF()`, которая принимает только один аргумент, `va_list`. Это сделано для того, чтобы программисты могли создавать дополнительные функции вывода по принципу `VDebugPrintF()` (многоточие нельзя передать из функции в функцию, зато мы *можем* передавать списки `va_list`).

10.1.2. Уровень детализации

Потратив время и силы на размещение множества инструкций print в стратегически важных участках своего кода, мы бы хотели просто оставить их там на случай, если они когда-нибудь понадобятся. Для этого большинство движков предоставляют какого-то рода механизм для управления уровнем *детализации* динамически, во время выполнения или с помощью командной строки. Когда установлен минимальный (обычно нулевой) уровень детализации, выводятся только сообщения о критически важных ошибках. Чем выше этот уровень, тем больше инструкций print, которые вы разместили в своем коде, начинает фигурировать в выводе.

Чтобы это реализовать, уровень детализации проще всего хранить в глобальной целочисленной переменной с названием наподобие `g_verbosity`. Затем мы должны предоставить функцию `VerboseDebugPrintF()`, принимающую уровень детализации, на котором (или с которого) начинают выводиться сообщения. Эту функцию можно реализовать следующим образом:

```
int g_verbosity = 0;

void VerboseDebugPrintF(int verbosity,
                        const char* format, ...)
{
```

```
    // Выводим сообщения, только если глобальный
    // уровень детализации достаточно высок.
    if (g_verbosity >= verbosity)
    {
        va_list argList;
        va_start(argList, format);

        VDebugPrintF(format, argList);

        va_end(argList);
    }
}
```

10.1.3. Каналы

Чрезвычайно полезной является также возможность распределения отладочного вывода по *каналам*. Один канал может содержать сообщения системы анимации, а другой, к примеру, использоваться для вывода сообщений системы физики.

На некоторых платформах, например PlayStation 3, отладочный вывод можно направить в один из 14 отдельных терминалов. Кроме того, сообщения из всех терминалов собираются в одно специальное окно. Благодаря этому разработчик может легко сосредоточиться на сообщениях, которые ему интересны. Работая над анимационной задачей, вы можете просто переключиться на соответствующий терминал и игнорировать любой другой вывод. При решении общей проблемы неизвестного происхождения можете искать подсказки в комбинированном терминале.

Другие платформы, такие как Windows, предоставляют лишь одну консоль для отладочного вывода. Но даже в этом случае разделение вывода на каналы может оказаться полезным. Сообщениям в разных каналах можно назначить разные цвета. Вы также можете реализовать *фильтрацию* и включать и выключать ее на этапе выполнения, чтобы выводить содержимое только нужных вам каналов. Например, если разработчик занимается отладкой проблемы, связанной с анимацией, он может просто отфильтровать все каналы, кроме анимационного.

Систему отладочного вывода на основе каналов можно довольно легко реализовать, добавив в функцию еще один аргумент, который определяет нужный нам канал. Мы можем пронумеровать каналы или, еще лучше, назначить им символические названия с помощью определения enum в C/C++. В качестве названий можно также использовать строки или идентификаторы в виде хешированных строк. Функция вывода может просто проверять список активных каналов и выводить только те сообщения, которые входят в их число.

Если количество каналов не превышает 32 или 64, для их идентификации можно применять 32- или 64-битную маску. Благодаря этому фильтрация каналов может быть сведена к заданию одного целого числа. Если бит в маске равен 1, соответствующий канал активен, если 0 — канал приглушен.

Использование Redis для управления отладочными каналами

Разработчики в студии Naughty Dog задействуют веб-интерфейс *Connector* в качестве средства вывода различных потоков отладочной информации, которая генерируется игровым движком во время выполнения. Игра разделяет отладочный текст на разные именованные каналы, каждый из которых относится к отдельной подсистеме (анимация, отрисовка, ИИ, звук и т. д.). Эти потоки данных собираются в Redis — легковесное хранилище типа «ключ — значение» (больше о Redis можно узнать на сайте http://redis.io). Интерфейс Connector дает возможность пользователям с легкостью просматривать и фильтровать содержимое этого хранилища в любом браузере.

10.1.4. Копирование вывода в файл

Копирование всего отладочного вывода в один или несколько журнальных файлов (например, по одному файлу для каждого канала) является хорошей идеей. Это позволяет диагностировать проблемы постфактум. В идеале журнальные файлы должны содержать *весь* вывод, независимо от уровня детализации и маски активных каналов. Таким образом, анализируя самый свежий журнальный файл, вы сможете отлавливать и отслеживать *неожиданные* проблемы.

Возможно, журнальные файлы имеет смысл *сохранять на диск* после каждого вызова функции отладочного вывода, чтобы не потерять последний буфер с полезной информацией, если игра вдруг перестанет работать. Последние выведенные данные чаще всего оказываются наиболее полезными для определения причины поломки, поэтому нужно убедиться в том, что журнальный файл всегда содержит самый свежий вывод. Конечно, сбрасывание буфера на диск может быть ресурсоемкой операцией. Поэтому имеет смысл выполнять ее после каждого отладочного вывода, только если вы не слишком активно ведете журнал или обнаружили, что на вашей конкретной платформе это действительно необходимо. Вы всегда можете предусмотреть в конфигурации своего движка параметр для включения и выключения возможности сбрасывать вывод на диск.

10.1.5. Отчеты о сбоях

Некоторые игровые движки генерируют во время сбоев специальный текстовый вывод и/или журнальные файлы. В большинстве операционных систем есть возможность установить обработчик исключений верхнего уровня, способный отлавливать основную массу сбоев. Это функция, в которой можно выводить разную полезную информацию. Вы даже можете предусмотреть функцию отправки отчета о сбое команде разработчиков по электронной почте. Эти сведения могут оказаться чрезвычайно полезными для программистов: наблюдая за тем, насколько часто со сбоями сталкиваются команды художников и дизайнеров, они могут убедиться в том, что задачи отладки нельзя откладывать!

Вот лишь несколько примеров информации, которую можно включить в отчет о сбое:

- текущий уровень (уровни), на котором находился игрок во время сбоя;
- местоположение игрока в виртуальном мире в момент сбоя;
- какое действие/анимацию выполнял игрок, когда игра вышла из строя;
- скрипт (-ы) игрового процесса, выполнявшийся во время сбоя (это может быть особенно полезным, если причиной сбоя стал скрипт!);
- трассировка стека. Большинство операционных систем предоставляют механизм для навигации по стеку вызовов, хотя такие стеки являются нестандартными и сильно зависят от платформы. С его помощью вы сможете вывести символические имена всех (не подставляемых) функций в стеке на момент сбоя;
- состояние всех аллокаторов памяти в движке (объем свободной памяти, степень фрагментации и т. д.). Такого рода данные могут пригодиться, если, к примеру, ошибка вызвана нехваткой памяти;
- любая другая информация, которая, по вашему мнению, может помочь в отслеживании причин сбоя;
- снимок экрана с игрой в момент, когда произошел сбой.

10.2. Средства отладочной отрисовки

Современные интерактивные игры почти полностью основаны на математических вычислениях. Математика используется для размещения, перемещения и наклона объектов в игровом мире, симуляции столкновений и рейкастинга (чтобы определить линию прямой видимости). И конечно же, мы можем применять произведение матриц для перевода объектов из локального в глобальное пространство с последующей их отрисовкой на экране. Почти все современные игры трехмерные, но даже в двухмерной игре визуализировать результаты всех этих математических вычислений может быть непросто. В связи с этим большинство хороших игровых движков предоставляют API для рисования цветных линий, простых фигур и трехмерного текста. Мы называем это *отладочной отрисовкой*, так как все, что мы выводим на экран с помощью данного механизма, предназначено для визуализации во время разработки и отладки и убирается перед выпуском игры.

API *отладочной отрисовки* может сэкономить вам уйму времени. Например, если вы пытаетесь понять, почему снаряды не попадают по вражеским персонажам, можете сделать одно из двух: проанализировать кучу чисел в отладчике или изобразить траектории снарядов в трехмерном пространстве игры в виде линий. Как думаете, что проще? Благодаря API отладочной отрисовки логические и математические ошибки сразу же становятся очевидными. Как говорится, лучше один раз увидеть, чем сто раз отладить.

Вот несколько примеров того, как отладочная отрисовка используется в движке Naughty Dog. Все снимки экрана, показанные далее, сделаны на одном из

Рис. 10.1. Визуализация игрока в области видимости NPC в игре The Last of Us: Remastered (снимок экрана с сайта https://beedge.neocities.org/), PlayStation 4

множества *тестовых игровых уровней*, которые мы применяем для проверки новых возможностей и отладки проблем в игре.

- На рис. 10.1 показано то, как вражеский NPC воспринимает игрока. Человечек, состоящий из палочек, представляет местоположение персонажа с точки зрения ИИ. Когда игрок исчезает из поля зрения NPC, этот человечек остается там, где его последний раз видели, даже если самому персонажу удается ускользнуть.
- На рис. 10.2 видно, как с помощью каркасной сферы можно визуализировать динамически расширяющийся радиус взрыва.

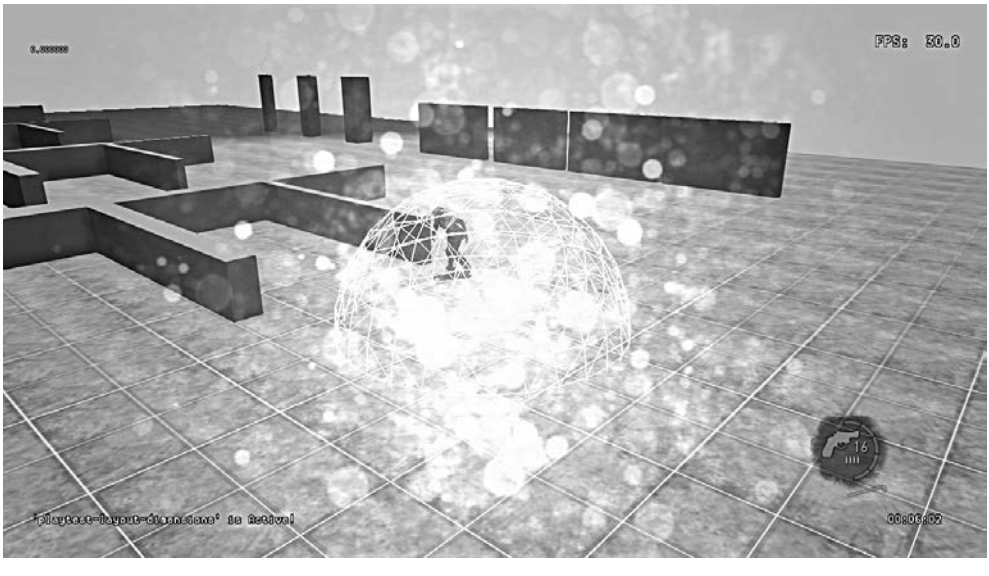


Рис. 10.2. Визуализация расширяющейся сферы взрыва в движке Naughty Dog

- На рис. 10.3 видны круги, с помощью которых можно визуализировать области поиска выступов, за которые может ухватиться Дрейк. Прямая линия обозначает выступ, за который персонаж держится в данный момент.
- На рис. 10.4 показан NPC в специальном режиме отладки. В этом режиме мозг персонажа, по сути, выключается и разработчик получает полный контроль над его движениями и действиями с помощью простого экранного меню. Разработчик может сориентировать камеру в нужном направлении и приказать персонажу пройти или пробежать к нужной точке. Персонажа можно заставить использовать ближайшее укрытие, сделать выстрел и т. д.

Рис. 10.3. Сферы и векторы, которые используются в системе свисания с выступов и перепрыгивания между ними в цикле игр Uncharted (снимок экрана с сайта https://beedge.neocities.org/), PlayStation 3

Рис. 10.4. Ручное управление действиями NPC с целью отладки в игре The Last of Us: Remastered (снимок экрана с сайта https://beedge.neocities.org/), PlayStation 4

10.2.1. API для отладочной отрисовки

API для отладочной отрисовки должен соответствовать следующим требованиям.

- Быть простым и легким в применении.
- Поддерживать набор часто используемых примитивов, в том числе:
  - линии;
  - сферы;
  - точки (обычно представлены небольшими крестиками или сферами, так как один пиксел очень сложно разглядеть);
  - оси координат (ось X обычно красного цвета, Y — зеленого, а Z — синего);
  - ограничивающие параллелепипеды;
  - текст с поддержкой форматирования.
- Быть достаточно гибким и позволять изменять разные аспекты отрисовки:
  - цвет;
  - толщину линий;
  - радиусы сфер;
  - размеры точек, длину осей координат и габариты других встроенных примитивов.
- Позволять рисовать примитивы как в пространстве игрового мира (полностью в 3D, используя матрицу перспективной проекции камеры), так и в плоскости экрана (в ортографической или, возможно, перспективной проекции). Примитивы виртуального пространства подходят для маркировки объектов в трехмерной сцене. Примитивы, привязанные к экрану, позволяют выводить отладочную информацию, которая не зависит от положения или поворота камеры.

- У пользователей должна быть возможность рисовать примитивы с включенной и выключенной *проверкой глубины*.
  - Когда проверка глубины включена, примитивы закрыты реальными объектами сцены. Это делает их глубину более наглядной, но при этом геометрия сцены может частично или полностью скрывать примитивы.
  - Когда проверка глубины выключена, примитивы парят над реальными объектами сцены. Это затрудняет восприятие их настоящей глубины, но при этом всегда видны все примитивы.
- У нас должна быть возможность обращаться к этому API из любого участка кода. Большинство движков отрисовки требуют, чтобы геометрия, которую нужно вывести, была готова на определенном этапе игрового цикла (обычно в конце обработки каждого кадра). Из этого следует, что система должна уметь накапливать все входящие запросы для отладочной отрисовки и отправлять их в подходящий момент.
- В идеале у каждого отладочного примитива должно быть определенное *время жизни*. Это период, на протяжении которого он будет оставаться на экране с момента, когда его запросили. Если код, который рисует примитив, вызывается в каждом кадре, время жизни может составлять один кадр (примитив будет оставаться на экране благодаря постоянному обновлению). Но если код для отрисовки примитива вызывается редко или с перерывами (это может быть функция, вычисляющая начальный вектор скорости снаряда), вам вряд ли захочется, чтобы он один раз мигнул на экране и исчез. В таких ситуациях у программиста должна быть возможность назначать отладочным примитивам более продолжительное время жизни — порядка нескольких секунд.
- Также важно, чтобы система отладочной отрисовки могла эффективно справляться с большим количеством примитивов. При выводе информации для 1000 игровых объектов число примитивов может быть огромным, поэтому при включении этой возможности игра должна оставаться отзывчивой.

API отладочной отрисовки в движке Naughty Dog выглядит примерно так:

```
class DebugDrawManager
{
public:
    // Добавляет прямой отрезок в очередь отладочной отрисовки
    void AddLine(const Point& fromPosition,
                 const Point& toPosition,
                 Color color,
                 float lineWidth = 1.0f,
                 float duration = 0.0f,
                 bool depthEnabled = true);

    // Добавляет в очередь отладочной отрисовки три оси, сходящиеся в точке
    void AddCross(const Point& position,
                  Color color,
```

```
                float size,
                float duration = 0.0f,
                bool depthEnabled = true);

// Добавляет каркас сферы в очередь отладочной отрисовки
void AddSphere(const Point& centerPosition,
                float radius,
                Color color,
                float duration = 0.0f,
                bool depthEnabled = true);

// Добавляет круг в очередь отладочной отрисовки
void AddCircle(const Point& centerPosition,
                const Vector& planeNormal,
                float radius,
                Color color,
                float duration = 0.0f,
                bool depthEnabled = true);

// Добавляет в очередь отладочной отрисовки набор осей координат,
// передающих положение и наклон заданного преобразования
void AddAxes(const Transform& xfm,
             Color color,
             float size,
             float duration = 0.0f,
             bool depthEnabled = true);

// Добавляет контур треугольника в очередь отладочной отрисовки
void AddTriangle(const Point& vertex0,
                 const Point& vertex1,
                 const Point& vertex2,
                 Color color,
                 float lineWidth = 1.0f,
                 float duration = 0.0f,
                 bool depthEnabled = true);

// Добавляет в очередь отладочной отрисовки ограничивающий
// параллелепипед, выровненный по осям координат
void AddAABB(const Point& minCoords,
             const Point& maxCoords,
             Color color,
             float lineWidth = 1.0f,
             float duration = 0.0f,
             bool depthEnabled = true);

// Добавляет в очередь отладочной отрисовки направленный
// ограничивающий параллелепипед
void AddOBB(const Mat44& centerTransform,
            const Vector& scaleXYZ,
            Color color,
```

```
                float lineWidth = 1.0f,
                float duration = 0.0f,
                bool depthEnabled = true);

    // Добавляет текстовую строку в очередь отладочной отрисовки
    void AddString(const Point& pos,
                   const char* text,
                   Color color,
                   float duration = 0.0f,
                   bool depthEnabled = true);
};

// Этот глобальный диспетчер отладочной отрисовки сконфигурирован
// для вывода примитивов в 3D с перспективной проекцией
extern DebugDrawManager g_debugDrawMgr;

// Этот глобальный диспетчер отладочной отрисовки рисует свои примитивы
// на плоскости экрана. Координаты точки (x,y) определяют ее положение
// на экране, а координата z содержит специальный код, сигнализирующий
// о том, какие координаты (x,y) мы используем: абсолютные (пикселы) или
// нормализованные, в диапазоне от 0,0 до 1,0 (во втором случае отрисовка
// не зависит от физического разрешения экрана)
extern DebugDrawManager g_debugDrawMgr2D;
```

Далее представлен пример API, в котором применяется данный код:

```
void Vehicle::Update()
{
    // Производим какие-то вычисления...

    // Рисуем отладочный вектор скорости.
    const Point& start = GetWorldSpacePosition();
    Point end = start + GetVelocity();
    g_debugDrawMgr.AddLine(start, end, kColorRed);

    // Производим какие-то другие вычисления...

    // Выводим имя персонажа и количество пассажиров.
    {
        char buffer[128];
        sprintf(buffer, "Vehicle %s: %d passengers",
                GetName(), GetNumPassengers());

        const Point& pos = GetWorldSpacePosition();
        g_debugDrawMgr.AddString(pos,
            buffer, kColorWhite, 0.0f, false);
    }
}
```

Вы можете заметить, что названия функций отрисовки начинаются с глагола add («добавить»), а не draw («нарисовать»). Дело в том, что отладочные примитивы обычно рисуются не сразу после вызова соответствующей функции. Вместо

этого они добавляются в список визуальных элементов, которые будут нарисованы позже. Большинство высокоскоростных движков 3D-отрисовки требуют, чтобы все визуальные элементы находились в структуре данных *сцены* — это делает возможным их эффективный вывод, который обычно происходит в конце игрового цикла. В главе 11 мы узнаем куда больше о том, как работают движки отрисовки.

10.3. Внутриигровые меню

У любого игрового движка есть множество возможностей и параметров конфигурации. На самом деле все основные подсистемы, включая отрисовку, анимацию, столкновения, физику, звук, сеть, механику управления персонажами, ИИ и др., предоставляют доступ к собственным узкоспециализированным конфигурационным параметрам. Очень важно, чтобы программисты, художники и игровые дизайнеры могли изменять эту конфигурацию на этапе выполнения без модификации исходного кода, повторной компиляции и компоновки исполняемого файла и перезапуска игры. Это может существенно сократить время, которое команда разработчиков тратит на отладку проблем и подготовку новых уровней или игровой механики.

Чтобы сделать этот процесс простым и удобным, можно предоставить систему *внутриигровых меню*. Элементы таких меню должны обладать рядом возможностей, включая следующие:

- переключение глобальных булевых параметров;
- изменение глобальных числовых значений — целых и с плавающей запятой;
- вызов произвольных функций, способных выполнять буквально любую задачу в рамках движка;
- вывод подменю, что делает возможным организацию меню иерархическим, удобным для навигации образом.

Внутриигровое меню должно вызываться простым и удобным способом — возможно, нажатием кнопок на джойстике (конечно, следует выбрать комбинацию, которая не потребуется во время нормальной игры). Вызов меню обычно останавливает игру. Это позволяет разработчику дождаться момента возникновения проблемы, поставить игру на паузу с помощью меню, откорректировать параметры движка, чтобы сделать проблему более наглядной, а затем продолжить игру для более глубокого анализа.

Давайте взглянем на то, как работает система меню в движке Naughty Dog. На рис. 10.5 показано меню верхнего уровня. Оно содержит подменю для всех основных подсистем движка. На рис. 10.6 мы опустились на один уровень вниз, в подменю Rendering (Отрисовка). Поскольку это крайне сложная система, у нее есть много дочерних меню, отвечающих за различные аспекты отрисовки. Чтобы выбрать способ отрисовки трехмерных мешей, переходим вниз еще на один уровень,

в подменю Mesh Options (Параметры мешей) (рис. 10.7). Там мы можем выключить все статические фоновые меши и оставить на экране только динамические, находящиеся на переднем плане. Это продемонстрировано на рис. 10.8 (ага, вот ты где, противный олень!).

Рис. 10.5. Главное меню разработки в игре The Last of Us: Remastered (снимок экрана с сайта https://www.gameenginebook.com)

Рис. 10.6. Подменю Rendering (Отрисовка) в игре The Last of Us: Remastered

Рис. 10.7. Подменю Mesh Options (Параметры мешей) в игре The Last of Us: Remastered

Рис. 10.8. Фоновые меши выключены (The Last of Us: Remastered)

10.4. Внутриигровая консоль

Некоторые движки предоставляют внутриигровую консоль — либо вместо системы внутриигровых меню, либо в дополнение к ней. Она служит командным интерфейсом для доступа к возможностям игрового движка, во многом похожим на *командную строку* DOS, которая позволяет пользователям обращаться к различным функциям операционной системы Windows, или на *командные оболочки* вроде csh, tcsh, ksh или bash, делающие то же самое в ОС семейства UNIX. По аналогии с системой меню консоль игрового движка позволяет разработчику просматривать и изменять глобальные параметры, а также выполнять произвольные команды.

Консоль не так удобна, как система меню, особенно для тех, кто не очень быстро набирает на клавиатуре. В то же время консоль может быть куда более мощной. Некоторые внутриигровые консоли предоставляют лишь небольшой список встроенных команд, что делает их такими же негибкими, как и меню. Но бывают

и те, в которых предусмотрен развитый интерфейс для использования практически любой возможности игрового движка. На рис. 10.9 показан снимок экрана с внутриигровой консолью в *Minecraft*.

Некоторые игровые движки поддерживают мощный скриптовый язык, с помощью которого программисты и игровые дизайнеры могут расширять встроенные возможности или даже создавать совершенно новые игры. Если внутриигровая консоль понимает этот язык, в ней можно делать то же самое, что и в скрипте, только интерактивно. Мы подробно рассмотрим скриптовые языки в разделе 16.9.

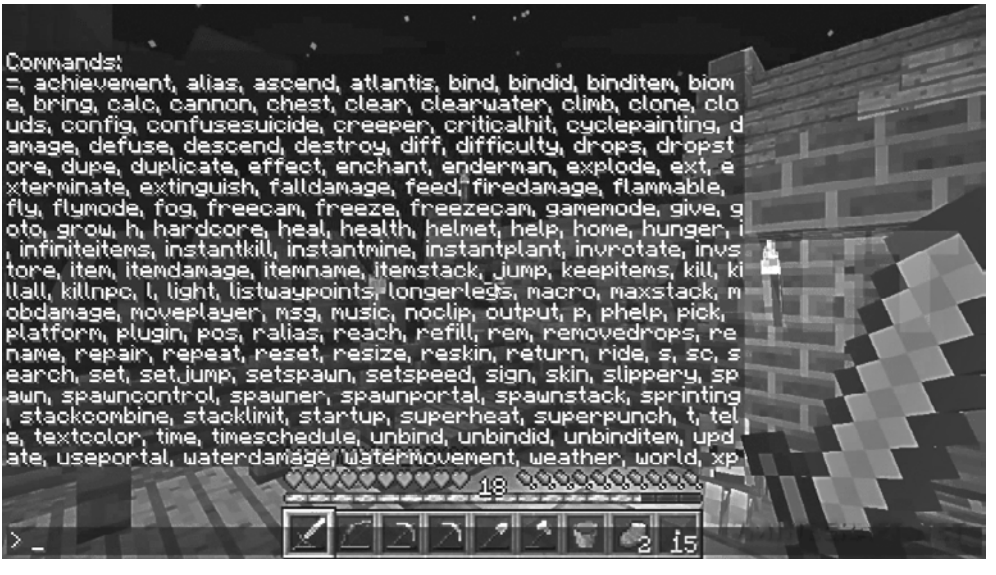


Рис. 10.9. Внутриигровая консоль в Minecraft выводится поверх основного экрана игры и выводит список доступных команд

10.5. Отладочные камеры и остановка игры

В идеале системы внутриигровых меню и консолей должны обладать двумя ключевыми возможностями: отсоединения камеры от персонажа игрока и перемещения ее по игровому миру для осмотра любых аспектов сцены, а также остановки, продолжения и пошагового выполнения игры (см. подраздел 8.5.5). Возможность управления камерой должна сохраняться, даже когда игра находится на паузе. Для этого мы можем позволить движку отрисовки и механизму управления камерой выполняться дальше, несмотря на остановку логического таймера игры.

Еще одной невероятно полезной возможностью для тщательного исследования анимации, эффектов частиц, поведения систем физики и столкновений, работы ИИ и т. д. является режим замедленного воспроизведения. Если позаботиться о том, чтобы все элементы игрового процесса обновлялись с помощью таймера, не привязанного к реальному времени, мы можем переключить игру в режим замедленного воспроизведения, обновляя этот таймер с заниженной частотой. Тот же подход можно использовать и для ускорения движений на экране. Это может пригодиться, когда нужно быстро пройти через длинные участки игрового процесса, чтобы добраться до интересующего места (не говоря уже о том, что это отличный способ для поднятия настроения, особенно в сочетании с плохим вокальным исполнением музыки из шоу Бенни Хилла...).

10.6. Читы

Во время разработки или отладки игры важно, чтобы пользователь в случае необходимости мог нарушать правила. Эта возможность носит меткое название *«читы»* (от cheat — «мошенничать»). Например, многие движки позволяют подхватывать персонажа игрока и перемещать его по игровому миру без учета столкновений, помогая ему проходить сквозь любые преграды. Это может быть чрезвычайно полезно для тестирования игрового процесса. Вместо того чтобы проходить игру, пытаясь добраться до нужного места, вы можете просто подхватить персонажа, доставить его туда, куда вам нужно, и затем возобновить обычный игровой процесс.

Далее перечислены некоторые (далеко не все) читы, которые могут вам пригодиться.

- *Неуязвимость игрока*. Во время тестирования возможностей игры и поиска ошибок разработчики часто не хотят утруждать себя защитой от врагов или беспокоиться о падении со слишком большой высоты.
- *Все оружие*. Часто в ходе тестирования полезно иметь возможность выдавать персонажу игрока оружие любого типа.
- *Бесконечные боеприпасы.* Пытаясь убить плохих парней, чтобы протестировать систему управления оружием и реакцию ИИ на попадания, вам вряд ли захочется тратить время на поиск патронов.
- *Выбор меша для персонажа игрока*. Если у персонажа игрока есть несколько «костюмов», вам может пригодиться возможность выбрать любой из них в целях тестирования.

Очевидно, что этот список можно продолжить на несколько страниц. Возможности здесь безграничны: вы можете добавлять любые читы, которые необходимы для разработки и отладки. Можете даже оставить некоторые из своих любимых читов в финальной версии игры, сделав их доступными для игроков. Для их актива-

ции обычно необходимо ввести неопубликованные *чит-коды* — это можно сделать с помощью джойстика, клавиатуры и/или путем выполнения определенных задач в игре.

10.7. Снимки экрана и запись видео

Еще одной чрезвычайно полезной возможностью является создание снимков экрана и запись их на диск в подходящем формате, таком как Windows Bitmap (`.bmp`), JPEG (`.jpg`) или Targa (`.tga`). Каждая платформа имеет свои особенности реализации данного механизма, но обычно для этого нужно обратиться к графическому API, который позволяет скопировать содержимое экранного буфера из видеопамяти в оперативную память, где его можно просканировать и преобразовать в нужный формат. Файлы изображений обычно записываются в заранее заданную папку на диске, и, чтобы не повторяться, их имена содержат дату и временную метку.

Вы можете дать пользователям возможность управлять различными параметрами создания снимков экрана. Вот несколько распространенных примеров.

- Они могут определить, должен ли снимок экрана содержать отладочные примитивы и текст.
- Могут также определить, должен ли снимок экрана содержать внутриигровые информационные элементы.
- Разрешение снимка. Некоторые движки позволяют захватывать снимки экрана высокого разрешения. Для этого может потребоваться изменение проекционной матрицы, чтобы сделать снимки четырех квадрантов экрана с нормальным разрешением, а затем объединить их в одно большое изображение.
- Простая анимация камеры. Например, вы можете позволить пользователю выбрать начальные и конечные положение/наклон камеры. Затем можно создать цепочку снимков экрана, последовательно интерполируя параметры камеры от начальной к конечной точке.

Некоторые движки предоставляют полноценный режим записи видео. Для этого последовательно с частотой вывода кадров в игре создаются снимки экрана, которые затем обрабатываются (либо позже, либо прямо во время выполнения), чтобы сгенерировать видеофайл в подходящем формате, таком как MPEG-2 (H.262) или MPEG-4 Part 10 (H.264). Но даже если ваш движок не поддерживает захват видео в реальном времени, вы всегда можете захватить вывод своей игровой консоли или ПК с помощью внешнего оборудования, такого как *Roxio Game Capture HD Pro*. А для Windows и Mac доступно огромное множество программных средств захвата видео, включая *Fraps* от Beepa, *Camtasia* от Camtasia Software, *Dxtory* от ExKode, *Debut* от NCH Software и *Action!* от Mirillis.

PlayStation 4 имеет встроенную поддержку публикации внутриигровых снимков экрана и видеоклипов. По ходу игрового процесса PS4 непрерывно захватывает видео последних 15 минут, проведенных пользователем в игре. Пользователь может в любой момент нажать на контроллере кнопку Share (Поделиться) и выбрать один из двух предложенных вариантов: либо сохранить снимок экрана или записанное видео на жесткий диск PS4 или флеш-накопитель, либо загрузить их в один из целого ряда онлайн-сервисов. В Naughty Dog мы используем данные средства для захвата видео и снимка экрана при сбое игры, это позволяет увидеть, что стало причиной неполадок.

Пользователи PlayStation 4 могут также транслировать прохождение игры в реальном времени. В ходе разработки к PS4 можно подключиться с помощью ПК (удаленно или находясь в офисе игровой студии), чтобы получить видеопоток происходящего в игре или даже управлять игрой удаленно с помощью контроллера PS4, подсоединенного к ПК по USB. Эта возможность может оказаться невероятно полезной в ситуациях, когда разработчику не удается воспроизвести ошибку на собственном оборудовании. Таким образом он может отлаживать проблему непосредственно на консоли игрока.

10.8. Внутриигровое профилирование

Игры работают в реальном времени, поэтому достижение и поддержание высокой частоты кадров (обычно 30 или 60 в секунду) имеют большое значение. Следовательно, одна из обязанностей разработчика игры состоит в том, чтобы обеспечить эффективное выполнение своего кода с учетом имеющегося бюджета. Как мы уже видели при обсуждении правила «80/20» в главе 2, существенная часть вашего кода, скорее всего, не нуждается в оптимизации. Чтобы понять, *какие именно* участки необходимо оптимизировать, следует *измерить производительность игры*.

В главе 2 мы рассмотрели различные сторонние средства профилирования. Все они имеют разного рода ограничения и могут быть вообще недоступны на игровой консоли. По этой причине, а также иногда просто для удобства многие игровые движки в том или ином виде предоставляют внутриигровые средства профилирования.

Внутриигровой профайлер, как правило, позволяет программисту помечать блоки кода, работу которых нужно проанализировать, и назначать им удобочитаемые имена. Профайлер измеряет время выполнения каждого такого блока с использованием высокоточного таймера в центральном процессоре и сохраняет результаты в память. При этом на экран выводятся актуальные показатели для каждого блока кода (примеры показаны на рис. 10.10 и 10.11). Данные зачастую предоставляются в нескольких формах, включая непосредственное количество циклов, время выполнения в микросекундах и процент от времени отрисовки всего кадра.

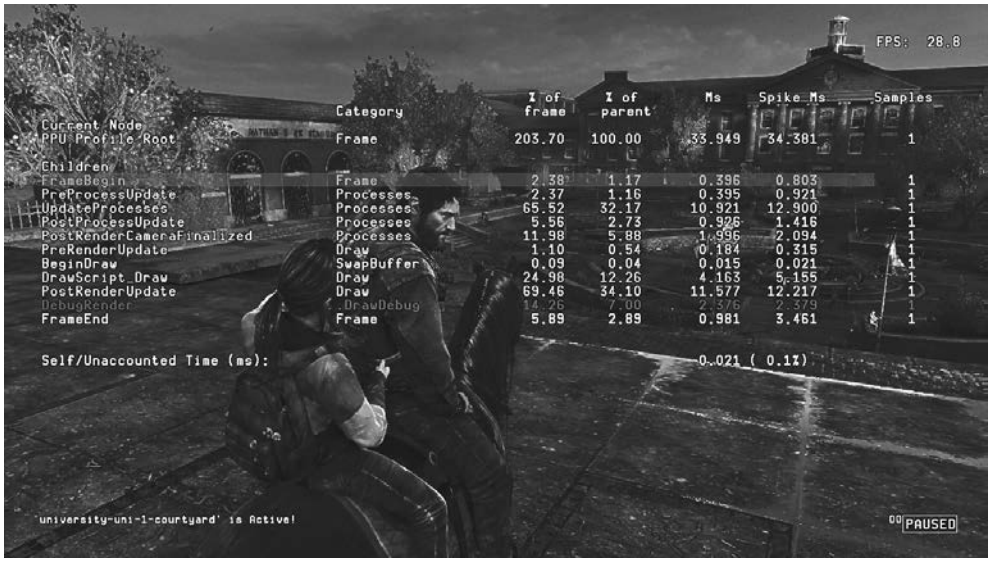


Рис. 10.10. Движок Naughty Dog предоставляет экран профилирования, с помощью которого пользователь может углубиться в иерархию вызовов и проанализировать их ресурсоемкость

10.8.1. Иерархическое профилирование

Компьютерные программы, написанные на императивных языках, являются *иерархическими* по своей природе: одна функция вызывает другую, а та, в свою очередь, — еще какие-то функции. Представьте, к примеру, что функция `a()` вызывает функции `b()` и `c()`, а функция `b()` вызывает `d()`, `e()` и `f()`. Это проиллюстрировано в следующем псевдокоде:

```
void a()
{
    b();
    c();
}

void b()
{
    d();
    e();
    f();
}
```

Рис. 10.11. Режим временной шкалы в The Lost Legacy Uncharted and The Last of Us показывает, когда именно выполняются разные операции в пределах отдельного кадра на одном из семи процессорных ядер PS4

```
void c() { ... }

void d() { ... }

void e() { ... }

void f() { ... }
```

Если предположить, что `a()` вызывается непосредственно из `main()`, иерархия вызовов функций будет выглядеть как на рис. 10.12.

При отладке программы *стек вызовов* показывает лишь снимок этого дерева. В частности, мы можем проследить путь от той функции в иерархии, которая *выполняется в настоящий момент*, до корневой функции дерева. В C/C++ корневой функцией обычно выступает `main()` или `WinMain()`, хотя, строго говоря, она не совсем корневая, так как ее саму вызывает функция начального запуска, которая входит в стандартную библиотеку времени выполнения C (C runtime, или CRT). Если, к примеру, задать точку останова внутри функции `e()`, стек вызовов будет выглядеть примерно так:

```
e()              ← Функция, выполняющаяся в настоящий момент.
b()
a()
main()
_crt_startup()   ← Корень иерархии вызовов.
```

Этот стек вызовов изображен на рис. 10.13 в виде маршрута от функции `e()` к корню иерархии.

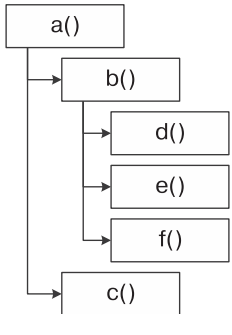


Рис. 10.12. Гипотетическая иерархия вызовов функций

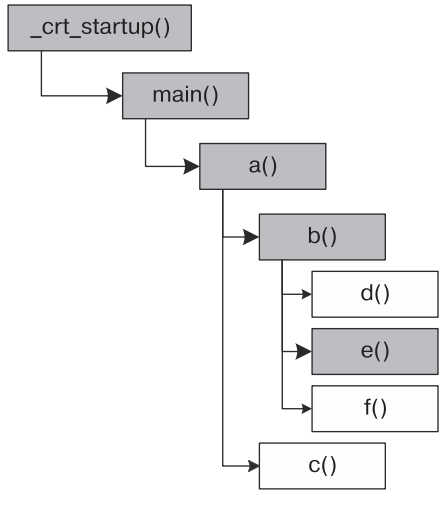


Рис. 10.13. Стек вызовов, который получается при задании точки останова внутри функции e()

Измерение времени выполнения в иерархии вызовов

Если измерить время выполнения отдельно взятой функции, полученный результат будет включать в себя время выполнения всех ее дочерних вызовов, а также потомков первого, второго и всех последующих уровней. Для правильной интерпретации собранных результатов профилирования необходимо принимать во внимание иерархию вызовов.

Многие коммерческие профайлеры могут *автоматически* по отдельности обрабатывать каждую функцию в программе. Это позволяет оценить время выполнения каждой функции, запускаемой в ходе профилирования, как *с учетом* вложенных вызовов, так и *без* них. Во втором случае измеряется только время, проведенное в самой функции (чтобы его получить, можно вычесть суммарное время работы всех потомков первого уровня из общего времени выполнения функции). Кроме того, некоторые профайлеры записывают количество вызовов каждой функции. Эти сведения могут пригодиться при оптимизации программы, позволяя разделять ресурсоемкие функции на две категории: те, которые выполняются долго сами по себе, и те, которые вызываются очень много раз.

Встроенные средства профилирования не обладают настолько продвинутыми возможностями и обычно требуют *ручного* разбора кода. Если главный цикл игрового движка имеет довольно простую структуру, возможно, удастся получить корректные данные, не углубляясь в детали и не анализируя иерархию вызовов. Типичный игровой цикл может выглядеть примерно так:

```
while (!quitGame)
{
    PollJoypad();
    UpdateGameObjects();
    UpdateAllAnimations();
    PostProcessJoints();
    DetectCollisions();
    RunPhysics();
    GenerateFinalAnimationPoses();
    UpdateCameras();
    RenderScene();
    UpdateAudio();
}
```

Для очень поверхностного профилирования такой игры можно было бы измерить время выполнения каждого существенного этапа игрового цикла:

```
while (!quitGame)
{
    {
        PROFILE(SID("Poll Joypad"));
        PollJoypad();
    }
```

```
    {
        PROFILE(SID("Game Object Update"));
        UpdateGameObjects();
    }
    {
        PROFILE(SID("Animation"));
        UpdateAllAnimations();
    }
    {
        PROFILE(SID("Joint Post-Processing"));
        PostProcessJoints();
    }
    {
        PROFILE(SID("Collision"));
        DetectCollisions();
    }
    {
        PROFILE(SID("Physics"));
        RunPhysics();
    }
    {
        PROFILE(SID("Animation Finaling"));
        GenerateFinalAnimationPoses();
    }
    {
        PROFILE(SID("Cameras"));
        UpdateCameras();
    }
    {
        PROFILE(SID("Rendering"));
        RenderScene();
    }
    {
        PROFILE(SID("Audio"));
        UpdateAudio();
    }
    }
```

В реальных условиях макрос `PROFILE()`, представленный ранее, был бы, скорее всего, реализован в виде класса, конструктор которого запускает таймер, а деструктор этот таймер останавливает и записывает время выполнения с заданным именем. Таким образом замеряется лишь работа кода внутри его блока, так как язык C++ автоматически создает и уничтожает объекты по мере того, как они входят в область видимости и покидают ее:

```
struct AutoProfile
{
    AutoProfile(const char* name)
```

```
    {
        m_name = name;
        m_startTime = QueryPerformanceCounter();
    }

    ~AutoProfile()
    {
        std::int64_t endTime = QueryPerformanceCounter();
        std::int64_t elapsedTime = endTime - m_startTime;

        g_profileManager.storeSample(m_name, elapsedTime);
    }

    const char*   m_name;
    std::int64_t  m_startTime;
};

#define PROFILE(name) AutoProfile p(name)
```

Проблема такого упрощенного подхода состоит в том, что он перестает работать на более глубоких уровнях вложенности вызовов. Например, если разместить дополнительные аннотации `PROFILE()` внутри функции `RenderScene()`, для интерпретации полученных результатов необходимо понимать, как устроена иерархия вызовов этой функции.

Чтобы решить эту проблему, можно позволить программисту, который помечает код, описывать иерархические связи между результатами профилирования. Например, любые цифры, полученные внутри функции `RenderScene()` с помощью `PROFILE(...)`, можно объявить дочерними по отношению к результатам измерений `PROFILE(SID("Rendering"))`. Эти связи обычно описываются отдельно от самих аннотаций, для чего измерения заранее разделяются на разные категории. Например, вот как можно настроить внутриигровой профайлер во время инициализации движка:

```
// Этот код объявляет различные категории (sample bins),
// каждая из которых содержит собственное название и название
// своего родителя (если таковой имеется).

ProfilerDeclareSampleBin(SID("Rendering"), nullptr);
    ProfilerDeclareSampleBin(SID("Visibility"), SID("Rendering"));
    ProfilerDeclareSampleBin(SID("Shaders"), SID("Rendering"));
        ProfilerDeclareSampleBin(SID("Materials"), SID("Shaders"));

    ProfilerDeclareSampleBin(SID("SubmitGeo"), SID("Rendering"));

ProfilerDeclareSampleBin(SID("Audio"), nullptr);

// ...
```

У этого подхода есть свои недостатки. В частности, он хорошо работает, когда у каждой функции в иерархии есть всего один родитель, но если попытаться профилировать функцию, которая вызывается из двух разных участков кода, ничего не получится. Причина этого должна быть довольно очевидной. Мы статически объявляем категории так, *будто* каждая функция в иерархии вызовов может существовать только в единственном экземпляре, хотя на самом деле она может встречаться много раз и в каждом новом случае у нее может быть другой родитель. В итоге полученные данные могут быть обманчивыми, так как время работы функции попадает лишь в одну из категорий, хотя в реальности его следовало бы распределить между категориями всех ее родителей. Большинство игровых движков даже не пытаются бороться с этой проблемой, потому что они в основном заинтересованы в профилировании более высокоуровневых функций, которые присутствуют только в каком-то одном участке иерархии вызовов. Однако при использовании встроенных профайлеров, которые можно встретить в большинстве игровых движков, это ограничение следует учитывать.

Конечно, мы могли бы написать куда более продвинутую систему профилирования, способную корректно работать с вложенными экземплярами AutoProfile. Это пример одного из множества компромиссов, неизбежных при проектировании игрового движка. Стоит ли выделять инженерные ресурсы на создание полностью иерархического профайлера или же обойтись чем-то более простым и потратить сэкономленное время на что-то другое? В конечном счете решать вам.

Также неплохо было бы учитывать, сколько *раз* вызывается та или иная функция. В приведенном ранее примере видно, что все функции, которые мы профилировали, вызываются ровно по одному разу. Но на более глубоких уровнях иерархии могут существовать вызовы, выполняемые чаще чем один раз в каждом кадре. Если мы установили, что на работу функции `x()` затрачивается 2 мс, необходимо понимать, что именно имеется в виду: один вызов продолжительностью 2 мс или 1000 таких вызовов на протяжении одного кадра. Отследить количество вызовов функции в одном кадре довольно просто: система профилирования может инкрементировать счетчик при получении результатов и сбрасывать его в начале каждого кадра.

10.8.2. Экспорт в Excel

Некоторые игровые движки позволяют сохранять результаты, полученные внутриигровым профайлером, в текстовый файл для последующего анализа. По моему мнению, лучше всего для этого подходит формат CSV (comma-separated values — «значения, разделенные запятыми»), так как его можно легко загрузить в электронную таблицу Microsoft Excel, где данные можно изменять и анализировать множеством разных способов. Я написал средство экспорта в этот формат для движка *Medal of Honor: Pacific Assault*. Столбцы соответствовали разным аннотированным

блокам, а каждая строка содержала результаты профилирования, полученные в рамках отдельного кадра во время выполнения игры. В первом столбце хранились номера кадров, а во втором — игровое время в секундах. Это позволяло разработчикам строить график изменения показателей производительности на протяжении какого-то времени и определять, сколько на самом деле занимало формирование каждого отдельного кадра. Добавив в экспортированную электронную таблицу простые формулы, мы могли вычислить частоту кадров, процент от общего времени выполнения и т. д.

10.9. Внутриигровые показатели использования памяти и обнаружение утечек

Помимо производительности (то есть частоты кадров), большинство игровых движков ограничены объемом памяти, доступной на целевой аппаратной платформе. Меньше всего это относится к играм для ПК, поскольку современные настольные ОС оснащены продвинутыми диспетчерами виртуальной памяти. Но даже в этом случае необходимо учитывать ограничения, накладываемые так называемыми минимальными системными требованиями: как гарантирует издатель и как утверждается на упаковке, игра обязана работать на компьютере с заданными характеристиками.

В связи с этим большинство игровых движков предоставляют собственные инструменты для отслеживания памяти. Они позволяют разработчикам следить за тем, сколько памяти занимает каждая подсистема движка и есть ли какие-либо утечки (то есть если выделенная память никогда не освобождается). Эти сведения помогут вам принимать обоснованные решения при попытке уменьшить использование памяти, чтобы игра могла уместиться на игровой консоли или компьютере нужной вам конфигурации.

Отслеживание того, сколько памяти на самом деле занимает игра, может оказаться на удивление непростой задачей. Можно было бы подумать, что для учета выделяемой и освобождаемой памяти достаточно просто завернуть `malloc()`/`free()` или `new`/`delete` в пару функций или макросов. Но на практике все всегда сложнее, и тому есть несколько причин.

1. *Зачастую у нас нет возможности контролировать выделение памяти в стороннем коде.* Очень высока вероятность того, что ваша игра в итоге будет скомпонована с какими-нибудь сторонними библиотеками, разве что вы пишете операционную систему, драйверы и игровой движок с нуля. Большинство качественных библиотек предоставляют *хуки выделения памяти*, чтобы вы могли заменить их аллокаторы своим собственным. Но иногда такой возможности

нет. Отслеживать память, выделяемую каждой сторонней библиотекой, которую использует ваш игровой движок, сложно, но *возможно*. Для этого следует тщательно выбирать такие библиотеки.

2. *Память бывает разной*. Например, в ПК есть два вида памяти: основная (RAM) и видеопамять (VRAM, размещена на вашем графическом адаптере и используется в основном для хранения текстур и данных о геометрии). Даже если вам удастся отследить все операции выделения и освобождения, протекающие в RAM, проанализировать применение VRAM будет практически невозможно. Дело в том, что графические API вроде DirectX скрывают от разработчика подробности выделения и использования видеопамяти. В игровых консолях все немного проще благодаря тому, что написание диспетчера VRAM зачастую ложится на вас. Это требует бóльших усилий по сравнению с применением DirectX, но по крайней мере вы получаете полную информацию о происходящем.
3. *Аллокаторы бывают разными*. Многие игры используют специальные аллокаторы, предназначенные для определенных задач. Например, в движке Naughty Dog есть *глобальная куча* для выделения памяти общего назначения, специальная куча для работы с памятью, выделяемой *игровыми объектами* по мере их появления и уничтожения в виртуальном мире, *куча для загрузки уровней*, данные в которую передаются в виде *потока* в ходе игрового процесса, аллокатор стека для *выделения в пределах одного кадра* (стек автоматически очищается в каждом кадре), аллокатор для *VRAM* и *куча отладочной памяти*, предназначенная только для тех операций выделения, которые не попадут в финальную версию игры. Когда игра запускается, каждый из этих аллокаторов занимает большой блок памяти и затем самостоятельно им управляет. Если отслеживать все вызовы new и delete, полученные результаты будут относиться лишь к одному из этих шести блоков. Чтобы получить какую-то полезную информацию, необходимо принимать во внимание все операции выделения внутри блоков памяти, принадлежащих *каждому* аллокатору.

Большинство профессиональных команд, занимающихся разработкой игр, прилагают существенные усилия к созданию в движке средств для отслеживания памяти, способных предоставлять точную и подробную информацию. Эти средства обычно умеют выводить свои данные в разном виде. Например, движок может сгенерировать подробный дамп всех операций выделения памяти, выполненных игрой в заданный период. Эти данные могут содержать сведения о максимальном объеме памяти, выделенном каждым аллокатором или игровой системой, что будет свидетельствовать о максимальном количестве физической RAM, которое им требуется. Некоторые движки также выводят поверх игры индикаторы с данными о занятой памяти. Эти индикаторы могут быть как табличными (рис. 10.14), так и графическими (рис. 10.15).

Рис. 10.14. Статистика использования памяти движком Naughty Dog в табличном виде

Кроме того, когда память близка к исчерпанию или полностью закончилась, качественный движок сообщает об этом как можно более полезным способом. При создании игр для ПК команда разработчиков обычно использует мощные компьютеры, количество RAM в которых больше, чем заявлено в минимальных системных требованиях. Точно так же в создании консольных игр применяются специальные *комплекты для разработки*, которые превосходят по объему памяти потребительские консоли. Поэтому в обоих случаях игра может продолжать работать, даже если она формально исчерпала всю память (то есть больше не помещается на потребительской консоли или ПК с минимальной конфигурацией). В такой ситуации игровой движок может вывести сообщение вида «Недостаточно памяти: этот уровень не будет работать в потребительской системе».

Рис. 10.15. Графический индикатор использования памяти, тоже из The Last of Us: Remastered (снимок экрана с сайта https://beedge.neocities.org/), PlayStation 4

Существует множество других способов, с помощью которых система игрового движка, следящая за памятью, может помочь разработчикам в выявлении проблем

на как можно более ранних этапах и с максимальным удобством. Вот лишь несколько примеров.

- Если модель не удается загрузить, в том месте игрового мира, где она должна была появиться, можно вывести красный объемный текст.
- Если не удается загрузить текстуру, объект можно раскрасить в уродливый розовый цвет, который явно не предназначен для финальной версии игры.
- Если не удается загрузить анимацию, персонаж может принять особую (возможно, забавную) позу, которая сигнализирует об этом факте, а имя недостающего ресурса можно отобразить над головой персонажа.

Ключевыми характеристиками качественных инструментов для анализа памяти являются предоставление точных сведений, вывод данных в удобном виде с явным выделением проблем и наличие контекстной информации, которая помогает разработчикам отслеживать первопричину возникающих неполадок.

Часть III

Графика, движение и звук

11 Движок рендеринга

Первое, что обычно приходит на ум при упоминании компьютеров и видеоигр, — это потрясающая трехмерная графика. Рендеринг в режиме реального времени — чрезвычайно обширная и глубокая тема, поэтому мы просто не сможем рассмотреть все ее тонкости в одной главе. К счастью, существует очень много прекрасных книг и других ресурсов, посвященных ей. Вывод трехмерной графики в режиме реального времени — это, пожалуй, одна из самых хорошо освещенных составляющих игрового движка. Соответственно, данная глава призвана обеспечить общее понимание технологии рендеринга в режиме реального времени и стать своеобразным трамплином для дальнейшего изучения. После ознакомления с этими страницами чтение других книг по 3D-графике будет казаться вам путешествием по знакомой территории. Возможно, вам даже удастся впечатлить своих друзей на одной из вечеринок.

Для начала заложим прочную основу из концепций, теоретических и математических построений, лежащих в основе любого движка 3D-рендеринга в режиме реального времени. Затем рассмотрим программные и аппаратные конвейеры, с помощью которых эти теоретические построения претворяются в реальность. Мы обсудим некоторые наиболее распространенные методы оптимизации и посмотрим, как они влияют на структуру конвейеров и API рендеринга в режиме реального времени, используемых в большинстве движков. Закончит главу обзор некоторых продвинутых методов рендеринга и моделей освещения, применяемых в современных игровых движках. На протяжении данной главы я буду давать вам ссылки на некоторые из моих любимых книг и другие ресурсы, с помощью которых вы сможете глубже понять рассматриваемые здесь темы.

11.1. Основы растеризации треугольников с буферизацией глубины

Если выделить суть этого процесса, то в ходе рендеринга трехмерной сцены выполняются следующие основные шаги.

- Описывается *виртуальная сцена*, обычно в терминах трехмерных поверхностей, представленных в некоторой математической форме.
- *Виртуальная камера* размещается и ориентируется для получения нужного вида сцены. Обычно камера моделируется как идеализированная фокальная точка,

на небольшом расстоянии от которой подвешена поверхность отображения, состоящая из *виртуальных светочувствительных элементов*, соответствующих элементам изображения (пикселам) целевого устройства отображения.

- Определяются *источники света*. Они обеспечивают световые лучи, которые будут взаимодействовать с объектами окружения, отражаться от них и в итоге попадать на считывающую изображение поверхность виртуальной камеры.
- Описываются *визуальные свойства* поверхностей, имеющихся в сцене. Эти свойства определяют, как свет должен взаимодействовать с каждой поверхностью.
- Движок рендеринга вычисляет цвет и интенсивность луча (лучей) света, идущего к фокальной точке камеры через каждый пиксел, составляющий прямоугольник отображения. Этот шаг называют *решением уравнения рендеринга* (или *уравнения затенения*).

Эта высокоуровневая схема рендеринга представлена на рис. 11.1.

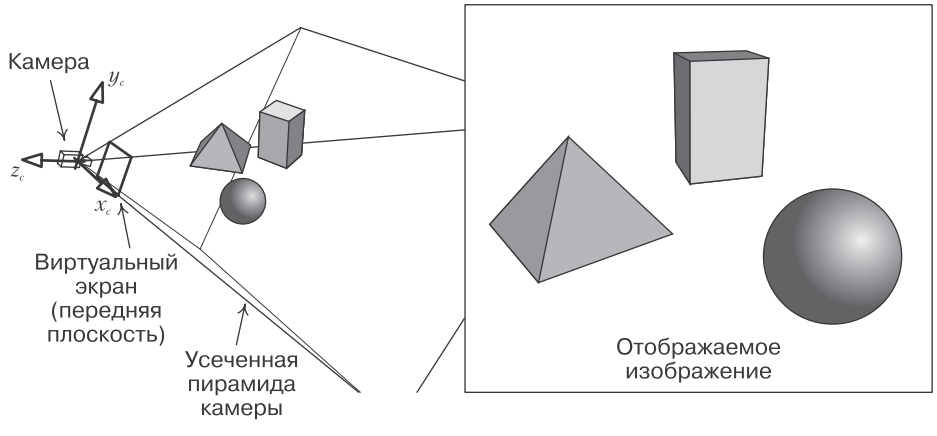


Рис. 11.1. Высокоуровневая схема рендеринга, используемая практически во всех технологиях компьютерной 3D-графики

Для реализации описанных ранее основных шагов рендеринга могут задействоваться самые разные технологии. Первостепенную роль обычно играет *фотореалистичность*, хотя иногда изображению в играх нужно придавать определенную стилизацию (выполнять его в стиле мультфильмов, угольных рисунков, акварельных картин и т. д.). В силу этого художники и инженеры по рендерингу обычно стремятся создавать предельно реалистичное описание свойств сцены и использовать модели переноса света, максимально точно отражающие физическую реальность. В этом смысле спектр технологий рендеринга варьируется от методов, способных работать в режиме реального времени за счет снижения визуальной точности воспроизведения, до методов, которые обеспечивают фотореалистичность, но не рассчитаны на работу в режиме реального времени.

Движки рендеринга в реальном времени циклически выполняют перечисленные ранее шаги, выводя отображаемые изображения с частотой 30, 50 или 60 кадров в секунду для создания иллюзии движения.

Это означает, что движок рендеринга в реальном времени имеет максимум 33,3 мс для генерирования каждого изображения (обеспечения частоты 30 кадров в секунду). Обычно доступно гораздо меньше времени, потому что часть полосы пропускания занимают системы игрового движка, относящиеся к анимации, ИИ, обнаружению столкновений, симуляции физики, звуку, механике игрока и другой логике игрового процесса. Если принять во внимание, что движки рендеринга для киноиндустрии часто затрачивают от нескольких минут до многих часов на рендеринг одного кадра, то качество современной компьютерной графики, отображаемой в режиме реального времени, просто потрясает!

11.1.1. Описание сцены

Сцена реального мира состоит из объектов. Некоторые из них твердые, как кирпич, а некоторые аморфные, как облако дыма, но каждый занимает определенный объем трехмерного пространства. Объект может быть *непрозрачным* (свет не может пройти через его объем), *прозрачным* (свет проходит сквозь него без рассеяния, и мы можем видеть довольно четкое изображение того, что находится позади) или *полупрозрачным* (свет проходит сквозь объект, но при этом рассеивается во всех направлениях, и в результате мы видим размытые цветные пятна, дающие лишь отдаленное представление о том, что находится позади него).

Для рендеринга непрозрачных объектов достаточно принять в расчет их *поверхности*. Поскольку поверхность непрозрачного объекта для света непроницаема, нам не нужно знать, что находится внутри него. Для рендеринга прозрачного или полупрозрачного объекта фактически нужно смоделировать, как свет отражается, преломляется, рассеивается и поглощается по мере прохождения сквозь его объем. Для этого нужно знать внутреннюю структуру и свойства объекта.

Однако большинство игровых движков обходят все эти сложности стороной. Они просто отрисовывают поверхности прозрачных и полупрозрачных объектов практически так же, как и поверхности непрозрачных объектов. Для описания степени прозрачности поверхности используется простой числовой показатель прозрачности — альфа-канал. Этот подход может приводить к различным визуальным аномалиям, например к неправильной отрисовке элементов поверхности в дальней части объекта, но во многих случаях можно найти приближенное значение, обеспечивающее достаточно реалистичный вид. Даже аморфные объекты, такие как облака дыма, часто отображают с помощью эффектов частиц, обычно представляющих собой множество полупрозрачных прямоугольных карточек. Поэтому можно с уверенностью сказать, что большинство игровых движков рендеринга ориентировано главным образом на рендеринг *поверхностей*.

Представления, используемые в профессиональных программных пакетах для рендеринга

Теоретически поверхность представляет собой двухмерное полотно, состоящее из неограниченного числа точек, находящихся в трехмерном пространстве. Однако такое описание, очевидно, не очень пригодно для практики. Чтобы компьютер мог

обрабатывать и отображать произвольные поверхности, нужен компактный способ их численного представления.

Некоторые поверхности можно точно описать в аналитической форме с помощью *параметрического уравнения поверхности*. Так, сфера с центром в начале координат может быть представлена уравнением $x^2 + y^2 + z^2 = r^2$. Однако параметрические уравнения не очень удобно использовать для моделирования произвольных форм.

В киноиндустрии поверхности часто представляют с помощью набора прямоугольных *патчей*, каждый из которых формируется из двухмерного сплайна, определенного небольшим количеством контрольных точек. Используются различные виды сплайнов, в том числе поверхности Безье (например, бикубические патчи, представляющие собой поверхность Безье третьего порядка, — подробнее см. по адресу https://ru.wikipedia.org/wiki/Поверхность_Безье), неоднородные рациональные B-сплайны (NURBS — см. https://ru.wikipedia.org/wiki/B-сплайн), треугольники Безье и N-патчи (также известные как *нормальные патчи* — см. ubm.io/1iGnvJ5). Моделирование поверхности с помощью патчей в какой-то мере напоминает покрытие статуи маленькими прямоугольниками из ткани или папье-маше.

В профессиональных движках рендеринга для киноиндустрии, таких как Pixar RenderMan, геометрические фигуры определяются путем *подразбиения поверхностей*. Каждая поверхность представляется с помощью сети управляющих многоугольников (почти как в случае сплайнов), но их можно последовательно разбивать на многоугольники меньшего размера, используя алгоритм Кэтмулла — Кларка. Обычно процесс разбиения продолжается до тех пор, пока размер отдельных многоугольников не станет меньше одного пиксела. Самое большое преимущество этого подхода состоит в том, что даже при сильном приближении камеры к поверхности изображение всегда можно разбить таким образом, чтобы были сглажены угловатые края контуров. Чтобы узнать больше о подразбиении поверхностей, ознакомьтесь с замечательной статьей: ubm.io/1lx6th5.

Триангулярные меши

Разработчики игр традиционно моделировали поверхности, используя триангулярные меши. При этом треугольники выступают в качестве кусочно-линейной аппроксимации поверхности во многом так же, как цепочка последовательно соединенных линейных отрезков — в качестве кусочно-линейной аппроксимации функции или кривой (рис. 11.2).

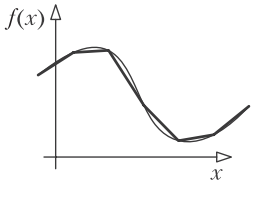


Рис. 11.2. Триангулярный меш является линейной аппроксимацией поверхности точно так же, как цепочка последовательно соединенных линейных отрезков может служить линейной аппроксимацией функции или кривой

Треугольник — наиболее подходящий тип многоугольника для рендеринга в реальном времени, так как обладает следующими полезными свойствами.

- *Треугольник — это самый простой тип многоугольника.* Если количество вершин меньше трех, у нас вообще не будет поверхности.

- *Треугольник всегда плоский.* Многоугольники других типов, с четырьмя или более вершинами, могут и не быть таковыми — если первые три вершины определяют плоскость, четвертая может находиться выше или ниже этой плоскости.
- *Треугольник остается треугольником при выполнении большинства видов преобразования, включая аффинное преобразование и перспективное проецирование.* В самом худшем случае, когда треугольник рассматривается со стороны его ребра, он вырождается в отрезок. При любой другой ориентации остается треугольником.
- *Практически все коммерческое аппаратное обеспечение для ускорения графики разрабатывается на основе алгоритма растеризации треугольников.* Начиная с самых первых ускорителей 3D-графики для ПК, аппаратное обеспечение для рендеринга создается почти исключительно на основе алгоритма растеризации треугольников. Этот выбор был сделан еще во времена первых программных растеризаторов, применявшихся в первых 3D-играх наподобие *Castle Wolfenstein 3D* и *Doom*. Нравится нам это или нет, но технологии, основанные на треугольниках, глубоко укоренились в нашей отрасли, и такое положение, видимо, сохранится надолго.

Тесселяция. *Тесселяция* — это процесс разбиения поверхности на некоторый набор отдельных многоугольников (обычно это либо четырехугольники, или *квадранты*, либо треугольники). Тесселяция поверхности на треугольники называется *триангуляцией*.

Одной из проблем той разновидности триангулярного меша, которая используется в играх, является то, что уровень тесселяции фиксируется художником в момент ее создания. При фиксированной тесселяции контур объекта может выглядеть угловатым (рис. 11.3) — это особенно заметно, когда объект находится близко к камере.

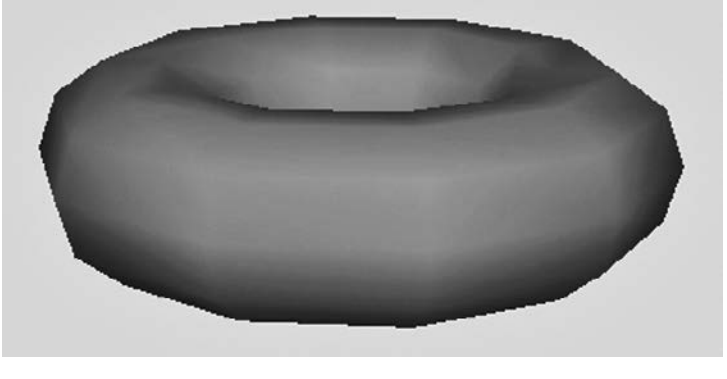


Рис. 11.3. Фиксированная тесселяция может привести к тому, что контур объекта будет выглядеть угловатым, особенно когда объект находится близко к камере

В идеале используемое нами решение должно обеспечивать любое нужное повышение уровня тесселяции при приближении объекта к виртуальной камере. Иными словами, следует обеспечить равномерную попиксельную плотность треугольников вне зависимости от того, близко или далеко находится объект. Воплотить в жизнь этот идеал позволяет подразбиение поверхностей, когда поверхности тесселируются с учетом расстояния до камеры таким образом, чтобы размер каждого треугольника был меньше одного пиксела.

В качестве аппроксимации к этому идеалу равномерной попиксельной плотности треугольников разработчики игр часто создают цепочку альтернативных версий каждого триангулярного меша с разными *уровнями детализации* (level of detail, LOD). Первый уровень LOD, часто называемый LOD 0, соответствует высокому уровню тесселяции и используется, когда объект находится близко к камере. Последующие уровни тесселируются с применением для каждого все более низкого разрешения (рис. 11.4). По мере удаления объекта от камеры движок производит переключение от уровня LOD 0 к уровню LOD 1, затем LOD 2 и т. д. Это позволяет движку рендеринга тратить бóльшую часть своего времени на преобразование и освещение вершин объектов, которые находятся ближе всего к камере и, следовательно, занимают на экране наибольшее количество пикселов.

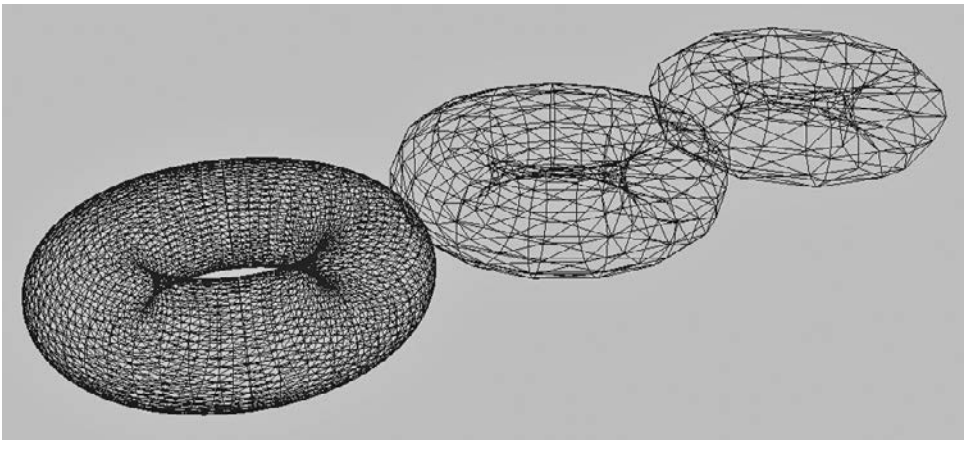


Рис. 11.4. В качестве аппроксимации к равномерной попиксельной плотности треугольников можно использовать цепочку сеток с разными LOD. Тор слева состоит из 5000 треугольников, по центру — из 450, справа — из 200

Некоторые игровые движки применяют методы *динамической тесселяции* к таким обширным мешам, как водные поверхности или ландшафт. При этом меш обычно представляется с помощью поля высот, определенного на некотором типе регулярного меша. Область, расположенная ближе всего к камере, тесселируется до полного разрешения меша. Области, удаленные от камеры, тесселируются с использованием с каждым разом все меньшего количества точек меша.

Еще один подход к динамической тесселяции и уровням детализации состоит в том, чтобы применять *прогрессивные меши*. При этом создается единственный меш высокого разрешения, который отображается, когда объект находится близко к камере. (Фактически это меш уровня LOD 0.) Он автоматически детесселируется при удалении объекта от камеры путем стягивания определенных ребер. Этот процесс автоматически генерирует полунепрерывную цепочку уровней LOD. Подробное описание технологии прогрессивного меша см. по адресу research.microsoft.com/en-us/um/people/hoppe/pm.pdf.

Построение триангулярного меша

Теперь, когда мы понимаем, что представляют собой триангулярные меши и почему они используются, кратко рассмотрим, как они строятся.

Порядок обхода. Треугольник определяется радиус-векторами трех его вершин, которые мы обозначим $\mathbf{p}_1$, $\mathbf{p}_2$ и $\mathbf{p}_3$. Ребра треугольника можно определить как простую разность радиус-векторов соседних вершин, например:

$$\mathbf{e}_{12} = \mathbf{p}_2 - \mathbf{p}_1;$$

$$\mathbf{e}_{13} = \mathbf{p}_3 - \mathbf{p}_1;$$

$$\mathbf{e}_{23} = \mathbf{p}_3 - \mathbf{p}_2.$$

Нормализованное векторное произведение любых двух ребер определяет единичную *нормаль грани* **N**:

$$\mathbf{N} = \frac{\mathbf{e}_{12}\mathbf{e}_{13}}{\left|\mathbf{e}_{12}\mathbf{e}_{13}\right|}.$$

Эти выкладки иллюстрирует рис. 11.5. Чтобы указать *направление* нормали грани, что является физическим смыслом векторного произведения ребер, необходимо определить, какую сторону треугольника следует считать передней, то есть расположенной на внешней поверхности объекта, а какую — задней, то есть внутренней. Это легко сделать, указав *порядок обхода* — по часовой стрелке (clockwise, CW) или против часовой стрелки (counterclockwise, CCW).

Большинство наборов низкоуровневых графических API позволяют *отбраковывать* обращенные назад треугольники на основе порядка обхода. Так, если в Direct3D мы присвоим параметру режима отбраковки `D3DRS_CULL` значение `D3DCULLMODE_CW`, то любой треугольник, вершины которого обходятся в экранном пространстве по часовой стрелке, будет считаться обращенным назад и не станет отрисовываться.

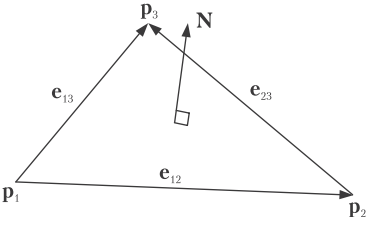


Рис. 11.5. Определение ребер и плоскости треугольника на основе его вершин

Отказ от рендеринга задних граней играет важную роль, потому что обычно лучше не тратить напрасно время на отрисовку треугольников, которые все равно не будут видны. Кроме того, рендеринг задних граней прозрачных объектов может создать визуальные аномалии. Хотя порядок обхода можно выбирать произвольным, он, безусловно, должен быть одинаковым для всех ресурсов игры. Несоблюдение единого порядка обхода — распространенная ошибка начинающих 3D-моделистов.

Списки треугольников. Самый простой способ определения меша состоит в том, чтобы просто перечислить вершины, сгруппировав их по три, где каждая тройка соответствует одному треугольнику. Такая структура данных называется *списком треугольников* (рис. 11.6).

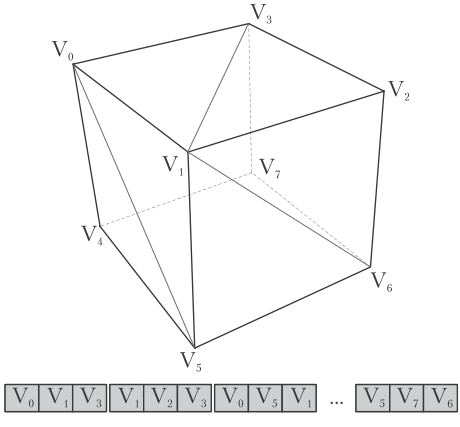


Рис. 11.6. Список треугольников

Индексированные списки треугольников. Как вы, наверное, заметили, многие вершины в списке треугольников, показанном на рис. 11.6, повторяются, часто по нескольку раз. Как мы увидим далее, вместе с каждой вершиной часто хранится большое количество метаданных, поэтому дублирование этих данных в списке треугольников — напрасная трата памяти. При этом также зря расходуется пропускная способность графического процессора, потому что преобразование и освещение продублированной вершины будут выполняться многократно.

В силу этих причин в большинстве движков рендеринга задействуется более эффективная структура данных, известная как *индексированный список треугольников*. Суть этого приема состоит в том, чтобы перечислить вершины один раз, не дублируя, а затем использовать легковесные *индексы* вершин, обычно занимающие лишь 16 бит, для определения троек вершин, составляющих треугольники.

Вершины хранятся в массиве, который называется *буфером вершин* (vertex buffer) в DirectX или *массивом вершин* (vertex array) в OpenGL. Индексы хранятся в отдельном буфере, который называется *индексным буфером* или *индексным массивом* (рис. 11.7).

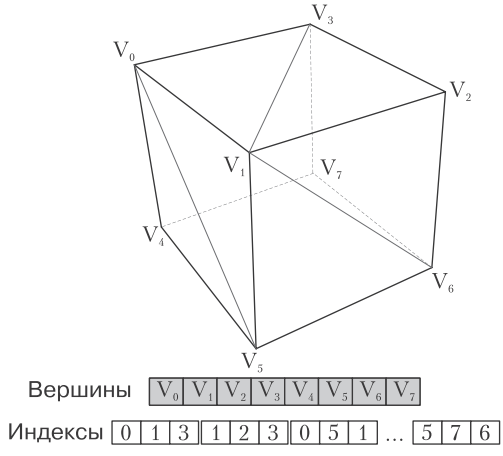


Рис. 11.7. Индексный список треугольников

Полосы и веера. В ходе рендеринга игр иногда используют специализированные структуры данных для сеток, известные как *полосы треугольников* и *веера треугольников*. Обе структуры позволяют в некоторой степени снизить дублирование вершин без помощи индексного буфера. Это обеспечивается за счет предварительного определения порядка расположения вершин и способа их объединения в треугольники.

В случае полосы первые три вершины определяют первый треугольник. Каждая последующая вершина образует совершенно новый треугольник совместно с двумя предыдущими вершинами. Для того чтобы в полосе треугольников сохранялся единый порядок обхода, две предыдущие вершины меняют местами при добавлении каждого нового треугольника. Полоса треугольников показана на рис. 11.8.

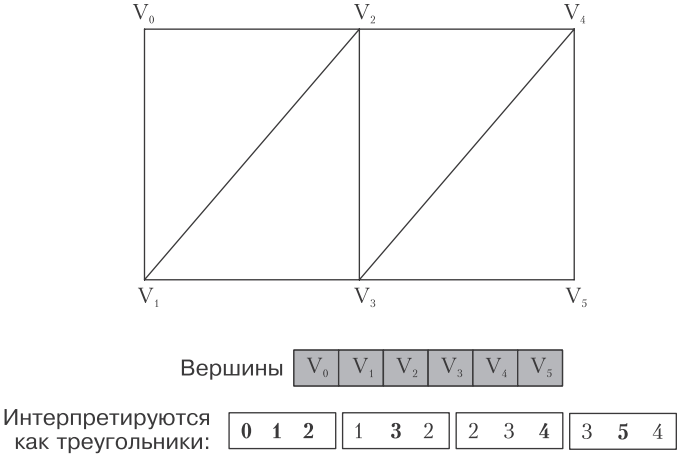


Рис. 11.8. Полоса треугольников

В случае веера первые три вершины определяют первый треугольник, а каждая последующая вершина образует новый треугольник совместно с предыдущей вершиной и первой вершиной веера (рис. 11.9).

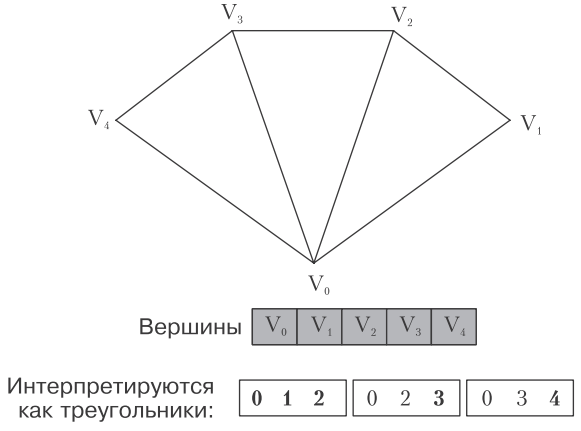


Рис. 11.9. Веер треугольников

Оптимизация кэша вершин. Когда графический процессор обрабатывает индексированный список треугольников, каждый треугольник может ссылаться на любую вершину в буфере вершин. Обработка вершин должна выполняться в порядке их появления в треугольниках, чтобы обеспечить целостность каждого из них на стадии растеризации. Поскольку обработку вершин выполняет вершинный шейдер, они кэшируются для повторного использования. Если последующий примитив ссылается на вершину, которая уже находится в кэше, то вместо того, чтобы повторно обрабатывать вершину, применяются ее обработанные атрибуты.

Полосы и веера используются отчасти потому, что потенциально помогают экономить память за счет работы без индексного буфера, а отчасти потому, что обычно позволяют улучшить когерентность кэша для обращений графического процессора к видеопамяти. Что еще лучше, мы можем задействовать *индексированную полосу* или *индексированный веер*, которые вдобавок к обеспечению преимуществ когерентности кэша за счет полосного или веерного порядка вершин практически исключают дублирование вершин, а это часто дает большую экономию памяти, чем устранение индексного буфера.

Оптимизировать кэш для индексированных списков треугольников можно и не ограничиваясь полосным или веерным порядком вершин. *Оптимизатор вершинного кэша* представляет собой автономный инструмент для обработки геометрии, который упорядочивает список треугольников таким образом, чтобы обеспечивалось оптимальное повторное использование вершин в кэше. Обычно в числе прочего он учитывает, какого размера вершинные кэши имеются у конкретного графического процессора и с помощью каких алгоритмов графический процессор определяет,

когда следует кэшировать вершины, а когда — удалять их из кэша. Так, например, оптимизатор вершинного кэша, включенный в библиотеку для обработки геометрии Edge компании Sony, способен обеспечивать производительность рендеринга на 4 % выше по сравнению с применением полос треугольников.

Пространство модели

Позиционные векторы вершин триангулярного меша обычно задаются относительно удобной локальной системы координат, называемой *пространством модели*, *локальным пространством* или *пространством объекта*. Начало координат пространства модели обычно находится либо в центре объекта, либо в другом удобном месте, например на полу между ногами персонажа или на земле в горизонтальной центроиде колес транспортного средства.

Как мы узнали в подразделе 5.3.9, осям пространства модели можно придать произвольный физический смысл, который, однако, обычно соответствует естественным направлениям движения модели вперед, влево, вправо и вверх. Чтобы обеспечить некоторую степень математической строгости, мы можем определить единичные векторы **L** (или **R**), **U** и **F** и отобразить их нужным образом на единичные базисные векторы **i**, **j** и **k** (и соответственно на оси *X*, *Y* и *Z*) в пространстве модели. Так, один из распространенных способов отображения выглядит следующим образом: **L** = **i**, **U** = **j** и **F** = **k**. Хотя способ отображения можно выбирать произвольным образом, важно, чтобы он был одинаковым для всех обрабатываемых движком моделей. На рис. 11.10 показан один из возможных способов отображения осей пространства модели для модели самолета.

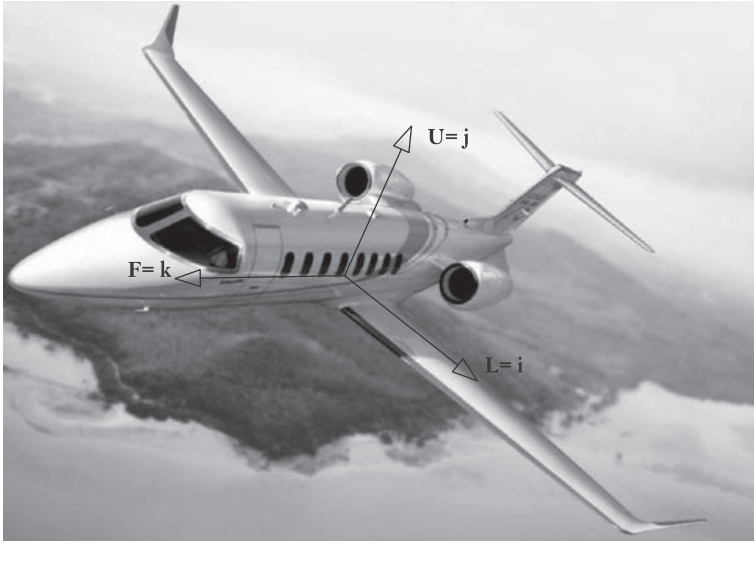


Рис. 11.10. Один из возможных способов отображения осей пространства модели

Мировое пространство и создание экземпляров сеток

Множество отдельных сеток объединяется в целостную сцену путем их размещения и ориентации в общей системе координат, называемой *мировым пространством*. Любой меш может использоваться в сцене многократно — примерами могут служить улица с множеством одинаковых фонарных столбов, безликая толпа солдат или рой атакующих игрока пауков. Каждый такой объект называют *экземпляром меша*.

Экземпляр меша содержит ссылку на разделяемые данные меша, а также матрицу преобразования, которая преобразует вершины меша из пространства модели в мировое пространство в контексте этого конкретного экземпляра. Эту матрицу называют матрицей *преобразования модели в мир*, или просто *мировой матрицей*. Используя обозначения из подраздела 5.3.10, можно записать эту матрицу следующим образом:

$$\mathbf{M}_{M\to W} = \begin{bmatrix} (\mathbf{RS})_{M\to W} & 0 \\ \mathbf{t}_M & 1 \end{bmatrix},$$

где верхняя матрица 3×3 $(\mathbf{RS})_{M\to W}$ обеспечивает вращение и масштабирование вершин пространства модели при их преобразовании в мировое пространство, а $\mathbf{t}_M$ — трансляция осей пространства модели, выраженная в координатах пространства мира. Если у нас есть единичные базисные векторы пространства модели $\mathbf{i}_M$, $\mathbf{j}_M$ и $\mathbf{k}_M$, выраженные в координатах мирового пространства, эту матрицу также можно записать следующим образом:

$$\mathbf{M}_{M\to W} = \left[\begin{array}{c|c} \mathbf{i}_M & 0 \\ \mathbf{j}_M & 0 \\ \mathbf{k}_M & 0 \\ \hline \mathbf{t}_M & 1 \end{array}\right].$$

Для заданной вершины, выраженной в координатах пространства модели, движок рендеринга вычисляет эквивалент в мировом пространстве следующим образом:

$$\mathbf{v}_W = \mathbf{v}_W \mathbf{M}_{M\to W}.$$

Матрицу $\mathbf{M}_{M\to W}$ можно рассматривать как описание положения и ориентации осей пространства модели, выраженное в координатах мирового пространства, либо как матрицу, преобразующую вершины из пространства модели в мировое пространство.

При рендеринге меша мировая матрица применяется также к нормалям его поверхности (см. подраздел 11.1.2). Если вы помните, в подразделе 5.3.11 говорилось, что для правильного преобразования нормальных векторов их необходимо умножать на обратную транспозицию мировой матрицы. Когда матрица не выполняет масштабирование или обрезку, можно корректно преобразовать нормальные

векторы, просто установив в ноль их компоненты W перед умножением на мировую матрицу, как описано в подразделе 5.3.6.

Меши некоторых фоновых элементов, таких как здания и ландшафт, абсолютно статичны и уникальны. Вершины этих сеток часто выражают в координатах мирового пространства, в результате чего их матрицы преобразования модели в мир являются тождественными и могут игнорироваться.

11.1.2. Описание визуальных свойств поверхности

Для правильного рендеринга и освещения поверхности необходимо располагать описанием ее *визуальных свойств*. В их число входит такая геометрическая информация, как направление нормали поверхности в различных ее точках. Сюда также входит описание того, как свет должен взаимодействовать с поверхностью, включая данные о диффузном цвете, коэффициенте блеска/отражения, шероховатости или текстуре, степени непрозрачности или прозрачности, показателе преломления и других оптических свойствах. Свойства поверхности могут включать в себя также определение того, как она должна изменяться с течением времени (например, как должна двигаться поверхность водоема или кожа анимированного персонажа при движении суставов его скелета).

Ключевым моментом при рендеринге фотореалистичных изображений является правильный учет поведения света при его взаимодействии с объектами сцены. Поэтому инженеры по рендерингу должны хорошо понимать, как ведет себя свет, как он распространяется в окружающей среде и как виртуальная камера воспринимает его и транслирует в цвета, сохраняемые в виде экранных пикселов.

Введение в свет и цвет

Свет — это электромагнитное излучение, которое в различных ситуациях может проявляться и как волна, и как частица. Цвет света определяется его *интенсивностью* I и *длиной волны* λ (или частотой f, где $f = 1 / \lambda$). К видимому диапазону относится свет с длиной волны от 740 нм (или частотой 430 ТГц) до 380 нм (750 ТГц). Луч света может состоять из света исключительно одной длины волны (то есть одного из цветов радуги, также называемых *спектральными цветами*) или представлять собой смесь из света различных длин волн. С помощью графика, называемого *спектральной диаграммой*, можно показать, какое количество света каждой частоты присутствует в заданном луче света. В белом свете в определенной степени присутствуют все длины волн, поэтому его спектральная диаграмма будет выглядеть примерно как прямоугольник, занимающий весь видимый диапазон. Чистый зеленый свет содержит только одну длину волны, поэтому его спектральная диаграмма будет выглядеть как один бесконечно узкий пик с частотой около 570 ТГц.

Взаимодействие света и объекта. Свет может очень разнообразно и сложно взаимодействовать с веществом. Его поведение отчасти определяется *средой*, через

которую он проходит, а отчасти — формой и свойствами *границ раздела* между средами разного типа (воздухом и твердым телом, воздухом и водой, водой и стеклом и т. д.). Формально любая поверхность, по сути, является просто границей раздела между двумя средами разного типа.

Несмотря на всю сложность света, с ним могут происходить только следующие четыре вещи:

- он может *поглощаться*;
- он может *отражаться*;
- он может *проходить* через объект, обычно *преломляясь* при этом;
- он может *рассеиваться* при прохождении сквозь очень узкие отверстия.

В большинстве движков для фотореалистичного рендеринга принимаются в расчет первые три варианта поведения, преломление света обычно не учитывается, поскольку оно, как правило, не производит заметного эффекта в большинстве сцен.

Любая поверхность поглощает только свет с некоторыми определенными длинами волн, отражая свет с другими длинами волн. Благодаря этому мы можем воспринимать цвета объектов. Так, например, когда белый свет падает на красный объект, поглощается свет со всеми длинами волн, за исключением присущей красному, и именно поэтому объект выглядит красным. Точно такой же эффект можно получить, освещая красным светом белый объект, — наши глаза не заметят никакой разницы.

Отражения могут быть *размытыми*, в этом случае входящий луч равномерно рассеивается во всех направлениях. Отражения могут быть *зеркальными*, в этом случае падающий луч света отражается напрямую или расширяется лишь до узкого конуса. Также отражения могут быть *анизотропными*, в этом случае способ отражения света от поверхности меняется в зависимости от того, под каким углом наблюдатель смотрит на нее.

Когда свет проходит через некоторый объем, он может *рассеиваться* (если объект полупрозрачен), частично *поглощаться* (например, цветным стеклом) или *преломляться* (при прохождении света через призму). В силу того что угол преломления зависит от длины волны, при преломлении света происходит его спектральное разложение. Именно поэтому мы наблюдаем радужный эффект при прохождении света через дождевые капли или стеклянные призмы. Также свет может войти в полутвердую поверхность, отразиться внутри объекта и выйти на поверхность в другой точке. Это так называемое *подповерхностное рассеяние* — эффект, который придает характерный теплый вид коже, воску и мрамору.

Цветовые пространства и цветовые модели. *Цветовая модель* — это трехмерная система координат для измерения цветов. *Цветовое пространство* — это определенный стандарт отображения числовых значений цвета в конкретной цветовой модели на цвета, воспринимаемые людьми в реальном мире. Цветовые модели обычно трехмерные, поскольку в наших глазах имеются три типа чувствительных к цвету колбочек, воспринимающих свет с разными длинами волн.

Наибольшее распространение в компьютерной графике получила цветовая модель RGB. В ней цветовое пространство представляется единичным кубом, вдоль осей которого откладываются относительные значения интенсивности красной, зеленой и синей составляющих света. Эти составляющие называются *цветовыми каналами*. В классической модели RGB-значения каждого канала варьируются в диапазоне от нуля до единицы. Значение (0, 0, 0) при этом соответствует черному цвету, а значение (1, 1, 1) — белому.

При сохранении цветов в растровом изображении могут использоваться различные форматы цвета. Формат цвета отчасти определяется тем *количеством битов на пиксел*, которое он занимает, а точнее, количеством битов, представляющих каждый цветовой канал. Так, формат RGB888 задействует 8 бит на каждый канал, что в сумме дает 24 бита на пиксел. Значения каждого канала при этом варьируются в диапазоне от 0 до 255, а не от 0 до 1. Формат RGB565 использует по 5 бит для красного и синего каналов и 6 бит для зеленого, что в сумме составляет 16 бит на пиксел. Формат палитры может использовать 8 бит на пиксел для хранения индексов, указывающих на 256 элементов цветовой палитры, каждый из которых может быть представлен в формате RGB888 или другом подходящем.

Для 3D-рендеринга используется также ряд других цветовых моделей. В подразделе 11.3.1 мы рассмотрим применение цветовой модели log-LUV для освещения *с расширенным динамическим диапазоном* (high dynamic range, HDR).

Непрозрачность и альфа-канал. Цветовые векторы RGB часто дополняются четвертым, так называемым альфа-каналом. Как упоминалось в подразделе 11.1.1, альфа-канал служит для оценки непрозрачности объекта. Значение альфа-канала, сохраняемое в пикселе изображения, представляет степень непрозрачности этого пиксела.

Цветовые форматы RGB могут расширяться с включением в них альфа-канала, расширенный формат называется RGBA или ARGB. Так, например, RGBA8888 — это формат с 32 битами на пиксел — по 8 бит на красный, зеленый, синий и альфа-каналы. RGBA5551 — это формат с 16 битами на пиксел, включая 1 бит на альфа-канал. В этом случае цвета могут быть либо совершенно непрозрачными, либо полностью прозрачными.

Атрибуты вершин

Простейший способ описания визуальных свойств поверхности состоит в том, чтобы указать эти свойства в ряде отдельных точек поверхности. Удобным местом для хранения свойств поверхности являются вершины меша, в этом случае их называют *атрибутами вершин*. Обычно каждая вершина триангулярного меша обладает некоторыми из рассмотренных далее атрибутов или всеми ими. Конечно, как инженеры по рендерингу, мы всегда можем определить любые дополнительные атрибуты, необходимые для получения на экране нужного визуального эффекта.

- *Позиционный вектор* $\mathbf{p}_i = [p_{ix}\ p_{iy}\ p_{iz}]$. Это вектор пространственного положения i-й вершины в меше. Обычно он указывается в локальном координатном пространстве объекта, называемом *пространством модели*.

- *Нормаль* вершины $\mathbf{n}_i = [n_{ix}\ n_{iy}\ n_{iz}]$. Этот вектор определяет единичную нормаль поверхности в вершине i. Он используется для повершинных динамических расчетов освещения.
- *Касательная* $\mathbf{t}_i = [t_{ix}\ t_{iy}\ t_{iz}]$ и *бикасательная* $\mathbf{b}_i = [b_{ix}\ b_{iy}\ b_{iz}]$ вершины. Это два единичных вектора, расположенных перпендикулярно по отношению друг к другу и к нормали вершины $\mathbf{n}_i$. Векторы $\mathbf{n}_i$, $\mathbf{t}_i$ и $\mathbf{b}_i$ совместно определяют набор координатных осей, называемый *касательным пространством*. Это пространство используется для различных попиксельных расчетов освещения, например наложения карт нормалей и карт окружения. (Бикасательную $\mathbf{b}_i$ иногда называют *бинормалью*, что несколько сбивает с толку, потому что она *не является* нормальной к поверхности.)
- *Диффузный цвет* $\mathbf{d}_i = [d_{iR}\ d_{iG}\ d_{iB}\ d_{iA}]$. Этот четырехэлементный вектор описывает диффузный цвет поверхности, выраженный в цветовом пространстве RGB. Обычно он включает в себя также спецификацию степени непрозрачности или значения *альфа-канала* (A) поверхности в вершине. Этот цвет может рассчитываться в автономном режиме (статическое освещение) или динамически (динамическое освещение).
- *Зеркальный цвет* $\mathbf{s}_i = [s_{iR}\ s_{iG}\ s_{iB}\ s_{iA}]$. Этот вектор описывает цвет зеркальной подсветки, которая должна появляться, когда свет напрямую отражается от блестящей поверхности на плоскость отображения виртуальной камеры.
- *Текстурные координаты* $\mathbf{u}_{ij} = [u_{ij}\ v_{ij}]$. *Текстурные координаты* позволяют натянуть двухмерное, а иногда и трехмерное растровое изображение на поверхность меша — данный процесс называют *наложением текстурных карт* или *текстурированием*. Текстурные координаты (u, v) указывают положение конкретной вершины в двухмерном нормированном координатном пространстве текстуры. На треугольник можно отобразить несколько текстур, соответственно, он может иметь несколько наборов текстурных координат. В приведенном ранее обозначении различные наборы текстурных координат показаны индексом j.
- *Весовые коэффициенты скининга* $\mathbf{k}_{ij} = [k_{ij}\ w_{ij}]$. В скелетной анимации вершины меша прикрепляются к отдельным суставам шарнирно сочлененного скелета. Каждая вершина должна указывать, к какому суставу она прикреплена, с помощью индекса k. На вершину могут воздействовать несколько суставов, и в этом случае окончательное ее положение определяется *средневзвешенной величиной* этих воздействий. Соответственно, влияние каждого сустава описывается с помощью весового коэффициента w. В общем случае вершина i может подвергаться нескольким воздействиям суставов j, каждое из которых описывается с помощью пары чисел (k_{ij}, w_{ij}).

Форматы вершин

Атрибуты вершин обычно хранятся в структуре данных, такой как структура (`struct`) языка C или класс языка C++. Компоновка такой структуры данных называется *форматом вершин*. Для разных сеток необходимы разные комбинации

атрибутов, что требует использования разных форматов вершин. Вот несколько примеров распространенных форматов вершин:

```
// Самый простой формат вершины – только положение
// полезно при выдавливании теневых объемов,
// обнаружении краев контуров, для рендеринга
// в стиле мультфильмов, z-предпрохода и т. д.)
struct Vertex1P
{
    Vector3  m_p;  // положение
};

// Типичный формат вершины с положением, нормалью вершины и одним набором
// текстурных координат.
struct Vertex1P1N1UV
{
    Vector3 m_p;     // положение
    Vector3 m_n;     // нормаль вершины
    F32     m_uv[2]; // текстурные координаты (u, v)
};

// Заскиненная вершина с положением, диффузным цветом, зеркальным
// цветом и четырьмя взвешенными воздействиями суставов.
struct Vertex1P1D1S2UV4J
{
    Vector3 m_p;  // положение
    Color4 m_d;   // диффузный цвет и прозрачность
    Color4 m_S;   // зеркальный цвет
    F32 m_uv0[2]; // первый набор текстурных координат
    F32 m_uv1[2]; // второй набор текстурных координат
    U8 m_k[4];    // четыре индекса суставов и...
    F32 m_w[3];   // три весовых коэффициента суставов для скиннинга
                  // (четвертый рассчитывается на основе первых трех)
};
```

Очевидно, что число возможных комбинаций атрибутов вершин — и, соответственно, возможных форматов вершин — чрезвычайно велико. (Теоретически количество возможных форматов становится неограниченным, если допустить использование любого количества текстурных координат и/или весовых коэффициентов суставов.) Жонглирование всеми этими форматами вершин — неизменный источник головной боли для любого программиста компьютерной графики.

Можно принять определенные меры по снижению количества форматов вершин, поддерживаемых движком. В существующих графических приложениях многие из теоретически возможных форматов вершин просто неудобны для использования или не могут быть обработаны графическим аппаратным обеспечением либо шейдерами игры. Чтобы как-то облегчить себе жизнь, разработчики игры иногда ограничиваются определенным подмножеством удобных/практически реализуемых форматов вершин. Например, они могут условиться, что у вершины должно быть только ноль, два или четыре весовых коэффициента суставов или не больше двух наборов текстурных координат. Поскольку некоторые графические

процессоры могут извлекать подмножество атрибутов из вершинной структуры данных, разработчики игры также могут использовать единый сверхформат для всех сеток, позволив оборудованию выбирать нужные атрибуты, исходя из требований шейдера.

Интерполяция атрибутов

Атрибуты в вершинах треугольника представляют собой грубое дискретное приближение к визуальным свойствам поверхности в целом. Действительно важную роль при рендеринге треугольника играют визуальные свойства тех его внутренних точек, которые видны через каждый экранный пиксел. То есть нам нужно знать значения атрибутов на *попиксельной*, а не на повершинной основе.

Простейший способ определения попиксельных значений атрибутов поверхности меша состоит в том, чтобы *линейно интерполировать* повершинные данные атрибутов. Применение интерполяции атрибутов к цветам вершин называется *затенением (шейдингом) по Гуро*. На рис. 11.11 показан пример применения затенения по Гуро к треугольнику, а рис. 11.12 демонстрирует, как этот метод может воздействовать на простой триангулярный меш. Интерполяция часто применяется и к другим видам информации об атрибутах вершин, например вершинным нормалям, текстурным координатам и глубине.

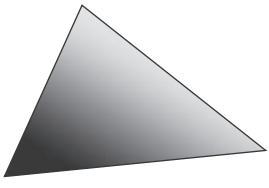


Рис. 11.11. Треугольник, затененный в вершинах по Гуро различными оттенками серого цвета

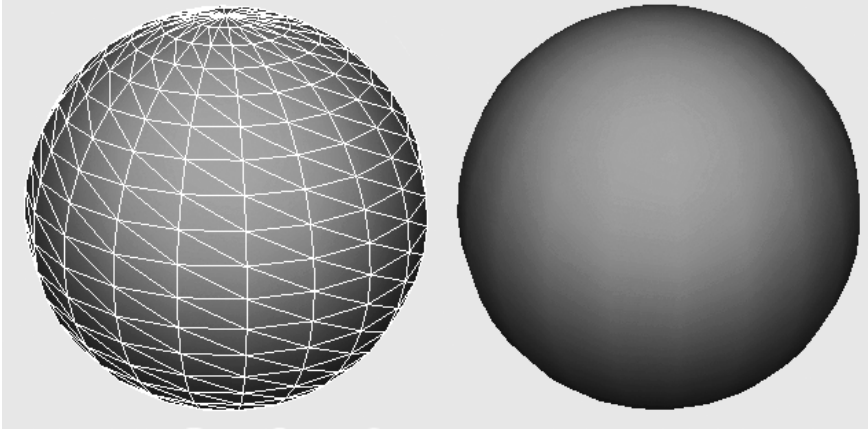


Рис. 11.12. Затенение по Гуро позволяет сделать угловатые объекты гладкими

Вершинные нормали и сглаживание. Как мы увидим в подразделе 11.1.3, *освещение* — это расчет цвета объекта в различных точках его поверхности на основе ее визуальных свойств и свойств падающего света. Простейший способ освещения меша состоит в том, чтобы выполнить *повершинный* расчет цвета поверхности. В этом случае на основе свойств поверхности и падающего света рассчитывается

диффузный цвет каждой вершины ($\mathbf{d}_i$), а затем цвета вершин интерполируются по треугольникам меша с помощью затенения Гуро.

Для определения того, как следует отражать луч света от точки поверхности, в большинстве моделей освещения используется вектор, *нормальный* к поверхности в точке падения светового луча. Поскольку мы рассчитываем освещение на повершинной основе, можем применять для этого вершинные нормали $\mathbf{n}_i$. При этом окончательный вид меша будет в значительной степени определяться направлением вершинных нормалей меша.

В качестве примера рассмотрим высокую узкую коробку, которая, естественно, имеет четыре стороны. Если нужно получить у нее четко очерченные края, можно определить вершинные нормали, перпендикулярные граням коробки. В таком случае при освещении каждого треугольника мы будем встречать один и тот же вектор нормали во всех трех вершинах, в результате чего полученное освещение будет выглядеть плоским и резко изменяющимся на углах коробки в соответствии с изменением вершинных нормалей.

Тот же меш коробки можно сделать больше похожим на гладкий цилиндр, определив нормали вершин, направленные радиально наружу от осевой линии коробки. В этом случае вершины всех треугольников будут иметь разные нормали, что даст различные цвета вершин. Затенение Гуро плавно интерполирует цвета вершин, в результате чего освещение будет плавно изменяться по всей поверхности. Этот эффект иллюстрирует рис. 11.13.

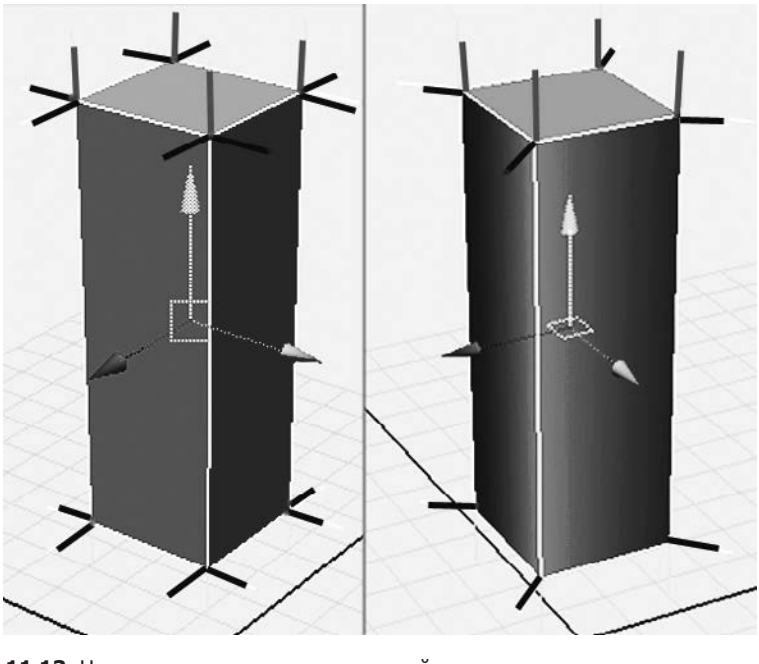


Рис. 11.13. Направление вершинных нормалей меша может сильно влиять на цвета, получаемые при повершинном расчете освещения

Текстуры

При сравнительно большом размере треугольников определение свойств на поверхностной основе может оказаться слишком грубым приближением. Линейная интерполяция атрибутов не всегда обеспечивает то, что нам нужно, зачастую приводя к нежелательным визуальным аномалиям.

В качестве примера рассмотрим задачу рендеринга яркого *зеркального блика*, возникающего при падении света на блестящий объект. При высоком уровне тесселяции меша поверхностное освещение в сочетании с затенением по Гуро обеспечит довольно хорошие результаты. Однако при слишком больших треугольниках погрешности линейной интерполяции зеркального блика могут стать раздражающе очевидными (рис. 11.14).

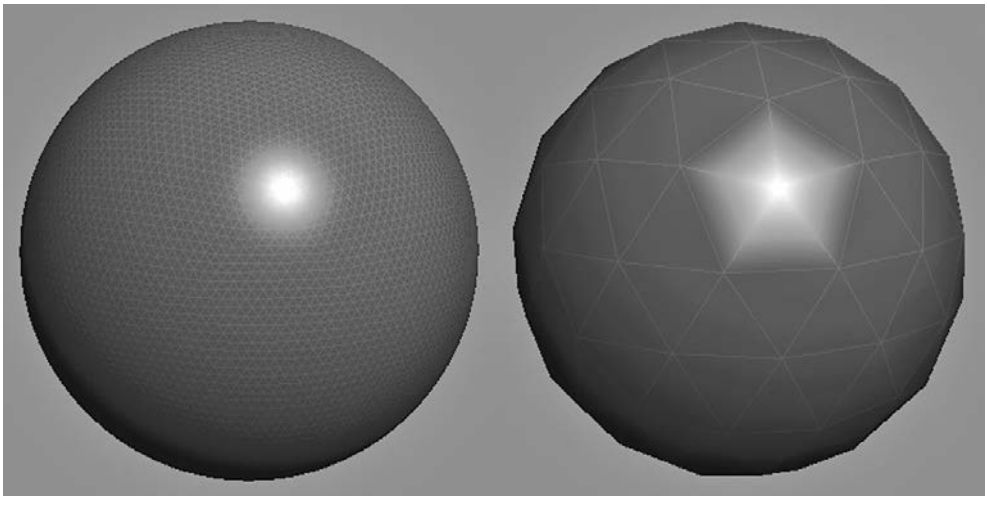


Рис. 11.14. Линейная интерполяция атрибутов вершин не всегда обеспечивает адекватное описание визуальных свойств поверхности, особенно при низком уровне тесселяции

Чтобы обойти ограничения поверхностных атрибутов поверхности, инженеры по рендерингу используют специальные растровые изображения, называемые *текстурными картами*. Текстура часто содержит информацию о цвете и обычно проецируется на треугольники меша. В этом случае она действует почти так же, как те забавные временные татуировки, которые мы наносили на руки, когда были детьми. Однако текстура может определять не только цвета, но и другие виды визуальных свойств поверхности. Кроме того, текстуру не обязательно проецировать на меш — например, ее можно задействовать как отдельную таблицу данных. Отдельные элементы изображения текстуры называют *текселами*, чтобы можно было отличать их от экранных пикселов.

В некоторых графических аппаратных средствах размер растрового изображения текстуры может быть равен только степени двойки. Так, часто используются текстуры размером 256×256, 512×512, 1024×1024 и 2048×2048, в то же время

большинство аппаратных средств позволяет брать текстуры любого размера при условии, что их можно поместить в видеопамять. Некоторые графические аппаратные средства накладывают дополнительные ограничения, например требуют, чтобы текстуры были квадратными, или снимают некоторые ограничения, например необходимость равенства размеров текстур степеням двойки.

Типы текстур. Наиболее распространенным типом текстур является так называемая *диффузная карта*, или *карта альбедо*. Такая текстура описывает диффузный цвет поверхности для каждого тексела поверхности и выступает в роли своего рода переводной картинки или камуфляжа.

В то же время в компьютерной графике используются и другие типы текстур, в том числе *карты нормалей* (содержат единичные векторы нормалей для каждого тексела, выраженные в RGB-значениях), *карты бликов* (определяют степень блеска поверхности для каждого тексела), *карты окружения* (содержат изображение окружающей среды для рендеринга отражений) и многие другие. Описание того, как можно применять различные типы текстур для реализации освещения на основе изображения и других эффектов, см. в подразделе 11.3.1.

На самом деле текстурные карты можно использовать для хранения любой информации, которая нужна при расчете освещения. Так, например, одномерную текстуру можно применять для хранения выборочных значений сложной математической функции, таблицы преобразования цветов или любой другой таблицы преобразования.

Текстурные координаты. Давайте рассмотрим, как следует проецировать на меш двухмерную текстуру. Для этого нужно определить двухмерную систему координат, известную как *пространство текстуры*. Текстурные координаты обычно представляются с помощью нормализованной пары чисел вида (u, v). Значения этих координат всегда варьируются от (0, 0) в нижнем левом углу текстуры до (1, 1) в верхнем правом углу. Нормализованные координаты позволяют применять одну и ту же систему координат вне зависимости от размеров текстуры.

Чтобы отобразить треугольник на 2D-текстуру, нужно просто определить пару текстурных координат (u_i, v_i) в каждой вершине i. Это обеспечит отображение треугольника на плоскость изображения в пространстве текстуры. Пример наложения текстуры показан на рис. 11.15.

Режимы адресации текстур. Выход текстурных координат за пределы диапазона [0, 1] не является чем-то недопустимым. Графическое аппаратное обеспечение может обрабатывать выход текстурных координат за пределы диапазона, используя любой из приведенных далее способов, которые называют *режимами адресации текстур*, при этом пользователь контролирует выбор режима.

- *Обертывание* (wrap). В этом режиме текстура многократно дублируется в каждом из направлений. Все текстурные координаты вида (ju, kv) эквивалентны координатам (u, v), где j и k — произвольные целые числа.
- *Зеркальное отображение* (mirror). Действует как режим обертывания с тем отличием, что текстура зеркально отражается относительно оси V для нечетных целых, кратных координате u, и относительно оси U для нечетных целых, кратных координате v.

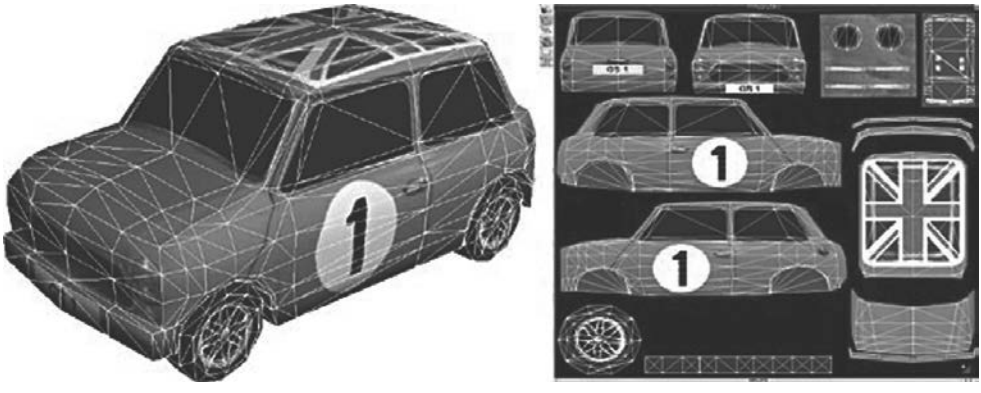


Рис. 11.15. Пример наложения текстуры. Показан вид треугольников в трехмерном пространстве и пространстве текстуры

- *Закрепление* (clamp). В этом режиме при выходе текстурных координат за пределы нормального диапазона происходит расширение цветов текселов, расположенных вокруг внешнего края текстуры.
- *Цвет рамки* (border color). В этом режиме для области за пределами диапазона координат текстуры [0, 1] используется цвет, произвольно указанный пользователем.

Эти режимы адресации текстур иллюстрирует рис. 11.16.

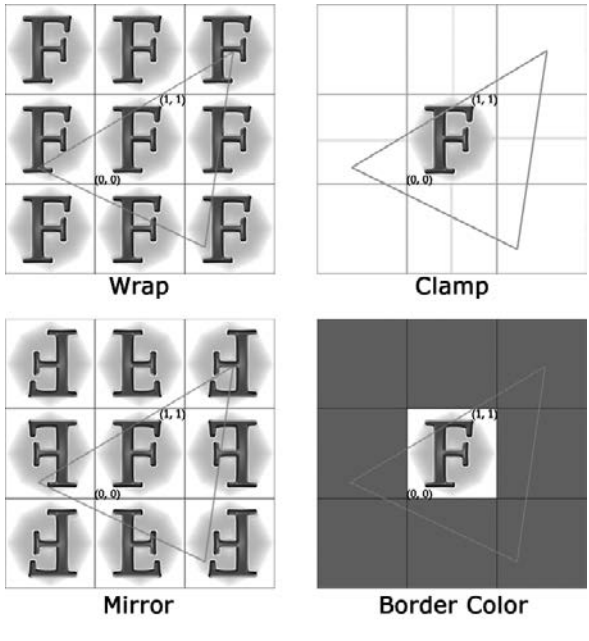


Рис. 11.16. Режимы адресации текстур

Форматы текстур. Растровые изображения текстур могут храниться на диске практически в любом формате изображения при условии, что ваш игровой движок содержит код, необходимый для считывания этого формата в память. Наиболее распространенными являются растровые форматы Targa (`.tga`), Portable Network Graphics (`.png`), разработанный компанией Microsoft BitMap Picture (`.bmp`) и TIFF (Tagged Image File Format — формат файлов изображений, снабженных тегами). В памяти текстуры обычно представляются в виде двухмерных (периодических) массивов пикселов с использованием различных цветовых форматов, таких как RGB888, RGBA8888, RGB565, RGBA5551 и т. д.

Большинство современных видеокарт и наборов графических API поддерживает *сжатые текстуры*. Так, DirectX поддерживает семейство сжатых форматов, известных как DXTn или S3 Texture Compression (S3TC), первоначально разработанных в S3 Graphics для использования в компьютерном графическом ускорителе. Мы не будем сейчас вдаваться в подробности, но основная идея состоит в том, чтобы разбить текстуру на блоки пикселов размером 4×4 и применить небольшую цветовую палитру для хранения цветов каждого блока. Подробнее о форматах сжатия текстур S3 можно прочитать по адресам en.wikipedia.org/wiki/S3_Texture_Compression и https://ru.wikipedia.org/wiki/S3TC.

Очевидным преимуществом сжатых текстур по сравнению с несжатыми является использование меньшего количества памяти. Еще одним неожиданным плюсом является то, что они ускоряют рендеринг. Сжатые по методу S3 текстуры обеспечивают это ускорение за счет более дружественных к кэшу схем доступа к памяти — блоки соседних пикселов размером 4×4 размещаются в одном 64- или 128-битном машинном слове, что позволяет одномоментно помещать в кэш более значительные части текстуры. Недостатком сжатых текстур являются артефакты сжатия. И хотя в большинстве случаев эти аномалии малозаметны, все же существуют ситуации, которые требуют использования несжатых текстур.

Тексельная плотность и MIP-текстурирование. Представим, что мы имеем дело с рендерингом полноэкранного квадранта (прямоугольника, состоящего из двух треугольников), где отображена текстура, разрешение которой в точности соответствует разрешению экрана. В этом случае каждый тексел отображается ровно на один экранный пиксел и можно сказать, что *тексельная плотность* (соотношение количества текселов и пикселов) равна единице. Однако если тот же квадрант будет рассматриваться с некоторого удаления, занимаемая им область экрана станет меньше. Поскольку разрешение текстуры при этом не изменится, тексельная плотность квадранта теперь будет больше единицы (то есть на каждый пиксел будет приходиться больше одного тексела).

Очевидно, что тексельная плотность не является фиксированной величиной — она изменяется при перемещении объекта с наложенной текстурой относительно камеры. Тексельная плотность влияет на потребление памяти и визуальное качество трехмерной сцены. Когда тексельная плотность намного меньше единицы, размер текселов значительно превышает размер экранных пикселов, что делает заметными края текселов и разрушает иллюзию. Когда тексельная плотность превышает единицу, на один экранный пиксел приходится сразу несколько текселов.

Это может вызвать появление *муара* (рис. 11.17). Что еще хуже, может происходить «уплывание» цвета пиксела или его мерцание из-за попеременного доминирования в цвете пиксела различных текселов, находящихся в его границах, при малейшем изменении угла или положения камеры. Рендеринг удаленного объекта с очень высокой плотностью текселов, если игрок *никогда* не приближается к нему, также может оказаться напрасной тратой памяти. В конце концов, зачем хранить в памяти такую четкую текстуру, если никто не увидит все эти подробности?

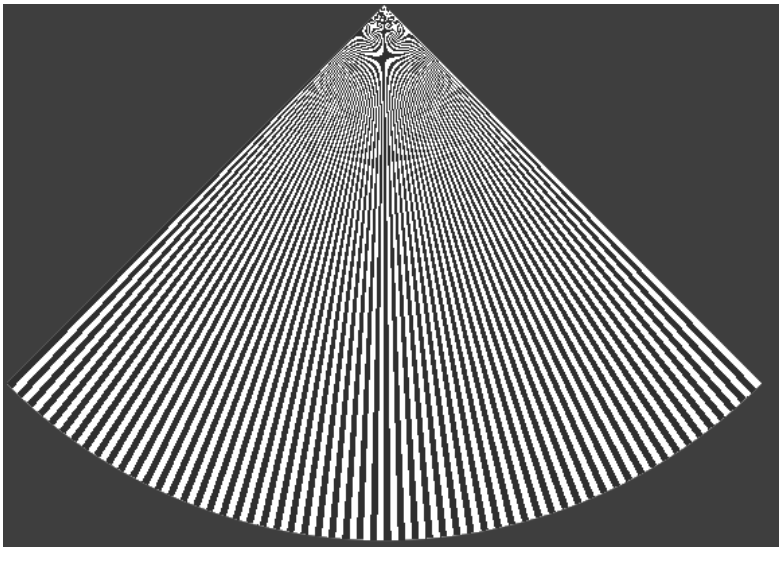


Рис. 11.17. Когда тексельная плотность превышает единицу, может возникнуть муар

В идеале тексельная плотность должна быть равной единице как для близких, так и для удаленных объектов. Хотя этого нельзя добиться в полной мере, мы можем приблизиться к идеалу, используя метод, называемый *множественным наложением текстур,* или *MIP-текстурированием* (от лат. multum in parvo — «многое в малом»). При этом для каждой текстуры создается ряд растровых изображений с последовательно уменьшающимся разрешением, где каждое следующее изображение в два раза меньше предыдущего по ширине и высоте. Каждое из этих изображений называется *MIP-картой* или *MIP-уровнем*. Так, например, текстура 64×64 будет иметь MIP-уровни со следующими размерами: 64×64, 32×32, 16×16, 8×8, 4×4, 2×2 и 1×1 (рис. 11.18).

После создания MIP-уровней текстуры графическое аппаратное обеспечение выбирает подходящий MIP-уровень, в зависимости от удаленности треугольника от камеры, таким образом, чтобы тексельная плотность была как можно ближе к единице. Так, если текстура занимает на экране область размером 40×40, то может быть выбран MIP-уровень размером 64×64, если занимает область размером 10×10, то может быть выбран MIP-уровень размером 16×16. Как мы увидим далее, используя метод *трилинейной фильтрации*, аппаратное обеспечение способно

сэмплировать два соседних MIP-уровня и смешивать полученные результаты. В данном случае отображение на область размером 10×10 можно выполнить смешиванием MIP-уровней размером 16×16 и 8×8.

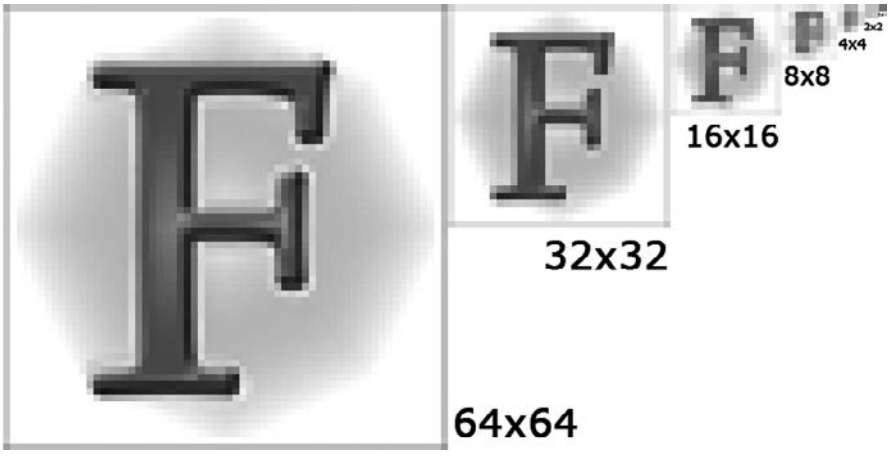


Рис. 11.18. MIP-уровни для текстуры размером 64×64

Тексельная плотность мирового пространства. Термин «тексельная плотность» можно использовать и для описания соотношения между количеством текселов и площадью мирового пространства на текстурируемой поверхности. Например, в случае двухметрового куба с наложенной текстурой размером 256×256 тексельная плотность составит $256^2 / 2^2 = 16\,384$. Чтобы отличать эту плотность от обсуждавшейся до сих пор тексельной плотности экранного пространства, будем называть ее *тексельной плотностью мирового пространства*.

Не нужно стремиться к тому, чтобы тексельная плотность мирового пространства была близкой к единице, и на практике конкретное значение обычно намного превышает единицу и полностью зависит от выбора единиц измерения площади мирового пространства. В то же время важно, чтобы наложение текстур на объекты производилось довольно единообразно в плане тексельной плотности мирового пространства. Так, например, обычно ожидается, что на каждой из шести сторон куба площадь текстуры будет одинаковой. В противном случае текстура на одной стороне куба будет иметь более низкое разрешение по сравнению с находящейся на другой стороне, что заметит игрок. Чтобы обеспечить единообразную тексельную плотность мирового пространства для всех объектов игры, многие игровые студии составляют рекомендации для команд художников и встраивают в движок средства визуализации тексельной плотности.

Фильтрация текстур. При рендеринге пиксела текстурируемого треугольника графическое аппаратное обеспечение сэмплирует растровое изображение текстуры, учитывая положение центра пиксела в ее пространстве. Поскольку в большинстве случаев текселы не отображаются на пикселы один к одному, центры пикселов могут находиться в любом месте текстурного пространства, в том числе на разде-

ляющей текселы границе. Поэтому для получения цвета фактически сэмплируемого тексела графическое аппаратное обеспечение обычно сэмплирует несколько текселов и смешивает полученные цвета. Это называется *фильтрацией текстур*.

Большинство видеокарт поддерживают следующие виды фильтрации текстур.

- *Выборка ближайшей точки*. Метод грубого приближения, при котором отбирается тот тексел, центр которого находится ближе всего к центру пиксела. При использовании MIP-текстурирования отбирается тот MIP-уровень, разрешение которого меньше всего отличается от идеального теоретического разрешения, при котором тексельная плотность мирового пространства равна единице, и в то же время превышает его.
- *Билинейная фильтрация*. При этом подходе вокруг центра пиксела отбираются четыре тексела и результирующий цвет вычисляется как средневзвешенное значение их цветов, где весовые коэффициенты зависят от удаленности центров текселов от центра пиксела. При использовании MIP-текстурирования выбирается ближайший MIP-уровень.
- *Трилинейная фильтрация*. При этом подходе производится билинейная фильтрация сразу на двух ближайших MIP-уровнях (уровнях с более высоким и более низким разрешением по сравнению с идеальным) с последующей линейной интерполяцией полученных результатов. Это позволяет устранить резкие визуальные границы между MIP-уровнями на экране.
- *Анизотропная фильтрация*. Как билинейная, так и трилинейная фильтрация отбирают квадратные блоки текселов размером 2×2. Это уместно, когда текстурируемая поверхность перпендикулярна направлению обзора, но плохо подходит для ее расположения под определенным углом к плоскости виртуального экрана. Анизотропная фильтрация отбирает текселы, находящиеся в пределах трапециевидной области, форма которой зависит от угла обзора, что повышает качество текстурируемых поверхностей, рассматриваемых под некоторым углом.

Материалы

Материал представляет собой полное описание визуальных свойств меша. Сюда входят определение текстур, отображаемых на поверхность меша, а также различные высокоуровневые свойства, например сведения о том, какие программы-шейдеры следует использовать при рендеринге меша, входные параметры для этих шейдеров и другие параметры, управляющие функциональностью аппаратного обеспечения для ускорения графики.

Хотя атрибуты вершин, по сути, являются составной частью описания свойств поверхности, они не считаются частью материала. В то же время они присутствуют в составе меша, поэтому пара из меша и материала содержит всю информацию, необходимую для рендеринга объекта. Пары из меша и материала иногда называют *рендер-пакетами*, кроме того, термин «геометрический примитив» также иногда трактуется в расширенном смысле — в него включаются пары из меша и материала.

В типичной 3D-модели используются сразу несколько материалов. Так, например, в модели человека будут отдельные материалы для волос, кожи, глаз, зубов и различной одежды. В силу этого меш обычно разбивается на подмеши, каждый из которых отображается на один материал. В движке рендеринга OGRE эта схема реализуется с помощью класса `Ogre::SubMesh`.

11.1.3. Основы освещения

Освещение лежит в основе рендеринга любой компьютерной графики. Без хорошего освещения сцена будет выглядеть плоской и искусственной, даже если она очень хорошо смоделирована во всех других отношениях. И наоборот, даже самая простая сцена может выглядеть чрезвычайно реалистично, если ее должным образом осветить. Прекрасным подтверждением этому может служить представленная на рис. 11.19 классическая коробка Корнелла.

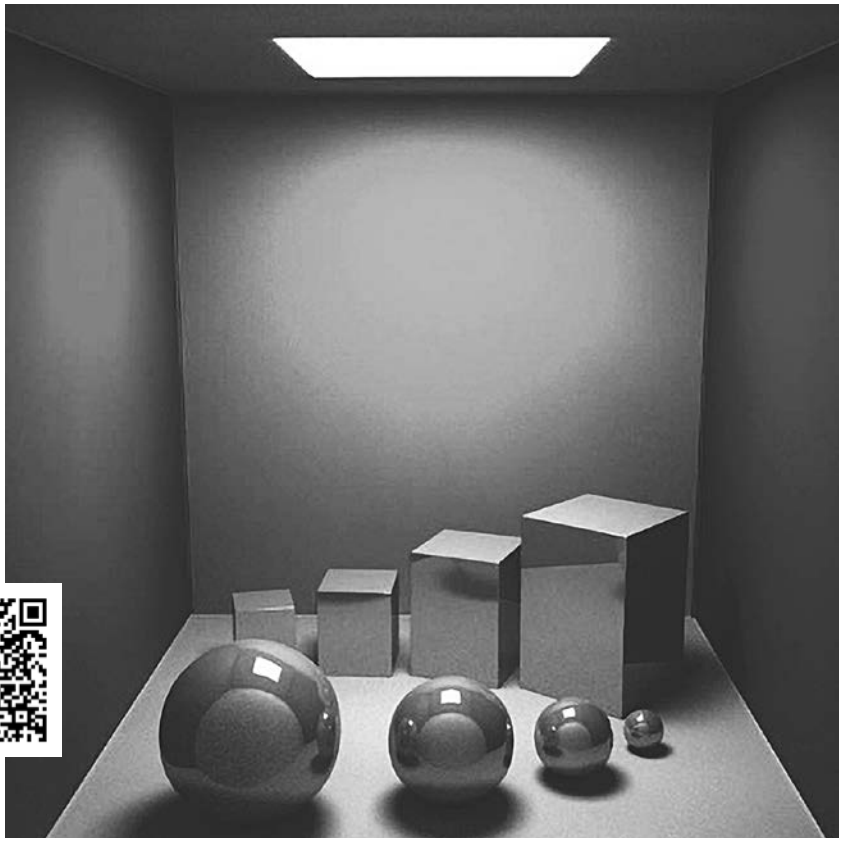


Рис. 11.19. Вариация на тему классической сцены — коробка Корнелла, показывающая, как удачное освещение может придать фотореалистичный вид даже самой простой сцене

Еще одной хорошей иллюстрацией важности освещения может служить представленный далее ряд снимков экрана из игры *The Last of Us: Remastered* студии Naughty Dog. На рис. 11.20 показано, как выглядит сцена из игры при ее рендеринге без использования текстур. На рис. 11.21 — та же сцена после применения диффузных текстур, а на рис. 11.22 — полностью освещенная. Обратите внимание на то, насколько возрастает реалистичность после использования освещения.

Рис. 11.20. Сцена из игры The Last of Us: Remastered при рендеринге без использования текстур (снимок экрана с сайта https://beedge.neocities.org/), PlayStation 4

Термин *«затенение» (шейдинг)* часто используется в более широком смысле, подразумевающем не только освещение, но и другие визуальные эффекты. При этом затенение может включать в себя процедурное смещение вершин для имитации движения водной поверхности, генерирование изгибов волос или меховых оболочек, тесселяцию поверхностей высокого порядка и практически любые другие вычисления, необходимые для рендеринга сцены.

Рис. 11.21. Сцена из игры The Last of Us: Remastered после применения диффузных текстур

Рис. 11.22. Сцена из игры The Last of Us: Remastered (снимок экрана с сайта https://www.gameenginebook.com) в полностью освещенном виде

В следующих подразделах мы рассмотрим основы освещения, необходимые для того, чтобы разобраться в графическом аппаратном обеспечении и конвейере рендеринга. После этого вернемся к теме освещения в разделе 11.3, где будут рассмотрены некоторые продвинутые методы освещения и затенения.

Локальные и глобальные модели освещения

В движках рендеринга используются различные математические модели взаимодействия света с поверхностью и объемом, называемые *моделями переноса света*. В простейших моделях учитывается только *прямое освещение*, при котором свет излучается, отражается от какого-то одного объекта сцены и падает непосредственно на плоскость отображения виртуальной камеры.

Простые модели называют *локальными моделями освещения*, поскольку в них учитываются только локальные эффекты, производимые светом на один объект, и не принимается в расчет взаимное влияние объектов на внешний вид друг друга. Неудивительно, что локальные модели были первыми моделями, получившими применение в играх, они используются в них и сегодня — локальное освещение иногда обеспечивает поразительно реалистичные результаты!

Однако подлинной фотореалистичности можно добиться, только принимая во внимание *непрямое освещение*, при котором свет многократно отражается от множества поверхностей, перед тем как попасть в виртуальную камеру. Модели освещения, в которых учитывается непрямое освещение, называются *глобальными моделями освещения*. Некоторые глобальные модели освещения предназначены для моделирования конкретного визуального явления, например для создания реалистичных теней, моделирования отражающих поверхностей, учета взаимного отражения между объектами (при этом цвет одного объекта влияет на цвета окружающих объектов) или моделирования каустических эффектов (интенсивных отражений от воды или блестящей металлической поверхности). Другие глобальные модели освещения предназначены для целостного отражения широкого спектра оптических явлений. Примерами таких технологий являются методы трассировки лучей и излучательности.

Глобальное освещение полностью описывается математической формулой, известной как *уравнение рендеринга* или *уравнение затенения*. Его предложил в 1986 году Дж. Т. Кайя в фундаментальной работе, представленной на конференции Специальной группы по компьютерной графике (SIGGRAPH). В какой-то степени любой метод рендеринга можно рассматривать как полное или частичное решение уравнения рендеринга, хотя разные методы могут различаться фундаментальным подходом к решению уравнения и используемыми допущениями, упрощениями и аппроксимациями. Чтобы узнать больше об уравнении рендеринга, см. статью по адресу https://ru.wikipedia.org/wiki/Уравнение_рендеринга, [2], [10] и практически любой другой текст о продвинутом рендеринге и освещении.

Модель освещения Фонга

Наиболее распространенной локальной моделью освещения, применяемой в игровых движках рендеринга, является *модель Фонга*. Она моделирует отраженный от поверхности свет как сумму трех составляющих.

- *Фоновая* составляющая моделирует общий уровень освещенности сцены. Это грубая аппроксимация количества присутствующего в сцене непрямого отраженного света. Благодаря отражениям непрямого света затененные области не выглядят абсолютно черными.
- *Диффузная* составляющая моделирует свет, равномерно отражающийся во всех направлениях от каждого прямого источника света. Это хорошее приближение

к тому, как в реальности свет отражается от матовой поверхности, такой как деревянный брусок или кусок ткани.

- *Зеркальная* составляющая моделирует яркие блики, которые можно наблюдать на блестящих поверхностях. Зеркальные блики появляются, когда направление обзора мало отличается от направления прямолинейного отражения луча, идущего от источника света.

На рис. 11.23 показано, как фоновая, диффузная и зеркальная составляющие совместно формируют окончательную интенсивность и цвет поверхности.

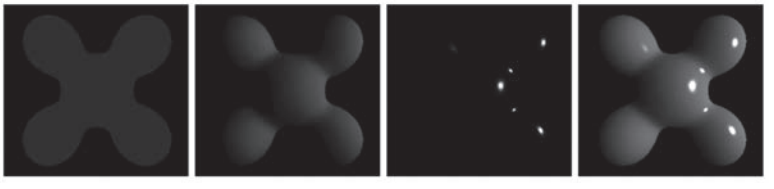


Рис. 11.23. Отражение Фонга вычисляется суммированием фоновой, диффузной и зеркальной составляющих

Чтобы рассчитать отражение Фонга в определенной точке поверхности, требуется ряд входных параметров. Поскольку в большинстве случаев модель Фонга независимо применяется ко всем трем цветовым каналам (R, G и B), все цветовые параметры в приведенных далее выкладках являются трехэлементными векторами. Входными данными для модели Фонга служат:

- вектор направления обзора $\mathbf{V} = [V_x\ V_y\ V_z]$, направленный от точки отражения к фокальной точке виртуальной камеры (то есть являющийся отрицательным по отношению к фронтальному вектору мирового пространства камеры);
- интенсивность фонового света для трех цветовых каналов $\mathbf{A} = [A_R\ A_G\ A_B]$;
- нормаль поверхности $\mathbf{N} = [N_x\ N_y\ N_z]$ в точке падения светового луча на поверхность;
- отражающие свойства поверхности:
  - коэффициент фонового отражения $\mathbf{k}_A = [k_{AR}\ k_{AG}\ k_{AB}]$;
  - коэффициент диффузного отражения $\mathbf{k}_D = [k_{DR}\ k_{DG}\ k_{DB}]$;
  - коэффициент зеркального отражения $\mathbf{k}_S = [k_{SR}\ k_{SG}\ k_{SB}]$;
  - экспонента зеркальных бликов α;
- а также для каждого источника света i:
  - цвет и интенсивность света $\mathbf{C}_i = [C_{iR}\ C_{iG}\ C_{iB}]$;
  - вектор направления $\mathbf{L}_i$ от точки отражения к источнику света.

В модели Фонга интенсивность **I** света, отраженного от точки, может быть выражена с помощью следующего векторного уравнения:

$$\mathbf{I} = \left(\mathbf{k}_A \otimes \mathbf{A}\right) + \sum_i \left[\mathbf{k}_D\left(\mathbf{NL}_i\right) + \mathbf{k}_S\left(\mathbf{R}_i\mathbf{V}\right)^\alpha\right] \otimes \mathbf{C}_i,$$

где суммирование производится по всем источникам света, воздействующим на рассматриваемую точку. Напомню, что оператором ⊗ обозначается *покомпонентное* умножение двух векторов — так называемое произведение Адамара.

Это выражение можно разбить на три скалярных уравнения, по одному для каждого цветового канала:

$$I_R = k_{AR}A_R + \sum_i \left[k_{DR}\left(\mathbf{NL}_i\right) + k_{SR}\left(\mathbf{R}_i\mathbf{V}\right)^\alpha\right] C_{iR};$$

$$I_G = k_{AG}A_G + \sum_i \left[k_{DG}\left(\mathbf{NL}_i\right) + k_{SG}\left(\mathbf{R}_i\mathbf{V}\right)^\alpha\right] C_{iG};$$

$$I_B = k_{AB}A_B + \sum_i \left[k_{DB}\left(\mathbf{NL}_i\right) + k_{SB}\left(\mathbf{R}_i\mathbf{V}\right)^\alpha\right] C_{iB}.$$

В этих уравнениях вектор $\mathbf{R}_i = [R_{ix}\ R_{iy}\ R_{iz}]$ является *отражением* вектора направления светового луча $\mathbf{L}_i$ относительно нормали поверхности **N**.

Вектор $\mathbf{R}_i$ можно легко рассчитать, применив векторную математику (рис. 11.24). Любой вектор можно выразить как сумму его нормальной и тангенциальной составляющих. Например, мы можем разбить вектор направления света **L** следующим образом:

$$\mathbf{L} = \mathbf{L}_N + \mathbf{L}_T.$$

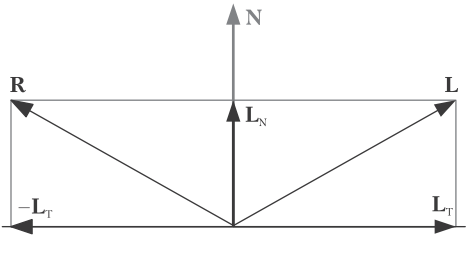


Рис. 11.24. Расчет вектора отраженного освещения R на основе вектора исходного освещения L и нормали поверхности N

Как известно, скалярное произведение (**NL**) представляет собой проекцию вектора **L** на поверхность (скалярную величину). Таким образом, нормальная составляющая $\mathbf{L}_N$ — это просто единичный вектор нормали **N**, масштабируемый значением данного скалярного произведения:

$$\mathbf{L}_N = (\mathbf{NL})\mathbf{N}.$$

Отраженный вектор **R** имеет такую же нормальную составляющую, как и вектор **L**, но *противоположную* тангенциальную составляющую ($-\mathbf{L}_T$). Следовательно, вектор **R** можно рассчитать так:

$$\mathbf{R} = \mathbf{L}_N - \mathbf{L}_T = \mathbf{L}_N - (\mathbf{L} - \mathbf{L}_N) = 2\mathbf{L}_N - \mathbf{L};$$

$$\mathbf{R} = 2(\mathbf{NL})\mathbf{N} - \mathbf{L}_T.$$

Это уравнение можно использовать для расчета всех значений $\mathbf{R}_i$ для каждого направления света $\mathbf{L}_i$.

Модель освещения Блинна — Фонга. Эта модель представляет собой разновидность затенения Фонга, в которой используется несколько иной способ расчета зеркального отражения. При этом вектор **H** определяется как вектор, расположенный посередине между векторами направления обзора **V** и направления света **L**. Тогда зеркальная составляющая будет равна выражению $(\mathbf{NH})^a$, а не выражению $(\mathbf{RV})^\alpha$, используемому в модели Фонга. Экспонента *a* несколько отличается от экспоненты α из модели Фонга, но ее значение выбрано таким образом, чтобы обеспечивать максимально точное соответствие зеркальной составляющей согласно модели Фонга.

Модель Блинна — Фонга позволяет повысить производительность в динамическом режиме за счет некоторого уменьшения точности, однако стоит сказать, что для отдельных типов поверхностей она обеспечивает даже более точное соответствие эмпирическим результатам, чем модель Фонга. Модель Блинна — Фонга была практически единственной моделью, которая использовалась на заре компьютерных игр и прошивалась в конвейерах с фиксированной функциональностью первых графических процессоров. Подробности см. по адресам https://en.wikipedia.org/wiki/Blinn-Phong_reflection_model и https://ru.wikipedia.org/wiki/Затенение_по_Фонгу.

График функции BRDF. Три составляющие в модели освещения Фонга представляют собой частные случаи обобщенной локальной модели отражения, известной как *функция двунаправленного распределения отражения* (bidirectional reflection distribution function, BRDF). Функция BRDF рассчитывает отношение исходящего (отраженного) излучения вдоль заданного направления обзора **V** к входящему излучению вдоль падающего луча **L**.

Функцию BRDF можно изобразить в виде полусферического графика, где радиальное расстояние от начала координат представляет интенсивность света, получаемую при наблюдении точки отражения с данного направления. *Диффузная* составляющая по модели Фонга равна $\mathbf{k}_D(\mathbf{NL})$. Она зависит только от входящего луча освещения **L**, но не от направления обзора **V**. Следовательно, ее значение будет одинаковым для всех углов обзора. Если построить график зависимости этой составляющей от угла обзора в трехмерной системе координат, он будет выглядеть как полусфера с центром в точке, в которой мы рассчитываем отражение Фонга. Как это выглядит в двухмерной системе координат, показано на рис. 11.25.

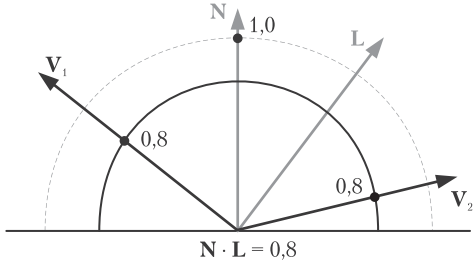


Рис. 11.25. Диффузная составляющая модели отражения Фонга зависит от NL, но не зависит от направления обзора V

Зеркальная составляющая по модели Фонга равна $\mathbf{k}_D(\mathbf{RV})^{\alpha}$. Она зависит и от направления освещения **L**, и от направления обзора **V**. Эта составляющая создает зеркальный блик, когда направление обзора незначительно отличается от отражения **R** направления освещения **L** относительно нормали поверхности. Однако ее значение быстро уменьшается при отклонении направления обзора от направления отраженного освещения. Как это выглядит в двухмерной системе координат, показано на рис. 11.26.

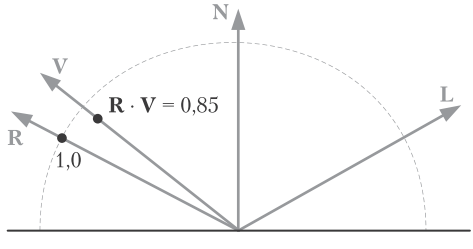


Рис. 11.26. Значение зеркальной составляющей модели отражения Фонга является максимальным, когда направление обзора V совпадает с направлением отраженного света R, и быстро уменьшается по мере отклонения вектора V от вектора R

Моделирование источников света

Помимо моделирования взаимодействий света с поверхностями, нам нужно описать имеющиеся в сцене источники света. Аналогично тому, как рендеринг в реальном времени подходит ко всему остальному, реальные источники света аппроксимируются с помощью различных упрощенных моделей.

Статическое освещение. Самый быстро реализуемый подход к расчету освещения состоит в том, чтобы вообще его не рассчитывать. При этом освещение выполняется в автономном режиме в удобное для вас время. Мы можем предварительно рассчитать отражение Фонга в вершинах меша и сохранить результаты в виде вершинных атрибутов, определяющих диффузный цвет. А также предварительно рассчитать освещение на попиксельной основе и сохранить результаты в виде

специальной разновидности текстурной карты, называемой *картой освещения*. Во время выполнения текстура карты освещения проецируется на объекты сцены для получения эффектов, производимых на них светом.

Возможно, вас удивляет, почему мы не можем просто внедрить карты освещения непосредственно в диффузные текстуры сцены? Этого не стоит делать по ряду причин. Во-первых, заносить освещение в диффузные текстурные карты не очень удобно потому, что их часто располагают мозаикой и/или многократно дублируют в пределах сцены. Вместо этого обычно создается единая карта освещения для каждого источника света, которая применяется к любым объектам, находящимся в зоне его влияния.

Такой подход позволяет обеспечить надлежащее освещение динамичных объектов, перемещающихся мимо источника света. Кроме того, карты освещения могут иметь другое (обычно более низкое) разрешение по сравнению с диффузными текстурными картами. Наконец, чистая карта освещения обычно лучше поддается сжатию, чем карта, содержащая информацию о диффузном цвете.

Фоновые источники света. *Фоновый источник света* соответствует фоновой составляющей в модели освещения Фонга. Эта составляющая не зависит от угла обзора и не имеет какого-либо определенного направления. Таким образом, фоновый источник света представляется одним цветовым значением, которое соответствует цветовой составляющей **A** в уравнении Фонга (и масштабируется в динамическом режиме значением коэффициента фонового отражения поверхности $\mathbf{k}_A$). Интенсивность и цвет фонового света могут варьироваться в зависимости от области игрового мира.

Направленные источники света. *Направленный источник света* моделирует источник света, который находится на практически бесконечном удалении от освещаемой поверхности — как солнце. Направленный источник света излучает параллельные лучи и не имеет конкретного местоположения в игровом мире. Поэтому направленный источник света представляется цветом источника света **C** и вектором направления **L**. Как выглядит направленный источник света, показано на рис. 11.27.

Точечные (всенаправленные) источники света. *Точечный (всенаправленный) источник света* имеет четко заданное положение в игровом мире и излучает свет равномерно во всех направлениях. Обычно считается, что интенсивность света уменьшается пропорционально квадрату расстояния от источника света, и за пределами заранее заданного максимального радиуса его эффекты просто принимаются равными нулю. Точечный источник света представляется его положением **P**, цветом/интенсивностью **C** и максимальным радиусом $r_{\max}$. Движок рендеринга применяет эффекты точечного источника света только к тем поверхностям, которые находятся в зоне его влияния (что является значительной оптимизацией). Точечный источник света показан на рис. 11.28.

Прожекторные источники света. *Прожекторный источник света* ведет себя как точечный источник, лучи которого ограничены конусообразной областью, как

у прожектора. Обычно задаются два угла, которые определяют два конуса — внутренний и внешний. Считается, что свет во внутреннем конусе имеет полную интенсивность. В то же время при увеличении угла от внутреннего конуса к внешнему интенсивность освещения падает и за пределами внешнего конуса считается равной нулю. Внутри обоих конусов интенсивность света также уменьшается пропорционально радиальному расстоянию. Прожекторный источник света представляется его позицией **P**, цветом **C**, вектором **L** центрального направления, максимальным радиусом r_{max} и углами внутреннего и внешнего конусов θ_{min} и θ_{max} соответственно. Прожекторный источник света показан на рис. 11.29.

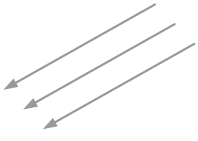


Рис. 11.27. Модель направленного источника света

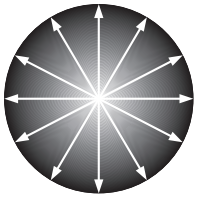


Рис. 11.28. Модель точечного источника света

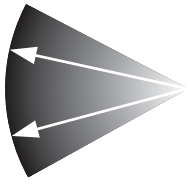


Рис. 11.29. Модель прожекторного источника света

Источники света с площадью. Все источники света, которые мы рассматривали до сих пор, излучают свет из некоторой идеализированной локальной или бесконечно удаленной точки. Реальный источник света практически всегда обладает ненулевой площадью, и именно поэтому создаются мягкие реалистичные тени и полутени разной интенсивности (umbra/penumbra).

Вместо того чтобы пытаться напрямую смоделировать источники света с площадью, инженеры по компьютерной графике часто прибегают к различным уловкам, позволяющим учесть их поведение. Например, чтобы сымитировать полутень, можно отбросить несколько теней и смешать результаты либо определенным образом размыть края резкой тени.

Излучающие объекты. Иногда присутствующие в сцене поверхности сами являются источниками света. Примером могут служить ручные фонарики, светящиеся хрустальные шары, пламя ракетного двигателя и т. д. Светящиеся поверхности можно смоделировать с помощью *излучающей текстурной карты* — текстуры, цвета которой всегда отображаются с максимальной интенсивностью, вне зависимости от окружающих условий освещения. Такие текстуры можно использовать для создания неоновых вывесок, фар автомобилей и т. д.

Рендеринг некоторых видов излучающих объектов выполняют, сочетая несколько методов. Например, рендеринг фонарика можно сделать с применением излучающей текстуры в случае, когда его луч направлен прямо на вас, с помощью совмещенного с ним прожекторного источника света, отбрасывающего свет на сцену, желтого полупрозрачного меша для имитации светового конуса, нескольких обращенных к камере прозрачных карт для воспроизведения бликов объектива

(или *эффекта цветения*, если движок поддерживает освещение с расширенным динамическим диапазоном) и проецируемой текстуры для создания каустического эффекта на освещаемых фонариком поверхностях. Прекрасным примером такого сочетания эффектов может служить реализация фонарика в игре *Luigi's Mansion* (рис. 11.30).

Рис. 11.30. Фонарик в игре Luigi's Mansion компании Nintendo (Wii) представляет собой сочетание множества визуальных эффектов, включая полупрозрачный конус для имитации луча, динамический прожекторный источник света для отбрасывания света на сцену, излучающую текстуру на объективе и обращенные к камере карты для имитации бликов объектива

11.1.4. Виртуальная камера

Используемая в компьютерной графике виртуальная камера устроена гораздо проще, чем реальная камера или человеческий глаз. Под камерой здесь понимается идеализированная фокальная точка, на небольшом расстоянии от которой подвешена прямоугольная виртуальная светочувствительная поверхность, называемая *прямоугольником отображения*.

Прямоугольник отображения состоит из меша квадратных или прямоугольных виртуальных светочувствительных элементов, каждый из которых соответствует одному экранному пикселу. Рендеринг можно рассматривать как процесс определения значений цвета и интенсивности света, регистрируемых каждым из этих элементов.

Пространство обзора

Фокальная точка виртуальной камеры является началом трехмерной системы координат, называемой *пространством обзора* или *пространством камеры*. Камера обычно смотрит вдоль положительной или отрицательной оси Z в пространстве

обзора, при этом ось Y направлена вверх, а ось X — влево или вправо. Типичные варианты левосторонней и правосторонней компоновки осей пространства обзора показаны на рис. 11.31. Положение и ориентацию камеры можно задать с помощью матрицы преобразования обзора в мир точно так же, как положение экземпляра меша в сцене задается с помощью его матрицы преобразования модели в мир. Если известны позиционный вектор и три единичных базисных вектора пространства камеры, выраженные в координатах мирового пространства, то матрицу преобразования обзора в мир можно записать аналогично тому, как была составлена матрица преобразования модели в мир:

$$\mathbf{M}_{V\to W} = \left[\begin{array}{c|c} \mathbf{i}_V & 0 \\ \mathbf{j}_V & 0 \\ \mathbf{k}_V & 0 \\ \hline \mathbf{t}_V & 1 \end{array}\right].$$

При рендеринге триангулярного меша его вершины сначала преобразуются из пространства модели в мировое пространство, а затем — из мирового пространства в пространство обзора. Для выполнения последнего преобразования нам требуется матрица преобразования мира в обзор, которая является обратной по отношению к матрице преобразования обзора в мир. Эту матрицу иногда называют *матрицей обзора*:

$$\mathbf{M}_{W\to V} = \mathbf{M}_{V\to W}^{-1} = \mathbf{M}_{\text{view}}.$$

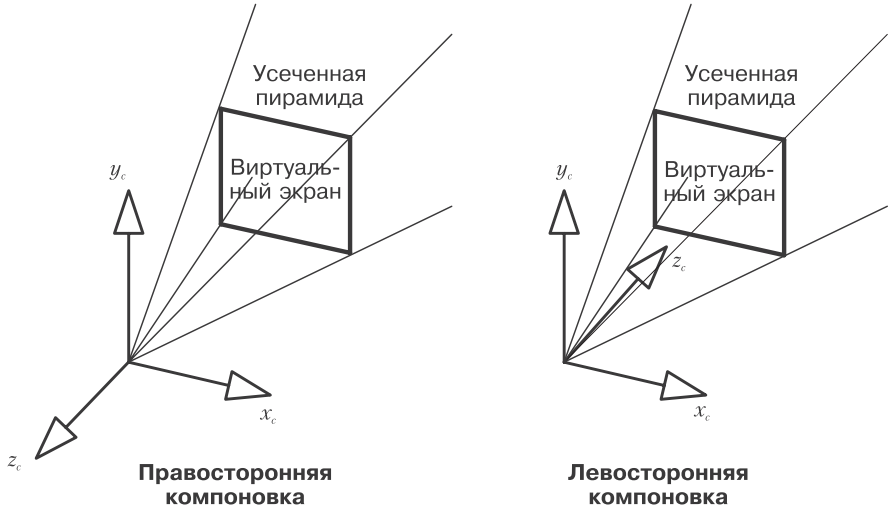


Рис. 11.31. Левосторонний и правосторонний варианты компоновки осей пространства камеры

Здесь следует проявлять осторожность. Тот факт, что матрица камеры инвертирована по отношению к матрицам объектов сцены, она является частым источником путаницы и ошибок среди начинающих разработчиков игр.

Матрица преобразования мира в обзор часто объединяется с матрицей преобразования модели в мир перед рендерингом конкретного экземпляра меша. Эта объединенная матрица называется в OpenGL *матрицей преобразования модели в обзор (model-view matrix)*. Она рассчитывается заранее, чтобы движку рендеринга оставалось выполнить только одну операцию умножения матриц при преобразовании вершин из пространства модели в пространство обзора:

$$\mathbf{M}_{M\to V} = \mathbf{M}_{M\to W}\mathbf{M}_{W\to V} = \mathbf{M}_{\text{modelview}}.$$

Проекции

Для рендеринга трехмерной сцены на двухмерной плоскости отображения задействуется специальная разновидность преобразования, называемая *проекцией*. Наиболее распространенным типом проекции в компьютерной графике является *перспективная* проекция, поскольку она имитирует типы изображений, создаваемых обычной камерой. При использовании этого типа проекции объекты кажутся меньше по мере удаления от камеры, то есть проявляется так называемое *перспективное уменьшение*.

В некоторых играх применяется и сохраняющая размеры *ортогональная* проекция, главным образом для рендеринга *видов в плане* (например, вида спереди, сбоку и сверху) трехмерных моделей или уровней игры для редактирования, наложения двухмерной графики на экран в качестве HUD-интерфейса и т. д. На рис. 11.32 показано, как будет выглядеть куб при рендеринге с использованием этих двух типов проекции.

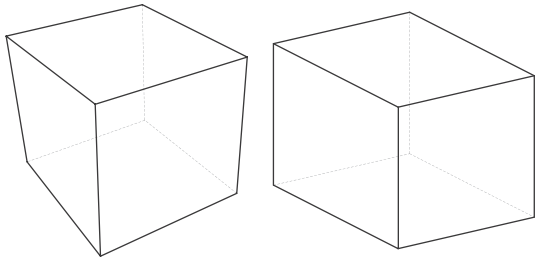


Рис. 11.32. Рендеринг куба с помощью перспективной *(слева)* и ортогональной *(справа)* проекций

Объем и усеченная пирамида обзора

Та область пространства, которую может видеть камера, называется *объемом обзора*. Объем обзора определяется шестью плоскостями. *Передняя плоскость* соответствует виртуальной светочувствительной поверхности, воспринимающей изображение.

Четыре боковые плоскости соответствуют краям виртуального экрана. *Задняя плоскость* используется в качестве меры оптимизации рендеринга, исключающей отрисовку очень удаленных объектов. Она также определяет верхний предел значений глубины, размещаемых в буфере глубины.

При рендеринге сцены с помощью перспективной проекции объем обзора представляет собой усеченную пирамиду (рис. 11.33). При ортогональной проекции объем обзора — это прямоугольная призма (рис. 11.34).

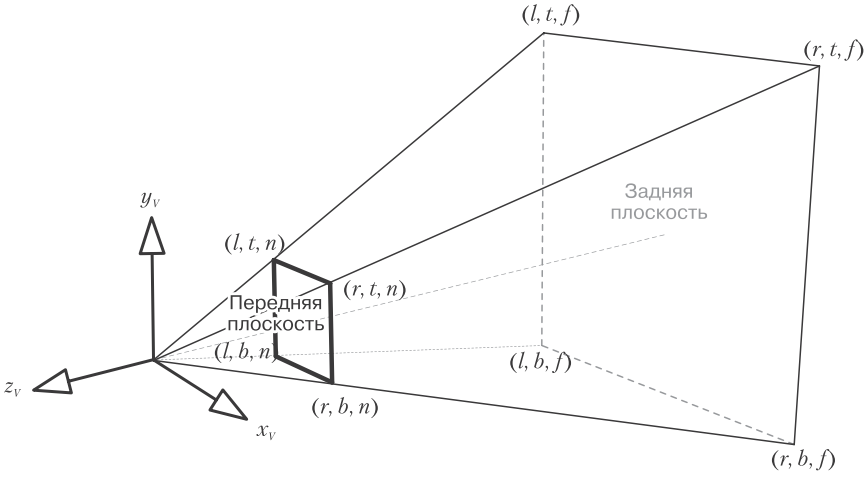


Рис. 11.33. Объем обзора перспективной проекции (усеченная пирамида)

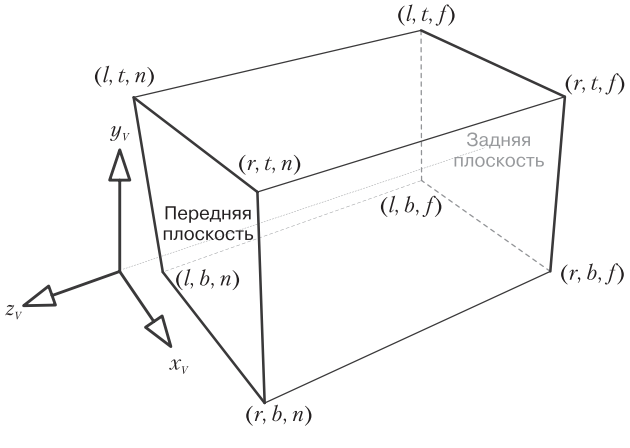


Рис. 11.34. Объем обзора ортогональной проекции

Шесть плоскостей объема обзора можно представить в компактном виде с помощью шести четырехэлементных векторов $(n_{ix}, n_{iy}, n_{iz}, d_i)$, где $\mathbf{n} = (n_x, n_y, n_z)$ — нормаль плоскости, а d — перпендикулярное расстояние от нее до начала координат.

Если нам больше подходит точечно-нормальное представление плоскости, мы можем описать плоскости с помощью шести пар векторов ($\mathbf{Q}_i$, $\mathbf{n}_i$), где $\mathbf{Q}$ — произвольная точка на плоскости, а $\mathbf{n}$ — нормаль плоскости. (В обоих случаях i — индекс, обозначающий плоскость.)

Проекция и однородное пространство отсечения

Как перспективная, так и ортогональная проекции преобразуют точки из пространства обзора в координатное пространство, называемое *однородным пространством отсечения*. Это трехмерное пространство на самом деле является просто видоизмененной версией пространства обзора. Пространство отсечения служит для приведения объема обзора пространства камеры к каноническому виду, который не зависит ни от типа *проекции*, используемой для преобразования трехмерной сцены в двухмерное пространство экрана, ни от *разрешения* и *соотношения сторон* экрана, на котором будет отображаться сцена.

Канонический объем обзора в пространстве отсечения представляет собой прямоугольную призму, располагающуюся от –1 до +1 вдоль осей X и Y. Вдоль оси Z объем обзора простирается либо от –1 до +1 (в OpenGL), либо от 0 до 1 (в DirectX). Эта система координат называется *пространством отсечения*, потому что плоскости объема обзора выровнены в нем по осям, что позволяет легко выполнять *усечение* треугольников до объема обзора (даже когда используется перспективная проекция). Канонический объем обзора в пространстве отсечения для OpenGL показан на рис. 11.35. Обратите внимание на то, что ось Z пространства отсечения направлена от нас, ось Y — вверх, а ось X — вправо. То есть однородное пространство отсечения обычно является *левосторонним*. Это объясняется тем,

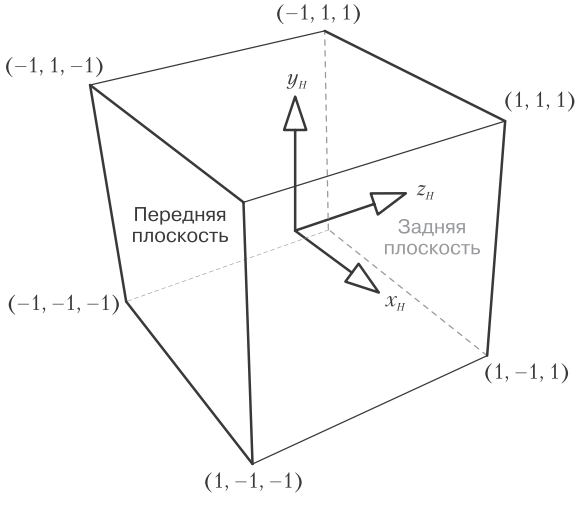


Рис. 11.35. Канонический объем обзора в однородном пространстве отсечения

что при левосторонней компоновке осей увеличение координаты z соответствует движению *вглубь* экрана, в то время как увеличение координат y и x, как обычно, соответствует движению вверх и вправо соответственно.

Перспективная проекция. Поскольку прекрасное разъяснение перспективной проекции уже представлено в подразделе 4.5.1 книги [32], мы не будем повторять его здесь. Вместо этого мы просто запишем далее матрицу перспективной проекции $\mathbf{M}_{V\to H}$. (Индекс $V\to H$ указывает, что данная матрица преобразует вершины из пространства обзора в однородное пространство отсечения.) Если допустить, что пространство обзора является правосторонним, то передняя плоскость пересекает ось Z при $z = -n$, а задняя плоскость — при $z = -f$. Левый, правый, нижний и верхний края виртуального экрана находятся соответственно в точках $x = l$, $x = r$, $y = b$ и $y = t$ передней плоскости. (Обычно, хотя и не всегда, виртуальный экран центрируется относительно оси Z пространства камеры, в таком случае $l = -r$ и $b = -t$.) Используя эти определения, можно вывести матрицу перспективной проекции для OpenGL:

$$\mathbf{M}_{V\to H} = \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{2nf}{f-n}\right) & 0 \end{bmatrix}.$$

В DirectX протяженность по оси Z объема обзора в пространстве отсечения находится в диапазоне [0, 1], а не [–1, 1], как в OpenGL.

Можно легко скорректировать матрицу перспективной проекции в соответствии с определениями, принятыми в DirectX:

$$\left(\mathbf{M}_{V\to H}\right)_{\text{DirectX}} = \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{nf}{f-n}\right) & 0 \end{bmatrix}.$$

Деление на координату z. При использовании перспективной проекции координаты x и y каждой вершины делятся на ее координату z. Это производит эффект

перспективного уменьшения. Чтобы понять, почему это происходит, попробуем умножить точку в пространстве обзора $\mathbf{p}_V$, выраженную в четырехэлементных однородных координатах, на матрицу перспективной проекции OpenGL:

$$\mathbf{p}_H = \mathbf{p}_V \mathbf{M}_{V \to H} = \begin{bmatrix} p_{Vx} & p_{Vy} & p_{Vz} & 1 \end{bmatrix} \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{2nf}{f-n}\right) & 0 \end{bmatrix}.$$

Результат умножения принимает следующий вид:

$$\mathbf{p}_H = \begin{bmatrix} a & b & c & -p_{Vz} \end{bmatrix}. \tag{11.1}$$

При преобразовании любого однородного вектора в трехмерные координаты компоненты x, y и z делятся на компоненту w:

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \equiv \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix}.$$

Таким образом, после деления уравнения (11.1) на однородную компоненту w, которая является просто отрицательной координатой z пространства обзора $-p_{Vz}$, имеем:

$$\mathbf{p}_H = \begin{bmatrix} \frac{a}{-p_{Vz}} & \frac{b}{-p_{Vz}} & \frac{c}{-p_{Vz}} \end{bmatrix} = \begin{bmatrix} p_{Hx} & p_{Hy} & p_{Hz} \end{bmatrix}.$$

Итак, однородные координаты пространства отсечения были разделены на координату z пространства обзора, что и производит эффект перспективного уменьшения.

Перспективно-корректная интерполяция вершинных атрибутов. Как упоминалось в подразделе 11.1.2, вершинные атрибуты интерполируются для определения подходящих для них значений внутри треугольника. Интерполяция атрибутов выполняется в *экранном пространстве*. Мы перебираем каждый экранный пиксел и пытаемся определить значение каждого атрибута в соответствующей точке *на поверхности треугольника*. При рендеринге сцены с использованием перспективной проекции это следует делать очень внимательно, чтобы не забыть учесть в расчетах перспективное уменьшение. Такая правильно выполненная интерполяция называется *перспективно-корректной* интерполяцией атрибутов.

Мы не будем здесь рассматривать соответствующие математические выкладки, достаточно сказать, что интерполированные значения атрибутов необходимо разделить на соответствующие координаты z (глубины) в каждой вершине.

Интерполированный атрибут процентной доли t расстояния между любой парой вершинных атрибутов A_1 и A_2 можно выразить следующим образом:

$$\frac{A}{p_z} = (1-t)\left(\frac{A_1}{p_{1z}}\right) + t\left(\frac{A_2}{p_{2z}}\right) = \mathrm{LERP}\left(\frac{A_1}{p_{1z}}, \frac{A_2}{p_{2z}}, t\right).$$

Прекрасное разъяснение математических выкладок, лежащих в основе перспективно-корректной интерполяции атрибутов, приводится в книге [32].

Ортогональная проекция. Ортогональная проекция определяется следующей матрицей:

$$\left(\mathbf{M}_{V\to H}\right)_{\mathrm{ortho}} = \begin{bmatrix} \left(\frac{2}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2}{t-b}\right) & 0 & 0 \\ 0 & 0 & \left(-\frac{2}{f-n}\right) & 0 \\ \left(-\frac{r+l}{r-l}\right) & \left(-\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & 1 \end{bmatrix}.$$

Это обычная матрица масштабирования и трансляции. (Вверху слева расположена диагональная матрица 3×3 неоднородного масштабирования, а в нижней строке — трансляция.) Поскольку объем обзора представляет собой прямоугольную призму и в пространстве обзора, и в пространстве отсечения, для преобразования вершин из одного пространства в другое нужно выполнить лишь масштабирование и трансляцию.

Экранное пространство и соотношение сторон

Экранное пространство — это двухмерная система координат, отсчет по осям которой выполняется в экранных пикселах. Ось X обычно направлена вправо, а ось Y — вниз, начало координат — в верхнем левом углу экрана. (Ось Y инвертирована по той причине, что ЭЛТ-мониторы сканируют экран сверху вниз.) Отношение ширины экрана к его высоте называют *соотношением сторон*. Наиболее распространенные соотношения сторон — 4:3 (в традиционных телевизорах) и 16:9 (в кинотеатрах и телевизорах формата HDTV) (рис. 11.36).

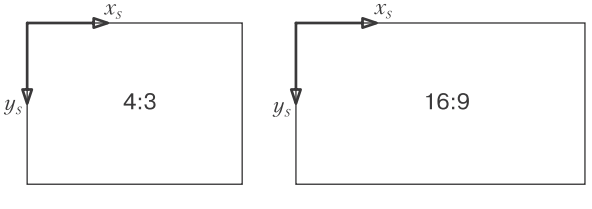


Рис. 11.36. Наиболее распространенные соотношения сторон экрана — 4:3 и 16:9

Рендеринг треугольников, представленных в однородном пространстве отсечения, можно выполнить, просто отрисовав их координаты (x, y) без учета координаты z. Но перед тем, как это сделать, мы должны масштабировать и сместить координаты пространства отсечения таким образом, чтобы они находились в экранном пространстве, а не в нормализованном единичном квадрате. Операция масштабирования и смещения называется *отображением на экран*.

Кадровый буфер

Готовое отрисованное изображение сохраняется в растровом цветовом буфере, который называют *кадровым буфером*. Обычно цвета пикселов сохраняются в формате RGBA8888, однако большинство видеокарт поддерживают и другие форматы кадрового буфера. Так, часто используется поддержка форматов RGB565, RGB5551 и одного или нескольких режимов палитры.

Аппаратное обеспечение устройства отображения (ЭЛТ-монитора, монитора с плоским экраном, телевизора формата HDTV и т. д.) считывает содержимое кадрового буфера с частотой 60 Гц (телевизоры стандарта NTSC, выпускаемые в Северной Америке и Японии) и 50 Гц (телевизоры стандарта PAL/SECAM, распространенные в Европе и множестве других регионов). Движки рендеринга обычно поддерживают минимум два кадровых буфера. Пока аппаратное обеспечение устройства отображения сканирует один буфер, движок рендеринга может обновлять содержимое второго буфера. Такой подход называют *двойной буферизацией*. Попеременное переключение буферов в течение *кадрового интервала гашения* (промежутка времени, в течение которого луч электронной пушки ЭЛТ-монитора возвращается к верхнему левому углу экрана), присущее двойной буферизации, гарантирует, что аппаратное обеспечение устройства отображения всегда будет сканировать полный кадровый буфер. Это позволяет избежать такого раздражающего эффекта, как *разрыв*, при котором в верхней части экрана отображается вновь отрисованное изображение, а в нижней — остатки предыдущего кадра.

В некоторых движках работают три кадровых буфера — как и ожидается, этот метод называют *тройной буферизацией*. Он используется для того, чтобы движок рендеринга мог начать работу над следующим кадром даже в том случае, когда аппаратное обеспечение устройства отображения не успевает просканировать предыдущий кадр. Например, аппаратное обеспечение может еще не закончить сканирование буфера A, когда движок уже завершил отрисовку буфера B. При тройной буферизации движок может приступить к отрисовке нового кадра в буфер C, вместо того чтобы простаивать, ожидая, пока аппаратное обеспечение устройства отображения завершит сканирование буфера A.

Цели рендеринга. Любой буфер, в который движок рендеринга записывает графику, называется *целью рендеринга*. Как мы увидим далее в этой главе, помимо кадровых буферов, движки рендеринга используют и множество внеэкранных целей рендеринга, включая буфер глубины, буфер трафарета и другие буферы для хранения промежуточных результатов рендеринга.

Растеризация треугольников и фрагменты

Чтобы получить изображение треугольника на экране, нужно заполнить те пикселы, которые он перекрывает. Этот процесс называется растеризацией. В ходе нее поверхность треугольника разбивается на части, называемые *фрагментами*, каждый из которых представляет небольшую область поверхности треугольника, соответствующую одному пикселу на экране. (При использовании мультисэмплирующего сглаживания фрагмент соответствует части пиксела — см. далее.)

Фрагмент можно считать пикселом-«студентом». Перед тем как он будет записан в кадровый буфер, он должен пройти ряд тестов (в дальнейшем описаны подробнее). Если он какой-то из них не пройдет, то будет отброшен. Фрагменты, прошедшие тесты, подвергаются затенению (то есть определяется их цвет), после чего цвет фрагмента либо записывается в кадровый буфер, либо смешивается с цветом уже записанного в буфер пиксела. Процесс превращения фрагмента в пиксел иллюстрирует рис. 11.37.

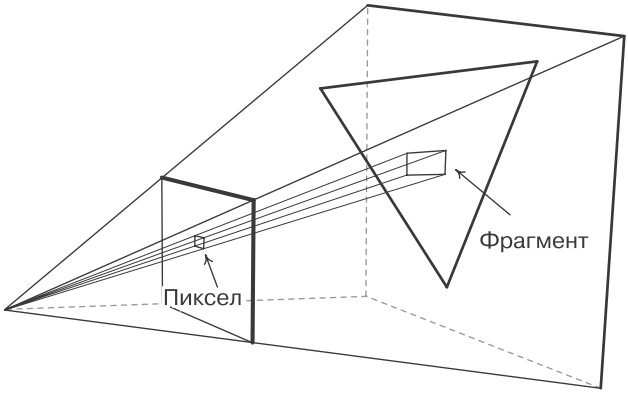


Рис. 11.37. Фрагмент — это небольшая область треугольника, соответствующая экранному пикселу. Он проходит через конвейер рендеринга, после чего либо отбрасывается, либо записывается в кадровый буфер в виде цвета

Окклюзия и буфер глубины

При отрисовке двух треугольников, которые перекрывают друг друга в экранном пространстве, нужно каким-то образом сделать так, чтобы треугольник, находящийся ближе к камере, отображался поверх второго треугольника. Этого можно добиться, всегда отрисовывая треугольники в порядке уменьшения удаленности (это так называемый алгоритм художника). Однако данный прием не сработает, если треугольники пересекаются между собой (рис. 11.38).

Для правильной реализации перекрытия (окклюзии) треугольников, вне зависимости от очередности их отрисовки, в движках рендеринга используется так называемая *буферизация глубины*, или *z-буферизация*. Буфер глубины — это полноэкранный буфер, который обычно содержит 24-битные целочисленные или, реже,

вещественные значения глубины для каждого пиксела в кадровом буфере. (Буфер глубины обычно хранится в 32-битном формате, при этом 24-битное значение глубины и восьмибитное значение трафарета упаковываются в 32-битное четверное слово каждого пиксела.) Каждый фрагмент имеет координату *z*, которая выявляет, насколько глубоко в экране он находится. (Глубина фрагмента определяется интерполяцией глубин вершин треугольника.) Когда цвет фрагмента записывается в кадровый буфер, его глубина сохраняется в соответствующем пикселе буфера глубины. Когда другой фрагмент (из другого треугольника) отрисовывается в том же пикселе, движок сравнивает глубину нового фрагмента с глубиной, уже записанной в буфере глубины. Если этот фрагмент находится ближе к камере, то есть имеет меньшую глубину, его значение заносится в пиксел кадрового буфера вместо старого значения. В противном случае он отбрасывается.

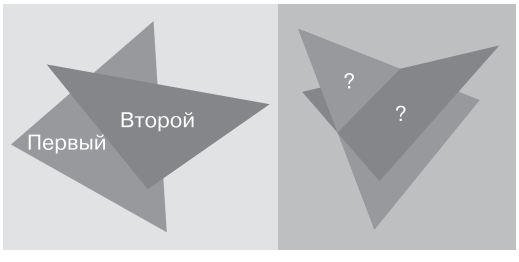


Рис. 11.38. Для правильной реализации окклюзии треугольников алгоритм художника отрисовывает треугольники в порядке уменьшения удаленности. Однако он дает сбой, когда треугольники пересекаются

Z-конфликт и w-буфер. При рендеринге параллельных поверхностей, расположенных очень близко друг к другу, важно, чтобы движок рендеринга мог различать глубины данных плоскостей. С этим никогда не было бы проблем, если бы буфер глубины имел бесконечную точность. К сожалению, глубина точности реального буфера ограниченная, поэтому, когда две плоскости находятся близко друг к другу, их значения глубины могут сливаться в одно дискретное значение. При этом пикселы более удаленной плоскости начинают «пробиваться» сквозь более близкую плоскость, что дает искажающий эффект, известный как *z-конфликт (z-fighting)*.

Чтобы свести к минимуму *z*-конфликт в пределах сцены, необходимо, чтобы точность оставалась одинаковой вне зависимости от того, насколько близко к камере расположена та или иная поверхность. Однако при использовании *z*-буферизации это не так. Точность измерения глубины по оси Z (p_{H_z}) в пространстве отсечения распределена неравномерно в пределах диапазона от передней до задней плоскости из-за деления на координату *z* пространства обзора. В соответствии с формой зависимости $1/z$ поблизости от камеры точность буфера глубины больше.

На рис. 11.39 представлен график функции $p_{H_z} = 1/p_{V_z}$, который демонстрирует этот эффект. Поблизости от камеры расстояние между двумя плоскостями в пространстве обзора Δp_{V_z} преобразуется в довольно большое приращение в пространстве отсечения Δp_{H_z}. Однако на значительном удалении от камеры то же самое

расстояние преобразуется в крошечное приращение в пространстве отсечения. Это ведет к *z*-конфликту, который очень быстро нарастает по мере удаления объектов от камеры.

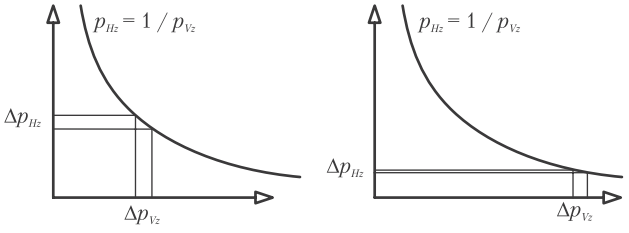


Рис. 11.39. График, показывающий, что поблизости от камеры точность измерения больше

Обойти эту проблему можно, сохраняя в буфере глубины не координаты *z* пространства отсечения (p_{H_z}), а координаты *z пространства обзора* (p_{V_z}). Поскольку координаты *z* пространства обзора линейно изменяются при увеличении удаленности от камеры, их использование для измерения глубины обеспечивает равномерную точность в пределах всего диапазона глубины.

Этот подход называется *w-буферизацией*, потому что координата *z* пространства обзора удобным для нас образом присутствует в компоненте *w* координат однородного пространства отсечения. (Если вы помните, из уравнения (11.1) следует, что $p_{H_w} = -p_{V_z}$.)

Здесь мы имеем дело с несколько запутанной терминологией. И *z*-буфер, и *w*-буфер содержат координаты, выраженные в *пространстве отсечения*. Однако с точки зрения координат *пространства обзора* *z*-буфер содержит значения $1/z$, то есть $1/p_{V_z}$, а *w*-буфер — значения *z*, то есть p_{V_z}!

Следует отметить, что метод *w*-буферизации чуть более затратный по сравнению с *z*-буферизацией. Это объясняется тем, что при использовании *w*-буферизации мы не можем выполнять линейную интерполяцию значений глубины напрямую. Приходится инвертировать значения глубины перед интерполяцией, а затем инвертировать результат перед сохранением его в *w*-буфере.

11.2. Конвейер рендеринга

Теперь, когда мы в общих чертах знаем, какой теоретический и практический фундамент лежит в основе растеризации треугольников, посмотрим, как она обычно проходит. В игровых движках рендеринга в реальном времени, описанных в разделе 11.1, высокоуровневые этапы рендеринга реализуются с использованием программно-аппаратной архитектуры, называемой *конвейером*. Конвейер — это просто упорядоченная цепочка имеющих конкретную цель вычислительных этапов, выполняемых над потоком входных данных для получения потока выходных данных.

Обычно каждый этап конвейера может выполняться независимо от других этапов. Отсюда вытекает одно из главных преимуществ конвейерной архитектуры, состоящее в том, что она хорошо поддается распараллеливанию. Одновременно

с выполнением первого этапа вычислений над некоторым элементом данных может выполняться второй этап вычислений над ранее полученными результатами первого этапа и так далее по цепочке.

Можно распараллеливать и отдельные этапы конвейера. Например, если разместить на кристалле N копий аппаратного обеспечения для выполнения определенного этапа вычислений, то этот этап вычислений можно будет параллельно выполнять над N элементами данных. Схема распараллеленного конвейера показана на рис. 11.40. В идеале все этапы вычислений должны выполняться параллельно (если не постоянно, то практически постоянно) с возможностью параллельной обработки элементов данных на определенных этапах.

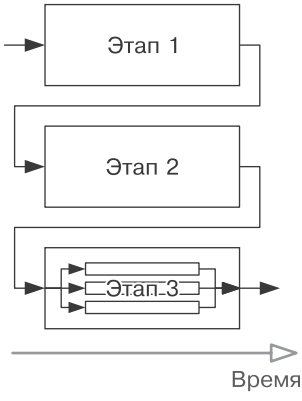


Рис. 11.40. Распараллеленный конвейер. Все вычисления выполняются так, чтобы можно было параллельно обрабатывать элементы данных на определенных этапах

Пропускная способность конвейера — это общее количество элементов данных, обрабатываемых в секунду. *Время задержки* конвейера — это время прохождения через конвейер одного элемента данных. Время задержки отдельного этапа представляет собой время прохождения одного элемента данных через этот этап. Время задержки самого медленного этапа конвейера определяет общую пропускную способность и среднее время задержки всего конвейера. Поэтому при проектировании конвейера рендеринга нужно постараться свести к минимуму и сбалансировать время задержки в пределах всего конвейера, не допуская узких мест. В хорошо спроектированном конвейере все этапы выполняются одновременно и ни на одном из них нет значительных простоев в ожидании, пока выполняется другой этап.

11.2.1. Общая схема конвейера рендеринга

В литературе, посвященной графике, конвейер рендеринга часто разбивают на три основных этапа. Мы же несколько расширим начало этого конвейера, отразив в нем использование автономных инструментов для создания сцен, которые в итоге отрисовывает игровой движок. Таким образом, наша схема конвейера включает в себя следующие высокоуровневые этапы.

- *Этап инструментов (в автономном режиме)*. Определяются геометрия и свойства поверхности (материалы).
- *Этап подготовки ресурсов (в автономном режиме)*. Данные геометрии и материалов преобразуются конвейером подготовки ресурсов в формат, пригодный для использования движком.
- *Этап приложения (центральный процессор)*. Потенциально видимые экземпляры меша выявляются и передаются вместе с материалами графическому аппаратному обеспечению для рендеринга.

- *Этап обработки геометрии (графический процессор)*. Вершины преобразуются, освещаются и проецируются в однородное пространство отсечения. Треугольники обрабатываются опциональным геометрическим шейдером, после чего выполняется их усечение до усеченной пирамиды.
- *Этап растеризации (графический процессор)*. Треугольники преобразуются во фрагменты, которые затеняются, подвергаются ряду тестов (*z*-тесту, α-тесту, тесту трафарета и т. д.) и, наконец, смешиваются с записью результатов в кадровый буфер.

Как конвейер рендеринга преобразует данные

Интересно проследить, как изменяется формат геометрических данных по мере их прохождения через конвейер рендеринга. На этапах инструментов и подготовки ресурсов мы имеем дело с мешами и материалами. На этапе приложения — с экземплярами меша и подмешами, с которыми ассоциированы соответствующие материалы. На этапе обработки геометрии каждый подмеш разбивается на отдельные вершины, которые затем обычно подвергаются параллельной обработке. По завершении этого этапа воссоздаются треугольники из полностью преобразованных и затененных вершин. На этапе растеризации каждый треугольник разбивается на фрагменты, которые либо отбрасываются, либо в конечном итоге записываются в кадровый буфер в виде цветов. Схема этого процесса показана на рис. 11.41.

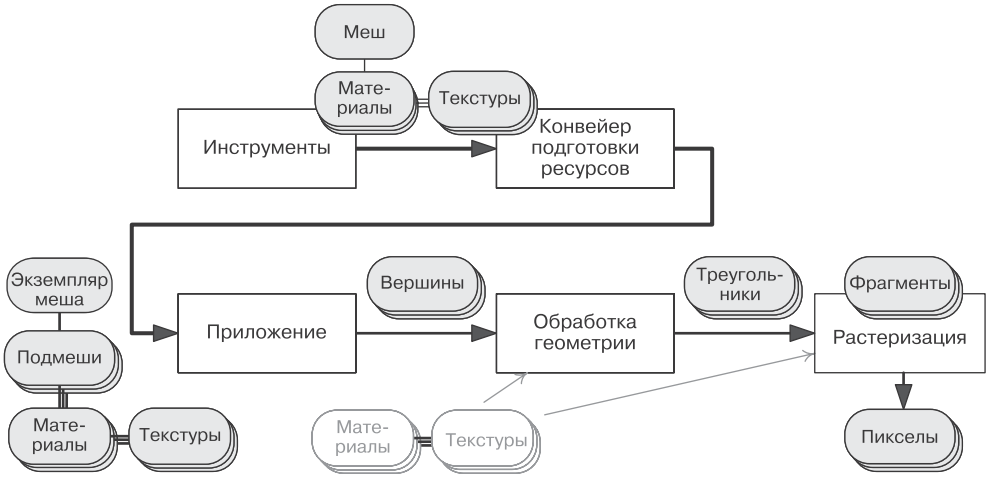


Рис. 11.41. Формат геометрических данных претерпевает значительные изменения по мере их прохождения через различные этапы конвейера рендеринга

Реализация конвейера

Первые два этапа конвейера рендеринга реализуются в автономном режиме чаще всего на компьютере с операционной системой Windows или Linux. Этап при-

ложения обычно выполняет центральный процессор с одним или несколькими ядрами, а этапы обработки геометрии и растеризации — графический процессор. В дальнейших разделах мы рассмотрим некоторые подробности этих этапов.

11.2.2. Инструментальный этап

На этом этапе 3D-моделисты выполняют меши в *приложении для создания цифрового контента*, таком как Maya, 3ds Max, Lightwave, Softimage/XSI, SketchUp и т. д.

Модели могут определяться с использованием любого удобного описания поверхности — неоднородных рациональных B-сплайнов (NURBS), квадрантов, треугольников и т. д. Однако они в любом случае тесселируются в треугольники до их рендеринга с помощью динамической части конвейера.

Вершины меша также могут быть заскинены (подвергнуты скинингу). Это подразумевает привязку каждой вершины к одному или нескольким суставам шарнирно сочлененной скелетной структуры с указанием весовых коэффициентов, определяющих относительную степень воздействия каждого сустава на вершину. Информация о скиннинге и скелет используются системой анимации для управления движениями модели (более подробно об этом написано в главе 12).

На инструментальном этапе художники также определяют материалы. Это подразумевает выбор шейдера для каждого материала, выбор текстур в соответствии с требованиями шейдера и указание параметров конфигурации и настроек каждого шейдера. Текстуры накладываются на поверхности, и определяются другие атрибуты вершин, часто путем их рисования с помощью удобных в использовании инструментов в приложении для создания контента.

Обычно материалы создаются с помощью коммерческого или разработанного самостоятельно редактора материалов. Иногда редактор материалов встраивается непосредственно в приложение для создания контента в виде плагина, но может быть и самостоятельным приложением. Некоторые редакторы материалов напрямую подключены к игре, что позволяет видеть, как материалы будут выглядеть в ней. Другие редакторы отображают результат 3D-визуализации в автономном режиме. Некоторые из них даже предоставляют художнику или программисту шейдеров возможность написать и отладить программу-шейдер. Такие инструменты позволяют быстро прототипировать визуальные эффекты, соединяя узлы различного типа друг с другом с помощью мыши. Обычно в этих инструментах есть окно просмотра внешнего вида готового материала. Примером такого инструмента является Fx Composer компании NVIDIA. К сожалению, компания прекратила выпуск обновлений для Fx Composer, так что этот инструмент поддерживает модели шейдеров только вплоть до DirectX 10. Однако она предлагает новый плагин для Visual Studio, который называется NVIDIA® Nsight™ Visual Studio Edition. Окно плагина Nsight показано на рис. 11.42, он предоставляет мощные возможности создания и отладки шейдеров. Движок Unreal Engine также предлагает графический редактор шейдеров, который называется Material Editor, его окно показано на рис. 11.43.

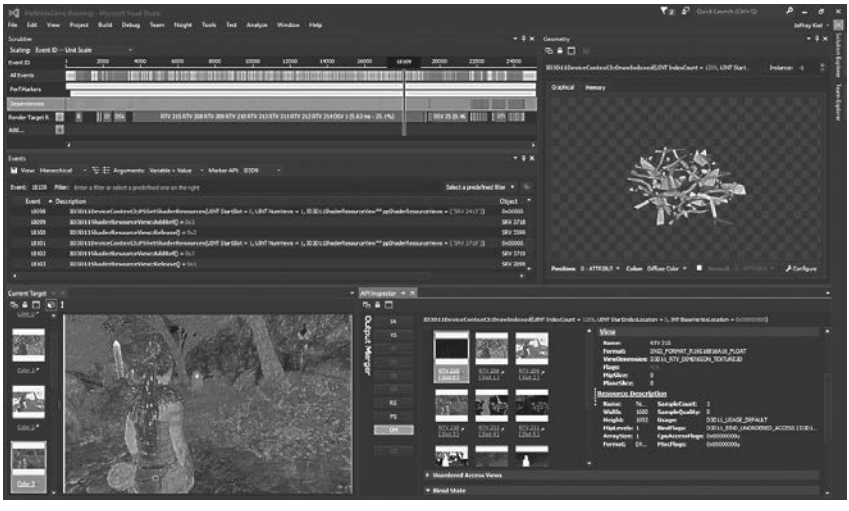


Рис. 11.42. Плагин NVIDIA® Nsight™ Visual Studio Edition позволяет легко писать и отлаживать программы-шейдеры с визуализацией результатов

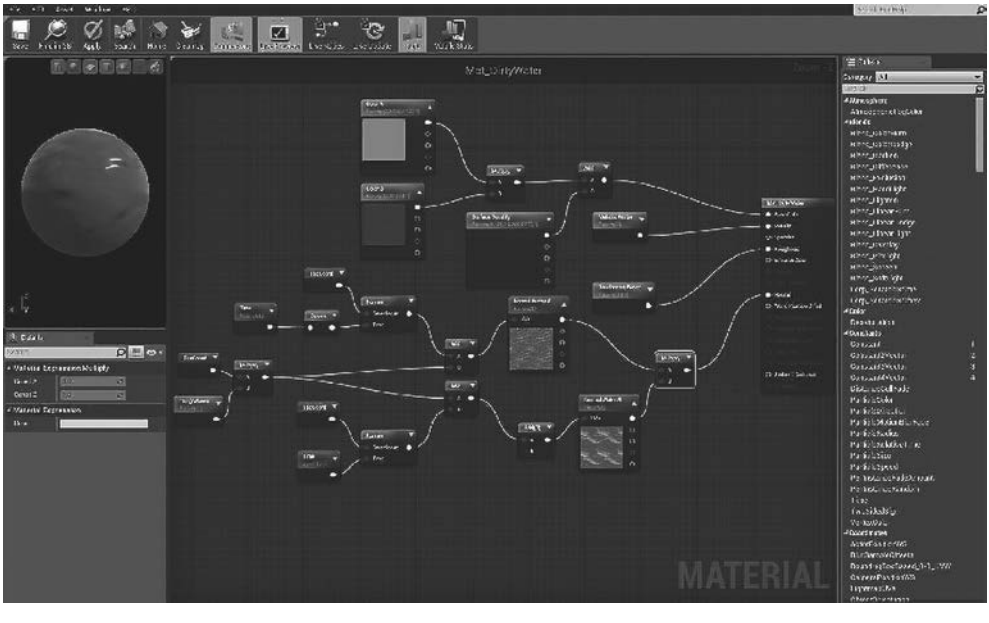


Рис. 11.43. Редактор Unreal Engine 4 Material Editor

Материалы могут храниться и администрироваться совместно с отдельными мешами. Однако это может привести к дублированию данных и выполнению лишней работы. В большинстве игр можно применять сравнительно небольшое количество материалов для создания множества объектов игры. Например, мы

можем определить стандартные часто используемые материалы, такие как дерево, камень, металл, пластик, ткань, кожа и т. д. Нет никакого смысла дублировать их внутри каждого меша. Вместо этого команда разработчиков игры обычно создает библиотеку доступных для выбора материалов, и отдельные меши ссылаются на эти материалы в слабосвязанной манере.

11.2.3. Этап подготовки ресурсов

Этап подготовки ресурсов сам по себе является конвейером, который иногда называют *конвейером подготовки ресурсов* или *конвейером инструментов*. Как мы видели в подразделе 7.2.1, он призван обеспечить экспорт, обработку и соединение воедино множества ресурсов различного типа. Например, 3D-модель включает в себя геометрию (вершинный и индексный буферы), материалы, текстуры и в некоторых случаях скелет. Конвейер подготовки ресурсов обеспечивает доступность и готовность к загрузке движком всех тех отдельных ресурсов, на которые ссылается 3D-модель.

Геометрические и материальные данные извлекаются из приложения для создания контента и, как правило, сохраняются в промежуточном платформенно-независимом формате. Затем эти данные подвергаются дальнейшему преобразованию в один или несколько платформенно-специфичных форматов в зависимости от того, сколько целевых платформ поддерживает движок. В идеале платформенно-специфичные ресурсы, полученные на этом этапе, должны быть готовыми к загрузке в память и применению в динамическом режиме с минимальной постобработкой или вообще без постобработки. Например, данные меша, предназначенные для Xbox One или PS4, могут выдаваться в виде индексного и вершинного буферов, готовых для использования графическим процессором. Для PS3 геометрия может выдаваться в виде сжатых потоков данных, готовых к загрузке посредством прямого доступа к памяти в синергичные процессорные блоки для их декомпрессии. Конвейер подготовки ресурсов часто учитывает потребности материала/шейдера при построении ресурсов. Например, если определенному шейдеру наряду с нормалью вершины потребуются касательный и бикасательный векторы, конвейер подготовки ресурсов может автоматически генерировать их.

Также на этапе подготовки ресурсов могут рассчитываться высокоуровневые структуры данных, называемые *графом сцены*. Например, может обрабатываться геометрия статического уровня для построения BSP-дерева (Binary Space Partitioning — бинарное разбиение пространства). (Как будет показано в подразделе 11.2.7, структуры данных графа сцены помогают движку рендеринга быстро определять, какие объекты следует отрисовывать при заданных положении и ориентации камеры.)

В автономном режиме в качестве составной части этапа подготовки ресурсов часто выполняются высокозатратные расчеты освещения. Эти расчеты так называемого *статического освещения* могут включать в себя расчет цвета освещения в вершинах меша (оно называется «запеченным» освещением вершин), построение *карт освещения* — текстурных карт с попиксельной информацией об освещении, расчет

коэффициентов *предрассчитанного переноса излучения* (precomputed radiance transfer, PRT), обычно представленных сферическими гармоническими функциями, и пр.

11.2.4. Конвейер графического процессора

Центральную роль в развитии графического аппаратного обеспечения играет специализированный тип микропроцессора, называемый *графическим процессором*. Как упоминалось в разделе 4.11, он призван обеспечивать максимальную пропускную способность графического конвейера, что достигается за счет активного распараллеливания таких задач, как обработка вершин и попиксельные расчеты затенения.

Так, пиковая производительность современного графического процессора AMD Radeon™ 7970 может доходить до 4 TFLOPS, что обеспечивается параллельным выполнением рабочих нагрузок 32 вычислительными блоками, каждый из которых содержит четыре векторных блока с 16-полосной обработкой по технологии SIMD (Single Instruction, Multiple Data — одиночный поток команд, множественный поток данных), которые, в свою очередь, выполняют конвейерную обработку фронтов волны вычислений, включающих в себя по 64 потока. Графический процессор предназначен для отрисовывания графики, но, поскольку современные графические процессоры полностью программируемые, можно применять их потрясающие вычислительные возможности для использования *вычислительных шейдеров*. Этот подход называется *выполнением вычислений общего назначения на графических процессорах*.

Практически все графические процессоры разбивают графический конвейер на подэтапы (рис. 11.44), описанные далее. Различной заливкой на схеме показано, является функциональность того или иного этапа программируемой, фиксированной настраиваемой или фиксированной ненастраиваемой.

Вершинный шейдер

Это полностью программируемый этап. Он отвечает за преобразование и затенение/освещение отдельных вершин. Входной информацией для него является одна вершина (хотя на практике параллельно обрабатывается множество вершин). Положение и нормаль вершины обычно выражаются в пространстве модели или мировом пространстве. Вершинный шейдер выполняет преобразование из пространства модели в пространство обзора с помощью матрицы преобразования модели в обзор. Также применяется перспективная проекция вместе с повершинными расчетами освещения и текстурирования, а также скиннинга для анимированных персонажей. Вершинный шейдер может выполнять процедурную анимацию, изменяя положение вершины. Примером такой анимации служит раскачивание листвы на ветру или волнообразное колебание водной поверхности. Выходной информацией данного этапа является полностью преобразованная и освещенная вершина, положение и нормаль которой выражены в однородном пространстве отсечения (см. подраздел 11.1.4).

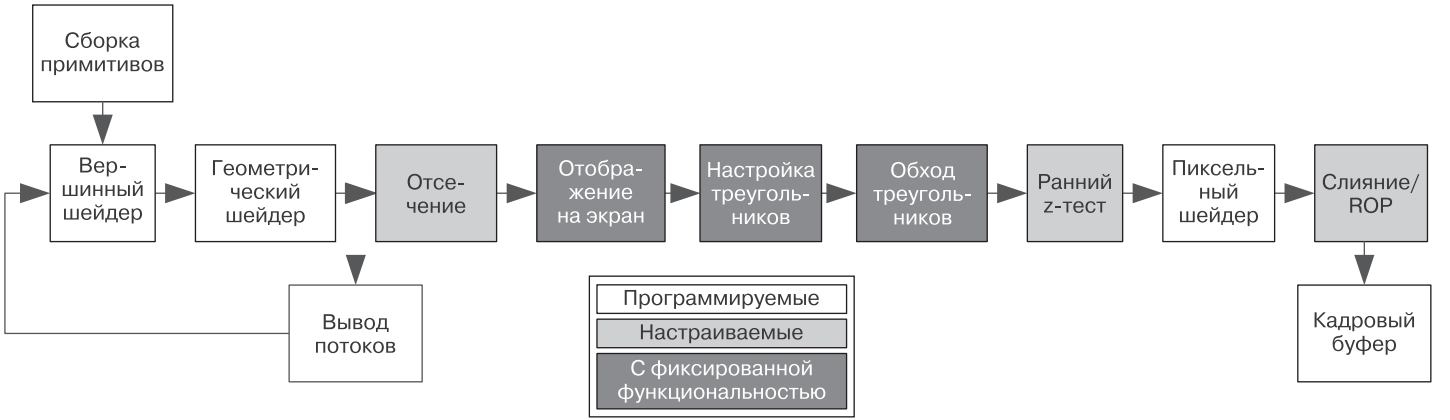


Рис. 11.44. Схема реализации этапов обработки геометрии и растеризации конвейера рендеринга в типичном графическом процессоре. Белой заливкой обозначены программируемые этапы, светло-серой — настраиваемые этапы, темно-серой — этапы с фиксированной функциональностью

В современных графических процессорах вершинный шейдер имеет полный доступ к текстурным данным, а раньше доступ к ним получали только пиксельные шейдеры. Это особенно полезно при использовании текстур в качестве отдельных структур данных, таких как карты высот или таблицы преобразования.

Геометрический шейдер

Этот опциональный этап также полностью программируем. Геометрический шейдер работает с целыми примитивами (треугольниками, линиями и точками) в однородном пространстве отсечения. Он способен выбраковывать и модифицировать входные примитивы, а также генерировать новые. Типичные области его применения — выдавливание теневых объемов (см. подраздел 11.3.3), рендеринг шести граней кубической карты (см. подраздел 11.3.1), выдавливание ребер меша вокруг контуров сетки, создание квадрантов частиц на основе точечных данных (см. подраздел 11.4.1), динамическая тесселяция, фрактальное подразбиение линейных отрезков для имитации молнии, моделирование ткани и т. д.

Вывод потоков

Некоторые графические процессоры позволяют записывать обратно в память данные, полученные в конвейере к этому моменту. Оттуда они могут быть возвращены в начало конвейера для дополнительной обработки. Эта функция называется *выводом потоков*.

Вывод потоков позволяет создавать ряд интересных визуальных эффектов без использования центрального процессора. Прекрасный пример — рендеринг волос. Часто волосы представляют в виде набора кубических сплайновых кривых. При этом раньше дело обстояло так: центральный процессор выполнял симуляцию физики волос и тесселяцию сплайнов на линейные отрезки, а после этого графический процессор делал рендеринг отрезков.

Теперь же благодаря потоковому выводу графический процессор может выполнять симуляцию физики в контрольных точках сплайнов волос в вершинном шейдере. Геометрический шейдер тесселирует сплайны, после чего задействуется функция вывода потоков для записи тесселированных вершинных данных в память. После этого поток линейных отрезков возвращается в начало конвейера для их рендеринга.

Отсечение

На данном этапе отсекаются части треугольников, выходящих за границы усеченной пирамиды. Для этого сначала выявляют вершины, расположенные за пределами усеченной пирамиды, а затем определяют точки пересечения ребер треугольника с плоскостями усеченной пирамиды. Точки пересечения становятся новыми вершинами, которые определяют один или несколько усеченных треугольников.

Данный этап обладает фиксированной, но в то же время в определенной мере настраиваемой функциональностью. Так, например, помимо плоскостей усеченной

пирамиды, можно использовать дополнительные пользовательские плоскости отсечения. Кроме того, можно задать такие настройки, чтобы отбраковывались треугольники, расположенные за пределами усеченной пирамиды.

Отображение на экран

На этапе отображения на экран выполняется масштабирование и перемещение вершин из однородного пространства отсечения в экранное пространство. Это этап с полностью фиксированной ненастраиваемой функциональностью.

Настройка треугольников

На этапе настройки треугольников инициализируется аппаратное обеспечение для растеризации с целью эффективного разбиения треугольников на фрагменты. Это ненастраиваемый этап.

Обход треугольников

На этом этапе все треугольники разбиваются на фрагменты, то есть растеризуются. Обычно генерируется один фрагмент на каждый пиксел, однако в случае применения определенных методов сглаживания могут создаваться сразу несколько фрагментов на каждый пиксел (см. подраздел 11.1.4). На данном этапе также интерполируются вершинные атрибуты с целью получения пофрагментных атрибутов для последующей обработки их пиксельным шейдером. Если необходимо, задействуется перспективно-корректная интерполяция. Это этап с фиксированной ненастраиваемой функциональностью.

Ранний z-тест

Многие видеокарты могут в этой точке конвейера проверять глубину фрагмента, отбрасывая его, если он перекрывается пикселом, который уже присутствует в кадровом буфере. Это позволяет полностью исключить для перекрытых фрагментов этап пиксельного шейдера, который может быть очень затратным.

Что удивительно, не все графические аппаратные средства поддерживают проверку глубины в этой точке конвейера. В старых моделях графических процессоров *z*-тест выполнялся вместе с альфа-тестом после запуска пиксельного шейдера. По этой причине данный этап называют *ранним z-тестом* или *ранним тестом глубины*.

Пиксельный шейдер

Это полностью программируемый этап. Здесь выполняется затенение (то есть освещение и другая обработка) каждого фрагмента. Пиксельный шейдер также может отбрасывать фрагменты, например, когда выясняется, что они абсолютно прозрачные. Пиксельный шейдер способен получать доступ к одной или нескольким текстурным картам, выполнять попиксельные расчеты освещения и любые другие операции, необходимые для определения цвета фрагментов.

Входной информацией для данного этапа является набор пофрагментных атрибутов, которые были интерполированы на основе вершинных атрибутов на этапе обхода треугольников. Выходная информация — один цветовой вектор, описывающий требуемый цвет фрагмента.

Этап слияния/растровых операций

Последний этап конвейера известен как этап *слияния* или *смешивания* либо, в терминологии компании NVIDIA, этап *растровых операций* (raster operations, ROP). Это непрограммируемый, но в значительной мере настраиваемый этап. Здесь к фрагментам применяется ряд тестов, в том числе тест глубины (см. подраздел 11.1.4), альфа-тест, при котором некоторые фрагменты могут быть отбракованы на основе значений альфа-канала фрагмента и пиксела, и тест трафарета (см. подраздел 11.3.3).

Если фрагмент проходит все тесты, его цвет смешивается (сливается) с цветом, который уже присутствует в кадровом буфере. Способ смешивания контролируется *функцией альфа-смешивания*, которая, имея жестко заданную базовую структуру, тем не менее позволяет выполнять широкий спектр операций смешивания за счет настройки ее операторов и параметров.

Чаще всего альфа-смешивание используется для рендеринга полупрозрачной геометрии. В этом случае применяется функция смешивания:

$$\mathbf{C}'_D = A_S\mathbf{C}_S + (1 - A_S)\mathbf{C}_D.$$

Подстрочными индексами S и D здесь обозначены «источник» (source) — поступающий на вход фрагмент и «цель» (destination) — пиксел в кадровом буфере соответственно. Таким образом, записываемый в кадровый буфер цвет ($\mathbf{C}'_D$) представляет собой *средневзвешенное значение* содержимого кадрового буфера ($\mathbf{C}_D$) и цвета рисуемого фрагмента ($\mathbf{C}_S$). Весовой коэффициент смешивания (A_S) — это значение альфа-канала поступающего фрагмента.

Чтобы обеспечить надлежащий результат альфа-смешивания, необходимо сортировать и отрисовывать имеющиеся в сцене полупрозрачные поверхности в порядке уменьшения удаленности *после* отрисовки непрозрачной геометрии в кадровый буфер. Это объясняется тем, что при выполнении альфа-смешивания значение глубины нового фрагмента *перезаписывает* значение глубины того пиксела, с которым он смешивается. То есть буфер глубины не учитывает прозрачность, за исключением случая отключения записи значений глубины. Если нужно отрисовать стопку полупрозрачных объектов поверх непрозрачного фона, то в идеале окончательный цвет пиксела должен представлять собой результат смешивания цвета непрозрачной поверхности с цветами *всех* полупрозрачных поверхностей в стопке. Если мы попытаемся отрисовать стопку не в порядке уменьшения удаленности, то некоторые полупрозрачные фрагменты не пройдут тест глубины и будут отброшены, результатом чего станет неполное смешивание (и странного вида изображение).

Можно определить и другие функции альфа-смешивания, не предназначенные для наложения прозрачности. Общее уравнение смешивания имеет вид

$\mathbf{C}'_D = (\mathbf{w}_S \otimes \mathbf{C}_S) + (\mathbf{w}_D \otimes \mathbf{C}_D)$, где весовые коэффициенты $\mathbf{w}_S$ и $\mathbf{w}_D$ могут быть выбраны программистом из предварительно определенного набора значений, включающего ноль, единицу, цвет источника или цели, альфа-значение источника или цели и результат вычитания из единицы цвета или альфа-значения источника или цели.

Оператор $\otimes$ представляет либо обычное умножение скалярной величины на вектор, либо покомпонентное умножение двух векторов (произведение Адамара — см. подраздел 5.2.4) в зависимости от типа данных коэффициентов $\mathbf{w}_S$ и $\mathbf{w}_D$.

11.2.5. Программируемые шейдеры

Теперь, когда мы в общих чертах знаем, что представляет собой конвейер графического процессора, давайте чуть подробнее рассмотрим его самую интересную часть — программируемые шейдеры. Архитектура шейдеров претерпела значительные изменения с момента их первого появления в DirectX 8. Первые *шейдерные модели* поддерживали только низкоуровневое программирование на ассемблере, и в пиксельном и вершинном шейдерах использовались сильно различающиеся наборы команд и регистров. В DirectX 9 появилась поддержка высокоуровневых C-подобных шейдерных языков, таких как Cg (C for graphics — C для графики), HLSL (High-Level Shading Language — высокоуровневый язык шейдеров, реализация языка Cg компании Microsoft) и GLSL (OpenGL Shading Language — язык шейдеров OpenGL). Нововведением версии DirectX 10 стал геометрический шейдер с унифицированной шейдерной архитектурой, которая в терминологии DirectX называется *шейдерной моделью 4.0*. В унифицированной шейдерной модели все три типа шейдеров поддерживают примерно одинаковый набор команд и имеют примерно одинаковый набор возможностей, включая возможность чтения текстурной памяти.

Шейдер принимает один элемент входных данных и преобразует его в ноль или более элементов выходных данных.

- На вход вершинного шейдера поступает вершина, положение и нормаль которой выражены в пространстве модели или мировом пространстве. На выходе вершинный шейдер выдает полностью преобразованную и освещенную вершину, выраженную в однородном пространстве отсечения.
- Входной информацией для геометрического шейдера являются один n-вершинный примитив — точка ($n = 1$), линейный отрезок ($n = 2$) или треугольник ($n = 3$) — и n дополнительных вершин, которые выступают в роли контрольных точек. На выходе данный шейдер выдает ноль или более примитивов, тип которых может отличаться от типа входного примитива. Например, геометрический шейдер может преобразовывать точки в квадранты, состоящие из двух треугольников, либо треугольники — в треугольники, при необходимости отбрасывая некоторые из них, и т. д.
- На вход пиксельного шейдера поступает фрагмент, атрибуты которого были интерполированы на основе трех вершин того треугольника, из которого он был получен. На выходе пиксельный шейдер выдает цвет, который записывается

в кадровый буфер (при условии успешного прохождения фрагментом теста глубины и других опциональных тестов). Пиксельный шейдер может явно отбраковывать фрагменты, не выдавая при этом никакой информации.

Доступ к памяти

Поскольку графический процессор реализует конвейер обработки данных, большое внимание при этом уделяется доступу к оперативной памяти. Программа-шейдер обычно не может выполнять чтение из памяти и запись в память напрямую. Ее доступ к памяти ограничен двумя методами: использованием регистров и текстурных карт.

В то же время следует отметить, что эти ограничения снимаются в системах с непосредственным совместным использованием памяти центральным и графическим процессорами. Например, интегральная система AMD Jaguar, применяемая в приставках PlayStation 4, — пример *гетерогенной системной архитектуры* (Heterogeneous System Architecture, HSA). В системах, не использующих архитектуру HSA, центральный и графический процессоры обычно представляют собой отдельные устройства, каждое из которых имеет собственную память и, как правило, размещается на отдельной плате. Передавать данные между двумя процессорами приходится посредством довольно медленной коммуникации с высоким временем задержки по специальной шине, такой как AGP или PCIe. В отличие от этого, при использовании архитектуры HSA центральный и графический процессоры совместно задействуют единое запоминающее устройство, которое называется *гетерогенной унифицированной архитектурой памяти* (heterogeneous Unified Memory Architecture, hUMA). Поэтому шейдерам, работающим в системе с hUMA, такой как PS4, можно передать в качестве входных данных *таблицу ресурсов шейдера* (Shader Resource Table, SRT). Это просто указатель на размещенную в памяти структуру (`struct`) языка C/C ++, чтение и запись которой могут осуществлять и центральный процессор, и шейдер, запущенный на графическом процессоре. На PS4 таблицы SRT используются вместо описанных далее константных регистров.

Регистры шейдеров. Шейдер может получать доступ к оперативной памяти косвенно посредством *регистров*. Все регистры графического процессора имеют 128-битный формат SIMD. Каждый из них может вместить четыре 32-битных вещественных или целочисленных значения, представляемых в языке Cg с помощью типа данных `float4`. Таким образом, содержимое каждого регистра может описывать четырехэлементный вектор в однородных координатах или цвет в формате RGBA, где каждая составляющая цвета представлена 32-битным числом с плавающей запятой. Матрицы могут быть представлены группами из трех или четырех регистров, представляемых в языке Cg с помощью встроенных матричных типов, таких как `float4x4`. Регистр графического процессора можно использовать также для размещения одного 32-битного скалярного значения, в этом случае значение обычно дублируется во всех четырех 32-битных полях. Некоторые графические процессоры могут производить действия над 16-битными полями, которые на-

зывают *полусловами*. (В языке Cg для этой цели предусмотрены встроенные типы `half4` и `half4x4`.)

Существует четыре разновидности регистров:

- *Входные регистры*. Являются основным источником входных данных для шейдера. Входные регистры вершинного шейдера содержат данные атрибутов, полученные непосредственно из вершин. Входные регистры пиксельного шейдера содержат интерполированные данные вершинных атрибутов, соответствующие одному фрагменту. Значения всех входных регистров автоматически устанавливаются графическим процессором перед вызовом шейдера.
- *Константные регистры*. Значения константных регистров устанавливаются приложением и могут изменяться от примитива к примитиву. Эти значения постоянны только с точки зрения программы-шейдера. Данные регистры являются вторичным источником входных данных для шейдера. Обычно в них размещаются матрица преобразования модели в обзор, матрица проекции, параметры освещения и любые другие необходимые шейдеру параметры, которые не входят в число вершинных атрибутов.
- *Временные регистры*. Предназначены для использования программой-шейдером и обычно хранят промежуточные результаты вычислений.
- *Выходные регистры*. Заполняются шейдером и являются его единственным средством вывода данных. Выходные регистры вершинного шейдера содержат такие вершинные атрибуты, как преобразованные позиционные и нормальные векторы в однородном пространстве отсечения, опциональные цвета вершин, текстурные координаты и т. д. Выходной регистр пиксельного шейдера содержит окончательный цвет затеняемого фрагмента.

Приложение предоставляет значения константных регистров, когда передает примитивы для рендеринга. Графический процессор автоматически копирует данные атрибутов вершин или фрагментов из видеопамяти в соответствующие входные регистры перед вызовом программы-шейдера, а по ее завершении записывает содержимое выходных регистров обратно в ОЗУ для передачи этих данных на следующий этап конвейера.

Графические процессоры обычно кэшируют выходные данные, чтобы их можно было использовать повторно, не пересчитывая. Так, например, в *кэше преобразованных вершин* размещаются последние обработанные вершины, выдаваемые вершинным шейдером. Если при этом какой-либо треугольник будет ссылаться на уже обработанную вершину, то она по возможности будет считана из кэша преобразованных вершин. Повторно вершинный шейдер будет вызван, только если за прошедшее время рассматриваемая вершина уже удалена из кэша для высвобождения места под вновь обработанные вершины.

Текстуры. Шейдер позволяет получить прямой доступ для чтения к *текстурным картам*. Данные текстур адресуются с помощью текстурных координат, а не абсолютных адресов памяти. Сэмплеры текстур графического процессора автоматически *фильтруют* данные текстур, при необходимости смешивая значения

смежных текселов или соседних MIP-уровней. Фильтрацию текстур можно отключать для получения прямого доступа к значениям определенных текселов. Это может быть полезно, например, при использовании текстурной карты в качестве таблицы данных.

Шейдеры могут выполнять *запись* в текстурные карты только в косвенной манере — путем рендеринга сцены во внеэкранный кадровый буфер, который интерпретируется как текстурная карта при последующих проходах рендеринга. Эта функция называется *рендерингом в текстуру*.

Введение в синтаксис высокоуровневых шейдерных языков

Высокоуровневые языки шейдеров, такие как Cg и GLSL, устроены во многом так же, как язык программирования C. Они позволяют программисту объявлять функции, определять простую структуру (`struct`) и выполнять арифметические действия. Однако, как уже упоминалось, программа-шейдер имеет доступ только к регистрам и текстурам. В силу этого структуры (`struct`) и переменные, объявляемые на языке Cg или GLSL, отображаются компилятором шейдеров непосредственно на регистры. Существует несколько способов определения такого отображения.

- *Использование семантики*. После переменных и членов структур (`struct`) можно через двоеточие ставить ключевые слова, которые называют *семантикой*. Семантика дает компилятору шейдеров указание связать переменную или переменную-член с определенным атрибутом вершины или фрагмента. Например, в вершинном шейдере объявить входную структуру (`struct`), члены которой будут отображены на атрибуты *положения* и *цвета* вершины, можно следующим образом:

```
struct VtxOut
{
    float4 pos : POSITION; // отображение на атрибут положения
    float4 color : COLOR;  // отображение на атрибут цвета
};
```

- *Отображение на вход или выход*. Компилятор может определить, на какие регистры следует отображать конкретную переменную или структуру (`struct`) — входные или выходные, исходя из контекста ее использования. Если переменная передается в качестве аргумента функции `main` программы-шейдера, то она относится к входным данным, если же в качестве возвращаемого значения функции `main`, то к выходным.

```
VtxOut vshaderMain(VtxIn in) // отображение на входные регистры
{
    VtxOut out;
    // ...
    return out;  // отображение на выходные регистры
}
```

- *Унифицированное объявление*. Чтобы получить доступ к данным, которые предоставляются приложением через константные регистры, можно объявить пере-

менную, используя ключевое слово `uniform`. Например, матрицу преобразования модели в обзор можно передать в вершинный шейдер следующим образом:

```
VtxOut vshaderMain(
    VtxIn in,
    uniform float4x4 modelViewMatrix)
{
    VtxOut out;
    // ...
    return out;
}
```

Арифметические операции можно выполнять, вызывая операторы в стиле языка C и встроенные функции там, где это уместно. Например, чтобы умножить входное положение вершины на матрицу преобразования модели в обзор, можно записать следующий код:

```
VtxOut vshaderMain(VtxIn in,
                   uniform float4x4 modelViewMatrix)
{
    VtxOut out;

    out.pos = mul(modelViewMatrix, in.pos);
    out.color = float4(0, 1, 0, 1); // зеленый цвет RGBA

    return out;
}
```

Извлекаются данные из текстур с помощью вызова специальных встроенных функций, считывающих значения текселов с указанными текстурными координатами. Есть несколько вариантов таких функций для чтения одномерных, двухмерных и трехмерных текстур в различных форматах как с фильтрацией, так и без нее. Имеются также специальные режимы адресации текстур для доступа к кубическим и теневым картам. Ссылки на сами текстурные карты объявляются с применением специального типа данных, называемого *сэмплером текстур*. Например, для объявления ссылки на обычную двухмерную текстуру используется тип данных `sampler2D`. Представленный далее простой пиксельный шейдер на языке Cg накладывает диффузную текстуру на треугольник:

```
struct FragmentOut
{
    float4 color : COLOR;
};

FragmentOut pshaderMain(float2 uv : TEXCOORD0,
                        uniform sampler2D texture)
{
    FragmentOut out;
    // чтение текселя с координатами (u,v)
    out.color = tex2D(texture, uv);

    return out;
}
```

Файлы эффектов

Программа-шейдер не приносит какой-то пользы сама по себе. Для вызова шейдера со значимыми входными данными конвейеру графического процессора требуется некоторая дополнительная информация. Например, нужно указать, каким образом определяемые приложением параметры, такие как матрица преобразования модели в обзор, параметры освещения и т. д., должны отображаться на переменные `uniform`, объявленные в программе-шейдере. Кроме того, некоторые визуальные эффекты требуют двух или более проходов рендеринга, однако программа-шейдер определяет набор операций только для одного прохода. При написании игры для платформы PC нужно будет определить запасные версии некоторых наиболее продвинутых эффектов рендеринга, способных работать даже на старых моделях видеокарт. Чтобы связать программы-шейдеры в единый визуальный эффект, потребуется воспользоваться файловым форматом, который называют *файлом эффекта*.

В различных движках рендеринга эффекты реализованы немного по-разному. В языке Cg формат файла эффекта называется *CgFX*. В движке рендеринга OGRE используется очень похожий на CgFX файловый формат, который называется *файлом материала*. В языке GLSL эффекты можно описать с помощью формата COLLADA, основанного на XML. Несмотря на эти различия, в большинстве случаев эффекты имеют следующую иерархическую структуру.

- В глобальной области видимости определяются структуры, программы-шейдеры, реализуемые в виде различных функций `main`, и глобальные переменные, которые отображаются на определяемые приложением константные параметры.
- Определяется одна или несколько *техник*. Техника представляет собой один из способов рендеринга определенного визуального эффекта. Обычно эффект реализуется с наилучшим качеством с помощью какой-то основной техники и, возможно, нескольких запасных техник, предназначенных для использования на менее мощном графическом аппаратном обеспечении.
- В пределах каждой техники определяется один или несколько *проходов*. Проход определяет, как следует отрисовывать одно полнокадровое изображение. Обычно это определение содержит ссылку на функцию `main` вершинного, геометрического и/или пиксельного шейдера, различные привязки параметров и опциональные настройки состояния рендеринга.

Дополнительная литература

В этом разделе мы лишь слегка коснулись того, что представляет собой высокоуровневое программирование шейдеров. Всестороннее изучение этой темы выходит за рамки данной книги. Намного более подробное введение в программирование шейдеров на языке Cg можно найти в руководстве по языку Cg, доступном на веб-сайте компании NVIDIA по адресу developer.nvidia.com/content/hello-cg-introductory-tutorial.

11.2.6. Сглаживание

После растеризации треугольника его края обычно имеют зубчатый вид — это та самая ступенчатость (или лестничный эффект), с которой сталкивался и которую успел полюбить (или возненавидеть) каждый из нас. В техническом смысле ступенчатость возникает потому, что мы используем дискретный набор пикселов для *сэмплирования* изображения, которое в действительности представляет собой плавный, непрерывный двухмерный сигнал. (Подробное обсуждение сэмплирования и ступенчатости см. в подразделе 14.3.2.)

Под *сглаживанием* понимается любой метод, позволяющий сократить степень проявления визуальных артефактов, вызываемых ступенчатостью. Существует множество различных способов сглаживания отрисованной сцены. Практически все они обеспечивают смягчение краев отрисованных треугольников за счет смешивания их с окружающими пикселами. В то же время каждый из этих методов обладает уникальными характеристиками в плане производительности, использования памяти и обеспечиваемого качества. На рис. 11.45 показана сцена, отрисованная сначала без сглаживания, а затем с помощью методов четырехкратного сглаживания MSAA и сглаживания FXAA компании NVIDIA.

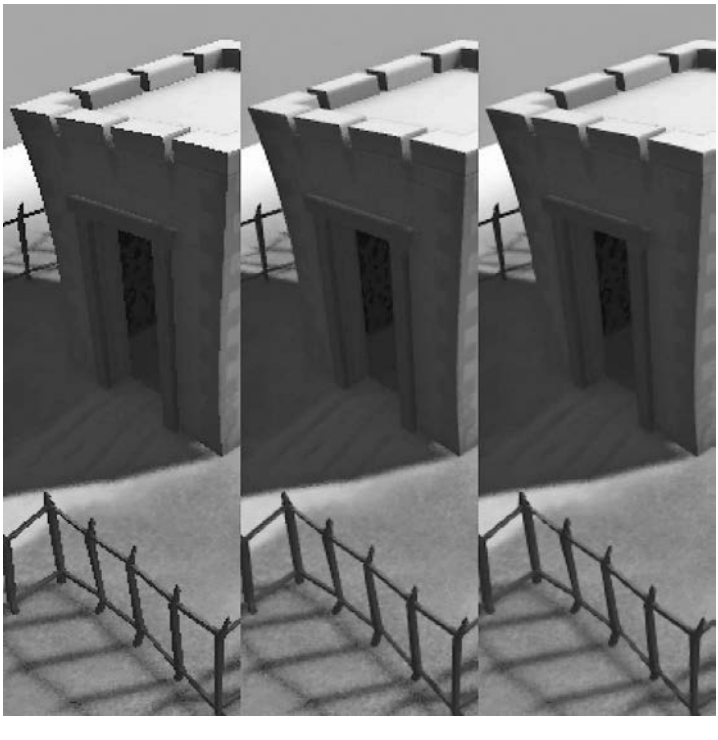


Рис. 11.45. Без сглаживания *(слева)*, четырехкратное сглаживание MSAA *(по центру)*, сглаживание FXAA компании NVIDIA *(справа)*. Изображение из «белой книги» по методу FXAA компании NVIDIA, автор — Тимоти Лоттес (bit.ly/1mIzCTv)

Полноэкранное сглаживание

При использовании метода полноэкранного сглаживания (Full-Screen Antialiasing, FSAA), называемого также *сглаживанием на основе суперсэмплирования* (supersampled antialiasing, SSAA), сцена отрисовывается в кадровый буфер большего размера по сравнению с фактическим размером экрана. Затем выполняется *понижающая дискретизация* (или *прореживание*) полученного увеличенного изображения до нужного разрешения. При четырехкратном суперсэмплировании отрисованное изображение в два раза больше экрана по ширине и высоте, в результате чего кадровый буфер занимает в четыре раза больший объем памяти. Это требует также в четыре раза большей вычислительной мощности графического процессора, поскольку пиксельный шейдер требуется запускать четыре раза для каждого экранного пиксела. Как видите, FSAA — невероятно затратный метод в смысле потребления памяти и использования вычислительной мощности графического процессора. Поэтому он редко применяется на практике.

Мультисэмплирующее сглаживание

Метод *мультисэмплирующего сглаживания* (MSAA) обеспечивает сравнимое с FSAA визуальное качество при гораздо меньшем потреблении пропускной способности графического процессора (и таком же уровне потребления видеопамяти). Метод MSAA основан на том, что благодаря естественному сглаживающему эффекту MIP-текстурирования ступенчатость обычно представляет проблему лишь *по краям* треугольников, но не в их внутренней части.

Чтобы понять принцип действия метода MSAA, нужно вспомнить, что процесс растеризации треугольника фактически сводится к выполнению трех отдельных операций:

- определению того, какие пикселы перекрывает (покрывает) треугольник;
- проверке того, не перекрыт ли каждый пиксел каким-либо другим треугольником (тест глубины);
- определению цвета каждого пиксела при условии, что результаты тестов покрытия и глубины указывают, что пиксел должен быть отрисован (затенение пикселов).

При растеризации треугольника без сглаживания все операции — и тест покрытия, и тест глубины, и затенение пикселов — выполняются в одной идеализированной точке каждого экранного пиксела, обычно расположенной в его центре. При сглаживании по методу MSAA тесты покрытия и глубины выполняются для N точек внутри каждого экранного пиксела, называемых точками *подвыборки*. Обычно N выбирают равным 2, 4, 5, 8 или 16. Однако при этом пиксельный шейдер запускается только *один раз* для каждого экранного пиксела вне зависимости от количества точек подвыборки. В силу этого метод MSAA выглядит намного предпочтительнее метода FSAA в плане пропускной способности графического процессора, поскольку операция затенения обычно гораздо затратнее, чем тесты покрытия и глубины. При N-кратном сглаживании MSAA выделяются в N раз большие буферы глубины, трафарета и цвета по сравнению с их обычным размером. Для каждого пиксела экрана эти буферы

содержат по N ячеек, то есть по одной ячейке на каждую точку подвыборки. При растеризации треугольника тесты покрытия и глубины выполняются N раз для N точек подвыборки в каждом фрагменте треугольника. Если хотя бы один из N тестов показывает, что фрагмент нужно отрисовать, один раз запускается пиксельный шейдер. Затем полученный от пиксельного шейдера цвет сохраняется *только* в тех ячейках, которые соответствуют точкам подвыборки,

По завершении отрисовки всей сцены выполняется понижающая дискретизация увеличенного буфера цвета для получения окончательного изображения с экранным разрешением. Этот процесс сводится к усреднению цветовых значений, размещенных в N ячейках точек подвыборки для каждого экранного пиксела. В итоге мы получаем сглаженное изображение при таком же уровне затрат на затенение, как и в случае изображения без сглаживания.

На рис. 11.46 показан результат растеризации треугольника без применения сглаживания. Рисунок 11.47 иллюстрирует результат четырехкратного сглаживания MSAA. Более подробное описание метода MSAA см. по адресу mynameismjp.wordpress.com/2012/10/24/msaa-overview.

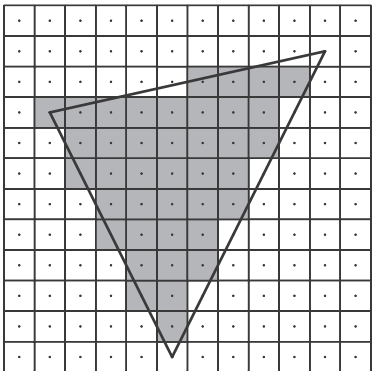


Рис. 11.46. Растеризация треугольника без сглаживания

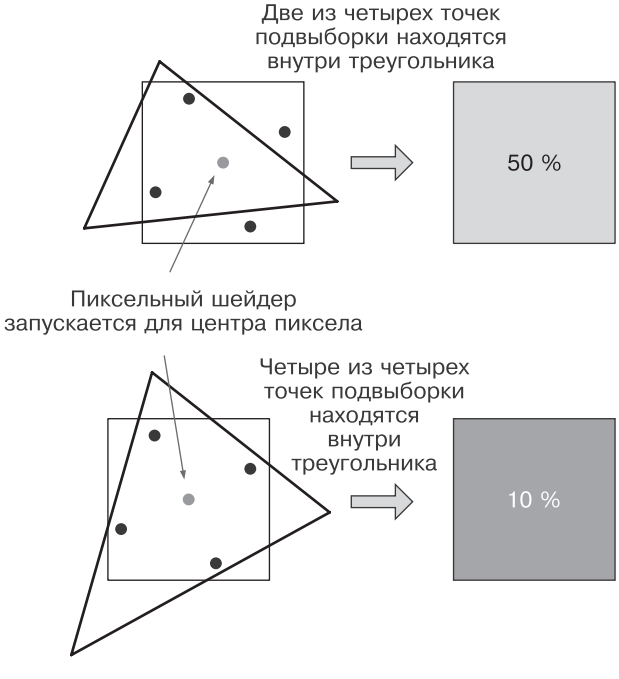


Рис. 11.47. Метод MSAA

Сглаживание с сэмплированием покрытия

Метод сглаживания с сэмплированием покрытия (Coverage Sample Antialiasing, CSAA) представляет собой оптимизированный вариант метода MSAA, впервые примененный компанией NVIDIA. При четырехкратном сглаживании CSAA пиксельный шейдер запускается один раз, тесты глубины и сохранения цвета выполняются для четырех точек подвыборки в каждом фрагменте, но тест покрытия пиксела выполняется для 16 точек подвыборки покрытия в каждом фрагменте. Это позволяет повысить степень детализации смешивания цветов по краям треугольников до уровня восьми- или шестнадцатикратного сглаживания MSAA при потреблении памяти и вычислительной мощности графического процессора на уровне четырехкратного сглаживания MSAA.

Морфологическое сглаживание

Метод морфологического сглаживания (Morphological Antialiasing, MLAA) применяется для исправления только тех областей сцены, в которых ступенчатость проявляется наиболее сильно. Сцена отрисовывается в ее обычном размере, а затем сканируется с целью выявления областей со ступенчатым узором. Если такие области найдены, они размываются для минимизации эффекта ступенчатости. Сходный с MLAA подход используется и в оптимизированном методе FXAA (Fast approximate antialiasing — быстрое приблизительное сглаживание) компании NVIDIA.

Более подробное описание метода MLAA см. по адресу intel.ly/2HhrQWX. Подробное описание метода FXAA можно найти по адресу bit.ly/1mIzCTv.

Субпиксельное морфологическое сглаживание

Субпиксельное морфологическое сглаживание (Subpixel Morphological Antialiasing, SMAA) обеспечивает более высокую точность на субпиксельном уровне за счет сочетания методов морфологического сглаживания (MLAA и FXAA) со стратегиями мультисэмплирования/суперсэмплирования (MSAA, SSAA). У SMAA низкий уровень затрат, сравнимый с уровнем затрат FXAA, но по сравнению с ним он меньше размывает окончательное изображение. Из-за этого данный метод заслуженно считают лучшим из существующих методов сглаживания. Подробное рассмотрение этой темы выходит за рамки данной книги, но вы найдете подробную информацию об SMAA по адресу www.iryoku.com/smaa/.

11.2.7. Этап приложения

Теперь, когда мы уже знаем, как работает графический процессор, рассмотрим этап конвейера, отвечающий за приведение его в действие, — этап приложения. На него возлагаются три задачи.

- *Определение видимости*. Чтобы не растрачивать ценные вычислительные ресурсы на обработку треугольников, которые никто никогда не увидит, графи-

ческому процессору нужно передавать только видимые (или по крайней мере *потенциально* видимые) объекты.

- *Передача геометрии графическому процессору для рендеринга.* Пары из подмеша и материала передаются графическому процессору путем вызова таких функций рендеринга, как `DrawIndexedPrimitive()` в DirectX или `glDrawArrays()` в OpenGL, или непосредственным выполнением последовательности команд графического процессора. Передаваемая геометрия может сортироваться для оптимизации производительности рендеринга. В случае, когда рендеринг сцены выполняется в несколько проходов, геометрия может передаваться несколько раз.
- *Управление параметрами шейдера и состоянием рендеринга.* На этапе приложения для каждого примитива настраиваются те унифицированные параметры, которые передаются шейдеру через константные регистры. Кроме того, на данном этапе требуется установить все настраиваемые параметры непрограммируемых этапов конвейера для надлежащей отрисовки каждого примитива.

В следующих разделах мы кратко рассмотрим, как эти задачи выполняются на этапе приложения.

Определение видимости

Меньше всего ресурсов тратится, когда мы вообще не отрисовываем треугольники. Поэтому крайне важно *отбраковать* объекты сцены, которые не вносят никакого вклада в окончательный результат рендеринга изображения, до передачи их графическому процессору. Процесс составления списка видимых экземпляров меша называется *определением видимости*.

Отбраковка по усеченной пирамиде. При отбраковке по усеченной пирамиде из списка рендеринга исключаются все объекты, которые целиком лежат за пределами усеченной пирамиды. Любой заданный экземпляр меша можно проверить на предмет того, расположен ли он внутри усеченной пирамиды, выполнив несколько простых тестов на основе данных об *ограничивающем объеме* объекта и шести плоскостях усеченной пирамиды. В качестве ограничивающего объема обычно используется сфера, потому что сферы очень легко поддаются отбраковке. Для этого необходимо сместить каждую плоскость усеченной пирамиды в наружном направлении на расстояние, равное радиусу сферы, а затем определить, с какой стороны каждой модифицированной плоскости находится центральная точка сферы. Если сфера находится с передней стороны по отношению к каждой из шести модифицированных плоскостей, значит, она находится внутри усеченной пирамиды.

На практике нам фактически даже не нужно перемещать плоскости усеченной пирамиды. Как можно вспомнить из уравнения (5.13), длину перпендикуляра h от точки до плоскости можно рассчитать, просто вставив точку непосредственно в уравнение плоскости следующим образом: $h = ax + by + cz + d = \mathbf{nP} - \mathbf{nP}_0$ (см. подраздел 5.6.3). Таким образом, нам нужно всего лишь вставить центральную точку

ограничивающей сферы в уравнение плоскости для каждой плоскости усеченной пирамиды, что даст значение h_i для каждой плоскости i, а затем сравнить эти значения с радиусом ограничивающей сферы, тем самым определив, с какой стороны плоскости она находится.

Описанная ранее в этом одразделе структура данных, называемая графом сцены, позволяет оптимизировать отбраковку по усеченной пирамиде за счет исключения из числа рассматриваемых тех объектов, ограничивающие сферы которых даже не стоит проверять на предмет расположения внутри усеченной пирамиды.

Окклюзия и потенциально видимые наборы. Даже когда объекты целиком расположены внутри усеченной пирамиды, они могут перекрывать друг друга. Удаление из списка видимых объектов, полностью перекрытых другими объектами, называется *отбраковкой окклюзии*. В насыщенных объектами сценах, наблюдаемых с уровня земной поверхности, объекты могут значительно перекрывать друг друга, и здесь отбраковка окклюзии играет крайне важную роль. В менее загроможденных объектами или наблюдаемых сверху сценах окклюзия может проявляться в гораздо меньшей степени, и затраты на ее отбраковку могут перевешивать преимущества от выполнения этого действия.

Отбраковать основную часть окклюзии крупномасштабной сцены можно предварительным вычислением *потенциально видимого набора* (Potentially Visible Set, PVS). PVS — это набор объектов сцены, которые могут быть видны в конкретно заданной точке обзора камеры. При вычислении набора PVS допускается ошибочное включение в него объектов, в действительности невидимых, но недопустимо исключать из него объекты, которые вносят некоторый вклад в отрисовываемую сцену.

Один из способов применения наборов PVS состоит в том, чтобы разделить уровень на определенные области. Каждая такая область снабжается списком других областей, видимых при нахождении в ней камеры.

Художники или дизайнеры игры могут вручную определять эти наборы PVS. Но более распространенная практика — генерирование наборов PVS на основе заданных пользователем областей с помощью автоматического автономного инструмента. Обычно такой инструмент выполняет рендеринг сцены, используя несколько случайным образом распределенных точек обзора в пределах области. При этом геометрия каждой области обозначается своим цветом, и вы можете увидеть список видимых областей, последовательно просматривая полученный кадровый буфер и переключаясь между цветами областей. Поскольку инструменты для автоматического генерирования наборов PVS не обеспечивают идеальных результатов, они, как правило, предоставляют пользователю некий механизм для настройки результатов путем либо размещения (вручную) предлагаемых точек обзора, либо составления (также вручную) списка областей, которые однозначно необходимо включить в набор PVS той или иной области или исключить из него.

Порталы. Еще один способ определения видимых частей сцены состоит в использовании *порталов*. При рендеринге с применением порталов игровой мир

разбивается на ряд квазизамкнутых областей, связанных друг с другом посредством проемов, например оконных или дверных. Эти проемы называются порталами и обычно представляются с помощью многоугольников, описывающих их границы.

При создании сцены с порталами сначала отрисовывается та область, в которой находится камера. Затем для каждого портала области строится соответствующая усеченная пирамида, образованная плоскостями, проходящими через фокальную точку камеры и каждое ребро ограничивающего многоугольника портала. Затем производится усечение содержимого соседних областей до объема усеченных пирамид порталов точно таким же образом, как выполняется усечение геометрии до усеченной пирамиды камеры. В результате этого в соседних областях отрисовывается только видимая геометрия. Данный метод иллюстрирует рис. 11.48.

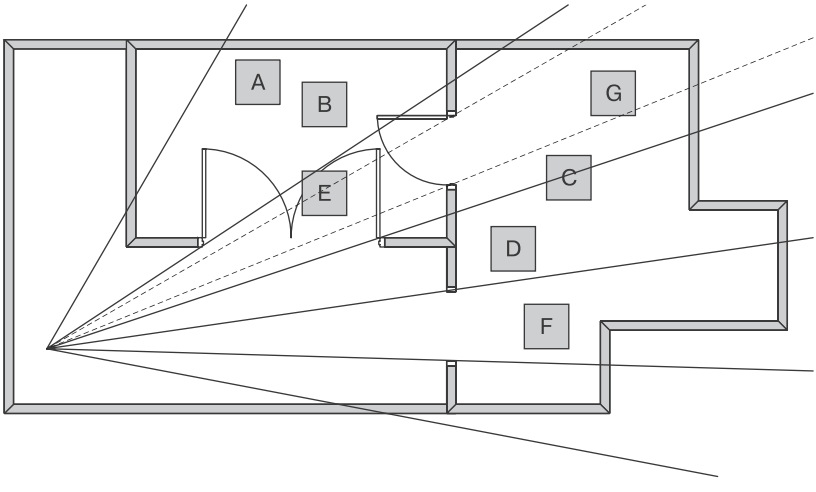


Рис. 11.48. Порталы используются для определения усеченных пирамид, после чего содержимое соседних областей усекается до объема этих усеченных пирамид. В данном примере будут отбракованы объекты A, B и D, потому что они лежат за пределами объема порталов, остальные объекты будут видны

Объемы окклюзии (антипорталы). Если перевернуть концепцию порталов с ног на голову, то мы сможем использовать пирамидальные объемы для определения тех областей сцены, которые *не могут* быть видны в силу того, что их перекрывает некий объект. Такие объемы называют *объемами окклюзии* или *антипорталами*. Чтобы построить объем окклюзии, необходимо найти края контуров всех перекрывающих объектов и протянуть плоскости наружу от фокальной точки камеры через каждый из этих краев. Затем можно проверить более удаленные объекты на предмет их нахождения внутри объемов окклюзии и отбраковать те из них, которые целиком расположены внутри этих объемов (рис. 11.49).

Порталы хорошо подходят для рендеринга закрытых помещений с небольшим количеством окон и дверных проемов между комнатами. В такой сцене порталы

занимают незначительную часть от общего объема усеченной пирамиды камеры, в результате чего мы можем отбраковать много объектов, находящихся за пределами порталов.

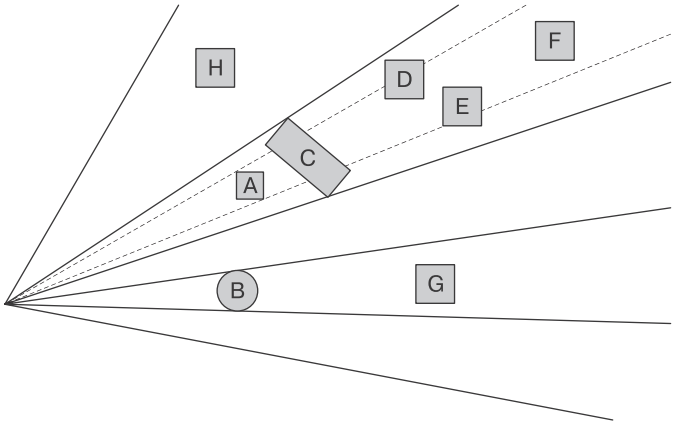


Рис. 11.49. Применение антипорталов, соответствующих объектам A, B и C, позволяет отбраковать объекты D, E, F и G. В результате этого остаются видимыми только объекты A, B, C и H

Антипорталы хорошо подходят для больших открытых пространств, где расположенные рядом объекты часто перекрывают значительную часть усеченной пирамиды камеры. В таком случае антипорталы охватывают довольно большую часть от общего объема усеченной пирамиды камеры, давая в результате много отбракованных объектов.

Передача примитивов

После создания списка видимых геометрических примитивов каждый отдельный примитив необходимо *передать* в конвейер графического процессора для отрисовки. Это может сделать вызовом функции `DrawIndexedPrimitive()` в DirectX или `glDrawArrays()` в OpenGL.

Состояние рендеринга. Как мы узнали из подраздела 11.2.4, многие этапы конвейера графического процессора обладают фиксированной, но в то же время настраиваемой функциональностью. Кроме того, программируемые этапы тоже в какой-то мере управляются с помощью настраиваемых параметров. Примеры настраиваемых параметров перечислены далее (это далеко не полный список):

- матрица преобразования мира в обзор;
- векторы направления света;
- привязки текстур (то есть сведения о том, какие текстуры следует использовать для того или иного материала/шейдера);
- режимы адресации и фильтрации текстур;

- масштаб времени для прокрутки текстур и других анимированных эффектов;
- *z*-тест (включение или отключение);
- параметры альфа-смешивания.

Набор всех настраиваемых параметров в рамках конвейера графического процессора называется *состоянием аппаратного обеспечения*, или *состоянием рендеринга*. На этапе приложения требуется правильно и полностью настроить состояние аппаратного обеспечения для каждого передаваемого примитива. В идеале настройки состояния должны полностью определяться материалом, ассоциированным с каждым подмешем. В таком случае задача этапа приложения сводится к итерации по списку видимых экземпляров меша и по каждой паре из подмеша и материала, установке состояния рендеринга на основе спецификаций материала и последующему вызову низкоуровневых функций передачи примитивов (`DrawIndexedPrimitive()`, `glDrawArrays()` или аналогичных).

Перетекание состояния. Если мы забудем установить какой-либо аспект состояния рендеринга при обходе передаваемых примитивов, то произойдет перетекание настроек, примененных к одному примитиву, на следующий примитив. *Перетекание состояния рендеринга* может проявиться, к примеру, в виде объекта с неправильно выбранными текстурой или световым эффектом. Очевидно, важно не допускать перетекания состояния на этапе приложения.

Список команд графического процессора. Коммуникация между этапом приложения и графическим процессором фактически осуществляется посредством списка команд. В этом списке настройки состояния рендеринга чередуются со ссылками на подлежащую отрисовке геометрию. Так, например, если требуется отрисовать объекты A и B, используя материал 1, а затем объекты C, D и E, применив материал 2, то список команд может выглядеть следующим образом.

- Установить состояние рендеринга для материала 1 (несколько команд, по одной на каждый параметр состояния рендеринга).
- Передать примитив A.
- Передать примитив B.
- Установить состояние рендеринга для материала 2 (несколько команд).
- Передать примитив C.
- Передать примитив D.
- Передать примитив E.

Если мы рассмотрим внутреннее устройство таких API-функций, как `DrawIndexedPrimitive()`, то увидим, что в действительности они просто составляют и передают списки команд графического процессора. Для некоторых приложений могут быть непозволительными даже затраты на эти API-вызовы. Для обеспечения максимальной производительности в некоторых игровых движках списки команд графического процессора создают вручную или вызовом низкоуровневых API рендеринга, таких как Vulkan (www.khronos.org/vulkan/).

Сортировка геометрии

Настройки состояния рендеринга являются глобальными, то есть они относятся к графическому процессору в целом. Таким образом, чтобы изменить настройки состояния рендеринга, необходимо очистить весь конвейер графического процессора перед применением новых настроек. Если не проявить здесь должной осмотрительности, это может привести к значительному снижению производительности.

Очевидно, изменять настройки рендеринга стоит как можно реже. Лучший способ заключается в том, чтобы отсортировать геометрию по материалу. Так мы сможем, задав настройки материала A, отрисовать всю ассоциированную с ним геометрию, после чего перейти к материалу B и т. д.

К сожалению, сортировка геометрии по материалу может отрицательно сказаться на производительности рендеринга, так как из-за этого часто происходит *перерисовка* — один и тот же пиксел заполняется несколькими перекрывающими друг друга треугольниками. Конечно, иногда перерисовка необходима и желательна, поскольку это единственный способ выполнить нужное альфа-смешивание присутствующих в сцене прозрачных и полупрозрачных поверхностей. Однако перерисовка *непрозрачных* пикселов — это всегда напрасная трата пропускной способности графического процессора.

Обеспечить отбрасывание перекрытых фрагментов до того, как до них доберется затратный пиксельный шейдер, призван ранний *z*-тест. Но, чтобы получить от него максимальный эффект, нужно отрисовывать треугольники в порядке возрастания удаленности. При этом в *z*-буфер сразу же заносятся самые близкие треугольники, и все фрагменты, получаемые из расположенных за ними более удаленных треугольников, можно сразу отбросить с минимальной перерисовкой или вообще без нее.

Z-предпроход спешит на помощь. Каким же образом можно разрешить конфликт между необходимостью сортировки геометрии по материалу и отрисовки непрозрачной геометрии в порядке возрастания удаленности? Ответом на этот вопрос является функция графического процессора, называемая *z-предпроходом*.

Z-предпроход подразумевает двукратную отрисовку сцены: первый раз для максимально эффективного генерирования содержимого *z*-буфера, второй — для заполнения кадрового буфера полной информацией о цвете (на этот раз благодаря использованию содержимого *z*-буфера — без перерисовки). Графический процессор обеспечивает специальный режим двойной скорости рендеринга, при котором только обновляется *z*-буфер при отключенных пиксельных шейдерах. Чтобы свести к минимуму затраты времени на генерирование содержимого *z*-буфера, в ходе этой фазы можно отрисовать непрозрачную геометрию в порядке возрастания удаленности. После этого пересортировать геометрию по материалу и отрисовать в полном цвете с минимальными изменениями состояния, чтобы обеспечить максимальную пропускную способность конвейера.

После отрисовки непрозрачной геометрии можно отрисовать прозрачные поверхности в порядке уменьшения удаленности. Такой несколько прямолинейный подход позволяет получить нужный результат альфа-смешивания. Отрисовывать

прозрачную геометрию в произвольном порядке позволяет метод *порядконезависимой прозрачности* (Order-Independent Transparency, OIT). Этот метод состоит в том, чтобы сохранять по нескольку фрагментов на каждый пиксел и выполнять сортировку и смешивание фрагментов каждого пиксела уже после отрисовки всей сцены. Это позволяет получать корректные результаты без предварительной сортировки геометрии, за что, однако, приходится платить высоким потреблением памяти, поскольку кадровый буфер должен быть достаточно большим для того, чтобы в нем поместились все полупрозрачные фрагменты для каждого пиксела.

Графы сцены

Миры современных игр могут быть невероятно большими. И поскольку в большинстве сцен основная часть геометрии находится за пределами усеченной пирамиды камеры, отбраковка по усеченной пирамиде всех этих объектов в явной форме, как правило, требует больших затрат ресурсов. Будет лучше, если вместо этого мы воспользуемся какой-то структурой данных, которая будет обеспечивать управление всей геометрией сцены и позволит быстро отбрасывать значительные части мира, которые явно не могут находиться внутри усеченной пирамиды камеры, до выполнения детальной отбраковки по усеченной пирамиде. В идеале эта структура данных должна также упрощать сортировку геометрии сцены либо в порядке возрастания удаленности для z-предпрохода, либо по материалу для полноцветного рендеринга.

Такую структуру данных обычно называют *графом сцены*, что является отсылкой к графоподобным структурам данных, которые часто применяют в движках рендеринга для киноиндустрии и таких инструментах для создания цифрового контента, как Мауа. При этом игровой граф сцены не обязательно должен быть графом, и на практике в качестве этой структуры данных обычно используется определенного вида дерево (которое, конечно, является частным случаем графа). Главная идея, лежащая в основе большинства этих структур данных, заключается в разделении трехмерного пространства таким образом, чтобы было легко отбрасывать сразу целые области, которые не пересекают усеченную пирамиду, не отбраковывая по отдельности каждый из объектов, находящихся внутри этих областей. Примерами таких структур являются деревья квадрантов и октантов, BSP-деревья, k-мерные деревья и методы пространственного хеширования.

Деревья квадрантов и октантов. *Дерево квадрантов* рекурсивным образом разбивает пространство на квадранты. Каждый уровень рекурсии представляется в нем в виде узла с четырьмя дочерними элементами, по одному на каждый квадрант. Квадранты обычно разделяются вертикально ориентированными выровненными по осям плоскостями, что придает им квадратную или прямоугольную форму. В то же время дерево квадрантов может разбивать пространство и на области произвольной формы.

Деревья квадрантов могут использоваться для хранения и организации практически любых пространственно распределенных данных. В контексте движков рендеринга они часто применяются для хранения отрисовываемых примитивов,

например экземпляров меша, подобластей геометрии ландшафта или отдельных треугольников большого статического меша, с целью оптимизации отбраковки по усеченной пирамиде. Отрисовываемые примитивы хранятся в листьях дерева, и обычно нужно стремиться к тому, чтобы в области каждого листа содержалось примерно одинаковое количество примитивов. Этого можно добиться, принимая решение о продолжении или прекращении дальнейшего разбиения, исходя из количества примитивов в области.

Чтобы определить, какие примитивы будут видимы, попадая в усеченную пирамиду камеры, необходимо обойти дерево от корня до листьев, проверяя каждую область на предмет пересечения с усеченной пирамидой. Если какой-то квадрант не пересекается с усеченной пирамидой, то с ней не пересекаются и все его дочерние области и можно прекратить обход этой ветви дерева. Это позволяет выявлять потенциально видимые примитивы намного быстрее, чем линейным поиском (обычно за время, равное $O(\log n)$). Пример разбиения пространства с помощью дерева квадрантов показан на рис. 11.50.

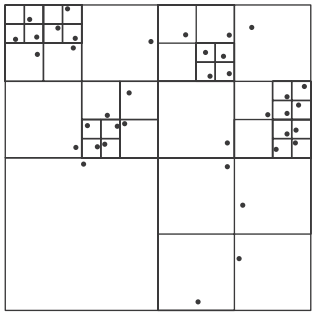


Рис. 11.50. Вид сверху на пространство, рекурсивным образом разбитое на квадранты для хранения в дереве квадрантов, на основе критерия присутствия одной точки в каждой области

Дерево октантов — это трехмерная разновидность дерева квадрантов, которая разбивает пространство на восемь подобластей на каждом уровне рекурсивного разбиения. Хотя обычно дерево октантов делит пространство на кубы или прямоугольные призмы, в общем случае это могут быть трехмерные области произвольной формы.

Деревья ограничивающих сфер. Аналогично тому, как деревья квадрантов или октантов разбивают пространство на прямоугольные области (в большинстве случаев), *дерево ограничивающих сфер* выполняет иерархическое подразбиение пространства на сферические области. При этом листья дерева содержат ограничивающие сферы отрисовываемых примитивов сцены. Эти примитивы объединяются в небольшие логические группы, и вычисляется общая ограничивающая сфера каждой группы.

Затем группы объединяются в группы большего размера, и этот процесс продолжается до тех пор, пока не будет получена единая группа с ограничивающей сферой, охватывающей весь виртуальный мир. Чтобы составить список потенциально видимых примитивов, необходимо обойти дерево от корня до листьев, проверяя каждую ограничивающую сферу на предмет пересечения с усеченной пирамидой, и продолжать рекурсивный обход только тех ветвей, которые ее пересекают.

BSP-деревья. Дерево бинарного разбиения пространства, или BSP-дерево, рекурсивным образом разбивает пространство на две половины до тех пор, пока объекты в пределах каждого полупространства не удовлетворят некоторым заранее определенным критериям (во многом так же, как дерево квадрантов разбивает

пространство на квадранты). BSP-деревья имеют множество применений, включая обнаружение столкновений, конструктивную блочную геометрию и, конечно, наиболее известное в трехмерной графике использование в качестве средства повышения производительности при отбраковке по усеченной пирамиде и сортировке геометрии. *K*-мерное дерево — это обобщение концепции BSP-дерева, применяемое к *k*-мерному пространству.

В контексте рендеринга BSP-дерево разбивает пространство одной плоскостью на каждом уровне рекурсии. Разбивающие плоскости могут выравниваться по осям, но чаще каждое разбиение осуществляется по плоскости некоторого треугольника сцены. После этого все остальные треугольники классифицируются как расположенные перед этой плоскостью или за ней. Любые треугольники, которые пересекают разбивающую плоскость, разделяются на три новых треугольника таким образом, чтобы каждый из них располагался либо полностью перед плоскостью, либо полностью за ней, либо копланарно с ней. В итоге мы получаем бинарное дерево с разбивающей плоскостью и одним или несколькими треугольниками в каждом внутреннем узле и треугольниками в листьях.

BSP-деревья можно использовать для отбраковки по усеченной пирамиде примерно таким же образом, как и деревья квадрантов, октантов и ограничивающих сфер. Но, когда BSP-дерево генерируется разбиением на отдельные треугольники, как описано ранее, его можно задействовать также для строгой сортировки треугольников в порядке возрастания или уменьшения удаленности. Это играло особенно важную роль для первых трехмерных игр, таких как *Doom*, в которых нельзя было воспользоваться преимуществами *z*-буфера и поэтому приходилось применять алгоритм художника (то есть отрисовывать сцену в порядке уменьшения удаленности), чтобы обеспечить надлежащую реализацию взаимной окклюзии треугольников.

При заданной точке обзора камеры в трехмерном пространстве алгоритм сортировки в порядке уменьшения удаленности выполняет обход дерева, начиная с его корня. В каждом узле проверяется, находится точка обзора перед разбивающей плоскостью этого узла или за ней. Если камера находится перед плоскостью узла, то сначала выполняется обход его задних дочерних элементов, затем отрисовка треугольников, расположенных в разбивающей его плоскости, и, наконец, обход передних дочерних элементов. Если же точка обзора камеры оказывается за разбивающей плоскостью узла, то сначала выполняется обход его передних дочерних элементов, затем отрисовка треугольников, расположенных в этой плоскости, и, наконец, обход задних дочерних элементов. Такая схема гарантирует, что обход наиболее удаленных от камеры треугольников будет выполнен раньше обхода более близких, что обеспечит сортировку в порядке уменьшения удаленности. Поскольку данный алгоритм обходит *все* треугольники сцены, порядок обхода не зависит от направления обзора камеры. Чтобы обойти только видимые треугольники, потребуется выполнить дополнительный этап отбраковки по усеченной пирамиде. На рис. 11.51 показано простое BSP-дерево, а также схема обхода для указанного положения камеры.

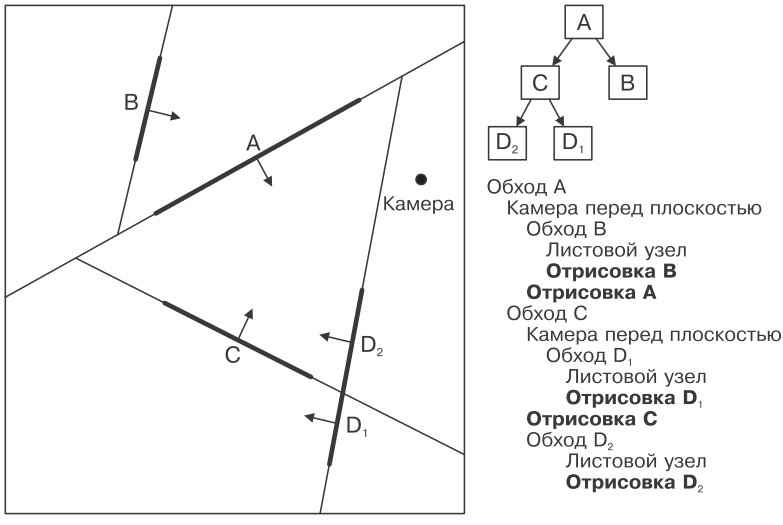


Рис. 11.51. Пример обхода треугольников BSP-дерева в порядке уменьшения удаленности. Для простоты треугольники повернуты ребром к представленной здесь двухмерной плоскости, однако в реальности ориентация в пространстве треугольников и разбивающих плоскостей может быть произвольной

Всестороннее рассмотрение алгоритмов генерирования и использования BSP-деревьев выходит за рамки данной книги. Более подробная информация о них есть по адресу www.gamedev.net/reference/articles/article657.asp.

Выбор графа сцены

Существует много разновидностей графа сцены, и выбор конкретной структуры данных для игры зависит от того, сцены какого характера планируется в ней отрисовывать. Чтобы выбрать правильно, нужно четко понимать, что потребуется и (это еще важнее!) *не* потребуется при рендеринге сцен конкретной игры.

Например, если вы создаете игру в жанре файтинга, где два персонажа ведут поединок на ринге с преимущественно статичным окружением, то можно обойтись вообще без графа сцены. Если игра протекает в основном в закрытых помещениях, стоит использовать BSP-дерево или систему порталов. Если действие происходит вне помещений в местности со сравнительно ровным ландшафтом и в основном применяется вид сцены сверху (как, например, в играх в жанре стратегии или симулятора бога), то высокую скорость рендеринга может обеспечить простое дерево квадрантов. Но если для происходящей на открытом воздухе сцены задействуется точка обзора одного из расположенных на земле персонажей, то, вероятно, потребуется использовать дополнительные механизмы отбраковки. Насыщенные объектами сцены могут сильно выиграть от применения системы объемов окклюзии (антипорталов), потому что в них обычно много перекрывающих вид объектов. В то же время, если речь идет о сцене на открытом воздухе с малым числом объектов, добавление системы антипорталов вряд ли принесет какие-либо преимущества и даже может плохо сказаться на частоте кадров.

В конечном счете выбирать граф сцены следует, исходя из точных данных, полученных путем фактического измерения производительности вашего движка рендеринга. Вы можете быть сильно удивлены, когда узнаете, на что на самом деле затрачиваются ресурсы! Но узнав это, вы сможете выбрать структуры данных графа сцены и/или другие средства оптимизации для решения конкретных проблем.

11.3. Продвинутые методы освещения и глобальное освещение

Для рендеринга фотореалистичных сцен нужны физически точные алгоритмы глобального освещения. Всестороннее их рассмотрение выходит за рамки книги, поэтому в следующих разделах мы лишь кратко коснемся методов, наиболее широко распространенных в современной игровой индустрии. Это должно дать вам общее представление о данных методах и послужить отправной точкой для дальнейшего изучения. Прекрасное всестороннее рассмотрение этой темы есть в книге [10].

11.3.1. Освещение на основе изображения

В ряде продвинутых методов освещения и затенения активно используются данные изображения, как правило, в виде двухмерных текстурных карт. Это так называемые алгоритмы *освещения на основе изображения*.

Наложение карт нормалей

Карта нормалей определяет вектор направления нормали поверхности в каждом текселе. Это позволяет 3D-моделисту снабдить движок рендеринга очень подробным описанием формы поверхности, не нуждаясь в высоком уровне тесселяции модели, который был бы нужен в случае предоставления той же информации с помощью нормалей вершин. При использовании карты нормалей один плоский треугольник может выглядеть так, как будто он состоит из миллионов крошечных треугольников. Пример наложения карты нормалей показан на рис. 11.52.

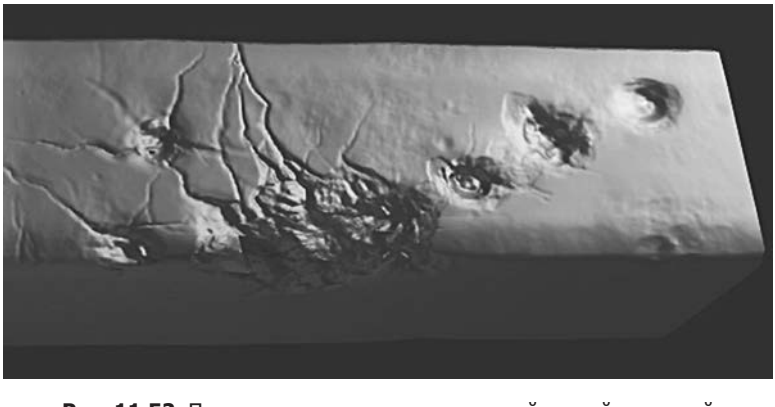


Рис. 11.52. Пример поверхности с наложенной картой нормалей

Векторы нормалей обычно выражаются в значениях цветовых RGB-каналов текстуры с подходящим смещением для компенсации того обстоятельства, что значения RGB-каналов всегда положительные, в то время как составляющие вектора нормалей могут быть отрицательными. Иногда в текстуре сохраняются только две координаты, поскольку третью легко вычислить в динамическом режиме, исходя из допущения, что нормали поверхности являются единичными векторами.

Карты высот: рельефное, параллактическое и смещающее наложение

Как следует из названия, *карта высот* определяет высоту идеализированной поверхности, расположенной выше или ниже поверхности треугольника. Обычно карты высот выглядят как изображение в оттенках серого цвета, поскольку требуется только одно значение высоты на каждый тексел. Карты высот могут использоваться для рельефного, параллактического и смещающего наложения — трех методов, позволяющих придать плоской поверхности эффект варьирования по высоте.

При рельефном наложении карта высот служит низкозатратным средством для генерирования нормалей поверхности. Этот метод широко использовался на заре развития трехмерной графики и почти не применяется сегодня — в большинстве современных игровых движков информация о нормалях поверхности сохраняется явным образом в виде *карты нормалей*, а не вычисляется на основе карты высот.

При наложении параллактической окклюзии на основе информации карты высот выполняется искусственная корректировка текстурных координат, применяемых при рендеринге плоской поверхности, для имитации наличия на поверхности элементов, которые корректно перемещаются при перемещении камеры. (Этот метод использовался для создания декалей, имитирующих эффект попадания пули, в серии игр *Uncharted* студии Naughty Dog.)

При смещающем наложении создаются реальные элементы поверхности путем тесселяции с последующим выдавливанием поверхностных многоугольников, опять же с использованием карты высот для определения величины смещения каждой вершины. Это дает в итоге самый убедительный визуальный эффект, который к тому же сам по себе обеспечивает надлежащую реализацию окклюзии и затенения, благодаря тому что генерируется реальная геометрия. Рисунок 11.53 позволяет сравнить результаты рельефного, параллактического и смещающего наложений. На рис. 11.54 показан пример реализации смещающего наложения в DirectX 9.

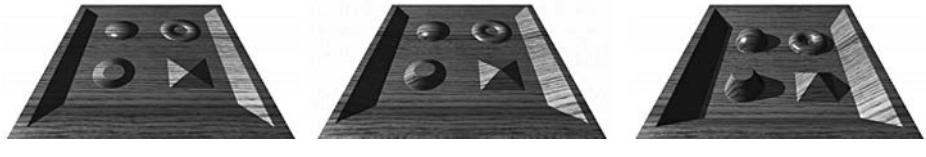


Рис. 11.53. Сравнение результатов рельефного наложения *(слева)*, наложения параллактической окклюзии *(по центру)* и смещающего наложения *(справа)*

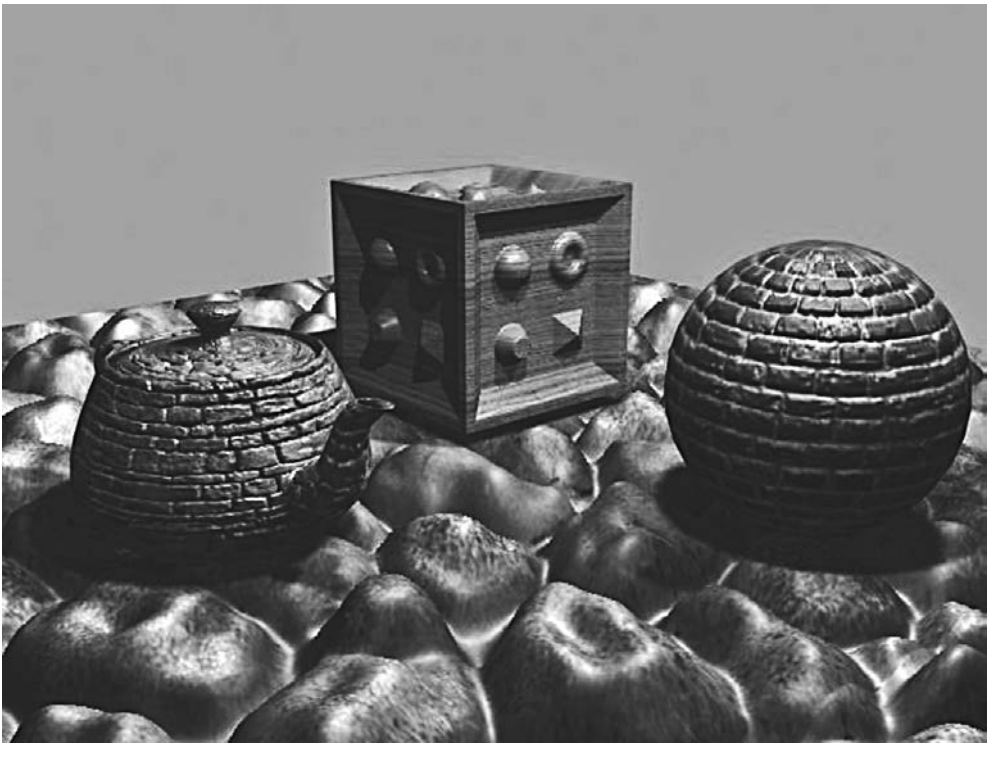


Рис. 11.54. Пример реализации смещающего наложения в DirectX 9. Простая исходная геометрия тесселируется в динамическом режиме для получения элементов поверхности

Карты зеркального отражения/бликов

Прямолинейное отражение света от блестящей поверхности называется *зеркальным отражением*. Интенсивность зеркального отражения зависит от величины углов между направлением обзора, направлением света и нормалью поверхности. Как мы видели в подразделе 11.1.3, интенсивность зеркальной составляющей света можно выразить как $k_S(\mathbf{RV})_\alpha$, где $\mathbf{R}$ — отражение вектора направления света относительно нормали поверхности, $\mathbf{V}$ — направление обзора, k_S — общий коэффициент зеркального отражения поверхности, α — степень зеркального отражения.

У многих поверхностей отражательная способность неравномерная. Например, когда у человека лицо вспотело или испачкано, его влажные участки выглядят блестящими, а сухие или испачканные — матовыми. Мы можем определить высокодетализированную информацию об отражательной способности в виде специальной текстурной карты, которую называют *картой зеркального отражения*.

Сохранив значения коэффициента k_S в текселах карты зеркального отражения, мы сможем контролировать степень зеркального отражения, применяемого в каждом тексселе. Такие карты зеркального отражения иногда называют *картами бликов*.

Или *масками зеркального отражения*, поскольку, присваивая текселам нулевые значения, мы можем маскировать те части поверхности, для которых не требуется зеркальное отражение. Сохранив в карте зеркального отражения значения степени α, мы сможем контролировать степень фокусировки зеркальных бликов в каждом тексселе. Такие текстуры называют *картами степени зеркального отражения*. Пример применения карты бликов представлен на рис. 11.55.

Рис. 11.55. Этот снимок экрана игры Fight Night Round 3 компании EA показывает, как можно использовать карту бликов для контроля степени зеркального отражения, применяемой в каждом тексселе поверхности

Наложение карт окружения

Карта окружения выглядит как панорамная фотография окружающей среды, снятая с точки обзора объекта сцены, и охватывает все 360° по горизонтали и либо 180, либо 360° по вертикали. Такая карта описывает общие условия освещения вокруг объекта и обычно используется для низкозатратной отрисовки отражений.

Наиболее широкое распространение получили два формата таких карт — *сферические* и *кубические карты окружения*. Сферическая карта окружения выглядит как фотография, снятая с помощью объектива «рыбий глаз», и обрабатывается таким образом, как если бы она была наложена на внутреннюю поверхность сферы с бесконечным радиусом и центром в отрисовываемом объекте. Проблемой сферических карт является то, что для обращения к ним используются сферические координаты. Это обеспечивает хорошее разрешение как по горизонтали, так и по вертикали вдоль экваториальной линии сферы. Однако по мере уменьшения вертикального

угла с приближением направления обзора к вертикали разрешение текстуры вдоль горизонтальной оси уменьшается до одного тексела. В качестве решения этой проблемы были предложены кубические карты.

Кубическая карта выглядит как фотография, составленная из снимков, сделанных с шести основных направлений обзора — сверху, снизу, слева, справа, спереди и сзади. При рендеринге кубическая карта обрабатывается таким образом, как если бы она была наложена на шесть бесконечно удаленных внутренних поверхностей коробки с центром в отрисовываемом объекте.

Чтобы считать тексел карты окружения, соответствующий точке **P** на поверхности объекта, необходимо взять луч, идущий из камеры в точку **P**, и отразить его относительно нормали поверхности в точке **P**. Затем отследить отраженный луч до точки его пересечения со сферой или кубом карты окружения. Значение тексела в этой точке используется при затенении точки **P**.

Трехмерные текстуры

Современное графическое аппаратное обеспечение поддерживает также трехмерные текстуры. Трехмерную текстуру можно рассматривать как своего рода стопку двухмерных текстур. Графический процессор знает, как обращаться к трехмерной текстуре и выполнять ее фильтрацию для заданных трехмерных текстурных координат (u, v, w).

Трехмерные текстуры могут быть полезны для описания внешнего вида или объемных свойств объекта. Например, мы можем отрисовать мраморную сферу и разрезать ее произвольной плоскостью. При этом получим непрерывный и корректный вид текстуры на всем разрезе вне зависимости от того, где именно он будет сделан, поскольку текстура будет четко определена и непрерывна в пределах всего объема сферы.

11.3.2. Освещение с расширенным динамическим диапазоном

Такое устройство отображения, как телевизор или ЭЛТ-монитор, может воспроизводить ограниченный диапазон интенсивности света, или, иначе говоря, динамический диапазон. Именно поэтому значения цветовых каналов в кадровом буфере ограничены диапазоном от нуля до единицы. Однако в реальном мире интенсивность света может быть сколь угодно большой. Попыткой охватить весь этот широкий диапазон интенсивности света является *освещение с расширенным динамическим диапазоном*, или *HDR-освещение* (High Dynamic Range).

HDR-освещение подразумевает расчет освещения без принудительного ограничения значений интенсивности. Получаемое изображение сохраняется в формате, допускающем запись значений интенсивности, превышающих единицу. Это позволяет передать без потери детализации и чрезвычайно темные, и чрезвычайно светлые участки изображения.

Перед выводом на экран посредством так называемого *тонального отображения* динамический диапазон изображения смещается и масштабируется в соответствии с динамическим диапазоном устройства отображения. Такой подход позволяет движку рендеринга воспроизводить множество реальных визуальных эффектов, таких как временная слепота, возникающая при выходе из темной комнаты в ярко освещенное пространство, или свечение вокруг объекта с яркой задней подсветкой — ореол.

Один из способов представления HDR-изображений состоит в том, чтобы обозначать каналы R, G и B 32-битными числами с плавающей запятой, а не 8-битными целыми числами. Еще один вариант — использовать совершенно другую цветовую модель. Так, для показа HDR-освещения часто применяют цветовую модель log-LUV. В ней цвет представляется с помощью канала интенсивности *(L)* и двух каналов цветности (*U* и *V*). Поскольку человеческий глаз более чувствителен к изменениям интенсивности, чем к изменениям цветности, канал *L* сохраняется в 16-битном формате, а каналы *U* и *V* — в 8-битном. Кроме того, для расширения охватываемого динамического диапазона для канала *L* используется логарифмическая шкала (с основанием 2).

11.3.3. Глобальное освещение

Как упоминалось в подразделе 11.1.3, под глобальным освещением понимается класс алгоритмов освещения, в которых учитываются множественные взаимодействия света с объектами сцены по мере его движения от источника к виртуальной камере. Глобальное освещение учитывает, например, такие эффекты, как тени, возникающие, когда одна поверхность заслоняет другую, отражения, каустики и своеобразное перетекание цвета одного объекта на расположенные вокруг объекты. В следующих разделах мы кратко рассмотрим ряд наиболее распространенных методов глобального освещения. С помощью некоторых из них воспроизводится один отдельно взятый эффект, такой как тени или отражения. С помощью других, например метода трассировки лучей или метода излучательности, создается целостная модель глобального переноса света.

Рендеринг теней

Тени возникают, когда какая-то поверхность препятствует распространению света. В идеализированном случае тени, отбрасываемые точечным источником света, должны иметь четко очерченный край, однако в реальном мире края теней обычно немного размыты — это называется полутенью. Полутень возникает по той причине, что реальные источники света занимают некоторую площадь и, соответственно, испускаемые ими световые лучи падают на края объектов под разными углами.

Двумя наиболее распространенными методами рендеринга теней являются методы *теневых объемов* и *теневых карт*. Мы кратко рассмотрим оба в последующих разделах. И первый, и второй метод в общем случае предусматривает разделение

объектов сцены на три категории: объекты, которые отбрасывают тени, объекты, на которые накладываются тени, и объекты, которые следует полностью исключить из рассмотрения при рендеринге теней. Аналогично определяется, какие источники света должны отбрасывать тени, а какие — нет. Эта важная оптимизация снижает количество комбинаций из источника света и объекта, подлежащих обработке при отрисовке теней в сцене.

Теневые объемы. При использовании метода теневых объемов каждый отбрасывающий тень объект рассматривается из точки обзора создающего тени источника света и выявляются края контура этого объекта. Края выдавливаются в направлении лучей, испускаемых источником света. Это дает в результате новый геометрический объект, определяющий тот объем пространства, в котором свет заслоняется рассматриваемым объектом. Пример применения этого метода показан на рис. 11.56.

Рис. 11.56. Теневой объем, созданный выдавливанием краев контура отбрасывающего тень объекта, наблюдаемого из точки обзора источника света

Теневой объем используется для генерирования теней с помощью специального полноэкранного буфера, называемого буфером трафарета. Он содержит по одному целочисленному значению на каждый экранный пиксел. Мы можем маскировать

рендеринг значениями буфера трафарета — например, можно настроить графический процессор таким образом, чтобы отрисовывались только те фрагменты, трафаретные значения которых не равны нулю. Мы также можем настроить графический процессор так, чтобы отрисовываемая геометрия обновляла значения в буфере трафарета различными полезными способами.

Для рендеринга теней сначала отрисовывается сцена с получением незатененного изображения в кадровом буфере, а также точного *z*-буфера. Буфер трафарета очищается таким образом, чтобы он содержал нули во всех пикселах. Затем каждый теневой объем отрисовывается с использованием точки обзора камеры так, чтобы треугольники, обращенные вперед, увеличивали значение в буфере трафарета на единицу, а треугольники, обращенные назад, уменьшали это значение на единицу. Разумеется, в тех областях экрана, где теневой объем вообще не отображается, значения буфера трафарета останутся равными нулю. Буфер трафарета будет содержать нули и там, где видны как передняя, так и задняя грани теневого объема, потому что в таком случае передняя грань увеличивает трафаретное значение, а задняя его уменьшает. В тех областях, где задняя грань теневого объема заслоняется реальной геометрией сцены, трафаретное значение будет равно единице. После этого мы уже будем знать, какие пикселы экрана находятся в тени. Соответственно, можно будет отрисовать тени в третьем проходе, просто затеняя те области экрана, где значение буфера трафарета не равно нулю.

Теневые карты. Метод наложения теневых карт, по сути, представляет собой пофрагментное выполнение теста глубины с использованием точки обзора источника света, а не точки обзора камеры. Сцена отрисовывается в два этапа. Сначала генерируется текстура *теневой карты* путем отрисовки сцены с использованием точки обзора источника света и сохранения содержимого буфера глубины. Затем выполняется обычная отрисовка сцены с применением теневой карты для проверки каждого фрагмента на предмет его расположения в тени. В каждом фрагменте сцены теневая карта указывает, заслонен или нет источник света какой-либо геометрией, расположенной ближе к источнику света, аналогично тому, как *z*-буфер указывает, перекрыт или нет фрагмент каким-либо треугольником, расположенным ближе к камере.

Теневые карты содержат только информацию о глубине — сведения о том, насколько удален от источника света каждый тексел. В силу этого для рендеринга теневых карт обычно используется режим двойной скорости аппаратного обеспечения с доступом только к *z*-буферу, поскольку нас интересует лишь информация о глубине. Если источник света точечный, теневая карта отрисовывается с применением перспективной проекции, если он направленный — с помощью ортогональной проекции.

Чтобы отрисовать сцену, применяя теневую карту, сначала нужно отрисовать ее как обычно, с использованием точки обзора камеры. Затем для каждой вершины каждого треугольника рассчитывается положение в *световом пространстве*, то есть в том же пространстве обзора, в котором была сгенерирована теневая

карта. Координаты светового пространства можно интерполировать в пределах треугольника точно так же, как и любые другие атрибуты вершин. Это позволяет определить положение каждого фрагмента в световом пространстве. Чтобы выяснить, находится заданный фрагмент в тени или нет, необходимо преобразовать координаты (x, y) фрагмента в световом пространстве в текстурные координаты (u, v) в пределах теневой карты. После этого сравнить координату z фрагмента в световом пространстве со значением глубины, сохраненным в соответствующем текселе теневой карты. Если координата z фрагмента в световом пространстве показывает, что он расположен дальше от источника света, чем тексел теневой карты, то он, видимо, заслонен каким-то другим геометрическим объектом, находящимся ближе к источнику света, то есть затенен. Аналогично, если координата z фрагмента в световом пространстве показывает, что он расположен ближе к источнику света, чем тексел теневой карты, то он ничем не заслонен и не находится в тени. Эту информацию можно использовать для того, чтобы соответствующим образом скорректировать цвет фрагмента. Применение теневых карт иллюстрирует рис. 11.57.

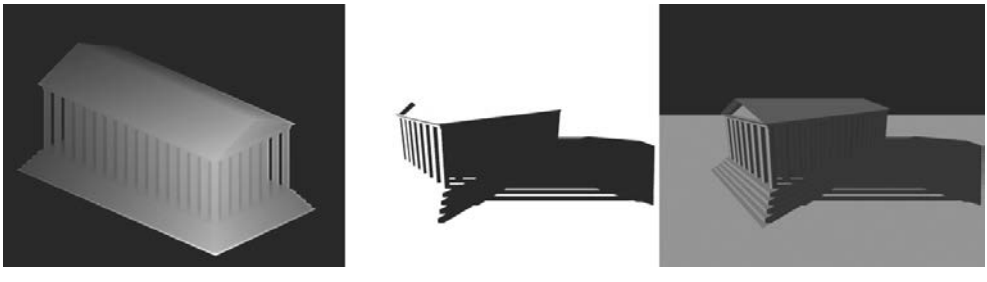


Рис. 11.57. Изображение слева представляет собой теневую карту — содержимое z-буфера, отрисовываемое с помощью точки обзора конкретного источника света. Пикселы центрального изображения черные там, где результат теста глубины светового пространства отрицательный (фрагмент в тени), и белые там, где он положительный (фрагмент не в тени). Справа представлен результат рендеринга сцены с тенями

Окклюзия окружения

Метод *окклюзии окружения* предназначен для моделирования *контактных теней* — мягких теней, возникающих в том случае, когда сцена освещается только фоновым источником света. Окклюзия окружения фактически определяет общую степень доступности для света каждой точки поверхности. Так, например, внутренняя поверхность трубы менее доступна для фонового света, чем внешняя. В случае размещения трубы вне помещения в пасмурный день ее внутренняя поверхность будет выглядеть темнее, чем внешняя.

На рис. 11.58 показано, как с помощью окклюзии окружения можно создать тени под автомобилем, в его колесных нишах и в швах между панелями кузова. Для того чтобы рассчитать уровень окклюзии окружения в некоторой точке

поверхности, необходимо построить полусферу очень большого радиуса с центром в этой точке и определить, какая часть площади полусферы видна из рассматриваемой точки.

Рис. 11.58. Автомобиль, отрисованный с использованием окклюзии окружения. Обратите внимание на более темные области под автомобилем и в колесных нишах

Для статических объектов эти данные можно рассчитать заранее в автономном режиме, потому что окклюзия окружения не зависит от направления обзора и направления падающего света. Обычно их сохраняют в виде текстурной карты, содержащей значения уровня окклюзии окружения для каждого тексела поверхности.

Отражения

Рис. 11.59. Зеркальные отражения в игре The Last of Us: Remastered (снимок экрана с сайта https://www.polygon.com), реализованные отрисовкой сцены в текстуру и последующим наложением этой текстуры на поверхность зеркала

Отражения возникают, когда свет отражается от поверхности с высокой отражающей способностью, создавая на поверхности изображение другой части сцены. Существует несколько способов реализации отражений. Для создания общих отражений окружающей среды на поверхностях блестящих объектов используют карты окружения. Для формирования прямолинейных отражений на плоских поверхностях, таких как зеркала, следует отразить положение камеры относительно плоскости отражающей поверхности, а затем отрисовать сцену в текстуру с использованием этой отраженной точки обзора. После этого нужно во втором проходе наложить полученную текстуру на отражающую поверхность (рис. 11.59).

Каустики

Каустики — это яркие зеркальные блики, возникающие при интенсивном отражении или преломлении от поверхности с высокой отражательной способностью, например водной или полированной металлической поверхности. Если отражающая поверхность находится в движении (как водная поверхность), каустические эффекты мерцают и «плавают» по той поверхности, где они проявляются. Каустические эффекты можно создать, проецируя текстуру с квазислучайными яркими бликами (возможно, анимированную) на некие поверхности. Пример применения этого метода показан на рис. 11.60.

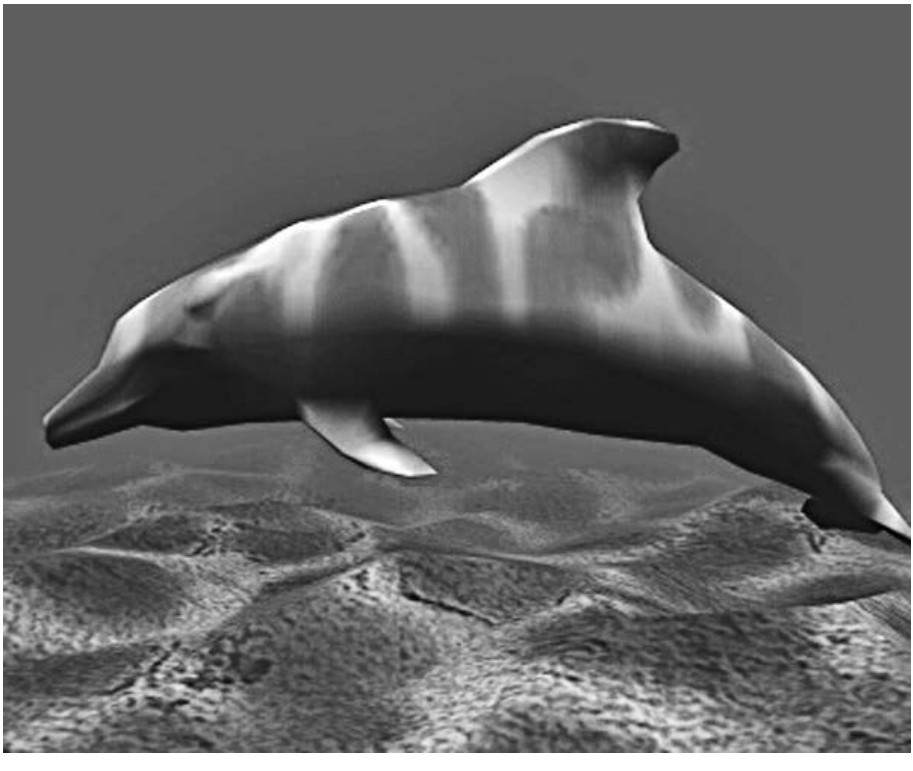


Рис. 11.60. Каустики от водной поверхности, созданные проецированием анимированной текстуры на другие поверхности

Подповерхностное рассеяние

Если свет входит в поверхность в одной точке, рассеивается под ней и снова выходит в другой точке, это называется *подповерхностным рассеянием*. Именно это явление вызывает теплое сияние человеческой кожи, восковых и мраморных статуй (рис. 11.61). Подповерхностное рассеяние описывается более сложным вариантом

функции BRDF (см. подраздел 11.1.3), известным как *функция двунаправленного распределения отражения поверхностного рассеяния* (bidirectional surface scattering reflectance distribution function, BSSRDF).

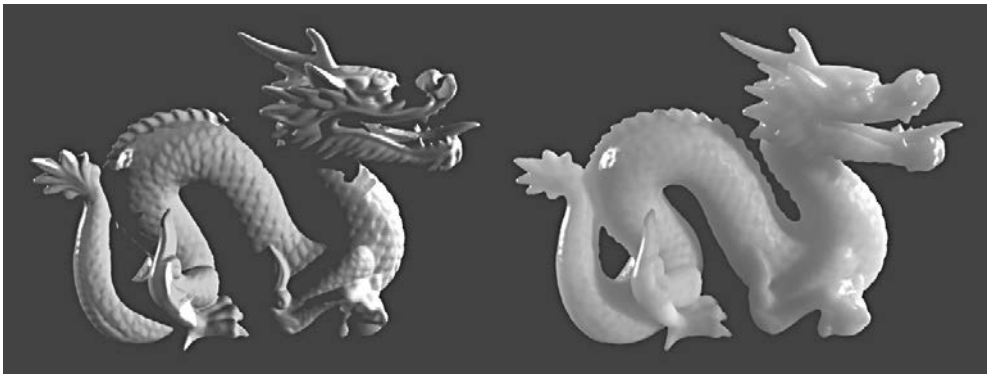


Рис. 11.61. *Слева* представлен результат отрисовки дракона без имитации подповерхностного рассеяния (то есть с использованием модели освещения BRDF). *Справа* — результат отрисовки того же дракона с имитацией подповерхностного рассеяния (то есть с применением модели BSSRDF). Изображения предоставлены Руи Вангом из Университета Вирджинии

Существует несколько способов имитации подповерхностного рассеяния. При реализации метода, основанного на использовании карт глубины, сначала отрисовывается теневая карта, которая затем применяется для определения того, какое расстояние необходимо преодолеть световому лучу, чтобы пройти сквозь объект, перекрывающий его путь, а не того, какие пикселы находятся в тени.

После этого для находящейся в тени стороны объекта задается искусственная диффузная составляющая освещения. Ее интенсивность обратно пропорциональна расстоянию, которое требуется пройти свету, чтобы выйти с противоположной стороны объекта. Это обеспечивает некоторое свечение объекта с противоположной источнику света стороны, но только в тех местах, где его толщина сравнительно невелика. Узнать больше о методах имитации подповерхностного рассеяния можно по адресу http.developer.nvidia.com/GPUGems/gpugems_ch16.html.

Предрассчитанный перенос излучения

Предрассчитанный перенос излучения (precomputed radiance transfer, PRT) — популярный метод, имитирующий эффекты методов рендеринга на основе излучательности в режиме реального времени. Он реализуется путем предварительного вычисления и сохранения полного определения способа взаимодействия с поверхностью (отражения, преломления, рассеивания и т. д.) светового луча, падающего под любым возможным углом. Это позволяет на этапе выполнения быстро считывать реакцию на конкретный падающий световой луч и преобразовывать ее в очень точные результаты освещения.

В общем случае реакция на свет в конкретной точке поверхности представляет собой сложную функцию, определенную на полусфере с центром в этой точке. Чтобы сделать основанный на PRT метод пригодным для практического применения, эту функцию нужно представить в некоем компактном виде. Общепринятый подход состоит в том, чтобы аппроксимировать данную функцию с помощью линейной комбинации сферических гармонических базисных функций. По сути, это трехмерная разновидность выражения простой скалярной функции $f(x)$ в виде линейной комбинации смещенных и масштабированных синусоидальных волн.

Подробное рассмотрение метода PRT выходит за рамки книги. Чтобы узнать о нем больше, см. статью по адресу web4.cs.ucl.ac.uk/staff/j.kautz/publications/prtSIG02.pdf. Методы освещения на основе PRT демонстрируются в примере программы для DirectX, доступном в комплекте средств разработки DirectX SDK (подробности см. по адресу msdn.microsoft.com/en-us/library/bb147287.aspx).

11.3.4. Отложенный рендеринг

При использовании традиционного рендеринга на основе растеризации треугольников все расчеты освещения и затенения выполняются для фрагментов треугольников в мировом пространстве, пространстве обзора или касательном пространстве. Проблемой данного метода является его неизбежная расточительность в плане расхода ресурсов. Во-первых, мы делаем работу, которая может оказаться ненужной, — после затенения вершин треугольника на этапе растеризации зачастую выясняется, что треугольник нужно полностью отбраковать по результатам *z*-теста. Использование раннего *z*-теста помогает устранить ненужные расчеты пиксельных шейдеров, однако не спасает положение. Во-вторых, при отрисовке сложной сцены с большим количеством источников света приходится создавать множество различных версий вершинных и пиксельных шейдеров, рассчитанных на разное количество источников света, разные их типы, разное количество весовых коэффициентов скиннинга и т. д.

Альтернативным способом затенения, позволяющим устранить значительную часть этих проблем, является *отложенный рендеринг*. При отложенном рендеринге большая часть расчетов освещения выполняется в экранном пространстве, а не в пространстве обзора. Сначала мы эффективно отрисовываем сцену, не беспокоясь об освещении. На этом этапе вся информация, которая может потребоваться в дальнейшем для освещения пикселов, сохраняется в глубоком кадровом буфере, называемом *G-буфером*. По завершении отрисовки сцены на основе информации, сохраненной в G-буфере, рассчитываются освещение и затенение. Такой подход обычно обеспечивает намного бóльшую эффективность по сравнению с выполнением освещения в пространстве обзора, не требует использования множества вариантов шейдеров и позволяет сравнительно легко воспроизводить некоторые очень привлекательные эффекты.

Хотя G-буфер может быть физически реализован как набор буферов, концептуально это единый кадровый буфер, содержащий множество информации о параметрах освещения и поверхности объектов сцены в каждом экранном пикселе. Типичный G-буфер может содержать следующие попиксельные атрибуты: глубину, нормаль поверхности в пространстве обзора или мировом пространстве, диффузный цвет, степень зеркального отражения и даже коэффициенты предрассчитанного переноса излучения (PRT). Некоторые типичные компоненты G-буфера демонстрирует ряд снимков экрана из игры *Killzone 2* студии Guerrilla Games (рис. 11.62).

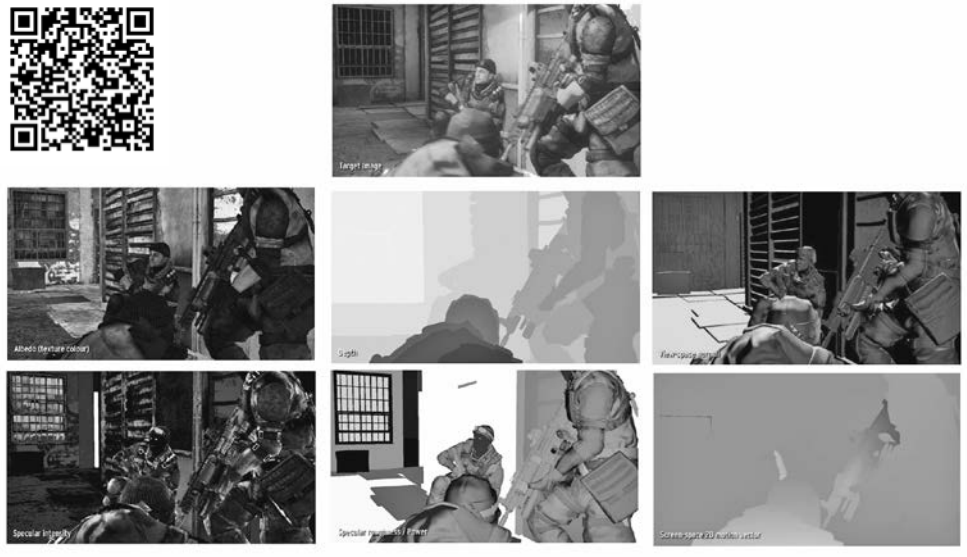


Рис. 11.62. Снимки экрана из игры Killzone 2 студии Guerrilla Games, демонстрирующие некоторые типичные компоненты G-буфера, используемые при отложенном рендеринге. Сверху представлен окончательный результат рендеринга. Ниже по часовой стрелке от верхнего левого угла: альбедо (диффузный цвет), глубина, нормаль в пространстве обзора, двухмерный вектор движения в экранном пространстве (для размытия движения), степень и интенсивность зеркального отражения

Всестороннее рассмотрение отложенного рендеринга выходит за рамки данной книги, однако ребята из студии Guerrilla Games подготовили отличную презентацию по этой теме, которая доступна по адресу www.slideshare.net/guerrillagames/deferred-rendering-in-killzone-2-9691589.

11.3.5. Физически корректное затенение

При использовании традиционных игровых движков освещения художникам и специалистам по освещению приходится настраивать обширный набор порой не слишком интуитивно понятных параметров многочисленных разнородных си-

стем движка рендеринга, чтобы добиться желаемого вида игровой сцены. Этот процесс может требовать больших затрат времени и сил. Что еще хуже, настройки параметров, которые хорошо работают при одних условиях освещения, могут не работать при других. Чтобы избавиться от этих проблем, программисты рендеринга все чаще работают с моделями *физически корректного затенения*.

Такая модель является попыткой аппроксимировать то, как свет распространяется и взаимодействует с материалами в реальном мире, позволяющей художникам и специалистам по освещению настраивать параметры шейдеров, используя интуитивно понятные, существующие в реальном мире величины, выраженные в существующих единицах. Всестороннее рассмотрение физически корректного затенения выходит за рамки книги, вы можете начать углубленное изучение этой темы со статьи, расположенной по адресу www.marmoset.co/toolbag/learn/pbr-theory.

11.4. Визуальные эффекты и наложения

Конвейер рендеринга, который мы обсуждали до сих пор, осуществляет главным образом рендеринг трехмерных твердых объектов. Помимо него, обычно используется также ряд специализированных систем рендеринга, выполняющих рендеринг таких визуальных элементов, как эффекты частиц, декали (накладываемые геометрические фигуры небольшого размера для отображения дырок от пуль, трещин, царапин и других элементов поверхности), волосы и мех, дождь и снегопад, водные поверхности и другие специфические визуальные эффекты. Могут также применяться полноэкранные постэффекты, такие как виньетка (уменьшение яркости и насыщенности по краям экрана), размытие движения, размытие по глубине резкости, искусственная/улучшенная колоризация и т. д.

Наконец, обычно реализуются также система меню игры и HUD-интерфейс путем отрисовки текста и другой двухмерной или трехмерной графики в экранном пространстве с наложением ее поверх трехмерной сцены.

Всестороннее рассмотрение этих систем движка выходит за рамки книги. В следующих разделах я сделаю лишь краткий обзор этих систем рендеринга и укажу, где можно найти дополнительную информацию.

11.4.1. Эффекты частиц

Система рендеринга частиц выполняет рендеринг аморфных объектов, таких как клубы дыма, искры, пламя и т. д., называемых *эффектами частиц*. В число ключевых черт, отличающих эффекты частиц от других видов отрисовываемой геометрии, входит следующее.

- Они состоят из очень *большого количества сравнительно простых* геометрических фигур — чаще всего из простых карт, называемых *квадрантами* и, в свою очередь, состоящих из двух треугольников.

- Эти геометрические фигуры часто *обращены к камере* (то есть представляют собой билборды), следовательно, движок должен принять определенные меры к тому, чтобы нормали граней каждого квадранта всегда были направлены на фокальную точку камеры.
- Их материалы почти всегда *полупрозрачны*. В силу этого на эффекты частиц накладываются строгие ограничения в отношении *порядка рендеринга*, которые не распространяются на большинство непрозрачных объектов сцены.
- Частицы *анимируются* множеством способов. Их положение, ориентация, размеры (масштаб), текстурные координаты и различные параметры затенения варьируются от кадра к кадру. Эти изменения определяются с помощью либо создаваемых вручную анимационных кривых, либо процедурных методов.
- Частицы обычно постоянно *размножаются* и *уничтожаются*. *Эмиттером частиц* называется некоторая логическая сущность в игровом мире, создающая частицы с определенной пользователем скоростью. Частицы уничтожаются при достижении ими предопределенной «плоскости смерти», по истечении заданной для них продолжительности жизни или при выполнении некоторых других определенных пользователем критериев.

Эффекты частиц могут отрисовываться с помощью обычной геометрии триангуляционных сеток с использованием соответствующих шейдеров. Однако в силу наличия у них перечисленных ранее уникальных характеристик в реальных игровых движках они всегда реализуются с помощью специализированной системы анимации и рендеринга эффектов частиц. Несколько примеров эффектов частиц показаны на рис. 11.63.

Рис. 11.63. Имитация пламени, дыма и пулевых трасс в игре Uncharted 3: Drake's Deception (снимок экрана с сайта http://gunfighter.ru/)

Проектирование и реализация системы частиц — весьма обширная тема, для объяснения которой требуется не одна глава. Чтобы узнать больше о системах частиц, см. [2, раздел 10.7], [16, раздел 20.5], [11, раздел 13.7] и [12, подраздел 4.1.2].

11.4.2. Декали

Декаль — это сравнительно небольшой геометрический объект, который накладывается поверх обычной геометрии сцены, позволяя изменять внешний вид поверхности динамическим образом. Декали используются для воспроизведения пулевых отверстий, следов обуви, царапин, трещин и т. д.

В современных движках принято моделировать декаль как прямоугольную область, которая проецируется вдоль луча на сцену. В результате образуется прямо-

угольная призма в трехмерном пространстве, и поверхность, которую она пересекает первой, становится поверхностью декаля. Треугольники пересеченной геометрии извлекаются и обрезаются по четырем ограничивающим плоскостям проецируемой призмы декаля. На полученные в результате этого треугольники накладывается нужная текстура декаля путем генерирования соответствующих текстурных координат для каждой вершины. Затем треугольники с наложенной текстурой отрисовываются поверх обычной сцены, часто с использованием параллактического наложения для придания им иллюзии глубины и с небольшим смещением по оси *Z* (оно обычно реализуется небольшим смещением передней плоскости) для исключения *z*-конфликта с той геометрией, на которую они накладываются. В итоге мы получаем пулевое отверстие, царапину или какую-то иную модификацию поверхности. Несколько декалей, воспроизводящих пулевые отверстия, показаны на рис. 11.64.

Рис. 11.64. Декали с параллактическим наложением в игре Uncharted 3: Drake's Deception (снимок экрана с сайта https://www.gameenginebook.com)

Чтобы узнать больше о создании и рендеринге декалей, см. [9, раздел 4.8] и [32, раздел 9.2].

11.4.3. Эффекты окружения

Любая игра, местом действия которой является хотя бы отчасти природное или реалистичное окружение, требует применения определенных эффектов окружения. Обычно они реализуются с помощью специализированных систем рендеринга. В следующих разделах мы кратко рассмотрим некоторые из наиболее распространенных таких систем.

Небо

Небо в игровом мире должно содержать яркие детали. Однако с технической точки зрения оно находится чрезвычайно далеко от камеры, поэтому мы не можем моделировать его в соответствующем реальности виде, что вынуждает нас использовать специализированные методы рендеринга.

Один из самых простых подходов состоит в том, чтобы заполнить кадровый буфер текстурой неба до отрисовки любой трехмерной геометрии. Эту текстуру необходимо отрисовывать с соотношением между текселами и пикселами примерно 1:1, чтобы разрешение текстуры приблизительно или точно равнялось разрешению экрана. Текстуру неба можно вращать и прокручивать в соответствии с движениями камеры в игре. Кроме того, при рендеринге неба нужно проследить за тем, чтобы для всех пикселов в кадровом буфере было задано максимально

возможное значение глубины. Это гарантирует, что после сортировки элементы трехмерной сцены всегда будут располагаться поверх неба. Такой способ отрисовки неба используется, например, в известном хите среди игр для игровых автоматов *Hydro Thunder*.

На современных игровых платформах с высоким уровнем затрат на затенение пикселов рендеринг неба часто выполняется *после* рендеринга остальной части сцены. При этом сначала очищается *z*-буфер путем внесения в него максимального значения координаты *z*. Затем отрисовывается сцена.

Наконец, после этого отрисовывается небо с включенным *z*-тестированием, отключенной записью в *z*-буфер и использованием в *z*-тесте значения, на единицу меньше максимального. В результате небо отрисовывается только там, где оно не закрыто более близкими объектами, такими как ландшафт, здания и деревья. Отрисовка неба в последнюю очередь гарантирует, что соответствующий пиксельный шейдер будет запущен для минимально возможного количества экранных пикселов.

В играх, в которых игрок может смотреть в любом направлении, можно использовать *небесный купол* или *небесную коробку*. Они всегда отрисовываются с центром в точке текущего расположения камеры, в результате чего поверхность купола или коробки кажется бесконечно удаленной, вне зависимости от того, куда перемещается камера в игровом мире. Как и в случае применения текстуры неба, небесный купол или коробка отрисовываются до отрисовки любой другой трехмерной геометрии с установкой максимального значения координаты *z* для всех пикселов в кадровом буфере. Подобное означает, что эти купол или коробка могут быть совсем небольшими по сравнению с другими объектами сцены. Размер купола или коробки не имеет значения — главное, чтобы при отрисовке неба был заполнен весь кадровый буфер. Чтобы узнать больше о рендеринге неба, см. [2, раздел 10.3] и [44, с. 253].

Специализированные системы рендеринга и анимации часто задействуются и для реализации облаков. В первых компьютерных играх, таких как *Doom* и *Quake*, облака представляли собой просто плоскости с прокручивающимися по ним полупрозрачными текстурами. Более поздние технологии отображения облаков включают в себя методы, основанные на использовании обращенных к камере карт (билбордов), эффектов частиц и объемных облачных эффектов.

Ландшафт

Задачей системы ландшафта является моделирование поверхности земли и создание своего рода холста для размещения других статических и динамических элементов. Иногда ландшафт моделируется напрямую в таком программном пакете, как Maya. Но когда игрок может видеть довольно сильно удаленный задний план, обычно необходимо использовать определенную систему динамической тесселяции или иную LOD-систему. Кроме того, иногда требуется также

ограничить объем данных, необходимых для представления очень больших открытых площадок.

Одним из популярных подходов к моделированию больших областей ландшафта является создание ландшафта на основе *поля высот*. Этот подход позволяет ограничиться сравнительно небольшим объемом данных, поскольку поле высот обычно сохраняется в виде текстурной карты в оттенках серого цвета. В большинстве систем ландшафта, использующих поля высот, сначала производится тесселяция горизонтальной плоскости ($y = 0$) в регулярную решетку, после чего определяются высоты вершин ландшафта путем сэмплирования текстуры поля высот. Количество треугольников на единицу площади может варьироваться в зависимости от расстояния от камеры, что позволяет отображать крупномасштабные удаленные элементы и в то же время обеспечивать приличную степень детализации близлежащего ландшафта. Пример ландшафта, определенного с помощью растрового изображения поля высот, показан на рис. 11.65.

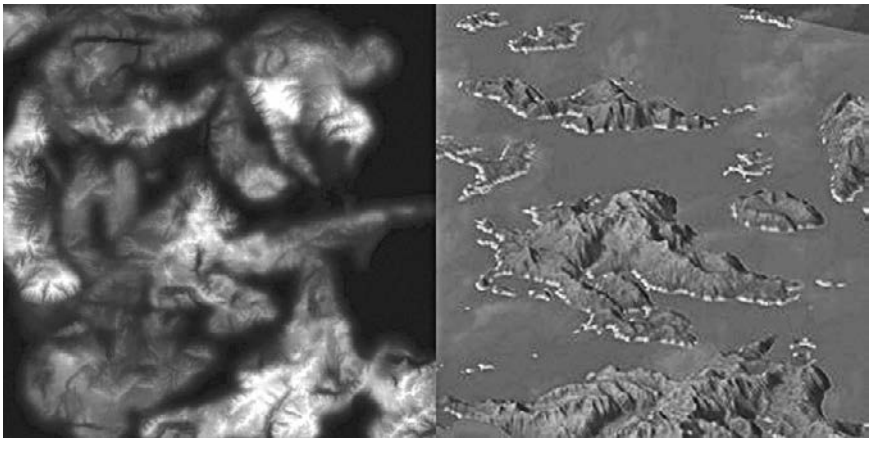


Рис. 11.65. Полутоновое растровое изображение поля высот *(слева)* может использоваться для управления вертикальным положением вершин в решетчатом меше ландшафта *(справа)*. Здесь плоскость водной поверхности пересекается с мешем ландшафта, образуя острова

Системы ландшафта обычно обеспечивают специализированные инструменты для непосредственного рисования поля высот, вырезания таких элементов ландшафта, как дороги, реки, и т. д. При этом процесс наложения текстур в системе ландшафта часто представляет собой смешивание содержимого четырех или более текстур. Это позволяет художникам «врисовывать» траву, почву, гравий и другие элементы ландшафта, просто проявляя один из текстурных слоев. Для обеспечения плавных переходов текстуры слои могут перекрестно смешиваться. Некоторые инструменты для создания ландшафта позволяют также вырезать участки ландшафта для вставки зданий, траншей и других специализированных элементов

в виде обычной сеточной геометрии. В одних движках инструменты для создания ландшафта интегрируются непосредственно в редактор игрового мира, а в других они существуют отдельно.

Конечно, создание ландшафта с использованием поля высот — это лишь один из множества способов моделирования поверхности земли в игре. Чтобы узнать больше о рендеринге ландшафта, см. [8, разделы 4.16–4.19] и [9, раздел 4.2].

Вода

Рендеры воды уже стали привычным явлением в мире игр. Существует много различных видов воды: океаны, бассейны, реки, водопады, фонтаны, струи, лужи и влажные твердые поверхности. Каждый из них обычно требует применения специальной технологии рендеринга, а некоторые — еще и динамической имитации движения. Для больших водоемов может потребоваться динамическая тесселяция или другая LOD-методология наподобие тех, которые используются в системах ландшафта.

Иногда системы рендеринга воды взаимодействуют с игровой системой динамики твердых тел (плавание, воздействие водяных струй и т. д.) и с игровым процессом (скользкие поверхности, механика плавания и ныряния, поднятие объектов вертикальными струями воды и т. д.). Водные эффекты часто создаются сочетанием разнородных технологий и подсистем рендеринга. Например, в случае водопада могут использоваться специализированные водные шейдеры, прокручиваемые текстуры, эффекты частиц для имитации тумана в нижней части водопада, похожее на декаль наложение для имитации пены — и это далеко не все.

Существующие сегодня игры предлагают порой действительно потрясающие водные эффекты, и есть надежда, что активные исследования таких технологий, как динамика жидкости в реальном времени, обеспечат нам еще более богатые возможности и большую реалистичность моделирования воды. Для получения дополнительной информации о методах рендеринга и моделирования воды см. [2, разделы 9.3, 9.5 и 9.6], [8, разделы 2.6 и 5.11] и [15].

11.4.4. Наложения

В большинстве игр используются HUD-интерфейсы, внутриигровые графические пользовательские интерфейсы и системы меню. Эти *наложения* обычно состоят из двухмерной или трехмерной графики, отрисовываемой непосредственно в пространстве обзора или экранном пространстве.

Наложения обычно создаются после отрисовки основной сцены при отключенном *z*-тестировании, чтобы гарантировать их отображение поверх трехмерной сцены. Двухмерные наложения чаще всего реализуются путем отрисовки квадрантов (пар треугольников) в экранном пространстве с использованием ортогональной проекции. Рендеринг трехмерных наложений может выполняться с помощью ортогональной проекции или обычной перспективной проекции с размещением геометрии в пространстве обзора, что обеспечивает отслеживание движений камеры.

Текст и шрифты

Система текста/шрифтов игрового движка обычно реализуется как особая разновидность двухмерного (иногда трехмерного) наложения. Система рендеринга текста, по сути, должна быть способна отображать последовательность символьных глифов, составляющих строку текста, с использованием различных вариантов их ориентации на экране.

Шрифт часто реализуется с помощью так называемого *глиф-атласа* — текстурной карты, содержащей необходимые глифы. Она обычно содержит только значения альфа-канала для каждого пиксела, которые указывают, в какой мере пиксел покрыт внутренней частью глифа. Файл описания шрифта предоставляет информацию о положении ограничивающей рамки каждого глифа внутри текстуры, о таких параметрах расположения шрифта, как кернинг, смещение базовой линии и т. д. Глиф отрисовывается путем отрисовки квадранта, координаты (u, v) которого соответствуют положению ограничивающей рамки нужного глифа внутри текстурной карты атласа. Текстурная карта дает альфа-значение, а цвет указывается отдельно, что позволяет создавать глифы любого цвета с помощью одного и того же атласа.

Другой способ рендеринга шрифтов состоит в том, чтобы использовать библиотеку шрифтов, такую как FreeType (https://www.freetype.org/). Библиотека FreeType позволяет игре или иному приложению считывать шрифты в любом из множества доступных форматов, включая TrueType (TTF) и OpenType (OTF), и отрисовывать глифы в размещенное в памяти пиксельное изображение, применяя любой размер шрифта в пунктах. Каждый глиф библиотеки FreeType отрисовывается с помощью соответствующих кривых Безье, что дает очень точные результаты.

В приложении в режиме реального времени, таком как игра, библиотека FreeType обычно используется для предварительной отрисовки необходимых глифов в атлас, который, в свою очередь, применяется для отрисовки глифов в виде простых квадрантов в каждом кадре. Но если вы встроите библиотеку FreeType или аналогичную в свой движок, это позволит отрисовывать те или иные глифы в атлас на лету или по мере необходимости. Это может быть полезным при рендеринге текста на языке с очень большим количеством возможных глифов, таком как китайский или корейский.

Еще один способ рендеринга высококачественных символьных глифов состоит в том, чтобы задействовать для определения глифов *знаковые поля расстояний*. При этом глифы отрисовываются в пиксельное изображение (как и при использовании библиотеки FreeType), однако записываемое для каждого пиксела значение уже не представляет собой альфа-значение «покрытия». Вместо этого каждый пиксел содержит снабженное знаком расстояние от своего центра до ближайшего края глифа. Расстояния имеют отрицательный знак внутри глифа и положительный — с внешней стороны его контура. При отрисовке глифа из текстурного атласа со знаковым полем расстояний пиксельный шейдер использует расстояния для вычисления высокоточных значений альфа-канала. Это дает текст с плавными очертаниями, вне зависимости от расстояния и угла обзора. Чтобы узнать больше

о рендеринге текста с применением знаковых полей расстояний, найдите в Интернете статью Константина Кэфера «Отрисовка текста с помощью знаковых полей расстояний в Mapbox GL» или статью Криса Грина из компании Valve под названием «Улучшенное увеличение с альфа-тестированием для векторных текстур и спецэффектов».

Глифы можно отрисовывать и непосредственно с помощью определяющих их контуры кривых Безье. Библиотека рендеринга шрифтов Slug компании Terathon Software LLC позволяет выполнять основанный на контурах рендеринг глифов в графическом процессоре, что делает этот метод пригодным для использования в игровых приложениях в режиме реального времени.

Хорошая система текста/шрифтов должна учитывать особенности различных языков в отношении набора символов и направления чтения. Процесс размещения символов в текстовой строке называют *формированием* строки. В зависимости от языка символы размещаются слева направо или справа налево, все они выравниваются по общей *базовой линии*. Интервал между символами отчасти определяется параметрами, заданными создателем шрифта (и сохраненными в файле шрифта), а отчасти — настройками *кернинга*, определяющими контекстные корректировки межсимвольного интервала.

Некоторые текстовые системы предоставляют такие интересные возможности, как разнообразные способы анимации символов в пределах экрана, анимация отдельных символов и т. д. Однако при создании игровой системы шрифтов важно помнить о том, что нужно реализовывать только *действительно необходимые* возможности. Например, вряд ли стоит снабжать игровой движок продвинутой анимацией текста, если в игре не требуется отображать анимированный текст.

11.4.5. Гамма-коррекция

Выходной сигнал большинства ЭЛТ-мониторов нелинейно зависит от уровня яркости. То есть если передать на ЭЛТ-монитор линейно возрастающие значения R, G и B, то полученное в итоге изменение изображения на экране будет восприниматься человеческим глазом как нелинейное. Темные области изображения будут выглядеть темнее, чем должны выглядеть (рис. 11.66).

Кривая гамма-отклика типичного ЭЛТ-монитора довольно просто моделируется с помощью формулы:

$$V_{\text{out}} = V_{\text{in}}^{\gamma},$$

где $\gamma_{\text{CRT}} > 1$. Для компенсации этого эффекта передаваемые на ЭЛТ-монитор значения цвета обычно подвергаются обратному преобразованию (то есть используется значение $\gamma_{\text{corr}} < 1$). Поскольку значение γ_{CRT} типичного ЭЛТ-монитора составляет 2,2, обычно корректирующее значение $\gamma_{\text{corr}} \approx 1/2{,}2 = 0{,}455$. Эти кривые кодирования и декодирования гаммы показаны на рис. 11.67.

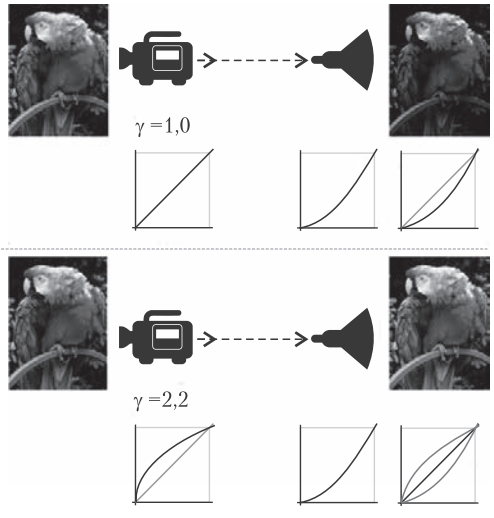


Рис. 11.66. Влияние гамма-отклика ЭЛТ-монитора на качество изображения и внесение соответствующей поправки (изображение предоставлено онлайн-энциклопедией www.wikipedia.org)

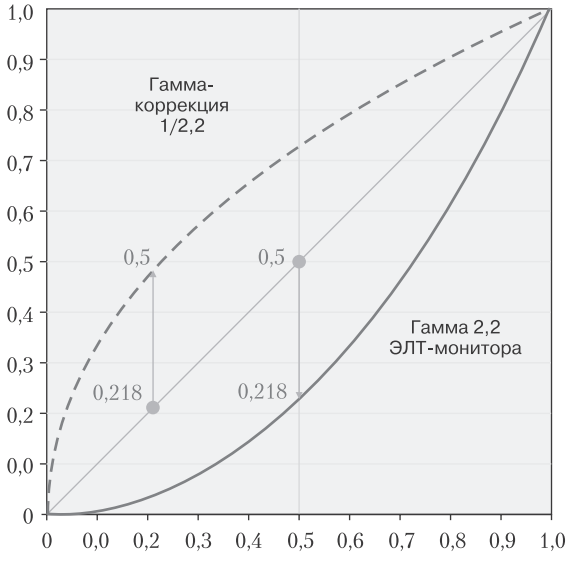


Рис. 11.67. Кривые кодирования и декодирования гаммы (изображение предоставлено онлайн-энциклопедией www.wikipedia.org)

Движок 3D-рендеринга может выполнять гамма-кодирование, чтобы гарантировать, что итоговое изображение будет содержать цветовые значения, должным образом подвергнутые гамма-коррекции. Однако здесь возникает проблема:

растровые изображения, используемые для представления текстурных карт, часто сами по себе подвергаются гамма-коррекции. Учитывая это, высококачественные движки рендеринга выполняют гамма-декодирование текстур перед рендерингом, а затем гамма-кодирование итоговой сцены для правильного воспроизведения ее цветов на экране.

11.4.6. Полноэкранные постэффекты

Полноэкранные *постэффекты* применяются к отрисованной трехмерной сцене для придания ей дополнительной реалистичности или стилизованного вида. Они часто реализуются пропусканием всего содержимого экрана через пиксельный шейдер, который применяет нужный (-ые) эффект (-ы). Это может быть обеспечено рендерингом полноэкранного квадранта с наложенной на него текстурой, содержащей сцену без применения фильтра. Вот несколько примеров полноэкранных постэффектов.

- *Размытие движения*. Обычно реализуется отрисовкой буфера векторов скорости в экранном пространстве и применением этого векторного поля для выборочного размытия отрисованного изображения. Размытие обеспечивается передачей *ядра свертки* по изображению (подробности см. в книге [5], в главе «Сглаживание и повышение резкости изображений с использованием дискретной свертки», написанной Д. А. Шумахером).
- *Размытие по глубине резкости*. Данный эффект можно реализовать использованием содержимого буфера глубины для регулировки степени размытия, применяемого к каждому пикселу.
- *Виньетка*. Данный кинематографический эффект сводится к уменьшению яркости и насыщенности изображения по краям экрана для повышения драматичности. Иногда он реализуется буквальной отрисовкой текстурного наложения поверх экрана. Разновидность этого эффекта — классический круговой эффект, используемый, когда игрок смотрит в бинокль или прицел оружия.
- *Колоризация*. Цвета экранных пикселов могут произвольно модифицироваться в качестве постэффекта. Например, обесцветив все цветовые каналы, кроме красного, можно воспроизвести потрясающий по силе воздействия эффект сцены с маленькой девочкой в красном пальто из фильма «Список Шиндлера».

11.5. Дополнительная литература

В весьма ограниченных рамках этой главы мы рассмотрели много тем, затронув лишь верхушку айсберга. Нет никаких сомнений в том, что многие из них вам потребуется изучить более подробно. Я настоятельно рекомендую ознакомиться с книгой [27], где вы найдете прекрасный обзор всего процесса создания трехмерной компьютерной графики и анимации для игр и киноиндустрии. Книга [2]

содержит очень подробное описание технологий, лежащих в основе современного рендеринга в реальном времени, а книга [16] хорошо известна как исчерпывающий справочник по всему, что имеет отношение к компьютерной графике.

К числу отличных изданий по 3D-рендерингу можно также отнести [11], [12] и [49]. Математика 3D-рендеринга очень хорошо освещена в книге [32]. Библиотека любого программиста компьютерной графики будет неполной без одной или нескольких книг из серии *Graphics Gems* ([5], [20], [22], [28] и [42]) и/или серии *GPU Gems* ([15], [40] и [44]). Конечно, этот короткий список литературы может служить лишь отправной точкой. Развиваясь как программист игр, вы найдете множество других превосходных книг по рендерингу и шейдерам.

12 Системы анимации

В большинстве современных 3D-игр центральную роль играют *персонажи* — часто это люди или человекоподобные существа, иногда животные или монстры. Отличительной чертой персонажей является то, что они должны двигаться плавно и органично. Это порождает множество дополнительных технических проблем, помимо тех, которые требуется решить в случае моделирования и анимации таких твердых объектов, как транспортные средства, снаряды, футбольные мячи и фигуры тетриса. Задачу наделения персонажей естественно выглядящими движениями решает компонент движка, называемый *системой анимации персонажей*.

Как мы увидим далее, система анимации предоставляет разработчикам игр мощный набор инструментов, которые можно применять не только к персонажам, но и к другим объектам. Возможности системы анимации могут с успехом задействоваться в любом мало-мальски подвижном объекте игры. Поэтому, когда вы видите в игре транспортное средство с подвижными частями, шарнирно сочлененный механизм, слегка покачивающиеся на ветру деревья или даже взрывающееся здание, велика вероятность того, что в этом объекте хотя бы отчасти используется анимационная система игрового движка.

12.1. Разновидности анимации персонажей

Технологии анимации персонажей прошли долгий путь развития со времен серии игр *Donkey Kong*. В первых играх использовались простейшие методы имитации движений живых существ. По мере роста возможностей игрового аппаратного обеспечения стало возможным применение более сложных методов в режиме реального времени.

Сегодня в распоряжении разработчиков игр имеется множество мощных методов анимации. В этом разделе мы кратко коснемся эволюции анимации персонажей и в общих чертах рассмотрим три метода, которые получили наибольшее распространение в современных игровых движках.

12.1.1. Келевая анимация

Предшественником всех методов игровой анимации является *традиционная рисованная анимация*. Речь идет о методе, который использовался уже в первых мультипликационных фильмах. Иллюзия движения создается отображением последовательности быстро сменяющихся неподвижных изображений — *кадров*. 3D-рендеринг в реальном времени можно рассматривать как электронную разновидность традиционной анимации, в которой перед зрителем проходит последовательность неподвижных полноэкранных изображений, создавая иллюзию движения.

Особой разновидностью традиционной анимации является *келевая анимация*. *Кель* — это прозрачный лист целлулоида (cel — от слова celluloid), на котором можно рисовать изображения. Размещая анимированную последовательность келей поверх фиксированного фонового рисунка, можно создавать иллюзию движения, не перерисовывая раз за разом статический фон.

Аналогом келевой анимации в компьютерной графике является так называемая *спрайтовая анимация*. Спрайт — это небольшое растровое изображение, которое можно накладывать поверх полноэкранного фонового изображения, не искажая его. Обычно оно отрисовывается с помощью специализированного графического аппаратного обеспечения. То есть в двухмерной игровой анимации спрайт играет ту же роль, которую играл кель в традиционной анимации. Этот метод анимации был основным в эпоху двухмерных игр. На рис. 12.1 показана хорошо известная последовательность спрайтов, применявшаяся для создания иллюзии бегущего гуманоидного персонажа практически во всех играх для приставок Mattel Intellivision. Эта последовательность кадров организована таким образом, чтобы она плавно проигрывалась и при бесконечном повторении, — такой подход называется *циклической анимацией*. Пользуясь современной терминологией, данную анимацию можно назвать *циклом бега*, поскольку она создает иллюзию того, что персонаж бежит. Обычно персонаж имеет несколько циклов анимации: различные циклы бездействия, цикл ходьбы и цикл бега.

Рис. 12.1. Последовательность спрайтов, применявшаяся в большинстве игр для приставок Intellivision

12.1.2. Жесткая иерархическая анимация

Системы анимации первых 3D-игр, таких как *Doom*, незначительно отличались от спрайтовой анимации: монстры в этих играх представляли собой не что иное, как обращенные к камере квадранты, создающие иллюзию движения за счет отображения на них последовательности текстурных растровых изображений, называемых

анимированными текстурами. Эта техника используется и сегодня для объектов с низким разрешением и/или удаленных, таких как толпа на стадионе или множество солдат, сражающихся на заднем плане. Однако развитие трехмерной графики потребовало применения для высококачественных персонажей переднего плана более совершенных методов анимации персонажей.

Изначально для трехмерной анимации персонажей использовали метод *жесткой иерархической анимации*. При этом персонаж моделируется как набор жестких фигур. Так, гуманоидный персонаж обычно разбивается на следующие составные части: таз, торс, верхние и нижние части рук, верхние и нижние части ног, кисти рук, ступни ног, голова. Эти жесткие части связываются друг с другом в иерархическом порядке, подобно тому как соединены в суставах кости млекопитающих животных. Это заставляет персонаж двигаться естественным для него образом. Так, например, когда начинает двигаться верхняя часть руки, за ней автоматически следуют нижняя часть руки и кисть. Обычно в качестве корневого узла иерархии выступает таз, его непосредственными дочерними узлами являются торс и верхние части ног и т. д.:

```
Pelvis
    Torso
        UpperRightArm
            LowerRightArm
                RightHand
        UpperLeftArm
            UpperLeftArm
                LeftHand
        Head
    UpperRightLeg
        LowerRightLeg
            RightFoot
    UpperLeftLeg
        UpperLeftLeg
            LeftFoot
```

Большой проблемой жесткой иерархической анимации является то, что тело персонажа часто начинает выглядеть не очень привлекательно из-за «растрескивания» в суставах (рис. 12.2). Этот метод хорошо подходит для роботов и машин, которые действительно состоят из жестких компонентов, но не выдерживает никакой критики в случае его применения к персонажам, которые хоть в какой-то мере обладают плотью.

12.1.3. Повершинная анимация и морфинг-мишени

Жесткая иерархическая анимация, часто выглядит неестественно именно в силу своей жесткости. Поэтому нужно перемещать отдельные вершины таким образом, чтобы треугольники могли растягиваться, обеспечивая более естественный вид движений.

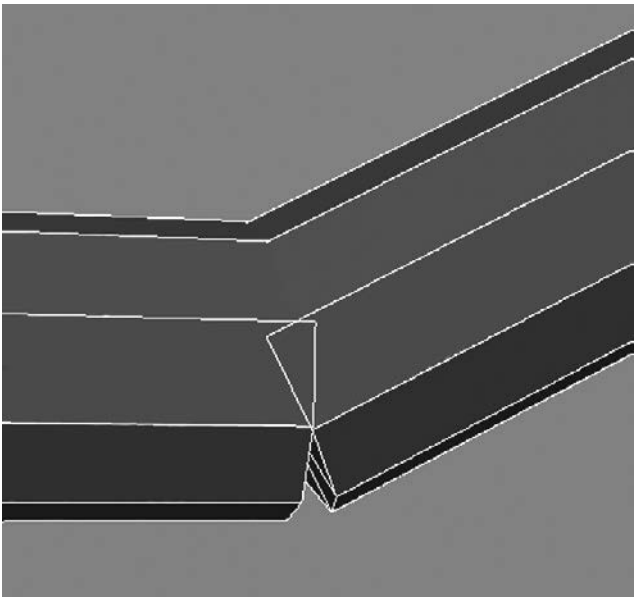


Рис. 12.2. «Растрескивание» в суставах — большая проблема жесткой иерархической анимации

Помимо других способов, для этого можно использовать такой прямолинейный метод, как *повершинная анимация*. При этом вершины сетки анимируются художником, после чего данные о движении экспортируются и используются в качестве указания игровому движку о том, как следует перемещать каждую из них в динамическом режиме. Данный метод позволяет деформировать сетку практически любым вообразимым способом (ограничением является лишь степень тесселяции поверхности). Однако это информационно емкий метод, поскольку для каждой вершины сетки требуется сохранять изменяющиеся во времени данные о движении. В силу этого он практически не применяется в играх в режиме реального времени.

В некоторых играх в режиме реального времени используется разновидность этой техники, известная как *анимация по морфинг-мишеням*. При этом аниматор перемещает вершины сетки, задействуя сравнительно небольшой набор фиксированных крайних поз. Анимация выполняется *смешиванием* двух или большего количества фиксированных поз в динамическом режиме. Положение каждой вершины рассчитывается посредством простой линейной интерполяции (linear interpolation, LERP) между положениями вершины в каждой из крайних поз.

Метод морфинг-мишеней часто используется для лицевой анимации, потому что у человеческого лица чрезвычайно сложная анатомия и оно приводится в движение примерно 50 мышцами. Это дает аниматору полный контроль над каждой

Рис. 12.3. Набор лицевых морфинг-мишеней для персонажа Элли из игры The Last of Us: Remastered (снимок экрана с сайта https://www.gameenginebook.com)

вершиной лицевой сетки, позволяя реализовывать как едва уловимые, так и очень большие движения, хорошо имитирующие мускулатуру лица. На рис. 12.3 показано, как может выглядеть набор лицевых морфинг-мишеней.

По мере роста вычислительных мощностей некоторые студии вместо морфинг-мишеней начали использовать суставные лицевые риги, включающие в себя сотни суставов. Другие студии сочетают оба метода, создавая базовое выражение лица с помощью суставных ригов и внося небольшие изменения с помощью морфинг-мишеней.

12.1.4. Скиновая анимация

По мере дальнейшего роста возможностей игрового аппаратного обеспечения был разработан метод *скиновой анимации*. Он обладает почти всеми плюсами поверхностной анимации и анимации по морфинг-мишеням, позволяя деформировать треугольники анимируемой сетки. Все это сочетается с намного более высокой эффективностью в плане производительности и потребления памяти, характерной для жесткой иерархической анимации. С помощью этого метода можно довольно реалистично имитировать движение кожи и одежды.

Скиновая анимация была впервые применена в таких играх, как *Super Mario 64*, и по сей день остается наиболее распространенным методом анимации как в индустрии компьютерных игр, так и в кино. Персонажи многих известных современных игр и кинофильмов как минимум частично, а то и полностью анимировались с помощью этого метода. Это, в частности, динозавры из фильма «Парк юрского периода», Солид Снейк из серии игр *Metal Gear*, Голлум из фильма «Властелин колец», Натан Дрейк из игры *Uncharted*, Базз Лайтер из мультфильма «История игрушек», Маркус Феникс из серии игр *Gears of War* и Джоэл из серии игр *The Last of Us*. Далее в этой главе мы сосредоточимся главным образом на изучении скиновой/скелетной анимации.

Как и в жесткой иерархической анимации, в скиновой анимации составляется *скелет* из жестких костей. Но вместо того, чтобы отрисовывать эти жесткие части на экране, их оставляют скрытыми. К суставам скелета привязывается так называемый скин — плавная непрерывная триангулярная сетка, вершины которой отслеживают движения суставов. Для каждой вершины можно определить весовые коэффициенты влияния нескольких суставов, что обеспечивает реалистичное растяжение скина при их движении.

На рис. 12.4 представлен игровой персонаж Горностай Крэнк *(Crank the Weasel)*, созданный Эриком Браунингом в 2001 году для компании Midway Home Entertainment. Внешняя оболочка, или скин, Крэнка состоит из сетки треуголь-

ников, как и любая другая 3D-модель. А внутри него мы видим жесткие кости и суставы, которые приводят скин в движение.

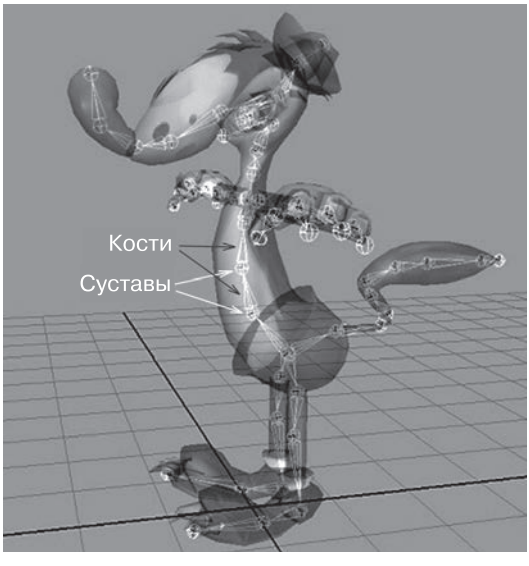


Рис. 12.4. Горностай Крэнк с внутренней скелетной структурой

12.1.5. Методы анимации как методы сжатия данных

В идеализированно гибкой системе анимации у аниматора будет контроль буквально над каждым бесконечно малым участком поверхности объекта. Конечно, такой подход приведет к созданию анимации, потенциально содержащей бесконечно большой объем данных. Анимация вершин триангулярной сетки является, по сути, упрощением этого идеального случая — мы *сжимаем* информацию, необходимую для описания анимации, ограничиваясь только возможностью перемещать вершины. (Анимация набора контрольных точек представляет собой аналог вершинной анимации для моделей, состоящих из патчей более высокого порядка.) Морфинг-мишени можно рассматривать как дополнительный уровень сжатия, обеспечиваемый за счет наложения на систему дополнительных ограничений, допускающих перемещение вершин только по линейным траекториям между фиксированным количеством заранее заданных положений. Скелетная анимация, по сути, представляет собой еще один способ сжатия данных вершинной анимации путем наложения ограничений. В этом случае ограничение состоит в том, что сравнительно большое количество вершин должно отслеживать движения сравнительно небольшого количества суставов скелета.

Решая, какой из методов анимации лучше выбрать в том или ином случае, часто полезно думать о них как о методах сжатия, во многом аналогичных методам

сжатия видео. В общем случае нужно выбирать метод анимации, который обеспечивает наилучшее сжатие, не создавая неприемлемых визуальных артефактов. Скелетная анимация дает наилучшее сжатие, когда движение одного сустава транслируется в движение множества вершин. Так, скелет позволяет очень эффективно перемещать конечности персонажа, поскольку они ведут себя почти как твердые тела.

Однако движения лица обычно намного сложнее и требуют более независимого перемещения отдельных вершин. Для того чтобы скелетный метод обеспечивал достаточно реалистичную анимацию лица, количество используемых суставов должно быть ненамного меньше количества вершин в сетке, что снижает его эффективность в качестве метода сжатия. Это один из главных доводов в пользу того, чтобы использовать для анимации лица метод морфинг-мишеней, а не скелетный подход. (Другой веский довод таков: применение морфинг-мишеней обычно более привычный для аниматоров способ работы.)

12.2. Скелеты

Скелет состоит из *иерархии* жестких элементов, называемых *суставами*. Хотя в игровой индустрии под суставами и костями часто понимают одно и то же, такое употребление слова «кости» не совсем корректно. Если судить с технической точки зрения, то суставы — это объекты, которыми напрямую манипулирует аниматор, а кости — пустое пространство между ними. В качестве примера рассмотрим тазобедренный сустав модели Горностая Крэнка. Он соединен с четырьмя другими суставами (в хвосте, позвоночнике и двух ногах) и в силу этого образует четыре выходящие из него кости (рис. 12.5). Игровые движки не обращают никакого

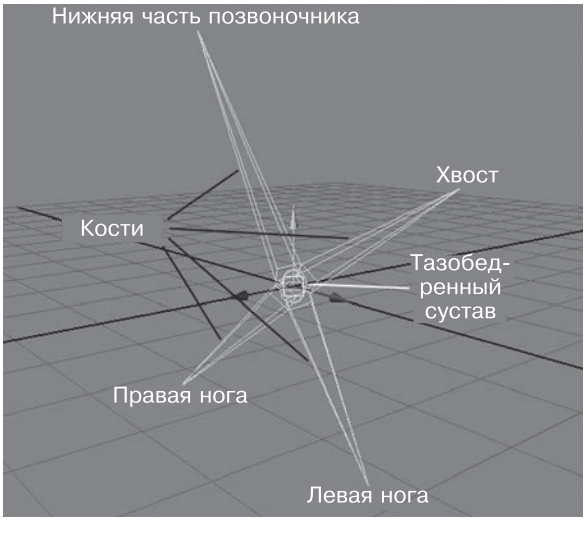


Рис. 12.5. Тазобедренный сустав персонажа соединен с четырьмя другими суставами (в хвосте, позвоночнике и двух ногах) и, соответственно, образует четыре кости

внимания на кости — для них важны только суставы. Поэтому, когда вы слышите слово «кость» в контексте игровой индустрии, знайте, что при этом почти всегда имеются в виду суставы.

12.2.1. Скелетная иерархия

Как мы уже упоминали, суставы в скелете образуют иерархию или древовидную структуру. Один сустав выбирается корневым, а все остальные становятся по отношению к нему дочерними, внучатыми и т. д. Типичная иерархия суставов для скиновой анимации выглядит практически так же, как типичная иерархия для жесткой иерархической анимации. Так, иерархия суставов гуманоидного персонажа может выглядеть примерно так, как показано на рис. 12.6.

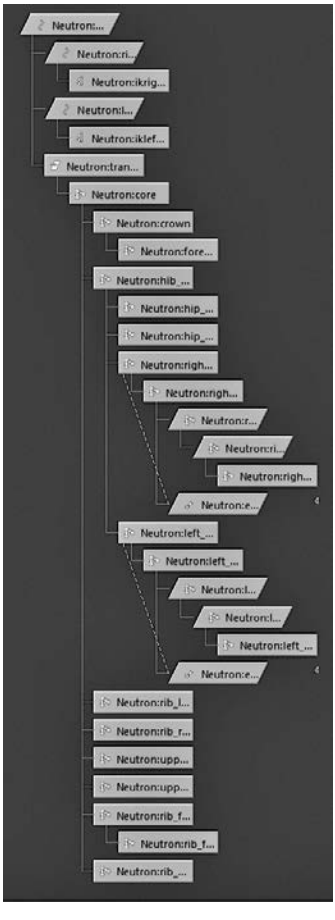


Рис. 12.6. Пример скелетной иерархии, представленной в виде гиперграфа в приложении Maya

Любому суставу присваивается индекс в диапазоне от 0 до $N - 1$. Поскольку каждый сустав имеет один и только один родительский сустав, иерархическая структура скелета может быть полностью определена путем сохранения в каждом суставе индекса его родительского сустава. Так как у корневого сустава нет родительского сустава, его индексу родительского сустава обычно присваивается недопустимое значение, например –1.

12.2.2. Представление скелета в памяти

Для представления скелета обычно используется небольшая структура данных верхнего уровня, содержащая массив структур данных отдельных суставов. Чаще всего выбирается такой порядок следования суставов в массиве, при котором дочерний сустав располагается после родительского. Это означает, что корнем скелета всегда является нулевой сустав.

Для обращения к отдельным суставам в пределах структуры данных обычно используются *индексы суставов*. Дочерний сустав ссылается на родительский, указывая его индекс. Аналогичным образом в заскиненной триангулярной сетке вершина ссылается по индексу на суставы, к которым привязана. Это гораздо эффективнее, чем ссылаться на суставы по имени, с точки зрения как потребления памяти (если принять максимальное число суставов в скелете равным 256, индекс сустава может занимать всего 8 бит), так и затрат времени на поиск нужного сустава (мы можем мгновенно перейти к нужному суставу в массиве, используя его индекс).

Структуры данных отдельных суставов обычно содержат следующую информацию:

- *имя* сустава в виде строки либо ее хэшированного 32-битного идентификатора;
- индекс родительского сустава в скелете;
- *обратную матрицу преобразования позы привязки* для данного сустава. Поза привязки сустава представляет собой данные о том, какими были положение, ориентация и масштаб этого сустава в момент его связывания с вершинами сетки скина. По причинам, которые будут подробно рассмотрены в последующих разделах, обычно сохраняется *обратная матрица* этого преобразования.

Типичная структура данных скелета может выглядеть примерно так:

```
struct Joint
{
    Matrix4x3   m_invBindPose; // обратная матрица преобразования позы привязки
    const char* m_name;        // удобное для чтения человеком имя сустава
    U8          m_iParent;     // индекс родительского сустава или 0xFF
                               // в случае корневого сустава
};

struct Skeleton
{
    U32     m_jointCount;  // количество суставов
    Joint*  m_aJoint;      // массив суставов
};
```

12.3. Позы и положения

Какой бы подход ни использовался для создания анимации (жесткий иерархический или скиновый/скелетный), она всегда воспроизводится постепенно. Чтобы создать иллюзию движения, тело персонажа последовательно принимает статические *позы*, которые быстро чередуются на экране обычно с частотой 30 или 60 *поз в секунду* (на самом деле, как вы увидите в подразделе 12.4.1, вместо строгого отображения каждой позы часто используют *интерполяцию* между соседними позами). В скелетной анимации поза скелета напрямую управляет вершинами меша, а позиционирование является основным инструментом аниматора, с помощью которого тот вдыхает жизнь в своих персонажей. Поэтому очевидно, что прежде, чем мы сможем анимировать скелет, нужно понять, как придать ему *позу*.

Скелет позиционируется за счет произвольного вращения, перемещения и иногда масштабирования его суставов. *Положение* сустава определяется его положением, направлением и масштабом относительно какой-то системы отсчета. Положение сустава обычно представлено в виде матрицы размерностью 4×4 или 4×3 или же структуры данных SRT (scale, quaternion rotation, vector translation — масштаб, кватернионное вращение, векторное перемещение). Поза скелета — это просто набор положений всех его суставов, который обычно имеет вид массива матриц или структур SRT.

12.3.1. Поза привязки

На рис. 12.7 показаны две позы одного и того же скелета. Слева мы видим специальную *позу привязки*, которую иногда называют *позой отсчета* или *позой покоя*. Это состояние трехмерного меша, в котором он находится перед привязкой к скелету (отсюда и название). Иными словами, это поза, которую принял бы обычный, нескиновый треугольный меш без какого-либо скелета. Эту позу также называют *Т-образной*, поскольку персонаж напоминает по форме букву «Т» — стоит со слегка расставленными ногами и вытянутыми руками. Эта стойка была выбрана потому, что в ней конечности отдалены от туловища и друг друга, что облегчает процесс привязки вершин к суставам.

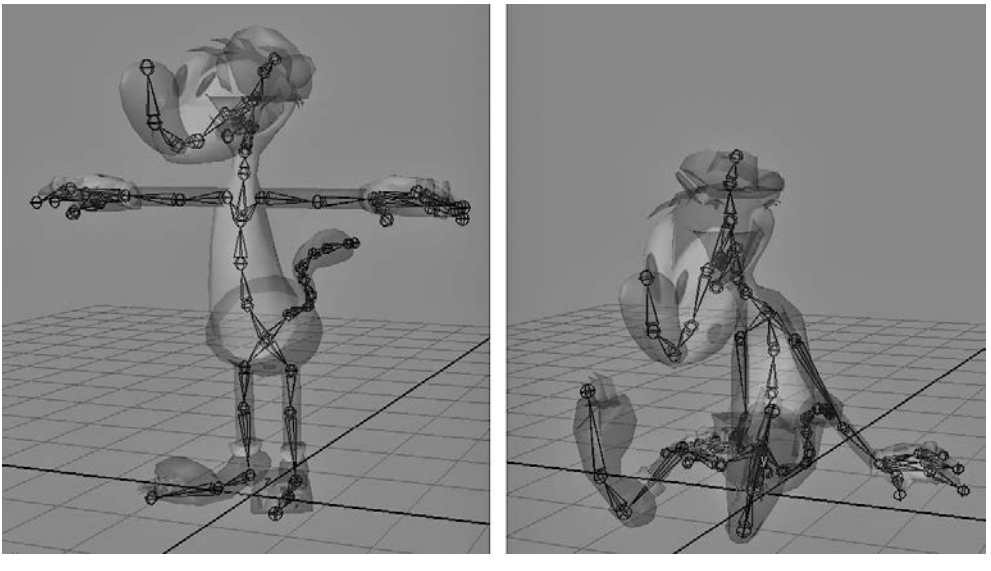


Рис. 12.7. Две позы одного и того же скелета. Слева показана поза привязки

12.3.2. Локальные позы

Положения дочерних суставов чаще всего указываются относительно *родительских*. Это обеспечивает естественное движение суставов. Например, если мы повернем плечевой сустав, а положения локтя, запястья и пальцев оставим без изменений, вокруг плеча жестко повернется вся рука, как и ожидалось. Позу или положение, заданные относительно родительского сустава, иногда называют *локальными*. Локальные позы почти всегда хранятся в формате SRT. Причины этого мы исследуем при обсуждении слияния анимации.

Многие пакеты 3D-моделирования, такие как Maya, отображают суставы в виде небольших сфер. Однако, помимо положения, у сустава также есть поворот и масштаб, поэтому его визуализация может быть немного обманчивой. На самом деле

сустав определяет координатное пространство, которое принципиально ничем не отличается от других пространств, рассмотренных ранее, таких как пространство модели, пространство игрового мира или пространство просмотра. Поэтому его лучше представить себе в виде набора осей в прямоугольной системе координат. Maya дает пользователю возможность отображать локальные оси координат сустава (рис. 12.8).

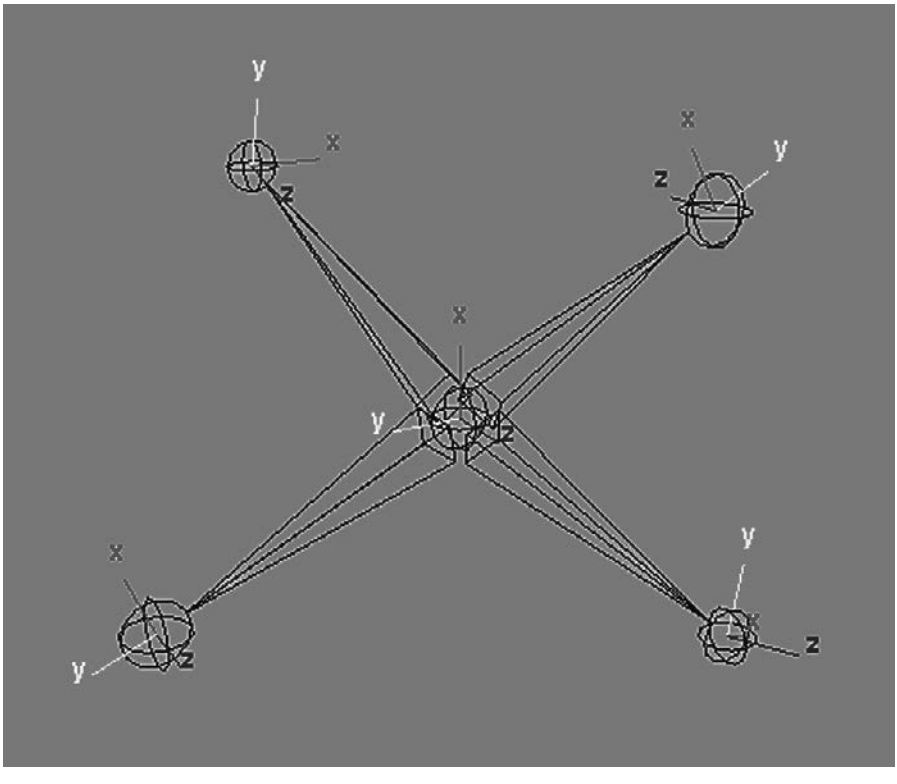


Рис. 12.8. Каждый сустав в скелетной иерархии определяет оси координат в локальном пространстве, известном как пространство сустава

В математическом смысле положение сустава — не что иное, как *аффинное преобразование*. Положение сустава j можно записать в виде матрицы преобразования $\mathbf{P}_j$ размером 4×4, которая состоит из вектора преобразования $\mathbf{T}_j$, диагональной матрицы масштабирования $\mathbf{S}_j$ размером 3×3 и матрицы вращения $\mathbf{R}_j$ того же размера. Позу всего скелета $\mathbf{P}_{\text{skel}}$ можно записать в виде набора всех поз $\mathbf{P}_j$, где j находится в диапазоне от 0 до $N - 1$:

$$\mathbf{P}_j = \begin{bmatrix} \mathbf{S}_j\mathbf{R}_j & 0 \\ \mathbf{T}_j & 0 \end{bmatrix}; \; \mathbf{P}_{\text{skel}} = \left\{\mathbf{P}_j\right\}\Big|_{j=0}^{N-1}.$$

Масштаб суставов

Некоторые игровые движки исходят из того, что суставы никогда не масштабируются, в этом случае $\mathbf{S}_j$ считается единичной матрицей и просто опускается. Иногда подразумевается, что масштаб, если он присутствует, является *однородным*, то есть одинаковым во всех трех плоскостях. В этом случае его можно представить в виде единого скалярного значения s_j. Некоторые движки разрешают даже *неоднородное* масштабирование, в результате чего масштаб можно лаконично представить как трехэлементный вектор $\mathbf{s}_j = \begin{bmatrix} s_{jx} & s_{jy} & s_{jz} \end{bmatrix}$. Элементы $\mathbf{s}_j$ соответствуют трем диагональным элементам матрицы масштабирования $\mathbf{S}_j$ размером 3×3, так что это не вектор *как таковой*. Игровые движки почти никогда не поддерживают сдвиги, поэтому значение $\mathbf{S}_j$ почти никогда не представлено полной матрицей масштабирования/сдвига размером 3×3, хотя оно, несомненно, *могло бы* ею быть.

Опуская или ограничивая масштаб в позе или анимации, мы получаем ряд преимуществ. Очевидно, использование представления с меньшим количеством измерений может сэкономить память (однородное масштабирование требует единого скалярного значения с плавающей запятой для каждого сустава и в каждом кадре, тогда как для неоднородного нужны три таких значения, а для полной матрицы масштабирования/смещения размером 3×3 — целых девять). Однородное масштабирование также гарантирует, что сфера привязки сустава никогда не превратится в эллипсоид, как при неоднородном масштабе. Это существенно упрощает математический аспект проверок на усечение и столкновение в движке, который выполняет такие проверки для каждого сустава.

Представление положения суставов в памяти

Как уже упоминалось, положения суставов обычно хранятся в формате SRT. В C++ такая структура данных может выглядеть следующим образом (элемент Q идет первым, чтобы обеспечить правильное выравнивание и оптимальную упаковку структуры, — догадываетесь почему?):

```
struct JointPose
{
    Quaternion m_rot;   // R
    Vector3    m_trans; // T
    F32        m_scale; // S (только однородный масштаб)
};
```

Если допускается неоднородный масштаб, положение сустава можно определить так:

```
struct JointPose
{
    Quaternion m_rot;   // R
    Vector4    m_trans; // T
    Vector4    m_scale; // S
};
```

Далее показано, каким образом можно представить локальную позу всего скелета (подразумевается, что массив `m_aLocalPose` делается динамическим, чтобы вместить количество экземпляров `JointPose`, равное числу суставов в скелете):

```
struct SkeletonPose
{
    Skeleton*  m_pSkeleton;  // скелет + количество суставов
    JointPose* m_aLocalPose; // локальные положения суставов
};
```

Положение сустава в виде изменения основания

Важно помнить, что *локальное* положение сустава задается относительно его ближайшего родителя. Любое аффинное преобразование можно считать переносом точек и векторов из одного пространства в другое. Поэтому, если применить преобразование положения сустава $\mathbf{P}_j$ к точке или вектору, которые находятся в системе координат сустава *j*, результатом будут те же точка или вектор, но уже в пространстве родительского сустава.

По традиции, сложившейся в предыдущих главах, мы условимся, что направление преобразования обозначается подстрочными знаками. Поскольку положение сустава берет точки и векторы из *дочернего* пространства (C) и переносит его в *родительский* сустав (P), можно записать это как $(\mathbf{P}_{\mathrm{C}\to\mathrm{P}})_j$. Как вариант можно ввести функцию $p(j)$, которая возвращает родительский индекс сустава *j*, и обозначить локальное положение этого сустава $\mathbf{P}_{j\to p(j)}$.

В некоторых случаях точки и векторы необходимо преобразовывать в противоположном направлении — из *родительского пространства* в пространство *дочернего* сустава. Такое преобразование является обратным локальному положению сустава. Математически это выглядит так: $\mathbf{P}_{p(j)\to j} = \left(\mathbf{P}_{j\to p(j)}\right)^{-1}$.

12.3.3. Глобальные позы

Иногда положение сустава удобнее выразить в пространстве модели или игрового мира. Это называется *глобальным положением или позой*. Некоторые движки выражают глобальные позы в виде матрицы, другие — в формате SRT.

В математическом смысле положение сустава ($j \to \mathrm{M}$) в пространстве модели можно получить, двигаясь по скелетной иерархии от заданного сустава до самого корня, умножая при этом локальные позы ($j \to p(j)$). Рассмотрим иерархию, представленную на рис. 12.9. Родительское пространство корневого сустава определяется в качестве пространства модели, поэтому $p(0) \equiv \mathrm{M}$. Таким образом, положение сустава J_2 в пространстве модели можно записать так:

$$\mathbf{P}_{2\to\mathrm{M}} = \mathbf{P}_{2\to1}\mathbf{P}_{1\to0}\mathbf{P}_{0\to\mathrm{M}}.$$

Аналогично положение сустава J_5 в пространстве модели — это просто:

$$\mathbf{P}_{5\to M} = \mathbf{P}_{5\to 4}\mathbf{P}_{4\to 3}\mathbf{P}_{3\to 0}\mathbf{P}_{0\to M}.$$

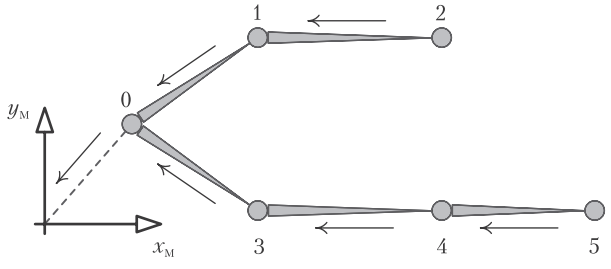


Рис. 12.9. Чтобы вычислить глобальное положение, можно пройтись по иерархии от заданного сустава к корню и основанию пространства модели, объединяя в процессе (локальные) преобразования «потомок — родитель» для каждого сустава

В целом глобальное положение (преобразование «сустав — модель») любого сустава j можно представить так:

$$\mathbf{P}_{j\to M} = \prod_{i=j}^{0} \mathbf{P}_{i\to p(i)}. \tag{12.1}$$

Здесь подразумевается, что после каждой итерации i в результате становится $p(i)$, а $p(0) \equiv M$.

Представление глобального положения в памяти

Мы можем расширить структуру данных `SkeletonPose`, добавив в нее глобальное положение, как показано далее. Опять динамически выделяем массив `m_aGlobalPose` в зависимости от количества суставов в скелете:

```
struct SkeletonPose
{
    Skeleton* m_pSkeleton;     // скелет + количество суставов
    JointPose* m_aLocalPose;   // локальные положения суставов
    Matrix44*  m_aGlobalPose;  // глобальные положения суставов
};
```

12.4. Клипы

В анимационных фильмах еще до начала съемок тщательно планируются все аспекты каждой сцены — движения всех персонажей и неодушевленных предметов и даже перемещения камеры. Это означает, что всю сцену можно анимировать

в виде одной длинной непрерывной последовательности кадров. Когда персонажи находятся вне области видимости камеры, их не нужно анимировать.

В игровой анимации все иначе. Игры интерактивны, поэтому мы не можем предугадать заранее, как будут двигаться и вести себя персонажи. Игрок имеет полную власть над своим персонажем и обычно над камерой (хотя бы частично). Даже решения неигровых персонажей, управляемых компьютером, сильно зависят от непредсказуемых действий игрока. В связи с этим игровая анимация почти никогда не представляет собой длинную непрерывную последовательность кадров. Вместо этого перемещения персонажа приходится разбивать на большое количество мелких движений, которые мы называем *анимационными клипами* или просто *анимациями*.

Каждый клип заставляет персонажа выполнить одно четко определенное действие. Некоторые клипы делаются циклическими, например цикл ходьбы или бега. Бывают клипы, которые воспроизводятся лишь один раз, например бросание объекта или спотыкание и падение на землю. Одни клипы охватывают все тело персонажа (прыжок вверх), а другие относятся лишь к какой-то его части (допустим, он машет правой рукой). Движения любого персонажа игры обычно разбиваются буквально на тысячи клипов.

Единственным исключением из этого правила является ситуация, когда персонажи используются в неинтерактивной части игры, известной как *IGC* (in-game cinematic — «внутриигровое видео»), *NIS* (noninteractive sequence — «неинтерактивные сцены») или *FMV* (full-motion video — «полностью подвижное видео»). Неинтерактивные сцены обычно используются для показа сюжетных элементов, которые не очень хорошо вписываются в игровой процесс. Они создаются таким же образом, как и фильмы на основе компьютерной анимации, хотя в них часто задействуются внутриигровые ресурсы, такие как меши, скелеты и текстуры персонажей. Термины IGC и NIS обычно относятся к неинтерактивным сценам, которые отрисовываются самим игровым движком в реальном времени. Термин FMV обозначает заранее сгенерированные сцены в MP4, WMV или другом видеоформате, которые воспроизводит видеоплеер движка во время игры в полноэкранном режиме.

QTE (quick time event — «быстрое событие») — это полуинтерактивная разновидность такого рода анимации. В QTE игрок должен нажать кнопку в нужный момент неинтерактивной сцены, чтобы увидеть успешную анимацию и продолжить игру. В противном случае воспроизводится анимация неудачи и игроку приходится повторять попытку с возможной потерей жизни или другими отрицательными последствиями.

12.4.1. Локальная временная шкала

Можно считать, что каждый анимационный клип имеет локальную временную последовательность, которую обычно обозначают переменной t. В начале клипа

$t = 0$, а в конце — $t = T$, где T — продолжительность клипа. Каждое уникальное значение переменной t называется *временным индексом*. Пример показан на рис. 12.10.

Рис. 12.10. Локальная временная шкала анимации с разными позами в выбранных временных индексах (снимок экрана с сайта https://beedge.neocities.org)

Интерполяция поз и непрерывное время

Важно понимать, что частота, с которой кадры выводятся на экране, не обязательно должна совпадать с частотой, с которой аниматор создает позы. В кинематографической и игровой анимации позы почти никогда не меняются каждые 1/30 или 1/60 секунды. Вместо этого аниматор генерирует так называемые *ключевые позы* или *ключевые кадры* в определенные моменты клипа, а компьютер вычисляет промежуточное положение персонажа с помощью линейной или нелинейной интерполяции (рис. 12.11).

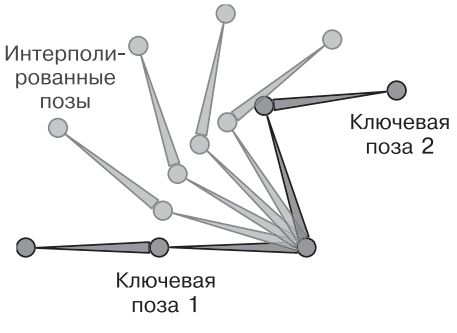


Рис. 12.11. Аниматор создает довольно небольшое количество ключевых поз, а движок генерирует промежуточные позы с помощью интерполяции

Благодаря тому, что движок умеет *интерполировать* позы (о чем мы подробнее поговорим позже в этой главе), мы можем получить позу персонажа в *любой момент* клипа, а не только в кадрах с целочисленными индексами. Иными словами, временная шкала клипа является *непрерывной*. В компьютерной анимации переменная времени t имеет не *целое*, а *вещественное* значение (с плавающей запятой).

Кинематографическая анимация не использует весь потенциал непрерывной природы своей временной шкалы, так как имеет строго фиксированную частоту смены кадров в секунду: 24, 30 или 60. В фильме зритель видит позы персонажа в кадрах 1, 2, 3 и т. д. — необходимости вычислять позу в кадре 3,7 попросту нет. Поэтому аниматор не уделяет особого, да и вообще какого-либо внимания тому, как выглядит персонаж между кадрами с целыми индексами.

Для сравнения: в игре, которая отрисовывается в реальном времени, частота кадров всегда немного колеблется в зависимости от текущей нагрузки на процессор

и видеокарту. К тому же течение времени в игровой анимации может меняться, чтобы, например, ускорить или замедлить движения персонажа. Поэтому в интерактивной игре анимационные клипы почти *никогда* не разбиваются по кадрам с целыми номерами. Теоретически, если скорость воспроизведения равна 1,0, клип должен состоять из кадров 1, 2, 3 и т. д. Но на практике игрок может видеть кадры 1,1, 1,9, 3,2 и т. д. А если течение времени замедлить до 0,5, игрок может увидеть кадры 1,1, 1,4, 1,9, 2,6, 3,2 и т. д. Мы можем даже повернуть время вспять, чтобы воспроизвести анимацию задом наперед. Таким образом, в игровой анимации время является как *непрерывным*, так и *непостоянным*.

Единицы времени

Поскольку анимация имеет непрерывную временную шкалу, время лучше измерять в секундах. Если продолжительность кадра определена заранее, то для этого можно использовать такие единицы, как *кадры*. Обычно в игровой анимации кадр длится от 1/30 до 1/60 секунды. Однако здесь важно не ошибиться, сделав переменную времени t целочисленной, иначе она будет учитывать только целые кадры. Вне зависимости от выбранных единиц времени значение t должно быть вещественным: числом с фиксированной запятой или целым числом, измеряющим очень маленькие интервалы времени — значительно меньше одного кадра. Это делается для того, чтобы время измерялось с достаточной точностью для выполнения таких операций, как межкадровая анимация или изменение скорости воспроизведения.

Кадры и семплы

К сожалению, в игровой индустрии термин «*кадр*» имеет несколько значений. Это может вызвать большую путаницу. Иногда кадром называют *отрезок времени* длиной 1/30 или 1/60 секунды. Но в других контекстах это может быть *отдельный момент времени* (например, когда речь идет о позе персонажа в кадре 42).

Лично я предпочитаю для описания отдельного момента времени использовать термин «*семпл*», а отрезки времени длиной 1/30 или 1/60 секунды называю *кадрами*. Например, односекундная анимация, созданная с частотой 30 кадров в секунду, содержала бы 31 *семпл* и занимала 30 *кадров* (рис. 12.12). Термин «семпл» позаимствован из области обработки сигналов. Сигнал непрерывного времени, то есть функцию $f(t)$, можно преобразовать в набор дискретных точек данных, разбив его на временные отрезки одинаковой длины. Этот процесс называют *дискретизацией*, подробнее о нем рассказывается в подразделе 14.3.2.

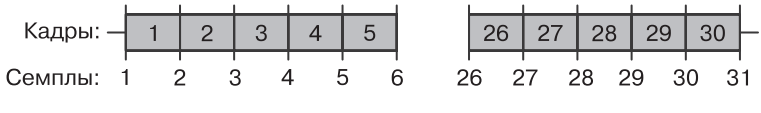


Рис. 12.12. Анимация длиной 1 с делится на 30 кадров и содержит 31 семпл

Кадры, семплы и циклические клипы

Клип, который должен воспроизводиться снова и снова, называется *циклическим*. Если взять две копии односекундного клипа (30 кадров, 31 семпл) и состыковать конец одной с началом другой, семпл 31 первого клипа будет точно совпадать с семплом 1 второго (рис. 12.13). Чтобы клип был как следует зациклен, конечная поза персонажа должна быть идентична находящейся в самом начале. Это, в свою очередь, означает, что последний семпл циклического клипа (в нашем примере это семпл 31) является избыточным, поэтому многие игровые движки его пропускают.

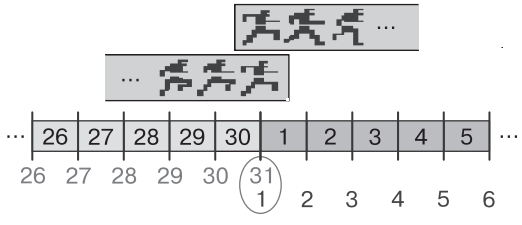


Рис. 12.13. Последний семпл циклического клипа совпадает с его первым семплом и, следовательно, является избыточным

Из этого вытекают два правила, определяющие количество семплов и кадров в любом анимационном клипе.

- Если клип *не циклический*, анимация из N кадров будет иметь $N + 1$ уникальных семплов.
- Если клип *циклический*, последний семпл лишний, поэтому анимация из N кадров будет иметь N уникальных семплов.

Нормализованное время (фаза)

Иногда бывает удобно использовать единицу нормализованного времени u, в этом случае $u = 0$ в начале анимации и $u = 1$ в конце анимации, независимо от ее продолжительности T. Нормализованное время иногда называют *фазой* анимационного клипа, поскольку u ведет себя как фаза синусоиды, если анимация зациклена (рис. 12.14).

Нормализованное время может пригодиться при синхронизации двух или более анимационных клипов с различной абсолютной продолжительностью. Например, мы можем плавно перейти от двухсекундного бегового цикла (60 кадров) к трехсекундному циклу ходьбы (90 кадров). Чтобы этот переход выглядел естественно, нужно убедиться в том, что оба клипа постоянно синхронизированы и положение ног идеально совпадает.

Рис. 12.14. Анимационный клип с единицами нормализованного времени

Мы можем этого добиться, подогнав нормализованное время начала ходьбы u_{walk} под индекс нормализованного времени бега u_{run}. Затем будем воспроизводить оба клипа с нормализованной частотой, чтобы они оставались синхронизированными. Это существенно проще и надежней, чем синхронизация с использованием индексов абсолютного времени t_{walk} и t_{run}.

12.4.2. Глобальная временная шкала

У каждого персонажа в игре, как и у любого анимационного клипа, есть временная шкала, только не локальная, которая отсчитывается с 0 в начале клипа, а глобальная, которая начинается, когда персонаж впервые появляется в игровом мире либо с началом уровня или самой игры. В этой книге для измерения глобального времени будем использовать переменную, обозначенную τ, чтобы не путать ее с локальным временем t.

Воспроизведение анимации можно представить как *привязывание* локальной временной шкалы клипа к глобальной временной шкале персонажа. Например, на рис. 12.15 проиллюстрировано воспроизведение анимационного клипа A начиная с глобального времени $\tau_{start} = 102$ с.

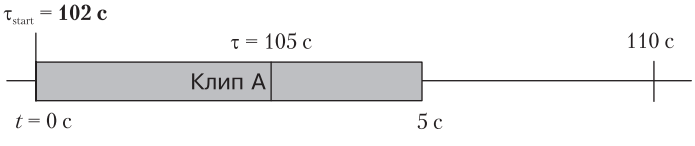


Рис. 12.15. Воспроизведение анимационного клипа A начиная с 102-й секунды глобального времени

Как мы видели ранее, воспроизведение циклической анимации похоже на выкладывание бесконечного количества копий клипа на глобальной временной шкале так, чтобы начало стыковалось с концом. Воспроизведение циклической анимации ограниченное количество раз можно изобразить как выкладывание определенного количества копий клипа (рис. 12.16).

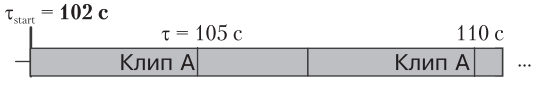


Рис. 12.16. Воспроизведение циклической анимации, представленное в виде стыковки множества копий клипа

Масштабирование клипа *по времени* ускоряет или замедляет его воспроизведение по сравнению с тем, как он был анимирован изначально. Чтобы этого добиться, мы просто изменяем размер изображения клипа, наложенного на глобальную временную шкалу. Этот процесс естественнее всего представить в виде *скоро-*

сти воспроизведения, которую обозначим *R*. Например, если анимация должна воспроизводиться с удвоенной скоростью ($R = 2$), локальную временную шкалу нужно уменьшить в два раза ($1 / R = 0{,}5$) перед привязкой к глобальной шкале (рис. 12.17).

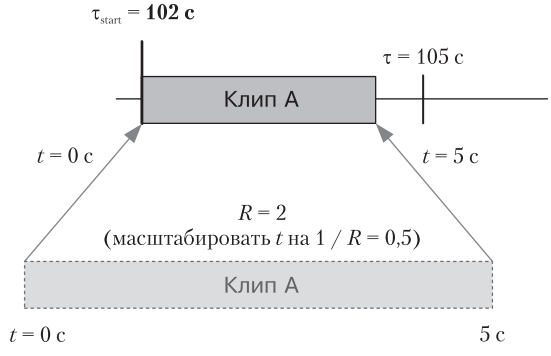


Рис. 12.17. Воспроизведение анимации с удвоенной скоростью соответствует масштабированию его локального времени с коэффициентом 0,5

Воспроизведение клипа задом наперед соответствует использованию временной шкалы размером –1 (рис. 12.18).

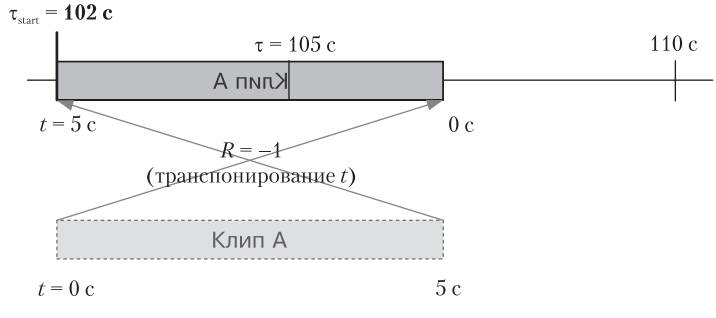


Рис. 12.18. Воспроизведение клипа от конца к началу соответствует временной шкале размером –1

Чтобы привязать анимационный клип к глобальной временной шкале, нужно знать о нем следующее:

- глобальное время начала τ_{start};
- скорость воспроизведения *R*;
- продолжительность *T*;
- сколько раз его нужно повторить (обозначим это количество *N*).

Располагая данной информацией, мы можем привязать любое глобальное время τ к соответствующему локальному времени t и наоборот. Это можно сделать с помощью следующих двух уравнений:

$$t = (\tau - \tau_{\text{start}})R; \tag{12.2}$$

$$\tau = \tau_{\text{start}} + \frac{1}{R}t.$$

Если анимация не циклическая ($N = 1$), мы должны втиснуть значение t в подходящий диапазон $[0, T]$, прежде чем использовать его для получения семпла из клипа:

$$t = \text{clamp}\left[(\tau - \tau_{\text{start}})R\right]_0^T.$$

Если анимация находится в бесконечном цикле ($N = \infty$), для размещения t в корректном диапазоне нужно взять остаток от деления результата на продолжительность T. Чтобы этого достичь, используется оператор `modulo` (`mod` или `%` в C/C++), как показано далее:

$$t = \left((\tau - \tau_{\text{start}})R\right) \bmod T.$$

Если клип повторяется *конечное* число раз ($1 < N < \infty$), сначала нужно втиснуть t в диапазон $[0, NT]$ а затем взять остаток от деления *этого* результата на T. Это позволит получить подходящий диапазон для дискретизации клипа:

$$t = \left(\text{clamp}\left[(\tau - \tau_{\text{start}})R\right]_0^{NT}\right) \bmod T.$$

Большинство игровых движков работают непосредственно с локальными шкалами анимации и не используют глобальную шкалу напрямую. Однако прямое применение глобального времени дает крайне полезные преимущества. Например, делает синхронизацию клипов тривиальной задачей.

12.4.3. Сравнение глобального и локального таймеров

Система анимации должна отслеживать временные индексы всех активных клипов. Для этого у нас есть два варианта.

- *Локальный таймер*. У каждого клипа есть собственный локальный таймер, который обычно представлен временным индексом с плавающей запятой и измеряется в секундах, кадрах или нормализованных единицах времени (в последнем случае его часто называют *фазой* анимации). В момент, когда клип начинает проигрываться, индекс времени t, как правило, принимается за ноль. Чтобы анимация продвигалась вперед во времени, мы инкрементируем локальные таймеры каждого клипа в отдельности. Если клип воспроизводится со скоростью R, не равной единице, степень инкрементации локального таймера нужно масштабировать с коэффициентом R.

- *Глобальный таймер*. У персонажа есть глобальный таймер, который обычно отсчитывает секунды, и каждый клип записывает глобальное время τ_{start}, когда начинает воспроизводиться. Значения локальных таймеров клипов *вычисляются* на основе этой информации с помощью уравнения (12.2).

Выгодной стороной метода с локальным таймером является его простота, это наиболее очевидный выбор при проектировании системы анимации. Однако у применения глобального таймера есть явные преимущества, особенно в сфере синхронизации клипов в контексте одного или нескольких персонажей в сцене.

Синхронизация анимационных клипов с помощью локального таймера

Как уже отмечалось, если использовать локальный таймер, начало локальной временной шкалы клипа ($t = 0$) обычно подгоняется под момент, когда он начинает воспроизводиться. Поэтому, чтобы синхронизировать два клипа или больше, мы должны воспроизвести их в один и тот же момент игрового времени. Эта на первый взгляд простая задача может заметно усложниться, если команды для запуска анимации приходят из разных подсистем движка.

Представьте, к примеру, что мы хотим синхронизировать анимацию удара, который наносит персонаж игрока, с анимацией ответной реакции неигрового персонажа. Проблема в том, что удар инициирован подсистемой игрока в ответ на нажатие кнопки на джойстике, в то время как анимация реакции неигрового персонажа (NPC) на удар воспроизводится подсистемой искусственного интеллекта (ИИ). Если код ИИ в игровом цикле выполняется *перед* кодом игрока, между ударом и реакцией возникнет отставание в один кадр. Если код игрока выполняется раньше, чем код ИИ, возникнет обратная проблема, когда NPC попытается ударить игрока. Если для взаимодействия между подсистемами используется механизм обмена сообщениями (событиями), может появиться дополнительная задержка (подробнее об этом — в разделе 16.8). Проблема проиллюстрирована на рис. 12.19.

```
void GameLoop()
{
    while (!quit)
    {
        // предварительные обновления...
        UpdateAllNpcs(); // реагируем на событие удара из последнего кадра

        // еще обновления...

        UpdatePlayer(); // нажата кнопка удара – запускаем анимацию удара
                        // и передаем событие NPC, чтобы тот среагировал
                        // еще больше обновлений...
    }
}
```

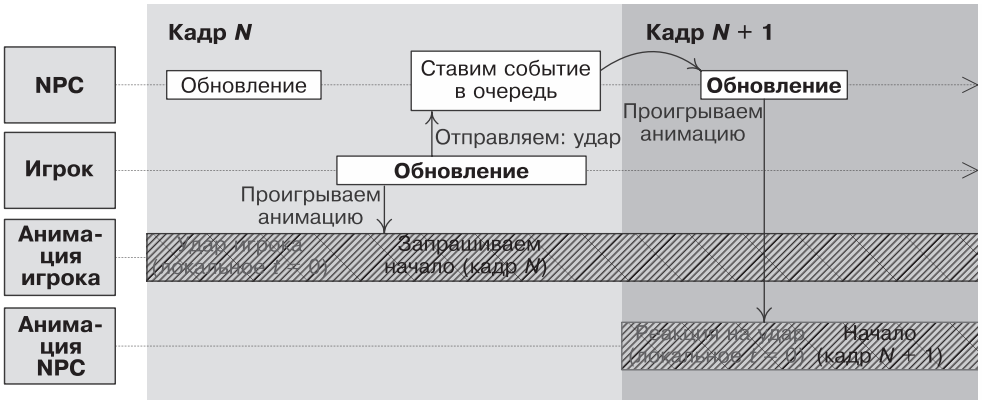


Рис. 12.19. Порядок выполнения разных систем игрового процесса может создать проблемы с синхронизацией анимации, если используются локальные таймеры

Синхронизация анимационных клипов с помощью глобального таймера

Использование глобального таймера позволяет смягчить многие проблемы, связанные с синхронизацией, поскольку начальное время ($\tau = 0$) по определению одинаково для всех клипов. Если глобальное начальное время совпадает для двух и более клипов, анимация на момент старта будет идеально синхронизированной. Если их скорость воспроизведения тоже совпадает, они останутся синхронизированными без смещения. Теперь нам неважно, *когда* выполняется код, инициирующий каждый клип. Даже если код ИИ, который воспроизводит реакцию на удар, отстанет на один кадр от кода удара игрока, для синхронизации этих двух клипов достаточно будет взять глобальное время начала удара и сделать его глобальным временем начала анимации удара (рис. 12.20).

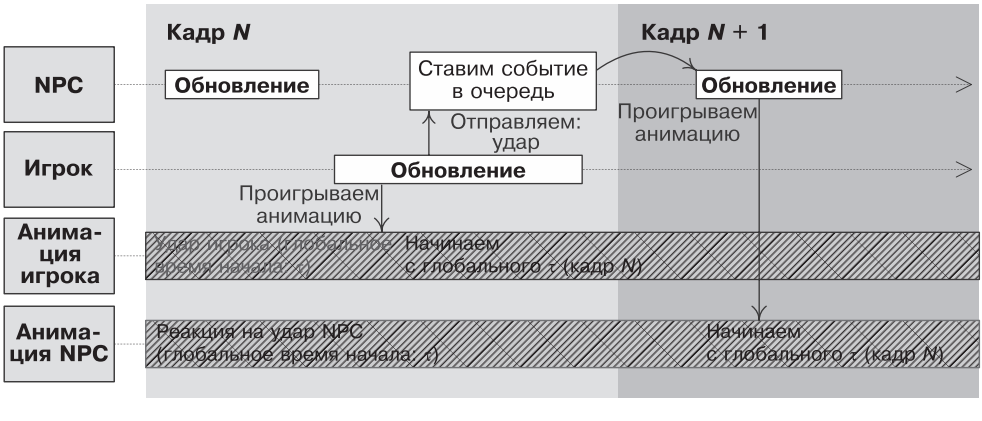


Рис. 12.20. Синхронизация анимационных клипов с помощью глобального таймера

Конечно, нужно следить за тем, чтобы глобальные таймеры двух персонажей совпадали, но это просто. Мы можем либо отрегулировать начальное глобальное время с учетом расхождений в таймерах персонажей, либо сделать так, чтобы у всех персонажей в игре был общий таймер.

12.4.4. Простой формат анимационных данных

Обычно для извлечения анимационных данных из файла сцены Maya дискретизируется поза скелета с частотой 30 или 60 семплов в секунду. Семпл содержит полноценную позу каждого сустава в скелете. Позы обычно хранятся в формате SRT: для каждого сустава *j* коэффициент масштабирования представляет собой либо единое скалярное значение S_{jx} с плавающей запятой, либо трехэлементный вектор $\mathbf{S}_j = \left[S_{jx} \; S_{jy} \; S_{jz} \right]$. Коэффициентом вращения, конечно, выступает четырехэлементный кватернион $\mathrm{Q}_j = \left[Q_{jx} \; Q_{jy} \; Q_{jz} \; Q_{jw} \right]$. Коэффициент смещения записывается как $\mathbf{T}_j = \left[T_{jx} \; T_{jy} \; T_{jz} \right]$. Иногда говорят, что анимация каждого сустава содержит до десяти *каналов*, имея в виду десять элементов $\mathbf{S}_j$, Q_j и $\mathbf{T}_j$ (рис. 12.21).

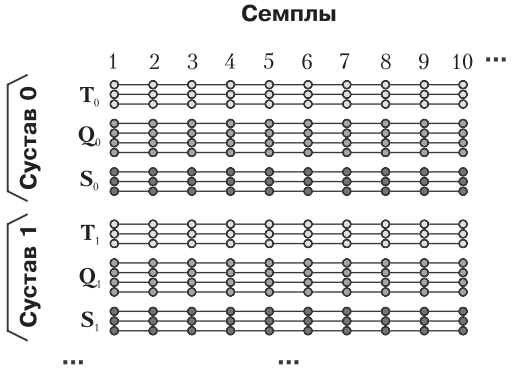


Рис. 12.21. Несжатый анимационный клип содержит десять каналов данных с плавающей запятой в каждом семпле и для каждого сустава

В C++ анимационный клип можно представить множеством разных способов. Вот один из вариантов:

```
struct JointPose { ... }; // формат SRT, определенный так, как показано ранее

struct AnimationSample
{
    JointPose* m_aJointPose;           // массив положений суставов
};

struct AnimationClip
{
       Skeleton*        m_pSkeleton;
       F32              m_framesPerSecond;
       U32              m_frameCount;
       AnimationSample* m_aSamples;       // массив семплов
    bool             m_isLooping;
};
```

Анимационный клип создается для определенного скелета и обычно подходит только для него. В связи с этим структура данных `AnimationClip` содержит ссылку на свой скелет `m_pSkeleton` (в настоящем игровом движке вместо указателя `Skeleton*` может использоваться уникальный ID скелета, в данном случае предполагается, что движок позволяет быстро и удобно искать скелеты по их уникальным идентификаторам).

Мы исходим из того, что количество поз `JointPose` в массиве `m_aJointPose` каждого семпла совпадает с количеством суставов в скелете. Число семплов в массиве `m_aSamples` зависит от того, сколько у нас кадров и должен ли клип быть циклическим. Если анимация не зациклена, количество семплов вычисляется как `(m_frameCount + 1)`. В противном случае оно равно `m_frameCount`, так как последний семпл идентичен первому и обычно опускается.

Важно понимать, что в настоящем игровом движке анимационные данные не хранятся в таком упрощенном формате. Как мы увидим в разделе 12.8, они обычно *сжимаются* различными способами для экономии памяти.

12.4.5. Непрерывные канальные функции

Семплы анимационного клипа являются лишь определениями непрерывных функций от времени. Можно считать, что для каждого сустава предусмотрены десять функций от времени со скалярными значениями или две функции с векторными значениями и одна с кватернионным. Как показано на рис. 12.22, в теории эти *канальные функции* являются плавными и непрерывными на протяжении всей локальной временной шкалы клипа, за исключением явно заданных разрывов, таких как переключения между камерами. Но на практике многие игровые движки выполняют *линейную* интерполяцию между семплами, и в этом случае используются *кусочно-линейные приближения* исходных непрерывных функций (рис. 12.23).

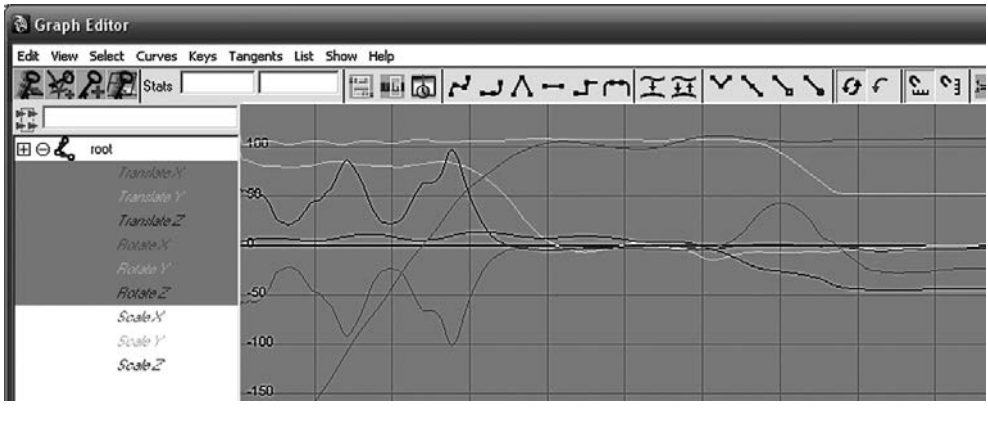


Рис. 12.22. Анимационные семплы в клипе определяют непрерывные функции от времени

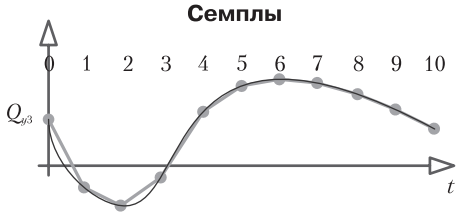


Рис. 12.23. Многие игровые движки используют кусочно-линейное приближение при интерполяции канальных функций

12.4.6. Метаканалы

Многие игры позволяют определять для анимации дополнительные метаканалы данных. Эти каналы могут кодировать игровую информацию, которая не имеет прямого отношения к позиционированию скелета, но должна синхронизироваться с анимацией.

Довольно часто разработчики создают специальный канал, который содержит *триггеры событий* в различные моменты времени (рис. 12.24). Каждый раз, когда индекс локального времени анимации проходит один из этих триггеров, игровому движку отправляется *событие*, на которое тот может как-то среагировать (мы подробно обсудим события в главе 16). С помощью триггеров событий часто отмечают моменты анимации, когда нужно воспроизвести определенный звук или эффект частиц. Например, когда правая или левая нога касается земли, можно инициировать соответствующий звук и отобразить облачко пыли.

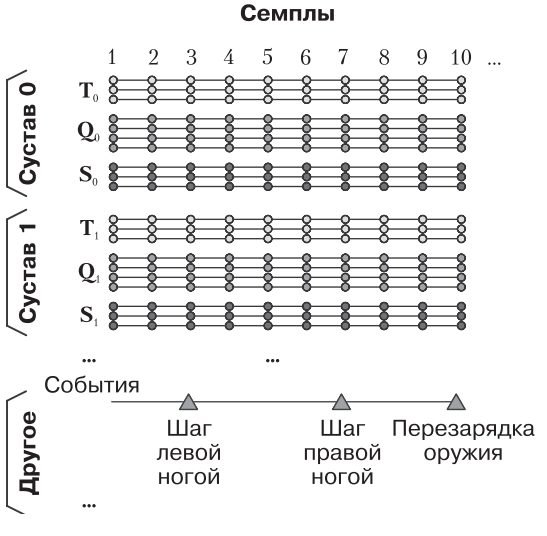


Рис. 12.24. Чтобы синхронизировать с анимацией аудиоэффекты, эффекты частиц и другие события игры, в анимационный клип можно добавить специальный канал с триггерами событий

Еще один распространенный подход состоит в том, чтобы позволить анимировать специальные суставы, известные в Maya как *локаторы*, вместе с суставами скелета. Поскольку сустав или локатор — это всего лишь аффинное преобразование, с помощью специальных суставов можно закодировать позицию и положение в пространстве практически любого объекта в игре.

Типичное применение анимированных локаторов заключается в определении того, как во время анимации должна размещаться игровая камера. В Maya локатор привязывается к камере, а сама она анимируется вместе с суставами персонажа (-ей) в сцене. Этот локатор экспортируется и используется в игре для перемещения камеры во время анимации. Поле зрения (фокусное расстояние) и, возможно, другие ее атрибуты тоже можно анимировать, сохраняя соответствующие данные в один или несколько дополнительных *каналов с плавающей запятой*.

Далее перечислены другие примеры каналов анимации, не относящихся к суставам:

- прокрутка координат текстур;
- анимация текстур (частный случай прокрутки координат текстур, когда кадры размещаются линейно внутри текстуры, которая затем прокручивается на один кадр в каждой итерации);
- анимированные параметры материалов (цвет, зеркальность, прозрачность и т. д.);
- анимированные параметры освещения (радиус, угол раствора конуса, интенсивность, цвет и т. д.);
- любые другие параметры, которые должны меняться со временем и каким-то образом синхронизироваться с анимацией.

12.4.7. Отношения между мешами, скелетами и клипами

UML-диаграмма на рис. 12.25 демонстрирует взаимодействие данных анимационного клипа со скелетами, позами, мешами и другой информацией в игровом движке. Обратите особое внимание на *количество элементов* и *направление* связей между этими классами. Количество элементов указано рядом с одним из концов стрелки, соединяющей классы: единица обозначает один экземпляр класса, а звездочка — много экземпляров. Любой отдельно взятый тип персонажа имеет один скелет, один или несколько мешей и один или несколько анимационных клипов. Скелет является центральным объединяющим элементом. Скины, которые к нему крепятся, не имеют никакого отношения к клипам. Точно так же клипы предназначены для определенного скелета, но им ничего не известно о мешах скина. Эти связи проиллюстрированы на рис. 12.26.

Игровые дизайнеры часто пытаются свести к минимуму количество уникальных скелетов в игре, так как для каждого скелета, как правило, требуется новый полный набор анимационных клипов. Чтобы все выглядело так, будто игра наполнена множеством разных персонажей, для каждого скелета по возможности следует предусмотреть несколько мешей. Это позволит всем персонажам использовать общий набор анимаций.

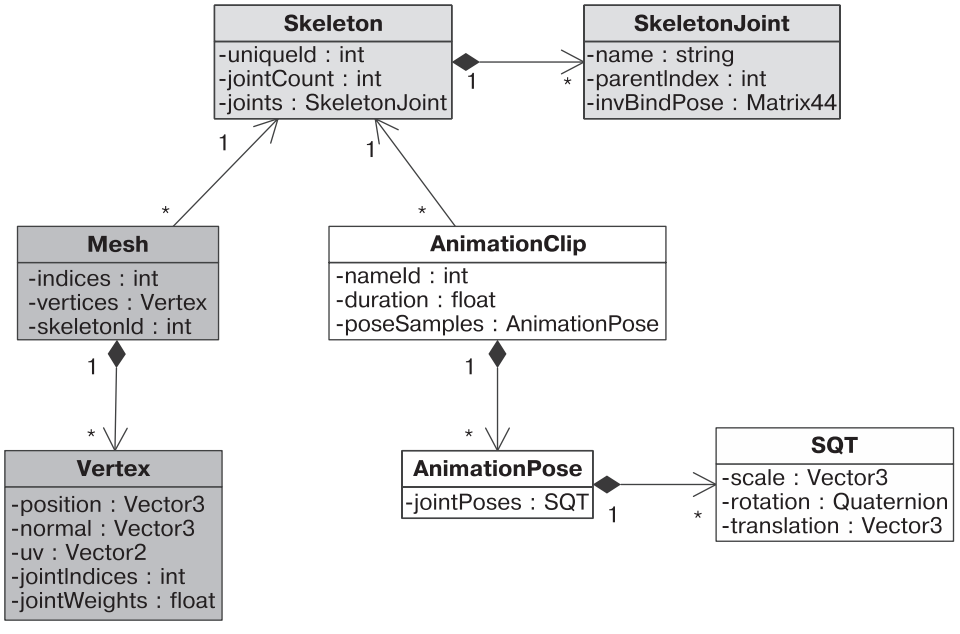


Рис. 12.25. UML-диаграмма разделяемых ресурсов анимации

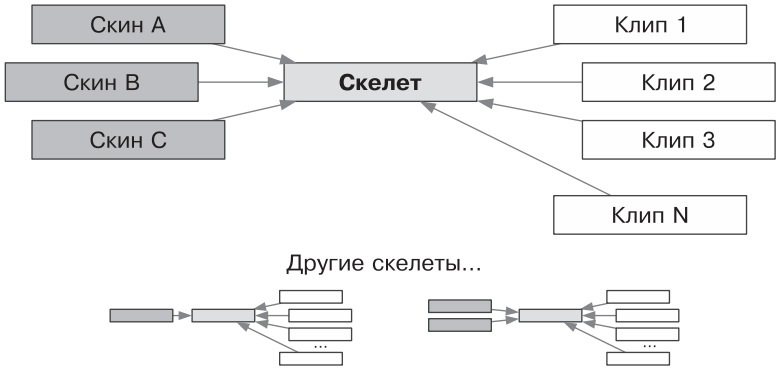


Рис. 12.26. Много анимационных клипов и один или несколько мешей, предназначенных для одного скелета

Перенос анимации

Ранее упоминалось, что анимация обычно совместима лишь с одним скелетом. Это ограничение можно преодолеть с помощью методик *переноса анимации* (animation retargeting).

Под переносом понимается анимирование скелета с помощью клипа, предназначенного для другого скелета. Если два скелета морфологически идентичны,

перенос может свестись к простому переназначению индексов суставов. Но если идеального совпадения нет, процесс усложняется. В Naughty Dog аниматоры определяют специальную *позу переноса*. В ней зафиксированы основные различия между позами привязки оригинального и нового скелетов, что позволяет системам переноса корректировать исходные позы на этапе выполнения, чтобы они выглядели более естественно для нового персонажа.

Существуют и более продвинутые методики для переноса анимации между разными скелетами. Подробнее об этом можно почитать в статьях Людовика Дютрива с соавторами *Feature Points Based Facial Animation Retargeting* (https://bit.ly/2HL9Cdr) и Криса Хекера *Real-time Motion Retargeting to Highly Varied User-Created Morphologies* (https://bit.ly/2vviG3x).

12.5. Скининг и генерация палитры матриц

Мы уже видели, как позиционировать скелет путем вращения, смещения и, возможно, масштабирования его суставов. И знаем, что любую позу скелета математически можно представить в виде набора локальных ($\mathbf{P}_{j \to p(j)}$) или глобальных ($\mathbf{P}_{j \to \mathrm{M}}$) преобразований поз, по одному для каждого сустава j. Пришло время исследовать процесс привязки вершин трехмерного меша к позиционированному скелету, известный как *скининг*.

12.5.1. Сведения о скининге для отдельной вершины

Мы используем вершины меша скина, чтобы прикрепить его к скелету. Каждая вершина может быть *привязана* к одному или нескольким суставам. В первом случае вершина точно повторяет движение этого сустава, во втором позиция вершины становится *взвешенным средним* позиций, которые она приняла бы в случае привязки к каждому суставу по отдельности.

Чтобы наложить меш на скелет, 3D-художник должен предоставить следующую дополнительную информацию о каждой из вершин:

- *индекс* или *индексы* суставов, к которым она привязана;
- *весовой коэффициент* каждого сустава, определяющий его влияние на итоговую позицию вершины.

Подразумевается, что сумма весовых коэффициентов равна единице, как при вычислении любого взвешенного среднего.

Обычно игровой движок ограничивает максимальное количество суставов, к которым можно прикрепить одну вершину. Обычно этот лимит равен четырем суставам, и тому есть целый ряд причин. Прежде всего, восьмибитные индексы суставов можно упаковать в 32-битное слово, что довольно удобно. Кроме того, большинство людей не могут увидеть разницу, когда количество суставов для одной вершины превышает четыре, тогда как при двух, трех и четырех суставах различия довольно очевидны.

Поскольку весовые коэффициенты суставов в сумме должны давать единицу, последний из них можно опустить, многие так и делают (его можно вычислить во время выполнения: $w_3 = 1 - (w_0 + w_1 + w_2)$). Таким образом, типичная структура данных с привязанными вершинами может выглядеть так:

```
struct SkinnedVertex
{
    float  m_position[3];    // (Px, Py, Pz)
    float  m_normal[3];      // (Nx, Ny, Nz)
    float  m_u, m_v;         // координаты текстуры (u,v)
    U8     m_jointIndex[4];  // индексы суставов
    float  m_jointWeight[3]; // веса суставов (последний вес опускается)
};
```

12.5.2. Математический аспект скининга

Вершины меша скина повторяют движения сустава или суставов, к которым привязаны. Чтобы описать этот процесс математически, нужно подобрать матрицу, которая перемещает вершины меша с исходной позиции (поза привязки) на новую, соответствующую текущей позе скелета. Мы будем называть ее *матрицей скининга*.

Вершины меша до привязки и после нее указываются в пространстве модели независимо от того, находится ли скелет в позе привязки. Поэтому преобразование между позой привязки и текущей позой, выполняемое искомой матрицей, происходит в пространстве модели. В отличие от других преобразований, которые мы видели до сих пор (например, между пространствами модели и игрового мира или между пространствами игрового мира и просмотра), матрица скининга *не* приводит к изменению основания. Она перемещает вершины на новое место, но при этом они всегда находятся в пространстве модели — как до преобразования, так и после.

Простой пример: скелет с одним суставом

Давайте выведем простое уравнение для матрицы скининга. Чтобы не усложнять раньше времени, возьмем скелет с одним суставом. Таким образом, мы будем работать с двумя системами координат: пространством модели и пространством сустава, которые обозначим нижними индексами M и J соответственно. Оси координат сустава начинаются в позе привязки (пусть ее обозначает верхний индекс B). Во время анимации оси сустава изменяют позицию и направление в пространстве модели — обозначим *текущую позу* верхним индексом C.

Теперь представим, что к суставу привязана одна вершина. В позиции привязки ее позиция в пространстве модели выглядит как $\mathbf{v}_M^B$. В ходе скининга вычисляется новая позиция пространства модели в текущей позе, $\mathbf{v}_M^C$ (рис. 12.27).

Ключевой аспект поиска матрицы скининга для заданного сустава состоит в понимании того, что позиция вершины, привязанной к суставу, будет *постоянной*, если выразить ее в *пространстве координат этого сустава*. Таким образом, мы переводим позицию вершины в позе привязки пространства модели в пространство сустава, перемещаем сустав в текущую позу и, наконец, возвращаем вершину

в пространство модели. Итогом переходов из пространства модели в пространство сустава и обратно будет перемещение вершины из позы привязки в текущую позу.

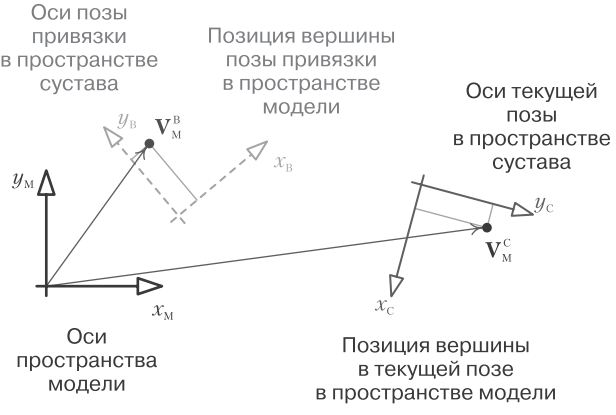


Рис. 12.27. Поза привязки и текущая поза простого скелета с одним суставом, к которому прикреплена одна вершина

Обратимся к рис. 12.28 и предположим, что вершина $\mathbf{v}_{\text{M}}^{\text{B}}$ в пространстве модели имеет координаты (4, 6) (скелет находится в позе связывания). Переводим эту вершину в аналогичную позицию в пространстве сустава, координаты которой $\mathbf{v}_j$, как как показано на диаграмме, — примерно (1, 3). Поскольку вершина привязана к суставу, в его пространстве ее координаты *всегда* остаются (1, 3), независимо от того, как он двигается. Когда сустав переходит в нужную текущую позицию, мы преобразуем координаты вершины обратно в пространство модели (обозначим их $\mathbf{v}_{\text{M}}^{\text{C}}$). На диаграмме они равны примерно (18, 2). Таким образом, преобразование скининга переместило вершину в пространстве модели из (4, 6) в (18, 2), что явилось прямым результатом перехода сустава из положения привязки в текущее положение, показанное на диаграмме.

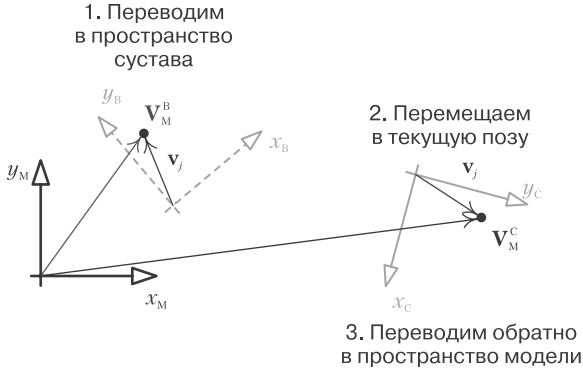


Рис. 12.28. Переведя позицию вершины в пространство сустава, мы можем использовать ее, чтобы отслеживать перемещения этого сустава

Если рассматривать проблему с математической точки зрения, *положение привязки* сустава *j* в пространстве модели можно представить в виде матрицы $\mathbf{B}_{j\to\mathrm{M}}$. Она преобразует точку или вектор, чьи координаты выражены в пространстве сустава *j*, в эквивалентный набор координат в пространстве модели. Теперь рассмотрим вершину, координаты которой выражены в пространстве модели со скелетом в позе привязки. Чтобы перевести эти координаты в пространство сустава *j*, достаточно умножить их на *обратную* матрицу позы привязки, $\mathbf{B}_{\mathrm{M}\to j} = \left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}$:

$$\mathbf{v}_j = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{B}_{\mathrm{M}\to j} = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}. \tag{12.3}$$

Точно так же можно представить *текущее положение* сустава (то есть любое положение, кроме положения привязки) в виде матрицы $\mathbf{C}_{j\to\mathrm{M}}$. Чтобы перевести значение $\mathbf{v}_j$ из пространства сустава обратно в пространство модели, умножаем его на матрицу текущего положения:

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}}.$$

Если расширить $\mathbf{v}_j$ с помощью уравнения (12.3), получится формула, которая может переместить вершину из позиции в позе привязки в позицию текущей позы:

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}} = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}\mathbf{C}_{j\to\mathrm{M}} = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{K}_j. \tag{12.4}$$

Объединенная матрица $\mathbf{K}_j = \left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}\mathbf{C}_{j\to\mathrm{M}}$ называется *матрицей скининга*.

Переход к скелету с несколькими суставами

В предыдущем примере рассматривался лишь один сустав. Однако выведенные уравнения применимы к любому суставу в любом скелете, который только можно представить, потому что мы сформулировали все с точки зрения глобальных поз, то есть преобразований между пространствами сустава и модели. Таким образом, чтобы вычисления подходили для скелета с несколькими суставами, нужно внести лишь два небольших изменения.

1. С помощью уравнения (12.1) мы должны убедиться в том, что матрицы $\mathbf{B}_{j\to\mathrm{M}}$ и $\mathbf{C}_{j\to\mathrm{M}}$ для заданного сустава вычисляются как следует. $\mathbf{B}_{j\to\mathrm{M}}$ и $\mathbf{C}_{j\to\mathrm{M}}$ — это всего лишь эквиваленты позы привязки и соответственно текущей позы матрицы $\mathbf{P}_{j\to\mathrm{M}}$, которая используется в этом уравнении.
2. Мы должны вычислить массив матриц скининга $\mathbf{K}_j$, по одной для каждого сустава *j*. Этот массив называется *палитрой матриц* и передается движку отрисовки при выводе меша скина. Отрисовщик берет каждую вершину, находит для нее в палитре матрицу скининга подходящего сустава и применяет эту матрицу для перемещения вершины из позы привязки в текущую позу.

Здесь следует отметить, что матрица текущей позы $\mathbf{C}_{j\to\mathrm{M}}$ меняется в каждом кадре по мере того, как персонаж принимает все новые позы. Однако обратная матрица позы привязки остается неизменной на протяжении всей игры, поскольку

поза привязки скелета фиксируется на этапе создания модели. В связи с этим матрицу $\left(\mathbf{B}_{j\to \mathrm{M}}\right)^{-1}$ обычно кэшируют вместе со скелетом и не просчитывают во время выполнения. Анимационные движки, как правило, вычисляют локальные позы для каждого сустава ($\mathbf{C}_{j\to p(j)}$), затем используют уравнение (12.1), чтобы преобразовать их в глобальные позы ($\mathbf{C}_{j\to \mathrm{M}}$), и в конце умножают каждую глобальную позу на соответствующую закэшированную обратную матрицу привязки $\left(\mathbf{B}_{j\to \mathrm{M}}\right)^{-1}$, чтобы сгенерировать матрицу скининга ($\mathbf{K}_j$) для каждого сустава.

Добавление преобразования вида «модель — мир»

Каждую вершину в итоге необходимо перевести из пространства модели в пространство игрового мира. В связи с этим некоторые движки заранее умножают палитру матриц скининга на преобразование объекта вида «модель — мир». Это может оказаться полезной оптимизацией, так как при вычислении скиннинговой геометрии движок отрисовки может пропустить по одной операции умножения для каждой вершины (хорошая экономия, если речь идет об обработке сотен тысяч вершин!).

Чтобы добавить сюда преобразование вида «модель — мир», его достаточно дописать к уравнению матриц скининга:

$$\left(\mathbf{K}_j\right)_{\mathrm{W}} = \left(\mathbf{B}_{j\to \mathrm{M}}\right)^{-1}\mathbf{C}_{j\to \mathrm{M}}\mathbf{M}_{\mathrm{M}\to \mathrm{W}}.$$

Некоторые движки встраивают преобразование вида «модель — мир» в матрицы скининга подобным образом, другие — нет. Это решение принадлежит исключительно команде программистов и зависит от всевозможных факторов. Например, этого точно *не стоит* делать в ситуации, когда одна анимация применяется одновременно к нескольким персонажам, — данная методика, известная как *дублирование анимации*, иногда используется для анимирования большой группы персонажей. В этом случае преобразование вида «модель — мир» следует выполнять отдельно, чтобы все персонажи группы могли брать одну и ту же палитру матриц.

Привязка вершины к нескольким суставам

Если вершина привязана более чем к одному суставу, мы вычисляем ее итоговую позицию так, будто она соединена с каждым суставом по отдельности, затем вычисляем позицию каждого сустава в пространстве модели и в конце берем *среднее взвешенное* значение полученной позиции. Веса указываются создателем персонажа и должны в сумме давать единицу (в противном случае их необходимо заново нормализировать с помощью инструментального конвейера).

Общая формула для среднего взвешенного N величин от a_0 до a_{N-1} с весами от w_0 до w_{N-1} при $\sum w_i = 1$ выглядит так:

$$a = \sum_{i=0}^{N-1} w_i a_i.$$

Это подходит и для векторных величин $\mathbf{a}_i$. Таким образом мы можем расширить уравнение (12.4) для N суставов с индексами от j_0 до j_{N-1} и весами от w_0 до w_{N-1}:

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_{\mathrm{M}}^{\mathrm{B}} \mathbf{K}_{j_i},$$

где $\mathbf{K}_{j_i}$ — матрица скининга для сустава j_i.

12.6. Слияние анимации

Термин «*слияние анимации*» обозначает любую методику, которая позволяет получать итоговую позу персонажа на более чем одном анимационном клипе. Если точнее, слияние объединяет две или более *входные позы* для создания *выходной позы* скелета.

Слияние обычно затрагивает как минимум две позы и генерирует вывод в один и тот же момент времени. В данном контексте слияние генерирует из двух или больше анимаций множество новых клипов, которые иначе пришлось бы создавать вручную. Например, слиянием анимации прихрамывания с анимацией нормальной походки можно получить разные промежуточные степени повреждения, которые персонаж демонстрирует во время ходьбы. Еще один пример — слияние анимации прицеливания вправо и влево: можно сделать так, чтобы персонаж прицеливался под любым углом в пределах между этими двумя крайностями. Слияние можно использовать для интерполяции между двумя противоположными выражениями лица, позами, режимами передвижения и т. д.

Этот метод позволяет также находить промежуточное состояние между двумя известными позами в *разные* моменты времени. Это может понадобиться в ситуации, когда нужно найти позу персонажа в момент, который не совпадает ни с одним дискретным кадром, доступным в анимационных данных. Мы также можем применить временное слияние, чтобы плавно перейти от одной анимации к другой. Для этого постепенно переходим от исходной анимации к итоговому состоянию за короткий промежуток времени.

12.6.1. Линейная интерполяция

Представьте, что у нас есть скелет с N суставами и двумя позами, $\mathbf{P}_{A\,\mathrm{skel}} = \left\{ \left(\mathbf{P}_A\right)_j \right\}\Big|_{j=0}^{N-1}$ и $\mathbf{P}_{B\,\mathrm{skel}} = \left\{ \left(\mathbf{P}_B\right)_j \right\}\Big|_{j=0}^{N-1}$, и мы хотим найти промежуточную позу $\mathbf{P}_{\mathrm{LERP\,skel}}$ между этими двумя крайностями. Это можно сделать путем *линейной интерполяции* (linear interpolation, LERP) между локальными позами каждого отдельного сустава в двух исходных позах скелета. Записать это можно следующим образом:

$$\left(\mathbf{P}_{\mathrm{LERP}}\right)_j = \mathrm{LERP}\left(\left(\mathbf{P}_A\right)_j, \left(\mathbf{P}_B\right)_j, \beta\right) = (1-\beta)\left(\mathbf{P}_A\right)_j + \beta\left(\mathbf{P}_B\right)_j. \qquad (12.5)$$

Интерполированная поза всего скелета — это просто набор интерполированных поз всех суставов:

$$\mathbf{P}_{\text{LERP skel}} = \left\{ \left(\mathbf{P}_{\text{LERP}} \right)_j \right\} \Big|_{j=0}^{N-1}. \tag{12.6}$$

В этих уравнениях β называется *процентом* или *коэффициентом слияния*. Когда β = 0, итоговая поза скелета в точности совпадает с $\mathbf{P}_{A\ \text{skel}}$, а если β = 1, итоговая поза соответствует $\mathbf{P}_{B\ \text{skel}}$. Когда β находится между нулем и единицей, итоговая поза существует в промежутке между этими двумя крайностями (см. рис. 12.11).

Мы упустили одну небольшую деталь: здесь происходит линейная интерполяция *поз суставов*, то есть интерполяция *матриц преобразования* размером 4 × 4. Но как мы видели в главе 5, интерполировать матрицы напрямую не очень практично. Это одна из причин, почему локальные позы обычно выражаются в формате SRT: это позволяет применить операцию LERP, описанную в подразделе 5.2.5, к каждому компоненту SRT по отдельности. Линейная интерполяция компонента смещения **T** внутри SRT представляет собой обычный вектор LERP:

$$\left(\mathbf{T}_{\text{LERP}} \right)_j = \text{LERP}\left(\left(\mathbf{T}_A \right)_j, \left(\mathbf{T}_B \right)_j, \beta \right) = (1-\beta)\left(\mathbf{T}_A \right)_j + \beta \left(\mathbf{T}_B \right)_j. \tag{12.7}$$

Линейная интерполяция компонента вращения R является кватернионом LERP или SLERP (сферической линейной интерполяции):

$$\begin{aligned} \left(\mathrm{Q}_{\text{LERP}} \right)_j &= \text{normalise}\left(\text{LERP}\left(\left(\mathrm{Q}_A \right)_j, \left(\mathrm{Q}_B \right)_j, \beta \right) \right) = \\ &= \text{normalise}\left((1-\beta)\left(\mathrm{Q}_A \right)_j + \beta \left(\mathrm{Q}_B \right)_j \right). \end{aligned} \tag{12.8}$$

или

$$\begin{aligned} \left(\mathrm{Q}_{\text{SLERP}} \right)_j &= \text{SLERP}\left(\left(\mathrm{Q}_A \right)_j, \left(\mathrm{Q}_B \right)_j, \beta \right) = \\ &= \frac{\sin\left((1-\beta)\theta \right)}{\sin(\theta)} \left(\mathrm{Q}_A \right)_j + \frac{\sin(\beta\theta)}{\sin(\theta)} \left(\mathrm{Q}_B \right)_j. \end{aligned} \tag{12.9}$$

Наконец, линейная интерполяция компонента масштабирования S является либо скалярной, либо векторной операцией LERP в зависимости от вида масштабирования (однородного или неоднородного), поддерживаемого движком:

$$\left(\mathbf{S}_{\text{LERP}} \right)_j = \text{LERP}\left(\left(\mathbf{S}_A \right)_j, \left(\mathbf{S}_B \right)_j, \beta \right) = (1-\beta)\left(\mathbf{S}_A \right)_j + \beta \left(\mathbf{S}_B \right)_j \tag{12.10}$$

или

$$\left(S_{\text{LERP}} \right)_j = \text{LERP}\left(\left(S_A \right)_j, \left(S_B \right)_j, \beta \right) = (1-\beta)\left(S_A \right)_j + \beta \left(S_B \right)_j. \tag{12.11}$$

При линейной интерполяции между двумя позами скелета наиболее естественно выглядящим состоянием обычно является то, в котором поза каждого сустава интерполируется независимо от других в пространстве ближайшего родителя этого сустава. Иными словами, слияние, как правило, применяется

к *локальным позам*. Если попытаться совместить глобальные позы прямо в пространстве модели, результаты будут выглядеть неправдоподобно с биомеханической точки зрения.

Поскольку слияние происходит с локальными позами, линейная интерполяция позы любого сустава совершенно не зависит от интерполяции других суставов скелета. Это означает, что в многопроцессорных архитектурах операции LERP можно полностью распараллелить.

12.6.2. Способы применения слияния методом LERP

Итак, разобравшись с основами слияния методом LERP, можем рассмотреть некоторые типичные способы задействования этой операции в игровой разработке.

Временная интерполяция

Как упоминалось в подразделе 12.4.1, игровая анимация почти никогда не делится на кадры с целочисленными индексами. Частота кадров не является постоянной, поэтому пользователь будет видеть кадры вроде 0,9, 1,85 и 3,02, а не 1, 2 и 3, как можно было бы ожидать. Кроме того, некоторые методы сжатия анимации подразумевают хранение лишь отдельных ключевых кадров, равномерно разнесенных по локальной временной шкале клипа с одинаковыми интервалами. Как бы то ни было, нам нужен какой-то механизм для вычисления промежуточного состояния между дискретными позами, из которых на самом деле состоит анимационный клип.

Обычно для получения таких промежуточных поз используется слияние методом LERP. Представьте, к примеру, что анимационный клип содержит семплы поз с равномерными интервалами 0, Δt, $2\Delta t$, $3\Delta t$ и т. д. Чтобы получить позу в момент $t = 2{,}18\Delta t$, нужно просто найти линейную интерполяцию между позами в точках $2\Delta t$ и $3\Delta t$, используя процент слияния $\beta = 0{,}18$.

В целом, мы можем найти позу в момент t, находящуюся между двумя дискретными позами в промежутке между t_1 и t_2, следующим образом:

$$\mathbf{P}_j(t) = \mathrm{LERP}\left(\mathbf{P}_j(t_1), \mathbf{P}_j(t_2), \beta(t)\right) = \tag{12.12}$$

$$= \left(1-\beta(t)\right)\mathbf{P}_j(t_1) + \beta(t)\mathbf{P}_j(t_2), \tag{12.13}$$

Коэффициент слияния $\beta(t)$ можно получить из соотношения:

$$\beta(t) = \frac{t-t_1}{t_2-t_1}. \tag{12.14}$$

Непрерывность движения: плавный переход

Игровые персонажи анимируются путем объединения большого количества мелких анимационных клипов. Если ваши аниматоры знают свое дело, движения персонажа будут выглядеть естественно и физически правдоподобно *в рамках* каждого отдельного клипа. Однако достичь того же качества при переходе от одного клипа

к другому крайне сложно. Именно эти переходы являются причиной подавляющего большинства рывков, которые можно наблюдать в игровой анимации.

В идеале нам хотелось бы, чтобы движения каждой части тела персонажа были безупречно плавными даже между клипами. Иными словами, трехмерные маршруты, по которым проходят суставы при перемещении скелета, не должны содержать внезапных скачков. Это так называемая *непрерывность C0* (рис. 12.29).

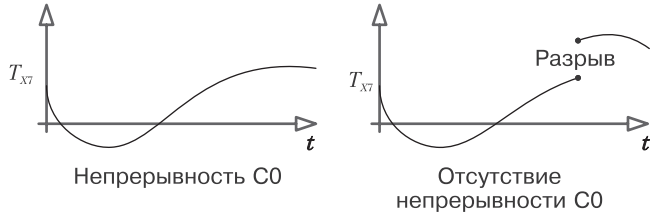


Рис. 12.29. Канальная функция слева обладает непрерывностью C0, а маршрут справа — нет

Непрерывными должны быть не только сами траектории движения, но и их производные первого порядка (кривые скорости). Это *непрерывность C1* (или непрерывность скорости и импульса). Чем выше порядок непрерывности, тем качественнее и реалистичнее выглядят движения анимированного персонажа. Нам, вероятно, следует стремиться к *непрерывности C2*, когда непрерывны и вторые производные траектории движения (кривые ускорения).

Строгая математическая непрерывность уровня C1 и выше часто оказывается недостижимой. Но чтобы получить приятную степень непрерывности движения C0, можно применить слияние анимации на основе LERP. Это обычно позволяет довольно точно симулировать непрерывность C1. Такого рода слияние методом LERP в контексте перехода между клипами иногда называют *плавным переходом*. Этот подход может создать нежелательные искажения, такие как проблема скользящих ног, поэтому его необходимо использовать с осторожностью.

Чтобы выполнить плавный переход между двумя анимациями, мы частично перекрываем временные шкалы клипов (в разумных пределах), а затем совмещаем эти клипы. Процент слияния β в начальный момент времени момент t_{start} равен нулю. Это означает, что в начале плавного перехода мы видим только клип А. Затем мы постепенно увеличиваем значение β, пока в момент t_{end} оно не достигнет единицы. На этом этапе на экране виден только клип Б, а клип А больше не нужен. Отрезок времени, на протяжении которого происходит плавный переход ($\Delta t_{\text{blend}} = t_{\text{end}} - t_{\text{start}}$), иногда называют *временем слияния*.

Разновидности плавных переходов. Существует два распространенных способа выполнения плавного слияния.

- *Плавный переход*. Клипы А и Б воспроизводятся одновременно по мере роста β от нуля к единице. Чтобы это работало как следует, оба клипа должны быть

циклическими, а их временные шкалы должны быть синхронизированными, чтобы позиции ног и рук в одном клипе примерно совпадали с соответствующими позициями в другом (без этого переход может выглядеть совершенно неестественно) (рис. 12.30).

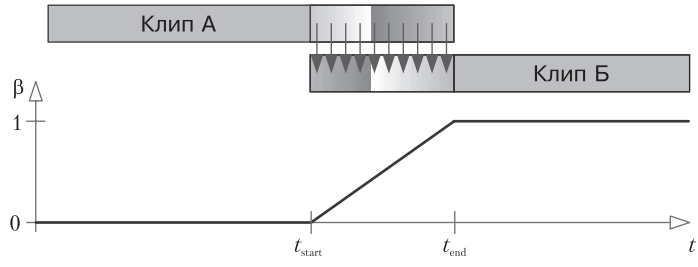


Рис. 12.30. Плавный переход, в ходе которого локальные таймеры обоих клипов продолжают работать

- *Замороженный переход*. Локальный таймер клипа А останавливается в момент, когда начинает воспроизводиться клип Б. Поза скелета из клипа А замораживается, пока клип Б плавно перебирает движение на себя. Такого рода переходное слияние хорошо работает в ситуациях, когда клипы не связаны друг с другом и не могут быть синхронизированы (как в случае с *плавным* переходом) (рис. 12.31).

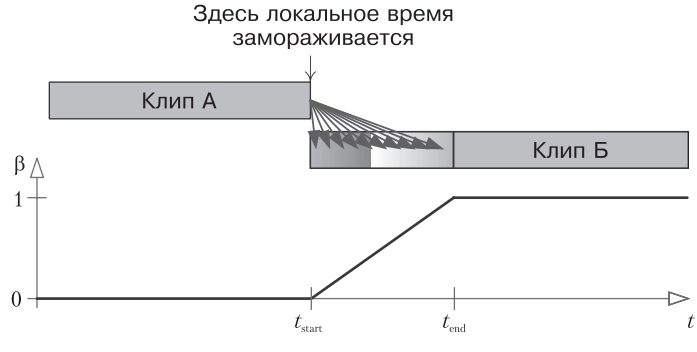


Рис. 12.31. Замороженный переход, в ходе которого останавливается локальный таймер клипа А

Мы также можем влиять на то, как изменяется коэффициент β во время перехода. На рис. 12.30 и 12.31 коэффициент слияния увеличивается линейно. Чтобы сделать переход еще более гладким, изменение β можно выполнять в соответствии с кубической функцией, такой как кривая Безье первого порядка. В контексте исчезающего клипа это называется *кривой замедления* (ease-out), в контексте клипа, который появляется, — *кривой ускорения* (ease-in) (рис. 12.32).

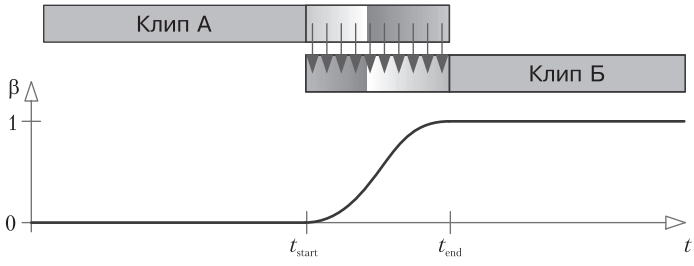


Рис. 12.32. Плавный переход с кубической кривой ускорения/замедления, примененной к коэффициенту слияния

Уравнение для кривой ускорения/замедления Безье приводится далее. Оно возвращается к значению β в любой момент t в рамках интервала слияния. β_{start} — коэффициент слияния в начале интервала t_{start}, а β_{end} — итоговый коэффициент слияния в момент t_{end}. Параметр u — *нормализованное время* между t_{start} и t_{end}, и для удобства мы введем также *обратное* нормализованное время $v = 1 - u$. Обратите внимание на то, что касательные к кривой Безье T_{start} и T_{end} сделаны равными коэффициентам слияния β_{start} и β_{end}, поскольку это дает кривую подходящей формы:

$$u = \left(\frac{t - t_{start}}{t_{end} - t_{start}} \right);\ v = 1 - u;$$

$$\beta(t) = \left(v^3\right)\beta_{start} + \left(3v^2u\right)T_{start} + \left(3vu^2\right)T_{end} + \left(u^3\right)\beta_{end} =$$

$$= \left(v^3 + 3v^2u\right)\beta_{start} + \left(3vu^2 + u^3\right)\beta_{end}.$$

Ключевые позы. Самое время упомянуть о том, что непрерывности движения можно достичь и *без* слияния, если аниматоры сделают так, чтобы последняя поза предыдущего клипа совпадала с первой позой следующего. На практике часто используют набор *ключевых поз*. Например, у нас могут быть ключевые позы для персонажа, стоящего прямо, присевшего, лежащего на животе и т. д. Мы можем достичь непрерывности C0, если в начале и в конце каждого клипа персонаж всегда оказывается в одной из ключевых поз. Для этого достаточно убедиться в том, что при совмещении анимаций ключевые позы совпадают. Непрерывности C1 или более высокого порядка тоже можно добиться, если движение персонажа в конце одного клипа плавно переходит в движение в начале следующего клипа. Для этого создается одна плавная анимация, которая затем разбивается на два или большее количество клипов.

Перемещение в определенном направлении

Слияние анимации на основе LERP часто применяется к перемещениям персонажа. Когда живой человек ходит или бегает, он способен изменить направление движения двумя основными способами. Во-первых, может повернуть все свое тело,

в этом случае он всегда смотрит по направлению движения. Это называют *осевым движением*, поскольку при изменении направления человек поворачивается вокруг своей вертикальной оси. Во-вторых, он может всегда смотреть в одном направлении, независимо от того, куда идет: вперед, назад или в сторону (в мире игр это называется *стрейфом*). Таким образом, направление его движения не зависит от того, куда он смотрит. Я буду называть данный метод *направленным движением*, так как он часто используется для того, чтобы в ходе перемещения прицел оружия или глаза персонажа были направлены на цель. Два стиля движения проиллюстрированы на рис. 12.33.

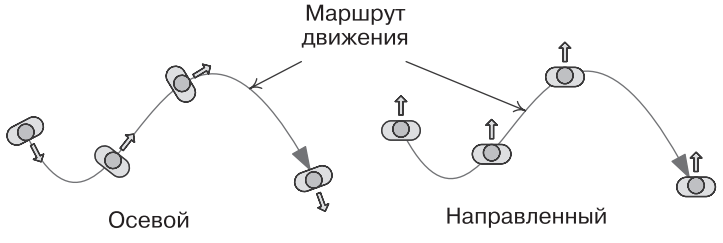


Рис. 12.33. На осевом маршруте персонаж поворачивается по направлению движения вокруг вертикальной оси. На направленном маршруте направление движения может не совпадать с тем, куда смотрит персонаж

Направленное перемещение. Чтобы реализовать *направленное перемещение*, аниматор создает три отдельных циклических клипа: для движения вперед, стрейфа влево и стрейфа вправо. Я буду называть их *клипами направленного перемещения*. Эти три клипа размещаются вдоль полукруга, где прямо — это 0°, влево — 90°, а вправо — –90°. Зафиксировав персонажа по направлению 0°, мы ищем на полукруге нужное направление движения, выбираем два соседних клипа и совмещаем их с помощью слияния методом LERP. Процент слияния β зависит от того, насколько близок угол движения к углам двух соседних клипов (рис. 12.34).

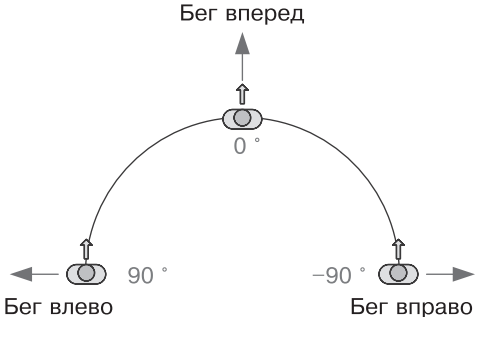


Рис. 12.34. Направленное перемещение можно реализовать слиянием циклических клипов, которые описывают движение в каждом из четырех основных направлений

Обратите внимание на то, что в слиянии не участвует движение назад, в результате чего не получился полный круг. Это вызвано тем, что слияние стрейфа бега задом наперед обычно не может выглядеть естественно. Проблема в том, что при движении влево персонаж, как правило, заносит правую ногу перед левой, чтобы переход к бегу строго вперед выглядел правильно. Аналогично движение вправо обычно сделано так, что левая нога ставится перед правой. Если попытаться совместить такую анимацию стрейфа непосредственно с бегом задом наперед, одна нога начнет пересекать другую, что выглядит чрезвычайно неловко и неестественно. У этой проблемы существует множество решений. Один из возможных вариантов состоит в определении полусферических слияний, по одному для движения вперед и назад, у каждого из них будет анимация стрейфа, рассчитанная на корректную работу при слиянии с бегом в соответствующем прямом направлении. Переходя от одного полукруга к другому, мы можем воспроизводить специальную анимацию перехода, чтобы персонаж успел отрегулировать свою походку и скрещивание ног подходящим способом.

Осевое перемещение. Для реализации *осевого перемещения* можно воспроизвести цикл движения вперед, изменяя направление путем поворота персонажа вокруг его вертикальной оси. Это движение будет более естественным, если тело персонажа не остается строго вертикальным, — людям свойственно немного наклоняться в ту сторону, куда они поворачивают. Мы могли бы попытаться слегка наклонить вертикальную ось всего персонажа, но в результате одна нога получилась бы утопленной в землю, а другая оторвалась бы от земли. Чтобы получить более естественный результат, можно анимировать три разновидности обычной ходьбы или бега вперед: строго вперед, резко влево и резко вправо. Затем, чтобы реализовать нужный угол наклона, выполнить слияние методом LERP между ходьбой вперед и резким поворотом.

12.6.3. Сложные слияния методом LERP

В настоящих игровых движках персонажи используют широкий набор сложных слияний, предназначенных для разных ситуаций. Для удобства и простоты в применении мы можем рассортировать часто используемые сложные слияния. В следующих подразделах рассмотрим несколько популярных разновидностей таких слияний.

Обобщенное одномерное слияние методом LERP

Слияние методом LERP можно легко расширить до более чем двух анимационных клипов, используя методику, которую я называю *обобщенным слиянием*. Мы вводим новый параметр слияния b, который находится в любом подходящем линейном диапазоне, например от -1 до $+1$, от 0 до 1 или даже от 27 до 136. В произвольных точках вдоль этого диапазона можно разместить любое количество клипов (рис. 12.35). Для любого заданного значения b мы выбираем два соседних клипа и совмещаем их с помощью уравнения (12.5). Если два соседних клипа находятся

в точках b_1 и b_2, процент слияния β можно определить, применив методику, аналогичную использованной в уравнении (12.14):

$$\beta(t) = \frac{b - b_1}{b_2 - b_1}. \tag{12.15}$$

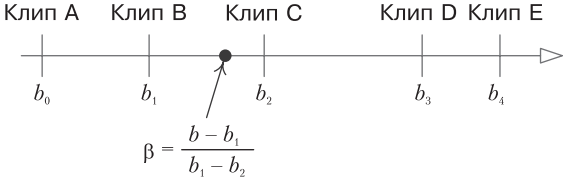


Рис. 12.35. Обобщенное линейное слияние *N* анимационных клипов

Направленное перемещение — это частный случай одномерного слияния методом LERP. Мы просто выпрямляем круг, вдоль которого были размещены направленные анимационные клипы, и используем угол поворота движения 0 в качестве значения b (с диапазоном от –90 до 90°) (рис. 12.36).

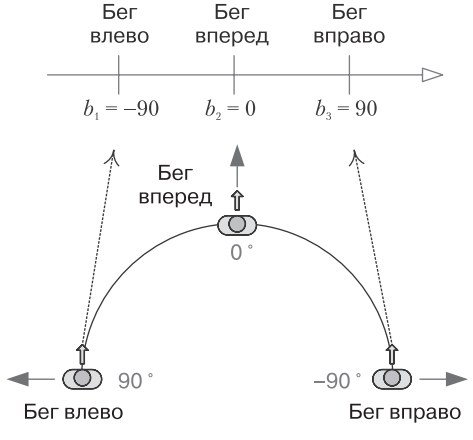


Рис. 12.36. Направленные клипы, используемые при направленном перемещении, можно считать частным случаем одномерного слияния методом LERP

Простое двухмерное слияние методом LERP

Иногда возникает необходимость в плавном изменении сразу *двух* аспектов движения персонажа. Например, мы можем захотеть, чтобы персонаж был способен целиться в вертикальной и горизонтальной плоскостях или в процессе движения варьировал длину шага и ширину постановки ног. Для достижения подобных результатов можем сделать одномерное слияние методом LERP двухмерным.

Если заранее известно, что в двухмерное слияние входят четыре клипа и они повернуты в направлении четырех углов прямоугольной области, для получения совмещенной позы достаточно выполнить два одномерных слияния. Обобщенный

коэффициент слияния b превращается в двухмерный вектор слияния $\mathbf{b} = \begin{bmatrix} b_x & b_y \end{bmatrix}$. Если $\mathbf{b}$ находится в пределах прямоугольной области, ограниченной четырьмя клипами, мы можем определить итоговую позу.

1. Используя горизонтальный коэффициент слияния b_x, находим две промежуточные позы: первую между двумя верхними анимационными клипами, вторую — между двумя нижними. Для нахождения этих двух поз можно выполнить два простых одномерных слияния.
2. Используя вертикальный коэффициент слияния b_y, находим итоговую позу. Для этого совмещаем две промежуточные позы методом LERP.

   Этот метод проиллюстрирован на рис. 12.37.

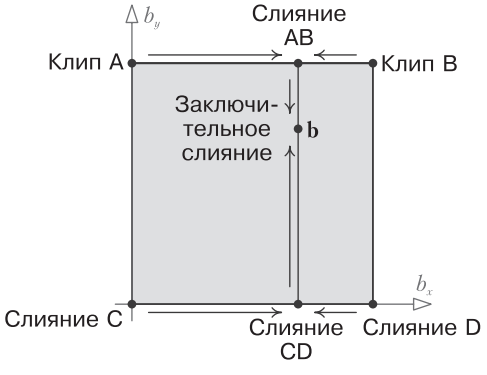


Рис. 12.37. Простая схема двухмерного слияния четырех клипов в углах прямоугольной области

Треугольное двухмерное слияние методом LERP

Простое двухмерное слияние, рассмотренное в предыдущем разделе, работает только в том случае, если анимационные клипы, которые мы хотим совместить, находятся в углах прямоугольной области. Но как выполнить слияние произвольного количества клипов, размещенных в произвольных точках двухмерного пространства?

Допустим, нам нужно совместить три анимационных клипа. Каждый клип, обозначенный индексом i, имеет определенную координату слияния $\mathbf{b}_i = \begin{bmatrix} b_{ix} & b_{iy} \end{bmatrix}$ в двухмерном пространстве, эти три координаты формируют треугольник. У каждого из трех клипов есть набор поз суставов $\left\{ (\mathbf{P}_i)_j \right\}\Big|_{j=0}^{N-1}$, где $(\mathbf{P}_i)_j$ — поза сустава j согласно клипу i, а N — количество суставов в скелете. Мы хотим найти интерполированную позу скелета, относящуюся к произвольной точке $\mathbf{b}$ внутри треугольника (рис. 12.38).

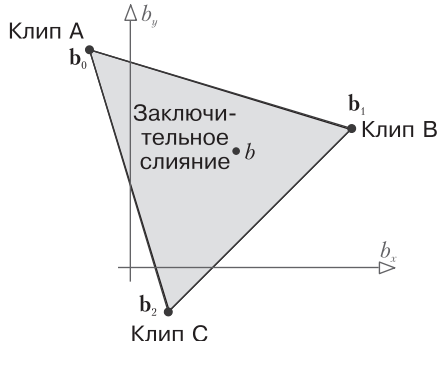


Рис. 12.38. Двухмерное слияние трех анимационных клипов

Но как вычислить LERP-слияние трех анимационных клипов? К счастью, ответ прост: функция LERP может работать с любым количеством входящих параметров, поскольку она, в сущности, представляет собой обычное *среднее взвешенное*. Как и в случае с любым другим взвешенным средним, все веса в сумме должны давать единицу. При совмещении двух входящих параметров мы просто использовали весовые коэффициенты β и $(1 - \beta)$, сумма которых, естественно, равна 1. В случае с тремя параметрами используем три весовых коэффициента, α, β и $\gamma = (1 - \alpha - \beta)$, после чего вычисляем LERP следующим образом:

$$\left(\mathbf{P}_{\text{LERP}}\right)_j = \alpha\left(\mathbf{P}_0\right)_j + \beta\left(\mathbf{P}_1\right)_j + \gamma\left(\mathbf{P}_2\right)_j. \tag{2.16}$$

Имея двухмерный вектор слияния $\mathbf{b}$, мы находим весовые коэффициенты слияния α, β и γ, для чего ищем *барицентрические координаты* точки $\mathbf{b}$ относительно треугольника, сформированного из трех клипов в двухмерном пространстве (https://ru.wikipedia.org/wiki/Барицентрические_координаты). В целом барицентрические координаты точки $\mathbf{b}$ внутри треугольника с вершинами $\mathbf{b}_1$, $\mathbf{b}_2$ и $\mathbf{b}_3$ представляют собой три скалярных значения (α, β, γ), удовлетворяющих уравнениям:

$$\mathbf{b} = \alpha\mathbf{b}_0 + \beta\mathbf{b}_1 + \gamma\mathbf{b}_2 \tag{12.17}$$

и

$$\alpha + \beta + \gamma = 1.$$

Это именно те весовые коэффициенты, которые мы искали для взвешенного среднего из трех клипов. Барицентрические координаты проиллюстрированы на рис. 12.39.

Стоит отметить, что введение в уравнение (12.17) барицентрических координат (1, 0, 0), (0, 1, 0) и (0, 0, 1) дает нам $\mathbf{b}_0$, $\mathbf{b}_1$ и $\mathbf{b}_2$ соответственно. Аналогично, введение этих весов слияния в уравнение (12.16) даст позы $\left(\mathbf{P}_0\right)_j$, $\left(\mathbf{P}_1\right)_j$ и $\left(\mathbf{P}_2\right)_j$ соответственно для каждого сустава. Более того, барицентрическая координата $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ находится в геометрическом центре треугольника и обеспечивает *равномерное* слияние трех поз. Это именно то, чего можно было бы ожидать.

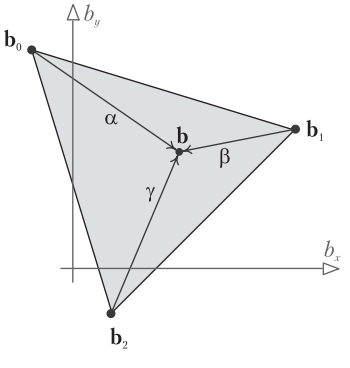


Рис. 12.39. Различные барицентрические координаты внутри треугольника

Обобщенное двухмерное слияние методом LERP

Методику с барицентрическими координатами можно применить к любому количеству анимационных клипов, размещенных в произвольных точках двухмерного пространства. Мы не станем разбирать этот подход целиком, но основная идея заключается в использовании *триангуляции Делоне* (https://ru.wikipedia.org/wiki/Триангуляция_Делоне) для поиска треугольников на основе местоположения разных

анимационных клипов $\mathbf{b}_i$. Сделав это, мы можем найти треугольник, который включает в себя нужную нам точку $\mathbf{b}$, а затем выполнить LERP-слияние трех клипов, как описано ранее. Студия EA Sports (Ванкувер) реализовала этот подход в своем анимационном фреймворке *ANT* и применяла его в играх *FIFA soccer* (рис. 12.40).

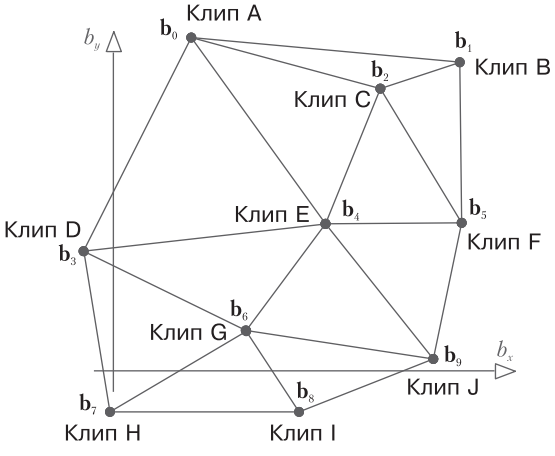


Рис. 12.40. Триангуляция Делоне для произвольного количества анимационных клипов, размещенных в произвольных точках двухмерного пространства слияния

12.6.4. Частичное скелетное слияние

Человек может управлять разными частями своего тела по отдельности. Например, во время ходьбы я могу помахать правой рукой и одновременно указать на что-то пальцем левой руки. Для реализации такого рода движений можно использовать методику *частичного скелетного слияния*.

Как вы помните из уравнений (12.5) и (12.6), при обычном слиянии методом LERP для всех суставов скелета применялся один и тот же коэффициент β. Частичное скелетное слияние дополняет эту идею, позволяя варьировать коэффициенты для каждого отдельного сустава. Иными словами, для каждого сустава j определяется отдельный процент слияния β_j. Набор этих процентов для всего скелета $\left\{\beta_j\right\}\Big|_{j=0}^{N-1}$ иногда называют *маской слияния*, так как его можно использовать для маскировки определенных суставов, обнуляя их коэффициенты.

Допустим, мы хотим, чтобы персонаж помахал кому-то правой рукой. Более того, чтобы он мог это сделать во время ходьбы, бега и стоя на месте. Чтобы реализовать это действие с помощью частичного слияния, аниматор создает три анимации для всего тела: «*идти*», «*бежать*» и «*стоять*», а также клип «*махать*». Затем выполняет маску слияния, коэффициенты которой равны нулю (исключе-

ние составляют правое плечо, локоть, запястье и суставы пальцев — для них они равны единице):

$$\beta_j = \begin{cases} 1, \text{ когда } j \text{ принадлежит правой руке,} \\ 0 \text{ в противном случае.} \end{cases}$$

Результатом LERP-слияния клипов «*махать*» и «*идти*»/«*бежать*»/«*стоять*» с использованием маски слияния будет персонаж, который идет, бежит или стоит и одновременно машет правой рукой.

Частичное слияние полезно, но оно обычно делает движения персонажа неестественными. Это происходит по двум основным причинам.

- Резкое изменение коэффициентов слияния отдельных суставов может привести к тому, что движения разных частей тела будут выглядеть не связанными между собой. В нашем примере коэффициенты слияния резко меняются в суставе правого плеча. Таким образом, движения верхнего отдела позвоночника, шеи и головы основаны на одной анимации, а движения правого плеча и руки — на совершенно другой. Это может выглядеть странно. Проблему можно немного смягчить, если менять коэффициенты слияния постепенно, а не сразу (в примере можно установить коэффициент 0,9 для правого плеча, 0,5 — для верхней части позвоночника и 0,2 — для шеи и средней части позвоночника).
- В реальности движения человеческого тела всегда взаимосвязаны. Например, если человек бежит, мы можем ожидать, что при взмахе его рука будет двигаться сильнее, чем когда он стоит на месте. Но если использовать частичное слияние, анимация правой руки будет неизменной, независимо от того, что происходит с остальными частями тела. Эту проблему сложно преодолеть с помощью частичного слияния. Поэтому многие разработчики предпочитают методику под названием «*аддитивное слияние*», результаты которой выглядят более естественно.

12.6.5. Аддитивное слияние

Аддитивное слияние подходит к проблеме совмещения анимации совершенно по-новому. Оно вводит новый вид анимации — «*клип разницы*», которая, как подсказывает ее название, представляет разницу между двумя обычными анимационными клипами. Добавив к последнему клип разницы, мы можем получить интересные вариации поз и движений персонажа. По своей сути клипы разницы кодируют *изменения*, которые нужно внести в одну позу, чтобы превратить ее в другую. В игровой индустрии их часто называют *аддитивными анимационными клипами*. В этой книге будем применять термин «*клип разницы*», так как он более точно описывает происходящее.

Рассмотрим два входящих клипа: *исходный* (S) и *клип отсчета* (R). По идее, клип разницы D = S – R. Если добавить D к оригинальному клипу отсчета, мы

опять получим исходный клип (S = D + R). Мы также можем генерировать анимации, находящиеся на полпути между R и S, для чего к R нужно прибавить какую-то долю D — это во многом похоже на то, как LERP-слияние находит промежуточную анимацию между двумя крайностями. Но настоящая прелесть метода аддитивного слияния в том, что полученный клип разницы можно добавлять не только к клипу отсчета, но и к другим, не связанным с ним клипам. Назовем такие анимации *целевыми клипами* и обозначим их символом T.

Например, если в клипе отсчета персонаж бежит как обычно, а в исходном клипе он демонстрирует бег и усталость, клип разницы будет содержать только изменения, необходимые для того, чтобы сделать бегущего персонажа усталым. Если применить эту разницу к персонажу в процессе ходьбы, итоговой анимацией может быть усталый пешеход. Мы можем создать множество интересных и очень естественно выглядящих анимаций, добавляя клип разницы к нормальным анимационным клипам. Или сгенерировать набор разных клипов, каждый из которых, будучи примененным к целевой анимации, даст уникальный эффект.

Математическая формулировка

Анимация разницы D определяется как разница между какой-то исходной анимацией S и какой-то анимацией отсчета R. Поэтому, по идее, поза разницы (в отдельный момент времени) составляет D = S – R. Конечно, речь идет о позах суставов, а не о скалярных величинах, поэтому мы не можем их просто так вычитать. Обычно поза сустава представляет собой матрицу аффинного преобразования размером 4×4, которая переводит точки и векторы из локального пространства дочернего сустава в пространство его родителя. Эквивалентом вычитания для матрицы является умножение на обратную матрицу. Поэтому, имея исходную позу $\mathbf{S}_j$ и позу отсчета $\mathbf{R}_j$ для любого сустава скелета j, мы можем определить позу разницы $\mathbf{D}_j$ в этом суставе следующим образом (в данном примере опускаем подстрочные записи $C \rightarrow P$ и $j \rightarrow p(j)$, так как здесь изначально подразумевается, что мы имеем дело с матрицами позы «потомок — родитель»).

$$\mathbf{D}_j = \mathbf{S}_j \mathbf{R}_j^{-1}.$$

Добавление позы разницы $\mathbf{D}_j$ к целевой позе $\mathbf{T}_j$ создает новую аддитивную позу $\mathbf{A}_j$. Для этого достаточно объединить преобразование разницы и целевое преобразование, как показано далее:

$$\mathbf{A}_j = \mathbf{D}_j \mathbf{T}_j = \left(\mathbf{S}_j \mathbf{R}_j^{-1}\right)\mathbf{T}_j. \tag{12.18}$$

Чтобы убедиться в корректности результата, можно посмотреть, что произойдет, если добавить позу разницы обратно к оригинальной позе отсчета:

$$\mathbf{A}_j = \mathbf{D}_j \mathbf{R}_j = \mathbf{S}_j \mathbf{R}_j^{-1} \mathbf{R}_j = \mathbf{S}_j.$$

Иными словами, как и ожидалось, добавление анимации разницы D к оригинальной анимации отсчета R дает исходную анимацию S.

Временна́я интерполяция клипов разницы. Как мы узнали в подразделе 12.4.1, игровые анимации почти никогда не разбиваются на кадры с целочисленными индексами. Чтобы найти позу в произвольный момент времени t, часто приходится выполнять *временну́ю интерполяцию* между соседними семплами позы в моменты t_1 и t_2. К счастью, клипы разницы, как и их неаддитивные аналоги, можно интерполировать по времени. Мы можем применить уравнения (12.12) и (12.14) к клипам разницы, как если бы это были обычные анимации.

Следует отметить, что анимацию разницы можно вычислить, только если входящие клипы S и R одинаковой продолжительности. В противном случае у нас бы был отрезок времени, на протяжении которого один из клипов, S или R, является неопределенным, а это означает, что неопределенным будет и клип D.

Коэффициент аддитивного слияния. В играх часто возникает необходимость в частичном слиянии разных анимаций для изменения степени его воздействия. Например, если клип разницы заставляет персонажа повернуть голову на 80° вправо, слияние его 50%-ной версии даст поворот на 40°.

Чтобы этого достичь, вновь обратимся к старому знакомому LERP. Мы хотим выполнить интерполяцию между оригинальной целевой анимацией и новым клипом, полученным в результате полного применения анимации разницы. Для этого расширим уравнение (12.18) следующим образом:

$$\mathbf{A}_j = \mathrm{LERP}\left(\mathbf{T}_j, \mathbf{D}_j\mathbf{T}_j, \beta\right) = \left(1-\beta\right)\left(\mathbf{T}_j\right) + \beta\left(\mathbf{D}_j\mathbf{T}_j\right). \tag{12.19}$$

Как говорилось в главе 5, мы не можем совмещать матрицы напрямую методом LERP. Поэтому уравнение (11.16) необходимо разбить на три отдельные интерполяции для S, Q и T по аналогии с тем, как мы это делали в уравнениях (12.7)–(12.11).

Сравнение аддитивного и частичного слияния

Аддитивное слияние в некоторых аспектах похоже на частичное. Например, мы можем взять разницу между клипами со стоящим на месте персонажем, в одном из которых он машет правой рукой. Результат будет почти таким же, как при использовании частичного слияния для получения машущей правой руки. Однако аддитивные слияния меньше подвержены проблеме выглядящих бессвязными анимаций, которые получаются при частичном слиянии. Это вызвано тем, что при аддитивном слиянии мы не заменяем анимацию для подмножества суставов и не интерполируем между двумя потенциально не связанными между собой позами. Вместо этого мы, скорее, добавляем движение к оригинальной анимации — возможно, для всего скелета. Благодаря этому анимация разницы знает, как изменить позу персонажа, чтобы заставить его сделать что-то конкретное,

например выглядеть уставшим, повернуть голову в определенном направлении или помахать рукой. Эти изменения можно применить к довольно широкому диапазону анимационных клипов, и результаты часто выглядят очень естественными.

Ограничения аддитивного слияния

Аддитивная анимация, безусловно, не панацея. Она добавляет движение в существующий клип, поэтому ей свойственно вызывать излишнее вращение суставов скелета, особенно в случае одновременного применения нескольких клипов разницы. В качестве простого примера представьте себе целевую анимацию, в ходе которой левая рука персонажа сгибается под углом 90°. Если добавить анимацию разницы, которая тоже поворачивает локоть на 90°, итоговым результатом будет поворот на 90° + 90° = 180°. В итоге предплечье войдет в верхнюю часть руки, что для большинства людей будет не очень удобно!

Очевидно, мы должны с осторожностью выбирать клип отсчета и целевые клипы, которые к нему применяются. Вот некоторые практические рекомендации.

- Минимизируйте вращение бедер в клипе отсчета.
- Плечевые и локтевые суставы в клипе отсчета обычно должны находиться в нейтральных позах, чтобы минимизировать излишнее вращение рук при добавлении клипа разницы к другим целям.
- Аниматоры должны создавать отдельные клипы разницы для каждой ключевой позы, например стоя с прямой спиной, присев, лежа на животе и т. д. Это позволяет аниматору учитывать, как вел бы себя в каждой из этих позиций живой человек.

Эти рекомендации могут послужить хорошим ориентиром, но научиться создавать и применять разные клипы можно либо путем проб и ошибок, либо работая с аниматорами, у которых уже есть опыт создания и использования анимации разницы. Если ваша команда еще не использовала аддитивное слияние, будьте готовы потратить на изучение этого искусства существенное количество времени.

12.6.6. Способы применения аддитивного слияния

Изменение позиции

Одним из самых поразительных приемов применения аддитивного слияния является *изменение позиции*. Для каждой нужной нам позиции создается анимация разницы длиной один кадр. В случае совмещения одного из таких клипов с базовой анимацией позиция персонажа кардинально меняется, при этом он продолжает выполнять свое основное действие (рис. 12.41).

Отклонения в движении

Когда человек бежит, его движения со временем меняются, особенно когда он чем-то занят (например, стрельбой по врагам). Аддитивное слияние можно использовать для придания монотонным движениям элемента случайности или эффекта отвлеченности (рис. 12.42).

Рис. 12.41. Однокадровые анимации разницы А и Б заставляют персонажа принять две совершенно разные позиции. Персонаж взят из Uncharted: Drake's Fortune (снимок экрана с сайта https://beedge.neocities.org)

Рис. 12.42. Аддитивное слияние можно применять для добавления вариативности к монотонной анимации покоя (снимок экрана с сайта https://beedge.neocities.org)

Прицеливание и фокусировка взгляда

Еще одно распространенное применение аддитивного слияния состоит в том, чтобы разрешить персонажу оглядываться вокруг или нацеливать оружие. Для этого персонаж сначала анимируется при выполнении какого-то действия, например бега с головой или оружием, направленными строго вперед. Затем аниматор поворачивает голову или оружие в крайнее правое положение и сохраняет однокадровую или многокадровую анимацию разницы. Процесс повторяется для поворотов влево и вниз. Затем четыре анимации разницы можно аддитивно совместить с оригинальным клипом, в котором голова направлена строго прямо, чтобы заставить персонажа смотреть вправо, влево, вниз или в любом промежуточном направлении.

Угол поворота определяется коэффициентом аддитивного слияния каждого клипа. Например, если совместить 100 % клипа с поворотом вправо, персонаж повернет голову вправо настолько, насколько это возможно. Слияние 50%-ного поворота влево заставит его повернуть голову под углом, который составляет половину от максимального поворота влево. Мы также можем добавить сюда повороты вверх или вниз, чтобы получить диагональное вращение (рис. 12.43).

Рис. 12.43. Аддитивное слияние можно использовать для прицеливания (снимок экрана с сайта https://beedge.neocities.org)

Перегрузка осей времени

Интересно отметить, что ось времени анимационного клипа может не использоваться для представления времени. Например, с помощью трехкадрового анимационного клипа движку можно предоставить три позы прицеливания: влево — в кадре 1, прямо перед собой — в кадре 2 и вправо — в кадре 3. Чтобы заставить персонажа целиться вправо, мы можем просто зафиксировать локальный таймер анимации прицеливания на кадре 3. Чтобы выполнить 50%-ное слияние прицеливания прямо перед собой и вправо, следует задать кадр 2,5. Это отличный пример использования существующих возможностей движка для выполнения новых задач.

12.7. Постобработка

После позиционирования скелета с помощью одного или нескольких анимационных клипов и совмещения результатов посредством линейной интерполяции или аддитивного слияния часто возникает необходимость изменить позу персонажа перед окончательной отрисовкой. Это называют *постобработкой анимации*. В данном разделе мы рассмотрим несколько наиболее распространенных разновидностей этого процесса.

12.7.1. Процедурная анимация

Процедурной называют любую анимацию, которая генерируется во время выполнения, а не описывается в виде данных, экспортированных из таких анимационных программ, как Maya. Иногда для начального позиционирования скелета используются клипы, анимированные вручную, а затем, на этапе постобработки, полученная поза каким-то образом модифицируется с применением процедурной анимации. Процедурная анимация может использоваться также в качестве ввода для системы, которая заменяет собой заранее сгенерированный клип.

Представьте, что мы используем обычный анимационный клип, чтобы кузов автомобиля слегка приподнимался и опускался по мере движения по неровной дороге. Направление, в котором едет автомобиль, определяет игрок. Мы хотим отрегулировать вращение колес и руля, чтобы оно выглядело более убедительным на поворотах. Это можно сделать постобработкой позы, сгенерированной в результате анимации. Предположим, что в оригинальной анимации передние колеса направлены строго прямо, а руль находится в нейтральной позиции. Мы можем использовать текущий угол поворота для создания вокруг вертикальной оси кватерниона, который отклонит передние колеса нужным образом. Этот кватернион можно умножить на канал Q сустава переднего колеса, чтобы получить его итоговую позу. Точно так же мы генерируем кватернион вокруг оси рулевой колонки и умножаем

его на канал Q сустава руля, чтобы руль выглядел повернутым. Эти изменения вносятся в *локальную позу* еще до вычисления глобальной и до генерации палитры матриц (см. раздел 12.5).

Рассмотрим еще один пример. Мы хотим придать деревьям и кустам в игровом мире естественные колебания на ветру и сделать так, чтобы они наклонялись, когда персонаж проходит рядом. Чтобы этого добиться, деревья и кусты можно смоделировать в виде скиновых мешей с простыми скелетами. Вместо клипов, анимированных вручную, или в дополнение к ним можно использовать процедурную анимацию, чтобы суставы двигались естественным образом. Мы также можем применить к вращению разных суставов одну или несколько синусоид или функцию шума Перлина, чтобы они качались на ветру. Или, например, когда персонаж движется по области с кустом или травой, можем наклонять кватернионы их корневых суставов радиально наружу, чтобы все выглядело так, будто они сгибаются под весом персонажа.

12.7.2. Инверсная кинематика

Представим, что у нас есть анимационный клип, в котором персонаж нагибается, чтобы поднять с земли какой-то объект. В Maya этот клип выглядит замечательно, но на реальном игровом уровне земля не бывает идеально плоской, поэтому иногда рука персонажа промахивается мимо объекта или проходит сквозь него. На этот случай мы бы хотели откорректировать финальную позу скелета, чтобы рука выравнивалась точно по целевому объекту. Для этого можно использовать методику под названием «*инверсная кинематика*» (ИК).

Обычный анимационный клип является примером *прямой кинематики* (ПК). В этом случае ввод представляет собой набор поз суставов, а вывод — глобальную позу и матрицу скининга для каждого сустава. Обратная кинематика подходит к проблеме с противоположной стороны: на ввод подается желаемая глобальная поза одного сустава, известного как *конечный эффектор*. В качестве результата мы хотим получить *локальные* позы других суставов в скелете, которые приведут конечный эффектор в нужное положение.

В математическом смысле ИК сводится к задаче *минимизации ошибок*. Как и у большинства задач минимизации, решений здесь может быть сколько угодно: одно, несколько или ни одного. Это логично: если я попытаюсь дотянуться до дверной ручки, которая находится на другом конце комнаты, мне придется к ней подойти. ИК работает лучше всего, когда начальная поза скелета находится довольно близко к желаемой цели. Это помогает алгоритму сосредоточиться на ближайшем решении, расходуя на вычисление разумное количество времени. ИК в действии показана на рис. 12.44.

Представьте себе скелет из двух суставов, каждый из которых может вращаться только вокруг одной оси. Вращение этих суставов можно описать в виде двухмерного углового вектора $\theta = [\theta_1\ \theta_2]$. Множество всех возможных углов для

суставов представляет собой так называемое двухмерное *пространство конфигураций*. Естественно, в более сложных скелетах, в которых суставы обладают большей степенью свободы, пространство конфигураций становится многомерным, но описанные здесь концепции, одинаково хорошо подходят для любого количества измерений.

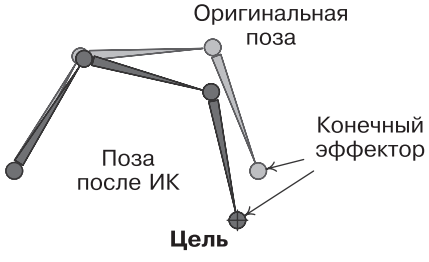


Рис. 12.44. Инверсная кинематика пытается привести сустав конечного эффектора к целевой глобальной позе, минимизируя расхождения (ошибки) между ними

Теперь представьте, что мы строим трехмерную диаграмму, в которой для каждого сочетания вращения суставов (то есть для каждой точки в двухмерном пространстве конфигураций) проводим линию от конечного эффектора к цели (рис. 12.45). Впадины на этой трехмерной поверхности представляют области, в которых конечный эффектор максимально приближен к цели. Если высота поверхности равна нулю, это означает, что цель достигнута. Затем инверсная кинематика пытается найти на этой поверхности минимумы (самые нижние точки).

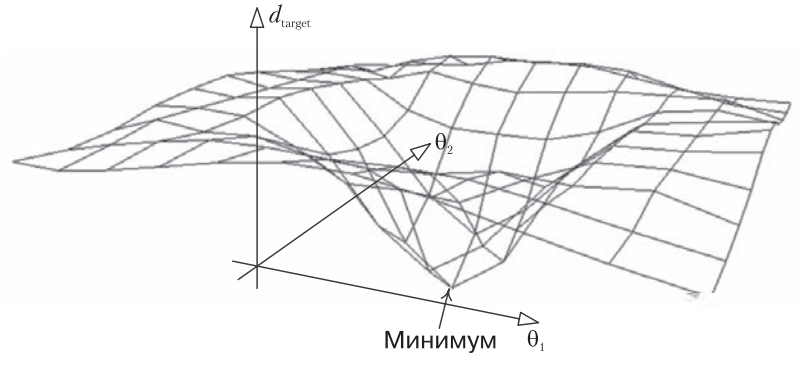


Рис. 12.45. Трехмерная диаграмма расстояния от конечного эффектора до цели для каждого сустава в двухмерном пространстве конфигураций. ИК находит локальный минимум

Мы не станем вдаваться в подробности решения задачи минимизации ИК. Больше об ИК можно почитать на странице https://ru.wikipedia.org/wiki/Инверсная_кинематика и в статье Джейсона Уэбера *Constrained Inverse Kinematics* [47].

12.7.3. Тряпичные куклы

Когда персонаж умирает или теряет сознание, его тело обмякает. В таких ситуациях мы хотим добиться физического реализма с учетом окружения. Для этого можно использовать *тряпичную куклу* — набор физически симулируемых твердых тел, каждое из которых представляет собой полутвердую часть тела персонажа, такую как предплечье или бедро. Твердые тела соединяются друг с другом с помощью суставов скелета таким образом, чтобы сделать движения безжизненного тела естественными. Положение и направление твердых тел определяется физической системой и затем используется для изменения положения и направления ключевых суставов в скелете персонажа. Передача данных из физической системы в скелет обычно выполняется на этапе постобработки.

Чтобы по-настоящему понять, как работает физика тряпичной куклы, сначала нужно разобраться с принципом работы подсистем столкновений и физической симуляции. Более подробно тряпичные куклы рассматриваются в подразделах 13.4.8 и 13.5.3.

12.8. Методы сжатия

Анимационные данные могут занимать много памяти. Поза одного сустава может состоять из десяти каналов (три для смещения, четыре для вращения и до трех для масштаба). Если предположить, что каждый канал содержит четырехбайтовое число с плавающей запятой, односекундный клип каждого сустава, состоящий из 30 семплов, занял бы 4 байта · 10 каналов · 30 семплов = 1200 байт — примерно 1,17 КиБ. В скелете из 100 суставов (что очень скромно по сегодняшним меркам) 1 с несжатого анимационного клипа заняла бы 117 КиБ. Если игра содержит 1000 с анимации (что немного для современной игры), общий набор данных займет чудовищные 114,4 МиБ. Это крайне много, учитывая, что у PlayStation 3 всего 256 МиБ основной RAM и еще столько же видеопамяти. Конечно, в PS4 доступно 8 ГиБ RAM, но лучше сделать анимацию намного более насыщенной и разнообразной, чем тратить память без необходимости. В связи с этим разработчики игр прилагают существенные усилия для сжатия анимационных данных, чтобы предусмотреть как можно больше разнообразных движений при минимальном потреблении памяти.

12.8.1. Пропуск каналов

Одним из простых способов уменьшения размера анимационного клипа является пропуск ненужных каналов. Многие персонажи не требуют неоднородного масштабирования, поэтому три канала, выделенные для масштаба, можно свести к одному. В некоторых играх канал масштабирования можно пропустить для всех

суставов, кроме разве что лицевых. Кости человекоподобных персонажей обычно не могут растягиваться, поэтому мы можем избавиться от смещения во всех суставах, кроме корневого, лицевых и иногда ключичных. Наконец, кватернионы всегда нормализованы, поэтому можно хранить лишь по три компонента в каждом квате (например, x, y и z), а четвертый компонент (например, w) воссоздавать во время выполнения.

Для дальнейшей оптимизации каналы, чьи позы не меняются на протяжении всей анимации, можно хранить в виде единого семпла в момент $t = 0$ с добавлением одного бита, который сигнализирует о том, что канал остается неизменным для всех остальных значений t.

Пропуск каналов может существенно уменьшить размер анимационного клипа. Персонаж с сотней суставов, которые не смещаются и не масштабируются, требует лишь 303 канала: по три для кватернионов каждого сустава и еще три для смещения корневого сустава. Сравните это с 1000 каналами, которые потребовались бы, если бы каждый из 100 суставов был десятиканальным.

12.8.2. Квантование

Еще один способ сделать анимацию более компактной состоит в уменьшении размера каждого канала. Значение с плавающей запятой обычно хранится в 32-битном формате IEEE, в котором 23 бита отводятся для значащей части числа и 8 бит — для экспоненты. Но во многих анимационных клипах такие точность и диапазон излишни. При хранении кватерниона значения канала всегда находятся в диапазоне [–1, 1]. Для величины 1 экспонента 32-битного числа с плавающей запятой в формате IEEE равна нулю, а 23 бита мантиссы дают точность вплоть до седьмого знака после запятой. Опыт показывает, что кватернион можно хорошо закодировать и с точностью 16 бит, поэтому мы реально тратим 16 бит на канал, если храним наши кваты с использованием 32-битных чисел с плавающей запятой.

Преобразование 32-битного значения с плавающей запятой в формате IEEE в n-битное целочисленное представление называется *квантованием*. На самом деле эта операция состоит из двух этапов. *Кодирование* — процесс преобразования оригинального значения с плавающей запятой в квантованное целое число. *Декодирование* — процесс восстановления приближенного к оригиналу значения с плавающей запятой из квантованного целого числа (мы можем восстановить лишь *приближенные* к оригиналу данные — квантование является методом сжатия *с потерями*, поскольку оно фактически уменьшает количество битов точности, использованных для представления значения).

Чтобы закодировать значение с плавающей запятой в виде целого числа, сначала следует разделить диапазон возможных входящих значений на N *интервалов* одинакового размера. Затем нужно определить, в каком интервале находится заданное значение с плавающей запятой, и представить его в виде *целочисленного индекса* этого интервала. Для декодирования квантованного значения мы преоб-

разуем целочисленный индекс в формат с плавающей запятой, после чего сдвигаем и масштабируем его в изначальный диапазон. N обычно подбирается таким образом, чтобы соответствовать диапазону возможных целых значений, которые могут быть представлены n-битным целым числом. Например, если закодировать 32-битное значение с плавающей запятой в виде 16-битного целого числа, количество интервалов будет $N = 2^{16} = 65\,536$.

В колонке *Inner Product* журнала *Game Developer Magazine* помещена отличная статья Джонатана Блоу на тему скалярного квантования с плавающей запятой (https://bit.ly/2J92oiU). В этой статье описываются два способа привязки значения с плавающей запятой к интервалу во время процесса кодирования: можно либо *усечь* число по ближайшей границе интервала в сторону убывания *(T-кодирование)*, либо *округлить* его по центру текущего интервала *(R-кодирование)*. Вместе с этим предлагаются два подхода к воссозданию значения с плавающей запятой из его целочисленного представления: можно вернуть либо биты *по левую сторону* интервала, к которому было привязано оригинальное значение *(L-воссоздание)*, либо значение центра интервала *(C-воссоздание)*. Это дает четыре метода кодирования/декодирования: TL, TC, RL и RC. Следует избегать TL и RC, так как они обычно меняют количество энергии в наборе данных, что часто имеет катастрофические последствия. TC — наиболее эффективный метод с точки зрения пропускной способности, но у него есть большая проблема: он не поддерживает точное представление нулевого значения (значение 0.0f после декодирования превращается в небольшое положительное число). Таким образом, лучшим выбором обычно считается RL, и именно его я здесь продемонстрирую.

В статье рассматривается лишь квантование положительных чисел с плавающей запятой, поэтому для простоты будем считать, что входящие значения находятся в диапазоне [0, 1]. Но мы всегда можем сместить и масштабировать в [0, 1] любой диапазон с плавающей запятой. Например, диапазон каналов кватерниона, который выглядит как [–1, 1], можно привести к [1, 1], добавив к нему 1 и затем разделив его на 2.

Далее приведены две функции для кодирования входящего значения с плавающей запятой в диапазоне [0, 1] в n-битное целое число и его декодирования методом RL, описанным Джонатаном Блоу. Квантованное значение всегда возвращается в виде 32-битного беззнакового целого числа (`U32`), но мы используем лишь n его младших битов в соответствии с аргументом `nBits`. Например, если передать `nBits==16`, полученный результат можно безопасно привести к `U16`:

```
U32 CompressUnitFloatRL(F32 unitFloat, U32 nBits)
{
    // Определяем количество интервалов
    // на основании того, сколько битов попросили сгенерировать.
    U32 nIntervals = 1u << nBits;

    // Переводим входящее значение из диапазона [0, 1]
    // в диапазон [0, nIntervals – 1]. Мы вычитаем один
```

```
    // интервал, потому что нужно вместить в nBits бит
    // как можно большее исходящее значение.
    F32 scaled = unitFloat * (F32)(nIntervals - 1u);

    // Наконец, округляем до центра ближайшего
    // интервала. Для этого прибавляем 0.5f и затем
    // отсекаем интервал с ближайшим индексом
    // в сторону убывания (путем приведения к U32).
    U32 rounded = (U32)(scaled + 0.5f);

    // На случай некорректных входящих значений.
    if (rounded > nIntervals - 1u)
        rounded = nIntervals - 1u;
        return rounded;
}

F32 DecompressUnitFloatRL(U32 quantized, U32 nBits)
{
    // Определяем количество интервалов на основе
    // того, сколько битов мы использовали при кодировании значения.
    U32 nIntervals = 1u << nBits;

    // Декодируем простым приведением U32 к F32, и затем
    // масштабируем по размеру интервала.
    F32 intervalSize = 1.0f / (F32)(nIntervals - 1u);
    F32 approxUnitFloat = (F32)quantized * intervalSize;

    return approxUnitFloat;
}
```

Для поддержки любых входящих значений в диапазоне [min, max] можно воспользоваться следующими функциями:

```
U32 CompressFloatRL(F32 value, F32 min, F32 max,
                    U32 nBits)
{
    F32 unitFloat = (value - min) / (max - min);
    U32 quantized = CompressUnitFloatRL(unitFloat,
                                        nBits);

    return quantized;
}

F32 DecompressFloatRL(U32 quantized, F32 min, F32 max,
                      U32 nBits)
{
    F32 unitFloat = DecompressUnitFloatRL(quantized,
                                          nBits);

    F32 value = min + (unitFloat * (max - min));
    return value;
}
```

Вернемся к изначальной проблеме со сжатием каналов анимации. Для сжатия и разжатия четырех компонентов кватерниона с помощью 16-битных каналов мы просто вызываем функции `CompressFloatRL()` и `DecompressFloatRL()` с min = –1, max = 1 и n = 16:

```
inline U16 CompressRotationChannel(F32 qx)
{
    return (U16)CompressFloatRL(qx, -1.0f, 1.0f, 16u);
}

inline F32 DecompressRotationChannel(U16 qx)
{
    return DecompressFloatRL((U32)qx, -1.0f, 1.0f, 16u);
}
```

Сжатие смещения требует немного больше усилий по сравнению со сжатием вращения, так как его диапазон, в отличие от каналов кватерниона, теоретически неограниченный. К счастью, на практике суставы персонажа не перемещаются слишком далеко, поэтому мы можем определить допустимый диапазон движения и считать некорректной любую анимацию, в которой смещение выходит за его пределы. Исключением из этого правила являются внутриигровые кинематографические сцены: во время анимирования IGC в пространстве игрового мира смещение корневых суставов персонажа может быть очень большим. Чтобы решить эту проблему, мы можем выбирать диапазон допустимых смещений для каждой анимации или каждого сустава в зависимости от того, сколько смещений может быть выполнено в одном клипе. Поскольку в разных клипах диапазон данных может варьироваться, его необходимо хранить вместе со сжатой анимацией. Это прибавит крошечный объем данных к каждому анимационному клипу, но в целом увеличение будет несущественным:

```
// Мы будем использовать радиус 2 м – вы можете выбрать другой.
F32 MAX_TRANSLATION = 2.0f;

inline U16 CompressTranslationChannel(F32 vx)
{
    // Привязываемся к корректному диапазону...
    if (vx < -MAX_TRANSLATION)
        vx = -MAX_TRANSLATION;
    if (vx > MAX_TRANSLATION)
        vx = MAX_TRANSLATION;

    return (U16)CompressFloatRL(vx,
                -MAX_TRANSLATION, MAX_TRANSLATION, 16);
}

inline F32 DecompressTranslationChannel(U16 vx)
{
    return DecompressFloatRL((U32)vx,
                -MAX_TRANSLATION, MAX_TRANSLATION, 16);
}
```

12.8.3. Частота дискретизации и пропуск семплов

Большой размер анимационных данных обычно объясняется тремя факторами: во-первых, поза каждого сустава может содержать свыше десяти каналов с плавающей запятой; во-вторых, скелет содержит много суставов (250 и больше у человекоподобных персонажей на PS3 и Xbox 360 и более чем 800 в играх для PS4 и Xbox One); в-третьих, поза персонажа обычно дискретизируется с высокой частотой (например, 30 кадров в секунду). Мы уже видели несколько способов решения первой проблемы. Со второй проблемой ничего не поделаешь, так как мы не можем уменьшить количество суставов персонажей с высоким разрешением. К третьей проблеме можно подойти с двух сторон.

- *Снизить общее число семплов.* Некоторые клипы выглядят хорошо, даже если их экспортировать с частотой 15 семплов в секунду. Это позволяет уменьшить анимационные данные в два раза.
- *Пропускать некоторые семплы.* Если на протяжении какой-то части клипа данные канала меняются линейным или близким к нему образом, мы можем пропустить на этом интервале все семплы, кроме первого и последнего, а затем на этапе выполнения восстановить их с помощью линейной интерполяции.

Второй метод не совсем простой и требует, чтобы каждый семпл хранил информацию о *времени*. Эти дополнительные данные могут нивелировать весь выигрыш, полученный от пропуска семплов. Но в некоторых игровых движках данный подход оказался успешным.

12.8.4. Сжатие на основе кривых

Одним из самых мощных, простых в использовании и хорошо продуманных API для работы с анимацией является Granny от Rad Game Tools. Granny хранит анимацию не как последовательность семплов поз, разделенных одинаковыми интервалами, а в виде набора неоднородных нерациональных B-сплайнов n-го порядка, которые описывают изменения в каналах сустава S, Q и T. Благодаря применению B-сплайнов каналы с большой кривизной можно закодировать с помощью лишь нескольких точек данных.

Для экспорта анимации Granny дискретизирует позы суставов с равными интервалами аналогично тому, как это делается в традиционных клипах. Затем в каждом канале Granny сохраняет набор B-сплайнов в дискретном наборе данных с заданным отклонением. В итоге получается анимационный клип, размер которого намного меньше, чем у его аналога с однородной дискретизацией и линейной интерполяцией (рис. 12.46).

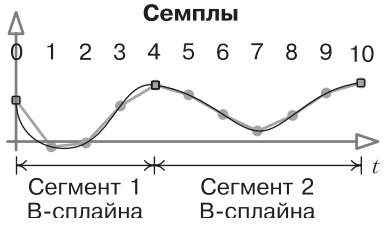


Рис. 12.46. Один из видов сжатия анимации состоит в сохранении B-сплайнов в каналах анимационного клипа

12.8.5. Сжатие с использованием вейвлетов

Еще один способ сжатия анимационных данных заключается в применении *теории обработки сигналов* — в частности, методики *вейвлет-сжатия*. Вейвлет — это функция с амплитудой, которая колеблется, как волна, но с очень небольшой продолжительностью, словно едва заметная рябь на поверхности воды. При использовании в обработке сигналов вейвлет-функции тщательно подбираются для получения нужных характеристик.

В вейвлет-сжатии кривая анимации разбивается на сумму ортонормированных вейвлетов, что во многом похоже на представление произвольного сигнала в виде потока дельта-функций или суммы синусоид. Обработка сигналов и линейные стационарные системы подробно обсуждаются в разделе 14.2, представленные там концепции служат фундаментом для понимания вейвлет-сжатия. Полноценное обсуждение этих методик выходит далеко за рамки данной книги, но вы можете почитать о них в Интернете. Чтобы найти вводные статьи по этой теме, используйте ключевое слово «вейвлет». В блоге Николаса Фречетта есть замечательная статья *Animation Compression: Signal Processing*, рассказывающая о реализации вейвлет-сжатия в игре *Thief* (2014) студии Eidos Montreal.

12.8.6. Выборочные загрузка и потоковая передача

Самым экономным анимационным клипом можно считать тот, который находится вне памяти. Большинству игр не нужно держать в памяти сразу все клипы. Некоторые из них применимы только к определенным классам персонажей, и если эти классы не встречаются в текущем уровне, их можно не загружать. Бывают клипы, необходимые лишь в определенные моменты игры. Их можно загружать в память непосредственно перед воспроизведением и выгружать сразу после завершения.

Большинство игр загружают основной набор анимационных клипов во время загрузки самой игры и постоянно хранят их в памяти. Это относится к основным движениям персонажа игрока и анимациям, применяемым к регулярно встречающимся в игре объектам, таким как оружие или бонусы. Остальные клипы обычно загружаются по мере необходимости. Некоторые игровые движки загружают анимационные клипы по отдельности, но многие упаковывают их в логические группы, которые можно загружать и выгружать как единое целое.

12.9. Конвейер анимации

Операции, выполняемые низкоуровневым анимационным движком, формируют *конвейер*, который преобразует весь ввод (анимационные клипы и параметры слияния) в нужный нам вывод (локальные и глобальные позы плюс палитра матриц для отрисовки).

Конвейер принимает на вход один или несколько клипов и коэффициентов слияния для каждого анимируемого персонажа и объекта в игре, совмещает их и генерирует в качестве результата одну локальную позу скелета. Вместе с этим вычисляются глобальная поза скелета и палитра матриц скининга для движка отрисовки. Обычно при этом предоставляются хуки постобработки, которые позволяют модифицировать локальную позу перед генерацией итоговой глобальной позы и палитры матриц. На этом этапе к скелету применяются инверсная кинематика, физика тряпичной куклы и другие виды процедурной анимации.

Конвейер состоит из следующих этапов.

1. *Разжатие клипа и извлечение поз.* На этом этапе разжимаются данные каждого отдельного клипа и извлекается статическая поза для нужного индекса времени. В результате для каждого клипа генерируется локальная поза скелета. Она может содержать информацию для всех суставов *(поза всего тела)*, подмножества суставов *(частичная поза)* или *позы разницы*, которая применяется в аддитивном слиянии.
2. *Слияние поз.* На этом этапе входящие позы совмещаются с помощью LERP-слияния для всего тела, LERP-слияния для части скелета и/или аддитивного слияния. В результате получается единая локальная поза для всех суставов скелета. Естественно, этот этап выполняется только в случае совмещения более чем одного клипа, в противном случае сразу берется вывод этапа 1.
3. *Генерация глобальной позы.* На этом этапе мы проходимся по скелетной иерархии и объединяем локальные позы суставов, чтобы сгенерировать единую глобальную позу.
4. *Постобработка.* Это необязательный этап, на котором можно модифицировать локальные и/или глобальные позы скелета перед окончанием их формирования. Постобработка используется для инверсной кинематики, физики тряпичной куклы и других модификаций на основе процедурной анимации.
5. *Повторное вычисление глобальных поз.* Многие операции постобработки принимают на вход информацию о глобальной позе, а в качестве результата возвращают локальные позы. В таких случаях мы должны заново вычислить глобальную позу на основе измененной локальной. Очевидно, что операции постобработки, которым не нужны сведения о глобальной позе, можно выполнить между этапами 2 и 3, чтобы избежать повторного вычисления.
6. *Генерация палитры матриц.* После генерации итоговой глобальной позы матрица глобальной позы каждого сустава умножается на соответствующую обратную матрицу позы привязки. Результатом этого этапа является палитра матриц скининга, которую можно передать на вход движку отрисовки.

Типичный конвейер анимации изображен на рис. 12.47.

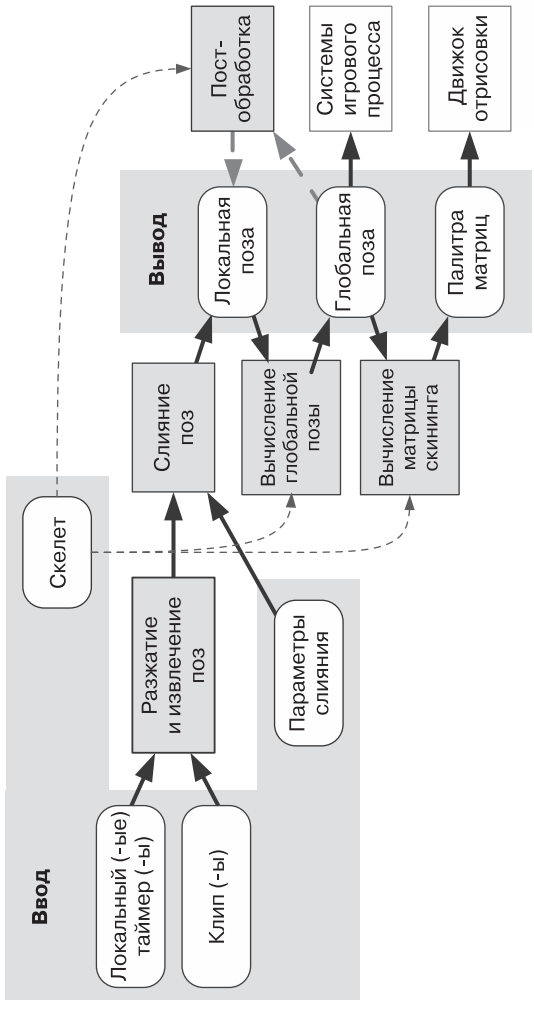


Рис. 12.47. Типичный конвейер анимации

12.10. Конечные автоматы действий

Действия игрового персонажа (стоять, идти, бежать, прыгать и т. д.) лучше всего моделируются с помощью конечного автомата, который многие называют *конечным автоматом действий* (action state machine, ASM). Подсистема ASM размещается над конвейером анимации и предоставляет интерфейс на основе состояний, доступный для использования практически любым высокоуровневым кодом игры.

Любое состояние в ASM относится к слиянию одновременных анимационных клипов произвольной сложности. Некоторые состояния могут быть очень простыми — например, состояние стояния может содержать лишь одну анимацию для всего тела. Другие состояния могут быть более сложными. Состояние бега может относиться к полукруговому слиянию стрейфа влево, бега вперед и стрейфа вправо под углами –90, 0 и 90° соответственно. Состояние «бег со стрельбой» может включать полукруговое направленное слияние, узлы аддитивного или частичного совмещения скелета (чтобы персонаж мог направлять оружие вверх, вниз, влево и вправо), а также дополнительные слияния, позволяющие персонажу оглядываться вокруг с задействованием глаз, головы и плеч. Могут использоваться и дополнительные аддитивные клипы для управления общей позой, походкой и расстоянием между ногами персонажа во время перемещения. Чтобы сделать его более «человечным», движения можно варьировать случайным образом.

Кроме того, ASM персонажа обеспечивает *плавный* переход от одного состояния к другому. Плавность достигается за счет слияния итоговых поз обоих состояний.

Большинство высококачественных анимационных движков позволяют различным частям тела персонажа выполнять одновременно разные, не зависящие или частично зависящие друг от друга действия. Например, он может бежать, целиться, стрелять из оружия, используя руки, и в то же время проговаривать строчку из диалога с помощью лицевых суставов. Движения разных частей тела обычно не обладают идеальной синхронностью: некоторые из них ведущие, другие — ведомые (например, поворот начинается с головы, за которой следуют плечи, бедра и в самом конце ноги). В традиционной анимации эта широко известная методика называется *предвосхищением* [51]. Чтобы реализовать такого рода сложное движение, для управления одним персонажем можно предусмотреть несколько независимых конечных автоматов. Обычно каждый конечный автомат находится в отдельном *слое состояния* (рис. 12.48). Позы, генерируемые ASM каждого слоя, совмещаются в заключительную составную позу.

Все это означает, что в любой момент итоговая поза скелета персонажа основана на разных анимационных клипах. Поэтому нужен какой-то механизм для отслеживания всех активных клипов и описания того, как именно их следует совместить, чтобы получить заключительную позу того или иного персонажа. В целом это можно сделать двумя способами.

1. *С помощью плоского среднего взвешенного.* При использовании этого метода движок хранит плоский список всех анимационных клипов, которые в насто-

ящий момент участвуют в формировании финальной позы, у каждого из них есть один вес слияния. Чтобы получить итоговую позу, клипы совмещаются как одно большое среднее взвешенное значение.

2. *С помощью деревьев слияния*. В этом методе каждый клип, участвующий в формировании позы, представлен листовым узлом дерева. Внутренние узлы представляют различные операции слияния, которые применяются к клипам, многие из них объединяются для формирования состояний действий. Для описания плавных переходов вводятся дополнительные узлы слияния. Затем многоуровневый ASM совмещает позы, полученные из состояний действий в каждом слое. Таким образом, итоговая поза персонажа генерируется у основания этого потенциально сложного дерева слияния.

Рис. 12.48. Многоуровневый конечный автомат, в котором переходы между состояниями каждого слоя не зависят от времени. В этом примере базовый слой описывает позу и движение всего тела персонажа. Слой вариаций привносит разнообразие, применяя к позе персонажа аддитивные клипы. Наконец, два слоя жестикуляции (один аддитивный, другой частичный) позволяют персонажу нацеливаться или указывать на окружающие его объекты игрового мира

12.10.1. Подход с плоским средним взвешенным

При использовании плоского среднего взвешенного каждый анимационный клип, который воспроизводится в текущий момент для заданного персонажа, имеет вес слияния, сигнализирующий о том, какую роль он должен играть в итоговой позе. Ведется плоский список всех *активных* анимационных клипов,

то есть тех, у которых вес слияния не равен нулю. Для вычисления финальной позы скелета мы извлекаем позу с подходящим индексом времени для каждого из N активных клипов. Затем вычисляем для каждого сустава в скелете простое среднее взвешенное векторов смещения, кватернионов вращения и коэффициентов масштабирования, полученных из N активных клипов. Так получаем итоговую позу скелета.

Уравнение среднего взвешенного для набора из N векторов $\{\mathbf{v}_i\}$:

$$\mathbf{v}_{\text{avg}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_i \Big/ \sum_{i=0}^{N-1} w_i.$$

Если весовые коэффициенты *нормализованы*, то есть в сумме дают единицу, это уравнение можно упростить:

$$\mathbf{v}_{\text{avg}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_i,$$

когда

$$\sum_{i=0}^{N-1} w_i = 1.$$

В случае, когда $N = 2$, а $w_0 = (1 - \beta)$ и $w_1 = \beta$, среднее взвешенное сводится к кватерниону, уже знакомому по линейной интерполяции (LERP) между двумя векторами:

$$\mathbf{v}_{\text{avg}} = w_0 \mathbf{v}_A + w_1 \mathbf{v}_B = (1-\beta)\mathbf{v}_A + \beta \mathbf{v}_B = \text{LERP}[\mathbf{v}_A, \mathbf{v}_B, \beta].$$

Эту же формулу со средним взвешенным можно применить и к кватернионам, если рассматривать их как четырехэлементные векторы.

Пример: OGRE

Именно таким образом работает анимационная система OGRE. `Ogre::Entity` представляет экземпляр трехмерного меша (это может быть, к примеру, конкретный персонаж, гуляющий по игровому миру). `Entity` агрегирует объект `Ogre::AnimationStateSet`, который, в свою очередь, ведет список объектов `Ogre::AnimationState`, по одному для каждого активного клипа. Класс `Ogre::AnimationState` показан в следующем фрагменте кода (для ясности было опущено несколько несущественных деталей):

```
/** Представляет состояние анимационного клипа
    и степень его влияния (вес) на общую позу персонажа.
*/
class AnimationState
{
protected:
    String       mAnimationName;   // ссылка на клип
```

```
    Real        mTimePos;       // локальный таймер
    Real        mWeight;        // вес слияния
    bool        mEnabled;       // активна ли эта анимация?
    bool        mLoop;          // является ли анимация циклической?

public:
    /// Функции API...
};
```

Каждый экземпляр `AnimationState` отслеживает локальный таймер и вес слияния одного анимационного клипа. При вычислении итоговой позы скелета конкретной копии `Ogre::Entity` анимационная система OGRE просто перебирает все активные объекты `AnimationState` в своем списке `AnimationStateSet`. Поза скелета извлекается из анимационного клипа в соответствии с его состоянием в момент времени с индексом, который задан таймером этого состояния. Затем для каждого сустава в скелете вычисляется среднее взвешенное N векторов смещения, кватернионов вращения и масштабов, дающее в итоге финальную позу скелета.

Интересно, что у OGRE нет такого понятия, как скорость воспроизведения *(R)*. В противном случае мы могли бы ожидать наличия в классе `Ogre::AnimationState` свойства наподобие:

```
Real   mPlaybackRate;
```

Конечно, мы по-прежнему можем замедлять и ускорять воспроизведение анимации, регулируя количество времени, которое передается в функцию `addTime()`, но, к сожалению, OGRE не имеет встроенной поддержки масштабирования по времени.

Пример: Granny

Анимационная система Granny от Rad Game Tools (http://www.radgametool.com/granny.html) предоставляет механизм плоского слияния анимации на основе среднего взвешенного, подобный используемому в OGRE. Granny поддерживает одновременное воспроизведение любого количества клипов для отдельного персонажа. Состояние каждой активной анимации хранится в структуре данных `granny_control`. Granny вычисляет среднее взвешенное, чтобы определить итоговую позу, автоматически нормализуя веса всех активных клипов. В этом смысле архитектуры Granny и анимационной системы OGRE практически идентичны.

Сильной стороной Granny является работа с временем. В этой системе используется подход с глобальным таймером, рассмотренный в подразделе 12.4.3. Это позволяет циклически проигрывать каждый клип сколько угодно раз, в том числе бесконечно. Клипы также могут масштабироваться по времени, отрицательный масштаб позволяет воспроизводить анимацию задом наперед.

Плавные переходы с плоским средним взвешенным

Если в архитектуре анимационного движка используется плоское среднее взвешенное, плавные переходы реализуются изменением весов самих клипов. Как вы помните, клипы с весом $w_j = 0$ не участвуют в формировании текущей позы персонажа, для ее вычисления берутся только клипы с ненулевым весом, которые совмещаются с усреднением. Если мы хотим плавно перейти от клипа A к клипу B, то просто увеличиваем вес w_B и одновременно уменьшаем вес w_A (рис. 12.49).

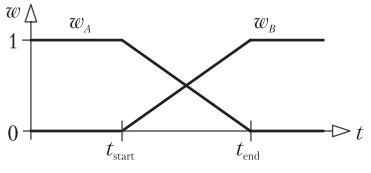


Рис. 12.49. Реализация простого плавного перехода от клипа A к клипу B в архитектуре на основе среднего взвешенного

Плавные переходы в архитектуре со средним взвешенным требуют чуть больше усилий, если нужно перейти от одного сложного слияния к другому. Представьте, к примеру, что персонаж должен перейти от ходьбы к прыжкам. Предположим, что ходьба генерируется на основе среднего значения трех клипов, A, B и C, а для прыжка используется среднее между клипами D и E.

Мы хотим, чтобы персонаж плавно перешел от ходьбы к прыжкам, не влияя на то, как эти две анимации выглядят по отдельности. То есть хотим во время перехода уменьшить общий вес клипов ABC и увеличить общий вес клипов DE, но так, чтобы *относительные веса* каждой из этих групп оставались неизменными. Обозначив коэффициент слияния λ, мы можем удовлетворить это требование, задав нужные веса для *обеих* групп клипов и затем умножив веса исходной группы на $(1 - \lambda)$, а веса конечной группы — на λ.

Чтобы убедиться в работоспособности этого подхода, рассмотрим конкретный пример. Представьте, что до перехода от ABC к DE мы имеем следующие ненулевые весовые коэффициенты: $w_A = 0{,}2$, $w_B = 0{,}3$ и $w_C = 0{,}5$. После перехода ненулевые весовые коэффициенты должны выглядеть так: $w_D = 0{,}33$ и $w_E = 0{,}66$. Таким образом, мы задаем следующие значения:

$$w_A = (1-\lambda)(0{,}2);\ w_B = (1-\lambda)(0{,}3);\ w_C = (1-\lambda)(0{,}5);$$
$$w_D = \lambda(0{,}33);\ w_E = \lambda(0{,}66). \qquad (12.20)$$

Из уравнений (12.20) можно сделать следующие выводы.

1. Когда $\lambda = 0$, итоговая поза является корректным слиянием клипов A, B и C, клипы D и E в этом не участвуют.
2. Когда $\lambda = 1$, итоговая поза является корректным слиянием клипов D и E, клипы A, B и C в этом не участвуют.

3. Когда $0 < \lambda < 1$, *относительные* весовые коэффициенты обеих групп, ABC и DE, остаются корректными, хотя они больше не дают в сумме единицу (сумма весовых коэффициентов в ABC равна $(1 - \lambda)$, а в DE — λ).

Для реализации этого подхода мы должны отслеживать логическое группирование клипов, даже несмотря на то, что на самом низком уровне состояния всех клипов хранятся в одном большом, плоском массиве, таком как `Ogre::AnimationStateSet` в OGRE. В примере выше система должна знать, что A, B и C формируют одну группу, а D и E — другую и что мы хотим перейти от ABC к DE. Для этого, помимо плоского массива состояний клипов, необходимо хранить дополнительные метаданные.

12.10.2. Деревья слияния

Некоторые анимационные движки представляют состояние клипа персонажа не как плоское среднее взвешенное, а скорее в виде дерева операций слияния. Дерево слияния анимации является примером того, что в области создания компиляторов известно как *синтаксическое дерево* или *дерево выражений*. Внутренние узлы такого дерева представлены операторами, ввод для которых предоставляют листовые узлы (если точнее, внутренние и листовые узлы представляют *нетерминальные* и соответственно *терминальные* символы грамматики).

Далее мы еще раз кратко пройдемся по различным видам слияния анимации, с которыми познакомились в подразделах 12.6.3 и 12.6.5, и посмотрим, как каждый из них можно представить в виде дерева выражений.

Деревья слияния с двоичной линейной интерполяцией

Как мы уже видели в подразделе 12.6.1, операция слияния с двоичной линейной интерполяцией (LERP) принимает на вход две позы и совмещает их в одну итоговую позу. Вес слияния β определяет долю, которую должна занимать вторая исходная поза в итоговом результате, доля первой позы — $(1 - \beta)$. Это можно представить в виде двоичного дерева выражений (рис. 12.50).

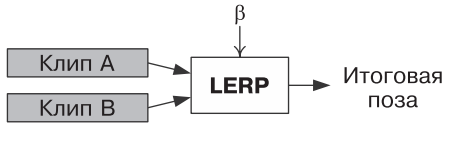


Рис. 12.50. Двоичное слияние методом LERP, представленное двоичным деревом выражений

Деревья обобщенного одномерного слияния

Из подраздела 12.6.3 мы узнали следующее: чтобы было удобно определять обобщенное одномерное слияние методом LERP, произвольное количество клипов

можно разместить вдоль линейной шкалы. Нужное нам совмещение клипов вдоль этой шкалы определяется коэффициентом слияния *b*. Такое слияние можно представить в виде оператора с *n* параметрами (рис. 12.51).

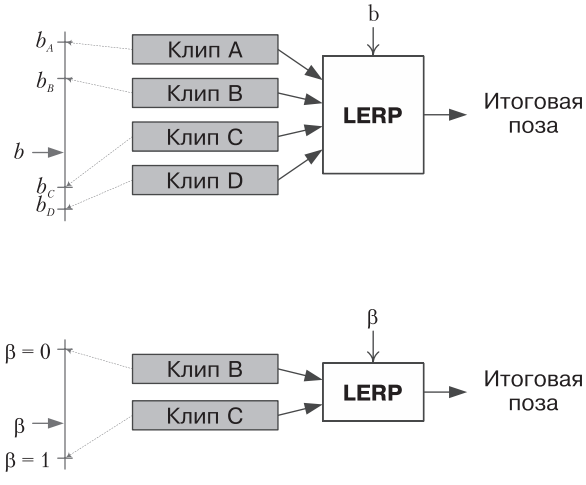


Рис. 12.51. С помощью дерева выражений с множеством входящих параметров можно представить обобщенное одномерное слияние. Такое дерево всегда можно сделать двоичным для любого заданного коэффициента слияния

При наличии конкретного значения *b* такое слияние всегда можно сделать двоичным. Мы просто подаем на вход два клипа, прилегающие к *b*, и вычисляем вес слияния β по формуле (12.15).

Деревья двухмерного слияния методом LERP

В подразделе 12.6.3 мы видели, как двухмерное LERP-слияние можно реализовать путем каскадного совмещения результатов двух двоичных LERP-слияний. Если нужная нам точка двоичного слияния $\mathbf{b} = \begin{bmatrix} b_x & b_y \end{bmatrix}$, эту операцию можно представить в виде дерева (рис. 12.52).

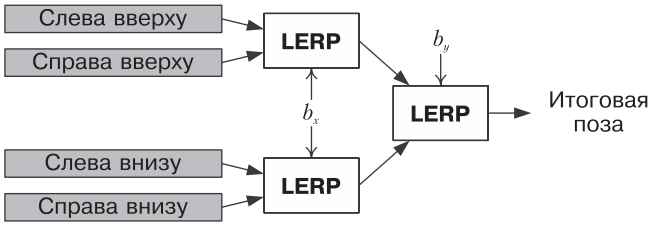


Рис. 12.52. Простое двухмерное LERP-слияние, реализованное в виде каскада двоичных слияний

Деревья аддитивного слияния

В подразделе 12.6.5 описывалось аддитивное слияние. Это двоичная операция, поэтому ее можно представить в виде узла двоичного дерева (рис. 12.53). Один весовой коэффициент слияния β определяет долю аддитивной анимации в итоговом результате: при β = 0 аддитивный клип никак не влияет на результат, а при β = 1 его влияние максимально.

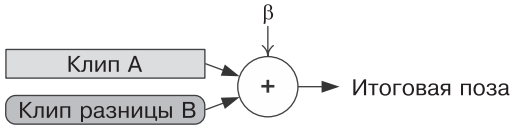


Рис. 12.53. Аддитивное слияние, представленное в виде двоичного дерева

С узлами аддитивного слияния нужно обращаться осторожно, поскольку входящие параметры не являются взаимозаменяемыми, как и в большинстве разновидностей операторов слияния. Первым входящим параметром служит обычная поза скелета, а вторым — специальная *поза разницы*, или *аддитивная поза*. Позу разницы можно применить *только* к обычной позе, и результатом аддитивного слияния будет еще одна обычная поза. Из этого следует, что в качестве аддитивного ввода для узла слияния всегда должен выступать листовой узел, тогда как обычный ввод может быть представлен как листовым, так и внутренним узлом. Если к персонажу нужно применить сразу несколько анимационных клипов, мы должны использовать каскадное двоичное дерево, передавая аддитивные клипы только в качестве аддитивных параметров (рис. 12.54).

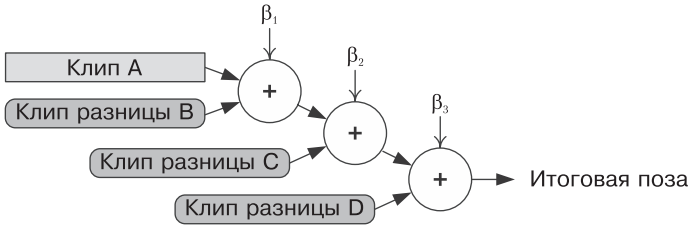


Рис. 12.54. Чтобы аддитивно совместить несколько поз в базовую позу, необходимо применить каскадное двоичное дерево выражений

Деревья многоуровневого слияния

В начале данного раздела отмечалось, что сложные движения персонажа можно генерировать объединением разных независимых конечных автоматов в *слои состояний*. Итоговые позы, полученные из ASM каждого слоя, совмещаются

в объединенную финальную позу. Это можно представить в виде совмещения деревьев слияния каждого активного состояния в одно сверхдерево (рис. 12.55).

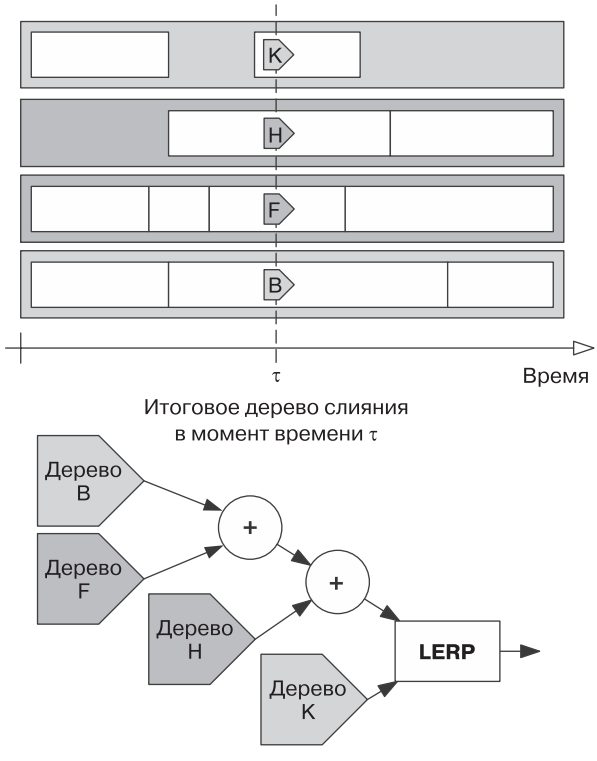


Рис. 12.55. Многоуровневый конечный автомат преобразует деревья слияния разных состояний в единое объединенное дерево

Плавные переходы с применением деревьев слияния

Часто переходы между состояниями внутри одного слоя многоуровневого ASM нужно делать плавными. Решение этой задачи с помощью ASM на основе деревьев выражений является чуть более интуитивно понятным, чем при использовании архитектуры со средним взвешенным. Независимо от того, переходим мы между двумя клипами или сложными слияниями, подход всегда один и тот же: чтобы получить плавный переход, мы просто вводим промежуточный узел двоичной линейной интерполяции между корнями деревьев слияния каждого состояния.

Как и прежде, обозначим коэффициент слияния узла плавного перехода символом λ. В качестве верхнего входящего параметра выступает дерево слияния исходного состояния (это может быть один клип или сложное слияние), а нижним параметром служит дерево конечного состояния (тоже клип или сложное слияние). Во время перехода λ увеличивается от нуля к единице. Когда $\lambda = 1$, переход за-

вершается, а узел плавного перехода методом LERP и его входящее дерево можно удалить. В результате дерево нижнего входящего параметра становится корнем общего дерева слияния для заданного слоя состояний, тем самым завершая переход (рис. 12.56).

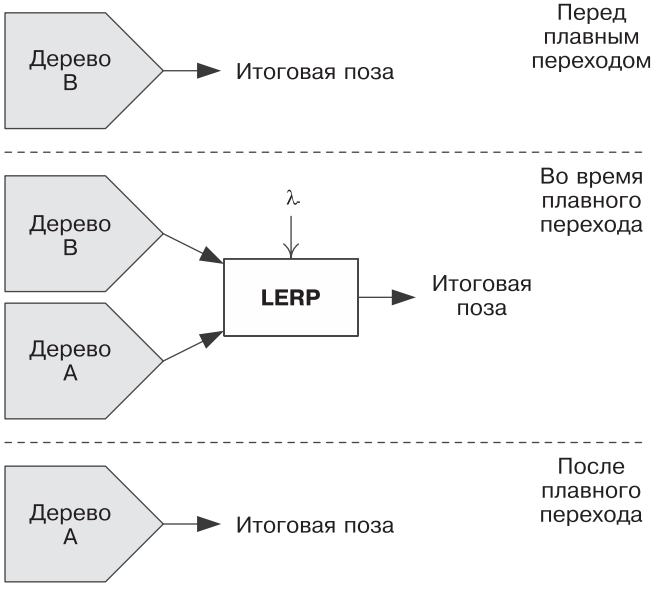


Рис. 12.56. Плавный переход между произвольными деревьями слияния A и B

12.10.3. Параметры состояния и дерева слияния

Анимация и системы управления центральными персонажами игры обычно являются результатом совместной работы аниматоров, игровых дизайнеров и программистов. Этим разработчикам нужно как-то определить состояния, из которых складывается ASM персонажа, описать структуру каждого дерева слияния и выбрать клипы, которые будут подаваться им на вход. Все это можно прописать прямо в коде, однако современные игровые движки позволяют определять состояния анимации с помощью *данных*. Это сделано для того, чтобы пользователь мог создавать новые, удалять ненужные и изменять существующие состояния и наблюдать за результатами своих действий без существенных задержек. Иными словами, главная задача анимационного движка на основе данных состоит в том, чтобы сделать возможной *быструю итеративную* разработку.

Для построения дерева слияния любой сложности требуются всего четыре отдельных типа атомарных узлов: клипы, двоичные слияния методом LERP, двоичные аддитивные слияния и, возможно, тернарные (треугольные) LERP-слияния. Из этих узлов можно сформировать почти любое вообразимое дерево слияния.

Дерево слияния, состоящее исключительно из атомарных узлов, может очень быстро стать большим и громоздким. В связи с этим многие игровые движки для удобства позволяют заранее определять собственные составные типы узлов. Примером этого являются узлы *N*-мерного линейного слияния, которые мы обсуждали в подразделах 12.6.3 и 12.10.2. Можно представить себе мириады сложных типов, каждый из которых предназначен для определенной задачи, которую нужно решать в разрабатываемой игре. В футбольном симуляторе может быть узел, позволяющий персонажу вести мяч, в военной игре — специальный узел, который отвечает за прицеливание и стрельбу. Игра с единоборствами может содержать отдельные узлы для каждого боевого приема, который выполняет персонаж. Способность определять собственные типы узлов дает безграничные возможности.

Существуют разнообразные способы определения состояния анимации. Некоторые игровые движки используют простой минималистичный подход, позволяя описывать состояния анимации в незамысловатом текстовом формате. Иногда для этого предусмотрен удобный графический редактор, который дает возможность формировать состояния анимации путем перетаскивания на холст отдельных компонентов, таких как клипы и узлы слияния, и соединения их произвольным образом. Такие редакторы обычно поддерживают предварительный просмотр, благодаря чему пользователь сразу же может узнать, как будет выглядеть его персонаж в игре. По моему мнению, выбор того или иного метода не сильно сказывается на качестве конечного продукта — главное, чтобы пользователь мог довольно быстро и легко вносить изменения и видеть их результаты.

Пример: движок Naughty Dog

В анимационном движке, который студия Naughty Dog использует в играх серий *Uncharted* и *The Last of Us*, применяется простой подход к определению состояний анимации на основе текстовых файлов. Мы давно работаем с языком Lisp (см. подраздел 16.9.5), так что состояния в движке Naughty Dog записываются с помощью видоизмененной версии языка программирования Scheme, который, в свою очередь, является разновидностью Lisp. Существует два основных типа состояний — *простые* и *сложные*.

Простые состояния. Содержат один анимационный клип, например:

```
(define-state simple
    :name "pirate-b-bump-back"
    :clip "pirate-b-bump-back"
    :flags (anim-state-flag no-adjust-to-ground)
)
```

Пусть вас не пугает синтаксис в стиле Lisp. Этот блок кода всего лишь определяет состояние `"pirate-b-bump-back"` с одноименным анимационным клипом. С помощью параметра `:flags` пользователи могут указывать булевы свойства состояния.

Сложные состояния. Содержат произвольное дерево аддитивных или LERP-слияний. Например, следующее состояние определяет дерево с одним узлом двоичного слияния методом LERP и подает ему на вход два клипа (`"walk-l-to-r"` и `"run-l-to-r"`):

```
(define-state complex
    :name "move-l-to-r"
    :tree
        (anim-node-lerp
            (anim-node-clip "walk-l-to-r")
            (anim-node-clip "run-l-to-r")
        )
)
```

С помощью аргумента `:tree` пользователь может описать любое дерево слияния, состоящее из аддитивных или LERP-узлов, а также узлов, воспроизводящих отдельные анимационные клипы.

Дальше мы можем проследить внутреннюю работу примера (`define-state simple ...`). Он, вероятно, определяет сложное дерево слияния, состоящее из одного узла с клипом:

```
(define-state complex
    :name "pirate-b-unimog-bump-back"
    :tree (anim-node-clip "pirate-b-unimog-bump-back")
    :flags (anim-state-flag no-adjust-to-ground)
)
```

Следующее сложное состояние демонстрирует, как из узлов можно формировать каскадные деревья смешивания произвольной глубины:

```
(define-state complex
    :name "move-b-to-f"
    :tree
        (anim-node-lerp
            (anim-node-additive
                (anim-node-additive
                    (anim-node-clip "move-f")
                    (anim-node-clip "move-f-look-lr")
                )
                (anim-node-clip "move-f-look-ud")
            )
            (anim-node-additive
                (anim-node-additive
                    (anim-node-clip "move-b")
                    (anim-node-clip "move-b-look-lr")
                )
                (anim-node-clip "move-b-look-ud")
            )
        )
)
```

Этот код соответствует дереву, показанному на рис. 12.57.

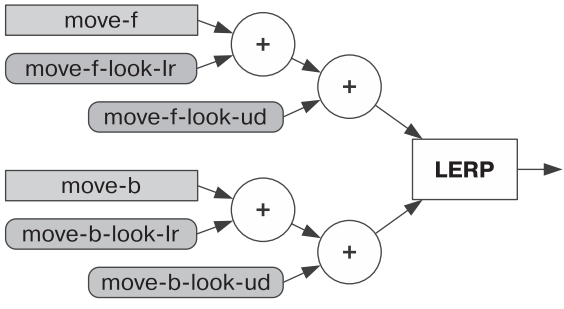


Рис. 12.57. Дерево слияния, относящееся к демонстрационному состоянию "move-b-to-f"

Быстрые темпы разработки. Чтобы достичь быстрых темпов разработки, команда аниматоров Naughty Dog использует четыре важных инструмента.

1. Внутриигровое средство просмотра позволяет поместить персонажа прямо в игру и управлять им с помощью игрового меню.
2. Простая утилита командной строки позволяет перекомпилировать анимационные скрипты и на лету перезагружать их в запущенной игре. Чтобы слегка поправить анимацию персонажа, пользователь может внести изменения в текстовый файл с описанием соответствующего состояния, быстро перезагрузить состояния анимации и сразу же увидеть результат своих действий в игре.
3. Движок постоянно отслеживает все переходы между состояниями каждого персонажа на протяжении последних нескольких секунд игрового процесса. Это дает возможность остановить игру и в буквальном смысле перемотать анимацию назад. Таким образом мы можем тщательно ее исследовать и устранить проблемы, замеченные во время игры.
4. Движок Naughty Dog также предлагает множество инструментов для динамического обновления. Например, если изменить анимационный клип в Maya, он практически мгновенно обновится в игре.

Пример: Unreal Engine 4

Unreal Engine 4 (UE4) предоставляет своим пользователям пять инструментов для работы со скелетными анимациями и мешами: Skeleton Editor, Skeletal Mesh Editor, Animation Editor, Animation Blueprint Editor и Physics Editor.

- Skeleton Editor, в сущности, является средством скелетного моделирования, (также известно как риггинг). С помощью этого инструмента пользователи

могут просматривать и модифицировать скелеты, добавлять *сокеты* к суставам и проверять, как скелет двигается. В других движках сокеты иногда называют *креплениями* или *точками крепления* (см. подраздел 12.11.1).

- Skeletal Mesh Editor дает возможность редактировать свойства мешей, которые крепятся к анимируемому скелету.
- Animation Editor позволяет импортировать и создавать анимационные ресурсы, а также управлять ими. В этом редакторе можно синхронизировать клипы (которые в UE4 называются последовательностями (Sequence)) и регулировать их сжатие. Клипы можно группировать в заранее подготовленные пространства слияния (Blend Spaces), а внутриигровые кинематографические сцены — описывать путем создания анимационных монтажей (Animation Montages).
- Animation Blueprint Editor позволяет задействовать возможности визуальной системы скриптования Animation Blueprint Editor из состава Unreal Engine для управления анимационными конечными автоматами персонажа. Этот редактор изображен на рис. 12.58.
- С помощью Physics Editor пользователи могут моделировать иерархию твердых тел, определяющую движения скелета при симуляции физики тряпичной куклы.

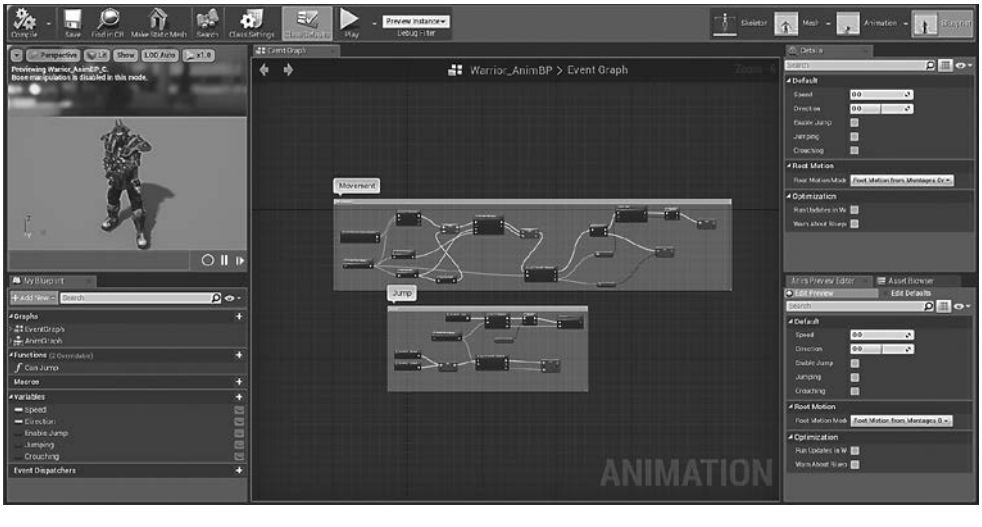


Рис. 12.58. Анимационный редактор Blueprints из состава Unreal Engine 4

Полноценное обсуждение анимационных инструментов Unreal Engine выходит за рамки этой книги, но вы можете узнать о них больше, если поищете в Интернете «Анимация skeletal mesh в Unreal».

12.10.4. Переходы

Для создания высококачественного анимированного персонажа мы должны тщательно управлять *переходами* между состояниями в ASM и следить за тем, чтобы переходы между анимационными клипами были плавными и аккуратными. Большинство современных анимационных движков предоставляют механизм для описания таких переходов в виде данных. В этом разделе мы посмотрим, как подобные механизмы работают.

Виды переходов

Переходами между состояниями можно управлять множеством разных способов. Если известно, что заключительная поза исходного состояния в точности совпадает с первой позой конечного состояния, мы можем просто переключиться из одного в другое. В противном случае можно использовать плавный переход, однако этот способ подходит не всегда. Например, плавный переход от позы лежа к позе стоя попросту не может выглядеть естественно. Здесь требуется одна или несколько отдельных анимаций. В таких случаях для конечного автомата часто создают специальные *переходные состояния*, которые используются исключительно между двумя другими состояниями и не могут выступать самостоятельными устойчивыми узлами. Тем не менее это полноценные состояния, поэтому они могут представлять собой деревья слияния любой сложности. Это дает максимальную гибкость при создании специально анимированных переходов.

Параметры перехода

При описании конкретного перехода между двумя состояниями обычно нужно указывать параметры, определяющие то, как именно этот переход будет осуществляться. Среди них можно выделить следующие.

- *Исходное и конечное состояния*. К какому состоянию (или состояниям) применяется этот переход?
- *Тип перехода*. Этот переход немедленный или плавный? Требует ли он промежуточного переходного состояния?
- *Продолжительность*. Если переход плавный, необходимо знать, сколько времени он занимает.
- *Тип кривой ускорения/замедления*. Для плавного перехода можно указать тип кривой ускорения/замедления, которая будет использоваться для изменения коэффициента слияния.
- *Рамки перехода*. Некоторые переходы могут выполняться только на определенном отрезке локальной временной шкалы исходной анимации. Например, переход от удара кулаком к реакции на столкновение имеет смысл только после замаха рукой. Если попытаться выполнить этот переход во время замаха, он будет отклонен (или же вместо него может быть выбран другой переход).

Матрица переходов

Количество возможных переходов между состояниями обычно очень велико, поэтому их описание может оказаться нелегкой задачей. Максимальное количество переходов в конечном автомате с n состояниями равно n^2. Представьте двухмерную равностороннюю матрицу, вдоль вертикальной и горизонтальной осей которой перечислены все имеющиеся состояния. Получается своеобразная таблица, с помощью которой можно задать любой переход от любого состояния вдоль вертикальной оси к любому состоянию вдоль горизонтальной оси.

В реальной игре такая матрица переходов обычно оказывается полупустой, поскольку не все переходы возможны. Например, от состояния смерти, как правило, нельзя перейти ни к какому другому состоянию. Точно так же, скорее всего, не получится перейти от вождения сразу к плаванию без как минимум одного промежуточного состояния, которое заставит персонажа выпрыгнуть из транспортного средства. Количество уникальных записей в таблице может быть существенно меньше, чем даже число корректных переходов между состояниями. Это вызвано тем, что часто описание одного перехода можно использовать для множества разных пар состояний.

Реализация матрицы переходов

Матрица переходов реализуется множеством способов. Мы можем задействовать приложение для работы с электронными таблицами, чтобы оформить все переходы в виде матрицы. Можем также позволить создавать переходы в том же текстовом файле, в котором описываются состояния действий. Если для редактирования состояний предусмотрен графический пользовательский интерфейс, в нем тоже можно реализовать переходы. В следующих разделах мы кратко рассмотрим несколько примеров реализации матриц переходов из реальных игровых движков.

Пример: шаблонные переходы в Medal of Honor: Pacific Assault. В *Medal of Honor: Pacific Assault (MOHPA)* мы реализовали поддержку описания шаблонных переходов, пользуясь тем, что матрица переходов заполняется не полностью. Исходное и конечное состояния в описании каждого перехода могли содержать звездочки (*), играющие роль символов обобщения. Это позволяло указывать один переход по умолчанию между двумя любыми состояниями (с помощью синтаксиса `from="*" to="*"`) и затем с легкостью уточнять эту глобальную конфигурацию для целых категорий состояний. Мы могли опускаться до уровня отдельных пар состояний, если это требовалось. Матрица переходов в *MOHPA* выглядела примерно так:

```
<transitions>
    <!-- глобальные параметры по умолчанию -->
    <trans from="*" to="*"
           type=frozen duration=0.2>
```

```
    <!-- параметры по умолчанию для любых ходьбы и бега -->
    <trans from="walk*" to="run*"
           type=smooth
           duration=0.15>

    <!-- отдельные параметры для распрямления из пригнутого состояния
      -- (действуют только на отрезке между 2 и 7,5 с
      -- на локальной временной шкале) -->
    <trans from="*prone" to="*get-up"
           type=smooth
           duration=0.1
           window-start=2.0
           window-end=7.5>
    ...
</transitions>
```

Пример: полноценные переходы в Uncharted. В некоторых анимационных движках высокоуровневый игровой код инициирует переходы путем явного указания имени конечного состояния. Проблема подобного решения в том, что вызывающий код должен знать такие подробности, как имена состояний, и какие переходы допустимы в определенном состоянии.

В движке Naughty Dog мы решили эту проблему, превратив переходы между состояниями из вспомогательных деталей реализации в полноценные сущности. Каждое состояние предоставляет список допустимых переходов к другим состояниям, а каждый переход имеет уникальное имя. Имена были стандартизированы так, чтобы по ним было понятно, какие *последствия* имеет переход. Например, если переход называется walk («идти»), он *всегда* меняет любое состояние на состояние ходьбы. Каждый раз, когда высокоуровневый код для управления анимацией хочет перейти от состояния A к состоянию B, он запрашивает имя перехода, а не само конечное состояние. Если такой переход удается найти и он оказывается допустимым, мы его выполняем, в противном случае он отклоняется.

В следующем примере описываются четыре перехода: reload, step-left, step-right и fire. Строчка (`transition-group ...`) инициирует заранее определенную группу переходов, что полезно в случаях, когда один и тот же набор переходов используется в разных состояниях. Команда (`transition-end ...`) определяет переход, который выполняется при достижении конца локальной временной шкалы состояния, если к этому моменту не был инициирован никакой другой переход:

```
(define-state complex
    :name "s_turret-idle"
    :tree (aim-tree
              (anim-node-clip "turret-aim-all--base")
              "turret-aim-all--left-right"
              "turret-aim-all--up-down"
          )
    :transitions (
        (transition "reload" "s_turret-reload"
            (range – -) :fade-time 0.2)
```

```
        (transition "step-left" "s_turret-step-left"
            (range – -) :fade-time 0.2)

        (transition "step-right" "s_turret-step-right"
            (range – -) :fade-time 0.2)

        (transition "fire" "s_turret-fire"
            (range – -) :fade-time 0.1)

        (transition-group "combat-gunout-idle^move")

        (transition-end "s_turret-idle")
    )
)
```

Изящество этого подхода может показаться не совсем очевидным. Его основная задача состоит в том, чтобы позволить редактирование переходов и состояний декларативным путем в виде данных, не требуя во многих случаях внесения изменений в исходный код на C++. Такая степень гибкости достигается за счет ограждения кода, управляющего анимацией, от сведений о структуре графа состояний. Представьте, к примеру, что у нас есть десять состояний ходьбы — обычное, характерное для испуганного человека, вприсядку, при ранении и т. д. Все они могут перейти к состоянию прыжка, но некоторым из них могут понадобиться разные анимационные клипы (например, нормальный прыжок, прыжок от испуга, прыжок из приседа, прыжок при ранении и т. д.). Для каждого из состояний ходьбы определяем переход с простым названием «прыжок». Мы можем направить его к одному обобщенному состоянию прыжка, просто чтобы с чего-то начать. Позже некоторые из этих переходов можно будет модифицировать, чтобы они вели к видоизмененным состояниям прыжка. Можно даже ввести промежуточные состояния между некоторыми видами ходьбы и соответствующими прыжками. Мы получаем возможность вносить любые изменения в граф состояний и параметры переходов, не трогая при этом исходный код на C++, — главное, чтобы *имена* переходов оставались прежними.

12.10.5. Управляющие параметры

С точки зрения программиста, организация всех весовых коэффициентов слияния, скоростей воспроизведения и других управляющих параметров сложного анимированного персонажа может оказаться непростой задачей. Разные весовые коэффициенты слияния по-разному влияют на анимацию персонажа. Например, один коэффициент может отвечать за направление движения, другие — за скорость перемещения, горизонтальную и вертикальную позиции прицела, поворот головы/глаз и т. д. Нужно каким-то образом сделать все эти весовые коэффициенты доступными для кода, который ими управляет.

В архитектуре с плоским средним взвешенным есть плоский список всех анимационных клипов, которые можно применить к персонажу. Состояние каждого

клипа содержит весовой коэффициент слияния, скорость воспроизведения и, возможно, другие управляющие параметры. Код, управляющий персонажем, должен находить состояния отдельных клипов по именам и соответствующим образом модифицировать их весовые коэффициенты слияния. Интерфейс для этого получается простым, однако бóльшая часть ответственности за регулирование коэффициентов слияния ложится на систему управления персонажем. Например, чтобы изменить направление бега, эта система должна знать, что действие «бег» состоит из набора анимационных клипов с названиями вроде StrafeLeft, RunForward, StrafeRight и RunBackward. Ей придется искать состояния этих клипов и вручную регулировать все четыре коэффициента слияния, чтобы повернуть анимацию бега под определенным углом. Разумеется, управление параметрами анимации на таком низком уровне может быть очень хлопотным, к тому же это делает исходный код менее понятным.

При использовании деревьев слияния возникают проблемы другого рода. Благодаря древовидной структуре клипы естественным образом группируются по функциональным модулям. Сложные движения персонажа можно инкапсулировать в составных узлах. Это два полезных преимущества по сравнению с применением плоского среднего взвешенного. Тем не менее управляющие параметры спрятаны внутри дерева. Код, управляющий горизонтальным поворотом головы и глаз, должен заранее знать структуру дерева слияния, чтобы иметь возможность найти подходящие узлы и изменить их параметры.

Разные анимационные движки решают эти проблемы по-разному.

- *Поиск узлов*. Некоторые движки позволяют высокоуровневому коду *искать* узлы слияния в дереве. Узлам, которые могут нам понадобиться, можно назначить специальные имена. Скажем, если узел управляет прицеливанием оружия по горизонтали, его можно назвать HorizAim. Управляющий код может просто поискать узел с определенным именем, и в случае успеха мы будем знать, какой эффект будет иметь изменение его весового коэффициента слияния.
- *Именованные переменные*. Некоторые движки позволяют назначать имена отдельным управляющим параметрам. Чтобы модифицировать их значения, код управления персонажем может найти их по имени.
- *Управляющая структура*. В некоторых движках все управляющие параметры персонажа содержатся в простой структуре данных, такой как массив значений с плавающей запятой или структура языка C. Узлы дерева (деревьев) слияния привязываются к конкретным управляющим параметрам — либо непосредственно в коде с помощью определенных элементов структуры, либо путем поиска по имени или индексу.

Конечно, существует множество других альтернатив. Каждый анимационный движок подходит к этой проблеме по-своему, но в итоге получается примерно один и тот же результат.

12.11. Ограничения

Мы уже видели, как с помощью ASM можно описывать сложные деревья слияния и как матрица переходов позволяет управлять сменой состояний. Еще одним важным аспектом управления анимацией персонажа является ограничение его движений или движений других объектов сцены. Например, может возникнуть необходимость в ограничении оружия таким образом, чтобы оно всегда находилось в руке персонажа, который его несет.

Мы можем ввести ограничения для двух персонажей, чтобы выровнять их положение при рукопожатии. Часто ограничивают ноги персонажа, чтобы они были на одном уровне с полом, и его руки, чтобы подогнать их под ступеньки стремянки или руль автомобиля. В этом разделе кратко обсудим, как такие ограничения реализуются в типичной системе анимации.

12.11.1. Крепления

Практически все современные игровые движки позволяют прикреплять объекты друг к другу. В самом простом случае это достигается за счет ограничения положения и/или поворота конкретного сустава J_A скелета объекта A таким образом, чтобы он совпадал с суставом J_B скелета объекта B. Крепление обычно представляет собой отношение вида «родитель — потомок». Когда двигается скелет родителя, дочерний объект подстраивается под заданное ограничение. Однако движения потомка обычно не влияют на родительский скелет (рис. 12.59).

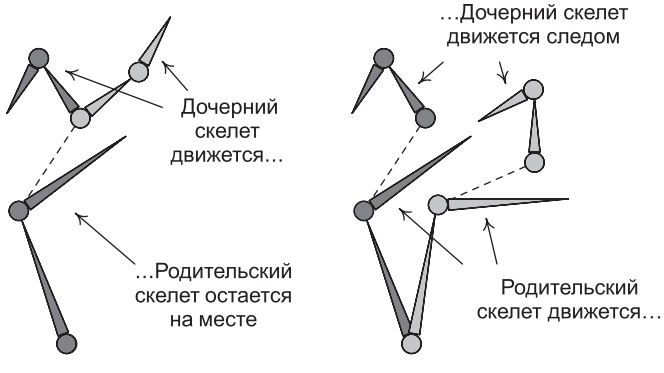


Рис. 12.59. Крепление, демонстрирующее, что движение родителя вызывает движение потомка, но не наоборот

Иногда для удобства между родительским и дочерним суставами можно ввести *смещение*. Например, размещая пистолет в руке персонажа, мы можем ограничить сустав рукоятки так, чтобы он совпадал с суставом правого запястья персонажа. Но в результате положение пистолета в руке может оказаться не совсем правильным. Чтобы решить эту проблему, в одном из двух скелетов можно создать специальный

сустав. Например, мы могли бы добавить к скелету персонажа сустав RightGun, сделать его дочерним по отношению к запястью и разместить таким образом, чтобы в случае прикрепления к нему сустава рукоятки оружие в руке персонажа выглядело естественно. Однако проблема этого подхода в том, что он увеличивает количество суставов в скелете. Каждому суставу требуются ресурсы для слияния анимации и вычисления палитры матриц, к тому же ему нужна память для хранения ключевых поз анимации. Поэтому такой подход часто оказывается непрактичным.

Мы знаем, что дополнительный сустав, предназначенный для крепления, не влияет на позу персонажа — он лишь создает еще одно преобразование между родительским и дочерним суставами. Следовательно, нужно как-то пометить суставы, которые должны игнорироваться конвейером слияния анимации, но при этом могут использоваться для крепления объектов. Такие особые суставы иногда называют *точками крепления* (рис. 12.60).

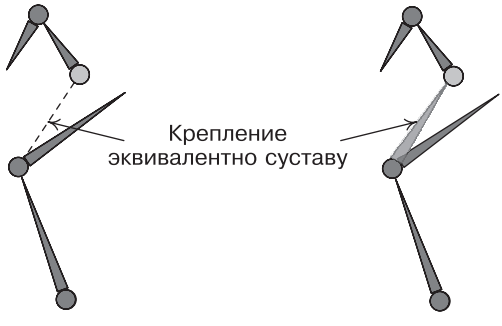


Рис. 12.60. Точка крепления ведет себя как дополнительный сустав между родителем и потомком

Точки крепления можно моделировать в Maya как обычные суставы или локаторы, хотя многие игровые движки определяют их более удобным способом. Например, они могут быть описаны в текстовом файле ASM или с помощью специального графического интерфейса в анимационном редакторе. Это позволяет аниматорам сосредоточится лишь на суставах, влияющих на внешний вид персонажа, тогда как возможности управления креплениями предоставляются тем, кому они нужны, — игровым дизайнерам и программистам.

12.11.2. Выравнивание объектов

С выходом каждой новой игры взаимодействия между игровыми персонажами и окружающей средой становятся все более сложными и утонченными. Поэтому важно иметь систему, которая позволяла бы в ходе анимации выравнивать персонажей и объекты друг относительно друга. Ее можно использовать как для внутриигровых кинематографических сцен, так и для интерактивных элементов игрового процесса.

Представьте, что аниматор, работающий в Maya или другом редакторе анимации, создает сцену с двумя персонажами и объектом — дверью. Два персонажа

пожимают друг другу руки, затем один из них открывает дверь и оба через нее проходят. Аниматор может позаботиться о том, чтобы все три объекта в этой сцене были идеально выровнены. Но при экспорте анимация превращается в три отдельных клипа, которые воспроизводятся для трех отдельных объектов игрового мира. Перед началом воспроизведения этой анимационной последовательности персонажи могут находиться под управлением ИИ или игрока. Как убедиться в том, что на этот момент все три объекта размещены должным образом друг относительно друга?

Контрольные локаторы

Хорошим решением будет добавить во все три анимационных клипа общую точку отсчета. В Maya аниматор может создать внутри сцены *локатор* (обычное трехмерное преобразование, во многом похожее на сустав скелета) и разместить его в любом удобном месте. На самом деле, как мы вскоре увидим, его местоположение и наклон неважны. Локатор помечается таким образом, чтобы инструменты для экспорта анимации знали, что он требует особого обращения.

После экспорта трех анимационных клипов инструменты сохраняют в их файлы положение и наклон контрольного локатора в виде координат относительно локального пространства каждого объекта. Позже, когда эти клипы воспроизводятся в игре, анимационный движок может узнать относительные положение и наклон контрольного локатора в каждом из трех случаев. Затем он может преобразовать точки отсчета трех объектов так, чтобы все три контрольных локатора совпали в пространстве игрового мира. Контрольный локатор во многом ведет себя и даже может быть реализован как *точка крепления* (см. подраздел 12.11.1). В итоге все три объекта будут выровнены друг относительно друга точно так же, как они были расставлены в оригинальной сцене в Maya.

На рис. 12.61 проиллюстрировано, как в Maya можно разместить дверь и двух персонажей из рассмотренного примера. На рис. 12.62 можно видеть, что после экспорта контрольный локатор содержится в каждом анимационном клипе и выражен в локальном пространстве координат всех объектов. Во время игры все три его экземпляра выравниваются по локатору глобального пространства, что позволяет добиться корректного размещения объектов (рис. 12.63).

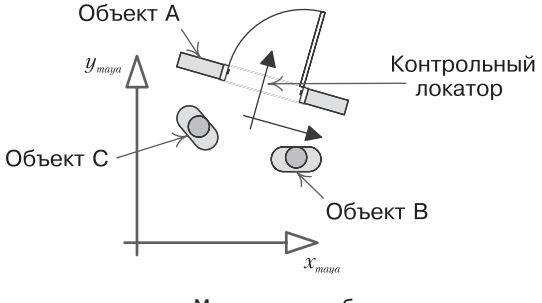


Рис. 12.61. Оригинальная сцена в Maya с тремя объектами и контрольным локатором

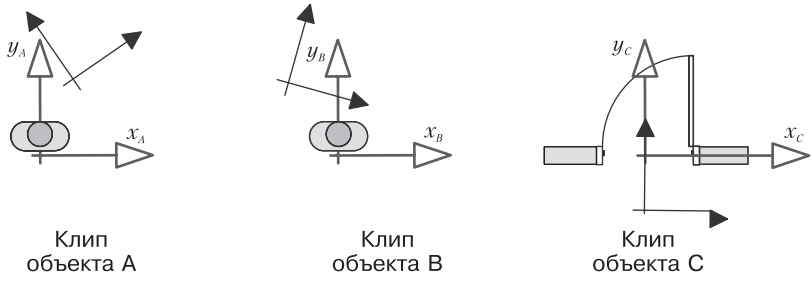


Рис. 12.62. Контрольный локатор сохраняется в анимационном файле каждого объекта

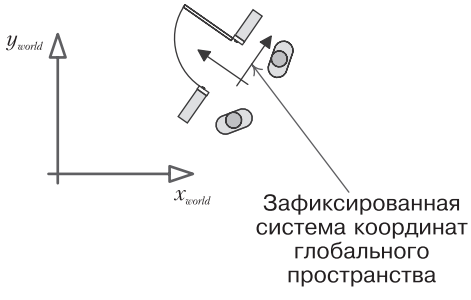


Рис. 12.63. На этапе выполнения контрольные преобразования локального пространства выравниваются по контрольному локатору пространства игрового мира, в результате чего объекты размещаются в нужных позициях

Поиск точки отсчета в глобальном пространстве

Я не упомянул одну важную деталь: кто решает, какими должны быть положение и наклон контрольного локатора в глобальном пространстве? Каждый анимационный клип описывает преобразование контрольного локатора в пространстве координат *своего объекта*. Нам нужно каким-то образом определить, где этот локатор должен находиться в пространстве игрового мира.

В примере с дверью и двумя персонажами, пожимающими друг другу руки, один объект имеет фиксированную позицию (дверь). Поэтому в качестве рабочего решения можно запросить у двери местоположение контрольного локатора и выровнять по нему обоих персонажей. Команды, с помощью которых это делается, могут быть похожими на следующий псевдокод:

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB)
{
```

```
    // Находим преобразование контрольного локатора
    // в глобальном пространстве в анимации двери.
    Transform refLoc = getReferenceLocatorWs(door,
        "shake-hands-door");

    // Воспроизводим анимацию двери без перемещения
    // (она уже находится в правильной позиции)
    playAnimation("shake-hands-door", door);

    // Воспроизводим клипы двух персонажей
    // относительного контрольного локатора
    // в глобальном пространстве, полученного из двери.
    playAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

В качестве альтернативного варианта можно определить преобразование контрольного локатора в глобальном пространстве отдельно от трех объектов сцены. Например, контрольный локатор можно разместить в игровом мире с помощью редактора (см. подраздел 15.3). В этом случае приведенный ранее псевдокод следует исправить примерно таким образом:

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB,
    Actor& refLocatorActor)
{
    // Находим преобразование контрольного локатора
    // в глобальном пространстве с помощью обычного
    // запроса к преобразованию независимого объекта
    // (предположительно размещенного в игровом мире вручную).
    Transform refLoc = getActorTransformWs(
        refLocatorActor);

    // Воспроизводим все анимации относительно контрольного локатора
    // в глобальном пространстве, полученного ранее.
    playAnimationRelativeToReference("shake-hands-door",
        door, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

12.11.3. Инверсная кинематика движений захвата рукой

Иногда во время игры два объекта размещаются не совсем правильно друг относительно друга, даже несмотря на использование крепления. Например, персонаж может держать винтовку в правой руке, а левой рукой поддерживать ложе. Когда персонаж целится под определенными углами, можно заметить, что левая рука неправильно расположена относительно винтовки. Отклонения такого рода вызваны слиянием методом LERP. Даже если в клипах A и B суставы выровнены идеально, нет никакой гарантии, что после LERP-слияния они сохранят правильные позиции.

Чтобы решить эту проблему, можно откорректировать положение левой руки с помощью *инверсной кинематики*. Основная идея в том, что мы сначала определяем нужную конечную позицию сустава, а затем применяем к нему ИК, продвигаясь вверх по иерархии к его родителю и дальше (обычно эта цепочка состоит из двух, трех или четырех суставов). Сустав, позицию которого мы пытаемся исправить, называется *конечным эффектором*. Эта операция изменяет положение родительского сустава (или суставов) конечного эффектора, чтобы как можно сильнее приблизить его к нужной позиции.

API ИК-системы обычно принимает форму запросов для включения инверсной кинематики в отдельно взятую цепочку суставов или исключения из нее с возможностью задания конечной целевой точки. Сами вычисления инверсной кинематики, как правило, выполняются внутри низкоуровневого конвейера анимации. Это позволяет выбрать для них подходящий момент — после вычисления промежуточных локальных и глобальных поз скелета, но перед генерацией заключительной палитры матриц.

Некоторые движки позволяют заранее определять цепочки ИК. Например, мы можем определить одну цепочку ИК для левой руки, одну для правой и по одной для каждой ноги. Предположим, что в данном примере для определения цепочки ИК используется имя ее конечного эффектора (некоторые движки могут применять индексы, дескрипторы или другие уникальные идентификаторы, но сути это не меняет). Функция для включения вычислений ИК может выглядеть так:

```
void enableIkChain(Actor& actor,
                   const char* endEffectorJointName,
                   const Vector3& targetLocationWs);
```

А вот пример функции для выключения цепочки ИК:

```
void disableIkChain(Actor& actor,
                    const char* endEffectorJointName);
```

Обычно включение и выключение ИК происходит нечасто, а вот конечное местоположение в глобальном пространстве приходится обновлять в каждом кадре (если цель перемещается). Поэтому низкоуровневый конвейер анимации всегда предоставляет какой-то механизм для обновления активной целевой точки ИК. Например, этот конвейер может поддерживать многократный вызов `enableIkChain()`.

При первом вызове включается цепочка ИК и устанавливается ее целевая точка. Во всех последующих вызовах эта точка просто обновляется. Для поддержания целей ИК в актуальном состоянии их также можно привязать к динамическим объектам игры. Например, цель ИК может быть задана в виде дескриптора твердого игрового объекта или сустава анимированного скелета.

Инверсная кинематика хорошо подходит для внесения мелких исправлений в расположение сустава, когда тот уже довольно близок к своей цели. Она приносит куда меньше пользы, когда между текущим и желаемым положением существует большое расхождение. Стоит также отметить, что большинство алгоритмов ИК вычисляют лишь *позицию* сустава. Вам, возможно, придется написать дополнительный код, чтобы *наклон* конечного эффектора тоже соответствовал цели. ИК не панацея, и ее отрицательное влияние на производительность может быть существенным. Поэтому применяйте ее с осторожностью.

12.11.4. Извлечение движений и инверсная кинематика ног

Обычно анимация перемещения персонажей в играх должна выглядеть реалистично, с привязкой к земле. Одним из важнейших факторов реалистичного перемещения является то, скользят ли ноги персонажа по поверхности. Скольжение можно преодолеть множеством способов, самые распространенные из них — извлечение движений и инверсная кинематика ног.

Извлечение движений

Представим, что мы анимировали ходьбу персонажа вперед вдоль прямой линии. Работая в Maya или другом анимационном пакете, аниматор делает так, чтобы персонаж выполнил один полный шаг вперед сначала левой ногой, затем правой. Итоговый анимационный клип называется *циклом перемещения*, поскольку во время ходьбы персонажа в игре он должен выполняться в бесконечном цикле. Аниматор следит за тем, чтобы нога персонажа опиралась о землю и не скользила в ходе движения. Персонаж перемещается из исходной позиции (в кадре 0) в новую (в конце цикла) (рис. 12.64).

Обратите внимание на то, что исходная точка в локальном пространстве персонажа остается неподвижной на протяжении всего цикла ходьбы. В результате, делая шаг вперед, он оставляет начальную позицию за плечами. Если воспроизвести эту анимацию в цикле, персонаж будет делать один шаг вперед и затем сразу возвращаться туда, где находился на момент первого кадра в клипе. Очевидно, в игре это работать не будет.

Для исправления ситуации нужно избавиться от перемещения персонажа вперед, чтобы его исходное положение в локальном пространстве всегда оставалось под его центром массы. Для этого можно обнулить смещение вперед корневого сустава скелета персонажа. В итоговом анимационном клипе персонаж будет идти на месте (рис. 12.65).

Рис. 12.64. В пакете для создания анимации персонаж движется вперед и все выглядит так, будто его ноги опираются о землю

Рис. 12.65. Цикл ходьбы после обнуления движения вперед корневого сустава

Чтобы ноги опирались о землю, как в оригинальной сцене в Maya, нужно сделать так, чтобы в каждом кадре персонаж перемещался на нужное расстояние. Мы можем измерить, как далеко продвинулся персонаж, разделить данное значение на время, которое для этого потребовалось, и тем самым получить среднюю скорость движения. Но при ходьбе вперед персонаж не перемещается с постоянной скоростью. Это особенно заметно в случае хромоты (ускорение движения, когда поврежденная нога касается земли, и замедление, когда персонаж опирается на здоровую ногу), но то же самое относится к любому естественно выглядящему циклу ходьбы.

Таким образом, прежде чем обнулить перемещение вперед корневого сустава, нужно сохранить анимационные данные в специальном канале для извлечения движений. Эти данные можно использовать в игре для покадрового перемещения вперед локальной исходной точки персонажа на то расстояние, на которое сместился корневой сустав в Maya. В итоге персонаж будет идти именно так, как было задумано, но вместе с ним перемещается и его исходная точка в локальном пространстве, что позволяет как следует зациклить анимацию (рис. 12.66).

Если в процессе односекундной анимации персонаж перемещается на 1,2 м вперед, мы знаем, что его средняя скорость равна 1,2 м/с. Для того чтобы персонаж перемещался с другой скоростью, мы можем просто изменить частоту воспроизведения анимации цикла ходьбы. Например, чтобы уменьшить скорость до 0,6 м/с, воспроизведем анимацию в два раза медленнее ($R = 0{,}5$).

Рис. 12.66. Цикл ходьбы в игре. Извлеченные данные о движении корневого сустава применяются к исходной точке персонала в локальном пространстве

Инверсная кинематика ног

Извлечение движений хорошо справляется с привязкой ног персонажа к земле при движении вперед по прямой (или, точнее, по маршруту, который предусмотрел аниматор). Однако то, как перемещается и поворачивается реальный игровой персонаж, может не совпадать с оригинальным маршрутом движения, анимированным вручную (например, при перемещении по неровной поверхности). Это приводит к дополнительному скольжению ног.

Одним из решений этой проблемы является коррекция скольжения с помощью ИК. Основная идея состоит в том, чтобы проанализировать анимацию и определить, в какие отрезки времени нога полностью касается земли. В момент контакта мы записываем позицию ноги в пространстве игрового мира. Во всех последующих кадрах в рамках полученного отрезка времени с помощью ИК исправляем позу ноги, чтобы она оставалась в нужном нам месте. Эта методика выглядит довольно просто, но, чтобы движение выглядело и ощущалось естественным, может потребоваться много усилий. Это поэтапный и скрупулезный процесс. К тому же некоторые естественные для человека движения, такие как удлинение шага при повороте, невозможно симулировать, используя одну лишь инверсную кинематику.

Кроме того, существует большая разница между *внешним видом* анимации и *восприятием* персонажа, особенно если им управляет игрок. Обычно отзывчивость и увлекательность системы управления игровым персонажем важнее, чем создание идеальной анимации. Учитывая все сказанное, к добавлению в игру ИК ног и механизма извлечения движений следует относиться серьезно. Будьте готовы к тому, что это потребует множества проб и ошибок: вы должны найти такой баланс, чтобы ваш персонаж хорошо *выглядел* и им было *приятно* управлять.

12.11.5. Другие виды ограничений

В движок анимации игры можно добавить множество других систем ограничения.

- *Направление взгляда*. Это способность персонажа фокусировать взгляд на определенных точках интереса в окружающей среде. Добиться этого можно с помощью одних только глаз, либо глаз и головы, либо совместного движения глаз, головы и всей верхней половины тела. Такое ограничение иногда реализуется с применением ИК или процедурных смещений суставов, однако более естественный результат можно получить при аддитивном слиянии.
- *Выравнивание по укрытию*. Это способность персонажа идеально выравниваться относительно объекта, служащего укрытием. Чтобы его реализовать, часто используют метод на основе контрольного локатора, описанный ранее.

- *Вход и выход из укрытия*. Если персонаж способен укрываться, для процессов входа и выхода из укрытия обычно необходимо применять слияние анимации и отдельные клипы.
- *Средства обхода*. Способность персонажа проходить над препятствием, под ним, мимо или через него может существенно оживить игру. Для этого часто используются специальные анимационные клипы и контрольный локатор, которые позволяют добиться правильного выравнивания относительно преодолеваемого препятствия.

13 Столкновения и динамика твердого тела

В реальном мире твердые объекты являются... твердыми по своей природе. В целом они ведут себя предсказуемо и, например, сами по себе не проходят друг сквозь друга. Но в игровом мире объекты делают только то, что мы им приказываем, поэтому игровые программисты должны специально заботиться о том, чтобы один элемент игры не проходил сквозь другой. За это отвечает компонент, который является одним из центральных в любом игровом движке, — *система обнаружения столкновений*.

Часто система столкновений тесно интегрирована с *физическим движком*. Конечно, физика — обширная область, и в большинстве современных игровых движков под ней понимают скорее симуляцию *динамики твердого тела*. *Твердое тело* — это идеально твердый цельный объект, который нельзя деформировать. Под *динамикой* имеется в виду процесс определения того, как такие твердые тела *двигаются* и *взаимодействуют* с течением времени под воздействием разных *сил*. Симуляция динамики твердого тела позволяет делать движения игровых объектов высокоинтерактивными и беспорядочными, но при этом естественными. Этого эффекта куда сложнее достичь, если для перемещения элементов используются готовые клипы.

В симуляции динамики активно применяется система обнаружения столкновений. Это позволяет как следует смоделировать различные аспекты физического поведения объектов, включая отскок друг от друга, скольжение с учетом трения, перекатывание и переход в неподвижное состояние. Конечно, систему обнаружения столкновений можно использовать отдельно, без симуляции динамики — у многих игр вообще нет системы для моделирования физики. Но игры, в которых объекты перемещаются по двух- или трехмерному пространству, должны как-то следить за столкновениями.

В этой главе мы исследуем архитектуру типичной системы обнаружения столкновений и типичного физического движка, моделирующего динамику твердого тела. В ходе рассмотрения компонентов этих двух тесно связанных между собой механизмов уделим внимание математическим и теоретическим аспектам, лежащим в их основе.

13.1. Нужна ли в вашей игре физика?

Сейчас у большинства игровых движков есть какие-то механизмы для физической симуляции. Геймеры попросту ожидают некоторых физических эффектов, таких как анимация смерти по принципу тряпичной куклы. Эффекты наподобие движения веревок, ткани, волос или более сложных физически моделируемых элементов могут придать игре нечто *неуловимое*, что выделяет ее на фоне конкурентов. В последние годы некоторые игровые студии начали экспериментировать с продвинутыми физическими симуляциями, включая приближенные к естественным эффекты колебания жидкости в реальном времени и моделирование деформируемых тел. Использование физики в игре чревато определенными затратами, и прежде, чем браться за реализацию исчерпывающего списка физических возможностей в игре, мы должны как минимум разобраться с тем, на какие компромиссы придется пойти.

13.1.1. Что можно делать с системой симуляции физики

Вот лишь несколько возможностей и игровых элементов, которые предоставляет система симуляции физики в игре.

- Обнаружение столкновений между динамическими объектами и статической геометрией игрового мира.
- Симуляция твердых тел с полной свободой движения под воздействием тяготения и других сил.
- Системы пружин.
- Строения и структуры, которые можно разрушать.
- Бросание лучей и фигур (для определения поля зрения, воздействий пули и т. д.).
- Многогранники срабатывания (определяют, когда объект входит в заранее заданные участки игрового мира, покидает их или находится внутри).
- Сложные механизмы (краны, головоломки с движущимися платформами и т. д.).
- Ловушки (такие как лавина валунов).
- Транспортные средства с реалистичной подвеской, которые можно водить.
- Эффекты смерти персонажа по принципу тряпичной куклы.
- Оживленная тряпичная кукла — реалистичная смесь традиционной анимации и физики тряпичной куклы.
- Болтающиеся предметы (фляга, ожерелье, мечи), довольно реалистичные волосы, движение одежды.
- Симуляция ткани.
- Симуляция поверхности воды и плавучих предметов.
- Распространение звука.

Это далеко не полный список.

Здесь следует отметить, что физическую симуляцию можно реализовывать не только во время игры, но и в рамках этапа предварительной обработки, чтобы сгенерировать анимационный клип. Для средств анимации, таких как Maya, доступен целый ряд дополнений подобного рода. Этот подход используется также в пакете *Endorphin*[1] от NaturalMotion, Inc. (www.naturalmotion.com/endorphin.htm). В этой главе мы ограничимся обсуждением симуляции динамики твердого тела на этапе выполнения, но при планировании игрового проекта следует помнить об аналогичных инструментах, которыми можно пользоваться во время разработки.

13.1.2. Делает ли физика игру интересной?

Наличие системы динамики твердого тела еще не гарантирует, что игра будет увлекательной. Очень часто хаотичная физическая симуляция лишь ухудшает впечатления от игры. Занимательность, связанная с физикой, зависит от многих факторов: качества симуляции и ее интеграции с другими системами движка, выбора элементов игрового процесса, подчиняющихся физическим законам и управляемых напрямую, влияния физических элементов на цели, стоящие перед игроком, и способности его персонажа, а также жанра игры.

Рассмотрим несколько общих игровых жанров и обсудим то, как система динамики твердого тела может вписаться в каждый из них.

Симуляторы

Основная задача симулятора — сделать игровой процесс как можно более близким к реальности. В качестве примеров можно привести игры из циклов *Flight Simulator*, *Gran Turismo* и *NASCAR Racing*. Очевидно, что реализм, обеспечиваемый системой динамики твердого тела, отлично подходит для такого рода игр.

Физические головоломки

Смысл физических головоломок — дать возможность пользователю порезвиться с динамически симулируемыми игрушками. Поэтому, естественно, механика такого рода игр почти полностью основана на физике. Примерами этого жанра являются *Bridge Builder*, *The Incredible Machine*, онлайн-игра *Fantastic Contraption* и *Crayon Physics* для iPhone.

Жанр «песочница»

В играх этого жанра может не быть никаких целей, но также они могут содержать множество необязательных задач. Обычно игрок просто слоняется по игровому миру и пытается понять, что можно сотворить с теми или иными игровыми

[1] NaturalMotion предлагает версию Endorphin под названием Euphoria, предназначенную для работы на этапе выполнения.

объектами. Примерами игр-песочниц являются *Besiege*, *Spore*, цикл *LittleBigPlanet* и, конечно же, *Minecraft*.

Игры-песочницы могут выиграть от реалистичной симуляции динамики, особенно если основная увлекательность заключается в реалистичном или близком к тому взаимодействии между объектами игрового мира. В таких случаях физика может быть занятной сама по себе. Хотя многие игры жертвуют реалистичностью, чтобы сделать игровой процесс более развлекательным, например, за счет нереально мощных взрывов, необычной силы тяжести и т. д. Поэтому симуляцию динамики часто приходится корректировать разными способами, чтобы добиться нужных ощущений.

Игры с акцентом на сюжете и достижении целей

Игра, ориентированная на достижение целей, имеет правила и определенные задачи, не выполнив которые игрок не может продвинуться дальше. Для сюжетной игры самое главное — рассказать историю. Интеграция системы физики в такого рода игры может представлять определенные сложности. Обычно *реалистичная симуляция* требует ослабления *контроля* происходящего, а это может помешать игроку достичь целей или воспрепятствовать развитию сюжета.

Представьте, к примеру, игру-платформер, в которой большое внимание уделяется персонажу. Мы хотим, чтобы движения игрока были занятными и простыми, но не обязательно реалистичными с точки зрения физики. В военной игре было бы неплохо сделать так, чтобы взрыв моста выглядел реалистично, но при этом мы не хотим, чтобы его развалины заблокировали единственный путь, которым игрок может пройти вперед. В таких играх физика может не только не добавить интереса, но и ухудшить игровой процесс, если физически симулируемое поведение объектов игрового мира мешает игроку выполнять поставленные перед ним задачи. Поэтому разработчики должны применять физику осторожно и рассудительно, контролируя поведение симуляции так, чтобы оно не препятствовало игровому процессу. Также было бы неплохо дать возможность игроку самостоятельно выбираться из сложных ситуаций. Хорошим примером этого служат игры из цикла *Halo*, в которых игрок может нажать кнопку X, чтобы поставить на колеса перевернутое транспортное средство.

13.1.3. Влияние физики на игру

Добавление в игру физической симуляции может иметь самые разные последствия для проекта и игрового процесса. Вот лишь несколько примеров в контексте разных аспектов игровой разработки.

Влияние на архитектуру

- *Предсказуемость*. Неизбежные хаос и изменчивость, которые отличают физически симулируемое поведение от анимированного, являются источником непредсказуемости. Если какое-то действие в игре должно всегда происходить

определенным образом, обычно лучше анимировать его, чем пытаться заставить динамическую симуляцию каждый раз выдавать нужное движение.

- *Тонкая настройка и контроль*. Законы физики, если их как следует смоделировать, являются неизменными. В игре мы можем откорректировать силу тяжести или коэффициент упругого восстановления твердого тела, что позволяет в некоторой степени вернуть контроль за происходящим. Однако результаты изменения физических параметров часто бывает сложно предугадать и четко представить. Чтобы заставить персонажа двигаться в нужном направлении, намного проще изменить анимацию ходьбы, чем какую-то физическую силу.
- *Непредсказуемое поведение*. Иногда физика делает некоторые аспекты игры непредсказуемыми. Таковы, например, прыжок с помощью гранатомета в *Team Fortress Classic*, взрыв автомобиля Warthog в высоком полете в *Halo* и летучие доски для серфинга в *PsyOps*.

В целом, архитектура игры должна диктовать требования к физике игрового движка, а не наоборот.

Влияние на инженерные аспекты игры

- *Инструментарий и процесс разработки*. На построение и обслуживание хорошего конвейера для вычисления столкновений/физики может уйти некоторое время.
- *Пользовательский интерфейс*. Каким образом пользователь управляет физическими объектами игрового мира? Он по ним стреляет? Заходит в них? Подбирает их? Держит их в виртуальных руках, как в *Trespasser*? Или использует гравитационное ружье, как в *Half-Life 2*?
- *Обнаружение столкновений*. Модели столкновений, предназначенные для применения в симуляции динамики, могут потребовать большего внимания к деталям и большей осторожности при создании, чем их аналоги, не основанные на физике.
- *ИИ*. При наличии физически симулируемых объектов прокладывание пути может быть не слишком предсказуемым. Возможно, движку придется учитывать динамические точки укрытия, которые можно перемещать или взрывать. Может ли ИИ использовать физику с пользой для себя?
- *Некорректное поведение объектов*. Небольшое пересечение анимированных объектов обычно не имеет существенного (или вообще какого-либо) отрицательного эффекта. Но если динамика объектов симулируется, они могут отскакивать друг от друга непредсказуемым образом или сильно дергаться. Возможно, стоит разрешить небольшое взаимное проникновение объектов за счет фильтрации столкновений. Или следует предусмотреть механизмы, которые будут контролировать чтобы объекты полностью останавливались и засыпали.
- *Физика тряпичной куклы*. Тряпичные куклы требуют очень тонкой настройки и часто ведут себя нестабильно во время симуляции. Из-за анимации части

тела персонажа могут пересекать другие области столкновения — когда персонаж превращается в тряпичную куклу, такое взаимопроникновение способно кардинально ухудшить стабильность. Нужно принять меры, чтобы этого избежать.

- *Графика*. Движение, основанное на физике, может повлиять на ограничивающие многогранники видимых объектов, которые по замыслу должны были быть статическими или более предсказуемыми. Наличие разрушаемых зданий и объектов может нарушить работу некоторых методов предварительного просчета теней и освещения.
- *Сетевые и многопользовательские возможности*. Физические эффекты, не влияющие на игровой процесс, можно симулировать на каждом клиентском устройстве по отдельности (и независимо). Но если физика сказывается на игровом процессе (как в случае с траекторией полета гранаты), она должна быть просчитана на сервере и точно реплицирована на всех клиентах.
- *Запись и воспроизведение*. Возможность записи игрового процесса и его последующего воспроизведения очень полезна для отладки/тестирования и может стать интересным элементом самой игры. Эту функцию сложно реализовать, так как она требует, чтобы каждая подсистема движка вела себя детерминированно и во время воспроизведения все происходило точно так, как в момент записи. Если ваша система симуляции физики не является детерминированной, это может стать большой ложкой дегтя.

Влияние на внешний вид

- *Усложнение инструментов и рабочих процессов*. Придание объектам массы, силы трения, ограничений и других атрибутов, необходимых для симуляции динамики, усложняет жизнь и художественному отделу.
- *Более сложный контент*. Нам может понадобиться несколько визуально идентичных версий объекта с разными параметрами столкновения и динамики, предназначенных для разных целей, например разрушаемая и неразрушаемая версии.
- *Потеря контроля*. Непредсказуемость физически симулируемых объектов может усложнить управление художественной композицией сцены.

Другие последствия

- *Влияние на разные аспекты игры*. Добавление в игру симуляции динамики требует тесного взаимодействия между программистами, художниками, звукоинженерами и дизайнерами.
- *Влияние на выпуск игры*. Физика может сделать разработку проекта более дорогой, создать сложности и риски, связанные с техническими и организационными моментами.

Большинство студий сейчас предпочитают использовать в своих играх системы динамики твердого тела, даже несмотря на возможные последствия. Если тщательно подходить к планированию и принимать взвешенные решения, добавление физики в игру может быть оправданным и плодотворным. И как мы увидим в дальнейшем, сторонние промежуточные механизмы делают физику доступней, чем когда-либо прежде.

13.2. Промежуточный слой для столкновений/физики

Написание систем столкновений и симуляции динамики твердого тела — непростая задача, отнимающая много времени. Подсистема столкновений/физики может составлять существенную часть исходного кода типичного игрового движка. И все это нужно написать и поддерживать!

К счастью, сейчас есть целый ряд надежных высококачественных движков столкновений/физики, как коммерческих, так и с открытым кодом. Некоторые из них перечислены далее. Обсуждение преимуществ и недостатков разных физических SDK вы найдете на игровых интернет-форумах, например www.gamedev.net/community/forums/topic.asp?topic_id=463024.

13.2.1. ODE

Как можно догадаться по названию, ODE (Open Dynamics Engine) (www.ode.org) — это SDK с открытым исходным кодом для симуляции столкновений и динамики твердого тела. По своим возможностям он похож на коммерческий продукт Havok. Среди его преимуществ можно выделить бесплатность (большой плюс для мелких игровых студий и учебных проектов!) и доступность всех исходных текстов (что существенно упрощает отладку и дает возможность модифицировать физический движок для нужд конкретной игры).

13.2.2. Bullet

Bullet — это открытая библиотека для обнаружения столкновений и симуляции физики, которая применяется как в игровой, так и в киноиндустрии. Ее движок столкновений интегрирован с симуляцией динамики, но благодаря специальным хукам его можно использовать отдельно или в сочетании с другими физическими движками. Она поддерживает *непрерывное обнаружение столкновений* (continuous collision detection, CCD), известное также как обнаружение *момента столкновения* (time of impact, TOI). Как мы вскоре увидим, это может быть чрезвычайно полезно при симуляции мелких быстро движущихся объектов. Bullet SDK можно загрузить на странице code.google.com/p/bullet/.

13.2.3. TrueAxis

TrueAxis — это еще один SDK для столкновений/физики. Он бесплатен для некоммерческого использования. Больше о TrueAxis можно узнать на сайте trueaxis.com.

13.2.4. PhysX

Библиотека PhysX изначально называлась Novodex, компания Ageia разрабатывала и распространяла этот продукт в рамках стратегии продвижения собственного физического сопроцессора. Затем компания NVIDIA купила и переписала этот проект, чтобы добиться совместимости со своими видеокартами (хотя он способен работать целиком на центральном процессоре, без графического адаптера). Он доступен по ссылке www.nvidia.com/object/nvidia_physx.html. Один из аспектов маркетинговой стратегии Ageia и NVIDIA состоял в том, что версия SDK, предназначенная для центрального процессора, должна быть полностью бесплатной, чтобы увеличить рыночную долю физического сопроцессора. За определенную плату разработчики могут получить весь исходный код и возможность модифицировать библиотеку под свои нужды. Сейчас PhysX является частью APEX — масштабируемого многоплатформенного фреймворка для симуляции динамики от NVIDIA. Проект PhysX/APEX доступен для Windows, Linux, Mac, Android, Xbox 360, PlayStation 3, Xbox One, PlayStation 4 и Wii.

13.2.5. Havok

Havok является золотым стандартом коммерческих SDK для симуляции физики. Он обладает одним из самых богатых наборов возможностей и демонстрирует отличную производительность на всех поддерживаемых платформах (это также самое дорогое решение). Havok состоит из основного движка столкновений/физики и ряда необязательных подключаемых модулей, включая систему симуляции транспортных средств, систему моделирования разрушаемых окружений и полноценный анимационный пакет, тесно интегрированный с системой физики тряпичной куклы в Havok. Эта библиотека доступна для Xbox 360, PlayStation 3, Xbox One, PlayStation 4, PlayStation Vita, Wii, Wii U, Windows 8, Android, Apple Mac и iOS. Больше о Havok можно узнать на сайте www.havok.com.

13.2.6. Physics Abstraction Layer (PAL)

Physics Abstraction Layer (PAL) — это библиотека с открытым исходным кодом, которая позволяет разработчикам использовать в одном проекте сразу несколько SDK для симуляции физики. Она предоставляет хуки для PhysX (Novodex), Newton, ODE, OpenTissue, Tokamak, TrueAxis и еще нескольких SDK. Больше о PAL можно почитать на странице www.adrianboeing.com/pal/index.html.

13.2.7. Digital Molecular Matter

Компания Pixelux Entertainment S.A., базирующаяся в Женеве (Швейцария), создала уникальный физический движок Digital Molecular Matter (DMM), который использует методы конечных элементов для симуляции динамики деформируемых тел и разрушаемых объектов. Его можно задействовать как в ходе разработки, так и на этапе выполнения. Он был выпущен в 2008 году и применялся в игре *Star Wars: The Force Unleashed* студии LucasArts. Обсуждение механики деформируемых тел выходит за рамки этой книги, но вы можете узнать больше о DMM на сайте www.pixeluxentertainment.com.

13.3. Система обнаружения столкновений

Основная задача системы обнаружения столкновений игрового движка состоит в определении, *соприкоснулись* ли те или иные объекты игрового мира. Чтобы ответить на этот вопрос, движок представляет каждый логический объект в виде одной или нескольких геометрических *фигур*. Обычно они довольно простые: сферы, параллелепипеды, капсулы и т. д. Но можно использовать и более сложные примитивы. Система столкновения определяет, *пересекаются* (то есть перекрываются) или нет те или иные фигуры в любой заданный момент времени. Это, в сущности, механизм для проверки обычных геометрических пересечений.

Конечно, система столкновений не ограничивается определением того, пересекаются фигуры или нет. Она также предоставляет полезную информацию о характеристиках каждого контакта, с помощью которой можно предотвратить появление на экране нереалистичных визуальных аномалий, таких как *взаимное проникновение* объектов. Обычно, чтобы этого добиться, перед отрисовкой следующего кадра такие объекты разводят в пространстве. Столкновения могут служить *опорой* объекта: один или несколько контактов в совокупности позволят объекту прийти в состояние покоя и достичь равновесия с силой тяжести и/или другими силами, которые на него воздействуют. Столкновения можно использовать и для других целей: например, взорвать ракету, когда она долетит до цели, или дать игроку заряд здоровья, когда он проходит через парящую в воздухе аптечку. Часто самым активным клиентом системы столкновений является движок симуляции твердого тела — с ее помощью он имитирует физически реалистичное поведение, такое как отскок, перекатывание, скольжение и переход в состояние покоя. Но даже игры без системы физики могут интенсивно задействовать механизм обнаружения столкновений.

В этой главе мы в общих чертах рассмотрим принцип работы движков обнаружения столкновений в реальном времени. Если вы хотите исследовать эту тему поглубже, ей посвящен целый ряд замечательных книг, включая [14], [48] и [11].

13.3.1. Элементы, которые могут сталкиваться между собой

Если мы хотим, чтобы конкретный логический объект игры мог сталкиваться с другими объектами, нам нужно обеспечить для него *представление столкновения*, описывающее его форму, положение и ориентацию в игровом мире. Это отдельная структура данных, не имеющая отношения к *представлению игрового процесса* объекта (к коду и данным, определяющим его роль и поведение в игре) и его визуальному представлению (это может быть экземпляр треугольного меша, поверхность с разбиением, эффект частиц или какой-то другой визуальный элемент).

В контексте обнаружения пересечений предпочтение в целом следует отдавать простым фигурам (с геометрической и математической точек зрения). Например, чтобы было легче вычислять столкновения, камень можно смоделировать в виде сферы, автомобиль может иметь форму прямоугольного параллелепипеда, а примерным представлением человеческого тела может быть набор взаимосвязанных *капсул*, похожих на те, в которые расфасовывают лекарства. В идеале к более сложным фигурам стоит прибегать, только когда упрощенное представление не позволяет достичь желаемого поведения в игре. На рис. 13.1 приведены примеры использования простых фигур для приблизительного очерчивания многогранников с целью обнаружения столкновений.

Рис. 13.1. Для приближенного описания областей столкновения игровых объектов часто применяются простые геометрические формы

В Havok для описания специальных твердых тел, которые принимают участие в обнаружении столкновений, используется термин *collidable* («такой, который может сталкиваться»). Каждый такой объект представлен экземпляром класса C++ `hkpCollidable`. В PhysX это называют *акторами* твердых объектов и применяют для их представления экземпляры класса `NxActor`. В обеих этих библиотеках элемент с возможностью столкновения содержит две порции информации: *фигуру* и *преобразование*. Фигура описывает геометрическую форму объекта, а преобразование — его положение и наклон в игровом мире. Последнее необходимо по трем причинам.

1. Строго говоря, фигура описывает лишь форму объекта, то есть это сфера, параллелепипед, капсула или какой-то другой объемный многогранник. Она может также указывать на размер объекта — радиус сферы или габариты параллелепипеда. Но центр фигуры обычно совпадает с точкой отсчета координат, а ее канонический наклон определяется относительно осей. Поэтому, чтобы она имела подходящие местоположение и наклон в глобальном пространстве, ее сначала необходимо преобразовать.
2. Многие объекты в игре являются динамическими. Перемещение фигуры произвольной сложности в пространстве может отрицательно сказаться на производительности, если при этом учитывать все ее *элементы* (вершины, плоскости и т. д.) по отдельности. Но благодаря преобразованию этот процесс не требует большого количества ресурсов независимо от того, насколько простой или сложной является фигура.
3. Описание некоторых более сложных видов фигур может занимать довольно много места в памяти. Поэтому будет полезно, если несколько объектов смогут использовать одно и то же описание фигуры. Например, в гоночной игре многие автомобили имеют идентичную информацию о форме. В этом случае все они могут использовать одну и ту же фигуру для описания автомобиля.

У каждого конкретного игрового объекта может быть один элемент для столкновений (если это простое твердое тело), несколько таких элементов (каждый из которых, к примеру, представляет один твердый компонент роботизированной руки на шарнирах) или даже ни одного (если ему не нужны функции обнаружения столкновений).

13.3.2. Мир столкновений/физики

Для отслеживания всех элементов с поддержкой столкновений обычно применяется структура данных под названием *мир столкновений*. Это полное представление игрового мира, созданное специально для использования системой обнаружения столкновений. В Havok это экземпляр класса `hkpWorld`, а в PhysX — `NxScene`. В ODE для этого предусмотрен класс `dSpace` (на самом деле это корень

иерархии геометрических форм, которая представляет все игровые фигуры, способные сталкиваться).

Хранение всей информации о столкновениях в приватной структуре данных имеет ряд преимуществ по сравнению с применением для этого самих игровых объектов. К примеру, игровой мир должен содержать элементы collidable только для тех объектов игры, которые теоретически могут столкнуться друг с другом. Благодаря этому система столкновений может игнорировать любые структуры данных, которые ее не касаются. Кроме того, такой подход позволяет максимально эффективно организовать данные о столкновениях. Система столкновений, например, может воспользоваться преимуществами, которые дает когерентность кэша, чтобы добиться наилучшей производительности. Мир столкновений также служит эффективным механизмом инкапсуляции, который в целом помогает сделать код более понятным, удобным для поддержки и тестирования, а также потенциально пригодным для повторного использования.

Мир физики

Система динамики твердого тела, если таковая присутствует в игре, обычно тесно интегрирована с системой столкновений. Обычно обе эти системы задействуют общую структуру данных мира, а твердое тело в симуляции, как правило, связано с отдельным элементом collidable системы столкновений. Этот подход традиционен для физических движков, так как системе физики необходимо часто запрашивать подробную информацию о столкновениях. Система столкновений обычно работает в пассивном режиме, принимая *инструкции* от системы физики. Последняя просит ее выполнить проверку на столкновение как минимум один, а иногда и несколько раз на каждом этапе симуляции. В связи с этим мир столкновений часто называют *миром столкновений/физики* или просто *миром физики*.

Каждое динамическое твердое тело в физической симуляции обычно связано с отдельным объектом collidable в системе столкновений, хотя не всем объектам collidable нужны динамические твердые тела. Например, в Havok каждое твердое тело представлено экземпляром класса `hkpRigidBody` и содержит указатель на одну копию `hkpCollidable`. В PhysX концепции collidable и твердых тел совмещены в едином классе `NxActor`, хотя физические свойства твердого тела хранятся отдельно, в экземпляре `NxBodyDesc`. Оба SDK позволяют зафиксировать положение и наклон твердого тела в пространстве, в результате оно будет служить лишь объектом collidable и не станет участвовать в симуляции динамики.

Несмотря на тесную интеграцию, большинство физических SDK по крайней мере пытаются отделить библиотеку столкновений от симуляции динамики твердого тела. Это позволяет использовать систему столкновений в качестве самостоятельной библиотеки (что важно для игр, которым не нужна физика, но которые должны обнаруживать столкновения). Это также означает, что игровые студии *теоретически* могут полностью заменить систему столкновений из физического SDK, не переписывая при этом симуляцию динамики (на практике это может оказаться не так просто!).

13.3.3. Концепции геометрических форм

Знакомая всем концепция *формы* предмета (см. https://en.wikipedia.org/wiki/Shape и ru.wikipedia.org/wiki/Форма_предмета) имеет под собой прочную математическую основу. В нашем контексте под формой можно понимать область пространства с некой границей, которая четко отделяет то, что *внутри*, от того, что *снаружи*. В двухмерном пространстве форма имеет площадь, а ее граница описывается кривой линией или как минимум тремя прямыми гранями (в последнем случае это *полигон*). В трехмерном пространстве у формы есть объем, а ее граница имеет вид либо изогнутой поверхности, либо набора полигонов (во втором случае она называется *многогранником*).

Важно отметить, что некоторые виды игровых объектов, такие как ландшафт, реки или тонкие стены, лучше всего представлять в виде *поверхностей*. В трехмерном пространстве поверхность является двухмерным геометрическим элементом с *передней* и *задней* частями, но без внутренней и наружной частей. В качестве примеров можно привести плоскости, треугольники, поверхности с разбиением и поверхности, состоящие из набора соединенных треугольников или других полигонов. Большинство SDK для обнаружения столкновений предоставляют поддержку стандартных поверхностей и считают их разновидностью *формы* наряду с закрытыми областями.

Библиотеки столкновений традиционно позволяют придавать поверхностям объем с помощью опционального параметра вытеснения. Этот параметр определяет, насколько толстой должна быть поверхность. Это позволяет избежать случаев, когда не удается обнаружить столкновения между мелкими, быстро движущимися объектами и бесконечно тонкими поверхностями (это так называемая проблема пролета пули сквозь бумагу — см. подраздел 13.3.5).

Пересечение

Наверное, любой интуитивно понимает, что такое *пересечение*. Этот термин позаимствован из теории множеств (ru.wikipedia.org/wiki/Пересечение_множеств). Пересечение двух множеств состоит из подмножества элементов, принадлежащих обоим множествам. В геометрии пересечение двух фигур — это просто бесконечно большое множество всех точек, находящихся внутри каждой из них.

Контакт

В играх нас обычно не интересует нахождение пересечения в самом строгом смысле как набора точек. Мы просто хотим знать, пересекаются два объекта или нет. В случае столкновения соответствующая система обычно предоставляет дополнительную информацию о природе контакта. Это, к примеру, позволяет развести объекты физически правдоподобным и эффективным образом.

Система столкновений обычно упаковывает информацию о контакте в удобную структуру данных, экземпляры который можно создавать для каждого обнаруженного контакта. Например, Havok возвращает контакты в виде экземпляров класса `hkContactPoint`. Часто такая информация содержит *разделяющий вектор*, вдоль

которого можно развести объекты, чтобы эффективно вывести их из состояния столкновения. Обычно она включает в себя список элементов collidable, вошедших в контакт, с описанием отдельных пересекшихся фигур и, возможно, даже сведениями о том, какие именно части этих фигур соприкоснулись. Система может вернуть дополнительные параметры, такие как скорость и направление движения тел, спроецированные на разделяющую нормаль.

Выпуклость

Одной из важнейших концепций в области обнаружения столкновений является разделение между *выпуклыми* и *невыпуклыми* (то есть *вогнутыми*) формами. Строго говоря, форма называется выпуклой, если ни один луч, выпущенный изнутри нее, не может пересечь ее поверхность больше одного раза. Представьте, что вы заворачиваете форму в целлофан: если она выпуклая, под целлофаном не останется участков, заполненных воздухом. Таким образом, в двухмерном пространстве окружности, прямоугольники и треугольники выпуклые, а персонаж Pac Man — нет. То же самое справедливо и для трехмерного пространства.

Как мы вскоре увидим, выпуклость имеет большое значение, так как обнаружение пересечений между выпуклыми формами обычно является более простым и эффективным с точки зрения вычислений, чем между вогнутыми. Подробнее о выпуклых формах можно почитать на странице https://ru.wikipedia.org/wiki/Выпуклое_множество.

13.3.4. Примитивы столкновения

Системы столкновений обычно умеют работать с довольно ограниченным набором фигур. Иногда их называют *примитивами столкновения*, поскольку они выступают элементарными строительными блоками, из которых можно сформировать более сложные формы. В этом разделе кратко рассмотрим некоторые из наиболее распространенных разновидностей примитивов столкновения.

Сферы

Простейшей трехмерной областью является сфера. И, как можно было бы ожидать, это самый эффективный примитив столкновения. Сфера описывается центральной точкой и радиусом. Эту информацию удобно упаковать в четырехэлементный вектор с плавающей запятой — формат, который особенно хорошо подходит для математических библиотек на основе SIMD.

Капсулы

Капсула состоит из цилиндра с двумя полусферами на концах. Это форма, которую можно получить при перемещении сферы из точки A в точку B (хотя между статической капсулой и сферой, движение которой формирует капсульную форму, есть важные различия, поэтому сравнение неидеально). Капсулы часто описываются в виде двух точек и радиуса (рис. 13.2). Капсулы являются более эффективными для пересечения, чем цилиндры или параллелепипеды, поэтому их часто исполь-

зуют для моделирования объектов цилиндрической или близкой к тому формы, таких как конечности человеческого тела.

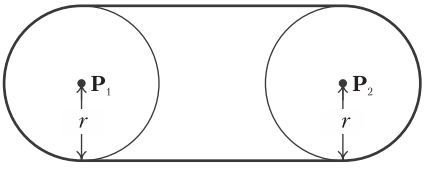


Рис. 13.2. Капсулу можно представить в виде двух точек и радиуса

Ограничивающие параллелепипеды, выровненные по осям

Ограничивающий параллелепипед, выровненный по осям (axis-aligned bounding box, AABB), — это прямоугольная область (ее правильнее называть *кубоидом*), чьи грани расположены параллельно осям координат. Конечно, выравнивание параллелепипеда по осям одной системы координат вовсе не гарантирует, что он будет выровнен в другой. Поэтому об AABB можно говорить лишь в контексте одной конкретной системы, в которой он выравнивался.

AABB может быть удобно определен двумя точками: одна содержит минимальные координаты прямоугольника вдоль каждой из трех главных осей, а другая содержит максимальные координаты (рис. 13.3).

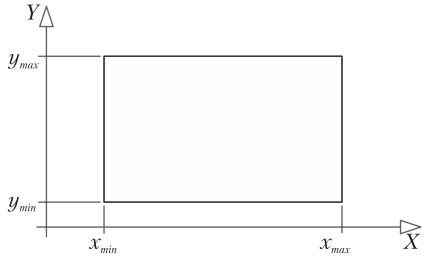


Рис. 13.3. Параллелепипед, выровненный по осям

Основным преимуществом параллелепипедов, выровненных по осям координат, является то, что их пересечение с другими AABB очень легко определить. Огромный недостаток использования AABB связан с тем, что они должны все время оставаться выровненными по осям, иначе утрачивается их вычислительная эффективность. Это означает, что AABB, применяемый для приближенного описания формы игрового объекта, приходится заново вычислять каждый раз, когда этот объект меняет наклон. Даже если объект похож на параллелепипед, в случае его внеосевого вращения AABB плохо справляется с описанием его формы (рис. 13.4).

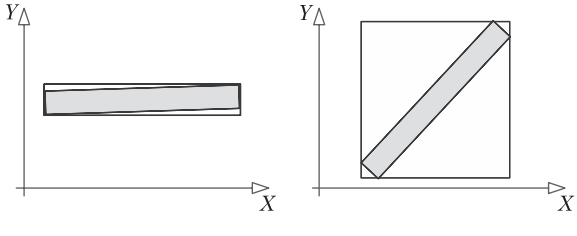


Рис. 13.4. AABB точное описывает прямоугольный объект, только если главные оси последнего строго выровнены по осям координат

Направленные ограничивающие параллелепипеды

Если разрешить AABB вращаться относительно его системы координат, получится так называемый направленный ограничивающий параллелепипед (oriented bounding box, OOB). Он часто представлен в виде трех половинных измерений (полуширины, полуглубины и полувысоты) и преобразования, которое определяет его положение и наклон относительно осей координат. OOB — распространенный примитив столкновения, поскольку он лучше вмещает в себя объекты с произвольным наклоном и при этом его довольно легко описать.

Дискретные ориентированные многогранники

Дискретный ориентированный многогранник (discrete oriented polytope, DOP) — это более общая разновидность AABB и OBB. Он имеет выпуклую форму, которая примерно повторяет форму объекта. Чтобы сформировать DOP, можно взять несколько бесконечных плоскостей и сместить их вдоль векторов их нормалей таким образом, чтобы они прилегали к объекту, форму которого нужно описать. AABB — это DOP из шести плоскостей (6-DOP), нормали которых лежат параллельно осям координат. OBB — это тоже 6-DOP, в котором нормали плоскостей параллельны основным естественным осям объекта. *k*-DOP состоит из произвольного количества плоскостей *k*. Формирование DOP часто начинают с OOB нужного объекта, затем с помощью дополнительных плоскостей грани и/или углы OBB обрезают под углом 45°, чтобы получить более близкую форму. Пример *k*-DOP показан на рис. 13.5.

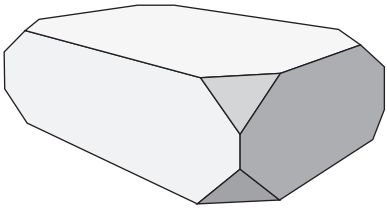


Рис. 13.5. OBB, у которого обрезаны все восемь углов, можно назвать 14-DOP

Произвольно выпуклые области

Большинство движков столкновений позволяют 3D-художнику описывать произвольно выпуклые области в таких редакторах, как Maya. Художник создает форму из полигонов (треугольников или квадов). Чтобы убедиться в том, что треугольники формируют выпуклый многогранник, их анализируют на этапе разработки с помощью какого-то инструмента. Если форма проходит проверку на выпуклость, ее треугольники преобразуются в набор плоскостей (по сути, в *k*-DOP), представленных *k* уравнениями или сочетанием *k* точек и *k* векторов нормалей (если форма оказалась невыпуклой, ее можно представить в виде полигонального супа, о котором пойдет речь в следующем разделе) (рис. 13.6).

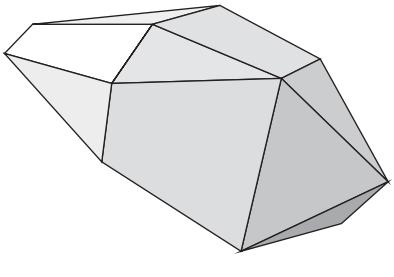


Рис. 13.6. Произвольно выпуклую область можно представить в виде набора пересекающихся плоскостей

Для проверки пересечения выпуклых областей требуется больше ресурсов, чем для более простых геометрических примитивов, которые мы обсуждали до сих пор. Но, как будет показано в подразделе 13.3.5, к ним можно применять высокоэффективные алгоритмы поиска пересечений вроде GJK, рассчитанные на выпуклые формы.

Полигональный суп

Некоторые системы столкновений поддерживают абсолютно произвольные невыпуклые формы, которые обычно состоят из треугольников или других простых полигонов. В связи с этим такого рода формы часто называют *полигональными супами*. Полигональные супы часто применяют для моделирования сложной статической геометрии, такой как ландшафт или здания (рис. 13.7).

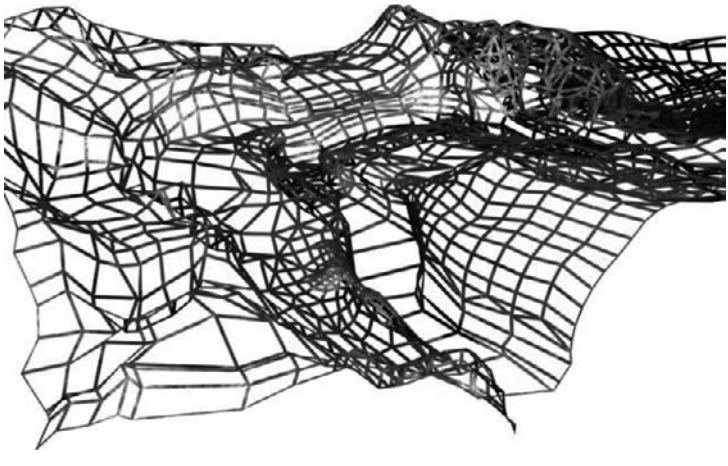


Рис. 13.7. Полигональный суп часто используется для моделирования сложных статических поверхностей, таких как ландшафт или здания

Как можно представить, обнаружение столкновений с полигональным супом — наиболее ресурсоемкая разновидность подобных проверок. Движок столкновений должен проверить каждый отдельный треугольник и правильно отреагировать на ложные срабатывания при пересечении с общими гранями смежных треугольников. В итоге в большинстве игровых проектов полигональный суп стараются применять только к объектам, которые не принимают участия в симуляции динамики.

Есть ли у полигонального супа внутренняя часть? В отличие от выпуклых и простых форм полигональный суп может представлять не только объем, но и открытую поверхность. Хотя часто его форма содержит недостаточно информации для того, чтобы система столкновений могла определить разницу между открытой поверхностью и замкнутым объемом. Из-за этого бывает сложно понять, в каком направлении необходимо послать объект, пересекшийся с полигональным супом, чтобы вывести его из состояния столкновения.

К счастью, эту проблему нельзя назвать неразрешимой. У каждого треугольника в полигональном супе есть передняя и задняя части, которые определяются порядком обхода его вершин. Поэтому полигональному супу можно специально придать такую форму, чтобы порядок обхода вершин всех полигонов был согласованным, то есть сделать так, чтобы смежные треугольники всегда имели одинаковое «направление». Таким образом, понятия «перед» и «зад» распространяются на весь полигональный суп. Мы также можем хранить информацию о том, открытый он или замкнутый (если предположить, что это можно установить еще на этапе разработки). Так что в контексте замкнутой формы перед и зад можно интерпретировать как «снаружи» и «внутри» или наоборот — в зависимости от того, какими правилами вы руководствовались при построении полигонального супа.

Мы также можем имитировать внутреннюю и внешнюю части некоторых *открытых* полигональных супов, то есть поверхностей. Например, если ландшафт в игре представлен в виде открытого полигонального супа, мы можем просто решить, что передняя часть поверхности всегда направлена в противоположную от Земли сторону. Из этого следует, что перед всегда должен совпадать с внешней частью. Чтобы реализовать это на практике, нам, вероятно, пришлось бы каким-то образом модифицировать движок столкновений, чтобы он понимал установленные нами правила.

Составные формы

Если для адекватного описания формы объекта недостаточно одной фигуры, мы можем использовать *набор* фигур. Например, стул можно смоделировать из двух параллелепипедов: одного для спинки, другого для сиденья и всех четырех ножек (рис. 13.8).

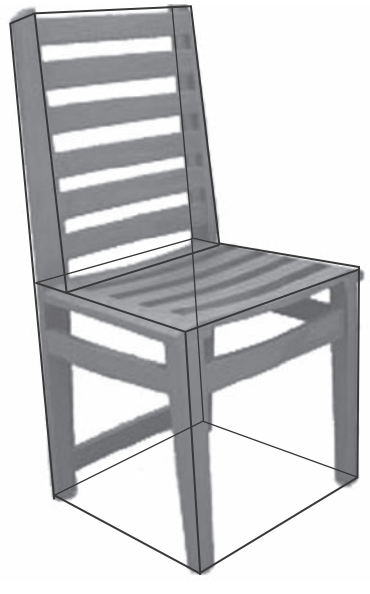


Рис. 13.8. Стул можно смоделировать с помощью двух взаимосвязанных параллелепипедов

В случае с моделированием невыпуклых объектов составная форма часто служит более эффективной альтернативой полигональному супу, две и более выпуклые области часто дают лучшую производительность, чем один полигональный суп. Более того, некоторые системы столкновений при проверке на пересечение могут использовать выпуклую ограничивающую область составной формы целиком. В Havok это называют *среднефазным* (midphase) обнаружением столкновений. Как показано на рис. 13.9, система столкновений сначала проверяет выпуклые ограничивающие области двух составных форм. Если они не пересекаются, системе не нужно проверять на столкновение отдельные внутренние формы.

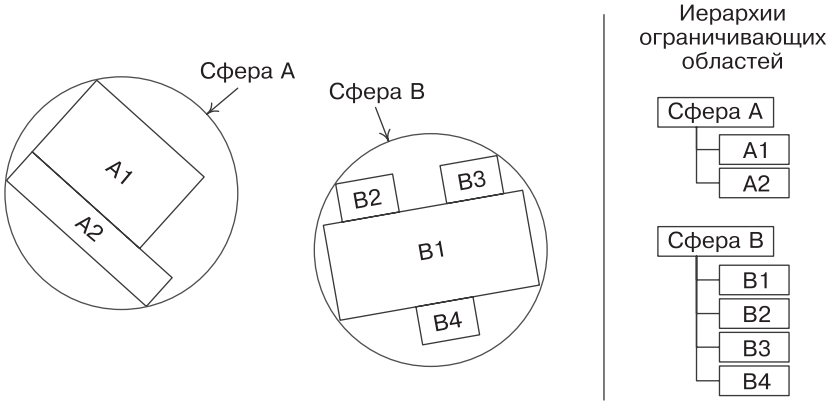


Рис. 13.9. Системе столкновений нужно проверять отдельные элементы | двух составных форм, только если обнаружится, что их выпуклые ограничивающие области (в данном случае сферы A и B) пересекаются

13.3.5. Проверки на столкновение и аналитическая геометрия

Система столкновений использует вычисления на основе *аналитической геометрии* (математического описания трехмерных областей и поверхностей) для обнаружения пересечения двух фигур. Больше об этой глубокой и обширной области исследований можно почитать на странице ru.wikipedia.org/wiki/Аналитическая_геометрия. В этом разделе мы познакомимся с концепциями, которые стоят за аналитической геометрией, рассмотрим несколько общих примеров и затем обсудим обобщенный алгоритм проверки на пересечение GJK для произвольно выпуклых многогранников.

Точка относительно сферы

Чтобы определить, находится ли точка **p** внутри сферы, можно построить разделяющий вектор **c** между ней и центром сферы и затем проверить его длину. Если он длиннее радиуса сферы *r*, точка находится за пределами сферы, а если нет, то внутри нее:

$$\mathbf{s} = \mathbf{c} - \mathbf{p}.$$

Если $|\mathbf{s}| \leq r$, то **p** находится внутри.

Одна сфера относительно другой

Определить, пересекаются ли две сферы, почти так же просто, как и в случае со сферой и точкой. Мы снова строим вектор **s**, который на этот раз соединяет центры двух сфер, измеряем его длину и сравниваем ее с суммой радиусов двух сфер.

Если длина вектора меньше суммы радиусов или равна ей, сферы пересекаются, в противном случае пересечения нет:

$$\mathbf{s} = \mathbf{c}_1 - \mathbf{c}_2, \tag{13.1}$$

если $|\mathbf{s}| \le (r_1 + r_2)$, то сферы пересекаются.

Чтобы избежать операции вычисления квадратного корня, которая требуется для поиска длины вектора **s**, мы можем возвести в квадрат все уравнение. В итоге (13.1) превратится в:

$$\mathbf{s} = \mathbf{c}_1 - \mathbf{c}_2;$$

$$|\mathbf{s}|^2 = \mathbf{s} \cdot \mathbf{s}.$$

Если $\mathbf{s}|^2 \le (r_1 + r_2)^2$, то сферы пересекаются.

Теорема о разделяющей линии[1]

Большинство систем определения столкновений активно используют так называемую *теорему о разделяющей гиперплоскости* (https://en.wikipedia.org/wiki/Hyperplane_separation_theorem и ru.wikipedia.org/wiki/Теорема_об_опорной_гиперплоскости). Она гласит, что, если можно найти такую ось, *проекции* двух выпуклых форм на которую не пересекаются, это означает, что не пересекаются сами формы. Если такой оси не существует *и* если формы выпуклые, мы можем быть уверены в том, что они пересекаются (если формы вогнутые, они могут не пересекаться, даже несмотря на отсутствие разделяющей оси; это одна из причин, почему при обнаружении столкновений мы чаще всего отдаем предпочтение выпуклым формам).

Проще всего эту теорему проиллюстрировать в двухмерном пространстве. Она, в сущности, утверждает, что, если можно найти такую линию, по одну сторону которой целиком находится объект A, а по другую — объект B, это означает, что объекты A и B не пересекаются. Эта линия называется *разделяющей*, и разделяющая ось всегда ей *перпендикулярна*. Если найти разделяющую линию, очень легко убедиться в том, что теорема действительно верна: достаточно лишь взглянуть на проекции форм на ось, перпендикулярную разделяющей линии.

Проекция двухмерной *выпуклой* формы на ось походит на тень, которую объект отбрасывает на тонкий провод. Это всегда отрезок, который соответствует максимальной протяженности объекта в направлении оси. Мы также можем представить себе проекцию в виде минимальной и максимальной координат на оси, что можно записать как полностью замкнутый интервал $[c_{min}, c_{max}]$. Как можно видеть на рис. 13.10, когда между двумя формами существует разделяющая линия, их проекции на разделяющей оси не перекрываются. Однако они могут перекрываться на других, неразделяющих осях.

---

[1] Теорема о плоскости, но здесь в основном речь идет о двухмерном пространстве, поэтому используется термин «линия» (можно также «прямая»). Терминология здесь применяется очень свободно (дальше автор называет это теоремой о разделяющей оси). — *Примеч. пер.*

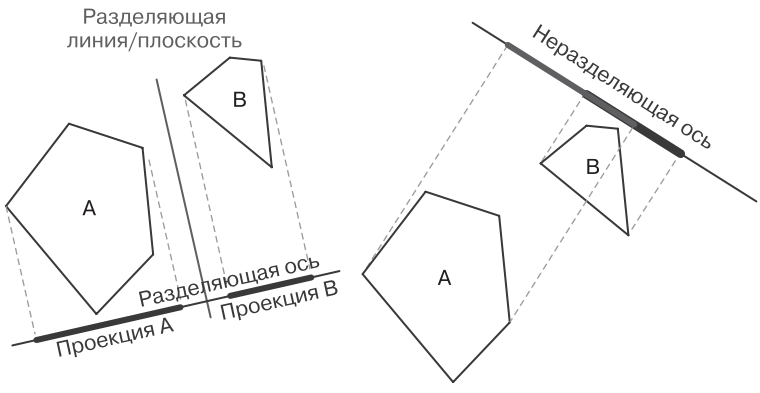


Рис. 13.10. Проекции двух фигур на разделяющую ось всегда имеют вид двух непересекающихся отрезков. Проекции этих же фигур на неразделяющую ось могут пересекаться. Если разделяющей оси не существует, фигуры пересекаются

В трехмерном пространстве разделяющая линия превращается в плоскость, но разделяющая ось так и остается осью, то есть бесконечной линией. Опять же проекция трехмерной *выпуклой* формы на ось является отрезком прямой, который можно представить в виде полностью замкнутого интервала $[c_{min}, c_{max}]$.

Некоторые разновидности форм обладают свойствами, которые делают выбор разделяющих осей очевидным. Чтобы обнаружить пересечение между формами A и B, мы можем последовательно спроецировать их на каждую возможную разделяющую ось и проверить, пересекаются (то есть перекрываются) два спроецированных отрезка $\left[c_{A\,min}, c_{A\,max}\right]$ и $\left[c_{B\,min}, c_{B\,max}\right]$ или нет. С математической точки зрения отрезки не пересекаются, если $c_{A\,max} < c_{B\,min}$ или $c_{B\,max} < c_{A\,min}$. Если интервалы проекций на одной из возможных разделяющих осей не пересекаются, это означает, что мы нашли разделяющую ось и знаем наверняка, что две формы не пересекаются.

Одним из практических примеров этого принципа является проверка на пересечение двух сфер. Если пересечения нет, ось, параллельная прямому отрезку, соединяющему центральные точки сфер, всегда будет корректной разделяющей осью (и, возможно, не единственной — все зависит от того, насколько далеко друг от друга находятся две сферы). Представьте себе предел, при котором две сферы почти касаются друг друга, но еще не вошли в контакт. В этом случае *единственной* разделяющей осью будет та, которая проходит параллельно прямой, соединяющей центры. По мере расхождения сфер мы можем поворачивать ось все сильнее и сильнее в любом направлении (рис. 13.11).

AABB относительно AABB

Чтобы определить, пересекаются ли два AABB, мы можем опять применить теорему о разделяющей оси. То, что грани обоих AABB гарантированно параллельны общему набору осей координат, говорит о том, что разделяющая ось, если таковая существует, будет одной из этих осей координат.

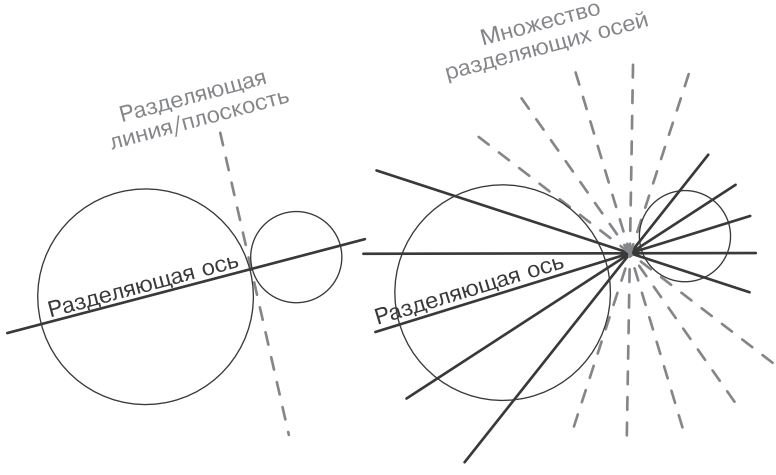


Рис. 13.11. Когда две сферы находятся на бесконечно малом расстоянии друг от друга, единственная разделяющая ось проходит параллельно отрезку, соединяющему центральные точки этих двух сфер

Таким образом, чтобы проверить два AABB (назовем их A и B) на пересечение, мы просто анализируем минимальные и максимальные координаты двух параллелепипедов отдельно вдоль каждой оси. По оси *X* у нас есть два отрезка, $\left[x_{\text{A min}}, x_{\text{A max}}\right]$ и $\left[x_{\text{B min}}, x_{\text{B max}}\right]$, соответствующие отрезки существуют и по осям *Y* и *Z*. Если они пересекаются во *всех трех случаях*, значит, два AABB пересекаются, в противном случае пересечения нет. Примеры пересекающихся и непересекающихся AABB показаны на рис. 13.12 (для наглядности мы сделали их плоскими). Углубленное обсуждение пересечений AABB вы найдете на странице www.gamasutra.com/features/20000203/lander_01.htm.

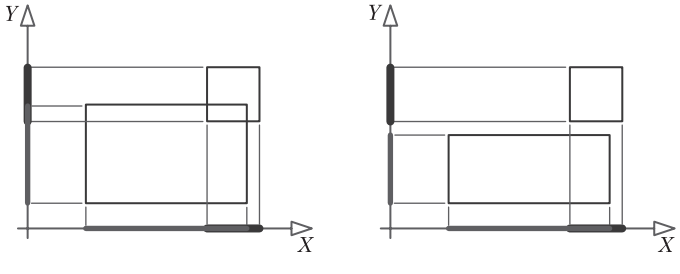


Рис. 13.12. Двухмерный пример двух пересекающихся и непересекающихся AABB. Обратите внимание на то, что вторая пара AABB пересекается только по оси *X*, но не по оси *Y*

Обнаружение столкновений выпуклых форм: алгоритм GJK

Для обнаружения пересечений произвольных выпуклых *многогранников* (в двухмерном пространстве речь идет о выпуклых полигонах) существует очень эффективный алгоритм, известный как GJK. Он назван в честь создателей, Гилберта,

Джонсона и Кирти из Мичиганского университета. Этому алгоритму и его разновидностям посвящено много статей, включая оригинальную (ieeexplore.ieee.org/document/2083?arnumber=2083), а также две отличные презентации в формате PowerPoint: одну представил Кристер Эриксон на конференции SIGGRAPH (realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt), а другую создал Джино ван ден Берген (www.laas.fr/~nic/MOVIE/Workshop/Slides/Gino.vander.Bergen.ppt). Но, наверное, самым простым и интересным описанием этого алгоритма является учебное видео Implementing GJK (автор Кейси Муратори), доступное по адресу mollyrocket.com/849. Эти материалы настолько хороши, что здесь я изложу лишь саму суть GJK, а подробности вы можете найти в названных источниках.

Алгоритм GJK основан на геометрической операции, известной как *разность Минковского*. Несмотря на необычное название, она довольно проста: мы берем каждую точку внутри формы B и последовательно вычитаем ее из каждой точки внутри формы A. Полученное множество точек $\{(\mathbf{A}_i - \mathbf{B}_j)\}$ и будет разностью Минковского.

У разности Минковского есть одно полезное свойство: если применить ее к двум выпуклым формам, она будет *содержать начало координат* тогда и только тогда, когда эти две формы пересекаются. Доказательство этого утверждения выходит за рамки данной книги, но мы можем интуитивно понять, почему оно является верным: просто помните, что под пересечением форм A и B на самом деле имеется в виду, что внутри A есть такие точки, которые находятся *также* внутри B. Вычитая каждую точку, составляющую B, из каждой точки, составляющей A, мы можем ожидать, что в какой-то момент нам попадется точка, общая для двух форм. В таком случае вычитание даст ноль, поэтому разность Минковского будет содержать начало координат тогда и только тогда, когда сферы A и B имеют общие точки (рис. 13.13).

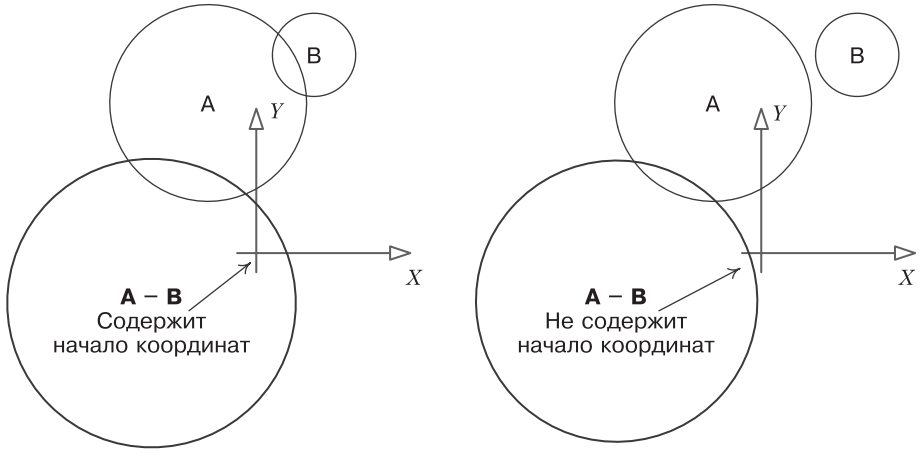


Рис. 13.13. Разность Минковского для двух пересекающихся выпуклых форм содержит начало координат. Если формы не пересекаются, начало координат не входит в разницу Минковского

Разность Минковского для двух выпуклых форм сама является выпуклой формой. Но нас интересует лишь ее *оболочка*, а не все ее внутренние точки. Основной принцип работы GJK состоит в поиске *тетраэдра* (то есть формы с четырьмя вершинами, созданной из треугольников), размещенного на оболочке разности Минковского и включающего в себя начало координат. Если такую фигуру можно найти, формы пересекаются, в противном случае — нет.

Тетраэдр — это частный случай геометрического объекта, известного как *симплекс*. Но пусть вас не смущает необычное название — это всего лишь набор точек. Симплекс, состоящий из одной точки, является точкой, симплекс из двух точек — это отрезок прямой, симплекс из трех точек — треугольник, а симплекс из четырех точек — тетраэдр (рис. 13.14).

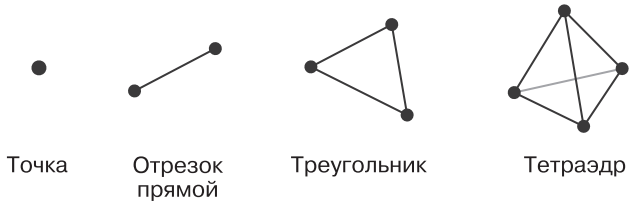


Рис. 13.14. Симплекс с одной, двумя, тремя и четырьмя точками

GJK является итеративным алгоритмом, который начинается с одноточечного симплекса, размещенного в произвольном месте внутри оболочки разности Минковского. Он постепенно пытается построить симплексы более высокого порядка, потенциально содержащие начало координат. На каждой итерации цикла мы берем текущий симплекс и смотрим, в каком направлении относительно него находится начало координат. Затем находим в этом направлении *вспомогательную вершину* разности Минковского, то есть вершину выпуклой оболочки, которая находится ближе всего к началу координат в выбранном направлении. Мы добавляем эту новую точку в симплекс, повышая тем самым его порядок (то есть точка становится отрезком прямой, отрезок — треугольником, а треугольник превращается в тетраэдр). Если в результате добавления этой точки полученный симплекс включает в себя начало координат, на этом можно заканчивать: мы знаем, что две формы пересекаются. В то же время, если не удается найти вспомогательную точку, которая находится ближе к началу координат, чем к симплексу, это означает, что мы туда никогда не дойдем, следовательно, две формы *не* пересекаются (рис. 13.15).

Чтобы как следует разобраться в алгоритме GJK, вам нужно ознакомиться со статьями и видео, которые я назвал ранее. Надеюсь, что представленное здесь описание подтолкнет вас к более глубокому исследованию. По крайней мере вы теперь можете впечатлить своих друзей на вечеринке, упомянув аббревиатуру GJK (но лучше не пытайтесь проделать то же самое на собеседовании, если не разобрались в этом алгоритме как следует!).

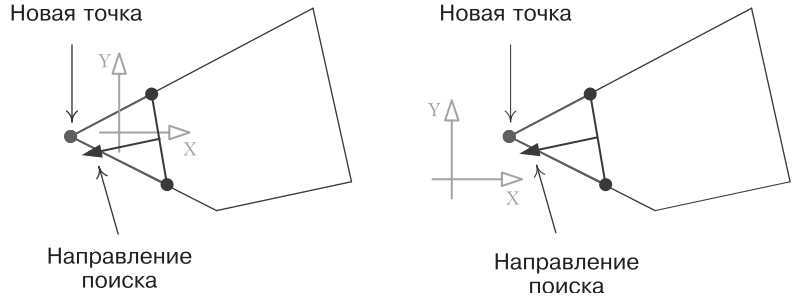


Рис. 13.15. Если при добавлении точки к текущему симплексу возникает новая форма, содержащая начало координат, это означает, что формы пересекаются. Если нет такой вспомогательной точки, которая могла бы приблизить симплекс к началу координат, формы не пересекаются. Это суть алгоритма GJK

Пересечение других фигур

Мы не станем рассматривать пересечение других фигур, так как эта тема хорошо раскрыта в других книгах, таких как [11], [14] и [48]. Ключевым моментом здесь является то, что таких комбинаций великое множество. На самом деле для N разных форм потребовалось бы $O(N^2)$ попарных проверок. Своей сложностью движок столкновений в значительной степени обязан количеству возможных вариантов пересечений, которые необходимо учитывать. Это одна из причин, почему авторы таких движков обычно пытаются ограничить число разных примитивов — это существенно сужает диапазон столкновений, которые нужно уметь обнаруживать (а также объясняет популярность алгоритма GJK: он способен одним махом обнаруживать столкновения фигур *любой* формы, меняться от одной формы к другой может лишь *вспомогательная функция*, используемая в алгоритме).

Эта задача имеет и практический аспект: как написать код, который выбирает подходящую функцию для проверки на столкновение двух произвольных форм? Многие движки столкновений применяют метод *двойной диспетчеризации* (en.wikipedia.org/wiki/Double_dispatch). В случае единичной диспетчеризации (то есть при использовании виртуальных функций) то, какую именно абстрактную функцию следует вызывать во время выполнения, зависит от типа одного объекта. При двойной диспетчеризации концепция с виртуальными функциями расширяется до типов двух объектов. Это можно реализовать в виде двухмерной поисковой таблицы, ключами в которой выступают типы проверяемых объектов. В качестве альтернативы можно сделать так, чтобы виртуальная функция на основе типа объекта A вызывала вторую виртуальную функцию на основе типа объекта B.

Рассмотрим пример из реальной жизни. Для обработки отдельных сценариев пересечения в Havok задействуются объекты, которые называют *агентами столкновений* (они наследованы от класса `hkpCollisionAgent`). В число классов агентов входят `hkpSphereSphereAgent`, `hkpSphereCapsuleAgent`, `hkpGskConvexConvexAgent` и т. д.

Для определения типа агента используется двухмерная таблица диспетчеризации, управляемая классом `hkpCollisionDispatcher`. Ожидаемо, задача этого класса состоит в том, чтобы быстро найти подходящий тип агента для заданной пары объектов collidable, которые проверяются на столкновение, и затем вызвать его с этими двумя объектами в качестве аргументов.

Обнаружение столкновений между движущимися телами

До сих пор мы рассматривали только статические проверки на пересечение двух неподвижных объектов. Когда объекты движутся, все усложняется. Движение в играх обычно симулируется в дискретные моменты времени. Простое решение заключается в том, чтобы учитывать положение и наклон твердых тел в каждый отдельный момент и применять статические проверки на пересечение для каждого снимка мира столкновений. Этот подход работает до тех пор, пока объекты не начинают двигаться слишком быстро относительно собственных габаритов. На самом деле этот метод дает настолько хорошие результаты, что во многих движках столкновений/физики, включая Havok, он используется по умолчанию.

Однако для мелких быстро движущихся объектов этот подход не годится. Представьте себе объект, который движется так быстро, что расстояние, пройденное им между двумя дискретными точками во времени, *превышает его собственные размеры*, измеренные в направлении движения. Если положить на плоскость два соседних снимка игрового мира, можно заметить, что теперь между изображениями быстро движущегося объекта есть промежуток. И если в него попадет какой-то другой объект, мы пропустим столкновение с ним. Эта проблема, иллюстрируемая на рис. 13.16, называется «пуля сквозь бумагу» и известна также как «туннельный эффект». В следующих разделах описывается несколько способов преодоления этой проблемы.

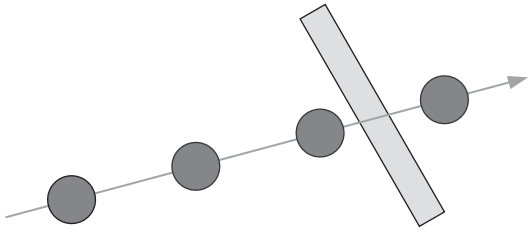


Рис. 13.16. Небольшой быстро движущийся объект способен оставлять промежутки в траектории движения между соседними снимками мира столкновений. Это означает, что столкновение можно упустить

Вытянутые фигуры. Чтобы избежать туннельного эффекта, можно использовать *вытянутые фигуры*. Чтобы вытянуть фигуру, мы постепенно перемещаем ее из одной точки в другую. Например, вытянутая сфера представляет собой капсулу, а вытянутый треугольник — треугольную призму (рис. 13.17).

Вместо статических снимков мира столкновений мы можем проверять на пересечение вытянутые фигуры, полученные при изменении положения и наклона между двумя соседними снимками. Этот подход сводится к *линейной интерполяции* движе-

ния объектов collidable между снимками, поскольку фигуры обычно вытягиваются вдоль прямых отрезков между текущим и предыдущим снимками.

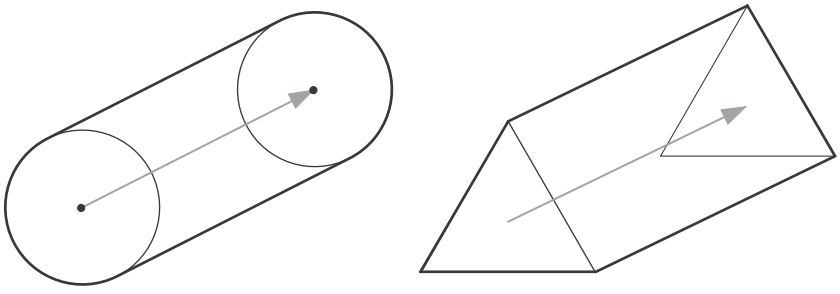


Рис. 13.17. Вытянутая сфера становится капсулой, а вытянутый треугольник превращается в треугольную призму

Конечно, линейная интерполяция может плохо описывать траекторию быстро движущихся объектов collidable. Если объект движется по изогнутой траектории, мы теоретически должны вытягивать его форму вдоль этой кривой. К сожалению, выпуклая форма, вытянутая таким образом, перестает быть выпуклой, что может сделать проверку на столкновение куда более сложной и требовательной к ресурсам.

Кроме того, если выпуклая фигура, которую мы вытягиваем, вращается, результат может оказаться невыпуклым, даже если он вытянут вдоль прямого отрезка. Как видно на рис. 13.18, мы *всегда* можем получить выпуклую форму за счет линейной экстраполяции контуров фигур из предыдущего и текущего снимков, однако полученный результат не обязательно очень точно описывает реальное движение формы на заданном отрезке времени. Иными словами, линейная интерполяция в целом не подходит для вращающихся объектов. Поэтому, если вращение допускается, проверка пересечения вытянутых фигур становится намного более сложной и требовательной к ресурсам, чем ее статический аналог, основанный на снимках.

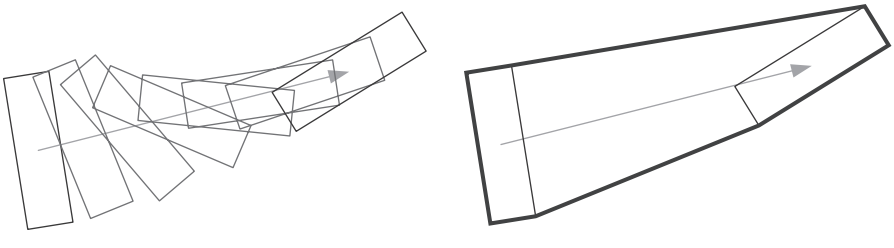


Рис. 13.18. Вращающийся объект, вытянутый вдоль прямого отрезка, может не быть выпуклым *(слева)*. Линейная интерполяция движения позволяет получить выпуклую форму *(справа)*, но она может довольно неточно описывать то, что произошло на временном отрезке

С помощью вытянутых фигур можно убедиться в том, что между статическими снимками состояния мира столкновений не было пропущено никаких пересечений.

Однако в случае линейной интерполяции изогнутых траекторий или при вращении объектов collidable результаты обычно получаются неточными, поэтому, в зависимости от нужд игры, вам могут понадобиться более детализированные методики.

Непрерывное обнаружение столкновений. Еще один способ борьбы с туннельным эффектом состоит в использовании методики под названием «*непрерывное обнаружение столкновений*» (continuous collision detection, CCD). Цель CCD — найти самый ранний *момент столкновения* (time of impact, TOI) двух движущихся объектов на заданном отрезке времени.

Алгоритмы CCD по своей природе, как правило, итеративные. Положение и наклон каждого объекта collidable отслеживаются как в текущий, так и в предыдущий момент времени. С помощью этой информации можно по отдельности линейно интерполировать перемещение и вращение, что позволяет получить приближенное преобразование объекта collidable в любой момент между текущим и предыдущим. Затем алгоритм ищет самый ранний TOI вдоль траектории движения. Для этого существует целый ряд широко распространенных поисковых алгоритмов, включая метод *консервативного продвижения* Брайана Миртича, который отбрасывает луч на сумму Минковского, или поиск минимального TOI для пары отдельных деталей формы. В своей исследовательской работе http://gamedevs.org/uploads/continuous-collision-detection-and-physics.pdf Эрвин Кауманс из Sony Interactive Entertainment описывает некоторые из этих алгоритмов наряду с собственным инновационным вариантом консервативного продвижения.

13.3.6. Оптимизация производительности

Обнаружение столкновений сильно нагружает центральный процессор. Этому есть две причины.

1. Вычисления, необходимые для определения того, пересекаются две формы или нет, сами по себе являются нетривиальными.
2. Большинство виртуальных миров содержат много объектов, и с ростом их количества стремительно увеличивается число проверок на пересечение.

Чтобы обнаружить пересечение n объектов, нужно проверить каждую возможную пару, в результате чего получится алгоритм класса $O(n^2)$ — это самый очевидный способ. Но на практике используются куда более эффективные алгоритмы. Движки столкновений обычно применяют какую-то разновидность пространственного хеширования (bit.ly/1fLtX1D), пространственного разбиения или иерархии ограничивающих объемных областей, чтобы уменьшить количество необходимых проверок на пересечение.

Временная когерентность

Один из распространенных методов оптимизации состоит в использовании *временной когерентности*, которую еще называют *покадровой*. Когда объекты collidable движутся не очень быстро, их положение и наклон в смежные моменты времени

обычно мало чем различаются. Часто определенную информацию можно кэшировать на протяжении нескольких кадров, избегая ее повторного вычисления. Например, в Havok агенты столкновения (`hkpCollisionAgent`), как правило, не меняются между кадрами, что позволяет повторно применять результаты, полученные в предыдущие моменты времени. Это происходит до тех пор, пока результаты не потеряют свою актуальность в ходе движения объекта collidable.

Пространственное разбиение

Основная идея пространственного разбиения состоит в том, чтобы существенно уменьшить количество объектов collidable, которые нужно проверять на пересечение, для чего пространство делится на участки меньшего размера. Если можно точно определить, что два объекта не занимают один и тот же участок, то для них не нужно выполнять подробную проверку на пересечение.

Для разбиения пространства с целью оптимизации обнаружения столкновений можно использовать различные иерархические схемы, такие как октодеревья, BSP-деревья (binary space partitioning — «двоичное разбиение пространства»), *k*-мерные деревья или сферические деревья. Эти структуры делят пространство разными способами, но у всех них есть общее — иерархический подход. В корне дерева происходит разбиение на большие участки, которые с продвижением по иерархии становятся все меньше, пока не будет достигнута нужная степень детализации. Затем мы можем пройтись по дереву, найти группу потенциально сталкивающихся объектов и проверить, пересекаются ли они на самом деле. Благодаря древовидному разбиению мы знаем, что объекты из разных параллельных ветвей не могут столкнуться друг с другом.

Широкофазное, среднефазное и узкофазное обнаружение

Havok использует трехуровневый подход, чтобы уменьшить количество объектов, которые нужно проверять на столкновение в каждый дискретный момент времени.

- Сначала выполняются грубые проверки на основе AABB, чтобы определить, какие объекты могут пересекаться. Это называется *широкофазным* обнаружением столкновений.
- Дальше проверяются грубые ограничивающие области на основе составных фигур. Это *среднефазное* обнаружение столкновений. Например, если фигура состоит из трех сфер, ограничивающая область может иметь вид четвертой сферы, которая окружает остальные три. Элементы составной фигуры тоже могут быть составными, поэтому составной объект collidable в целом имеет иерархию ограничивающей области. Мы проходимся по этой иерархии в поисках дочерних фигур, которые могут пересекаться.
- В конце на пересечение проверяются отдельные примитивы объекта. Это *узкофазное* обнаружение столкновений.

Алгоритм sweep and prune. Во всех основных движках столкновений/физики, таких как Havok, ODE и PhysX, широкофазное обнаружение основано на алгоритме *sweep and prune* («найти и обрезать»; ru.wikipedia.org/wiki/Sweep_and_prune). Его основная суть в том, чтобы отсортировать начальные и конечные координаты AABB, принадлежащие объектам collidable, вдоль трех главных осей, а затем поискать пересекающиеся AABB, перебирая отсортированный список. Алгоритмы этого семейства могут использовать покадровую когерентность (см. подраздел 13.3.6), чтобы уменьшить количество операций сортировки с $O(n \log n)$ до $O(n)$ (это обеспечивает предсказуемое время выполнения). Покадровая когерентность может также помочь с обновлением AABB при вращении объектов.

13.3.7. Запросы о столкновениях

Помимо прочего, система обнаружения столкновений должна отвечать на гипотетические вопросы о пересечении объемных областей в игровом мире, например такие.

- Если выпустить пулю из оружия игрока в заданном направлении, во что она сначала попадет и попадет ли вообще?
- Может ли автомобиль переместиться из точки A в точку B, не врезавшись ни во что по пути?
- Найти все вражеские объекты в заданном радиусе от игрока.

Обычно такие операции называют *запросами о столкновениях*.

Самым распространенным видом запросов является *след столкновений (collision cast)*. Он определяет, с чем столкнулся бы гипотетический объект, если бы перемещался по миру столкновений вдоль луча или отрезка прямой. Следы отличаются от других операций обнаружения столкновений тем, что никак не влияют на другие объекты игрового мира. Именно поэтому мы говорим, что след дает ответ на *гипотетический* вопрос об объектах collidable в игре.

Рейкастинг

Самым простым видом следов столкновений является *отбрасывание луча* (ray cast, или *рейкастинг*), хотя это не совсем правильное название. На самом деле мы отбрасываем *направленный отрезок прямой*, иными словами, лучи всегда начинаются в точке ($\mathbf{p}_0$) и заканчиваются в точке ($\mathbf{p}_1$). Мы проверяем, какие объекты пересечет этот отрезок в игровом мире. При обнаружении каких-либо пересечений возвращается точка или точки контакта.

В системах рейкастинга отрезок прямой обычно описывается в виде начальной точки $\mathbf{p}_0$ и дельта-вектора $\mathbf{d}$, который при сложении с $\mathbf{p}_0$ дает конечную точку $\mathbf{p}_1$. Любые точки на этом отрезке можно найти с помощью следующего *параметрического уравнения*, в котором параметр t находится в диапазоне от нуля до единицы:

$$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d},\ t \in [0, 1].$$

Очевидно, что $\mathbf{p}_0 = \mathbf{p}(0)$, а $\mathbf{p}_1 = \mathbf{p}(1)$. Кроме того, любую точку контакта вдоль отрезка можно описать уникальным образом, если указать значение параметра *t*, соответствующее контакту. Большинство API для рейкастинга возвращают точки контакта в виде значений *t* или позволяют получить значения вызовом еще одной функции.

Большинство систем обнаружения столкновений способны вернуть *самый ранний контакт*, то есть точку контакта, которая находится ближе всего к $\mathbf{p}_0$ и соответствует наименьшему значению *t*. Некоторые системы также могут вернуть полный список всех объектов collidable, пересеченных лучом или отрезком прямой. Информация, возвращаемая для каждого контакта, обычно содержит значение *t*, уникальный идентификатор затронутого объекта и, возможно, другие сведения вроде нормали к поверхности в точке контакта и различные свойства задетой формы или поверхности. Структура данных для точки контакта может выглядеть так:

```
struct RayCastContact
{
    F32    m_t;            // значение t для этого контакта
    U32    m_collidableId; // с каким объектом мы столкнулись?
    Vector m_normal;       // нормаль к поверхности в точке контакта
    // другая информация...
};
```

Применение рейкастинга. Рейкастинг активно используется в играх. Например, мы можем спросить у системы столкновений, находится ли персонаж B в прямой области видимости персонажа A. Чтобы определить это, просто направляем прямой луч из глаз персонажа A на грудь персонажа B. Если персонаж B задет этим лучом, мы знаем, что A может видеть B. Но если луч сталкивается с каким-то другим объектом *до* того, как достигнет персонажа B, значит, поле зрения блокируется этим объектом. Рейкастинг применяется в системах оружия (например, для определения мест попадания пуль), механике игрока (например, чтобы понять, есть ли под ногами персонажа твердая поверхность), системах ИИ (для проверки поля зрения, нацеливания, запросов о движениях и т. д.), системах управления транспортными средствами (например, чтобы определить местоположение покрышек автомобиля и привязать их к земле) и т. д.

Отбрасывание форм

Еще один распространенный запрос к системе столкновений позволяет узнать, как далеко могла бы пройти воображаемая выпуклая фигура вдоль направленного отрезка прямой, прежде чем столкнуться с чем-то твердым. Это называют *отбрасыванием форм* или *сфер*, когда в качестве отбрасываемой формы используется сфера (в Havok это называется *линейным отбрасыванием*). Как и в случае с лучами, чтобы описать отбрасывание формы, обычно указывают начальную точку $\mathbf{p}_0$, расстояние $\mathbf{d}$, которое нужно преодолеть, и, конечно же, тип, размеры и наклон фигуры, которую мы хотим отбросить.

При отбрасывании выпуклых форм возможны два сценария.

1. Отбрасываемая форма, находясь в начальной точке, уже пересекает по меньшей мере один объект collidable или прикасается к нему, что не дает ей двигаться дальше.
2. Отбрасываемая форма ни с чем не пересекается в своей начальной точке, поэтому может преодолеть ненулевое расстояние с заданной траекторией.

В первом случае система столкновений обычно сообщает об одном или нескольких контактах между отбрасываемой формой и всеми объектами, которые она изначально пересекает. Эти контакты могут находиться как *внутри* отбрасываемой формы, так и на ее *поверхности* (рис. 13.19).

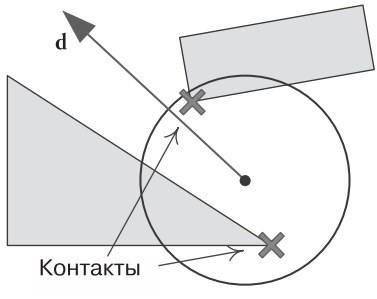


Рис. 13.19. Если отбрасываемая сфера изначально с чем-то пересекается, она не сможет двигаться дальше; к тому же внутри нее может находиться множество точек контакта

Во втором случае форма может пройти ненулевое расстояние вдоль отрезка прямой. Обычно она сталкивается с не более чем одним объектом. Но если выбрать подходящую траекторию, она может столкнуться сразу с несколькими объектами. И конечно, если столкновение происходит с невыпуклым полигональным супом, мы можем получить одновременно несколько точек контакта. Можно с уверенностью сказать, что множественные столкновения *возможны* независимо от того, какую выпуклую форму мы отбрасываем. В этом случае все контакты будут исключительно на ее *поверхности* и не смогут оказаться внутри (поскольку мы уже знаем, что в начале своего движения она ни с чем не пересекалась). Этот сценарий проиллюстрирован на рис. 13.20.

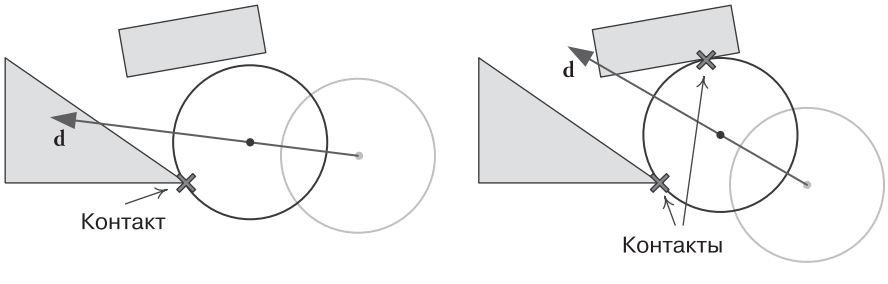


Рис. 13.20. Если в начальной точке отбрасываемая форма ни с чем не пересекается, она может пройти ненулевое расстояние вдоль отрезка прямой и все контакты (если таковые обнаружатся) будут возникать исключительно на ее поверхности

Как и при рейкастинге, некоторые API для отбрасывания форм сообщают только о *самом раннем* контакте, который случился с формой, но бывают и такие, которые позволяют форме двигаться дальше по гипотетической траектории и возвращают все произошедшие контакты (рис. 13.21).

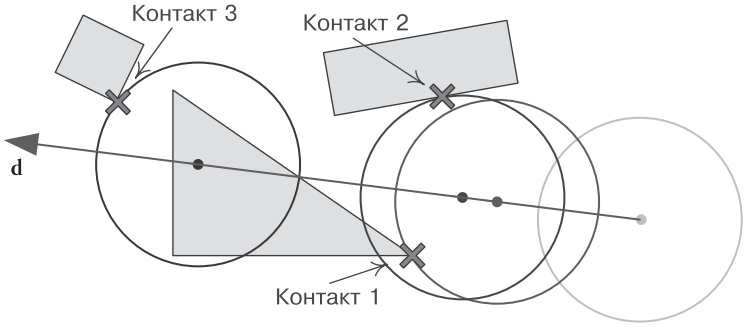


Рис. 13.21. API для отбрасывания форм возвращает все контакты, а не только самый ранний

Отбрасывание формы всегда возвращает чуть более сложную информацию о контакте, чем рейкастинг. Здесь нельзя ограничиться лишь одним или несколькими значениями *t*, поскольку они относятся лишь к центральной точке фигуры вдоль траектории. Это не позволяет узнать, где произошел контакт с объектом collidable: на поверхности или внутри фигуры. В результате большинство API для отбрасывания форм возвращают как значение *t*, так и саму точку контакта, а также другую полезную информацию, например, о том, с каким объектом мы столкнулись, нормаль к поверхности в точке контакта и т. д.

В отличие от API для рейкастинга любая система отбрасывания форм должна быть способна сообщать о множественных контактах. Мы не можем возвращать лишь один контакт с наименьшим значением *t*, так как форма может задеть сразу несколько разных объектов игрового мира или несколько точек одного невыпуклого объекта. В связи с этим системы столкновений обычно возвращают массив или список структур данных с точками контактов. Каждая структура может иметь следующий вид:

```
struct ShapeCastContact
{
    F32    m_t;            // значение t для этого контакта
    U32    m_collidableId; // какой объект collidable мы задели?
    Point  m_contactPoint; // местоположение самого контакта
    Vector m_normal;       // нормаль к поверхности в точке контакта
    // другая информация...
};
```

Полученный список точек контактов обычно имеет смысл сгруппировать по отдельным значениям *t*. Например, самый ранний контакт может описываться группой точек в списке с общим минимальным значением *t*. Важно понимать, что система столкновений может сортировать точки контактов по *t*, а может и не делать этого. Во втором случае почти всегда стоит выполнить такую сортировку вручную. Таким образом, если взглянуть на элемент в списке, рядом с ним всегда можно будет увидеть самые ранние точки столкновения по траектории движения формы.

Отбрасывание форм. С помощью отбрасывания сферы можно определить, блокируется ли виртуальная камера объектами игрового мира. Отбрасывание сфер

и капсул часто задействуется и при реализации движений персонажа. Например, чтобы передвинуть персонажа вперед по неровной поверхности, мы можем отбросить сферу или капсулу из его ног в направлении движения. Затем можем отрегулировать положение персонажа по вертикали с помощью второго отбрасывания, чтобы он всегда касался земли. Если сфера задевает очень короткое вертикальное препятствие вроде бордюра, она может немного приподняться. Если вертикальное препятствие слишком высокое, как стена, отбрасываемая сфера может проскользить вдоль него горизонтально. В следующем кадре персонаж переместится туда, где остановилась сфера.

Фантомы

Иногда игра должна определить, какие объекты collidable находятся внутри какой-то определенной объемной области игрового мира. Например, нам может потребоваться список всех врагов в заданном радиусе от персонажа игрока. Для этих целей в Havok предусмотрен специальный тип объектов collidable, известный как *фантом*.

Фантом по своему действию во многом похож на отбрасываемую форму с нулевым вектором расстояния **d**. В любой момент у фантома можно запросить список его контактов с другими объектами игрового мира. Он возвращает эти данные практически в таком же формате, как и отбрасываемая форма с нулевым расстоянием.

Но, в отличие от отбрасываемой формы, фантом является статическим. Это означает, что он может в полной мере применять оптимизацию на основе временной когерентности, которая используется движком столкновений при обнаружении пересечения настоящих объектов. На самом деле единственное отличие фантома от обычных объектов collidable состоит в том, что они его не видят (он также не принимает участия в симуляции динамики). Это позволяет ему отвечать на гипотетические вопросы о том, с чем бы он столкнулся, если бы он был «настоящим» объектом collidable, но при этом он никак не влияет на другие объекты, включая другие фантомы, в мире столкновений.

Другие типы запросов

Некоторые движки столкновений, помимо следов, поддерживают и другие виды запросов. Например, в Havok можно запросить *ближайшие точки* на самых близких объектах collidable в мире столкновений.

13.3.8. Фильтрация столкновений

Игровые разработчики нередко хотят иметь возможность разрешать и запрещать столкновения между определенными типами объектов. Например, большинству объектов позволено проходить сквозь поверхность воды: мы можем применить симуляцию плавучести, чтобы поддерживать их на плаву, или же позволить им уйти на дно, но в любом случае не хотим, чтобы поверхность воды выглядела твердой.

Большинство движков столкновений позволяют принимать или отклонять контакт между объектами collidable на основе определенных игровых критериев. Это называется *фильтрацией столкновений*.

Маски и объединение столкновений в слои

Один из распространенных подходов к фильтрации — классифицировать объекты в игровом мире по категориям, а затем использовать справочную таблицу, чтобы определить, разрешено ли конкретным категориям конфликтовать друг с другом. Например, в Havok объект может быть членом одного (и только одного) уровня коллизий. Фильтр столкновений по умолчанию в Havok, представленный экземпляром класса `hkpGroupFilter`, поддерживает 32-битную маску для каждого уровня, каждый бит которой сообщает системе, может ли этот конкретный уровень сталкиваться с одним из других уровней.

Обратные вызовы в ответ на столкновения

Еще один метод фильтрации состоит в том, чтобы при обнаружении столкновения срабатывала *функция обратного вызова*. Эта функция может проанализировать обстоятельства столкновения и принять решение о том, следует его принять или отклонить, руководствуясь соответствующими критериями. Havok поддерживает и этот подход. Когда в мир столкновений вносятся первые точки контактов, вызывается функция `contactPointAdded()`. Если позже контакт признается корректным (то есть если его время TOI самое раннее), инициируется обратный вызов `contactPointConfirmed()`. В обеих этих функциях приложение может принять решение об отклонении контакта.

Материалы столкновений в разных играх

Игровым разработчикам часто приходится разбивать объекты игры на категории: отчасти, чтобы контролировать то, как они сталкиваются (как в случае с фильтрацией столкновений), и отчасти, чтобы управлять второстепенными последствиями, выбирая звуки или эффекты частиц, которые генерируются при попадании одного объекта в другой. Например, мы могли бы различать дерево, камень, металл, грязь, воду и человеческую плоть.

Для этого во многих играх предусмотрен механизм разделения сталкивающихся объектов на категории, во многом похожий на *систему материалов*, которая задействуется в движке отрисовки. На самом деле некоторые группы игровых разработчиков используют для такой категоризации термин «*материал столкновений*». Основной смысл этого в том, чтобы назначить поверхности каждого объекта collidable набор свойств, которые определяют ее поведение с точки зрения физики и столкновения. В число этих свойств могут входить звуковые эффекты и эффекты частиц, физические параметры вроде коэффициентов упругого восстановления или трения, сведения о фильтрации столкновений и любые другие данные, которые могут потребоваться игре.

В случае с простыми выпуклыми примитивами свойства столкновения обычно назначаются форме целиком. Если речь идет о формах на основе полигонального супа, свойства могут относиться к отдельным треугольникам. Учитывая вероятность второго варианта, связь между примитивом столкновения и его материалом обычно стараются сделать как можно более компактной. Как правило, для этого используют 8-, 16- или 32-битное целое число или указатель на данные материала. Это число служит индексом глобального массива со структурами данных, которые содержат подробные сведения о столкновениях.

13.4. Динамика твердого тела

В игровом движке особенно важную роль играет *кинематика* объектов — то, как они перемещаются со временем. Многие игровые движки включают в себя *систему физики*, предназначенную для симуляции движений объектов в виртуальном игровом мире так, чтобы это выглядело реалистично. Строго говоря, игровые движки обычно озабочены лишь одной областью физики — *динамикой*, которая изучает влияние *сил* на движение объектов. До недавнего времени системы игровой физики были почти полностью сфокусированы на конкретном подразделе этой области, известной как *классическая динамика твердого тела*. Как можно догадаться по этому названию, в ходе симуляции физики в игре допускаются два важных упрощения.

- *Классическая (ньютоновская) механика*. Подразумевается, что объекты в физической модели подчиняются законам Ньютона. Их размеры достаточно велики для того, чтобы пренебречь квантовыми эффектами, а скорости не настолько высоки, чтобы учитывать релятивистские эффекты.
- *Твердые тела*. Все объекты в симуляции являются идеально твердыми и не могут деформироваться. Иными словами, их форма никогда не меняется. Эта идея хорошо сочетается с предположениями, которые делает система обнаружения столкновений. Более того, предположение о твердости тел существенно упрощает математические вычисления, необходимые для симуляции динамики сплошных объектов.

Движки игровой физики умеют также налагать различные *ограничения* на движения твердых тел в виртуальном мире. Самым распространенным ограничением является непроницаемость — объекты не могут проходить друг сквозь друга. Следовательно, когда система физики обнаруживает пересекающиеся тела, она пытается смоделировать реалистичную *реакцию на столкновение*[1]. Это одна из основных причин, почему физический движок и система обнаружения столкновений тесно связаны между собой.

Большинство физических систем также позволяют разработчикам игр устанавливать дополнительные ограничения, чтобы получить реалистичное взаимодействие

[1] Если используется система непрерывного обнаружения столкновений, реакцией может быть предотвращение пересечения как такового.

между физически симулируемыми твердыми телами. Это может включать шарниры, призматические соединения (ползунки), шаровые опоры, колеса, тряпичные куклы (для эмуляции потерявших сознание или мертвых персонажей) и т. д.

Система физики обычно использует структуру данных из мира столкновений и, в сущности, инициирует выполнение алгоритма обнаружения столкновений в ходе процесса обновления в каждый дискретный момент времени. Обычно движок столкновений поддерживает отношение «один к одному» между твердыми телами в симуляции динамики и объектами collidable. Например, в Havok объект `hkpRigidBody` содержит ссылку на один конкретный экземпляр `hkpCollidable` (хотя можно создать такой объект collidable, у которого нет твердого тела). В PhysX эти две концепции интегрированы немного теснее: `NxActor` служит как объектом collidable, так и твердым телом, участвующим в симуляции динамики. Эти твердые тела и связанные с ними объекты обычно хранятся в структуре-синглтоне под названием *«мир столкновений/физики»* (иногда, просто *«мир физики»*).

С точки зрения игрового процесса между твердыми телами в физическом движке и логическими объектами, из которых состоит виртуальный мир, обычно существуют определенные различия. Положение и наклон игровых объектов могут быть продиктованы симуляцией физики. Для этого в каждом кадре мы обращаемся к физическому движку за преобразованиями всех твердых тел, а затем каким-то образом применяем их к преобразованиям соответствующих игровых объектов. В то же время движение игрового объекта, подчиняющегося какой-то другой части движка (например, системе анимации или системе управления персонажем), может влиять на положение и наклон твердого тела в мире физики. Как упоминалось в подразделе 13.3.1, один логический игровой объект может быть представлен в мире физики одним или несколькими твердыми телами. Такие простые предметы, как камень, оружие или бочка, могут быть связаны лишь с одним твердым телом. Но выразительный персонаж или сложный механизм могут состоять из множества взаимосвязанных твердых частей.

Дальнейший материал этой главы будет посвящен исследованию принципов работы физических движков. Мы познакомимся с теорией, лежащей в основе симуляции динамики твердого тела. Затем рассмотрим некоторые из наиболее распространенных возможностей систем физики и посмотрим, каким образом физический движок можно интегрировать в игру.

13.4.1. Некоторые основы

На тему классической динамики твердого тела создано великое множество замечательных книг, статей и презентаций. Прочный фундамент при изучении теории аналитической механики поможет заложить [17]. Еще более актуальными для нашего обсуждения являются книги [13], [29] и [39], где речь идет именно о физических симуляциях, которые применяются в играх. Издания вроде [2], [11] и [32] содержат главы о динамике твердых тел в играх. В журнале *Game Developer Magazine* опубликован цикл полезных статей Криса Хекера об игровой физике. Крис выложил их и множество других интересных ресурсов на странице

chrishecker.com/Rigid_Body_Dynamics. Рассел Смит, основной автор ODE, создал информативную презентацию о симуляции динамики в играх (www.ode.org/slides/parc/dynamics.pdf).

В этом разделе я кратко представлю фундаментальные теоретические концепции, лежащие в основе большинства игровых физических движков. Это очень сжатое изложение, поэтому некоторые подробности я буду вынужден опустить. Настоятельно рекомендую вам, прочитав эту главу, ознакомиться хотя бы с парочкой дополнительных ресурсов, перечисленных ранее.

Единицы измерения

Большинство симуляций динамики твердого тела выполняются в системе единиц измерения МКС, в которой расстояние измеряется в метрах (м), масса в килограммах (кг), а время — в секундах (с). Отсюда и название — МКС.

При желании систему физики можно сконфигурировать для работы с другими единицами измерения, но если вы сделаете это, вам нужно будет убедиться в том, что это не вызовет несогласованности в вашей симуляции. Например, такие константы, как ускорение свободного падения (**g**), которые в системе МКС измеряются в метрах в секунду в квадрате (м/с^2), придется выразить в выбранной вами системе измерения. Большинство игровых студий используют МКС, чтобы не усложнять себе жизнь.

Разделение линейной и угловой динамики

Твердое тело называют *свободным*, если оно может беспрепятственно перемещаться вдоль всех трех декартовых осей и вращаться вокруг них. Говорят, что у такого тела есть шесть *степеней свободы*.

Возможно, вам покажется удивительным то, что движение свободного твердого тела можно разделить на два независимых компонента.

- *Линейная динамика*. Это описание движения тела без учета каких-либо вращательных эффектов (с помощью одной лишь линейной динамики можно описать движение идеализированной *материальной точки*, то есть массы бесконечно малого размера, которая не может вращаться).
- *Угловая динамика*. Это описание вращательного движения тела.

Возможность разделять линейный и угловой компоненты движения твердого тела чрезвычайно полезна при анализе или симуляции его поведения. Это означает, что мы можем просчитать линейное движение тела без учета вращения, как будто это идеализированная материальная точка, и наложить на результат угловое движение, чтобы получить полное описание перемещений тела.

Центр массы

С точки зрения *линейной* динамики свободное твердое тело ведет себя так, будто вся его масса сконцентрирована в одной точке — *центре массы* (ЦМ (center of

mass, CM)). Это, в сущности, точка балансировки тела для всех возможных направлений. Иными словами, масса твердого тела распределяется равномерно во всех направлениях вокруг его центра массы.

Центр массы тела однородной плотности находится в его *барицентре*. То есть если разделить тело на N очень маленьких частей, сложить их позиции в виде суммы векторов и затем разделить на N, можно получить примерное местоположение центра массы с довольно хорошей точностью. Если плотность тела неоднородная, позицию каждой части необходимо *взвесить* в соответствии с его *массой*. Это означает, что центр массы на самом деле представляет собой *среднее взвешенное* позиций этих частей. Таким образом мы имеем

$$\mathbf{r}_{\text{CM}} = \frac{\sum_{\forall i} m_i \mathbf{r}_i}{\sum_{\forall i} m_i} = \frac{\sum_{\forall i} m_i \mathbf{r}_i}{m},$$

где m — общая масса тела; $\mathbf{r}$ — *вектор радиуса* или *вектор положения*, проведенный из начала координат глобального пространства к заданной точке (по мере приближения размеров и масс маленьких частей к нулю их суммы становятся интегралами в пределе).

Центр массы всегда находится внутри выпуклого тела, а если тело вогнутое, он может располагаться за его пределами (например, где бы находился центр массы буквы «C»?).

13.4.2. Линейная динамика

С точки зрения линейной динамики позицию твердого тела можно полностью описать с помощью вектора положения $\mathbf{r}_{\text{CM}}$, направленного из начала координат глобального пространства к центру массы тела (рис. 13.22). Мы используем систему МКС, поэтому позиция измеряется в метрах. В дальнейшем будем опускать нижний индекс CM, поскольку и так понятно, что здесь описывается движение центра массы тела.

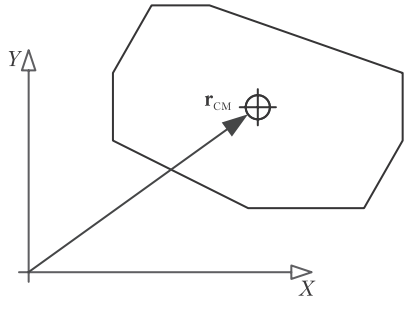


Рис. 13.22. С точки зрения линейной динамики для полного описания позиции твердого тела достаточно одной лишь позиции его центра массы

Линейный вектор скорости и линейное ускорение

Линейный вектор скорости твердого тела определяет скорость и направление, в котором движется ЦМ тела. Это векторная величина, которая обычно измеряется в метрах в секунду. Вектор скорости является производной первого порядка от положения, поэтому мы можем записать:

$$\mathbf{v}(t) = \frac{d\mathbf{r}(t)}{dt} = \dot{\mathbf{r}}(t),$$

где $\dot{\mathbf{r}}$ — производная по времени. Дифференцирование вектора эквивалентно дифференцированию каждого его компонента по отдельности, поэтому:

$$v_x(t) = \frac{dr_x(t)}{dt} = \dot{r}_x(t).$$

То же самое можно повторить для компонентов y и z.

Линейное ускорение — это производная первого порядка от линейного вектора скорости, взятая по времени, или производная второго порядка от положения ЦМ тела по времени. Это тоже векторная величина, и ее обычно обозначают **a**. Поэтому можем записать:

$$\mathbf{a}(t) = \frac{d\mathbf{v}(t)}{dt} = \dot{\mathbf{v}}(t) = \frac{d^2\mathbf{r}(t)}{dt^2} = \ddot{\mathbf{r}}(t).$$

Сила и импульс

Сила — это то, что заставляет объект с массой ускоряться или замедляться. Сила имеет как величину, так и направление в пространстве, поэтому она всегда представлена вектором. Часто ее обозначают **F**. Когда к телу применяется N сил, их общее влияние на линейное движение тела вычисляется простым сложением их векторов:

$$\mathbf{F}_{\text{net}} = \sum_{i=1}^{N} \mathbf{F}_i.$$

Знаменитый второй закон Ньютона гласит, что сила пропорциональна ускорению и массе:

$$\mathbf{F}(t) = m\mathbf{a}(t) = m\ddot{\mathbf{r}}(t). \tag{13.2}$$

Закон Ньютона подразумевает, что сила измеряется в килограмм-метрах на секунду в квадрате (кг-м/с^2). Эта единица измерения называется *ньютоном*.

Если умножить линейный вектор скорости тела на его массу, получится величина, известная как *линейный импульс*, обычно ее обозначают **p**:

$$\mathbf{p}(t) = m\mathbf{v}(t).$$

Уравнение (13.2) справедливо, если масса остается постоянной. Но если она меняется (как в случае с ракетой, которая постепенно сжигает топливо, превращая

его в энергию), это уравнение не совсем верно. На самом деле оно должно выглядеть так:

$$\mathbf{F}(t) = \frac{d\mathbf{p}(t)}{dt} = \frac{d\left(m(t)\mathbf{v}(t)\right)}{dt}.$$

Конечно, если масса постоянная, это уравнение сводится к более знакомому нам $\mathbf{F} = m\mathbf{a}$ и может быть вынесено за знак производной. Линейный импульс не имеет большого значения. Однако само понятие импульса пригодится при обсуждении угловой динамики.

13.4.3. Решение уравнений движения

Ключевая проблема динамики твердого тела состоит в том, чтобы определить движение тела с учетом известных нам сил, которые на него влияют. В линейной динамике это означает поиск $\mathbf{v}(t)$ и $\mathbf{r}(t)$ при условии, что мы знаем результирующую силу $\mathbf{F}_{\text{net}}(t)$ и, возможно, другие параметры, такие как положение и вектор скорости в один из прошедших моментов времени. Как мы увидим в дальнейшем, это сводится к решению двух обыкновенных дифференциальных уравнений: одного для поиска $\mathbf{v}(t)$ при наличии $\mathbf{a}(t)$, другого — для поиска $\mathbf{r}(t)$ при наличии $\mathbf{v}(t)$.

Сила как функция

Сила может быть постоянной, но ее также можно представить в виде функции от времени, как показано ранее. Сила также может быть функцией положения тела, его вектора скорости или любого количества других величин. Поэтому уравнение силы в целом следует записать так:

$$\mathbf{F}(t, \mathbf{r}(t), \mathbf{v}(t)...) = m\mathbf{a}(t). \tag{13.3}$$

Это можно выразить в виде вектора положения и его первой и второй производных:

$$\mathbf{F}\left(t, \mathbf{r}(t), \dot{\mathbf{r}}(t)\ldots\right) = m\ddot{\mathbf{r}}(t).$$

Например, сила, прилагаемая пружиной, пропорциональна тому, насколько она была сжата относительно положения покоя. В одномерном пространстве, если положение покоя пружины равно $x = 0$, можно записать:

$$F(t, x(t)) = -kx(t),$$

где k — *постоянная упругости пружины*.

В качестве еще одного примера можно привести силу затухания механического амортизатора, пропорциональную вектору скорости его положения. В одномерном представлении можно записать:

$$F(t, v(t)) = -bv(t),$$

где b — *коэффициент вязкого затухания*.

Обыкновенное дифференциальное уравнение

В общем виде *обыкновенное дифференциальное уравнение* (ОДУ) представляет собой функцию от одной независимой переменной и различных производных этой функции. Если независимая переменная является временем, а функция выглядит как $x(t)$, ОДУ можно записать как отношение вида:

$$\frac{d^n x}{dt^n} = f\left(t, x(t), \frac{dx(t)}{dt}, \frac{d^2x(t)}{dt^2} \ldots \frac{d^{n-1}x(t)}{dt^{n-1}}\right).$$

Если перефразировать, производная n-го порядка от $x(t)$ выражается функцией f, которая принимает в качестве аргументов время t, положение $x(t)$ и любое количество производных от $x(t)$, *порядок* которых *меньше* n.

Как мы видели в уравнении (13.3), сила в целом является функцией от времени, положения и вектора скорости:

$$\ddot{\mathbf{r}}(t) = \frac{1}{m}\mathbf{F}\left(t, \mathbf{r}(t), \dot{\mathbf{r}}(t)\right).$$

Это явно подходит под определение ОДУ. Мы хотим решить это уравнение, чтобы найти $\mathbf{v}(t)$ и $\mathbf{r}(t)$.

Аналитические решения

В редких случаях дифференциальные уравнения движения можно решить *аналитическим* способом. Это означает, что мы можем найти простое замкнутое выражение, которое описывает положение тела для *всех возможных значений* времени t. Распространенным примером является вертикальное движение снаряда с постоянным ускорением под воздействием силы тяжести $\mathbf{a}(t) = [0, g, 0]$, где $g = -9{,}8$ м/с^2. В данном случае ОДУ движения сводится к:

$$\ddot{y}(t) = g.$$

Проинтегрировав выражение один раз, получим:

$$\dot{y}(t) = gt + v_0,$$

где v_0 — скорость по вертикали в момент времени $t = 0$. Если проинтегрировать его второй раз, получится знакомое решение:

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0,$$

где y_0 — начальное положение объекта на вертикальной оси.

Несмотря на все это, в игровой физике решения почти никогда нельзя найти аналитическим путем. Отчасти это вызвано тем, что замкнутые решения для некоторых дифференциальных уравнений попросту неизвестны. Более того, игра представляет собой интерактивную симуляцию, поэтому мы, как правило, должны предугадать, как силы будут изменяться со временем. Это делает невозможным

нахождение простых замкнутых выражений для положения и вектора скорости игрового объекта в виде функции от времени.

У этого эмпирического правила, конечно же, есть исключения. Например, с помощью замкнутых выражений часто вычисляют вектор скорости, который нужно придать снаряду, чтобы попасть по заранее известной цели.

13.4.4. Численное интегрирование

По названным ранее причинам в игровых физических движках используют метод под названием «*численное интегрирование*». С его помощью дифференциальные уравнения можно решить *пошагово*, получая решение для каждого следующего момента времени из предыдущего. Временные интервалы обычно имеют примерно одинаковую длину, которая обозначается Δt. Имея положение и вектор скорости тела в текущий момент времени t_1 и зная, что сила выражена функцией от времени, положения и/или вектора скорости, мы хотим найти положение и вектор скорости для следующего момента времени $t_2 = t_1 + \Delta t$. Иными словами, нам нужно получить $\mathbf{r}(t_2)$ и $\mathbf{v}(t_2)$ из $\mathbf{r}(t_1)$, $\mathbf{v}(t_1)$ и $\mathbf{F}(t, \mathbf{r}, \mathbf{v})$.

Явный метод Эйлера

Одним из простейших численных решений ОДУ является так называемый *явный метод Эйлера*. Этот интуитивный подход часто применяется при разработке новых игр. Предположим, нам уже известен текущий вектор скорости и, чтобы найти положение тела в следующем кадре, мы хотим решить такое ОДУ:

$$\mathbf{v}(t) = \dot{\mathbf{r}}(t). \tag{13.4}$$

Используя явный метод Эйлера, мы выражаем вектор скорости не в метрах в секунду, а в метрах за один кадр, для чего умножаем его на дельту времени и затем добавляем расстояние, пройденное за один кадр, к текущему положению, чтобы найти новую позицию в следующем кадре. В результате получается приближенное решение ОДУ (13.4):

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t. \tag{13.5}$$

С помощью аналогичного подхода можно найти вектор скорости тела в следующем кадре, зная результирующую силу, приложенную в этом кадре. Приближенное решение такого ОДУ:

$$\mathbf{a}(t) = \frac{\mathbf{F}_{\text{net}}(t)}{m} = \dot{\mathbf{v}}(t) -$$

явным методом Эйлера выглядит так:

$$\mathbf{v}(t_2) = \mathbf{v}(t_1) + \frac{\mathbf{F}_{\text{net}}(t)}{m}\Delta t. \tag{13.6}$$

Интерпретации явного метода Эйлера. В уравнении (13.5) на самом деле предполагается, что вектор скорости тела на протяжении всего отрезка времени остается постоянным. Поэтому мы можем использовать *текущий* вектор скорости тела, чтобы предсказать его положение в *следующем* кадре. Следовательно, изменение положения $\Delta\mathbf{r}$ в промежутке между t_1 и t_2 составляет $\Delta\mathbf{r} = \mathbf{v}(t_1)\Delta t$. Если представить себе график движения тела по времени, мы берем *уклон* функции в момент t_1, то есть просто $\mathbf{v}(t_1)$, и экстраполируем его линейно на следующий момент времени t_2. Как можно видеть на рис. 13.23, линейная экстраполяция может давать не совсем точную оценку положения в следующий момент $\mathbf{r}(t_2)$, но она работает довольно хорошо, если вектор скорости остается более или менее постоянным.

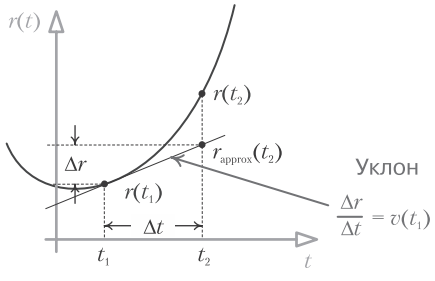


Рис. 13.23. В явном методе Эйлера уклон r(t) в момент t_1 используется для получения приближенного значения r(t_2) путем линейной экстраполяции r(t_1)

Этот рисунок подсказывает нам еще одну интерпретацию явного метода Эйлера — приближенное значение производной. Любая производная по определению является отношением двух бесконечно малых разностей — в нашем случае $d\mathbf{r}/dt$. Метод Эйлера сводит это к отношению двух *конечных разностей*. Иными словами, $d\mathbf{r}$ превращается в $\Delta\mathbf{r}$, а dt — в Δt. Из этого следует:

$$\frac{d\mathbf{r}}{dt} \approx \frac{\Delta\mathbf{r}}{\Delta t};\ \mathbf{v}(t_1) \approx \frac{\mathbf{r}(t_2) - \mathbf{r}(t_1)}{t_2 - t_1},$$

что опять можно упростить до уравнения (13.5). На самом деле это приближение корректно, только когда вектор скорости не меняется на отрезке времени. Его также можно использовать в пределе, так как Δt стремится к нулю (при достижении нуля оно становится *абсолютно* верным). Очевидно, этот же способ анализа можно применить и к уравнению (13.6).

Свойства численных методов

Я уже упоминал, что явный метод Эйлера не совсем точен. Давайте исследуем этот факт подробнее. У численного решения обыкновенного дифференциального уравнения есть три важных взаимосвязанных свойства.

- *Сходимость.* Временной интервал Δt стремится к нулю. Приближается ли при этом примерное решение к настоящему?

- *Степень*. Насколько серьезной является погрешность конкретного численного решения ОДУ? Такие погрешности обычно пропорциональны какой-то степени длины временного интервала Δt, поэтому их часто записывают с помощью О большого (например, $O(\Delta t^2)$). Говорят, что для конкретного численного метода степени n погрешность равна $O(\Delta t^{(n+1)})$.
- *Стабильность*. Свойственно ли численному решению стабилизироваться? Если численный метод добавляет в систему энергию, векторы скоростей объектов рано или поздно «взорвутся» и система станет *нестабильной*. Однако, если численный метод заберет энергию из системы, возникнет эффект амортизации и система будет *стабильной*.

Понятие степени нуждается в дополнительном разъяснении. Погрешность численного метода обычно измеряется сравнением его приближенного уравнения с разложением бесконечного ряда Тейлора для точного решения ОДУ. Затем мы упрощаем выражение, вычитая одно уравнение из другого. Оставшиеся элементы ряда Тейлора представляют собой ошибку, свойственную данному методу. Например, явное уравнение Эйлера выглядит так:

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t.$$

Разложение бесконечного ряда Тейлора для точного решения:

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + \frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \frac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + \ldots$$

где $\mathbf{r}^{(3)}$ — третья производная по времени. Таким образом, погрешность представлена всеми элементами, идущими за $\mathbf{v}\Delta t$, и имеет степень $O(\Delta t^2)$ (поскольку этот элемент по своим размерам существенно превышает элементы более высоких степеней):

$$\mathbf{E} = \frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \frac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + \ldots = O(\Delta t^2).$$

Чтобы явно выразить погрешность метода, ее часто приписывают в конце уравнения в виде О большого. Например, самой верной формой записи уравнения явного метода Эйлера является следующая:

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + O(\Delta t^2).$$

Говорят, что явный метод Эйлера является методом первого порядка, поскольку он точен вплоть до элемента ряда Тейлора с Δt в первой степени *включительно*. В целом, если погрешность метода равна $O(\Delta t^{(n+1)})$, это метод n-го порядка.

Альтернативы явному методу Эйлера

Явный метод Эйлера довольно активно применяется в простых интеграционных задачах в играх, давая наилучшие результаты, когда вектор скорости остается более или менее постоянным. Однако из-за существенных погрешностей и плохой стабильности он не используется в симуляции динамики общего назначения. Для решения ОДУ существует множество других способов, включая обратный

метод Эйлера (тоже первого порядка), метод среднего Эйлера (второго порядка) и семейство методов Рунге — Кутты (особенно популярен метод Рунге — Кутты четвертого порядка, сокращенно RK4). Мы не станем рассматривать их подробно, так как в Интернете и литературе о них можно найти множество информации. Хорошей отправной точкой при их изучении может послужить страница «Википедии» en.wikipedia.org/wiki/Numerical_methods_for_ordinary_differential_equations.

Метод Стёрмера — Верле

Наверное, среди численных методов решения ОДУ в современных интерактивных играх чаще всего применяют метод Стёрмера — Верле, поэтому я остановлюсь на нем отдельно. У него есть два варианта: обычный и *с учетом скорости*. Мы рассмотрим оба, но без теории и глубоких объяснений — они описаны во множестве документов и на веб-страницах, посвященных данной теме (для начала можете почитать ru.wikipedia.org/wiki/Метод_Стёрмера_—_Верле).

Выгодными особенностями обычного метода Стёрмера — Верле являются достижение более высокого порядка (низкая погрешность), относительная простота, низкая требовательность к ресурсам и то, что он позволяет за один шаг найти положение, выраженное непосредственно с точки зрения ускорения (обычно приходится выполнять два шага: от ускорения к вектору скорости, а затем к положению). Формула выводится за счет сложения двух разложенных рядов Тейлора, один из которых движется вперед во времени, а другой — назад:

$$\mathbf{r}(t_1+\Delta t)=\mathbf{r}(t_1)+\dot{\mathbf{r}}(t_1)\Delta t+\frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2+\frac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3+O(\Delta t^4);$$

$$\mathbf{r}(t_1-\Delta t)=\mathbf{r}(t_1)-\dot{\mathbf{r}}(t_1)\Delta t+\frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2-\frac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3+O(\Delta t^4).$$

Если сложить эти два выражения, соответствующие отрицательные и положительные элементы взаимно уничтожат друг друга. В результате мы получим положение в следующий момент времени в зависимости от ускорения и двух известных позиций в текущем и предыдущем моментах. Это обычный метод Стёрмера — Верле:

$$\mathbf{r}(t_1+\Delta t)=2\mathbf{r}(t_1)-\mathbf{r}(t_1-\Delta t)+\mathbf{a}(t_1)\Delta t^2+O(\Delta t^4).$$

Метод Стёрмера — Верле можно выразить с точки зрения результирующей силы:

$$\mathbf{r}(t_1+\Delta t)=2\mathbf{r}(t_1)-\mathbf{r}(t_1-\Delta t)+\frac{\mathbf{F}_{\text{net}}(t_1)}{m}\Delta t^2+O(\Delta t^4).$$

В этом выражении явно отсутствует вектор скорости. Однако его можно найти с помощью следующего несколько неточного приближения (это лишь один из вариантов):

$$\mathbf{v}(t_1+\Delta t)=\frac{\mathbf{r}(t_1+\Delta t)-\mathbf{r}(t_1)}{\Delta t}+O(\Delta t).$$

Метод Стёрмера — Верле с учетом скорости

Метод Стёрмера — Верле *с учетом скорости* является более распространенным. Он состоит из четырех шагов и для облегчения решения разделяет временной интервал на две части. Если нам известно $\mathbf{a}(t_1)=\frac{1}{m}\mathbf{F}(t_1,\mathbf{r}(t_1),\mathbf{v}(t_1))$, мы можем сделать следующее.

1. Вычислить $\mathbf{r}(t_1+\Delta t)=\mathbf{r}(t_1)+\mathbf{v}(t_1)\Delta t+\frac{1}{2}\mathbf{a}(t_1)\Delta t^2$.
2. Вычислить $\mathbf{v}\left(t_1+\frac{1}{2}\Delta t\right)=\mathbf{v}(t_1)+\frac{1}{2}\mathbf{a}(t_1)\Delta t$.
3. Определить $\mathbf{a}(t_1+\Delta t)=\mathbf{a}(t_2)=\frac{1}{m}\mathbf{F}(t_2,\mathbf{r}(t_2),\mathbf{v}(t_2))$.
4. Вычислить $\mathbf{v}(t_1+\Delta t)=\mathbf{v}\left(t_1+\frac{1}{2}\Delta t\right)+\frac{1}{2}\mathbf{a}(t_1+\Delta t)\Delta t$.

Обратите внимание на то, что на третьем шаге функция силы зависит от положения и вектора скорости в *следующий* момент времени, $\mathbf{r}(t_2)$ и $\mathbf{v}(t_2)$. Мы уже вычислили $\mathbf{r}(t_2)$ на первом шаге, поэтому у нас есть вся нужная информация при условии, что сила не зависит от вектора скорости. Если такая зависимость существует, мы должны найти приближенное значение вектора скорости в следующем кадре, возможно, с использованием явного метода Эйлера.

13.4.5. Угловая динамика в двухмерном пространстве

До сих пор мы занимались анализом линейных движений центра массы тела, которое ведет себя словно материальная точка. Как уже говорилось, свободное твердое тело может вращаться вокруг своего ЦМ. Это означает, что мы можем наложить его угловое движение на линейное движение его центра массы, чтобы получить полное описание перемещения тела. Раздел физики, изучающий вращательные движения тела в ответ на приложенные силы, называется *угловой динамикой*.

В двухмерном пространстве угловая динамика почти ничем не отличается от линейной. У каждой линейной величины есть угловой аналог, и с математической точки зрения все выглядит довольно изящно. Сначала исследуем двухмерную угловую динамику. Как вы сами увидите, при переходе в трехмерное пространство все станет немного запутаннее, но всему свое время!

Наклон и угловая скорость

В двухмерном пространстве любое твердое тело можно представить как тонкий лист бумаги (иногда в литературе такое тело называют *плоской пластинкой* (plane lamina)). Все линейные движения происходят в плоскости *xy*, а все вращения производятся вокруг оси *Z* (представьте себе деревянные куски головоломки, скользящие по поверхности для настольного хоккея).

Наклон твердого тела в 2D полностью описывается углом θ, измеряемым в радианах относительно какой-то заранее условленной оси вращения. Например, мы можем решить, что θ = 0, когда гоночный автомобиль сориентирован параллельно оси *X* и его передняя часть направлена в сторону ее убывания. Этот угол, конечно же, является функцией от времени, поэтому мы обозначим его $\theta(t)$.

Угловая скорость и угловое ускорение

Вектор угловой скорости позволяет определить, как быстро меняется угол вращения тела. В двухмерном пространстве это скалярная величина, поэтому правильнее называть ее просто угловой *скоростью*. Она обозначается скалярной функцией $\omega(t)$ и измеряется в радианах в секунду (рад/с). Угловая скорость представляет собой производную угла наклона $\theta(t)$ по времени:

Угловое представление:	Линейное представление:
$\omega(t) = \frac{d\theta(t)}{dt} = \dot{\theta}(t)$	$\mathbf{v}(t) = \frac{d\mathbf{r}(t)}{dt} = \dot{\mathbf{r}}(t).$

И как можно было бы ожидать, угловое ускорение определяет, как быстро меняется угловая скорость. Оно обозначается $\alpha(t)$ и измеряется в радианах на секунду в квадрате (рад/с^2):

Угловое представление:	Линейное представление:
$\alpha(t) = \frac{d\omega(t)}{dt} = \dot{\omega}(t) = \ddot{\theta}(t)$	$\mathbf{a}(t) = \frac{d\mathbf{v}(t)}{dt} = \dot{\mathbf{v}}(t) = \ddot{\mathbf{r}}(t).$

Момент инерции

Вращательным эквивалентом массы является величина, известная как *момент инерции*. По аналогии с тем, как масса описывает, насколько легко или сложно изменить линейный вектор скорости материальной точки, момент инерции измеряет усилия, необходимые для изменения угловой скорости твердого тела, вращающегося вокруг определенной оси. Если масса тела сконцентрирована рядом с точкой вращения, поворот вокруг этой оси будет относительно легким и, следовательно, момент инерции будет меньшим по сравнению с моментом инерции тела, чья масса распределена вдалеке от оси вращения.

Поскольку сейчас речь идет о двухмерной угловой динамике, осью вращения всегда будет *Z*, а момент инерции тела представлен простым скалярным значением. Последнее обычно обозначается символом *I*. Мы не станем углубляться в то, как именно вычисляется момент инерции. Полную цепочку уравнений можно найти в [17].

Вращательный момент

До сих пор мы исходили из того, что все силы применяются к центру массы твердого тела. Но в целом сила может быть приложена к любой его точке. Как мы уже

видели, если линия действия силы проходит через ЦМ, движение тела будет исключительно линейным. В противном случае, помимо обычного линейного движения, возникнет *вращательный момент* (рис. 13.24).

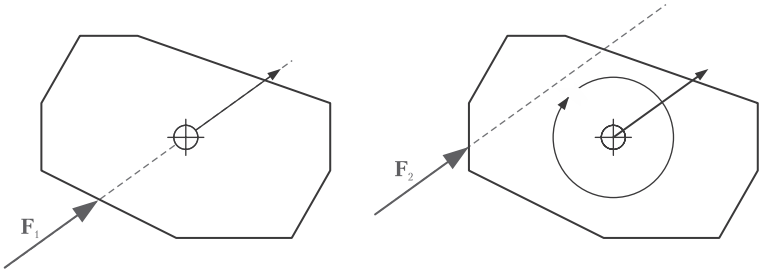


Рис. 13.24. *Слева*: сила, приложенная к ЦМ тела, вызывает чисто линейное движение. *Справа*: сила, приложенная не по центру, приведет к появлению вращательного момента, что создаст как линейное, так и вращательное движение

Вращательный момент можно вычислить с помощью векторного произведения. Сначала мы выражаем точку применения силы в виде вектора **r**, который направлен к ней из центра массы тела (иными словами, вектор **r** находится в *пространстве тела*, началом системы координат которого является ЦМ) (рис. 13.25). Вращательный момент **N**, вызванный силой **F**, приложенной к точке **r**, составляет:

$$\mathbf{N} = \mathbf{r} \times \mathbf{F}. \tag{13.7}$$

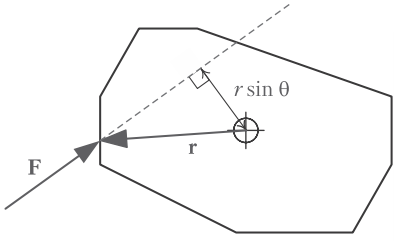


Рис. 13.25. Вращательный момент вычисляется с помощью векторного произведения точки применения силы в пространстве тела (то есть относительно его ЦМ) и вектора силы. Для наглядности векторы показаны в двухмерном пространстве: если бы это можно было изобразить, вектор вращательного момента был бы направлен в страницу

Из уравнения (13.7) следует, что чем дальше от центра массы прикладывается сила, тем выше вращательный момент. Это объясняет, почему с помощью рычага можно сдвинуть тяжелый предмет. Еще это объясняет, почему сила, приложенная непосредственно к центру массы, не вызывает вращательного момента и вращения как такового — в этом случае величина вектора **r** равна нулю.

Когда к твердому телу применяется две силы или более, векторы моментов вращения, созданные каждой из них, можно сложить по аналогии с тем, как слагаются

сами силы. Поэтому в целом нас интересует результирующий вращательный момент $\mathbf{N}_{\text{net}}$.

В двухмерном пространстве векторы **r** и **F** должны находиться на плоскости *xy*, поэтому момент **N** всегда будет направлен вдоль положительной или отрицательной оси *Z*. В связи с этим мы будем обозначать двухмерный вращательный момент в виде скалярной величины N_z, которая является компонентом *z* вектора **N**.

Вращательный момент имеет такое же отношение к угловому ускорению и инерции, как сила к линейному ускорению и массе:

Угловое представление:	Линейное представление:
$N_z(t) = I\alpha(t) = I\dot{\omega}(t) = I\ddot{\theta}(t)$	$\mathbf{F}(t) = m\mathbf{a}(t) = m\dot{\mathbf{v}}(t) = m\ddot{\mathbf{r}}(t).$

(13.8)

Решение угловых уравнений движения в двухмерном пространстве

Если взять двухмерный пример, для решения угловых уравнений движения можно использовать те же методы численного интегрирования, которые мы применяли в задаче с линейной динамикой. Далее показана пара ОДУ, которые мы хотим решить:

Угловое представление:	Линейное представление:
$N_{\text{net}}(t) = I\dot{\omega}(t);$	$\mathbf{F}_{\text{net}}(t) = m\dot{\mathbf{v}}(t);$
$\omega(t) = \dot{\theta}(t)$	$\mathbf{v}(t) = \dot{\mathbf{r}}(t).$

Их приближенные решения, полученные явным методом Эйлера, выглядят так:

Угловое представление:	Линейное представление:
$\omega(t_2) = \omega(t_1) + I^{-1}N_{\text{net}}(t_1)\Delta t;$	$\mathbf{v}(t_2) = \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t_1)\Delta t;$
$\theta(t_2) = \theta(t_1) + \omega(t_1)\Delta t$	$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t.$

Конечно, мы могли бы воспользоваться другими, более точными численными методами вроде метода Стёрмера — Верле с вектором скорости (для компактности я опускаю линейный случай, можете сравнить это с шагами, приведенными в подразделе 13.4.4).

1. Вычислить $\theta(t_1 + \Delta t) = \theta(t_1) + \omega(t_1)\Delta t + \frac{1}{2}\alpha(t_1)\Delta t^2.$
2. Вычислить $\omega\left(t_1 + \frac{1}{2}\Delta t\right) = \omega(t_1) + \frac{1}{2}\alpha(t_1)\Delta t.$
3. Вычислить $\alpha(t_1 + \Delta t) = \alpha(t_2) = I^{-1}N_{\text{net}}(t_2, \theta(t_2), \omega(t_2)).$
4. Вычислить $\omega(t_1 + \Delta t) = \omega\left(t_1 + \frac{1}{2}\Delta t\right) + \frac{1}{2}\alpha(t_1 + \Delta t)\Delta t.$

13.4.6. Угловая динамика в трехмерном пространстве

Угловая динамика в трехмерном пространстве является чуть более сложной темой по сравнению с двухмерным аналогом, хотя основные концепции, конечно же, очень похожи. В следующем разделе я кратко опишу принцип работы угловой динамики в 3D, уделяя первоочередное внимание вещам, которые обычно смущают новичков в этой области. Если хотите узнать больше, можете почитать цикл статей Гленна Фидлера, посвященных этой теме, они доступны по адресу gafferongames.com/game-physics/physics-in-3d/. Еще одним полезным ресурсом является статья Дэвида Бараффа из Института робототехники при Университете Карнеги — Мелона. Она называется *An Introduction to Physically Based Modeling*, и ее можно загрузить, перейдя по ссылке www-2.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf.

Тензор инерции

Масса твердого тела может совершенно по-разному распределяться по трем осям координат. Поэтому следует ожидать, что по каждой из этих осей у него будут разные моменты инерции. Например, длинный тонкий стержень должен легко вращаться вокруг своей длинной оси, поскольку в непосредственной близости от нее сконцентрирована вся масса. По этой же причине стержень будет довольно сложно вращать вокруг короткой оси, так как масса распределяется от нее намного дальше. Это действительно так, и именно поэтому, чтобы вращаться быстрее, фигуристы подтягивают руки и ноги поближе к телу.

В трехмерном пространстве вращательная масса твердого тела представлена матрицей размером 3 × 3, известной как *тензор инерции*. Ее обычно обозначают символом **I** (как и прежде, я не стану здесь описывать, как вычисляется тензор инерции, подробности ищите в [17]):

$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}.$$

Элементы, размещенные по диагонали матрицы, являются моментами инерции тела по трем его главным осям: I_{xx}, I_{yy} и I_{zz}. Остальные элементы называются *произведениями инерции*. Они равны нулю, если тело симметрично по всем трем главным осям, как, например, прямоугольный параллелепипед. Когда они не равны нулю, производимые ими движения обычно реалистичны с точки зрения физики, но выглядят несколько необычно, и большинство игроков может посчитать их неправильными. В связи с этим в игровых физических движках тензор инерции часто упрощают до трехэлементного вектора $\begin{bmatrix} I_{xx} & I_{yy} & I_{zz} \end{bmatrix}$.

Направление в трехмерном пространстве

Мы уже знаем, что наклон твердого тела в двухмерном пространстве можно описать одним углом θ, который измеряет вращение вокруг оси *Z* (если предположить, что движение происходит в плоскости *xy*). В трехмерном пространстве направление тела можно представить с помощью трех углов Эйлера $\left[\theta_x\ \theta_y\ \theta_z\right]$, каждый из которых соответствует повороту вокруг одной из трех осей прямоугольной системы координат. Но, как мы видели в главе 5, для углов Эйлера характерны проблемы складывания рамок и их может быть сложно использовать в математических вычислениях. Поэтому направление тела чаще описывают либо в виде матрицы **R** размером 3 × 3, либо с помощью единичного кватерниона q. В этой главе будем применять исключительно второй вариант.

Как вы помните, кватернион — это четырехэлементный вектор, компоненты *x*, *y* и *z* которого можно интерпретировать как единичный вектор **u**, размещенный вдоль оси вращения и масштабированный по синусу половинного угла, при этом компонент *w* вектора **u** является косинусом половинного угла:

$$\mathrm{q} = \left[q_x\ q_y\ q_z\ q_w\right] = \left[\mathbf{q}\ q_w\right] = \left[\mathbf{u}\sin\frac{\theta}{2}\ \cos\frac{\theta}{2}\right].$$

Ориентация тела, конечно же, является функцией от времени, поэтому нужно записать его как q(*t*).

Опять же мы должны выбрать произвольное направление, которое будет обозначать нулевое вращение. Например, мы можем сказать, что передняя часть каждого объекта по умолчанию будет размещаться вдоль положительной оси *Z* глобального пространства, а *Y* и *X* будут находиться сверху и слева соответственно. Любой неединичный кватернион будет служить для вращения объекта от его канонического направления в глобальном пространстве. Каноническое направление выбирается произвольным образом, однако оно, безусловно, должно быть одним и тем же для всех ресурсов игры.

Вектор угловой скорости и вращательный момент в трехмерном пространстве

Вектор угловой скорости в трехмерном пространстве обозначается ω(*t*), и его нельзя свести к скалярной величине. Он может быть представлен в виде единичного вектора **u**, который определяет оси вращения и масштабирован по двухмерному вектору угловой скорости $\omega_u = \dot{\theta}_u$ тела вокруг оси *U*. Следовательно:

$$\boldsymbol{\omega}(t) = \omega_u(t)\mathbf{u} = \dot{\theta}_u(t)\mathbf{u}.$$

В линейной динамике мы уже видели, что, если на тело не воздействуют никакие силы, его линейное ускорение равно нулю, а вектор линейной скорости остается постоянным. То же самое справедливо и для двухмерной угловой динамики: если у двухмерного тела нет вращательного момента, его угловое ускорение α равно нулю, а угловая скорость ω вокруг оси *Z* не меняется.

К сожалению, в трехмерном пространстве это *не так*. Оказывается, что, даже когда твердое тело вращается не под воздействием каких-либо сил, вектор его угловой скорости $\omega(t)$ может быть непостоянным, поскольку ось вращения способна непрерывно менять наклон. Вы можете наблюдать этот эффект: попытайтесь раскрутить в воздухе перед собой прямоугольный объект, такой как деревянный брусок. Если подбросить его так, чтобы он вращался вокруг своей самой короткой оси, движение будет стабильным, а направление оси — более или менее постоянным. То же самое произойдет, если раскрутить брусок вокруг его самой длинной оси. Однако вращение вокруг оставшейся оси (не самой длинной, но и не самой короткой) будет крайне нестабильным (попробуйте сами: найдите где-нибудь деревянный брусок и покрутите его разными способами, а когда закончите, не забудьте вернуть на место). При вращении объекта кардинальным образом меняются направление и сама ось вращения (рис. 13.26).

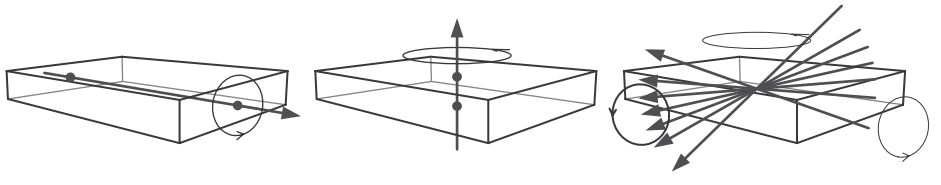


Рис. 13.26. Прямоугольный объект, вращающийся вокруг своей самой короткой или длинной оси, имеет постоянный вектор угловой скорости. Но если вращение происходит вокруг оси среднего размера, направление вектора угловой скорости кардинально меняется

То, что вектор угловой скорости может меняться в отсутствие вращательных моментов, означает, что вектор угловой скорости не является сохраняющимся. Однако связанная с ним величина под названием *«угловой момент»* остается постоянной и, следовательно, *является* сохраняющейся в отсутствие каких-либо сил. Угловой момент служит вращательным эквивалентом импульса (линейного момента):

Угловое представление:	Линейное представление:
$\mathbf{L}(t) = \mathbf{I}\boldsymbol{\omega}(t)$	$\mathbf{p}(t) = m\mathbf{v}(t).$

Как и в линейном случае, угловой момент $\mathbf{L}(t)$ является трехэлементным вектором. Но разница в том, что вращательная масса (тензор инерции) — не скалярная величина, а скорее матрица размером 3×3. Таким образом, выражение $\mathbf{I}\boldsymbol{\omega}$ вычисляется путем умножения матриц:

$$\begin{bmatrix} L_x(t) \\ L_y(t) \\ L_z(t) \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} \omega_x(t) \\ \omega_y(t) \\ \omega_z(t) \end{bmatrix}.$$

Поскольку вектор угловой скорости ω не сохраняется, в симуляциях динамики он не играет роль первичной величины, как линейный вектор $\mathbf{v}$. Вместо этого в качестве первичной величины берется угловой момент $\mathbf{L}$. Момент угловой скорости

является вторичной величиной, которая на каждом шаге симуляции определяется только после вычисления **L**.

Вращательный момент в трехмерном пространстве

В трехмерном пространстве вращательный момент по-прежнему вычисляется путем векторного произведения вектора радиального положения точки приложения силы и самого вектора силы ($\mathbf{N} = \mathbf{r} \times \mathbf{F}$). Уравнение (13.8) все еще справедливо, но его всегда записывают с помощью углового момента, поскольку вектор угловой скорости не является сохраняющимся:

$$\mathbf{N} = \mathbf{I}\boldsymbol{\alpha}(t) = \mathbf{I}\frac{d\boldsymbol{\omega}(t)}{dt} = \frac{d}{dt}\left(\mathbf{I}\boldsymbol{\omega}(t)\right) = \frac{d\mathbf{L}(t)}{dt}.$$

Решение уравнений углового движения в трехмерном пространстве

При решении уравнений углового движения в трехмерном пространстве может возникнуть соблазн воспользоваться тем же подходом, который мы применяли для линейного и плоского углового движения. В результате запишем следующие дифференциальные уравнения движения:

Угловое 3D-представление:	Линейное представление:
$\mathbf{N}_{\text{net}}(t) = \mathbf{I}\dot{\boldsymbol{\omega}}(t);$	$\mathbf{F}_{\text{net}}(t) = m\dot{\mathbf{v}}(t);$
$\boldsymbol{\omega}(t) = \dot{\boldsymbol{\theta}}(t)$	$\mathbf{v}(t) = \dot{\mathbf{r}}(t).$

Затем по наитию можно использовать явный метод Эйлера и записать приближенные решения этих ОДУ таким образом:

Угловое 3D-представление:	Линейное представление:
$\boldsymbol{\omega}(t_2) = \boldsymbol{\omega}(t_1) + \mathbf{I}^{-1}\mathbf{N}_{\text{net}}(t_1)\Delta t;$	$\mathbf{v}(t_2) = \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t)\Delta t;$
$\boldsymbol{\theta}(t_2) = \boldsymbol{\theta}(t_1) + \boldsymbol{\omega}(t_1)\Delta t$	$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t.$

На самом деле это *неправильно*. Дифференциальные уравнения трехмерного углового движения имеют два важных отличия по сравнению с их линейными и двухмерными вращательными аналогами.

1. Сначала мы ищем не вектор угловой скорости ω, а сразу угловой момент **L**. Затем вектор угловой скорости вычисляется как вторичная величина на основе **I** и **L**. Так делается потому, что угловой момент сохраняется, а вектор угловой скорости — нет.
2. При вычислении направления на основе вектора угловой скорости возникает проблема: последний состоит из *трех элементов*, тогда как направление представлено *четырехэлементным* кватернионом. Как можно записать отношение между кватернионом и вектором в виде ОДУ? Ответ прост: никак (по крайней

мере не напрямую). Однако мы можем преобразовать вектор угловой скорости в кватернион и затем применить немного странно выглядящее уравнение, которое связывает кватернионы направления и угловой скорости.

Оказывается, если выразить направление твердого тела в виде кватерниона, его производная будет относиться к вектору угловой скорости тела следующим образом. Сначала мы сформируем *кватернион угловой скорости*. Помимо трех компонентов вектора угловой скорости, *x*, *y* и *z*, он будет содержать компонент ω со значением 0:

$$\boldsymbol{\omega} = \begin{bmatrix} \omega_x & \omega_y & \omega_z & 0 \end{bmatrix}.$$

Теперь дифференциальное уравнение, связывающее кватернионы направления и угловой скорости, выглядит так (не станем углубляться в причины):

$$\frac{d\boldsymbol{\omega}(t)}{dt} = \dot{\mathrm{q}}(t) = \frac{1}{2}\boldsymbol{\omega}(t)\mathrm{q}(t).$$

Следует помнить, что $\omega(t)$, как описано ранее, — это *кватернион* угловой скорости, а $\omega(t)\mathrm{q}(t)$ — произведение *кватернионов* (подробнее об этом говорилось в подразделе 5.4.2).

Поэтому ОДУ движения на самом деле нужно записать так (обратите внимание на то, что я также представил эти уравнения с точки зрения линейного момента, чтобы подчеркнуть схожесть двух случаев):

Угловое 3D-представление:	Линейное представление:
$\mathbf{N}_{\text{net}}(t) = \dot{\mathbf{L}}(t);$	$\mathbf{F}_{\text{net}}(t) = \dot{\mathbf{p}}(t);$
$\boldsymbol{\omega}(t) = \mathbf{I}^{-1}\mathbf{L}(t);$	$\mathbf{v}(t) = m^{-1}\mathbf{p}(t);$
$\boldsymbol{\omega}(t) = \begin{bmatrix} \boldsymbol{\omega}(t) & 0 \end{bmatrix};$	$\mathbf{v}(t) = \dot{\mathbf{r}}(t).$
$\frac{1}{2}\boldsymbol{\omega}(t)\mathrm{q}(t) = \dot{\mathrm{q}}(t)$	

Если использовать явный метод Эйлера, итоговое приближенное решение угловых ОДУ в трехмерном пространстве будет выглядеть так:

$$\mathbf{L}(t_2) = \mathbf{L}(t_1) + \mathbf{N}_{\text{net}}(t_1)\Delta t = \mathbf{L}(t_1) + \Delta t\sum_{\forall i}\left(\mathbf{r}_i \times \mathbf{F}_i(t_1)\right); \quad (\text{векторы})$$

$$\boldsymbol{\omega}(t_2) = \begin{bmatrix} \mathbf{I}^{-1}\mathbf{L}(t_2) & 0 \end{bmatrix}; \; \mathrm{q}(t_2) = \mathrm{q}(t_1) + \frac{1}{2}\boldsymbol{\omega}(t_1)\mathrm{q}(t_1)\Delta t. \quad (\text{кватернионы})$$

Кватернион направления $\mathrm{q}(t)$ следует периодически нормализовать, чтобы нивелировать последствия погрешности, которые неизбежно накапливаются при работе с числами с плавающей запятой.

Как и прежде, явный метод Эйлера взят здесь лишь в качестве примера. В реальном движке мы бы применили метод Стёрмера — Верле с вектором скорости, RK4 или какой-то другой подход с лучшими точностью и стабильностью.

13.4.7. Реакция на столкновение

До сих пор мы подразумевали, что наши твердые тела ни с чем не сталкиваются и их движение ничем не ограничено. Система симуляции динамики должна обеспечивать реалистичное поведение тел при столкновении и следить за тем, чтобы по завершении этапа симуляции они никогда не оставались взаимно пересекающимися. Это называют *реакцией на столкновение*.

Энергия

Прежде чем переходить к обсуждению реакции на столкновение, следует разобраться с концепцией *энергии*. Когда сила перемещает тело на какое-то расстояние, говорят, что она выполнила *работу*. Работа представляет изменение энергии, то есть сила либо привносит энергию в систему твердых тел (как в случае с взрывом), либо отнимает ее (как в результате трения). Энергия бывает двух видов. Тело обладает потенциальной энергией V просто по факту своего расположения относительно силового поля, такого как гравитационное или магнитное (чем выше тело находится над поверхностью Земли, тем больше его потенциальная гравитационная энергия). Кинетическая энергия T обусловлена тем, что тело движется относительно других тел в системе. Общая энергия $E = V + T$ изолированной системы тел является *сохраняющейся* величиной, то есть она остается постоянной, если система не теряет или не получает дополнительную энергию.

Кинетическую энергию, возникающую в результате линейного движения, можно записать как:

$$T_{\text{linear}} = \frac{1}{2}mv^2 -$$

или с помощью векторов импульса и скорости:

$$T_{\text{linear}} = \frac{1}{2}\mathbf{p} \cdot \mathbf{v}.$$

Кинетическую энергию, возникающую в результате вращательного движения тела, можно выразить аналогично:

$$T_{\text{angular}} = \frac{1}{2}\mathbf{L} \cdot \boldsymbol{\omega}.$$

Концепции энергии и ее сохранения могут оказаться чрезвычайно полезными при решении всевозможных задач в физике. В следующем разделе мы увидим, какую роль она играет в определении реакции на столкновения.

Импульсная реакция на столкновение

Когда в реальном мире сталкиваются два тела, происходит сложная череда событий. Тела немного сжимаются и расширяются, меняют векторы своего движения и теряют энергию, которая переходит в звук и тепло. В большинстве симуляций динамики твердых тел, происходящих в реальном времени, все эти подробности

вычисляются приближенно с помощью простой модели, основанной на анализе моментов силы и кинетической энергии сталкивающихся объектов. Эта модель называется *законом Ньютона о восстановлении для моментальных столкновений без трения*, в нее заложены следующие упрощения.

- Сила столкновения действует на протяжении бесконечно малого периода времени, что превращает ее в так называемый идеализированный *импульс*. В результате столкновения векторы скорости тел меняются *моментально*.
- В точке контакта поверхностей тел нет никакого трения. Иными словами, импульс, стремящийся разделить тела во время столкновения, является нормалью к обеим поверхностям — у импульса столкновения нет тангенциального компонента (конечно, это идеализированный случай; мы поговорим о трении далее).
- Природу сложных субмолекулярных взаимодействий между телами во время столкновения можно с определенной степенью приближения представить в виде величины под названием *«коэффициент упругого восстановления»*, которую принято обозначать ε. Этот коэффициент определяет то, сколько энергии теряется во время столкновения. Когда $\varepsilon = 1$, столкновение получается идеально упругим, без потери энергии (представьте себе два бильярдных шара, которые ударяются друг о друга в воздухе). Когда $\varepsilon = 0$, столкновение является идеально *неупругим* (или идеально *пластичным*), а кинетическая энергия тел теряется. Тела слипнутся и продолжат двигаться вместе в направлении, в котором двигался их общий центр массы до столкновения (как будто склеились куски теста).

Анализ столкновения основан на том, что момент сохраняется. Поэтому для двух тел, 1 и 2, можно написать:

$$\mathbf{p}_1 + \mathbf{p}_2 = \mathbf{p}_1' + \mathbf{p}_2'$$

или

$$m_1\mathbf{v}_1 + m_2\mathbf{v}_2 = m_1'\mathbf{v}_1' + m_2'\mathbf{v}_2',$$

где символы со штрихами представляют моменты и векторы скорости после столкновения. Кинетическая энергия системы тоже сохраняется, но мы также должны учесть энергию, перешедшую в звук и тепло, для чего введем дополнительный элемент потери энергии T_{lost}:

$$\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1'v_1'^2 + \frac{1}{2}m_2'v_2'^2 + T_{\text{lost}}.$$

Если столкновение является идеально *упругим*, потеря энергии T_{lost} равна нулю. Если оно идеально *пластичное*, T_{lost} будет равно исходной кинетической энергии системы, суммарная кинетическая энергия (со штрихом) обнулится, а тела после столкновения останутся сцепленными.

Чтобы просчитать столкновение с помощью ньютоновского закона восстановления, применим к двум телам идеализированный *импульс*. Импульс — это как

сила, которая действует на протяжении бесконечно короткого периода времени и тем самым вызывает мгновенные изменения вектора скорости тела, к которому она приложена. Мы можем обозначить импульс $\Delta\mathbf{p}$, так как он представляет собой изменение момента ($\Delta\mathbf{p} = m\Delta\mathbf{v}$). Однако в физической литературе вместо этого в основном используется символ $\hat{\mathbf{p}}$ (произносится «р со шляпкой»), поэтому сделаем то же самое.

Мы исходим из того, что при столкновении не происходит никакого трения, поэтому вектор импульса должен быть нормалью к обеим поверхностям в точке контакта. Иными словами, $\hat{\mathbf{p}} = \hat{p}\mathbf{n}$, где $\mathbf{n}$ — единичный вектор нормали к обеим поверхностям (рис. 13.27). Если предположить, что эта нормаль направлена к телу 1, само тело получит импульс $\hat{\mathbf{p}}$, в то же время тело 2 получит тот же импульс, но в противоположном направлении $-\hat{\mathbf{p}}$. Следовательно, моменты двух тел после столкновения можно записать с применением их моментов до столкновения и импульса $\hat{\mathbf{p}}$:

$$\begin{gathered}\mathbf{p}_1' = \mathbf{p}_1 + \hat{\mathbf{p}};\ \mathbf{p}_2' = \mathbf{p}_2 - \hat{\mathbf{p}};\\ m_1\mathbf{v}_1' = m_1\mathbf{v}_1 + \hat{\mathbf{p}};\ m_2\mathbf{v}_2' = m_2\mathbf{v}_2 - \hat{\mathbf{p}};\\ \mathbf{v}_1' = \mathbf{v}_1 + \frac{\hat{\mathbf{p}}}{m_1}\mathbf{n};\ \mathbf{v}_2' = \mathbf{v}_2 + \frac{\hat{\mathbf{p}}}{m_2}\mathbf{n}.\end{gathered} \tag{13.9}$$

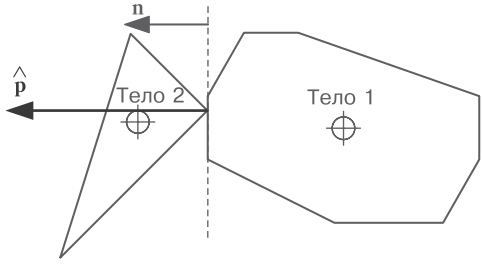


Рис. 13.27. При столкновении без трения импульс направлен вдоль нормали к обеим поверхностям в точке контакта. Эта нормаль определяется единичным вектором n

Коэффициент упругого восстановления создает ключевую связь между векторами относительных скоростей тел до и после столкновения. Если центры массы тел имеют векторы скорости до и после, коэффициент упругого восстановления ε можно определить так:

$$\left(\mathbf{v}_2' - \mathbf{v}_1'\right) = \varepsilon\left(\mathbf{v}_2 - \mathbf{v}_1\right). \tag{13.10}$$

Если решить уравнения (13.9) и (13.10), временно предположив, что тела не могут вращаться, получится:

$$\hat{\mathbf{p}} = \hat{p}\mathbf{n} = \frac{(\varepsilon+1)\left(\mathbf{v}_2\cdot\mathbf{n} - \mathbf{v}_1\cdot\mathbf{n}\right)}{\left(1/m_1\right) + \left(1/m_2\right)}\mathbf{n}.$$

Обратите внимание на то, что, если коэффициент упругого восстановления равен 1 (идеально упругое столкновение) и масса тела 2 фактически безгранична

(скажем, как в случае с бетонной дорогой), получается, что $(1 / m_2) = 0$, $\mathbf{v}_2 = 0$, в результате чего данное выражение сводится к отображению вектора скорости другого тела от нормали контакта, как мы и ожидали:

$$\hat{\mathbf{p}} = -2m_1(\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n};$$

$$\mathbf{v}_1' = \frac{\mathbf{p}_1 + \mathbf{p}_2}{m_1} = \frac{m_1\mathbf{v}_1 - 2m_1(\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n}}{m_1} = \mathbf{v}_1 - 2m_1(\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n}.$$

Решение будет выглядеть не так аккуратно, когда мы начнем учитывать вращение тел. Нам придется работать с векторами скорости точек контакта двух тел, а не их центров массы, а импульс нужно будет вычислить так, чтобы придать телам реалистичный эффект вращения, вызванный столкновением. Я не стану углубляться в подробности, однако вы можете прочесть замечательную статью Криса Хекера с описанием линейных и вращательных аспектов реакции на столкновение: http://chrishecker.com/images/e/e7/Gdmphys3.pdf. Теория, лежащая в основе реакции, более полно объясняется в [17].

Штрафные силы

Еще один подход к реакции на столкновение состоит в том, чтобы добавить в симуляцию воображаемые *штрафные силы*. Штрафная сила ведет себя как тугая пружина, прикрепленная к точкам контакта двух тел, которые только-только пересеклись. Она вызывает нужную нам реакцию на столкновение на протяжении короткого (но не бесконечно короткого) периода времени. Благодаря этому подходу постоянная упругости пружины k фактически определяет продолжительность пересечения, а коэффициент затухания b в какой-то мере ведет себя как коэффициент упругого восстановления. Когда $b = 0$, затухания нет, энергия не теряется, а столкновение получается идеально упругим. С увеличением b столкновение становится все более пластичным.

Давайте кратко рассмотрим некоторые положительные и отрицательные стороны использования штрафных сил для реакции на столкновения. Преимущество этого подхода в том, что штрафные силы легко понять и реализовать. Кроме того, они хорошо работают при взаимном пересечении трех и более тел. Эту ситуацию очень сложно разрешить, если принимать во внимание только отдельные пары объектов. Хорошим примером является демонстрационная симуляция Sony PS3, во время которой в ванную высыпают множество резиновых уточек: несмотря на очень большое количество столкновений, она выглядела красиво и работала стабильно. Штрафная сила является отличным способом этого достичь.

К сожалению, штрафные силы реагируют на проникновение (то есть относительное положение), а не на относительный вектор скорости, поэтому их направление может показаться неожиданным, особенно при высокоскоростных столкновениях. Классическим примером является легковой автомобиль, врезающийся в грузовик лоб в лоб. Легковой автомобиль низкий, а грузовой — высокий.

Если использовать только метод штрафных сил, можно легко получить ситуацию, когда штрафная сила окажется вертикальной, а не горизонтальной, чего можно было бы ожидать, исходя из векторов скорости двух транспортных средств. В итоге передняя часть грузовика может подскочить и легковой автомобиль проедет под ним.

В целом метод штрафных сил хорошо работает при столкновениях на низких скоростях, но когда объекты движутся быстро, он дает плохие результаты. Вы можете объединить этот подход с другими методами разрешения столкновений, чтобы получить баланс между стабильностью при большом количестве взаимных проникновений и более интуитивным поведением на высоких скоростях.

Разрешение столкновений с помощью ограничений

Как мы увидим в подразделе 13.4.8, большинство систем физики позволяют накладывать различные виды ограничений на движение тел в симуляции. Если воспринимать столкновения в качестве ограничений, которые не дают объектам пересечься, мы можем воспользоваться универсальным механизмом разрешения ограничений в симуляции. Если этот механизм работает быстро и выдает высококачественные визуальные результаты, это может стать хорошим видом реакции на столкновения.

Трение

Трение — это сила, которая возникает между телами, находящимися в непрерывном контакте, и препятствует их движению друг относительно друга. Существует множество разновидностей трения. *Статическое трение* — это сопротивление, которое мы ощущаем, когда пытаемся подвинуть статический объект на какой-то поверхности. *Динамическое трение* — это сила сопротивления, которая возникает, когда объекты движутся друг относительно друга. *Сила трения скольжения* — это разновидность динамического трения, препятствующая движению объекта, когда тот скользит по поверхности. *Трение качения* — разновидность статического или динамического трения, которая действует в точке контакта между колесом (или другим круглым объектом) и поверхностью, по которой оно катится. Если поверхность очень грубая, трение качения достаточно сильное для того, чтобы колесо не проскальзывало. В этом случае оно играет роль статического трения. Если поверхность довольно гладкая, колесо может скользить, в результате чего возникает динамическое трение. В точке контакта двух движущихся тел мгновенно возникает *трение столкновения* (это сила, которую мы игнорировали во время обсуждения ньютоновского закона восстановления ранее). Разные виды ограничений тоже могут испытывать трение. Например, ржавые шарниры или поперечные балки создают вращательное трение, когда их пытаются повернуть.

Чтобы понять принцип работы трения, рассмотрим пример. Трение линейного скольжения пропорционально весу объекта, направленному перпендикулярно поверхности, по которой тот скользит. Вес объекта обусловлен силой тяжести $\mathbf{G} = m\mathbf{g}$, которая всегда направлена вниз. Компонента этой силы, действующая перпенди-

кулярно поверхности, наклоненной под углом θ относительно горизонтальной плоскости, равна $G_N = mg\cos\theta$. Следовательно, сила трения

$$f = \mu mg\cos\theta,$$

где константа пропорциональности μ называется *коэффициентом трения*. Эта сила действует под углом к поверхности в направлении, противоположном предпринятому или фактическому движению объекта (рис. 13.28).

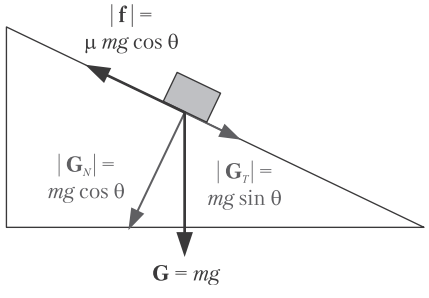


Рис. 13.28. Сила трения f пропорциональна перпендикулярной компоненте веса объекта

На рис. 13.28 также показана компонента силы тяжести, направленная под углом к поверхности, $G_T = mg\sin\theta$. Эта сила заставляет объект ускоряться вниз по плоскости, однако при наличии силы трения скольжения она нейтрализуется за счет f. Таким образом, результирующая сила, действующая под углом к поверхности:

$$F_{\text{net}} = G_T - f = mg\left(\sin\theta - \mu\cos\theta\right).$$

Если угол наклона делает выражение в скобках равным нулю, объект будет скользить по поверхности с постоянной скоростью (если изначально он движется) или находиться в состоянии покоя. Если выражение больше нуля, объект станет ускоряться. Если оно меньше нуля, объект будет замедляться и в итоге остановится.

Сваривание

Когда объект скользит по полигональному супу, возникает еще одна проблема. Как вы помните, полигональный суп — это, в сущности, набор не связанных между собой полигонов (обычно треугольников). Когда объект скользит от одного треугольника к другому, система обнаружения столкновений генерирует дополнительные ложные контакты: она думает, что объект вот-вот столкнется с гранью следующего треугольника (рис. 13.29).

У этой проблемы есть целый ряд решений. Например, мы можем проанализировать набор контактов и отклонить те из них, которые выглядят ложными, руководствуясь различными эвристическими правилами и, возможно, какой-то информацией о контактах объекта в предыдущем кадре (например, если мы знаем, что объект скользил по поверхности и что нормаль контакта возникает в результате его приближения к грани текущего треугольника, эту нормаль можно отклонить). В Havok этот подход использовался до выхода версии 4.5.

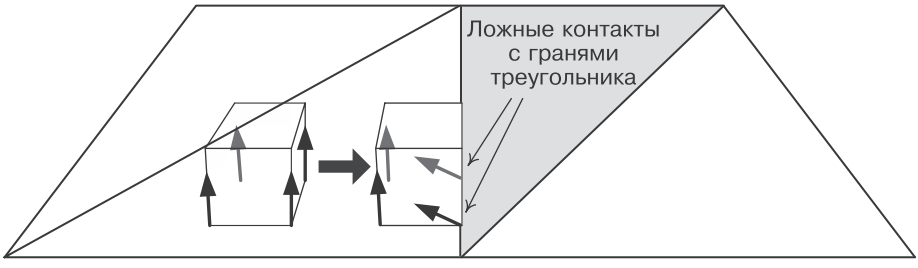


Рис. 13.29. Когда объект скользит по полигональному супу, у него могут возникать ложные контакты с гранями треугольников

В Havok 4.5 была реализована новая методика, которая позволяет хранить в меше сведения о смежности треугольников. Благодаря этому система обнаружения столкновений знает, какие грани являются внутренними, и может быстро и надежно отбрасывать ложные срабатывания. В Havok это решение называется *свариванием* (welding), поскольку грани треугольников в полигональном супе фактически привариваются друг к другу.

Переход в состояние покоя, острова и засыпание

Когда трение, затухание или что-то другое забирает всю энергию из системы симуляции, объекты в конечном счете останавливаются. Это выглядит как естественный результат симуляции — нечто вытекающее из дифференциальных уравнений движения. К сожалению, в настоящей компьютерной симуляции переход в состояние покоя никогда не достигается так просто. Различные факторы, такие как погрешность операций с плавающей запятой, неточности в вычислении сил упругого восстановления и вычислительная неустойчивость, могут вызвать вечную вибрацию объектов, не давая им остановиться. В связи с этим большинство физических движков используют различные эвристические методы для обнаружения ситуаций, когда вместо ожидаемого прекращения движения объекты продолжают колебаться. Чтобы объекты остановились, из системы можно удалить остаточную энергию, также их можно резко останавливать, когда вектор их средней скорости опускается ниже какого-то порогового значения.

Когда объект по-настоящему останавливается, то есть входит в состояние равновесия, нам больше не нужно интегрировать его уравнения движения в каждом кадре. Чтобы оптимизировать производительность, большинство физических движков позволяют *усыплять* динамические объекты. Это позволяет временно исключить их из симуляции, хотя они по-прежнему могут участвовать в столкновениях. Если на спящий объект начинают действовать какие-либо сила или импульс или если он теряет один из контактов, который поддерживал его равновесие, он просыпается, чтобы его динамическая симуляция могла продолжаться.

Критерии засыпания. Чтобы решить, следует ли усыплять тело, можно руководствоваться различными критериями. В некоторых ситуациях для обеспечения

надежности этого процесса могут потребоваться дополнительные усилия. Например, у длинного маятника может быть очень маленький угловой момент, но при этом его движение будет четко видно на экране.

Среди наиболее распространенных критериев определения равновесия можно выделить следующие.

Тело *поддерживается*. Это означает, что у него есть как минимум три точки контакта или хотя бы один *плоскостной* контакт, которые позволяют ему достичь равновесия с силой тяжести и любыми другими силами, которые могут на него воздействовать.

Линейный и угловой моменты тела ниже заранее определенного порога.

Скользящее среднее линейного и углового моментов ниже заранее определенного порога.

Общая *кинетическая энергия* тела $\left(T = \frac{1}{2}\mathbf{p} \cdot \mathbf{v} + \frac{1}{2}\mathbf{L} \cdot \boldsymbol{\omega} \right)$ ниже заранее определенного порога. Кинетическая энергия обычно нормализуется по массе, чтобы для всех тел с любой массой можно было использовать единый порог.

Движение тела, которое вот-вот должно уснуть, может *постепенно затухать*, чтобы остановка получилась плавной, а не резкой.

Острова симуляции. Havok и PhysX выполняют дополнительную оптимизацию, автоматически объединяя объекты, которые взаимодействуют между собой или могут начать это делать в ближайшем будущем, в множества под названием *«острова симуляции»*. Каждый остров может моделироваться отдельно от других, что очень способствует оптимизации когерентности кэша и параллельной обработке.

В Havok и PhysX засыпают не отдельные тела, а целые острова симуляции. Этот подход имеет свои плюсы и минусы. Очевидно, возможность усыплять целые группы взаимодействующих между собой объектов дает более существенный прирост производительности. В то же время остров остается активным, пока не заснут все его объекты без исключения. Но похоже, что преимущества в целом перевешивают недостатки, поэтому острова симуляции, скорее всего, будут применяться и в будущих версиях этих SDK.

13.4.8. Ограничения

Неограниченное твердое тело имеет шесть степеней свободы: оно может смещаться во всех трех измерениях и вращаться вокруг всех трех осей прямоугольной системы координат. *Ограничения* уменьшают степени свободы либо частично, либо полностью. Их можно использовать для моделирования в игре всевозможных интересных ситуаций. Вот лишь несколько примеров:

- качающаяся люстра (ограничение «точка к точке»);
- дверь, которую можно открыть ногой, захлопнуть, сорвать с петель (шарнирное ограничение);

- механизм колеса автомобиля (ограничение оси с затухающими пружинами для подвески);
- поезд или автомобиль с прицепом (ограничение тугой пружины или стержня);
- веревка или цепь (цепочка тугих пружин или стержней);
- тряпичная кукла (специальное ограничение, имитирующее поведение различных суставов человеческого скелета).

В следующих разделах мы кратко пройдемся по этим и некоторым другим видам ограничений, которые обычно доступны в SDK для симуляции физики.

Ограничения вида «точка к точке»

«Точка к точке» — самое простое ограничение. Оно ведет себя как шаровидный сустав: тела могут двигаться как угодно при условии, что заданная точка одного тела совпадает с заданной точкой другого (рис. 13.30).

Тугие пружины

Ограничение тугой пружины во многом похоже на ограничение «точка к точке», только оно удерживает две точки на заданном расстоянии. Все выглядит так, будто между двумя точками находится невидимый стержень (рис. 13.31).

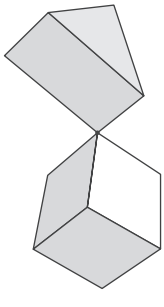


Рис. 13.30. Ограничение «точка к точке» требует, чтобы точка на теле A совпадала с точкой на теле B

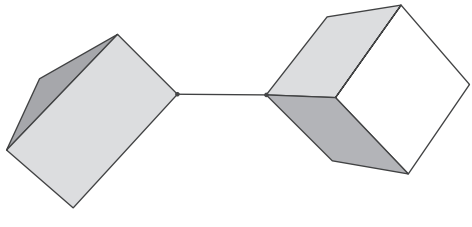


Рис. 13.31. Ограничение тугой пружины требует, чтобы точка на теле A находилась на определенном расстоянии от точки на теле B

Шарнирные ограничения

Шарнирное ограничение оставляет вращательному движению лишь одну степень свободы вокруг оси шарнира. *Неограниченный шарнир* ведет себя как поперечная балка, позволяя выполнять полное вращение. Шарниры часто *ограничивают*, чтобы они могли поворачиваться только вокруг одной оси и только на определенный угол. Например, односторонняя дверь может вращаться лишь по дуге 180°, иначе она начнет проходить через стену. Точно так же дверь, открывающаяся в обе стороны, может вращаться лишь по дуге ±180°. Шарнирные ограничения могут

иметь определенную степень трения в виде вращательного момента, направленного против вращения шарнира вокруг его оси. Пример частичного шарнирного ограничения приведен на рис. 13.32.

Призматические ограничения

Призматические ограничения ведут себя как поршень (рис. 13.33): движение тела имеет лишь одну степень свободы, относящуюся к смещению. Призматическое ограничение может позволять вращение вокруг оси смещения поршня. Оно, конечно же, может быть полным или частичным и включать в себя трение.

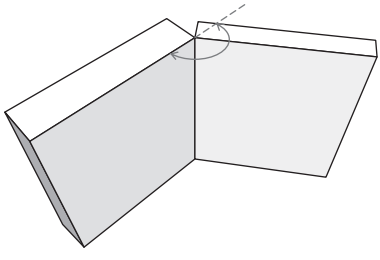


Рис. 13.32. Частичное шарнирное ограничение имитирует поведение двери

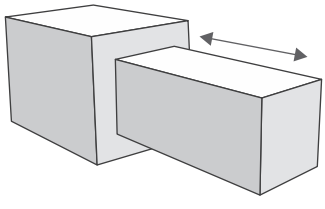


Рис. 13.33. Призматическое ограничение ведет себя как поршень

Другие распространенные виды ограничений

Конечно, существуют и другие виды ограничений. Вот лишь несколько примеров.

- *Плоскостное*. Объекты могут двигаться только в одной плоскости.
- *Колесовидное*. Обычно это шарнирное ограничение с полным вращением в сочетании с какого-то рода вертикальной подвеской, которая симулируется с помощью механизма затухающих пружин.
- *Блочное*. В этом особом ограничении воображаемая веревка, соединяющая два тела, проходит через блок (https://ru.wikipedia.org/wiki/Блок_(механика)). Тела движутся вдоль веревки по принципу рычага.

Ограничения могут быть ломкими, это означает, что в случае применения достаточной силы они могут разрушаться. Кроме того, игра может включать и выключать ограничение, руководствуясь собственными критериями того, в какой момент оно должно быть преодолено.

Цепочки ограничений

Иногда длинные цепи связанных между собой объектов сложно симулировать так, чтобы они оставались стабильными. Это вызвано итеративной природой механизма ограничений. На такой случай предусмотрены *цепочки ограничений*, они содержат информацию о том, как именно соединены объекты. Это позволяет механизму ограничений сделать цепь более стабильной.

Тряпичные куклы

Тряпичная кукла — это физическая симуляция движений человеческого тела в случае смерти или потери сознания, то есть бесчувственного. Тряпичные куклы создаются соединением между собой твердых тел — по одному для каждой полутвердой части тела. Например, у нас могут быть капсулы для ступней, голеней, бедер, ладоней, плеч, предплечий, головы и, возможно, еще несколько для туловища, чтобы симулировать гибкость спины.

Твердые тела в тряпичной кукле соединяются между собой с помочью ограничений, созданных специально для того, чтобы имитировать движения суставов, на которые способно настоящее человеческое тело. Для повышения стабильности симуляции обычно применяются цепочки ограничений.

Симуляция тряпичной куклы всегда тесно интегрирована с системой анимации. По мере движения тряпичной куклы в мире физики мы извлекаем положение и степень вращения твердых тел, а затем используем эту информацию для смещения и вращения определенных суставов в анимированном скелете. Получается, что тряпичная кукла — это лишь разновидность *процедурной анимации*, управляемая системой физики (подробнее о скелетной анимации говорится в главе 12).

Конечно, реализовать тряпичную куклу не так просто, как я это обрисовал. Например, не всегда есть прямое соответствие между твердыми телами тряпичной куклы и суставами скелета — обычно первых меньше, чем последних. Следовательно, нужен какой-то механизм, который мог бы привязать твердые тела к суставам (он должен знать, какому суставу соответствует каждое твердое тело в тряпичной кукле). В скелете могут быть дополнительные суставы, размещенные между теми, которые управляются твердыми телами тряпичной куклы, поэтому система связывания должна уметь находить правильные преобразования поз для промежуточных суставов. Это не точная наука. Чтобы получить естественно выглядящую тряпичную куклу, требуются чувство прекрасного и/или знания в области человеческой биомеханики.

Механизированные ограничения

Ограничения могут быть механизированными в том смысле, что внешняя часть движка, такая как система анимации, может опосредованно управлять смещением и направлением твердых тел в тряпичной кукле.

Для примера возьмем локтевой сустав. Локоть ведет себя практически как ограниченный шарнир с углом свободного вращения чуть меньше 180° (на самом деле он может вращаться и в продольном направлении, но здесь мы это проигнорируем). Чтобы механизировать это ограничение, смоделируем локоть в виде *вращательной пружины*. Она имеет вращательный момент, пропорциональный ее углу отклонения от некоего заранее определенного направления покоя, $N = -k(\theta - \theta_{rest})$. Теперь представьте, что мы меняем угол покоя извне, делая так, чтобы он всегда совпадал с углом локтевого сустава в анимированном скелете.

При изменении этого угла пружина теряет равновесие и получает вращательный момент, который стремится вернуть ее в положение с θ_{rest}. В отсутствие каких-либо других сил или вращательных моментов твердые тела будут в точности следовать движению локтевого сустава в анимированном скелете. Но если в системе появятся другие силы (например, если предплечье столкнется с неподвижным объектом), они повлияют на общее движение локтевого сустава, позволяя ему реалистично отклониться от анимации. Это создает иллюзию того, что человек изо всех сил пытается двигаться определенным образом (то есть идеально повторять движения, обусловленные анимацией), но иногда из-за ограничений физического мира ему это не удается (например, рука может за что-то зацепиться, когда он пытается ею взмахнуть) (рис. 13.34).

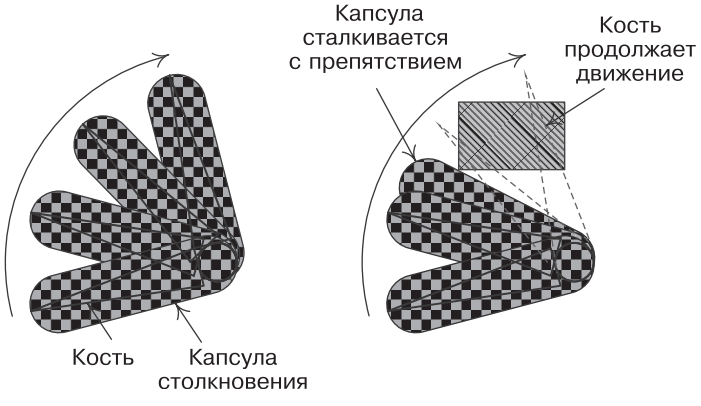


Рис. 13.34. При наличии механизированного ограничения тряпичной куклы и отсутствии любых дополнительных сил или вращающих моментов твердое тело, представляющее предплечье, может в точности следовать движениям анимированного локтевого сустава (*слева*). Если какое-то препятствие блокирует движение тела, оно реалистично отклонится от движения анимированного локтевого сустава

13.4.9. Управление движениями твердых тел

Помимо естественных движений под воздействием силы тяжести и в ответ на столкновения с другими объектами сцены в большинстве игр требуется дополнительный контроль за поведением твердых тел, например:

- вентилятор пытается сдуть любой объект, который находится в зоне его действия;
- автомобиль тянет за собой прицеп во время движения;
- тяговый луч воздействует на космический корабль;
- антигравитатор заставляет объекты парить в воздухе;
- речной поток увлекает плавающие объекты вниз по течению.

Этот список можно продолжать. Большинство физических движков обычно предоставляют своим пользователям различные способы влиять на поведение тел в симуляции. В следующих разделах мы выделим наиболее распространенные из этих механизмов.

Сила тяжести

В большинстве игр, действие которых происходит на поверхности Земли или какой-то другой планеты (или на космическом корабле с искусственной гравитацией), всегда присутствует сила тяжести. На самом деле это не совсем сила, а скорее константа ускорения (грубо говоря), поэтому она одинаково действует на все тела независимо от их массы. Благодаря ее вездесущности и особой природе величина и направление ускорения свободного падения в большинстве SDK заданы в виде глобальных параметров (если вы разрабатываете космическую игру, силу тяжести можно обнулить, чтобы исключить из симуляции).

Применение сил

К телам в игровой симуляции физики можно приложить любое количество сил. Сила всегда действует на протяжении конечного периода времени (если действие происходит моментально, мы называем ее *импульсом*, подробнее об этом говорится в дальнейшем). Силы в игре часто динамичны по своей природе: они меняют направление и/или величину в каждом кадре. Поэтому в большинстве SDK для симуляции физики функция применения силы вызывается в каждом кадре на протяжении действия этой силы. Сигнатура такой функции обычно выглядит примерно так: `applyForce(const Vector& forceInNewtons)`, подразумевается, что время действия силы равно Δt.

Применение вращательного момента

Если вектор действия приложенной силы проходит через центр массы тела, не возникает никакого вращательного момента, меняется только его линейное ускорение. Если сила приложена не по центру, ускорение, которое она создаст, будет как линейным, так и вращательным. Тело может получить и *сугубо вращательный момент*, если к его точкам, равноудаленным от центра массы, применить две одинаковые, но противоположные силы. Линейные движения, вызванные этими силами, нивелируют друг друга, так как с точки зрения линейной динамики обе силы приложены к центру массы. В результате остается лишь вращательное воздействие. Подобные парные силы, генерирующие вращательный момент, называются *парами сил* (ru.wikipedia.org/wiki/Пара_сил). Для этих целей может быть предоставлена специальная функция вида `applyTorque(const Vector& torque)`. Но если в вашем SDK симуляции физики нет функции `applyTorque()`, вы всегда можете написать ее сами и сделать так, чтобы она создавала подходящую пару сил.

Применение импульсов

Как говорилось ранее, *импульс* представляет собой мгновенное изменение вектора скорости (или, если точнее, изменение момента). Строго говоря, импульс — это сила, которая действует на протяжении бесконечно короткого промежутка времени. Но теоретически самая малая продолжительность приложения силы в симуляции с дискретным временем равна Δt, что слишком долго для того, чтобы как следует имитировать импульс. Поэтому большинство физических движков предоставляют функцию с сигнатурой вида `applyImpulse(const Vector& impulse)`, позволяющую применять импульсы к телам. Импульсы бывают двух видов, линейные и угловые, и в хорошем SDK должны быть предусмотрены функции для каждого из них.

13.4.10. Отдельный шаг симуляции столкновений/физики

Итак, разобравшись с теорией и некоторыми техническими подробностями реализации системы столкновений и физики, мы можем уделить некоторое внимание тому, как эти системы выполняют обновления в каждом кадре.

Каждый движок столкновений/физики выполняет во время обновления следующие основные действия. Порядок их выполнения может зависеть от SDK, но чаще всего мне встречался примерно такой подход.

1. Силы и вращательные моменты, действующие на тела в мире физики, интегрируются по Δt для определения гипотетического положения и направления движения тел в следующем кадре.
2. Вызывается библиотека обнаружения столкновений, чтобы определить, были ли сгенерированы какие-либо новые контакты между объектами в результате их гипотетических движений (тела обычно отслеживают свои контакты, чтобы использовать преимущества временной когерентности, следовательно, на каждом шаге симуляции движку столкновений нужно лишь определить, какие предыдущие контакты были потеряны и какие новые контакты добавлены).
3. Разрешение столкновений часто происходит за счет применения импульсов или штрафных сил либо в рамках этапа разрешения ограничений, представленного далее. В зависимости от SDK этот шаг может включать в себя непрерывное обнаружение столкновений, которое называют также временем обнаружения столкновений (time of impact detection, TOI).
4. Ограничения удовлетворяются с помощью соответствующего механизма.

По завершении шага 4 некоторые тела могли сместиться по сравнению с гипотетическими позициями, полученными на шаге 1. Это смещение может вызвать дополнительные пересечения между объектами или нарушение ранее разрешенных ограничений. Таким образом, мы повторяем все четыре шага (иногда только шаги 2–4 в зависимости от того, как разрешаются столкновения и ограничения) до тех

пор, пока не будут успешно разрешены все столкновения и удовлетворены все ограничения или не исчерпается заранее определенное число итераций. Во втором случае механизм разрешения столкновений фактически сдается в надежде на то, что в последующих кадрах симуляции все разрешится само собой. Это позволяет избежать всплесков производительности за счет распределения нагрузки, связанной с вычислением столкновений и разрешением ограничений, между несколькими кадрами. Но при слишком больших погрешностях или слишком длинных/изменчивых промежутках времени это может вызвать поведение, которое выглядит некорректным. Чтобы постепенно решить эти проблемы, в симуляцию можно добавить штрафные силы.

Механизм разрешения ограничений

Механизм разрешения ограничений — это, в сущности, итеративный алгоритм, который пытается одновременно удовлетворить большое количество ограничений, *минимизируя расхождения* между реальными позициями и направлениями тел в мире физики и их идеальными позициями и направлениями, которые эти ограничения определяют. То есть это фактически итеративный алгоритм минимизации погрешностей.

Давайте сначала посмотрим на то, как механизм разрешения ограничений ведет себя в тривиальной ситуации с двумя телами, соединенными одним шарниром. На каждом шаге симуляции физики численный интегратор будет находить новые гипотетические преобразования для этих двух тел. Затем механизм разрешения ограничений определит их относительное положение и вычислит расхождение между позициями и направлениями их общей оси вращения. При этом предпринимается попытка минимизировать или устранить любые обнаруженные расхождения за счет перемещения тел. Поскольку в системе существуют только эти два тела, на второй итерации данного шага не должно быть найдено никаких новых контактов, в результате чего будет установлено, что единственное шарнирное ограничение теперь удовлетворено. Поэтому цикл может завершиться без дальнейших итераций.

Если необходимо удовлетворить сразу несколько ограничений, может потребоваться больше итераций. Иногда во время итераций численный интегратор может переместить тела, нарушив их ограничения, тогда как механизм разрешения ограничений стремится вернуть их обратно. При некоторой доле везения и за счет тщательно выверенного подхода к минимизации расхождений в механизме разрешения ограничений этот цикл с обратной связью должен в конце концов сойтись на корректном решении. Однако результат не всегда бывает точным. Поэтому в играх с физическими движками иногда можно наблюдать невозможные, казалось бы, явления: например цепи, которые растягиваются, создавая небольшие промежутки между звеньями, объекты, пересекающиеся на короткое время, или шарниры, мгновенно выходящие за допустимый диапазон. Механизм разрешения ограничений стремится свести погрешности к минимуму, устранить их полностью не всегда возможно.

Различия между движками

Приведенное ранее описание, конечно же, является упрощенной версией того, что на самом деле происходит при обновлении каждого кадра в движке физики/столкновений. То, как именно и в каком порядке проходят разные этапы вычислений, зависит от конкретного SDK для симуляции физики. Например, некоторые типы ограничений моделируются в виде сил и вращательных моментов, которые обрабатываются на шаге численного интегрирования, а не самим механизмом разрешения ограничений. Столкновение может быть просчитано перед интегрированием, а не после него. Столкновения могут быть разрешены любым из множества способов. Мы лишь хотим, чтобы вы получили представление о том, как работают эти системы. Чтобы разобраться в подробностях работы того или иного SDK, стоит почитать его документацию и, наверное, изучить исходный код (если нужные вам участки доступны в открытом виде). Любопытный и усердный читатель может для начала загрузить Open Dynamics Engine (ODE) и/или PhysX, так как эти два SDK бесплатны, и поэкспериментировать с ними. Много полезной информации можно найти в вики проекта ODE, которая находится по адресу opende.sourceforge.net/wiki/index.php/Main_Page.

13.5. Интеграция физического движка в игру

Очевидно, что сама по себе система столкновений/физики не слишком полезна — она должна быть интегрирована в игровой движок. В этом разделе обсудим наиболее распространенные элементы взаимодействия между движком столкновений/физики и остальным кодом игры.

13.5.1. Связывание игровых объектов и твердых тел

Твердые тела и объекты collidable в мире столкновений/физики являются не более чем абстрактными математическими описаниями. Чтобы их можно было использовать в игре, они должны быть каким-то образом привязаны к своему визуальному представлению на экране. Твердые тела обычно не выводятся напрямую, если не считать этап отладки. Вместо этого с их помощью описываются форма, размер и физическое поведение логических объектов, из которых состоит виртуальный мир игры. Игровые объекты подробно рассмотрим в главе 16, а пока что станем руководствоваться интуитивным пониманием и считать их логическими элементами игрового мира, такими как персонажи, транспортные средства, оружие, парящие в воздухе бонусы и т. д. Таким образом, связь между твердым телом в мире физики и его визуальным представлением на экране обычно не является прямой и реализуется за счет промежуточного игрового объекта (рис. 13.35).

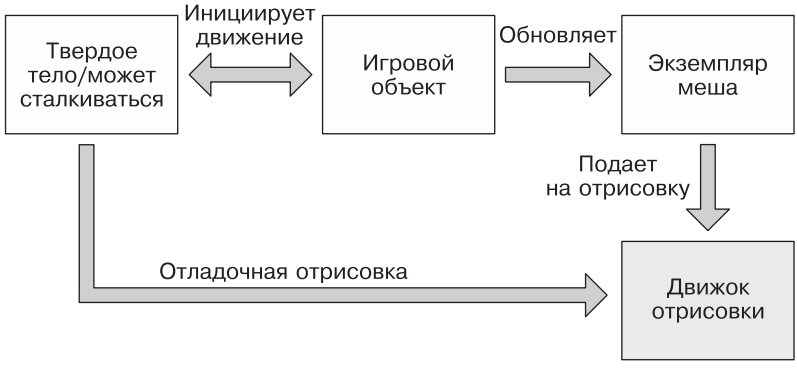


Рис. 13.35. Твердые тела привязываются к своим визуальным представлениям посредством игровых объектов. Обычно также в целях отладки предоставляется возможность прямой отрисовки, чтобы можно было визуализировать положение твердых тел

В целом игровой объект представлен в мире столкновений/физики произвольным количеством твердых тел или не представлен вообще. Далее приведены три возможных сценария.

- *Отсутствие твердых тел.* Игровые объекты, у которых в мире физики нет никаких твердых тел, ведут себя как нематериальные объекты, потому что не участвуют в симуляции столкновений. Это могут быть декоративные элементы, которые не взаимодействуют с игроками и другими персонажами, например пролетающие над головой птицы или видимые, но недостижимые участки игрового мира. Этот сценарий может относиться к объектам, столкновения которых по какой-то причине обрабатываются вручную — без помощи движка столкновений/физики.
- *Одно твердое тело.* Простейшие игровые объекты могут быть представлены одним-единственным твердым телом. В этом случае форма объекта collidable твердого тела выбирается таким образом, чтобы она была близка к форме визуального представления игрового объекта, а положение/направление твердого тела и игрового объекта должны в точности совпадать.
- *Несколько твердых тел.* Некоторые сложные игровые объекты представлены в мире столкновений/физики несколькими твердыми телами. Это касается персонажей, различного оборудования, транспортных средств или любых других объектов, состоящих из нескольких твердых элементов, которые могут двигаться друг относительно друга. Такие игровые объекты обычно используют скелет, то есть иерархию аффинных преобразований, чтобы отслеживать положение своих составляющих (хотя, безусловно, это можно делать и другими способами). Твердое тело и сустав скелета обычно связываются таким образом, чтобы их положение и направление соответствовали друг другу. Суставы скелета могут двигаться за счет анимации, в этом случае связанные с ними твердые тела просто повторяют их движения. Вместо этого также можно задействовать систему

физики, которая определяет положение и направление твердых тел и тем самым косвенно влияет на положение и направление суставов. Связь между суставом и твердым телом может иметь вид «один к одному», но это не обязательно — некоторые суставы могут полностью управляться анимацией, а другие — быть привязанными к твердым телам.

Связью между игровыми объектами и твердыми телами, конечно же, должен управлять движок. Обычно каждый игровой объект сам отвечает за свои твердые тела: создает и уничтожает их по мере необходимости, добавляет их в мир физики и удаляет оттуда, поддерживает связь между положением каждого твердого тела и одного из своих суставов и/или своим общим положением. Для работы со сложными игровыми объектами, состоящими из нескольких твердых тел, может использоваться некий класс-обертка. Это ограждает разные типы игровых объектов от таких низкоуровневых задач, как управление коллекцией твердых тел, и позволяет сделать это управление согласованным.

Тела, движимые физической симуляцией

Если в вашей игре есть система динамики твердых тел, можно предположить, что мы хотим сделать так, чтобы движения как минимум некоторых объектов игры были полностью обусловлены симуляцией. Осколки, взрывающиеся здания, камни, катящиеся по склону холма, пустые обоймы и гильзы — все это примеры объектов, движимых физикой.

Чтобы привязать твердое тело, движимое физикой, к его игровому объекту, мы проходим через разные этапы симуляции и затем запрашиваем у физического движка его положение и направление. Затем применяем полученное преобразование либо ко всему игровому объекту, либо к его суставу или какой-то другой структуре данных внутри него.

Пример: создание сейфа со съемной дверцей. Твердые тела, которые привязывают к суставам скелета, часто *ограничивают*, чтобы добиться от них желаемого движения. Рассмотрим, как можно было бы смоделировать сейф со съемной дверцей.

Предположим, что визуально сейф представляет собой треугольный меш с двумя дочерними мешами: одним для корпуса, другим для дверцы. Для управления движениями этих элементов используется скелет с двумя суставами. Корневой сустав привязан к корпусу сейфа, а дочерний — к дверце. Это сделано таким образом, чтобы вращение дверного сустава выглядело так, будто дверца (точнее, ее меш) открывается и закрывается.

Геометрия столкновений сейфа тоже разбивается на два взаимосвязанных элемента: один для корпуса, другой для дверцы. Эти две части задействуются для создания в мире столкновений/физики двух отдельных твердых тел. Тело корпуса сейфа привязывается к корневому суставу скелета, а тело дверцы — к дверному суставу. Затем в мир физики вводится шарнирное ограничение, чтобы во время симуляции тело дверцы как следует поворачивалось относительно корпуса.

Движения двух твердых тел, представляющих корпус и дверцу, используются для обновления преобразований двух суставов скелета. Когда система анимации сгенерирует палитру матриц скелета, движок отрисовки выведет дочерние меши корпуса и дверцы там, где в мире физики находятся их твердые тела.

Если в какой-то момент дверцу нужно снести, мы можем нарушить ограничение и применить импульсы к твердым телам, чтобы те взлетели в воздух. С точки зрения игрока все будет выглядеть так, будто дверца и корпус превратились в отдельные объекты. Но на самом деле это по-прежнему один игровой объект и один треугольный меш с двумя суставами и двумя твердыми телами.

Тела, движимые игровым процессом

В большинстве игр есть такие игровые объекты, которые необходимо перемещать без учета физики. Их движения могут определяться анимацией или маршрутом на основе сплайна либо ими может управлять живой игрок. Часто такие объекты должны участвовать в симуляции столкновений, чтобы от них могли отскакивать другие элементы, движимые физикой, но при этом система физики не должна никак влиять на их движения. На этот случай в большинстве физических движков предусмотрен специальный тип твердых тел, *управляемых игровым процессом* (в Havok они называются телами на основе ключевых кадров).

На такие тела не действует сила тяжести. Система физики считает их бесконечно тяжелыми (обычно это выражается в нулевой массе, которая считается некорректной для тел в физической симуляции). Благодаря бесконечной массе симулируемые силы и импульсы столкновений принципиально не могут изменить вектор скорости тела, движимого игровым процессом.

Для перемещения такого тела в мире физики мы не можем просто задавать ему положение и направление соответствующего игрового объекта из кадра в кадр. Это создавало бы разрывы, которые было бы сложно сгладить в физической симуляции (например, если тело, управляемое физическим движком, пересечется с телом, которое контролируется игровым процессом, у первого не будет информации об импульсе второго, а без этого нельзя обработать столкновение). В связи с этим тела, движимые игровым процессом, обычно перемещаются с помощью импульсов — мгновенных изменений вектора скорости, которые, если их проинтегрировать по времени, разместят эти тела в нужном месте в конце временного интервала. Большинство SDK для симуляции физики предоставляют удобную функцию, которая вычисляет линейный и угловой импульсы, необходимые для достижения в следующем кадре нужных положения и направления. Чтобы такие тела могли остановиться, нужно не забыть обнулить вектор скорости, иначе они будут бесконечно продолжать движение по последней ненулевой траектории.

Пример: анимированная дверца сейфа. Вернемся к примеру с сейфом и съемной дверцей. Представьте, что персонаж должен подойти к сейфу, набрать код, открыть дверцу, положить внутрь деньги и запереть сейф. Позже другой персонаж должен достать эти деньги менее цивилизованным способом — с помощью взрывчатки. Для этого к сейфу нужно добавить еще один дочерний меш для кодового замка и до-

полнительный сустав, который позволяет его прокручивать. Замок может обойтись без твердого тела, разве что мы хотим, чтобы он тоже отлетел при взрыве.

Когда в ходе анимации персонаж открывает и закрывает сейф, его твердые тела можно перевести в режим управления игровым процессом. В результате анимация будет управлять движениями суставов, которые, в свою очередь, будут двигать твердые тела. Позже, когда сейф нужно будет взорвать, мы можем переключить твердые тела в режим физической симуляции, избавиться от шарнирного ограничения, применить импульс и наблюдать за тем, как отлетает дверца.

Как вы уже, наверное, заметили, в этом конкретном примере можно обойтись и без шарнирного ограничения. Оно нужно только в случае, если в какой-то момент дверца должна оставаться открытой и если мы хотим, чтобы она покачивалась естественным образом при контакте с другими объектами или перемещении сейфа.

Зафиксированные тела

Большинство игровых миров состоят как из статической геометрии, так и из динамических объектов. Для моделирования статических компонентов в большинстве SDK симуляции физики предусмотрены особые *зафиксированные тела*. Своим поведением они немного напоминают тела, движимые игровым процессом, но при этом полностью исключены из симуляции динамики. Они фактически участвуют только в обнаружении столкновений. В большинстве игр такая оптимизация может дать существенный прирост производительности, особенно если речь идет о небольшом количестве динамических объектов, перемещаемых по огромному статическому миру.

Типы движений в Havok

В Havok любые виды твердых тел представлены экземплярами класса `hkpRigidBody`. Каждый экземпляр содержит поле для задания его *типа движения*, которое говорит системе о том, должно ли тело быть зафиксированным, а если нет, чем должны быть обусловлены его движения: игровым процессом (в Havok это называется телом на основе ключевых кадров) или системой физики (что в Havok называется динамическим телом). Если вы создали зафиксированное тело, тип его движений нельзя будет изменить. В противном случае можно менять его тип динамически во время выполнения. Эта возможность чрезвычайно полезна. Например, объектом в руке персонажа может управлять игровой процесс. Но, как только персонаж его уронит или выбросит, управление перейдет к системе симуляции динамики. В Havok для этого достаточно поменять тип движений в момент, когда персонаж выпускает объект из руки.

Тип движений также дает движку Havok некоторое представление о тензоре инерции динамического тела. Движения динамического типа делятся на подкатегории: динамические с инерцией сферы, динамические с инерцией параллелепипеда и т. д. Используя знание о типе движения тела, Havok может принять решение применить различные оптимизации на основе предположений о внутренней структуре тензора инерции.

13.5.2. Обновление симуляции

Физическую симуляцию, естественно, нужно периодически обновлять — обычно в каждом кадре. Для этого недостаточно просто заново пройти все этапы симуляции (численное интегрирование, обработку столкновений и применение ограничений). Мы также должны поддерживать связи между игровыми объектами и твердыми телами. Если игре требуется прикладывать какие-либо силы или импульсы к произвольному твердому телу, это тоже должно происходить в каждом кадре. Чтобы полностью обновить физическую симуляцию, необходимо выполнить следующие шаги.

- *Обновить твердые тела, движимые игровым процессом.* Преобразования всех твердых тел, движимых игровым процессом, обновляются таким образом, чтобы они совпадали со своими аналогами (игровыми объектами или суставами) из игрового мира.
- *Обновить фантомы.* Фантомная форма ведет себя как объект collidable, управляемый игровым процессом, у которого нет соответствующего твердого тела. Она используется для выполнения определенных запросов о столкновениях. Местоположение всех фантомов обновляется до этапа симуляции, чтобы при обнаружении столкновений они находились в нужных местах.
- *Обновить силы, применить импульсы и отрегулировать ограничения.* Обновляются все силы, которые прилагаются в игре, и любые импульсы, вызванные игровыми событиями, произошедшими в текущем кадре. При необходимости регулируются ограничения (например, мы можем проверить разрушаемый шарнир, чтобы определить, был ли он разрушен, если это действительно так, следует сообщить физическому движку о необходимости убрать ограничение).
- *Выполнить этап симуляции.* Мы уже говорили, что движки столкновений и физики следует периодически обновлять. Это действие включает в себя *численное интегрирование* уравнений движения для получения физического состояния всех тел в следующем кадре, выполнение алгоритма *обнаружения столкновений*, чтобы добавить контакты во все твердые тела в мире физики или удалить их оттуда, *обработку столкновений* и *применение ограничений*. В зависимости от SDK этапы обновления могут быть инкапсулированы внутрь единой атомарной функции `step()`, или же их можно будет выполнять по отдельности.
- *Обновить объекты, движимые физической симуляцией.* Из мира физики извлекаются преобразования всех объектов, движимых симуляцией, а затем эта информация используется для обновления преобразований соответствующих игровых объектов или суставов.
- *Найти фантомы.* После физической симуляции считываются контакты всех фантомных фигур, которые затем задействуются для принятия решений.
- *Запросить следы столкновений.* Выполняется отбрасывание лучей или фигур — либо синхронно, либо асинхронно. Когда результаты этих запросов станут доступны, различные системы движка используют их для принятия решений.

Эти задачи обычно реализуются в порядке, в котором они перечислены, исключение составляет процесс отбрасывания лучей и фигур, который теоретически можно выполнить на любом этапе игрового цикла. Несомненно, обновление тел, движимых игровым процессом, и приложение сил и импульсов должно происходить в первую очередь для того, чтобы полученные результаты были доступны в ходе симуляции. Аналогично игровые объекты, управляемые системой физики, должны обновляться после симуляции, чтобы мы могли использовать самые актуальные преобразования тел. Отрисовка обычно происходит в самом конце игрового цикла. Это позволяет вывести на экран согласованное представление виртуального мира в заданный момент.

Выбор подходящего момента для запросов о столкновениях

Чтобы обратиться к системе столкновений за свежей информацией, нужно выполнить запросы о столкновениях (отбрасывании лучей и фигур) после завершения физической симуляции в текущем кадре. Однако симуляция обычно реализуется ближе к завершению цикла, после того как игровая логика приняла бóльшую часть решений и определила новое местоположение всех физических тел, управляемых игровым процессом. Так когда же мы должны запрашивать данные о столкновениях?

На этот вопрос нет простого ответа. Возможны несколько вариантов, и в большинстве игр используются все или некоторые из них.

- *Принимаем решение в зависимости от состояния последнего кадра.* Во многих случаях корректные решения можно принять на основе информации о столкновениях из последнего кадра. Например, чтобы решить, должен ли игрок начать падать в текущем кадре, нам, наверное, нужно знать, стоял ли он на чем-то в предыдущем. В этом случае мы можем смело выполнять запросы о столкновениях до физической симуляции.
- *Допускаем отставание на один кадр.* Даже если действительно нужно знать, что происходит в *текущем кадре*, мы, вероятно, можем смириться с отставанием на один кадр при получении информации о столкновениях. Обычно это касается только тех случаев, когда объекты движутся не очень быстро. Например, мы можем переместить объект в одном кадре, а в следующем проверить, попал ли он в поле зрения игрока. Игрок может совсем не заметить отставания запроса. Если это действительно так, можно выполнить запросы о столкновениях до физической симуляции (получая информацию, актуальную для предыдущего кадра), а затем использовать полученные результаты в конце текущего кадра так, как будто они являются *приближенным* состоянием системы столкновений.
- *Выполняем запрос после физической симуляции.* Еще один подход состоит в том, чтобы выполнять некоторые запросы после этапа симуляции. Это имеет смысл в ситуациях, когда решения на основе результатов запроса можно принять ближе к концу цикла. Например, таким образом можно реализовать визуальный эффект, который зависит от информации о столкновениях.

Однопоточное обновление

Далее показан пример очень простого однопоточного игрового цикла:

```
F32 dt = 1.0f/30.0f;

for (;;) // главный игровой цикл
{
    g_hidManager->poll();

    g_gameObjectManager->preAnimationUpdate(dt);
    g_animationEngine->updateAnimations(dt);
    g_gameObjectManager->postAnimationUpdate(dt);

    g_physicsWorld->step(dt);
    g_animationEngine->updateRagDolls(dt);

    g_gameObjectManager->postPhysicsUpdate(dt);
    g_animationEngine->finalize();

    g_effectManager->update(dt);

    g_audioEngine->udate(dt);

     // и т. д.

    g_renderManager->render();

    dt = calcDeltaTime();
}
```

В данном примере игровые объекты обновляются в три этапа: один раз перед анимацией (во время которой они, к примеру, могут подготовить и поместить в очередь новые клипы), один раз после того, как анимационная система вычислила итоговые локальные позы и предварительную глобальную позу, но перед генерацией итоговой глобальной позы и палитры матриц, и еще раз после физической симуляции.

- Местоположение всех твердых тел, движимых игровым процессом, как правило, обновляется в `preAnimationUpdate()` или `postAnimationUpdate()`. Каждое преобразование этих тел должно совпадать с координатами игрового объекта, которому оно принадлежит, или сустава в скелете владельца.
- Местоположение каждого тела, движимого физической симуляцией, обычно считывается в функции `postPhysicsUpdate()` и используется для обновления координат игрового объекта или суставов его скелета.

Одним из важных аспектов является частота, с которой выполняется физическая симуляция. Большинство операций численного интегрирования, алгоритмов обнаружения столкновений и механизмов разрешения ограничений работают лучше всего, когда время между итерациями (Δt) остается постоянным. Обычно

обновление данных о физике/столкновениях имеет смысл выполнять строго один раз в 30 или 60 секунд и затем регулировать частоту кадров общего игрового цикла. Если частота кадров в игре опускается ниже установленного значения, лучше не пытаться подогнать под нее временные интервалы системы физики, а позволить симуляции визуально замедлиться.

13.5.3. Пример использования систем столкновений и физики в игре

Чтобы сделать обсуждение столкновений и физики более предметным, в общих чертах рассмотрим несколько распространенных примеров того, как эти системы обычно задействуются в настоящих играх.

Игровые объекты на основе простых твердых тел

Многие игры содержат простые физически симулируемые объекты, такие как оружие, камни, которые можно подбирать и бросать, пустые обоймы, мебель, предметы на полках, по которым можно стрелять, и т. д. Чтобы их реализовать, можно создать отдельный класс игровых объектов со ссылкой на твердое тело в мире физики (например, `hkpRigidBody`, если используется Havok). Мы также можем создать класс подключаемого компонента, который отвечает за физику и столкновения простого твердого тела, что позволит добавить эту возможность в игровой объект практически любого типа.

Простые физические объекты обычно меняют тип своих движений на этапе выполнения. Когда они находятся в руке персонажа, ими управляет игровой процесс, а когда их бросают, перемещения определяются физической симуляцией.

Следы от выстрелов

Независимо от того, как вы относитесь к виртуальному насилию, факт остается фактом: лазерные пушки и обычное оружие в том или ином виде являются важной частью большинства игр. Давайте посмотрим, как они обычно реализуются.

Иногда для реализации снарядов/пуль используется рейкастинг. В кадре, в котором происходит выстрел, мы отбрасываем луч, определяем, какой объект был затронут, и немедленно изображаем попадание.

К сожалению, подход с рейкастингом не учитывает время полета снаряда и его слегка нисходящую траекторию, обусловленную силой тяжести. Если в игре эти детали имеют значение, снаряды можно смоделировать с помощью настоящих твердых тел, которые перемещаются по миру физики/столкновений с ограниченной скоростью. Это особенно актуально для относительно медленных объектов, таких как метательные предметы или ракеты. Мы в Naughty Dog применяли этот метод для бросания кирпичей в *The Last of Us*.

При реализации лазерных лучей и снарядов необходимо учитывать множество аспектов. Некоторые из них описываются далее.

Рейкастинг для пуль. При использовании рейкастинга для проверки попадания пули возникает вопрос: откуда исходит луч — из фокусной точки камеры или из дула ружья в руках персонажа игрока? Эта проблема особенно остро стоит в шутерах *от третьего лица*, в которых луч из оружия игрока не совпадает с лучом, который выходит из фокусной точки камеры и проходит через прицел в центре экрана. Это может приводить к ситуациям, когда прицел визуально сфокусирован на цели, но поле зрения персонажа (который показан на экране в третьем лице) явно блокируется каким-то препятствием, что делает невозможным попадание в цель. Обычно, чтобы игрок чувствовал, что он стреляет туда, куда целится, и чтобы на экране все это выглядело правдоподобно, применяются различные трюки.

Несоответствия между столкновением и видимой геометрией. Несоответствия между геометрией столкновений и видимой геометрией могут приводить к ситуациям, когда игрок видит цель через небольшую трещину или выглядывая за край какого-то объекта, но с точки зрения геометрии столкновения это часть твердого тела, поэтому попадание невозможно (обычно эта проблема касается только персонажа игрока). В качестве решения, чтобы определить, попал ли луч в цель, вместо запроса о столкновении можно использовать запрос к движку отрисовки. Например, во время одного из этапов отрисовки можно сгенерировать текстуру, в которой каждый пиксел хранит уникальный идентификатор своего игрового объекта. Мы можем обратиться к этой текстуре, чтобы определить, занимает ли этот пиксел (или пикселы) враг или какая-то другая подходящая цель.

Прицеливание в динамических условиях. Если снаряду требуется какое-то время, чтобы достичь цели, персонажам, управляемым ИИ, возможно, придется стрелять с опережением.

Эффекты попадания. При попадании пули в цель, вероятно, следует воспроизвести какой-то звук или эффект частиц, наложить текстуру с повреждением или выполнить другие действия.

В движке Unreal для этого применяется система *физических материалов*. Видимой геометрии можно назначить не только визуальные, но и физические материалы. В первом случае описывается то, как поверхность выглядит, а во втором — то, как она реагирует на физические взаимодействия, включая звуки попаданий, эффекты частиц, иллюстрирующие контакт с пулей, следы от пуль и т. д. (подробнее об этом говорится на странице udn.epicgames.com/Three/PhysicalMaterialSystem.html).

В Naughty Dog мы используем очень похожую систему: геометрию столкновений можно пометить с помощью *атрибутов полигонов* (polygon attributes, PAT), которые определяют физическое поведение, такое как звуки шагов. Но с пулевыми попаданиями работают по-особому, поскольку они должны взаимодействовать непосредственно с видимыми объектами, а не с грубой геометрией столкновений. Поэтому видимым материалам можно при желании назначить *эффект пули*, который определяет, как будет выглядеть попадание, какой звук будет слышен и какие текстуры будут наложены при попадании любых снарядов, способных поразить эту поверхность.

Гранаты

Гранаты в играх иногда реализуются в виде объектов, полностью управляемых физической симуляцией. Но это приводит к существенной потере контроля. Это можно частично нивелировать, применяя к гранате разные искусственные силы или импульсы. Например, при первом отскоке гранаты можно задействовать сильное воздушное сопротивление, чтобы она не слишком далеко отлетела от цели.

На самом деле доходит до того, что в некоторых играх движением гранат управляют полностью вручную. Дугообразную траекторию ее полета можно вычислить заранее, последовательно отбрасывая лучи, это позволяет определить, какую цель она поразит, если ее бросить. Траекторию можно даже показать игроку с помощью какого-то элемента интерфейса. В случае броска игра перемещает гранату по дуге и впоследствии может контролировать ее отскок, чтобы он не был слишком сильным, но при этом выглядел естественно.

Взрывы

Взрывы в играх обычно состоят из нескольких элементов: визуального эффекта вроде огненного шара и дыма, звуковых эффектов для имитации грохота взрыва и его воздействия на объекты игрового мира, а также растущего радиуса поверхности взрыва, затрагивающего все на своем пути.

Когда объект находится в радиусе взрыва, его здоровье обычно уменьшается, также требуется придать ему какое-то движение, чтобы сымитировать эффект ударной волны. Это можно сделать с помощью анимации (так, например, лучше всего реализовать реакцию персонажа на взрыв) или посредством одной лишь симуляции динамики. Во втором случае взрыв должен применять импульсы к любому подходящему объекту в своем радиусе. Направление этих импульсов довольно просто вычислить: они, как правило, радиальные, поэтому нужно нормализовать вектор, проведенный из эпицентра взрыва к центру затронутого объекта, и затем масштабировать его в соответствии с магнитудой взрыва (возможно, с постепенным ослабеванием по мере удаления от эпицентра).

Взрывы могут взаимодействовать и с другими системами движка. Например, мы можем применить силу к системе анимации листвы, чтобы трава, кусты и деревья моментально наклонились от ударной волны.

Разрушаемые объекты

Разрушаемые объекты — неотъемлемый элемент многих игр. Им присущи довольно специфичные свойства: поначалу, в неповрежденном состоянии, они должны выглядеть цельными, но в то же время должны быть способны распадаться на много отдельных частей. Иногда эти части откалываются одна за другой, чтобы объект можно было постепенно обкромсать, но в некоторых случаях для полного разрушения достаточно одного катастрофического взрыва.

Системы симуляции деформируемых тел, такие как DMM, могут сделать процесс разрушения естественным. Но разрушаемые объекты можно реализовать и с помощью динамики твердых тел. Для этого модель объекта обычно разделяется на несколько частей, которые могут отломиться, и каждой из них назначается отдельное твердое тело. Этот подход, к примеру, применяется в Havok Destruction.

Чтобы оптимизировать производительность и/или улучшить качество графики, можно использовать специальные неповрежденные версии визуальных и физических объектов, которые являются цельными и неделимыми. Непосредственно перед процессом разрушения их можно заменить поврежденными версиями. Но в некоторых случаях лучше сделать так, чтобы объект изначально состоял из отдельных частей, — например, когда речь идет о груде кирпичей или горе кастрюль.

Чтобы смоделировать составной объект, мы можем просто собрать вместе кучу твердых тел и отдать их поведение на откуп системе физики. Это может сработать, если задействовать высококачественный физический движок (хотя добиться желаемых результатов иногда непросто). Но если мы хотим получить качество, как в голливудских фильмах, простого набора твердых тел будет недостаточно.

Например, мы можем захотеть определить структуру объекта. Некоторые его куски могут быть *неразрушаемыми*, например основание стены или шасси автомобиля. Некоторые могут быть *неструктурными*: при попадании в них они не просто отваливаются, но и применяют силы к другим частям, которые на них опираются. Некоторые куски могут быть *взрывчатыми*: при попадании они создают дополнительные взрывы или распространяют повреждение по всей структуре. Некоторые части объекта могут служить подходящими точками укрытия только для определенных персонажей. Из этого следует, что система разрушаемых объектов может как-то взаимодействовать с системой укрытий.

Разрушаемые объекты могут иметь такое свойство, как здоровье. Повреждения способны накапливаться до тех пор, пока объект в итоге не разрушится; каждая отдельная часть может обладать собственным здоровьем, и, чтобы она отвалилась, по ней нужно попасть несколько раз. Мы также можем задействовать ограничения, чтобы разрушенные куски свисали с объекта, не отделяясь от него полностью.

Возможно, структурам нужно какое-то время, чтобы полностью обвалиться. Например, если на одном конце моста произошел взрыв, обрушение должно происходить постепенно, пока не распространится по всей длине, благодаря чему мост будет казаться массивным. Это еще одна возможность, которую нельзя просто так реализовать с помощью системы физики, иначе в острове симуляции одновременно активизируются все твердые тела. В такого рода эффектах движениями может управлять игровой процесс, но это требует тщательного планирования.

Механика персонажа

В таких играх, как боулинг, пинбол или *Marble Madness*, роль главного персонажа играет шар, который катается по воображаемому игровому миру. Можно было бы смоделировать такой шар в виде свободно передвигающегося твердого тела

в физической симуляции и управлять его движениями в ходе игрового процесса за счет применения к нему сил и импульсов.

Но в играх с настоящими персонажами такой подход обычно нежелателен. Передвижения человекообразных или животных персонажей, как правило, намного сложнее контролировать с помощью сил и импульсов. Вместо этого персонажи обычно моделируются как набор твердых тел в форме капсул, движения которых обусловлены игровым процессом, при этом каждое из них привязано к суставу анимированного скелета. Такие тела в основном используются для обнаружения пулевых попаданий или генерации дополнительных эффектов (например, когда рука персонажа сталкивает какой-то объект со стола). Поскольку этими телами управляет игровой процесс, они не могут избежать пересечения с неподвижными объектами мира физики, поэтому ответственность за то, чтобы движения персонажа выглядели правдоподобно, ложится на аниматора.

Для перемещения персонажа по виртуальному миру большинство игр отбрасывают сферы или капсулы, чтобы проверить выбранное направление движения. Столкновения обрабатываются вручную. Это позволяет делать разные крутые вещи:

- вызывать скольжение персонажа вдоль стены, если он наткнулся на нее под острым углом;
- позволять персонажу подпрыгивать над низкими бордюрами, чтобы он в них не застревал;
- не давать персонажу переходить в состояние падения, когда он сходит с низкого бордюра;
- не давать персонажу подниматься по слишком крутым склонам (в большинстве игр есть пороговое значение угла, после превышения которого персонаж не может продолжать восхождение и соскальзывает вниз);
- подстраивать анимацию под столкновения.

Чтобы понять последний пункт, представьте себе персонажа, который вбегает в стену под прямым углом: мы можем позволить ему бесконечно топтаться на месте или же замедлить его анимацию. Есть и более элегантные решения — например, воспроизвести анимацию, в которой персонаж прикладывает руку к стене и стоит без дела, пока не изменится направление движения.

Havok предоставляет систему управления персонажем, которая берет на себя многие из этих нюансов (рис. 13.36). Персонаж моделируется в виде фантома в форме капсулы, движущегося в каждом кадре в поиске нового положения. Он обладает коллекцией контактных плоскостей, очищенных от всего лишнего, чтобы избежать дрожания при столкновении. Эту коллекцию можно анализировать в каждом кадре, чтобы определить, как лучше всего переместить персонажа, отрегулировать анимацию и т. д.

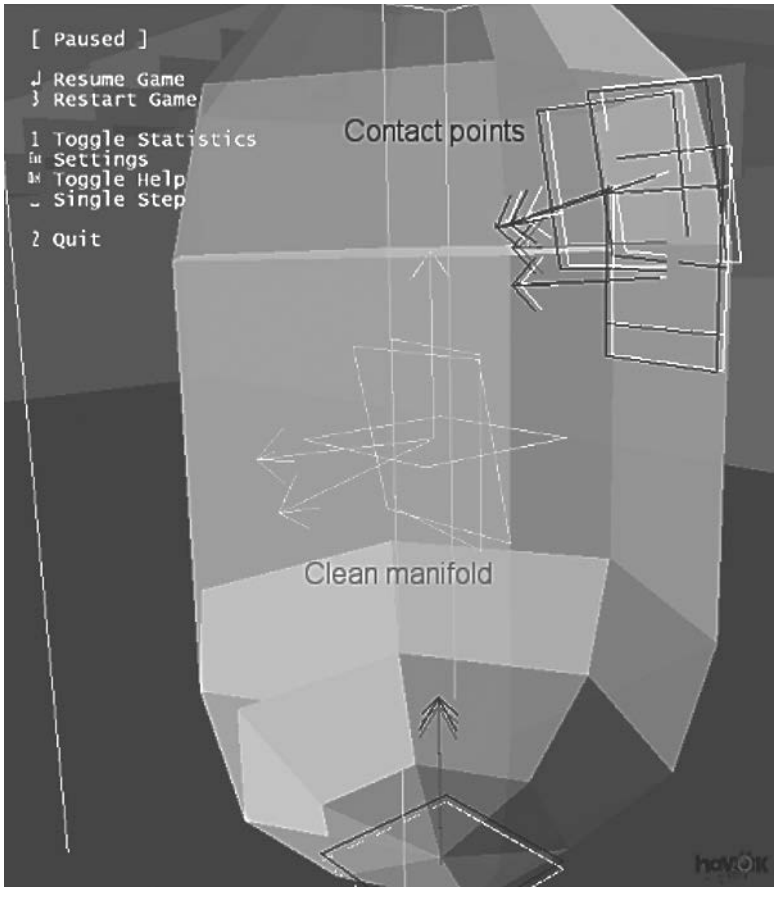


Рис. 13.36. В Havok персонажем управляет специальная система, которая моделирует его в виде фантома в форме капсулы. Этот фантом хранит набор контактных плоскостей без лишних деталей, которые движок может использовать для принятия решений о движении

Столкновения камеры с игровым миром

Во многих играх камера следует за персонажем игрока или автомобилем, и часто ее можно вращать или частично контролировать. При этом важно, чтобы она никогда не пересекала геометрию сцены, так как это может разрушить иллюзию реализма. Поэтому часто система управления камерой является еще одним важным клиентом движка столкновений.

Основная идея большинства механизмов, обрабатывающих столкновения камеры, состоит в том, чтобы окружить виртуальную камеру одной или несколькими фантомными сферами или слоем отбрасывания сферических фигур, который срабатывает, когда камера близка к столкновению. В качестве реакции мы можем

отрегулировать положение и/или направление камеры, чтобы избежать потенциального столкновения еще до прохождения сквозь соответствующий объект.

Это звучит довольно просто, но в действительности мы имеем дело с невероятно сложной проблемой, которую придется долго решать методом проб и ошибок. Чтобы вы понимали, скольких усилий это может потребовать, скажу лишь, что во многих игровых студиях системой управления камерой на протяжении всего проекта занимается отдельный инженер. У нас нет возможности подробно рассмотреть обнаружение и обработку столкновений камеры, но следующий список должен дать вам общее представление о самых важных аспектах, на которые следует обращать внимание.

- Чтобы избежать столкновений, можно увеличивать масштаб камеры, это хорошо работает в самых разных ситуациях. В игре от третьего лица масштаб без особых проблем можно увеличивать, пока не будет достигнуто представление от первого лица (главное, чтобы в процессе камера не пересекла голову персонажа).
- Горизонтальный угол камеры, как правило, не следует изменять слишком сильно в ответ на столкновения, так как это обычно нарушает работу элементов управления игроком, которые привязаны к камере. Но небольшая коррекция по горизонтали может дать хороший результат — это зависит от того, чем должен заниматься игрок в этот момент. Если он целится, а вы сместите прицел, чтобы не дать камере с чем-то столкнуться, это может его разозлить. Но если он просто перемещается по виртуальному миру, изменение в направлении камеры может выглядеть совершенно естественно. В связи с этим коррекцию горизонтального угла камеры имеет смысл разрешать только в случае, когда главный персонаж не находится в гуще битвы.
- Вертикальный угол камеры можно до некоторой степени откорректировать, но важно не переборщить, иначе игрок потеряет ощущение горизонта и направит камеру вниз, на макушку персонажа!
- Некоторые игры позволяют двигать камеру по дуге, лежащей в вертикальной плоскости, которая может быть описана в виде сплайна. Это позволяет интуитивно управлять масштабом и вертикальным углом камеры с помощью одного элемента HID, такого как вертикальное отклонение левого аналогового стика (именно так работает камера в движке Naughty Dog). Чтобы избежать столкновения с объектами игрового мира, камеру можно переместить по той же дуге, которая при этом может быть приплюснута по горизонтали. Помимо этого, возможен целый ряд других подходов.
- Важно учитывать не только то, что находится сзади и возле камеры, но и то, что перед ней. Например, что должно произойти, когда между камерой и персонажем игрока появится колонна или другой персонаж? В некоторых играх объекты, загораживающие обзор игрока, становятся полупрозрачными, иногда камера увеличивает масштаб или уходит в сторону, чтобы избежать столкновения. Это может создать неудобства для играющего! От того, как вы решите подобного рода проблемы, может зависеть, насколько качественной будет выглядеть игра.

Но даже если принять во внимание эти и многие другие проблемные ситуации, ощущения от управления камерой все равно могут быть не самыми лучшими! Всегда выделяйте достаточно времени на тонкую настройку системы столкновений камеры.

Интеграция системы тряпичных кукол

В подразделе 13.4.8 мы узнали, как с помощью специальных типов ограничений можно связать вместе коллекцию твердых тел, чтобы имитировать поведение бесчувственного (мертвого или потерявшего сознание) человека. А сейчас исследуем несколько проблем, которые возникают при интеграции физики тряпичной куклы в игру.

Как мы уже видели, общие движения персонажа, находящегося в сознании, обычно определяются за счет отбрасывания фигур или перемещения по виртуальному миру фантомных форм. Мелкие движения тела персонажа, как правило, основаны на анимации. Иногда к конечностям прикрепляют твердые тела, движимые игровым процессом, чтобы игрок мог нацеливать оружие или опрокидывать другие объекты в виртуальном мире.

Система тряпичных кукол начинает действовать, когда персонаж теряет сознание. Его конечности смоделированы в виде твердых тел в форме капсул, соединенных с помощью ограничений и прикрепленных к суставам анимированного скелета. Движения этих тел симулирует система физики, и суставы скелета соответствующим образом обновляются, что позволяет физическому движку управлять движениями персонажа.

В физике тряпичных кукол могут использоваться совсем не те твердые тела, которые были прикреплены к суставам персонажа, когда тот был жив. Подобное вызвано тем, что эти две модели столкновений имеют очень разные требования. Твердыми телами живого персонажа управляет игровой процесс, поэтому нас не заботит их потенциальное пересечение. На самом деле мы обычно хотим, чтобы они пересекались, — таким образом можно избавиться от любых отверстий, через которые может пройти выстрел вражеского персонажа. Но когда персонаж превращается в тряпичную куклу, взаимного пересечения твердых тел необходимо избегать, так как это может спровоцировать систему обработки столкновений на генерацию импульсов, которые будут разбрасывать конечности в стороны! В связи с этим живая и бесчувственная версии персонажа обычно имеют совершенно разные представления в системе столкновений/физики.

Есть еще одна проблема: как перейти из сознательного состояния в бессознательное? Простое LERP-слияние двух поз, движимых анимацией и физической симуляцией, обычно работает не очень хорошо, так как последняя очень быстро отклоняется от первой (слияние двух совершенно не связанных между собой поз, как правило, выглядит неестественно). В связи с этим во время перехода имеет смысл использовать механизированные ограничения (см. подраздел 13.4.8).

Персонажи, пребывающие в сознании (то есть когда их твердыми телами управляет игровой процесс), часто проходят сквозь фоновую геометрию. Это означает,

что в момент перехода персонажа в режим тряпичной куклы (движимой физической симуляцией) его твердые тела могут находиться внутри другого сплошного объекта. Это может создать огромные импульсы, которые сделают поведение тряпичной куклы в игре довольно бурным. Чтобы избежать этих проблем, лучше всего тщательно продумать анимацию смерти, чтобы конечности персонажа по возможности избегали столкновений. Кроме того, когда движениями управляет игровой процесс, важно следить за столкновениями с помощью фантомов или соответствующих функций обратного вызова, чтобы персонаж мог немедленно переключиться в режим тряпичной куклы, когда любая часть его тела коснется чего-то твердого.

Но даже если принять эти меры, тряпичные куклы все равно могут застрять в других объектах. В попытках сделать их поведение естественным чрезвычайно важной возможностью могут оказаться односторонние столкновения. Если конечность частично проникает в стену, она обычно выталкивается наружу, а не просто застревает. Однако одностороннее столкновение не решает всех проблем. Например, при быстром движении персонажа или не совсем правильно выполняемом переходе в бессознательный режим одно твердое тело тряпичной куклы может оказаться на противоположной стороне тонкой стены. В результате персонаж зависает в воздухе вместо того, чтобы как следует опуститься на землю.

Еще одним свойством тряпичных кукол, которое может пригодиться, является способность персонажа прийти в себя и подняться на ноги. Чтобы это реализовать, нужно найти подходящую анимацию вставания. Нам нужен клип, в нулевом кадре которого поза максимально приближена к позе тряпичной куклы в состоянии покоя (переход в это состояние в целом совершенно непредсказуем). Чтобы найти подходящий клип, можно сопоставить позы лишь нескольких ключевых суставов, например тазобедренных и плечевых. Еще можно использовать механизированные ограничения, чтобы вручную поместить тряпичную куклу в позу, которая позволит ей подняться из состояния покоя.

В качестве последнего замечания следует упомянуть, что задание ограничений тряпичной куклы может быть непростым занятием. В целом нам нужно, чтобы конечности двигались свободно, но в пределах возможного с точки зрения биомеханики. Это одна из причин, почему при создании тряпичных кукол применяют специализированные типы ограничений. Тем не менее вы должны понимать: тряпичные куклы будут выглядеть не слишком хорошо, если вы не приложите к работе с ними определенные усилия. Высококачественные физические движки вроде Havok предоставляют широкий выбор инструментов для создания контента, которые позволяют художникам настраивать ограничения в таких DCC-пакетах, как Maya, а затем проверять в реальном времени, как они могли бы выглядеть в игре.

В целом, заставить физику тряпичной куклы *работать* в игре не очень сложно, но чтобы она *хорошо* выглядела, вам придется как следует поработать! Многие вещи в игровой разработке создаются методом проб и ошибок и отнимают много времени, особенно если вы впервые имеете дело с тряпичными куклами.

13.6. Расширенные физические возможности

Симуляция динамики твердого тела с ограничениями может охватить невероятно широкий диапазон физических эффектов в игре. Но некоторые возможности такой системе явно недоступны. В последнее время проводятся исследования, которые призваны расширить область применения физических движков. Вот лишь несколько примеров.

- *Деформируемые тела*. С улучшением аппаратных возможностей и появлением более эффективных алгоритмов физические движки начинают предоставлять поддержку деформируемых тел. Отличным примером является движок DMM.
- *Ткань*. Ткань можно смоделировать в виде слоя материальных точек, соединенных жесткими пружинами. Этот материал чрезвычайно сложно имитировать, учитывая множество трудностей, связанных со столкновениями между тканью и другими объектами, численной стабильностью симуляции и т. д. Тем не менее сейчас многие игры и сторонние SDK для симуляции физики, такие как Havok, предоставляют мощные и хорошо ведущие себя модели ткани, которые можно применять в играх и других приложениях в режиме реального времени.
- *Волосы*. Волосы можно смоделировать в виде большого количества тонких физически симулируемых фиберов. Есть и более простой подход: на слои ткани можно наложить такие текстуры, чтобы они внешне напоминали волосы, при этом симуляцию ткани можно отрегулировать так, чтобы они правдоподобно шевелились на голове у персонажа. Именно так работают волосы Хлои в *Uncharted: The Lost Legacy*. Симуляция и отрисовка волос по-прежнему являются областью активных исследований, и качество этого аспекта игр, несомненно, будет улучшаться.
- *Симуляция поверхности воды и плавучесть*. В играх уже давно симулируется поверхность воды и плавучих предметов. Плавучесть можно реализовать с помощью системы особых случаев, а не как часть самого физического движка, ее можно смоделировать также посредством сил в рамках физической симуляции. Естественные колебания на поверхности воды часто являются лишь визуальным эффектом и никак не влияют на физические объекты в симуляции. С точки зрения физики поверхность воды часто выступает плоскостью, которая может двигаться при существенных колебаниях. Но некоторые разработчики игр и исследователи на этом не останавливаются и делают поверхность воды динамической, с поднимающимися волнами, реалистичной имитацией течения и т. д. Одним из хороших примеров является вода в симуляторе бога *From Dust*.
- *Симуляция общей динамики жидкостей*. Сейчас динамика жидкостей в основном реализуется посредством специализированных библиотек симуляции. Но в этой области проводятся активные исследования, и некоторые игры уже применяют симуляцию жидкостей для создания поразительных визуальных эффектов. Например, в цикле *LittleBigPlanet* с помощью двухмерной симуляции

жидкости обеспечивается имитация дыма и огня, а PhysX SDK предоставляет трехмерную *позиционную* симуляцию жидкости, которая показывает потрясающе реалистичные результаты.

- *Моделирование звука с учетом физики*. Когда физически симулируемые объекты сталкиваются, отскакивают, катятся и скользят, важно сгенерировать подходящие звуки, чтобы усилить реалистичность симуляции. В играх эти звуки можно создавать управляемым воспроизведением заранее подготовленных аудиоклипов. Но динамическое моделирование звука активно исследуется и постепенно становится достойной альтернативой этому способу.
- *GPGPU*. С ростом производительности графических адаптеров появилась новая тенденция: использовать их невероятно мощные параллельные вычисления не только для решения графических задач. Одной из очевидных областей применения вычислений на основе графических адаптеров общего назначения (general-purpose GPU, GPGPU) является симуляция столкновений и физики. Например, движок симуляции ткани в Naughty Dog был переписан так, чтобы он мог полностью работать на видеокарте PlayStation 4.

14 Звук

Если вам когда-нибудь приходилось смотреть фильм ужасов с выключенными динамиками, вы должны понимать, насколько важную роль играет звук в создании подходящей атмосферы (если нет, попробуйте — это откроет вам уши!). Неважно, фильм это или видеоигра, звук может сделать ваше времяпрепровождение захватывающим, эмоциональным и незабываемым или же скучным и вялым.

Современные игры погружают игрока в реалистичную (или полуреалистичную, но стилизованную) виртуальную среду. На графический движок возлагается задача точного и как можно более правдоподобного воспроизведения происходящего глазами игрового персонажа с соблюдением художественного стиля игры. Так же и звуковой движок обязан точно и правдоподобно воспроизводить то, что персонаж, находящийся в виртуальном мире, должен слышать своими ушами (соблюдая при этом художественный замысел и тональный стиль игры). Современные звукоинженеры называют аудиосистему *движком рендеринга звука* (audio rendering engine), чтобы подчеркнуть, что она имеет много общего с движком отрисовки.

В этой главе мы исследуем как теоретическую, так и практическую сторону создания аудио для высокобюджетных игр. Познакомимся с такой областью математики, как *теория обработки сигналов*, которая лежит в основе почти любого аспекта аудиотехнологий, включая цифровые запись и воспроизведение звука, фильтрацию, реверберацию и другие эффекты, воспроизводимые цифровым сигнальным процессором (digital signal processor, DSP). Мы будем обсуждать игровой звук с точки зрения разработчика ПО, знакомясь с широко применяемыми программными аудиоинтерфейсами и компонентами, из которых состоит типичный движок рендеринга звука, а также посмотрим, как аудиосистема связана с другими частями игрового движка. Вы узнаете, как были реализованы моделирование акустики окружающей среды и диалоги между персонажами в популярной игре *The Last of Us* от Naughty Dog. Так что пристегните ремни, держитесь крепче за руль и наслаждайтесь шумной поездкой!

14.1. Физика звука

Звук — это волна давления, которая распространяется в воздухе или какой-то другой среде. Звуковая волна образует череду областей сжатия и разрежения относительно среднего атмосферного *давления*, поэтому и *амплитуда* звуковой волны измеряется в соответствующих единицах системы СИ — паскалях (Па). Один паскаль — это сила величиной 1 ньютон (Н), применяемая к плоскости площадью 1 м^2 (1 Па = 1 Н/м^2 = 1 кг/(м·с^2)).

Мгновенное звуковое давление — это давление окружающей среды (мы будем считать его константой) плюс возмущение, вызванное звуковой волной в один определенный момент времени:

$$p_{\text{inst}} = p_{\text{atmos}} + p_{\text{sound}}.$$

Конечно, звук — это динамическое явление, так как давление звука изменяется со временем. Мы можем выразить мгновенное звуковое давление в виде функции от времени — $p_{\text{inst}}(t)$. В *теории обработки сигналов* (области математики, которая лежит в основе практически любого аспекта аудиотехнологий) такая функция называется *сигналом*. На рис. 14.1 проиллюстрирован сигнал типичной звуковой волны $p(t)$, колеблющейся в районе среднего атмосферного давления.

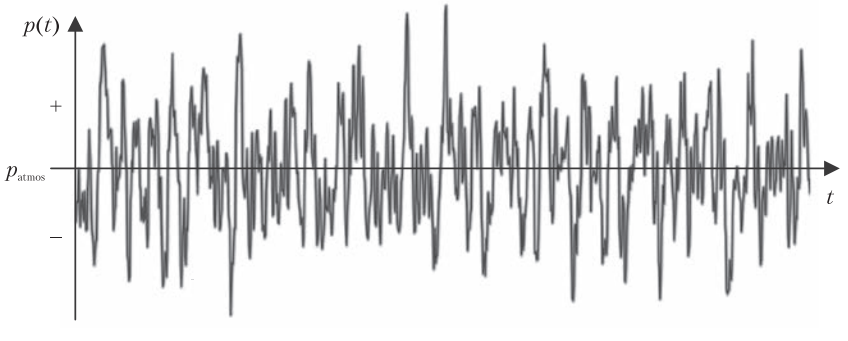


Рис. 14.1. Сигнал p(t) можно использовать для моделирования акустического давления звука на каком-то отрезке времени

14.1.1. Свойства звуковых волн

Когда музыкальный инструмент издает длинную стабильную ноту, сигнал звукового давления получается *периодическим*. Это означает, что волна состоит из *повторяющихся участков*, характерных для этого конкретного вида инструментов. *Период T* любого периодического сигнала равен минимальному количеству времени, которое проходит между смежными участками. Например, у синусоидальной звуковой волны период — это время между соседними пиками или впадинами. В СИ период обычно измеряется в секундах (с) (рис. 14.2).

Частота волны обратно пропорциональна ее периоду ($f = 1 / T$), измеряется в герцах (Гц) и означает количество колебаний в секунду. Строго говоря, колебание является безразмерной величиной, поэтому герц обратно пропорционален секунде (1 Гц = 1 / 1 с).

Многие ученые используют величину, известную как *угловая частота* и обозначаемую ω. Это просто частота колебаний, измеряемая не в *колебаниях* в секунду, а в *радианах* в секунду. Поскольку одно полное круговое вращение — это 2π радиан, $\omega = 2\pi f = 2\pi / T$. Угловая частота очень полезна при анализе синусоидальных волн, потому что круговое движение в двухмерном пространстве, спроецированное на одну из осей, дает синусоидальное движение.

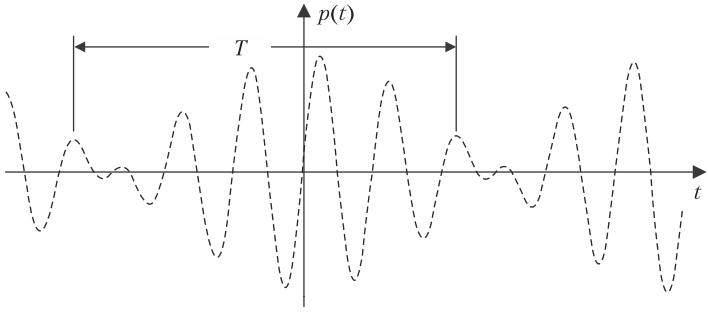


Рис. 14.2. Период T произвольного периодического сигнала — это минимальный временной интервал между повторяющимися участками звуковой волны

Величина *смещения* периодического сигнала, такого как синусоида, влево или вправо по оси времени называется *фазой*. Фаза — это относительное понятие. Например, sin(t) является разновидностью сигнала cos (t) с фазой, смещенной на $+\frac{\pi}{2}$ по оси T, то есть $\sin(t) = \cos\left(t - \frac{\pi}{2}\right)$. Аналогично, cos ($t$) — это sin ($t$) с фазой, смещенной на $-\frac{\pi}{2}$, то есть $\cos(t) = \sin\left(t + \frac{\pi}{2}\right)$ (рис. 14.3).

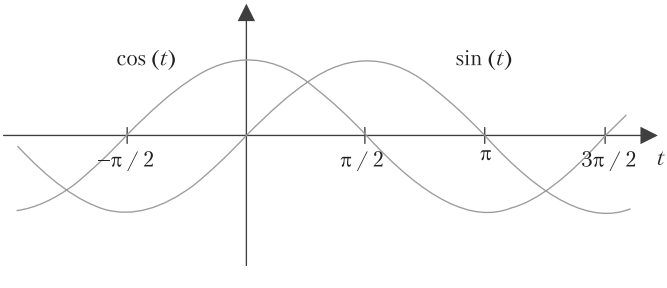


Рис. 14.3. Функции синуса и косинуса различаются лишь сдвигом фазы

Скорость v, с которой волна распространяется в той или иной среде, зависит от вещества, из которого состоит эта среда, и ее физических свойств, включая агрегатное состояние (твердое, газообразное, жидкое), температуру, давление и плотность.

В сухом воздухе с температурой 20 °C скорость звука равна примерно 343,2 м/с, или 1235,6 км/ч.

Длина λ синусоидальной волны измеряет расстояние между соседними пиками или впадинами. Она в определенной степени зависит от частоты, но, поскольку это *пространственная* величина, на нее влияет также скорость волны. Если точнее, $\lambda = v / f$, где v — скорость волны (м/с), а f — ее частота (Гц) или (1/с). Секунды в числителе и знаменателе сокращаются, и у нас остается длина волны, измеряемая в метрах.

14.1.2. Субъективная громкость звука и децибел

Чтобы мы могли судить о громкости звуков, которые слышим, наши уши непрерывно *усредняют* амплитуду звукового сигнала в течение короткого скользящего временного интервала. Эффект усреднения хорошо описывается величиной, известной как *действующее звуковое давление*, которая определяется как среднее квадратическое значение (root mean square, RMS) мгновенного звукового давления, измеренного на определенном отрезке времени.

Если последовательно выполнить n измерений звукового давления p_i через равные промежутки времени, получится RMS звукового давления

$$p_{\text{RMS}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} p_i^2}. \tag{14.1}$$

Однако наши уши оценивают давление непрерывно, а не в отдельные моменты времени. Если представить, что мгновенное давление звука измеряется непрерывно на отрезке времени от T_1 до T_2, сумма в уравнении (14.1) превращается в интеграл:

$$p_{\text{RMS}} = \sqrt{\frac{1}{T_2 - T_1}\int_{T_1}^{T_2} \big(p(t)\big)^2 \, dt}. \tag{14.2}$$

Но это еще не все. На самом деле субъективная громкость пропорциональна *интенсивности звука I*, которая, в свою очередь, пропорциональна *квадрату* RMS давления звука:

$$I \propto p_{\text{RMS}}^2.$$

Люди могут воспринимать звуковое давление в очень широком диапазоне: от шуршания кусочка бумаги, упавшего на землю, до ударной волны, когда самолет преодолевает звуковой барьер. Чтобы охватить такой широкий динамический диапазон, интенсивность звука обычно измеряют в *децибелах* (дБ). Децибел — это *логарифмическая* величина, которая выражает *отношение* между двумя значениями. Благодаря использованию логарифмической шкалы децибел позволяет свести широкий диапазон измерений к относительно узкому диапазону значений. На самом деле децибел равен 1/10 *бела* — единицы измерения, названной в честь Александра Грейама Белла.

Когда интенсивность звука измеряется в децибелах, ее называют *уровнем звукового давления* (УЗД) и обозначают символом L_p. Уровень звукового давления

определяется отношением интенсивности звука (то есть давления в квадрате) к начальной интенсивности p_{ref}, которая представляет нижний предел слышимости человеком. Поэтому получается:

$$L_p = 10\log_{10}\left(\frac{p_{\text{RMS}}^2}{p_{\text{ref}}^2}\right) = 20\log_{10}\left(\frac{p_{\text{RMS}}}{p_{\text{ref}}}\right).$$

Число 20 возникает из-за того, что квадрат, который выносят за логарифм, эквивалентен умножению на 2. Эталонное звуковое давление в воздухе традиционно записывают как p_{ref} = 20 μПа (СКЗ). Больше о звуковом давлении, физике звука и человеческом слухе можно почитать в [6].

Кстати, если вы с трудом припоминаете этот материал из школьных уроков, следующие тождества помогут освежить вашу память о логарифмах:

$$\begin{aligned}
&\log_b x = c, && \text{когда} && b^c = x \text{ (определение)};\\
&\log_b 1 = 0, && \text{потому что} && b^0 = 1;\\
&\log_b b = 1, && \text{потому что} && b^1 = b;\\
&\log_b (x \cdot y) = \log_b x + \log_b y, && \text{потому что} && b^c \cdot b^d = b^{c+d};\\
&\log_b (x / y) = \log_b x - \log_b y, && \text{потому что} && b^c / b^d = b^{c-d};\\
&\log_b x^d = d \log_b x, && \text{потому что} && \left(b^c\right)^d = b^{cd}.
\end{aligned} \qquad (14.3)$$

В уравнении (14.3) b, x и y являются положительными вещественными числами, где $b \neq 1$, c и d — это любые вещественные числа, где $c = \log_b x$ и $d = \log_b y$ (или, если записать иначе, $b^c = x$ и $b^d = y$).

Кривые равной громкости

Человеческое ухо по-разному реагирует на звуковые волны разной частоты. Мы наиболее чувствительны к частотам в диапазоне от 2 до 5 кГц. По мере отдаления частоты от этого диапазона в том или ином направлении для поддержания субъективной громкости на том же уровне необходима все более высокая интенсивность (то есть давление) звука.

На рис. 14.4 показан ряд кривых, которые соответствуют разным уровням субъективной громкости, — так называемые *кривые равной громкости*. Как можно видеть, чтобы обеспечить одну и ту же субъективную громкость, для низких и высоких частот необходимо применять большее давление, чем для частот среднего диапазона. Иными словами, если амплитуда звукового давления волны остается неизменной, более низкие и высокие частоты будут звучать для человеческого уха не так громко, как средние. Нижняя кривая равной громкости соответствует самому тихому акустическому сигналу, который мы только можем услышать, — это так называемый *абсолютный порог слышимости*. Самая верхняя кривая проходит через болевой порог человека, который равен примерно 120 дБ для слышимых звуков.

Больше о кривых равной громкости и кривых Флетчера — Мансона, на которых они основаны, можно почитать, прейдя по ссылке bit.ly/2HfCjCs.

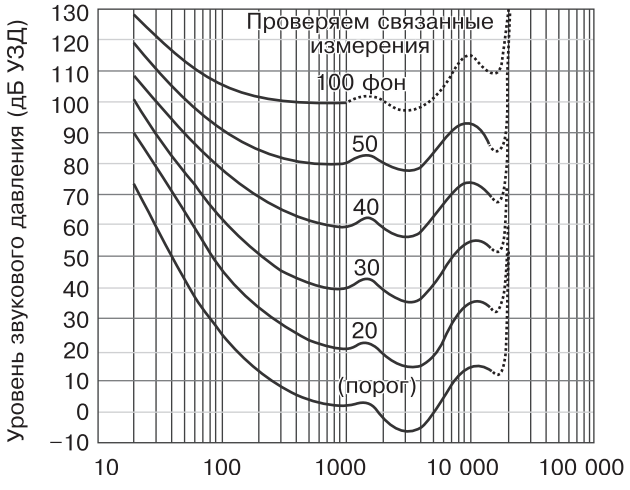


Рис. 14.4. Кривые равной громкости

Полоса слышимых частот

Обычный взрослый человек может слышать звуки с частотой от 20 до 20 000 Гц (20 кГц), хотя с возрастом верхний порог обычно понижается. Кривые равной громкости помогают объяснить, почему человеческое ухо может воспринимать звуки лишь в ограниченной полосе частот. При понижении или повышении частоты для поддержания одного и того же уровня субъективной громкости требуется все более высокое звуковое давление. Когда частота приближается к нижнему или верхнему порогу слышимости, кривые начинают выглядеть как вертикальные асимптоты. Это означает, что для достижения громкости, которую мы могли бы воспринять, требуется практически бесконечно высокое звуковое давление. То есть восприятие звука человеком за пределами полосы слышимых частот фактически сводится к нулю.

14.1.3. Распространение звуковой волны

Волна звукового давления, как и любая другая, распространяется в пространстве и может *поглощаться* поверхностями и *отражаться* от них, *огибать* углы (это называется *дифракцией*), проходить через узкие щели и *преломляться* при пересечении границы между разными средами. Звуковые волны не проявляют никакой *поляризации*[1], поскольку колебания звукового давления происходят в направлении распространения волны, что делает волну *продольной*, а не перпендикулярно ему, как в световых волнах, которые являются *поперечными*. В играх обычно

[1] Звуковые волны в твердых телах могут быть поперечными и, следовательно, демонстрировать поляризацию.

моделируются поглощение, отражение и иногда дифракция, то есть незначительное огибание углов, виртуальных звуковых волн, а вот эффекты преломления в целом игнорируются, так как человеку их сложно заметить.

Ослабевание по мере удаления

Если представить себе источник звука, который равномерно излучает сигнал в открытом пространстве с идеально неподвижным воздухом, интенсивность волны звукового давления будет ослабевать с увеличением пройденного расстояния в соответствии с законом $1/r^2$, тогда как само давление будет понижаться по принципу $1/r$:

$$p(r) \propto \frac{1}{r};\ I(r) \propto \frac{1}{r^2},$$

где r — это радиальное расстояние от слушателя или микрофона до источника звука, а давление и интенсивность выражены как функции от r.

Если быть более точным, уровень звукового давления для сферически излучаемой (ненаправленной) волны в открытом пространстве можно записать так:

$$L_p(r) = L_p(0) + 10\log_{10}\left(\frac{1}{4\pi r^2}\right) = L_p(0) - 10\log_{10}\left(4\pi r^2\right),$$

где $L_p(r)$ — УЗД у слушателя в виде функции от его радиального расстояния до источника звука, а $L_p(0)$ представляет неослабленную, или оригинальную, интенсивность звука в его источнике.

Источники звука не всегда являются ненаправленными. Например, когда большая плоская стена отражает звуковую волну, она выступает *направленным* источником звука: отраженные волны распространяются в одном направлении, а границы изменения давления проходят практически параллельно.

Мегафон проецирует звук в определенном направлении, но с *коническим* ослабеванием. Это означает, что интенсивность звуковых волн максимальна вдоль центральной линии проецируемого конуса, но ослабевает по мере увеличения угла между этой линией и слушателем.

Различные модели излучения звуковых сигналов проиллюстрированы на рис. 14.5.

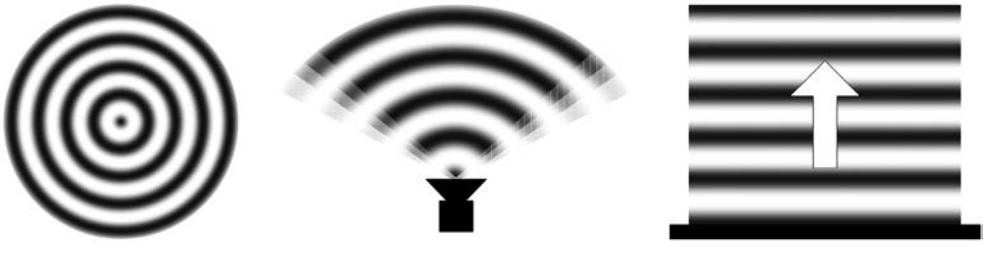


Рис. 14.5. Три источника звука с разными моделями излучения сигнала (в плоском виде для большей наглядности). *Слева направо:* ненаправленная, коническая и направленная

Атмосферное поглощение

По мере увеличения расстояния r до источника звука звуковое давление уменьшается пропорционально $1/r$ из-за рассеивания энергии по мере геометрического расширения формы волны. Это одинаково влияет на все звуки любой частоты. Интенсивность звука тоже падает при удалении от источника, но причиной этого служит атмосферное поглощение, которое действует неравномерно по всему спектру частот. В целом эффект поглощения усиливается с повышением частоты звука.

Это напоминает мне историю, которую я слышал в средней школе. Как-то ночью по тихой сельской улице шла женщина. Она услыхала незнакомые низкие звуки с длинными интервалами. Заинтригованная тем, откуда берутся такие странные звуки, она пошла туда, откуда они раздавались. С каждым шагом она слышала их все отчетливей, а интервалы между ними стали казаться более короткими. Спустя несколько минут звуки превратились в красивую мелодию. Женщина подошла к открытому окну и увидела в комнате человека, играющего на альте. Музыкант прекратил играть и поздоровался, а женщина спросила его, зачем он издавал случайные ноты несколько минут назад. Он ответил: «Я этого не делал — я играл эту мелодию с самого начала». Странные звуки, которые слышала женщина, можно объяснить тем, что из-за атмосферного поглощения низкочастотный звук слышен на большем расстоянии, чем высокочастотный. Больше о распространении звуковых волн можно узнать на странице www.sfu.ca/sonic-studio/handbook/Sound_Propagation.html.

На интенсивность звуковых волн, распространяющихся в той или иной среде, влияют и другие факторы. В целом ослабевание зависит от расстояния, частоты, температуры и влажности. По адресу sengpielaudio.com/calculator-air.htm находится онлайн-калькулятор, который позволит вам поэкспериментировать с этими параметрами.

Сдвиг фазы и интерференция

Когда в пространстве пересекаются несколько звуковых волн, их амплитуды суммируются — это называется *суперпозицией*. Представьте себе две периодические звуковые волны одинаковой частоты (простейший пример — две синусоиды). Если волны *совпадают по фазе* (то есть совпадают их пики и впадины), они будут *усиливать* друг друга и в результате получится волна с большей амплитудой. Точно так же, если волны *сдвинуты по фазе*, пики одной из них могут *компенсировать* впадины другой и наоборот и итоговая волна будет иметь меньшую или даже нулевую амплитуду.

Взаимодействие нескольких волн называется *интерференцией*. *Усиливающая интерференция* возникает, когда волны усиливают друг друга с увеличением амплитуды. *Ослабляющая интерференция* — это когда волны взаимно нивелируются, что в итоге дает меньшую амплитуду.

Важную роль в этом явлении играет частота волн. Волны с похожими частотами просто увеличивают или уменьшают результирующую амплитуду. Но если частоты

сильно различаются, мы можем наблюдать эффект *биения* — это когда расхождение частот приводит к периодическому изменению сдвига фазы, в результате чего амплитуда то увеличивается, то уменьшается.

Интерференция может возникнуть между двумя совершенно не связанными между собой звуковыми сигналами или когда один и тот же сигнал проходит от источника к слушателю разными путями. Во втором случае разница в пройденном расстоянии выражается в сдвиге фазы, который может создать либо усиливающую, либо ослабляющую интерференцию в зависимости от величины сдвига.

Гребенчатая фильтрация. Интерференция может создавать так называемый эффект *гребенчатой фильтрации*. Это происходит, кода звуковые волны отражаются от поверхностей таким образом, что некоторые частоты в них либо полностью подавляются, либо складываются и принимают пиковое значение. В результате получается амплитудно-частотная характеристика (АЧХ, см. подраздел 14.2.5) со множеством узких пиков и впадин, которые, если их вывести на графике, напоминают расческу или гребенку (отсюда и название). Этот эффект может серьезно влиять на воспроизведение и запись звука — как в виде нежелательного искажения, так и иногда в качестве полезного свойства. Существование гребенчатой фильтрации — одна из ключевых причин того, что акустическая обработка помещения обычно является более разумным капиталовложением, чем высококачественная звуковая аппаратура: если в помещении возникает эффект гребенчатой фильтрации, попытки добиться от оборудования плоской АЧХ будут пустой тратой времени. Итан Вайнер сделал отличную видеопрезентацию на эту тему: www.realtraps.com/video_comb.htm.

Реверберация и эхо

В любой среде со звукоотражающими поверхностями к слушателю от источника в целом доходят волны трех видов.

- *Прямые (сухие) волны*. Звуковые волны, доходящие до слушателя напрямую от источника, не сталкиваясь ни с какими препятствиями. Это так называемый *прямой*, или *сухой*, звук.
- *Ранние отражения (эхо)*. Звуковые волны, достигающие пользователя непрямым путем — после отражения от окружающих поверхностей (с частичным поглощением). Им требуется для этого больше времени, так как они преодолевают более длинный путь. В связи с этим между прибытием прямых и отраженных звуковых волн существует *задержка*. Первая группа отраженных звуковых волн, которую мы слышим, взаимодействовала лишь с одной или двумя поверхностями, поэтому она состоит из относительно чистых сигналов, которые воспринимаются человеком как отдельные копии звука — то, что мы называем *эхом*.
- *Поздние реверберации (хвост)*. Когда звуковые волны отражаются от окружающих поверхностей больше одного-двух раз, они накладываются друг на друга настолько, что наш мозг больше не в состоянии различить отдельные

эхо. Это называется *поздними реверберациями* или *диффузным хвостом*. Благодаря свойствам отражающих поверхностей амплитуды волн ослабляются по-разному. И поскольку отраженные звуковые волны задерживаются, возникающие сдвиги фазы создают интерференцию. В результате некоторые частоты затухают сильнее других. Когда говорят об *акустике* помещения, в основном имеют в виду влияние поздних реверберaций на субъективное качество или тембр звука.

Эхо и реверберация в сочетании с сухим звуком создают так называемый *«мокрый»* звук. На рис. 14.6 проиллюстрированы «мокрые» и сухие составляющие одного резкого хлопка.

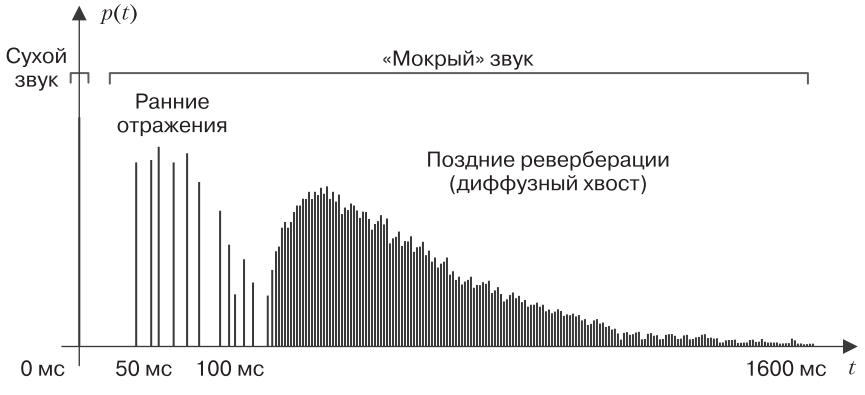


Рис. 14.6. Прямые звуковые волны, ранние отражения и поздние реверберации

Ранние отражения и поздние реверберации могут многое сказать о том, в какого рода пространстве мы находимся. *Предзадержка* — временной интервал между прибытием прямых и самых первых отраженных звуковых волн. По предзадержке наш мозг способен определить примерный размер комнаты или пространства, в котором мы слышим звук. *Задержка* — это время, которое уходит на полное затухание отраженных звуковых волн. С помощью этого свойства наш мозг может понять, какая часть звука была поглощена окружением, что дает косвенное представление о том, из чего состоит место, в котором мы находимся. Например, небольшая ванная комната, отделанная плиткой, будет создавать поздние реверберации с очень короткой предзадержкой (ввиду своего маленького размера), но с длинным периодом затухания, так как плитка эффективно отражает звуковые волны и у нее малое поглощение. Большое помещение с гранитными стенами, такое как Центральный вокзал Нью-Йорка, будет иметь намного более длинную предзадержку и намного большее эхо, но затухание будет мало отличаться от того, которое получается в ванной, облицованной плиткой.

Если повесить в этой ванной комнате занавески или отделать стены деревянными панелями вместо плитки, предзадержка останется прежней, а вот затухание

и другие факторы, такие как *плотность* (время между отдельными отражениями) и *диффузия* (скорость повышения плотности), изменятся. Это объясняет, почему мы можем судить о том, где находимся, даже с завязанными глазами, или как слепые люди учатся ориентироваться на улице с одной лишь тростью. Звук дает нам *много* информации о нашем окружении!

Термин «*реверберация*» используется для описания качества звука с точки зрения его «мокрых» составляющих. На ранних этапах развития индустрии аудиозаписи звукоинженеры почти не могли повлиять на реверберацию, полностью полагаясь на форму и отделку комнаты, в которой записывался звук. Позже появились простые методы создания искусственной реверберации, от использования динамика и микрофона в ванной комнате, как делал Билл Путман (основатель Universal Audio), до применения длинных металлических пластин или пружин, которые создавали задержку в звуковом сигнале. В наши дни для этого существуют цифровые устройства. Например, с помощью цифровых сигнальных процессоров (digital signal processor, DSP) и/или программного обеспечения не только воссоздается естественная реверберация в записываемых звуковых эффектах и музыке, но и записываются интересные искажения, которые обычно не встречаются в реальной жизни. Больше о цифровой обработке сигналов мы узнаем в разделе 14.2. О реверберации можно почитать на странице www.uaudio.com/blog/the-basics-of-reverb.

Безэховая камера — это комната, предназначенная для полного подавления отраженных звуковых волн. Для этого стены, пол и потолок отделывают толстыми поролоновыми панелями, которые поглощают практически все отраженные звуковые волны. В результате в ухо слушателя или в микрофон попадает только прямой (сухой) звук. Звучание в безэховых камерах имеет совершенно мертвый тембр. Они используются для записи чистых звуков, лишенных реверберации. Такие звуки идеально подходят для подачи на вход DSP-конвейера, давая звукоинженеру максимальный контроль за тембром звучания.

Звук в движении: эффект Доплера

Если вы когда-нибудь стояли на железнодорожном переезде во время приближения поезда, то могли наблюдать *эффект Доплера* в действии. Вначале звук поезда кажется высоким, но с сокращением расстояния он постепенно понижается. Звуковые волны распространяются в воздухе с более или менее постоянной скоростью, но источник звука (в данном случае поезд) тоже движется. Звуковые волны, движущиеся в том же направлении, что и поезд, сплющиваются, а те, что направлены в противоположную сторону, — растягиваются. И в том и в другом случае величина изменения пропорциональна разнице между скоростью звука в воздухе и скоростью поезда, проходящего через тот же воздух. Таким образом, частота сплющенных волн увеличивается, так как расстояние между пиками и впадинами звуковой волны фактически уменьшается, что делает звук более высоким. И наоборот, частота растянутых волн понижается, что дает более низкий звук. Этот эффект был назван в честь австрийского физика Кристиана Доплера, который обнаружил его в 1842 году.

Эффект Доплера возникает, и когда движется не источник звука, а слушатель. В целом величина сдвига зависит от *вектора относительной скорости* между слушателем и источником. Сдвиг Доплера приводит к изменению частоты и может быть вычислен следующим образом:

$$f' = \left(\frac{c + v_l}{c + v_s} \right) f,$$

где f — исходная частота; f' — частота после сдвига Доплера, наблюдаемая слушателем; c — скорость звука в воздухе; v_l и v_s — скорости слушателя и источника звука соответственно. Если скорость источника очень мала по сравнению со скоростью звука, это отношение можно выразить приближенно:

$$f' = \left(\frac{1 + (v_l - v_s)}{c} \right) f = \left(\frac{1 + \Delta v}{c} \right) f.$$

Данное выражение делает очевидным вектор относительной скорости Δv. Наглядное представление эффекта Доплера проиллюстрировано в следующей GIF-анимации: en.wikipedia.org/wiki/File:Dopplereffectsourcemovingrightatmach0.7.gif.

14.1.4. Восприятие положения в пространстве

Слуховая сенсорная система человека эволюционировала таким образом, чтобы мы имели достаточно точное представление о том, откуда исходят окружающие нас звуки. На восприятие положения источника звука влияет целый ряд факторов.

- *Затухание с увеличением расстояния* дает нам примерное понимание того, насколько удален источник звука. Но для этого мы должны иметь какое-то представление о его громкости на близком расстоянии, чтобы было с чем сравнивать.
- В результате *атмосферного поглощения* высокие частоты исчезают по мере удаления источника звука от слушателя. Это может сыграть важную роль в восприятии разницы между, скажем, человеком, который разговаривает нормальным голосом, но находится далеко, и человеком, находящимся поблизости, но разговаривающим тихо.
- Наличие *двух ушей*, по одному слева и справа, дает нам хорошую позиционную информацию. Звук, который приходит справа, в правом ухе будет звучать громче, чем в левом. Также появляется *интерауральный интервал* длиной примерно 1 мс, так как звук достигнет одного уха чуть раньше, чем другого. И наконец, сама голова препятствует распространению звуков, поэтому ухо, расположенное с противоположной от источника звука стороны головы, получит немного приглушенную его версию по сравнению с ближним ухом. В научной литературе это называют *интерауральной разностью интенсивности*.
- *Форма уха* тоже имеет значение. Наши уши слегка наклонены вперед, поэтому звуки, поступающие сзади, получаются совсем чуть-чуть приглушенными по сравнению с теми, которые мы слышим впереди.

- *Передаточная функция головы* — это математическая модель воздействия, которое оказывают складки наших ушей *(ушные раковины)* на звуки, поступающие с разных направлений.

14.2. Математика звука

Обработка сигналов и теория систем — это области математики, которые лежат в основе практически любой аудиотехнологии. Они активно используются и в широком спектре других технологических и инженерных направлений, включая обработку изображений и машинное зрение, аэронавтику, электронику, динамику жидкостей и многие другие. В этом разделе мы пройдемся по ключевым концепциям, связанным с сигналами и теорией систем. Это станет хорошим подспорьем при изучении более продвинутых аспектов игрового звука позже в этой главе (к тому же это важная область математической теории, которая может пригодиться любому разработчику игр, — два зайца одним выстрелом!). Глубокие знания этой темы можно получить, прочитав [41].

14.2.1. Сигналы

Сигнал — это любая функция с одной или несколькими независимыми переменными, которая, как правило, описывает поведение некоего физического явления. В разделе 14.1 мы использовали сигнал $p(t)$ для представления давления звуковой волны сжатия по времени. Конечно, это далеко не единственный вид сигналов. С помощью сигнала $v(t)$ можно изобразить вольтаж, генерируемый микрофоном на каком-то отрезке времени, $w(t)$ может смоделировать колебание давления воды в трубах, а $f(t)$ — представить изменяющуюся численность популяции лисиц в какой-то экосистеме.

При изучении теории сигналов мы часто называем независимую переменную временем и обозначаем ее символом t, но на самом деле она может представлять какую-то другую величину, к тому же быть не единственной. Например, двухмерное изображение в оттенках серого можно представить в виде сигнала $i(x, y)$, где две независимые переменные — это ортогональные оси координат, а значение сигнала i — это интенсивность изображения в каждом пикселе. Цветное изображение можно было бы представить в виде трех сигналов: $r(x, y)$ для красного канала, $g(x, y)$ — для зеленого и $b(x, y)$ — для синего.

Непрерывная дискретность

Примеры с двухмерными изображениями, приведенные ранее, подчеркивают важное различие между двумя основными типами сигналов — *непрерывными* и *дискретными*.

Если независимая переменная является *вещественным числом* ($t \in \mathbb{R}$), сигнал называют *непрерывным*. В этой главе мы будем использовать символ t для обозна-

чения непрерывного времени, а функции обозначим с помощью круглых скобок, например $x(t)$, чтобы не забывать, что работаем с непрерывным сигналом.

Если независимая переменная является *целым числом* ($n \in \mathrm{I}$), сигнал называют *дискретным*. Станем применять n для обозначения дискретного времени, а функции записывать с помощью квадратных скобок, например $x[n]$, чтобы не забывать, что речь идет о сигналах дискретного времени. Следует отметить, что *значение* сигнала дискретного времени по-прежнему может быть вещественным числом ($[n] \in \mathbb{R}$), единственной характерной чертой такого сигнала является то, что *независимая* переменная должна быть целым числом ($n \in \mathrm{I}$).

Как мы видели на рис. 14.1, сигнал непрерывного времени можно изобразить в виде графика обычной функции с временем t по горизонтальной оси и со значением сигнала $p(t)$ — по вертикальной. Сигнал дискретного времени $x[n]$ можно представить похожим образом, хотя независимая переменная функции n может принимать лишь целочисленные значения (рис. 14.7). Представьте, что из сигнала непрерывного времени выбираются отдельные точки. Этот процесс называется *дискретизацией* или *оцифровкой* (или *аналого-цифровым преобразованием*), и он лежит в основе цифровой записи и воспроизведения звука. Подробнее о дискретизации — в подразделе 14.3.2.

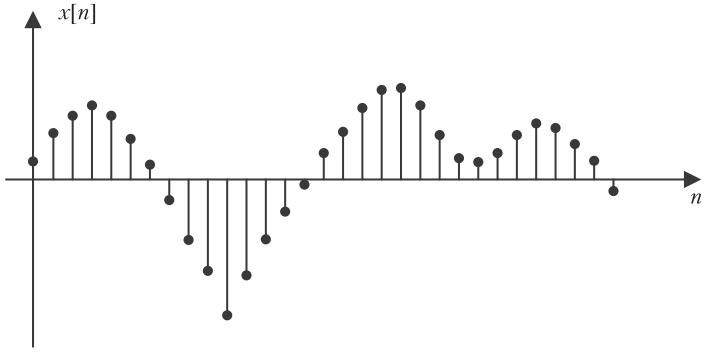


Рис. 14.7. Значение сигнала дискретного времени $x[n]$ определено только для целочисленных n

14.2.2. Преобразование сигналов

В дальнейшем нам потребуется понимание некоторых базовых методов манипуляции сигналами путем изменения их независимых переменных. Например, чтобы *отразить* сигнал относительно $t = 0$, мы просто меняем t на $-t$ в его уравнении. Чтобы *сдвинуть сигнал по времени вправо* (то есть в *положительном* направлении) на расстояние s, меняем t на $s - t$. Для сдвига по времени в *отрицательном направлении — влево* — нужно заменить t на $t + s$. Мы также можем растягивать и сжимать область сигнала, масштабируя независимую переменную. Эти простые преобразования проиллюстрированы на рис. 14.8.

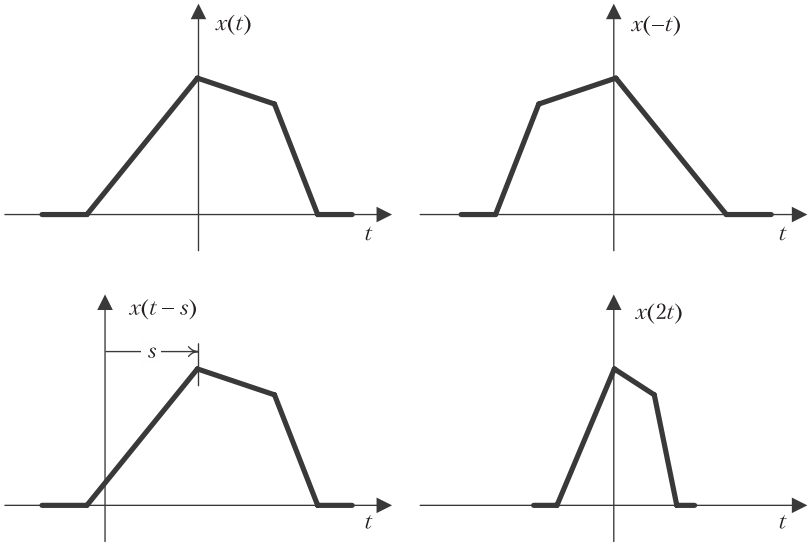


Рис. 14.8. Простые манипуляции с независимой переменной сигнала

14.2.3. Линейные стационарные системы

В контексте теории обработки сигналов под *системой* понимают любые устройство или процесс для преобразования *входного* сигнала в *выходной*. Это определение можно использовать для описания и анализа многих реальных систем, которые возникают при обработке звука, включая микрофоны, динамики, аналого-цифровые преобразователи, модули реверберации, эквалайзеры, фильтры и даже акустические свойства комнаты, а также манипуляции ими.

В качестве простого примера можно привести *усилитель*. Это система, которая увеличивает амплитуду входного сигнала с *коэффициентом передачи A*, то есть в A раз. Если взять сигнал $x(t)$, система усиления преобразует его в $y(t) = Ax(t)$.

Систему называют *стационарной*, если сдвиг по времени во входном сигнале вызывает идентичный сдвиг в выходном. Иными словами, поведение системы не меняется со временем.

Система является *линейной*, если обладает свойством *суперпозиции*. То есть если входной сигнал состоит из *взвешенной суммы* других сигналов, результатом будет взвешенная сумма отдельных сигналов, которые можно было бы получить, если бы каждый входной сигнал прошел через систему по отдельности.

Линейные стационарные системы (ЛСС) чрезвычайно полезны по двум причинам. Во-первых, их поведение является интуитивно понятным и относительно простым в использовании с математической точки зрения. Во-вторых, многие реальные физические системы в таких областях, как теория распространения звука, электроника, механика, динамика жидкостей и др., могут быть смоделированы

в виде ЛСС. В связи с этим при обсуждении аудиотехнологий ограничимся только системами этого вида.

Любую систему можно представить как черный ящик с входным и выходным сигналами (рис. 14.9).

Рис. 14.9. Система в виде черного ящика

Если использовать такое представление, простые системы можно легко объединять в сложные, например:

- соединить выход системы A со входом системы B и в результате получить составную систему, которая последовательно выполняет операции A и B (это называется *последовательным соединением*);
- объединить выход двух систем;
- использовать в качестве входа системы ее же выход. В таком случае получается так называемая *петля обратной связи*.

Примеры всех этих соединений показаны на рис. 14.10.

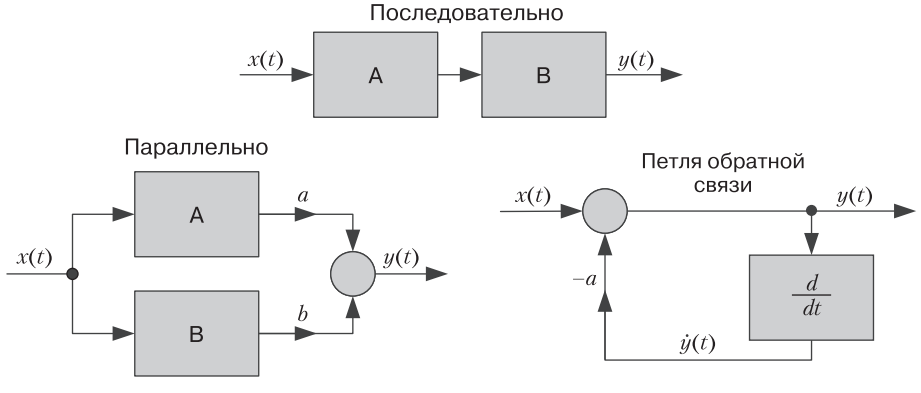


Рис. 14.10. Системы можно связывать между собой разными способами. При последовательном соединении y(t) = B(A(x(t))). При параллельном соединении y(t) = aA(x(t)) + bB(x(t)). В петле обратной связи y(t) = x(t) – aẏ(t)

Одно очень важное свойство всех систем ЛСС состоит в том, что их взаимные связи являются *порядконезависимыми*. Поэтому, если у нас есть последовательная связь между системами A и B, мы можем поменять эти системы местами и получить тот же результат.

14.2.4. Импульсная характеристика систем ЛСС

Это, конечно, хорошо, что мы можем рассуждать о системах, преобразующих входной сигнал в выходной, и с легкостью рисовать диаграммы связей между ними. Но как представить работу системы математически?

Как вы помните из подраздела 14.2.3, если сигнал на входе *линейной* системы состоит из линейной комбинации (взвешенной суммы) нескольких входных сигналов, то сигнал на выходе этой системы также будет линейной комбинацией (взвешенной суммой) нескольких выходных сигналов (если каждый входной сигнал пропустить через систему по отдельности). Таким образом, если мы сможем представить произвольный входной сигнал в виде взвешенной суммы элементарных сигналов, это должно позволить нам описать поведение системы как реакцию на эти элементарные сигналы.

Единичный импульс

Если мы хотим описать входной сигнал в виде линейной последовательности элементарных сигналов, возникает вопрос: какой элементарный сигнал следует использовать? По причинам, которые станут понятными чуть позже, остановим свой выбор на *единичном импульсе*. Этот сигнал принадлежит к семейству *функций сингулярности (особенности)*, которые всегда содержат как минимум один разрыв (сингулярность), — отсюда и название.

С точки зрения дискретного времени единичный импульс $\delta[n]$ выглядит крайне просто — это сигнал, значение которого равно нулю везде, кроме точки $n = 0$, где оно равно 1:

$$\delta[n] = \begin{cases} 1, & \text{если } n = 0, \\ 0 & \text{в остальных случаях.} \end{cases}$$

Единичный импульс в дискретном времени проиллюстрирован на рис. 14.11.

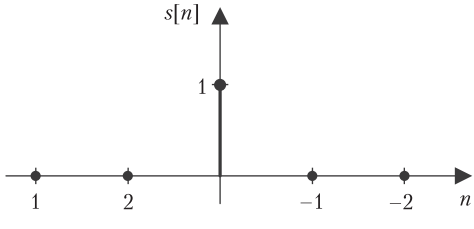


Рис. 14.11. Единичный импульс с точки зрения дискретного времени

С точки зрения непрерывного времени единичный импульс $\delta(t)$ имеет чуть более сложное определение. Это функция, значение которой равно нулю везде, кроме точки $t = 0$, в которой она бесконечна, при этом *площадь* под кривой равна 1.

Чтобы понять, как может выглядеть определение такой странной функции, представьте себе *прямоугольную* функцию $b(t)$, значение которой равно нулю везде, кроме интервала $[0, T)$, где оно равно $1 / T$. Площадь под кривой — это всего лишь площадь прямоугольника — ширина, умноженная на высоту, или $T(1 / T) = 1$. Теперь представьте предел $T \to 0$. В этом случае ширина прямоугольника стремится к нулю, а высота — к бесконечности, но площадь по-прежнему остается равной 1 (рис. 14.12).

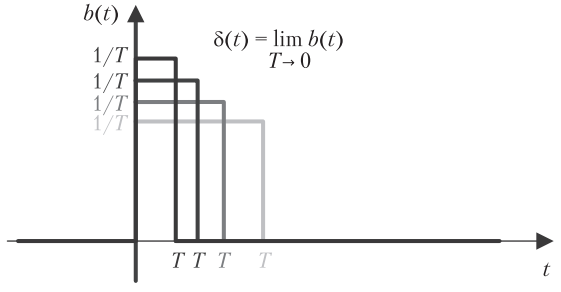


Рис. 14.12. Единичный импульс можно определить как предел прямоугольной функции b(t), ширина которой стремится к нулю

Функция единичного импульса обычно обозначается $\delta(t)$ (дельта-функция). Ее формальное определение может выглядеть так:

$$\delta(t) = \lim_{T \to 0} b(t),$$

где:

$$b(t) = \begin{cases} 1/T, \text{ если } t \geq 0 \text{ и } t < T, \\ 0 \text{ в остальных случаях.} \end{cases}$$

Как видно на рис. 14.13, единичный импульс обычно изображается в виде стрелки, высота которой соответствует площади под кривой (так как настоящая высота функции в точке $t = 0$ бесконечна).

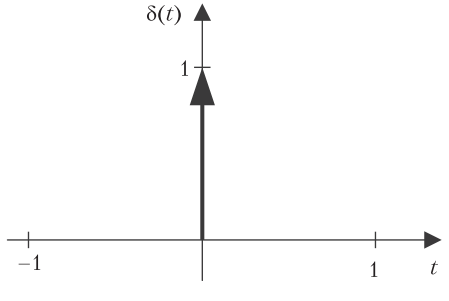


Рис. 14.13. Значение функции единичного импульса δ(t) равно нулю везде, кроме точки t = 0, где оно бесконечно. Это можно изобразить в виде стрелки, высота которой равна 1, подчеркивая тем самым, что площадь под кривой равна 1

Представление сигнала в виде последовательности импульсов

Итак, мы узнали, что такое сигнал единичного импульса. Теперь посмотрим, сможем ли мы описать произвольный сигнал $x[n]$ в виде линейной последовательности импульсов (забегая наперед, отвечаю: сможем).

Функция $\delta[n-k]$ является дискретным единичным импульсом, сдвинутым по времени, чье значение равно нулю везде, кроме времени $n = k$, где оно равно 1. Иными словами, единичный импульс $\delta[n-k]$ происходит в момент времени k. Рассмотрим импульс с конкретным значением k, скажем $k = 3$. Убедимся в том, что его высота совпадает со значением исходной функции в момент $k = 3$. Для этого масштабируем импульс по $x[3]$ и получим $x[3]\delta[n-3]$. Если проделать то же самое для всех возможных значений k, получится последовательность импульсов вида $x[k]\delta[n-k]$. Сложим все эти масштабированные, сдвинутые по времени функции вместе и получим еще одно представление исходного сигнала $x[n]$:

$$x[n] = \sum_{k=-\infty}^{+\infty} x[k]\delta[n-k]. \tag{14.4}$$

Я не стану приводить здесь строгое доказательство, но, наверное, не так уж сложно поверить в то, что выполнение тех же действий в условиях непрерывного времени даст практически такой же результат. Единственное отличие в том, что уравнение (14.4) для непрерывного времени превращается в интеграл. Представьте себе бесконечную последовательность единичных импульсов $\delta(t-\tau)$, сдвинутых по времени и происходящих в разные моменты τ. Мы можем записать произвольный сигнал $x(t)$ аналогично тому, как делали это в случае с дискретным временем:

$$x(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)\delta(t-\tau)d\tau. \tag{14.5}$$

Свертка

Уравнение (14.4) показывает, как можно представить сигнал $x[n]$ в виде *линейной последовательности* простых сдвинутых по времени сигналов $\delta[n-k]$. Представим, что мы пропускаем через систему только *один* из взвешенных входящих импульсов ($x[k]\delta[n-k]$), неважно, какой именно. Если выбрать $k = 0$, получится входной сигнал $x[0]\delta[n]$.

Мы будем использовать запись $x[n] \Longrightarrow y[n]$, чтобы показать, что входной сигнал $x[n]$ преобразуется системой ЛСС в выходной $y[n]$. Поэтому можно записать:

$$x[0]\delta[n] \Longrightarrow y[n].$$

Значение $x[0]$ — это просто константа. И поскольку мы имеем дело с линейной системой, то выходной сигнал $y[n]$ будет точно такой же константой, умноженной на реакцию системы на единичный импульс $\delta[n]$. Давайте опишем реакцию системы на элементарный единичный импульс с помощью сигнала $h[n]$:

$$\delta[n] \Longrightarrow h[n].$$

Сигнал $h[n]$ называется *импульсной характеристикой* системы. Поэтому можем записать реакцию системы на простой входной сигнал следующим образом:

$$x[0]\delta[n] \Longrightarrow x[0]h[n].$$

Концепция импульсной характеристики проиллюстрирована на рис. 14.14.

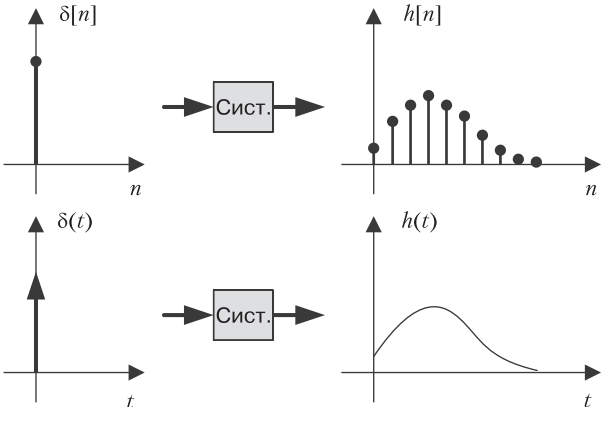


Рис. 14.14. Примеры импульсной характеристики системы с дискретным и непрерывным временем

Реакция системы ЛСС на единичный сдвинутый по времени импульс — сдвинутый по времени импульс ($\delta[n-k] \Rightarrow h[n-k]$). Поэтому для значений k, отличных от нуля, все будет работать точно так же, только теперь оба сигнала, входной и выходной, будут *сдвинуты по времени* на величину k:

$$x[k]\delta[n-k] \Rightarrow x[k]h[n-k].$$

Чтобы вычислить реакцию системы на всю последовательность сигналов $x[n]$, просто суммируем результаты для каждой отдельной составляющей, сдвинутой по времени:

$$\sum_{k=-\infty}^{+\infty} x[k]\delta[n-k] \Rightarrow \sum_{k=-\infty}^{+\infty} x[k]h[n-k].$$

Иными словами, сигнал на выходе системы можно записать так:

$$y[n] \Rightarrow \sum_{k=-\infty}^{+\infty} x[k]h[n-k]. \tag{14.6}$$

Это очень важное уравнение называется *сверточной суммой*. Для удобства введем математический оператор *, который обозначает операцию свертки:

$$x[n] * h[n] \Rightarrow \sum_{k=-\infty}^{+\infty} x[k]h[n-k]. \tag{14.7}$$

Уравнения (14.6) и (14.7) позволяют вычислить реакцию системы ЛСС $y[n]$ на произвольный входной сигнал $x[n]$ при наличии лишь *импульсной характеристики* системы $h[n]$. То есть сигнал импульсной характеристики $h[n]$ *полностью описывает систему* ЛСС.

Свертка с непрерывным временем. Ранее, чтобы не усложнять, мы работали с дискретным временем. В случае с непрерывным временем все происходит

примерно так же. Единственное отличие в том, что суммирование превращается в интеграл, поэтому в уравнения необходимо добавить дифференциал $d\tau$.

Если подать произвольный сигнал $x(t)$ на вход системы ЛСС с непрерывным временем, выходной сигнал можно записать следующим образом:

$$y(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau. \tag{14.8}$$

Как и прежде, используем оператор * для обозначения свертки:

$$x(t) * h(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau. \tag{14.9}$$

По аналогии со сверточной суммой интеграл в уравнениях (14.8) и (14.9) тоже называется *сверточным*.

Визуализация свертки

Давайте попробуем визуализировать операцию свертки в случае с непрерывным временем. Чтобы найти $y(t) = x(t) * h(t)$ для отдельного значения t, скажем $t = 4$, нужно выполнить следующие шаги (рис. 14.15).

1. Построить график $x(\tau)$, используя τ в качестве переменной времени, так как t не меняется (в данном примере $t = 4$).
2. Построить график $h(t - \tau)$. Это можно записать как $h(-\tau + t)$. Поскольку τ инвертируется, мы знаем, что импульсная характеристика отображается относительно $\tau = 0$. Также знаем, что сигнал был сдвинут *влево* на $t = 4$ единицы, так как мы прибавили t к независимой переменной.
3. Умножить два сигнала по всей оси T.
4. Проинтегрировать от $-\infty$ до $+\infty$ вдоль оси T, чтобы найти площадь под результирующей кривой. Это будет значение $y(t)$ в конкретный момент времени t (в нашем случае $t = 4$).

Помните: чтобы определить входной сигнал $y(t)$ целиком, эту процедуру необходимо повторить для всех возможных значений t.

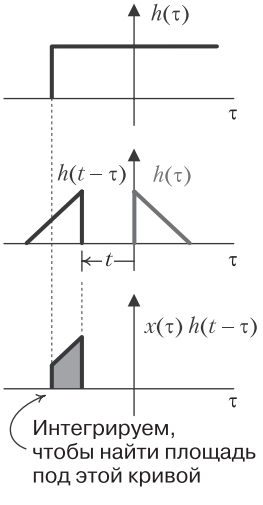


Рис. 14.15. Визуальное представление операции свертки с непрерывным временем

Некоторые свойства свертки

По своим свойствам операция свертки на удивление похожа на обычное умножение. Она является:

- *коммутативной* — $x(t) * h(t) = h(t) * x(t)$;
- *ассоциативной* — $x(t) * h_1(t) * h_2(t) = x(t) * h_1(t) * h_2(t)$;
- *дистрибутивной* — $x(t) * h_1(t) * h_2(t) = x(t) * h_1(t) + x(t) * h_2(t)$.

14.2.5. Частотная область и преобразование Фурье

Чтобы получить определение таких понятий, как импульсная характеристика и свертка, мы описали сигнал как взвешенную сумму единичных импульсов. А также представили сигнал в виде взвешенной суммы синусоид. Во втором случае мы, в сущности, разбиваем сигнал на его *частотные компоненты*. Это позволяет вывести еще одну невероятно мощную математическую концепцию — *преобразование Фурье*.

Синусоидальный сигнал

Синусоидальный сигнал возникает, когда круговое движение в двухмерном пространстве проецируется на одну ось. Аудиосигнал в виде синусоиды создает чистый тон на определенной частоте.

Самым элементарным синусоидальным сигналом является функция синуса (или косинуса). Сигнал $x(t) = \sin t$ равен 0 в точках $t = 0$, π и 2π, равен 1 в точке $t = \frac{\pi}{2}$ и -1 в точке $t = \frac{3\pi}{2}$.

Самое общее представление синусоидального сигнала с вещественными значениями выглядит так:

$$x(t) = A \cos(\omega_0 t + \phi). \tag{14.10}$$

Здесь A обозначает *амплитуду* синусоидальной волны (то есть максимальное и минимальное значения синусоидальной волны равны A и соответственно $-A$). ω_0 — это *угловая частота*, измеряемая в радианах в секунду (частота и угловая частота подробно обсуждались в подразделе 14.1.1); ϕ обозначает *сдвиг фазы* (тоже измеряемый в радианах), который смещает косинусоидальную волну влево или вправо по оси времени.

Когда $A = 1$, $\omega_0 = 1$ и $\phi = 0$, уравнение (14.10) сводится к $x(t) = \cos t$. Когда $\varphi = \frac{\pi}{2}$, это выражение превращается в $x(t) = \sin t$. Функции cos и sin представляют проекции кругового движения на горизонтальную и вертикальную оси соответственно.

Комплексный экспоненциальный сигнал

На самом деле косинусоидальная функция — не самый лучший способ представить сигнал в виде суммы синусоид. Выражения можно сделать куда более простыми и элегантными, если использовать *комплексные числа*. Чтобы понять, как это работает, необходимо рассмотреть математику комплексных чисел, в частности их умножение. Так что наберитесь терпения — как только мы закончим, все сразу же прояснится.

Краткое введение в комплексные числа. Вы, наверное, помните из школьных уроков математики, что комплексное число — это своего рода двухмерная величина,

состоящая из двух частей — вещественной и мнимой. Любое комплексное число можно записать как $c = a + jb$, где a и b — вещественные числа, а $j = \sqrt{-1}$ — *мнимое*. Вещественная часть c обозначается $a = \mathrm{Re}(c)$, а мнимая — $b = \mathrm{Im}(c)$.

Вещественное число можно представить в виде «вектора» $[a, b]$ в двухмерном пространстве, известном как *комплексная плоскость*. Однако важно помнить, что комплексные числа и векторы *не* являются взаимозаменяемыми — в математическом смысле их поведение существенно различается.

Абсолютная величина комплексного числа определяется как длина представления его двухмерного вектора на комплексной плоскости $|c| = \sqrt{a^2 + b^2}$. Угол между вектором и вещественной осью называется *аргументом*: $\arg c = \tan^{-1}(b / a)$ (иногда аргумент комплексного числа называют *фазой*; как мы увидим, термин «фаза» тесно связан со сдвигом фазы ϕ в уравнении (14.10)). Абсолютная величина и аргумент комплексного числа изображены на рис. 14.16.

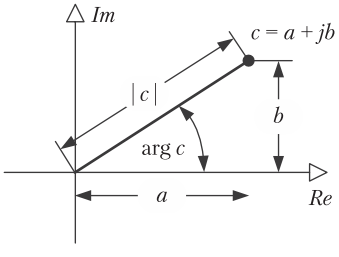


Рис. 14.16. Величина $|c| = \sqrt{a^2 + b^2}$ комплексного числа — это его длина в комплексной плоскости, а $\arg c = \tan^{-1}(b/a)$ — это угол между ним и осью Re

Умножение и вращение комплексных чисел. Мы не станем рассматривать все свойства комплексных чисел. Их углубленное обсуждение можно найти, перейдя по ссылке http://www.math.wisc.edu/~angenent/Free-Lecture-Notes/freecomplexnumbers.pdf. Но есть математическая операция, в которой мы заинтересованы, — комплексное *умножение*.

Комплексные числа умножаются алгебраически, без скалярных или векторных произведений:

$$c_1 c_2 = (a_1 + jb_1)(a_2 + jb_2) = (a_1 a_2) + j(a_1 b_2 + a_2 b_1) + j^2 b_1 b_2 =$$
$$= (a_1 a_2 - b_1 b_2) + j(a_1 b_2 + a_2 b_1). \qquad (14.11)$$

Если вычислить абсолютную величину и аргумент (угол) произведения $c_1 c_2$, окажется, что эта абсолютная величина равна *произведению* абсолютных величин исходных чисел, а аргумент равен *сумме* их аргументов:

$$|c_1 c_2| = |c_1||c_2|;$$
$$\arg(c_1 c_2) = \arg c_1 + \arg c_2. \qquad (14.12)$$

Тот факт, что умножение комплексных чисел приводит к *сложению* их углов (аргументов), означает, что комплексное умножение создает *вращение* в комплексной плоскости. Если абсолютная величина c_1 равна 1 ($|c_1| = 1$), модуль произведения будет равен абсолютной величине c_2 ($|c_1 c_2| = |c_2|$). В данном случае произведение представляет *чистое вращение* c_2 на угол, равный $\arg c_1$ (рис. 14.17). Если $|c_1| \neq 1$,

модуль произведения будет *масштабирован* на $|c_1|$, и в результате c_2 станет демонстрировать *спиральное движение* на комплексной плоскости.

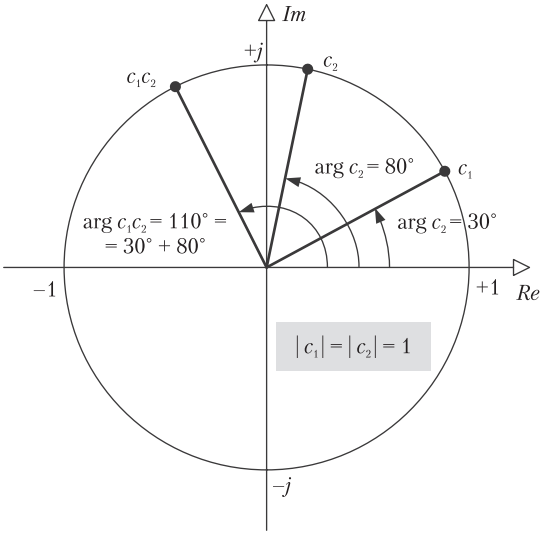


Рис. 14.17. Умножение двух комплексных чисел, абсолютные величины которых равны 1, создает чистое вращение в комплексной плоскости

Это объясняет, почему кватернионы *единичной длины* работают как операции вращения в трехмерном пространстве! Кватернион, в сущности, представляет собой четырехмерное комплексное число с одной вещественной и тремя мнимыми составляющими. Поэтому в трехмерном пространстве он придерживается тех же основных правил, что и комплексное число на плоскости.

То, что в результате комплексного умножения получается вращение, становится логичным, если взглянуть на процесс многократного умножения j на j:

$$1 \cdot j = j;\ j \cdot j = \sqrt{-1}\sqrt{-1} = -1;\ -1 \cdot j = -j;\ -j \cdot j = 1\ldots$$

Таким образом, умножение j на само себя эквивалентно *повороту* единичного вектора на 90° в комплексной плоскости. На самом деле это происходит при умножении j на *любое* другое комплексное число (рис. 14.18).

Формулы Эйлера и комплексной экспоненты. Если модуль комплексного числа $|c| = 1$, то функция $f(n) = c^n$, где n относится к возрастающей последовательности положительных вещественных чисел, проходит *по кругу* в комплексной плоскости. Любой круговой контур в двухмерном пространстве проецирует синусоидальную и косинусоидальную кривые на вертикальную и горизонтальную оси соответственно (рис. 14.19).

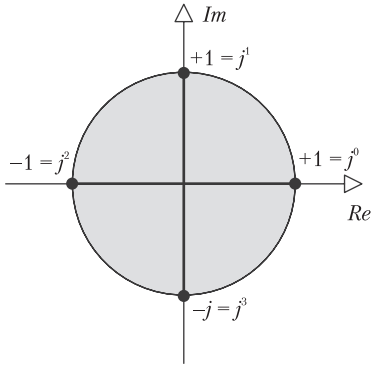


Рис. 14.18. Умножение мнимого числа j = $\sqrt{-1}$ на само себя выглядит как поворот единичного вектора на 90° в комплексной плоскости

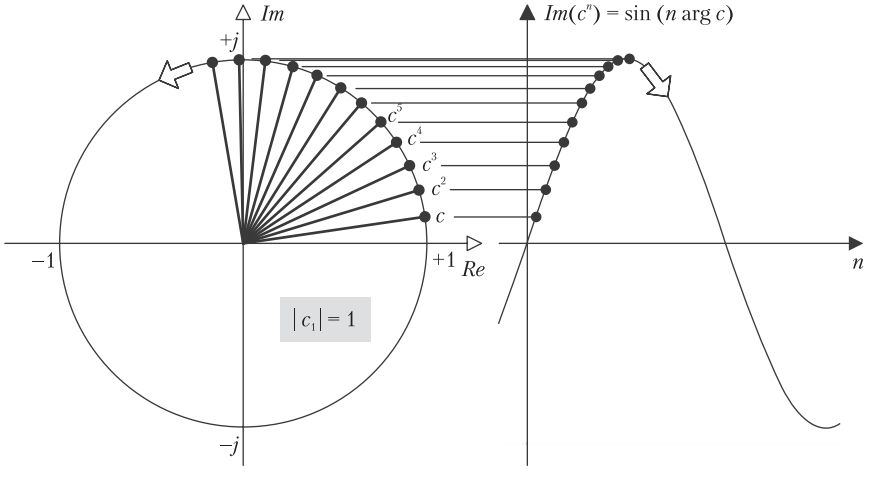


Рис. 14.19. Многократное умножение комплексного числа на само себя создает круговой контур в комплексной плоскости и проецирует синусоиду на любую ось, проходящую через начало координат

Возведение *комплексного* числа в *вещественную* степень *(c^n)* создает вращение в комплексной плоскости и, следовательно, проецирует синусоиду на любую ось в том же пространстве. Оказывается, того же результата можно добиться, если возвести *вещественное* число в *комплексную* степень *(n^c)*. Это означает, что уравнение (14.10) можно записать с помощью комплексных чисел:

$$e^{j\omega_0 t} = \cos\omega_0 t + j\sin\omega_0 t,\ t \in \mathbb{R}; \qquad (14.13)$$

$$\mathrm{Re}\left[e^{j\omega_0 t}\right] = \cos\omega_0 t;\ \mathrm{Im}\left[e^{j\omega_0 t}\right] = \sin\omega_0 t,$$

где $e \approx 2{,}71828$ — вещественное трансцендентное число, являющееся основанием натурального логарифма.

Уравнение (14.13) является одним из самых важных в математике. Оно называется *формулой Эйлера*. Сам факт, что оно работает, остается в какой-то степени загадкой даже для некоторых опытных математиков. Чтобы объяснить эту теорему, можно взглянуть на разложение ряда Тейлора e^{jt} или производную e^x, в которой x становится комплексным числом. Но в нашем случае можно просто положиться на интуицию, ведь мы уже видели, как комплексное умножение создает вращение в комплексной плоскости.

Ряд Фурье

Итак, вооружившись математической теорией, мы должны записать синусоиды с помощью комплексных чисел. Давайте еще раз обратим внимание на представление сигнала в виде суммы синусоид.

Проще всего это сделать, когда сигнал *периодический*. В этом случае можем записать его как сумму *гармонически связанных* синусоид:

$$x(t) = \sum_{k=-\infty}^{+\infty} a_k e^{j(k\omega_0)t}. \tag{14.14}$$

Мы называем это представлением сигнала в виде *ряда Фурье*. Здесь комплексные экспоненциальные функции $e^{j(k\omega_0)t}$ выступают синусоидальными компонентами, из которых мы формируем сигнал. Эти составляющие *гармонично связаны* между собой в том смысле, что у каждого из них есть частота, которая является целочисленным множителем k так называемой *фундаментальной частоты* ω_0. Коэффициенты a_k представляют долю, занимаемую каждой из гармоник в сигнале $x(t)$.

Преобразование Фурье

Полноценное обсуждение этой темы выходит за пределы данной книги, но в нашем случае нужно сказать (без какого-либо доказательства!), что *любой* достаточно стабильный сигнал[1], даже не обязательно *периодический*, можно представить в виде линейной комбинации синусоид. В целом произвольный сигнал может содержать компоненты с *любыми* частотами, а не только теми, которые имеют гармоническую связь. Таким образом, дискретный набор гармонических коэффициентов a_k в уравнении (14.14) превращается в континуум значений, которые описывают долю каждой частоты в сигнале.

Мы можем представить себе новую функцию $X(\omega)$, в которой роль независимой переменной играет частота ω, а не время t и значение которой представляет долю каждой частоты, присутствующей в оригинальном сигнале $x(t)$. Можно сказать, что $x(t)$ — это представление сигнала с помощью *временной области*, а $X(\omega)$ — с помощью *частотной области*.

---

[1] У всех сигналов, отвечающих так называемым условиям Дирихле, есть преобразования Фурье, следовательно, в нашем контексте их можно назвать достаточно стабильными.

Каждое из этих представлений можно математически выразить друг через друга, используя *преобразование Фурье*:

$$X(\omega) = \int_{-\infty}^{+\infty} x(t) e^{-j\omega t} dt; \tag{14.15}$$

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) e^{j\omega t} d\omega. \tag{14.16}$$

Если сравнить уравнение (14.16) с рядом Фурье из уравнения (14.14), можно заметить определенное сходство. Вместо описания абсолютных величин частотных компонентов с помощью дискретного ряда коэффициентов a_k мы описываем их с помощью непрерывной функции $X(\omega)$. Но в обоих случаях функция $x(t)$ представлена суммой синусоид.

Диаграммы Боде

В целом, если применить преобразование Фурье к сигналу с вещественным значением, получится сигнал с *комплексным значением* ($X(\omega) \in C$). Преобразование Фурье часто иллюстрируют с помощью двух графиков. Например, мы можем изобразить его вещественную и мнимую составляющие. Можем также построить отдельные графики для его абсолютной величины и аргумента (угла) — этот подход известен как *диаграмма Боде* (рис. 14.20).

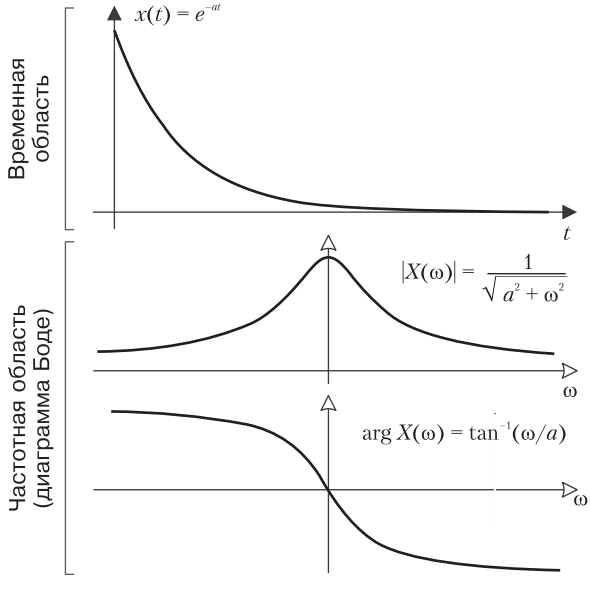


Рис. 14.20. Преобразование Фурье позволяет получить сигнал частотной области с комплексными значениями. Для визуализации этого сигнала с комплексными значениями с учетом его абсолютной величины и фазы (или аргумента) используется диаграмма Боде

Быстрое преобразование Фурье

Для вычисления преобразования Фурье с дискретным временем существует коллекция быстрых алгоритмов с довольно метким названием — *быстрое преобразование Фурье* (БПФ). Больше о БПФ можно почитать на странице ru.wikipedia.org/wiki/Быстрое_преобразование_Фурье.

Преобразования Фурье и свертка

Интересно, что свертка во временной области эквивалентна умножению в частотной области и наоборот. Имея систему с импульсной характеристикой $h(t)$, мы знаем, что ее выход $y(t)$ в ответ на вход $x(t)$ будет следующим:

$$y(t) = x(t) * h(t).$$

Если взять частотную область, то при наличии преобразований Фурье для импульсной характеристики $H(\omega)$ и входа $X(\omega)$ можно найти преобразование Фурье для выхода:

$$Y(\omega) = X(\omega)\, H(\omega).$$

Невероятный, но в то же время очень удобный результат. Иногда удобнее свертку производить на временной оси, используя импульсную характеристику системы $h(t)$, а умножение в частотной области — выполнять с помощью *амплитудно-частотной характеристики* (АЧХ) $H(\omega)$.

Оказывается, системам ЛСС свойственна так называемая *двойственность*: если поменять местами время и частоту, математические правила, действующие в системе, практически не изменятся. Например, чтобы понять, как работает *модуляция* сигнала (умножение одного сигнала на другой) во временной области, можно понаблюдать за сверткой преобразований Фурье двух сигналов на частотной оси. Два способа представления одной и той же задачи всегда лучше, чем один!

Фильтрация

Преобразование Фурье позволяет визуализировать набор частот, из которых состоит практически любой аудиосигнал. *Фильтр* — это система ЛСС, которая ослабляет входящие частоты в выбранном диапазоне, оставляя все остальные без изменений. Фильтр *нижних частот* сохраняет нижние частоты, ослабляя высокие. Фильтр *высоких частот* имеет противоположный эффект: высокие частоты остаются нетронутыми, а низкие ослабляются. *Полосовой* фильтр ослабляет как низкие, так и высокие частоты, но не трогает частоты в ограниченной *полосе пропускания*. *Режекторный* фильтр делает наоборот: оставляет без изменений низкие и высокие частоты, но ослабляет определенную ограниченную *полосу подавления*.

Фильтры используются в *эквалайзерах* стереосистем для подавления или усиления определенных частот в зависимости от пользовательских настроек. С помощью фильтров также можно подавлять шум, если его спектр не совпадает со спектром полезного сигнала на частотной оси. Например, если высокочастотный шум плохо

сказывается на звучании более низкочастотного голоса или музыки, фильтр низких частот может его устранить.

АЧХ $H(\omega)$ идеального фильтра выглядит как прямоугольник с высотой 1 в полосе пропускания и 0 в полосе подавления. Если умножить ее на преобразование Фурье входного сигнала $X(\omega)$, выходной сигнал $Y(\omega) = X(\omega)\ H(\omega)$ сохранит все пропущенные частоты в исходном виде, а подавленные частоты будут обнулены. АЧХ идеального фильтра показана на рис. 14.21.

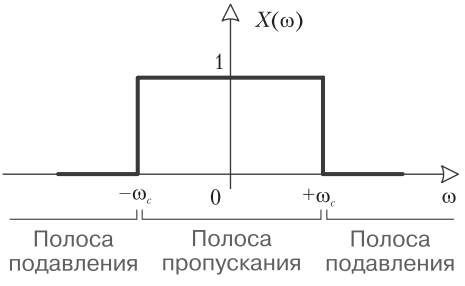


Рис. 14.21. АЧХ $H(\omega)$ идеального фильтра имеет полосу пропускания 1 и полосу подавления 0

Конечно, идеальный фильтр с полным пропуском и полным подавлением определенных частот — это, наверное, не совсем то, что нам нужно. В АЧХ большинства реальных фильтров полоса пропускания плавно переходит в полосу подавления. Это помогает в ситуациях, когда между нужными и нежелательными частотами нет четкого разделения. На рис. 14.22 показана АЧХ фильтра низких частот с плавным переходом.

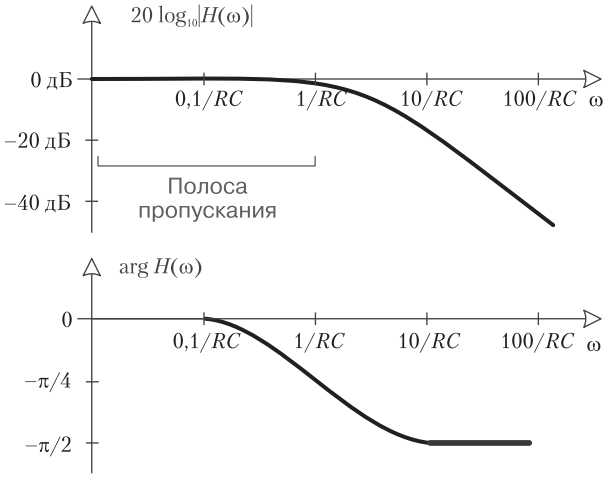


Рис. 14.22. АЧХ $H(\omega)$ для низкочастотного RC-фильтра (резисторно-конденсаторного) с плавным переходом. Горизонтальные и вертикальные оси на обоих графиках имеют логарифмическую шкалу

В звуковой аппаратуре с высокой точностью воспроизведения *эквалайзеры* обычно дают возможность пользователям регулировать басы, средние и высокие частоты в итоговом сигнале. Эквалайзер, в сущности, — это просто набор фильтров, рассчитанных на разные частотные диапазоны, которые последовательно применяются к звуковому сигналу.

Теория фильтрации является необъятной областью, охватить которую мы попросту не в состоянии. Куда больше информации на эту тему можно найти в главе 6 и [41].

14.3. Аудиотехнологии

Прежде чем переходить к полноценному обсуждению программного обеспечения, из которого состоит звуковой движок игры, нужно убедиться в том, что мы хорошо ориентируемся в аудиотехнологиях, звуковом оборудовании и терминологии, которую используют профессионалы в данной области.

14.3.1. Аналоговые аудиотехнологии

Первые звуковые устройства были аналоговыми. Это был самый простой способ записи, изменения и воспроизведения звуковых волн сжатия, так как звук сам по себе является аналоговым физическим явлением. В этом разделе мы кратко исследуем некоторые ключевые аналоговые аудиотехнологии.

Микрофоны

Микрофон — это преобразователь, который переводит звуковую волну в электронный сигнал. В микрофонах применяются различные технологии для преобразования механических колебаний давления звуковой волны в соответствующий сигнал, основанный на колебаниях электрического напряжения. *Динамические микрофоны* используют электромагнитную индукцию, а *конденсаторные* — изменение емкости. Существуют и другие виды микрофонов, которые создают сигнал напряжения за счет пьезоэлектрического эффекта и модуляции света.

Разные микрофоны обладают различными профилями чувствительности, которые называют также *характеристиками направленности*. Эти характеристики описывают то, насколько чувствителен микрофон к звукам, поступающим под тем или иным углом относительно его центральной оси. Ненаправленные микрофоны одинаково чувствительны ко всем направлениям, график двунаправленных имеет две доли в форме восьмерки, кардиоидные характеризуются практически однонаправленным профилем чувствительности, их график имеет форму, напоминающую сердце, — отсюда и название. Некоторые распространенные характеристики направленности проиллюстрированы на рис. 14.23.

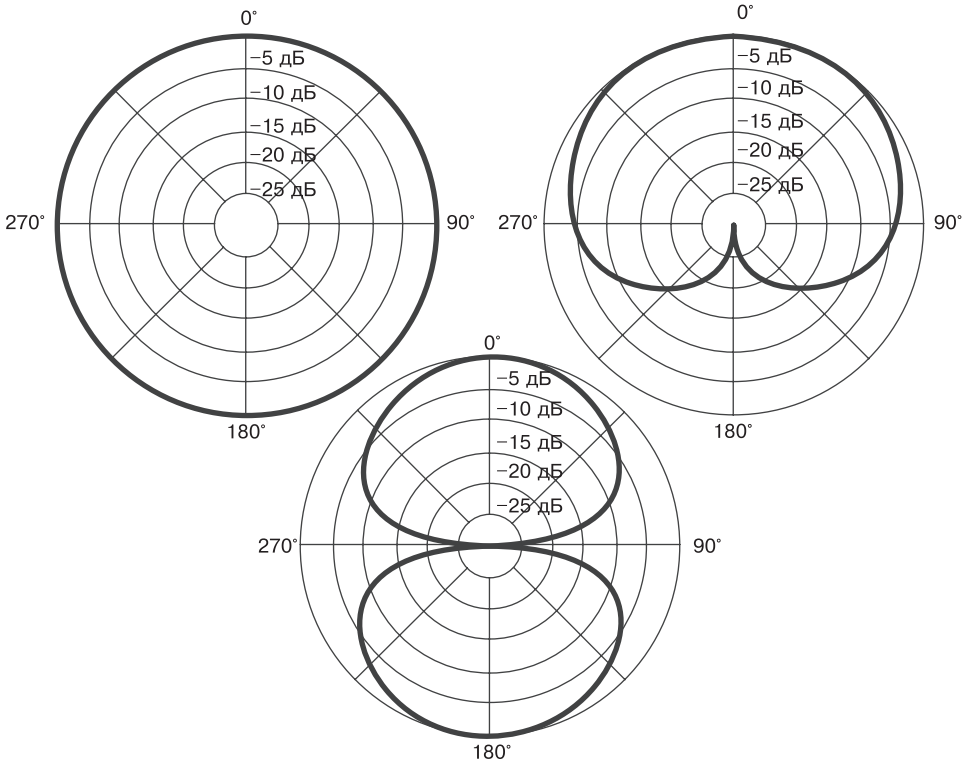


Рис. 14.23. Три типичных вида направленности микрофонов (по часовой стрелке, начиная с левого верхнего): ненаправленный, кардиоидный и двунаправленный

Динамики

Динамик — это, в сущности, микрофон наоборот. Он преобразует колебания напряжения (входящий сигнал) в вибрации мембраны, которые, в свою очередь, создают колебания давления в воздухе, формирующие звуковую волну.

Компоновка динамиков: стерео

Звуковые системы обычно поддерживают несколько каналов вывода. Стереоустройства вроде iPod, звуковая система в автомобиле, переносной музыкальный центр вашего дедушки — все они поддерживают как минимум два динамика — для левого и правого стереоканалов. У некоторых высококачественных стереосистем есть две дополнительные «пищалки» — крошечные динамики, способные воспроизводить самые высокочастотные звуки в левом и правом каналах. Это позволяет сделать основные динамики более крупными, с лучшей передачей басов. Некоторые устройства также поддерживают сабвуфер или динамик с LFE (low-frequency

effects — «низкочастотные эффекты»), их иногда обозначают 2.1: 2 — левый и правый каналы, а .1 — LFE-динамик.

Сравнение наушников и динамиков. Существует большая разница между стереодинамиками, работающими в просторном помещении, и стереонаушниками. Стереодинамики обычно размещают перед слушателем на каком-то расстоянии друг от друга. Это означает, что звуковые волны, издаваемые *левым* динамиком, попадают в том числе и в *правое* ухо и наоборот. Если динамик удален от слушателя, его волны будут доходить с небольшой задержкой (сдвигом фазы) и незначительным затуханием. Звуковые волны со сдвинутой фазой, исходящие из удаленного динамика, обычно *влияют* на волны, издаваемые более близким источником. Это должны учитывать производители аудиооборудования, которые стремятся обеспечить как можно более высокое качество звука.

Наушники прилегают непосредственно к ушам, поэтому левый и правый каналы идеально изолированы и между ними не возникает *интерференции*. Кроме того, звук попадает сразу в наружный слуховой проход, поэтому отсутствуют передаточные эффекты, связанные с формой ушных раковин (см. подраздел 14.1.4). Это означает, что слушатель получает меньше пространственной информации.

Компоновка динамиков: объемный звук

Системы *объемного звука* в домашних кинотеатрах обычно делятся на две категории: 5.1 и 7.1. Как вы уже наверняка догадались, эти числа обозначают пять или семь основных динамиков плюс один сабвуфер. Такие системы обеспечивают эффект погружения, создавая реалистичный звуковой ландшафт за счет сочетания *позиционной* информации и высокоточного воспроизведения звука (см. подраздел 14.1.4). Система 5.1 имеет такие основные каналы[1]: центральный, фронтальный левый, фронтальный правый, тыловой левый, тыловой правый. В системах 7.1 есть еще два динамика для более объемного звука, которые размещаются по обе стороны от слушателя. Одними из самых популярных систем объемного звука являются Dolby Digital AC-3 и DTS. На рис. 14.24 показан принцип размещения динамиков в типичном домашнем кинотеатре формата 7.1.

Технологии Dolby Surround, Dolby Pro Logic и Dolby Pro Logic II предназначены для превращения исходного стереосигнала в объемный звук. Стереосигналу не хватает позиционной информации, необходимой для динамиков в конфигурации 5.1, но технологии Dolby позволяют сгенерировать ее приближенную версию эвристическим способом, используя различные характерные участки, найденные в исходном стереосигнале.

---

[1] См. https://ru.wikipedia.org/wiki/Объёмный_звук#Система_обозначений_каналов_звука_(Формат_громкоговорителей). — *Примеч. пер.*

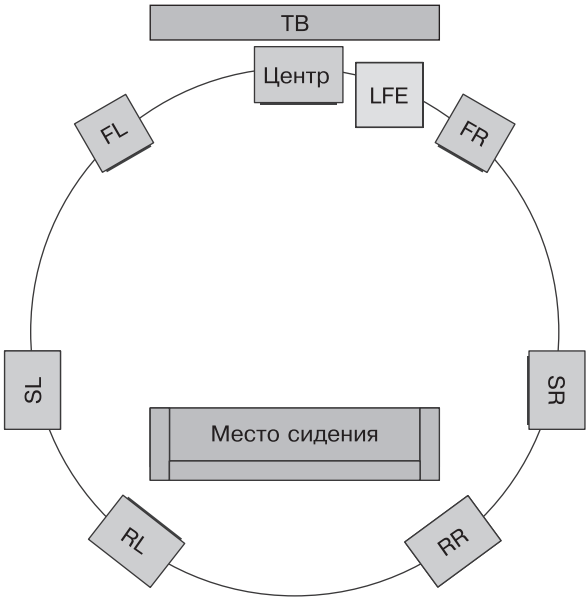


Рис. 14.24. Расстановка динамиков в домашнем кинотеатре с системой объемного звука 7.1

Уровни аналогового сигнала

Звуковые сигналы можно передавать на разных уровнях напряжения. Микрофон обычно генерирует напряжение с достаточно низкой амплитудой — это так называемый *микрофонный уровень*. Для соединения разных компонентов используется *линейный уровень* с более высоким напряжением. В профессиональной аудиоаппаратуре и потребительской электронике задействуются совершенно разные напряжения линейного уровня. Профессиональное оборудование в основном предназначено для работы с линейными уровнями с размахом сигнала от 2,191 В до 3,472 В. Напряжение сигнала линейного уровня в потребительских устройствах может немного разниться, но в большинстве случаев оно составляет 1,0 В, причем некоторые устройства способны работать с входным напряжением до 2,0 В. При подключении аудиооборудования уровни входного и выходного сигналов обязательно должны быть подходящими. Если подать слишком высокое напряжение, может произойти клиппирование сигнала. А если напряжение окажется слишком низким, звук получится более тихим, чем задумывалось.

Усилители

Напряжение низкого уровня, генерируемое микрофоном, необходимо *усилить*, чтобы энергии, которая возникает в динамиках, было достаточно для создания слышимых звуковых волн. *Усилитель* — это аналоговый электронный компонент, входной и выходной сигналы которого почти ничем не отличаются, если

не считать существенного увеличения *амплитуды* последнего. Вместе с амплитудой фактически увеличивается и *мощность* сигнала. Повышенное напряжение берется из какого-то источника питания и затем модулируется таким образом, чтобы имитировать поведение входящего сигнала по времени. Иными словами, усилитель *модулирует* выходное напряжение своего источника питания так, чтобы он совпадал по форме с входным сигналом существенно меньшего напряжения.

Основной составляющей усилителя является *транзистор* — широко известное и совершенно гениальное изобретение, лежащее в основе многих современных электронных устройств, включая венец цифровой эволюции — компьютер. Транзистор использует полупроводящий материал для согласования напряжения в двух изолированных между собой цепях. Таким образом, сигнал с низким напряжением может управлять цепью с более высоким напряжением. Это именно то, что требуется от усилителя. Мы не станем подробно обсуждать, как работают транзисторы и усилители. Но если вам интересно, можете подогреть свой аппетит, посмотрев замечательное видео о принципе работы самого первого транзистора: https://www.youtube.com/watch?v=RdYHljZi7ys. Больше об усилителях можно почитать, перейдя по ссылке https://ru.wikipedia.org/wiki/Усилитель.

Коэффициент усиления A усилителя определяется отношением выходной мощности (P_{out}) к входящей (P_{in}). Как и уровень давления звука, коэффициент передачи обычно измеряется в децибелах:

$$A = 20\log_{10}\left(\frac{P_{out}}{P_{in}}\right).$$

Регуляторы уровня и коэффициента передачи

Регулятор уровня, которые также называют *аттенюатором*, в сущности, является устройством, обратным усилителю. Вместо того чтобы увеличивать амплитуду электрического сигнала, он ее *уменьшает*, при этом оставляя нетронутыми все остальные аспекты волны. В домашнем кинотеатре цифро-аналоговый преобразователь генерирует напряжение с очень маленькой амплитудой. Усилитель повышает мощность этого сигнала до максимально безопасного уровня, при превышении которого происходит отсечение и искажение звука (или даже повреждение аппаратуры). Затем регулятор уровня понижает максимальную выходную мощность, чтобы получить звук нужной громкости.

Регулятор уровня устроен намного проще, чем усилитель. Чтобы его соорудить, достаточно поместить резистор переменного сопротивления *(потенциометр)* где-то между выходом усилителя и динамиками. При минимальном сопротивлении (нулевом или близком к тому) амплитуда входящего сигнала не меняется, и в результате звук воспроизводится с максимальной громкостью. Когда сопротивление находится на самой высокой отметке, амплитуда входящего сигнала максимально подавляется и полученный звук имеет минимальную громкость.

Если ваша домашняя стереосистема позволяет регулировать уровень в децибелах, вы, наверное, заметили, что все доступные значения — отрицательные. Дело в том, что регулятор уровня подавляет вывод усилителя. Громкость попрежнему определяется коэффициентом передачи, однако на вход подается максимальная мощность усилителя, а выходная мощность — это уровень, выбранный пользователем:

$$A = 10\log_{10}\left(\frac{P_{\text{volume}}}{P_{\text{max}}}\right),$$

и пока $P_{\text{volume}} < P_{\text{max}}$, он будет оставаться отрицательным.

Разъемы и подключение аналоговых устройств

Аналоговый монофонический звуковой сигнал на основе напряжения можно передавать по двум проводам, стереосигнал требует трех проводов (два канала плюс общее заземление). Провода могут находиться внутри устройства, в этом случае их обычно называют *шиной*. Но они могут быть и внешними и использоваться для соединения разных устройств.

Внешняя проводка, как правило, подключается к аудиооборудованию либо через прямой «зажим», либо через винтовой разъем, аналогичный тому, который имеется на высококачественных динамиках. Применяются и стандартные разъемы. В качестве примеров можно привести разъемы RCA и TRS (tip/ring/sleeve — кончик/кольцо/гильза, использовался телефонными операторами в начале XX века), мини-разъем TRS (можно встретить в iPod, мобильных телефонах и большинстве звуковых карт), пронумерованные разъемы (в основном применяются в высококачественных микрофонах и усилителях) и многие другие.

Качество аудиокабелей может сильно варьироваться. Чем толще их жилы, тем меньше сопротивление, следовательно, сигнал может передаваться на бóльшее расстояние с допустимым уровнем затухания. Дополнительное экранирование способно уменьшить помехи. И конечно, выбор металла, из которого состоят кабели и разъемы, может сказаться на качестве соединения.

14.3.2. Цифровые аудиотехнологии

Появление компакт-дисков (CD) ознаменовало переориентацию звуковой индустрии на цифровые средства хранения и обработки. Цифровые технологии открывают множество новых возможностей: от уменьшения размера и увеличения емкости носителей до применения мощного аппаратного и программного обеспечения для синтеза звука и манипуляции им невиданными доселе способами. Аналоговые носители остались в прошлом, а аналоговые аудиосигналы используются только там, где они необходимы, — в микрофонах и динамиках.

Как мы уже видели в подразделе 14.2.1, различие между аналоговыми и цифровыми аудиотехнологиями в точности соответствует разделению между сигналами *непрерывного* и *дискретного времени* в теории обработки сигналов.

Преобразование аналогового сигнала в цифровой: импульсно-кодовая модуляция

Чтобы записанный звук можно было использовать в цифровой системе, такой как компьютер или игровая консоль, колебания напряжения в аналоговом аудиосигнале нужно преобразовать в цифровой вид. Импульсно-кодовая модуляция (pulse-code modulation, PCM) — это стандартный метод кодирования дискретизированного аналогового звукового сигнала таким образом, чтобы его можно было хранить в памяти компьютера, передавать по цифровой телефонной сети или записывать на компакт-диски.

В процессе импульсно-кодовой модуляции напряжение замеряется с определенной периодичностью. Полученную информацию можно хранить в виде чисел с плавающей запятой или *квантовать* для представления их в виде целых чисел фиксированной длины (обычно 8, 16, 24 или 32 бита). Затем последовательность снятых показаний напряжения сохраняется в массив в оперативной памяти или записывается на носитель для длительного хранения. Процесс измерения единого аналогового сигнала и перевода его в квантованную численную форму называется *аналого-цифровым (АЦ) преобразованием*. Для этого обычно применяется специальное оборудование. Если в ходе этого процесса используются интервалы одинаковой длины, он называется *дискретизацией* (или *семплингом*). Аппаратный или программный компонент, выполняющий АЦ-преобразование и/или дискретизацию, называют аналого-цифровым преобразователем (АЦП).

Говоря математическим языком, мы создаем *дискретную* версию $p[n]$ звукового сигнала непрерывного времени $p(t)$ так, чтобы для каждого семпла было справедливым уравнение $p[n] = p(nT_s)$, где n — неотрицательное целое число, применяемое для индексации семплов, а T_s — временной интервал между двумя семплами, известный как *период дискретизации*. Основы этого процесса проиллюстрированы на рис. 14.25.

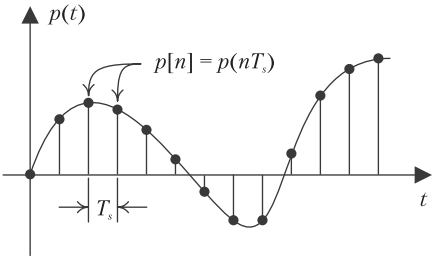


Рис. 14.25. Сигнал дискретного времени можно считать дискретизированной версией сигнала непрерывного времени

Цифровой сигнал, полученный в результате дискретизации методом PCM, характеризуется двумя важными свойствами.

- *Частота дискретизации*. Это частота, с которой снимались показания напряжения (семплы). Теоретически аналоговый сигнал можно записать в цифровом

виде без *каких-либо* потерь, если он дискретизируется с частотой, *вдвое* превосходящей его самую высокочастотную составляющую. Этот несколько удивительный и невероятно полезный факт известен как *теорема Котельникова*. Как говорилось в подразделе 14.1.2, люди способны воспринимать звуки в ограниченной полосе частот — от 20 Гц до 20 кГц. Поэтому все звуковые сигналы, которые нас интересуют, имеют ограниченную полосу и могут быть записаны с высокой точностью при помощи частоты дискретизации чуть выше 40 кГц (для качественной передачи голоса достаточно более узкой полосы частот — от 300 Гц до 3,4 кГц, поэтому системы цифровой телефонии могут обойтись частотой дискретизации 8 кГц).

- *Битовая глубина*. Так называется количество битов, которое используется для представления каждого квантованного показания напряжения. *Ошибка квантования* — это погрешность, возникающая при округлении снимаемых показаний напряжения к ближайшему квантованному значению. При прочих равных условиях увеличение битовой глубины уменьшает ошибку квантования и тем самым улучшает качество записываемого звука. В несжатых форматах аудиоданных обычно применяют глубину 16 бит. Иногда эту величину называют *разрешением*.

Теорема Котельникова. Теорема Котельникова утверждает, что, если сигнал непрерывного времени с *ограниченной полосой частот* (то есть сигнал, преобразование Фурье которого всегда дает ноль за пределами заданной полосы) оцифровать с достаточно высокой частотой дискретизации, полученный результат можно превратить обратно в сигнал непрерывного времени, *не отличающийся* от оригинала. Минимальная частота дискретизации, которая делает это утверждение справедливым, называется *частотой Найквиста*:

$$\omega_s > 2\omega_{max},$$

где

$$\omega_s = \frac{2\pi}{T_s}.$$

Очевидно, что существование этой теоремы делает возможным применение цифровых технологий в обработке звука. Без нее цифровой звук никогда бы не звучал так же хорошо, как аналоговый, и компьютеры не играли бы важной роли в создании профессионального аудио, как происходит сегодня.

Мы не станем вдаваться в мельчайшие подробности теоремы Котельникова. Но, чтобы получить общее представление о том, как она работает, достаточно осознать, что в результате дискретизации сигнала с одинаковыми временными интервалами ее частотный спектр (преобразование Фурье) дублируется по оси частот снова и снова. Чем выше частота дискретизации, тем более растянутыми получаются копии частотного спектра сигнала. Поэтому, если оригинальный сигнал имеет ограниченную полосу частот и частота дискретизации достаточно высокая, мы можем гарантировать, что копии частотного спектра будут находиться на достаточном расстоянии друг от друга, чтобы не пересекаться. В этом случае мы можем идеально

восстановить оригинальный спектр с помощью фильтра низких частот, который отбрасывает все копии, кроме оригинала. Но если частота дискретизации слишком низкая, копии спектра будут пересекаться. Это называется *эффектом наложения*. В результате мы не можем восстановить спектр оригинального сигнала в первозданном виде. На рис. 14.26 проиллюстрирована дискретизация с наложением и без.

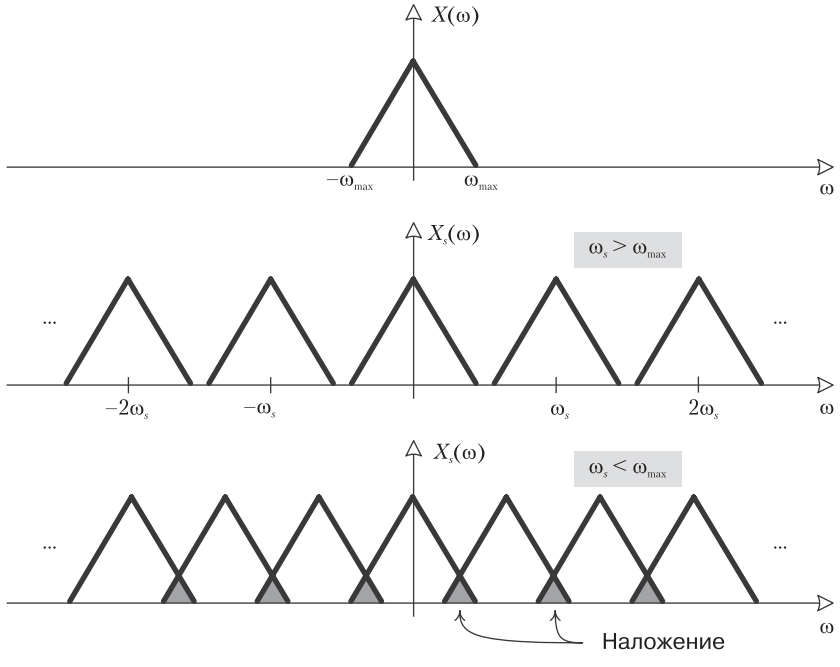


Рис. 14.26. Частоты в спектре с ограниченной полосой равны нулю везде, за исключением этой ограниченной полосы *(вверху)*. Если частота дискретизации превышает частоту Найквиста, копии спектра не пересекаются и оригинальный сигнал можно восстановить в первозданном виде *(посредине)*. Если частота дискретизации слишком низкая, копии спектра пересекаются и полученный результат страдает от эффекта наложения *(внизу)*

Перевод цифрового сигнала в аналоговый вид: демодуляция

При воспроизведении цифрового звукового сигнала требуется процесс, обратный аналого-цифровому преобразованию. Мы называем его *цифроаналоговым преобразованием* (ЦАП), что довольно логично. Этот процесс называют также *демодуляцией*, поскольку он обращает вспять результаты импульсно-кодовой модуляции.

Оборудование для ЦАП генерирует аналоговое напряжение, соответствующее каждому результату измерений в цифровом сигнале, который хранится в памяти в виде массива квантованных PCM-значений. Если передавать в ЦАП значения с частотой, с которой происходила дискретизация в ходе PCM, и если эта частота была достаточно высокой согласно теореме Котельникова, полученный аналоговый сигнал должен в точности совпадать с оригиналом.

На практике, когда в аналоговую цепь подается напряжение с последовательно меняющимися дискретными уровнями, часто возникают нежелательные высокочастотные колебания, так как оборудование пытается быстро переходить от одного уровня напряжения к другому. Чтобы избавиться от этих колебаний, ЦА-аппаратура обычно содержит фильтр низких или высоких частот, что позволяет обеспечить точное воспроизведение оригинального аналогового сигнала. Подробнее о фильтрации говорилось в подразделе 14.2.5.

Цифровые аудиоформаты и кодеки

Для хранения PCM-звука на диске и передачи его по сети существуют различные форматы данных. У каждого из них своя история, и каждый имеет положительные и отрицательные стороны. Некоторые, например AVI, в действительности являются контейнерами, способными инкапсулировать цифровые звуковые сигналы в разных форматах.

Некоторые аудиоформаты хранят PCM-данные в несжатом виде. Другие применяют различные методы сжатия, чтобы уменьшить размер файла и объем передаваемого трафика. Иногда это приводит к *потере* информации, то есть в процессе компрессии/декомпрессии теряется какая-то часть оригинального сигнала. Существуют также способы сжатия *без потерь*, благодаря которым оригинальные PCM-данные можно восстановить в первозданном виде.

Рассмотрим несколько наиболее распространенных аудиоформатов.

- *Обычные PCM-данные без заголовков* иногда используются в ситуациях, когда метаинформация о сигнале, такая как частота дискретизации и битовая глубина, известна заранее.
- *LPCM* (линейная версия PCM) — это аудиоформат без сжатия, поддерживающий до восьми звуковых каналов с частотой дискретизации от 48 до 96 кГц и семплы размером 16, 20 или 24 бита. Линейным этот формат делает то, что амплитуда измеряется с помощью линейной (а не, скажем, логарифмической) шкалы.
- *WAV* — это формат файлов без сжатия, созданный компаниями Microsoft и IBM. Он повсеместно применяется в операционных системах Windows. Расшифровывается аббревиатура как waveform audio file format — «формат аудиофайлов в форме волны», хотя изредка его ошибочно называют audio for Windows — «аудио для Windows». На самом деле это представитель семейства форматов, известного как *RIFF* (resource interchange file format — «файловый формат обмена ресурсами»). RIFF-файл состоит из кусков, каждый из которых содержит четырехсимвольный код (FOURCC), определяющий содержимое куска, и поле с его размером. Битовый поток в WAV-файлах соответствует формату линейной импульсно-кодовой модуляции (LPCM). WAV-файлы могут содержать также сжатый звук, но это встречается нечасто.
- *WMA* (Windows Media Audio) — патентованная технология сжатия звука, разработанная компанией Microsoft в качестве альтернативы MP3. Подробности ищите на странице ru.wikipedia.org/wiki/Windows_Media_Audio.

- *AIFF* (audio interchange file format — «файловый формат обмена аудио») — это формат, разработанный компанией Apple Computer, Inc. и широко используемый на компьютерах Macintosh. По аналогии с WAV/RIFF файл AIFF обычно содержит несжатые PCM-данные и состоит из кусков, в начале каждого из которых имеются четырехсимвольный код и поле с размером. У формата AIFF есть версия с поддержкой сжатия, AIFF-C.
- *MP3* — это формат сжатых аудиофайлов с потерями, который стал фактическим стандартом в большинстве цифровых аудиопроигрывателей и широко используется в играх и мультимедийных системах/сервисах. Его полное название — MPEG-1 (или MPEG-2) audio layer III. Сжатие в MP3 позволяет добиться десятикратного уменьшения размера файлов с малозаметными отличиями от оригинального несжатого звука. Такие результаты достигаются за счет *перцептуального кодирования* — методики, устраняющей те части звукового сигнала, которые для большинства людей находятся за гранью восприятия.
- *ATRAC* расшифровывается как Adaptive Transform Acoustic Coding. Это семейство патентованных методов сжатия звука, разработанных компанией Sony. Данный формат изначально создавался для того, чтобы носители MiniDisc (тоже изобретенные Sony) могли вместить аудио с тем же временем воспроизведения, что и CD, занимая при этом намного меньше места и демонстрируя ухудшение качества, которое невозможно заметить. Подробнее об этом — по ссылке ru.wikipedia.org/wiki/ATRAC.
- *Ogg Vorbis* — это открытый формат файлов, поддерживающий сжатие с потерями. Ogg обозначает «контейнер», который обычно используется в сочетании с форматом данных Vorbis.
- *Dolby Digital (AC-3)* — формат сжатия с потерями, поддерживающий разные конфигурации каналов — от моно до объемного звука 5.1.
- *DTS* — это набор технологий для кинотеатров, разработанный компанией DTS, Inc. Цифровой аудиоформат DTS Coherent Acoustics поддерживает передачу через интерфейсы S/PDIF (см. подраздел 14.3.2) и используется в DVD и LaserDisc.
- *VAG* — патентованный формат аудиофайлов, доступный для применения всеми разработчиками на PlayStation 3. Он использует *адаптивную дифференциальную* версию PCM (ADPCM) — метод аналого-цифрового преобразования на основе PCM. Дифференциальная версия PCM (DPCM) хранит не абсолютные значения семплов, а разницу между ними, что делает возможным более эффективное сжатие сигнала. Высокой степени сжатия позволяет добиться также то, что в DPCM частота дискретизации варьируется динамически.
- *MPEG-4 SLS*, *MPEG-4 ALS* и *MPEG-4 DST* — форматы, поддерживающие сжатие без потерь.

Это далеко не полный список. Существует огромное количество форматов аудиофайлов и еще больше алгоритмов сжатия/разжатия. Чтобы познакомиться

с этой увлекательной областью, можете почитать старую добрую «Википедию»: ru.wikipedia.org/wiki/Цифровые_аудиоформаты. Отличную информацию о звуковых форматах можно найти на сайте «PlayStation 3 Secrets» (bit.ly/2HOVtvR).

Параллельные и чередующиеся звуковые данные

Чтобы организовать многоканальные звуковые данные, семплы каждого монофонического канала можно хранить в отдельном буфере. Например, в случае со звуковым сигналом в конфигурации 5.1 вам понадобились бы шесть параллельных буферов (рис. 14.27).

Параллельные каналы

C[n]	FL[n]	FR[n]	RL[n]	RR[n]	LFE[n]
C[n + 1]	FL[n + 1]	FR[n + 1]	RL[n + 1]	RR[n + 1]	LFE[n + 1]
C[n + 2]	FL[n + 2]	FR[n + 2]	RL[n + 2]	RR[n + 2]	LFE[n + 2]
…	…	…	…	…	…

Рис. 14.27. Шестиканальные (5.1) данные в формате PCM с параллельными каналами

Многоканальные звуковые данные могут также *чередоваться* внутри одного буфера. В этом случае все семплы для каждого временного индекса группируются и размещаются в заранее определенном порядке. На рис. 14.28 показан чередующийся буфер PCM, содержащий шестиканальный (5.1) звуковой сигнал.

Чередующиеся каналы

C[n]
FL[n]
FR[n]
RL[n]
RR[n]
LFE[n]
C[n + 1]
FL[n + 1]
FR[n + 1]
…

Рис. 14.28. Шестиканальные (5.1) данные в формате PCM с чередующимися каналами

Разъемы и подключение цифровых устройств

S/PDIF (Sony/Philips Digital Interconnect Format) — технология соединения устройств с *цифровой* передачей звуковых сигналов. Исключает появление шумов, вызванных аналоговыми подключениями. В реальности соединение стандарта S/PDIF выполняется с помощью коаксиальных или оптоволоконных кабелей, известных как TOSLINK.

Независимо от физического интерфейса (коаксиального или оптоволоконного TOSLINK) протокол S/PDIF способен передавать двухканальный 24-битный несжатый звук в формате LPCM со стандартной частотой дискретизации в диапазоне от 32 до 192 кГц. Однако не всякое оборудование поддерживает полный диапазон частот дискретизации. Один и тот же интерфейс может использоваться для передачи

закодированного битового аудиопотока (например, в формате сжатия с потерями Dolby Digital или DTS) с битрейтом от 32 до 640 кбит/с в случае с Dolby Digital и от 768 до 1536 кбит/с в случае с DTS.

Несжатые многоканальные (то есть с более чем двумя стереоканалами) данные в формате LPCM в потребительском звуковом оборудовании можно передавать только по HDMI (high-definition multimedia interface — «мультимедийный интерфейс высокой четкости»). HDMI-разъемы подходят для передачи как несжатого цифрового видео, так и сжатого/несжатого цифрового звука. Этот стандарт поддерживает битрейт вплоть до 36,86 Мбит/с для многоканального или потокового (в виде потока битов) звука. Но этот показатель варьируется в зависимости от видеорежима: полноценный битрейт доступен только в режимах, начиная с 720p/50 Гц. Подробнее об этом можно почитать в спецификации HDMI, в разделе Video Dependency. Альтернативные разъемы с высокой пропускной способностью от компании Apple, DisplayPort и Thunderbolt, во многом похожи на HDMI.

Звуковые сигналы иногда передают по USB. В большинстве игровых консолей разъем USB предназначен только для подключения наушников.

Также возможна беспроводная передача звуковых сигналов. Для этого чаще всего применяется стандарт Bluetooth.

14.4. Рендеринг звука в 3D

Итак, мы познакомились с физикой звука, математикой, лежащей в основе обработки сигналов, и различными технологиями, которые используются для записи и воспроизведения звука. В этом разделе обсудим, как применить все эти теоретические и технические знания в игровом движке, чтобы получить реалистичные звуковые ландшафты с эффектом присутствия.

Любой игре, действие которой происходит в трехмерном виртуальном мире, необходим какого-то рода *движок рендеринга 3D-звука*. Высококачественная система 3D-звука должна создавать насыщенный и правдоподобный звуковой ландшафт, который соответствует событиям, происходящим в трехмерном мире, помогает в продвижении сюжета и остается верным тональному дизайну игры.

- На вход эта система принимает мириады трехмерных *звуков*, которые исходят из всевозможных участков игрового мира: шаги, голоса, звуки столкновения разных объектов, выстрелы, звуки окружающей среды, такие как шум ветра или дождя, и т. д.
- Ее выходом является набор звуковых каналов, которые при воспроизведении с помощью динамиков как можно более правдоподобно передают то, что услышал бы игрок, если бы он действительно находился в виртуальном мире игры.

В идеале звуковой движок должен генерировать полноценный объемный звук в конфигурации 7.1 или 5.1, поскольку это дает игроку наиболее полную позиционную информацию. Но поддержка стерео тоже должна присутствовать, так как не у всех есть модные системы домашнего кинотеатра, к тому же некоторые люди используют наушники, чтобы не будить соседей.

Звуковой движок игры отвечает также за воспроизведение звуков, которые возникают не в игровом мире. Это относится к фоновой музыке, звукам игрового меню, закадровому голосу рассказчика, голосу персонажа игрока (особенно в шутерах от первого лица) и, возможно, некоторым звукам окружающей среды. Это так называемые *2D-звуки*. Они должны воспроизводиться непосредственно в динамиках после смешивания с объемным/пространственным выводом звукового движка.

14.4.1. Краткий обзор процесса рендеринга 3D-звука

Далее перечислены основные обязанности, возложенные на движок 3D-звука.

- *Синтез звука* — это процесс генерации звуковых сигналов в соответствии с событиями, происходящими в игровом мире. Для этого можно воспроизводить заранее записанные звуковые *клипы* или генерировать звуки *процедурно*, на этапе выполнения.
- *Спатиализация* — это процесс, создающий иллюзию того, что каждый 3D-звук доносится из соответствующего места в игровом мире. Эффект достигается за счет управления амплитудой каждой звуковой волны, то есть ее коэффициентом передачи и уровнем, двумя способами:
  - *затухание* в зависимости от расстояния определяет общую громкость звука для получения информации о его радиальном расстоянии от слушателя;
  - *панорамирование* определяет относительную громкость звука в каждом доступном динамике, чтобы предоставить информацию о том, *откуда* доносится звук.
- *Акустическое моделирование* повышает реализм сгенерированного звукового ландшафта путем имитации ранних отражений и поздних реверберации, которые характеризуют окружающее пространство. При этом учитываются препятствия, которые частично или полностью блокируют путь от источника звука к слушателю. Некоторые звуковые движки моделируют также *атмосферное поглощение*, зависящее от частоты (см. подраздел 14.1.3), и/или HRTF-эффекты (см. подраздел 14.1.4).
- При перемещениях источника звука и слушателя может учитываться также *эффект Доплера*.
- *Микширование* — это процесс управления относительной громкостью каждого 2D- и 3D-звука в игре. С одной стороны, он обусловливается физикой, а с другой — эстетическими соображениями звукоинженеров.

14.4.2. Моделирование мира звука

Чтобы сгенерировать звуковой ландшафт виртуального мира, мы должны сначала описать этот мир движку. Модель мира звука состоит из следующих элементов.

- *Источники 3D-звука*. Каждый 3D-звук в игровом мире представляет собой монофонический звуковой сигнал, исходящий из определенной *позиции*. Мы также должны сообщить движку о его *векторе скорости*, *характере направленности*

(ненаправленный, конический, плоскостной) и *радиусе действия*, за пределами которого его не слышно.

- *Слушатель*. Это виртуальный микрофон, расположенный в игровом мире. Он имеет *позицию*, *вектор скорости* и *направление*.
- *Модель окружающего пространства*. Она описывает *геометрию*, *свойства* поверхностей и объектов, присутствующих в игровом мире, и/или *акустические свойства* пространства, в котором происходит действие.
- *Позиции* слушателя и источника звука, а также *характер направленности* последнего используются в расчете *затухания в зависимости от расстояния*. *Направление* слушателя определяет систему координат, в которой вычисляется *угловая позиция* звука. Этот угол определяет *панорамирование* — относительную громкость в пяти или шести основных динамиках объемного звука (в зависимости от конфигурации 5.1 или 7.1). *Вектор относительной скорости* между источником и слушателем нужен для применения *эффекта Доплера*. И последнее, но не менее важное: *модель окружающего пространства* используется для симуляции акустики с учетом частичного или полного блокирования пути распространения звука.

14.4.3. Затухание в зависимости от расстояния

Затухание в зависимости от расстояния уменьшает громкость 3D-звука по мере его радиального отдаления от слушателя.

Минимальное и максимальное расстояние затухания

В игровом мире, как правило, очень много источников звука. Но, учитывая ограничения, связанные с оборудованием и пропускной способностью процессора, мы попросту не в состоянии сгенерировать их все. Да это и не нужно, так как слушатель все равно не может услышать то, что происходит за пределами определенного радиуса от него. По этой причине каждый источник звука имеет параметры затухания (fall-off, FO).

FO min — это минимальный радиус (обозначим его r_{min}), в пределах которого звук совершенно не затухает и его можно слышать на полной громкости. FO max — это максимальный радиус (обозначим его r_{max}), за пределами которого источник звука считается неслышным и, следовательно, может быть проигнорирован. Между FO min и FO max должен осуществляться плавный переход от полной громкости к нулевой.

Переход к нулю

Плавный переход от максимальной громкости к нулевой можно выполнять линейно. Для некоторых типов звуков линейное затухание вполне подходит.

Как мы узнали из подраздела 14.1.3, интенсивность звука, тесно связанная с нашим восприятием громкости, уменьшается с увеличением радиального расстояния по принципу $1 / r^2$. Коэффициент передачи, пропорциональный амплитуде

давления звука, снижается в соответствии с правилом $1 / r$. Поэтому для перехода от полной громкости к нулевой правильнее всего использовать кривую $1 / r$.

У функции $1 / r$ есть одна проблема: ввиду своей асимптотности она никогда не достигает нуля, независимо от того, насколько велико значение r. Чтобы это исправить, мы можем сместить кривую немного вниз, чтобы она пересекала ось R в точке r_{max}. Или же просто обнулять интенсивность звука для всех $r > r_{max}$.

Нарушая правила

Работая над *The Last of Us*, звукоинженеры Naughty Dog обнаружили, что при затухании громкости диалогов персонажей по принципу $1 / r^2$ речь очень быстро становилась неразборчивой даже при скромной удаленности. Это создавало серьезные проблемы, особенно на тех участках игры, которые нужно было преодолеть незамеченным и на которых разговоры между вражескими персонажами служили не только средством развития сюжетной линии, но и тактическим элементом.

Для решения этой проблемы звукоинженеры использовали сложную кривую затухания, благодаря которой понижение громкости диалогов происходило более медленно, с точки зрения слушателя, ускорялось на средних дистанциях и затем снова замедлялось, когда расстояние от слушателя становилось очень большим. Таким образом, речь оставалась разборчивой на бóльших расстояниях, но при этом сохранялось затухание, кажущееся естественным.

Кроме того, кривая затухания диалогов динамически регулировалась во время выполнения в зависимости от текущего уровня напряженности в игре, то есть от того, знают ли враги о присутствии игрока, ищут ли они его и вовлечены ли в прямое противостояние с ним. Благодаря этому голоса в *The Last of Us* распространяются на бóльшие расстояния в режиме скрытности, но не становятся слишком громкими во время боя.

Наконец, чтобы подсластить результат, можно добавить реверберацию. Это позволит голосам персонажей огибать углы, даже если прямой путь полностью блокируется. Это невероятно полезно в ситуациях, в которых четкая слышимость разговоров важнее моделирования реалистичного затухания.

При проектировании модели 3D-звука можно использовать множество уловок. Но что бы вы ни делали, всегда помните, что *требования к игре* стоят на первом месте и для их удовлетворения все средства хороши. Не волнуйтесь, законы физики не обидятся!

Атмосферное затухание

Как сказано в подразделе 14.1.3, низкочастотные звуки затухают в атмосфере не так сильно, как высокочастотные. В некоторых играх, в том числе *The Last of Us* от Naughty Dog, для имитации этого явления используется низкочастотный фильтр. Он применяется к каждому 3D-звуку, полоса пропускания которого опускается к все более низким частотам по мере увеличения расстояния между источником звука и слушателем.

14.4.4. Панорамирование

Панорамирование — это методика, создающая иллюзию того, что 3D-звук поступает с определенного направления. Регулируя громкость (то есть коэффициент передачи) звука в каждом доступном динамике, мы можем создать *иллюзорную картину* его распространения в трехмерном пространстве. Этот метод панорамирования называется *амплитудным*, так как для предоставления слушателю угловой информации мы регулируем только амплитуду звуковых волн, генерируемых отдельными динамиками, вместо использования фазовых сдвигов, реверберации или фильтрации для формирования позиционной картины. Иногда этот подход называют *IID-панорамированием*, потому что он основан на эффектах восприятия интерауральной разности интенсивности (interaural intensity difference, IID).

Термин «панорамирование» происходит от старой технологии, в которой для управления относительной громкостью левого и правого динамика в стереосистеме использовался панорамный потенциометр — резистор переменного сопротивления. Если перевести потенциометр в одно из крайних положений, звук будет слышен только в одном динамике — левом или правом, если оставить его в центральном положении, звук будет распределяться равномерно между ними.

Чтобы понять, как это работает, представьте, что вы находитесь в центре круга, по контуру которого в разных точках размещены динамики. Радиус круга равен примерному расстоянию между отдельным динамиком и слушателем.

Если речь идет о стереосистеме, динамики находятся по обе стороны от центра под углом ±45°. Стереонаушники размещаются под углом ±90° (при этом радиус намного меньше). В объемном звуке вида 7.1 учитываются только семь основных динамиков (канал LFE не предоставляет никакой позиционной информации), которые размещаются примерно так, как показано на рис. 14.29. В системе 5.1 мы просто опускаем окружной тыловой левый и окружной тыловой правый[1] динамики.

Пока что будем считать каждый 3D-звук точечным источником. Чтобы панорамировать звук, нужно сначала определить *азимутальный* (горизонтальный) угол. Его следует измерять в локальном пространстве слушателя, чтобы нулевое значение соответствовало позиции прямо перед ним. Дальше нужно определить два динамика, которые находятся на нашем круге *по соседству* с этим азимутальным углом. Мы преобразуем угол в процентное отношение длины дуги между двумя динамиками к длине всего круга. Затем используем это относительное значение для определения коэффициента передачи звука в каждом динамике.

Чтобы выразить это математически, обозначим азимутальный угол звука θ_s. Углы двух прилегающих динамиков будут называться θ_1 и θ_2. Относительное панорамное направление β можно вычислить так:

$$\beta = \frac{\theta_s - \theta_1}{\theta_2 - \theta_1}.$$

[1] См. https://ru.wikipedia.org/wiki/Объёмный_звук#Система_обозначений_каналов_звука_(Формат_громкоговорителей). — *Примеч. пер.*

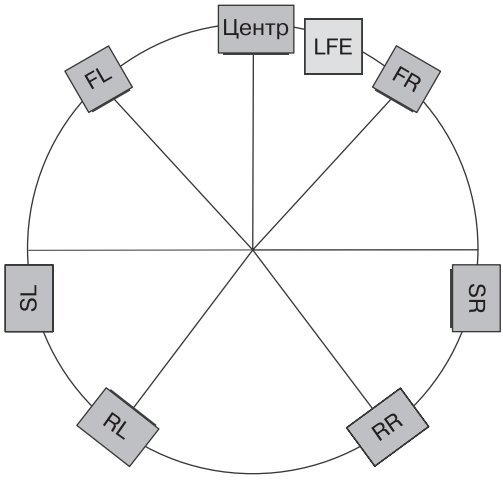


Рис. 14.29. Размещение динамиков при панорамировании системы 7.1

Это уравнение проиллюстрировано на рис. 14.30.

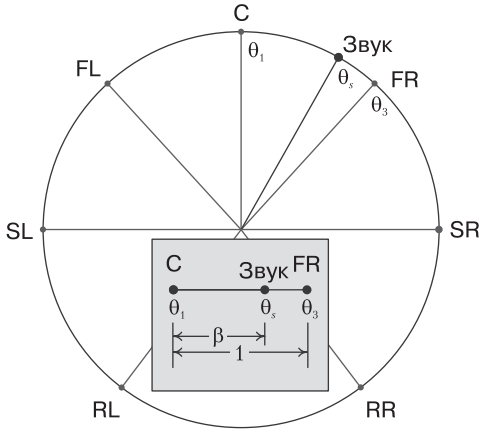


Рис. 14.30. Если обращаться со звуком как с точечным источником, относительное панорамное направление β вычисляется между двумя динамиками, находящимися в непосредственной близости к источнику

Постоянное панорамное усиление

Очевидным решением является использование значения β для выполнения простой линейной интерполяции между коэффициентами передачи двух динамиков. Если исходное усиление звука (до панорамирования) равно A, коэффициенты его передачи в каждом динамике можно вычислить так:

$$A_1 = (1-\beta)A;\ A_2 = \beta A.$$

Это называют *постоянным панорамным усилением*, поскольку совокупное усиление $A = A_1 + A_2$ остается неизменным и не зависит от значений θ_s и β.

Основная проблема этого подхода в том, что он не дает ощущения постоянной *громкости* при перемещении источника звука по акустическому полю. Коэффициент передачи определяет амплитуду волны давления звука, поэтому от него зависит *уровень звукового давления* (УЗД). Но, как мы уже знаем из подраздела 14.1.2, в человеческом восприятии громкость пропорциональна *интенсивности* или *мощности* звуковой волны, которые равны УЗД *в квадрате*.

Чтобы проиллюстрировать эту проблему, представим, что звук исходит из точки, равноудаленной от двух динамиков. В соответствии с постоянным панорамным усилением коэффициенты A_1 и A_2 должны быть равны $\frac{1}{2}A$. Но в таком случае общая мощность получается $A_2^2 = \left(\frac{1}{2}A\right)^2 + \left(\frac{1}{2}A\right)^2 = \frac{1}{2}A^2$. Иными словами, громкость звука будет *в два раза меньшей* по сравнению с ситуацией, когда источник совпадает с правым или левым динамиком.

Закон постоянной панорамной мощности

Очевидно, что для поддержания постоянной субъективной громкости звука при перемещении источника относительно слушателя необходимо обеспечить постоянную *мощность*. Этот принцип называется *законом постоянной панорамной мощности*.

Данный закон можно реализовать очень простым способом. Вместо линейной интерполяции коэффициентов передачи мы вычисляем их в виде синуса и косинуса относительного усиления β:

$$A_1 = \sin\left(\frac{\pi}{2}\beta\right)A;\ A_2 = \cos\left(\frac{\pi}{2}\beta\right)A.$$

Еще раз представим, что источник звука находится на равном расстоянии от двух динамиков ($\beta = \frac{1}{2}$). В соответствии с постоянной панорамной мощностью коэффициенты передачи двух динамиков будут $A_1 = A_2 = \frac{1}{\sqrt{2}}A$. Следовательно, общая мощность $A_1^2 + A_2^2 = \left(\frac{1}{\sqrt{2}}A\right)^2 + \left(\frac{1}{\sqrt{2}}A\right)^2 = A^2$. Это справедливо для любых значений β, поэтому мощность A^2 остается неизменной независимо от того, в какой точке круга находится источник звука.

Для соблюдения закона постоянной панорамной мощности звукоинженеры часто руководствуются «правилом 3 децибел»: если звук нужно равномерно распределить между двумя динамиками, громкость каждого нужно уменьшить на 3 дБ по сравнению с тем, как если бы звук воспроизводился только одним из них. Значение –3 дБ вытекает из $\log_{10}\left(\frac{1}{\sqrt{2}}\right) \approx -0{,}15$. Усиление напряжения (или амплитуды) определяется как $20 \log_{10} (A_{out}/A_{in})$, поэтому $20 \cdot (-0{,}15) = -3$ дБ (число 20 перед

логарифмом определяется тем, что 1 децибел, равный 1/10 бела, умножается на 2, так как мы имеем дело с A^2, а не с A).

Пространство для маневра

В результате панорамирования звуки в некоторых ситуациях полностью генерируются одним или двумя (или тремя и больше) динамиками. Представим, что два смежных динамика издают очень громкий звук с одинаковой максимальной мощностью. Что произойдет, если он будет исходить только из одного динамика? Вероятно, тот попросту выйдет из строя, так как закон постоянной панорамной мощности требует превысить его максимальную громкость в два раза.

Чтобы избежать этой проблемы, мы должны искусственно занижать максимальные коэффициенты передачи всех звуков, так что при худшем варианте развития событий динамик, воспроизводящий звук, не будет перегружен. Искусственно занижая максимальную громкость, мы оставляем себе некоторое *пространство для маневра*.

Этот подход применим и к *микшированию*. Когда микшируются два и больше звука, их амплитуды суммируются. Можно оставить некоторый зазор на крайний случай, когда одновременно проигрывается большое количество громких звуков.

Центрировать или не центрировать?

Исторически сложилось так, что центральный канал в кинотеатрах используется для речи, панорамирование между разными динамиками выполняется только для звуковых эффектов. Суть этого приема в том, что персонажи, которые разговаривают, обычно находятся в кадре, поэтому зрители ожидают, что их голоса будут раздаваться спереди и по центру. Удобным побочным эффектом этого подхода является отделение речи от остальных звуков в фильме, благодаря чему громкие звуковые эффекты не могут занять всю резервную мощность и заглушить диалог.

В трехмерных играх совсем другая ситуация. Игрок обычно хочет, чтобы речь доносилась из правильного места. Если повернуть камеру на 180°, диалог должен повернуться вокруг звукового поля на тот же угол. В связи с этим речь в играх обычно не привязывается к центральному динамику, а участвует в панорамировании вместе со звуковыми эффектами.

Конечно, это возвращает нас к проблеме с пространством для маневра: громкие выстрелы могут полностью заглушить разговор. Мы в Naughty Dog боремся с этим, пытаясь найти золотую середину: диалог всегда отчасти воспроизводится в центральном канале, а отчасти распределяется между остальными динамиками вместе со звуковыми эффектами.

Фокусирование

Когда источник звука находится далеко от слушателя, можно считать его точечным. Мы просто вычисляем единый азимутальный угол и подаем его на вход системы постоянного панорамного усиления. Но когда источник звука приближается или даже проникает в круг, определяющий радиальное расстояние от динамиков до

слушателя, его больше нельзя моделировать в виде точки, представленной одним углом.

Представьте, что мы движемся по направлению к источнику звука и затем минуем его. Сначала его слышно лишь из фронтальных динамиков, а затем он должен как-то перейти к тыловым. Если звук смоделирован в виде точечного источника, нам не остается ничего иного, кроме как мгновенно переключить его с передних каналов на задние.

В идеале звук, как мы его себе представляем, должен постепенно разворачиваться по панорамному кругу по мере приближения. Таким образом, чем ближе он к слушателю, тем большую его часть можно воспроизводить из боковых динамиков. Когда слушатель и источник находятся в одной позиции, звук можно распределить по всем семи (пяти) каналам. Затем, пройдя мимо, мы можем плавно сместить его к тыловым динамикам и уменьшить коэффициент передачи, оставляя источник позади.

Эти и другие приемы становятся возможными, если источник звука моделируется не как точка, а как *дуга* на панорамном круге. Представьте себе, что каждый источник звука имеет произвольную форму в трехмерном пространстве, но его *проекция* на панорамный круг формирует *клин* под определенным углом. Это аналогично концепции *телесного угла*, которую часто используют для обсчета объемного света в трехмерной графике (подробнее об этом рассказывается на странице ru.wikipedia.org/wiki/Телесный_угол).

Назовем угол, образуемый спроецированным источником звука, *фокусным* и обозначим его α. Точечный источник можно считать крайним случаем, в котором $\alpha = 0$. Фокусный угол изображен на рис. 14.31.

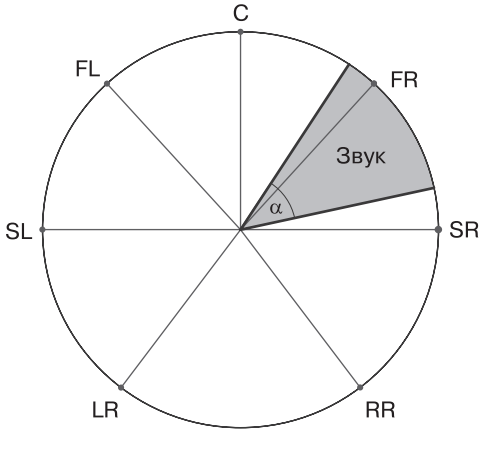


Рис. 14.31. Фокусный угол определяет проекцию расширенного источника звука на окружность динамиков

Чтобы сгенерировать звук с ненулевым фокусным углом, нужно сначала определить подмножество динамиков, которые либо пересекают его дугу, спроецированную на панорамный круг, либо прилегают к ней. Затем интенсивность/мощность

звука следует распределить между этими динамиками, чтобы создать иллюзию источника звука, растягивающегося по спроецированной дуге.

Существует несколько способов распределения звука между подходящими динамиками. Например, мы можем сделать так, чтобы все динамики *внутри фокусного* клина имели одинаковую максимальную мощность, а те, что прилегают к дуге, были немного приглушены, — так можно создать эффект затухания. Но какой бы метод мы ни выбрали, всегда необходимо соблюдать закон постоянной панорамной мощности. Поэтому коэффициенты передачи должны быть заданы таким образом, чтобы сумма их квадратов (то есть их суммарная мощность) была равна квадрату коэффициента передачи исходного источника звука до панорамирования.

Вертикальная плоскость

В конфигурациях со стерео- и объемным звуком динамики расположены примерно на одном уровне в горизонтальной плоскости. Это делает непростой задачу позиционирования звуков сверху и снизу относительно ушей слушателя.

В идеале, конечно, следовало бы смоделировать настоящее многоканальное (periphonic) звуковое поле, размещая динамики сферически. Технология под названием *Ambisonics* (en.wikipedia.org/wiki/Ambisonics) обеспечивает как плоскостную, так и сферическую конфигурацию динамиков. Но она не поддерживается ни одной игровой консолью, по крайней мере пока. Компания Sony в настоящее время предоставляет технологию 3D-звука в своей «платиновой» беспроводной гарнитуре для PS4, и ее поддержка начинает появляться в играх. Но даже при наличии этой технологии игры все равно должны уметь работать со звуковыми системами типа 5.1 и 7.1.

Оказывается, придать звуковым образам некоторую степень вертикальности можно с помощью концепции *фокуса*. Для этого все звуки *проецируются* на горизонтальную плоскость, а затем для тех из них, проекции которых находятся слишком близко к панорамному кругу или внутри него, применяется ненулевой фокусный угол. Если источник звука находится сверху от слушателя, но удален от него, он будет сгенерирован практически так же, как и любой другой звук. Но если звук проходит над головой, мы распределяем его между несколькими динамиками, создавая иллюзорный образ внутри панорамного круга. Если прибавить к этому затухание в зависимости от расстояния и атмосферное поглощение, зависящее от частоты, мы можем предоставить слушателю достаточно информации, чтобы ему казалось, что звук исходит сверху или снизу.

Дополнительный материал по панорамированию

Основы закона постоянной панорамной мощности можно найти здесь: www.rs-met.com/documents/tutorials/PanRules.pdf. Следующий сайт тоже является отличным источником информации по данной теме: www.music.miami.edu/programs/mue/Research/jwest/Chap_3/Chap_3_IID_Based_Panning_Methods.html.

Докторская диссертация Вилле Пукки из Хельсинкского политехнического университета, доступная по адресу aaltodoc.aalto.fi/bitstream/handle/123456789/2345/

isbn9512255324.pdf?sequence=1, содержит четкое описание проблемы спатиализации, рассматривает метод векторного амплитудного панорамирования (vector based amplitude panning, VBAP) и предлагает обширный список дополнительной литературы.

Стоит также почитать работу Дэвида Гризингера *Stereo and Surround Panning in Practice*, доступную по адресу www.davidgriesinger.com/pan_laws.pdf. На сайте Дэвида можно найти огромное количество исследований о восприятии звука и технологиях воспроизведения аудио.

14.4.5. Распространение, реверберация и акустика

Даже если бы мы реализовали затухание в зависимости от расстояния, панорамирование и эффект Доплера, наш движок 3D-звука все равно не смог бы сгенерировать реалистичный звуковой ландшафт. Дело в том, что многие слуховые сигналы, с помощью которых мы, люди, можем представить себе окружающее пространство, создаются за счет таких эффектов, как ранние отражения, поздняя реверберация и передаточная функция головы, возникающих из-за того, что звуковые волны доходят до наших ушей разными путями. Любую методику, которая учитывает, каким образом звуковая волна распространяется в пространстве, можно отнести к *моделированию распространения звука*.

В научных исследованиях и в сфере интерактивных медиа и игр используется множество разных подходов. Все эти технологии можно разделить на три основные категории.

- *Геометрический анализ* — это попытка смоделировать пути, по которым проходят звуковые волны.
- *Модели на основе восприятия* предназначены для воспроизведения того, что воспринимает ухо, с помощью ЛСС-модели акустики окружающего пространства.
- *Специальные методы* используются для вычисления различных приближенных значений для создания достаточно точной акустической модели с минимальными требованиями к размеру памяти и/или вычислительной мощности.

Следующая исследовательская работа содержит хороший обзор методов, входящих в первую категорию: www-sop.inria.fr/reves/Nicolas.Tsingos/publis/presence03.pdf. В этом разделе мы кратко обсудим моделирование систем ЛСС и подробнее рассмотрим несколько специальных методов, которые обычно оказываются более пригодными для применения в реальных играх.

Моделирование эффектов распространения с помощью системы ЛСС

Представьте, что я стою в комнате с различными предметами, выполненными из разных материалов. Вдруг возникает звук. Он отражается, преломляется, сталкивается с препятствиями и в итоге достигает моих ушей. Если подумать, траектория распространения звуковых волн не так уж и важна. Единственное, что влияет

на мое восприятие, — это конкретная суперпозиция сухих направленных звуковых волн, а также разного рода запоздалые, приглушенные или каким-то иным образом искаженные непрямые «мокрые» волны.

Оказывается, все эти эффекты можно смоделировать с помощью линейной стационарной системы (ЛСС). Теоретически, если нам известна *импульсная характеристика* комнаты для двух заданных точек, представляющих источник звука и слушателя, мы можем в точности определить, как *любой* звук, который воспроизводится в месте расположения источника, должен звучать в месте нахождения слушателя. Для этого достаточно выполнить свертку сухого звука и импульсной характеристики:

$$p_{\text{wet}}(t) = p_{\text{dry}}(t) * h(t).$$

На первый взгляд этот подход выглядит как панацея от всех проблем. Но в действительности он совсем не такой простой и практичный, как может показаться. Определить импульсную характеристику пространства в реальном времени довольно легко: вы можете сгенерировать короткий щелчок, имитирующий единичный импульс $\delta(t)$, и записанный сигнал будет приблизительно описывать $h(t)$. Но, чтобы определить $h(t)$ в виртуальном пространстве, приходится выполнять сложную и требовательную к ресурсам симуляцию для каждого игрового пространства. Кроме того, чтобы как следует смоделировать акустику комнаты, эти вычисления следует выполнить для большого количества пар «источник — слушатель», разбросанных по всему игровому миру. Размер полученных данных будет огромным. И наконец, саму операцию свертки тоже нельзя назвать легковесной — в прошлом игровым консолям и звуковым картам не хватало мощности, чтобы выполнять ее для каждого звука в игре в режиме реального времени.

В наши дни игровое оборудование становится все мощнее и подход к моделированию распространения звука на основе свертки имеет все больше смысла. Например, Мицах Тейлор с коллегами создал демонстрацию реверберации со сверткой в реальном времени, показывающую обнадеживающие результаты (см. intel.ly/2J8Gpsu). Несмотря на это в большинстве игр данный подход попрежнему не используется. Наиболее распространенными все еще остаются специальные методы и приближенные вычисления модели реверберации в окружающем пространстве.

Области реверберации

Один из популярных подходов к моделированию «мокрых» характеристик пространства для воспроизведения звука состоит в ручном разбиении игрового мира на области, каждой из которых назначаются подходящие параметры реверберации, такие как предварительная задержка, затухание, плотность и диффузия (подробнее об этом говорится в подразделе 14.1.3). При перемещении слушателя по этим областям мы можем включать подходящие режимы реверберации: если игрок заходит

в просторную комнату, отделанную плиткой, можем усилить эхо; когда он находится в небольшом чулане — устранить практически все эффекты реверберации, чтобы получить очень сухой звук.

Плавный переход между параметрами реверберации по мере движения слушателя по игровому пространству — удачное решение. Для изменения каждого параметра можно использовать линейную интерполяцию. Коэффициент смешивания лучше всего привязать к тому, как далеко вглубь области зашел слушатель. Представьте, к примеру, перемещение между улицей и помещением через дверной проем. Мы можем создать область вокруг дверного проема, внутри которой происходит смешивание. Если слушатель находится за пределами этой области, параметры реверберации улицы должны иметь коэффициент смешивания 100 %, а аналогичные параметры помещения — 0 %. Если слушатель находится посреди области смешивания, параметры должны распределяться поровну. Как только слушатель выйдет за пределы области, находясь в здании, соотношение параметров улицы и помещения будет 0/100 (рис. 14.32).

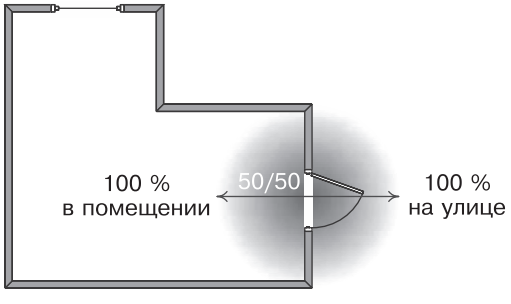


Рис. 14.32. Плавный переход между параметрами реверберации в зависимости от положения слушателя — хорошая идея

Частичное и полное блокирование звука

При использовании областей для определения акустики игровых пространств мы обычно назначаем каждой из них *одну* функцию импульсной характеристики или *один* набор параметров реверберации. Это позволяет описать основные свойства каждого игрового пространства, например большого зала, отделанного кафелем, маленького стенного шкафа с верхней одеждой, плоской равниной на открытом воздухе и т. д. Но при этом воспроизведение акустики, обусловленной разными препятствиями, получается не совсем идеальным. Представьте, к примеру, квадратную комнату с массивной колонной по центру. Если источник звука находится в одном из углов, тембр, который мы будем слышать при перемещении по комнате, будет сильно варьироваться в зависимости от того, перекрывает ли колонна прямой путь распространения звука. Если применить для такой комнаты лишь один набор параметров реверберации, эти тонкости передать не удастся.

В качестве решения можно попытаться как-то смоделировать геометрию и свойства материалов окружающего пространства, определить, как препятствия влияют на звуковые волны, и затем использовать результаты этого анализа для изменения базовых параметров реверберации, связанных с помещением.

На рис. 14.33 показаны три варианта воздействия объектов и поверхностей игрового мира на передачу звуковых волн.

- *Полное блокирование*. В этой ситуации звук не может свободно доходить от источника к слушателю. Мы по-прежнему можем слышать полностью заблокированный звук, если, к примеру, на его пути находится лишь тонкая стена или дверь. С точки зрения слушателя происходит одно из двух: либо звук полностью отсутствует, либо его сухие и «мокрые» компоненты затухают и/или приглушаются.
- *Частичное блокирование*. В этой ситуации блокируется только прямой путь между источником и слушателем, но звук все еще способен распространяться окольными путями. Частичное блокирование может возникнуть, к примеру, когда источник звука находится за автомобилем, колонной или другим препятствием. Сухой компонент частично заблокированного звука либо полностью отсутствует, либо существенно приглушен, «мокрый» компонент может быть искажен из-за того, что проходит более длинный путь с отражениями.
- *Ограниченное распространение*. В этой ситуации между источником и слушателем есть прямой путь, но окольные пути блокируются тем или иным способом. Это может произойти, когда звук, сгенерированный в одной комнате, проходит через узкий проем, такой как дверь или окно, и достигает слушателя. Сухой компонент звука сохраняет первозданный вид, а «мокрый» затухает, глушится или, если проем очень узкий, полностью отсутствует.

Анализируем прямой путь. Определить, заблокирован прямой путь или нет, несложно. Мы просто отбрасываем луч (см. подраздел 13.3.7) от слушателя к каждому источнику звука. Если луч проходит, путь свободен. Если нет — звук заблокирован.

С помощью рейкастинга можно также смоделировать прохождение звука *через* стены и другие препятствия. Мы отбрасываем луч от источника к слушателю и запрашиваем свойства материалов тех поверхностей, в которые уперлись, чтобы определить, какая часть звуковой энергии поглощается. Если энергия частично проходит дальше, можем отбросить еще один луч из точки на другой стороне препятствия и продолжить анализ пути к слушателю. Если вся энергия звука была поглощена, мы можем заключить, что его не слышно. Но если луч доходит до слушателя, не потеряв при этом всю энергию без остатка, можем ослабить коэффициент передачи сухого компонента на соответствующую величину, чтобы симулировать передачу звука.

Анализ окольного пути. Определение того, блокируется ли окольный путь, — более сложная задача. В идеале можно было бы выполнить какого-то рода поиск (возможно, A*), чтобы узнать, существует ли путь между источником и слушателем, насколько он ослабляет и отражает звук. На практике же из-за высокой на-

грузки на процессор и память такая *трассировка пути* используется редко. В конце концов, мы как игровые разработчики не очень-то заинтересованы в создании точных физических моделей, достойных Нобелевской премии. Мы всего лишь хотим создать *правдоподобный* звуковой ландшафт с *эффектом присутствия*.

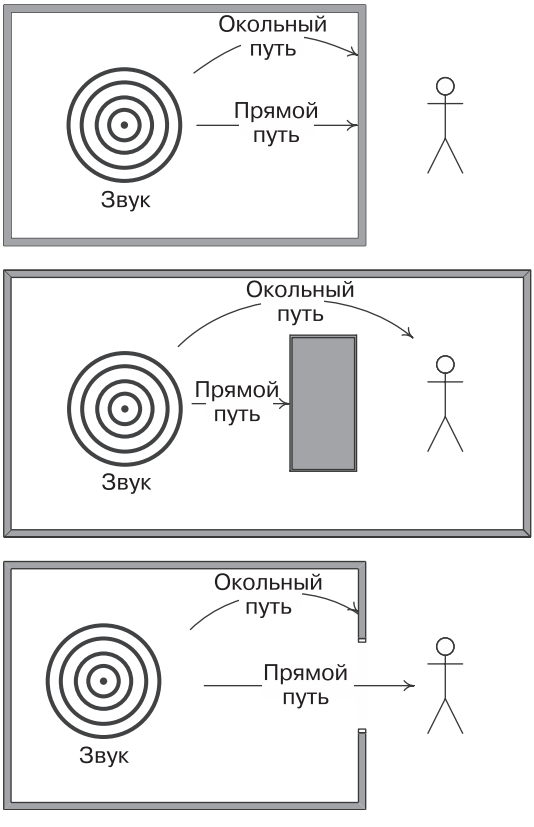


Рис. 14.33. *Сверху вниз*: полное блокирование, частичное блокирование, ограниченное распространение

Но не отчаивайтесь. Существует множество способов получения *приближенной* модели окольного пути звука. Например, если мы задействуем *области реверберации* для моделирования общей акустики в различных пространствах игры (см. предыдущий подраздел), с их помощью можно определить наличие окольного пути. Например, можно руководствоваться простыми эмпирическими правилами.

1. Если источник и слушатель находятся в одной и той же области, будем считать, что окольный путь существует.
2. Если источник и слушатель находятся в разных областях, будем считать, что окольный путь заблокирован.

Если прибавить к этим допущениям результаты рейкастинга прямого пути, можно получить четыре отдельных сценария: свободное распространение, полное блокирование, частичное блокирование, ограниченное распространение.

Учет дифракции. Любая волна, проходящая через узкий проем или затрагивающая какой-то угол, расширяется (рис. 14.34). Это явление называется *дифракцией*. Благодаря ему звук можно слышать из-за углов как при наличии прямого пути, главное, чтобы угловая разность между прямым и изогнутым путями не была слишком большой.

Чтобы определить, может ли звук преломиться и достичь слушателя, вокруг центрального прямого луча можно отбросить несколько изогнутых. Большинство движков столкновений не поддерживают трассировку изогнутых путей, но мы можем эмулировать эту возможность, отбрасывая сразу несколько прямых лучей. На рис. 14.35 показан простой пример, в котором от источника звука к слушателю отбрасываются пять лучей: один для прямого пути и еще по два — для двух изогнутых. Строго говоря, к каждому изогнутому пути, который мы желаем трассировать, применяется *кусочно-линейное приближение*.

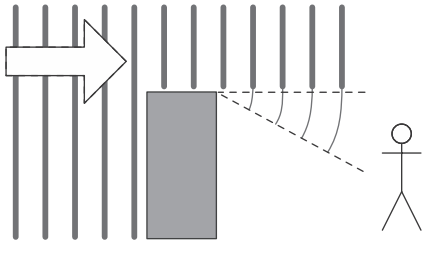


Рис. 14.34. Благодаря дифракции сухой компонент звука четко слышен даже в случае блокирования прямого пути

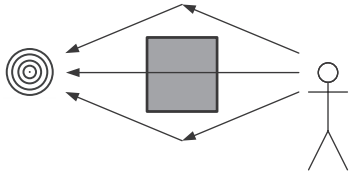


Рис. 14.35. Изогнутые лучи можно эмулировать отбрасыванием нескольких прямых лучей

Если прямой луч блокируется, а изогнутый доходит до слушателя, это означает, что слушатель находится в области дифракции за близлежащим углом и должен слышать звук.

Применение модели с использованием реверберации и коэффициента передачи. До сих пор мы обсуждали определение того, блокируются или нет прямые и окольные пути. Этот анализ может дать представление также об акустическом влиянии полного и частичного блокирования (например, звук, проходящий через стену, можно приглушить, звук, проходящий длинный путь с отражениями, способен создать сильную реверберацию). Возникает вопрос: как задействовать эту информацию при рендеринге звука?

Один из простых подходов состоит в ослаблении сухого и «мокрого» компонентов звука по отдельности в зависимости от того, как блокируются прямой и окольный пути, полностью или частично. Чтобы получить более точные результаты, к каждому компоненту звука можно применить разную степень реверберации с учетом эвристических данных, собранных при определении пути (или путей)

распространения звуковой волны. У каждой игры свои требования, и это один из тех случаев, когда наиболее подходящий вариант определяется методом проб и ошибок!

Смешивание частично заблокированных звуков. Если вам вдруг вздумается реализовать все, о чем мы говорили в предыдущих разделах, вы заметите одну вопиющую проблему. По мере того как источник звука переходит в каждое из четырех состояний, описанных ранее (например, от свободного распространения к частичному блокированию), его тембр и громкость будут резко меняться. Существует множество способов смягчения переходов. Вы можете, к примеру, применять *гистерезис*, то есть замедлять реакцию звуковой системы на изменения каждого звука в частично заблокированном состоянии, и затем использовать эту короткую задержку для плавного перехода между параметрами реверберации. Однако задержка может быть заметной, поэтому данное решение нельзя назвать идеальным.

Для игр из циклов *Uncharted* и *The Last of Us* старший звукоинженер Naughty Dog Джонатан Ланье изобрел патентованную систему, которую назвал *моделированием стохастического распространения*. Чтобы не выдать секреты фирмы, могу лишь сказать, что в этой системе к каждому источнику звука отбрасывается множество лучей, прямых и непрямых, и результаты попаданий/непопаданий накапливаются на протяжении многих кадров. На основе этих данных можно сгенерировать вероятностную модель степени блокирования сухого и «мокрого» компонентов каждого источника звука. Это позволяет плавно переключать звук из полностью заблокированного в полностью свободное состояние без заметных рывков.

Звуковые порталы в The Last of Us

В ходе работы над игрой *The Last of Us* программистам Naughty Dog нужно было моделировать то, как звук на самом деле распространяется в пространстве. Если вражеский NPC разговаривает, находясь в длинном коридоре, соединенном с комнатой, в которой находится игрок, мы хотим, чтобы его голос было слышно из *дверного проема*, а не через стену вдоль прямого пути.

Чтобы этого добиться, мы задействовали сеть взаимосвязанных областей. Области делились на две категории: *комнаты* и *порталы*. Для поиска пути от каждого источника звука к слушателю мы использовали информацию и связанности, которые звукоинженер указал при компоновке областей. Если источник и слушатель находились в одной комнате, мы задействовали проверенный метод с анализом степени блокирования (частичное, полное или ограниченное распространение), который применялся в цикле *Uncharted*. Но если источник и слушатель находились в разных комнатах, напрямую соединенных порталом, звук воспроизводился так, *будто* его источник размещался в самом портале. Мы обнаружили, что во всех реальных сценариях, возникавших в игре, достаточно было лишь одного перехода по графу связанности комнат. Конечно, я опускаю множество важных подробностей, но общий принцип работы этой системы проиллюстрирован на рис. 14.36.

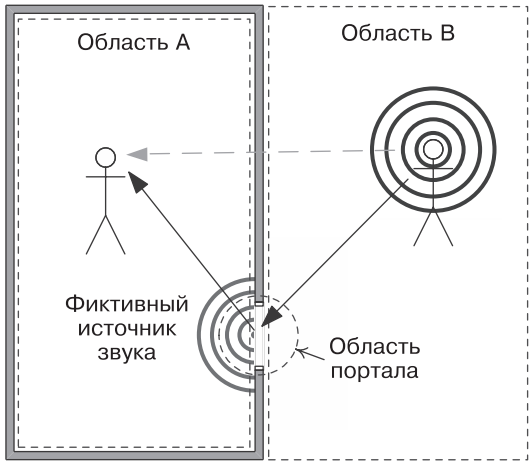


Рис. 14.36. Модель распространения звука на основе порталов, которую студия Naughty Dog, Inc. задействовала в The Last of Us

Дополнительный материал по акустике окружающего пространства

Моделирование распространения звука и акустический анализ — это области, в которых ведутся активные исследования. С улучшением возможностей аппаратного обеспечения в игровой индустрии применяются все более продвинутые методики. Далее перечислены несколько ссылок, которые должны подогреть ваш аппетит. Но если вы хотите растянуть удовольствие на много часов, можете просто поискать в Google «распространение звука» и «акустическое моделирование»!

- *Real-Time Sound Propagation in Video Games*, Жан-Франсуа Гэ из Ubisoft Montreal (bit.ly/2HdBiLc).
- *Modern Audio Technologies in Games*, презентация А. Меньшикова на GDC 2003 (bit.ly/2J7FYyD).
- *3D Sound in Games*, Джейк Симпсон (bit.ly/2HfVFTU).

14.4.6. Эффект Доплера

Как говорилось в подразделе 14.1.3, эффект Доплера — это изменение частоты в зависимости от *вектора относительной скорости* между источником и слушателем: $\mathbf{v}_{rel} = \mathbf{v}_{source} - \mathbf{v}_{listener}$. Чтобы приблизительно описать это изменение, мы можем просто масштабировать звуковой сигнал по времени. В результате получается так называемый эффект бурундука (chipmunk effect), с которым мы все знакомы благодаря группе «*Элвин и бурундуки*»: ускоренный звук становится более высо-

ким. Поскольку мы имеем дело с цифровыми сигналами (дискретного времени), масштабирование по времени можно выполнить за счет изменения частоты дискретизации (см. подраздел 14.5.4). Но это не совсем правильный способ, так как ускорение и замедление звука могут быть заметными.

В идеале сдвиг частоты не должен отражаться на временной оси. Этого можно добиться разными способами, включая такие методы, как *фазовый вокодер* и *TDHS* (time domain harmonic scaling). Их полное описание выходит за рамки данной книги, но вы можете узнать о них больше по адресу www.dspdimension.com/admin/time-pitch-overview.

Сдвиг частот независимо от времени — чрезвычайно мощная технология, которую может поддерживать звуковой движок. Ее полезность отчасти объясняется тем, что она позволяет выполнять масштабирование по времени без изменения частоты. Поэтому вы можете не только делать звуки более высокими или низкими, не изменяя временные параметры для эффекта Доплера, но и менять скорость их воспроизведения без влияния на тональность. Таким образом можно создавать всевозможные крутые эффекты.

14.5. Архитектура звукового движка

До сих пор мы обсуждали концепции и методики, стоящие за рендерингом 3D-звука, а также теории и технологии, на которых они основаны. В этом разделе сосредоточимся на архитектуре программных и аппаратных компонентов, которые используются для создания движков рендеринга объемного звука.

Как и большинство компьютерных систем, ПО для генерации звука в игровом движке состоит из набора слоев и программных компонентов (рис. 14.37).

- В основе этой структуры всегда лежит *аппаратное обеспечение*. Оно как минимум обеспечивает схему для управления цифровыми или аналоговыми звуковыми выходами, которая соединяют наш компьютер или игровую консоль с наушниками, телевизором либо системой домашнего кинотеатра с объемным звуком. Звуковое оборудование также может реализовать некоторые функции программного обеспечения, размещенного в верхней части стека, предоставляя аппаратные реализации кодеков, микшеров, пружинных реверберatorов, модулей звуковых эффектов, синтезаторов звуковых форм и/или DSP-чипы. Все вместе это часто называют *звуковой картой*, так как аудиовозможности ПК иногда обеспечиваются подключаемой периферийной картой.
- В персональных компьютерах оборудование обычно инкапсулируется в слое *драйверов*, что позволяет ОС поддерживать звуковые карты от разных производителей.
- И в ПК, и в консолях оборудование и драйверы обычно обернуты в низкоуровневый *прикладной программный интерфейс* (application programming interface,

API), благодаря которому программистам не нужно управлять оборудованием и драйверами напрямую.

- Поверх всего этого построен *движок 3D-звука*.

Рис. 14.37. Аппаратно-программный стек

Аппаратно-программный стек по своим возможностям обычно аналогичен *многоканальному микшерному пульту* (ru.wikipedia.org/wiki/Микшерный_пульт), который часто используют в студиях звукозаписи и живых концертах (рис. 14.38). Этот пульт может принимать на вход довольно большое количество аудиосигналов, поступающих от микрофонов и/или электронных инструментов. Можно фильтровать эти сигналы, пропускать их через эквалайзер, а также применять к ним реверберацию и другие эффекты. Звукоинженеры смешивают их и выставляют им подходящую громкость. Итоговый вывод подается на колонки (если это живое выступление) или записывается с разбиением на отдельные каналы.

Точно так же аппаратно-программный аудиостек должен принимать на вход большое количество 2D- и 3D-сигналов, обрабатывать их разными способами, объединять их, задавая подходящие коэффициенты передачи, и, наконец, панорамировать по исходящим каналам динамиков, чтобы игрок получил иллюзию трехмерного звукового ландшафта.

14.5.1. Конвейер обработки звуков

Как мы узнали из подраздела 14.4.1, процесс рендеринга 3D-звука состоит из нескольких шагов.

- Для каждого 3D-звука должен быть *синтезирован* сухой цифровой сигнал (в формате PCM).

Рис. 14.38. Многоканальный микшерный пульт от Focusrite с 72 входами и 48 выходами

- Затем применяется затухание в зависимости от расстояния, чтобы у слушателя было ощущение удаленности источника звука. Сигнал подвергают *реверберации*, чтобы смоделировать акустику виртуального пространства и предоставить слушателю пространственную информацию. В результате создается «мокрый» сигнал.
- «Мокрый» и сухой сигналы *панорамируются* (независимо) на один или несколько динамиков, чтобы создать итоговый образ каждого звука в трехмерном пространстве.
- Панорамированные многоканальные сигналы всех 3D-звуков *микшируются* в единый многоканальный сигнал, который, пройдя через линейку параллельных ЦА-преобразователей и усилителей, подается на аналоговые выходы для динамиков или напрямую на цифровые разъемы, такие как HDMI или S/PDIF.

Мы явно представляем себе процесс рендеринга 3D-звука в виде *конвейера*. И поскольку игровой мир обычно содержит большое количество источников звука, этот конвейер существует в нескольких экземплярах, работающих параллельно. В связи с этим его иногда называют *графом обработки звука*. Это действительно граф с взаимосвязанными компонентами, которые в конечном счете ведут к горстке аудиоканалов, составляющих итоговый микшированный панорамированный вывод. Этот граф в общем виде показан на рис. 14.39.

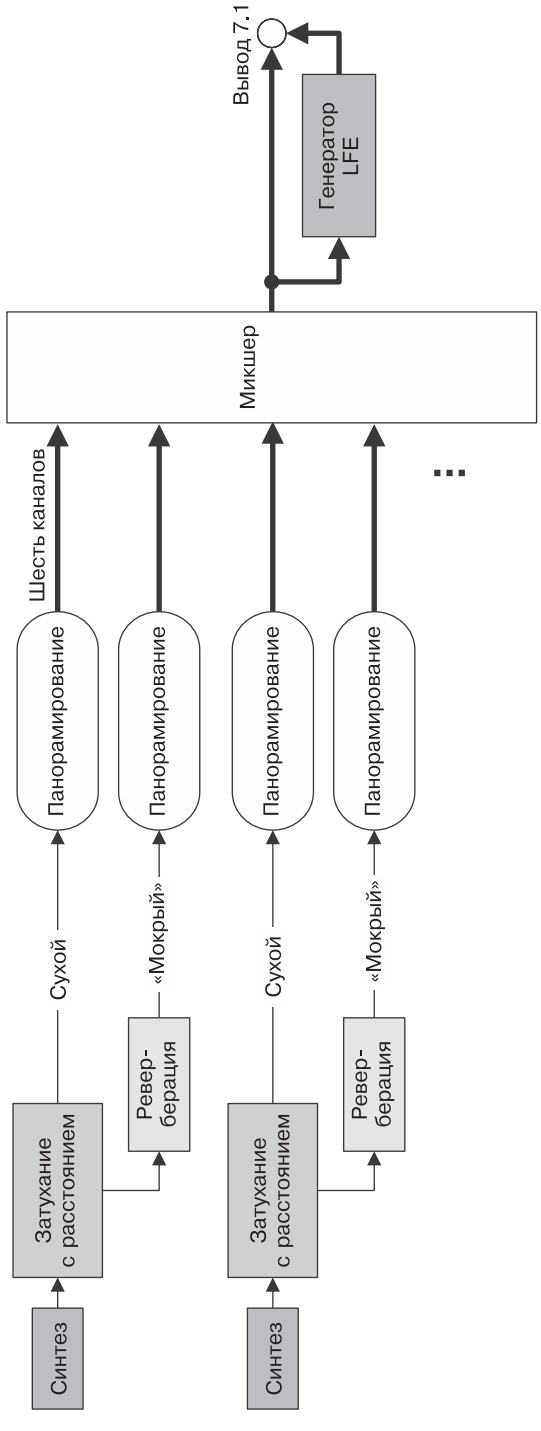


Рис. 14.39. Граф (конвейер) обработки звука

14.5.2. Концепции и терминология

Прежде чем углубляться в детали работы конвейера обработки звука, следует познакомиться с несколькими концепциями и терминами, которые используются для их описания.

Голоса

Каждый 2D- или 3D-сигнал, проходящий через граф рендеринга звука, называется *голосом*. Истоки этого термина — в ранней электронной музыке: синтезатор проигрывал музыкальные ноты посредством набора генераторов звуковых сигналов, которые назывались голосами.

Синтезатор содержит определенное количество цепей для генерации звуковых сигналов, поэтому в электронной музыке говорят о количестве одновременно генерируемых голосов. Точно так же движок рендеринга звука в игре обычно содержит ограниченное количество кодеков, модулей реверберации и т. д. Максимальное число голосов, поддерживаемое тем или иным аппаратно-программным стеком, зависит от того, сколько независимых параллельных дорожек проходит через звуковой граф. Это число в целом определяется объемом доступной памяти и ограничениями аппаратных и/или вычислительных ресурсов. Данный показатель иногда называют уровнем *полифонии* системы.

2D-голоса. Конвейер рендеринга звука в игре должен уметь работать с 2D-сигналами, такими как музыка, звуковые эффекты меню, закадровый голос и т. д. 2D-голоса обрабатываются тем же конвейером. Основное различие между 2D- и 3D-обработкой состоит в следующем.

- 2D-звуки изначально являются многоканальными, по одному каналу для каждого динамика, а 3D-звуки генерируются в виде сухих монофонических сигналов. В связи с этим 2D-звуки не проходят через панорамирование.
- В 2D-звук могут быть изначально встроены реверберация или другие эффекты. В таком случае ему не нужен модуль реверберации движка рендеринга.

Таким образом, 2D-звуки обычно подаются в конвейер непосредственно перед главным микшером, где они объединяются с 3D-звуками для получения итогового микса.

Шины

Взаимные соединения компонентов, из которых состоит звуковой граф, называют *шинами*. В электронике шина представляет собой цепь, основная задача которой состоит в объединении других цепей. В программном смысле шина — это не более чем логическая конструкция, описывающая наличие связей между составляющими.

14.5.3. Голосовая шина

На рис. 14.40 приведена более подробная схема конвейера компонентов, через который звуковой движок пропускает отдельный 3D-голос. В следующих пунктах мы подробно исследуем каждый из них и узнаем, почему они связаны между собой именно таким образом.

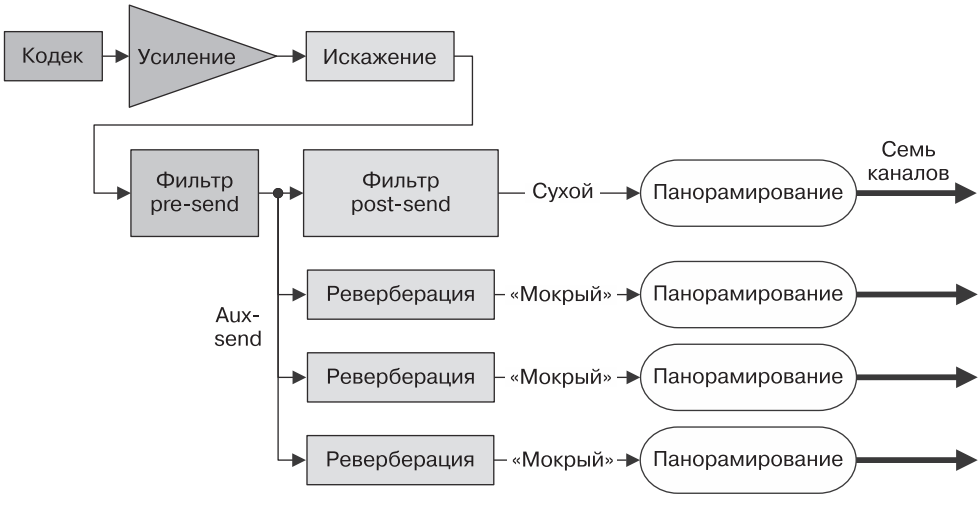


Рис. 14.40. Конвейер, через который проходит отдельный 3D-голос при движении по звуковому графу

Синтез звука: кодеки

Звуковой сигнал проходит через граф рендеринга в цифровом виде. Термин *«синтез»* обозначает процесс генерации таких цифровых сигналов. Иногда для этого просто воспроизводят заранее записанный аудиоклип. Сигнал можно сгенерировать также процедурным способом с опциональным добавлением одной или нескольких фундаментальных волновых функций (синусоиды, прямоугольной волны, гребенки и т. д.) и/или применением различных фильтров к гармонично насыщенному шуму. В большинстве игр почти всегда используются готовые аудиоклипы, поэтому мы будем рассматривать только этот вариант.

Заранее записанный аудиоклип может быть передан игровому движку в любом из великого множества сжатых и несжатых форматов, которые применяются в наши дни (см. подраздел 14.3.2). Каноническим форматом, который принимают на вход различные компоненты графа обработки звука, считается сырой PCM. Поэтому для преобразования каждого исходного аудиоклипа в PCM-поток используется аппаратный или программный *кодек*. Кодек распознает исходный формат данных, при необходимости распаковывает содержимое файла и передает его голосовой шине для дальнейшей обработки звуковым графом.

Регулятор усиления

Громкость каждого исходного звука в трехмерном мире можно контролировать разными способами. Во время записи аудиоклипа мы можем задать уровни для получения звука нужной громкости. Или заранее обработать клип, чтобы отрегулировать его усиление. На этапе выполнения можем динамически управлять громкостью клипа, задействуя *регулятор усиления* внутри звукового графа. Подробнее о регуляторе усиления говорится в подразделе 14.3.1.

Aux Send

Когда звукоинженер, находясь в звукозаписывающей студии или на концерте, хочет применить к звуку какие-то эффекты, он может вывести сигнал из многоканального микшерного пульта, пропустить его через блок эффектов и затем вернуть обратно для дальнейшей обработки. Для этого на микшерском пульте существуют специальные вспомогательные каналы AUX со своими выходами (aux send).

В контексте графа обработки звука термин aux send используется аналогично: он описывает точку разветвления в конвейере — разделение одного сигнала на два параллельных. Один из этих сигналов относится к сухому компоненту звука, другой проходит через модуль реверберации и/или других эффектов для создания «мокрого».

Реверберация

Путь «мокрого» сигнала обычно пролегает через компонент, который добавляет ранние отражения и позднюю реверберацию. Последняя может быть реализована с помощью свертки, как описано в подразделе 14.4.5. Если свертка не подходит для обработки в реальном времени либо из-за отсутствия модуля DSP в консоли или ПК, либо ввиду нехватки ресурсов процессора и/или памяти, этого эффекта можно добиться использованием *пружинного реверберatора* (reverb tank). Это, в сущности, система буферизации, кэширующая копии звука с задержкой, которые затем микшируются с оригиналом, чтобы имитировать ранние отражения и/или позднюю реверберацию. К этому добавляется *фильтр*, симулирующий эффекты интерференции и общего затухания высокочастотных компонентов в отраженных звуковых волнах.

Фильтр pre-send

Конвейер обработки голоса обычно содержит фильтр, который применяется перед разветвлением aux send и, следовательно, влияет как на сухой, так и на «мокрый» компоненты звука. Этот фильтр называется *pre-send*. Он в основном используется для моделирования явлений, которые возникают в источнике звука. Например, с помощью фильтра pre-send можно имитировать речь человека в противогазе.

Фильтр post-send

Обычно после разветвления aux send доступен еще один фильтр, который действует только на сухой компонент звука. Он может пригодиться для моделирования эффекта приглушения, вызванного препятствиями на прямом пути звука. В Naughty Dog мы используем фильтр post-send еще и для реализации затухания отдельных частот, вызванного атмосферным поглощением (см. подраздел 14.1.3).

Модули панорамирования

Сухой и «мокрый» компоненты 3D-звука остаются монофоническими на протяжении всего пути через голосовую шину. В самом конце конвейера каждый из этих двух моносигналов должен быть распределен между двумя стереодинамиками/наушниками или пятью/семью динамиками объемного звука. В связи с этим любая шина 3D-голоса заканчивается как минимум двумя модулями панорамирования: один предназначен для сухого сигнала, еще один или больше — для «мокрого». Компоненты могут панорамироваться по-разному. Сухой сигнал распределяется в соответствии с местоположением источника. Панорамирование «мокрого» сигнала можно сделать не таким сфокусированным, чтобы имитировать то, как отраженные звуковые волны доносятся с разных направлений. Если звук исходит из узкого дверного проема, панорамная дуга «мокрого» сигнала может составлять лишь несколько градусов. Но если слушатель находится в центре объемного зала, такой сигнал, наверное, стоит распределить на весь панорамный круг, то есть все динамики должны одинаково воспроизводить его.

14.5.4. Главный микшер

Вывод каждого модуля панорамирования имеет вид многоканальной шины, содержащей сигналы для каждого нужного нам выходного канала (стерео- или объемного звука). Обычно в игре одновременно воспроизводится множество 3D-звуков. *Главный микшер* принимает на вход все эти многоканальные сигналы и объединяет их в единый многоканальный сигнал, который затем выводится на динамики.

В зависимости от деталей реализации главный микшер может быть как аппаратным, так и полностью программным. В первом случае разработчик звуковой карты может выбирать между цифровым и аналоговым микшированием, во втором по понятным причинам микширование может быть исключительно цифровым.

Аналоговое микширование

Аналоговый микшер, в сущности, представляет собой цепь с суммированием: амплитуды отдельных входящих сигналов складываются, после чего амплитуда результирующей волны уменьшается в соответствии с нужным диапазоном напряжения.

Цифровое микширование

Микширование может выполняться в цифровом виде с помощью ПО, которое работает на специальном DSP-чипе или процессоре общего назначения. Цифровой микшер принимает на вход несколько потоков данных в формате PCM и выдает на выход единый PCM-поток.

Перед цифровым микшером стоит более сложная задача, чем перед аналоговым, так как PCM-каналы, которые он объединяет, могли быть записаны с разной частотой дискретизации и/или битовой глубиной. Чтобы входящие сигналы микшера были приведены к единому формату, все они должны пройти через *преобразование глубины* и *частоты дискретизации*. После этого микширование становится тривиальным. Мы просто складываем все входящие семплы в каждый дискретный момент времени, а затем при необходимости корректируем итоговую амплитуду, чтобы объединенный сигнал находился в нужном диапазоне громкости.

Преобразование глубины дискретизации

Если входящие сигналы микшера имеют разную *битовую глубину*, к ним можно применить *преобразование глубины дискретизации*, чтобы привести к единому формату. Это простая операция. Мы превращаем квантованные значения входящих семплов в числа с плавающей запятой и квантуем их заново, но уже с подходящей битовой глубиной. Мельчайшие подробности квантования рассмотрены в подразделе 12.8.2.

Преобразование частоты дискретизации

Если входящие сигналы имеют разную *частоту дискретизации*, это необходимо исправить еще до микширования. Теоретически сигнал нужно преобразовать в аналоговый вид и заново дискретизировать с нужной частотой (это можно сделать с помощью аппаратных ЦА- и АЦ-преобразователей). Но на практике повторная дискретизация восстановленного аналогового сигнала обычно создает нежелательный шум, поэтому преобразование почти всегда идет с помощью алгоритма, который применяется непосредственно к цифровому потоку данных в формате PCM.

Чтобы понимать принцип работы этих алгоритмов, необходимо как следует разобраться в теории обработки сигналов (см. раздел 14.2), однако это выходит за рамки данной книги. В некоторых простых случаях эта концепция воспринимается довольно легко. Например, чтобы *удвоить* частоту дискретизации, мы можем интерполировать смежные семплы и вставить полученные значения в качестве *новых* семплов, тем самым увеличивая их общее число в два раза. На самом деле все не так просто: например, мы должны позаботиться о том, чтобы в результирующем сигнале отсутствовал эффект наложения. Подробное описание преобразования частоты дискретизации можно найти на странице ru.wikipedia.org/wiki/Передискретизация.

14.5.5. Главная шина вывода

После микширования голоса обрабатываются *главной шиной вывода*. Это набор компонентов для обработки исходящего сигнала перед его подачей на динамики. Типичный пример такой шины показан на рис. 14.41, а ее составляющие кратко описаны далее. Каждый звуковой движок имеет свои особенности и может включать не все названные модули. Некоторые движки могут содержать также дополнительные компоненты, которые здесь не упомянуты.

- *Предусилитель*. Позволяет отрегулировать уровень мастер-сигнала, прежде чем пропускать его дальше по шине.

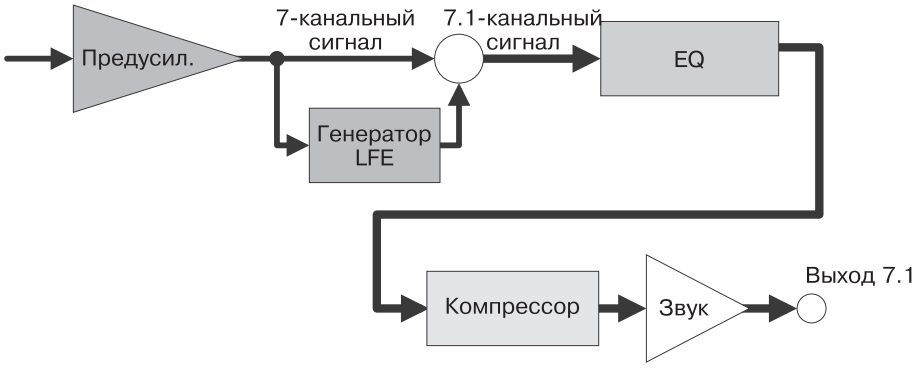


Рис. 14.41. Типичная главная шина вывода

- *Генератор LFE*. Как упоминалось в подразделе 14.4.4, модуль панорамирования работает только с двумя, пятью или семью основными динамиками системы стерео- или объемного звука. Канал LFE (сабвуфер) не участвует в позиционировании трехмерного образа сигнала. *Генератор LFE* — это компонент, который извлекает из итогового объединенного сигнала самые низкие частоты и подает их в канал LFE.
- *Эквалайзер*. Большинство звуковых движков предоставляют *эквалайзер* (EQ) в том или ином виде. Как говорилось в подразделе 14.2.5, эквалайзер позволяет усиливать или ослаблять сигнал в пределах определенных полос частот. Обычно частотный спектр делится на 4–10 полос, уровень в которых можно регулировать по отдельности.
- *Компрессор*. Компрессор *сжимает динамический диапазон* (dynamic range compression, DRC) звукового сигнала. Он понижает уровень самой громкой части звука и/или повышает уровень самых тихих участков и делает это автоматически, анализируя показатели уровней входящего сигнала и динамически регулируя сжатие. Подробнее о DRC можно узнать по адресу ru.wikipedia.org/wiki/Компрессор_аудиосигнала.
- *Главный регулятор усиления*. Этот компонент позволяет управлять общей громкостью всей игры.

- *Выходы*. Выход главной шины представляет собой набор выходов аналоговых сигналов, соответствующих каналам динамиков и/или цифровых многоканальных выходов HDMI или S/PDIF, которые позволяют подавать сигнал на телевизор или систему домашнего кинотеатра.

14.5.6. Реализация шины

Аналоговые шины

Аналоговая шина представляет собой ряд параллельных электронных соединений. Для передачи монофонического аудиосигнала нужны два параллельных провода или две контактные дорожки на печатной схеме: одна для сигнала напряжения, другая — для заземления.

Аналоговая шина работает практически мгновенно. Сигнал представляет собой непрерывное физическое явление и немедленно передается от предыдущего компонента к следующему. Такие цепи получаются довольно простыми. Единственная реальная сложность состоит в том, чтобы гарантировать согласование уровней напряжения и электрического сопротивления для входного и выходного сигналов.

Цифровые шины

Для обеспечения мгновенной связи между цифровыми компонентами теоретически можно использовать простую цифровую схему. Но для этого соединенные модули должны быть идеально синхронизированы: как только один генерирует байт данных, другой сразу же должен его потребить. В противном случае этот байт будет утерян.

Чтобы преодолеть проблемы синхронизации, присущие соединениям двух цифровых компонентов, обычно на входе и/или выходе каждого из них размещают *кольцевой буфер*, который используется двумя клиентами: один из них *читает*, другой *записывает*. Чтобы это работало, буфер хранит два указателя, или индекса, которые называются *читающей головкой* и *записывающей головкой*. Первый клиент считывает данные, перемещая читающую головку по буферу, и, дойдя до конца, начинает чтение сначала. Второй клиент сохраняет данные в буфер, перемещая записывающую головку, и тоже возвращается в начало, когда доходит до конца. Ни одна из этих головок не может обогнать другую, что гарантирует отсутствие между клиентами конфликтов, таких как чтение еще не записанных данных или перезапись того, что считывается в текущий момент.

Чтобы подключить, скажем, цифровой выход кодека к цифровому входу ЦАП, проще всего использовать *разделяемый* кольцевой буфер. Он будет обеспечивать запись со стороны кодека и чтение со стороны ЦАП.

Этот подход выглядит простым, но работает только в случае, когда оба компонента имеют доступ к одной и той же физической памяти. Этого легко достичь, если они работают в разных *потоках*. Но если они представляют собой отдельные *процессы*, каждый со своим собственным виртуальным адресным пространством, операционная система должна предоставить механизм для отображения одного

и того же участка физической памяти в виртуальные адресные пространства каждого из процессов. Обычно это возможно только тогда, когда процессы выполняются на одном ядре или на разных ядрах одного многоядерного компьютера.

Если компоненты выполняются на разных ядрах, которые не могут разделять память (например, если один работает на ПК, а другой — на подключаемой звуковой карте), у каждого из них должны быть собственные входной и выходной буферы. Данные должны *копироваться* из выходного буфера одного компонента во входной буфер другого. Этого можно добиться за счет контроллера прямого доступа к памяти (direct memory access controller, DMAC), как в случае с передачей данных между PPU и SPU в PS3. Для этого также можно использовать специальную шину, такую как PCI Express (PCIe, доступна почти везде), которая позволяет соединять ядра центрального процессора с подключаемым периферийным оборудованием ПК.

Латентность шины

Для воспроизведения звука игры и приложения должны периодически передавать аудиоданные кодекам, которые, по большому счету, управляют выводом динамиков. Мы называем это *подачей* звука. Частота подачи звука определяет качество его воспроизведения. Если пакеты отправляются слишком медленно, устройство считает все данные и *опустошит* буфер еще до появления новых пакетов. В результате программное обеспечение будет не успевать, что приведет к прерыванию звука. Если пакеты отправляются слишком быстро, PCM-буферы могут *переполниться* и вызвать потерю пакетов. Из-за этого звук может воспроизводиться скачками.

Размер входящих и исходящих буферов, из которых состоит цифровая шина, определяет *латентность* звуковой системы, то есть задержку, которую эта шина создает. Если сделать буферы очень маленькими, задержка будет минимальной, но вместе с этим возрастет нагрузка на процессор, которому придется чаще подавать данные в буфер. В то же время большие буферы используют процессор не так активно, но при этом имеют более высокую латентность. Вместо размера буфера (в байтах) принято измерять латентность аппаратных аудиокомпонентов (в миллисекундах). Дело в том, что размер буфера зависит от формата данных и степени сжатия, которая поддерживается кодеком, но на самом деле нас интересует задержка, возникающая при воспроизведении высококачественного звука.

Какая латентность считается приемлемой? Это зависит от приложения. В профессиональных аудиосистемах она должна быть очень короткой — порядка 0,5 мс. Это связано с тем, что перед синхронизацией между собой (а иногда и с видеосигналом) звуковые сигналы часто проходят через цепочки аудиоустройств. Латентность, возникающая в каждом из компонентов, затрудняет синхронизацию.

В игровых консолях допускается более существенная латентность, ведь нас интересует лишь синхронизация звука и графики. Если рендеринг игры происходит с частотой 60 кадров в секунду, это означает, что на каждый кадр отводится 1/60 = 16,6 мс. То есть если звук задерживается не более чем на 16 мс, мы можем быть уверены в том, что он будет воспроизводиться синхронно с графикой, отрисованной в том же кадре. На самом деле движки отрисовки во многих играх использу-

ют двойную или тройную буферизацию, что создает задержку длиной один или два кадра между запросом на вывод и появлением соответствующей картинки на экране телевизора. Задержку может создавать и сам телевизор, поэтому в игре с тройным буфером и частотой кадров 60 Гц допускается латентность звука $3 \cdot 16 = 48$ мс и больше. Контроллер DMA в PlayStation 3 срабатывает каждые 5,5 мс, поэтому аудиосистемы в этой консоли обычно сконфигурированы так, чтобы аудиобуфер мог хранить звук, длина которого кратна 5,5 мс.

14.5.7. Управление ресурсами

Аудиоклипы

Элементарным аудиоресурсом является *клип* — отдельный цифровой ресурс со своей локальной временной шкалой (по аналогии с анимационным клипом). Иногда его называют *звуковым буфером*, так как цифровые данные семплов хранятся в буфере. Клип может содержать как монофонические звуковые сигналы (характерно для ресурсов с 3D-звуком), так и многоканальный звук (обычно это можно встретить в 2D-ресурсах или источниках стереозвука в 3D-ресурсах). Он может храниться в файле любого формата, который поддерживает ваш движок.

Звуковые очереди

Звуковые очереди — это коллекции аудиоклипов с метаинформацией, которая описывает, как их следует обрабатывать и воспроизводить. Это, как правило, основной механизм, с помощью которого игра может инициировать воспроизведение звуков (движок может даже не поддерживать проигрывание отдельных клипов). Звуковые очереди — удобное средство разделения труда: звукоинженеры могут создавать их заранее, не беспокоясь о том, как и на каком этапе игры они будут использоваться. А программисты могут с легкостью проигрывать соответствующие клипы в ответ на подходящие в игре события, не беспокоясь о подробностях воспроизведения.

Набор клипов в очереди можно интерпретировать и воспроизводить разными способами. Каждый клип может представлять один из шести каналов в немикшированной музыкальной записи формата 5.1. Очередь может состоять и из набора необработанных звуков, которые выбирают случайным образом для создания разнообразия. Мы также можем сделать так, чтобы необработанные звуки в очереди проигрывались в заранее определенном порядке. В метаданных обычно указано, как нужно воспроизводить звуки: один раз или в цикле.

Некоторые звукоинженеры позволяют очереди предоставлять один или несколько дополнительных аудиоклипов, которые воспроизводятся только в случае, если главный звук был прерван посредине. Например, голосовая очередь может содержать звук гортанной смычки (https://ru.wikipedia.org/wiki/Гортанная_смычка), который проигрывается, только если персонажа перебили. С помощью этого механизма можно предусмотреть и отдельный хвостовой звук, который воспроизводится при остановке зацикленной очереди. Например, когда пулемет перестает издавать циклические звуки выстрелов, в качестве хвостового клипа можно воспроизвести подходящее затухающее эхо.

Метаданные очереди иногда определяют способ воспроизведения звука (в 2D или 3D), FO min, FO max, кривую затухания источника звука, принадлежность к группе (см. подраздел 14.5.8) и, возможно, разные спецэффекты, фильтрацию или выравнивание, которые следует применять при воспроизведении. В звуковом движке Scream от Sony, который студия Naughty Dog использовала в своих игровых циклах *Uncharted* и *The Last of Us*, очередь может содержать произвольный скриптовый код, дающий звукоинженеру полный контроль над воспроизведением инкапсулированных звуковых ресурсов с помощь очереди.

Воспроизведение очереди. Любой звуковой движок, поддерживающий концепцию очередей, предоставляет API для их воспроизведения. Этот интерфейс обычно служит основным, а иногда и *единственным* механизмом, с помощью которого код игры может инициировать проигрывание звуков.

В целом, с помощью этого API программист может определить способ воспроизведения очереди (в качестве 2D- или 3D-звука), предоставить позицию в трехмерном пространстве и вектор скорости, указать, нужно ли зацикливать звук и откуда берется исходный буфер — из памяти или из потока. Помимо этого, обычно можно управлять громкостью и другими аспектами воспроизведения.

Большинство API возвращают вызывающему коду *дескриптор звука*. С его помощью программа может отслеживать воспроизведение звука и, следовательно, модифицировать и преждевременно его останавливать. Этот механизм обычно реализуется в виде индекса в глобальной таблице дескрипторов, а не как обычный указатель на данные, описывающие экземпляр звука. Благодаря этому при естественном завершении воспроизведения дескриптор можно автоматически обнулить. Это позволяет сделать систему потокобезопасной: если один поток остановит звук, другие потоки смогут автоматически увидеть, что их дескрипторы этого звука стали недействительными.

Библиотеки звуков

Движок 3D-звука управляет *огромным* количеством ресурсов. Игровой мир содержит множество объектов, каждый из которых может генерировать разнообразные звуки. Помимо объемных аудиоэффектов, в нем есть музыка, речь, звуки меню и т. д.

Все эти аудиоданные громадного размера, поэтому их нельзя хранить целиком в памяти. В то же время отдельные аудиоклипы слишком мелкие и многочисленные для того, чтобы управлять ими по отдельности. В связи с этим большинство игровых движков пакуют звуковые клипы и очереди в большие модули, которые называются *библиотеками звуков* или *звуковыми банками*.

Некоторые библиотеки звуков загружаются при запуске игры и постоянно присутствуют в памяти. Например, нам всегда нужна коллекция звуков, издаваемых персонажем, поэтому ее выгружать не стоит. Другие библиотеки могут загружаться и выгружаться динамически, в зависимости от потребностей игры. Например, звуки уровня A могут быть не нужны на уровне B, поэтому библиотеку A можно загружать только во время игры на соответствующем уровне. В игре *The Last of Us* от Naughty Dog шум дождя, текущей воды и скрип балок на грани обрушения загружались, только когда игрок находился в покосившемся здании в Бостоне.

Некоторые звуковые движки поддерживают *перемещение* библиотек звуков в памяти. Благодаря этой возможности полностью устраняются проблемы с фрагментацией памяти, которые в противном случае могли бы возникнуть из-за постоянной загрузки и выгрузки множества библиотек разного размера во время игрового процесса. Подробнее о перемещении памяти говорится в подразделе 6.2.2.

Потоковая передача звуков

Некоторые звуки слишком длинны для того, чтобы их было удобно хранить в памяти целиком. Распространенными примерами являются музыка и речь. Для таких случаев во многих игровых движках предусмотрена поддержка *потокового аудио*.

Эта возможность обусловлена тем фактом, что при воспроизведении звука нужны лишь данные о сигнале в текущем индексе времени и вокруг него. Для реализации потоковой передачи используются относительно небольшие кольцевые буферы. Перед воспроизведением звука мы загружаем в буфер небольшую его часть и проигрываем его как обычно. Звуковой конвейер извлекает данные из буфера по мере воспроизведения, позволяя загружать туда все новую информацию. И пока мы успеваем наполнять буфер до извлечения всего его содержимого, звук будет проигрываться без перебоев.

14.5.8. Микширование в игре

Если проигрывать все звуки, доносящиеся от каждого игрового объекта, с корректным затуханием, спатиализацией и акустической моделью, применяя все те методы и технологии, которые мы обсуждали до сих пор, что получится в результате? Можно было бы ожидать такого ответа: «Невероятно правдоподобный звуковой ландшафт с эффектом присутствия, который получит престижные награды и сделает нас богатыми!» Но в реальности мы получили бы какофонию.

Чем хорошая игра отличается от великолепной, так это *микшированием* — тем, какие звуки вы слышите и в каких пропорциях и, что немаловажно, какие звуки до вас не доходят. Задача звукоинженера в игровом проекте состоит в создании итогового *микса*, который:

- звучит реалистично и создает эффект присутствия;
- не слишком отвлекает, раздражает или нагружает слух;
- эффективно передает всю информацию, относящуюся к игровому процессу и/или сюжету;
- поддерживает настроение и тональность, которые всегда соответствуют происходящим событиям и общей концепции игры.

В ходе микширования приходится объединять всевозможные звуки, включая музыку, речь, звуки окружающей среды, такие как шум дождя и ветра, жужжание насекомых и скрипы в старых зданиях, звуковые эффекты наподобие выстрелов, взрывов и шума автомобильных моторов, звуки столкновений, скольжения и вращения, издаваемые физически симулируемыми объектами.

Чтобы микширование в игре соответствовало всем этим требованиям, используются различные методики. Некоторые из них мы проанализируем в следующих пунктах.

Группы

Самый очевидный способ улучшить микширование в игре состоит в правильном выборе уровней для всех исходных звуков в трехмерном мире. Важно сделать так, чтобы звуки имели подходящие коэффициенты усиления *друг относительно друга*. Например, выстрелы должны быть громче шагов.

В некоторых играх громкость определенных звуков должна изменяться динамически. Часто возникает необходимость в управлении целыми категориями звуков. Например, во время неистового сражения имеет смысл повысить уровни музыки и выстрелов, но при этом приглушить вспомогательные звуковые эффекты. А во время тихих сцен, когда персонажи разговаривают друг с другом, можно немного усилить речь и ослабить звуки окружающей среды, чтобы диалог было лучше слышно.

В связи с этим многие звуковые движки поддерживают концепцию *групп*, позаимствованную у нашего старого приятеля — многоканального микшерного пульта. Панель микширования позволяет направлять группы звуковых входов в промежуточную микшерную цепь, организовывая их в единый групповой сигнал. Коэффициент усиления этого сигнала можно регулировать с помощью одного ползунка, что позволяет звукоинженеру управлять громкостью всех входящих сигналов сразу.

В мире программного обеспечения группы реализуют простым распределением звуковых очередей по категориям, без физического микширования их сигналов. Например, очередь может быть помечена как музыка, звуковой эффект, оружие, строчка диалога и т. д. Движок может предоставлять средства управления всеми звуками в каждой категории с помощью одного значения. Звукоинженеру обычно доступны удобные и простые API-вызовы, которые позволяют останавливать, перезапускать и заглушать воспроизведение отдельных групп.

Некоторые движки по аналогии с микшерными пультами обеспечивают механизм физического объединения групп звуков в единый звуковой сигнал. В движке Scream от Sony это называется генерацией *предварительного подмикса* (pre-master submix). После того как подмикс установил относительное усиление сигналов в группе, результирующий сигнал можно пропустить через дополнительные фильтры или другие этапы обработки. Это дает звукоинженеру дополнительный контроль над микшированием в игре.

Эффект ducking

Ducking — это временное уменьшение уровня определенных звуков, чтобы другие звуки было лучше слышно. Например, когда персонажи разговаривают, мы можем понизить уровень фонового шума, чтобы сделать диалог более отчетливым.

Этот эффект можно инициировать множеством разных способов. Наличие звука какого-то определенного типа может служить поводом для приглушения другой

категории звуков. Это можно сделать и программным образом. Для этого сгодится любой подходящий механизм.

Снижение уровня, вызванное эффектом ducking, обычно выполняется через систему категоризации групп: когда воспроизводится какая-то категория звуков, она может автоматически приглушить одну или несколько других категорий (в разной степени). Или же код игры может вызвать функцию для программного приглушения звуков.

Эффект ducking может быть реализован также параллельной подачей одного звукового сигнала на дополнительный вход компрессора динамического диапазона (dynamic range compressor, DRC), размещенного на другой голосовой шине. Как вы помните из подраздела 14.5.5, DRC анализирует уровни сигналов и автоматически регулирует их громкость. Если подать на вход DRC параллельный сигнал, он проанализирует его, прежде чем изменять уровень. Таким образом, мы можем сделать так, чтобы повышение громкости одного сигнала приводило к понижению громкости другого.

Предустановки шины и снимки параметров микширования

Многие звуковые движки позволяют звукоинженерам устанавливать и сохранять конфигурацию, а затем загружать и легко применять ее на этапе выполнения. В движке Scream от Sony это достигается за счет двух основных возможностей: *предустановок шины* (bus presets) и *снимков параметров микширования* (mix snapshots).

Предустановка шины — это набор конфигурационных параметров, определяющих поведение компонентов на одной шине (голосовой шине или шине главного вывода). Это может быть определенное сочетание эффектов реверберации, которые, скажем, имитируют акустику просторного открытого зала, салона автомобиля или небольшой кладовки. Или настройки DRC для шины главного вывода. Многие предустановки подобного рода могут быть созданы звукоинженерами и активированы в подходящие моменты игрового процесса.

Снимки параметров микширования имеют аналогичную концепцию, но в контексте регуляции усиления. Мы можем заранее определить коэффициенты передачи различных каналов в группе и затем в случае необходимости применить их во время выполнения.

Ограничение количества сигналов

Ограничение количества сигналов позволяет определить, сколько звуков может воспроизводиться одновременно. Например, если у нас есть 20 NPC, каждый из которых стреляет, мы можем озвучить лишь трех или четырех из них, находящихся ближе всего к слушателю. Это важно по двум причинам: во-первых, это отличный способ предотвратить какофонию, во-вторых, звуковой движок обычно способен одновременно воспроизводить ограниченное количество голосов. Последнее может быть вызвано как аппаратными ограничениями (например, у звуковой карты есть всего N кодеков), так и наличием программных лимитов на память и использование процессора, поэтому ресурсы следует применять грамотно.

Лимиты для отдельных групп. Иногда разные группы звуков имеют разные ограничения количества сигналов. Например, мы можем указать, что одновременно должны озвучиваться не больше трех персонажей и воспроизводиться не больше четырех выстрелов, пяти звуковых эффектов и двух звуковых дорожек.

Задание приоритетов и заимствование голосов. В трехмерной игре с множеством динамических элементов количество звуков, проигрываемых в любой отдельно взятый момент, может превышать число голосов, доступных в системе. Некоторые звуковые движки поддерживают большое или даже бесконечное количество *виртуальных голосов*. Каждый виртуальный голос представляет звук, который формально воспроизводится, но может быть временно остановлен или приглушен, чтобы освободить драгоценные аппаратные или программные ресурсы. Движок использует различные критерии для динамического определения виртуальных голосов, которые в тот или иной момент следует привязать к реальным.

Один из самых простых способов ограничения числа одновременно проигрываемых звуков — задание максимального радиуса для каждого источника 3D-сигнала. Как сказано в подразделе 14.4.3, это радиус FO max. Если источник находится на большем расстоянии от слушателя, он считается неслышимым, а его виртуальный голос временно приглушается или останавливается, освобождая ресурсы для других голосов. Процесс автоматического приглушения называется *заимствованием голосов*.

Еще один распространенный подход состоит в назначении каждой очереди или группе очередей *приоритета*. Когда одновременно воспроизводится слишком много виртуальных голосов, мы можем приглушить (позаимствовать) те из них, которые имеют более низкий приоритет.

У звукового движка могут быть и другие механизмы для управления разными аспектами алгоритма заимствования голосов. Например, очереди можно выделить минимальное время воспроизведения, по истечении которого ее голос может быть позаимствован. В процессе заимствования звук можно останавливать не резко, а с плавным ослаблением. А некоторые очереди могут получать временный иммунитет от заимствования, чтобы они воспроизводились, несмотря на приоритет.

Микширование внутриигровых кинематографических звуков

В условиях обычного игрового процесса местоположение слушателя или виртуального микрофона обычно совпадает с позицией камеры или находится рядом с ней, а источники звуков моделируются в соответствии с их реальными координатами в виртуальном мире. Затухание в зависимости от расстояния, определение прямого и окольного путей распространения звука, ограничение количества голосов — все это основано на реалистичном положении объектов.

Однако во время *внутриигровых кинематографических сцен*, когда игровой процесс приостанавливается в угоду сюжету, камера часто удаляется от головы игрока. Это может кардинально нарушить работу системы 3D-звука. Мы могли бы просто привязать местоположение слушателя/микрофона к позиции камеры, но такое решение подходит не всегда. Например, если есть длинная сцена с двумя разговаривающими персонажами, нам, наверное, следует сделать так, чтобы их голоса можно

было слышать, если они физически находятся слишком далеко. В этом случае микрофон лучше отвязать от камеры и искусственно приблизить к персонажам.

Микширование внутриигровых кинематографических сцен во многом похоже на аналогичный процесс в фильмах. Поэтому звукоинженер должен уметь нарушать правила и делать то, что с физической точки зрения не совсем реалистично.

14.5.9. Обзор звуковых движков

К этому моменту уже должно быть очевидно, что создание движка 3D-звука — масштабное начинание. К счастью, этой задачей уже занимается множество людей, благодаря усилиям которых нам доступен широкий выбор программного обеспечения для работы со звуком, готового к использованию практически в исходном виде. Это касается как низкоуровневых звуковых библиотек, так и полнофункциональных движков рендеринга 3D-звука.

В следующих пунктах мы рассмотрим некоторые из наиболее распространенных звуковых библиотек и движков. Некоторые из них предназначены для конкретной платформы, другие являются межплатформенными.

Windows: Universal Audio Architecture

На ранних этапах развития компьютерных игр возможности и архитектуры звуковых карт в ПК существенно варьировались в зависимости от платформы и производителя. Компания Microsoft пыталась объединить все это разнообразие внутри своего API DirectSound, который поддерживался в Windows Driver Model (WDM) и драйвере Kernel Audio Mixer (KMixer). Однако производителям никак не удавалось договориться об общем наборе возможностей и стандартных интерфейсов, поэтому реализация одних и тех же функций в разных звуковых картах зачастую существенно различалась. Из-за этого операционной системе приходилось иметь дело с огромным количеством несовместимых между собой интерфейсов драйверов.

Но с выходом Windows Vista компания Microsoft представила новый стандарт под названием Universal Audio Architecture (UAA). Стандартный API драйверов UAA поддерживал ограниченный набор возможностей, а все остальное реализовывалось на прикладном уровне (производители оборудования по-прежнему могут предоставлять дополнительные функции аппаратного ускорения, но теперь они доступны только в фирменных нестандартных драйверах). Появление UAA ограничило конкурентные преимущества известных производителей звуковых карт, таких как Creative Labs, а вместе с тем, как и ожидалось, мы получили надежный полнофункциональный стандарт, удобный для использования как в играх, так и в обычных приложениях.

Еще одной положительной чертой стандарта UAA стало улучшение совместимости звуковых возможностей разных приложений. Во времена DirectSound игра могла полностью контролировать звуковую карту, блокируя воспроизведение звуков, генерируемых операционной системой, почтовым клиентом и т. д. UAA позволила ОС получить полный контроль за итоговым сигналом, который воспроизводится

динамиками компьютера. Благодаря этому разные приложения наконец научились разумно и предсказуемо разделять звуковую карту. Чтобы больше узнать об UAA, поищите в Интернете *Universal Audio Architecture*.

Стандарт UAA реализован в Windows в виде API Windows Audio Session (WASAPI), который на самом деле не предназначался для использования в играх. Самые продвинутые возможности обработки в нем поддерживаются только в программном режиме с ограниченным применением аппаратного ускорения. Вместо WASAPI в играх обычно задействуется API XAudio2, о котором пойдет речь в следующем подразделе.

XAudio2

XAudio2 — это мощный низкоуровневый API, предоставляющий доступ к звуковому оборудованию в Xbox 360, Xbox One и Windows. Это замена DirectAudio, которая позволяет использовать широкий спектр возможностей с аппаратным ускорением, включая программируемые эффекты DSP, подмикширование, работу с большим количеством сжатых и несжатых аудиоформатов, а также переменную частоту дискретизации для снижения нагрузки на центральный процессор.

XAudio2 лежит в основе библиотеки рендеринга 3D-звука под названием X3DAudio. Эти API доступны в том числе на платформе Windows, их можно задействовать в играх для ПК. Когда-то компания Microsoft предлагала мощную программу для редактирования звука, XACT (cross-platform audio creation tool — «межплатформенное средство создания аудио»), которая предназначалась для использования совместно с XNA Game Studio, но ни XNA, ни XACT больше не поддерживаются.

Scream и BoomRangBuss

Для PS3 и PS4 студия Naughty Dog применяет движок 3D-звука Scream от Sony и его библиотеки синтеза BoomRangBuss.

Аудиооборудование в PlayStation 3 во многом похоже на устройство, совместимое с UAA: оно поддерживает до восьми звуковых каналов для полноценного объемного звука в формате 7.1, а также аппаратный микшер и выходные разъемы HDMI, S/PDIF, USB/Bluetooth и аналоговый выход. Эти возможности объединены в набор системных библиотек, таких как libaudio, libsynth и libmixer, поверх них разработчики игр могут создавать собственные программные аудиостеки. Компания Sony предоставляет многофункциональный стек с поддержкой 3D-звука под названием Scream, который игровые студии могут использовать в готовом виде. Он доступен для платформ PS3, PS4 и PSVita. Его архитектура имитирует полноценный многоканальный микшерный пульт.

Поверх Scream студия Naughty Dog реализовала собственную фирменную аудиосистему с трехмерным окружением для применения в циклах *Uncharted* и *The Last of Us*. Эта система поддерживает стохастическую модель частичного/полного блокирования звуковых сигналов и систему рендеринга звука на основе порталов, позволяющую генерировать чрезвычайно реалистичный звуковой ландшафт.

Advanced Linux Sound Architecture (ALSA). В Linux есть свой эквивалент модели драйверов UAA под названием Advanced Linux Sound. Это компонент ядра, который заменил собой оригинальную систему Open Sound System (OSSv3) в качестве стандартного интерфейса к звуковым возможностям для приложений и игр. Больше об ALSA можно узнать на странице www.alsa-project.org/main/index.php/Main_Page.

QNX Sound Architecture (QSA). QNX Sound Architecture — это звуковой API для ОС QNX Neutrino, функционирующий на уровне ядра. Как игровой разработчик, вы, наверное, никогда не столкнетесь с QNX. Но документация этой системы позволяет получить хорошее представление о концепциях и типичном наборе возможностей звукового оборудования. Ознакомиться с ней можно по адресу www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_audio%2Fmixer.html.

Многоплатформенные движки 3D-звука

Сейчас существует целый ряд готовых к использованию межплатформенных движков 3D-звука. Далее перечислены наиболее известные из них.

- *OpenAL* — это межплатформенный API для рендеринга 3D-звука, специально созданный для имитации архитектуры графической библиотеки OpenGL. Его ранние версии имели открытый исходный код, но теперь это ПО распространяется по лицензии. Ряд поставщиков предоставляют собственные реализации API OpenAL, включая OpenAL Soft (kcat.strangesoft.net/openal.html) и AeonWave-OpenAL (www.adalin.com).
- *AeonWave 4D* — аудиобиблиотека от Adalin B.V. с поддержкой Windows и Linux. Цена доступная.
- *FMOD Studio* — программа для редактирования звука с профессионально выглядящим пользовательским интерфейсом (www.fmod.org). Она включает в себя полноценный API для работы с 3D-звуком, который позволяет воспроизводить ресурсы, созданные в FMOD Studio, в режиме реального времени на платформах Windows, Mac, iOS и Android.
- *Miles Sound System* — это популярный промежуточный аудиослой от Rad Game Tools (www.radgametools.com/miles.htm). Он предоставляет мощный граф обработки звука и доступен практически на любой игровой платформе, которую только можно себе представить.
- *Wwise* — движок рендеринга 3D-звука от Audiokinetic (www.audiokinetic.com). Его особенность в том, что он не основан на концепциях и возможностях многоканального микшерного пульта, а предоставляет звукоинженерам и программистам уникальный интерфейс, состоящий из игровых объектов и событий.
- И конечно, нельзя не упомянуть движок *Unreal Engine*, который имеет собственную систему рендеринга 3D-звука и мощный интегрированный инструментарий (www.unrealengine.com). Для более глубокого знакомства с возможностями и инструментами для работы со звуком см. [45].

14.6. Звуковые возможности, характерные для разных видов игр

Помимо конвейера рендеринга 3D-звука, игры обычно реализуют разного рода возможности и подсистемы, обусловленные конкретными требованиями.

- *Поддержка разделенного экрана*. В многопользовательских трехмерных играх, поддерживающих разделенный экран, должен быть какой-то механизм разделения одного набора динамиков между *несколькими игроками*, находящимися в одной комнате.
- *Звук на основе физической симуляции*. Играм, поддерживающим динамические физически симулируемые объекты вроде обломков, разрушаемых предметов и тряпичных кукол, нужно как-то проигрывать подходящие звуки в ответ на столкновения, скольжение, вращение и разрушение.
- *Система динамической музыки*. Во многих сюжетных играх музыка должна в реальном времени адаптироваться к настроению и напряженности игровых событий.
- *Система диалогов*. Персонажи, управляемые ИИ, могут показаться куда более реалистичными, если умеют разговаривать между собой и с игроком.
- *Синтез звука*. Некоторые движки по-прежнему позволяют синтезировать звуки с нуля путем сочетания разных волновых функций (синусоид, косинусоид, пилообразных колебаний и т. д.) с разными уровнями и частотами. Продвинутые методы синтеза тоже находят применение в играх в режиме реального времени.
  - *Синтезаторы музыкальных инструментов* воспроизводят естественное звучание аналоговых музыкальных инструментов, не используя заранее записанные аудиоклипы.
  - *Синтез звука на основе физической симуляции* охватывает широкий спектр методик, предназначенных для точного воспроизведения звуков, которые должен издавать объект в процессе физического взаимодействия с виртуальной средой. Такие системы задействуют информацию о контакте, импульсе, силе, вращательном моменте и деформации, предоставляемую современными движками физической симуляции, в сочетании со свойствами материала объекта и его геометрической формы для синтеза подходящих звуков при столкновении, скольжении, вращении, сгибании и т. д. Вот лишь несколько ссылок для исследования этой захватывающей темы: gamma.cs.unc.edu/research/sound, gamma.cs.unc.edu/AUDIO_MATERIAL, www.cs.cornell.edu/projects/sound и ccrma.stanford.edu/~bilbao/booktop/node14.html.
  - *Синтезаторы автомобильных двигателей* пытаются воспроизвести звуки, издаваемые транспортным средством, с учетом таких свойств, как ускорение, число оборотов в минуту, нагрузка на виртуальный двигатель и механические движения (в трех играх серии *Uncharted* от Naughty Dog использовалось динамическое моделирование автомобильных двигателей в том или ином виде, хотя формально эти системы не были синтезаторами, так как их вывод состоял из плавных переходов между заранее записанными звуками).

- *Синтезаторы артикуляционной речи* генерируют человеческую речь с нуля, применяя трехмерную модель речевого тракта человека. VocalTractLab (www.vocaltractlab.de) — бесплатный инструмент для изучения синтеза речи и экспериментов в этой области.

- *Моделирование толпы*. Играм с большими скоплениями людей (зрителями, прохожими и т. д.) нужен какой-то механизм для их озвучивания. Недостаточно просто проигрывать множество человеческих голосов, наложенных друг на друга. Обычно толпу моделируют в виде нескольких звуковых слоев, включая фоновый шум и речь отдельных людей.

Мы не можем рассмотреть в одной главе все пункты этого списка. Но выделим несколько страниц для обзора некоторых наиболее распространенных возможностей, характерных для определенных видов игр.

14.6.1. Поддержка разделенного экрана

Поддержка многопользовательского режима с разделенным экраном — нетривиальная задача, поскольку вам придется иметь дело с *несколькими слушателями* в виртуальном игровом мире, которые находятся в одной комнате и используют один и тот же набор динамиков. Если просто панорамировать звук несколько раз, по одному для каждого слушателя, и затем равномерно смикшировать результат в динамики, полученный звук может оказаться не совсем удачным. Идеального решения не существует. Например, если игрок А находится в непосредственной близости от взрыва, а игрок Б — вдалеке, звук взрыва будет громко и ясно слышен обоим игрокам. Лучшее, что здесь можно сделать, — состряпать гибридное решение, в котором одни звуки обрабатываются физически правильным способом, а другие подтасовываются, чтобы с точки зрения игроков все звучало как можно более логично.

14.6.2. Система диалогов

Даже если созданные нами игровые персонажи выглядят словно фотокопии живых людей и двигаются на удивление естественно, без умения реалистично *разговаривать* они не покажутся игроку настоящими. Речь передает критически важную с точки зрения игрового процесса информацию. Это ключевой элемент повествования. Он цементирует эмоциональную связь между игроком и персонажами игры. Речь может быть решающим фактором в восприятии игроком *интеллекта* персонажей, управляемых системой ИИ.

На конференции Game Developer's Conference (GDC) в 2002 году Крис Бутчер и Джейми Гризмер из компании Bungie представили презентацию *The Illusion of Intelligence: The Integration of AI and Level Design in Halo* (bit.ly/1g7FbhD). В ней они поделились историей о том, насколько важной может оказаться речь для передачи игроку мотивации персонажей на основе ИИ. Когда в игре *Halo* убивают лидера отряда из альянса Covenant, его подчиненные разбегаются в панике. Во время многочисленных игровых тестов никто не понимал, что причиной отступления рядовых была смерть их лидера. В конце концов рядовым назначили реплики

примерно такого характера: «Лидер мертв — бежим отсюда!» Только после этого тестировщики сумели уловить суть происходящего!

В этом разделе мы исследуем фундаментальные подсистемы, которые можно встретить в системе диалогов практически любой игры, где персонажи играют ключевую роль. Также обсудим определенные методики и технологии, с помощью которых студии Naughty Dog удалось создать насыщенные и реалистичные разговоры в *The Last of Us*. Больше информации и внутриигровые видеоролики с демонстрацией системы диалогов можно найти в моем выступлении на GDC 2014 под названием *Context-Aware Character Dialog in The Last of Us*, доступном в форматах PDF и QuickTime по ссылке www.gameenginebook.com.

Озвучивание персонажа

Озвучить персонажа легко — достаточно просто проигрывать подходящие заранее записанные звуки каждый раз, когда он должен что-то сказать. Но в реальности все немного сложнее. Игровые движки обычно имеют довольно развитые системы диалогов, и тому есть много причин. Вот лишь несколько из них.

- Нам нужно как-то организовать все реплики, которые персонаж может когда-либо произнести, и назначить им уникальные идентификаторы, чтобы игра могла их инициировать при необходимости.
- Мы должны убедиться в том, что каждый уникальный персонаж в игре имеет узнаваемый голос, который никогда не меняется. Например, каждому охотнику в главе Pittsburgh игры *The Last of Us* был назначен один из восьми уникальных голосов, чтобы во время сражения не было двух охотников, звучащих одинаково.
- Мы можем не знать заранее, какой персонаж должен будет произнести ту или иную реплику, поэтому иногда один и тот же текст озвучивают несколько актеров. Таким образом, в нужный момент для произнесения реплики можно будет выбрать подходящий голос.
- Обычно произносимый текст должен быть очень разнообразным, поэтому большинство систем диалогов предоставляют средства для случайного выбора реплик из набора возможных вариантов.
- Звуковые ресурсы с речью часто довольно продолжительны и поэтому занимают много места в памяти. Многие строчки диалогов являются частью кинематографических сцен и произносятся лишь один раз на протяжении игры. С учетом всего этого хранить ресурсы с речью в памяти обычно неоправданно. Вместо этого, как правило, такие ресурсы передаются по требованию в виде *потока* (см. подраздел 14.5.7).

Издаваемые людьми звуки, например, при поднятии тяжести, перепрыгивании препятствия или получении удара под дых управляются той же системой, которая отвечает за разговорные диалоги. Это во многом вызвано тем, что подобные звуки должны совпадать с голосом персонажа. Поэтому для их воспроизведения тоже можно использовать систему диалогов.

Определение строчки диалога

В большинстве систем диалогов есть некий промежуточный слой между *запросом* речи и выбором конкретного *аудиоклипа*, который нужно проиграть. Программист или дизайнер запрашивает *логические* строчки диалога, каждая из которых представлена уникальным идентификатором в виде строки или, еще лучше, хеша (см. подраздел 6.4.3). Затем дизайнеры могут назначить каждой логической реплике один или несколько аудиоклипов, чтобы обеспечить необходимое разнообразие с точки зрения особенностей голоса и того, что именно будет произнесено.

Представим, к примеру, логическую реплику такого плана: «У меня закончились патроны». Назначим ей уникальный идентификатор `'line-out-of-ammo` (одинарная кавычка в начале говорит о том, что эта строка хешируется). Также предположим, что эти слова могут произнести десять персонажей: персонаж игрока (назовем его Дрейк), приятельница игрока (назовем ее Элена) и еще восемь вражеских персонажей (с именами в диапазоне от pirate-a до pirate-h). Нам понадобится какая-то структура данных для определения готовых звуковых ресурсов, из которых состоит эта отдельная логическая строчка диалога.

В Naughty Dog звукоинженеры описывают логические реплики с помощью языка программирования Scheme с видоизмененным синтаксисом. В представленном далее примере будем использовать аналогичный формат. Но детали реализации здесь неважны — нас интересует лишь структура данных:

```
(define-dialog-line 'line-out-of-ammo
   (character 'drake
      (lines
        drk-out-of-ammo-01   ;; "Черт побери, я пустой!"
        drk-out-of-ammo-02   ;; "Блин, нужно больше патронов."
        drk-out-of-ammo-03   ;; "Ох, вот теперь я ПО-НАСТОЯЩЕМУ разозлился."
      )
    )
    (character 'elena
      (lines
        eln-out-of-ammo-01   ;; "Помогите, я пустой!"
        eln-out-of-ammo-02   ;; "Еще остались патроны?"
      )
    )
    (character 'pirate-a
      (lines
        pira-out-of-ammo-01   ;; "Я пустой!"
        pira-out-of-ammo-02   ;; "Нужно больше боеприпасов!"
        ;; ...
      )
    )
    ;; ...
    (character 'pirate-h
      (lines
```

```
        pirh-out-of-ammo-01   ;; "Я пустой!"
        pirh-out-of-ammo-02   ;; "Нужно больше боеприпасов!"
        ;; ...
      )
    )
)
```

Обычно вместо определения реплик в виде единой монолитной структуры данных, как здесь, стоит использовать отдельные файлы для разных персонажей. Например, все реплики Дрейка могут находиться в одном файле, реплики Элены — в другом, а реплики всех пиратов — в третьем. Это поможет звукоинженерам фокусироваться на своей работе и не мешать друг другу. А также позволит более эффективно управлять памятью. Например, если в какой-то области игры нет никаких пиратов, данные с их диалогами можно выгрузить. По той же причине данные с диалогами имеет смысл разделять по уровням.

Воспроизведение строчки диалога

Имея доступ к этим данным, система диалогов может с легкостью преобразовать запрос логической реплики вроде `'line-out-of-ammo` в конкретный аудиоклип. Для этого ей нужно найти в таблице идентификатор голоса определенного персонажа и случайно выбрать одну из возможных реплик, которые он может произнести.

Также обычно имеет смысл предусмотреть механизм, который будет предотвращать слишком частое повторение реплик. Чтобы этого добиться, их индексы можно хранить в массиве, содержимое которого перемешивается случайным образом. Для выбора реплики мы просто перебираем элементы массива. Дойдя до конца, массив опять перемешивается, но так, чтобы последние произнесенные слова не оказались в самом начале. Это позволит избавиться от повторений и сделать так, чтобы реплики персонажей звучали непредсказуемо.

Строчки диалогов обычно запрашиваются кодом игрового процесса на C++, Java, C# или любом другом языке, на котором написана игра. Дизайнеры игры могут использовать для этого также скрипты (Lua, Python и т. д.). Обычно API системы диалогов пытаются сделать довольно простым. Если программисту ИИ или дизайнеру приходится лезть из кожи вон, чтобы воспроизвести какую-то реплику, персонажи в итоге могут получиться необычно молчаливыми! Поэтому интерфейс лучше сделать простым, по принципу «нажал и забыл». Оставьте тяжелую работу программисту, который пишет систему диалогов.

Например, в *Uncharted 3: Drake's Deception* код на C++ мог заставить персонажа произнести реплику путем вызова метода `PlayDialog()` из класса `Npc`. Код принятия решений в системе ИИ был усеян этими вызовами, что позволило воспроизводить подходящие строчки диалогов в ключевые моменты игры. Вот как это выглядело:

```
void Skill::OnEvent(const Event& evt)
{
    Npc* pNpc = GetSelf(); // Получаем указатель на NPC
```

```
    switch (evt.GetMessage())
    {
    case SID("player-seen"):
        // проигрываем реплику...
        pNpc->PlayDialog(SID("line-player-seen"));
        // ...и движемся к ближайшему укрытию
        pNpc->MoveTo(GetClosestCover());
        break;
    // ...
    }

    // ...
}
```

Приоритет и прерывание

Что произойдет, если персонаж, который уже разговаривает, получит запрос на проигрывание другой реплики? А если он получит несколько запросов в одном и том же кадре? В обоих случаях для разрешения неоднозначных ситуаций имеет смысл использовать *систему приоритетов*.

Чтобы реализовать такую систему, нужно назначить каждой реплике уровень приоритета. Когда приходит запрос на воспроизведение строки диалога, система проверяет приоритеты той строчки, которая произносится в текущий момент (если это действительно происходит), и той, которую запросили в данном кадре. Если текущая реплика имеет более высокий приоритет, она продолжает проигрываться, а запрос игнорируется. Если выигрывает запрошенная реплика или персонаж молчит, она будет произнесена и в случае необходимости прервет текущую речь.

Сама реализация прерывания речи не совсем тривиальна. Мы не можем выполнить плавный переход, то есть понизить уровень воспроизводимого звука и повысить уровень нового, так как применительно к отдельно взятому персонажу это будет звучать странно и неправильно. В идеале перед воспроизведением новой реплики следовало бы проиграть звук гортанной смычки. Возможно, стоит даже позволить персонажу произнести короткую фразу, свидетельствующую об удивлении тем, что его перебили, и только *потом* проигрывать новую строчку. Система диалогов в *The Last of Us* не предусматривает таких тонкостей. Она просто останавливает текущую реплику и сразу же начинает новую. В большинстве случаев это звучит довольно хорошо. Конечно, у любой игры есть уникальный подход к речи и то, что работает в одном проекте, может не сработать в другом. Поэтому, как говорится, на вкус и цвет...

Разговоры

В *The Last of Us* разработчики хотели добиться того, чтобы вражеские NPC звучали так, будто они на самом деле между собой разговаривают. Это означает, что персонажи должны уметь обмениваться довольно длинными цепочками реплик.

В *Uncharted 4: A Thief's End* и *Uncharted: The Lost Legacy* была аналогичная история: мы хотели, чтобы персонажи поддерживали беседу во время поездок на джипе по Мадагаскару и Индии. Эти беседы могли прерываться (например, когда игрок решал выйти из джипа, чтобы осмотреть окрестности) и продолжаться с того момента, на котором они были прерваны (когда игрок возвращался в машину).

Разговоры в *The Last of Us*, *Uncharted 4* и *The Lost Legacy* состоят из логических *сегментов*. Каждый из них относится к отдельной логической реплике, которую в ходе разговора произносит определенный актер, и имеет уникальный идентификатор. С помощью этих идентификаторов сегменты объединяются в разговор. В качестве примера просмотрим, как описать следующий диалог:

А: Эй, нашел что-нибудь?

Б: Нет, я уже целый час ищу, и все впустую.

А: Ну тогда заткнись и ищи дальше!

В системе диалогов Naughty Dog этот разговор можно выразить так:

```
(define-conversation-segment 'conv-searching-for-stuff-01
    :rule []
    :line 'line-did-you-find-anything
               ;; "Эй, нашел что-нибудь?"
    :next-seg 'conv-searching-for-stuff-02
)
(define-conversation-segment 'conv-searching-for-stuff-02
    :rule []
    :line 'line-nope-not-yet
               ;; "Нет, я уже целый час ищу..."
    :next-seg 'conv-searching-for-stuff-03
)
(define-conversation-segment 'conv-searching-for-stuff-03
    :rule []
    :line 'line-shut-up-keep-looking
               ;; "Ну тогда заткнись и ищи дальше!"
)
```

На первый взгляд этот синтаксис может показаться немного громоздким. Но, как мы увидим далее, такое разбиение разговора позволяет добиться отличной гибкости. Например, таким образом можно интуитивно и достаточно удобно описывать ветвящиеся диалоги.

Прерывание разговора

Раньше мы видели, как с помощью простой системы приоритетов можно справляться с прерываниями и разрешать конфликтные ситуации, когда одновременно запрашивается сразу несколько логических реплик.

Ту же систему можно использовать и для *разговоров*. Но в этом случае ее реализация должна быть чуть сложнее. Представьте, к примеру, диалог между персонажами А и Б. А что-то говорит, затем Б отвечает, в то время как А ждет своей очереди. Когда Б разговаривает, к А приходит запрос на воспроизведение совершенно

другой реплики. Формально А в этот момент молчит, поэтому согласно правилам расстановки приоритетов, которые применяются к каждому персонажу по отдельности, проблемы быть не должно и реплика произносится. Но в зависимости от контекста эти слова могут быть совершенно неуместными.

А: Эй, нашел что-нибудь?

Б: Нет, я уже целый час ищу, и...

А: Гляди, блестящий предмет! (*Неуместная реплика, прерывающая разговор*).

Б: ...Все впустую.

Чтобы решить эту проблему в *The Last of Us*, мы сделали *разговор* полноценной частью системы. Когда он инициируется, система знает, что все его участники заняты, даже если кто-то из них молчит. У каждого разговора есть приоритет, и соответствующие правила применяются к нему в целом, а не к отдельным репликам его участников. Поэтому, когда персонаж А, к примеру, получает запрос на проигрывание фразы «Гляди, блестящий предмет!», системе известно, что в этот момент он участвует в диалоге: «Эй, нашел что-нибудь?»... Строчка «Гляди, блестящий предмет!», предположительно, является такой же или менее приоритетной, чем текущий разговор, поэтому прерывание не допускается.

Если прерывающая реплика имеет более высокий приоритет (например, «Святые угодники, он в нас целится!»), ей позволено прервать существующий диалог. В этом случае будут прерваны все его участники. В итоге прерывание будет звучать естественно и осмысленно.

А: Эй, нашел что-нибудь?

Б: Нет, я уже целый час ищу, и...

А: Святые угодники, он в нас целится! *(Прерывание в связи с более приоритетным разговором)*.

Б: Держи его! *(Изначальный разговор прерван новым, персонажи А и В переключаются в боевой режим.)*

Эксклюзивность

Помимо прочего, в игре *The Last of Us* использовалась концепция *эксклюзивности*. Любые реплики или разговоры можно сделать *неэксклюзивными*, *эксклюзивными для группировки* или *глобально эксклюзивными*. Такое разделение позволяет определить, как будет прерван разговор или отдельная реплика.

- *Неэксклюзивные* реплики и разговоры могут воспроизводиться *поверх* других реплик и разговоров. Например, в поисках игрока один охотник вполне может промычать себе под нос: «Хм, здесь никого нет», а другой: «Меня это уже начинает напрягать». Они не разговаривают друг с другом, поэтому то, что эти слова произносятся одновременно, совершенно естественно.
- Реплики и разговоры, *эксклюзивные для группировки*, прерывают другие реплики и разговоры в рамках группировки персонажа. Например, если охотник в процессе поисков заметит игрока (Джоэла), то может закричать: «Он здесь!»

Другим охотникам следует немедленно прекратить разговоры, так как они в соответствии с нашим замыслом должны слышать друг друга. К тому же они должны продемонстрировать, что их коллективное внимание сосредоточено на игроке. Если в этот момент Элли, компаньон Джоэла, пытается предупредить его шепотом, нам, наверное, не стоит ее прерывать. Она не член банды охотников, и то, что она хочет сообщить Джоэлу, важно вне зависимости от того, заметили его охотники или нет.

- *Глобально эксклюзивные* реплики и разговоры прерывают любую другую речь без учета принадлежности говорящих к группировкам. Это может пригодиться в ситуациях, когда *каждый* персонаж в пределах слышимости должен отреагировать на произнесенные слова.

Возможность выбирать и ветвящиеся разговоры

Зачастую разговоры желательно организовать так, чтобы они зависели от действий игрока, решений, принимаемых ИИ-персонажами, и/или других аспектов состояния игрового мира. При создании или редактировании разговоров сценаристы и звукоинженеры предпочитают контролировать не только то, какие реплики будут произнесены, но и логические условия, определяющие ветвление разговора в тот или иной момент игрового процесса. Таким образом, творческие люди получают возможность творить без помощи программиста.

Студия Naughty Dog создала такую систему для использования в *The Last of Us*. Она была в какой-то степени навеяна проектом, разработанным ранее компанией Valve. Элан Раскин описал его в презентации *Rule Databased for Contextual Dialog and Game Logic*, представленной на конференции Game Developer's Conference в 2012 году (www.gdcvault.com/play/1015317/AI-driven-Dynamic-Dialog-through). И хотя системы диалогов, которые используют Naughty Dog и Valve, во многом существенно различаются, идеи, лежащие в их основе, похожи. Здесь будет описана система, которая применяется в Naughty Dog, так как автор лучше всего с ней знаком.

В нашей системе диалогов каждый сегмент разговора может иметь одну или несколько альтернативных реплик. У каждой альтернативы внутри сегмента есть *критерии выбора*. Если они удовлетворяются, используется соответствующая реплика, в противном случае она игнорируется.

Критериев может быть несколько. Каждый из них представляет собой простое логическое выражение, которое возвращает булево значение. В качестве примера можно привести выражения (`'health > 5`) и (`'player-death-count == 1`). Если критериев больше одного, они объединяются с использованием булева оператора И, при этом *каждый* из них должен удовлетворяться.

Далее приведен пример сегмента разговора с тремя альтернативами, зависящими от здоровья персонажа, произносящего эти слова:

```
(define-conversation-segment 'conv-player-hit-by-bullet
  (
    :rule [ ('health < 25) ]
```

```
    :line 'line-i-need-a-doctor
                ;; "У меня сильное кровотечение… нужен доктор!"
  )
  (
    :rule [ ('health < 75) ]
    :line 'line-im-in-trouble
                ;; "Вот это я влип."
  )
  (
    :rule [ ] ;; отсутствие критерия эквивалентно оператору "else"
    :line 'line-that-was-close
                ;; "Ах! Я этого больше не допущу!"
  )
)
```

Ветвящийся диалог. Разбивая разговор на сегменты, каждый из которых может содержать одну или несколько альтернативных реплик, мы получаем возможность выстраивать ветвящиеся диалоги. Рассмотрим, к примеру, разговор, во время которого Элли (компаньон игрока в *The Last of Us*) видит, что в Джоэла (персонажа игрока) стреляют, и спрашивает его, все ли с ним в порядке. Если тот не ранен, диалог звучит так.

Элли: Ты живой?

Джоэл: Ага, я в порядке.

Элли: Господи! Не высовывайся!

Если в Джоэла попали, разговор проходит иначе:

Элли: Ты живой?

Джоэл (задыхаясь): Не совсем.

Элли: У тебя кровь течет!

Этот ветвящийся диалог можно изобразить с помощью синтаксиса, описанного ранее:

```
(define-conversation-segment 'conv-shot-at--start
  (
    :rule [ ]
    :line 'line-are-you-ok ;; "Ты живой?"
    :next-seg 'conv-shot-at--health-check
    :next-speaker 'listener ;; *** см. комментарии ниже
  )
)

(define-conversation-segment 'conv-shot-at--health-check
  (
    :rule [ (('speaker 'shot-recently) == false) ]
    :line 'line-yeah-im-fine ;; "Ага, я в порядке."
    :next-seg 'conv-shot-at--not-hit
    :next-speaker 'listener ;; *** см. комментарии ниже
  )
  (
    :rule [ (('speaker 'shot-recently) == true) ]
```

```
    :line 'line-not-exactly ;; "(задыхаясь) Не совсем."
    :next-seg 'conv-shot-at--hit
    :next-speaker 'listener ;; *** см. комментарии ниже
  )
)

(define-conversation-segment 'conv-shot-at--not-hit
  (
    :rule [ ]
    :line 'line-keep-head-down ;; "Господи! Не высовывайся!"
  )
)

(define-conversation-segment 'conv-shot-at--hit
  (
    :rule [ ]
    :line 'line-youre-bleeding ;; "У тебя кровь течет!"
  )
)
```

Тот, кто говорит, и тот, кто слушает. Ветвящийся диалог, представленный здесь, имеет несколько тонких аспектов. В любой отдельно взятый момент один из двух участников беседы говорит, а другой слушает. Затем в ходе разговора они меняются местами снова и снова. В первом сегменте, `'conv-shot-at--start`, говорит Элли, а слушает Джоэл. При переходе к следующему сегменту, `'conv-shot-at--health-check`, мы указываем значение `'listener` для поля `:next-speaker`. Таким образом, система знает, что текущий слушатель (Джоэл) должен что-то сказать в следующем сегменте, поменявшись местами с Элли. В том же сегменте мы проверяем, был ли говорящий персонаж недавно ранен, используя критерии `(('speaker 'shot-recently) == false)` и `(('speaker 'shot-recently) == true)`. Поскольку очередь говорить перешла к Джоэлу, все работает так, как и ожидалось.

Подобная абстрактная система выглядит не такой уж полезной в контексте разговора между двумя ведущими персонажами вроде Джоэла и Элли. Однако определение диалога абстрактным способом делает этот подход довольно гибким. Например, то же определение можно использовать для описания разговора, в котором Джоэл спрашивает Элли, все ли с ней в порядке. Все благодаря тому, что в определении диалога не уточняется, какой именно персонаж произносит ту или иную реплику. Более того, в случае с вражескими персонажами разговоры обязательно должны быть обобщенными, поскольку мы не знаем заранее, кто из них будет разговаривать. Чтобы имитировать болтовню врагов во время боя, персонажи обычно выбираются динамически. Это должно работать независимо от того, кто именно был выбран.

В системе говорящих/слушающих персонажей число участников разговора может равняться двум-трем. Система диалогов Naughty Dog поддерживает до трех слушателей, хотя в подавляющем большинстве наших разговоров участвовали два персонажа.

Словари фактов. В критериях фигурируют символьные величины, такие как `'health` и `'player-death-count`. Они организованы в виде *словарей* — структур данных, которые, в сущности, представляют собой таблицы с парами «ключ — значение». Мы называем их *словарями фактов* (табл. 14.1).

Таблица 14.1. Пример словаря фактов

Ключ	Значение	Тип данных
'name	'ellie	StringId
'faction	'buddy	StringId
'health	82	int
'is-joels-friend	true	bool
...	...	...

Если внимательно рассмотреть таблицу, можно заметить, что каждое значение в словаре имеет тип данных. Иными словами, значения в словаре являются *вариантами*. Вариант — это объект данных, способный хранить значения разных типов по аналогии с union в C или C++. Но, в отличие от union, вариант хранит также информацию о типе данных, которые он содержит в текущий момент. Это позволяет проверять тип значений перед их использованием, а также приводить данные одного типа к другому. Например, если вариант содержит целочисленное значение `42`, мы можем сделать так, чтобы он вернул число с плавающей запятой, `42.0f`.

В *The Last of Us* у каждого персонажа был собственный словарь фактов с информацией о его здоровье, типе оружия, степени осведомленности и т. д. Свой словарь был и у каждой группировки персонажей. Это позволяло описывать факты, относящиеся к группировке в целом, — например, сколько ее участников все еще живы. Мы также применяли глобальный словарь фактов (синглтон) со сведениями об игре в целом, безотносительно к группировкам. В нем хранились такие данные, как количество времени, проведенное игроком в игре, название текущего уровня, количество попыток игрока выполнить то или иное задание и т. д.

Синтаксис критерия. При написании критерия доступен синтаксис для поименного извлечения фактов из любого словаря. Например, выражение `(('self 'health) > 5)` говорит системе о том, что она должна взять словарь персонажа, найти в нем значение факта `'health` и проверить, превышает ли оно 5. Аналогично выражение `(('global 'seconds-playing) <= 23.5)` означает, что система должна найти факт `'seconds-playing` в глобальном словаре и убедиться в том, что его значение меньше или равно 23,5 с.

Если пользователь не указал словарь явно, как в случае с `('health > 5)`, система ищет факт с заданным именем в заранее определенном порядке. Сначала проверяется словарь персонажа. Если там ничего не найдено, поиск выполняется по словарю, относящемуся к группировке персонажа. Если и он не увенчался успехом, система переходит к глобальному словарю. Такой порядок поиска позволяет звукоинженерам описывать критерии наиболее компактно, хотя код при этом получается не слишком конкретным и ясным.

Диалоги, зависящие от контекста

Работая над *The Last of Us*, мы хотели, чтобы вражеские персонажи осмысленно озвучивали местоположение игрока. Если игрок прячется на складе, они должны кричать: «Он на складе!» Если укрывается за автомобилем, должна звучать фраза: «Он за машиной!» Эту возможность довольно легко реализовать, а благодаря ей персонажи выглядят невероятно смышлеными.

Чтобы этого добиться, звукоинженеры делят игровые миры на области. Каждой из них назначаются одна или две *метки местоположения*. Первая описывает *конкретный* участок, такой как «за барной стойкой» или «на кассе», а вторая обозначает более общее место: «в магазине», «на улице» и т. д.

Чтобы определить, какую строчку диалога нужно проиграть, система проверяет, в каких областях находятся игрок и вражеский NPC. Если они в одной и той же *общей* области, для выбора реплик используется *конкретная* метка. В противном случае мы выбираем реплику с учетом метки общей области игрока. Если враг и игрок столкнулись в магазине, мы можем выбрать строчку вроде «Он у окна!». Но если NPC зашел в магазин, а игрок идет по улице, мы можем услышать: «Он на улице! Держи его!» Принцип работы этой системы проиллюстрирован на рис. 14.42.

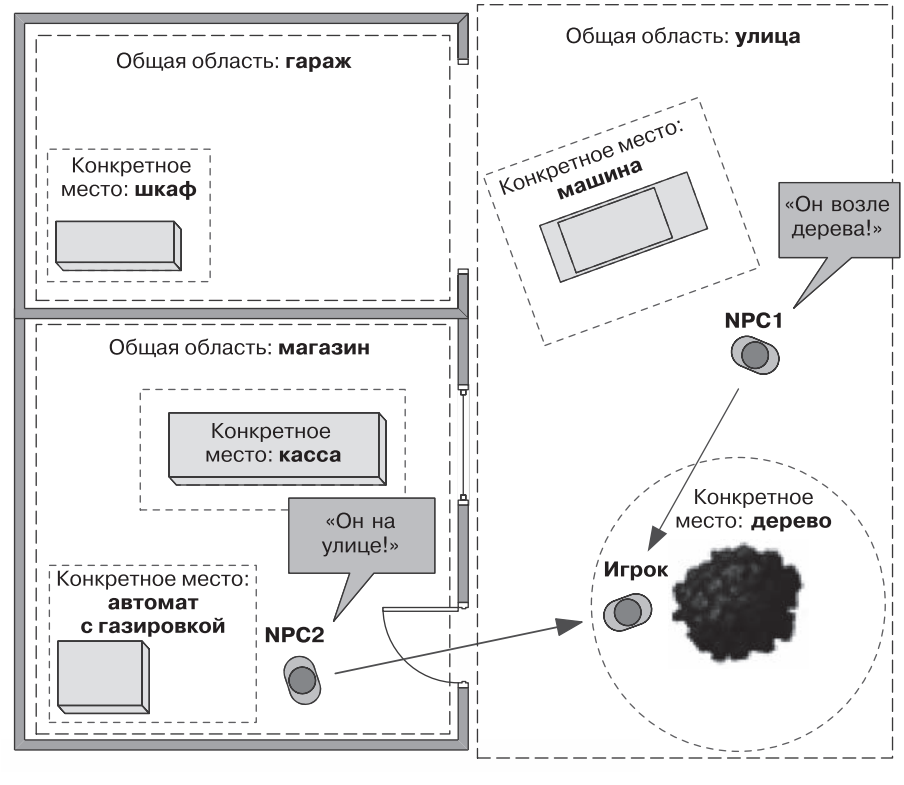


Рис. 14.42. Общие и конкретные области для выбора реплик в зависимости от контекста

Этот простой механизм оказался невероятно мощным. Его было непросто подготовить из-за огромного количества реплик и их сочетаний, которые нужно было записать и сконфигурировать, но результат, полученный в игре, стоил потраченных усилий.

Действия, связанные с диалогом

Реплики, произносимые без жестикуляции, обычно выглядят жутковато и нереалистично. Иногда, например в кинематографических сценах, речь является частью анимации персонажа. Но чаще во время разговора персонаж занимается чем-то другим: идет, бежит или стреляет. В идеале, чтобы сделать речь более живой, следует добавить какие-нибудь жесты.

В *The Last of Us* мы реализовали систему жестов, используя технологию аддитивной анимации (см. подраздел 12.6.5). Эти жесты можно было вызывать вручную в коде на C++ и скриптах. Кроме того, к каждой реплике можно было привязать скрипт, временная шкала которого синхронизировалась с издаваемым звуком. Это позволило инициировать жесты в ключевые моменты разговора.

14.6.3. Музыка

Музыка — чрезвычайно важный аспект практически любой хорошей игры. Она задает настроение, создает атмосферу напряженности. Она может сделать сцену эмоциональной или неловкой. На музыкальную систему игрового движка обычно возложены такие обязанности:

- обеспечивать возможность воспроизведения музыкальных дорожек в виде потоковых аудиоклипов, поскольку такие клипы почти всегда оказываются слишком большими, чтобы хранить их в памяти;
- обеспечивать музыкальное разнообразие;
- изменять музыку в соответствии с событиями, происходящими в игре;
- плавно переходить от одной музыкальной композиции к другой;
- смешивать музыку с другими звуками игры подходящим и приятным для слуха образом;
- позволять временно приглушать музыку, чтобы сделать более разборчивыми определенные звуки или разговоры в игре;
- разрешать временно прерывать воспроизводимую музыкальную дорожку с помощью так называемых *стингеров* — коротких аудиоклипов или звуковых эффектов;
- позволять останавливать музыку и начинать ее воспроизведение заново (игроку вряд ли захочется на протяжении всей игры слышать величественные мелодии в исполнении настоящего оркестра!).

В целом мы хотим, чтобы музыка менялась в соответствии со степенью напряженности и/или эмоциональным настроем событий, происходящих в игре.

Чтобы этого добиться, можно создать *плей-листы*, рассчитанные на различную обстановку. Каждый из них содержит один или несколько музыкальных фрагментов, которые проигрываются последовательно или в случайном порядке. Когда напряженность и настроение игры меняются (например, с началом или завершением битвы, при показе трогательных видеороликов и т. д.), музыкальная система об этом узнает и выбирает подходящие плей-листы. В некоторых играх для разных уровней напряженности предусмотрен стек музыкальных подборок: спокойная, когда поблизости нет врагов, напряженная, когда игрок приближается к ничего не подозревающей вражеской группировке, ошеломляющая, когда происходит начальный контакт, и динамичная во время боя.

Еще одним способом привязки музыки к событиям игры являются *стингеры*. Это короткие музыкальные клипы или звуковые эффекты, которые могут временно прерывать текущую музыкальную дорожку или воспроизводиться поверх нее (при этом уровень громкости основной дорожки понижается). Например, когда в поле зрения игрока впервые попадает новый враг, мы можем проиграть зловещий грохочущий звук, сообщающий о близкой опасности. А когда игрок умирает, можем быстро переключиться на фрагмент со «смертельной» музыкой. И в той и в другой ситуации следует использовать стингеры.

Плавный переход между разными звуковыми потоками — не очень простая задача. Мы не можем бездумно смешивать не связанные музыкальные фрагменты и ожидать неизменно хороших результатов. Текущая и следующая звуковые дорожки могут иметь разный темп и ритм. Главное здесь — подобрать подходящий момент для перехода. Если темп не совпадает, переход можно сделать быстрым, более длинные переходы звучат хорошо, когда темп близок к идентичному. Правильное сочетание достигается методом проб и ошибок. Даже просто чтобы как следует зациклить музыкальный фрагмент, звукоинженерам приходится выполнять тонкую настройку.

Игровая музыка — обширная тема, заслуживающая куда больше внимания. Если вы хотите лучше ее изучить, отличной отправной точкой может послужить книга [45].

Часть IV
Игровой процесс

15 Введение в системы игрового процесса

До этого момента все, о чем говорилось в данной книге, относилось к технологиям. Мы убедились в том, что игровой движок — это сложная многоуровневая программная система, работающая поверх оборудования, драйверов и ОС целевого компьютера. Увидели, как низкоуровневые системы движка предоставляют сервисы, необходимые для остальных его компонентов; как HID-устройства — джойстик, клавиатура, мышь и пр. — могут позволить игроку передавать движку входящие сигналы; как механизм отрисовки выводит на экран трехмерные изображения; как анимационные системы обеспечивают естественные движения персонажей и объектов; как система столкновений обнаруживает и обрабатывает взаимное проникновение примитивов; как физический движок обеспечивает реалистичное движение объектов; как движок трехмерного звука генерирует звуковой ландшафт с эффектом погружения в игровой мир. Но, несмотря на широкий спектр мощных возможностей, предоставляемых этими компонентами, их совокупность *по-прежнему* не позволяет получить настоящую игру!

Игра определяется не своими технологиями, а *игровым процессом*. Игровой процесс можно определить как общее впечатление от игры. Больше конкретики в эту концепцию можно внести с помощью термина «*игровая механика*», под которым обычно понимают набор *правил*, регулирующих взаимодействия между разными элементами игры. Он также определяет *цели*, стоящие перед игроком, *критерии* успеха и неудачи, *возможности* игрового персонажа, количество и виды *неигровых элементов*, существующих в виртуальном мире игры, и *общий поток* игрового процесса. Во многих играх эти элементы переплетаются с убедительным сюжетом и богатым набором персонажей. Но, как доказывают успешные игры-головоломки наподобие *Tetris*, сюжет и персонажи точно не являются обязательной частью видеоигры. В своем докладе *A Survey of 'Game' Portability* (www.dcs.shef.ac.uk/intranet/research/resmes/CS0705.pdf) Ахмед бин-Субаих, Стив Мэддок и Даниэла Романо из Шеффилдского университета называют коллекцию про-

граммных систем, используемых для реализации игрового процесса, *G-фактором* игры. В следующих главах мы рассмотрим важнейшие инструменты и движки для определения *игровой механики* (или *игрового процесса*, или *G-фактора*) и управления ею.

15.1. Структура игрового мира

Архитектура игрового процесса сильно зависит от жанра и конкретной игры. Тем не менее большинство трехмерных и значительная часть двухмерных игр в той или иной степени соответствуют нескольким структурным шаблонам проектирования. Мы обсудим эти шаблоны в следующих разделах, но, пожалуйста, помните о том, что всегда найдутся игры, которые не вписываются в эту концепцию.

15.1.1. Элементы мира

События большинства видеоигр происходят в двух- или трехмерном виртуальном *игровом мире*. Обычно он состоит из множества отдельных элементов, которые можно разделить на две категории: статические и динамические. К статическим элементам относят ландшафт, здания, дороги, мосты и практически все, что не двигается и не оказывает активного влияния на игровой процесс. Динамические элементы включают в себя персонажи, транспортные средства, оружие, плавающие бонусы и аптечки, коллекционные предметы, излучатели частиц, динамическое освещение, невидимые участки, с помощью которых отслеживаются важные игровые события, сплайны, определяющие траекторию объектов, и т. д. Структура игрового мира проиллюстрирована на рис. 15.1.

Игровой процесс обычно сосредоточен в динамических элементах игры. Конечно, компоновка статического фона имеет огромное влияние на то, какой получится игра. Например, шутер с системой укрытий будет не очень интересным, если все события происходят внутри большой пустой прямоугольной комнаты. Однако программные системы, реализующие игровой процесс, делают основной акцент на обновлении местоположения, направления и внутреннего состояния динамических объектов, так как это элементы, изменяющиеся со временем. Термин «*игровое состояние*» обозначает текущее состояние всех динамических объектов в игровом мире в целом.

Рис. 15.1. Игровой мир из The Last of Us (придуман и разработан студией Naughty Dog для PlayStation 3) со статическими и динамическими элементами (скриншот с сайта https://beedge.neocities.org)

Соотношение динамических и статических элементов тоже варьируется в зависимости от игры. Большинство трехмерных игр содержат относительно небольшое число динамических объектов, перемещающихся внутри довольно большой статической области. Другие игры, такие как аркадная классика *Asteroids* или ретрохит для Xbox 360 под названием *Geometry Wars*, не имеют никаких особых статических элементов (за исключением разве что черного фона). Динамические игровые элементы обычно требуют больше ресурсов, чем статические, поэтому в большинстве трехмерных игр их количество ограничено. Но чем выше соотношение динамических и статических объектов, тем более живым кажется пользователю игровой мир. По мере того как игровое оборудование становится все более мощным, это соотношение постепенно растет.

Следует отметить, что различие между динамическими и статическими элементами игрового мира часто оказывается не очень четким. Например, в аркадной игре *Hydro Thunder* водопады были динамическими в том смысле, что их текстуры содержали анимацию, у их основания был динамический эффект тумана и их можно было размещать в игровом мире независимо от ландшафта или поверхности воды. Однако с инженерной точки зрения водопады представляли собой статические элементы, поскольку они никак не взаимодействовали с гоночными катерами (не считая того, что скрывали от игрока секретные бонусы и проходы). Разные игровые движки по-разному проводят границу между статическими и динамическими объектами, а некоторые и вовсе их не различают, то есть любой элемент может быть динамическим.

Различие между статикой и динамикой используется в основном для оптимизации: зная, что состояние объекта не изменится, мы можем делать меньше работы. Например, если нам известно, что ограждение статическое и не меняет своего положения, для него можно заранее вычислить статические вертексы, карты освещения, карты теней, информацию об объемном свете или коэффициенты сферических гармоник для предварительного обсчета передачи сияния (precomputed radiance transfer, PRT). Почти любые вычисления, которые необходимо выполнять для динамических элементов, в случае со статикой лучше сделать заранее или вовсе пропустить.

Игры с разрушаемым окружением являются примером нечеткого разграничения статических и динамических объектов. Например, мы могли бы создать три версии каждого статического объекта: неповрежденную, поврежденную и полностью разрушенную. Большую часть времени эти фоновые объекты являются частью статического мира, но во время взрыва их можно динамически подменить, чтобы создать иллюзию повреждения. На самом деле это разделение представляет собой лишь два крайних случая в целом ряде возможных оптимизаций. Граница между двумя этими категориями (если мы ее вообще провели) смещается в зависимости от выбранных методик оптимизации и подстраивается под нужды игровой архитектуры.

Статическая геометрия

Геометрия статических элементов часто определяется с помощью таких инструментов, как Maya. Это может быть одна гигантская полигональная сетка, но ее можно разбить на отдельные части. Статические пропорции сцены иногда формируются за счет *геометрии дублирования*. Так называется метод сохранения памяти, в котором относительно небольшое количество уникальных полигональных сеток отрисовывают несколько раз в разных частях игрового мира и в разных направлениях, чтобы создать иллюзию разнообразия. Например, вы можете создать пять разных участков стены небольшого размера и затем объединить их случайным образом, сгенерировав тем самым множество уникально выглядящих стен.

Статические визуальные элементы и данные о столкновениях можно также вывести из *геометрии брашей*. Этот вид геометрии берет свое начало в движках семейства Quake. *Браш* описывает форму в виде выпуклых граней, каждая из которых привязана к набору плоскостей. Геометрию брашей быстро и легко создавать, она хорошо интегрируется в движки отрисовки, основанные на BSP-деревьях. Браши могут оказаться чрезвычайно полезными, когда нужно быстро убрать из игрового мира часть содержимого. Это делает возможным раннее тестирование игрового процесса, когда для этого не нужно много ресурсов. Если компоновка доказала свою состоятельность, команда художников может либо создать карту текстур и подстроить подходящим образом геометрию брашей, либо заменить ее более подробными ресурсами с полигональными сетками. В то же время, если уровень необходимо перепроектировать, геометрию брашей можно легко поправить, причем от команды художников не потребуется больших усилий.

15.1.2. Области игрового мира

Если виртуальный мир, в котором разворачиваются события игры, очень большой, его обычно разделяют на отдельные участки, которые называют *областями* (chunks), в русском языке чаще используются термины *«уровни»* или *«карты»*. В любой момент игрок, как правило, видит лишь несколько областей, по которым продвигается в ходе игры.

Изначально концепцию уровней придумали для того, чтобы разнообразить игровой процесс в условиях ограниченной памяти ранних игровых консолей. В память помещался только один уровень, но игрок мог перемещаться между ними, получая куда более яркие впечатления от игры. С тех пор игровая архитектура эволюционировала на множестве разных направлений, и в наши дни линейные игры, основанные на уровнях, значительно утратили популярность. Некоторые игры, в сущности, по-прежнему линейны, однако переходы между

областями не настолько очевидны для игрока, как раньше. В некоторых случаях применяется топология в виде звезды — игрок начинает в центральной области и может переходить на произвольные уровни (возможно, только после их разблокирования). Иногда используется графовая модель, в которой области соединяются друг с другом случайным образом. В то же время некоторые игры создают иллюзию огромного открытого мира, варьируя уровень детализации (level-of-detail, LOD), чтобы снизить потребление памяти и улучшить производительность.

Несмотря на разнообразие современных игровых архитектур, все игровые миры, кроме самых крошечных, по-прежнему делятся на какого-то рода области. Это вызвано рядом причин. Прежде всего тем, что объем памяти все еще является важным ограничением (и это не изменится до тех пор, пока на рынке не появятся устройства с бесконечной памятью!). Кроме того, области игрового мира служат удобным механизмом для управления общим продвижением по игре. Они могут также выступать средством разделения труда: за создание каждой области и управление ею может отвечать относительно небольшая группа дизайнеров и художников. Области игрового мира проиллюстрированы на рис. 15.2.

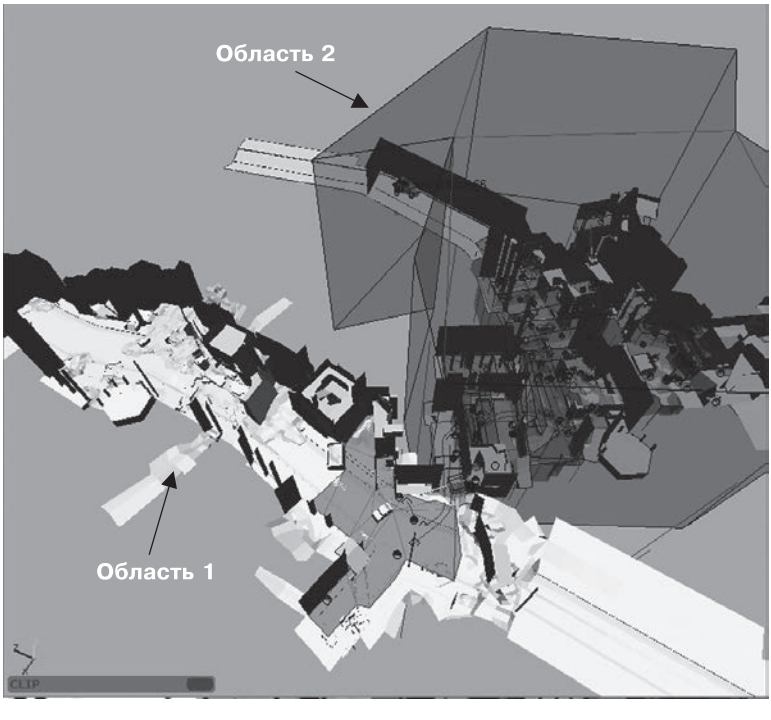


Рис. 15.2. Многие игровые миры разделяют на области. Это вызвано разными причинами: ограниченностью памяти, необходимостью контролировать прохождение игры или потребностью в разделении труда во время разработки

15.1.3. Высокоуровневый игровой поток

Высокоуровневый поток игры определяет последовательность, дерево или граф *целей* для игрока. Цели иногда называют *задачами*, *стадиями*, *уровнями* (термин, применимый и к областям игрового мира) или *волнами*, если игра в основном заключается в борьбе с полчищами наступающих врагов. Высокоуровневый поток также определяет, как выглядит успешное достижение цели (например, «избавиться от всех врагов и заполучить ключ») и наказание за провал (например, «вернуться в начало текущей области, возможно, с потерей жизни в процессе»). В сюжетных играх этот поток может включать в себя встроенные видеоролики, которые помогают игроку лучше понять развитие сюжета. Такие ролики называют *внутриигровым видео* или *неинтерактивными сценами*. Сцены, отрисованные заранее в полноэкранном формате, обычно обозначаются аббревиатурой FMV (fullmotion video — «полностью подвижное видео»).

Ранние игры привязывали достижение целей непосредственно к игровым областям (отсюда и двойное значение термина «уровень»). Например, в *Donkey Kong* каждый новый уровень ставит перед Марио новую цель — достичь вершины структуры и перейти на следующий уровень. Однако в современной игровой разработке линейная зависимость между игровыми областями и целями не слишком популярна. Каждая цель связана с одной или несколькими областями игрового мира, но эта связь сознательно делается опосредованной. Гибкий подход позволяет изменять цели игры независимо от разделения игрового мира, что крайне полезно с логистической и практической точек зрения при разработке игры. Многие игры объединяют цели в более крупные разделы игрового процесса, которые часто называют *главами* или *актами*. Архитектура типичного игрового процесса показана на рис. 15.3.

15.2. Реализация динамических элементов: игровые объекты

Динамические элементы игры обычно проектируются в объектно-ориентированном стиле. Это интуитивно понятный и естественный подход, который хорошо ложится на то, как дизайнер игры представляет себе устройство виртуального мира. Он может визуализировать персонажей, транспортные средства, парящие аптечки, взрывающиеся бочки и множество других динамических объектов, движущихся в игре. Поэтому желание создавать эти объекты и манипулировать ими в редакторе игрового мира совершенно естественно. Точно так же для программистов вполне естественно реализовывать динамические элементы в виде во многом автономных агентов, которые создаются на этапе выполнения. В этой книге практически любой динамический элемент игрового мира мы будем называть *игровым объектом* (ИО). Но в данной индустрии этот термин вовсе не является стандартным. Игровые объекты часто называют *сущностями*, *акторами* или *агентами*, и это далеко не полный список.

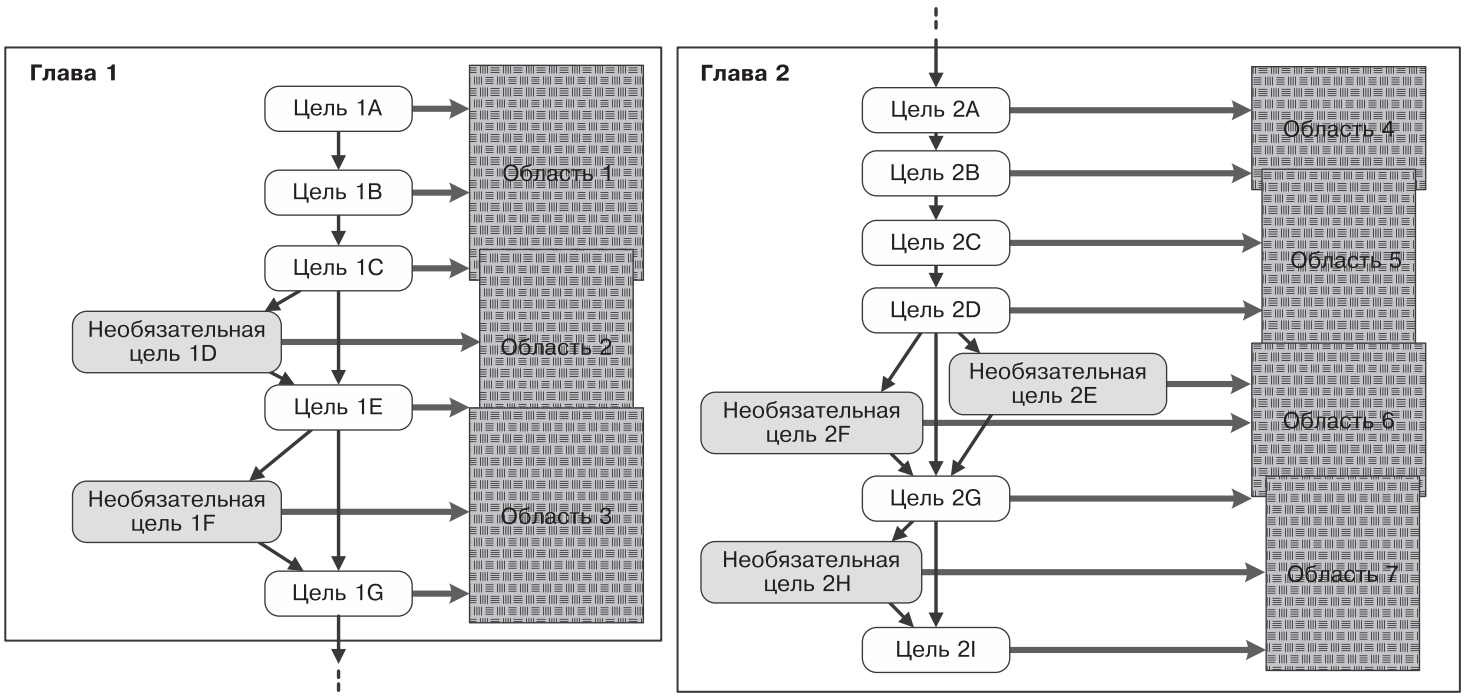


Рис. 15.3. Цели игрового процесса обычно организованы в виде последовательности, дерева или обобщенного графа, и каждая из них относится к одной или нескольким областям игрового мира

Как принято в объектно-ориентированном дизайне, игровой объект, в сущности, является набором *атрибутов* (его текущее состояние) и *поведений* (как это состояние меняется со временем и в ответ на события). Игровые объекты обычно классифицируются по *типу*. Разные типы объектов имеют разную структуру атрибутов и разное поведение. Все *экземпляры* определенного типа имеют одну и ту же структуру атрибутов и один набор поведений, но *значения* этих атрибутов могут разниться (следует отметить, что если *поведение* игрового объекта зависит от данных, которые задаются, скажем, через скриптовый код или набор правил, определяющих реакцию объекта на события, то поведение тоже может различаться от экземпляра к экземпляру).

Если взять *экземпляр* и *тип*, то ключевым является последний. Например, в игре *Pac-Man* есть четыре типа игровых объектов: привидения, шарики, таблетки силы (power pill) и Pac-Man. Однако в любой момент на экране может быть до четырех экземпляров типа «привидение», 50–100 экземпляров типа «шарик», четыре экземпляра таблеток силы и один экземпляр типа Pac-Man.

Большинство объектно-ориентированных систем имеют какой-то механизм *наследования* атрибутов и/или поведения. Наследование поощряет повторное использование кода и дизайна. То, как именно это работает, зависит от конкретной игры, однако наследование поддерживается в том или ином виде в большинстве игровых движков.

15.2.1. Объектные модели игры

В информатике термин «*объектная модель*» имеет два связанных между собой, но отдельных значения. Это может быть набор возможностей, предоставляемых определенным языком программирования или формального проектирования. В качестве примера можно привести *объектные модели* C++ или OMT. Также это может быть конкретный интерфейс объектно-ориентированного программирования (то есть набор классов, методов и взаимных связей, созданный для решения определенной проблемы). Одним из примеров может служить *объектная модель Microsoft Excel*, которая позволяет внешним программам управлять Excel различными способами (подробное обсуждение термина «*объектная модель*» вы найдете по адресу en.wikipedia.org/wiki/Object_model).

В этой книге мы используем термин «*объектная модель игры*» для описания механизмов, с помощью которых игровой движок позволяет моделировать и симулировать динамические элементы в виртуальном игровом мире. В этом смысле данный термин объединяет в себе аспекты *обоих* определений, приведенных ранее.

- Объектная модель игры — это конкретный интерфейс объектно-ориентированного программирования, предназначенный для решения определенной проблемы или симуляции определенного набора элементов, из которых состоит определенная игра.
- Вместе с тем объектная модель игры часто расширяет язык программирования, на котором написан ее движок. Если игра реализована на процедурном языке, таком как C, объектно-ориентированные механизмы могут быть добавлены самими

программистами. Но даже если игра написана на объектно-ориентированном языке вроде C++, такие продвинутые возможности, как рефлексия, постоянное хранение и сетевая репликация, часто добавляются отдельно. Объектная модель игры иногда сочетает в себе возможности нескольких языков. Например, игровой движок может объединять компилируемый язык, такой как C или C++, со скриптовым языком наподобие Python, Lua или Pawn, предоставляя унифицированную объектную модель, с которой можно работать в обоих языках.

15.2.2. Игровые объекты на этапах проектирования и выполнения

Объектная модель, доступная дизайнерам в редакторе игрового мира (обсуждается в разделе 15.4), может отличаться от той, которая используется для реализации игры на этапе выполнения.

- Инструментальная объектная модель игры может быть реализована во время выполнения с помощью языка, не поддерживающего объектно-ориентированную парадигму, такого как C.
- Тип ИО, созданный на этапе проектирования, может быть реализован во время выполнения в виде набора классов, а не одного класса, как можно было бы ожидать.
- Каждый ИО, созданный на этапе проектирования, во время выполнения может представлять собой лишь уникальный идентификатор, тогда как все его состояние может храниться в таблицах или наборах слабо связанных между собой объектов.

Таким образом, у игры есть две отдельные, но тесно связанные между собой объектные модели.

- *Объектная модель этапа проектирования* представляет собой набор *типов игровых объектов*, которые дизайнер видит в своем редакторе игрового мира.
- *Объектная модель этапа выполнения* состоит из тех языковых конструкций и программных систем, которые программисты использовали для ее реализации. Объектная модель этапа выполнения может совпадать с моделью этапа проектирования или накладываться на нее, но эти модели могут быть и совершенно разными.

В некоторых игровых движках граница между моделями этапов проектирования и выполнения размыта или вовсе отсутствует. В других она ярко выражена. Иногда реализация разделяется между инструментарием и средой выполнения. А иногда итоговая реализация выглядит совершенно иначе по сравнению с тем, как она представлена в редакторе. Некоторые аспекты этой реализации почти всегда просачиваются в оригинальный дизайн. Дизайнеры должны учитывать, как будут влиять создаваемые ими игровой мир, правила игрового процесса и поведение объектов на производительность и потребление памяти. Тем не менее практически все игровые

движки предоставляют какой-то вид объектной модели на этапе проектирования и ее соответствующую реализацию для среды выполнения.

15.3. Игровые движки на основе данных

На ранних этапах развития игровой индустрии игры в основном писались вручную. Инструментарий если и существовал, то был примитивным. Это работало, так как игры обычно были крошечными и планка была установлена не слишком высоко. Это отчасти было вызвано тем, что раннее игровое оборудование было способно лишь на примитивные графику и звук.

В наши дни сложность игр возросла на порядки, а стандарт качества так высок, что игры часто сравнивают с компьютерными спецэффектами для голливудских блокбастеров. Игровые студии значительно увеличились, но объемы игрового контента растут еще быстрее. С выходом консолей восьмого поколения, таких как Xbox One и PlayStation 4, разработчики регулярно говорят о том, что им приходится производить в десять раз больше контента, чем для предыдущего поколения, в то время как размеры их команд не претерпели радикальных изменений. Ввиду этой тенденции игровые студии должны быть способны выпускать огромные объемы контента крайне эффективным образом.

Инженерные ресурсы часто становятся узким местом в процессе производства, так как количество высококачественных программистов ограничено, а их финансовые запросы высоки, к тому же производительность программистов обычно намного меньше производительности художников и дизайнеров игры, что связано со сложностью компьютерного программирования. В большинстве студий сейчас считают, что создание как минимум части контента следует доверить людям, которые отвечают за этот процесс, — дизайнерам и художникам. Когда поведение игры частично или полностью определяется *данными*, которые генерируют художники и дизайнеры, а не только *программным обеспечением*, созданным программистами, можно сказать, что движок *основан на данных*.

Архитектуры, основанные на данных, могут повысить эффективность командной работы за счет полноценного вовлечения всех сотрудников и уменьшения нагрузки на программистов. Они также могут увеличить *темпы разработки*. То, что разработчик вносит небольшое исправление в содержимое игры или полностью переделывает целый уровень, позволяет ему быстро увидеть результаты внесенных изменений и при этом в идеале не обращаться за помощью к программисту. Это экономит ценное время и позволяет команде повысить качество игры до очень высокого уровня.

Тем не менее важно понимать, что за размещение данных во главе угла приходится платить высокую цену. Чтобы дизайнеры и художники могли создавать игровой контент, им необходимо дать подходящие инструменты. Нужно поменять код среды выполнения, чтобы он умел эффективно обрабатывать широкий спектр входных данных. Отдельный инструментарий должен быть предусмотрен и в самой игре, чтобы художники и дизайнеры могли видеть результаты своей работы

и отлаживать проблемы. На написание, тестирование и поддержку всего этого программного обеспечения придется потратить много времени.

К сожалению, многие студии поспешно выбирают архитектуру на основе данных, не потрудившись изучить, как она повлияет на конкретный игровой дизайн и специфические нужды членов их команды. В итоге они могут перестараться и создать переусложненные инструменты и системы движка, которые трудно использовать, полные ошибок и практически неспособные адаптироваться к изменчивым требованиям проекта. Ирония в том, что, стремясь воспользоваться преимуществами архитектуры на основе данных, команда демонстрирует намного меньшую производительность, чем та, которую обеспечивали старомодные методы с ручным написанием кода.

У любого игрового движка должны быть какие-то компоненты, основанные на данных, но при выборе этих компонентов следует проявлять чрезвычайную осторожность. Очень важно взвесить стоимость создания функции, основанной на данных или ускоряющей темпы разработки, и то количество времени, которое она должна сэкономить в процессе работы над проектом. Также при проектировании и реализации инструментов и систем движка на основе данных всегда следует помнить о принципе KISS (keep it simple, stupid). Перефразируя Альберта Эйнштейна: все в игровом движке должно быть настолько просто, насколько это возможно, но не проще.

15.4. Редактор игрового мира

Мы уже обсуждали инструменты для создания ресурсов на основе данных, такие как Maya, Photoshop, Havok и т. д. Они генерируют отдельные ресурсы, которые затем потребляют движок отрисовки, система анимации, аудиосистема, физический движок и т. д. Аналогом этих инструментов в сфере игрового процесса является *редактор игрового мира* — программа (или набор программ), которая позволяет создавать уровни и наполнять их статическими и динамическими элементами.

Все коммерческие игровые движки предоставляют редактор игрового мира в том или ином виде.

- Широко известный инструмент под названием *Radiant* используется для создания карт в движках семейства *Quake* и *Doom*. Снимок окна Radiant показан на рис. 15.4.
- Движок *Source* от компании Valve, лежащий в основе таких игр, как *Half-Life 2*, *The Orange Box*, *Team Fortress 2*, цикла *Portal*, цикла *Left 4 Dead* и *Titanfall*, предоставляет редактор под названием *Hammer* (ранее известный как *Worldcraft* и *The Forge*). Снимок окна Hammer показан на рис. 15.5.
- CRYENGINE от Crytek предоставляет мощный пакет инструментов для создания и редактирования игровых миров. Эти инструменты поддерживают редактирование в реальном времени сразу в нескольких игровых средах на разных платформах как в двухмерном, так и в настоящем стереоскопическом трехмерном режиме. Редактор Sandbox от Crytek изображен на рис. 15.6.

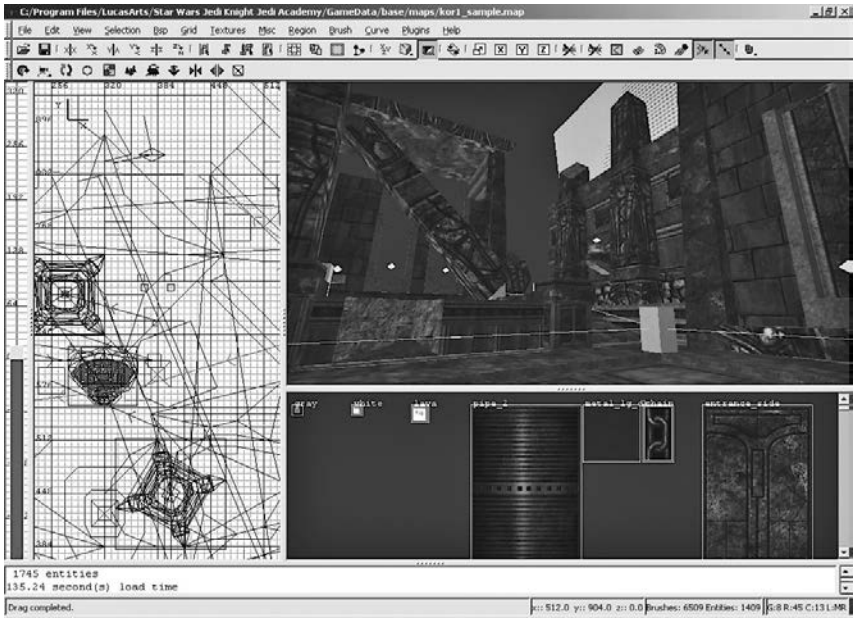


Рис. 15.4. Редактор игрового мира Radiant для движков семейства Quake и Doom

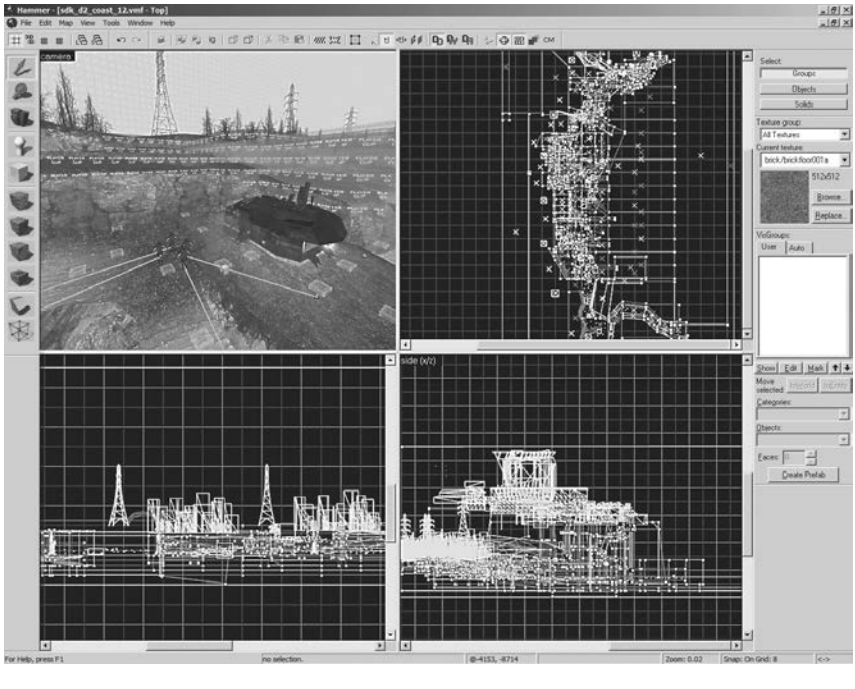


Рис. 15.5. Редактор Hammer от Valve для движка Source

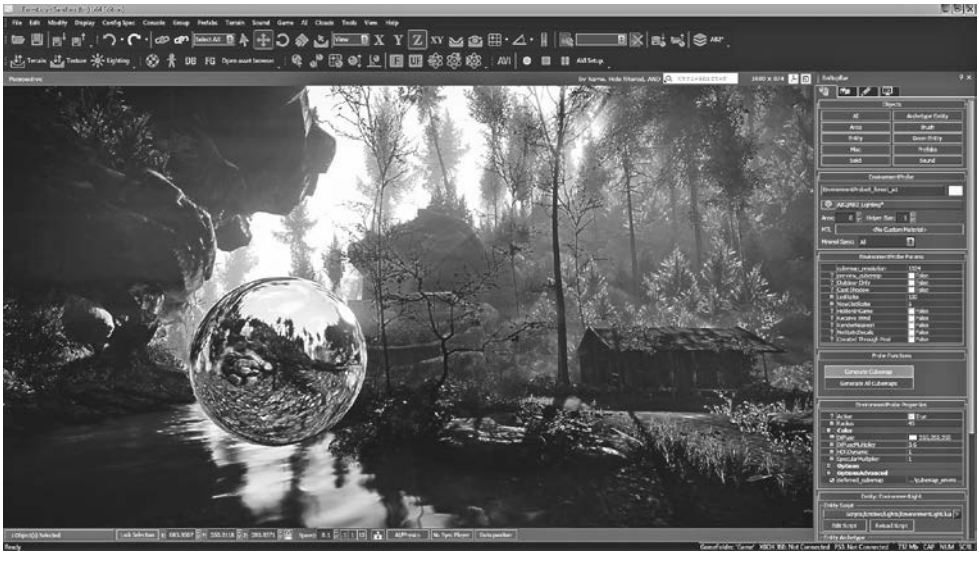


Рис. 15.6. Редактор Sandbox для CRYENGINE

Редактор игрового мира обычно позволяет задавать начальные состояния игровых объектов, то есть значения их атрибутов. Большинство редакторов также дают пользователям возможность тем или иным образом контролировать *поведение* динамических элементов. Контроль может быть реализован в виде конфигурационных параметров на основе данных (например, объект А изначально должен быть невидимым, объект Б должен атаковать игрока сразу после создания, объект В может гореть и т. д.), а поведением можно управлять с помощью скриптового языка, что сдвигает область ответственности дизайнера игры в мир программирования. Некоторые редакторы даже позволяют определять совершенно новые типы игровых объектов с минимальным вмешательством со стороны программистов или вообще без него.

15.4.1. Типичные возможности редактора игрового мира

Устройство и внешний вид редакторов игрового мира могут сильно разниться, но большинство из них предоставляют довольно стандартный набор возможностей. Некоторые из них рассмотрены далее.

Создание игровых областей и управление ими

Единицей создания игрового мира обычно является область (также известная как уровень или карта — см. подраздел 15.1.2). Редактор игрового мира, как правило, позволяет создавать новые области, переименовывать, разбивать на части, объ-

единять и удалять существующие. Каждую область можно связать с одним или несколькими статическими мешами и/или другими статическими элементами данных, такими как навигационные карты ИИ, описания выступов, за которые может ухватиться игрок, определения точек укрытия и т. д. В некоторых движках область определяется единственным фоновым мешем и не может без него существовать. Бывают движки, которые допускают независимое существование областей, они могут определяться ограничивающими многогранниками (такими как AABB, OBB или произвольными полигональными участками) и заполняться любым количеством мешей и/или брашей (см. подраздел 1.7.2).

Некоторые редакторы предоставляют отдельные инструменты для создания ландшафта, воды и других специальных статических элементов. Иногда для их редактирования используются стандартные DCC-приложения, при этом их помечают так, чтобы конвейер обработки ресурсов и/или движок выполнения знали, что они особенные (например, в играх из циклов *Uncharted* и *The Last of Us* вода была выполнена в виде полигонального меша, на который накладывался специальный материал, сигнализирующий о том, что его следует воспринимать как воду). Иногда специальные элементы игрового мира создаются и редактируются с помощью отдельного независимого инструмента. Например, для создания поля высот ландшафта в *Medal of Honor: Pacific Assault* применялась видоизмененная версия инструмента, доставшегося от другой команды внутри Electronic Arts. Это было более разумно, чем пытаться интегрировать редактор ландшафта в Radiant — редактор игрового мира, который в то время задействовался в проекте.

Визуализация игрового мира

Важно, чтобы пользователь редактора игрового мира имел возможность визуализировать его содержимое. В связи с этим практически любой подобный редактор позволяет просматривать игровой мир в трехмерном представлении с перспективой и/или в виде двухмерной ортографической проекции. Панель просмотра часто делится на четыре секции: по одной для разрезов сверху, сбоку и спереди и еще одна — для трехмерного представления с перспективой.

Некоторые редакторы позволяют просматривать игровой мир с помощью собственного встроенного движка отрисовки. Другие сами интегрированы в редакторы трехмерной геометрии наподобие Maya или 3ds Max, чтобы иметь возможность использовать их панели просмотра. Бывают такие редакторы, которые умеют взаимодействовать с игровым движком и применять его для отрисовки трехмерного представления с перспективой. Некоторые из них даже интегрированы в сам движок.

Навигация

Очевидно, что редактор, который не умеет перемещаться по игровому миру, был бы довольно бесполезным. В ортографическом представлении важно иметь возможность прокрутки и масштабирования. В трехмерном представлении используются различные методы управления камерой. У вас должна быть возможность фокусироваться на отдельном объекте и вращаться вокруг него. Вам также

должен быть доступен режим наблюдения, в котором камера вращается вокруг собственной точки фокуса и может двигаться вперед, назад, вверх и вниз и поворачиваться влево и вправо.

Некоторые редакторы предоставляют множество удобных средств навигации. К ним относятся возможности выделять объекты и фокусироваться на них одним нажатием клавиши, сохранять различные важные положения камеры и затем переходить между ними, разнообразные режимы движения для быстрой навигации и точного управления камерой, история навигации в стиле веб-браузера, с помощью которой можно перемещаться по игровому миру, и т. д.

Выделение

Основная задача игрового мира — помочь пользователю наполнить этот мир статическими и динамическими элементами. Поэтому важно, чтобы он мог выбирать отдельные объекты для редактирования. Некоторые редакторы поддерживают только одиночное выделение, а более продвинутые продукты позволяют выбрать сразу несколько элементов. Объект можно выделять в ортографическом или трехмерном представлении, используя инструмент rubber-band box и соответственно метод рейкастинга. Многие редакторы также выводят все элементы игрового мира в виде прокручиваемого списка или дерева, чтобы их можно было искать и выбирать по имени. В некоторых продуктах выделение можно пометить и сохранить для дальнейшей работы.

Игровые миры часто забиты под завязку. Поэтому иногда выбрать нужный объект оказывается непросто, поскольку он закрыт другими объектами. Эта проблема решается разными путями. Если вы находитесь в трехмерном режиме и используете рейкастинг, то можете перебрать все объекты, которые пересекаются лучом, а не просто выделить ближайший из них. Многие редакторы позволяют временно скрывать из виду выделенные объекты. Таким образом, если вам не удалось получить нужный объект с первого раза, можете скрыть то, что оказалось на пути, и попробовать еще раз. Как вы увидите в следующем разделе, слои тоже могут быть эффективным механизмом для наведения порядка и облегчения процесса выбора объектов.

Слои

Некоторые редакторы позволяют группировать объекты в заранее определенные пользовательские *слои*. Эта возможность невероятно полезна, так как позволяет организовывать содержимое игрового мира логичным образом. Вы можете скрывать и показывать целые слои, чтобы уменьшить беспорядок на экране. Слои могут быть разноцветными, чтобы их проще было распознавать. Этот механизм — важная часть стратегии разделения труда. Например, когда команда, отвечающая за освещение, работает над какой-то областью игрового мира, она может скрыть все элементы сцены, которые не относятся к свету.

Кроме того, если редактор игрового мира умеет загружать и сохранять отдельные слои, это позволяет избегать конфликтов, когда несколько специалистов одновременно работают над одним и тем же уровнем. Например, все освещение

может храниться в одном слое, вся фоновая геометрия — в другом, а все персонажи ИИ — в третьем. Поскольку слои автономны, команды, отвечающие за освещение, фон и NPC, могут работать одновременно над одной и той же областью игрового мира.

Таблица свойств

Статические и динамические элементы, наполняющие область игрового мира, обычно обладают разными свойствами (атрибутами), которые пользователь может редактировать. Это могут быть простые пары вида «ключ — значение», ограниченные элементарными атомарными типами данных, такими как булевы значения, целые и дробные числа, а также строки. В некоторых редакторах поддерживаются более сложные свойства, включая массивы и вложенные составные структуры данных. Могут поддерживаться еще более сложные типы данных, такие как векторы, RGB-цвета и ссылки на внешние ресурсы (аудиофайлы, меши, анимации и т. д.).

Большинство редакторов игрового мира отображают атрибуты текущего выделенного объекта (или объекты) в виде таблицы свойств с прокруткой (рис. 15.7). Она позволяет пользователю просматривать текущие значения каждого атрибута

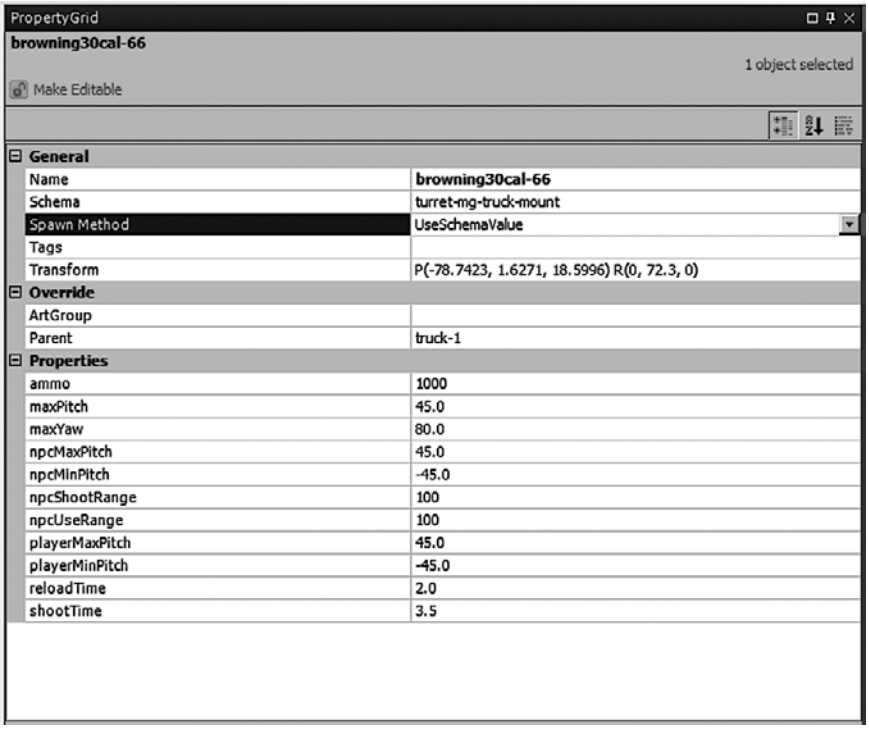


Рис. 15.7. Типичная таблица свойств

и редактировать их путем ввода текста, установки флажков, выбора раскрывающихся списков, прокрутки счетчика и т. д.

Редактирование нескольких выбранных объектов. Редакторы, поддерживающие множественное выделение, позволяют редактировать сразу несколько объектов. На экран выводится список всех атрибутов каждого объекта в выделении. Если атрибут имеет одно и то же значение для всех объектов, оно отображается как есть и его обновление применяется к атрибутам всех выбранных объектов. Если атрибут имеет разные значения для разных объектов, таблица свойств обычно оставляет его пустым. Если ввести в поле таблицы новое значение, оно перезапишет значения всех выбранных объектов, приводя их к единому виду.

Еще одна проблема возникает в ситуации, когда выделение содержит неоднородный набор объектов, то есть объектов разных типов. У каждого типа может быть свое уникальное множество атрибутов, поэтому таблица свойств должна отображать только атрибуты, общие для всех объектов в выделении. Это все равно может оказаться полезным, так как типы игровых объектов часто наследуются от одного и того же базового типа. Например, у большинства объектов есть позиция и положение в пространстве. В неоднородных выделениях пользователь по-прежнему может редактировать эти общие атрибуты, хотя более специфические свойства временно скрываются.

Произвольные свойства. Обычно набор свойств, принадлежащих объекту, и типы данных этих свойств определяются на уровне типа объекта. Например, у отображаемого объекта есть позиция, положение в пространстве, масштаб и меш, тогда как у освещения есть позиция, положение в пространстве, свет, интенсивность и тип света. Некоторые редакторы также позволяют пользователям определять произвольные свойства для отдельных экземпляров. Эти свойства обычно собраны в плоский список пар «ключ — значение». Пользователь может выбрать имя (ключ) каждого произвольного свойства, его тип и значение. Это может быть крайне полезно в ходе создания прототипов новых возможностей игрового процесса или реализации одноразовых сценариев.

Вспомогательные средства для размещения и выравнивания объектов

Редактор игрового мира по-особому относится к некоторым свойствам объектов. Обычно позицию, положение в пространстве и масштаб можно изменять с помощью специальных инструментов в ортографических и трехмерных окнах просмотра (как в Maya или Max). Кроме того, для связывания ресурсов тоже часто требуются специальные средства. Например, если мы поменяем меш, связанный с объектом игрового мира, редактор должен отобразить этот меш в ортографическом и трехмерном представлениях. Таким образом редактор должен относиться к этим свойствам по-особенному — он не может работать с ними, как с большинством других свойств объекта.

Многие редакторы игрового мира в дополнение к базовым инструментам для перемещения, вращения и масштабирования предоставляют целый ряд вспомогательных средств для размещения и выравнивания объектов. Многие из этих возможностей в значительной степени позаимствованы из коммерческих продуктов для 3D-моделирования и работы с графикой, таких как Photoshop, Maya, Visio и пр. Это относится к привязке к сетке, привязке к ландшафту, выравниванию по объекту и многому другому.

Специальные типы объектов

Особого отношения к себе со стороны редактора игрового мира требуют не только некоторые свойства, но и определенные типы объектов, включая следующие.

- *Источники света*. Источники света не содержат меш, поэтому в редакторах игрового мира они обычно представлены специальными значками. Редактор может также попытаться передать приблизительное воздействие источников света на геометрию сцены, чтобы дизайнеры могли, перемещая их в режиме реального времени, хорошо представлять, как сцена будет выглядеть в целом.
- *Излучатели частиц*. Визуализация эффектов частиц тоже может быть проблематичной, если редактор основан на автономном движке отрисовки. В таких случаях излучатели частиц могут быть показаны лишь в виде значков, хотя возможно также попробовать эмулировать соответствующий эффект в самом редакторе. Конечно, если редактор встроен в игру или способен взаимодействовать с запущенной игрой для изменения сцены в реальном времени, это не составит проблемы.
- *Источники звука*. Как обсуждалось в главе 14, трехмерный движок отрисовки моделирует источники звука в виде 3D-точек или многогранников. Было бы удобно иметь для этого специальные инструменты внутри редактора игрового мира. Например, звуковые дизайнеры оценили бы возможность визуализации максимального радиуса всенаправленного излучателя звука или вектор направления и конус, если речь идет о направленном излучателе.
- *Регион*. Регион — это объем пространства, который используется игрой для обнаружения важных событий (например, того, что объект входит в какое-то пространство или покидает его) или разграничения участков для разных задач. Некоторые игровые движки позволяют моделировать регионы только в виде сфер или направленных параллелепипедов, а в других для этого можно применять произвольные выпуклые полигональные примитивы при просмотре сверху, со строго горизонтальными сторонами. Иногда регионы могут использовать более сложную геометрию, такую как k-DOP (см. подраздел 13.3.4). Если регионы всегда сферические, дизайнеры, возможно, смогут обойтись атрибутом Радиус в таблице свойств, но для определения и редактирования размеров региона

произвольной формы почти наверняка потребуется узкоспециализированный инструмент.

- *Сплайны.* Сплайн — это трехмерная кривая, определяемая набором контрольных точек и, возможно, их касательных векторов в зависимости от того, какой вид математических кривых вы используете. Сплайны Кэтмелла — Рома применяются чаще всего, так как они полностью определяются набором контрольных точек (без касательных), через каждую из которых проходит кривая. Но вне зависимости от того, какой тип сплайнов поддерживается, редактор игрового мира должен иметь возможность выводить их в окнах обзора, а у пользователя должны быть средства для манипулирования отдельными контрольными точками. Некоторые редакторы поддерживают два режима выделения: Общий для выбора объектов сцены и Точный для выбора отдельных компонентов выделенного объекта, таких как контрольные точки сплайна или вершины региона.
- *Навигационные меши для ИИ.* Во многих играх для навигации NPC используются алгоритмы нахождения пути внутри доступных для перемещения регионов игрового мира. Эти регионы нужно определить заранее, и редактор обычно является ключевым инструментом, с помощью которого дизайнеры ИИ создают, визуализируют и редактируют эти регионы. Например, *навигационный меш* — это двухмерный треугольный меш, предоставляющий системе поиска пути простое описание границ региона, доступного для перемещения, а также информацию о связях.
- *Другие дополнительные данные.* Очевидно, что у каждой игры есть собственные требования к данным. Редактор игрового мира может предоставить для этих данных дополнительные средства визуализации и редактирования. В качестве примера можно привести описание вспомогательных элементов — окон, дверных проемов, возможных точек атаки или защиты — в рамках игрового пространства для системы ИИ или геометрические свойства, описывающие такие элементы, как точки укрытия или выступы, предназначенные для использования персонажем игрока и/или NPC.

Сохранение и загрузка областей игрового мира

Конечно, редактор игрового мира нельзя назвать полноценным, если он не умеет загружать и сохранять игровые области. Детализация загрузки и сохранения областей существенно варьируется от движка к движку. Некоторые движки хранят каждую область в отдельном файле, тогда как другие позволяют загружать и сохранять отдельные слои. Форматы данных, используемые в разных движках, тоже различаются: одни могут быть двоичными, другие — текстовыми, как XML или JSON. У каждого подхода есть свои плюсы и минусы, однако возможность загрузки и сохранения областей игрового мира в том или ином виде присутствует во всех

редакторах, и любой игровой движок способен загружать эти области на этапе выполнения, чтобы в них можно было играть.

Быстрое внесение изменений

Хороший редактор игрового мира обычно поддерживает какое-то динамическое обновление для быстрой правки. Некоторые редакторы работают внутри самой игры, позволяя пользователю сразу же видеть последствия внесенных им изменений. Иногда редактор напрямую подключается к запущенной игре. Но большинство из них работают полностью автономно или как дополнения для DCC-приложений, таких как Lightwave или Maya. Эти инструменты иногда позволяют перезагружать измененные данные динамически, прямо в запущенной игре. То, как именно это реализовано, не так уж важно — главное, что пользователи получают довольно короткий *цикл внесения изменений* (время между внесением изменения в игровой мир и проявлением его последствий в игре). Следует понимать, что этот цикл не должен быть мгновенным. Время внесения изменений должно соответствовать их масштабу и частоте. Например, мы можем ожидать, что модификация максимального уровня здоровья персонажа будет очень быстрой операцией, но при существенных изменениях среды освещения для целой игровой области можем смириться с куда более продолжительным временем ожидания.

15.4.2. Интегрированные средства управления ресурсами

В некоторых движках редактор игрового мира интегрируется с другими аспектами управления базой данных ресурсов, такими как определение свойств меша и материала, создание анимации, деревья смешивания, анимационные конечные автоматы, настройка свойств объекта, связанных с физикой и столкновениями, управление текстурами и т. д. (обсуждение базы данных игровых ресурсов приведено в подразделе 7.2.1).

Возможно, наиболее известным примером реализации такого подхода является редактор *UnrealEd*, используемый для создания контента игр, основанных на Unreal Engine. UnrealEd интегрирован непосредственно в игровой движок, поэтому любые изменения, сделанные в редакторе, применяются к динамическим объектам прямо в запущенной игре. Это позволяет легко достичь высоких темпов внесения изменений. Но UnrealEd — это далеко не только редактор игрового мира, на самом деле это полноценный пакет для создания контента. Он управляет всей базой данных игровых ресурсов, включая анимации, аудиоклипы, треугольные меши, текстуры, материалы, шейдеры и многое другое. UnrealEd предоставляет своим пользователям унифицированный визуальный редактор, который позволяет просматривать и изменять любые ресурсы в базе данных в режиме реального времени, становясь отличным подспорьем для быстрого и эффективного процесса разработки. Снимки нескольких окон UnrealEd показаны на рис. 15.8 и 15.9.

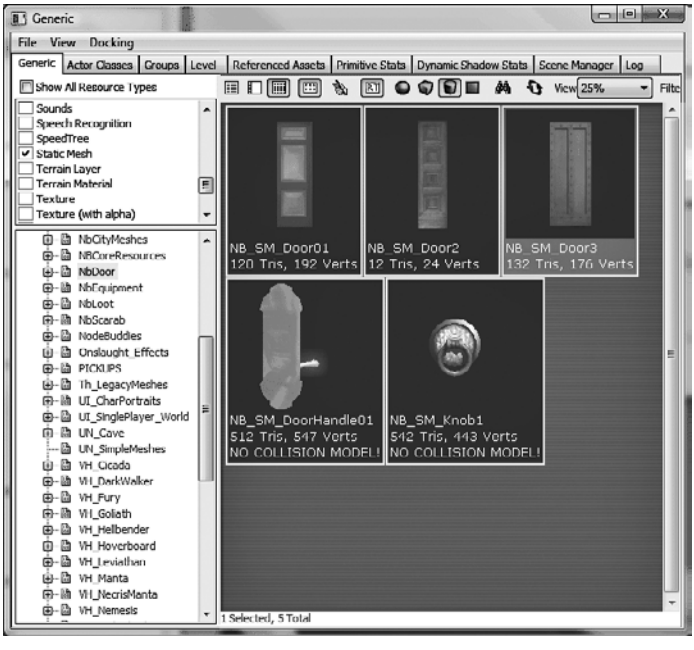


Рис. 15.8. UnrealEd's Generic Browser дает доступ к базе данных всех ресурсов игры

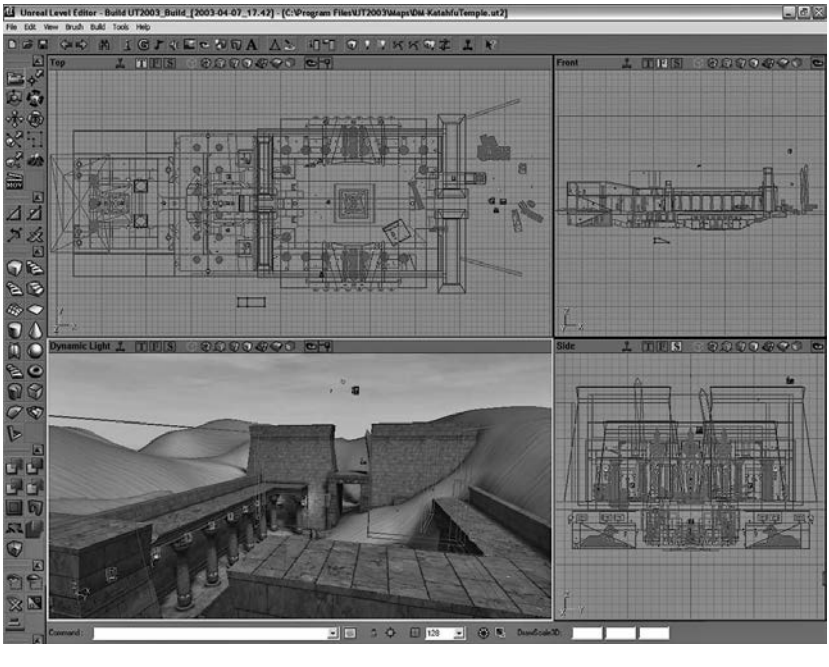


Рис. 15.9. UnrealEd также предоставляет редактор игрового мира

Расходы на обработку данных

В подразделе 7.2.1 мы узнали, что конвейер обработки ресурсов (asset conditioning pipeline, ACP) переводит ресурсы игры из их исходного формата в форматы, поддерживаемые игровым движком. Обычно это двухэтапный процесс. Сначала ресурс экспортируется из DCC-приложения в платформонезависимый (переносимый) формат, содержащий только те данные, которые относятся к игре. Затем ресурс обрабатывается и сохраняется в формате, оптимизированном для определенной платформы. В мультиплатформенных проектах переносимый ресурс на втором этапе превращается в несколько ресурсов, рассчитанных на разные платформы.

Одним из ключевых различий между конвейерами является то, в какой именно момент выполняется второй этап оптимизации для конкретной платформы. UnrealEd реализует его сразу, когда ресурсы импортируются в редактор. Это обеспечивает высокую скорость внесения изменений во время проектирования уровней. Однако это может сделать редактирование таких исходных ресурсов, как меши, анимации, аудио и т. д., более болезненным. Движки наподобие *Source* или *Quake* занимаются оптимизацией ресурсов при формировании уровня, прямо перед запуском игры. *Halo* дает возможность пользователю изменять исходные ресурсы в любое время. Их преобразование в оптимизированный формат происходит при начальной загрузке в движок, а результаты кэшируются, чтобы этот этап не выполнялся без нужды при каждом запуске игры.

16 Системы организации игрового процесса на этапе выполнения

16.1. Компоненты системы организации игрового процесса

Большинство игровых движков предоставляют пакет программных компонентов, играющий роль фреймворка, на основе которого можно создавать уникальные правила, цели и динамические элементы игры. У этих компонентов нет общепринятого названия, мы будем называть их совокупность *системой организации игрового процесса* движка. Если провести границу между игровым движком и самой игрой, эти системы окажутся прямо посередине. Теоретически они могут не иметь прямого отношения к конкретной игре. Однако в реальности почти всегда содержат детали, связанные с определенными жанром или игрой. Границу между движком и игрой лучше всего, наверное, представить себе в виде одного большого градиента, который соединяет их, охватывая все эти компоненты. В некоторых случаях доходит до того, что система организации игрового процесса находится полностью *над* границей между движком и игрой. Наиболее серьезные различия между игровыми движками относятся к проектированию и реализации их компонентов игрового процесса. Тем не менее у них на удивление много общего, и именно об этом пойдет речь далее.

У каждого игрового движка свой подход к проектированию программного обеспечения для игрового процесса. Однако у большинства из них есть основные подсистемы в том или ином виде.

- *Объектная модель игры на этапе выполнения*. Это реализация абстрактной объектной модели, которая предоставляется игровым дизайнерам в редакторе игрового мира.
- *Управление уровнями и их потоковая загрузка*. Эта система загружает и выгружает содержимое виртуальных миров, в которых происходит игровой процесс. Во многих движках данные уровней передаются в память поточным способом

в процессе игры, создавая тем самым иллюзию большого бесшовного мира, хотя на самом деле он разбит на отдельные области.

- *Обновление объектной модели в реальном времени*. Чтобы игровые объекты виртуального мира обладали независимым поведением, каждый из них должен периодически обновляться. Именно здесь всевозможные системы игрового движка объединяются в единое целое.
- *Обмен сообщениями и обработка событий*. Большинству игровых объектов нужно взаимодействовать друг с другом. Обычно это делается с помощью абстрактной системы обмена сообщениями. Межобъектные сообщения, сигнализирующие об изменениях в состоянии игрового мира, часто называют *событиями*. Поэтому система обмена сообщениями во многих студиях называется *системой событий*.
- *Скриптование*. Программирование высокоуровневой игровой логики на таких языках, как C или C++, может быть хлопотным. Чтобы улучшить продуктивность, сделать возможным быстрое внесение изменений и передать часть ответственности членам команды — не программистам, в игровые движки часто встраивают поддержку скриптовых языков. Эти языки могут быть текстовыми, как Python или Lua, и графическими, как Blueprints в Unreal.
- *Управление целями и потоком игры*. Эта подсистема управляет целями, стоящими перед игроком, и общим игровым потоком, что обычно описывается в виде последовательности, дерева или обобщенного графа игровых целей. Цели часто объединяются в главы, особенно в играх, в которых сюжет играет большую роль (что можно сказать о многих современных проектах). Система управления игровым потоком отвечает за общее развитие игры, следит за тем, какие цели достигнуты игроком, и пропускает игрока из одной области игрового мира в другую при достижении соответствующих целей. Некоторые дизайнеры называют это сплайном игры.

Самой сложной из этих систем является, наверное, объектная модель этапа выполнения. Она обычно предоставляет все или как минимум большую часть следующих возможностей.

- *Динамическое создание и уничтожение объектов*. В ходе игрового процесса динамические элементы виртуального мира часто появляются и исчезают. Аптечки испаряются, если их подобрать, взрывы возникают и пропадают, а вражеское подкрепление таинственным образом выходит из-за угла, когда вы уже думали, что прошли уровень. Многие игровые движки предоставляют систему для управления памятью и другими ресурсами, связанными с динамическим созданием игровых объектов. А иногда динамическое создание и удаление объектов вообще недоступно.
- *Привязка к низкоуровневым системам движка*. Каждый объект имеет какую-то связь с одной или несколькими внутренними системами движка. Большинство игровых объектов представлены на экране в виде треугольных мешей.

У некоторых из них есть эффекты частиц. Многие генерируют звук. Некоторые анимированы. Часто они стыкуются между собой, а некоторые динамически симулируются физическим движком. Одна из основных задач системы организации игрового процесса состоит в том, чтобы дать каждому игровому объекту доступ к возможностям систем движка, от которых он зависит.

- *Моделирование поведения объектов в реальном времени.* По своей сути игровой движок является динамической компьютерной симуляцией агентной модели. Проще говоря, игровому движку необходимо постоянно обновлять состояние игровых объектов динамическим образом. Обновления могут поводиться в определенном порядке, продиктованном, с одной стороны, зависимостями между объектами, с другой — их зависимостью от различных подсистем движка, и с третьей — взаимозависимостями между самими этими подсистемами.
- *Возможность создавать новые типы игровых объектов.* Требования любой игры меняются и эволюционируют по мере ее разработки. При этом важно иметь достаточно гибкую модель игровых объектов, чтобы их новые типы можно было с легкостью добавлять и делать доступными в редакторе игрового мира. В идеале у вас должна быть возможность создать новый тип объекта на основе одних лишь данных. Однако во многих движках для этого по-прежнему требуются услуги программиста.
- *Уникальные идентификаторы объектов.* Виртуальные миры, как правило, содержат сотни, если не тысячи отдельных игровых объектов различных типов. Вы должны иметь возможность идентифицировать или найти любой из них на этапе выполнения. Это означает, что все они должны иметь какого-то рода уникальные идентификаторы. Лучшим идентификатором было бы удобное для запоминания имя, но мы должны учитывать накладные расходы, которые повлечет за собой использование строк в процессе игры. Наиболее эффективным выбором являются числовые идентификаторы, но игровым дизайнерам очень сложно с ними работать. Вероятно, оптимальным решением будет применение хешированных строк (см. подраздел 6.4.3): они так же эффективны, как и целые числа, но при этом их можно преобразовать в исходные строки, которые легче воспринимать.
- *Запрашивание игровых объектов.* Система организации игрового процесса должна предоставлять средства поиска объектов внутри игрового мира. Возможно, нам нужно получить объект по его уникальному идентификатору или все объекты определенного типа. Иногда возникает необходимость в выполнении сложных запросов, основанных на произвольных критериях, например «найти всех врагов в радиусе 20 м от персонажа игрока».
- *Ссылки на игровые объекты.* Нам нужен какой-то механизм, с помощью которого мы могли бы *ссылаться* на найденные объекты — либо кратко, внутри одной функции, либо на протяжении более длинных отрезков времени. Ссылками на объекты могут быть и обычные указатели на экземпляры классов в C++, и что-то более сложное, например дескриптор или умный указатель с подсчетом ссылок.

- *Поддержка конечного автомата*. Многие типы игровых объектов лучше всего моделировать в виде конечного автомата. Некоторые игровые движки позволяют объекту находиться в одном из множества возможных состояний, каждое из которых имеет свои атрибуты и особенности поведения.
- *Сетевая репликация*. В сетевых многопользовательских играх компьютеры соединяются между собой по локальной сети или Интернету. Состояние отдельного игрового объекта обычно принадлежит одному компьютеру, который им управляет. Но это состояние также необходимо *реплицировать* (передать) другим участникам игры, чтобы все игроки имели согласованное представление объекта.
- *Сохранение и загрузка/хранение объектов*. Многие игровые движки позволяют сохранять объекты игрового мира на диск с возможностью последующей загрузки. Это может поддерживать систему сохранения игры в любой момент, реализовывать сетевую репликацию или быть основным средством загрузки игровых областей, созданных в редакторе игрового мира. Для хранения объектов обычно требуются определенные возможности языка, такие как *динамическая идентификация типов данных* (runtime type identification, RTTI), *рефлексия* и *абстрактные классы*. RTTI и рефлексия предоставляют программному обеспечению средства динамического определения *типа* объекта, а также информацию об *атрибутах* и *методах* его класса на этапе выполнения. Абстрактные классы позволяют создавать объекты без предварительного задания имени класса — это очень полезно при сериализации экземпляров объекта в память или на диск. Если у выбранного вами языка нет встроенной поддержки RTTI, рефлексии и абстрактных классов, их можно добавить вручную.

Оставшаяся часть главы посвящена подробному рассмотрению каждой из этих подсистем.

16.2. Архитектуры объектной модели времени выполнения

В редакторе игрового мира дизайнер имеет дело с абстрактной объектной моделью, определяющей различные типы динамических элементов, которые могут существовать в игре, их поведение и возможные атрибуты. На этапе выполнения система организации игрового процесса должна предоставить конкретную реализацию этой объектной модели. Это, безусловно, самая большая часть любой такой системы.

Реализация объектной модели на этапе выполнения может не иметь ничего общего с абстрактной инструментальной моделью. Например, она вообще может быть реализована не на объектно-ориентированном языке или в ней может использоваться набор взаимосвязанных экземпляров класса для представления одного абстрактного игрового объекта. Какой бы ни была архитектура объектной модели времени выполнения, она должна точно воспроизводить типы, атрибуты и поведение объектов, объявленные в редакторе игрового мира.

Объектная модель времени выполнения — это внутриигровое проявление абстрактной инструментальной модели, с которой дизайнеры работают в редакторе. Ее архитектура может существенно варьироваться, но большинство игровых движков выполнены в одном из двух основных архитектурных стилей.

- *Объектный стиль*. Каждый инструментальный игровой объект представлен на этапе выполнения одним или несколькими взаимосвязанными экземплярами класса. У каждого объекта есть набор *атрибутов* и *поведений*, инкапсулированных внутри класса (или классов), экземпляром которого является объект. Игровой мир — это не что иное, как коллекция игровых объектов.
- *Стиль на основе свойств*. Каждый инструментальный игровой объект представлен всего лишь уникальным идентификатором в виде целого числа, хеша или строки. *Свойства* всех игровых объектов распределены по множеству таблиц, каждая из которых хранит свойства определенного типа, и доступны по идентификатору объекта (вместо того чтобы храниться централизованно в одном или нескольких взаимосвязанных экземплярах класса). Сами свойства часто реализуются в виде экземпляров заранее определенных классов. *Поведение* игрового объекта косвенно определяется набором свойств, из которых он состоит. Например, если у объекта есть свойство «Здоровье», это означает, что он может быть поврежден, способен потерять здоровье и в итоге умереть. Наличие свойства MeshInstance будет говорить о том, что объект может быть отрисован в 3D в виде экземпляра треугольного меша.

У каждого из этих архитектурных стилей есть определенные плюсы и минусы. Мы подробно их исследуем и отметим те аспекты, в которых один стиль может иметь существенное превосходство перед другим.

16.2.1. Архитектуры объектного стиля

В архитектурах *объектного стиля* каждый логический игровой объект реализуется в виде экземпляра класса или, возможно, нескольких связанных между собой экземпляров. Под это общее определение подходит множество архитектур. В следующих пунктах мы исследуем несколько наиболее распространенных.

Простая модель объектного стиля в игре Hydro Thunder на C

Для реализации объектной модели игры не обязательно использовать объектно-ориентированный язык вроде C++. Например, аркадный хит *Hydro Thunder*, созданный Midway Home Entertainment в Сан-Диего, был полностью написан на C. В этой игре применена очень простая объектная модель, состоящая всего из нескольких типов объектов:

- лодок (управляемых игроком и ИИ);
- парящих значков ускорения синего и красного цвета;
- анимированных объектов внешней среды (животные вдоль гоночной трассы и т. д.);

- поверхности воды;
- трамплинов;
- водопадов;
- эффектов частиц;
- секторов гоночной трассы (двухмерные полигональные области, соединенные друг с другом и формирующие водный участок для проведения гонок);
- статической геометрии (ландшафт, зеленые насаждения, строения вдоль трассы и т. д.);
- двухмерных элементов приборной панели.

На рис. 16.1 показаны несколько снимков экрана *Hydro Thunder*. Обратите внимание на парящие значки ускорения в обоих примерах и на прыгающую на мосту обезьяну на левом изображении (это один из анимированных объектов окружающей среды).

Рис. 16.1. Снимки экрана аркадной игры Hydro Thunder

У *Hydro* была структура языка C под названием `World_t`, которая отвечала за хранение содержимого игрового мира (то есть отдельно взятой гоночной трассы) и управление им. Она хранила указатели на массивы игровых объектов разных типов. Статическая геометрия была выполнена в виде одного экземпляра меша. Поверхность воды, водопады и эффекты частиц представлены отдельными узкоспециализированными структурами данных. Лодки, значки ускорения и другие динамические объекты игры представлены экземплярами структуры общего назначения под названием `WorldOb_t` (то есть объект мира). В *Hydro* в соответствии с определением, которое было дано в настоящей главе, это был эквивалент *игрового объекта*.

Структура данных `WorldOb_t` содержала поля, определяющие позицию и ориентацию объекта в пространстве, трехмерный меш для его отрисовки, набор сфер столкновения, сведения о состоянии простой анимации (игра поддерживала только строгую иерархическую анимацию), физические свойства вроде скорости, массы и плавучести, а также другие атрибуты, общие для всех динамических объектов игры. Кроме того, у каждого экземпляра `WorldOb_t` было три указателя на пользовательские данные типа `void*`, пользовательскую функцию для обновления и пользовательскую функцию отрисовки. Игра *Hydro Thunder*, строго говоря, не была объектно-ориентированной, однако ее движок *Hydro* расширял процедурный язык (C) для поддержки элементарной реализации двух важных аспектов ООП — *наследования* и *полиморфизма*. Указатель на пользовательские данные позволял игровому объекту любого типа хранить информацию о собственном состоянии, характерную для своего типа, но при этом наследовать возможности, общие для всех объектов игры. Например, лодка *Banshee* имела иной механизм ускорения, чем *Rad Hazard*, и каждому из этих механизмов требовались особые данные о состоянии для управления анимацией включения и выключения ускорителя. Два других указателя играли роль *виртуальных функций*, позволяя объектам игрового мира демонстрировать полиморфное поведение (с помощью функции обновления) и иметь полиморфный внешний вид (благодаря функции отрисовки).

```
struct WorldOb_s
{
    Orient_t m_transform;     /* позиция/поворот */
    Mesh3d*  m_pMesh;         /* трехмерный меш */
    /* ... */
    void*    m_pUserData;     /* собственное состояние */

    void     (*m_pUpdate)();  /* полиморфное обновление */
    void     (*m_pDraw)();    /* полиморфная отрисовка */
};
typedef struct WorldOb_s WorldOb_t;
```

Монолитные иерархии классов

Стремление классифицировать типы игровых объектов по их месту в иерархии совершенно естественно. Обычно это подталкивает программистов к выбору объектно-ориентированных языков, которые поддерживают наследование. Иерархия классов — наиболее интуитивный и простой способ представления коллекции взаимосвязанных типов объектов. Поэтому неудивительно, что этот иерархический подход используется в большинстве коммерческих игровых движков.

На рис. 16.2 показана простая иерархия классов, на основе которой можно было бы реализовать игру *Pac-Man*. Как это часто бывает, она берет свое начало в классе `GameObject`, который может предоставлять какие-то механизмы, необходимые объектам любых типов, например RTTI или сериализацию. Класс `MovableObject` представляет любой объект с позицией и ориентацией в простран-

стве, `RenderableObject` делает объект доступным для отрисовки (в традиционной версии *Pac-Man* для этого задействовались бы спрайты, а в современных трехмерных разновидностях речь может идти о треугольных мешах). От `RenderableObject` происходят классы для привидений, персонажа Pac-Man, шариков и таблеток силы, из которых состоит игра. Это гипотетический пример, но он иллюстрирует основные идеи, лежащие в основе большинства иерархий классов игровых объектов: общие, универсальные возможности обычно размещаются у основания иерархии, а чем конкретнее становится функциональность, тем сильнее она от него отдаляется.

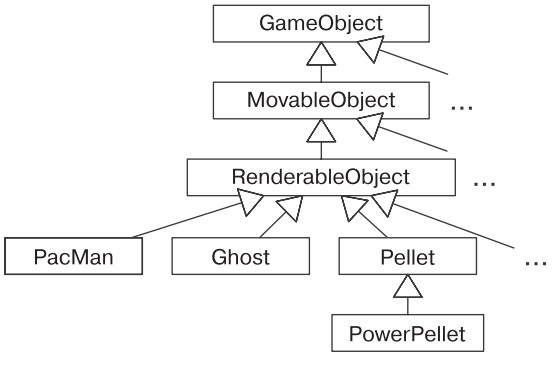


Рис. 16.2. Гипотетическая иерархия классов в игре Pac-Man

Вначале иерархия классов игровых объектов обычно выглядит компактной и простой, и в таком виде она может служить мощным и интуитивно понятным способом описания коллекции типов. Но с ростом она имеет тенденцию углубляться и в то же время расширяться, что делает ее, как я это называю, *монолитной*. Монолитные иерархии возникают в ситуации, когда практически все классы в объектной модели игры происходят от одного общего базового класса. Классическим примером является объектная модель Unreal Engine (рис. 16.3).

Проблемы глубоких и широких иерархий

Монолитные иерархии классов часто создают проблемы разработчикам игр, и этому есть много разных причин. Чем глубже и шире становится иерархия классов, тем ярче выражаются эти проблемы. Далее исследуем самые распространенные из них.

Понимание, поддержка и редактирование классов. Чем глубже класс находится в иерархии, тем сложнее его понять, поддерживать и редактировать. Это связано с тем, что для понимания класса нужно разобраться во всех его родительских классах. Например, изменение поведения безобидно выглядящей виртуальной функции в дочернем классе может нарушить допущения, сделанные в одном из множества его родителей, из-за чего возникнут неочевидные ошибки, которые будет сложно отыскать.

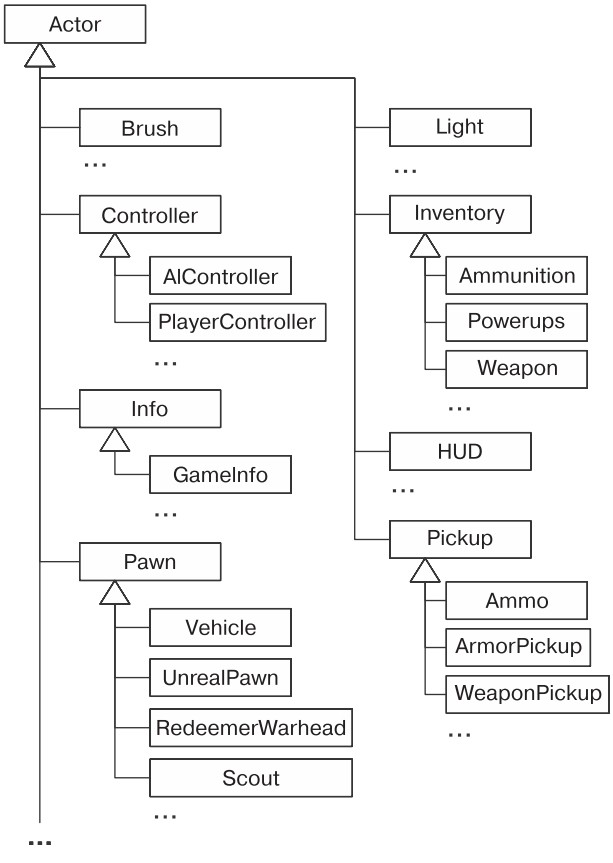


Рис. 16.3. Фрагмент иерархии классов игровых объектов в движке Unreal

Невозможность описания многоуровневых систем. Иерархия по своей природе классифицирует объекты в соответствии с определенной системой критериев, известной как систематика. Например, биологическая систематика классифицирует все живые существа по их генетическому сходству, используя дерево из восьми уровней: домен, царство, тип, класс, порядок, семейство, род и вид. На каждом уровне этого дерева применяется свой критерий для распределения бесчисленных форм жизни на нашей планете по более узким категориям.

Одной из важнейших проблем любой иерархии является то, что на каждом отдельном уровне она способна классифицировать объекты лишь вдоль одной оси — в соответствии с одним конкретным набором критериев. После выбора критериев классификация вдоль совершенно других осей становится сложной или вовсе невозможной. Например, биологическая систематика классифицирует объекты в соответствии с генетическими особенностями, оставляя без внимания цвет организмов. Чтобы классифицировать организмы по цвету, нужна совершенно другая древовидная структура.

В объектно-ориентированном программировании это ограничение иерархической классификации часто проявляется в виде широких, глубоких и запутанных иерархий классов. Если проанализировать дерево классов настоящей игры, можно обнаружить, что ее структура пытается объединить в себе целый ряд разных критериев классификации. В некоторых случаях разработчики идут на уступки для добавления нового типа объектов, чьи характеристики не были изначально заложены в иерархию. Представьте, к примеру, логичную на первый взгляд иерархию классов, описывающую разные типы транспортных средств (рис. 16.4).

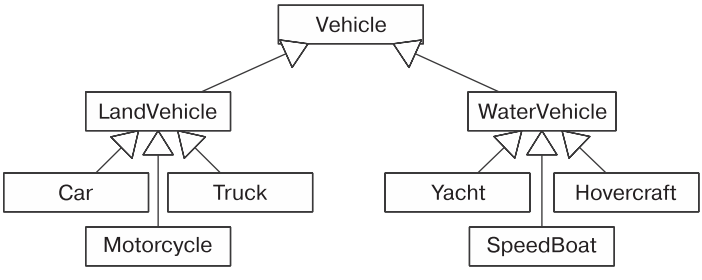


Рис. 16.4. Логичная на первый взгляд иерархия классов, описывающая разные виды транспортных средств

Что произойдет, когда дизайнеры игры сообщат программистам о том, что для нее теперь требуются *автомобили-амфибии*? Такое транспортное средство не вписывается в существующую систематику. В итоге программисты могут запаниковать или, что более вероятно, исковеркать иерархию классов, сделав ее уродливой и подверженной ошибкам.

Множественное наследование — смертельный бриллиант. Одно из решений проблемы с автомобилями-амфибиями состоит в использовании множественного наследования (МН) из C++ (рис. 16.5). На первый взгляд все выглядит хорошо. Однако множественное наследование в C++ создает ряд практических проблем. Например, с его помощью можно создать объект, который содержит несколько копий членов своего базового класса. Такую ситуацию называют смертельным бриллиантом или бриллиантом смерти (подробнее об этом — в подразделе 3.1.1).

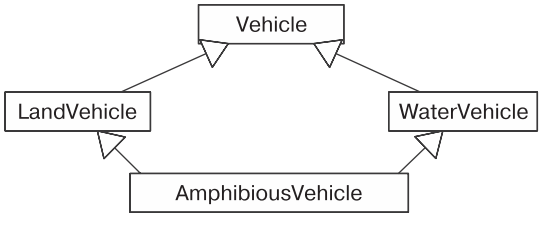


Рис. 16.5. Иерархия классов для автомобилей-амфибий в форме бриллианта

Трудности использования МН для построения рабочей, простой в обслуживании и понятной иерархии классов обычно перевешивают преимущества.

В результате большинство игровых студий запрещают или существенно ограничивают применение множественного наследования в своих иерархиях классов.

Классы-примеси. Некоторые студии допускают использование урезанной разновидности МН, в которой класс может иметь любое количество непосредственных родителей, но только одного предка второго уровня. Иными словами, класс может быть наследником только одного класса в основной иерархии, но при этом иметь любое количество предков-примесей (самостоятельных классов без предков). Это позволяет вынести общую функциональность в примеси, а затем подключать ее к любым участкам основной иерархии, на которых она требуется (рис. 16.6). Но как вы увидите в дальнейшем, вместо наследования классов обычно лучше задействовать композицию или агрегацию.

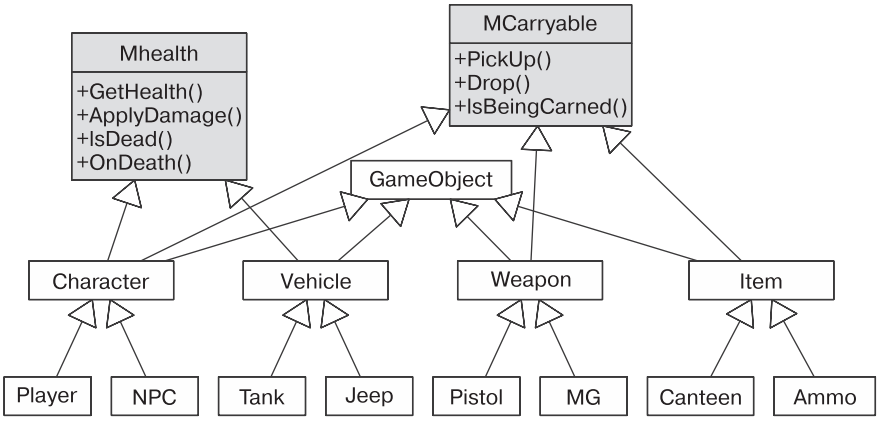


Рис. 16.6. Иерархия классов с примесями. Класс-примесь MHealth вводит понятие здоровья и возможность умереть в любом классе, который его наследует. Класс-примесь MCarryable делает наследующий его объект доступным для переноса персонажем (Character)

Эффект раздутия. При проектировании монолитной иерархии классов корневой класс или классы выглядят очень простыми и предоставляют только минимальный набор возможностей. Но с появлением в игре все новой функциональности желание сделать так, чтобы у нескольких классов был общий код, начинает раздувать иерархию.

Представьте, к примеру, что изначально плавать могут только деревянные ящики. Но, увидев, как круто эти ящики смотрятся в воде, дизайнеры игры начинают просить о создании других плавающих объектов, таких как персонажи, клочки бумаги, транспортные средства и т. д. Поскольку плавучесть не входила в число оригинальных критериев классификации на момент проектирования иерархии, программисты быстро столкнулись с необходимостью внедрения этого свойства в классы, не имеющие никакой иерархической связи. На множественное наследование в этой индустрии смотрят косо, поэтому программисты решают переместить код плавучести вверх по иерархии в базовый класс, общий для всех объектов, способных держаться на воде. То, что некоторые из объектов, наследованных от этого

базового класса, не могут плавать, видится менее серьезной проблемой по сравнению с дублированием кода в разных классах (чтобы четко очертить это различие, программисты могут даже добавить булево поле с именем вроде `m_bCanFloat`). В конце концов плавучесть становится возможностью корневого класса иерархии (вместе с практически любой другой функциональностью игры).

Классическим примером эффекта раздутия является класс `Actor` в Unreal. Он содержит данные и код для управления отрисовкой, анимацией, физикой, взаимодействием с окружающей средой, аудиоэффектами, сетевой репликацией для многопользовательских игр, созданием/удалением объектов, перебором акторов (возможностью перебирать все акторы, удовлетворяющие определенному критерию, и выполнять с ними какие-то действия) и трансляцией сообщений. То, что функции могут уплывать вверх и раздувать классы, близкие к основе монолитной иерархии, затрудняет инкапсуляцию возможностей различных подсистем движка.

Упрощение иерархии за счет композиции

Наверное, самой распространенной причиной возникновения монолитных иерархий классов является чрезмерное использование в объектно-ориентированном проектировании отношения вида IS-A. Например, программист, занимающийся пользовательским интерфейсом игры, может решить, что класс `Window` (окно) должен происходить от класса `Rectangle` (прямоугольник), так как все окна прямоугольные. Однако окно *не является* прямоугольником, оно лишь его *содержит*, определяя с его помощью свои границы. Поэтому более реалистичным решением этой конкретной проблемы проектирования будет включение экземпляра класса `Rectangle` в класс `Window` или предоставление последнему указателя либо ссылки на `Rectangle`.

В объектно-ориентированном проектировании отношение HAS-A называют *композицией*. Если использовать этот подход, класс А будет содержать либо непосредственный *экземпляр* класса Б, либо *указатель* или *ссылку* на него. Строго говоря, термин «композиция» подразумевает, что класс А должен *владеть* классом Б. Это означает, что вместе с экземпляром класса А автоматически создается экземпляр класса Б, а при уничтожении экземпляра А удаляется и экземпляр Б. Но мы можем также связывать классы друг с другом через указатель или ссылку, чтобы одному классу не приходилось управлять жизненным циклом другого. Такой подход обычно называют *агрегацией*.

Преобразование IS-A в HAS-A. Преобразование отношений IS-A в HAS-A может помочь уменьшить ширину, глубину и сложность иерархии классов в игре. Чтобы это проиллюстрировать, взглянем на гипотетическую монолитную иерархию (рис. 16.7). Корневой класс `GameObject` предоставляет некоторую базовую функциональность, необходимую всем игровым объектам, такую как RTTI, рефлексия, сохранение с помощью сериализации, сетевая репликация и т. д. Класс `MovableObject` представляет любой игровой объект, способный *изменяться*, то есть менять позицию, положение в пространстве и при желании масштаб. `RenderableObject` добавляет возможность быть отрисованным на экране (не все

объекты нужно отрисовывать, например, невидимый класс `TriggerRegion` можно было бы вывести прямо из `MovableObject`). Класс `CollidableObject` предоставляет своим экземплярам информацию о столкновениях. Класс `AnimatingObject` позволяет анимировать свои экземпляры с использованием скелетно-суставной иерархии. И наконец, `PhysicalObject` дает своим экземплярам возможность участвовать в физической симуляции, например, когда твердое тело в игровом мире попадает под воздействие гравитации и начинает отскакивать.

У этой иерархии классов есть одна большая проблема: она ограничивает диапазон типов игровых объектов, которые мы можем создавать. Если нам нужен объект с симуляцией физики, мы должны наследовать его класс от `PhysicalObject`, даже если ему не нужна скелетная анимация. Если требуется объект с поддержкой столкновений, он должен наследовать `CollidableObject`, хотя может быть невидимым и, следовательно, не нуждаться в услугах `RenderableObject`.

Второй проблемой иерархии, показанной на рис. 16.7, является то, что она затрудняет расширение функциональности существующих классов. Представьте, к примеру, что с целью поддержки морфинга (https://ru.qwertyu.wiki/wiki/Morph_target_animation) мы вывели из `AnimatingObject` два новых класса, `SkeletalObject` и `MorphTargetObject`. Если мы хотим, чтобы их можно было физически симулировать, нам придется превратить `PhysicalObject` в два почти идентичных класса, один из которых наследуется от `SkeletalObject`, а другой — от `MorphTargetObject`. Также можем обратиться к множественному наследованию.

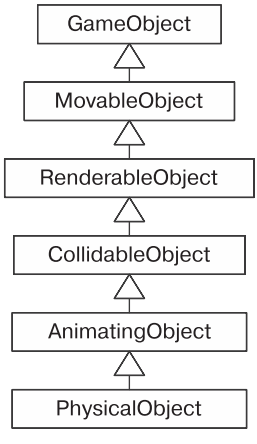


Рис. 16.7. Гипотетическая иерархия классов игровых объектов, которая устанавливает связи исключительно с помощью наследования

Чтобы решить эту проблему, различные возможности `GameObject` можно вынести в отдельные классы, каждый из которых будет иметь одну четко определенную обязанность. Такие классы иногда называют *компонентами* или *служебными объектами*. Компонентная архитектура позволяет выбирать для каждого типа игрового объекта только те возможности, которые нам нужны. Кроме того, она делает возможными поддержку, расширение и рефакторинг отдельных функций без изменения другого кода. К тому же отдельные компоненты проще понимать и тестировать, поскольку они не связаны между собой. Некоторые компонентные классы напрямую соответствуют конкретным подсистемам движка, таким как отрисовка, анимация, столкновения, физика, звук и т. д. Таким образом, когда эти подсистемы нужно интегрировать для совместного использования в определенном игровом объекте, они по-прежнему остаются четко определенными и инкапсулированными.

На рис. 16.8 показано, как могла бы выглядеть иерархия классов после превращения ее в компоненты. В этой переработанной архитектуре класс `GameObject` играет роль хаба, который содержит указатели на каждый из определенных нами

опциональных компонентов. Компонент `MeshInstance` служит заменой классу `RenderableObject` — он представляет экземпляр треугольного меша и инкапсулирует знания о том, как его отрисовывать. Точно так же компонент `AnimationController` заменяет `AnimatingObject`, предоставляя возможности скелетной анимации объекту `GameObject`. Класс `Transform` заменяет `MovableObject`, сохраняя позицию, положение в пространстве и масштаб объекта. Класс `RigidBody` описывает геометрию столкновений игрового объекта и предоставляет для своего экземпляра `GameObject` интерфейс к низкоуровневым системам, отвечающим за столкновения и физику, заменяя собой `CollidableObject` и `PhysicalObject`.

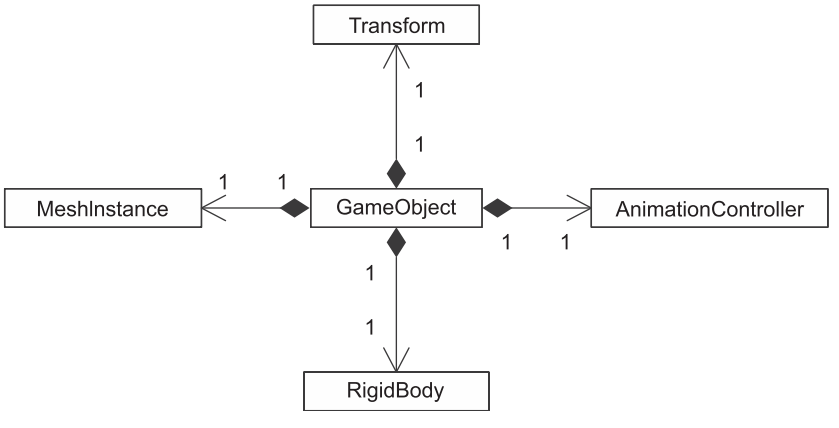


Рис. 16.8. Гипотетическая иерархия классов игровых объектов, переработанная с акцентом на композицию вместо наследования

Создание компонентов и владение ими. В такого рода архитектуре центральный класс (хаб) обычно владеет своими компонентами, то есть управляет их жизненным циклом. Но откуда `GameObject` знает, какие компоненты нужно создавать? У этой задачи есть множество решений, самое простое из которых заключается в предоставлении корневому классу `GameObject` указателей на все возможные компоненты. Каждый уникальный тип игровых объектов является потомком класса `GameObject`. В конструкторе `GameObject` все указатели на компоненты инициализируются с помощью значения `nullptr`. Конструктор каждого дочернего класса может создавать любые нужные ему компоненты. Для удобства конструктор по умолчанию `GameObject` умеет удалять все эти компоненты автоматически. В такой архитектуре иерархия классов, берущая начало в `GameObject`, служит основной систематикой для тех видов объектов, которые мы хотим видеть в своей игре, а компоненты классов играют роль дополнительных подключаемых возможностей.

Одна из возможных реализаций логики создания и удаления компонентов в подобного рода иерархии показана далее. Однако имейте в виду, что данный код является лишь примером — подробности реализации могут существенно

варьироваться, даже если речь идет о движках, которые, в сущности, используют один и тот же подход к иерархии классов.

```
class GameObject
{
protected:
    // Наша трансформация (позиция, поворот, масштаб).
    Transform            m_transform;

    // Стандартные компоненты:
    MeshInstance*        m_pMeshInst;
    AnimationController* m_pAnimController;
    RigidBody*           m_pRigidBody;

public:
    GameObject()
    {
        // По умолчанию не задаем никаких компонентов.
        // Дочерние классы переопределят.
        m_pMeshInst = nullptr;
        m_pAnimController = nullptr;
        m_pRigidBody = nullptr;
    }

    ~GameObject()
    {
        // Автоматически удаляем любые компоненты, созданные дочерними
        // классами (удаление нулевых указателей не является ошибкой).
        delete m_pMeshInst;
        delete m_pAnimController;
        delete m_pRigidBody;
    }

    // ...
};

class Vehicle : public GameObject
{
protected:
    // Добавляем еще несколько компонентов специально
    // для транспортных средств…
    Chassis* m_pChassis;
    Engine* m_pEngine;
    // ...

public:
    Vehicle()
    {
        // Создаем стандартные компоненты GameObject.
        m_pMeshInst = new MeshInstance;
        m_pRigidBody = new RigidBody;
```

```
        // ВНИМАНИЕ: мы исходим из того, что контроллеру анимации
        // необходимо предоставить ссылку на экземпляр меша, чтобы он
        // мог передать ему палитру матриц.
        m_pAnimController
         = new AnimationController(*m_pMeshInst);

        // Создаем компоненты, относящиеся к транспортным средствам.
        m_pChassis = new Chassis(*this,
                                 *m_pAnimController);
        m_pEngine = new Engine(*this);
    }

    ~Vehicle()
    {
        // Здесь нужно удалять только компоненты, относящиеся
        // к транспортным средствам, так как стандартные компоненты
        // удаляют автоматически в GameObject.
        delete m_pChassis;
        delete m_pEngine;
    }
};
```

Обобщенные компоненты

Еще одна, более гибкая, но вместе с тем более сложная в реализации, альтернатива состоит в предоставлении корневому классу игровых объектов обобщенного связного списка компонентов. Компоненты в такой архитектуре обычно происходят от одного общего базового класса — это позволяет перебирать связный список и выполнять полиморфные операции, такие как запрашивание типа каждого компонента или передача им по очереди событий для обработки. Благодаря такому подходу класс корневого объекта игры может ничего не знать о доступных типах компонентов, что во многих случаях позволяет создавать новые типы без его изменения. Кроме того, мы можем сделать так, чтобы конкретный игровой объект содержал произвольное количество экземпляров каждого типа компонентов (жестко закодированная архитектура допускает лишь заранее определенное количество, которое зависит от того, сколько указателей на каждый компонент существует внутри класса игровых объектов).

Подобная архитектура проиллюстрирована на рис. 16.9. Она сложнее в реализации, чем модель с жестко определенными компонентами, поскольку игровой объект должен быть написан в полностью обобщенном виде. Соответственно, классы компонентов не делают никаких предположений о том, какие еще компоненты могут существовать в контексте того или иного игрового объекта. Выбор между жестко определенным набором указателей и связным списком компонентов — не самый простой. Ни один из этих подходов не обладает очевидным превосходством — у каждого из них есть свои преимущества и недостатки, и разные студии выбирают разные архитектуры.

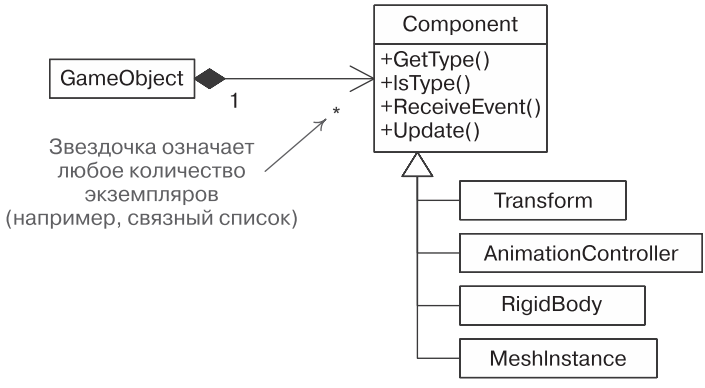


Рис. 16.9. Связный список компонентов может обеспечить гибкость, скрыв от центрального игрового объекта подробности о любом отдельно взятом компоненте

Чистые компонентные модели

Что получится, если довести идею компонентного подхода до абсолюта? Для этого пришлось бы вынести *всю* функциональность корневого класса `GameObject` в различные компонентные классы. В итоге класс `GameObject` буквально превратился бы в контейнер без какого-либо поведения, он по-прежнему содержал бы уникальный идентификатор и кучу ссылок на свои компоненты, но в остальном у него не было бы никакой логики. Может, от него вообще стоит избавиться? Для этого каждому компоненту можно назначить уникальный идентификатор игрового объекта. В итоге все они будут логически связаны между собой. Если также предусмотреть средство быстрого поиска компонентов по их идентификаторам, необходимость в центральном классе `GameObject` полностью отпадет. Я буду называть подобного рода архитектуру (рис. 16.10) *чистой компонентной моделью*.

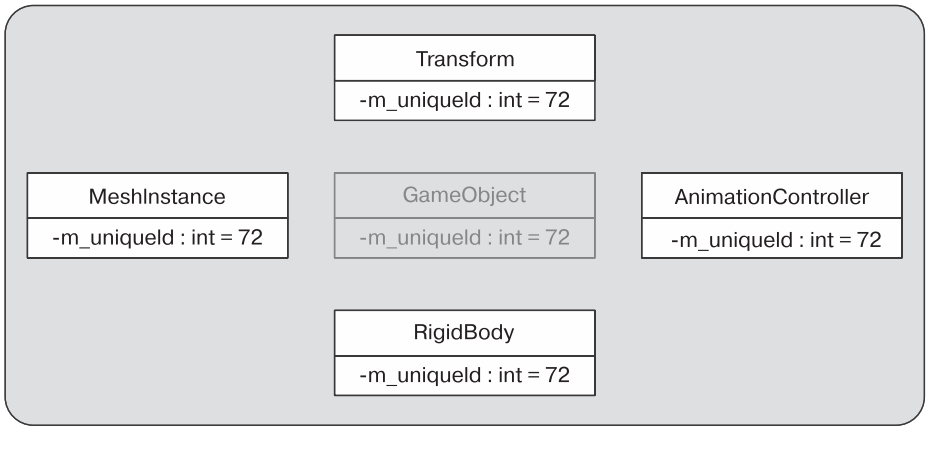


Рис. 16.10. В чистой компонентной модели логический игровой объект состоит из множества компонентов, связанных между собой косвенно, с помощью общего уникального идентификатора

Чистая компонентная модель не такая простая, как может показаться на первый взгляд, и у нее есть собственные проблемы. Например, нам все еще нужно как-то определять конкретные типы игровых объектов для игры и делать так, чтобы вместе с экземпляром типа создавались подходящие компоненты. Раньше за создание компонентов отвечала иерархия `GameObject`. Теперь же мы можем воспользоваться шаблоном проектирования «фабрика», который определяет для каждого типа игровых объектов по одному фабричному классу с виртуальным конструктором. Последний переопределяется для создания подходящих компонентов для каждого отдельного типа. Но мы также можем обратиться к модели на основе данных, когда типы игровых объектов определяются в текстовом файле, с которым при создании экземпляра типа сверяется движок.

Еще одной проблемой сугубо компонентной архитектуры является межкомпонентное взаимодействие. Центральный класс `GameObject`, игравший роль хаба, организовывал обмен сообщениями между разными компонентами. В чистой компонентной модели должен быть какой-то способ, с помощью которого один игровой объект сможет общаться с другим. Для этого компоненты могут искать друг друга по уникальному идентификатору игрового объекта. Но нам, наверное, захочется создать куда более эффективный механизм, например заранее объединить компоненты в кольцевой связный список.

В том же смысле отправка сообщений одним игровым объектом другому может оказаться непростой, если применять чисто компонентную модель. Мы больше не можем обращаться к экземпляру `GameObject`, поэтому нужно либо точно знать, с каким компонентом мы хотим общаться, либо транслировать сообщения всем компонентам, из которых состоит игровой объект. Ни один из этих вариантов нельзя назвать идеальным.

Чистая компонентная модель может использоваться и используется в реальных игровых проектах. У подобных архитектур есть свои преимущества и недостатки, но опять же они не имеют однозначного превосходства над альтернативными подходами. Если вы не занимаетесь теоретическими исследованиями в отделе R&D, вашим выбором должна быть та архитектура, работая с которой вы чувствуете себя наиболее комфортно и уверенно и которая лучше всего отвечает потребностям создаваемой игры.

16.2.2. Архитектуры на основе свойств

Программистам, которые часто работают с объектно-ориентированными языками программирования, свойственно воспринимать окружающий мир в виде объектов, обладающих атрибутами (полями с данными) и поведением (методами, функциями класса). *Модель на основе объектов* может выглядеть так.

- Объект 1:
  - позиция = (0, 3, 15);
  - поворот = (0, 43, 0).

- Объект 2:
  - позиция = (–12, 0, 8);
  - здоровье = 15.
- Объект 3:
  - поворот = (0, –87, 10).

Однако вместо объектов можно сосредоточиться на атрибутах. Сначала определяется набор всех свойств, которые может иметь игровой объект. Затем для каждого свойства создается таблица со значениями, которые это свойство имеет в каждом содержащем его игровом объекте. В качестве ключей для значений свойств выступают уникальные идентификаторы объектов. Вот как будет выглядеть *модель на основе свойств*.

- Позиция:
  - объект 1 = (0, 3, 15);
  - объект 2 = (–12, 0, 8).
- Поворот:
  - объект 1 = (0, 43, 0);
  - объект 3 = (0, –87, 10).
- Здоровье:
  - объект 2 = 15.

Объектные модели на основе свойств успешно применяются во многих коммерческих играх, включая *Deus Ex 2* и цикл *Thief*. Подробнее о том, как именно организованы объектные системы в этих проектах, мы поговорим в дальнейшем.

Архитектура на основе свойств больше напоминает базу данных, чем объектную модель. Каждый атрибут ведет себя словно таблица в реляционной БД, где уникальный идентификатор объекта играет роль *первичного ключа*. Конечно, в объектно-ориентированном подходе объекты обладают не только *атрибутами*, но и *поведением*. Но где же реализовать поведение, если есть только таблицы свойств? Ответ на этот вопрос зависит от конкретного движка, но чаще всего поведение реализуется одним из следующих способов:

- в самих свойствах;
- с помощью скриптового кода.

Давайте исследуем эти приемы глубже.

Реализация поведения в классах свойств

Каждый тип свойств можно реализовать в виде *класса свойств*. Свойство может быть как простым (одно булево значение или число с плавающей запятой), так и сложным (треугольный меш с отрисовкой на экране или «мозг» ИИ). Каждый класс свойств может обладать поведением в виде жестко определенных методов

(функций класса). Общее поведение конкретного игрового объекта определяется совокупностью поведения всех его свойств.

Например, если игровой объект содержит экземпляр свойства `Health`, его можно повредить и в конце концов уничтожить или убить. Когда игровой объект кто-то атакует, свойство `Health` может реагировать на это, соответственно уменьшая его здоровье. Свойства одного игрового объекта могут взаимодействовать между собой, демонстрируя совместное поведение. Например, при обнаружении атаки свойство `Health` может послать сообщение свойству `AnimatedSkeleton`, чтобы игровой объект воспроизвел подходящую анимацию в ответ на повреждение. Или, если свойство `Health` обнаруживает, что игровой объект будет разрушен или убит, оно может попросить `RigidBodyDynamics` активировать физическую симуляцию взрыва или поведения мертвого тела типа «тряпичная кукла».

Реализация поведения с помощью скриптов

Еще один вариант — хранение значений свойств в виде обычных данных в одной или нескольких таблицах, играющих роль БД. При этом поведение игровых объектов реализуется в виде скриптового кода. У каждого игрового объекта может быть специальное свойство с названием вроде `ScriptId`, которое содержит блок со скриптовым кодом (функцию или объект, если сам скриптовый язык объектно-ориентированный), отвечающим за поведение объекта. Игровой объект может использовать этот код также для обработки событий, возникающих в игровом мире. Подробнее о системах событий и языках для скриптования игр мы поговорим в разделах 16.8 и 16.9 соответственно.

В некоторых движках, основанных на свойствах, основной набор встроенных классов свойств предоставляется инженерами, но у дизайнеров и программистов есть возможность реализовывать совершенно новые типы свойств в скриптах. Этот подход успешно применялся в проекте *Dungeon Siege*.

Сравнение свойств и компонентов

Важно отметить, что многие авторы, упомянутые далее в пункте «Дополнительный материал», понимают под компонентом то, что я здесь называю объектом свойства. В подразделе 16.2.1 термин «компонент» использовался для обозначения подобъекта в объектно-ориентированной модели, что не совсем то же самое, что объект свойства.

Тем не менее объекты свойств во многом очень тесно связаны с компонентами. В обоих подходах один логический игровой объект состоит из нескольких подобъектов. Основное различие заключается в том, для чего эти подобъекты предназначены. В архитектуре на основе свойств каждый подобъект определяет отдельный атрибут самого игрового объекта (например, здоровье, визуальное представление, инвентарь, магические способности и т. д.), тогда как в компонентной (объектно-ориентированной) модели подобъекты часто представляют собой интерфейсы к тем или иным низкоуровневым подсистемам движка (отрисовка, анимация, столкновения и динамика и т. д.). Это различие настолько тонкое, что

во многих случаях им можно пренебречь. Вы можете называть свою архитектуру *чистой компонентной моделью* или *моделью на основе свойств*, но результат их применения будет практически одинаковым: логический игровой объект, состоящий из коллекции подобъектов, которые формируют его поведение.

Преимущества и недостатки архитектур на основе свойств

У подхода, основанного на атрибутах, есть целый ряд потенциальных преимуществ. Он обычно отличается более эффективной работой с памятью, так как нам нужно хранить значения только применяемых свойств, то есть никогда не будет игровых объектов с неиспользуемыми атрибутами. К тому же такую модель легче организовать вокруг данных: дизайнеры могут создавать новые атрибуты без перекомпиляции игры, потому что определение классов игровых объектов не меняется. Программистов приходится привлекать только в случае, когда нужно добавить совершенно новый тип свойств (если предположить, что этого нельзя сделать с помощью скриптов).

Кроме того, архитектура на основе свойств может лучше работать с кэшем, чем объектно-ориентированная модель, поскольку данные одного типа хранятся в памяти *по соседству*. Это обычный метод оптимизации для современного игрового оборудования, в котором доступ к памяти является куда более затратным, чем выполнение инструкций и вычислений (например, в PlayStation 3 один кэш-промах эквивалентен выполнению буквально тысяч инструкций процессора). Последовательное размещение данных в оперативной памяти может снизить частоту кэш-промахов или свести их к нулю, поскольку при доступе к одному элементу массива в ту же линию кэша автоматически загружаются элементы, находящиеся по соседству. Такой подход к организации данных иногда называют *структурой массивов* — это альтернатива более традиционному методу с *массивом структур*. Различия между этими двумя принципами компоновки памяти проиллюстрированы далее с помощью фрагмента кода (следует отметить, что мы бы не стали реализовывать объектную модель игры именно так, этот пример — лишь демонстрация того, как в архитектуре на основе свойств вместо одного массива со сложными объектами обычно возникает множество смежных массивов с данными похожих типов):

```
static const U32 MAX_GAME_OBJECTS = 1024;

// Традиционный подход в виде массива структур.

struct GameObject
{
    U32 m_uniqueId;
    Vector m_pos;
    Quaternion m_rot;
    float m_health;

    // ...
};

GameObject g_aAllGameObjects[MAX_GAME_OBJECTS];
```

```
// Более эффективный с точки зрения кэширования подход в виде структуры массивов.

struct AllGameObjects
{
    U32 m_aUniqueId[MAX_GAME_OBJECTS];
    Vector m_aPos[MAX_GAME_OBJECTS];
    Quaternion m_aRot[MAX_GAME_OBJECTS];
    float m_aHealth[MAX_GAME_OBJECTS];

// ...
    };

AllGameObjects g_allGameObjects;
```

У моделей, основанных на атрибутах, есть свои проблемы. Например, когда игровой объект представляет собой всего лишь груду свойств, отслеживать отношения между ними не так-то просто. Реализация специфического широкомасштабного поведения за счет объединения узких возможностей, принадлежащих группе объектов свойств, может оказаться затруднительной. А еще подобные системы сложно отлаживать, так как программист не может просто засунуть игровой объект в окно просмотра отладчика и проанализировать все его свойства сразу.

Дополнительный материал

На тему архитектур, основанных на свойствах, существует ряд интересных презентаций для PowerPoint, которые были представлены выдающимися инженерами в сфере разработки игр на различных конференциях.

- *Фермьер Р.* Creating a Data Driven Engine // Game Developer's Conference, 2002.
- *Билас С.* A Data-Driven Game Object System // Game Developer's Conference, 2002.
- *Дюран А.* Building Object Systems: Features, Tradeoffs, and Pitfalls // Game Developer's Conference, 2003.
- *Чейтлэйн Д.* Enabling Data Driven Tuning via Existing Tools // Game Developer's Conference, 2003.
- *Черч Д.* Object Systems. Представлена на конференции игровых разработчиков в Сеуле (Республика Корея), 2003. Конференция организована Крисом Хекером, Кэйси Муратори, Джоном Блоу и Дагом Черчем, chrishecker.com/images/6/6f/ObjSys.ppt.

16.3. Форматы областей игрового мира

Как мы уже видели, область игрового мира обычно содержит *статические* и *динамические* элементы. Статическая геометрия может быть представлена одним большим треугольным мешем или состоять из нескольких мешей поменьше. Каждый меш может иметь несколько *экземпляров* — например, один меш двери может

использоваться во всех дверных проемах в игровой области. Статические данные, как правило, содержат информацию о столкновениях, хранящуюся в виде полигонального супа — набора выпуклых форм и/или более простых геометрических примитивов вроде поверхности, параллелепипеда, капсулы или сферы. Другие статические элементы могут включать в себя *регионы*, позволяющие обнаруживать события или очерчивать участки игрового мира, *навигационный меш* для ИИ, набор отрезков, составляющих *края* фоновой геометрии, за которые может ухватиться персонаж игрока, и т. д. Мы не станем углубляться в детали этих форматов, поскольку большинство из них обсуждалось в предыдущих разделах.

Динамическая часть области игрового мира содержит некое представление игровых объектов внутри этой области. Игровой объект представляет собой совокупность *атрибутов* и *поведения*; последнее определяется либо напрямую, либо опосредованно, по *типу* объекта. В объектно-ориентированной архитектуре этот тип определяет, экземпляр какого именно класса (или классов) представляет объект на этапе выполнения. В модели на основе свойств поведение игрового объекта определяется совокупностью поведения его свойств, но от типа по-прежнему зависит, какие свойства этот объект должен иметь (можно сказать, что свойства объекта составляют его тип). Поэтому для каждого игрового объекта в области игрового мира обычно предусмотрены:

- *исходные значения атрибутов объекта*. Область игрового мира определяет состояние каждого объекта в момент его появления в игре. Атрибуты объекта могут храниться во множестве разных форматов. Далее рассмотрены несколько популярных вариантов;
- *некая спецификация типа объекта*. В объектно-ориентированном движке это может быть строка, хеш или какая-то другая разновидность уникального идентификатора. В архитектуре, основанной на свойствах, тип может храниться как есть или косвенно определяться коллекцией своих свойств/атрибутов.

16.3.1. Двоичные образы объектов

Каждый игровой объект можно сохранять на диске отдельно в виде двоичного образа, который является точным представлением того, как он выглядит в памяти во время выполнения. Это делает процесс загрузки игровых объектов тривиальным. Загрузив в память область игрового мира, мы получаем готовые образы всех наших объектов, поэтому их можно оставить неизменными.

Ну, не совсем неизменными. Сохранение двоичных образов «живых» экземпляров классов в C++ проблематично по ряду причин. Например, *указатели* и *виртуальные таблицы* требуют особого обращения, а для данных внутри каждого экземпляра класса, возможно, придется *менять порядок следования байтов* (эти методики подробно описаны в подразделе 7.2.2). Более того, двоичные образы объектов негибки и неустойчивы к внесению изменений. Игровой процесс — один из наиболее динамичных и нестабильных аспектов любого игрового проекта, поэтому важно выбрать такой формат хранения данных, который способствует быстрой

разработке и способен выдерживать частые обновления. Так что двоичные образы обычно не самый лучший вариант для хранения данных игровых объектов, хотя этот формат может подойти для более стабильных структур данных, таких как меш или геометрия столкновений.

16.3.2. Описание сериализованных игровых объектов

Сериализация — еще один метод сохранения внутреннего состояния объекта в файл на диске. Этот подход обычно переносимый и более простой в реализации по сравнению с двоичными образами. Чтобы сериализовать и сохранить объект на диск, у него запрашивают поток данных, содержащий достаточно информации для того, чтобы позже по нему можно было восстановить оригинал. Когда объект десериализуется с диска обратно в память, создается экземпляр подходящего класса, а затем считывается поток атрибутов, инициализирующий внутреннее состояние объекта. Если исходный поток сериализованных данных был полным, новый объект должен быть идентичен оригинальному во всех отношениях.

Поддержка сериализации встроена в некоторые языки программирования. Например, C# и Java предоставляют стандартизированные механизмы для сериализации экземпляров объектов в текстовый формат XML и обратно. У языка C++, к сожалению, таких встроенных механизмов нет, однако для него было создано множество систем сериализации как внутри, так и за пределами игровой индустрии. Мы не станем углубляться в подробности написания системы сериализации на C++. Вместо этого я опишу формат данных и несколько основных механизмов, которые требуются для использования сериализации в этом языке.

Сериализация данных — это не то же самое, что двоичный образ объекта. Обычно для нее применяется более удобный и переносимый формат. XML — популярный формат для сериализации объектов, поскольку стандартизован и имеет хорошую поддержку. Он в какой-то мере понятен человеку и отлично подходит для описания иерархических структур данных, которые часто возникают при сериализации коллекций взаимосвязанных игровых объектов. К сожалению, скорость разбора XML очень низкая, что может затянуть загрузку областей игрового мира. Поэтому некоторые игровые движки задействуют проприетарные двоичные форматы, которые, по сравнению с XML-текстом, более компактны и быстры в разборе.

Многие игровые движки (и неигровые системы сериализации объектов) переориентировались на альтернативу XML — текстовый формат данных JSON (http://www.json.org). Помимо прочего, JSON повсеместно используется для обмена информацией во Всемирной паутине. Например, взаимодействие с Facebook API происходит исключительно с помощью JSON.

Сериализация объекта на диск и обратно обычно реализуется одним из двух основных способов.

- Можно добавить к базовому классу пару функций с названиями вроде `SerializeOut()` и `SerializeIn()` и сделать так, чтобы каждый дочерний класс предоставлял для них собственную реализацию, которая знает, как сериализовать атрибуты этого конкретного класса.

- Можно реализовать для классов на C++ систему *рефлексии*, затем написать обобщенный механизм, способный автоматически сериализовать любой объект в C++, для которого доступна информация о рефлексии.

Термин «*рефлексия*» используется в разных языках, включая C#. В сущности, данные рефлексии — это описание содержимого класса на этапе выполнения. Они содержат сведения об имени класса, его свойствах, типе каждого свойства и его сдвиге в образе памяти объекта, а также информацию обо всех методах класса. Имея данные рефлексии для произвольных классов в C++, мы могли бы довольно легко написать систему сериализации объектов общего назначения.

Нетривиальной частью системы рефлексии в C++ является генерация соответствующих сведений для всех нужных нам классов. Это можно сделать инкапсуляцией всех свойств класса в макросе `#define`, который извлекает необходимую информацию о рефлексии, предоставляя виртуальную функцию, переопределяемую каждым дочерним классом. Эта функция возвращает для своего класса соответствующие данные рефлексии, закодированные вручную в виде структуры или с помощью какого-то другого изобретательного подхода.

Помимо сведений об атрибутах, поток данных сериализации обязательно включает в себя имя или уникальный идентификатор *класса* или *типа* каждого объекта, с помощью которого создается экземпляр соответствующего класса, когда объект сериализуется с диска обратно в память. Этот идентификатор может храниться в виде строки, хеша или какого-то другого вида уникального ID.

К сожалению, C++ не позволяет создать экземпляр класса по одной лишь строке с его именем или идентификатору. Имя класса должно быть известно на момент компиляции, поэтому программисту следует указать его вручную (например, `new ConcreteClass`). Чтобы обойти это ограничение языка, любая система сериализации объектов в C++ содержит какого-то рода *фабрику классов*. Фабрика может быть реализована множеством разных способов, самым простым из которых является использование таблицы данных, которая привязывает имя/идентификатор каждого класса к какой-то функции или объекту-функтору, написанному вручную специально для создания экземпляра этого конкретного класса. Зная имя или идентификатор класса, мы можем просто найти в этой таблице подходящую функцию или функтор и создать с их помощью нужный экземпляр.

16.3.3. Спаунеры и схемы типов

И у двоичных образов объектов, и у форматов сериализации есть серьезный недостаток. И те и другие определяются на этапе выполнения в рамках реализации типов игровых объектов, которые они хранят, следовательно, редактор игрового мира должен знать подробности реализации среды выполнения игрового движка. Например, чтобы сохранить двоичный образ гетерогенной коллекции игровых объектов, редактор игрового мира должен либо быть напрямую скомпонованным с кодом среды выполнения игрового движка, либо содержать код, специально написанный для генерации блоков байтов, полностью совпадающих со структурой игровых объектов на этапе выполнения. Данные сериализации не так жестко

привязаны к реализации игровых объектов, но опять же редактору игрового мира нужен доступ либо к методам класса `SerializeIn()` и `SerializeOut()` (что требует непосредственной компоновки с кодом игрового объекта в среде выполнения), либо к информации о рефлексии этого класса.

Чтобы предотвратить связывание между редактором игрового мира и кодом среды выполнения движка, описание игровых объектов можно абстрагировать таким способом, который не зависит от реализации. Для каждого объекта в файле с областью игрового мира будет предусмотрен небольшой блок данных, который часто называют *спаунером*. Спаунер — это легковесное сугубо информационное представление игрового объекта, которое можно использовать на этапе выполнения для создания его экземпляров и инициализации. Оно содержит идентификатор инструментального *типа* игрового объекта и таблицу простых пар типа «ключ — значение», которые описывают его исходные атрибуты. В число этих атрибутов часто входит метод преобразования модели в экземпляры игрового мира, так как у большинства игровых объектов есть определенные позиция, положение и масштаб в виртуальном пространстве. При появлении игрового объекта создается экземпляр подходящего класса (или классов) в соответствии с типом спаунера. Затем эти объекты времени выполнения обращаются к словарю с парами «ключ — значение», чтобы инициализировать свои свойства.

Спаунер можно сконфигурировать так, чтобы он создавал игровые объекты сразу после загрузки, или он может оставаться неактивным, пока в какой-то момент во время игры его не попросят создать тот или иной экземпляр. Спаунеры можно реализовать в виде полноценных объектов с удобным и функциональным интерфейсом, которые, помимо атрибутов, могут хранить полезные метаданные. С их помощью не только загружают игровые объекты. Например, в движке Naughty Dog дизайнеры использовали спаунеры для определения важных точек или осей координат в игровом мире. Они назывались *позиционными спаунерами* или *спаунерами-локаторами*. Локаторы имеют множество применений в игре, например:

- определение точек интереса для персонажей ИИ;
- определение осей координат, относительно которых можно полностью синхронно воспроизвести набор анимаций;
- определение позиции, в которой должен возникнуть эффект частиц или аудиоэффект;
- определение точек маршрута вдоль гоночной трассы.

Этот список можно продолжать.

Схемы типов объектов

Атрибуты и поведение игрового объекта определяются его типом. В редакторе игрового мира с использованием механизма спаунеров тип игрового объекта может быть представлен в виде информационной *схемы*, которая определяет коллекцию атрибутов, доступных пользователю во время создания или редактирования объекта этого типа. На этапе выполнения инструментальный тип можно привязать

к классу или коллекции классов, экземпляры которых требуются для загрузки игрового объекта заданного типа. Это можно реализовать как непосредственно в коде, так и с помощью таблиц данных.

Схемы типов можно хранить в обычном текстовом файле, который может быть прочитан редактором игрового мира и проанализирован/модифицирован его пользователями. Например, файл со схемой может выглядеть так:

```
enum LightType
{
    Ambient, Directional, Point, Spot
}
type Light
{
    String       UniqueId;
    LightType    Type;
    Vector       Pos;
    Quaternion   Rot;
    Float        Intensity : min(0.0), max(1.0);
    ColorARGB    DiffuseColor;
    ColorARGB    SpecularColor;
    // ...
}

type Vehicle
{
    String          UniqueId;
    Vector          Pos;
    Quaternion      Rot;
    MeshReference   Mesh;
    Int             NumWheels : min(2), max(4);
    Float           TurnRadius;
    Float           TopSpeed : min(0.0);
    // ...
}

//...
```

Этот пример раскрывает несколько важных моментов. Вы могли заметить, что вместе с именами атрибутов определяются их *типы данных*. Они могут быть простыми, такими как строки, целые числа, значения с плавающей запятой, или специализированными — наподобие векторов, кватернионов, ARGB-цветов или ссылок на такие особые типы ресурсов, как меши, данные о столкновениях и т. д. В примере мы даже предусмотрели механизм для определения перечисляемых типов, таких как `LightType`. Еще одним неочевидным моментом является то, что схема типов объектов предоставляет редактору игрового мира дополнительную информацию, например, какой тип элементов пользовательского интерфейса следует использовать при редактировании этого атрибута. Иногда выбор элемента пользовательского интерфейса для атрибута диктуется его типом — строки обычно редактируются с помощью поля ввода, для булевых значений предусмотрены флажки, а векторы представлены тремя полями для координат *x*, *y* и *z* или,

возможно, специализированным графическим компонентом, предназначенным для работы с векторами в 3D. Схема также может содержать метаинформацию для пользовательского интерфейса, например минимально и максимально допустимые значения для целочисленных и дробных атрибутов, наборы возможных вариантов для раскрывающихся списков и т. д.

Некоторые игровые движки поддерживают наследование схем типов по аналогии с классами. Например, каждый игровой объект должен знать свой *тип* и иметь *уникальный идентификатор*, чтобы на этапе выполнения его можно было отличить от остальных игровых объектов. Эти атрибуты можно указать в схеме верхнего уровня, от которой происходят все остальные схемы.

Значения атрибутов по умолчанию

Как вы можете себе представить, количество атрибутов в типичной схеме игровых объектов может быть довольно большим. Это означает, что игровым дизайнерам необходимо указывать много информации для каждого экземпляра игрового объекта, который он размещает в игровом мире, независимо от его типа. Определение *значений по умолчанию* для многих атрибутов схемы может оказаться чрезвычайно полезным. Это даст возможность дизайнерам игры создавать стандартные экземпляры игровых объектов, прилагая минимальные усилия, и при этом позволит им редактировать значения атрибутов в случае необходимости.

Значениям по умолчанию присуща одна проблема, которая проявляется при их изменении. Представьте, что игровые дизайнеры изначально хотели, чтобы орки наносили 20 единиц урона. После многих месяцев разработки они могут решить, что оркам не хватает силы, и сделать так, чтобы по умолчанию те наносили урон 30 единиц. Теперь любой новый орк, появляющийся в игровом мире, будет при ударе забирать 30 единиц здоровья, если не изменить это поведение вручную. А как же те орки, которые были созданы в области игрового мира до изменения? Неужели нам придется их отыскивать и менять урон, наносимый каждым из них, на 30?

В идеале система спаунеров должна автоматически распространять изменения значений по умолчанию на все существующие экземпляры, атрибуты которых не были явно переопределены. Для реализации этой возможности можно просто игнорировать пары «ключ — значение» для атрибутов, которые совпадают со значением по умолчанию. Если у спаунера нет какого-то атрибута, вместо него можно подставить значение по умолчанию (здесь предполагается, что у игрового движка есть доступ к файлу со схемой типов, чтобы он мог прочитать стандартные значения атрибутов; если делать это инструментальным способом, для распространения новых значений по умолчанию нужно будет просто пересобрать все игровые области, затронутые этим изменением). В нашем примере у большинства спаунеров для существующих орков не было бы пары «ключ — значение» `HitPoints`, если только, конечно, урон одного из спаунеров не был изменен вручную. Таким образом, когда значение по умолчанию меняется с 20 на 30, эти орки автоматически начинают его использовать.

Некоторые движки позволяют переопределять значения по умолчанию в дочерних типах. Например, схема типа под названием `Vehicle` может содержать атрибут

TopSpeed со стандартным значением 80 миль в час. Унаследованная от него схема типа Motorcycle может переопределить значение по умолчанию для TopSpeed, увеличив его до 100 миль в час.

Некоторые преимущества спаунеров и схем типов

Ключевыми преимуществами разделения спаунера и реализации игрового объекта являются *простота*, *гибкость* и *надежность*. С точки зрения управления данными работать с парами «ключ — значение» намного проще, чем с двоичными образами объектов, которые требуют исправления указателей или пользовательских форматов сериализованных объектов. Также применение пар «ключ — значение» делает формат данных чрезвычайно гибким и устойчивым к изменениям. Если игровой объект встречает пару, о которой ничего не знает, он может ее просто проигнорировать. Но если ему не удается найти нужную пару «ключ — значение», он может воспользоваться значением по умолчанию. Благодаря этому такой формат данных устойчив к изменениям, которые вносят как дизайнеры, так и программисты.

Кроме того, спаунеры упрощают архитектуру и реализацию редактора игрового мира, так как ему достаточно лишь знать, как работать с наборами пар «ключ — значение» и схемами типов объектов. Ему не нужно делить никакой код со средой выполнения игрового движка, и его связь с реализацией движка остается очень слабой.

Спаунеры и архетипы дают игровым дизайнерам и разработчикам много возможностей и обеспечивают большую гибкость. Дизайнеры могут создавать схемы типов новых игровых объектов внутри редактора игрового мира самостоятельно или с минимальной помощью со стороны программистов. Программист может реализовать эти новые типы объектов в среде выполнения, когда у него будет для этого время. Ему не нужно сразу же предоставлять реализацию каждого нового типа объектов, чтобы поддерживать игру в рабочем состоянии. Данные о новых объектах могут спокойно храниться в файле игровой области вне зависимости от того, есть у них реализация или нет, а реализация может существовать без соответствующих данных в файле игровой области.

16.4. Загрузка и потоковая передача игровых миров

Чтобы преодолеть разрыв между локальным редактором игрового мира и объектной моделью времени выполнения, нам нужно как-то загружать области игрового мира в память и выгружать их оттуда, когда они больше не нужны. Система загрузки игровых миров имеет две функции: управление вводом/выводом файлов, необходимых для загрузки с диска в память игровых уровней и других нужных нам ресурсов, а также выделение и очистка памяти для этих ресурсов. Движку также нужно управлять *появлением* и *уничтожением* объектов в игре. Это касается как выделения и очистки памяти для этих объектов, так и создания для них экземпляров подходящих классов. В следующих разделах мы поговорим о загрузке игровых миров и о том, как обычно работают системы создания объектов.

16.4.1. Простая загрузка уровней

Наиболее прямолинейный подход к загрузке уровней, который применялся во всех старейших видеоиграх, состоит в том, что в любой момент в памяти может находиться только одна область (то есть уровень) игрового мира. При запуске игры и в промежутке между уровнями игрок видит статический или слегка анимированный двухмерный экран загрузки.

Управление памятью в такого рода архитектуре выглядит довольно просто. Как говорилось в подразделе 7.2.2, система выделения памяти на основе стека прекрасно подходит для загрузки уровней по одному. При запуске игры на дно стека помещаются все ресурсы, которые могут понадобиться на всех уровнях игры. Здесь мы будем называть их LSR-ресурсами (load and stay resident — «загрузиться и остаться резидентными») (https://ru.wikipedia.org/wiki/Резидентная_программа). После того как LSR-ресурсы полностью загрузятся, записывается указатель стека. Каждая область игрового мира вместе со всеми мешами, текстурами, звуками, анимациями и другими ресурсами загружается в стеке поверх LSR. Когда игрок прошел уровень, всю память можно освободить, сбросив указатель стека к верхнему блоку LSR-ресурсов, а после этого вместо старого уровня загрузить новый (рис. 16.11).

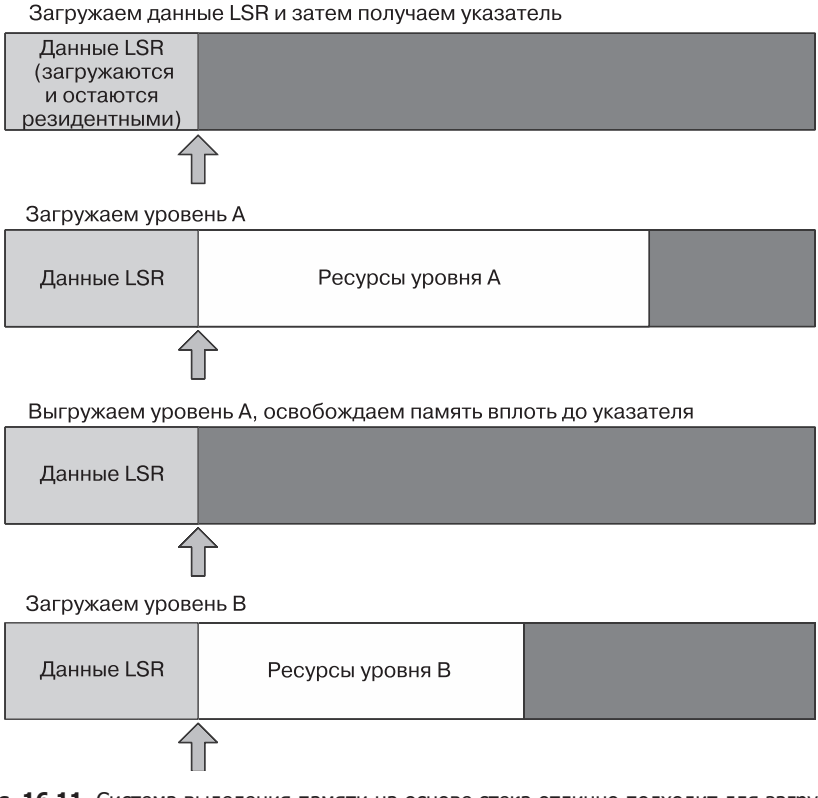


Рис. 16.11. Система выделения памяти на основе стека отлично подходит для загрузки игровых уровней по одному

Несмотря на свою простоту этот подход имеет ряд недостатков. Например, игровой мир показывается игроку в виде отдельных областей — таким образом нельзя создать большой, единый, бесшовный виртуальный мир. Еще одна проблема состоит в том, что во время загрузки ресурсов уровня игровой мир отсутствует в памяти. Поэтому игроку приходится глядеть на двухмерный экран загрузки.

16.4.2. Шаг в направлении бесшовной загрузки: шлюзы

Чтобы не раздражать игрока скучными экранами загрузки, лучше всего позволить ему продолжать игру, пока в память загружается следующая область игрового мира с соответствующими ресурсами. Одна из простых реализаций такого похода заключается в разделении памяти, которую мы выделяем для ресурсов игрового мира, на два блока одного размера. Уровень А можно загрузить в первый блок и сделать его доступным для игрока, а затем загрузить уровень Б во второй блок, используя библиотеку потокового ввода/вывода (в этом случае код загрузки работает в отдельном потоке выполнения). Большой проблемой этого подхода является то, что максимальный размер каждого уровня уменьшается вдвое по сравнению с тем, когда уровни загружаются по одному.

Подобного эффекта можно достичь, разделив память игрового мира на два блока разного размера: большой блок может содержать полноценный уровень, а маленький способен вместить только крошечную игровую область — шлюз.

При запуске игры загружаются обе области. Игрок проходит полноценный уровень и попадает в шлюз. На этом этапе ворота или какое-то другое препятствие не дают ему увидеть пройденный полноценный уровень или вернуться в него. Затем большую область можно выгрузить и вместо нее загрузить следующую. Во время загрузки игрок занят выполнением в шлюзе какого-то задания. Таким заданием может быть простая прогулка из одного конца коридора в другой или что-то более интересное вроде головоломки или битвы с врагами.

Загрузка большой игровой области в то время, как игрок находится в шлюзе, возможна благодаря асинхронному файловому вводу/выводу. Подробнее об этом говорилось в подразделе 7.1.3. Следует отметить, что система шлюзов *не* освобождает нас от вывода экрана загрузки при запуске новой игры, так как в этот момент в памяти нет игрового мира, в котором можно было бы действовать. Но после попадания в игровой мир игроку больше не придется смотреть на экран загрузки, и все благодаря шлюзам и асинхронному чтению данных.

Похожий принцип использовался в *Halo* для Xbox. Большие области игрового мира соединялись между собой за счет более мелких ограниченных зон. Во время игры в *Halo* каждые 5–10 мин встречаются участки, которые не дают вам вернуться назад. В *Jak 2* для PlayStation 2 тоже применялся подход со шлюзами. Игровой мир состоял из центральной области (главный город) с рядом ответвлений, каждое из которых соединялось с хабом с помощью небольшого ограниченного региона.

16.4.3. Потоковая загрузка игрового мира

Многие игровые архитектуры нацелены на то, чтобы игрок чувствовал, будто он находится в огромном, едином, бесшовном мире. В идеале не должно быть периодического попадания в ограниченные регионы-шлюзы — лучше, если мир разворачивается перед игроком максимально естественно и правдоподобно.

Для поддержки бесшовных миров в современных игровых движках используется методика *потоковой загрузки*. Ее можно реализовать множеством способов. Перед нами стоят две основные цели: загружать данные в то время, как пользователь вовлечен в обычный игровой процесс, и управлять памятью таким образом, чтобы избежать *фрагментации*, но при этом сделать возможными загрузку и выгрузку данных по мере продвижения игрока по игровому миру.

У консолей и ПК последнего поколения куда больше памяти, чем у их предшественников, поэтому теперь в памяти можно одновременно хранить несколько игровых областей. Мы могли бы, к примеру, разделить адресное пространство на три буфера одинакового размера. Сначала мы загружаем в них области А, Б и В, позволяя игроку начать с первой из них. Когда он, перейдя в область Б, находится достаточно далеко для того, чтобы предыдущая область не была видна, можем выгрузить область А и начать загрузку в первый буфер области Г. Когда область Б больше не видна, ее можно удалить и загрузить Д. Круговорот буферов может продолжаться до тех пор, пока игрок не достигнет конца единого игрового мира.

Проблема с таким грубым подходом к потоковой загрузке мира состоит в том, что он жестко ограничивает размер игровой области. Все области в игре должны иметь примерно один и тот же объем — быть достаточно большими для того, чтобы заполнить собой бóльшую часть одного из трех буферов памяти и в то же время не выйти за его пределы.

Одним из решений этой проблемы является использование более тонкого разделения памяти. Вместо потоковой загрузки довольно больших частей игрового мира мы можем разделить все ресурсы игры, включая игровые области, динамические меши, текстуры и банки анимации, на блоки данных одинакового размера. А далее задействовать блочную систему выделения памяти на основе пула (наподобие описанной в подразделе 7.2.2) для загрузки и выгрузки ресурсов по мере необходимости, не беспокоясь о фрагментации памяти. Этот подход, в сущности, применяется в движке Naughty Dog (вместе с ним задействуются сложные методики для использования свободного пространства в не полностью занятых областях).

Определение того, какие ресурсы нужно загружать

Один из вопросов, возникающих в ходе работы с системой выделения памяти с множеством мелких блоков для потоковой загрузки игрового мира, звучит так: откуда движок узнает, какие ресурсы следует загружать в тот или иной момент игрового процесса? В движке Naughty Dog мы применяем относительно простой

механизм *регионов для загрузки уровней*, чтобы управлять процессом загрузки и выгрузки ресурсов.

События всех игр серий *Uncharted* и *The Last of Us* разворачиваются в нескольких разных географических зонах. Например, в *Uncharted: Drake's Fortune* игрок находится в джунглях и на острове. Каждый из этих миров существует в едином однородном пространстве, разделенном на множество территориально смежных участков. Все участки охватываются простым выпуклым многогранником — *регионом*; частично регионы перекрываются. Каждый из них содержит список областей игрового мира, которые должны быть в памяти в момент, когда игрок в нем находится.

Игрок всегда пребывает внутри одного или нескольких регионов. Чтобы определить набор игровых областей, которые должны содержаться в памяти, мы просто берем *объединение* списков областей каждого региона, вмещающего в себя персонажа Натана Дрейка. Система загрузки уровней периодически проверяет основной список и сравнивает его с набором областей игрового мира, которые уже находятся в памяти. Если область исчезает из списка, она выгружается, освобождая тем самым все занимаемые ею блоки. Если в списке появляется новая область, она загружается в любые блоки, которые удается найти. Регионы для загрузки уровней и игровые области проектируются таким образом, чтобы игрок никогда не видел исчезновения выгружаемой области и чтобы между тем, как область начинает загружаться, и моментом, когда ее содержимое впервые становится видимым игроку, проходило достаточно времени для того, чтобы она успела полностью загрузиться в память (рис. 16.12).

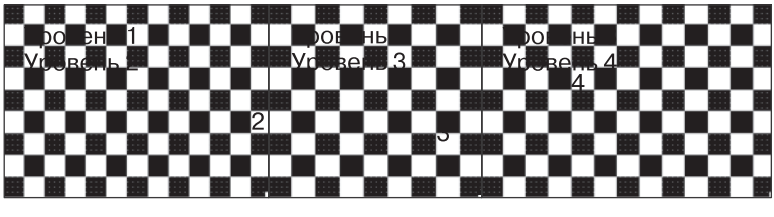


Рис. 16.12. Игровой мир разделен на области. Все регионы для загрузки уровней содержат список запрашиваемых областей и организованы таким образом, чтобы появление или исчезновение области никогда не происходило в поле зрения игрока

PlayGo на PlayStation 4

В PlayStation 4 появилась новая возможность под названием PlayGo, которая делает процесс загрузки игры намного менее болезненным по сравнению с традиционной покупкой Blu-ray. PlayGo загружает минимальный набор данных, необходимый для прохождения первого участка игры. Остальной игровой контент незаметно загружается в фоне, пока вы играете. Конечно, для работы этого механизма необходимо, чтобы игра поддерживала бесшовную потоковую загрузку уровней, описанную ранее.

16.4.4. Управление памятью для создания объектов

После загрузки игрового мира в память нужно как-то управлять процессом *появления* в нем динамических объектов. У большинства игровых движков есть какого-то рода система создания объектов, отвечающая за создание экземпляра или экземпляров классов, из которых состоит каждый игровой объект, и за удаление игровых объектов, когда они больше не нужны. Одной из основных задач такой системы является динамическое выделение памяти для появляющихся объектов. Динамическое выделение может быть медленным, поэтому нужно сделать все для того, чтобы оно было как можно более эффективным. И поскольку размеры игровых объектов могут существенно варьироваться, их динамическое выделение способно привести к фрагментации памяти и вызвать тем самым ее преждевременную нехватку. Существует ряд методов управления памятью игровых объектов. В следующих разделах мы исследуем несколько самых распространенных из них.

Предварительное выделение памяти для новых объектов

Чтобы решить проблемы со скоростью выделения и фрагментацией памяти, некоторые игровые движки принимают суровые меры, попросту запрещая динамическое выделение памяти в ходе игрового процесса. Области игрового мира по-прежнему могут загружаться и выгружаться динамически, но во время загрузки они сразу же создают все свои динамические элементы. После этого ни один игровой объект не может быть создан или уничтожен. Можно сказать, что эта методика подчиняется закону сохранения игровых объектов: после загрузки игрового мира его объекты больше не создаются и не уничтожаются.

Этот подход позволяет избежать фрагментации памяти, поскольку размер всех объектов в игровой области известен заранее и не меняется. Это означает, что еще в редакторе игрового мира можно выделить память для всех игровых объектов и сделать ее частью данных самой игровой области. Таким образом, все игровые объекты создаются из одного участка памяти, который задействуется для загрузки игрового мира и его ресурсов, поэтому они подвержены фрагментации не более, чем любые другие ресурсы. Еще одним преимуществом этого подхода является то, что использование памяти игрой становится очень предсказуемым. Можно не волноваться о том, что в какой-то момент в игровом мире неожиданно появится большая группа игровых объектов и вызовет нехватку памяти.

В то же время эта методика может довольно сильно ограничивать возможности игровых дизайнеров. Появление игровых объектов можно симулировать, создавая их в редакторе игрового мира, при этом изначально они должны оставаться невидимыми и бездействующими. Позже, чтобы объект появился, его можно активировать и сделать видимым. Однако игровым дизайнерам нужно предугадать общее количество объектов каждого типа, которые могут понадобиться, еще во время их создания в редакторе. Если они захотят предоставить игроку бесконечное количество аптечек, оружия, врагов или каких-то других объектов, им придется выработать механизм повторного использования игровых элементов, иначе ничего не получится.

Динамическое выделение памяти для новых объектов

Игровые дизайнеры, наверное, отдали бы предпочтение движку, который поддерживает настоящее динамическое создание объектов. И хотя данный подход сложнее реализовать по сравнению со статическим, это можно сделать несколькими разными способами.

И снова главной проблемой является фрагментация памяти. Поскольку разные типы игровых объектов, а иногда и разные экземпляры одного и того же типа объектов, имеют разный размер, мы не можем воспользоваться нашей любимой системой выделения памяти на основе пула, устойчивой к фрагментации. К тому же игровые объекты обычно удаляются не в том порядке, в котором они были созданы, поэтому механизм выделения на основе стека тоже не годится. Похоже, что единственным выбором является подверженная фрагментации куча. К счастью, с фрагментацией можно бороться множеством разных способов. Несколько наиболее популярных из них мы исследуем в следующих разделах.

По одному пулу памяти для каждого типа объектов. Если отдельные экземпляры каждого типа игровых объектов всегда имеют одинаковый размер, для каждого типа можно использовать свой пул памяти. На самом деле в пул необходимо объединять любые объекты одинакового размера, поэтому в нем может находиться несколько типов.

Это позволяет полностью исключить фрагментацию памяти. Однако ограничение данного подхода состоит в том, что нам придется поддерживать множество отдельных пулов. Также нужно будет сделать обоснованное предположение о том, сколько объектов каждого типа понадобится. Слишком большое количество элементов в одном пуле приведет к пустой трате памяти, а если их будет слишком мало, мы не сможем удовлетворить все запросы на создание в ходе выполнения, в результате чего объекты не смогут появиться в игре.

Выделение небольших участков памяти. Идею предоставления по одному пулу для каждого типа объектов можно сделать более жизнеспособной, если разрешить выделять объект из пула, элементы которого больше самого объекта. Это может существенно сократить количество нужных пулов, хотя при этом в каждом из них может теряться какое-то количество памяти.

Например, мы можем создать набор пулов, размер элементов в каждом из которых будет в два раза больше, чем в предыдущем, — скажем, 8, 16, 32, 64, 128, 256 и 512 байт. Можем выбрать размеры, которые подходят для какого-то другого сценария использования, или основать свой выбор на статистике, собранной из работающей игры.

При каждой попытке выделения игрового объекта мы ищем наименьший пул, размер элементов которого превышает размер самого объекта или равен ему. Нужно смириться с тем, что в некоторых случаях место будет затрачиваться впустую. Но зато исчезнут все проблемы с фрагментацией памяти — довольно неплохой компромисс. Если нам когда-нибудь придет запрос на выделение, который превышает по размеру самый крупный пул, мы всегда можем перенаправить его к системе

выделения общего назначения на основе кучи, так как фрагментация больших блоков памяти куда менее проблематична, чем маленьких.

Этот способ выделения памяти может устранить фрагментацию (для объектов, которые влезают в один из пулов). Он также может существенно ускорить выделение небольших блоков данных, так как для удаления свободных элементов из связного списка пул манипулирует двумя указателями, что является куда менее ресурсоемкой операцией, чем универсальный подход с выделением памяти в куче.

Перемещение памяти. Еще один способ устранения фрагментации заключается в борьбе с изначальной проблемой. Этот подход известен как перемещение памяти. Он подразумевает перемещение блоков памяти в смежные свободные участки. Перемещать память просто, однако мы двигаем живые объекты, поэтому необходимо тщательно исправить все указатели на перемещаемые блоки. Подробнее об этом говорилось в подразделе 6.2.2.

16.4.5. Сохраненные игры

Многие игры позволяют игроку сохранить данные о пройденных уровнях, выйти из игры, а через какое-то время загрузить ее именно в том состоянии, в котором она была оставлена. Система сохранения игры похожа на механизм загрузки областей игрового мира: она умеет загружать свое состояние с файла на диске или с карты памяти. Однако к ней предъявляются другие требования, поэтому ее обычно реализуют отдельно или частично, используя механизм загрузки игрового мира.

Чтобы понять, в чем заключаются различия между требованиями к этим двум системам, сравним области игрового мира и файлы с сохраненными играми. Игровые области определяют начальное состояние всех динамических объектов виртуального мира, также они содержат полное описание всех статических элементов. Статическая информация, такая как фоновые меши и данные о столкновениях, как правило, занимает много места на диске. Поэтому область иногда разбивают на несколько файлов, а общий объем данных, который с ней связан, обычно довольно велик.

Файл с сохраненной игрой тоже должен содержать информацию о текущем состоянии элементов игрового мира. Но в нем не нужно хранить копию данных, которые можно получить из игровой области. Например, не обязательно сохранять статическую геометрию. К тому же сохраненная игра не должна содержать все подробности о состоянии каждого объекта. Некоторые объекты не влияют на игровой процесс, поэтому ими можно пренебречь. В некоторых случаях можно обойтись частичным сохранением состояния. Система сохранения игры считается удачной, если игрок не может увидеть различий в состоянии игрового мира до выхода и после его загрузки или если для игрока они неважны. Таким образом, файлы с сохранениями обычно имеют намного меньший размер, чем файлы игровых областей: они могут активнее использовать сжатие и пропускать ненужную информацию. Это было особенно важно, когда множество файлов с сохраненными играми

требовалось уместить на одну из тех крошечных карт памяти, которые применялись в старых консолях. Но даже сегодня, когда консоли имеют большие жесткие диски и подключены к системе облачного сохранения, файлы с сохраненными играми лучше делать как можно более компактными.

Контрольные точки

Еще один подход к сохранению игрового процесса состоит в том, чтобы ограничиться определенными этапами игры, известными как *контрольные точки*. Его преимуществом является то, что бóльшая часть информации о состоянии игры хранится в текущей области (или областях) игрового мира вблизи каждой контрольной точки. Эти данные никогда не меняются, независимо от того, кто играет, поэтому их не нужно сохранять вместе с игрой. Благодаря этому файлы с сохраненной игрой, основанные на контрольных точках, могут быть чрезвычайно компактными и состоять из названия последней достигнутой точки и, возможно, каких-то сведений о текущем состоянии игрового персонажа, например о здоровье игрока, количестве оставшихся жизней, предметах, находящихся в его инвентаре, имеющемся оружии и боеприпасах. Некоторые игры, основанные на контрольных точках, не сохраняют даже эту информацию — при каждой загрузке игрок начинает игру в заранее известном состоянии. Конечно, недостатком такой системы является то, что она может не понравиться пользователю, особенно если контрольных точек мало и между ними большое расстояние.

Сохранение в любом месте

Некоторые игры поддерживают такую возможность, как *сохранение в любом месте*. Как понятно из названия, она позволяет сохранить состояние игры буквально в любой момент игрового процесса. Реализация этого подхода требует существенного увеличения файлов с сохраненными играми. Текущее местоположение и внутреннее состояние каждого игрового объекта, влияющего на игровой процесс, должны быть сохранены и загружены при очередном запуске игры.

При использовании такой архитектуры файл с сохраненной игрой содержит практически ту же информацию, что и игровая область, за исключением статических компонентов игрового мира. Некоторые форматы данных можно применять для обеих систем, хотя в некоторых случаях эта задача может оказаться невыполнимой, например, когда формат данных для игровых областей специально сделан гибким, а формат сохранения игры сжимается для экономии места.

Как уже упоминалось, чтобы уменьшить количество данных, которые необходимо размещать в файле с сохраненными играми, можно опустить не слишком важные игровые объекты и некоторые незначительные подробности других элементов. Например, нам не нужно помнить временной индекс каждой анимации, которая сейчас воспроизводится, или точный импульс/скорость каждого твердого тела в физической симуляции. Мы можем рассчитывать на то, что воспоминания игроков неидеальны, и сохранять приблизительное состояние игры.

16.5. Ссылки на объекты и запросы к игровому миру

Каждый игровой объект должен обладать уникальным идентификатором, чтобы его можно было отличить от других объектов в игре, найти во время выполнения, использовать для межобъектного взаимодействия и т. д. Уникальные идентификаторы в равной степени полезны и на этапе разработки, так как с их помощью можно идентифицировать и искать игровые элементы в редакторе игрового мира.

На этапе выполнения нам нужные различные механизмы для поиска игровых объектов. Для этого можно задействовать уникальный идентификатор объекта, его тип или произвольный набор критериев. Часто возникает необходимость в запросах, основанных на удаленности, например, чтобы найти всех враждебных пришельцев в радиусе 10 м от игрока.

Отыскав игровой объект с помощью запроса, мы должны как-то на него сослаться. В таких языках, как C или C++, ссылки на объекты могут быть реализованы в виде указателей или с использованием чего-то более утонченного, например дескрипторов или умных указателей. Время жизни ссылки на объект может сильно варьироваться: от периода выполнения одной функции до многих минут. В следующих разделах мы сначала исследуем различные способы реализации ссылок на объекты, а затем рассмотрим виды запросов, которые могут потребоваться в ходе написания игрового процесса, и то, как эти запросы можно реализовать.

16.5.1. Указатели

В C или C++ ссылку на объект проще всего реализовать в виде указателя (или ссылки в C++). Указатели являются мощным механизмом, но при этом остаются максимально простыми и понятными. Тем не менее для указателей характерен целый ряд проблем.

- *Покинутые объекты*. В идеале у каждого объекта должен быть *владелец* — другой объект, ответственный за управление его жизненным циклом, за его создание и удаление, когда он больше не нужен. Но указатели никак с этим не помогают. В итоге может возникнуть *покинутый* объект, все еще находящийся в памяти, но больше никому не нужный и на который не ссылается ни один объект в системе.
- *Устаревшие указатели*. В идеале при удалении объекта следует обнулить все указатели на него. Если этого не сделать, получится устаревший указатель, который указывает на ныне свободный блок памяти, принадлежавший реальному объекту. Попытка чтения из устаревшего указателя или записи в него может вызвать сбой в программе или ее некорректное поведение. Устаревшие указатели может быть сложно отыскать, так как они способны оставаться рабочими на протяжении некоторого времени после удаления объекта. Сбой происходит позже, когда поверх освобожденного блока выделяется новый объект и данные реально меняются.

- *Недопустимые указатели*. Программист может хранить в указателе любой адрес, в том числе совершенно недопустимый. Разыменование нулевых указателей — распространенная проблема. Чтобы от нее уберечься, можно использовать отладочный макрос, который позволяет убедиться в том, что мы не разыменовываем нулевой указатель. Бывает и так, что в качестве указателя по ошибке интерпретируется участок данных, в этом случае его разыменовывание может заставить программу прочитать или записать практически случайный адрес в памяти. Это обычно приводит к сбою или другим существенным проблемам, которые может оказаться очень сложно отладить.

Многие игровые движки активно применяют указатели, так как это самый быстрый, эффективный и простой в использовании способ реализации ссылок на объекты. Но опытные программисты относятся к указателям с осторожностью, а некоторые игровые студии выбирают более сложные виды ссылок на объекты. Это может быть вызвано необходимостью или желанием задействовать более безопасные методы программирования. Например, если игровой движок перемещает выделенные блоки данных во время выполнения, чтобы избавиться от фрагментации (см. подраздел 6.2.2), обычные указатели не подойдут. Нужно сделать одно из двух: либо использовать тип ссылок на объекты, устойчивый к перемещению памяти, либо вручную и на ходу исправить все указатели во всех перемещаемых блоках.

16.5.2. Умные указатели

Умный указатель — это небольшой объект, который внешне ведет себя как обычные указатели в C/C++, но позволяет избежать большинства присущих им проблем. В самом простом виде этот объект содержит поле со стандартным указателем и предоставляет набор перегруженных операторов, которые в большинстве случаев позволяют ему демонстрировать соответствующее поведение. Указатели могут быть разыменованы, поэтому операторы * и -> перезагружаются так, чтобы возвращать соответственно ссылку и указатель на объект, на который мы ссылаемся. Это поведение, которого мы ожидаем. Перезагружаются также операторы +, -, ++ и --, так как указатели поддерживают арифметические операции.

Поскольку умный указатель является объектом, он может содержать метаданные и/или предпринимать дополнительные шаги, недоступные обычному указателю. Например, в нем может храниться информация, позволяющая узнать о том, что объект, на который он указывает, удаляется. Если это случится, он может начать возвращать нулевой адрес.

Умные указатели могут помочь также с управлением жизненным циклом объектов, взаимодействуя друг с другом для определения количества ссылок на определенный объект. Это называется *подсчетом ссылок*. Когда количество умных указателей, ссылающихся на тот или иной объект, уменьшается до нуля, мы можем быть уверены в том, что этот объект больше не нужен, поэтому его можно автоматически удалить. Благодаря этому программистам больше не нужно волноваться

о покинутых объектах и о том, чем они владели. Подсчет ссылок обычно лежит в основе систем сборки мусора, имеющихся в современных языках программирования вроде Java и Python.

У умных указателей есть и недостатки. Например, их легко реализовать, но сложно реализовать правильно. Необходимо учитывать огромное количество разных случаев, поэтому тот факт, что класс `std::auto_ptr` из оригинальной стандартной библиотеки C++ не подходит во многих ситуациях, является общепризнанным. К счастью, большинство этих проблем были решены в C++11 с появлением трех классов умных указателей: `std::shared_ptr`, `std::weak_ ptr` и `std::unique_ptr`.

Эти классы были смоделированы на основе мощного механизма умных указателей из библиотеки шаблонов Boost C++. Библиотека предоставляет шесть разновидностей умных указателей:

- `boost::scoped_ptr` — указатель на один объект с одним владельцем;
- `boost::scoped_array` — указатель на массив объектов с единым владельцем;
- `boost::shared_ptr` — указатель на объект, жизненным циклом которого управляют несколько владельцев;
- `boost::shared_array` — указатель на массив объектов, жизненными циклами которых управляют несколько владельцев;
- `boost::weak_ptr` — указатель, не владеющий объектом, на который он ссылается, и не уничтожающий его автоматически (подразумевается, что за управление его жизненным циклом отвечает `shared_ptr`);
- `boost::intrusive_ptr` — указатель, реализующий подсчет ссылок, исходя из того, что объект, на который он ссылается, сам считает эти ссылки. Польза этих указателей в том, что их размер совпадает с размером стандартного указателя в C++ (так как им не требуется механизм подсчета ссылок) и их можно создавать непосредственно на его основе.

Корректная реализация класса умных указателей может оказаться очень сложной задачей. Чтобы понять, что я имею в виду, взгляните на документацию для умных указателей в Boost (www.boost.org/doc/libs/1_36_0/libs/smart_ptr/smart_ptr.htm). Там поднимаются всевозможные вопросы, включая следующие:

- типобезопасность умных указателей;
- возможность использования умных указателей для неполных типов;
- корректное поведение умного указателя при возникновении исключения;
- накладные расходы на этапе выполнения (могут быть большими).

Как-то я участвовал в проекте, в котором предпринимались попытки реализовать собственные умные указатели, и в связи с этим вплоть до завершения работы нам приходилось исправлять всевозможные ошибки. Я советую держаться от умных указателей подальше. Если же вам нужно их использовать, выберите зрелую реализацию, такую как стандартная библиотека C++11 или Boost, а не пишите собственную.

16.5.3. Дескрипторы

Дескриптор во многом ведет себя как умный указатель, но его легче реализовать и он меньше подвержен разным проблемам. Это, в сущности, целочисленный индекс для глобальной *таблицы дескрипторов*, а она, в свою очередь, хранит указатели на объекты, на которые эти дескрипторы ссылаются. Чтобы создать дескриптор, мы ищем в таблице адрес нужного объекта и сохраняем его индекс. Чтобы разыменовать дескриптор, вызывающий код просто находит соответствующий слот в таблице и разыменовывает указатель, который там находится (рис. 16.13).

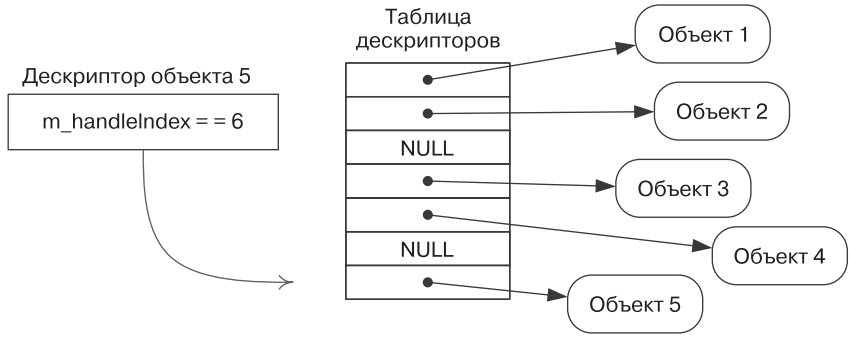


Рис. 16.13. Таблица дескрипторов содержит стандартные указатели на объекты. Дескриптор является лишь индексом для этой таблицы

Благодаря такой непрямой связи дескрипторы являются куда более безопасными и гибкими по сравнению со стандартными указателями. При удалении объекта мы можем просто обнулить его слот в таблице дескрипторов. Таким образом все существующие дескрипторы этого объекта автоматически превратятся в нулевые ссылки. Дескрипторы также поддерживают перемещение объектов в памяти. Для этого нужно найти адрес объекта в таблице и обновить его соответствующим образом. Опять же в результате все существующие дескрипторы объекта автоматически обновятся.

Дескриптор можно реализовать в виде обычного целого числа. Однако индексы таблицы дескрипторов обычно заворачивают в простой класс, чтобы предоставить удобный интерфейс для создания и разыменовывания дескрипторов.

Дескрипторы потенциально могут ссылаться на устаревший объект. Представьте, к примеру, что мы создаем дескриптор объекта А, который занимает в таблице дескрипторов слот 17. Позже объект А удаляется, а слот 17 обнуляется. Вслед за этим создается объект Б и в результате стечения обстоятельств попадает в слот 17. Если в этот момент у нас все еще есть дескрипторы объекта А, они внезапно начнут ссылаться на объект Б, а не на ноль. Нам такое поведение, конечно же, не подходит.

У проблемы с устаревшими дескрипторами есть простое решение: добавить в каждый дескриптор уникальный идентификатор объекта. Таким образом, при создании дескриптор объекта А содержит не только индекс слота 17, но и ID «А».

Когда вместо А в таблице дескрипторов создается объект Б, у всех оставшихся дескрипторов А будет подходящий индекс, но не тот идентификатор. Благодаря этому дескрипторы устаревшего объекта А после разыменования будут по-прежнему возвращать ноль, а не указатель на объект Б, которого мы не ожидаем.

В следующем фрагменте кода показано, как можно реализовать простой класс дескрипторов. Обратите внимание на то, что мы включили индекс дескриптора в сам объект `GameObject`. Это позволит очень быстро создавать новые дескрипторы данного объекта, причем не потребуется искать его адрес в таблице, чтобы определить его индекс.

```
// Чтобы создание новых дескрипторов было эффективным,
// мы храним в классе GameObject уникальный ID и индекс
// дескриптора объекта.

class GameObject
{
private:
    // ...

    GameObjectId m_uniqueId;        // уникальный ID объекта
    U32          m_handleIndex;     // ускоренное создание дескрипторов
    friend class GameObjectHandle;  // доступ к ID и индексу

    // ...

public:
    GameObject() // конструктор
    {
        // Уникальный ID может быть взят из редактора игрового мира
        // или присвоен динамически во время выполнения.
        m_uniqueId = AssignUniqueObjectId();
        // Чтобы присвоить индекс дескриптора, мы ищем первый
        // свободный слот в таблице дескрипторов.
        m_handleIndex = FindFreeSlotInHandleTable();

        // ...
    }
    // ...

};
// Эта константа определяет размер таблицы дескрипторов и, следовательно,
максимальное
// количество игровых объектов,
// которые могут существовать одновременно.
static const U32 MAX_GAME_OBJECTS = 2048;

// Это глобальная таблица дескрипторов — простой
// массив указателей на экземпляры GameObject.
static GameObject* g_apGameObject[MAX_GAME_OBJECTS];
```

```
// Это простой класс дескрипторов для игровых объектов.
class GameObjectHandle
{
private:
    U32 m_handleIndex;        // индекс в таблице дескрипторов
    GameObjectId m_uniqueId;  // уникальный ID позволяет избежать
                              // устаревания дескрипторов

public:
    explicit GameObjectHandle(GameObject& object) :
        m_handleIndex(object.m_handleIndex),
        m_uniqueId(object.m_uniqueId)
    {
    }

    // Эта функция разыменовывает дескриптор.
    GameObject* ToObject() const
    {
        GameObject* pObject
        = g_apGameObject[m_handleIndex];

        if (pObject != nullptr
        && pObject->m_uniqueId == m_uniqueId)
        {
            return pObject;
        }
        return nullptr;
    }
};
```

Этот пример рабочий, но неполный. Здесь следует реализовать семантику копирования, предоставить дополнительные варианты конструкторов и т. д. Элементы глобальной таблицы дескрипторов могут содержать дополнительную информацию, а не только указатели на каждый игровой объект. И конечно, реализация таблицы дескрипторов с жестко заданным размером не является единственным возможным подходом — системы дескрипторов могут различаться в разных движках.

Следует отметить, что у глобальной таблицы дескрипторов есть один положительный побочный эффект — она предоставляет готовый список всех активных игровых объектов в системе. С помощью этой таблицы можно, к примеру, быстро и эффективно перебрать все элементы игрового мира. В некоторых случаях это упрощает реализацию других видов запросов.

16.5.4. Запросы игровых объектов

Любой игровой движок предоставляет как минимум несколько способов поиска игровых объектов на этапе выполнения. Мы будем называть их *запросами игровых объектов*. Самый простой вид запросов заключается в получении игрового объекта по его уникальному идентификатору. Но в настоящих игровых движках это далеко

не единственный способ. Далее перечислено лишь несколько примеров того, какие виды запросов могут понадобиться разработчикам игры.

- Найти всех вражеских персонажей, находящихся в поле зрения игрока.
- Перебрать все игровые объекты определенного типа.
- Найти все разрушаемые игровые объекты со здоровьем больше 80 %.
- Нанести урон всем игровым объектам в радиусе взрыва.
- Перебрать все игровые объекты на пути пули или другого снаряда, начиная с ближайшего.

Этот список мог бы растянуться на многие страницы, а его содержание, конечно же, напрямую зависит от архитектуры конкретной игры.

Чтобы сделать запросы игровых объектов максимально гибкими, мы можем представить себе базу данных общего назначения, поддерживающую запросы произвольного вида с произвольными поисковыми критериями. В идеале наша база данных игровых объектов должна выполнять все эти запросы чрезвычайно эффективно и быстро, максимально задействуя все доступные аппаратные и программные ресурсы.

В реальности такое идеальное сочетание гибкости и невероятной скорости обычно недостижимо. Вместо этого игровые студии, как правило, определяют виды запросов, которые им больше всего будут нужны во время разработки игры, и реализуют специальные структуры данных, чтобы ускорить эти конкретные запросы. По мере того как возникает необходимость в новых запросах, инженеры могут использовать уже имеющиеся структуры данных, если же им не удается достичь достаточной скорости — создать новые. Вот несколько примеров специализированных структур данных, которые могут ускорить определенные типы запросов игровых объектов.

- *Поиск игровых объектов по уникальному идентификатору*. Указатели или дескрипторы для игровых объектов можно хранить в хеш-таблице или двоичном дереве поиска с уникальными идентификаторами в качестве ключей.
- *Перебор всех объектов, удовлетворяющих определенному критерию*. Игровые объекты можно заранее сортировать по связным спискам с учетом различных критериев, если они известны заблаговременно. Например, мы можем сформировать список всех игровых объектов определенного типа или хранить список всех объектов, находящихся в окружности определенного радиуса, в центре которой располагается игрок.
- *Поиск всех объектов на пути снаряда или между игроком и какой-то целевой точкой*. Для такого рода запросов обычно используется система столкновений. Большинство систем столкновений поддерживают быстрый рейкастинг, а некоторые из них позволяют отбрасывать на игровой мир не только лучи, но и разные примитивы вроде сфер или произвольных многогранников, чтобы определить, какие объекты те затрагивают (см. подраздел 13.3.7).

- *Поиск всех объектов в пределах заданного региона или радиуса.* Мы можем выбрать для хранения игровых объектов некую пространственную структуру данных на основе хешей. Для этого достаточно наложить на игровой мир горизонтальную сетку или что-то более сложное вроде дерева квадрантов, октодерева, k-мерного дерева или другой структуры данных для кодирования пространственной удаленности.

16.6. Обновление игровых объектов в реальном времени

Любым игровым движкам, как простейшим, так и более сложным, нужен какой-то механизм для периодического обновления внутреннего состояния каждого игрового объекта. *Состояние* игрового объекта может определяться значениями всех его *атрибутов* (их иногда называют *свойствами* или *членами класса* в языке C++). Например, состояние шарика в игре *Pong* описывается его позицией на экране (x, y) и вектором скорости (скорость плюс направление движения). Поскольку игры представляют собой динамичные симуляции с привязкой ко времени, состояние игрового объекта описывает его конфигурацию в *определенный момент*. Иными словами, время в игровом мире является *дискретным*, а не *непрерывным* (хотя, как мы позже увидим, изменение состояния объектов лучше воспринимать как непрерывный процесс, который дискретизируется движком, так как это позволяет избежать распространенных проблем).

Далее мы будем обозначать состояние объекта i в произвольный момент времени t как $\mathbf{S}_i(t)$. С математической точки зрения такое использование векторного обозначения не совсем верно, но это позволяет помнить о том, что состояние игрового объекта ведет себя как гетерогенный n-мерный вектор, содержащий всевозможные данные разных типов. Следует отметить, что термин «состояние» здесь имеет не такой смысл, как в контексте *конечного автомата*. Да, игровой объект можно реализовать в виде одного или множества конечных автоматов, но в этом случае их текущее состояние будет лишь частью общего вектора состояния объекта $\mathbf{S}(t)$.

Большинство низкоуровневых подсистем движка (отрисовка, анимация, столкновения, физика, звук и т. д.) требуют периодического обновления, и система игровых объектов не исключение. Как мы уже видели в главе 8, для обновления обычно используется единый главный цикл, который называют *игровым циклом*. Практически все игровые движки обновляют состояние своих объектов в этом цикле — иными словами, они обращаются с объектной моделью игры, как с любой другой подсистемой, требующей периодического обслуживания.

Таким образом, обновление игровых объектов можно воспринимать как процесс определения состояния каждого объекта в текущий момент времени $\mathbf{S}_i(t)$ с учетом его состояния в предыдущий момент $\mathbf{S}_i(t - \Delta t)$. После обновления состояния всех объектов текущее время t становится предыдущим, $(t - \Delta t)$, и этот процесс повторяется на протяжении всей игры. Обычно в движке предусмотрены один или

несколько *таймеров* — один для отслеживания точного реального времени и, возможно, несколько дополнительных, которые могут быть привязаны к реальному времени, а могут — и нет. Эти таймеры предоставляют движку абсолютное время t и/или разницу во времени Δt между итерациями игрового цикла. Таймер, на основе которого происходит обновление состояния игровых объектов, обычно может отклоняться от реального времени. Благодаря этому можно останавливать, замедлять, ускорять игровые объекты или даже работать задом наперед — это определяется потребностями архитектуры игры. Кроме того, данные возможности бесценны в ходе отладки и разработки.

Как вы уже знаете из главы 1, система обновления игровых объектов является примером того, что в информатике называют *динамическим, агентным компьютерным моделированием в реальном времени*. Таким системам также присущи некоторые аспекты *дискретно-событийного моделирования* (подробнее о событиях — в разделе 16.8). Это хорошо изученные области информатики, и у них есть много применений вне сферы интерактивных развлечений. Игры являются одной из более сложных разновидностей агентного моделирования. Как вы вскоре увидите, периодическое обновление состояния игровых объектов в динамичном, интерактивном виртуальном окружении может быть на удивление сложным, если делать это правильно. Программисты могут узнать много нового об обновлении игровых объектов, более широко изучив область агентного и дискретного событийного моделирования. Точно так же исследователи в этих областях могут что-нибудь почерпнуть из архитектуры игровых движков!

Реализация любой высокоуровневой системы может слегка (или радикально) различаться от движка к движку. Но, как и прежде, большинство игровых студий сталкиваются с одними и теми же проблемами, а некоторые шаблоны проектирования приживаются снова и снова практически во всех движках. В этом разделе мы исследуем общие проблемы и их общие решения. Пожалуйста, помните о том, что некоторые движки могут подходить к решению описанных здесь проблем совершенно иначе, а у каких-то игровых архитектур есть уникальные проблемы и у нас нет возможности рассмотреть их все.

16.6.1. Простой подход (который не работает)

Самый простой подход к обновлению состояния коллекции игровых объектов выглядит так: перебрать все элементы коллекции, вызывая для каждого из них по очереди виртуальную функцию с именем вроде `Update()`. Обычно это делается при каждой итерации главного игрового цикла, то есть для каждого *кадра*. Классы игровых объектов могут предоставлять собственные реализации функции `Update()`, чтобы выполнять задачи, необходимые для продвижения состояния этого типа объектов к следующему дискретному индексу времени. Разницу во времени по сравнению с предыдущим кадром можно передать в функцию обновления, чтобы объекты правильно оценивали ход времени. В простейшем случае сигнатура функции `Update()` может выглядеть примерно так:

```
virtual void Update(float dt);
```

В дальнейшем будем исходить из того, что наш движок имеет монолитную иерархию объектов, в которой каждый игровой объект представлен одним экземпляром одного класса. Этот принцип можно легко применить практически к любой объектно-ориентированной архитектуре. Например, для обновления объектной модели на основе компонентов мы могли бы вызывать `Update()` для каждого компонента, из которых состоит игровой объект, или только для центрального объекта, который затем обновит все связанные с ним компоненты так, как считает нужным. Этот же подход можно использовать и в архитектурах на основе свойств. Для этого некую разновидность функции `Update()` можно вызывать в каждом кадре для каждого экземпляра свойства.

Говорят, дьявол кроется в деталях, поэтому исследуем здесь два важных вопроса. Во-первых, каким образом следует хранить коллекцию всех игровых объектов? И во-вторых, за обновление чего должна отвечать функция `Update()`?

Хранение коллекции активных игровых объектов

Коллекцией активных игровых объектов часто управляет класс-синглтон с именем вроде `GameObjectManager`. Такая коллекция, как правило, должна быть динамической, поскольку в процессе игры объекты появляются и исчезают. Таким образом, простым и эффективным вариантом является *связный список* указателей, умных указателей или дескрипторов игровых объектов (некоторые игровые движки не поддерживают динамическое создание и удаление объектов игры, в этом случае можно использовать *массив* указателей, умных указателей или дескрипторов игровых объектов фиксированной длины). Как мы увидим в дальнейшем, в большинстве движков вместо обычного плоского связного списка для отслеживания игровых объектов задействуются более сложные структуры данных. Но пока, чтобы не усложнять, в качестве такой структуры данных можно применить связный список.

Обязанности функции Update()

Функция `Update()` игрового объекта берет на себя основную работу по определению его состояния в текущем дискретном индексе времени $\mathbf{S}_i(t)$ с учетом предыдущего состояния $\mathbf{S}_i(t - \Delta t)$. Для этого могут потребоваться симуляция поведения твердого тела, дискретизация заранее подготовленной анимации, реакция на события, возникшие на текущем временном отрезке, и т. д.

Большинство игровых объектов взаимодействуют с одной или несколькими подсистемами движка. Им могут потребоваться анимация, отрисовка, излучение эффектов частиц, воспроизведение звука, столкновение с другими объектами или статической геометрией и т. д. У каждой из этих подсистем есть внутреннее состояние, которое тоже нужно периодически обновлять, обычно один раз в каждом кадре. Его обновление непосредственно из функции `Update()` игрового объекта может показаться разумным и интуитивным шагом. Возьмем, к примеру, следующую гипотетическую функцию обновления для объекта `Tank`:

```
virtual void Tank::Update(float dt)
{
    // Обновляем состояние самого танка.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // Теперь обновляем низкоуровневые подсистемы движка
    // от имени этого танка (плохая идея… см. далее!)
    m_pAnimationComponent->Update(dt);
    m_pCollisionComponent->Update(dt);
    m_pPhysicsComponent->Update(dt);
    m_pAudioComponent->Update(dt);
    m_pRenderingComponent->draw();
}
```

Если структурировать функции `Update()` таким образом, весь игровой цикл можно построить вокруг обновления игровых объектов, например:

```
while (true)
{
    PollJoypad();

    float dt = g_gameClock.CalculateDeltaTime();

    for (each gameObject)
    {
        // Эта гипотетическая функция Update() обновляет
        // все подсистемы движка!
        gameObject.Update(dt);
    }

    g_renderingEngine.SwapBuffers();
}
```

Какой бы заманчивой ни была продемонстрированная здесь идея, применять ее в игровых движках коммерческого уровня обычно нельзя. В следующих разделах мы исследуем некоторые проблемы, присущие этому упрощенному подходу, и рассмотрим распространенные способы решения каждой из них.

16.6.2. Влияние пакетных обновлений на производительность

Большинство низкоуровневых подсистем движка имеют чрезвычайно высокие требования к производительности. Они работают с огромными объемами данных, и в каждом кадре им необходимо как можно быстрее выполнять большое количество вычислений. Поэтому большинство таких подсистем выигрывают от *пакетного обновления*. Например, намного эффективнее обновлять большое количество анимаций в ходе одного этапа, чем обновлять анимацию каждого объекта среди

прочих операций, не имеющих к этому никакого отношения, таких как обнаружение столкновение, физическая симуляция и отрисовка.

В большинстве коммерческих игровых движков каждая подсистема обновляется напрямую или опосредованно в главном цикле игры, а не в функции `Update()` каждого игрового объекта. Если игровому объекту нужны возможности конкретной подсистемы, он просит ее выделить от своего имени некое состояние, относящееся только к ней. Например, если объект хочет быть отрисованным с помощью треугольного меша, он может попросить подсистему отрисовки выделить для него *экземпляр меша* (как описывалось в подразделе 11.1.1, это отдельный экземпляр треугольного меша; отслеживается его положение, ориентация и масштаб в игровом пространстве независимо от того, видим он или нет, данные о материале и любая другая информация, имеющая значение). В движке отрисовки хранится коллекция экземпляров мешей. Он может управлять ими по своему усмотрению, чтобы максимизировать собственную производительность во время выполнения. Игровой объект определяет то, *как* отрисовывается экземпляр меша, не напрямую, а манипулируя его свойствами. После того как все объекты получили шанс обновиться, движок отрисовки выводит все видимые экземпляры мешей в ходе одного эффективного пакетного обновления.

При использовании пакетного обновления функция `Update()` отдельного игрового объекта, такого как наш гипотетический танк, может выглядеть так:

```
virtual void Tank::Update(float dt)
{
    // Обновляем состояние самого танка
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // Управляем свойствами различных подсистем
    // движка,но НЕ обновляем их прямо здесь…

    if (justExploded)
    {
        m_pAnimationComponent->PlayAnimation("explode");
    }

    if (isVisible)
    {
        m_pCollisionComponent->Activate();
        m_pRenderingComponent->Show();
    }
    else
    {
        m_pCollisionComponent->Deactivate();
        m_pRenderingComponent->Hide();
    }

    // и т. д.
}
```

В итоге игровой цикл будет иметь примерно такой вид:

```
while (true)
{
    PollJoypad();

    float dt = g_gameClock.CalculateDeltaTime();

    for (each gameObject)
    {
        gameObject.Update(dt);
    }

    g_animationEngine.Update(dt);
    g_physicsEngine.Simulate(dt);
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_audioEngine.Update(dt);
    g_renderingEngine.RenderFrameAndSwapBuffers();
}
```

Пакетное обновление имеет много преимуществ с точки зрения производительности, среди которых можно выделить такие.

- *Максимальная когерентность кэша*. Пакетное обновление позволяет подсистеме движка достичь максимальной когерентности кэша, поскольку все данные, относящиеся к отдельным объектам, хранятся внутри и могут быть размещены в едином непрерывном участке оперативной памяти.
- *Минимальное дублирование вычислений*. Глобальные вычисления можно выполнить один раз и затем использовать повторно, не дублируя их в каждом игровом объекте.
- *Снижение частоты повторного выделения ресурсов*. Подсистемам движка в ходе обновления часто приходится выделять память и/или другие ресурсы и управлять ими. Если обновление определенной подсистемы чередуется с обновлениями других подсистем движка, эти ресурсы необходимо освобождать и выделять заново для каждого обрабатываемого объекта в игре. Но если обновления выполняются вместе, ресурсы можно выделять лишь один раз для каждого кадра и затем задействовать их во всех объектах пакета.
- *Эффективная конвейеризация*. Многие подсистемы движка выполняют практически идентичный набор вычислений для каждого отдельного объекта в игровом мире. При пакетном обновлении можно использовать принцип распределения/объединения, чтобы разделить крупные рабочие задания между несколькими ядрами процессора. Такого рода параллелизм невозможен при обработке изолированных объектов.

Выигрыш в производительности не единственный довод в пользу пакетного обновления. Некоторые подсистемы движка попросту не работают, если их обновлять из каждого объекта. Например, если мы пытаемся вычислить столкновения в системе из нескольких динамических твердых тел, приемлемое решение

невозможно получить, рассматривая каждый объект по отдельности. Взаимное проникновение этих объектов должно быть улажено групповым образом: либо с помощью итеративного подхода, либо путем решения линейной системы.

16.6.3. Взаимные зависимости объектов и подсистем

Даже если не принимать во внимание производительность, упрощенный подход с обновлением в каждом объекте не годится, если игровые объекты *зависят* друг от друга. Например, человеческий персонаж может держать в руках кошку. Чтобы вычислить позу скелета кошки в глобальной системе координат, сначала нужно произвести те же вычисления для человека. Из этого следует, что для корректной работы игры важен *порядок*, в котором обновляются объекты.

Похожая проблема возникает, если *подсистемы* движка зависят друг от друга. Например, симуляция физики тряпичной куклы должна обновляться совместно с движком анимации. Обычно система анимации генерирует промежуточную позу скелета в локальной системе координат. Эти суставные преобразования переводятся в глобальные координаты и применяются к системе связанных между собой твердых тел, которые приблизительно соответствуют скелету в рамках физической симуляции. Твердые тела симулируются физическим движком с течением времени, затем суставы скелета размещаются в своих итоговых позициях покоя. В конце движок анимации вычисляет позу в глобальной системе координат и палитру матриц скиннинга. И опять-таки, чтобы получить корректные результаты, обновления подсистем анимации и физики должны выполняться в определенном порядке. Такие взаимные зависимости между подсистемами — распространенное явление в игровых движках.

Поэтапные обновления

Чтобы учитывать взаимные зависимости между подсистемами, их обновления можно вручную закодировать в правильном порядке внутри игрового цикла. Например, чтобы учесть взаимодействие между движком анимации и физикой тряпичной куклы, мы можем написать что-то вроде следующего:

```
while (true) // главный цикл игры
{
    // ...

    g_animationEngine.CalculateIntermediatePoses(dt);
    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // также симулирует тряпичные куклы
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
    g_animationEngine.FinalizePoseAndMatrixPalette();

    // ...
}
```

Следует проявить осторожность, чтобы обновлять состояние игровых объектов на подходящем этапе игрового цикла. Обычно для этого недостаточно вызывать в каждом кадре из каждого объекта функцию `Update()`. Игровым объектам могут потребоваться промежуточные результаты вычислений, производимые различными подсистемами движка. Например, игровой объект может запросить воспроизведение анимации до того, как система анимации начнет обновляться. Однако тот же объект может захотеть процедурно поправить промежуточную позу, сгенерированную системой анимации, до того, как эта поза будет задействована системой физической симуляции тряпичной куклы, и/или до того, как будут сгенерированы итоговая поза и палитра матриц. Из этого следует, что объект необходимо обновить дважды: первый раз до того, как анимация вычислит свои промежуточные позы, второй раз — после.

Многие игровые движки позволяют своим объектам выполнять логику обновления по нескольку раз за кадр. Например, Naughty Dog (движок, на котором основаны игры из циклов *Uncharted* и *The Last of Us*) обновляет игровые объекты три раза: перед слиянием анимации, после слияния, но перед генерацией итоговой позы, и после ее генерации. Этого можно достичь, предусмотрев в каждом классе игровых объектов три виртуальные функции, которые играют роль хуков. При использовании такого подхода игровой цикл выглядит примерно так:

```
while (true) // главный цикл игры
{
    // ...

    for (each gameObject)
    {
        gameObject.PreAnimUpdate(dt);
    }

    g_animationEngine.CalculateIntermediatePoses(dt);

    for (each gameObject)
    {
        gameObject.PostAnimUpdate(dt);
    }

    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // также симулирует тряпичные куклы
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
    g_animationEngine.FinalizePoseAndMatrixPalette();

    for (each gameObject)
    {
        gameObject.FinalUpdate(dt);
    }

    // ...
}
```

Мы можем обеспечить игровым объектам столько этапов обновления, сколько посчитаем нужным. Но здесь следует быть осторожными, поскольку перебор игровых объектов и вызов виртуальной функции для каждого из них могут оказаться накладными. Кроме того, некоторым объектам не нужны все этапы обновления, поэтому их перебор будет пустой тратой ресурсов процессора.

На самом деле приведенный ранее пример не совсем реалистичный. Перебор всех объектов игры напрямую с целью вызова хуков `PreAnimUpdate()`, `PostAnimUpdate()` и `FinalUpdate()` был бы крайне неэффективным, поскольку лишь небольшая их часть требует выполнения какой-либо логики в каждом хуке. Этот подход также нельзя назвать гибким, так как он поддерживает только игровые объекты: если после завершения анимации нам потребуется обновить систему частиц, мы не сможем этого сделать. И наконец, это создает лишнюю *связанность* между низкоуровневыми подсистемами движка и игровыми объектами.

Куда лучшим выбором с точки зрения проектирования был бы универсальный механизм обратных вызовов. В такой архитектуре система анимации предоставляет средства, с помощью которых любой клиентский код (игровые объекты или другие подсистемы движка) может *зарегистрировать функцию обратного вызова* для каждого из трех этапов обновления (до анимации, после анимации и заключительного). Система анимации будет перебирать все зарегистрированные функции и вызывать их, не имея представления об игровых объектах *как таковых*. Этот подход максимально увеличивает производительность, поскольку только те клиенты, которым действительно *нужно* обновиться, регистрируют функции обратного вызова, выполняющиеся для каждого кадра. Также это максимально повышает гибкость и устраняет лишнее связывание системы игровых объектов и других подсистем движка, так как обратные вызовы может регистрировать любой клиент, а не только игровой объект.

Обновления на основе бакетов

При наличии *межобъектных* зависимостей поэтапное обновление, описанное ранее, необходимо слегка модифицировать. Дело в том, что зависимости между объектами могут вызывать создание противоречивых правил, определяющих порядок обновления. Представьте, к примеру, что объект Б содержится в объекте А. Также предположим, что объект Б можно обновлять только после того, как объект А был *полностью* обновлен, включая вычисление его итоговой позы в глобальной системе координат и палитры матриц. Это помешает нам выполнить пакетное обновление анимации для всех игровых объектов вместе, чтобы добиться от системы анимации максимальной производительности.

Межобъектные зависимости можно представить в виде леса, состоящего из деревьев зависимостей. Игровые объекты, не имеющие родителей (не зависящие от других объектов), являются корневыми. Объект, зависящий непосредственно от одного из таких корневых объектов, находится на первом уровне потомков в одном из деревьев леса. Объект, зависящий от потомка первого уровня, становится потомком второго уровня и т. д. (рис. 16.14).

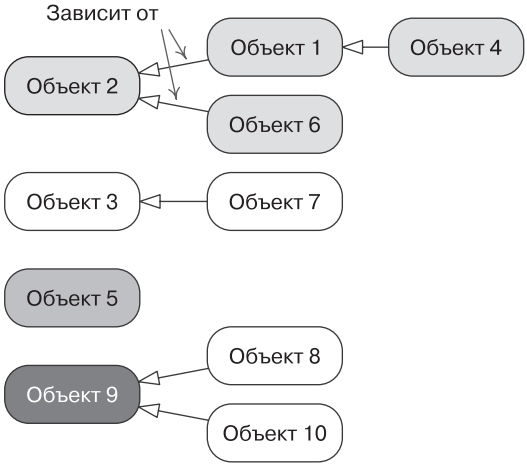


Рис. 16.14. Обновление зависящих друг от друга объектов можно представить в виде леса, состоящего из деревьев зависимостей

Чтобы решить проблему с конфликтующим порядком обновлений, можно разделить объекты на независимые группы, которые мы будем называть *бакетами* ввиду отсутствия более подходящего термина. Первый бакет состоит из всех корневых объектов в лесу, второй бакет — из всех потомков первого уровня, третий бакет — из всех потомков второго уровня и т. д. Для каждого бакета выполняется полное обновление игровых объектов и подсистем движка, включая все стадии. Таким образом мы проходимся по всем бакетам, от начала до конца.

Теоретически глубина деревьев в лесу зависимостей является неограниченной. Но на практике она оказывается довольно небольшой. Например, у нас могут быть персонажи с оружием, которые могут находиться на движущейся платформе или в автомобиле. Для этого в лесу зависимостей понадобятся лишь три уровня и, следовательно, только три бакета: один для платформ/автомобилей, другой — для персонажей и еще один — для оружия, которое они держат в руках. Многие игровые движки жестко ограничивают глубину леса зависимостей, чтобы использовать фиксированное количество бакетов (при условии, что они вообще применяют систему бакетов — игровой цикл можно спроектировать множеством других способов).

Вот как может выглядеть поэтапный пакетный цикл обновления на основе бакетов:

```
enum Bucket
{
    kBucketVehiclesPlatforms,
    kBucketCharacters,
    kBucketAttachedObjects,
    kBucketCount
};

void UpdateBucket(Bucket bucket)
```

```
{
    // ...

    for (each gameObject in bucket)
    {
        gameObject.PreAnimUpdate(dt);
    }

    g_animationEngine.CalculateIntermediatePoses
     (bucket, dt);

    for (each gameObject in bucket)
    {
        gameObject.PostAnimUpdate(dt);
    }

    g_ragdollSystem.ApplySkeletonsToRagDolls(bucket);
    g_physicsEngine.Simulate(bucket, dt); // тряпичные куклы и т. д.
    g_collisionEngine.DetectAndResolveCollisions
     (bucket, dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons(bucket);
    g_animationEngine.FinalizePoseAndMatrixPalette
     (bucket);

    for (each gameObject in bucket)
    {
        gameObject.FinalUpdate(dt);
    }
    // ...
}

void RunGameLoop()
{
    while (true)
    {
        // ...

        UpdateBucket(kBucketVehiclesAndPlatforms);
        UpdateBucket(kBucketCharacters);
        UpdateBucket(kBucketAttachedObjects);
        // ...

        g_renderingEngine.RenderSceneAndSwapBuffers();
    }
}
```

В реальности все может быть немного сложнее. Например, некоторые подсистемы, такие как физический движок, могут не поддерживать концепцию бакетов (возможно, они взяты из стороннего SDK или обновлять их с помощью бакетов непрактично). Но мы использовали ее в Naughty Dog для реализации всех игр в циклах *Uncharted* и *The Last of Us*. Поэтому данный подход доказал свои практичность и эффективность.

Несогласованное состояние объектов и запаздывание одного кадра

Давайте еще раз вернемся к обновлению объектов, но теперь будем учитывать локальное представление времени в каждом из них. В разделе 16.6 говорилось, что состояние игрового объекта i в момент времени t можно представить в виде вектора состояния $\mathbf{S}_i(t)$. При обновлении объекта мы превращаем вектор его предыдущего состояния $\mathbf{S}_i(t_1)$ в вектор нового состояния $\mathbf{S}_i(t_2)$, где $t_2 = t_1 + \Delta t$.

Теоретически переход состояния всех игровых объектов из t_1 в t_2 происходит мгновенно и параллельно (рис. 16.15). Но если предположить, что игровой цикл однопоточный, мы на самом деле обновляем объекты один за другим: циклически их перебираем и по очереди вызываем для каждого какого-то рода функцию обновления. Если остановить программу посреди этого цикла, состояние половины игровых объектов будет обновлено до $\mathbf{S}_i(t_2)$, а другая половина по-прежнему будет в состоянии $\mathbf{S}_i(t_1)$. Таким образом, если в ходе цикла обновления запросить текущее время у двух игровых объектов, они могут вернуть разные результаты! Более того, все объекты могут находиться в частично обновленном состоянии в зависимости от того, в какой именно момент был прерван цикл. Например, слияние анимации поз может уже закончиться, а результаты физической симуляции и столкновений могут быть еще не применены. Это подводит нас к следующему правилу: «Состояние всех игровых объектов является согласованным *до* и *после* цикла обновления, но *в самом цикле* согласованность не гарантируется».

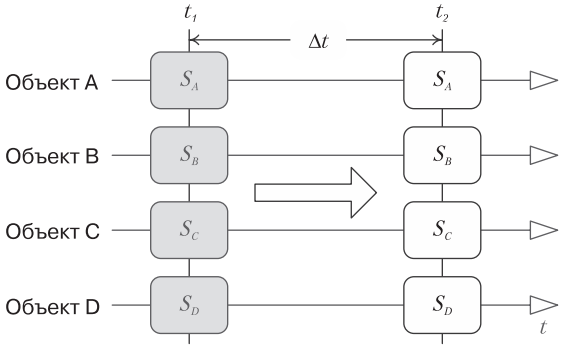


Рис. 16.15. Теоретически состояние всех игровых объектов обновляется мгновенно и параллельно на каждой итерации игрового цикла

Это проиллюстрировано на рис. 16.16.

Несогласованность в состоянии игровых объектов во время цикла обновления становится источником большой путаницы и ошибок, которые допускают даже профессионалы в игровой индустрии. Проблемы чаще всего возникают, когда в цикле обновления игровые объекты запрашивают друг у друга информацию о состоянии (это означает, что они связаны между собой). Например, если объект Б запрашивает скорость и направление объекта А, чтобы определить собственные

скорость и направление в момент t, программист должен четко понимать, какое именно состояние он хочет прочитать — *предыдущее*, $\mathbf{S}A(t_1)$, или новое, $\mathbf{S}A(t_2)$. Если вам нужно новое состояние, но объект A еще не обновился, вы сталкиваетесь с проблемой порядка обновления, которая может вызвать ошибки из категории, известной как *запаздывание одного кадра*. Такого рода ошибки возникают в ситуациях, когда состояние одного объекта отстает на один кадр от состояния других. На экране это проявляется в виде рассинхронизации игровых элементов.

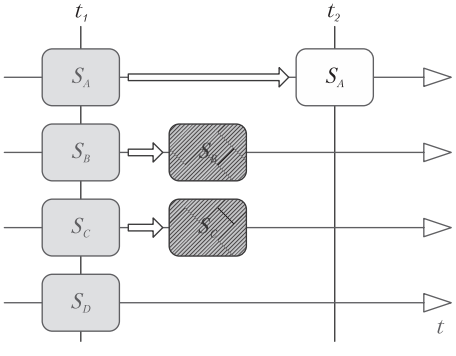


Рис. 16.16. На практике состояние игровых объектов обновляется последовательно. Это означает, что в какой-то произвольный момент работы цикла обновления одни объекты будут считать текущим временем t_2, а другие — t_1. Некоторые объекты могут быть обновлены лишь частично, поэтому их состояние будет несогласованным на внутреннем уровне. В сущности, состояние таких объектов находится где-то между t_2 и t_1

Кэширование состояния объектов

Как говорилось ранее, одно из решений этой проблемы заключается в том, чтобы распределить игровые объекты по бакетам. Но этот подход неидеален: он накладывает довольно нелогичные ограничения на то, как объекты могут запрашивать информацию о состоянии друг у друга. Если игровому объекту A нужен вектор *обновленного* состояния $\mathbf{S}_Б(t_2)$ объекта Б, то последний должен находиться в *ранее обновленном* бакете. Точно так же если объекту A нужен вектор *предыдущего* состояния $\mathbf{S}_Б(t_1)$ объекта Б, то последний должен находиться в *еще не обновленном* бакете. Объект A никогда не должен запрашивать вектор состояния у объекта из собственного бакета, поскольку согласно приведенному ранее правилу такие векторы могут быть обновлены лишь частично. В лучшем случае вы не будете уверены в том, в какой момент времени вы обращаетесь к состоянию другого объекта, t_1 или t_2.

Для улучшения согласованности можно сделать так, чтобы каждый игровой объект *кэшировал* вектор своего предыдущего состояния $\mathbf{S}_i(t_1)$ в ходе вычисления вектора нового состояния $\mathbf{S}_i(t_2)$, а не *перезаписывал* его во время обновления. Это решение имеет два очевидных преимущества. Во-первых, любой объект может безопасно запрашивать вектор предыдущего состояния любого другого объекта не-

зависимо от того, в каком порядке они обновляются. Во-вторых, вектор полностью согласованного состояния $\mathbf{S}_i(t_1)$ будет доступен всегда, даже во время обновления вектора нового состояния. Насколько мне известно, у этой методики нет стандартного названия, поэтому я буду называть ее *кэшированием состояния*.

Еще одной положительной стороной кэширования является то, что с помощью интерполяции можно вычислить состояние объекта в любой момент времени между t_2 и t_1. Именно с этой целью физический движок Havok хранит предыдущее и текущее состояния каждого твердого тела в симуляции.

Отрицательной стороной кэширования состояния является удвоенное потребление памяти по сравнению с перезаписывающим обновлением. К тому же оно решает только часть проблемы, потому что, в отличие от полностью согласованного состояния в момент t_1, новое состояние в момент t_2 все еще может страдать от несогласованности. Тем не менее этот подход может принести пользу, если применять его разумно. Он во многом вдохновлен архитектурными принципами чистого *функционального программирования* (см. подраздел 16.9.2). В чистых функциональных языках все операции выполняются функциями с четко определенными вводом и выводом и без побочных эффектов. Все данные считаются постоянными и неизменяемыми: вместо того чтобы изменять входящее значение, мы всегда создаем совершенно новые данные.

Использование временных меток

Простой и незатратный способ улучшения согласованности состояния игровых объектов заключается в применении временных меток. Это позволяет легко определить, к какому моменту времени относится вектор состояния объекта — предыдущему или текущему. Любой код, запрашивающий состояние другого игрового объекта внутри цикла обновления, может самостоятельно проверить временную метку, чтобы убедиться в получении подходящей информации.

Использование временных меток не помогает справиться с несогласованностью состояния во время обновления бакета. Но мы можем установить глобальную или статическую переменную, чтобы следить за тем, какой бакет обновляется в настоящий момент. Предполагается, что каждый игровой объект знает, в каком бакете он находится. Поэтому мы можем сравнить бакет объекта, к которому обращаемся, с бакетом, обновляемым в данный момент, и утверждать, что они не совпадают, чтобы защититься от запросов несогласованного состояния.

16.7. Применение конкурентности к обновлению игровых объектов

В главе 4 мы исследовали *аппаратный параллелизм* и то, как воспользоваться оборудованием, «заточенным» под параллельные вычисления, которое стало нормой в последних игровых консолях. Для этого используются методики *конкурентного программирования*. В разделе 8.6 мы познакомились с рядом подходов, которые

позволяют игровому движку пользоваться преимуществами параллельной обработки. Теперь пришло время поговорить о том, как конкурентность и параллелизм можно применить к обновлению состояния игровых объектов.

16.7.1. Конкурентные подсистемы движка

Больше всего от параллельной обработки, несомненно, выиграют те части движка, которые предъявляют самые высокие требования к производительности: отрисовка, анимация, звук и физика. Поэтому независимо от того, как обновляется объектная модель — в одном потоке или на нескольких ядрах, она должна уметь *взаимодействовать* с низкоуровневыми подсистемами движка, которые практически наверняка являются многопоточными.

Если движок поддерживает *систему заданий* общего назначения (см. подраздел 8.6.4), мы можем использовать ее для того, чтобы сделать выполнение подсистем конкурентным. В этом случае каждое обновление подсистемы может быть инициировано отдельным заданием в каждом кадре. Но будет лучше, если каждая подсистема будет создавать в каждом кадре сразу несколько заданий. Например, система анимации может инициировать по одному заданию для каждого объекта игрового мира, которому требуется слияние анимации. Позже в том же кадре, когда система анимации вычисляет матрицу мира и скиннинг, она может воспользоваться методикой распределения/объединения, чтобы распределить эту работу между доступными ядрами.

При конкурентном обновлении низкоуровневых подсистем движка следует убедиться в том, что их *интерфейсы* являются *потокобезопасными*. Мы хотим быть уверены в том, что никакой внешний код не попадет в состояние гонки, конкурируя с внешними клиентами или внутренними механизмами самой подсистемы. Для этого во всех внешних вызовах каждой подсистемы обычно применяются блокировки.

Если система задействует *потоки пользовательского уровня* (корутины или фиберы), нужно использовать циклические блокировки, чтобы сделать ее типобезопасной. Но если она обновляется в системных потоках, для этой цели подойдут и мьютексы.

Если определенная подсистема движка гарантированно работает только во время конкретного этапа игрового цикла, мы можем применить утверждение *lock-not-needed* (блокировка не требуется), не блокируя при этом ее участки. Подробнее об этих полезных утверждениях можно почитать в подразделе 4.9.7.

Конечно, остается еще один вариант: мы можем попытаться задействовать *неблокируемые* структуры данных для реализации критически важных (разделяемых) ресурсов подсистемы. Подобные структуры сложно создать, и у некоторых типов до сих пор нет неблокируемой версии. Таким образом, если вы выбрали этот подход, лучше применять его только к подсистемам с самыми строгими требованиями к производительности.

16.7.2. Асинхронный подход к проектированию

При взаимодействии с конкурентными подсистемами движка (например, в ходе обновления игровых объектов) необходимо мыслить *асинхронно*. Поэтому во время выполнения ресурсоемких операций следует избегать вызова *блокирующих функций* — таких, которые выполняют свою работу непосредственно в контексте вызывающего потока, блокируя его или само задание до тех пор, пока эта работа не будет закончена. Вместо этого крупные или затратные задания при любой возможности следует запрашивать с помощью *неблокирующих функций*, которые передают эти запросы на выполнение другим потокам, ядрам или процессорам и затем немедленно возвращают контроль вызывающему коду. Вызывающий поток или задание могут продолжать выполнение других задач (например, обновлять другие игровые объекты), пока не придут результаты запроса. Позже в том же или следующем кадре эти результаты можно будет принять и что-то с ними сделать.

Например, игра может запросить отбрасывание луча в игровой мир, чтобы определить, находится ли вражеский персонаж в области видимости игрока. В синхронной архитектуре луч отбрасывается сразу в ответ на запрос и функция, которая за это отвечает, немедленно возвращает результат, как показано далее:

```
SomeGameObject::Update()
{
    // ...

    // Отбрасываем луч, чтобы понять, находится
    // ли враг в поле зрения игрока.
    RayCastResult r = castRay(playerPos, enemyPos);

    // Теперь обрабатываем результаты…
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Игрок может видеть врага.
        // ...
    }

    // ...
}
```

В асинхронной архитектуре для отбрасывания луча вызывается функция, которая просто подготавливает и ставит в очередь соответствующее задание, после чего немедленно возвращается. Вызывающий поток или задание может продолжать работу, пока запрос обрабатывается другим процессором или ядром. Позже, после завершения задания по отбрасыванию луча, вызывающий поток или задание могут принять результаты запроса и обработать их:

```
SomeGameObject::Update()
{
    // ...
```

```
    // Отбрасываем луч, чтобы понять, находится ли враг в поле зрения игрока.
    RayCastResult r;
    requestRayCast(playerPos, enemyPos, &r);
    // Занимаемся другими задачами, пока другой
    // процессор или ядро выполняет для нас рейкастинг.

    // ...

    // Хорошо, мы больше не можем сделать ничего полезного.
    // Ждем результатов выполнения задания по отбрасыванию луча.
    // Если задание завершится, эта функция немедленно вернет результат.
    // В противном случае главный поток будет бездействовать,
    // пока не получит результаты...
    waitForRayCastResults(&r);

    // Обрабатываем результаты...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Игрок может видеть врага.
        // ...
    }

    // ...
}
```

Часто асинхронный код инициирует запрос в одном кадре, а получает результаты в следующем. В таких случаях можно встретить код следующего вида:

```
RayCastResult r;
bool rayJobPending = false;

SomeGameObject::Update()
{
    // ...

    // Ждем результатов выполнения задания по отбрасыванию луча
    // из последнего кадра.
    if (rayJobPending)
    {
        waitForRayCastResults(&r);

        // Обрабатываем результаты...
        if (r.hitSomething() && isEnemy(r.getHitObject()))
        {
            // Игрок может видеть врага.
            // ...
        }
    }
    // Отбрасываем новый луч для следующего кадра.
    rayJobPending = true;
    requestRayCast(playerPos, enemyPos, &r);

    // Занимаемся другими делами...
    // ...
}
```

Когда следует выполнять асинхронные запросы

Особенно каверзным аспектом внедрения асинхронного, пакетного подхода в синхронный, пошаговый код является определение того, *когда* в игровом цикле следует инициировать запрос, ждать результатов и использовать их. Чтобы облегчить себе эту задачу, попытайтесь ответить на следующие вопросы.

- *Как рано мы можем инициировать запрос?* Чем раньше мы выполним запрос, тем больше вероятность, что результаты появятся тогда, когда они нам все еще нужны. Это делает эффективность применения процессора максимальной, так как главный поток не будет простаивать в ожидании завершения асинхронного запроса. Поэтому в ходе обработки кадра нужно определить самый ранний момент, когда нам доступна вся информация, необходимая для выполнения запроса, и тут же его выполнить.
- *Как долго мы можем ждать, прежде чем нам потребуются результаты этого запроса?* Возможно, вторую часть операции можно выполнить на позднем этапе цикла обновления. Вероятно, мы можем допустить опоздание на один кадр и обновить текущее состояние объекта с помощью результатов из предыдущего кадра (некоторые подсистемы вроде ИИ допускают и более продолжительные задержки, так как они обновляются лишь раз в несколько секунд). Во многих случаях код, использующий результаты запроса, можно отложить до поздних этапов обработки кадра. Это потребует от вас немного внимания, рефакторинга и, возможно, дополнительного кэширования промежуточных данных.

16.7.3. Зависимости заданий и степень параллелизма

Чтобы можно было наилучшим образом воспользоваться платформой для параллельных вычислений, все ядра постоянно должны быть заняты. Если параллелизация движка основана на системе заданий, каждая итерация игрового цикла будет состоять из сотен или даже тысяч заданий, выполняющихся параллельно. Однако *зависимости* между ними могут сделать использование доступных ядер неоптимальным. Если задание Б принимает на вход данные, сгенерированные заданием А, это означает, что задание Б не сможет стартовать, пока не завершится А. Таким образом, между заданиями Б и А возникает *зависимость*.

Степень параллелизма (degree of parallelism, DOP) системы, известная также как *степень конкурентности* (degree of concurrency, DOC), измеряет максимально возможное (в теории) количество заданий, которые могут выполняться параллельно в тот или иной момент. Степень параллелизма группы заданий можно определить, нарисовав *граф зависимостей*. В нем задания выступают узлами дерева, а отношения «родитель — потомок» представляют собой зависимости. Количество листьев дерева говорит о степени параллелизма конкретной коллекции заданий. На рис. 16.17 показаны деревья зависимостей между заданиями и их степени параллелизма.

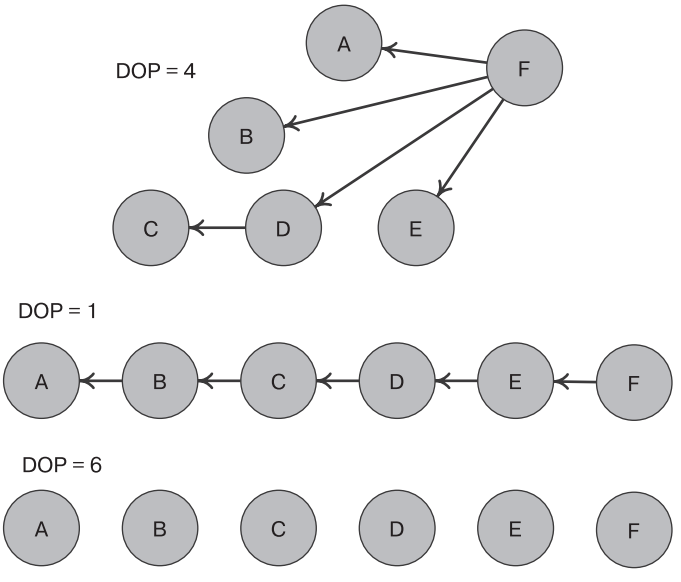


Рис. 16.17. Три дерева зависимостей между заданиями. Узлы дерева представляют задания, а стрелки обозначают зависимости между ними. Количество листьев в таком дереве указывает на степень параллелизма системы

Чтобы максимально эффективно использовать ресурсы процессора в компьютере с поддержкой параллелизма, DOP системы должна совпадать с числом доступных ядер или превышать его. В первом случае мы получим максимальную пропускную способность, а во втором производительность снизится, так как некоторые задания должны выполняться последовательно, но при этом ни одно из ядер не простаивает. Но если DOP системы меньше количества ядер, то некоторым ядрам будет нечем заняться.

Каждый раз, когда задание вынуждено ждать, пока свою работу не завершат другие задания, от которых оно зависит, в системе возникает *точка синхронизации*. Каждая такая точка представляет собой вероятность пустой траты ценных ресурсов процессора, пока задание ожидает завершения выполнения своих зависимостей (рис. 16.18).

Чтобы максимально повысить эффективность использования оборудования, можно попытаться повысить DOP системы за счет уменьшения числа зависимостей между заданиями. Мы также можем попробовать найти какую-то другую работу, которую нужно выполнять в периоды простоя. Чтобы снизить или устранить влияние точки синхронизации, ее можно *отсрочить*. Представьте, к примеру, что задание Г не может начать работу, пока не завершатся задания А, Б и В. Если попытаться запланировать выполнение задания Г перед завершением А, Б и В, ему, очевидно, придется какое-то время бездействовать. Но если отсрочить задание Г так, чтобы оно инициировалось намного позже завершения А, Б и В, можно быть уверенными в том, что оно будет простаивать (рис. 16.19). В своей презентации

Diving Down the Concurrency Rabbit Hole (http://cellperformance.beyond3d.com/articles/public/concurrency_rabit_hole.pdf) Майк Актон утверждает, что «секретом оптимизированной конкурентной архитектуры является *задержка*». Вот что он имеет в виду: отсрочив точки синхронизации, мы можем уменьшить или устранить время простоя в конкурентной системе.

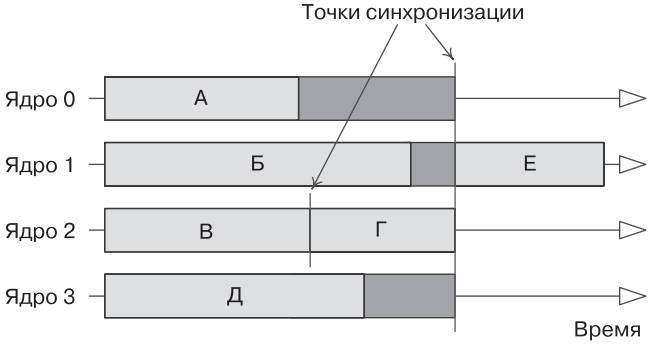


Рис. 16.18. Точка синхронизации возникает всякий раз, когда одно задание зависит от завершения одного или нескольких других заданий. Здесь задание Г зависит от В, а задание Е — от А, Б, Г и Д

Рис. 16.19. Задание Г зависит от заданий А, Б и В. *Вверху*: если попытаться выполнить Г на ядре 2 сразу после В, ядро будет бездействовать, пока не завершится Б. *Внизу*: если отсрочить начало работы Г до тех пор, пока не завершатся А, Б и В, ядро 2 можно освободить для выполнения других заданий и тем самым устранить время простоя, вызванное точкой синхронизации

16.7.4. Распараллеливание самой объектной модели игры

Объектные модели в играх известны тем, что их сложно распараллеливать, и тому есть несколько причин. Игровые объекты обычно очень сильно связаны между собой и с многочисленными подсистемами движка. Межобъектные зависимости возникают из-за того, что игровые объекты постоянно общаются между собой и запрашивают во время своего обновления состояние друг друга. Эти взаимодействия обычно происходят по нескольку раз в ходе цикла обновления и могут быть непредсказуемыми и крайне чувствительными к вводу со стороны игрока и событиям, происходящим в игровом мире. Это усложняет процесс *конкурентного* обновления объектов с использованием нескольких потоков или ядер процессора.

Тем не менее конкурентное обновление объектной модели игры, несомненно, выполнимая задача. Например, в Naughty Dog конкурентная объектная модель была реализована во время работы над *The Last of Us: Remastered*, когда мы переносили движок с PS3 на PS4. В этом подразделе мы исследуем некоторые проблемы, с которыми вы можете столкнуться при конкурентном обновлении игровых объектов, а также несколько решений, применяемых в нашем движке. Конечно, эти методики — лишь один из возможных подходов к проблеме, другие движки могут решать ее по-своему. Кто знает, может быть, *вы* придумаете совершенно иной способ борьбы с досадными проблемами, присущими конкурентному обновлению объектной модели игры!

Обновление игровых объектов с помощью заданий

Если у игрового движка есть система заданий, она, скорее всего, будет применяться для распараллеливания различных низкоуровневых подсистем, таких как анимация, звук, столкновения/физика, отрисовка, файловый ввод/вывод и т. д. Так почему бы таким же образом не распараллелить обновления игровых объектов, превратив их в задания? Этот подход используется в движке Naughty Dog. Он работает, однако его корректная и эффективная реализация является нетривиальной задачей.

В подразделе 16.6.3 упоминалось о том, что из-за наличия взаимных зависимостей между игровыми объектами нужно контролировать порядок их обновления. Для этого все объекты в игре можно распределить по N бакетам таким образом, чтобы объекты в бакете Б зависели только от объектов в бакетах от 0 до Б – 1. Если выбрать этот путь, все игровые объекты в каждом бакете можно будет обновлять в виде заданий, которые соответствующая система станет распределять между доступными ядрами (вперемешку с другими заданиями, которые волей случая выполняются в то же время). Примерно такой подход применяется в движке Naughty Dog:

```
void UpdateBucket(int iBucket)
{
    job::Declaration aDecl[kMaxObjectsPerBucket];
    const int count = GetNumObjectsInBucket(iBucket);
```

```
    for (int jObject = 0; jObject < count; ++jObject)
    {
        job::Declaration& decl = aDecl[jObject];
        decl.m_pEntryPoint = UpdateGameObjectJob;
        decl.m_param = reinterpret_cast<uintptr_t>(
            GetGameObjectInBucket(iBucket, jObject));
    }
    job::KickJobsAndWait(aDecl, count);
}
```

Другой подход к работе с межобъектными зависимостями состоит в том, чтобы сделать их явными. В этом случае мы бы, вероятно, имели какой-то механизм для объявления зависимости между, скажем, объектом А и объектами Б, В и Г. Такие зависимости сформировали бы простой *направленный граф*. Чтобы начать обновление, мы бы инициировали задания для всех игровых объектов, у которых нет зависимостей (представленных узлами графа зависимостей без исходящих ребер). По завершении задания мы бы переходили к каждому зависимому игровому объекту и ждали, пока не завершится обновление всех других объектов, которые от него зависят. Этот процесс повторялся бы снова и снова, пока не обновился бы весь граф игровых объектов.

Если граф зависимостей между игровыми объектами содержит какие-либо *циклы*, это может стать проблемой. Циклические зависимости, состоящие из двух и более объектов, являются признаком того, что эти объекты нельзя обновить в том порядке, в котором они связаны между собой. Мы можем решить эту проблему двумя путями: либо полностью устранить циклы, изменив то, как объекты взаимодействуют между собой, для чего направленный граф нужно сделать *ациклическим* (directed acyclic graph, DAG), либо изолировать эти скопления циклически зависимых игровых объектов и обновлять каждое скопление последовательно на одном ядре.

Асинхронное обновление игровых объектов

В подразделе 16.7.1 упоминалось о том, что обновление игровых объектов обычно происходит асинхронно. Например, для рейкастинга вместо *блокирующей функции* делается асинхронный *запрос*, который обрабатывается системой столкновений на каком-то будущем этапе формирования кадра. Позже, в текущем или следующем кадре, мы принимаем результаты рейкастинга и как-то их используем.

Этот подход хорошо работает и в ситуации, когда конкурентное обновление (на нескольких ядрах) распространяется и на сами игровые объекты. Но если система заданий основана на *потоках пользовательского уровня* (корутинах или фиберах), *блокирующие вызовы* тоже являются подходящим вариантом. Это будет работать, так как корутины обладают уникальным свойством — они способны *уступить* выполнение другой корутине и через какое-то время, когда эта другая корутина также уступит выполнение, продолжить работу. В системе заданий, основанной на фиберах, такой как в движке Naughty Dog, задания не являются корутинами *как таковыми*, но они имеют точно такую же способность — могут уснуть посреди

выполнения, а позже их можно разбудить для продолжения работы с того места, на котором они остановились.

Вот пример рейкастинга, реализованного с помощью блокирующего вызова в системе, основанной на корутинах или фиберах:

```
SomeGameObject::Update()
{
    // ...

    // Отбрасываем луч, чтобы понять, находится ли
    // враг в поле зрения игрока.
    RayCastResult r = castRayAndWait(playerPos, enemyPos);

    // Спим...

    // Просыпаемся, когда готовы результаты рейкастинга!

    // Обрабатываем результаты...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Игрок может видеть врага...
        // ...
    }

    // ...
}
```

Обратите внимание на то, что эта реализация почти ничем не отличается от *плохого* примера, представленного в подразделе 16.7.2! Оказывается, благодаря потокам пользовательского уровня наше задание может-таки воспользоваться вызовом блокирующей функции! На рис. 16.20 показано, как это происходит. Задание, в сущности, было разделено на две части: одна выполняется до вызова блокирующей операции рейкастинга, а другая — после.

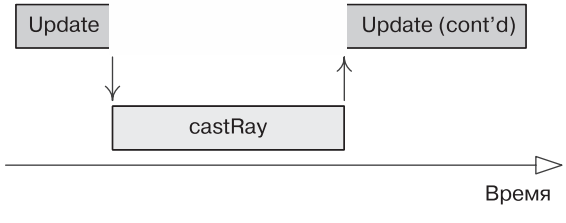


Рис. 16.20. Если наша система заданий реализована на основе потоков пользовательского уровня, задание можно прервать для выполнения асинхронной операции и затем возобновить его, когда она будет завершена

Благодаря этому механизму мы можем реализовать такие функции системы заданий, как `WaitForCounter()` и `KickJobsAndWait()`. Эти функции *блокируют* свои задания, делая их неактивными и позволяя работать другим заданиям, пока соответствующий счетчик (или счетчики) не достигнет нуля.

Блокирование во время обновления игровых объектов

Обновление с использованием бакетов помогает решить проблемы, возникающие из-за зависимостей между объектами. Оно гарантирует, что игровые объекты обновляются в правильном глобальном порядке (например, железнодорожный вагон обновляется раньше объектов, которые в нем находятся). Оно также может быть полезным для выполнения запросов между объектами. Объект в бакете Б может безопасно запросить состояние объектов в бакетах Б – Δ и Б + Δ, где $\Delta > 0$, так как в обоих случаях нам известно, что содержимое этих бакетов не обновляется одновременно с объектами из бакета Б. Однако проблема с запаздыванием кадра никуда не девается: если в кадре N объект из бакета Б обращается к объекту из бакета Б – Δ, он получит его состояние в том же кадре N, но, обратившись к объекту из бакета Б + Δ, он получит его состояние на момент, когда актуальным был последний кадр ($N - 1$), поскольку оно еще не обновилось.

Таким образом, если система обновления основана на бакетах, мы можем безопасно обращаться к объектам из *других* бакетов, не нуждаясь ни в каких блокировках. Но что, если взаимодействовать или выполнять запросы друг к другу должны объекты из *одного* бакета? В таком случае мы опять можем столкнуться с состоянием гонки, и это нельзя игнорировать, надеясь на лучшее.

У вас может появиться соблазн создания в объектной модели игры единой глобальной блокировки (циклической или мьютексной). Ее может установить каждый объект внутри определенного бакета (который, возможно, взаимодействует с другими объектами в ходе обновления), а после окончания работы снять. Это, несомненно, гарантирует отсутствие конкуренции за ресурсы между объектами, которые общаются друг с другом. Но вместе с тем дает крайне нежелательный эффект: все игровые объекты внутри одного бакета обновляются *последовательно*, что, в сущности, делает конкурентное обновление однопоточным! Дело в том, что такая блокировка сделала бы невозможным одновременное обновление любых двух игровых объектов, даже если они не взаимодействуют между собой.

К этой проблеме можно подойти совершенно по-разному. В ранних версиях Naughty Dog мы создали глобальную систему блокирования, но блокировка устанавливалась, только когда дескриптор игрового объекта *разыменовывался* внутри его функции обновления. В нашем движке обращение к игровым объектам происходит по дескрипторам, а не по стандартным указателям — это сделано для поддержки дефрагментации памяти. Поэтому, чтобы получить обычный указатель на объект, сначала необходимо разыменовать его дескриптор. Это дает прекрасную возможность обнаружить, что один игровой объект собирается взаимодействовать с другим. Устанавливая блокировки только в ситуациях, когда взаимодействие на самом деле может произойти, мы в определенной степени сумели восстановить конкурентность. Однако система блокирования была сложной, неудобной в работе и приводила к неэффективному использованию процессора во время обновления бакетов.

Снимки объектов

Если проанализировать взаимные зависимости между объектами в настоящем игровом движке, можно заметить, что подавляющее большинство их взаимодействий во время обновления заключаются в запрашивании состояния. Иными словами, игровой объект А запрашивает текущее состояние объектов Б, В и т. д. На самом деле ему нужна лишь *копия* этого состояния, доступная *только для чтения*. Ему не нужно взаимодействовать с самими объектами (которые теоретически в этот самый момент могут обновляться конкурентным образом).

Учитывая все это, было бы разумно сделать так, чтобы каждый объект предоставлял *снимок* своего текущего состояния — копию, доступную только для чтения, которую без блокировок и потенциальной конкуренции за ресурсы может получить любой другой игровой объект в системе. Снимки, в сущности, являются разновидностью методики *кэширования состояния*, описанной в подразделе 16.6.3. Мы в Naughty Dog использовали этот подход для игр *The Last of Us: Remastered*, *Uncharted 4: A Thief's End* и *Uncharted: The Lost Legacy*.

Вот как снимки работают в движке Naughty Dog: в начале каждого обновления бакета мы просим все игровые объекты обновить свои снимки. Эти обновления никогда не требуют обращения к другим объектам, поэтому их можно выполнять конкурентно и без блокировок. Когда все объекты обновились, мы запускаем задания для обновления их внутреннего состояния. Это тоже происходит конкурентным образом. Каждый раз, когда объекту из бакета Б нужно запросить состояние другого объекта, у него есть три варианта.

- При обращении к объекту в бакете Б – Δ он может запросить либо его непосредственное состояние, либо снимок.
- При обращении к объекту в бакете Б он запрашивает снимок.
- При обращении к объекту в бакете Б + Δ он опять может запросить либо его непосредственное состояние, либо снимок.

Обработка межобъектных изменений

Снимки позволяют избежать блокировок, когда объекты *читают* состояние друг друга. Но они никак не помогают против конкуренции за ресурсы, которая возникает, когда одному игровому объекту нужно *изменить* состояние другого, находящегося в том же бакете. Чтобы справиться с этой ситуацией, Naughty Dog применяет сочетание следующих эмпирических правил и методик.

- При любой возможности избегайте межобъектных изменений.
- Межобъектные изменения *между* бакетами безопасны, но...
- Межобъектные изменения *внутри* одного бакета следует выполнять осторожно:
  - с помощью блокировок;
  - помещая запрос на изменения в очередь, вместо того чтобы применять их немедленно. Очередь запросов защищена блокировкой, и обработка ее содержимого откладывается до тех пор, пока не завершится обновление бакета.

Межобъектные изменения в рамках одного бакета можно выполнить еще одним способом — инициировав в *следующем* бакете задание, которое должно их синхронизировать. Откладывая действие до следующего бакета, мы можем быть уверены в том, что объекты, с которыми имеем дело, завершили обновление и не будут обновляться одновременно с нашими операциями. Мы реализовали это в Naughty Dog для синхронизации движений в рукопашной схватке игрока и NPC.

Будущие улучшения

Система обновления на основе бакетов, описанная в этом разделе, далеко не идеальна. Она не настолько эффективна, как могла бы быть, так как каждый переход между бакетами создает в игровом цикле *точку синхронизации*. В такие моменты некоторые ядра процессора могут бездействовать в ожидании того, когда все игровые объекты в бакете закончат обновляться.

Создание снимков тоже не идеальное решение проблем с зависимостями между бакетами, поскольку оно рассчитано только на запросы для чтения, а для межобъектных изменений по-прежнему требуются блокировки. Обновление самих снимков тоже может быть затратным, хотя здесь можно применить простую оптимизацию: генерировать снимки для тех объектов только тогда, когда они нужны.

У этой проблемы много решений. Чтобы найти наиболее подходящий способ конкурентного обновления игровых объектов, лучше всего поэкспериментировать. Надеюсь, в этом разделе вы нашли полезные идеи, которые помогут продвинуться вперед, когда вы будете практиковаться самостоятельно!

16.8. События и передача сообщений

Игры по своей природе являются событийными. *Событие* — это то, что происходит во время игрового процесса и может нас заинтересовать. Взрыв, момент, когда враг увидел игрока, подбор аптечки — все это события. Игре обычно нужно как-то уведомить о событии заинтересованные игровые объекты и позволить им реагировать на интересующие их события различными способами — мы называем это *обработкой* событий. Разные типы игровых объектов по-разному реагируют на события. Это ключевой аспект их поведения, который не менее важен, чем постепенное изменение состояния при отсутствии внешнего ввода. Например, поведение мячика в игре *Pong* определяется его скоростью и направлением, реакцией на столкновение со стенкой или ракеткой и тем, что происходит, когда один из игроков по нему не попал.

16.8.1. Проблема со статически типизированным связыванием функций

Чтобы уведомить объект о произошедшем событии, можно просто вызвать его метод (функцию — член класса). Например, когда происходит взрыв, мы можем найти в игровом мире все объекты, находящиеся в его радиусе досягаемости,

и вызвать из каждого из них виртуальную функцию с именем вроде `OnExplosion()`. Это проиллюстрировано в следующем псевдокоде:

```
void Explosion::Update()
{
    // ...

    if (ExplosionJustWentOff())
    {

        GameObjectCollection damagedObjects;
        g_world.QueryObjectsInSphere(GetDamageSphere(),
                                     damagedObjects);
        for (each object in damagedObjects)
        {
            object.OnExplosion(*this);
        }
    }

    // ...
}
```

Вызов `OnExplosion()` является примером *статически типизированного позднего связывания функций*. Связывание функций — это процесс определения того, какую реализацию функции следует выполнить по определенному месту вызова. В итоге эта реализация привязывается к вызову. Считается, что виртуальные функции, такие как обработчик `OnExplosion()`, имеют *позднее связывание*. Это означает, что на этапе компиляции неизвестно, *какая* из множества ее потенциальных реализаций будет вызвана, — это прояснится только во время выполнения, когда мы будем знать тип целевого объекта и сможем выбрать подходящую реализацию. Вызов функции является *статически типизированным*, потому что компилятор *знает*, какую реализацию нужно вызывать для того или иного типа объектов. Например, ему известно, что для целевого объекта `Tank` подходит метод `Tank::OnExplosion()`, а для `Crate` — `Crate::OnExplosion()`.

Проблема статически типизированного связывания функций в том, что оно делает реализацию менее гибкой. Например, виртуальная функция `OnExplosion()` требует, чтобы все игровые объекты наследовались от общего базового класса. Более того, этот класс должен *объявить* виртуальную функцию `OnExplosion()`, даже если не все игровые объекты могут реагировать на взрывы. На самом деле, если мы решим обрабатывать события с помощью статически типизированных виртуальных функций, нам придется объявить в классе `GameObject` виртуальные функции для *всех возможных событий* в игре! Это затруднит добавление новых событий в систему. Мы не сможем создавать события в виде данных — например, внутри редактора игрового мира. К тому же отдельные типы или экземпляры объектов не смогут подписываться лишь на те события, которые их интересуют, — каждый объект в игре будет знать обо всех возможных событиях, даже если у него

для них не предусмотрено никакой реакции, то есть его функции-обработчики являются пустыми.

Что нам действительно нужно для обработки событий, так это *динамически типизированное позднее связывание функций*. Некоторые языки программирования имеют встроенную поддержку этой возможности (например, делегаты в C#). В других языках программистам приходится реализовывать ее вручную. К этой проблеме можно подойти по-разному, но чаще всего все сводится к модели на основе данных. Иными словами, понятие вызова функции инкапсулируется в виде объекта, который передается на этапе выполнения, что позволяет реализовать динамически типизированный вызов функции с поздним связыванием.

16.8.2. Инкапсуляция события в виде объекта

Событие, в сущности, состоит из двух компонентов: *типа* (взрыв, ранение друга, момент, когда игрока заметили, подбор аптечки и т. д.) и *аргументов*. Аргументы предоставляют подробности о событии. (Какой урон нанес взрыв? Какой именно друг был ранен? Где был замечен игрок? Сколько здоровья было в аптечке?) Эти два компонента можно инкапсулировать в виде объекта, как показано далее в довольно упрощенном фрагменте кода:

```
struct Event
{
    const U32 MAX_ARGS = 8;

    EventType  m_type;
    U32        m_numArgs;
    EventArg   m_aArgs[MAX_ARGS];
};
```

В некоторых игровых движках это называется не *событиями*, а *сообщениями* или *командами*. Эти названия подчеркивают тот факт, что информирование объекта о событии по своей сути эквивалентно передаче ему сообщения или команды.

На практике объекты событий обычно являются не такими простыми. Мы можем реализовать разные их типы, наследуя их, к примеру, от единого корневого класса. Аргументы могут быть реализованы в виде связного списка или динамически выделяемого массива с произвольным количеством элементов разных типов.

Инкапсуляция события или сообщения в виде объекта имеет множество преимуществ.

- *Единая функция обработки событий*. Поскольку типы событий закодированы внутри объектов, любое их количество может быть представлено экземпляром одного класса (или корневого класса в иерархии наследования). Это означает, что нам нужна лишь *одна* виртуальная функция для обработки событий *любых* типов (например, `virtual void OnEvent(Event& event);`).
- *Постоянное хранение*. В отличие от вызовов функций, чьи аргументы выходят из области видимости после того, как функция завершает работу, объект

события хранит свой тип и аргументы в виде данных. Благодаря этому он является постоянным. Можно поместить его в очередь для последующей обработки, транслировать его копию нескольким получателям и т. д.

- *Слепая маршрутизация событий.* Объект может передать полученное им событие другому объекту, и для этого ему не нужно ничего знать об этом событии. Например, если автомобиль получает событие «*Выходите*», он может передать его всем своим пассажирам, тем самым позволяя им покинуть салон, при этом самому автомобилю ничего не известно о том, что значит выходить.

Концепция инкапсуляции событий/сообщений/команд в виде объектов широко применяется во многих областях информатики. Ее можно встретить не только в игровых движках, но и в таких системах, как графические пользовательские интерфейсы, распределенные средства коммуникации, и многих других. В широко известной книге *Gang of Four* [19] это называется шаблоном проектирования «команда».

16.8.3. Типы событий

Существует множество способов различения типов событий. Один из простых подходов, который применяется в C и C++, состоит в определении глобального перечисления, которое привязывает каждый тип события к уникальному целому числу:

```
enum EventType
{
    EVENT_TYPE_LEVEL_STARTED,
    EVENT_TYPE_PLAYER_SPAWNED,
    EVENT_TYPE_ENEMY_SPOTTED,
    EVENT_TYPE_EXPLOSION,
    EVENT_TYPE_BULLET_HIT,
    // ...
}
```

Этот подход прост и эффективен, так как чтение, запись и сравнение целых чисел происходят чрезвычайно быстро. Но у него есть несколько проблем. Во-первых, информация обо *всех* типах событий в игре хранится централизованно, что можно рассматривать как разновидность нарушения инкапсуляции (мнения об этом разнятся, хорошо это или плохо, судите сами). Во-вторых, типы событий встроены в код, поэтому нельзя будет легко добавить их в виде данных. В-третьих, перечислители представляют собой обычные указатели, поэтому они должны размещаться в определенном порядке. Если кто-то случайно создаст новый тип события посреди списка, указатели всех последующих идентификаторов изменятся, что может вызвать проблемы, если эти идентификаторы хранятся в файлах с данными. Таким образом, систему типизации событий на основе перечисления можно использовать в небольших демонстрационных программах и прототипах, но она не очень хорошо масштабируется и совсем не годится для настоящих игр.

Типы событий можно кодировать в виде строк. Этот подход ничем не ограничен. Чтобы добавить в систему новый тип, достаточно придумать для него название. Но это влечет за собой множество проблем, включая высокую вероятность кон-

фликтов имен, банальные опечатки, увеличение расхода памяти для хранения строк и медленные операции сравнения целых чисел. Вместо обычных строк можно применить их хеши, это позволит устранить проблемы с производительностью и снизить потребление памяти, однако риск конфликтов имен и опечаток по-прежнему остается. Тем не менее многие игровые студии, включая Naughty Dog, считают, что чрезвычайная гибкость строк и строковых идентификаторов, а также возможность описывать события в виде данных стоят того, чтобы пойти на риск.

Некоторые проблемы, присущие идентификации событий с помощью строк, можно нивелировать за счет специальных инструментов. Например, мы можем хранить названия всех типов событий в центральной базе данных, для добавления новых типов можно предоставить пользовательский интерфейс. Конфликты имен могут обнаруживаться автоматически во время создания новых событий, и пользователю можно запретить дублирование типов. Для выбора событий можно предоставить раскрывающийся список, чтобы не заставлять пользователя запоминать их названия и вводить их вручную. База данных также может хранить метаинформацию о каждом типе событий, включая документацию о его назначении и правильном использовании, а также сведения о количестве и типах аргументов, которые оно поддерживает. Этот подход способен прекрасно работать, но не следует забывать о затратах, связанных с подготовкой такой системы и ее обслуживанием, поскольку их нельзя назвать незначительными.

16.8.4. Аргументы событий

Аргументы событий обычно ведут себя по аналогии с аргументами функций, предоставляя получателю информацию о событии, которая может оказаться полезной. Их можно реализовать всевозможными способами.

Для каждого уникального типа событий можно вывести отдельную разновидность класса `Event`. Аргументы могут быть встроены в код в виде свойств класса, например:

```
class ExplosionEvent : public Event
{
    Point m_center;
    float m_damage;
    float m_radius;
};
```

В качестве альтернативы аргументы событий можно представить в виде набора *вариантов*. Вариант — это объект данных, способный хранить сразу несколько типов данных. Обычно он содержит информацию о типе данных, который хранится в текущий момент, а также сами данные. В системе событий аргументами могут выступать целые и дробные числа, булевы значения и хешированные строки. Таким образом, класс варианта в C или C++ может выглядеть так:

```
struct Variant
{
    enum Type
```

```
    {
        TYPE_INTEGER,
        TYPE_FLOAT,
        TYPE_BOOL,
        TYPE_STRING_ID,
        TYPE_COUNT // количество уникальных типов
    };

    Type       m_type;

    union
    {
        I32    m_asInteger;
        F32    m_asFloat;
        bool   m_asBool;
        U32    m_asStringId;
    };
};
```

Набор вариантов внутри `Event` можно реализовать в виде массива с небольшим максимальным количеством элементов (4, 8 или 16). Так мы можем ограничить число аргументов, которые разрешается передавать вместе с событием. Это позволяет также обойти проблему динамического выделения памяти для полезных данных в каждом аргументе события, что можно считать значительным преимуществом, особенно в консольных играх с ограниченной памятью.

Набор вариантов можно реализовать в виде структуры данных с динамически изменяющимся размером, такой как массив переменной длины (например, `std::vector`) или связный список (например, `std::list`). Это существенно повышает гибкость по сравнению с использованием фиксированного размера, но влечет за собой расходы на динамическое выделение памяти. Здесь отлично подошел бы механизм выделения на основе пула, если считать, что все экземпляры `Variant` одинакового размера.

Аргументы событий в виде пар «ключ — значение»

Фундаментальной проблемой *индексированного* набора аргументов событий является зависимость от порядка. И отправитель, и получатель события должны знать, в каком именно порядке перечислены аргументы. Иначе это может вызвать неразбериху и возникновение ошибок. Например, вы можете случайно пропустить или добавить лишний обязательный аргумент.

Этой проблемы можно избежать, реализовав аргументы событий в виде пар «ключ — значение». Все аргументы однозначно определяются по своим ключам, поэтому они могут размещаться в любом порядке, а необязательные аргументы можно и вовсе пропускать. Набор аргументов можно реализовать в виде закрытой или открытой хеш-таблицы с ключами в качестве хешей, но это может быть и массив, связный список или двоичное дерево поиска, состоящее из пар «ключ —

значение». Эти идеи проиллюстрированы в табл. 16.1. Вариантов существует множество, выбор конкретной реализации не так уж и важен, главное — удовлетворить требования игры и сделать это эффективно.

Таблица 16.1. Аргументы объекта события можно реализовать в виде набора пар «ключ — значение». Ключи позволяют предотвратить проблему зависимости от порядка, так как однозначно идентифицируют каждый аргумент

Ключ	Тип	Значение
«событие»	stringid	«взрыв»
«радиус»	float	10,3
«урон»	Int	25
«граната»	bool	true

16.8.5. Обработчики событий

Когда игровой объект получает событие, сообщение или команду, он должен как-то отреагировать. Этот процесс называется *обработкой событий*, и его обычно реализуют в виде функции или фрагмента скриптового кода, известного как *обработчик событий* (позже мы еще вернемся к теме скриптования в играх).

Обработчик часто представляет собой одну виртуальную компилируемую или скриптовую функцию, способную обработать события любого типа, например `virtual void OnEvent(Event& event)`. В данном случае функция, как правило, содержит оператор ветвления вида `switch` или `if/else-if`, рассчитанный на различные типы событий, которые могут быть получены. Типичный обработчик событий может выглядеть так:

```
virtual void SomeObject::OnEvent(Event& event)
{
    switch (event.GetType())
    {
    case SID("EVENT_ATTACK"):
        RespondToAttack(event.GetAttackInfo());
        break;

    case SID("EVENT_HEALTH_PACK"):
        AddHealth(event.GetHealthPack().GetHealth());
        break;

    // ...

    default:
        // Нераспознанное событие.
        break;
    }
}
```

Как вариант, можно предусмотреть набор обработчиков, по одному для каждого типа событий (например, `OnThis()`, `OnThat()`...). Но, как уже отмечалось, создание множества разных обработчиков чревато проблемами.

MFC (Microsoft Foundation Classes) — набор инструментов для создания графических интерфейсов в Windows — славился своими *картами сообщений*. Этот механизм позволял привязать на этапе выполнения любое сообщение Windows к произвольной невиртуальной или виртуальной функции. Это избавило от необходимости объявлять обработчики для всех возможных событий Windows в едином корневом классе и в то же время позволило отказаться от крупных выражений `switch`, которые широко применялись в обработчиках сообщений Windows за пределами MFC. Однако такая система не стоит потраченных усилий, так как выражение `switch` отлично работает и является простым и понятным.

16.8.6. Распаковка аргументов событий

Представленный ранее пример не дает ответа на один важный вопрос: как извлечь данные из списка аргументов события типобезопасным способом? Например, `event.GetHealthPack()`, предположительно, возвращает игровой объект `HealthPack`, который, в свою очередь, как ожидается, предоставляет метод под названием `GetHealth()`. Подразумевается, что корневому классу `Event` (и, соответственно, любому другому типу аргументов событий в игре) известно об аптечках. Это, вероятно, не очень практичный подход. В настоящем движке могут существовать классы, наследованные от `Event`, которые предоставляют удобные API для доступа к данным, такие как `GetHealthPack()`. Или же обработчику событий придется вручную распаковывать данные и приводить их к подходящим типам. Второй подход выглядит не очень хорошо с точки зрения типобезопасности, но на практике это не такая уж большая проблема, потому что при распаковке аргументов мы всегда знаем тип события.

16.8.7. Цепочки обязанностей

Игровые объекты почти всегда так или иначе зависят друг от друга. Обычно они организованы в виде иерархии трансформации, что, к примеру, позволяет одному объекту опираться на другой или находиться в руке у персонажа. Кроме того, игровые объекты могут состоять из нескольких взаимодействующих между собой компонентов, имеющих топологию в виде звезды или слабосвязанного «облака». Спортивные игры могут вести список всех персонажей в команде. В целом взаимные отношения между игровыми объектами можно представить в виде одного или нескольких *графов* (не забывайте, что списки и деревья являются разновидностями графа). Несколько примеров с графами отношений показаны на рис. 16.21.

Часто в рамках подобных графов отношений имеет смысл предусмотреть возможность передачи событий от одного объекта к другому. Например, когда авто-

мобиль получает событие, будет удобно, если он сможет передать его всем своим пассажирам, а пассажиры смогут передать его объектам в своем багаже. Если событие получает многокомпонентный игровой объект, его, возможно, придется передать всем компонентам, чтобы каждый из них мог заняться его обработкой. Или когда событие пришло персонажу спортивной игры, у нас может возникнуть желание поделиться им со всеми членами его команды.

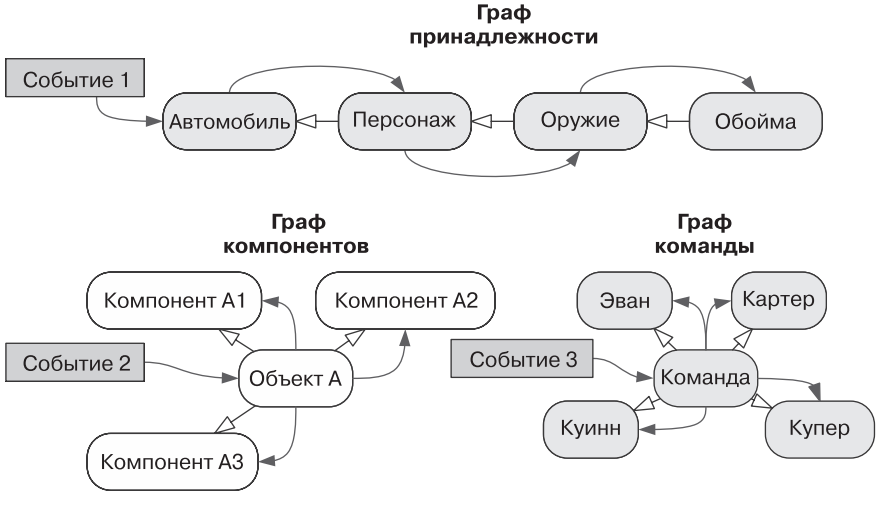


Рис. 16.21. Игровые объекты связаны между собой разными способами, и мы можем изобразить их отношения в виде графов. Любой такой граф может служить каналом распространения событий

Маршрутизация событий в графе объектов — распространенный шаблон проектирования в объектно-ориентированном, событийном программировании, который иногда называют *цепочкой обязанностей* [19]. Обычно порядок, в котором события передаются по системе, заранее определяется программистами. Событие передается первому объекту в цепочке, затем обработчик возвращает булево значение или числовой код, сигнализируя, распознал он это событие или нет. Если получатель взял на себя обработку события, процесс маршрутизации останавливается, в противном случае событие передается следующему получателю в цепочке. Обработчик, поддерживающий маршрутизацию событий в стиле цепочки обязанностей, может выглядеть так:

```
virtual bool SomeObject::OnEvent(Event& event)
{
    // Сначала вызываем обработчик базового класса.
    if (BaseClass::OnEvent(event))
    {
        return true;
    }
```

```
    // Теперь сами пытаемся обработать событие.
    switch (event.GetType())
    {
    case SID("EVENT_ATTACK"):
        RespondToAttack(event.GetAttackInfo());
        return false; // Событие можно передать другим обработчикам.

    case SID("EVENT_HEALTH_PACK"):
        AddHealth(event.GetHealthPack().GetHealth());
        return true; // Я обработал событие, не нужно его передавать.

    // ...

    default:
        return false; // Я не распознал это событие.
    }
}
```

Когда дочерний класс переопределяет обработчик событий, имеет смысл вызвать сначала реализацию базового класса (при условии, что потомок лишь дополняет реакцию родителя, а не заменяет ее). Иногда дочерний класс полностью переопределяет ответ базового класса, в таких случаях обработчик базового класса вызывать не следует. Это еще одна разновидность цепочки обязанностей.

Маршрутизация событий имеет и другие применения. Например, мы можем транслировать событие всем объектам в радиусе воздействия (скажем, во время взрыва). Чтобы это реализовать, можно воспользоваться механизмом запросов и найти все объекты внутри соответствующей сферы, а затем перенаправить им событие.

16.8.8. Подписка на события

Можно с уверенностью сказать, что большинству объектов в игре не нужно реагировать на все возможные события. Они в основном заинтересованы лишь в относительно небольшом наборе событий. Многоадресное вещание или трансляция событий могут отрицательно сказаться на производительности, так как нам нужно пройтись по группе объектов и вызвать из каждого из них обработчик, даже если объект не заинтересован в конкретном виде событий.

Чтобы сделать этот механизм более эффективным, можно разрешить игровым объектам *подписываться* на определенные виды событий. Например, мы можем вести список объектов, заинтересованных в отдельном типе событий, или каждый объект может содержать битовый массив, в котором каждый установленный бит сигнализирует о том, заинтересован ли он в событиях того или иного типа. Это позволит не вызывать незаинтересованные обработчики.

Что еще лучше, мы можем ограничить исходный запрос объектами, подписанными на событие, которое хотим транслировать. Например, во время взрыва можно запросить у системы столкновений все объекты в радиусе повреждений,

которые *к тому же* способны отреагировать на событие «*взрыв*». Это может сэкономить время в целом, поскольку мы избегаем взаимодействия с объектами, которые, как известно, не заинтересованы в транслируемом событии. Принесет ли этот подход в итоге какую-то выгоду, зависит от реализации механизма запросов и того, где быстрее фильтровать объекты: во время запроса или в ходе многоадресной передачи.

16.8.9. Использовать ли очередь

Большинство игровых движков позволяют обрабатывать события сразу после их отправки. В некоторых движках события также можно поместить в очередь для обработки в неопределенный момент в будущем. У этого подхода есть привлекательные достоинства, но при этом он существенно усложняет систему событий и создает ряд уникальных проблем. В следующих пунктах мы исследуем преимущества и недостатки очереди событий и заодно посмотрим, как такие системы реализуются.

Некоторые преимущества очереди событий

Далее рассматриваются некоторые положительные стороны использования очереди событий.

Выбор подходящего момента для обработки события. Мы уже видели, что к порядку обновления подсистем движка и игровых объектов следует подходить с осторожностью, чтобы обеспечить корректное поведение и максимизировать производительность на этапе выполнения. Определенные виды событий могут быть также очень чувствительны к тому, в какой именно момент игрового цикла они обрабатываются. Если обрабатывать все события сразу после их отправки, обработчики станут вызываться на непредсказуемых этапах игрового цикла и их будет сложно контролировать. Откладывая события с помощью очереди, программист может приступать к их обработке только в подходящий момент, когда это безопасно делать.

Возможность посылать события в будущее. Когда отправитель посылает событие, он обычно может указать время доставки. Например, мы можем сделать так, чтобы событие обрабатывалось чуть позже в том же или следующем кадре либо через несколько секунд после отправки. Это позволяет посылать события в будущее, что имеет ряд интересных применений. Мы можем реализовать простой таймер. Периодическое действие, такое как мигание света каждые 2 с, можно реализовать за счет события, чей обработчик выполняет периодическое задание и затем шлет в будущее новое событие того же типа с однократной задержкой.

Чтобы реализовать возможность посылать события в будущее, перед размещением в очереди им следует назначать время доставки. Когда согласно внутренним часам игры это время достигается или превышается, событие обрабатывается. Это можно легко реализовать, сортируя события в очереди по возрастанию времени

доставки. В каждом кадре мы берем первое событие в очереди и смотрим, когда оно должно быть доставлено. Если время доставки еще не наступило, мы немедленно прерываем обработку, так как нам известно, что все следующие события тоже находятся в будущем. Но если время доставки совпадает с текущим или уже прошло, мы извлекаем событие из очереди и обрабатываем его. Это продолжается до тех пор, пока мы не найдем событие, время доставки которого еще не наступило. Этот процесс проиллюстрирован в следующем псевдокоде:

```
// Эта функция вызывается как минимум один раз в каждом кадре.
// Ее задача – отправить все события, время доставки
// которых совпадает с текущим или уже прошло.

void EventQueue::DispatchEvents(F32 currentTime)
{
    // Проверяем следующее событие в очереди, но не удаляем его.
    Event* pEvent = PeekNextEvent();

    while (pEvent
        && pEvent->GetDeliveryTime() <= currentTime)
    {
        // Удаляем событие из очереди.
        RemoveNextEvent();

        // Передаем событие обработчику его получателя.
        pEvent->Dispatch();

        // Берем следующее событие в очереди (опять же не удаляя его).
        pEvent = PeekNextEvent();
    }
}
```

Приоритизация событий. Даже если события выстроены в очередь по времени доставки, порядок их обработки может оставаться неопределенным, если для нескольких событий указано одно и то же время. Это может происходить чаще, чем вы думаете, поскольку время доставки обычно квантуется в целое число кадров. Например, если два отправителя хотят, чтобы события были отправлены в «этом кадре», «следующем кадре» или «через семь кадров», их время доставки будет идентичным.

Чтобы избавиться от такой неопределенности, событиям можно назначать приоритет. Когда временные метки у двух событий совпадают, то из них, которое имеет более высокий приоритет, всегда должно обслуживаться первым. Этого легко достичь, если сначала отсортировать события в очереди по времени доставки, а затем те, у которых совпадает время доставки, расставить по приоритетам.

Мы можем поддерживать до 4 млрд уникальных приоритетов, закодировав их в виде обычного 32-битного целого числа, а можем ограничиться двумя-тремя уникальными уровнями, например низким, средним и высоким. В любом игровом

движке существует какое-то минимальное количество уровней приоритета, позволяющее избавиться от любых реальных неоднозначностей в системе. Как правило, стоит стремиться к минимуму. Если приоритетов слишком много, вам будет крайне сложно разобраться в том, какое событие станет обрабатываться первым в той или иной ситуации. Однако у каждой игры свои требования к системе событий, и у вас могут быть другие условия.

Некоторые проблемы с очередью событий

Усложнение системы событий. Для реализации системы событий с очередью требуется больше кода, нужны дополнительные структуры данных и более сложные алгоритмы по сравнению с системой, где события обрабатываются без задержки. Повышение сложности обычно затягивает разработку и увеличивает стоимость поддержки и развития системы во время создания игры.

Глубокое копирование событий и их аргументов. В случае незамедлительной обработки событий данные в их аргументах должны существовать только на время выполнения функции обработки и тех функций, которые она вызывает. Это означает, что события и их аргументы могут находиться буквально на любом участке памяти, включая стек вызовов. Например, мы могли бы написать функцию следующего вида:

```
void SendExplosionEventToObject(GameObject& receiver)
{
    // Размещаем аргументы события в стеке вызовов.
    Point centerPoint(-2.0f, 31.5f, 10.0f);
    F32   damage = 5.0f;
    F32   radius = 2.0f;

    // Размещаем событие в стеке вызовов.
    Event event("Explosion");
    event.SetArgFloat("Damage", damage);
    event.SetArgPoint("Center", &centerPoint);
    event.SetArgFloat("Radius", radius);

    // Посылаем событие, в результате чего
    // немедленно вызывается обработчик получателя, как показано далее.
    event.Send(receiver);
    //{
    // receiver.OnEvent(event);
    //}
}
```

Когда событие помещается в очередь, его аргументы должны храниться вне отправляющей его функции. Это означает, что прежде, чем помещать объект события в очередь, его нужно скопировать. Мы должны выполнить глубокое копирование, при котором копируется не только сам объект, но и все полезное

содержимое его аргументов, включая любые данные, на которые они ссылаются. Глубокое копирование событий гарантирует отсутствие «висящих» ссылок на данные, существующих только в области видимости отправляющей функции, и позволяет хранить событие неограниченное время. Представленный ранее пример функции, отправляющей событие, будет выглядеть почти так же, даже если использовать очередь. В следующем листинге показана реализация функции `Event::Queue()` — более сложной, чем ее аналог, `Send()` (это видно по коду, выделенному курсивом):

```
void SendExplosionEventToObject(GameObject& receiver)
{
    // Мы по-прежнему можем разместить аргументы события в стеке вызовов.
    Point centerPoint(-2.0f, 31.5f, 10.0f);
    F32   damage = 5.0f;
    F32   radius = 2.0f;

    // Событие все еще можно хранить в стеке вызовов.
    Event event("Explosion
    event.SetArgFloat("Damage", damage);
    event.SetArgPoint("Center", &centerPoint);
    event.SetArgFloat("Radius", radius);

    // Этот код помещает событие в очередь
    // получателя для последующей обработки.
    // Обратите внимание на то, что перед попаданием
    // в очередь событие должно пройти через
    // глубокое копирование, так как его оригинал
    // хранится в стеке вызовов и пропадет из
    // области видимости после завершения этой
    // функции.
    event.Queue(receiver);
    //{
    //   Event* pEventCopy = DeepCopy(event);
    //   receiver.EnqueueEvent(pEventCopy);
    //}
}
```

Динамическое выделение памяти для событий в очереди. Для глубокого копирования объектов событий требуется динамическое выделение памяти, которое, как я уже много раз отмечал, нежелательно в игровых движках: оно может оказаться высокозатратным и имеет тенденцию фрагментировать память. Тем не менее, если мы хотим размещать события в очереди, память для них придется выделять динамически.

Как и с любым другим динамическим выделением в игровых движках, предпочтение следует отдавать быстрому механизму, не подверженному фрагментации. Можно воспользоваться системой выделения на основе пула, но она подходит только для случаев, когда объекты событий одинакового размера, равно как и элементы данных, из которых состоят их аргументы. Так действительно может быть:

например, каждый аргумент может иметь тип `Variant`, как говорилось ранее. Но если объекты событий и/или их аргументы разных размеров, может пригодиться компактный аллокатор памяти, который, как вы помните, состоит из нескольких пулов, по одному для каждого из нескольких заранее определенных компактных размеров. При проектировании системы событий с использованием очереди всегда тщательно анализируйте требования к динамическому выделению памяти.

Конечно, здесь применимы и другие подходы. Например, в Naughty Dog память для событий в очереди выделяется в виде перемещаемых блоков. Подробнее о перемещаемой памяти можно почитать в подразделе 6.2.2.

Отладка сложных ситуаций. Если событие хранится в очереди, его непосредственный отправитель не занимается вызовом обработчика. Поэтому, в отличие от системы с немедленной обработкой, не представляется возможным узнать, откуда пришло событие. Мы не можем пройтись по стеку вызовов в отладчике и исследовать состояние отправителя или обстоятельства, при которых это событие отправлено. Это может немного осложнить отладку отложенных событий. Все еще более усугубляется, если события перенаправляются от одного объекта к другому.

Некоторые движки хранят отладочную информацию, которая позволяет отследить перемещение события по системе, но, как бы вы к этому ни подошли, отсутствие очереди намного упрощает отладку.

Использование очереди событий вызывает интересные ошибки с состоянием гонки, которые сложно отследить. Иногда, чтобы добиться правильного порядка обновления игровых объектов во время формирования кадра, события приходится отправлять по нескольку раз за один игровой цикл. Например, в ходе обновления определенной анимации обнаруживается, что она уже завершилась. В этом случае необходимо инициировать отправку события, обработчик которого захочет запустить новую анимацию. Очевидно, мы хотим избежать задержки в один кадр между концом первой анимации и началом следующей. Для этого сначала нужно обновить счетчик анимации (чтобы система обнаружила завершение анимации и послала событие), затем отправить событие (чтобы обработчик получил возможность запросить новую анимацию), после чего начнется слияние анимации (чтобы мы могли обработать и вывести первый кадр новой анимации). Это проиллюстрировано в следующем фрагменте кода:

```
while (true) // main game loop
{
    // ...

    // Обновляем счетчики анимации. Система может обнаружить
    // окончание клипа и послать событие EndOfAnimation.
    g_animationEngine.UpdateLocalClocks(dt);

    // Дальше происходит отправка событий. Это
    // позволяет обработчику EndOfAnimation начать при
    // необходимости новую анимацию в текущем кадре.
    g_eventSystem.DispatchEvents();
```

```
    // Наконец, начинается слияние всех активных анимаций,
    // включая любые новые клипы, запущенные ранее в этом кадре.
    g_animationEngine.StartAnimationBlending();

    // ...
}
```

16.8.10. Некоторые проблемы с незамедлительной отправкой событий

Отказ от очереди тоже чреват некоторыми проблемами. Например, незамедлительная обработка событий может привести к созданию чрезвычайно глубоких стеков вызовов. Объект А может послать объекту Б событие, в обработчике которого объект Б пошлет еще одно событие, которое пошлет еще одно и еще одно и т. д. В игровых движках, поддерживающих немедленную обработку событий, часто можно увидеть стеки вызовов наподобие следующего:

```
...
ShoulderAngel::OnEvent()
Event::Send()
Characer::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
HandleSoundEffect()
AnimationEngine::PlayAnimation()
Event::Send()
Character::OnEvent()
Event::Send()
Character::OnEvent()
Event::Send()
Character::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::Update()
GameWorld::UpdateObjectsInBucket()
Engine::GameLoop()
main()
```

В экстремальных ситуациях такое большое количество вызовов может исчерпать место в стеке, особенно если есть бесконечный цикл с отправкой событий. Но настоящая проблема состоит в том, что каждая функция, обрабатывающая события, должна быть полностью *реентерабельной*. Это означает, что обработчик можно вызывать рекурсивно, без нежелательных побочных эффектов. В качестве немного искусственного примера представьте себе функцию, которая инкрементирует значение глобальной переменной. Если инкрементация должна происхо-

дить только один раз в каждом кадре, эта функция *не является* реентерабельной, поскольку ее повторные рекурсивные вызовы инкрементируют переменную несколько раз.

16.8.11. Системы передачи событий/сообщений на основе данных

Системы событий дают программистам большую гибкость при работе со статически типизированным механизмом вызова функций, таким как в языках C или C++. Но это не предел мечтаний. До сих пор вся логика отправки и получения событий была встроена в код и, как следствие, находилась исключительно в руках программистов. Если спроектировать систему событий вокруг данных, ее мощь станет доступной и игровым дизайнерам.

Систему событий на основе данных можно реализовать множеством разных способов. Если говорить об одной крайности, когда все аспекты системы встроены в код, то можно добавить в нее поддержку простой конфигурации. Например, мы можем позволить дизайнерам определять то, как отдельные объекты или целые классы объектов реагируют на определенные события. Представьте, что при выборе объекта в игровом редакторе появляется список всех событий, которые тот может получить. Для управления каждым событием дизайнер мог бы использовать флажки и раскрывающиеся списки, выбирая поведение объекта из нескольких заданных в коде вариантов. Например, персонажи под управлением ИИ могут быть сконфигурированы для выполнения одного из следующих действий в ответ на событие PlayerSpotted: убежать, атаковать или проигнорировать. В некоторых настоящих коммерческих движках система событий в целом реализована именно так.

Теперь рассмотрим другую крайность: движок может предоставлять игровым дизайнерам простой скриптовый язык (эту тему подробно исследуем в разделе 16.9). В этом случае дизайнеры могут буквально писать код, который определяет реакцию того или иного вида объектов на определенный тип событий. В такой модели дизайнеры, в сущности, становятся программистами (работая с менее мощным, но более простым в применении и, как мы надеемся, менее подверженным ошибкам языком, чем настоящие программисты), поэтому их возможности неограниченны. Дизайнеры могут определять новые типы событий и отправлять, получать и обрабатывать их каким угодно способом. Это то, что мы делаем в Naughty Dog.

Проблема простой, конфигурируемой системы событий в том, что она может существенно ограничить автономность дизайнеров игры — то есть то, что они могут делать самостоятельно, без помощи программистов. В то же время у полностью скриптового решения есть недостатки: многие игровые дизайнеры не имеют профессиональной подготовки в сфере программирования, поэтому изучение и использование скриптового языка может оказаться для них сложной задачей. К тому же не имеющие опыта написания скриптов или программирования дизайнеры, вероятно, будут допускать больше ошибок в коде, чем их коллеги-программисты. Это может привести к неприятным сюрпризам во время альфа-тестирования.

В итоге некоторые игровые движки пытаются найти золотую середину. Они применяют развитые пользовательские интерфейсы, чтобы предоставить дизайнерам большую гибкость, но при этом не дают возможности работать с полноценным скриптовым языком. Один из вариантов — предоставление графического языка программирования в стиле блок-схем. Суть этой системы в том, чтобы пользователь имел ограниченный и контролируемый набор атомарных операций, которые мог бы компоновать произвольным образом. Например, в ответ на событие PlayerSpotted мог бы сформировать блок-схему, в результате реализации которой персонаж отступает к ближайшему укрытию, проигрывает анимацию, ждет 5 с и затем атакует. Пользовательский интерфейс может обеспечивать также обработку ошибок и проверку данных, чтобы предотвратить случайное появление проблем. Примером такой системы является Blueprints в движке Unreal (подробнее о нем — далее).

Коммуникационные системы для передачи данных

Когда систему событий, использующую вызовы функций, пытаются переориентировать на данные, возникает проблема: разные типы событий обычно оказываются несовместимыми. Например, представьте себе игру, в которой у игрока есть электромагнитное импульсное ружье. Его импульс выводит из строя освещение и электронные приборы, пугает мелких животных и создает ударную волну, из-за которой колышутся все близлежащие растения. У каждого из этих типов игровых объектов уже может быть нужная реакция. В ответ на событие Scare мелкие животные могут суетливо убегать, электронные устройства в ответ на событие TurnOff — выключаться, а растения, получив событие Wind, — качаться. Проблема в том, что электромагнитное ружье несовместимо с обработчиками всех этих игровых объектов. В итоге придется реализовать новый тип события (скажем, EMP) и написать для него отдельные обработчики в каждом типе игровых объектов, чтобы предусмотреть нужную реакцию.

Чтобы решить эту проблему, можно убрать из уравнения тип событий и относиться к этому так, будто мы *передаем потоки данных* от одного объекта к другому. В такой системе у каждого объекта будет как минимум один *входящий порт*, к которому можно подключить поток данных, и как минимум один исходящий, через который данные можно посылать другим объектам. Если предположить, что у нас есть какой-то механизм для соединения этих портов (например, пользовательский интерфейс, в котором это можно делать с помощью гибких линий), это позволит создавать поведение любой сложности. Возвращаясь к примеру: у электромагнитного ружья будет исходящий порт (скажем, Fire), который шлет булев сигнал. Большую часть времени он генерирует значение 0 (false), но когда ружье стреляет, оно извергает короткий (длиной один кадр) импульс со значением 1 (true). У других объектов игрового мира есть двоичные входящие порты, инициирующие различные реакции. Животные могут принимать на вход Scare, электронные устройства — TurnOn, а объекты с листвой — Sway. Если подключить исходящий порт ружья Fire к входящим портам этих объектов, открытие огня может вызвать нужный эффект

(следует отметить: прежде чем подключать выход ружья Fire к входу электронных устройств TurnOn, мы пропустили его через узел, который *инвертирует* свой ввод; это сделано потому, что во время выстрела они должны *выключаться*). Компоновочная диаграмма для этого примера показана на рис. 16.22.

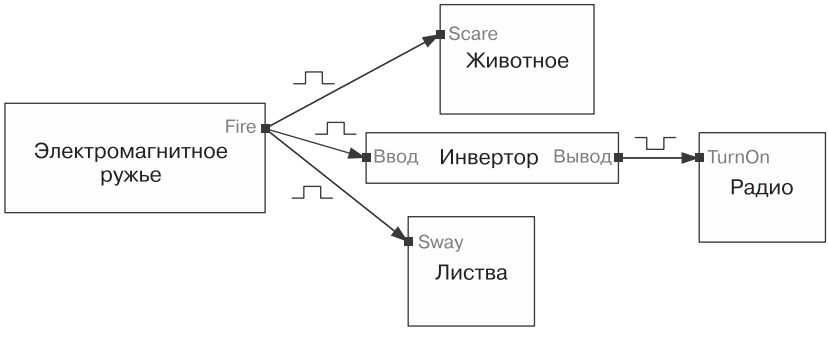


Рис. 16.22. При стрельбе электромагнитное ружье подает 1 на выход Fire. Этот выход можно соединить с входящим портом, ожидающим булева значения, чтобы инициировать реакцию, связанную с выходом

Программисты решают, какого рода порты будет иметь каждый тип игровых объектов. Затем дизайнеры с помощью графического интерфейса могут соединить эти порты произвольным образом, чтобы получить нужное поведение в игре. Программисты предоставляют и другие виды узлов для применения в этом графе. Они могут, например, инвертировать свой ввод, сгенерировать синусоиду или вывести текущее время игры в секундах.

Таким образом можно передавать разные типы данных. Некоторые порты генерируют или принимают булевы значения, а другие могут быть рассчитаны на действия с дробными числами. Бывают порты, которые работают с трехмерными векторами, светами, целыми числами и т. д. При использовании такой системы важно соединять только порты с совместимыми типами данных. Или же должен быть какой-то механизм для автоматического преобразования типов, когда типы соединяемых портов не совпадают. Например, если подключить дробный выход к булеву входу, любые значения меньше 0,5 будут автоматически превращаться в false, а значения, большие или равные 0,5, — в true. Этот подход лежит в основе графических систем событий, таких как Blueprints в Unreal Engine 4. Снимок экрана Blueprints показан на рис. 16.23.

Некоторые преимущества и недостатки визуального программирования

Преимущества графического пользовательского интерфейса перед простым текстовым скриптовым языком, наверное, довольно очевидны: простота в использовании, плавная кривая обучения с потенциалом для вспомогательных средств и всплывающих подсказок в помощь пользователю, строгая проверка на ошибки.

Среди недостатков интерфейса в стиле блок-схем можно выделить высокую стоимость разработки, отладки и поддержки такой системы, дополнительную сложность, которая может вызывать досадные ошибки, способные сорвать график работ, и тот факт, что иногда такие инструменты ограничивают возможности дизайнеров. Для сравнения: в качестве преимуществ текстового языка программирования можно назвать относительную простоту (это означает, что он не так сильно подвержен ошибкам), возможность легко искать и заменять участки исходного кода, а также свободу выбора — пользователь может работать в наиболее удобном для себя текстовом редакторе.

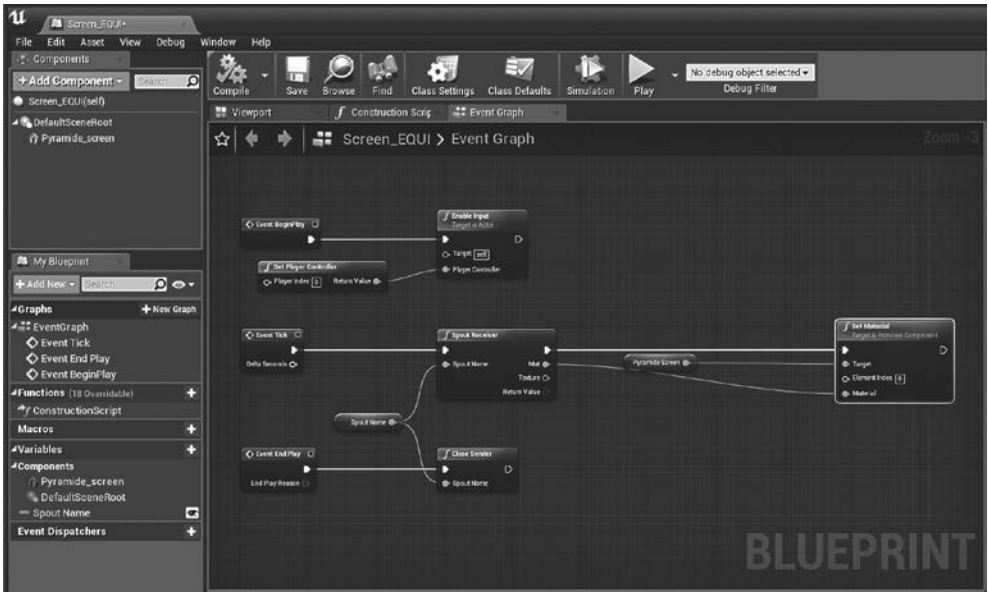


Рис. 16.23. Blueprints в Unreal Engine 4

16.9. Скрипты

Скриптовым можно назвать язык программирования, основное назначение которого — позволить пользователям изменять поведение приложения и управлять им. Например, с помощью языка Visual Basic можно изменить поведение Microsoft Excel, языки MEL и Python используются для изменения поведения Maya. В контексте игровых движков *скриптовый язык программирования* должен быть высокоуровневым и относительно простым для изучения, давая пользователю удобный доступ к большинству часто применяемых возможностей движка. Таким образом, скриптовый язык можно задействовать для разработки новых и модификации существующих игр, и для этого не обязательно быть программистом.

16.9.1. Для выполнения кода и описания данных

Следует провести четкое разграничение. Игровые скриптовые языки в целом делятся на две категории.

- *Языки описания данных*. Основная их цель — дать возможность пользователям создавать и наполнять структуры данных, которые позже загружает движок. Языки этого типа часто являются декларативными (см. далее) и выполняются/анализируются либо во время разработки, либо на этапе выполнения, когда данные попадают в память.
- *Скриптовые языки времени выполнения*. Эти языки предназначены для выполнения в контексте движка во время игры. Их обычно применяют для расширения или изменения встроенных возможностей объектной модели и/или других подсистем движка.

В этом разделе мы в основном сосредоточимся на скриптовых языках времени выполнения. Мы реализуем некоторые аспекты игрового процесса за счет расширения и изменения объектной модели игры.

16.9.2. Характеристики языков программирования

Прежде чем переходить к обсуждению скриптовых языков, полезно определиться с терминологией. Языков программирования существует великое множество, но все их можно классифицировать по относительно небольшому числу критериев.

- *Интерпретируемые и компилируемые языки*. Исходный код *компилируемого* языка транслируется программой, называемой компилятором, в машинный код, который может быть выполнен непосредственно процессором. Для сравнения: исходный код *интерпретируемого* языка анализируется прямо во время выполнения или заранее компилируется в платформонезависимый *байт-код*, который затем может быть выполнен *виртуальной машиной* (ВМ). Последняя ведет себя как эмулятор воображаемого процессора, а байт-код содержит список инструкций машинного языка, которые этот виртуальный процессор потребляет. Преимущество виртуальной машины состоит в том, что ее можно довольно легко перенести на практически любую аппаратную платформу и встроить в основное приложение, например в игровой движок. Цена, которую мы платим за такую гибкость, в основном заключается в скорости выполнения: ВМ обычно выполняет свои инструкции байт-кода намного медленнее, чем физический процессор выполняет инструкции машинного языка.
- *Императивные языки*. В императивном языке программа описывается в виде последовательных инструкций, каждая из которых выполняет какую-то операцию и/или меняет состояние данных в памяти. C и C++ являются императивными языками.
- *Декларативные языки*. Декларативный язык описывает, *что* нужно сделать, но не уточняет, *как* именно должны быть получены результаты. Это решение

принадлежит людям, реализующим язык. Примером декларативного языка является Prolog. Языки разметки, такие как HTML и TeX, тоже можно причислить к декларативным.

- *Функциональные языки.* Функциональные языки, формально являясь подмножеством декларативных, стремятся полностью избавиться от состояния. В функциональном языке программа определяется набором функций. Каждая функция получает свои результаты без побочных эффектов (то есть она не вносит в систему заметных изменений, если не считать исходящие данные). Программа создается путем передачи входящих данных от одной функции к другой, пока не будет сгенерирован желаемый результат. Эти языки обычно хорошо подходят для реализации конвейеров обработки данных. К тому же они обладают явными преимуществами при реализации многопоточных приложений, так как ввиду отсутствия изменяемого состояния им не нужно прибегать к мьютексному блокированию. В качестве примеров функциональных языков можно привести OCaml, Haskell и F#.
- *Процедурные и объектно-ориентированные языки.* В процедурном языке основным строительным блоком программы является *процедура* или *функция*. Процедуры и функции выполняют операции, вычисляют результаты и/или изменяют состояние различных структур данных в памяти. Для сравнения: основным строительным блоком программы в объектно-ориентированном языке служит *класс* — структура данных, тесно связанная с набором процедур/функций, которые знают, как управлять и манипулировать ее данными.
- *Языки с поддержкой рефлексии.* В таких языках информация о типах данных, компоновке свойств, отношениях между функциями и членами иерархии классов доступна для анализа на этапе выполнения. В языках, не поддерживающих рефлексию, большинство метаинформации известно только во время компиляции и лишь очень небольшая ее часть доступна для исполняемого кода. C# поддерживает рефлексию, а C и C++ — нет.

Отличительные характеристики игровых скриптовых языков

Здесь перечислены некоторые характеристики, выделяющие *скриптовые* языки программирования на фоне их *компилируемых* собратьев.

- *Интерпретируемость.* Большинство скриптовых языков интерпретируются виртуальной машиной и не подлежат компиляции. Это сделано для гибкости, переносимости и высоких темпов разработки (см. далее). Программа, представленная в виде платформонезависимого кода, может восприниматься движком как данные. Ее можно загрузить в память, как любой другой ресурс, не прибегая к помощи операционной системы, без которой, к примеру, не обойтись при загрузке DLL на ПК или PRX в PlayStation 3. Поскольку код выполняется виртуальной машиной, а не самим процессором, игровой движок получает значительную свободу в отношении того, как и когда запускать скриптовую программу.

- *Легковесность*. Большинство игровых скриптовых языков предназначены для использования во встраиваемых системах. Благодаря этому их ВМ просты и обладают довольно низким потреблением памяти.
- *Поддержка высоких темпов разработки*. При каждом изменении в компилируемом коде программа должна быть заново скомпилирована и скомпонована, а игра — перезапущена, иначе вы не увидите результатов этого изменения (разве что среда разработки поддерживает какую-то разновидность режима «изменить и продолжить»). Для сравнения: при редактировании скриптового кода результаты изменений обычно можно увидеть довольно быстро. Некоторые игровые движки умеют перезагружать скриптовый код на лету, без перезапуска игры. В любом случае время между внесением изменения и отображением результатов оказывается намного меньшим по сравнению с редактированием исходного кода компилируемых языков.
- *Удобство и простота в использовании*. Скриптовые языки часто подстраиваются под нужды конкретной игры. Они могут предусматривать возможности, которые делают выполнение распространенных действий более простым, интуитивно понятным и менее подверженным ошибкам. Например, игровой скриптовый язык может предоставлять функции или специальный синтаксис для поиска объектов по имени, отправки и обработки событий, остановки и изменения хода времени, запуска таймера, реализации конечного автомата, предоставления доступа к конфигурируемым параметрам в редакторе игрового мира (для применения игровыми дизайнерами) или даже для выполнения сетевой репликации в многопользовательских играх.

16.9.3. Некоторые распространенные игровые скриптовые языки

При реализации в игре системы скриптования времени выполнения необходимо принять фундаментальное решение: выбрать готовый язык (коммерческий или открытый) и подогнать его под свои нужды или спроектировать и написать собственный язык с нуля.

Создание совершенно нового языка обычно не стоит связанных с этим хлопот и затрат на поддержку на протяжении работы над проектом. К тому же вам будет сложно (если вообще возможно) найти игровых дизайнеров и программистов, уже знакомых с вашим нестандартным внутренним языком, поэтому учитывайте затраты на подготовку специалистов. Однако это, вне всяких сомнений, самый гибкий вариант, и гибкость может оправдать вложенные ресурсы.

Для многих студий удобнее выбрать довольно известный и зрелый скриптовый язык и дополнить его функциями, относящимися к конкретному игровому движку. Существует множество готовых скриптовых языков, развитых и надежных, которые применяются в большом количестве проектов как в игровой индустрии, так и за ее пределами.

В следующих пунктах мы исследуем ряд игровых скриптовых языков, как узкоспециализированных — предназначенных для определенных игр, так и универсальных, которые широко используются в игровых движках.

QuakeC

Джон Кармак из Id Software реализовал специальный скриптовый язык для игры *Quake* под названием *QuakeC* (QC). Это, в сущности, упрощенная разновидность языка программирования C с тесной интеграцией в движок *Quake*. Он не поддерживает указатели и определение произвольных структур, но позволяет удобно манипулировать *сущностями* (так в *Quake* называются игровые объекты), и с его помощью можно отправлять и принимать/обрабатывать игровые события. QuakeC является интерпретируемым, императивным, процедурным языком программирования.

Богатые возможности, которые язык QuakeC предоставлял игрокам, послужили одной из причин рождения того, что называется *сообществом моддинга*. Скриптовые языки и другие формы модификации игр на основе данных позволяют игрокам изменять всевозможные аспекты игрового процесса — как получать небольшие отклонения от оригинала, так и создавать совершенно новые игры.

UnrealScript

Наверное, самым известным примером отдельного скриптового языка является *UnrealScript* в Unreal Engine. Этот язык имеет синтаксис в стиле C++ и поддерживает большинство концепций, привычных для программистов на C и C++, включая классы, локальные переменные, циклы, массивы и структуры для организации данных, строки, идентификаторы на основе хешей (в Unreal они называются `FName`) и ссылки на объекты (но не произвольные указатели). UnrealScript является интерпретируемым, императивным, объектно-ориентированным языком программирования.

UnrealScript больше не поддерживается компанией Epic. Теперь для изменения поведения игр разработчики используют систему графического скриптования Blueprints или пишут код на C++.

Lua

Lua — это широко известный и популярный скриптовый язык, который легко интегрируется в приложения наподобие игровых движков. На веб-сайте Lua (http://www.lua.org/about.html) его называют «лидирующим скриптовым языком в играх».

Согласно тому же сайту у Lua имеется ряд ключевых преимуществ.

- *Надежность и зрелость*. Lua используется во множестве коммерческих продуктов, включая *Photoshop Lightroom* от Adobe и многие игры, такие как *World of Warcraft*.

- *Хорошая документация*. Lua имеет полное и простое для понимания справочное руководство [25], доступное как в Интернете, так и в виде книги. Этому языку посвящен целый ряд изданий, включая [26] и [50].
- *Отличная производительность на этапе выполнения*. Lua выполняет свой байт-код быстрее и эффективнее, чем многие другие скриптовые языки.
- *Переносимость*. Lua имеет встроенную поддержку всех разновидностей Windows и UNIX, мобильных устройств и встраиваемых микропроцессоров. Переносимость была заложена в этот язык с самого начала, поэтому его легко адаптировать под новые платформы.
- *Изначальная поддержка встраиваемых систем*. Lua имеет очень низкое потребление памяти — примерно 350 КиБ для интерпретатора и всех библиотек.
- *Простота, богатые возможности и расширяемость*. Ядро языка Lua очень компактное и простое, но оно поддерживает метамеханизмы, позволяющие расширять его встроенные возможности практически неограниченно. Например, сам по себе Lua не является объектно-ориентированным языком, но поддержку ООП можно добавить (и многие это делают) через метамеханизм.
- *Бесплатность, открытость*. Lua имеет открытый исходный код, который распространяется под очень либеральной лицензией MIT.

Lua — динамически типизируемый язык. Это означает, что типами обладают только значения, но не сами переменные (каждое значение содержит информацию о типе). Основной структурой данных в Lua служит *таблица*, известная также как ассоциативный массив. Это, в сущности, список пар «ключ — значение» с оптимизированным доступом к массиву по ключу.

Lua предоставляет удобный интерфейс к языку C — виртуальная машина Lua умеет вызывать функции, написанные на C, и манипулировать ими так, как будто они являются обычными функциями Lua.

Блоки кода в Lua, называемые *фрагментами*, считаются объектами первого сорта, и манипулировать ими может сама программа. Код может быть выполнен как в исходном виде, так и после компиляции в бай-код. Это позволяет виртуальной машине выполнять строки с кодом на Lua так, как будто они компилированы в саму программу. Lua также поддерживает некоторые продвинутые концепции программирования, такие как корутины. Это простая разновидность кооперативной многозадачности, в которой потоки уступают процессорные ресурсы явным образом, вместо того чтобы периодически чередоваться, как в системах с вытесняющей многозадачностью.

У Lua есть свои нюансы. Например, гибкий механизм привязки функций позволяет довольно легко переопределять важные глобальные функции вроде `sin()` для выполнения совершенно других действий, что обычно происходит непреднамеренно. Но в целом этот язык отлично зарекомендовал себя в игровом скриптовании.

Python

Python является процедурным, объектно-ориентированным, динамически типизируемым скриптовым языком, который отличается простотой в использовании, гибкостью и возможностью интеграции с другими языками. Как и Lua, Python часто выбирают для скриптования в играх. Далее перечислены некоторые из его самых сильных сторон согласно официальному сайту Python (http://www.python.org).

- *Четкий и понятный синтаксис.* Код на Python легко читать. Отчасти это связано с тем, что его синтаксис вынуждает применять определенный стиль отступов (на самом деле он анализирует пробельные символы, используемые в качестве отступов, чтобы определить контекст каждой строчки кода).
- *Поддержка рефлексии.* Python включает в себя мощные средства интроспекции на этапе выполнения. Его классы являются объектами первого сорта. Это означает, что их можно запрашивать и работать с ними так, как с любым другим объектом.
- *ООП.* Одним из преимуществ Python перед Lua является то, что поддержка ООП встроена в ядро языка. Это немного облегчает интеграцию Python с объектной моделью игры.
- *Модульность.* Python поддерживает иерархические пакеты, поощряя аккуратную системную архитектуру и хорошую инкапсуляцию.
- *Обработка ошибок на основе исключений.* Исключения упрощают обработку ошибок в Python, делая ее более элегантной и локальной по сравнению с языками без поддержки исключений.
- *Обширная стандартная библиотека, богатый выбор сторонних модулей.* У Python есть библиотеки практически на любой случай (правда!).
- *Встраиваемость.* Python можно легко встроить в приложение, такое как игровой движок.
- *Обширная документация.* Для Python существует много документации и практических руководств, как онлайн, так и в виде книг. Начать знакомство с языком можно с веб-сайта Python (http://www.python.org).

Синтаксис Python во многом напоминает синтаксис C (например, оператор `=` используется для присваивания, а `==` — для проверки равенства). Однако отступы в этом языке служат единственным средством определения *области видимости*, в отличие от C, где для этого предназначены фигурные скобки. Основными структурами данных в Python выступают *список* (последовательность атомарных значений или других вложенных списков с линейной индексацией) и *словарь* (таблица с парами «ключ — значение»). Обе они могут хранить экземпляры друг друга, что позволяет с легкостью создавать структуры любой сложности. Кроме того, *классы* (унифицированные коллекции элементов данных и функций) встроены прямо в язык.

Python поддерживает *утиную типизацию* — разновидность динамической типизации, в которой функциональный интерфейс объекта определяется его типом, а не статической иерархией наследования. Иными словами, классы, поддерживающие один и тот же интерфейс (то есть список функций с определенными сигнатурами), взаимозаменяемы. Это мощная парадигма: Python, по сути, позволяет использовать полиморфизм, не применяя при этом наследование. Утиная типизация в некоторых аспектах напоминает шаблонное метапрограммирование в C++, но она более гибкая, так как связывание вызывающей и вызываемой сторон происходит динамически на этапе выполнения. Название термина пошло от известной фразы, которую приписывают Джеймсу Уиткомбу Райли: «Если объект выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка». Больше об утиной типизации можно узнать на странице https://ru.wikipedia.org/wiki/Утиная_типизация.

Подведем итоги: язык Python прост в применении и изучении, легко встраивается в игровые движки, хорошо интегрируется с объектной моделью игры и может стать отличным мощным решением в качестве игрового скриптового языка.

Pawn/Small/Small-C

Pawn — это легковесный, динамически типизируемый, C-подобный скриптовый язык, созданный Марком Питером. Ранее известный как *Small*, он является развитием более раннего подмножества языка C под названием *Small-C*, написанного Роном Кейном и Джеймсом Хендриксом. Это интерпретируемый язык: исходный код компилируется в байт-код, известный как P-code, который интерпретируется виртуальной машиной во время выполнения.

Язык Pawn создавался с расчетом на умеренное потребление памяти и очень быстрое выполнение своего байт-кода. В отличие от C, переменные в Pawn имеют динамическую типизацию. Pawn также поддерживает конечные автоматы, включая локальные переменные состояния. Благодаря этой уникальной возможности данный язык хорошо подходит для многих игровых приложений. У Pawn есть качественная документация (http://www.compuphase.com/pawn/pawn.htm). Это проект с открытым исходным кодом, который можно использовать бесплатно под лицензией Zlib/libpng (http://www.opensource.org/licenses/zlib-license.php).

C-подобный синтаксис делает этот язык простым в изучении для любого программиста на C/C++, а также позволяет легко интегрировать его с игровыми движками, написанными на C. Поддержка конечных автоматов может быть очень полезной при разработке игры. Pawn успешно применяется в целом ряде игровых проектов, включая *Freaky Flyers* от Midway, зарекомендовав себя в качестве достойного скриптового языка для игр.

16.9.4. Архитектуры для скриптования

Скриптовый код можно применять в игровом движке множеством способов. В вашем распоряжении широкий диапазон всевозможных архитектур, от небольших скриптовых фрагментов, выполняющих простые функции от имени объекта или

подсистемы движка, до высокоуровневых скриптов, которые управляют выполнением игры. Вот лишь несколько вариантов.

- *Скриптовые функции обратного вызова.* Этот подход подразумевает, что функциональность движка в основном написана на компилируемом языке программирования, но некоторые ее ключевые аспекты поддаются модификации. Это часто делается с помощью *хуков* или *обратных вызовов* — пользовательских функций, которые вызываются движком с целью изменения его поведения. Конечно, хук может быть написан на компилируемом языке, но скриптование — тоже подходящий вариант. Например, при обновлении объекта в игровом цикле движок может сделать необязательный обратный вызов, написанный в виде скрипта. Это дает возможность пользователям влиять на то, как игровые объекты обновляются с течением времени.
- *Скриптовый обработчик событий.* Обработчик событий, в сущности, является особым видом хука, чье назначение состоит в том, чтобы позволить объекту реагировать на интересующие его ситуации в игровом мире (такие как взрыв) или в самом движке (например, нехватка памяти). Многие игровые движки позволяют пользователям писать хуки обработки событий как на компилируемом, так и на скриптовом языке.
- *Расширение или определение новых типов игровых объектов с помощью скриптов.* Некоторые скриптовые языки позволяют расширять типы игровых объектов, реализованные на компилируемом языке. На самом деле примером этого (в ограниченном масштабе) являются обратные вызовы и обработчики событий, но данную идею можно развить до того, чтобы позволить определение новых типов игровых объектов в скриптах. Это можно сделать с помощью *наследования* (делая скриптовый класс потомком класса, написанного на компилируемом языке) или *композиции* (подключая экземпляр скриптового класса к скомпилированному игровому объекту).
- *Скриптовые компоненты или свойства.* Если компоненты или свойства лежат в основе объектной модели игры, их создание можно частично или полностью возложить на скрипты. Этот подход использовался студией Gas Powered Games в проекте *Dungeon Siege*. Объектная модель игры основывалась на свойствах, которые можно было реализовать как на C++, так и на собственном скриптовом языке компании, *Skrit* (http://ds.heavengames.com/library/dstk/skrit/skrit). На момент завершения проекта у них насчитывалось 148 скриптовых типов свойств и 21 тип на C++.
- *Скриптовый движок.* Скрипт может лежать в основе всей системы движка. Например, объектную модель игры можно было бы полностью написать на скриптовом языке, обращаясь к скомпилированному коду движка только в случаях, когда есть необходимость в низкоуровневых компонентах.
- *Скриптовая игра.* Некоторые игровые движки переворачивают с ног на голову отношения между скриптовым и компилируемым языками. Скриптовый код у них стоит во главе угла, а компилируемый код движка выступает лишь

библиотекой, которая используется для доступа к определенным высокопроизводительным возможностям. Примером такой архитектуры является движок Panda3D (http://www.panda3d.org). Основанные на нем игры можно писать целиком на языке Python, а скомпилированный движок (написанный на C++) играет роль библиотеки, которая вызывается из скриптового кода (Panda3D также позволяет писать игры полностью на C++).

16.9.5. Возможности игровых скриптовых языков на этапе выполнения

Основное назначение многих скриптовых языков состоит в реализации функций игрового процесса, и это часто делается за счет дополнения и модификации объектной модели игры. В этом разделе рассмотрим некоторые из наиболее распространенных требований и возможностей такой скриптовой системы.

Взаимодействие с компилируемым языком программирования

Скриптовый язык не может приносить пользу сам по себе. Игровой движок должен иметь возможность выполнять скриптовый код, но не менее важно, чтобы скриптовый код сам мог инициировать операции внутри движка.

Виртуальная машина скриптового языка обычно встраивается в игровой движок. Он ее инициализирует, запускает скриптовый код, когда это необходимо, и управляет выполнением скриптов. Единица выполнения зависит от конкретного языка и реализации игры.

- В функциональных скриптовых языках основной единицей выполнения часто выступает *функция*. Для вызова скриптовой функции движок должен найти ее имя в байт-коде и загрузить виртуальную машину для ее выполнения или воспользоваться уже существующей ВМ.
- В объектно-ориентированных скриптовых языках основной единицей выполнения обычно является *класс*. В таких системах можно создавать и уничтожать объекты, а также вызывать методы (функции-члены) из отдельных экземпляров класса.

Часто бывает полезно организовать двунаправленное взаимодействие между скриптом и компилируемым кодом. Таким образом, большинство скриптовых языков позволяют вызывать компилируемый код из скриптов. Подробности зависят от конкретного языка и реализации, но основной подход обычно состоит в том, чтобы позволить реализовать некоторые скриптовые функции на компилируемом языке. Чтобы воспользоваться функцией движка, скриптовый код делает обычный вызов. Виртуальная машина видит, что у функции есть компилируемая реализация, находит ее адрес (возможно, по имени или какому-то уникальному идентификатору) и вызывает ее. Например, в Python все или некоторые методы класса или функции модуля могут быть реализованы на C. Python хранит структуру данных, известную

как *таблица методов*, которая связывает имя каждой скриптовой функции, представленное в виде строки, с адресом функции на C, которая ее реализует.

Реальный пример — язык DC в Naughty Dog. В качестве примера посмотрим на то, как в движок Naughty Dog был интегрирован скриптовый язык времени выполнения под названием DC.

DC является разновидностью языка Scheme (который, в свою очередь, происходит от Lisp). Фрагменты исполняемого кода в DC называются скриптовыми лямбдами и являются примерным эквивалентом функций или блоков кода в языках семейства Lisp. Лямбды, которые пишет программист, имеют уникальные глобальные имена. Компилятор DC преобразует их во фрагменты байтового кода, которые загружаются в память при выполнении игры и могут быть найдены по имени с помощью простого функционального интерфейса в C++.

Получив указатель на фрагмент байт-кода со скриптовой лямбдой, движок может его выполнить, вызвав функцию выполнения виртуальной машины и передав ей указатель на байт-код. Сама функция на удивление проста. Она работает в цикле, последовательно считывая и выполняя инструкции байт-кода. Дойдя до конца, она завершается.

ВМ содержит банк регистров, способных хранить любого рода данные, с которыми может работать скрипт. Это реализовано в виде вариантного типа данных — совокупности всех возможных типов (варианты обсуждались в подразделе 16.8.4). Одни инструкции приводят к загрузке данных в регистр, другие могут инициировать поиск и использование содержимого регистров. Существуют инструкции для выполнения любых математических операций, доступных в языке, а также условных проверок: реализации элементов (if...), (when...) и (cond...) языка DC.

Виртуальная машина также поддерживает стек вызовов функций. Скриптовые лямбды (то есть функции) в DC, определенные с помощью синтаксиса (defun...), могут вызывать друг друга. Как и в любом другом процедурном языке программирования, для отслеживания состояния регистров и обратного адреса при вызове одной функции из другой необходим стек. В виртуальной машине DC стек вызовов буквально состоит из банков регистров, каждая новая функция получает отдельный банк. Благодаря этому не нужно сохранять состояние регистров перед вызовом и затем восстанавливать его после того, как функция завершила работу. Когда ВМ встречает инструкцию, которая заставляет ее вызвать другую скриптовую лямбду, байт-код этой лямбды ищется по ее имени. Затем на стек помещается новый кадр, а выполнение переходит к первой инструкции в скриптовой лямбде. Когда виртуальной машине встречается инструкция `return`, кадр вместе с обратным адресом (на самом деле это просто индекс инструкций байт-кода в вызывающей скриптовой лямбде, сразу после той, которая вызвала функцию) снимается со стека.

Псевдокод, представленный далее, должен дать вам представление о том, как выглядит основной цикл обработки инструкций в виртуальной машине DC:

```
void DcExecuteScript(DCByteCode* pCode)
{
    DCStackFrame* pCurStackFrame
    = DcPushStackFrame(pCode);
```

```
// Продолжаем, пока в стеке не закончатся кадры
// (то есть пока не завершится скриптовая лямбда верхнего уровня).
while (pCurStackFrame != nullptr)
{
    // Получаем следующую инструкцию. Мы никогда не выйдем
    // за пределы стека, так как инструкция return всегда находится
    // в конце и снимает со стека текущий кадр снизу.
    DCInstruction& instr
     = pCurStackFrame->GetNextInstruction();

    // Выполняем операцию инструкции.
    switch (instr.GetOperation())
    {
    case DC_LOAD_REGISTER_IMMEDIATE:
        {
            // Берем ближайшее значение, загружаемое из инструкции.
            Variant& data = instr.GetImmediateValue();

            // Определяем также, в какой регистр его нужно сохранить.
            U32 iReg = instr.GetDestRegisterIndex();

            // Берем регистр стекового кадра.
            Variant& reg
             = pCurStackFrame->GetRegister(iReg);

            // Сохраняем ближайшие данные в регистр.
            reg = data;
        }
        break;

    // Другие операции загрузки и сохранения в регистр...

    case DC_ADD_REGISTERS:
        {
            // Определяем два регистра, которые нужно добавить.
            // Результат будет сохранен в регистре A.
            U32 iRegA = instr.GetDestRegisterIndex();
            U32 iRegB = instr.GetSrcRegisterIndex();

            // Берем из стека 2 варианта регистра.
            Variant& dataA
             = pCurStackFrame->GetRegister(iRegA);

            Variant& dataB
             = pCurStackFrame->GetRegister(iRegB);

            // Добавляем регистры и сохраняем их в регистре A.
            dataA = dataA + dataB;
        }
        break;
```

```
        // Другие математические операции...

        case DC_CALL_SCRIPT_LAMBDA:
            {
                // Определяем, в каком регистре хранится
                // имя скриптовой лямбды, которую нужно
                // вызвать (предположительно она была
                // загружена предыдущей инструкцией загрузки).
                U32 iReg = instr.GetSrcRegisterIndex();

                // Берем подходящий регистр, содержащий имя вызываемой лямбды.
                Variant& lambda
                 = pCurStackFrame->GetRegister(iReg);

                // Ищем байт-код лямбды по ее имени.
                DCByteCode* pCalledCode
                 = DcLookUpByteCode(lambda.AsStringId());

                // Теперь вызываем лямбду, поместив в стек новый кадр.
                if (pCalledCode)
                {
                    pCurStackFrame
                     = DcPushStackFrame(pCalledCode);
                }
            }
            break;

        case DC_RETURN:
            {
                // Просто снимаем кадр со стека. Если мы находимся в верхней
                // лямбде стека, эта функция вернет nullptrи цикл прервется.
                pCurStackFrame = DcPopStackFrame();
            }
            break;

        // Другие инструкции...

            // ...

        } // конец switch
    } // конец while
}
```

В приведенном примере предполагается, что глобальные функции `DcPushStackFrame()` и `DcPopStackFrame()` занимаются управлением стеком с банками регистров подходящим для нас образом и глобальная функция `DcLookUpByteCode()` способна найти любую скриптовую лямбду по ее имени. Мы не станем приводить здесь их реализацию, поскольку этот пример является лишь демонстрацией работы внутреннего цикла ВМ скрипта, а не полноценным рабочим листингом.

Скриптовые лямбды в DC могут также вызывать скомпилированные функции, то есть глобальные функции, написанные на C++, которые выступают интерфейсом к самому движку. Когда виртуальная машина встречает инструкцию, вызывающую скомпилированную функцию на C++, адрес этой функции ищется по ее имени в глобальной таблице, встроенной в код программистами движка. Если подходящая функция найдена, ее аргументы берутся из регистров в текущем кадре стека и затем она вызывается. Подразумевается, что эти аргументы всегда имеют тип `Vvariant`. Если скомпилированная функция возвращает результат, он тоже должен быть экземпляром `Variant`, и его значение будет сохранено в регистре текущего стекового кадра, чтобы им могли воспользоваться последующие инструкции.

Глобальная таблица функций может иметь следующий вид:

```
typedef Variant DcNativeFunction(U32 argCount,
                                 Variant* aArgs);

struct DcNativeFunctionEntry
{
    StringId            m_name;
    DcNativeFunction*   m_pFunc;
};
DcNativeFunctionEntry g_aNativeFunctionLookupTable[] =
{
    { SID("get-object-pos"), DcGetObjectPos },
    { SID("animate-object"), DcAnimateObject },
    // и т. д.
};
```

Далее показано, как может выглядеть реализация компилируемой функции в DC. Обратите внимание на то, как ей передается массив аргументов типа `Variant`. Функция может проверить, совпадает ли количество переданных ей аргументов с тем, которого она ожидает. Она может также проверить их тип (-ы) и подготовиться к обработке ошибок, которые авторы DC-скриптов, возможно, допустили при ее вызове. В Naughty Dog мы написали итератор, удобный для извлечения аргументов одного за другим и их проверки.

```
Variant DcGetObjectPos(U32 argCount, Variant* aArgs)
{
    // Итератор ожидает максимум два аргумента.
    DcArgIterator args(argCount, aArgs, 2);

    // Подготавливаем возвращаемое значение по умолчанию.
    Variant result;
    result.SetAsVector(Vector(0.0f, 0.0f, 0.0f));

    // Используем итератор для извлечения аргументов.
    // Он автоматически маркирует пропущенные или некорректные аргументы.
    StringId objectName = args.NextStringId();
    Point* pDefaultPos = args.NextPoint(kDcOptional);
```

```
    GameObject* pObject
        = GameObject::LookUpByName(objectName);
    if (pObject)
    {
        result.SetAsVector(pObject->GetPosition());
    }
    else
    {
        if (pDefaultPos)
        {
            result.SetAsVector(*pDefaultPos);
        }
        else
        {
            DcErrorMessage("get-object-pos: "
                           "Object '%s' not found.\n",
                           objectName.ToDebugString());
        }

    }
    return result;
}
```

Обратите внимание на то, как функция `StringId::ToDebugString()` выполняет обратный поиск, чтобы найти исходную строку по ее идентификатору. Для этого игровой движок должен вести какого-то рода базу данных, связывающую идентификаторы с их строками. Это может сильно упростить жизнь во время разработки, но из-за повышенного потребления памяти такой базы данных не должно быть в конечном продукте (имя функции `ToDebugString()` напоминает, что обратное преобразование идентификатора в строку должно выполняться только в целях отладки — сама игра никогда не должна полагаться на эту возможность!).

Ссылки на игровые объекты

Скриптовым функциям часто нужно взаимодействовать с игровыми объектами, которые сами могут быть частично или полностью реализованы на компилируемом языке движка. Ссылки на объекты, которые используются в компилируемых языках (например, указатели или ссылки в C++), могут не подойти для скриптового кода (последний, к примеру, может вообще не поддерживать указатели). Поэтому нам нужен надежный способ обращения к игровым объектам в скриптовом языке.

Этого можно добиться разными способами. Например, мы можем ссылаться на объекты в скрипте через непрозрачные числовые *дескрипторы*. Дескрипторы объектов в скриптовом коде можно получать по-разному. Они могут передаваться самим движком, или же мы можем выполнять какие-то запросы, чтобы получить их в определенном радиусе от игрока. Также можно находить дескрипторы, соответствующие определенным именам объектов. Затем скрипт может выполнять

операции с этими объектами, передавая их дескрипторы в качестве аргументов скомпилированным функциям. На стороне движка эти дескрипторы опять преобразуются в указатели на игровые объекты, затем с ними производятся какие-то действия.

Числовые дескрипторы отличаются своей простотой, так как их должно быть легко реализовать в любом скриптовом языке, поддерживающем целочисленные данные. Однако они могут быть запутанными и сложными в обращении. В качестве альтернативных дескрипторов можно использовать имена объектов, представленные в виде строк. У этого подхода есть некоторые интересные преимущества по сравнению с числовыми дескрипторами: строки более понятны и с ними легче работать. Они напрямую соответствуют именам объектов в редакторе игрового мира. Кроме того, мы можем зарезервировать некоторые имена, сделать их особенными. Например, в скриптовом языке Naughty Dog зарезервированное имя `self` всегда ссылается на объект, к которому подключен текущий скрипт. Это позволяет игровому дизайнеру написать скрипт, подключить его к объекту в игре, а затем воспроизвести с его помощью анимацию для этого объекта, написав (`animate 'self` *`название_анимации`*).

Конечно, использование строк в качестве дескрипторов объектов имеет и подводные камни. Строки обычно занимают больше памяти, чем целочисленные идентификаторы. И поскольку их длина может варьироваться, для их копирования требуется динамическое выделение памяти. Сравнение строк — медленная операция. Авторы скриптов склонны допускать опечатки при вводе имен игровых объектов, что способно породить ошибки. Кроме того, скриптовый код может перестать работать, если кто-то поменяет имя объекта в редакторе игрового мира, но забудет сделать то же самое в скрипте.

Большинство этих проблем можно решить, преобразовав строки любой длины в целые числа (это называется хешированием). В теории полученный хеш обладает лучшими качествами обоих подходов: он может быть прочитан пользователем, как обычная строка, но его производительность во время выполнения находится на уровне производительности целочисленных идентификаторов. Но чтобы это все заработало, ваш скриптовый язык должен каким-то образом поддерживать применение хешей в этом качестве. В идеале компилятор скрипта должен сам превращать строки в хеши. Таким образом, исполняемому коду вообще не придется иметь дело со строками, разве что в целях отладки: будет удобно, если отладчик сможет показывать строки, относящиеся к хешированным идентификаторам. Однако это реализуется не во всех скриптовых языках. В качестве альтернативы можно разрешить пользователю применять строки в скриптах и затем хешировать их во время выполнения при вызове скомпилированной функции.

Скриптовый язык DC использует для кодирования строковых идентификаторов концепцию символов, позаимствованную из языка программирования Scheme. В DC/Scheme конструкция `'foo` или ее развернутая форма `(quote foo)` соответствует строковому идентификатору `SID("foo")` в C++.

Получение и обработка событий в скриптах

Механизм событий широко применяется в большинстве игровых движков. Разрешение писать обработчики событий на скриптовом языке откроет перед нами богатые возможности модификации жестко закодированного поведения игры.

События обычно передаются отдельным объектам и обрабатываются в их контексте. Поэтому скриптовые обработчики должны быть каким-то образом связаны с объектами. Некоторые движки используют для этого систему типов игровых объектов: скриптовые обработчики можно регистрировать для каждого отдельного типа. Это позволяет разным видам объектов по-разному реагировать на одно и то же событие, но про этом гарантирует, что все экземпляры одного типа имеют единообразную реакцию. Сами обработчики событий могут быть обычными скриптовыми функциями или методами класса, если речь идет об объектно-ориентированном языке. В любом случае им обычно передают дескриптор определенного объекта, которому было послано событие (это действие похоже на то, как методам в C++ передается указатель `this`).

В некоторых движках скриптовые обработчики событий связаны с отдельными экземплярами объектов, а не с их типами. Благодаря этому разные экземпляры одного типа могут по-разному реагировать на одно и то же событие.

Конечно, существует множество других вариантов. Например, в движке Naughty Dog, который использовался в играх из циклов *Uncharted* и *The Last of Us*, скрипты сами по себе являются объектами. Они могут быть связаны с отдельными объектами игры, их можно подключать к регионам — выпуклым многогранникам, с помощью которых инициируются игровые события, но они могут существовать и в качестве полноценных элементов игрового мира. У каждого скрипта может быть несколько состояний, то есть в движке Naughty Dog скрипты играют роль конечных автоматов. Каждое состояние, в свою очередь, может иметь один или несколько блоков кода для обработки событий. Когда игровой объект получает событие, он может обработать его с помощью скомпилированного кода на C++. Но если он находит прикрепленный скриптовый объект, событие передается текущему состоянию скрипта. Если у состояния предусмотрен подходящий обработчик, он будет вызван. В противном случае скрипт просто игнорирует событие.

Отправка событий

Использование скриптов для обработки игровых событий, генерируемых движком, — безусловно, полезная возможность. Но будет еще полезней, если мы сможем генерировать события в скриптовом коде и передавать их либо движку, либо другим скриптам.

В идеале мы должны иметь возможность отправлять из скрипта события совершенно новых типов, а не только определенных заранее. Этого легко добиться, если типы событий представляют собой строки или строковые идентификаторы. Чтобы определить новый тип событий, автору скрипта достаточно выбрать для него подходящее название и указать его в своем коде. Это может стать чрезвычайно гиб-

ким механизмом взаимодействия между скриптами. Скрипт А может определить событие нового типа и послать его скрипту Б. Если у скрипта Б предусмотрен обработчик для событий данного типа, это позволит скрипту А общаться со скриптом Б. В некоторых игровых движках передача событий или сообщений является единственным поддерживаемым средством межобъектного взаимодействия в скриптах. Это может быть элегантным, но в то же время мощным и гибким решением.

Объектно-ориентированные скриптовые языки

Некоторые скриптовые языки являются полностью объектно-ориентированными, другие не поддерживают объекты напрямую, но предоставляют механизмы для реализации классов и объектов. Во многих движках игровой процесс построен на основе объектно-ориентированной модели того или иного вида. Поэтому поддержка какого-то рода ООП в скриптах логична.

Определение классов в скриптах. Класс, в сущности, представляет собой набор данных, связанных с какими-то функциями. Поэтому его можно реализовать в любом скриптовом языке, который позволяет определять новые структуры данных и предоставляет какие-то механизмы для хранения функций и работы с ними. Например, в Lua класс можно соорудить из таблицы, которая хранит свойства и методы.

Наследование в скриптах. Не все объектно-ориентированные языки поддерживают наследование. Однако наличие этой возможности может быть не менее полезным, чем в компилируемых языках программирования вроде C++.

Если говорить об игровых скриптовых языках, может наблюдаться наследование либо от одного скриптового класса к другому, либо от компилируемого класса к скриптовому. Если ваш скриптовый язык объектно-ориентированный, велика вероятность того, что первый вариант поддерживается изначально. Однако реализация второго варианта требует больших усилий даже в скриптовых языках, поддерживающих наследование. Для этого необходимо преодолеть различия между двумя языками и двумя низкоуровневыми объектными моделями. Мы не станем вдаваться в подробности, как это можно реализовать, поскольку важную роль здесь играет то, какие два языка вы пытаетесь интегрировать. UnrealScript — единственный известный мне язык, который позволяет прозрачно наследовать скриптовые классы от компилируемых.

Композиция/агрегация в скриптах. Для расширения иерархии классов вовсе не обязательно полагаться на наследование — для достижения аналогичного результата можно использовать композицию или агрегацию. В этом случае в скрипте нужно лишь иметь возможность определять классы и связывать их экземпляры с объектами, определенными в компилируемом языке программирования. Например, игровой объект может содержать указатель или ссылку на опциональный компонент, полностью написанный на скриптовом языке. Скриптовым компонентам, если они существуют, можно делегировать определенные ключевые обязанности. У них может быть функция `Update()`, которая вызывается при каждом обновлении объекта. Мы также можем позволить компоненту

регистрировать свои методы/функции-члены в качестве обработчиков событий. Когда событие посылается игровому объекту, реализованному на компилируемом языке, тот вызывает из скриптового компонента подходящий обработчик, позволяя тем самым автору скрипта модифицировать или расширять его поведение.

Скриптовый конечный автомат

Многие проблемы в игровом программировании могут исчезнуть сами собой в результате использования конечных автоматов. По этой причине концепция конечных автоматов встроена прямо в основную объектную модель некоторых движков. В таких движках у каждого игрового объекта может быть одно или несколько состояний, и именно в них, а не в самом игровом объекте, содержатся функции обновления, обработки событий и т. д. Простые объекты можно создавать на основе одного состояния, а более сложные имеют возможность определить несколько состояний, каждое со своим подходом к обновлению и реакцией на события.

Если ваш движок поддерживает объектную модель на основе состояний, будет логично обеспечить поддержку конечных автоматов и в скриптовом языке. Но даже если конечные автоматы не поддерживаются в основной объектной модели, вы все равно можете реализовать поведение, основанное на состояниях, на скриптовой стороне. Конечный автомат можно написать на любом языке, описывая состояния в виде классов, но у некоторых скриптовых языков для этого есть отдельный инструментарий. Объектно-ориентированные скриптовые языки могут иметь специальный синтаксис, позволяющий классу содержать несколько состояний. Они также могут предоставлять инструменты, с помощью которых автор скрипта сможет легко агрегировать объекты состояния внутри центрального хаба и затем прозрачным образом делегировать ему функции обновления и обработки событий. Но даже если у вашего скриптового языка нет таких возможностей, вы всегда можете воспользоваться методиками реализации конечных автоматов, придерживаясь заранее оговоренных правил во всех своих скриптах.

Многопоточные скрипты

Часто бывает полезно иметь возможность выполнять несколько скриптов параллельно. Это особенно актуально для современных аппаратных архитектур с высокой степенью параллелизации. Если несколько скриптов могут выполняться одновременно, это означает, что мы предоставляем *параллельные потоки выполнения* в скриптовом коде по аналогии с потоками, доступными в большинстве многозадачных операционных систем. Конечно, на самом деле скрипты могут работать не параллельно — если все они запущены на одном и том же процессоре, их выполнение будет чередоваться. Но с точки зрения автора скрипта это выглядит как парадигма параллельного программирования.

Большинство скриптовых систем, предоставляющих параллелизм, делают это посредством *кооперативной многозадачности*. Это означает, что скрипт выполняется до тех пор, пока сам не уступит ресурсы другому скрипту. Для сравнения:

при использовании *вытесняющей многозадачности* работу любого скрипта можно прервать в любой момент, чтобы выполнить другой скрипт.

Один из простых подходов к кооперативной многозадачности в скриптах состоит в том, чтобы разрешить коду засыпать в ожидании чего-то важного. Скрипт может ждать, пока не пройдет заданное количество секунд или не будет получено определенное событие. Он также может ждать, пока другой поток выполнения не достигнет заранее определенной точки синхронизации. Какой бы ни была причина, при засыпании скрипт попадает в список бездействующих скриптовых потоков и сообщает виртуальной машине о том, что она может приступить к выполнению другого подходящего скрипта. Система отслеживает условия, которые должны разбудить тот или иной спящий скрипт. Когда одно из этих условий выполнится, скрипт или скрипты, которые этого ждали, смогут продолжить выполнение.

Чтобы увидеть, как это работает, рассмотрим пример многопоточного скрипта. Он управляет анимациями двух персонажей и двери. Два персонажа должны подойти к двери, им для этого может понадобиться разное непредсказуемое количество времени. Пока персонажи двигаются к двери, мы приостанавливаем потоки скрипта. Когда оба они дойдут до цели, один из них откроет дверь с помощью соответствующей анимации. Следует отметить, что мы не хотим прописывать продолжительность анимации в сам скрипт. Таким образом, если аниматоры решат ее изменить, нам не придется возвращаться и править код. Когда ожидание анимации завершится, мы снова приостановим потоки. Далее представлен скрипт с простым C-подобным синтаксисом, который все это делает:

```
procedure DoorCinematic()
{
    thread Guy1()
    {
        // Просим guy1 подойти к двери.
        CharacterWalkToPoint(guy1, doorPosition);

        // Засыпаем, пока он туда не доберется.
        WaitUntil(CHARACTER_ARRIVAL);

        // Мы на месте. Сообщим об этом другим
        // потокам с помощью сигнала.
        RaiseSignal("Guy1Arrived");

        // Ждем, пока не появится другой персонаж.
        WaitUntil(SIGNAL, "Guy2Arrived");

        // Теперь просим guy1 воспроизвести анимацию "открыть дверь".
        CharacterAnimate(guy1, "OpenDoor");
        WaitUntil(ANIMATION_DONE);

        // Дверь открыта. Сообщим об этом другим потокам.
        RaiseSignal("DoorOpen");
```

```
        // Теперь пройдем через дверь.
        CharacterWalkToPoint(guy1, beyondDoorPosition);
    }

    thread Guy2()
    {
        // Просим guy2 подойти к двери.
        CharacterWalkToPoint(guy2, doorPosition);

        // Засыпаем, пока он туда не доберется.
        WaitUntil(CHARACTER_ARRIVAL);

        // Мы на месте. Сообщим об этом другим потокам
        // с помощью сигнала.
        RaiseSignal("Guy2Arrived");

        // Ждем, пока не появится другой персонаж.
        WaitUntil(SIGNAL, "Guy1Arrived");

        // Теперь ждем, пока другой персонаж не откроет нам дверь.
        WaitUntil(SIGNAL, "DoorOpen");

        // Дверь открыта. Теперь пройдем через нее.
        CharacterWalkToPoint(guy2, beyondDoorPosition);
    }
}
```

В приведенном примере подразумевается, что наш гипотетический скриптовый язык предоставляет простой синтаксис для определения потоков выполнения в рамках одной функции. Мы определяем два потока: один для `Guy1`, а другой — для `Guy2`.

Поток для `Guy1` просит персонажа подойти к двери и засыпает в ожидании его прибытия. Мы немного утрируем, но давайте представим, что скриптовый язык магически позволяет потоку заснуть и ждать, пока игровой персонаж не прибудет в заданную точку. В реальности, чтобы этого добиться, мы могли бы заставить персонажа отправить скрипту событие, которое пробудило бы поток.

Когда `Guy1` появляется у двери, его поток выполняет два действия, которые заслуживают отдельного объяснения. Сначала он генерирует сигнал `Guy1Arrived`. Затем засыпает в ожидании другого сигнала, `Guy2Arrived`. Если взглянуть на поток для `Guy2`, можно увидеть похожие действия, только в обратном порядке. Этот метод с генерацией одного сигнала и последующим ожиданием другого предназначен для синхронизации двух потоков.

В нашем гипотетическом скриптовом языке *сигнал* представляет собой обычный булев флаг с именем. Изначально флаг равен `false`, но когда поток вызывает `RaiseSignal(`*имя*`)`, его значение меняется на `true`. Другие потоки могут уснуть, ожидая, пока не станет истинным определенный именованный сигнал. Когда это произойдет, спящий поток (или потоки) проснется и продолжит работу. В этом при-

мере для синхронизации друг с другом два потока используют сигналы `Guy1Arrived` и `Guy2Arrived`. Каждый поток генерирует свой сигнал и затем ожидает сигнала другого потока. Неважно, какой сигнал придет первым, — потоки проснутся только тогда, когда сработают оба сигнала. И в этот момент будет достигнута идеальная синхронизация. На рис. 16.24 проиллюстрированы два возможных сценария, в одном из которых первым приходит `Guy1`, а во втором — `Guy2`. Как видите, порядок генерации сигналов не имеет значения — потоки всегда синхронизируются после получения обоих сигналов.

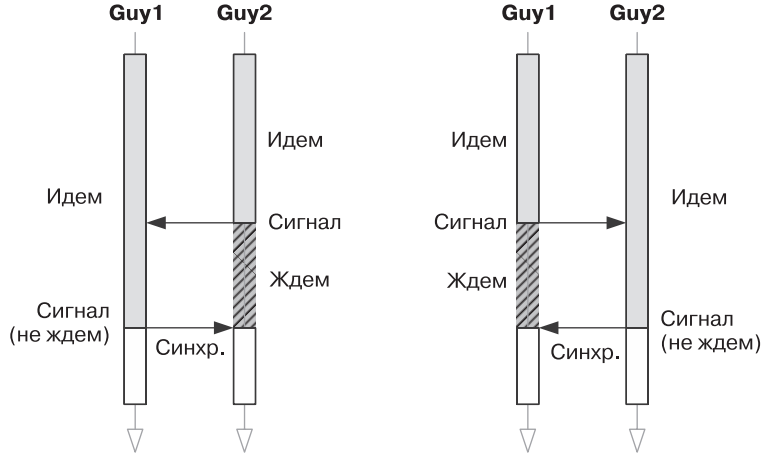


Рис. 16.24. Два примера, демонстрирующие то, как простой подход с генерацией одного сигнала и последующим ожиданием другого можно использовать для синхронизации двух потоков скрипта

16.10. Высокоуровневый игровой поток

Объектная модель игры предоставляет фундамент для реализации богатой и интересной коллекции типов игровых объектов, которыми можно наполнить игровой мир. Однако сама по себе эта модель позволяет описывать разновидности объектов и их поведение только по отдельности. Она ничего не говорит о целях игрока, о том, что произойдет в случае их достижения и какая судьба его постигнет, если он потерпит неудачу.

Для этого нужна система, управляющая высокоуровневым игровым потоком. Ее часто реализуют в виде конечного автомата. Каждое состояние обычно представляет собой отдельные цель или происшествие и относится к определенному месту действия внутри виртуального игрового мира. При выполнении какого-то задания конечный автомат переходит к следующему состоянию, а игроку предъявляется новый набор целей. Конечный автомат также определяет события, которые должны произойти, если игроку не удастся выполнить необходимые задания.

В таких ситуациях игрока часто отправляют в начало текущего состояния, чтобы он мог повторить попытку. Иногда после нескольких неудач у него заканчиваются жизни, и он переходит в главное меню, чтобы начать новую игру. Всем игровым потоком, от меню до первого и последнего игровых уровней, можно управлять с помощью высокоуровневого конечного автомата.

Система заданий, которую мы в Naughty Dog применяли в франшизах *Jak and Daxter*, *Uncharted* и *The Last of Us*, является примером такого подхода на основе конечных автоматов. Она поддерживает линейные последовательности состояний (в Naughty Dog они называются *заданиями*). А также позволяет разделять задание на две параллельные ветви, которые в итоге сливаются обратно в основную последовательность. Такой параллелизм выделяет граф заданий Naughty Dog на фоне обычных конечных автоматов, поскольку последние, как правило, способны находиться одновременно лишь в одном состоянии.

Часть V

Подведение итогов

17 Хотите сказать, что это еще не все?

Поздравляю! Вам удалось завершить путешествие по архитектурам игровых движков целыми и невредимыми. Надеюсь, вас не сильно потрепало и вы сумели узнать многое об основных компонентах, из которых состоит типичный игровой движок. Конечно, там, где заканчивается одно путешествие, начинается другое. Каждая тема, рассмотренная в этой книге, таит в себе много подробностей. С развитием технологий и компьютерного оборудования в играх возникает все больше и больше возможностей, и для их поддержки в движках появляются все новые подсистемы. Основное внимание в этой книге было сосредоточено на самих игровых движках. Мы даже не начали рассматривать захватывающий мир программирования игрового процесса, которому можно было бы посвятить много новых томов.

В следующих коротких разделах мы пройдемся по нескольким подсистемам движка и игрового процесса, которые нам не удалось подробно исследовать, и я дам дополнительный материал для тех, кто хочет узнать о них больше.

17.1. Некоторые подсистемы движка, которые мы не рассмотрели

17.1.1. Видеоплеер

Большинство игр включают в себя видеоплеер для воспроизведения заранее отрисованных роликов, которые еще называют полностью подвижным видео (full-motion video, FMV). Основными компонентами видеоплеера являются интерфейс к системе потокового ввода/вывода файлов (см. подраздел 7.1.3), кодек для декодирования сжатого видеопотока и какой-то механизм синхронизации с аудиосистемой для воспроизведения звуковой дорожки.

Существует множество стандартов и соответствующих кодеков для кодирования видео, каждый из которых подходит для определенного вида задач. Например, в Video CD (VCD) и DVD используются кодеки MPEG-1 и соответственно MPEG-2 (H.262). Стандарты H.261 и H.263 в основном предназначены для видеотелефонии. В играх часто применяются стандарты MPEG-4 part 2 (например, DivX), MPEG-4 Part 10/H.264, Windows Media Video (WMV) или Bink Video

(спецификация, которую компания Rad Game Tools, Inc. разработала специально для игр). Больше информации о видеокодеках можно найти на страницах http://en.wikipedia.org/wiki/Video_codec и http://www.radgametools.com/bnkmain.htm.

17.1.2. Сетевые возможности для многопользовательских игр

Концепции конкурентного программирования, рассмотренные в главе 4, имеют определенное отношение к архитектуре многопользовательских игр и разработке распределенных сетевых приложений, но в этой книге ни одной из этих тем не уделяется достаточно внимания. Для глубокого исследования многопользовательских сетевых возможностей см. [4].

17.2. Системы игрового процесса

Игра далеко не ограничивается одним лишь движком. Помимо основного слоя игрового процесса, рассмотренного в главе 16, существует богатый ассортимент систем, заточенных под определенные жанры и игры. Эти системы объединяют в единое целое многочисленные технологии игрового движка, описанные в этой книге, вдыхая жизнь в игру.

17.2.1. Игровая механика

Игровая механика, безусловно, является самым важным компонентом игрового процесса. Ее общий стиль в сочетании со стилем игрового процесса определяет каждый отдельный жанр, и, конечно, каждая игра в рамках определенного жанра имеет свои архитектурные особенности. Это очень обширная тема, включающая в себя интеграцию HID-систем, симуляцию движения, обнаружение столкновений, анимацию и звук, не говоря уже об интеграции с другими компонентами игрового процесса, такими как игровая камера, оружие, укрытия, механика обхода препятствий (лестниц, свисающих веревок и т. д.), системы управления транспортными средствами, механика решения головоломок и т. п.

Очевидно, что механика игрока так же разнообразна, как и сами игры, поэтому нет такого ресурса, где можно было бы изучить все ее разновидности. Вместо этого лучше исследовать отдельные игровые жанры. Играя, пытайтесь анализировать механику игрока. Затем попробуйте реализовать ее самостоятельно! И в качестве очень скромной отправной точки можно почитать обсуждение механики игрока в платформерах в стиле Mario [9, раздел 4.11].

17.2.2. Камеры

Система камер в игре почти так же важна, как и механика игрока. На самом деле камера может спасти или испортить игровой процесс. У любого жанра обычно есть свой стиль управления камерой, хотя, конечно, каждая игра в рамках одного жанра

делает все немного (а иногда и *совершенно*) иначе. Некоторые базовые приемы управления камерой можно найти в [8, раздел 4.3]. Далее перечислены виды камер, наиболее широко распространенных в трехмерных играх (пожалуйста, имейте в виду, что это далеко не полный список).

- *Камеры осмотра.* Камеры этого типа вращаются вокруг целевой точки и могут быть приближены к ней или удалены от нее.
- *Преследующие камеры.* Камеры этого типа широко применяются в платформерах, шутерах от третьего лица и играх, посвященных транспортным средствам. Они ведут себя подобно камере осмотра, сфокусированной на игровом персонаже, аватаре или транспортном средстве, но обычно немного отстают от игрока. Преследующие камеры имеют дополнительную логику для обнаружения и избегания столкновений, предоставляя игроку некоторую степень контроля за положением камеры относительно его персонажа.
- *Камеры от первого лица.* Камера от первого лица остается привязанной к виртуальным глазам игрового персонажа в ходе его перемещений по игровому миру. Игрок обычно получает полный контроль за направлением камеры с помощью мышки или джойстика. То, куда смотрит камера, напрямую связано с прицелом оружия игрока, которое обычно представлено в виде руки с пистолетом внизу и прицелом в центре экрана.
- *Камеры RTS.* Стратегии в реальном времени и симуляторы бога используют камеру, которая парит над ландшафтом и направлена вниз под углом. Камера может перемещаться по местности, но ее поперечная и вертикальная оси обычно находятся вне прямого контроля игрока.
- *Кинематографические камеры.* В большинстве трехмерных игр есть как минимум несколько кинематографических эпизодов, в которых камера летает по сцене в киношной манере, без привязки к какому-то игровому объекту. Такими движениями камеры обычно управляют аниматоры.

17.2.3. Искусственный интеллект

Еще одним важным компонентом большинства игр на основе персонажей является *искусственный интеллект* (ИИ). В основе системы ИИ обычно лежат такие технологии, как простой поиск пути (в котором часто применяется широко известный алгоритм A*), механизмы восприятия (поле зрения, конусы видимости, знания об окружающей среде и т. д.) и какого-то рода память или опыт.

Поверх этого фундамента создается управляющая логика персонажа. Она определяет, как заставить персонажа выполнить определенные действия, такие как перемещение по местности, навигация по необычному ландшафту, применение оружия, вождение транспортных средств, использование укрытия и т. д. Обычно для этого требуется сложное взаимодействие с системами столкновений, физической симуляции и анимации внутри движка. Управление персонажем подробно обсуждается в разделе 12.10.

Если подняться на уровень выше, у системы ИИ обычно есть логика для определения целей и принятия решений, а также, возможно, моделирования эмоционального состояния, группового поведения (координация, атака с флангов, поведение толпы и стада и т. д.) и в некоторых случаях продвинутые возможности вроде способности учиться на прошлых ошибках или адаптироваться к динамическому окружению.

Конечно, термин «искусственный интеллект» является одним из ярчайших примеров неправильного использования терминов в игровой индустрии. Игровой ИИ больше напоминает набор трюков, чем серьезную попытку симуляции человеческого мышления. У персонажей ИИ могут быть всевозможные эмоциональные состояния и прекрасно настроенное восприятие игрового мира. Но все это будет пустой тратой времени, если игрок не сможет *чувствовать* мотивацию персонажа.

Программирование ИИ — это обширная тема, и мы, безусловно, не уделили ей должного внимания в этой книге. Для получения дополнительной информации см. [8, раздел 3], [9, раздел 3], [18] и [47, раздел 3]. Вы также можете начать с презентации Криса Бутчера и Джейми Гризмера из компании Bungie под названием *The Illusion of Intelligence: The Integration of AI and Level Design in Halo*, которую они представили на GDC 2002 (http://bit.ly/1g7FbhD). И раз уж вы открыли браузер, поищите также «программирование игрового ИИ». Вы найдете ссылки на всевозможные обсуждения, публикации и книги, посвященные игровому ИИ. Вам также могут пригодиться такие ресурсы, как http://aigamedev.com и http://www.gameai.com.

17.2.4. Другие системы игрового процесса

Очевидно, что игра состоит далеко не только из механики игрока, камер и ИИ. В некоторых играх есть транспортные средства, доступные для вождения, специализированные виды оружия, возможность разрушать окружающую среду с помощью динамической физической симуляции, создавать собственных игровых персонажей, строить пользовательские уровни, головоломки... Конечно, список возможностей, относящихся к отдельным жанрам и играм, а также специальные программные системы, которые их реализуют, можно продолжать до бесконечности. Системы игрового процесса такие же насыщенные и разнообразные, как и сами игры. Наверное, вам, как игровому программисту, стоит начать следующее путешествие именно с этого!

Список литературы

1. *Abrash M.* Michael Abrash's Graphics Programming Black Book (Special Ed.). — Scottsdale, AZ: Coriolis Group Books, 1997. (Доступно в Интернете по адресу www.jagregory.com/abrash-black-book.)
2. *Akenine-Moller T., Haines E., Hoffman N.* Real-Time Rendering. 3rd Ed. — Wellesley, MA: A K Peters, 2008.
3. *Alexandrescu A.* Modern C++ Design: Generic Programming and Design Patterns Applied. — Reading, MA: Addison-Wesley, 2001.
4. *Armitage G., Claypool M., Branch P.* Networking and Online Games: Understanding and Engineering Multiplayer Internet Games. — N. Y.: John Wiley and Sons, 2006.
5. *Arvo J.* (editor). Graphics Gems II. — San Diego, CA: Academic Press, 1991.
6. *Bies D. A., Hansen C. H.* Engineering Noise Control. — 4rd Ed. — N. Y.: CRC Press, 2014.
7. *Booch G., Maksimchuk R. A., Engel M. W., Young B. J., Conallen J., Houston K. A.* Object-Oriented Analysis and Design with Applications, 3rd Ed. — Reading, MA: Addison-Wesley, 2007.
8. *DeLoura M.* (editor). Game Programming Gems. — Hingham, MA: Charles River Media, 2000.
9. *DeLoura M.* (editor). Game Programming Gems 2. — Hingham, MA: Charles River Media, 2001.
10. *Dutré P., Bala K., Bekaert P.* Advanced Global Illumination, 2nd Ed. — Wellesley, MA: A K Peters, 2006.
11. *Eberly D. H.* 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics. — San Francisco, CA: Morgan Kaufmann, 2001.
12. *Eberly D. H.* 3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic. — San Francisco, CA: Morgan Kaufmann, 2005.
13. *Eberly D. H.* Game Physics. — San Francisco, CA: Morgan Kaufmann, 2003.
14. *Ericson C.* Real-Time Collision Detection. — San Francisco, CA: Morgan Kaufmann, 2005.
15. *Fernando R.* (editor). GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. — Reading, MA: Addison-Wesley, 2004.

16. *Foley J. D., Dam A. van, Feiner S. K., Hughes J. F.* Computer Graphics: Principles and Practice in C. 2nd Ed. — Reading, MA: Addison-Wesley, 1995.
17. *Fowles G. R., Cassiday G. L.* Analytical Mechanics, 7th Ed. — Pacific Grove, CA: Brooks Cole, 2005.
18. *Funge J. D.* AI for Games and Animation: A Cognitive Modeling Approach. — Wellesley, MA: A K Peters, 1999.
19. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020.
20. *Glassner A. S.* (editor). Graphics Gems I. — San Francisco, CA: Morgan Kaufmann, 1990.
21. *Grama A., Gupta A., Karypis G., George K. G.* Introduction to Parallel Computing. 2nd Ed. — Reading, MA: Addison Wesley, 2003. (Доступно в Интернете по адресу srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf [sic].)
22. *Heckbert P. S.* (editor). Graphics Gems IV. — San Diego, CA: Academic Press, 1994.
23. *Hennessey J. L., Patterson D. A.* Computer Architecture: A Quantitative Approach. — San Francisco, CA: Morgan Kaufmann, 2011.
24. *Herlihy M., Shavit N.* The Art of Multiprocessor Programming. — San Francisco, CA: Morgan Kaufmann, 2008.
25. *Ierusalimschy R., Figueiredo L. H. de, Celes W.* Lua 5.1 Reference Manual. Lua.org, 2006.
26. *Ierusalimschy R.* Programming in Lua, Second Edition. Lua.org, 2006.
27. *Kerlow I. V.* The Art of 3-D Computer Animation and Imaging. 2nd Ed. — N. Y.: John Wiley and Sons, 2000.
28. *Kirk D.* (editor). Graphics Gems III. — San Francisco, CA: Morgan Kaufmann, 1994.
29. *Kodicek D.* Mathematics and Physics for Game Programmers. — Hingham, MA: Charles River Media, 2005.
30. *Koster R.* A Theory of Fun for Game Design. — Phoenix, AZ: Paraglyph, 2004.
31. *Lakos J.* Large-Scale C++ Software Design. — Reading, MA: Addison-Wesley, 1995.
32. *Lengyel E.* Mathematics for 3D Game Programming and Computer Graphics. — 2nd Ed. — Hingham, MA: Charles River Media, 2003.
33. *Little G. B.* Inside the Apple //e. Bowie, MD: Brady Communications Company, Inc., 1985. (Доступно в Интернете по адресу www.apple2scans.net/files/InsidetheIIe.pdf.)
34. *Luong T. V., Lok J. S. H., Taylor D. J., Driscoll K.* Internationalization: Developing Software for Global Markets. — N. Y.: John Wiley & Sons, 1995.
35. *Maguire S.* Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs. — Bellevue, WA: Microsoft Press, 1993.
36. *Meyers S.* Effective C++: 55 Specific Ways to Improve Your Programs and Designs. — 3rd Ed. — Reading, MA: Addison-Wesley, 2005.

37. *Meyers S.* More Effective C++: 35 New Ways to Improve Your Programs and Designs. — Reading, MA: Addison-Wesley, 1996.
38. *Meyers S.* Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library. — Reading, MA: Addison-Wesley, 2001.
39. *Millington I.* Game Physics Engine Development. — San Francisco, CA: Morgan Kaufmann, 2007.
40. *Nguyen H.* (editor). GPU Gems 3. — Reading, MA: Addison-Wesley, 2007.
41. *Oppenheim A. V., Willsky A. S.* Signals and Systems. — Englewood Cliffs, NJ: Prentice-Hall, 1983.
42. *Paeth A. W.* (editor). Graphics Gems V. — San Francisco, CA: Morgan Kaufmann, 1995.
43. *Pilato C. M., Collins-Sussman B., Fitzpatrick B. W.* Version Control with Subversion. 2nd Ed. — Sebastopol, CA: O'Reilly Media, 2008. (Известна также под названием *The Subversion Book*. Доступна в Интернете по адресу svnbook.red-bean.com.)
44. *Pharr M.* (editor). GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. — Reading, MA: Addison-Wesley, 2005.
45. *Stevens R., Raybould D.* The Game Audio Tutorial: A Practical Guide to Sound and Music for Interactive Games. — Burlington, MA: Focal Press, 2011.
46. *Stroustrup B.* The C++ Programming Language, Special Edition. 3rd Ed. — Reading, MA: Addison-Wesley, 2000.
47. *Treglia D.* (editor). Game Programming Gems 3. — Hingham, MA: Charles River Media, 2002.
48. *Bergen G. van den.* Collision Detection in Interactive 3D Environments. — San Francisco, CA: Morgan Kaufmann, 2003.
49. *Watt A.* 3D Computer Graphics, Third Edition. — Reading, MA: Addison Wesley, 1999.
50. *Whitehead J. II, McLemore B., Orlando M.* World of Warcraft Programming: A Guide and Reference for Creating WoW Addons. — N. Y.: John Wiley & Sons, 2008.
51. *Williams R.* The Animator's Survival Kit. — London, UK: Faber & Faber, 2002.

Джейсон Грегори

Игровой движок. Программирование и внутреннее устройство

Третье издание

Перевели с английского О. Сивченко, С. Черников

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Научные редакторы	*К. Рафалюк, А. Саидов*
Литературный редактор	*Н. Рощина*
Художник	*В. Мостипан*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2020. Наименование: книжная продукция. Срок годности: не ограничен.

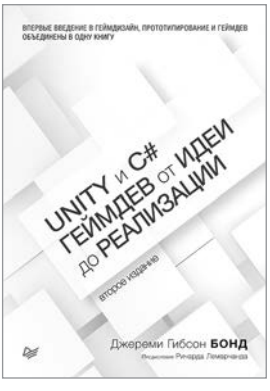
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 14.10.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 91,590. Тираж 700. Заказ 0000.

Джереми Гибсон Бонд

UNITY И C#. ГЕЙМДЕВ ОТ ИДЕИ ДО РЕАЛИЗАЦИИ
2-е издание



Впервые введение в геймдизайн, прототипирование и геймдев объединены в одну книгу. Если вы собираетесь заняться разработкой игр, то в первую очередь вам необходима информация о современных методах и профессиональных инструментах. Эти незаменимые знания можно получить в книге Джереми Гибсона Бонда. Кроссплатформенная разработка Unity позволяет создать игру, а затем с легкостью портировать куда угодно — от Windows и Linux до популярных мобильных платформ.

Начните путешествие в мир игровой индустрии прямо сейчас! Заявите гордо: «Я — геймдизайнер». Ведь если вас услышат другие, то вы будете стараться соответствовать своим словам. А что дальше? Как стать геймдизайнером? Ответы на эти вопросы дает книга Джереми Гибсона Бонда — геймдизайнера и профессора, который больше 10 лет учит других создавать великолепные игры и делает это сам.

Вы погрузитесь в увлекательный мир игровой индустрии, построите 8 реальных прототипов и овладеете всеми необходимыми инструментами.

Джозеф Хокинг

UNITY В ДЕЙСТВИИ. МУЛЬТИПЛАТФОРМЕННАЯ РАЗРАБОТКА НА C#

2-е международное издание

Unity зачастую представляют как набор компонентов, не требующих программирования, что в корне неверно. Для создания успешной игры необходимо многое: великолепная работа художника, программистские навыки, интересная история, увлекательный геймплей, дружная и слаженная работа команды разработчиков. А еще нельзя забывать про безупречную визуализацию и качественную работу на всех платформах — от игровых консолей до мобильных телефонов. Unity объединяет мощный движок, возможности профессионального программирования и творчества дизайнеров, позволяя воплотить в жизнь самые невероятные и амбициозные проекты.

Второе издание знаменитого бестселлера «Unity в действии» было полностью переработано, чтобы познакомить вас с новыми подходами и идеями, позволяющими максимально эффективно использовать Unity для разработки игр. Больше внимания уделено проектированию двумерных игр, фундаментальные концепции которых читатель может применить на практике и построить сложный двумерный платформер. Эту книгу можно смело назвать введением в Unity для профессиональных программистов. Джозеф Хокинг дает людям, имеющим опыт разработки, всю необходимую информацию, которая поможет быстро освоить новый инструмент и приступить к созданию новых игр. А учиться лучше всего на конкретных проектах и практических заданиях.

Осваивайте Unity и быстрее приступайте к созданию собственных игр!

Тайнан Сильвестр

ГЕЙМДИЗАЙН. РЕЦЕПТЫ УСПЕХА ЛУЧШИХ КОМПЬЮТЕРНЫХ ИГР ОТ SUPER MARIO И DOOM ДО ASSASSIN'S CREED И ДАЛЬШЕ

Что такое геймдизайн? Это не код, графика или звук. Это не создание персонажей или раскрашивание игрового поля. Геймдизайн — это симулятор мечты, набор правил, благодаря которым игра оживает.

Как создать игру, которую полюбят, от которой не смогут оторваться? Знаменитый геймдизайнер Тайнан Сильвестр на примере кейсов из самых популярных игр рассказывает, как объединить эмоции и впечатления, игровую механику и мотивацию игроков. Познакомьтесь с принципами дизайна, которыми пользуются ведущие студии мира!

Тайнан Сильвестр занимается геймдизайном больше 15 лет. За это время он успел поработать как над инди-проектами, так и над студийным блокбастером BioShock Infinite, но больше всего он известен благодаря RimWorld.