Данная книга посвящена алгоритмическим основам современной компьютерной графики. Описаны базовые математические понятия, такие как матрицы и кватернионы. Детально рассматривается физика освещения, включая физически-корректное освещение. Отдельные главы посвящены методу трассировки лучей и современному OpenGL. Дается реализация ряда специальных эффектов при помощи шейдеров на языке GLSL.

Весь исходный код доступен в репозитории на github.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
издательство
www.дмк.рф

ISBN 978-5-97060-779-4

Программирование компьютерной графики

Основные понятия и алгоритмы

Современный OpenGL

Алексей Боресков

Программирование компьютерной графики

Метод трассировки лучей

ДМК ПРЕСС
издательство

Алексей Боресков

Программирование компьютерной графики. Современный OpenGL

Москва, 2019

УДК 004.92
ББК 32.973
Б82

Боресков А. В.
Б82 Программирование компьютерной графики. Современный OpenGL. – М.: ДМК Пресс, 2019. – 372 с.: ил.

ISBN 978-5-97060-779-4

Данная книга посвящена основам современной компьютерной графики. Подробно рассматривается ряд чисто математических понятий, таких как матрицы и кватернионы, алгоритмы и API, а также физика освещения. Отдельные главы посвящены методу трассировки лучей и современному OpenGL. Рассматривается реализация ряда специальных эффектов при помощи шейдеров в OpenGL. Весь исходный код доступен в репозитории на github.

Издание будет полезно всем, кто планирует работать с компьютерной графикой.

УДК 004.92
ББК 32.973

ISBN 978-5-97060-779-4

Содержание

Посвящение

Эта книга посвящается памяти профессора Московского университета Евгения Викторовича Шикина. Под его руководством все мы постигали науку на факультете ВМК МГУ, занимались компьютерной графикой, дифференциальной геометрией, решали теоретические и прикладные задачи. Он одним из первых опубликовал в России книгу по компьютерной графике. В своих книгах он всегда уделял особое внимание математическим основам, которые другие авторы часто не считали необходимым приводить.

Евгений Викторович был выдающимся лектором, его лекции всегда были очень интересны и отличались образностью. Его стиль ведения лекций стал для всех нас тем эталоном, к которому мы все стремимся уже в своих лекциях.

Большой заслугой Евгения Викторовича было также и то, что именно благодаря ему дисциплина «Компьютерная графика» стала обязательной учебной дисциплиной сначала на факультете ВМК МГУ, а затем и в ряде других вузов. Евгений Викторович ввел новую форму экзамена для этого предмета – вместо теоретического экзамена предлагалось написать компьютерную программу, реализующую трехмерную динамическую сцену. Оценивалась не только сложность эффектов, которые реализовал автор, но и оригинальность замысла, а также качество программного кода.

Проверка экзаменационных работ превратилась для всех нас в увлекательное занятие, поскольку все работы были разные, а многие студенты создавали настоящие завораживающие виртуальные миры.

Сазонов В. В.
Березин С. Б.
Боресков А. В.

Благодарности

Хочется от всей души поблагодарить всех тех, кто своими замечаниями, советами и отзывами помогли сделать эту книгу лучше и полезнее. Особенную благодарность выражаю коллегам по Cadence Design Systems (как настоящим, так и бывшим) за многочисленные советы и замечания.

Введение

Для того чтобы уметь писать программы, строящие различные изображения (что и является целью компьютерной графики), нужно знать целый ряд понятий и алгоритмов.

Поскольку мы работаем с геометрическими объектами, то нам нужен способ их представления в компьютере, а также способ их преобразования.

Все геометрические объекты обычно представляются при помощи *координат*, т. е. упорядоченных наборов чисел (векторов) с введенными операциями над ними. В главе 1 мы рассмотрим координаты и преобразования в двухмерном случае (на плоскости), а в главе 3 – в трехмерном случае (в пространстве).

Также в главе 3 рассматриваются различные способы задания ориентации объектов в пространстве и задания поворотов. В качестве таких способов выступают *углы Эйлера* и *кватернионы*. Также в этой главе рассматриваются стандартные операции над кватернионами.

Существует большое количество часто используемых объектов и операций над ними как на плоскости, так и в пространстве. Соответственно, главы 2 и 4 рассматривают такие объекты и часто встречающиеся операции над ними на плоскости и в пространстве.

Когда мы имеем дело с большим количеством геометрических данных, то для ускорения работы с ними обычно используются специальные структуры данных, подобно тому как в системах управления базами данных используются различные типы индексов для ускорения операций над данными.

Использование таких структур данных (обычно называемых *пространственными индексами*) позволяет заметно сократить затраты на работу с большими массивами геометрических данных. Подобные структуры, также их построение и использование рассматриваются в главе 5.

Нашей конечной целью является построение изображений, т. е. наборов точек различных цветов. Соответственно, мы должны определиться с тем, что такое цвет и как его можно представить в компьютере. Вся глава 6 посвящена различным моделям для представления цвета, рассматриваются их преимущества для тех или иных задач.

Кроме того, в главе 6 также рассматривается такое понятие, как *гамма-коррекция*, моделирующее преобразование цвета в различных устройствах для отображения и получения изображений (мониторах, камерах и т. п.).

Практически все части современной компьютерной графики относятся к так называемой *растровой графике*, т. е. изображения представляются при помощи прямоугольных массивов точек – пикселов – различных цветов.

Соответственно, возникает задача перевода идеальных геометрических объектов, таких как отрезки, дуги, треугольники, в их растровое представление (процесс, называемый *растеризацией*). Вся глава 7 посвящена этой задаче.

Другой важной задачей, возникающей при построении трехмерных объектов (*рендеринге*), является задача *удаления невидимых поверхностей*. Она связана с тем, что при рендеринге трехмерных объектов мы проектируем их на двухмерную

плоскость. При этом одни объекты могут закрывать (делать полностью или частично невидимыми) другие объекты. Задачей является определить, какие именно объекты из числа имеющихся будут видны и что именно будет видно в каждой части строящегося изображения.

Для решения этой задачи существует большое количество различных методов. В главе 8 дается общий обзор методов и указываются их основные плюсы и минусы.

Обычно мы хотим построить изображения объектов, освещаемых различными источниками света (а не просто висящих в темноте). Для этого нам нужно разобраться с тем, как именно различные объекты взаимодействуют с падающим на них светом.

В главе 9 рассматривается, как можно моделировать это взаимодействие. Для начала рассматриваются самые простые эмпирические модели (например, Ламберта и Фонга). Далее осуществляется переход к строгим физически корректным моделям освещения (например, Кука–Торранса). Также в этой главе рассматриваются играющие большую роль в современной графике *сферические гармоники* и изучается их применение для расчета освещения.

Глава 10 посвящена методу трассировки лучей – простому и красивому методу, позволяющему строить фотореалистичные изображения с точным расчетом таких эффектов, как преломление и отражение света.

Метод трассировки лучей, хотя и дает очень высокое качество получаемых изображений, является довольно медленным. Поэтому во многих случаях (например, симуляторы, обучающие программы, компьютерные игры) требуется *графика реального времени*.

Под этим термином подразумевается возможность построения изображений с достаточно высокой частотой, обычно это не менее 24 кадров в секунду. Для получения графики с такой частотой и достаточно высоким качеством обычно используются программируемые графические процессоры (GPU, Graphics Processing Unit).

Существует много различных GPU, поэтому работа с ними обычно идет не напрямую, а через специальные API (Application Program Interface). Двумя из наиболее распространенных из этих API являются кроссплатформенный OpenGL и ориентированный на платформу Microsoft Window Direct3D. В главе 12 будет рассмотрен OpenGL версии 3.3. В этой главе будет рассмотрен как вывод отдельных примитивов, так и работа с буферами и написание простейших шейдеров (программ, выполняемых на GPU).

Однако сама библиотека OpenGL не занимается созданием окон для рендеринга, обработкой ввода/вывода и т. п. Для этого обычно используют вспомогательные библиотеки. В главе 11 мы рассмотрим сразу две подобные библиотеки – freeglut и GLFW. Также мы рассмотрим библиотеку GLEW, позволяющую получить доступ к различным расширениям OpenGL. Она позволяет получить доступ к многочисленным функциям, которые обычно содержатся в самом драйвере, и к более поздним версиям OpenGL (так, для платформы Microsoft Window библиотека OpenGL32.dll обычно поддерживает OpenGL версии 1.5 или даже ниже).

Одной из неотъемлемых частей современной компьютерной графики являются различные спецэффекты. В главе 13 рассматриваются реализации на OpenGL большого набора различных спецэффектов, начиная с самых простых и заканчи-

вая такими, как рендеринг меха, физически корректный рендеринг, вывод текста при помощи поля расстояний и т. п.

Почти для всех глав имеются многочисленные примеры исходного кода, которые доступны как в репозитории на github – **https://github.com/steps3d/graphicsbook**, так и на сайте автора **http://steps3d.narod.ru**. Этот исходный код использует определенный набор библиотек и инструментов, которые не содержатся в репозитории. Для сборки всех примеров применяется утилита cmake. При ее помощи можно легко создать проекты для компиляции примеров под определенную среду разработки и операционную систему.

Полный список используемых библиотек и инструментов содержится в приложении.

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или вебсайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Координаты и преобразования на плоскости

Координаты являются ключевым понятием, с которым мы будем сталкиваться на протяжении всей книги. Рассмотрение координат и преобразований мы начнем с двухмерного случая (2D). С одной стороны, одномерный случай слишком прост, а с другой – мы будем использовать изложенное для двухмерного случая как основу в более сложном, трехмерном (3D) случае.

Фактически координаты – это механизм, позволяющий вместо работы с геометрическими объектами на самом деле работать с алгебраическими объектами методами алгебры.

Основным нашим понятием будет понятие *координатной плоскости* – это такая плоскость, на которой каждой ее точке P сопоставлена некоторая уникальная пара чисел (x, y), называемая *координатами* этой точки.

Самым простым способом введения координат на плоскости является задание двух координатных осей Ox и Oy. *Координатная ось* – это прямая, на которой обозначены начало отсчета – точка O – и дополнительная точка E. Точка E служит для обозначения направления оси и задания на ней единицы длины. Начало координат O соответствует нулю, а точка E соответствует единице (рис. 1.1) (длина отрезка OE равна единице).

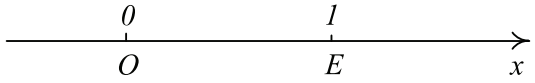

Рис. 1.1 ❖ Координатная ось Ox

В двухмерной системе координат обычно используются сразу две координатные оси с общим началом отсчета – точкой O. При этом чаще всего рассматриваются так называемые *декартовы* (Cartesian[1]) системы координат, в которых координатные оси перпендикулярны друг другу (см. рис. 1.2), но могут быть и недекартовы системы координат.

---

[1] Для многих важных понятий в скобках мы будем давать их английский эквивалент. Это связано с тем, что очень много литературы на компьютерной графике доступно именно на английском языке.

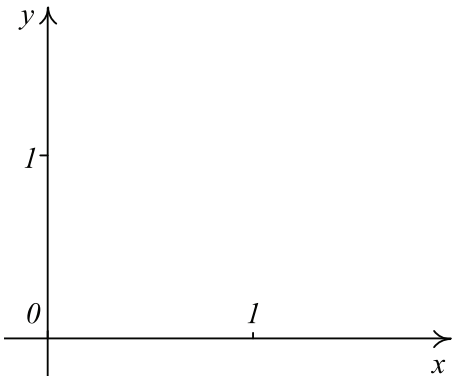

Рис. 1.2 ❖ Декартова система координат на плоскости

В качестве примера *недекартовой* давайте рассмотрим *полярную систему координат*. В этой системе координат используется всего одна ось (точнее, луч, выходящий из начала координат и идущий в положительном направлении). В качестве координат произвольной точки A выступают расстояние r от нее до начала координат O и угол ϕ между положительным направлением оси и лучом OA (рис. 1.3).

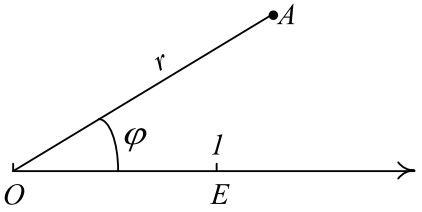

Рис. 1.3 ❖ Полярная система координат

Обратите внимание, что при $r = 0$ угол не определен.

Пусть у нас есть декартова система координат и некоторая точка A. Тогда мы можем получить координаты этой точки, просто спроектировав ее на каждую из координатных осей (вдоль направления другой оси) и взяв соответствующие значения проекций как координаты точки (рис. 1.4).

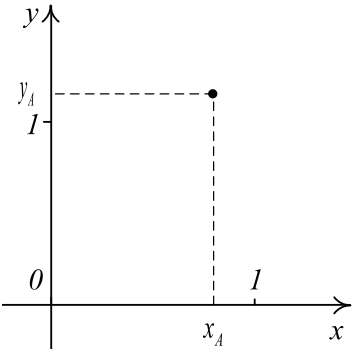

Рис. 1.4 ❖ Декартовы координаты точки A

На одной и той же плоскости мы можем ввести сразу несколько различных систем координат. В результате произвольной точке этой плоскости можно сопоставить сразу несколько различных пар координат (см. рис. 1.5).

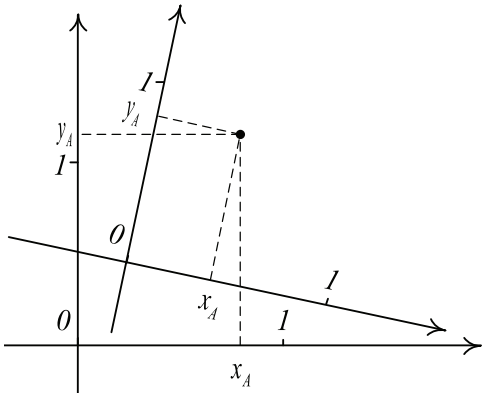

Рис. 1.5 ❖ Различные системы координат на плоскости

При помощи системы координат на плоскости устанавливается *взаимно однозначное* соответствие между множеством всех точек плоскости и множеством пар всевозможных вещественных чисел. Такое множество (пар) мы будем далее обозначать $\mathbb{R}^2$.

Пару чисел x_A и y_A, являющуюся координатами точки A, можно записать либо как строку $(x_A \quad y_A)$, либо как столбец $\begin{pmatrix} x_A \\ y_A \end{pmatrix}$. Всюду далее мы будем записывать координаты в виде столбцов. На самом деле, какая именно запись выбрана, не играет особой роли, обе они полностью эквивалентны, но нам нужно выбрать один какой-то определенный способ. Можно ввести операцию *транспонирования*, которая переводит строку в столбец и наоборот:

$$\begin{pmatrix} x & y \end{pmatrix}^T = \begin{pmatrix} x \\ y \end{pmatrix},$$

$$\begin{pmatrix} x \\ y \end{pmatrix}^T = \begin{pmatrix} x & y \end{pmatrix}.$$

Соответственно, вместо рассмотрения плоскости мы можем на самом деле рассматривать множество $\mathbb{R}^2$ и различные операции на нем. Вместо точек на плоскости мы будем далее рассматривать просто пары чисел, являющиеся координатами точек.

Если у нас есть точка A, то мы можем ввести вектор OA. Координатами этого вектора мы будем называть координаты точки A. Из школьного курса геометрии мы знаем, что векторы можно складывать между собой и умножать на числа. Давайте рассмотрим, что происходит с координатами векторов при их сложении и умножении на различные числа.

Далее тот факт, что некоторый вектор u имеет координаты $\begin{pmatrix} x_u \\ y_u \end{pmatrix}$, мы будем далее записывать следующим образом:

$$u = \begin{pmatrix} x_u \\ y_u \end{pmatrix}.$$

Пусть у нас есть два вектора $u = \begin{pmatrix} x_u \\ y_u \end{pmatrix}$, $v = \begin{pmatrix} x_v \\ y_v \end{pmatrix}$ и некоторое вещественное число α. Тогда легко можно убедиться в том, что сумме векторов u и v соответствует пара чисел, являющаяся покомпонентной суммой координат исходных векторов:

$$u + v = \begin{pmatrix} x_u + x_v \\ y_u + y_v \end{pmatrix}.$$

Аналогично можно убедиться в том, что при умножении вектора u на число α происходит покомпонентное умножение координат вектора u на число α:

$$\alpha \cdot u = \begin{pmatrix} \alpha x_u \\ \alpha y_u \end{pmatrix}.$$

Через O обозначим так называемый *нулевой вектор* $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Тогда мы можем для операций сложения векторов и умножения вектора на число записать следующие свойства:

$u + v = v + u$;

$(u + v) + w = u + (v + w)$;

$u + O = u$;

$1 \cdot u = u$;

$\alpha \cdot (u + v) = \alpha \cdot u + \alpha \cdot v$;

$(\alpha + \beta) \cdot u = \alpha \cdot u + \beta \cdot u$.

Также для произвольного вектора u можно ввести *обратный вектор* $-u$, такой что $u + (-u) = O$. Легко убедиться, что обратный вектор $-u$ всегда существует на самом деле и имеет следующие координаты:

$$-u = \begin{pmatrix} -x_u \\ -y_u \end{pmatrix}.$$

Тем самым для обратного вектора справедливо равенство

$-u = (-1) \cdot u$.

Помимо введенных операций сложения и умножения на числа, часто используется еще одна операция – *скалярное произведение* векторов (*dot product*). Скалярное произведение векторов – это операция, сопоставляющая произвольной паре векторов u и v некоторое вещественное число, обозначаемое (u, v) или $u \cdot v$. Скалярное произведение удовлетворяет следующим свойствам:

$(u,v) = (v,u)$;

$(u+v,w) = (u,w) + (v,w)$;

$(\alpha u,v) = \alpha(u,v)$;

$(u,u) \geq 0$.

При этом скалярный квадрат вектора (u, u) равен нулю тогда и только тогда, когда сам вектор u равен нулевому вектору. Скалярное произведение легко может быть записано через координаты следующим образом:

$(u, v) = u_x v_x + u_y v_y$.

Поскольку для любого вектора u всегда справедливо $(u, u) \geqslant 0$, то можно ввести величину, называемую *нормой*, или *длиной*, вектора и обозначаемую как $||u||$ или $|u|$, следующим образом:

$u = \sqrt{(u,u)}$.

Введенная таким образом норма вектора удовлетворяет следующим основным свойствам:

$||\alpha \cdot u|| = |\alpha| \cdot ||u||$;

$||u + v|| \leqslant ||u|| + ||v||$;

$||u|| = 0$ тогда и только тогда, когда $u = 0$;

$|(u, v)| \leqslant ||u|| \cdot ||v||$.

Два вектора, u и v, называются *ортогональными* (или *перпендикулярными*), если их скалярное произведение равно нулю, т. е. $(u, v) = 0$.

Используя скалярное произведение, можно не только ввести понятие перпендикулярности векторов, но и угол между ними. Угол φ между ненулевыми векторами u и v определяется из следующего равенства:

$$\cos\varphi = \frac{(u,v)}{||u|| \cdot ||v||}.$$

Кроме векторов, важную роль играют и *матрицы*. Матрица 2×2 – это упорядоченная запись четырех чисел a_{11}, a_{12}, a_{21} и a_{22} в виде таблицы из двух строк и двух столбцов:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$

Матрицу, состоящую из одних нулей, называют *нулевой* и обозначают O, а через I обозначают *единичную* матрицу следующего вида:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Элементы матрицы a_{11} и a_{22} называются *главной диагональю* матрицы A. Матрица, у которой все элементы, не лежащие на главной диагонали, равны нулю, называется *диагональной*.

Для матриц 2×2 можно, как и для векторов, ввести операции сложения и умножения на число, определив их покомпонентно:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{pmatrix};$$

$$\alpha \cdot \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} \alpha a_{11} & \alpha a_{12} \\ \alpha a_{21} & \alpha a_{22} \end{pmatrix}.$$

Эти операции удовлетворяют аналогичным свойствам соответствующих операций для векторов. Множество всех матриц 2×2 обозначают через $\mathbb{R}^{2\times 2}$.

Кроме уже введенных операций над матрицами, можно также ввести еще две – транспонирование матриц и перемножение матриц. Обратите внимание, что они не являются поэлементными, в отличие от сложения матриц и умножения матрицы на число. Транспонирование матриц определяется следующим образом:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix}.$$

Матрица C называется *произведением* матриц A и B (и обозначается $C = AB$), если элемент матрицы C с индексами i и j (т. е. элемент c_{ij}) задается следующей формулой:

$$c_{ij} = \sum_{k=1}^{2} a_{ik} b_{kj}.$$

Для операций транспонирования и умножения справедливы следующие свойства:

$(AB)C = A(BC)$;
$A(B + C) = AB + AC$;
$(\alpha A)B = \alpha(AB)$;
$AI = IA = A$;
$AO = OA = O$;
$(AB)^T = B^T A^T$;
$(A + B)^T = A^T + B^T$;
$(\alpha A)^T = \alpha A^T$;
$(A^T)^T = A$.

Обратите внимание, что в общем случае $AB \neq BA$ подобные операции называют *некоммутативными*. В этом легко можно убедиться на следующем примере:

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \neq \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Для матрицы A размером 2×2 также можно ввести понятие *детерминанта* матрицы, обозначаемого $\det A$, используя следующую формулу:

$$\det A = a_{11}a_{22} - a_{12}a_{21}.$$

Матрицы, детерминант которых равен нулю, называются *вырожденными*, все остальные матрицы называются *невырожденными*. Можно показать, что у вырожденной матрицы строки (или столбцы) пропорциональны друг другу.

Если матрица A невырожденная ($\det A \neq 0$), то всегда существует такая матрица B, называемая *обратной* к A, что выполняется равенство

$AB = BA = I$.

Матрица, обратная к матрице A, обозначается как A^{-1}. Для обратных матриц справедливы следующие свойства:

$(A^{-1})^{-1} = A$;

$\det A^{-1} = \dfrac{1}{\det A}$;

$(AB)^{-1} = B^{-1}A^{-1}$;

$(A^T)^{-1} = (A^{-1})^T$.

Для невырожденной матрицы A можно явно выписать ее обратную матрицу:

$$A^{-1} = \frac{1}{\det A}\begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix}.$$

Кроме операции перемножения матриц 2×2, можно также ввести операцию умножения матрицы 2×2 на вектор следующим образом:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{pmatrix}.$$

Для этой операции справедливы следующие свойства:

$(A + B)u = Au + Bu$;
$A(Bu) = (AB)u$;
$A(u + v) = Au + Av$;
$A(\alpha u) = \alpha Au$;
$(Au, v) = (u, A^T v)$.

Преобразования на плоскости

Под *преобразованием* на плоскости понимается любое отображение плоскости на себя $f : \mathbb{R}^2 \to \mathbb{R}^2$. Простейшим примером преобразования является *тождественное* преобразование $f(u) \equiv u$.

Композицией h преобразований f и g называется результат последовательного применения этих преобразований:

$h(u) = (fg)(u) = f(g(u))$.

Введем еще одно понятие – преобразование g называется *обратным* к преобразованию f (и обозначается как $g = f^{-1}$), если для всех векторов u выполнено равенство

$f(g(u)) = g(f(u)) = u$.

Среди всех преобразований на плоскости выделяется очень важный класс *линейных* преобразований. Преобразование f называется линейным, если для любых векторов u и v и любого вещественного числа α выполняются следующие равенства:

$f(u + v) = u + v$;
$f(\alpha u) = \alpha f(u)$.

Примером линейного преобразования может служить любое преобразование вида $f(u) = Au$, где A – произвольная матрица. Можно показать, что для любого линейного преобразования f всегда существует такая матрица A, что $f(u) = Au$ для всех векторов u. В то же время преобразование $f(u) = u + a$ не является линейным при ненулевом векторе a.

Тем самым вместо рассмотрения линейных преобразований на плоскости мы можем рассматривать просто различные матрицы, поскольку каждому линейному преобразованию соответствует преобразование, задаваемое какой-либо матрицей.

Для линейных преобразований обратному преобразованию соответствует обратная матрица, и композиции преобразований соответствует произведение матриц соответствующих преобразований.

Далее мы рассмотрим некоторые стандартные преобразования на плоскости и выпишем соответствующие им матрицы.

Масштабирование

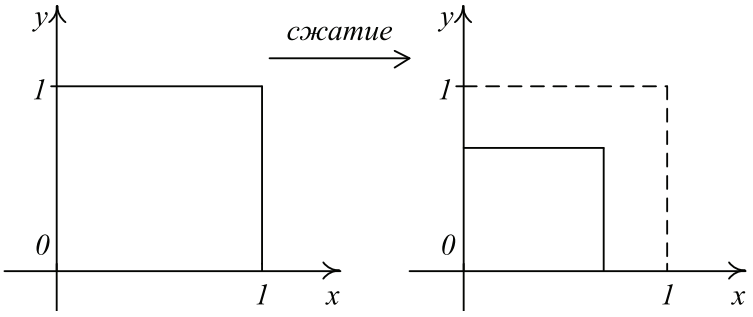

Рис. 1.6 ❖ Преобразование масштабирования

Простейшим линейным преобразованием на плоскости является *однородное масштабирование*. Оно задается при помощи следующей матрицы:

$$S_\lambda = \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}, \lambda > 0.$$

В случае $\lambda > 1$ мы получаем однородное растяжение, а при $0 < \lambda < 1$ – однородное сжатие. Поскольку $\det S_\lambda = \lambda^2 > 0$, то это преобразование всегда невырожденное, обратным к нему также является преобразование однородного масштабирования:

$$S_\lambda^{-1} = \begin{pmatrix} \frac{1}{\lambda} & 0 \\ 0 & \frac{1}{\lambda} \end{pmatrix}.$$

Помимо однородного масштабирования, есть еще и *неоднородное масштабирование*, когда степень растяжения/сжатия отличается для разных направлений (рис. 1.7). Неоднородное масштабирование задается матрицей

$$S_{\lambda\mu} = \begin{pmatrix} \lambda & 0 \\ 0 & \mu \end{pmatrix},\ \lambda > 0,\ \mu > 0.$$

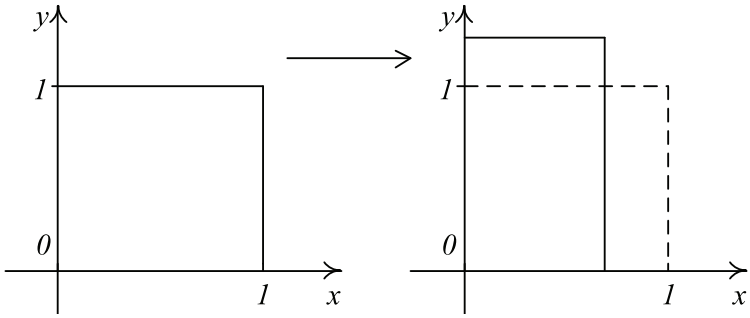

Рис. 1.7 ❖ Неоднородное масштабирование

Неоднородное масштабирование также является невырожденным и поэтому тоже всегда обратимо. Обратным к нему также является преобразование масштабирования со следующей матрицей:

$$S_{\lambda\mu}^{-1} = \begin{pmatrix} \frac{1}{\lambda} & 0 \\ 0 & \frac{1}{\mu} \end{pmatrix}.$$

Отражение

Еще одним простым линейным преобразованием на плоскости является отражение относительно координатной оси (рис. 1.8 и 1.9).

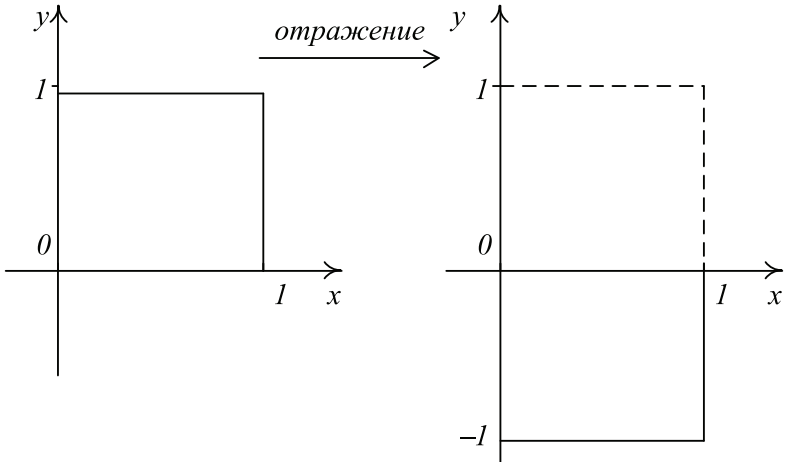

Рис. 1.8 ❖ Отражение относительно оси *Ox*

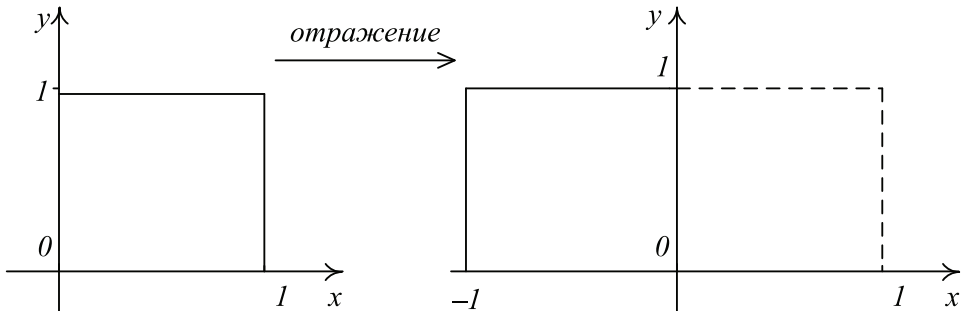

Рис. 1.9 ❖ Отражение относительно оси *Oy*

На плоскости у нас всего две координатные оси, соответственно, мы получаем два отражения – относительно оси *Ox* и относительно оси *Oy*. Соответствующие им матрицы приводятся ниже.

$$R_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix};$$

$$R_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Как и для преобразования масштабирования, соответствующие матрицы являются *диагональными*. Определитель для каждой из приведенных матриц равен –1, тем самым они обратимы, и обратная к матрице отражения совпадает с самой матрицей, т. е. $R_x^{-1} = R_x$, $R_y^{-1} = R_y$.

Поворот

Еще одним распространенным линейным преобразованием является *преобразование поворота* (rotation) на заданный угол φ вокруг начала координат *O* (рис. 1.10) против часовой стрелки.

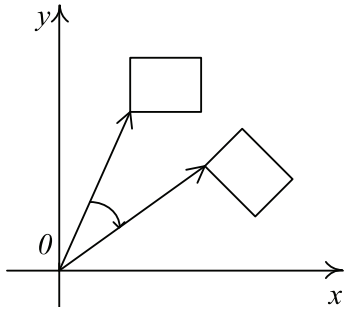

Рис. 1.10 ❖ Преобразование поворота

Это преобразование задается при помощи следующей матрицы:

$$R(\varphi) = \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}.$$

Легко убедиться в том, что для такой матрицы выполнены следующие свойства:

$\det R(\varphi) = 1$;
$R^{-1}(\varphi) = R^{T}(\varphi) = R(-\varphi)$.

Матрицей поворота по часовой стрелке на угол φ будет матрица $R(-\varphi)$.

Сдвиг

Последним линейным преобразованием, которое мы рассмотрим, будем преобразование *сдвига* (shear). В простейшем случае сдвиг может быть или вдоль оси *Ox*, или вдоль оси *Oy* (рис. 1.11).

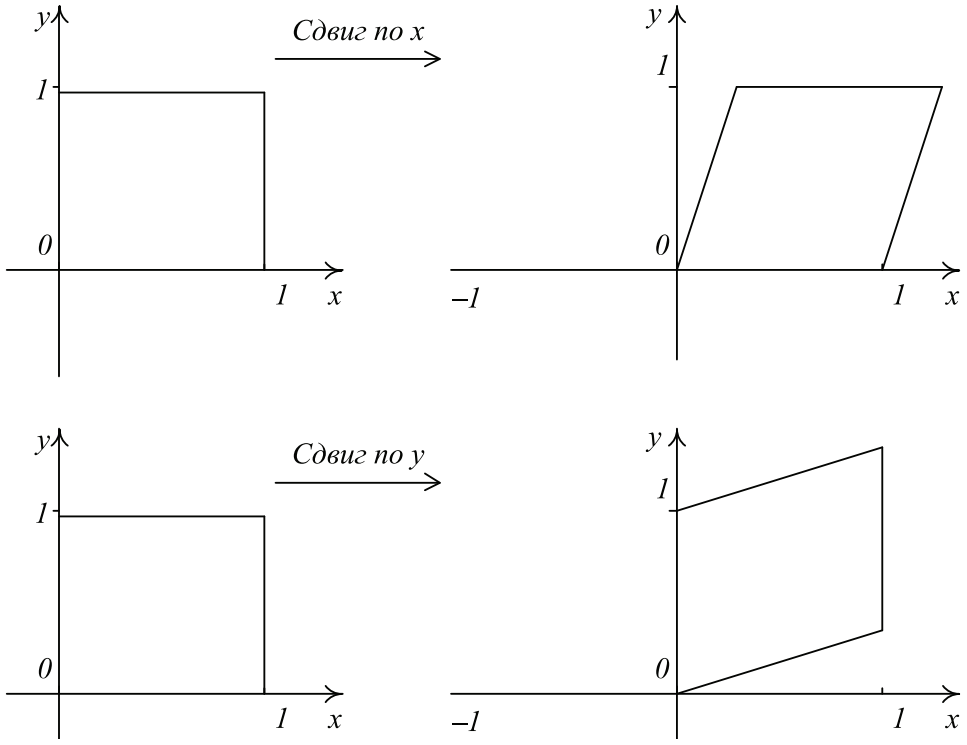

Рис. 1.11 ❖ Простейший случай преобразования сдвига

Матрицы, соответствующие преобразованиям сдвига вдоль осей *Ox* и *Oy*, задаются следующими формулами:

$$H_x(\lambda) = \begin{pmatrix} 1 & \lambda \\ 0 & 1 \end{pmatrix};$$

$$H_y(\lambda) = \begin{pmatrix} 1 & 0 \\ \lambda & 1 \end{pmatrix}.$$

Несложно убедиться в том, что $\det H_x(\lambda) = \det H_y(\lambda) = 1$, поэтому эти матрицы обратимы, и обратные к ним также являются матрицами сдвига:

$$H_x^{-1}(\lambda) = H_x(\lambda) = \begin{pmatrix} 1 & -\lambda \\ 0 & 1 \end{pmatrix};$$

$$H_y^{-1}(\lambda) = H_y(\lambda) = \begin{pmatrix} 1 & 0 \\ -\lambda & 1 \end{pmatrix}.$$

Существует общая форма преобразования сдвига, включающая в себя сдвиг сразу по обоим направлениям. Такое преобразование может быть записано в виде следующей матрицы:

$$H(\lambda,\mu) = \begin{pmatrix} 1 & \lambda \\ \mu & 1 \end{pmatrix}.$$

Обратите внимание, что для такой матрицы должно быть выполнено требование $\lambda\mu \neq 1$, иначе соответствующая матрица будет вырождена.

Составные преобразования

Можно показать, что любое невырожденное линейное преобразование на плоскости может быть представлено в виде произведения (композиции) элементарных преобразований – масштабирования, отражения и поворота.

Для начала давайте покажем, что преобразование поворота на самом деле можно выразить через три последовательных применения преобразования сдвига:

$R(\varphi) = H_x(\lambda)H_y(\mu)H_x(\nu)$.

Заменив все матрицы в этом выражении на свои значения и выполнив перемножение матриц, мы придем к следующему равенству:

$$\begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} = \begin{pmatrix} 1+\lambda\mu & \lambda+\nu+\lambda\mu \\ \mu & 1+\mu\nu \end{pmatrix}.$$

Отсюда можно получить выражения для коэффициентов, выразив их через тригонометрические функции угла φ:

$$\mu = -\sin\varphi;$$

$$\lambda = \nu = \frac{1-\cos\varphi}{\sin\varphi}.$$

Теперь давайте рассмотрим другой пример – преобразование, осуществляющее отражение относительно произвольной прямой, проходящей через начало координат (рис. 1.12).

Такая прямая может быть записана при помощи своего вектора нормали n в следующем виде:

$(p, n) = 0$.

Пусть наша прямая l образует угол φ с осью Ox, тогда ее нормаль n будет образовывать тот же самый угол φ, но уже с осью Oy. Считая, что нормаль n – это единичный вектор, мы можем записать его компоненты через угол φ следующим образом:

$$n_x = \cos\left(\varphi + \frac{\pi}{2}\right) = -\sin\varphi;$$

$$n_y = \cos\varphi.$$

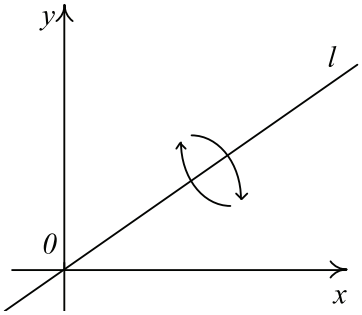

Рис. 1.12 ❖ Отражение относительно произвольной прямой

Тогда если мы выполним поворот на угол φ, то прямая l перейдет в координатную ось Ox. Но мы уже знаем, как осуществлять отражение относительно Ox, поэтому мы выполним это отражение и осуществим еще один поворот – на этот раз на угол $-\varphi$. В результате последнего поворота мы вернем прямую l обратно на свое место. Тем самым требуемое преобразование может быть записано в виде следующей суперпозиции: $R(-\varphi)R_x R(\varphi)$.

Поскольку мы уже знаем синус и косинус угла φ (через компоненты вектора нормали), то мы можем записать результирующую матрицу в следующем виде:

$$R(-\varphi)R_x R(\varphi) = \begin{pmatrix} n_y^2 - n_x^2 & -2n_x n_y \\ -2n_x n_y & n_x^2 - n_y^2 \end{pmatrix}.$$

Использование библиотеки GLM для работы с двухмерными векторами и матрицами

При использовании языка программирования C++ можно «завернуть» векторы и матрицы в классы, переопределив при этом все необходимые операторы. Это позволяет сильно упростить код и повысить его читаемость. Для этого уже есть готовое решение в виде популярной библиотеки `GLM`, которую можно легко скачать по следующему адресу: **http://glm.g-truc.net/**. Далее мы будем постоянно использовать данную библиотеку во всех примерах.

Данная библиотека состоит только из заголовочных файлов. Все вводимые классы и функции помещаются в пространство имен `glm`. Для представления двухмерных векторов используется класс `glm::vec2`, для представления двухмерных матриц – класс `glm::mat2`. Ниже приводится фрагмент кода, показывающий применение данной библиотеки.

```
#define GLM_FORCE_RADIANS           // углы будут задаваться в радианах
#define GLM_SWIZZLE
#include <glm/vec2.hpp>
#include <glm/matrix.hpp>
```

```
#include <glm/mat2x2.hpp>
#include <glm/geometric.hpp>

int main ( int argc, char * argv [] )
{
      glm::vec2 a(1.0f);            // вектор (1,1)
      glm::vec2 b = a + 1.0f;       // вектор (2,2)
      glm::vec2 c(2.0f,-1.0f);      // вектор (2,-1)
      glm::vec2 d = a + 2.0f*c;     // сложение и умножение на число
      glm::vec2 e = -d;
                                          // нормируем, т. е. делим на длину
      glm::vec2 f = glm::normalize(e);
      glm::vec2 a2(a.xy);
      glm::vec2 b2(a.xy());

      float dot2 = glm::dot(glm::vec2(1), d);  // скалярное произведение
      float len  = glm::length(e);             // длина вектора
      float dist = glm::distance(b, c);        // расстояние между векторами
      bool  res  = glm::all(glm::equal(a2, b2));

      glm::mat2 m (0, 1, 2, 3);
      glm::mat2 n  = matrixCompMult(m, m);     // покомпонентное перемножение
      glm::mat2 t  = transpose(m);             // транспонирование
      glm::mat2 mi = inverse(m);               // получаем обратную матрицу

      glm::vec2 u = mi * c + a;                // умножение матрицы на вектор

      return 0;
}
```

Помимо операторов, вводится также набор вспомогательных функций. В примере приведены лишь некоторые из них – `glm::length` (длина вектора), `glm::dot` (скалярное произведение двух векторов), `glm::distance` (расстояние между векторами), `glm::normalize` (получение единичного вектора, сонаправленного заданному), `glm::transpose` (транспонирование матрицы), `glm::invert` (обращение матрицы) и `glm::matrixCompMult` (поэлементное произведение двух матриц).

По мере необходимости мы будем и далее возвращаться к описанию библиотеки `GLM`. Сейчас укажем лишь одно – в этой библиотеке есть ряд функций, принимающих на вход значения угла. Для того чтобы во всех этих функциях угол задавался в радианах (а не в градусах), необходимо перед подключением заголовочных файлов `GLM` определить следующую переменную – `GLM_FORCE_RADIANS`.

В ряде случаев нам будет нужно получить указатель на элементы вектора (или матрицы) как на массив вещественных чисел. Ниже показывается, как это делается через `glm::value_ptr`.

```
glm::vec2 a ( 1, 2 );
glm::mat2 m ( 1, 2, 3, 4 );
float   * p1 = glm::value_ptr ( a );
float   * p2 = glm::value_ptr ( m );
```

Комплексные числа как координаты на плоскости

Помимо широко известных вещественных чисел (множество которых обычно обозначается как $\mathbb{R}$), есть еще и *комплексные числа*, множество которых обозначается как $\mathbb{C}$.

Под комплексным числом $z \in \mathbb{C}$ понимается упорядоченная пара из двух вещественных чисел (x, y). Первое число из этой пары (x) называется *действительной частью z*, а второе число (y) – его *мнимой частью*.

Для обозначения комплексных чисел обычно используется следующая форма записи: число с вещественной частью x и мнимой частью y обозначается как

$$z = x + iy.$$

Через i обозначена так называемая *мнимая единица*, т. е. число, квадрат которого равен –1:

$$i^2 = -1.$$

Обратите внимание, что обычное вещественное число можно рассматривать как комплексное число с равной нулю мнимой частью, т. е. число вида $x + i \cdot 0$.

Для комплексных чисел естественным образом вводятся операции сложения и умножения (получаемые просто раскрытием скобок и перегруппировкой членов):

$$z_1 + z_2 = (x_1 + x_2) + i \cdot (y_1 + y_2);$$
$$z_1 z_2 = (x_1 x_2 - y_1 y_2) + i \cdot (x_1 y_2 + x_2 y_1).$$

Тем самым множество всех комплексных чисел $\mathbb{C}$ можно рассматривать как двухмерное векторное пространство (определение векторного пространства будет дано в главе 3).

Кроме сложения и умножения, для комплексных чисел можно также определить операцию *сопряжения* – сопряженным к комплексному числу z называется следующее комплексное число:

$$\overline{z} = z^* = x - iy.$$

Модулем (абсолютной величиной) комплексного числа z называется величина

$$|z| = \sqrt{z\overline{z}} = \sqrt{x^2 + y^2}.$$

Для комплексных чисел можно ввести экспоненту. Для этого обычно используется формула Эйлера:

$$e^{i\varphi} = \cos\varphi + i \cdot \sin\varphi.$$

Тогда

$$e^{x+iy} = e^x(\cos y + i \cdot \sin y).$$

Кроме того, для комплексных чисел часто используется запись через полярные координаты:

$$z = x + i \cdot y = re^{i\varphi}.$$

Здесь

$r = |z|;$

$\cos\varphi = \frac{x}{|z|};$

$\sin\varphi = \frac{y}{|z|}.$

Таким образом, каждой точке на плоскости можно поставить в соответствие некоторое комплексное число z. При этом $|z|$ – это просто длина соответствующего вектора. Сложению комплексных чисел соответствует сложение векторов, умножению комплексного числа на вещественное соответствует умножение вектора на число.

При этом операция сопряжения – это просто отражение относительно Ox. Также можно очень просто представить операцию поворота на угол φ. Для этого давайте введем следующее единичное комплексное число:

$z_0 = \cos\varphi + i \cdot \sin\varphi.$

Тогда умножение на это число будет соответствовать повороту на угол φ:

$zz_0 = (x + iy) \cdot (\cos\varphi + i\sin\varphi) = (x\cos\varphi - y\sin\varphi) + i \cdot (y\cos\varphi + x\sin\varphi).$

Глава 2

Основные геометрические алгоритмы на плоскости

В этой главе мы рассмотрим некоторые часто встречающиеся алгоритмы для работы с геометрией на плоскости. В их число входят как самые простые (вроде построения прямой по двум точкам), так и довольно сложные алгоритмы (отсечение многоугольника, построение выпуклой оболочки, диаграммы Вороного и др.). Многие из этих алгоритмов сопровождаются примерами кода с использованием классов из библиотеки `GLM`.

В ряде приводимых алгоритмов возникает задача сравнения/классификации. При этом обычно нельзя сравнивать числа с плавающей точкой на точное равенство. Это связано с тем, что расчеты с такими числами чаще всего сопровождаются небольшими погрешностями. Кроме того, некоторые входные данные могут нести погрешности уже на входе.

Поэтому для геометрических сравнений обычно производится операция *регуляризации*, задача которой – обеспечить устойчивость результата при малых шевелениях входных данных. Таким образом, вместо явного сравнения с нулем обычно используется сравнение $|x| < \epsilon$. Если это неравенство выполнено для достаточно малого ϵ, то можно считать, что x равно нулю. Аналогично, проверка на положительность обычно имеет вид $x > \varepsilon$.

В этих проверках используется *абсолютная погрешность*. Но в ряде случаев необходимо также проверять и на относительную погрешность. Пусть надо сравнить два отрезка на совпадение. Обычно для этого просто сравниваются на совпадение концы отрезка. Однако если у нас есть два очень коротких отрезка (с длинами, меньшими ε), то проверка на абсолютную погрешность пройдет всегда (даже если они вообще перпендикулярны).

Однако не все такие отрезки можно считать равными. Поэтому в данном случае имеет смысл дополнить сравнение концов проверкой на *относительную погрешность*. Это значит, что сравнение проводится не с ϵ, а с величиной $\epsilon / \max(l_1, l_2)$. Здесь через l_1 и l_2 обозначены длины сравниваемых отрезков.

Таким образом, в ряде случаев мы приходим к комбинированным проверкам, когда проверяется как абсолютная, так и относительная погрешность.

Кроме того, если мы используем величину ϵ для сравнения длин/расстояний, то для сравнения площадей и объемов нужно уже использовать другие величины. Это связано с тем, что площади и объемы не являются линейными функциями от

размеров. Кроме того, могут быть сравнения углов, для которых также нужна своя допустимая погрешность.

Прямая и ее уравнение

Прямая является одним из самых базовых геометрических объектов. Будем считать, что на плоскости задана декартова система координат. В школьном курсе математики обычно рассматривается следующее уравнение прямой:

$$y = kx + b.$$

Однако это уравнение не подходит для задания вертикальных прямых, да и для задания почти вертикальных прямых оно тоже не очень удобно. Поэтому обычно используется уравнение прямой общего вида (легко убедиться в том, что приведенное выше уравнение является его частным случаем):

$$ax + by + c = 0.$$

В этом уравнении коэффициенты a и b не могут быть равными нулю одновременно, поэтому $a^2 + b^2 > 0$. Тем самым мы имеем полное право разделить это уравнение на $\sqrt{a^2 + b^2}$, получая при этом следующее уравнение:

$$\frac{a}{\sqrt{a^2+b^2}}x + \frac{b}{\sqrt{a^2+b^2}} + \frac{c}{\sqrt{a^2+b^2}} = 0.$$

Первые два члена этого уравнения являются не чем иным, как просто скалярным произведением следующих двух векторов:

$$u = \begin{pmatrix} x \\ y \end{pmatrix};$$

$$n = \begin{pmatrix} \frac{a}{\sqrt{a^2+b^2}} \\ \frac{b}{\sqrt{a^2+b^2}} \end{pmatrix}.$$

Обратите внимание, что $||n|| = 1$. Тем самым мы переписали уравнение прямой в двухмерном случае в следующем виде:

$$(n,u) + d = 0;$$

$$d = \frac{c}{\sqrt{a^2+b^2}}.$$

Здесь вектор n является *вектором нормали* к нашей прямой, а d – расстоянием вдоль этого вектора до начала координат (рис. 2.1). Обратите внимание, что поскольку расстояние берется вдоль вектора нормали, то оно может быть и отрицательным.

Если мы знаем некоторую точку u_0 на этой прямой, то уравнение прямой может быть записано в несколько ином виде:

$$(u - u_0, n) = 0.$$

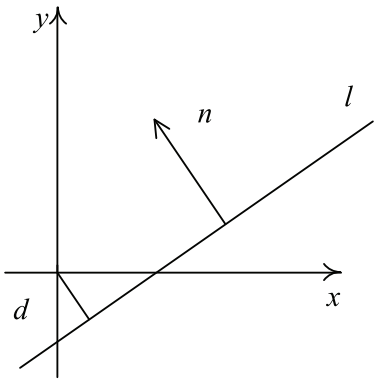

Рис. 2.1 ❖ Прямая на плоскости

Если прямая проходит через начало координат, то $d = 0$ и уравнение принимает вид: $(n, u) = 0$.

Однако уравнение прямой можно записать и в параметрическом виде, привязав каждую точку прямой к значению некоторого параметра t, пробегающему все множество вещественных чисел $\mathbb{R}$:

$$\begin{cases} x = x_0 + At \\ y = y_0 + Bt \end{cases}.$$

Переписав это уравнение в векторном виде, мы получим:

$$u = u_0 + lt.$$

Здесь u_0 – это некоторая точка на прямой (соответствующая значению параметра $t = 0$). Через l обозначен вектор направления (направляющий вектор) для этой прямой, обычно он считается единичным (рис. 2.2). При единичном направляющем векторе параметр t имеет смысл расстояния вдоль данной прямой от точки u_0.

Используя точку на прямой u_0 и направляющий вектор l, мы можем записать уравнение прямой еще в одном виде:

$$\frac{x - x_0}{l_x} = \frac{y - y_0}{l_y}.$$

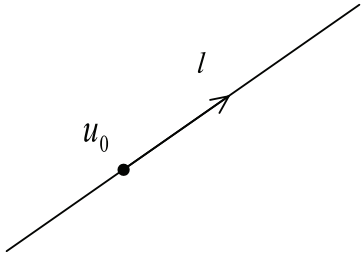

Рис. 2.2 ❖ Прямая

Пусть теперь у нас задан направляющий вектор l и мы хотим по нему построить вектор нормали n к данной прямой. Легко убедиться в том, что вектор n, задаваемый следующей формулой, всегда будет перпендикулярен вектору l и, кроме того, будет иметь ту же длину, что и вектор l.

$$n = \begin{pmatrix} l_y \\ -l_x \end{pmatrix}.$$

Аналогично по вектору нормали всегда можно построить направляющий вектор, перпендикулярный ему.

Прямая разбивает всю плоскость на две полуплоскости – положительную $(n, u) + d > 0$ и отрицательную $(n, u) + d < 0$.

Построение прямой, луча и отрезка по двум точкам

Пусть у нас теперь заданы две точки a и b и мы хотим построить прямую, проходящую через них. При этом мы будем считать, что эти точки не совпадают – только тогда данная прямая определена однозначно. Для получения уравнения данной прямой в параметрическом виде нам достаточно найти точку на этой прямой и ее направляющий вектор. Проще всего в качестве точки на прямой взять точку a, а в качестве направляющего вектора – вектор $l = b - a$. Тогда параметрическое уравнение прямой, проходящей через эти две точки, примет следующий вид:

$$u = a + (b - a)t.$$

Также это уравнение можно записать и в несколько измененном виде:

$$u = (1 - t)a + tb.$$

В случае прямой параметр t может принимать любые значения. Для луча он может принимать только неотрицательные значения. Для отрезка ab параметра t принимает значения лишь из отрезка [0, 1].

Определение положения точки относительно прямой или отрезка

Пусть у нас есть прямая l, задаваемая уравнением $(n, u) + d = 0$, и некоторая точка p. Рассмотрим, как можно определить положение этой точки относительно прямой l. Точка может лежать в полуплоскости, куда смотрит нормаль, в противоположной полуплоскости или же на самой прямой.

Для определения этого достаточно просто подставить координаты точки p в уравнение прямой и посмотреть на знак получающегося значения. Если $(n, p) + d > 0$, то точка p лежит в положительной полуплоскости. Если $(n, p) + d < 0$ – то в отрицательной. И наконец, когда $(n, p) + d = 0$, точка лежит на самой прямой. Довольно часто необходимо сравнивать значение не с 0, а некоторым достаточно малым ϵ.

Часто встречается модификация этой задачи – у нас есть отрезок ab и мы хотим определить положение точки относительно этого отрезка. Отличием от вышерассмотренного случая является то, что когда точка лежит на прямой, мы можем определить, лежит ли она до точки a, после точки b или же между ними (рис. 2.3).

Также полезно выделить случай, когда точка совпадает с одним из концов отрезка. Обратите внимание, что эта проверка делается с заданной точностью ε.

Но при этом данная классификация описывает взаимное расположение заданных объектов (прямой и точки) относительно друг друга и никак не зависит от системы координат. Тем самым между различными случаями существует качественное, а не количественное отличие. Обратите внимание, что данная классификация может быть рассмотрена как разбиение плоскости на набор областей и границ между ними. Далее внутри этих границ выделяются границы меньшей размерности и т. д. Тем самым мы, не теряя общности, можем различать краевые и частные случаи.

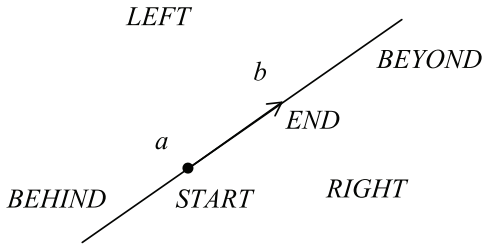

Рис. 2.3 ❖ Классификация точки относительно отрезка

В этом случае нормаль n не определена однозначно. Поэтому удобнее говорить не о том, что точка лежит в положительной или отрицательной полуплоскости, а о том, что точка лежит слева или справа по отношению к направленному отрезку ab.

Для определения того, с какой стороны лежит точка, нам достаточно найти площадь Δabp со знаком. Этот знак и будет определять интересующую нас сторону. Если эта площадь равна 0, то точка p лежит на прямой, проходящей через ab.

Для решения этой задачи можно воспользоваться следующим фрагментом кода:

```
int classify ( const glm::vec2& a, const glm::vec2& b, const glm::vec2& p )
{
      const glm::vec2 a2b  = b - a;       // от a к b
      const glm::vec2 a2p  = p - a;       // от a к p
      float           area = a2b.x*a2p.y - a2p.x*a2b.y;

      if ( area > EPS )
            return LEFT;

      if ( area < -EPS )
            return RIGHT;

      if ( glm::length ( p - a ) < EPS )
            return START;

      if ( glm::length ( p - b ) < EPS )
            return END;

      if ((a2b.x*a2p.x < 0) || (a2b.y*a2p.y < 0))
            return BEHIND;
```

```
      if ( glm::length(a2b) < glm::length(a2p) )
            return BEYOND;

      return BETWEEN;
}
```

Определение положения круга по отношению к прямой

Круг с центром в точке c и радиусом r можно описать при помощи следующего уравнения:

$$||u - c|| \leq r.$$

Пусть, кроме круга, у нас также есть прямая l, заданная уравнением $(n, u) + d = 0$. Будем считать, что вектор нормали n является единичным (если это не так, мы всегда можем разделить уравнение прямой на его длину и прийти к уравнению с единичным вектором нормали).

Тогда в этом уравнении $|d|$ – это расстояние от прямой до начала координат, а для произвольной точки p величина $|(n, p) + d|$ – это расстояние от этой точки до прямой (без знака).

Подставим центр круга в уравнение прямой. Тогда если $(n, c) + d > r$, то круг целиком лежит в положительной полуплоскости. Если $(n, c) + d < -r$, то круг целиком лежит в отрицательной полуплоскости. В противном случае прямая пересекает наш круг (рис. 2.4).

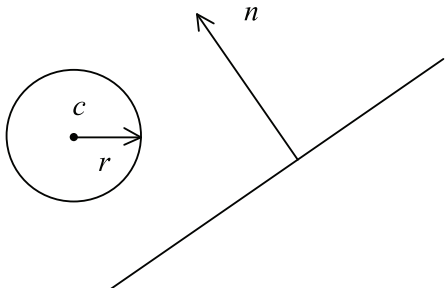

Рис. 2.4 ❖ Классификация круга относительно прямой

Таким образом, мы приходим к следующему фрагменту кода для определения положения круга относительно прямой:

```
int classifyLineCircle ( const glm::vec2& n, float d,
                         const glm::vec2& c, float r )
{
      const float signedDistance = glm::dot ( c, n ) + d;

      if ( signedDistance >= r + EPS )
            return IN_POSITIVE;

      if ( signedDistance <= -r - EPS )
            return IN_NEGATIVE;

      return IN_BOTH;
}
```

Прямоугольники со сторонами, параллельными осям координат, и их классификация по отношению к прямой

Одним из крайне простых и в то же время часто встречающихся объектов является прямоугольник. Из всех прямоугольников чаще всего используются прямоугольники, у которых ребра параллельны осям координат. Такие прямоугольники обычно называются *Axis Aligned Bounding Box* (AABB). Для задания такого прямоугольника достаточно всего четырех чисел: x_{min}, y_{min}, x_{max} и y_{max} (рис. 2.5).

Тогда такой прямоугольник задается следующими неравенствами:

$$\begin{cases} x_{min} \le x \le x_{max} \\ y_{min} \le y \le y_{max} \end{cases}.$$

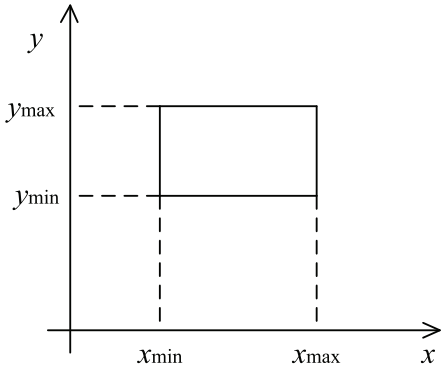

Рис. 2.5 ❖ Пример AABB

Теперь рассмотрим задачу определения положения *AABB* относительно прямой, заданной своим уравнением $(n, u) + d = 0$.

Простейшим решением было бы подставить все четыре вершины *AABB* в уравнение прямой и проверить знаки получающихся значений. Если все знаки совпадают, то объект находится целиком по одну сторону относительно прямой. В противном случае он пересекается прямой.

Однако на самом деле подобную проверку можно осуществить гораздо эффективнее. Для этого обратите внимание, что нормаль к прямой может лежать в одном из четырех квадрантов (рис. 12.6). И в зависимости от того, в каком квадранте лежит вектор нормали, найдется всего одна вершина, которая будет ближе всего вдоль направления нормали (рис. 2.7).

Пусть вектор нормали n имеет координаты n_x и n_y. Тогда координаты ближайшей вершины будут задаваться следующими формулами:

$$x_n = \begin{cases} x_{min}, n_x \ge 0 \\ x_{max}, n_x < 0 \end{cases};$$

$$y_n = \begin{cases} y_{min}, n_y \ge 0 \\ y_{max}, n_y < 0 \end{cases}.$$

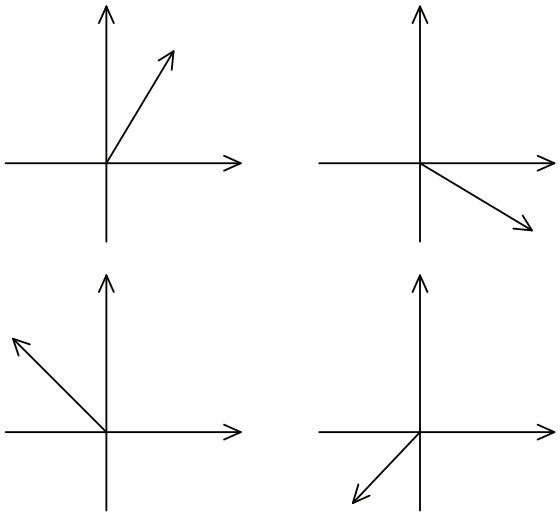

Рис. 2.6 ❖ Возможные варианты направления вектора нормали к прямой

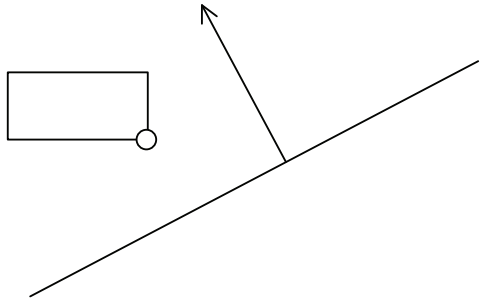

Рис. 2.7 ❖ Выбор ближайшей точки для заданного квадранта

Тогда алгоритм классификации *AABB* относительно прямой проверяет, лежит ли вершина p_n в положительной полуплоскости ($(n, p_n) + d \geq 0$). Если это так, то и весь *AABB* также лежит в положительной полуплоскости.

В противном случае мы находим вершину p_{-n}, лежащую напротив вершины p_n, и проверяем, лежит ли она в отрицательной полуплоскости. Если да, то и весь *AABB* также лежит в отрицательной полуплоскости.

Если же ни одно из этих условий не выполнено, то прямая пересекает AABB.

```
int classifyBox ( const glm::vec2& pMin, const glm::vec2& pMax,
                  const glm::vec2& n, float d )
{
      glm::vec2 pn (n.x >= 0 ? pMin.x : pMax.x, n.y >= 0 ? pMin.y : pMax.y);
      float     f = glm::dot ( n, pn ) + d;

      if ( f > EPS )
            return IN_POSITIVE;

      pn.x = n.x >= 0 ? pMax.x : pMin.x;
```

```
        pn.y = n.y >= 0 ? pMax.y : pMin.y;

        if ( glm::dot ( n, pn ) + d < EPS )
              return IN_BEGATIVE;

        return IN_BOTH;
}
```

Нахождение расстояния от точки до AABB

Пусть у нас есть *AABB* и некоторая точка *p*, от которой необходимо найти расстояние до этого *AABB*. Давайте сначала рассмотрим одномерную задачу – есть отрезок $[x_{min}, x_{max}]$, и нужно найти расстояние до него от точки *x*. Возможны следующие случаи: $x < x_{min}$, $x > x_{max}$ и $x_{min} < x < x_{max}$. В первом случае расстоянием будет $x_{min} - x$, во втором – $x - x_{max}$, и ноль в третьем случае.

Тогда в двухмерном случае нам нужно найти два двухмерных расстояния – от точки *x* до проекции AABB на ось *Ox* – $[x_{min}, x_{max}]$ и от точки *y* до проекции AABB на ось *Oy* – $[y_{min}, y_{max}]$. Обозначим их через Δx и Δy соответственно. Тогда искомое нами расстояние будет задаваться следующей формулой:

$$\sqrt{\Delta x^2 + \Delta y^2}.$$

Определение угла между двумя прямыми

Пусть у нас есть две прямые l_1 и l_2, заданные своими уравнениями $(n_1, u) + d_1 = 0$ и $(n_2, u) + d_2 = 0$ (будем считать нормали единичными). Тогда, для того чтобы найти угол между этими прямыми, нам достаточно найти угол φ между нормалями через их скалярное произведение:

$$\cos\varphi = (n_1, n_2).$$

Если же прямые заданы при помощи параметрических уравнений $u = u_{0,1} + l_1 t$ и $u = u_{0,2} + l_2 t$, то угол между этими прямыми равен углу между направляющими векторами l_1 и l_2 и также находится через их скалярное произведение:

$$\cos\varphi = (l_1, l_2).$$

Вычисление площади треугольника и многоугольника

Важной числовой характеристикой плоской фигуры является ее *площадь*. Проще всего найти площадь прямоугольника – она равна произведению его ширины и высоты. Также легко найти площадь круга – круг радиуса *r* имеет площадь πr^2.

Пожалуй, одной из самых простых фигур является треугольник. Пусть у нас есть треугольник, заданный своими вершинами – *a*, *b* и *c* (рис. 2.8).

Если через l_a, l_b и l_c обозначить длины сторон этого треугольника, то его площадь может быть выражена по формуле Герона:

$$S = \sqrt{p(p - l_a)(p - l_b)(p - l_c)}.$$

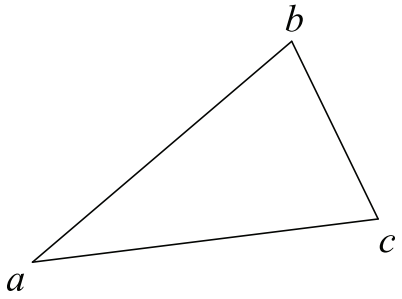

Рис. 2.8 ❖ Треугольник, заданный своими вершинами

В этой формуле через p обозначен полупериметр треугольника, задаваемый следующей формулой:

$$p = \frac{l_a + l_b + l_c}{2}.$$

Однако это не самый эффективный способ нахождения площади треугольника. С вычислительной точки зрения очень легко площадь может быть найдена через определитель специальной матрицы:

$$S = \frac{1}{2}\left|\det\begin{pmatrix} b_x - a_x & b_y - a_y \\ c_x - a_x & c_y - a_y \end{pmatrix}\right|.$$

Обратите внимание на использование модуля – знак этого определителя может быть как положительным, так и отрицательным в зависимости от того, в каком именно порядке заданы вершины треугольника – по часовой стрелке или против нее.

Рассмотрим теперь задачу вычисления площади плоского многоугольника без самопересечений. Будем считать, что есть некоторый многоугольник P, заданный своими вершинами $p_0, p_1, \ldots, p_{n-1}$. Тогда можно показать, что каким бы мы не выбрали вектор a, площадь P будет задаваться следующей формулой:

$$S = \frac{1}{2}\left|\sum_{i=0}^{n-2}\det\begin{pmatrix} x_i - a_x & y_i - a_y \\ x_{i+1} - a_x & y_{i+1} - a_y \end{pmatrix} + \det\begin{pmatrix} x_{n-1} - a_x & y_{n-1} - a_y \\ x_0 - a_x & y_0 - a_y \end{pmatrix}\right|.$$

Здесь через x_i и y_i обозначены компоненты i-й вершины многоугольника. В качестве вектора a проще всего взять нулевой вектор, в результате чего мы приходим к следующей формуле:

$$S = \frac{1}{2}\left|\sum_{i=0}^{n-2}\det\begin{pmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{pmatrix} + \det\begin{pmatrix} x_{n-1} & y_{n-1} \\ x_0 & y_0 \end{pmatrix}\right|.$$

Соответственно, процедуру вычисления площади многоугольника по его вершинам можно записать в следующем виде:

```
float signedArea ( const glm::vec2 p [], int n )
{
    float sum = p [0].x   * (p [1].y - p [n-1].y) +
                p [n-1].x * (p [0].y - p [n-2].y);

    for ( int i = 1; i < n - 1; i++ )
        sum += p [i].x * (p [i+1].y - p [i-1].y);

    return 0.5f * sum;
}
```

Определение направления обхода многоугольника

Пусть у нас есть некоторый многоугольник P, заданный своими вершинами p_0, p_1, ..., p_{n-1} (рис. 2.9). Будем считать, что этот многоугольник не является самопересекающимся.

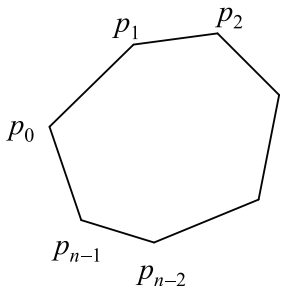

Рис. 2.9 ❖ Многоугольник на плоскости

Вершины, задающие этот многоугольник, могут быть заданы как в направлении по часовой стрелке, так и против. И иногда возникает необходимость определить, в каком именно направлении они заданы.

Для определения этого воспользуемся тем, что площадь многоугольника со знаком (рассмотренная ранее) зависит от направления обхода вершин – при смене направления обхода на противоположное знак меняется.

Тем самым для определения направления обхода вершин нам достаточно просто найти площадь многоугольника со знаком и проверить знак найденной площади. Это реализуется приводимой ниже функцией.

```
bool isClockwise ( const glm::vec2 p [], int n )
{
      return signedArea ( p, n ) > 0;
}
```

Проверка многоугольника на выпуклость

Произвольный многоугольник P называется *выпуклым* (convex), если для любых принадлежащих ему точек a и b весь отрезок ab целиком содержится внутри него (рис. 2.10).

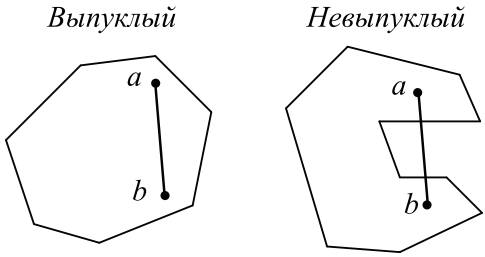

Рис. 2.10 ❖ Выпуклый и невыпуклый многоугольники

Есть и другая формулировка определения выпуклости: для каждого ребра все вершины многоугольника лежат по одну сторону от проходящей через это ребро прямой (рис. 2.11).

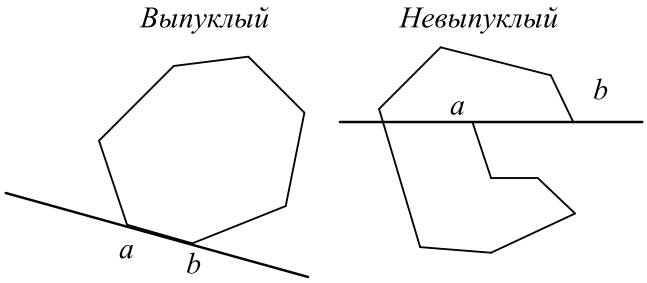

Рис. 2.11 ❖ Второе определение выпуклости

Используя это определение, мы можем легко написать функцию, которая будет проверять выпуклость многоугольника, просто перебирая по очереди все его ребра и проверяя, лежат ли остальные его вершины по одну сторону от прямой, проходящей через выбранное ребро.

```
bool isConvex ( const glm::vec2 v [], int n )
{
    glm::vec2 prev = v [n-1];
    glm::vec2 cur, dir, n;
    float     d, dp;
    int       sign, s;

    for ( int i = 0; i < n; i++ )
    {
        cur  = v [i];
        dir  = cur - prev;                  // вектор направления ребра
        n    = glm::vec2 ( dir.y, -dir.x ); // нормаль к ребру  [prev, cur]
        d    = - glm::dot ( n, cur );       // уравнение прямой (p,n) + d = 0
        sign = 0;                           // пока неизвестен

        for ( int j = 0; j < n; j++ )
        {
            dp = d + glm::dot ( n, v [j] );
```

```
            if ( fabs ( dp ) < EPS )        // слишком мал
               continue;

            s = dp > 0 ? 1 : -1;            // знак для v [j]

            if ( sign == 0 )
               sign = s;
            else
            if ( sign != s )
                return false;
         }
      }

      return true;
}
```

Нахождение пересечения двух прямых

Часто возникает следующая задача: пусть есть две прямые l_1 и l_2, и нужно найти точку их пересечения (или установить, что пересечения нет). Рассмотрим, как это можно сделать.

Пусть прямые заданы своими уравнениями: $(n_i, u) + d_i = 0$, $i = 0, 1$. Тогда для нахождения их пересечения нужно решить следующую *систему алгебраических уравнений* (СЛАУ):

$$\begin{pmatrix} n_{1x} & n_{1y} \\ n_{2x} & n_{2y} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Эта система будет иметь (и притом единственное) решение тогда и только тогда, когда определитель матрицы данной системы не равен нулю. Найти это решение можно, воспользовавшись *правилом Крамера*.

$$x_0 = \frac{1}{\Delta} \det \begin{pmatrix} -d_1 & n_{1y} \\ -d_2 & n_{2y} \end{pmatrix};$$

$$y_0 = \frac{1}{\Delta} \det \begin{pmatrix} n_{1x} & -d_1 \\ n_{2x} & -d_2 \end{pmatrix};$$

$$\Delta = \det \begin{pmatrix} n_{1x} & n_{1y} \\ n_{2x} & n_{2y} \end{pmatrix}.$$

В случае когда определитель системы равен нулю, наши прямые параллельные и возможны два случая – они или совпадают, или вообще не пересекаются. Случаю совпадения прямых соответствуют два варианта: $n_1 = n_2$ и $d_1 = d_2$ или $n_1 = -n_2$ и $d_1 = -d_2$.

В результате мы приходим к следующему коду:

```
bool doLinesCoincide ( const glm::vec2& n1, float d1,
                       const glm::vec2& n2, float d2 )
{
      if ( glm::dot ( n1 , n2 ) > 0 ) // считаем нормали единичными
```

```
            return fabs ( d1 - d2 ) < EPS;
      else
            return fabs ( d1 + d2 ) < EPS;
}

bool findLineIntersection ( const glm::vec2& n1, float d1,
                            const glm::vec2& n2, float d2,
                            glm::vec2& p )
{
      const float     det = n1.x * n2.y - n1.y * n2.x;

      if ( fabs ( det ) < EPS )
            return doLinesCoincide ( n1, d1, n2, d2 );

      p.x = (d2*n1.y - d1*n2.y)/det;
      p.y = (d1*n2.x - d2*n1.x)/det;

      return true;
}
```

Нахождение пересечения двух отрезков

К рассмотренной ранее задаче нахождения пересечения двух прямых очень близко находится задача нахождения пересечения двух отрезков.

Пусть у нас заданы два отрезка a_1b_1 и a_2b_2 и мы хотим проверить, пересекаются ли эти отрезки, и в случае пересечения найти точку их пересечения.

Первым нашим шагом будет построение прямых l_1 и l_2, проходящих через эти отрезки. Далее мы находим точку пересечения этих прямых и проверяем, принадлежит ли она каждому из этих отрезков (рис. 2.12).

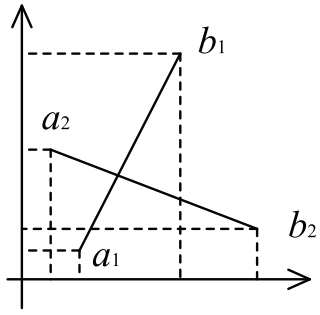

Рис. 2.12 ❖ Пересечение двух отрезков

С учетом всего этого можно написать следующий код:

```
bool findSegIntersection ( const glm::vec2& a1, const glm::vec2& b1,
                           const glm::vec2& a1, const glm::vec2& b2,
                           glm::vec2& p )
{
      const glm::vec2 l1 = b1 - a1;
      const glm::vec2 l2 = b2 - a2;
```

```
    const glm::vec2 n1 ( l1.y, -l1.x );
    const glm::vec2 n2 ( l2.y, -l2.x );
    const float     d1 = -glm::dot ( a1, n1 );
    const float     d2 = -glm::dot ( a2, n2 );

        // при помощи определителя проверяем
        // на параллельность
    const float t1 = n1.x*n2.y - n1.y*n2.x;
    const float t2 = n1.x*d2   - n2.x*d1;

    if ( fabs ( t1 ) < EPS && fabs ( t2 ) < EPS )
    {
        if ( fabs ( d1.x ) > EPS )   // проектируем на Ox
            return min( a1.x , a2.x ) <= min( b1.x , b2.x );
        else
        if ( fabs ( d1 . y ) > EPS ) // проектируем на Oy
            return min( a1.y , a2.y ) <= min( b1.y , b2.y );

        return false;     // ошибка
    }
                          // найдем пересечение
    const float     det = n1.x * n2.y - n1.y * n2.x;
    const glm::vec2 p0 ( d2*n1.y - d1*n2.y , d1*n2.x - d2*n1.x )/det;

    return min( a1.x, b1.x ) <= p0.x && p0.x <= max( a1.x, b1.x ) &&
           min( a1.y, b1.y ) <= p0.y && p0.y <= max( a1.y, b1.y );
}
```

Нахождение расстояния и ближайшей точки от заданной точки к прямой, лучу и отрезку

Пусть у нас есть прямая l и некоторая точка p, не лежащая на этой прямой (рис. 2.13). Тогда существует единственная точка q, которая является ближайшей к p точкой прямой l. При этом отрезок pq будет перпендикулярен прямой l.

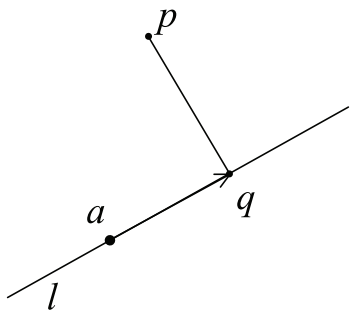

Рис. 2.13 ❖ Ближайшая точка на прямой

Пусть прямая l задается параметрическим уравнением $a + l \cdot t$. Тогда обозначим через $f(t)$ квадрат расстояния от точки $a + l \cdot t$ до точки p:

$$f(t) = ||a + l \cdot t - p||^2 = (a - p + l \cdot t, a - p + l \cdot t).$$

Воспользовавшись свойствами скалярного произведения, получаем:

$f(t) = ||a - p||^2 + 2(a - p, l)t + ||l||^2 t^2$.

Свое минимальное значение эта функция принимает при $t = t^*$, где

$$t^* = \frac{(a-p,l)}{\|l\|^2}.$$

Тогда ближайшая точка будет равна $q = a + l \cdot t^*$.

Пусть теперь вместо прямой мы имеем луч, который начинается в точке a и идет вдоль положительного направления t. Тогда мы, как и ранее, находим t^*. В случае если $t^* \geq 0$, ближайшей точкой будет точка, соответствующая этому значению параметра. В случае $t^* < 0$ ближайшей точкой будет точка a.

Последним случаем будет нахождение точки, ближайшей к заданному отрезку ab. Построим прямую, проходящую через этот отрезок, воспользовавшись следующим параметрическим уравнением:

$p = a + t(b - a)$.

Как и ранее, находим t^* (в качестве вектора l будет выступать вектор $b - a$). Если $t^* < 0$, то ближайшей будет точка a. Если $t^* > 1$, то ближайшей будет точка b. Иначе ближайшей будет точка $a + (b - a)t^*$.

```
float pointToSegDistance ( const glm::vec2& p, const glm::vec2& a,
                           const glm::vec2& b, float& t )
{
      const glm::vec2 delta = b - a;
      const glm::vec2 diff  = p - delta;

      if ( (t = glm::dot ( a, diff ) > 0 )
      {
            const float dt = glm::length(delta);

            if ( t < dt )
                  t    /= dt;
            else
                  t     = 1;
      }
      else
            t = 0;

      return glm::length(a + t*delta - p);
}
```

Проверка на попадание точки внутрь многоугольника

Пусть у нас задан многоугольник P без самопересечений и некоторая точка a и мы хотим проверить, лежит ли эта точка внутри многоугольника.

Стандартный способ проверки принадлежности точки многоугольнику основан на использовании *теоремы Жордана*. Согласно этой теореме, если мы выпустим из точки a произвольный луч, число пересечений луча с границей многоу-

гольника будет четным, если точка лежит вне многоугольника, и нечетным – если внутри (рис. 2.14).

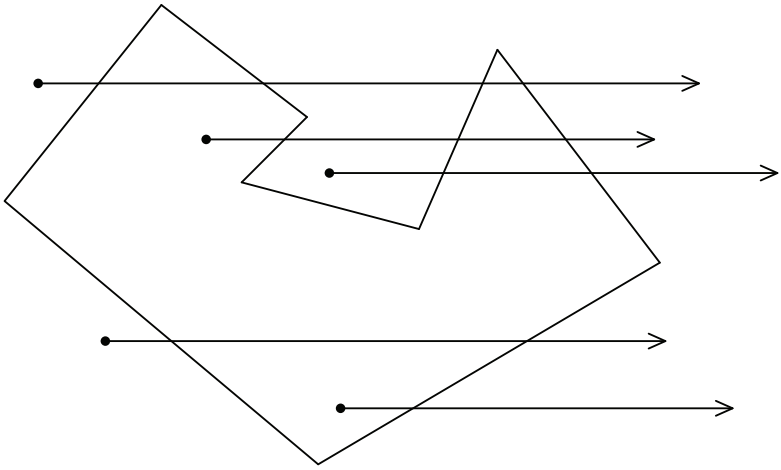

Рис. 2.14 ❖ Проверка на попадание точки внутрь многоугольника

Обычно из точки выпускается горизонтальный луч вправо. Однако при подсчете числа пересечений необходимо учесть те случаи, когда луч проходит через вершину или ребра многоугольника (рис. 2.15).

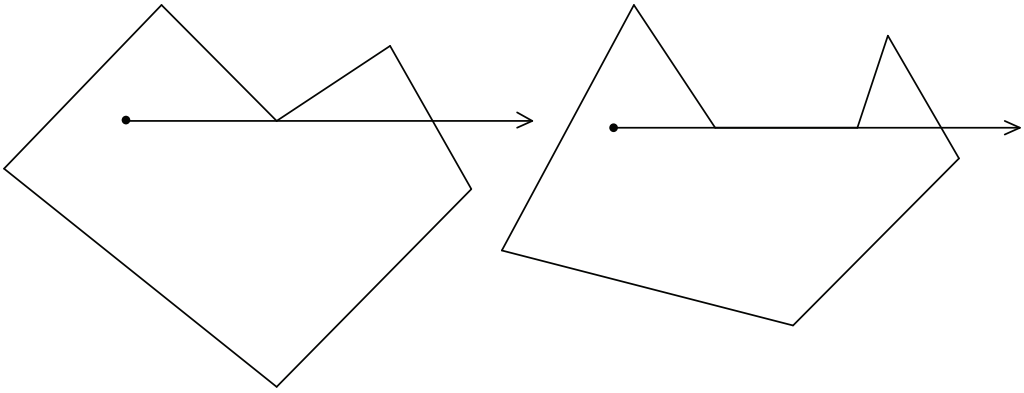

Рис. 2.15 ❖ Особые случаи при подсчете числа пересечений

Обычно в таких случаях считается, что вершина, лежащая на луче, находится немного выше луча. Это позволяет избежать ряда неприятных ситуаций. Ниже приводится код, который, используя данный алгоритм, проверяет, лежит ли точка внутри многоугольника, заданного массивом своих вершин.

```
bool ptInPoly( const glm::vec2 p [], int n, const glm::vec2& pt )
{
    int c = 0;
```

```
    for ( int i = 0, j = n - 1; i < n; j = i++ )
    {
        if ((((p.y [i] <= pt.y) && (pt.y < p.y [j])) ||
            ((p.y [j] <= pt.y) && (pt.y < p.y [i]))) &&
            (pt.x < (p.x [j]-p.x [i])*(pt.y-p.y [i])/
            (pt.p.y [j]-p.y [i])+p.x [i]))
               c = !c;
    }

    return c != 0;
}
```

Отсечение отрезка по выпуклому многоугольнику. Алгоритм Цируса – Бека

Одной из распространенных задач является отсечение заданного отрезка по выпуклому многоугольнику, заданному набором своих вершин $p_0, p_1, \ldots, p_{n-1}$.

Для начала запишем исходный отрезок ab в параметрическом виде:

$p(t) = a + (b - a)t, 0 \leq t \leq 1.$

Теперь рассмотрим произвольное ребро многоугольника $p_i p_{i+1}$. Для этого ребра можно ввести *внешнюю нормаль* n_i, направленную от многоугольника (рис. 2.16).

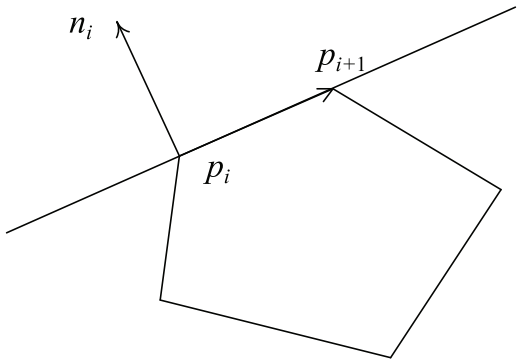

Рис. 2.16 ❖ Внешняя нормаль к ребру многоугольника

Тогда прямая, проходящая через отрезок $p_i p_{i+1}$, будет описываться уравнением $(p - p_i, n_i) = 0$. Эта прямая разбивает всю плоскость на две полуплоскости, и сам многоугольник будет целиком лежать в полуплоскости $(p - p_i, n_i) \leq 0$ в силу определения выпуклости и того факта, что нормаль внешняя.

Зафиксируем некоторое значение параметра t и рассмотрим, в какой именно полуплоскости лежит соответствующая ему точка отрезка $p(t)$. Для этого надо просто найти знак скалярного произведения $(p(t) - p_i, n_i)$.

Если $(p(t) - p_i, n_i) > 0$, то $p(t)$ лежит в той полуплоскости, где многоугольника заведомо нет, и поэтому вся часть отрезка ab до этого значения t должна быть отброшена (рис. 2.17, слева).

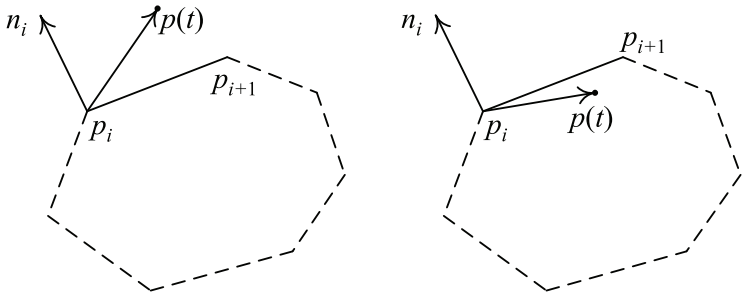

Рис. 2.17 ❖ Классификация $p(t)$ относительно отрезка $p_i p_{i+1}$

Очевидно, что граничной точкой, отделяющей часть отрезка ab, которую надо отсечь, является точка пересечения отрезков ab и $p_i p_{i+1}$. Для ее нахождения можно просто подставить уравнение отрезка $p(t) = a + t(b - a)$ в уравнение прямой, проходящей через отрезок $p_i p_{i+1}$. Из получающегося уравнения легко можно найти значение параметра t, соответствующее точке пересечения:

$$t = \frac{(p_i - a, n_i)}{(b - a, n_i)}.$$

Если $(b - a, n_i) \neq 0$, то мы находим точку пересечения t, которая на самом деле может лежать вне отрезка ab (если $t < 0$ или $t > 0$).

Случай $(b - a, n_i) = 0$ соответствует параллельным отрезкам ab и $p_i p_{i+1}$. Тогда можно использовать знак скалярного произведения $(p_i - a, n_i)$, для того чтобы понять, находится ли точка a внутри многоугольника или снаружи. Если же $(p_i - a, n_i) > 0$ (рис. 2.18 слева), то точка a (и вместе с ней весь отрезок ab) лежит вне полигона и весь отрезок должен быть отброшен.

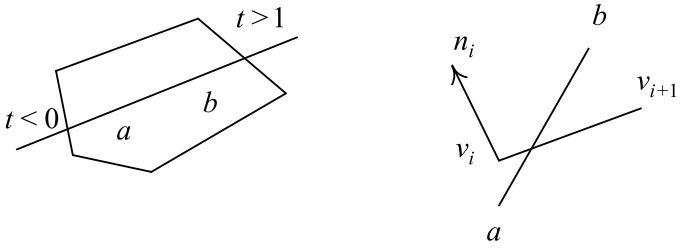

Рис. 2.18 ❖ Случай параллельных отрезков ab и $p_i p_{i+1}$

Рассмотрим более подробно невырожденный случай $(b - a, n_i) \neq 0$. Для него мы находим параметр t точки пересечения. Если этот параметр не лежит на отрезке $[0, 1]$, то нас он не интересует. В противном случае мы можем разделить все такие точки пересечения на два класса – точки «входа» и точки «выхода».

Пусть мы нашли некоторое значение t из отрезка $[0, 1]$. Тогда если $(b - a, n_i) > 0$, то мы будем считать, что соответствующая этому значению t точка $a + (b - a)t$ является точкой «выхода», т. е. при увеличении t мы двигаемся «от» многоугольника. В противном случае мы будем считать эту точку точкой «входа» (рис. 2.19).

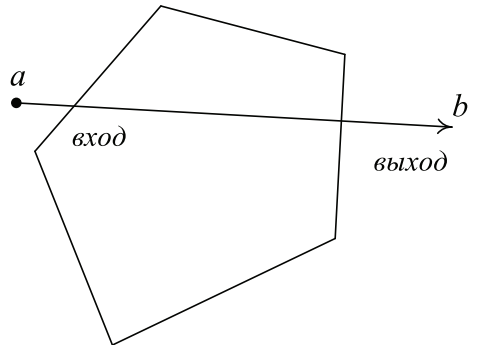

Рис. 2.19 ❖ Точки входа и выхода

Перебрав таким образом все ребра исходного многоугольника, мы получим два множества значений параметра t – входные и выходные. Но поскольку многоугольник является выпуклым, у него не более одного «настоящего» входного значения t_1 и одного «настоящего» выходного значения t_2. Найти эти «настоящие» значения довольно просто: t_1 – это максимальное входное значение, а t_2 – минимальное выходное.

Если мы получили в результате $t_1 > t_2$, то это значит, что весь отрезок ab целиком лежит вне нашего многоугольника и должен быть отброшен целиком. В противном случае остается часть отрезка, соответствующая значениям t от t_1 до t_2.

Полученный нами в результате алгоритм называется *алгоритмом Цируса–Бека* (Cyrus–Beck). Перед тем как перейти к коду, реализующему данный алгоритм, давайте рассмотрим, каким образом мы можем по заданному отрезку $p_i p_{i+1}$ построить внешнюю нормаль n_i. Ниже приводится простая формула, дающая нормаль к заданному отрезку, но это не обязательно внешняя нормаль:

$$n_i' = \begin{pmatrix} p_{i,y} - p_{i+1,y} \\ p_{i+1,x} - p_{i,x} \end{pmatrix}.$$

Для того чтобы проверить, является ли найденный вектор n_i' внешней нормалью, давайте найдем знак следующего выражения $(p_j - p_i, n_i')$, где j – некоторый индекс, кроме i и $i + 1$. Если это скалярное произведение отрицательно, то n_i' – действительно внешняя нормаль. В противном случае внешняя нормаль получается из n_i' умножением на -1.

В результате мы приходим к следующему коду:

```
glm::vec2 outerNormal ( const glm::vec2 p [], int n , int i )
{
      const int i1 = (i+1) % n;
      const int j  = (i+2) % n;
      glm::vec2 n ( p[i].y - p[i1].y, p[i1].x - p[i].y );

      if ( glm::dot ( p[j] - p[i], n ) > 0 )
            n = -n;

      return n;
}
```

```
bool clipSegByConvexPoly ( const glm::vec2 p [], int n,
                           glm::vec2& a, glm::vec2& b )
{
      const glm::vec2 d    = b - a;
      float           tIn  = -INFINITY;
      float                tOut = INFINITY;

      for ( int j = n - 1, i = 0; i < n; i++ )
      {
            const glm::vec2 n  = outerNormal ( p, n, j );
            const float          dn = glm::dot ( d, n );

            if ( fabs ( dn ) < EPS )      // отрезки параллельны
            {
                  if ( glm::dot ( p [j] - a, n ) > 0 )
                        return false;

                  continue;
            }

            const float      pn = glm::dot ( p[j] - a, n );
            const float      t  = pn / dn;

            if ( t > 0 )
                  tOut = min ( tOut, t );
            else
                  tIn = max ( tIn, t );

            if ( tOut <= tIn )
                  return false;
      }

      tIn  = max ( tIn, 0 );
      tOut = min ( tOut, 1 );

      if ( tOut <= tIn )
            return false;

      b = a + tOut * d;
      a = a + tIn  * d;

      return true;
}
```

Алгоритм отсечения Лянга–Барского

Одним из наиболее распространенных случаев отсечения отрезка является отсечение отрезка по *AABB*. В этом случае можно сильно оптимизировать алгоритм Цируса–Бека и прийти к *алгоритму Лянга–Барского* (Liang–Barsky). Ниже приводится код для него.

```
bool clipSegByRect ( const glm::vec2& rMin, const glm::vec2& rMax,
                     glm::vec2& a, glm::vec2& b )
{
       float      p [4], q [4];
       glm::vec2      d = b - a;
       float      t1 = 0, t2 = 0;

       p [0] = -d.x;
       p [1] = d.x;
       p [2] = -d.y;
       p [3] = d.y;
       q [0] = a.x - rMin.x;
       q [1] = rMax.x - a.x;
       q [2] = a.y - rMin.y;
       q [3] = rMax.y - a.y;

       for ( int i = 0; i < 4; i++ )
       {
            if ( fabs ( p [i] ) < EPS )      // отрезки параллельны
            {
                 if ( q [i] >= 0 )
                 {
                      if ( i < 2 )
                      {
                           if ( a.y < rMin.y )
                                a.y = rMin.y;

                          if ( b.y > rMax.y )
                                b.y = rMax.y;
                      }

                      if ( i > 1 )
                      {
                           if ( a.x < rMin.x )
                                a.x = rMin.x;

                           if ( b.x > rMax.x )
                                b.x = rMax.x;
                      }
                 }
            }
       }

      for ( int i = 0; i < 4; i++ )
      {
            if ( fabs ( p [i] ) < EPS )
                 continue;

            float      temp = q [i] / p [i];

            if ( p [i] < 0 )
            {
```

```
                if ( t1 <= temp )
                      t1 = temp;
          }
          else
          {
                if ( t2 > temp )
                      t2 = temp;
          }
     }


     if ( t1 >= t2 )
          return false;

     b = a + t2 * d;
     a = a + t1 * d;

     return true;
}
```

Отсечение многоугольника. Алгоритм Сазерленда–Ходжмана

Еще одной распространенной двухмерной задачей является разбиение заданного многоугольника прямой (рис. 2.20).

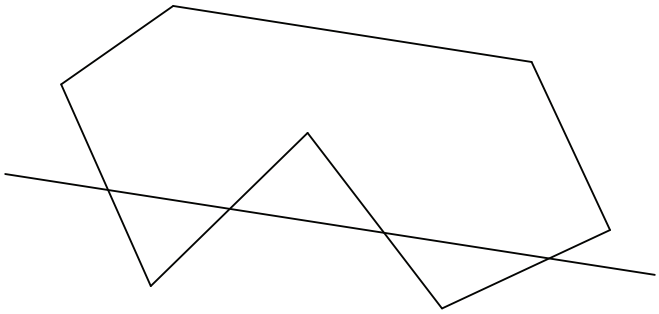

Рис. 2.20 ❖ Разбиение многоугольника прямой

Пусть у нас есть многоугольник P, заданный массивом своих вершин p_0, p_1, …, p_{n-1}, и есть некоторая прямая l, заданная своим уравнением $(p, n) + d = 0$. Задачей является определение частей P, лежащих в положительной полуплоскости $(p, n) + d > 0$. Обратите внимание, что в результате этой операции мы можем получить не один многоугольник, а несколько.

Применяемый для решения этой задачи *алгоритм Сазерленда–Ходжмана* довольно прост. Мы по очереди обрабатываем каждое ребро многоугольника. Пусть текущим обрабатываемым ребром является ребро sq, при этом мы будем считать, что вершина s уже была обработана при обработке предыдущего ребра. Мы будем использовать список выходных вершин.

При обработке мы проверяем каждую из вершин ребра на попадание в положительную полуплоскость. При этом возможны всего четыре различных случая (рис. 2.21):

- обе вершины лежат внутри положительной полуплоскости, тогда вершина *q* добавляется в выходной список;
- вершина *s* лежит внутри, а вершина *q* – снаружи. Тогда мы добавляем точку *i* пересечения ребра с прямой в выходной список;
- обе вершины *s* и *q* лежат снаружи, в выходной список не добавляется ничего;
- *s* снаружи, а *q* внутри. Тогда добавляем точку *i* пересечения ребра и прямой в выходной список.

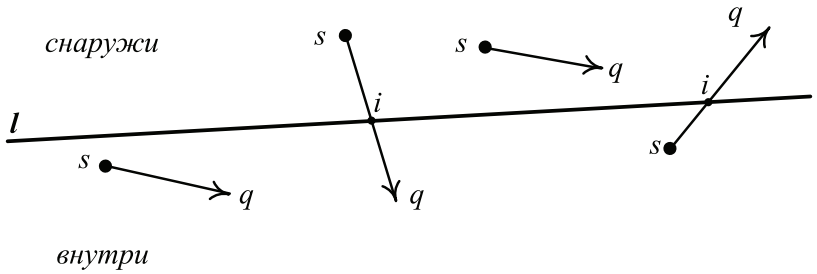

Рис. 2.21 ❖ Возможные положения текущего ребра относительно разбивающей прямой

После обработки всех ребер исходного многоугольника мы получим список вершин, задающих результат. Но при этом результат может состоять из нескольких многоугольников (рис. 2.22).

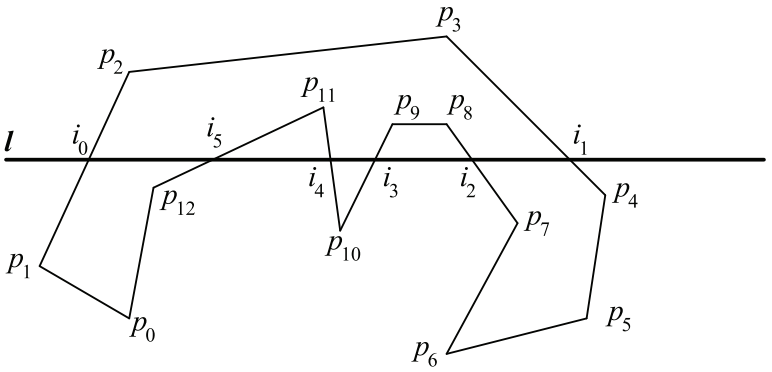

Рис. 2.22 ❖ Результат отсечения может состоять из нескольких многоугольников

Однако при этом основой всех выходящих многоугольников будут отрезки разбивающей прямой *l*, находящиеся внутри многоугольника. Поэтому можно взять все выходные вершины, лежащие на *l*, и отсортировать вдоль *l*. Тогда в результате мы получим набор отрезков на *l*, и каждый из этих отрезков будет определять свой многоугольник.

Отсечение многоугольника по выпуклому многоугольнику

Пусть у нас есть некоторый (не обязательно выпуклый) многоугольник P и нам нужно отсечь его по выпуклому многоугольнику Q. Для этого заметим, что любой выпуклый многоугольник может быть представлен как пересечение полуплоскостей, образованных прямыми, проходящими через его ребра.

Тогда, используя то, что $(A \cap B) \cap C = A \cap (B \cap C)$, мы можем представить отсечение многоугольника P по многоугольнику Q как результат применения ряда последовательных отсечений по прямым, проходящим через ребра многоугольника Q. Операция отсечения многоугольника по прямой была описана выше.

Барицентрические координаты

Пусть у нас есть треугольник, заданный своими вершинами a, b и c. Тогда этот треугольник является *выпуклой оболочкой* этих точек. Это значит, что для любой точки p из этого треугольника всегда найдутся три числа u, v и w такие, что

$p = a \cdot u + v \cdot b + w \cdot c$;
$u, v, w \geq 0$;
$u + v + w = 1$.

Эти три числа u, v и w называются *барицентрическими координатами* точки p. При этом сами вершины треугольника будут иметь координаты $(1 \quad 0 \quad 0)^T$, $(0 \quad 1 \quad 0)^T$ и $(0 \quad 0 \quad 1)^T$. Если точка p лежит строго внутри треугольника, то все компоненты ее барицентрических координат будут принадлежать интервалу (0, 1). У точек на ребрах одна из трех координат будет равной нулю.

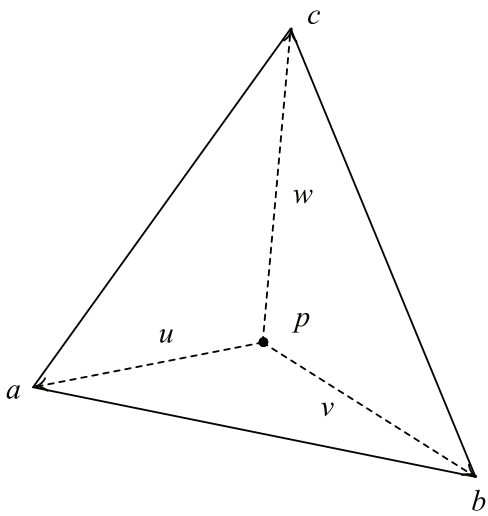

Рис. 2.23 ❖ Барицентрические координаты треугольника

Барицентрические координаты имеют ряд удобных свойств. Одним из них является возможность быстрой проверки точки на принадлежность треугольнику. Хотя для этой цели и можно использовать общий алгоритм проверки принадлеж-

ности точки многоугольнику, для треугольника можно реализовать гораздо более эффективный алгоритм.

Пусть у нас есть точка p. Перейдем от ее декартовых координат $(x \quad y)^T$ к барицентрическим координатам u, v и w. Так как $u + v + w = 1$, то мы можем выразить w через первые две координаты $w = 1 - u - v$. Обратите внимание, что мы для любой точки можем найти тройку барицентрических координат, но не всегда они будут лежать на отрезке [0, 1] – только если точка лежит внутри треугольника.

Тогда мы получим следующее уравнение для получения барицентрических координат:

$$p = c + u \cdot (a - c) + v(b - c).$$

Умножая это уравнение скалярно на $a - c$ и $b - c$, мы получим следующую систему линейных алгебраических уравнений:

$$\begin{cases} u(a-c,a-c)+v(a-c,b-c)=(p-c,a-c) \\ u(a-c,b-c)+v(b-c,b-c)=(p-c,b-c) \end{cases}.$$

Если исходный треугольник не вырожден, то определитель этой системы d не равен нулю:

$$d = (a - c, a - c)(b - c, b - c) - (a - c, b - c)^2.$$

В этом случае мы можем записать решение системы по правилу Крамера:

$$u = \frac{1}{d}\big((p-c,a-c)(b-c,b-c)-(p-c,b-c)(a-c,b-c)\big);$$

$$v = \frac{1}{d}\big((p-c,a-c)(a-c,b-c)-(p-c,b-c)(a-c,a-c)\big).$$

Таким образом, для того чтобы проверить, лежит ли точка p внутри треугольника abc, мы сперва вычисляем скалярные произведения $(a - c, a - c)$, $(a - c, b - c)$ и $(b - c, b - c)$. Далее мы вычисляем определитель при помощи этих скалярных произведений и находим первую координату u. Если она не лежит на [0, 1], то точка p гарантированно не лежит внутри треугольника abc.

Иначе мы вычисляем вторую координату v и проверяем, лежит ли она на [0, 1]. Если нет, то опять точка не может лежать внутри треугольника. Последняя проверка – лежит ли $u + v$ на [0, 1]. Если нет, то точка не лежит внутри треугольника. Если все три проверки успешно пройдены, то точка лежит внутри abc.

```
bool pointInTriangle ( const glm::vec2& a, const glm::vec2& b,
                       const glm::vec2& c, glm::vec2& p )
{
      const glm::vec2 a1 = a - c;
      const glm::vec2 b1 = b - c;
      const float     aa = glm::dot(a1, a1);
      const float     ab = glm::dot(a1, b1);
      const float     bb = glm::dot(b1, b1);
      const float     d  = aa*bb - ab*ab;

      if ( fabs ( d ) < EPS )
            return false;
```

```
    const glm::vec2 p1 = p - c;
    const float     pa = glm::dot(p1, a1);
    const float     pb = glm::dot(p1, b1);
    const float     u  = (pa*bb - pb*ab) / d;

    if ( u < 0 || u > 1 )
        return false;

    const float     v = (pa*ab - pb*aa) / d;

    if ( v < 0 || v > 1 )
        return false;

    return u + v <= 1;
}
```

Построение выпуклой оболочки, алгоритм Грэхема

Пусть у нас есть некоторый набор точек $S = \{p_0, p_1, \ldots, p_{n-1}\}$. Тогда для этого набора точек можно ввести понятие его *выпуклой оболочки* $\mathcal{CU}(S)$, являющейся объединением всех треугольников, построенных на точках множества S (рис. 2.24).

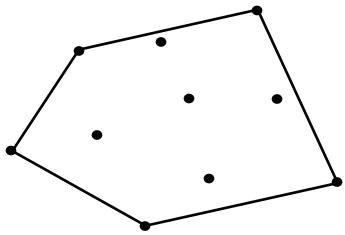

Рис. 2.24 ❖ Выпуклая оболочка множества точек

Рассмотрим, как можно построить выпуклую оболочку по заданному набору точек S. Одним из простых, но в то же время достаточно эффективных методов для построения выпуклой оболочки является *алгоритм Грэхема*.

Первым шагом этого алгоритма является выбор точки p_0, обладающей минимальной y-координатой. Если в множестве S есть несколько таких точек, то мы выбираем ту, которая обладает минимальной x-координатой.

После этого отсортируем все оставшиеся точки множества S по значению угла, между лучом из точки p_0 и положительным направлением оси Ox. Если у двух точек соответствующие углы равны, то меньшей считается точка, которая ближе к p_0.

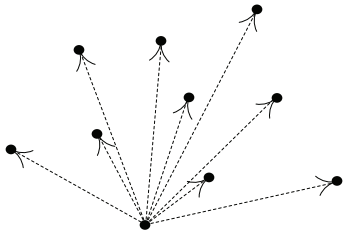

Рис. 2.25 ❖ Сортировка точек для алгоритма Грэхема

Для подобной сортировки совсем не обязательно переходить к полярным координатам и считать значения углов, достаточно использовать любую монотонную функцию от угла. Ниже приводится пример такой функции.

```
int orientation ( const glm::vec2& p, const glm::vec2&q,
                  const glm::vec2& r )
{
      const float val = (q.y - p.y)*(r.x - q.x) - (q.x - p.x)*(r.y - q.y);

      if ( fabs(val) < EPS )
            return COLLINEAR;

      return val > 0 ? CW : CCW;
}

int grahamCompare ( const glm::vec2& p1, const glm::vec2& p2 )
{
      const int orient = orientation ( p0, p1, p2 );

      if ( orient == COLLINEAR )
            return glm::distance (p0, p2) >= glm::distance (p0, p1) ? -1 : 1;

      return orient == CCW ? -1 : 1;
}
```

Дальше мы переименуем все оставшиеся точки в соответствии с порядком их сортировки как $p_1, p_2, \ldots, p_{n-1}$. После этого мы перебираем по очереди все эти точки, текущая выпуклая оболочка будет храниться на стеке. На каждом шаге мы рассматриваем очередную точку p_i и берем две точки s_j и s_{j-1} с вершины стека.

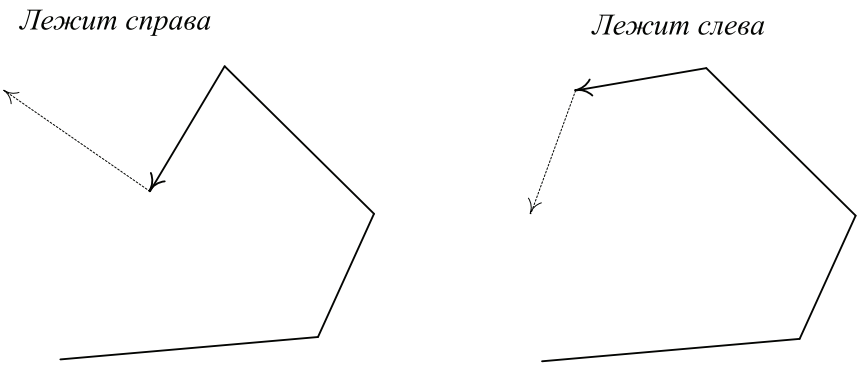

Рис. 2.26 ❖ Проверка очередной точки для алгоритма Грэхема

Точка p_i лежит либо слева от отрезка $s_{j-1}s_j$, либо справа (рис. 2.26). В нормальной выпуклой оболочке следующая точка должна лежать слева относительно прямой, проходящей через две предыдущие. Если же точка p_i лежит справа, то мы удаляем точку s_j с вершины стека, рассматриваем следующие две точки с вершины стека s_j и s_{j-1} и для них проверяем положение точки p_i и т. д. Этому алгоритму соответствует следующий код на C++:

```
glm::vec2 nextToTop ( std::stack<glm::vec2>& s )
{
      glm::vec2 top  = s.pop ();
      glm::vec2 next = s.top ();

      s.push ( top );

      return next;
}

std::vector<glm::vec2> convexHull ( glm::vec2 * p, int n )
{
      float yMin = p [0].y;
      int      iMin = 0;

      for ( int i = 1; i < n; i++)
      {
            float y = p [i].y;

            if ( (y < yMin) || ((y == yMin) && p[i].x < p[iMin.x]) )
            {
                  iMin = i;
                  yMin = p [i].y;
            }
      }

      std::swap ( p [0], p [iMin] );

      glm::vec2 p0 = p [0];

      std::sort ( &p [1], &p [n], grahamCompare );

      std::stack<glm::vec2> s;

      s.push ( p [0] );
      s.push ( p [1] );
      s.push ( p [2] );

      for ( int i = 3; i < n; i++ )
      {
            while ( !s.empty () &&
                  orientation ( nextToTop ( s ), s.top (), p [i] )) != CCW )
                        s.pop ();

            s.push ( p [i] );
      }

            // теперь стек содержит вершины выпуклой оболочки
      std::vector<glm::vec2>      hull ( s.size () );

      while ( !.s.empty () )
                  hull.push_front ( s.pop () );

      return hull;
}
```

Как можно заметить, сложность этого алгоритма определяется исключительно операцией сортировки, поскольку после сортировки каждая вершина обрабатывается не более двух раз. Таким образом, сложность алгоритма Грэхема составляет $O(n \cdot \log n)$.

Триангуляция Делоне. Диаграмма Вороного

Пусть у нас есть некоторый набор S точек p_0, p_1, ..., p_{n-1}. Тогда можно построить *триангуляцию S* – разбиение выпуклой оболочки S на набор треугольников, вершинами которых являются точки из множества S и при этом различные треугольники не имеют общих внутренних точек. При этом каждая точка из набора S является либо граничной, либо внутренней (рис. 2.27).

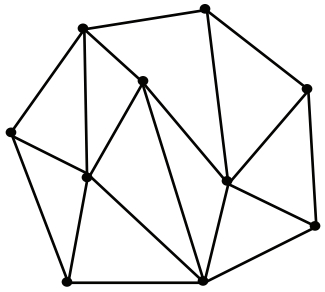

Рис. 2.27 ❖ Триангуляция набора точек

Справедливо следующее утверждение: пусть набор S содержит $n \geq 3$ точек и не все из них коллинеарные. Кроме того, пусть i точек являются внутренними. Тогда при любом способе триангуляции набора S будет получено $n + i - 2$ треугольника.

Однако, как видно по рис. 2.27, не все способы триангуляции одинаково хороши. Так, на рис. 2.27 справа мы видим несколько сильно вытянутых треугольников. Среди всех способов триангуляции выделяют *триангуляцию Делоне* (Delauney), являющуюся в некотором смысле оптимальной (рис. 2.28).

Триангуляция набора точек называется *триангуляцией Делоне*, если внутри каждой окружности, описанной вокруг произвольного треугольника, не содержится и одной из точек набора S.

Для того чтобы триангуляция Делоне вообще существовала, достаточно, чтобы в наборе S было не менее трех точек и все они не были коллинеарны (не лежали на одной прямой). В случае если никакие четыре точки из набора S не лежат на одной окружности, триангуляция Делоне единственна.

Рассмотрим теперь один из алгоритмов, который может быть использован для построения триангуляции Делоне.

Мы будем строить триангуляцию Делоне итеративно, на каждом шаге к текущей триангуляции будет добавляться один треугольник. Изначально триангуляция будет состоять всего из одного ребра границы выпуклой оболочки множества S. В конце работы мы получим полностью завершенную триангуляцию нашего множества. Каждая итерация подключает новый треугольник к границе текущей триангуляции множества S.

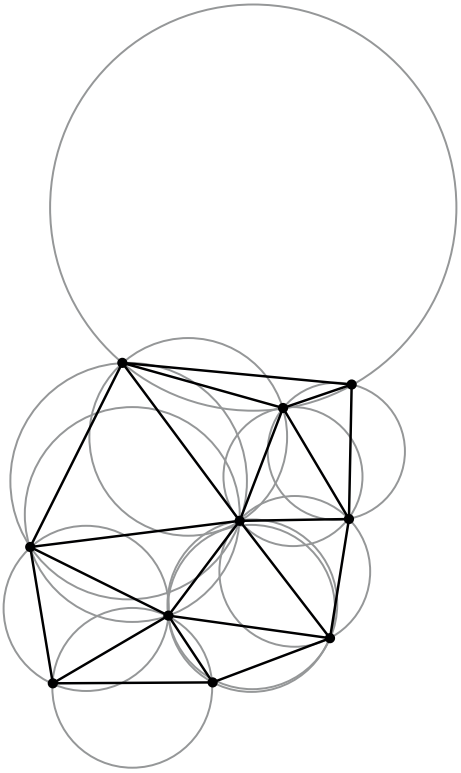

Рис. 2.28 ❖ Триангуляция Делоне

Обратите внимание, что для каждого ребра триангуляции есть две области, лежащие по разные стороны от прямой, проходящей через это ребро. Все ребра триангуляции Делоне можно разбить на следующие три класса:

- спящие ребра – те ребра, которые еще не были обработаны нашим алгоритмом;
- живые ребра – известные ребра, для которых известна только одна из двух примыкающих к нему областей;
- мертвые ребра – обнаруженные ребра, для которых известны обе примыкающие к нему области.

В самом начале работы алгоритма у нас есть всего одно живое ребро, принадлежащее границе выпуклой оболочки S. Все остальные ребра являются спящими. В ходе работы эти спящие ребра сперва станут живыми, а затем – мертвыми. Но при этом граница всегда будет состоять из живых ребер.

На очередном шаге мы выбираем некоторое ребро e из границы. Для этого ребра мы ищем вторую область, примыкающую к нему. Если эта область является треугольником, образованным ребром e и некоторой точкой p, то ребро e становится мертвым, а остальные два ребра этого треугольника переходят в следующее состояние: если ребро было спящим, то оно становится живым, а если оно было живым, то оно становится мертвым. Саму вершину p мы будем называть *сопряженной* к ребру e.

В случае когда искомая область не является треугольником (т. е. она не ограничена), ребро *e* становится мертвым (рис. 2.29).

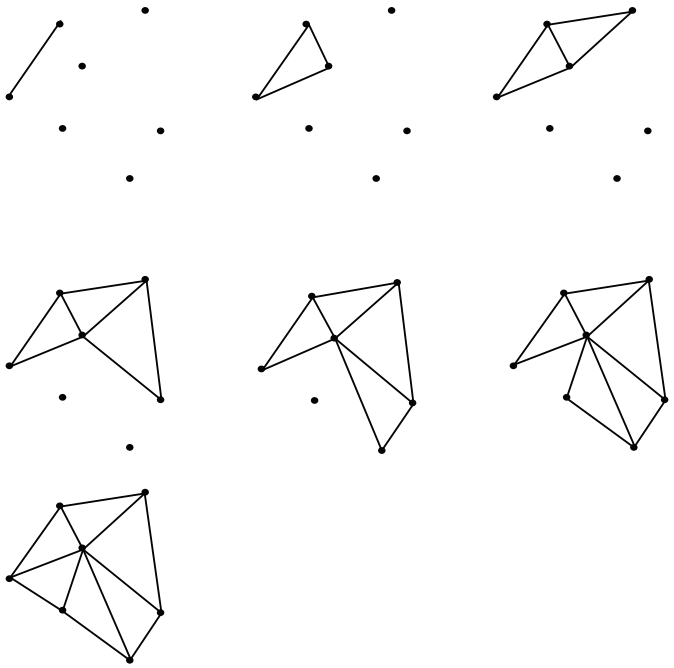

Рис. 2.29 ❖ Шаги построения триангуляции

Для начала работы алгоритма нам нужно какое-либо ребро, принадлежащее границе выпуклой оболочки *S*. Кроме того, нам нужно будет где-то хранить все живые ребра и находить ребро с минимальной y-координатой – для этого лучше всего подходит класс `std::unordered_set`.

```
std::list<triangle> triangulate ( const glm::vec2 * p, int n )
{
            glm::vec2                n, v;
            std::list<triangle>    * result = new std::list<triangle>;
            std::unordered_set<edge> frontier;

            edge e = hullEdge ( p, n );    // выбираем ребро

            frontier.insert ( e );

            while ( !frontier.empty () )
            {
                  e = removeMin ( frontier );

                  if ( mate ( e, p, n, v ) )
                  {
                        updateFrontier ( frontier, v, e.first );
                        updateFrontier ( frontier, e.second, v );
```

```
                        result -> push_front ( triangle ( e.first, e.second, v ) );
            }

            return result;
}
```

Функция updateFrontier служит для изменения состояния ребра из одного типа в следующий:

```
void updateFrontier ( std::unordered_set<edge>& frontier,
                      const glm::vec2& a, const glm::vec2& b )
{
      edge e = edge ( a, b );

      if ( std::set<edge>::iterator it = frontier.find ( e ) !=
     frontier.end () )
            frontier.erase ( it );
      else
            frontier.insert ( edge ( b, a ) );
}
```

Для нахождения ребра выпуклой оболочки служит следующая функция:

```
edge hullEdge ( glm::vec2 * p, int n )
{
      int m = 0, i = 1;

      for ( ; i < n; i++ )
            if ( p[i].x < p[m].x ||
                ((p[i].x == p [m].x) && (p[i].y < p[m].y)))
                  m = i;

      std::swap ( p [0], p [m] );

      for ( m = 1, i = 2; i < n; i++ )
      {
            int c = classify ( p [0], p [m], p [i] );

            if ((c == LEFT) || (c == BETWEEN)))
                  m = i;
      }

      return edge ( p [0], p [m] );
}
```

Нам осталось лишь реализовать функцию mate, которая для заданного ребра *ab* находит вершину *c*, такую что в окружность, проведенную через точки *a*, *b* и *c*, не попадает ни одной другой точки из множества *S*.

Для начала ограничим множество рассматриваемых точек только теми, которые лежат справа от отрезка *ab*.

Произвольной точке *c*, лежащей справа от отрезка *ab*, можно сопоставить центр окружности, проходящей через точки *a*, *b* и *c*. Нас интересует минимальная подобная окружность, т. е. нужно найти такую точку *c*, что окружность, проходящая

через *a*, *b* и *c*, будет минимальна. Очевидно, что минимальной окружности соответствует ближайший к *ab* центр окружности *Oc* (рис. 2.30).

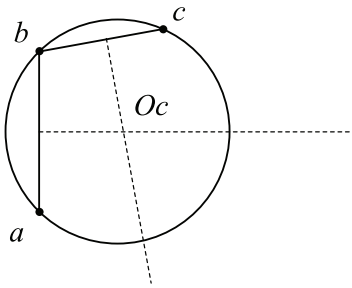

Рис. 2.30 ❖ Окружность, проходящая через точки *a*, *b* и *c*

При этом данный центр будет лежать на пересечении серединных перпендикуляров к отрезкам *ab* и *bc* и его можно описать параметром *t* вдоль серединного перпендикуляра к *ab*.

В результате мы приходим к следующей реализации функции *a*, *b* и *c*:

```
bool mate ( const edge& e, const glm::vec2 * p, int n, glm::vec2& v )
{
      glm::vec2 * bestVertex = nullptr;
      float       bestT      = FLT_MAX;
      edge        f = rotate ( e );

      for ( int i = 0; i < n; i++ )
            if ( classify ( e.first, e.second, p [p] ) == RIGHT )
            {
                  edge g = rotate ( edge ( e.second, p [i] ) );

                  intersect ( f, g, t );

                  if ( t < bestT )
                  {
                        bestVertex = &p [i];
                        bestT      = t;
                  }
            }

      if ( bestVertex )
      {
            v = *bestVertex;

            return true;
      }

      return false;
}
```

Ниже приводятся реализации вспомогательных функций.

```
int intersect ( const edge& f, const edge& e, float& t )
{
      glm::vec2      a = f.first;
      glm::vec2      b = f.second;
      glm::vec2      c = e.first;
      glm::vec2      d = e.second;
      glm::vec2      n ( d.y - c.y, c.x - d.x );
      float          denom = glm::dot ( n, b - a );

      if ( fabs ( denom ) < EPS )
      {
            int cls = classify ( e.first, e.second, f.first );

            t = FLT_MAX;

            if ((cls == LEFT) || (cls == RIGHT))
                  return PARALLEL;

            return COLLINEAR;
      }

      t = - glm::dot ( n, a -c ) / denom;

      return SKEW;
}

edge rotate ( const edge& e )
{
      glm::vec2      m = 0.5f * ( e.first + e.second );
      glm::vec2      v = e.second - e.first;
      glm::vec2      n ( v.y, -v.x );

      return edge ( m - 0.5f * n, m + 0.5f * n );
}
```

В этом примере использовался вспомогательный класс edge, для которого введена операция сравнения, чтобы его можно было поместить в std::set. Ниже приводится описание и реализация этого класса.

```
inline operator < ( const glm::vec2& a, const glm::vec2& b )
{
      if ( a.x < b.x )
            return true;

      if ( a.x > b.x )
            return false;

      return a.y < b.y;
}

class edge
{
public:
      glm::vec2 first, second;

      edge ( const glm::vec2& a, const glm::vec2& b ) : first ( a ), second ( b ) {}

      operator < ( const edge& e ) const
      {
```

```
            if ( first < e.first )
                  return true;

            if ( first > e.first )
                  return false;

            if ( second < e.second )
                  return true;

            return false;
      }
};
```

Рассмотренный выше пример лишь демонстрирует один из способов построения триангуляции Делоне, и его вычислительная сложность составляет $O(n^2)$.

Для работы с большими наборами данных лучше использовать готовые библиотеки, например CGAL (**http://www.cgal.org**).

Реализация булевых операций над многоугольниками. Метод построчного сканирования. Разложение на трапеции

Довольно часто возникает следующая задача – у нас есть два многоугольника (или набора многоугольников) *A* и *B*. Необходимо найти результат одной из следующих логических (булевых) операций над ними – *A and B*, *A or B*, *A xor B* или *A not B* (рис. 2.31).

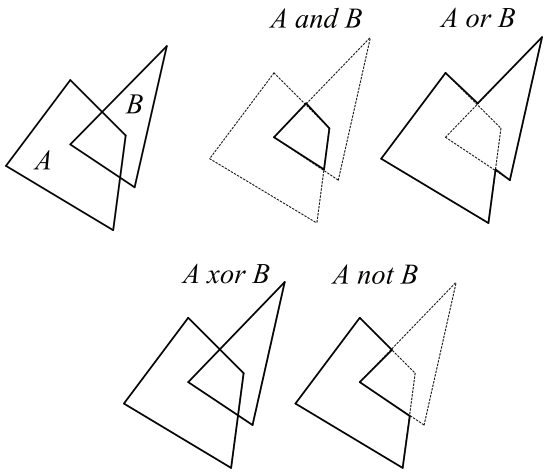

Рис. 2.31 ❖ Булевы операции над многоугольниками *A* и *B*

Одним из наиболее распространенных методов решения подобной задачи является метод *построчного сканирования* (scanline). Данный метод фактически понижает размерность задачи и сводит ее к серии легко решаемых одномерных задач.

Пусть, например, нам нужно найти пересечение (*and*) многоугольников *A* и *B* (рис. 2.32). Проведем вертикальную линию *l* и рассмотрим пересечение ею всех имеющихся многоугольников (включая и результат операции пересечения).

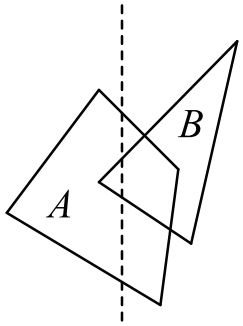

Рис. 2.32 ❖ Пересечение исходных многоугольников вертикальной прямой

Пусть пересечением l и A является множество отрезков a, а пересечением l и B – множество отрезков b:

$a = l \text{ and } A$;
$b = l \text{ and } B$.

Тогда пересечением l с результатом пересечения A и B будет на самом деле просто пересечение множеств одномерных отрезков a и b, так как $l \cap (A \cap B) = (l \cap A) \cap (l \cap B)$.

Таким образом, вдоль прямой l задача сводится к легко решаемому одномерному случаю. Перемещая прямую l слева направо, мы получим требуемое пересечение многоугольников A и B.

Однако можно заметить, что нам достаточно рассмотреть только такие вертикальные прямые, которые проходят через специальные точки-события.

В качестве таких точек выступают как вершины многоугольников A и B, так и точки пересечения ребер A и B друг с другом. Проведя анализ вдоль прямых, проходящих через эти точки-события, мы можем получить вершины результирующего многоугольника (рис. 2.33).

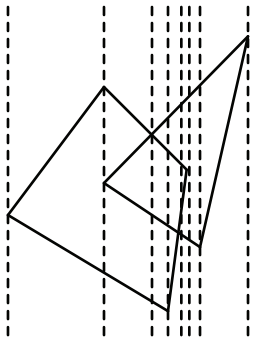

Рис. 2.33 ❖ Прямые, проходящие через точки-события

Для работы алгоритма нам понадобится упорядоченная по возрастанию x очередь q, которая будут содержать точки-события. Изначально мы заносим в q все вершины A и B.

Также нам понадобится таблица активных ребер a, изначально пустая. В этой таблице будут находиться только те ребра, обработка которых началась, но еще не завершилась (т. е. мы уже перешли правее левого конца ребра, но еще не дошли до правого).

По мере продвижения сканирующей линии слева направо мы будем добавлять новые ребра в a, удаляя при этом из a уже обработанные ребра. Также мы будем находить попарные пересечения ребер из a между собой и добавлять их в q.

В результате весь алгоритм можно представить следующим кодом на Python:

```
a = []
q = []
q.append ( a.vertices )       # Добавляем вершины A
q.append ( b.vertices )       # Добавляем вершины B
q.sort ( compareX )           # Сортируем по минимальной x-координате

while not q.empty():          # Пока есть хоть одно событие,
      e = q.pop ()            # снимаем ребро с минимальной x-координатой
      a.append ( e )          # Добавляем это ребро к таблице активных ребер
      a.sort ( compareY )
                              # Проверяем нужно ли удалить ребро
      if a.contains ( e.begin ) and a.contains ( e.end ):
            a.remove ( e )
                                        # Найдем пересечения ребер
      q1 = computeEdgeCrossings ( a )
                                        # Используем сортировку слиянием для вставки в q
      q.mergeSort ( q1, compareX )
                                        # Выполнить логическую операцию над a
      computeBoolean ( operation, a, e )
```

Данный метод позволяет находить результаты булевых операций над множеством многоугольников за $O(n \log n)$, где n – это общее число вершин всех входных многоугольников.

К сожалению, для очень больших объемов данных (например, миллионов многоугольников) алгоритм ведет себя не очень хорошо и требует очень больших вычислительных затрат, так как он не является *локальным*. Стандартным приемом борьбы с этим является разбиение входной геометрии на отдельные не взаимодействующие между собой группы многоугольников и применение метода построчного сканирования индивидуально к каждой такой группе.

В качестве альтернативного варианта может выступать разбиение всей входной геометрии на трапеции с вертикальными основаниями. Как известно, любой многоугольник (в том числе и с дырками внутри себя) можно представить в виде объединения не имеющих общих внутренних точек трапеций (рис. 2.34).

Тем самым вся входная геометрия может быть представлена как множество трапеций. Для обеспечения локальности можно автоматически разбивать слишком большие трапеции на небольшие части.

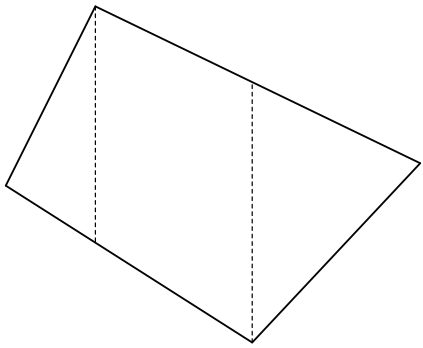

Рис. 2.34 ❖ Представление многоугольника в виде набора трапеций

Результат любой булевой операции над парой трапеций также всегда может быть представлен в виде набора трапеций. Таким образом можно отсортировать входные списки трапеций и разбить на отдельные, не взаимодействующие «окна». Тем самым мы получаем локальность – на результат в данном окне не влияет геометрия, содержащаяся в других окнах.

Глава 3

Координаты и преобразования в пространстве. Кватернионы

В этой главе мы попробуем обобщить ряд двухмерных понятий на трехмерное пространство. Мы рассмотрим координаты в трехмерном пространстве, основные операции над ними, а также базовые преобразования в пространстве. Кроме того, в этой главе будут рассмотрены способы задания ориентации трехмерных объектов, однородные координаты и кватернионы. Мы рассмотрим использование библиотеки GLM для работы с трехмерными и четырехмерными векторами, матрицами и кватернионами.

Векторы и матрицы в пространстве

В двухмерном случае векторы были столбцами, состоящими из двух чисел – координат x и y. В трехмерном случае к этим двум координатам добавляется третья координата z и векторы становятся столбцами из трех чисел:

$$u = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Как и на плоскости, мы можем ввести операции сложения векторов и умножения вектора на число поэлементным образом. Легко убедиться, что при таком способе введения операций основные свойства этих операций из главы 1 сохраняются.

$$\begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} + \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} = \begin{pmatrix} x_u + x_v \\ y_u + y_v \\ z_u + z_v \end{pmatrix}.$$

$$\alpha \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \begin{pmatrix} \alpha x_u \\ \alpha y_u \\ \alpha z_u \end{pmatrix}.$$

Множество всех таких троек вещественных чисел вместе с введенными подобным образом операциями сложения и умножения на число мы будем обозначать $\mathbb{R}^3$.

Также можно ввести в трехмерном пространстве скалярное произведение двух векторов, определив его следующим образом:

$$(u, v) = x_u x_v + y_u y_v + z_u z_v.$$

Из введенного таким образом скалярного произведения можно ввести длину вектора и угол между векторами. Легко убедиться в том, что все базовые свойства для скалярного произведения и длин из главы 1 также будут выполнены.

Однако, кроме скалярного произведения двух векторов, в трехмерном пространстве можно также ввести и их *векторное произведение* (cross product), обозначаемое как $[u, v]$ или $u \times v$. Векторное произведение двух трехмерных векторов является трехмерным вектором и задается следующим образом:

$$[u,v] = \begin{pmatrix} y_u z_v - y_v z_u \\ x_v z_u - x_u z_v \\ x_u y_v - x_v y_u \end{pmatrix}.$$

Можно заметить, что компоненты векторного произведения можно выразить через определители матриц 2×2, составленных из координат исходных векторов:

$$[u,v] = \left(\det \begin{pmatrix} y_u & z_u \\ y_v & z_v \end{pmatrix}, \det \begin{pmatrix} z_u & x_u \\ z_v & x_v \end{pmatrix}, \det \begin{pmatrix} x_u & y_u \\ x_v & y_v \end{pmatrix} \right).$$

Векторное произведение обладает следующими свойствами:

$[u, v] = -[v, u]$;
$[u, u] = 0$;
$([u, v], u) = ([u, v], v) = 0$;
$\|[u, v]\| = \|u\| \cdot \|v\| \cdot \sin\alpha$.

Из этих свойств следует, что векторным произведением двух векторов u и v является вектор w, перпендикулярный обоим векторам u и v. Длина этого вектора равна произведению длин векторов u и v на синус угла между ними. Однако одного этого недостаточно для точного определения направления векторного произведения, так как существует два противоположно направленных вектора, удовлетворяющих этим свойствам.

Соответственно, добавляется еще одно свойство – векторы u, v и w должны образовывать *правую тройку*, т. е. быть расположенными в пространстве как большой, указательный и средний пальцы правой руки (рис. 3.1).

Из свойств векторного произведения легко получить способ определения площади S треугольника, заданного своими вершинами a, b и c:

$$S = \frac{1}{2} \left\| [b-a, c-a] \right\|.$$

Также можно обобщить понятие матрицы. Матрицей 3×3 называется набор из 9 чисел, упорядоченных как таблица из трех строк и трех столбцов, как показано ниже:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

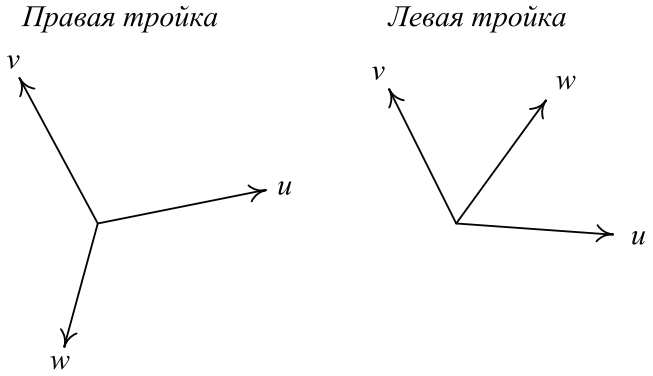

Рис. 3.1 ❖ Правая и левая тройки векторов в пространстве

Аналогично двухмерному случаю, мы можем ввести операции сложения матриц и умножения матрицы на число поэлементным образом. Также можно ввести операцию транспонирования матрицы:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}.$$

Множество всех матриц 3×3, составленных из вещественных чисел, с введенными таким образом операциями обозначается $\mathbb{R}^{3\times3}$.

Как и для матриц 2×2, можно ввести операцию перемножения двух матриц A и B как операцию, дающую в результате матрицу C размером 3×3, элементы которой задаются следующей формулой (обратите внимание, что в двухмерном случае работает та же самая формула, только суммирование идет до двух, а не до трех):

$$c_{ij} = \sum_{k=1}^{3} a_{ik} b_{kj}.$$

Через I мы будем обозначать единичную матрицу:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Легко можно убедиться в том, что основные свойства для матриц и операций над ними из главы 1 будут выполняться и в этом случае.

По аналогии с операцией перемножения матриц можно ввести операцию умножения матрицы 3×3 на трехмерный вектор, задав ее следующим образом:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y + a_{13}z \\ a_{21}x + a_{22}y + a_{23}z \\ a_{31}x + a_{32}y + a_{33}z \end{pmatrix}.$$

Для введенной таким образом операции справедливо (как и в двухмерном случае) следующее свойство:

$(Au, v) = (u, A^T v)$.

Еще для матрицы 3×3 можно ввести понятие *обратной матрицы* – если для матрицы A существует такая матрица B, что $AB = BA = I$, то матрица B называется обратной матрицей к матрице A и обозначается как A^{-1}.

Двухмерная матрица была обратима тогда и только тогда, когда ее определитель был не равен нулю. Для матрицы 3×3 также можно ввести определитель, обладающий этим же свойством – матрица обратима тогда и только тогда, когда ее определитель не равен нулю. Определитель матрицы 3×3 задается формулой

$$\det A = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}.$$

Формула эта довольно сложна – она состоит из всевозможных произведений трех элементов по одному элементу из каждой строки и каждого столбца матрицы. Есть простой способ запоминания данной формулы, называемый *правилом Сарруса* (рис. 3.2).

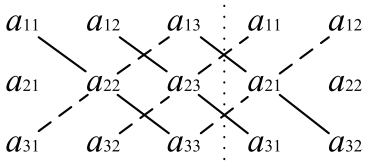

Рис. 3.2 ❖ Правило Сарруса

Мы добавляем к нашей матрице копии первых двух столбцов и далее просто строим шесть диагональных линий. Сначала линии идут слева направо и сверху вниз – они соответствуют знаку плюс, а затем линии идут слева направо снизу вверх – они соответствуют знаку минус.

Также определитель матрицы 3×3 можно разложить по любому столбцу или строке. Ниже приводится формула разложения определителя по первой строке:

$$\det A = a_{11}\det\begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{12}\det\begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} + a_{13}\det\begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}.$$

Можно показать, что все свойства определителя, сформулированные нами в главе 1, выполнены и в этом случае.

Преобразования в пространстве. Базовые преобразования

Так же, как и на плоскости, мы можем ввести понятие преобразования как отображения векторного пространства в себя $f: \mathbb{R}^3 \to \mathbb{R}^3$. Среди всех возможных преобразований обычно выделяют классы линейных и аффинных преобразований точно

так же, как мы это делали в двухмерном случае. На плоскости каждому линейному преобразованию всегда соответствует матрица 2×2, реализующая это преобразование. В пространстве каждому линейному преобразованию всегда соответствует матрица 3×3, множество всех таких матриц вместе с поэлементными операциями сложения и умножения на число обозначается $\mathbb{R}^{3\times3}$.

Среди всех возможных линейных преобразований рассматривается небольшое число основных – масштабирование, отражение, поворот и сдвиг.

Простейшим линейным преобразованием в пространстве является масштабирование. Оно задается при помощи диагональной матрицы со строго положительными элементами (коэффициентами масштабирования) на главной диагонали:

$$S_{\lambda\mu\nu} = \begin{pmatrix} \lambda & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \nu \end{pmatrix}, \lambda > 0, \mu > 0, \nu > 0.$$

Определитель этой матрицы равен произведению ее диагональных элементов, т. е. $\lambda\mu\nu$, что строго больше нуля. Поэтому данная матрица всегда обратима, причем обратная матрица также является матрицей масштабирования:

$$S^{-1}_{\lambda\mu\nu} = \begin{pmatrix} 1/\lambda & 0 & 0 \\ 0 & 1/\mu & 0 \\ 0 & 0 & 1/\nu \end{pmatrix}.$$

Если $\lambda = \mu = \nu$, то соответствующее масштабирование является *однородным*.

Другим крайне простым линейным преобразованием в пространстве является отражение. Но если на плоскости у нас было только два базовых отражения – относительно осей *Ox* и *Oy*, то в пространстве у нас будет отражение относительно не прямой, а плоскости. Таким образом, мы получим три базовых отражения – относительно плоскостей *Oxy*, *Oxz* и *Oyz*:

$$M_{xy} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix};$$

$$M_{yz} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & - \end{pmatrix};$$

$$M_{xz} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Определитель каждой из этих матриц равен –1, они обратимы, и обратной к каждой из них является сама матрица отражения, т. е. $M_{xy}^{-1} = M_{xy}$, $M_{yz}^{-1} = M_{yz}$ и $M_{xz}^{-1} = M_{xz}$.

В двухмерном случае у нас был только один базовый поворот – вокруг начала координат. В пространстве мы поворачиваем уже не вокруг точки, а вокруг прямой. Соответственно, у нас будет три базовых поворота – по одному для каждой координатной оси. Ниже приводятся их матрицы.

$$R_x(\varphi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{pmatrix}.$$

$$R_y(\varphi) = \begin{pmatrix} \cos\varphi & 0 & -\sin\varphi \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\varphi \end{pmatrix}.$$

$$R_z(\varphi) = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Обратите внимание, что каждая из этих матриц обладает важным свойством – транспонированная матрица совпадает с обратной. Если мы рассмотрим поворот вокруг произвольной прямой, проходящей через начало координат (чтобы преобразование было линейным), то это свойство сохраняется – обратная матрица поворота совпадет с транспонированной матрицей поворота.

Из этого, в частности, следует, что операция поворота сохраняет длины векторов и углы между ними, так как если R – это матрица поворота, то для любых векторов u и v справедливо $(Ru, Rv) = (u, R^T Rv) = (u, v)$.

Пусть у нас есть произвольная прямая, проходящая через начало координат, с заданным направляющим вектором l (который мы будем далее считать единичным). Тогда матрица поворота вокруг этой прямой на угол φ задается следующей фомулой:

$$R_l(\varphi) = I + \sin\varphi \cdot S + (1 - \cos\varphi) \cdot S^2.$$

В этой формуле через S обозначена следующая матрица:

$$S = \begin{pmatrix} 0 & l_z & -l_y \\ -l_z & 0 & l_x \\ l_y & -l_x & 0 \end{pmatrix}.$$

Операция сдвига (shear) может быть определена для каждой пары координатных осей, что дает в результате 6 базовых преобразований: $H_{xy}(\lambda)$, $H_{xz}(\lambda)$, $H_{yx}(\lambda)$, $H_{zx}(\lambda)$, $H_{yz}(\lambda)$ и $H_{zy}(\lambda)$. Здесь через первый индекс обозначено значение, вдоль какой именно из координатных осей будет изменение в ходе сдвига, а второй индекс – при помощи значения вдоль какой оси. Ниже приводится одна из этих матриц:

$$H_{xz}(\lambda) = \begin{pmatrix} 1 & 0 & \lambda \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Пример: отражение относительно плоскости

Давайте рассмотрим, как можно, используя базовые преобразования, построить преобразование отражения относительно произвольной плоскости, проходящей через начало координат.

Уравнение плоскости в пространстве имеет вид $(u, n) + d = 0$. Здесь n – это нормаль к плоскости, а d – это расстояние со знаком от плоскости до начала координат. В случае когда плоскость проходит через начало координат, это расстояние равно нулю, и уравнение принимает вид $(u, n) = 0$.

Первым нашим шагом будет поворот вокруг оси Ox на такой угол, чтобы повернутая нормаль n' полностью лежала в плоскости Oxz. Это значит, что $n'_y = 0$.

$$n' = R_x(\varphi)n = \begin{pmatrix} n_x \\ n_y\cos\varphi + n_z\sin\varphi \\ -n_y\sin\varphi + n_z\cos\varphi \end{pmatrix}.$$

Отсюда легко получить уравнение для второй компоненты вектора нормали:

$$n'_y = n_y\cos\varphi + n_z\sin\varphi = \sqrt{n_y^2 + n_z^2}\left(\frac{n_z}{\sqrt{n_y^2 + n_z^2}}\sin\varphi - \frac{-n_y}{\sqrt{n_y^2 + n_z^2}}\cos\varphi\right).$$

Так как сумма квадратов коэффициентов, стоящих перед $\sin\varphi$ и $\cos\varphi$, равна единице, то существует такой угол α, что

$$\cos\alpha = \frac{n_z}{\sqrt{n_y^2 + n_z^2}};$$

$$\sin\alpha = \frac{-n_y}{\sqrt{n_y^2 + n_z^2}}.$$

Тогда мы получаем следующее выражение для n'_y:

$$n'_y = \sqrt{n_y^2 + n_z^2}\cdot\sin(\varphi - \alpha).$$

Отсюда следует, что, взяв $\varphi = \alpha$, мы получим $n'_y = 0$, что нам и требовалось. На самом деле для построения матрицы поворота нам нужен не сам угол, а только его синус и косинус, которые мы и так уже знаем. Таким образом, мы можем записать нужную нам матрицу поворота в следующем виде:

$$R_x(\varphi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \dfrac{n_z}{\sqrt{n_y^2 + n_z^2}} & \dfrac{-n_y}{\sqrt{n_y^2 + n_z^2}} \\ 0 & \dfrac{n_y}{\sqrt{n_y^2 + n_z^2}} & \dfrac{n_z}{\sqrt{n_y^2 + n_z^2}} \end{pmatrix}.$$

Эта матрица преобразует вектор нормали n в следующий единичный вектор:

$$n' = \begin{pmatrix} n_x \\ 0 \\ \sqrt{n_y^2 + n_z^2} \end{pmatrix}.$$

Далее мы выполним еще один поворот – на этот раз вокруг оси *Oy* на некоторый угол ψ таким образом, чтобы $n''_z = 0$. Аналогично ранее рассмотренным выкладкам можно легко найти синус и косинус этого угла:

$$\sin\psi = -\sqrt{n_y^2 - n_z^2};$$
$$\cos\psi = n_x.$$

В результате этого мы получим следующий вектор нормали:

$$n'' = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

Тогда матрицу поворота $R_y(\psi)$ можно записать в таком виде:

$$R_y(\psi) = \begin{pmatrix} n_x & 0 & \sqrt{n_y^2 + n_z^2} \\ 0 & 1 & 0 \\ -\sqrt{n_y^2 + n_z^2} & 0 & n_x \end{pmatrix}.$$

Так как после этих поворотов плоскость, относительно которой нужно выполнить отражение, переходит в плоскость *Oyz*, то для выполнения отражения мы можем использовать матрицу M_{yz}. В результате мы приходим к следующей формуле:

$$M = R_x^T(\varphi)R_y^T(\psi)M_{yz}R_y(\psi)R_x(\varphi).$$

Однородные координаты

К сожалению, такое распространенное преобразование, как параллельный перенос (translation) на заданный ненулевой вектор *a* (задаваемое формулой $f(u) = u + a$), не является линейным и, значит, не может быть представлено в виде матрицы 3×3.

Для этого, аналогично двухмерному случаю, вводится понятие *аффинного преобразования* как суперпозиции линейного преобразования и преобразования переноса. Это преобразование может быть всегда представлено в виде:

$$f(u) = Mu + a.$$

Здесь *M* – некоторая матрица, а *a* – некоторый вектор.

Аффинное преобразование не может быть представлено в виде умножения на матрицу 3×3. Однако можно воспользоваться так называемыми *однородными координатами*, которые позволяют представить произвольное аффинное преобразование через матрицу 4×4.

Для этого давайте сопоставим каждому трехмерному вектору u следующий четырехмерный вектор h (который мы далее будем называть *однородным вектором*):

$$h = \begin{pmatrix} u_x \\ u_y \\ u_z \\ 1 \end{pmatrix}.$$

Таким образом, мы построили взаимно однозначное соответствие между множеством всех трехмерных векторов и множеством четырехмерных векторов, у которых последняя координата равна единице.

Подобно тому, как двухмерные векторы преобразовывались при помощи матриц 2×2, а трехмерные – при помощи матрицы 3×3, для преобразования четырехмерных векторов служат матрицы 4×4 (таблицы из четырех строк и четырех столбцов).

Рассмотрим матрицу 4×4 такого вида:

$$H = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_x \\ a_{21} & a_{22} & a_{23} & a_y \\ a_{31} & a_{32} & a_{33} & a_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Ее можно записать в следующей более компактной форме:

$$H = \begin{pmatrix} A & a \\ 0^T & 1 \end{pmatrix}.$$

Здесь через A обозначена верхняя левая подматрица 3×3 матрицы H, а через a – вектор, состоящий из первых трех элементов последнего столбца матрицы H.

Теперь рассмотрим результат умножения такой матрицы на вектор в однородных координатах. Несложно убедиться в справедливости следующей формулы:

$$\begin{pmatrix} A & a \\ 0^T & 1 \end{pmatrix} \begin{pmatrix} u \\ 1 \end{pmatrix} = \begin{pmatrix} Au + a \\ 1 \end{pmatrix}.$$

Мы видим, что в результате такого умножения мы снова получили вектор в однородных координатах, причем этот вектор соответствует аффинному преобразованию $f(u) = Au + a$. Тем самым мы просто можем дописать к трехмерному вектору четвертую координату, умножить на матрицу рассмотренного вида и в получившемся четырехмерном векторе взять первые три координаты (четвертая будет всегда равна единице). Таким образом, произвольное аффинное преобразование можно записать как умножение на матрицу 4×4.

У подобной записи аффинных преобразований есть очень полезное свойство – композиции аффинных преобразований соответствует матрица, равная произведению матриц соответствующих преобразований. Аналогично, обратному преобразованию соответствует обратная матрица.

Именно поэтому все современные графические API используют однородные координаты и матрицы 4×4 и все преобразования выполняют в однородных координатах.

Четырехмерные векторы, у которых последняя компонента не равна единице, также можно рассматривать как однородные координаты, при условии что эта компонента не равна нулю. Для этого вводится следующее простое правило: все четырехмерные векторы, отличающиеся умножением на константу, считаются *эквивалентными* (т. е. равными).

Тогда произвольный четырехмерный вектор (с не равной нулю четвертой компонентой) всегда будет эквивалентен вектору, получающемуся делением его на его четвертую компоненту:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \sim \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}.$$

Поэтому если рассмотреть графический конвейер для OpenGL или Direct3D (мы в деталях рассмотрим его позже), то там обязательно присутствует такой шаг, как *перспективное деление*, когда четырехмерный вектор делится на свою последнюю компоненту.

Жесткие преобразования

Среди всех аффинных преобразований выделяется один важный класс *жестких* (rigid) преобразований. Под жесткими понимаются аффинные преобразования, являющиеся композицией переноса и поворота:

$$f(u) = Ru + a.$$

Важным свойством жестких преобразований является сохранение углов между векторами и расстояний между ними, т. е. если у нас есть геометрический объект, заданный массивом вершин, и мы подвергаем его жесткому преобразованию, то он не деформируется. Многие используемые в компьютерной графике преобразования являются жесткими.

Для жесткого преобразования крайне просто построить обратное. Так как жесткое преобразование можно записать как суперпозицию переноса $T(a)$ и поворота $R(\varphi)$, то справедливо

$$f(u) = T(a)R(\varphi)u.$$

Тогда для обратного преобразования мы получаем:

$$f^{-1}(u) = (R(\varphi)T(a))^{-1}u = T^{-1}(a)R^{-1}(\varphi)u = T(-a)R^{T}(\varphi)u.$$

Обратите внимание, что если у нас есть жесткое преобразование, заданное матрицей 4×4, тогда соответствующее ему преобразование поворота будет задаваться верхней левой 3×3 подматрицей этой матрицы.

Преобразования нормали

Пусть у нас есть некоторый треугольник, который подвергается аффинному преобразованию. В результате применения этого преобразования треугольник изменится, но также изменится и вектор нормали к нему. Рассмотрим подробнее, каким именно образом происходит изменение нормали к этому треугольнику.

Итак, пусть есть треугольник, заданный своими вершинами $p_0p_1p_2$, и есть некоторое аффинное преобразование, заданное матрицей H. В результате применения этого преобразования к треугольнику мы получим новый треугольник с вершинами в точках Hp_0, Hp_1 и Hp_2 и нормалью n' (рис. 3.3).

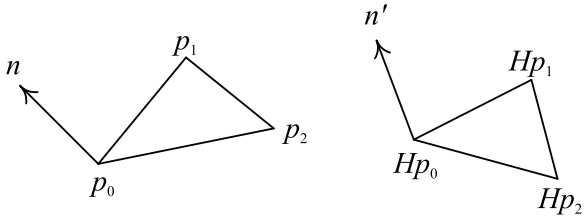

Рис. 3.3 ❖ Преобразование треугольника

Преобразование H состоит из переноса (задаваемого четвертым столбцом матрицы H) и линейного преобразования M (задаваемого верхней левой 3×3 подматрицей матрицы H). Поскольку перенос всех вершин на один и тот же вектор никак не влияет на нормаль к треугольнику, мы далее будем рассматривать только линейное преобразование M.

Так как n – это вектор нормали к треугольнику, то он должен быть перпендикулярен ко всем ребрам треугольника, т. е. справедливы следующие равенства:

$$(n, p_i - p_0) = 0, i = 1, 2.$$

С другой стороны, вектор n' – это перпендикуляр к преобразованному треугольнику, поэтому выполнены следующие равенства:

$$(n', Mp_i - Mp_0) = 0, i = 1, 2.$$

Используя свойства скалярного произведения, получаем:

$$0 = (n', Mp_i - Mp_0) = (n', M(p_i - p_0)) = (M^T n', p_i - p_0), i = 1, 2.$$

Таким образом, вектор $M^T n'$ также является вектором нормали к нашему треугольнику, т. е. отличается от исходного вектора n умножением на константу. Константа нас не интересует, поэтому мы можем считать, что $M^T n' = n$. Отсюда мы получаем формулу для преобразования нормали:

$$n' = (M^T)^{-1} n.$$

Проектирование. Параллельное проектирование

Несмотря на то что используемые для рендеринга геометрические объекты расположены в трехмерном пространстве, результатом рендеринга является плоское

(т. е. двухмерное) изображение (или последовательность изображений при рендеринге анимации).

Поэтому нам нужно некоторое преобразование, способное отображать трехмерные объекты на двухмерную плоскость. При этом крайне желательно, чтобы оно переводило прямые линии в прямые и не вносило серьезных искажений. Подобное преобразование называется *проектированием* (обратите внимание, что оно всегда вырождено) и является составной частью конвейера рендеринга.

Обычно используются всего два варианта проектирования – параллельное и перспективное. Чаще всего каждое из них задается не в общем виде, а в так называемой канонической форме – некотором упрощенном случае. При этом любое неканоническое проектирование может быть приведено к каноническому при помощи невырожденного аффинного преобразования.

Простейшим видом проектирования является *параллельное проектирование*. Рассмотрим подробнее, как именно оно задается.

Пусть у нас есть некоторая плоскость π (на которую осуществляется проектирование), задаваемая уравнением $(u, n) + d = 0$, и единичный вектор l, не параллельный этой плоскости (т. е. $(l, n) \neq 0$).

Тогда, чтобы получить проекцию произвольной точки p, мы проводим через эту точку прямую, параллельную вектору l. Так как эта прямая не может быть параллельна нашей плоскости, то она обязательно имеет с ней точку пересечения p', причем ровно одну. Эта точка и является проекцией точки p (рис. 3.4).

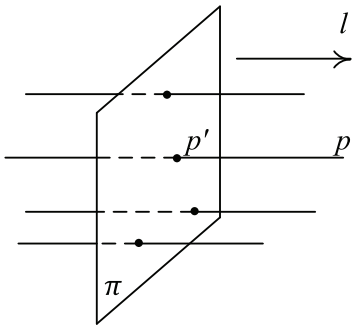

Рис. 3.4 ❖ Параллельная проекция точки

Прямую, проходящую через точку p с направляющим вектором l, можно описать при помощи следующего параметрического уравнения:

$$p(t) = p + l \cdot t.$$

Подставим это уравнение прямой в уравнение плоскости и найдем из него значение t, соответствующее точке пересечения:

$$(p + l \cdot t, n) + d = 0;$$

$$t = -\frac{d + (p, n)}{(l, n)}.$$

Отсюда легко можно получить саму точку p':

$$p' = p - \frac{d + (p,n)}{(l,n)} l.$$

Как видно из этой формулы, преобразование параллельного проектирования является аффинным, и поэтому оно может быть представлено в виде умножения на матрицу 4×4. Любое параллельное проектирование всегда можно невырожденным аффинным преобразованием привести к *каноническому* параллельному проектированию.

При каноническом параллельном проектировании проектирование осуществляется на плоскость *Oxy* и направление проектирования параллельно оси *Oz*. Этому соответствует следующая матрица:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Перспективное проектирование

Несмотря на свою простоту, параллельное проектирование обладает весьма серьезным недостатком. Давайте рассмотрим, как будет проектироваться куб, когда направление проектирования параллельно некоторым его ребрам (рис. 3.5).

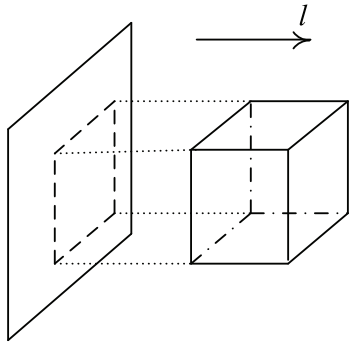

Рис. 3.5 ❖ Параллельное проектирование куба

Легко видно, что грани, параллельные направлению проектирования, перейдут просто в отрезки, а ребра, параллельные направлению проектирования, перейдут в точки. Это значит, что если в игре вроде FPS (First Person Shooter) мы смотрим вдоль коридора, то стены, а также пол и потолок этого коридора мы не увидим – они перейдут в отрезки. Это не совсем то, что мы обычно ожидаем в таких играх, поэтому для них такой способ проектирования непригоден.

Поэтому в ряде случаев используется другой, более сложный вид проектирования – *перспективное проектирование*. Этот способ проектирования лишен приведенного выше недостатка.

При перспективном проектировании мы, как и ранее, будем проектировать на некоторую плоскость π. Но на этот раз вместо направления проектирования мы

зададим *центр проектирования* – некоторую точку c, не лежащую на нашей плоскости. Эта плоскость разбивает все пространство на два полупространства. Мы будем проектировать только те точки, которые не лежат в том же полупространстве, что и центр проектирования c.

Для нахождения проекции произвольной точки p (лежащей в другом полупространстве, чем центр проектирования c) мы строим отрезок cp. Поскольку его концы лежат в разных полупространствах, то он всегда пересекает плоскость в единственной точке p'. Эту точку мы и будем считать искомой проекцией (рис. 3.6).

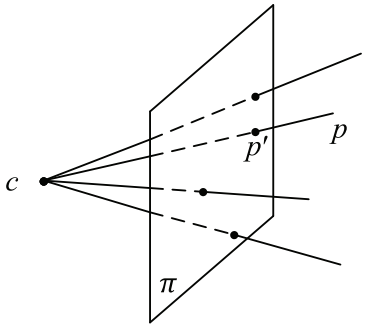

Рис. 3.6 ❖ Перспективное проектирование

Теперь давайте выведем формулы для перспективного проектирования. Далее мы будем выводить формулы для канонического перспективного проектирования – в качестве плоскости π выступает плокость Oxy, а центр проектирования лежит на положительной части оси Oz и имеет z-координату, равную единице.

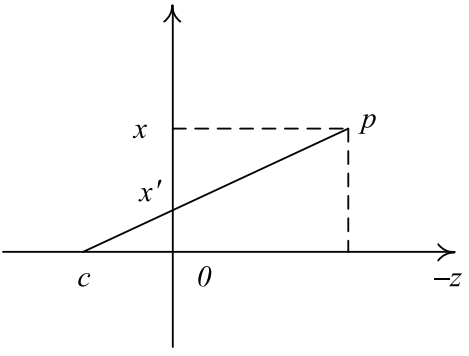

Рис. 3.7 ❖ Подобные треугольники для перспективного проектирования

Если мы отбросим координату y, то в результате всю получившуюся геометрию можно будет отобразить на плоскости (рис. 3.7). У нас возникает два подобных прямоугольных треугольника с общей вершиной в точке c. Отсюда мы можем записать следующее уравнение:

$$\frac{x}{1-z}=\frac{x'}{1}.$$

В этом уравнении через x' мы обозначили x-координату точки p'. Из этого уравнения мы получаем:

$$x' = \frac{x}{1-z}.$$

Аналогично мы можем найти вторую компоненту:

$$y' = \frac{y}{1-z}.$$

Легко видно, что получающееся преобразование не является аффинным, оно относится к классу дробно-линейных преобразований. Однако, несмотря на это, его можно записать через матрицу 4×4, воспользовавшись следующей эквивалентностью для однородных координат:

$$\begin{pmatrix} \frac{x}{1-z} \\ \frac{y}{1-z} \\ 0 \\ 1 \end{pmatrix} \sim \begin{pmatrix} x \\ y \\ 0 \\ 1-z \end{pmatrix}.$$

Таким образом, мы приходим к следующей матрице для канонического перспективного проектирования:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

Углы Эйлера. Задание ориентации в пространстве

Если в двухмерном случае ориентация задается всего лишь одним углом поворота, то в трехмерном случае все заметно сложнее. Леонард Эйлер показал, что произвольную ориентацию в пространстве можно выразить при помощи трех углов – *крена* (roll), *тангенса* (pitch) и *рыскания* (yaw, head), называемых углами Эйлера. Каждый из этих углов задает поворот вокруг координатных осей – z, x и y (рис. 3.8).

Таким образом, ориентация при помощи углов Эйлера задается следующей матрицей:

$$E(h, p, r) = R_{-z}(r)R_x(p)R_y(h).$$

Если выписать все возникающие матрицы и выполнить умножения, то мы приходим к следующей матрице:

$$E = \begin{pmatrix} \cos r \cdot \cos h - \sin r \cdot \sin p \cdot \sin h & -\sin r \cdot \cos p & -\sin h \cdot \cos r - \sin r \cdot \sin p \cdot \cos h \\ \sin r \cdot \cos h + \cos r \cdot \sin p \cdot \sin h & \cos r \cdot \cos p & -\sin r \cdot \sin h + \cos r \cdot \sin p \cdot \cos h \\ \cos p \cdot \sin h & -\sin p & \cos p \cdot \cos h \end{pmatrix}.$$

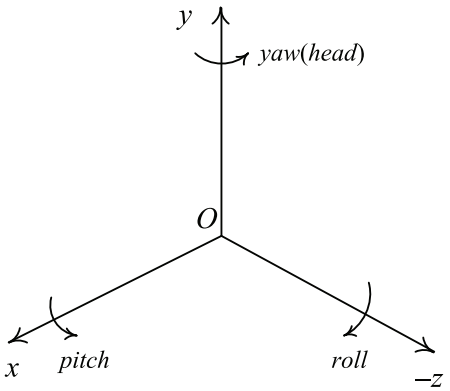

Рис. 3.8 ❖ Углы Эйлера

Поскольку такая матрица E является произведением трехмерных матриц поворота, то и она сама также является матрицей поворота, т. е. обратная к ней совпадает с транспонированной.

Несмотря на свою простоту и интуитивность, способу задания ориентации при помощи углов Эйлера присущ один очень серьезный недостаток. Давайте посмотрим, как будет выглядеть матрица $E(h, p, r)$ при $p = \frac{\pi}{2}$:

$$E\left(h, \frac{\pi}{2}, r\right) = \begin{pmatrix} \cos(r+h) & 0 & -\sin(r+h) \\ \sin(r+h) & 0 & -\cos(r+h) \\ 0 & -1 & 0 \end{pmatrix}.$$

Как видно, получившаяся матрица не зависит от h и r по отдельности, а только от их суммы. Это значит, что произошла потеря одной степени свободы, это явление называется *складыванием рамок* (gimbal lock).

Мы уже знаем, как построить результирующую матрицу поворота по трем углам Эйлера. Но, зная результирующую матрицу поворота, мы можем в ряде случаев восстановить и соответствующие углы Эйлера. Обозначим элементы этой матрицы через e_{ij}. Тогда можно прийти к следующим формулам:

$$e_{22} = -\sin p;$$

$$\frac{e_{12}}{e_{22}} = -\tan r;$$

$$\frac{e_{31}}{e_{33}} = \tan h.$$

Обратите внимание, что все эти формулы справедливы, только когда $\cos p \neq 0$. Если $\cos p = 0$, то мы можем найти лишь $r + h$.

Понятие линейного пространства и его размерности. Многомерные векторы и преобразования

Ранее мы уже сталкивались с двухмерными и трехмерными векторами, а когда вводили однородные координаты – то и с четырехмерными. Естественным образом возникает вопрос, можно ли и далее обобщить понятие вектора и что такое вообще вектор.

И в двухмерном, и в трехмерном случаях у нас было множество векторов ($\mathbb{R}^2$ и $\mathbb{R}^3$ соответственно), и над векторами из этих множеств вводились операции сложения и умножения на число, удовлетворяющие некоторым условиям. Таким образом, векторы – это просто элементы некоторого множества объектов с введенными над ними операциями. Именно так и определяется *линейное* (или *векторное*) пространство.

Множество L называется линейным (векторным) пространством (над множеством вещественных чисел) (а сами его элементы – векторами), если для всех его элементов определены операции сложения и умножения на число, удовлетворяющие следующим свойствам:

1) *коммутативность* $u + v = v + u$ *для всех* u *и* v *из* L;
2) *ассоциативность* $(u + v) + w = u + (v + w)$ *для всех* u, v *и* w *из* L;
3) *существует единственный элемент* 0 *такой, что* $0 + u = u + 0 = u$ *для всех* $u \in L$;
4) *для каждого* u *существует единственный элемент* $(-u)$ *такой, что* $u + (-u) = (-u) + u = 0$ *для всех* $u \in L$;
5) *для любых чисел* α *и* β *и любого* $u \in L$ *справедливо* $\alpha(\beta u) = (\alpha\beta)u$;
6) *для любого* u *справедливо* $1 \cdot u = u$;
7) *для любого числа* α *и любых* u *и* v *из* L *справедливо* $\alpha(u + v) = \alpha u + \alpha v$;
8) *для любых чисел* α *и* β *и любого* u *справедливо* $(\alpha + \beta)u = \alpha u + \beta u$.

Легко убедиться в том, что ранее рассмотренные множества векторов $\mathbb{R}^2$ и $\mathbb{R}^3$ вместе с введенными операциями полностью удовлетворяли всем этим свойствам.

Пусть у нас есть набор векторов $e_0, e_1, \ldots, e_{n-1}$. Тогда выражение вида $\alpha_0 e_0 + \alpha_1 e_1 + \cdots + \alpha_{n-1} e_{n-1}$ называется *линейной комбинацией* векторов $e_0, e_1, \ldots, e_{n-1}$.

Эти векторы называются *линейно независимыми*, если выражение $\alpha_0 e_0 + \alpha_1 e_1 + \cdots + \alpha_{n-1} e_{n-1}$ может равняться нулевому вектору тогда и только тогда, когда все коэффициенты $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ одновременно равны нулю. В противном случае они называются линейно зависимыми. Если набор векторов линейно зависим, то один из этих векторов может быть выражен через остальные векторы.

Число n называется размерностью линейного пространства (и обозначается $\dim L$), если существует n линейно независимых векторов, но любые $n + 1$ векторы всегда линейно зависимы. Линейно независимый набор векторов, в котором число векторов совпадает с размерностью пространства, называется *базисом*.

Легко убедиться в том, что размерность $\mathbb{R}^2$ равна двум, а размерность $\mathbb{R}^3$ – трем.

Можно рассмотреть множество упорядоченных четверок чисел, ввести в нем операции сложения и умножения на число поэлементным образом. В результате мы получим четырехмерное векторное пространство, обозначаемое как $\mathbb{R}^4$.

Аналогично множество всех матриц 2×2 также является четырехмерным пространством.

Вообще, если мы для произвольного натурального числа n рассмотрим множество упорядоченных наборов из n вещественных чисел и для них введем поэлементным образом операции сложения и умножения на число, то получим n-мерное векторное пространство, обозначаемое как $\mathbb{R}^n$.

Для этого пространства можно ввести понятие линейного преобразования. При этом каждому такому преобразованию будет соответствовать матрица $n \times n$, состоящая из n строк и n столбцов. Множество таких матриц образует линейное пространство, обозначаемое как $\mathbb{R}^{n \times n}$.

Если в векторном пространстве можно ввести понятие скалярного произведения, удовлетворяющего ранее рассмотренным свойствам, то такое пространство называется *евклидовым*. В $\mathbb{R}^n$ скалярное произведение можно ввести следующим образом:

$$(u, v) = \sum_{i=0}^{n-1} u_i v_i.$$

Используя скалярное произведение, можно ввести длину (норму вектора) $\|u\| = \sqrt{(u,u)}$ и угол между векторами (как мы делали ранее).

Все ранее рассмотренные векторные пространства были *конечномерными*. Давайте теперь рассмотрим примеры *бесконечномерных* векторных пространств. Под бесконечномерным пространством понимается такое векторное пространство, в котором для любого натурального числа n существует n линейно независимых векторов.

Через $C[a, b]$ обычно обозначают множество всех функций, определенных и непрерывных на отрезке $[a, b]$. Для таких функций легко ввести операции сложения и умножения (дающие в результате также непрерывные функции):

$(f + g)(x) = f(x) + g(x)$;
$(\alpha f)(x) = \alpha f(x)$.

Легко убедиться в том, что все свойства 1–8 выполнены. Тем самым $C[a, b]$ является векторным пространством. Это пространство *бесконечномерно* – несложно показать, что множество функций $1, x, x^2, x^3, x^4, \ldots, x^n$ линейно независимо для любого n.

Это пространство не является евклидовым – в нем нельзя ввести скалярное произведение, но в нем можно ввести понятие нормы, определив ее следующим образом:

$$\|f\|_C = \max_{x \in [a,b]} |f(x)|.$$

Тем самым это пространство является *нормированным*, для введенной таким образом нормы выполняются многие полезные свойства из конечномерного анализа. Например, можно ввести понятие сходимости, фундаментальности последовательности и т. д.

Теперь давайте рассмотрим пример бесконечномерного пространства со скалярным произведением. В качестве такого пространства мы рассмотрим $L_2[a, b]$ – множество всех определенных на $[a, b]$ функций, квадрат которых интегрируем (в смысле Лебега) на этом отрезке.

Операции сложения и умножения на число вводятся, как и для пространства непрерывных функций, а операцию скалярного произведения двух функций мы определим следующим образом:

$$(f,g)=\int_a^b f(x)g(x)dx.$$

Все необходимые свойства также выполнены, и мы можем ввести понятие нормы и угла.

Давайте теперь рассмотрим на отрезке $[0,\pi]$ множество функций $\sin(kx)$ и $\cos(kx)$. Легко показать, что они принадлежат $L_2[0,\pi]$ и являются линейно независимыми. Они образуют базис в этом пространстве. И на самом деле этот базис является ортогональным.

Теперь если мы запишем разложение функции по этому базису, то придем к обычному *ряду Фурье*. При этом формулы для коэффициентов ряда Фурье приобретают тот же смысл, что и для разложения по ортогональному базису в конечномерном случае, – это просто нормированные скалярные произведения.

Тем самым ряд Фурье – это просто разложение по ортогональному базису в бесконечномерном пространстве (функций).

Теперь давайте посмотрим, как в этом пространстве будет выглядеть линейное преобразование (линейный оператор):

$$(Af)(x)=\int_a^b K(x,y)f(y)dy.$$

Здесь через K обозначено так называемое *ядро*. Легко убедиться в том, что приведенная формула действительно задает линейный оператор в пространстве $L_2[a,b]$.

Системы координат в пространстве. Переходы между различными системами координат

Частным случаем линейно независимой системы векторов является набор ортогональных ненулевых векторов. Пусть у нас есть набор векторов $e_0, e_1, \ldots, e_{n-1}$, причем ни один из них не равен нулевому вектору и любые два различных вектора попарно перпендикулярны, т. е. $(e_i, e_j) = 0$, $i \neq j$. Тогда эти векторы будут линейно независимыми.

Если число линейно независимых векторов совпадает с размерностью пространства, то они называются *базисом* этого пространства. Тогда любой вектор может быть разложен по этому базису, т. е. представлен в виде $\alpha_0 e_0 + \alpha_1 e_1 + \cdots + \alpha_{n-1} e_{n-1}$, и сам набор $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ будет называться координатами вектора в этом базисе. Обратите внимание, что базисов может быть бесконечное количество и координаты одного и того же вектора в различных базисах будут отличаться.

Пусть у нас есть вектор p и некоторый базис $e_0, e_1, \ldots, e_{n-1}$. В общем случае задача нахождения координат вектора в этом базисе сводится к решению системы из n линейных уравнений с n неизвестными вида:

$$\alpha_0 e_0 + \alpha_1 e_1 + \cdots + \alpha_{n-1} e_{n-1} = p.$$

Однако в случае если базис ортогональный, то эта задача сильно упрощается. Давайте умножим приведенное выше векторное уравнение скалярно на e_i. В силу ортогональности базиса мы придем к следующему уравнению:

$$\alpha_i (e_i, e_i) = (p, e_i).$$

Так как $e_i \neq 0$, то отсюда легко получаем значение α_i.

Часто рассматривается не просто ортогональный базис, а *ортонормированный* базис, т. е. ортогональный базис, в котором все векторы имеют единичную длину. Для ортонормированного базиса справедливо $(e_i, e_j) = \delta_{ij}$. Здесь через δ_{ij} обозначен символ Кронекера. Для него всегда $\delta_{ii} = 1$, $\delta_{ij} = 0$, $i \neq j$. Тогда

$$\alpha_i = (p, e_i).$$

Каждый базис фактически задает свою систему координат, для случая ортонормированного базиса такая система координат называется *декартовой* (cartesian).

Пусть у нас есть базис e_0, e_1, e_2 в пространстве. Тогда если эти векторы образуют левую тройку векторов, то соответствующую систему координат мы будем называть *левосторонней*, иначе – *правосторонней*. В дальнейшем мы будем рассматривать только правосторонние системы координат.

По нашему ортонормированному базису построим матрицу R, в которой первая строка будет совпадать с вектором e_0, вторая строка – с вектором e_1, а третья – с вектором e_2. Обозначим это следующим образом:

$$R = (e_0|e_1|e_2)^T.$$

Тогда для нахождения новых координат p' точки p в этом базисе нам будет достаточно просто умножить R на p:

$$p' = Rp.$$

Рассмотрим теперь следующее скалярное произведение – RR^T. Легко убедиться в том, что в позиции (i, j) результирующей матрицы будет стоять скалярное произведение (e_i, e_j). В силу ортонормированности базиса получаем, что $RR^T = I$. Аналогично можно показать, что $R^TR = I$. Тем самым матрица R является матрицей поворота.

У ранее рассмотренных систем координат (базисов) начало (точка отсчета) совпадает с точкой O. Но это на самом деле не обязательно – новая система координат может иметь свое начало координат и определяться не только базисом e_0, e_1, e_2, но и началом координат p_0. Тогда координаты p' точки p в этой системе координат будут определяться по следующей формуле:

$$p = p_0 + Rp'.$$

Отсюда можно легко выразить новые координаты, используя то, что матрица R – это матрица поворота:

$$p' = R^T(p - p_0).$$

Как видно, операция перехода к другой системе координат является аффинной и, следовательно, может быть выражена при помощи матрицы 4×4.

Ортогонализация Грамма – Шмидта

Пусть у нас есть набор линейно независимых векторов $e_0, e_1, \ldots, e_{n-1}$ и мы хотим на его основе построить ортогональную систему векторов $u_0, u_1, \ldots, u_{n-1}$.

В качестве первого элемента новой системы мы возьмем сам вектор e_0. Следующий вектор мы будем искать по приводимой ниже формуле:

$u_1 = e_1 - \alpha_{10} u_0$.

Коэффициент α_{10} мы подберем таким образом, чтобы $(u_0, u_1) = 0$. Для этого достаточно взять этот коэффициент по следующей формуле:

$$\alpha_{10} = \frac{(e_1, u_0)}{(u_0, u_0)}.$$

Следующий элемент определим по аналогичной формуле:

$u_2 = e_2 - \alpha_{20} u_0 - \alpha_{21} u_1$.

Коэффициенты подбираются таким образом, чтобы $(u_0, u_2) = (u_1, u_2) = 0$. При этом мы знаем, что в силу построения $(u_0, u_1) = 0$. Для этого мы берем следующие коэффициенты:

$$\alpha_{20} = \frac{(e_2, u_0)}{(u_0, u_0)};$$

$$\alpha_{21} = \frac{(e_2, u_1)}{(u_1, u_1)}.$$

Продолжая этот процесс подобным образом, мы строим очередной вектор ортогональной системы по следующей формуле:

$$u_k = e_k - \sum_{i=0}^{k-1} \frac{(e_k, u_i)}{(u_i, u_i)} u_i.$$

Несложно убедиться в том, что для любой системы линейно независимых векторов мы в результате получим систему ортогональных векторов. Можно добавить в процесс нормирование каждого очередного получаемого вектора, тогда мы на выходе получим ортонормированную систему векторов. Этот процесс называется *ортогонализацией Грамма–Шмидта*.

Кватернионы. Задание поворотов и ориентации в пространстве при помощи кватернионов

Очень удобным и красивым способом представления ориентации (а также и произвольных поворотов) в трехмерном пространстве является использование *кватернионов*. Кватернионы были впервые введены в 1849 г. Уильямом Гамильтоном. Но в компьютерной графике их начали использовать только с 1985 года.

Кватернионы являются расширением комплексных чисел (которые, в свою очередь, являются расширением вещественных чисел). Как известно, произвольное комплексное число $c \in \mathbb{C}$ состоит из действительной и мнимой частей:

$c = x + iy$.

Здесь через i обозначена так называемая *мнимая единица* – число, квадрат которого равен –1 (т. е. $i^2 = -1$). Комплексные числа являются расширением множества вещественных чисел, и произвольный многочлен степени n имеет ровно n комплексных корней (с учетом кратности). Так, многочлен $z^2 + 1$ не имеет ни одного вещественного корня, но два комплексных – i и $-i$.

Для создания кватернионов была использована не одна мнимая единица, а целых три, обозначаемые как i, j и k. Тем самым произвольный кватернион (также состоящий из действительной и мнимой частей) может быть представлен в следующем виде:

$$q = w + ix + jy + kz.$$

Здесь w – это действительная часть кватерниона, а x, y, z – его мнимая часть. Используемые мнимые единицы независимы и удовлетворяют следующим условиям:

$$i^2 = j^2 = k^2 = -1;$$
$$ij = -ji = k;$$
$$jk = -kj = i;$$
$$ki = -ik = j.$$

Для кватернионов естественным образом вводятся операции сложения и умножения (получающиеся просто раскрытием скобок и перегруппировкой):

$$q_1 + q_2 = (w_1 + w_2) + (x_1 + x_2)i + (y_1 + y_2)j + (z_1 + z_2)k;$$
$$q_1q_2 = (w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2) + (w_1x_2 + w_2x_1 + y_1z_2 - y_2z_1)i + (w_1y_2 + w_2y_1 + x_2z_1 - x_1z_2)j + (w_1z_2 + w_2z_1 + x_1y_2 - x_2y_1)k.$$

Обратите внимание, что операция умножения кватернионов не коммутативна, т. е. в общем случае $q_1a_2 \neq q_2q_a$ (так как $i \cdot j \neq j \cdot i$).

Как и для комплексных чисел, для произвольного кватерниона q можно построить *сопряженный* ему кватернион q^*, задаваемый следующей формулой:

$$q^* = w - ix - jy - kz.$$

Для операции сопряжения выполнены следующие свойства:

$$(q^*)^* = q;$$
$$(pq)^* = q^*p^*;$$
$$(p + q)^* = p^* + q^*.$$

Также для любого кватерниона q выражение qq^* всегда вещественно (т. е. имеет нулевую мнимую часть) и всегда неотрицательно. Это позволяет ввести функцию $N(q)$ следующим образом:

$$N(q) = qq^* = w^2 + x^2 + y^2 + z^2.$$

Для введенной таким образом нормы справедливо:

$$N(q^*) = N(q);$$
$$N(p, q) = N(p)N(q).$$

Через эту функцию можно определить норму кватерниона как

$$\|q\| = \sqrt{N(q)}.$$

Произвольное вещественное число можно рассматривать как кватернион с нулевой мнимой частью. Для произвольного ненулевого кватерниона q можно ввести *обратный* кватернион q^{-1}, задав его при помощи формулы

$$q^{-1} = \frac{q^*}{N(q)}.$$

Легко убедиться в том, что

$qq^{-1} = q^{-1}q = 1.$

Существует и другая форма записи кватерниона – мнимая часть кватерниона q рассматривается как трехмерный вектор v, и тем самым кватернион записывается в виде $q = [w, v]$. С использованием этого обозначения произведение кватернионов можно переписать в следующем виде:

$q_1q_2 = [w_1w_2 - (v_1, v_2), w_2v_1 + w_1v_2 + v_1 \times v_2]$.

С другой стороны, кватернионы являются четырехмерными векторами, и для них можно ввести соответствующее скалярное произведение:

$(q_1, q_2) = w_1w_2 + x_1x_2 + y_1y_2 + z_1z_2$.

Кватернион q называется единичным, если $N(q) = 1$. Можно показать, что для произвольного единичного кватерниона q всегда найдется угол θ и единичный вектор v такие, что q можно представить в виде:

$q = [\sin\theta, v \cdot \cos\theta]$.

Можно показать по индукции, что для единичного кватерниона q и натурального числа n всегда справедливо следующее равенство:

$q^n = [\sin(n\theta), v \cdot \cos(n\theta)]$.

Для произвольного кватерниона $q = [w, v]$ на основе операции возведения в целочисленную степень можно ввести функцию e^q:

$$e^q = \sum_{n=0}^{\infty} \frac{q^n}{n!}.$$

Можно показать, что эта функция может быть записана в следующем виде:

$$e^q = e^w\left(\cos\|v\| + \frac{v}{\|v\|}\cos\|v\|\right).$$

Также можно ввести и обратную к ней функцию:

$$\log q = \log\|q\| + \frac{v}{\|v\|}\arccos\frac{w}{\|v\|}.$$

Пусть q – это единичный кватернион вида $[\sin\theta, v \cdot \cos\theta]$. Представим тогда произвольный трехмерный вектор p как кватернион вида $[0, p]$.

Теперь рассмотрим кватернион $p' = q \cdot [0, p] \cdot q^{-1}$. Его мнимая часть является трехмерным вектором, который на самом деле является результатом поворота исходного вектора p вокруг v на угол 2θ. Таким образом, произвольный единичный кватернион задает поворот (или ориентацию) в трехмерном пространстве.

Последовательное применение поворотов, задаваемых единичными кватернионами q_1 и q_2, соответствует повороту при помощи кватерниона, являющего произведением q_1 и q_2:

$q_2 \cdot (q_1 \cdot [0, p] \cdot q_1^{-1}) \cdot q_2^{-1} = (q_2 \cdot q_1) \cdot [0, p] \cdot (q_2, q_1)^{-1}$.

Для единичного кватерниона q можно выписать соответствующую ему матрицу поворота:

$$R(q) = \begin{pmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(xz+wy) \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) \end{pmatrix}.$$

Для задания поворотов и ориентации в трехмерном пространстве гораздо удобнее использовать не матрицы или углы Эйлера, а единичные кватернионы. Важно и часто используемой операцией над кватернионами является *сферическая линейная интерполяция* (slerp). Эта операция позволяет получить плавный переход от одной ориентации (q_1) до другой ориентации (q_2). Сферическая линейная интерполяция задается при помощи следующей формулы:

$$slerp(q_1, q_2, t) = \frac{\sin(\varphi(1-t))}{\sin\varphi} q_1 + \frac{\sin(t\varphi)}{\sin\varphi} q_2.$$

Здесь через φ обозначен угол, задаваемый формулой:

$$\cos\varphi = \frac{(q_1, q_2)}{\|q_1\| \cdot \|q_2\|}.$$

Скалярное произведение кватернионов считается просто как обычное скалярное произведение четырехмерных векторов.

Если у нас есть не два кватерниона, а целый набор $q_0, q_1, \ldots, q_{n-1}$ и мы хотим получить плавную интерполяцию между ними (т. е. плавный переход ориентации через всех них), то мы можем воспользоваться функцией `squad`.

Для определения этой функции нам понадобится сначала ввести вспомогательные контрольные точки

$$s_i = q_i \exp\left(-\frac{\log(q_{i+1}q_i^{-1}) + \log(q_{i-1}q_i^{-1})}{4}\right).$$

После этого на участке $q_{i-1}, q_i, q_{i+1}, q_{i+2}$ интерполирующая функция задается следующей формулой:

$$\text{squad}(q_i, q_{i+1}, s_i, s_{i+1}, t) = \text{slerp}(\text{slerp}(q_i, q_{i+1}, t), \text{slerp}(s_i, s_{i+1}, t), 2t(1-t)).$$

Использование библиотеки GLM для работы с 3- и 4-мерными векторами и матрицами, а также кватернионами

Библиотека `GLM` содержит в себе классы, служащие для представления трехмерных (`glm::vec3`) и четырехмерных (`glm::vec4`) векторов, матриц для их преобразования (`glm::mat3` и `glm::mat4`), а также класс для работы с кватернионами. Ниже приводится пример использования всех этих классов.

```
#define GLM_FORCE_RADIANS      // углы задаются в радианах
#define GLM_SWIZZLE
```

```
#include <glm/vec3.hpp>
#include <glm/vec4.hpp>
#include <glm/gtc/matrix_inverse.hpp>
#include <glm/gtc/quaternion.hpp>
#include <glm/gtx/quaternion.hpp>
#include <glm/geometric.hpp>

int main ()
{
      glm::vec3 a = glm::vec3(1.0f, 2.0f, 3.0f);
      glm::vec3 b = a.xyz;
      glm::vec3 c (a.xyz);
      glm::vec3 d (a.xyz());
      glm::vec3 e (a.x, a.yz);
      glm::vec3 g (1);
      glm::vec3 h = a + b;

      float     r = glm::dot   ( a, b );   // скалярное и векторное произведения
      glm::vec3 p = glm::cross ( b, c );

      glm::vec4 a4 = glm::vec4(1.0f, 2.0f, 3.0f, 4.0f);
      glm::vec4 b4 (a4.xy, a4.z, a4.w);

      glm::mat3 l(1.0f);      // единичная матрица
      glm::vec3 a1 = l * a;

      glm::mat3 mm ( glm::vec3(0.6f, 0.2f, 0.3f),
                     glm::vec3(0.2f, 0.7f, 0.5f),
                     glm::vec3(0.3f, 0.5f, 0.7f) );

      glm::mat3 mmi = glm::inverse(mm);               // получение обратной матрицы

      glm::mat3 m (0, 1, 2, 3, 4, 5, 6, 7, 8);
      glm::mat3 t = transpose(m);
      glm::quat q1 = glm::angleAxis ( M_PI * 0.25f,    // кватернион для направления и угла
                                      glm::vec3(0, 0, 1) );
      glm::quat n  = glm::normalize(q1);              // нормирование вектора
      float     l1 = glm::length(n);                  // длина вектора
      glm::quat q2 (1.0f, 0.0f, 0.0f, 1.0f);
      float     roll   = glm::roll  ( q1 );           // получение углов Эйлера по кватерниону
      float     pitch  = glm::pitch ( q1 );
      float     yaw    = glm::yaw   ( q1 );
      glm::vec3 angles = glm::eulerAngles(q1);
      glm::quat q3     = glm::slerp ( q1, q2, 0.2f );  // интерполяция кватернионов
      glm::mat4 rotate = glm::mat4_cast ( q1 );        // преобразование кватерниона в матрицу

      return 0;
}
```

Преобразование между кватернионом и базисом касательного пространства

В последующих главах мы будем часто сталкиваться с так называемым базисом *касательного пространства* (или TBN-базисом). Пока ограничимся только тем, что это ортонормированный базис, задаваемый при помощи трех векторов – *ка-*

сательного (tangent), *бинормали* (binormal) и *нормали* (normal). Можно установить взаимно однозначное соответствие между множеством таких базисов и множеством единичных кватернионов (так как каждый такой базис получается некоторым поворотом). Ниже приводится код (с использованием классов из библиотеки GLM), который для каждого такого базиса строит соответствующий ему единичный кватернион:

```
glm::quat tangentSpaceToQuat ( const glm::vec3 tangent,
                               const glm::vec3 binormal,
                               const glm::vec3 normal )
{
      glm::quat      q;

      q.x = normal.y   - binormal.z;
      q.y = tangent.z  - normal.x;
      q.z = binormal.x - tangent.y;
      q.w = 1.0f + tangent.x + binormal.y + normal.z;

      return glm::normalize ( q );
}
```

Следующая функция выполняет обратное преобразование – по заданному кватерниону строит соответствующий TBN-базис.

```
void quatToTangentSpace ( const glm::quat& q, glm::vec3& tangent,
                          glm::vec3& binormal, glm::vec3& normal )
{
      tangent = glm::vec3 ( 1, 0, 0 ) +
                glm::vec3 ( -2,  2, 2 ) * q.y * glm::vec3 ( q.y, q.z, q.w ) +
                glm::vec3 ( -2, -2, 2 ) * q.z * glm::vec3 ( q.z, q.w, q.x );

      binormal = glm::vec3 ( 0, 1, 0 ) +
                 glm::vec3 ( 2, -2,  1 ) * q.z * glm::vec3 ( q.w, q.z, q.y )+
                 glm::vec3 ( 2, -2, -2 ) * q.x * glm::vec3 ( q.y, q.x, q.w );

      normal = glm::vec3 ( 0, 0, 1 ) +
               glm::vec3 ( 2,  2, -2 ) * q.x * glm::vec3 ( q.z, q.w, q.y ) +
               glm::vec3 ( -2, 2, -2 ) * q.y * glm::vec3 ( q.w, q.z, q.y );
}
```

Собственные векторы и собственные числа матрицы

Давайте рассмотрим некоторую матрицу A (например, размером 3×3). Для этой матрицы рассмотрим следующее уравнение (λ – это некоторый параметр):

$$Au = \lambda u.$$

Данное уравнение можно переписать в виде обычной системы линейных уравнений с нулевой правой частью:

$$(A - \lambda E)u = 0.$$

Нулевой вектор всегда является решением этой системы. Давайте рассмотрим, есть ли у этой системы ненулевые решения.

Если $\det(A - \lambda E) \neq 0$, то существует обратная матрица $(A - \lambda E)^{-1}$, и тогда решение этой системы задается как произведение этой обратной матрицы на нулевой вектор. Таким образом, в этом случае, кроме нулевого решения, других решений нет.

В случае $\det(A - \lambda E) = 0$ матрицы система вырождена и у системы есть ненулевые решения. Требование обращения в ноль этого определителя дает обычное алгебраическое уравнение, степень которого равна порядку матрицы (т. е. в нашем случае это будет уравнение третьей степени).

Само такое уравнение называется *характеристическим уравнением* матрицы A, а его корни (если они есть) – *собственными числами* матрицы A.

Если λ – это собственное число матрицы A, то приведенная выше система уравнений имеет ненулевые решения, называемые *собственными векторами* матрицы A.

Обратите внимание, что не у каждой матрицы есть вещественные собственные числа и векторы. Однако справедливо утверждение: если для матрицы A выполнено $A = A^T$, то у этой матрицы всегда есть ровно столько собственных чисел, каков ее размер. Кроме того, собственные векторы, соответствующие различным собственным числам, ортогональны.

Мы будем использовать библиотеку `eig3x3` для нахождения собственных чисел и векторов матрицы 3×3.

Глава 4

Основные геометрические алгоритмы в пространстве

В этой главе мы рассмотрим некоторые распространенные и полезные алгоритмы в трехмерном случае вместе с примерами кода, демонстрирующими их использование.

Задание прямых и плоскостей в пространстве

Прямая в пространстве задается тем же самым параметрическим уравнением, что и на плоскости:

$$p(t) = p_0 + l \cdot t.$$

Здесь p_0 – некоторая точка на прямой, а l – направляющий вектор прямой (прямая будет параллельна этому вектору). Обычно используется единичный вектор l, тогда расстояние между двумя точками на прямой, соответствующими значениям параметра t_1 и t_2, будет равно $|t_1 - t_2|$.

Кроме того, по точке p_0 и направляющему вектору l можно записать и непараметрическое уравнение прямой:

$$\frac{x - x_0}{l_x} = \frac{y - y_0}{l_y} = \frac{z - z_0}{l_z}.$$

Используя это уравнение, можно легко найти расстояние от заданной точки q до этой прямой как минимальное расстояние между точками $p(t)$ и q. Для этого введем функцию $f(t)$ как квадрат расстояния между ними:

$$f(t) = ||p_0 + l \cdot t - q||^2 = (p_0 + l \cdot t - q, p_0 + l \cdot t - q).$$

После несложных преобразований приходим к

$$f(t) = (p_0 - q), p_0 - q) + 2t(p_0 - q, l) + t^2(l, l).$$

Эта функция достигает своего минимального значения при следующем значении параметра t:

$$t' = -\frac{(p_0 - q, l)}{(l, l)}.$$

Считая вектор l единичным, получаем следующее минимальное расстояние (в квадрате):

$f(t') = (p_0 - q, p_0 - q) - (p_0 - q, l)^2$.

Плоскость в пространстве задается при помощи вектора нормали n (который мы будем далее считать единичным) и расстояния со знаком до начала координат d в следующем виде:

$(p, n) + d = 0$.

Если мы знаем нормаль n и некоторую точку p_0, через которую проходит эта плоскость, то уравнение плоскости можно записать в следующем виде:

$(p - p_0, n) = 0$.

Тогда если у нас есть некоторая точка q, то расстоянием от нее до этой плоскости будет $|(q, n) + d)|$.

Также плоскость можно задать и следующим уравнением (на самом деле легко убедиться в том, что это просто покомпонентная запись предыдущего уравнения):

$ax + by + cz + d = 0$.

Проекция точки на прямую

Под проекцией произвольной точки на заданную прямую понимается ближайшая к ней точка на прямой. Фактически мы уже нашли ее в предыдущем параграфе – проекцией точки q на прямую $p_0 + l \cdot t$ будет следующая точка:

$$q' = p_0 + \frac{(q - p_0, l)}{(l, l)} l.$$

Проекция точки на плоскость

Под проекцией произвольной точки на заданную плоскость понимается ближайшая к ней точка на плоскости. На самом деле для нахождения этой проекции нам достаточно опустить из данной точки перпендикуляр к плоскости и найти точку пересечения этого перпендикуляра и плоскости.

Пусть плоскость задается уравнением $(p, n) + d = 0$ и вектор нормали n является единичным. Тогда ближайшая точка на плоскости к точке p_0 будет задаваться следующей формулой:

$p' = p_0 - ((p_0, n) + d) \cdot n$.

Задание прямой двумя точками. Задание плоскости тремя точками

Пусть в пространстве заданы две различные точки – a и b. Тогда существует единственная прямая, проходящая через эти точки. Эта прямая может быть задана при помощи следующего параметрического уравнения:

$p(t) = (1 - t)a + tb = a + t(b - a)$.

Если в этом уравнении ограничить значения t единичным отрезком [0, 1], то в результате мы получим не всю прямую, а только сам отрезок ab (соединяющий эти две точки).

Теперь пусть у нас заданы три различные точки a, b и c, не лежащие на одной прямой. Тогда существует единственная плоскость, проходящая через них. Для получения уравнения этой плоскости заметим, что векторы $b - a$ и $c - a$ должны быть параллельны этой плоскости. Тогда их векторное произведение будет нормалью к плоскости. Зная, что плоскость проходит через точку a, мы можем записать уравнение плоскости в следующем виде:

$$(p - a, [b - a, c - a]) = 0.$$

Проведение плоскости через прямую и точку

Пусть у нас есть прямая, заданная параметрическим уравнением $p(t) = p_0 + l \cdot t$, и некоторая точка q, не лежащая на этой прямой.

Рассмотрим вектор $q - p_0$. Он должен быть параллелен нашей плоскости (как и вектор l), кроме того, он не может быть параллелен l по условию. Тем самым у нас есть два вектора, параллельных плоскости и, следовательно, перпендикулярных ее нормали n. И мы можем определить эту нормаль следующей формулой:

$$n = [q - p_0, l].$$

Так как точка p_0 должна лежать на нашей плоскости, то уравнение плоскости можно записать в виде:

$$(p - p_0, n) = 0.$$

Проверка прямых и отрезков на параллельность и перпендикулярность. Нахождение углов между прямыми и отрезками

Пусть у нас есть два отрезка, заданных своими концами, – ab и cd. Тогда через каждый из этих отрезков будет проходить прямая, задаваемая следующими параметрическими уравнениями:

$$p_1(t) = a + (b - a)t;$$
$$p_2(t) = c + (d - c)t.$$

Углом между этими прямыми (отрезками) будет угол φ между векторами $b - a$ и $d - c$, который можно легко найти из их скалярного произведения:

$$\cos\varphi = \frac{(b-a, d-c)}{\|b-a\| \cdot \|d-c\|}.$$

Отрезки (прямые) будут перпендикулярны, если $(b - a, d - c) = 0$.

Проверка, лежит ли отрезок или прямая на заданной плоскости

Пусть у нас есть отрезок ab и некоторая плоскость, задаваемая уравнением $(p, n) + d = 0$. Этот отрезок лежит в заданной плоскости тогда и только тогда, когда оба его конца лежат на ней, т. е. когда $(a, n) + d = (b, n) + d = 0$.

Если же вместо отрезка ab у нас есть параметрически заданная прямая (через уравнение $a + lt$), то прямая будет лежать на плоскости, если точка a лежит на плоскости (т. е. $(a, n) + d = 0$) и отрезок l параллелен плоскости.

Вектор l параллелен плоскости тогда и только тогда, когда он перпендикулярен ее нормали, т. е. $(l, n) = 0$.

Проверка, пересекает ли отрезок/луч/прямая заданную плоскость

Пусть у нас есть некоторая плоскость π, заданная уравнением $(p, n) + d = 0$.

Рассмотрим прямую, заданную при помощи параметрического уравнения $p(t) = a + l \cdot t$, и найдем точку ее пересечения с плоскостью π. Для этого просто подставим уравнение прямой в уравнение плоскости:

$$0 = (a + l \cdot t, n) + d = (a, n) + d + t(l, n).$$

Отсюда при $(l, n) \neq 0$ мы получаем значение t', соответствующее точке пересечения:

$$t' = -\frac{d + (a, n)}{(l, n)}.$$

Найденное значение t' задает точку пересечения прямой и плоскости, которая сразу же может быть найдена подстановкой в уравнение прямой.

В случае $(l, n) = 0$ прямая параллельна плоскости. Тогда возможны два варианта. Если $(a, n) + d = 0$ (точка a лежит на плоскости), то прямая целиком лежит на плоскости и любая ее точка может выступать в качестве точки пересечения.

В случае $(a, n) + d \neq 0$ прямая параллельна плоскости, но не имеет с ней ни одной общей точки.

Теперь пусть у нас есть не прямая, а луч, задаваемый тем же самым уравнением $p(t) = a + l \cdot t$, но только мы рассматриваем лишь неотрицательные значения t, а сама точка a является началом луча.

Как и ранее, мы находим параметр точки пересечения t'. Если $t \geq 0$, то луч пересекает плоскость, в противном случае пересечения нет.

Пусть у нас теперь есть просто отрезок ab. Можно пойти ранее рассмотренным путем – получить уравнение прямой, найти параметр точки пересечения и проверить, лежит ли она на [0, 1]. Вместо этого мы покажем другой подход.

Сначала вычислим следующие два значения:

$f_a = (a, n) + d$;
$f_b = (b, n) + d$.

Если одно из этих значений равно нулю, то мы сразу получим нашу точку пересечения. Если оба равны нулю, то весь отрезок целиком лежит в плоскости.

Если же оба этих значения не равны нулю и имеют одинаковый знак, то оба конца отрезка лежат в одном и том же полупространстве и, следовательно, отрезок не может пересекать плоскость.

В случае когда эти значения имеют разные знаки, то мы можем легко найти точку пересечения из следующего параметрического уравнения:

$p' = (1 - t')a + t'b$.

Параметр t' легко находится по следующей формуле:

$$t' = \frac{f_a}{f_a - f_b}.$$

Отсюда мы получаем координаты точки пересечения:

$$p' = \frac{f_b}{f_b - f_a}a + \frac{f_a}{f_a - f_b}b.$$

Проверка, пересекает ли луч заданный треугольник

Пусть в пространстве задан некоторый треугольник abc и луч (или прямая) $p(t) = p_0 + l \cdot t$. Рассмотрим, каким образом мы можем найти точку пересечения луча/прямой с этим треугольником.

Произвольная точка p этого треугольника может быть представлена при помощи барицентрических координат (они отлично работают не только на плоскости, но и в пространстве):

$p = (1 - u - v)a + ub + vc, u, v \in [0, 1], u + v \in [0, 1]$.

Тогда задачу о нахождении пересечения луча с этим треугольником можно свести к следующей системе из трех линейных уравнений с тремя неизвестными (t, u и v):

$p_0 + lt = (1 - u - v)a + ub + vc$.

Введем специальные обозначения:

$e_1 = b - a$;
$e_2 = c - a$;
$s = p_0 - a$.

Тогда, используя правило Крамера, мы можем записать решение этой системы в следующем виде:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-l, e_1, e_2)} \begin{pmatrix} \det(s, e_1, e_2) \\ \det(-l, s, e_2) \\ \det(-l, e_1, s) \end{pmatrix}.$$

Здесь через $\det(a, b, c)$ обозначен определитель матрицы, составленный из заданных трех векторов. Можно убедиться, что этот определитель может быть записан в следующем виде:

$\det(a, b, c) = (-[a, c], b)$.

Тогда можно переписать формулу для решения системы как:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(x, e_1)} \begin{pmatrix} (e, e_2) \\ (x, s) \\ (y, l) \end{pmatrix}.$$

Здесь через x и y обозначены следующие векторы:

$x = [l, e_2]$;
$y = [s, e_1]$.

Найдя решение системы, мы проверяем, выполнены ли условия на барицентрические координаты: $u, v \geq 0$, $u, v \leq 1$, $u + v \leq 1$ – и на параметр t, в случае если ищется пересечение именно с лучом, а не с прямой. Весь этот алгоритм можно выразить при помощи следующей функции:

```
bool rayTriangleIntersect ( const ray& ray,
                            const glm::vec3& a,
                            const glm::vec3& b, const glm::vec3& c,
                            float& u , float& v, float& t )
{
      const glm::vec3 e1 = b - a;
      const glm::vec3 e2 = c - a;
      const glm::vec3 x  = glm::cross ( ray.dir, e2 );
      const float     d  = glm::dot ( e1, x );      // определитель системы

      if ( d > -EPS && d < EPS )
            return false;        // система не имеет решения

      const float       f = 1.0 / d;
      const glm::vec3 s = ray.org - a;

      u = f * glm::dot ( s, x );

      if ( u < 0 || u > 1 )
            return false;

      const glm::vec3 y = glm::cross ( s, e1 );

      v = f * glm::dot ( r.dir, y );

      if ( v < 0 || v > 1 || u + v > 1 )
            return false;

      t = f * glm::dot ( e2, y );

      return true;
}
```

Нахождение пересечения луча и OBB

Рассмотрим задачу нахождения пересечения луча $p(t) = p_0 + lt$, $t \geq 0$ и *OBB*.

Для начала введем важное понятие – *пластина* (slab). Это область пространства, заключенная между двумя параллельными плоскостями (рис. 4.1). Соответственно, уравнения этих плоскостей имеют общий вектор нормали n, но разные расстояния от начала координат d_1 и d_2.

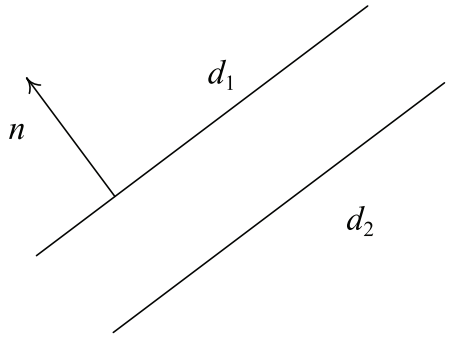

Рис. 4.1 ❖ Пластина

Для нахождения пересечения луча с таким объектом найдем два значения t – точку входа t_{min}, где луч входит внутрь этой области, и точку t_{max}, где луч выходит из нее. Эти точки находятся как точки пересечения луча и плоскости, при этом t_{min} – это минимальное значение из точек пересечения с данными плоскостями, а t_{max} – максимальное.

Тогда произвольный *OBB* можно представить как пересечение трех таких областей (рис. 4.2). Как и для двухмерной задачи нахождения пересечения отрезка и выпуклого многоугольника, мы находим пересечение луча со всеми шестью плоскостями и классифицируем эти точки как точки входа или точки выхода.

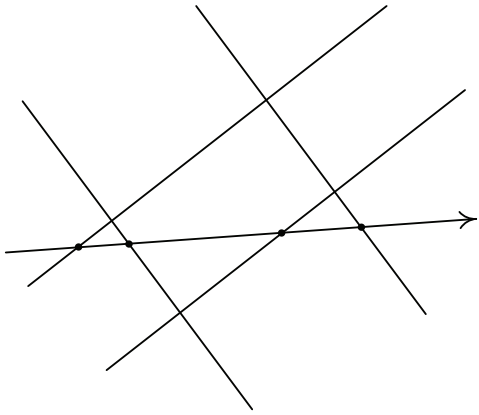

Рис. 4.2 ❖ Представление *OBB* как пересечение нескольких пластин

Тогда t_{min} будет наибольшая из всех точек входа, а t_{max} – наименьшая из всех точек выхода. Если $0 < t_{min} < t_{max}$, то луч имеет пересечение с данным OBB и точкой пересечения будет t_{min}. Если $t_{min} < 0 < t_{max}$, то это соответствует началу луча внутри *AABB* и мы получаем t_{max} как точку выхода луча.

Описанный алгоритм реализуется следующим фрагментом кода:

```
bool rayBoxIntersect ( const ray& ray, const obb& box, float& t )
{
      float tNear = -FLT_MAX;
```

```
        float tMax  = FLT_MAX;
        float t1, t2;

        for ( int i = 0; i < 3; i++ )
        {
              float vd = glm::dot ( ray.dir,    box.n [i] );
              float vo = glm::dot ( ray.origin, box.n [i] );

              if ( vd > EPS )             // t1 < t2, так как d1 [i] < d2 [i]
              {
                    t1 = -(vo + box.d2 [i]) / vd;
                    t2 = -(vo + box.d1 [i]) / vd;
              }
              else
              if ( vd < -EPS )            // t1 < t2, так как d1 [i] < d2 [i]
              {
                    t1 = -(vo + box.d1 [i]) / vd;
                    t2 = -(vo + box.d2 [i]) / vd;
              }
              else                        // abs ( vd ) < Threshold => луч параллелен пластине

              {
                    if ( vo < d1 [i] || vo > d2 [i] )
                          return false;
                    else
                          continue;
              }

              if ( t1 > tNear )
                    tNear = t1;

              if ( t2 < tFar )
                    if ( ( tFar = t2 ) < EPS )
                          return false;

              if ( tNear > tFar )
                    return false;
        }

        t = tNear;

        return t > EPS;
}
```

Нахождение пересечения луча и сферы

Пусть у нас задана сфера, описанная уравнением $||p - c|| = r$, а также задан луч, описанный уравнением $p(t) = p_0 + lt$.

Существует довольно простой геометрический метод нахождения пересечения луча со сферой. Для начала найдем расстояние от начала луча до ближайшей к центру сферы точки на луче. Этой точке соответствует значение параметра

$$t_0 = (c - p_0, l).$$

Далее найдем квадрат расстояния от этой точки до центра сферы:

$$d^2 = (c - p_0, c - p_0) - (c - p_0, l)^2.$$

Если $r^2 - d^2 < 0$, то луч проходит мимо сферы и пересечения нет. В противном случае по теореме Пифагора найдем расстояние m (рис. 4.3) и по нему соответствующие точкам пересечения значения параметра t:

$$m^2 = l^2 - d^2;$$
$$q = \sqrt{r^2 - m^2}.$$

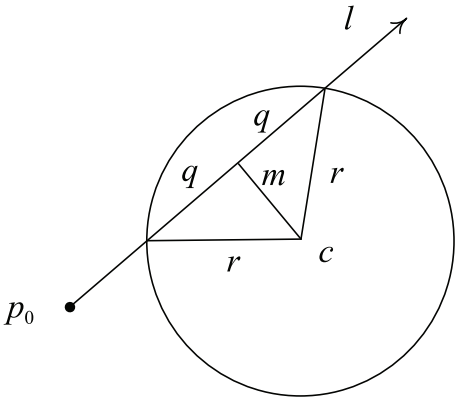

Рис. 4.3 ❖ Нахождение ближайшей точки пересечения луча со сферой

Ниже приводится соответствующий этому код на C++.

```
bool raySphereIntersect ( const ray& ray, const glm::vec3& c, float r, float& t )
{
      const glm::vec3 l  = c - ray.origin;            // вектор из начала луча к центру сферы
      const float     l2OC = glm::dot ( l, l );       // квадрат расстояния p0 - c
      const float     tca  = glm::dot ( l, ray.dir ); // кратчайшее расстояние
                                                      // от луча до центра
      float     t2hc = r*r - l2OC + tca*tca;
      float     t2;

      if ( t2hc <= 0.0f )
            return false;

      t2hc = sqrtf ( t2hc );

      if ( tca < t2hc )            // мы внутри сферы
      {
            t  = tca + t2hc;
            t2 = tca - t2hc;
      }
      else                         // мы снаружи сферы
      {
            t  = tca - t2hc;
            t2 = tca + t2hc;
      }

      if ( fabs ( t ) < EPS )
            t = t2;

      return t > EPS;
}
```

Проверка, пересекает ли плоскость заданную сферу

Пусть у нас есть плоскость, заданная уравнением $(p, n) + d = 0$, и сфера, заданная уравнением $||p - c|| = r$. Будем считать, что вектор нормали n в уравнении плоскости является единичным.

Тогда сфера будет пересекать плоскость тогда и только тогда, когда расстояние от ее центра c до этой плоскости меньше, чем ее радиус r. Тем самым для классификации сферы относительно плоскости мы просто подставляем координаты центра в уравнение плоскости.

Если $(c, n) + d > r$, то вся сфера целиком лежит в положительном полупространстве. Если $(c, n) + d < -r$, то вся сфера целиком лежит в отрицательном полупространстве. Иначе сфера пересекает плоскость.

Проверка, пересекает ли плоскость заданный *AABB*

Пусть, как и ранее, у нас есть плоскость, заданная уравнением $(p, n) + d = 0$, и *AABB*, заданный своими минимальными и максимальными координатами p_{min} и p_{max}.

Простейшим способом проверки классификации данного *AABB* относительно плоскости будет подстановка координат всех восьми его вершин в уравнение плоскости. Если знаки всех полученных значений совпадут, то *AABB* лежит по одну (определяемую этим знаком) сторону от плоскости и пересечения нет.

Однако есть и другой способ, позволяющий ограничиться проверкой всего двух вершин (вместо восьми). Этот способ особенно удобен, когда нужно проверить большое число *AABB* на пересечение с одной и той же плоскостью.

Исходя из направления вектора нормали n, можно легко найти две вершины, которые будут ближайшей и наиболее удаленной вершинами *AABB* относительно данной плоскости (рис. 4.4).

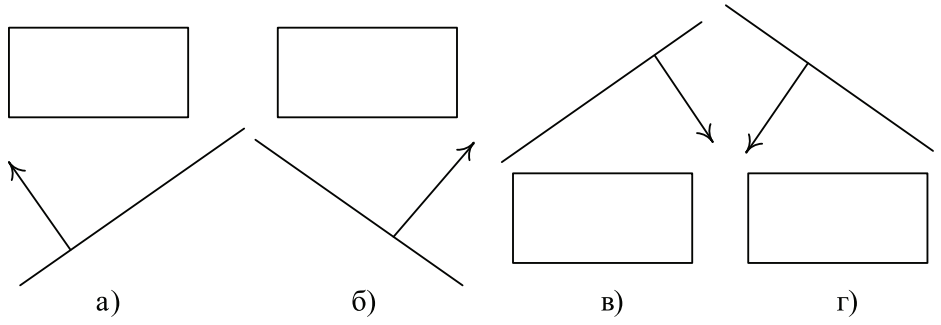

Рис. 4.4 ❖ Ближайшая и наиболее удаленная вершины *AABB* для плоскости

Как видно по рисунку, для каждой из трех координат мы выбираем значение из p_{min}, если соответствующая компонента вектора нормали отрицательна, и из p_{max}, если соответствующая компонента нормали положительна. В результате мы получаем ближайшую вдоль направления нормали вершину. Изменив схему выбора, мы получим вторую, наиболее удаленную вдоль направления нормали вершину.

Ниже приводится фрагмент кода, реализующий выбор ближайшей вершины:

```
inline glm::vec3 closestPoint ( const glm::vec3& n, const bbox: box )
{
      return glm::vec3 ( n.x <= 0 ? box.pMax.x : box.pMin.x,
                         n.y <= 0 ? box.pMax.y : box.pMin.y,
                         n.z <= 0 ? box.pMax.z : box.pMin.z );
}
```

Соответственно, для того чтобы *AABB* лежал по одну сторону от плоскости, необходимо и достаточно, чтобы обе эти вершины лежали в одном и том же полупространстве.

Телесный угол. Проверка на попадание в него

В трехмерной компьютерной графике часто возникает понятие *области видимости* (viewing frustum). Обычно такая область представляет из себя бесконечную или усеченную прямоугольную пирамиду (рис. 4.5).

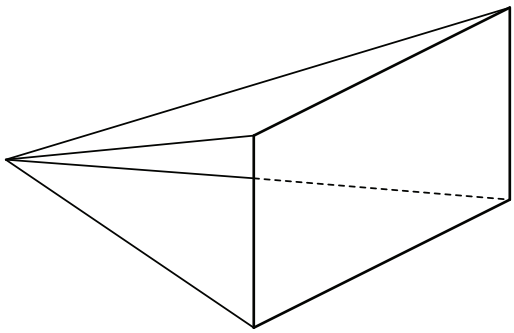

Рис. 4.5 ❖ Область видимости

Общим является то, что всегда такая область видимости определяется набором отсекающих плоскостей и является пересечением положительных полупространств, задаваемых этими плоскостями.

Давайте рассмотрим, каким образом мы можем проверить на попадание внутрь такой области точки, сферы и *AABB*.

Так как эта область представляет из себя пересечение набора положительных полупространств вида $(n_i, p) + d_i \geq 0$, то нам достаточно проверить, что проверяемый объект попадает внутрь каждого из участвующих полупространств, что мы уже раньше делали.

Определение, лежат ли две заданные прямые в одной плоскости

Пусть в пространстве у нас заданы две прямые при помощи своих параметрических уравнений $p(t) = a_i + l_i \cdot t$, $i = 1, 2$.

Если эти прямые не параллельны и лежат в одной и той же плоскости с нормалью n, то эта нормаль должна быть перпендикулярна каждому из направляющих векторов l_1 и l_2, т. е. мы можем считать, что $n = [l_1, l_2]$.

Параметр d уравнения этой плоскости легко можно найти, исходя из того что точка a_1 должна лежать в этой плоскости:

$d = -(n, a_1)$.

Тогда первая прямая будет полностью лежать в этой плоскости. Вторая прямая будет лежать в этой плоскости, только если a_2 лежит в ней, т. е. если $(n, a_2) = (n, a_1)$.

Осталось рассмотреть лишь случай, когда эти прямые параллельны (и $[l_1, l_2] = 0$). Но если прямые параллельны, то они всегда лежат в одной плоскости и нормаль к этой плоскости перпендикулярна как l_1, так и вектору $a_2 - a_1$. Тем самым мы находим требуемую нормаль как векторное произведение этих векторов. Коэффициент d находится так же, как и ранее.

Классификация двух прямых в пространстве

Пусть, как и в предыдущем пункте, у нас есть две прямые в пространстве и нам нужно классифицировать их – являются ли они параллельными, пересекающимися или скрещивающимися.

В качестве первого шага найдем

$n = [l_1, l_2]$.

Если $n = 0$, то эти прямые параллельны и осталось лишь проверить, не совпадают ли они. При совпадении вектор $a_2 - a_1$ должен быть параллелен l_1, т. е. их векторное произведение должно быть равно нулю. Если это не так, то эти прямые не совпадают.

Далее мы проверяем, не лежат ли эти прямые в одной плоскости, как уже было рассмотрено ранее. Если они не лежат в одной плоскости, то они являются скрещивающимися.

Если прямые лежат в одной плоскости и не параллельны, то они обязательно пересекаются в какой-то точке.

Нахождение расстояния между двумя прямыми в пространстве

Пусть, как и ранее, у нас есть две прямые в пространстве, заданные своими параметрическими уравнениями $p(t) = a_i + l_i t$, $i = 1, 2$.

Как и ранее, строим вектор нормали $n = [l_1, l_2]$. Если он равен нулю, то эти прямые параллельны и искомое расстояние равно расстоянию от a_2 до первой прямой.

Иначе мы строим пару параллельных плоскостей, задаваемых уравнениями $(p, n) + d_i = 0$. Здесь параметр $d_i = -(n, a_i)$. Каждая прямая лежит в соответствующей плоскости, и расстояние между этими прямыми равно расстоянию между этими плоскостями, которое есть

$$\frac{|d_1 - d_2|}{\|n\|}.$$

Проверка на пересечение двух треугольников в пространстве

Пусть у нас есть два треугольника в пространстве – T_1, заданный вершинами u_0, u_1, u_2, и T_2, заданный вершинами v_0, v_1, v_2. Эти треугольники лежат в плоскостях π_1 и π_2 соответственно. Нашей задачей является проверить эти треугольники на пересечение.

В качестве первого шага найдем параметры содержащих их плоскостей:

$n_1 = [u_1 - u_0, u_2 - u_0]$;
$d_1 = -(n_1, u_0)$;
$n_2 = [v_1 - v_0, v_2 - v_0]$;
$d_2 = -(n_2, v_0)$.

Далее мы находим расстояния и знаки для всех вершин T_1 до плоскости π_2 при помощи следующей функции:

$$d(u_i) = (n_2, u_i) + d_2.$$

Если все они отличны от нуля и имеют одинаковый знак, то треугольник T_1 целиком лежит по одну сторону от π_2 и пересечения нет.

В противном случае выполняем аналогичную проверку для T_2 и π_1. В результате этих двух проверок мы можем быстро отбросить целый ряд часто встречающихся случаев.

Если все $d(u_i) = 0$, $i = 0, 1, 2$, то оба треугольника лежат в одной плоскости. Этот случай мы рассмотрим чуть позже.

В противном случае плоскости π_1 и π_2 пересекаются, и их пересечением является некоторая прямая. В качестве направляющего вектора этой прямой будет выступать вектор $l = [n_1, n_2]$. Оба треугольника пересекают эту прямую (иначе они были бы отброшены ранее), и каждое из этих пересечений является отрезком этой прямой. Соответственно, треугольники будут пересекаться, только если пересекаются эти отрезки.

Рассмотрим теперь, каким образом можно найти пересечение T_1 и этой прямой (пересечение T_2 с прямой находится аналогично).

Пусть эта прямая задана уравнением $p(t) = p_0 + lt$. Тогда для начала спроектируем вершины u_0, u_1, u_2 на эту прямую при помощи следующей формулы (формула дает значение параметра вдоль прямой):

$$p(u_i) = (l, u_i - p_0).$$

Не ограничивая общности, будем далее считать, что вершины u_0 и u_2 лежат по одну сторону от π_2, а u_1 – по другую (рис. 4.6).

Найдем пересечение ребра u_0u_1 с этой прямой. Для этого достаточно найти значение параметра t, соответствующего этой точке пересечения.

Сам отрезок u_0u_1 можно также представить параметрически в виде $p(s) = u_0 + s(u_1 - u_0)$. Тогда уравнение для точки пересечения ребра и прямой может быть записано в виде:

$$u_0 + s(u_1 - u_0) = p_0 + lt.$$

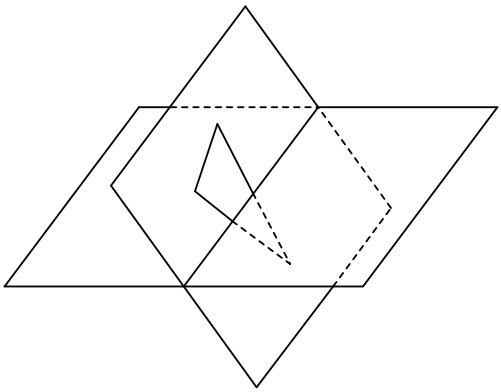

Рис. 4.6 ❖ Пересечение двух треугольников

Умножим это уравнение скалярно на n_2. Так как искомая точка лежит сразу в двух плоскостях, то $(p_0 + lt, n_2) = 0$. Отсюда мы сразу получаем уравнение для параметра s:

$$d(u_0) + s(d(u_1) - d(u_0)) = 0.$$

Отсюда легко находим s, а через него и параметр t. Аналогичным образом находится точка пересечения второго ребра нашего треугольника с прямой.

Найдя пересечения обоих треугольников с прямой, проверяем их на пересечение и получаем ответ.

Если оба треугольника лежат в одной плоскости, то мы можем легко свести задачу к двухмерной. Для этого давайте выберем координатную плоскость, для которой проекция треугольников на нее будет наибольшей (среди трех координатных плоскостей).

После этого мы просто выполняем обычный двухмерный тест на пересечение треугольников. Для этого мы сначала проверяем каждое ребро первого треугольника на пересечение с каждым ребром второго треугольника (как это делается, было уже рассмотрено в главе 2). Если данное пересечение есть, то эти два треугольника пересекаются.

В противном случае возможны два случая – треугольники либо вообще не пересекаются, либо один из них содержится полностью внутри другого. Для проверки этого берем произвольную вершину одного треугольника и проверяем на попадание внутрь другого треугольника. Аналогично берем вершину второго треугольника и проверяем на попадание внутрь первого треугольника. Если хотя бы в одном из этих случаев был получен положительный ответ, то эти треугольники пересекаются. В противном случае пересечения нет.

Глава 5

Структуры для работы с большими наборами геометрических данных

Во многих задачах приходится иметь дело с большими объемами геометрических данных. Типичной задачей, часто возникающей в подобных случаях, является нахождение всех объектов, удовлетворяющих какому-либо пространственному критерию.

К числу таких задач относится задача нахождения ближайшего пресечения заданного луча объектами, нахождение объектов, попадающих в заданную область пространства, нахождение всех пар объектов, расстояние между которыми меньше заданного, и многое другое.

Решение подобных задач «в лоб» (т. е. прямым перебором) требует затрат порядка $O(N^k)$, $k \geq 1$, что во многих случаях совершенно неприемлемо.

Похожая проблема возникает и при разработке различных баз данных, и там для решения этой проблемы используют так называемые *индексы*. Индекс – это некоторая специальная структура данных, позволяющая заметно снизить стоимость поиска объектов, удовлетворяющих заданных критериям. Одним из часто используемых индексов для баз данных являются Б-деревья, обеспечивающие логарифмическую скорость поиска (т. е. порядка $O(\log N)$).

В этой главе мы рассмотрим различные способы ускорения работы с большими объемами геометрических данных, включая наиболее распространенные *пространственные индексы* (spatial index), обеспечивающие скорость поиска от $O(\log^k n)$ до $O(1)$.

Ограничивающие тела

Часто возникает ситуация, когда нужно выполнить какую-либо проверку над сложным объектом или парой сложных объектов. В качестве подобной проверки может выступать проверка двух объектов на пересечение друг с другом или проверка объекта на пересечение с заданным лучом.

В случае сложных объектов (например, наборов из большого числа треугольников) выполнение подобной проверки «в лоб» может быть довольно дорогостоящим и, что самое важное, нередко вообще ненужным. Поэтому достаточно часто

вокруг каждого из таких объектов описывается некоторое простое тело, называемое *ограничивающим телом* (bounding volume, bounding box) (рис. 5.1).

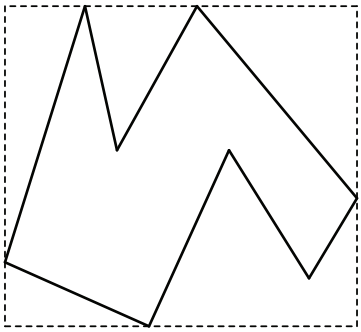

Рис. 5.1 ❖ Пример ограничивающего тела

Тогда вместо выполнения проверки над оригинальным объектом мы можем сначала выполнить проверку над ограничивающим телом (например, проверить на пересечение с заданным лучом). Если пересечения нет, то его не может быть и для исходного тела. В случае же, если есть пересечение с ограничивающим телом, то мы должны выполнить полноценную проверку для исходного объекта. Однако такой простой тест обычно позволяет сразу же отбросить подавляющее большинство случаев.

Подобное использование ограничивающих тел не позволяет полностью избавиться от явной проверки объектов, однако оно позволяет заметно снизить общее число проверок для сложных объектов, приводя тем самым к увеличению быстродействия.

Можно использовать различные типы ограничивающих тел. Минимальным требованием к ограничивающему телу является то, чтобы оно содержало исходный объект внутри себя. Также обычно желательно, чтобы ограничивающее тело было выпуклым. С этой точки зрения идеальным ограничивающим телом является выпуклая оболочка объекта.

Однако поскольку целью использования ограничивающих тел является увеличение быстродействия, то крайне важно, чтобы ограничивающее тело было достаточно простым, его можно было легко построить и для него легко можно было бы выполнять все необходимые проверки.

Поэтому выпуклая оболочка обычно является неудачным выбором – ее сложно строить и сложно выполнять необходимые проверки. Вместо нее обычно используют крайне простые выпуклые тела, содержащие исходный объект внутри себя. Ниже мы рассмотрим некоторые из наиболее распространенных ограничивающих тел.

Прямоугольный параллелепипед (AABB)

Одним из простейших видов ограничивающих тел являются прямоугольные параллелепипеды с ребрами, параллельными осям координат. Для их обозначения обычно используется термин *AABB* (Axis Aligned Bounding Box). Основным преимуществом этих ограничивающих тел является их крайняя простота (рис. 5.2).

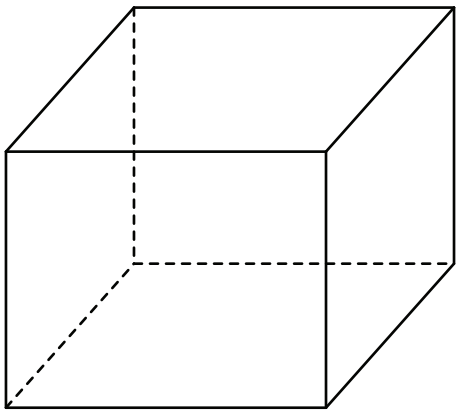

Рис. 5.2 ❖ Прямоугольный параллелепипед с ребрами, параллельными осям координат

Подобный объект в трехмерном случае можно представить при помощи экземпляров следующего класса:

```
class   bbox
{
      glm::vec3    minPoint;
      glm::vec3    maxPoint;

public:
      bbox ()
      {
            reset ();
      }

      bbox ( const glm::vec3& v1, const glm::vec3& v2 );
      bbox ( const bbox& b1, const bbox& b2 ) : minPoint ( b1.minPoint ),
                                                maxPoint ( b1.maxPoint )

      {
            merge ( b2 );
      }

      glm::vec3      getMinPoint () const
      {
            return minPoint;
      }

      glm::vec3      getMaxPoint () const
      {
            return maxPoint;
      }

      glm::vec3    getVertex ( int index ) const
      {
            return glm::vec3 ( index & 1 ? maxPoint.x : minPoint.x,
                               index & 2 ? maxPoint.y : minPoint.y,
                               index & 4 ? maxPoint.z : minPoint.z );
      }
```

```
glm::vec3      getCenter () const
{
      return (minPoint + maxPoint) * 0.5f;
}

glm::vec3      getSize () const
{
      return glm::vec3 ( maxPoint.x - minPoint.x,
                         maxPoint.y - minPoint.y,
                         maxPoint.z - minPoint.z );
}

bbox&     addVertex ( const glm::vec3& v );
bbox&   addVertices ( const glm::vec3 * v, int numVertices );

int     classify    ( const plane& plane ) const;

bool    isEmpty () const
{
      return minPoint.x > maxPoint.x || minPoint.y > maxPoint.y ||
             minPoint.z > maxPoint.z;
}

bool      contains ( const glm::vec3& pos ) const
{
      return pos.x >= minPoint.x && pos.x <= maxPoint.x &&
             pos.y >= minPoint.y && pos.y <= maxPoint.y &&
             pos.z >= minPoint.z && pos.z <= maxPoint.z;
}

bool      intersects ( const bbox& box ) const
{
      if ((maxPoint.x < box.minPoint.x) ||
          (minPoint.x > box.maxPoint.x) )
            return false;

      if ((maxPoint.y < box.minPoint.y) ||
          (minPoint.y > box.maxPoint.y) )
            return false;

      if ((maxPoint.z < box.minPoint.z) ||
          (minPoint.z > box.maxPoint.z) )
            return false;

      return true;
}

void    reset ()
{
      minPoint.x = FLT_MAX;
      minPoint.y = FLT_MAX;
      minPoint.z = FLT_MAX;
      maxPoint.x = -FLT_MAX;
      maxPoint.y = -FLT_MAX;
      maxPoint.z = -FLT_MAX;
}

void merge ( const bbox& box )
{
```

```
    if ( box.minPoint.x < minPoint.x )
       minPoint.x = box.minPoint.x;

    if ( box.minPoint.y < minPoint.y )
       minPoint.y = box.minPoint.y;

    if ( box.minPoint.z < minPoint.z )
       minPoint.z = box.minPoint.z;

    if ( box.maxPoint.x > maxPoint.x )
       maxPoint.x = box.maxPoint.x;

    if ( box.maxPoint.y > maxPoint.y )
       maxPoint.y = box.maxPoint.y;

    if ( box.maxPoint.z > maxPoint.z )
       maxPoint.z = box.maxPoint.z;
}

void      grow ( const glm::vec3& delta )
{
      minPoint -= delta;
      maxPoint += delta;
}

void      grow ( float delta )
{
      minPoint.x -= delta;
      minPoint.y -= delta;
      minPoint.z -= delta;
      maxPoint.x -= delta;
      maxPoint.y -= delta;
      maxPoint.z -= delta;
}

float volume () const
{
      glm::vec3      size = maxPoint - minPoint;

      return size.x * size.y * size.z;
}

float area () const
{
      glm::vec3      size = maxPoint - minPoint;

      return (size.x*size.y + size.x*size.z + size.y*size.z) * 2;
}

float      distanceTo ( const plane& plane ) const
{
      float      dist = plane.signedDistanceTo (
                        plane.makeNearPoint ( minPoint, maxPoint ) );

      if ( dist > 0 )
            return dist;

      dist = plane.signedDistanceTo ( plane.makeFarPoint ( minPoint, maxPoint ) );

      if ( dist < 0 )
```

```
                return dist;

            return 0;
      }
};
```

Если у нас есть объект, заданный при помощи набора вершин, то мы можем очень легко построить *AABB*, просто найдя минимальные и максимальные значения для каждой компоненты координат этих вершин.

```
bbox&      bbox :: addVertices ( const vec3 * v, int numVertices )
{
      for ( register int i = 0; i < numVertices; i++ )
      {
            if ( v [i].x < minPoint.x )
                  minPoint.x = v [i].x;

            if ( v [i].x > maxPoint.x )
                  maxPoint.x = v [i].x;

            if ( v [i].y < minPoint.y )
                  minPoint.y = v [i].y;

            if ( v [i].y > maxPoint.y )
                  maxPoint.y = v [i].y;

            if ( v [i].z < minPoint.z )
                  minPoint.z = v [i].z;

            if ( v [i].z > maxPoint.z )
                  maxPoint.z = v [i].z;
      }

      return *this;
}
```

Проверка подобных ограничивающих тел на пересечение между собой, а также с заданным лучом крайне проста и идентична двухмерному случаю, рассмотренному в главе 2.

Сфера

Еще одним простым и часто используемым ограничивающим телом является сфера, задаваемая центром и радиусом. Такая сфера может быть представлена при помощи экземпляров следующего класса:

```
class sphere
{
public:
      glm::vec3      center;
      float          radius;

      sphere ( const glm::vec3& c, float r ) : center ( c ), radius ( r ) {}
      sphere ( const glm::vec3 * v, int n );

      const glm::vec3 getCenter () const
      {
            return center;
      }
```

```
      float getRadius () const
      {
            return radius;
      }

      bool intersects ( const sphere& s ) const
      {
            return glm::length ( center - s.center ) <= radius + s.radius;
      }
};
```

Если у нас есть объект, заданный набором вершин v_i, то простейшим способом построения ограничивающей сферы для него будет следующий:

$$c = \frac{1}{n}\sum_{i=0}^{n-1} v_i;$$

$$r = \sqrt{\max\|c - v_i\|^2}.$$

Этот подход реализуется следующим конструктором:

```
sphere::sphere ( const glm::vec3 * v, int n )
{
      center = glm::vec3 ( 0 );
      radius = 0;

      for ( int i = 0; i < n; i++ )
            c += v [i];

      center /= (float) n;

      for ( int i = 0; i < n; i++ )
      {
            float r = glm::length ( v [i] - center );

            if ( r > radius )
                  radius = r;
      }
}
```

Проверка двух сфер на пересечение также крайне проста – пересечение имеет место тогда и только тогда, когда расстояние между центрами этих сфер не больше суммы их радиусов:

```
      bool intersects ( const sphere& s ) const
      {
            return glm::length ( center - s.center ) <= radius + s.radius;
      }
```

Проверка на пересечение сферы с заданным лучом реализуется фрагментом кода ранее рассмотренной функции `raySphereIntersect`.

k-DOP

Иногда *AABB* или сфера оказывается не очень удачным выбором, и требуется более точное приближение к форме заключенного внутри объекта. Хорошим вариантом такого ограничивающего тела, где можно легко управлять точностью приближения, является так называемый *k-DOP* (Descrete Oriented Polytope).

Возьмем четное число k и зафиксируем $k/2$ единичных векторов $n_0, \ldots, n_{\frac{k}{2}-1}$. Обратите внимание, что число $k/2$ должно быть больше размерности пространства. Чем больше k, тем выше точность приближения к исходному объекту.

Далее для каждого выбранного вектора n_i строим две плоскости $\pi_{i,0}$ и $\pi_{i,1}$, такие что наш объект будет заключен между ними и расстояние между этими плоскостями будет минимальным (рис. 5.3). При этом вектор n_i будет являться вектором нормали к обеим этим плоскостям. Тем самым эти плоскости будут описываться следующими уравнениями:

$(p, n_i) + d_{i,0} = 0;$
$(p, n_i) + d_{i,1} = 0.$

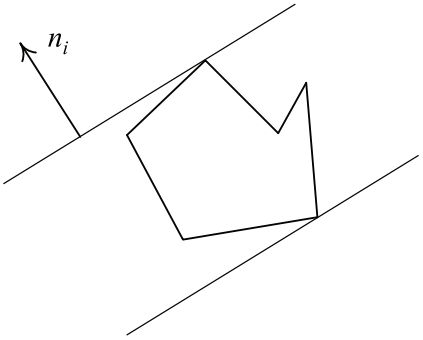

Рис. 5.3 ❖ Объект, заключенный между парой плоскостей

Область пространства, заключенная между такой парой плоскостей, обычно называется *slab*. Сам же *k-DOP* определяется как пересечение таких областей для всех векторов n_i (рис. 5.4).

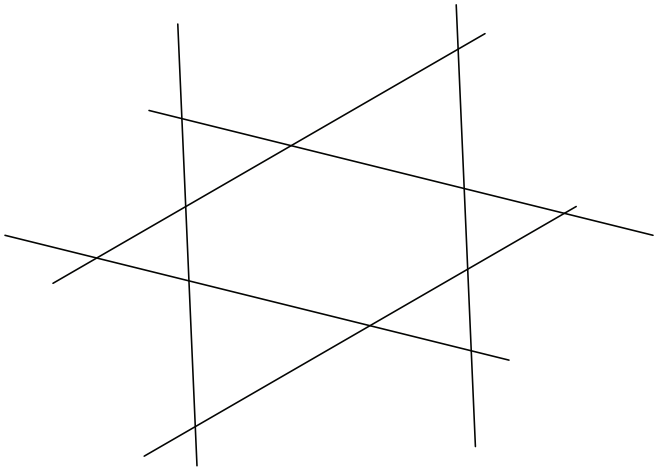

Рис. 5.4 ❖ *k-DOP* как пересечение slab'ов

Обычно мы фиксируем набор векторов n_i и для него строим различные *k-DOP*'ы. Поэтому нет никакой необходимости хранить эти векторы в каждом *k-DOP*'е. Именно с этой целью и вводится класс kdopBase. Сам же класс *k-DOP* можно легко представить следующим образом:

```
class kdopBase
{
public:
      enum
      {
            maxCount = 20
      };

private:
      glm::vec3      n [maxCount];
      int                  count;

public:
      kdopBase ( const glm::vec3 *, int );

      int getCount () const
      {
            return count;
      }

      const glm::vec3& getNormal ( int i ) const
      {
            return n [i];
      }
};

class kdop
{
      kdopBase * base;
      float      dMin [kdopBase::maxCount];
      float      dMax [kdopBase::maxCount];

public:
      kdop ( kdopBase * theBase );
      kdop ( const kdop& );

      void addVertices ( const glm::vec3 * v, int n );
      void merge       ( const kdop& );

      bool intersect ( const kdop& ) const;
      int  classify  ( const plane& ) const;
      bool intersect ( const ray& r, float& t1, float& t2 ) const;
};
```

Здесь класс kdopBase служит для задания набора векторов, служащих для построения *k-DOP*, а сам *k-DOP* представлен при помощи объектов класса kdop.

Ниже приводится функция, которая по заданному набору вершин строит *k-DOP*.

```
void kdop :: addVertices ( const glm::vec3 * v, int n )
{
      for ( int i = 0; i < n; i++ )
            for ( int j = 0; j < base -> getCount (); j++ )
            {
                  float      d = glm::dot ( v [i], base -> getNormal ( j ) );

                  if ( d < dMin [j] )
                        dMin [j] = d;

                  if ( d > dMax [j] )
                        dMax [j] = d;
            }
}
```

Процедура проверки этих ограничивающих тел (при условии что они опираются на один и тот же набор нормалей) крайне проста – для каждого вектора n_i мы проверяем, что соответствующие отрезки $[d_{i,0}, d_{i,1}]$ для обоих ограничивающих тел пересекаются. Если хотя бы для одного вектора n_i пересечения нет, то нет и пересечения *k-DOP*'ов.

```
bool kdop :: intersects ( const kdop& k ) const
{
      assert ( base == k.base );

      for ( int i = 0; i < base -> getCount (); i++ )
      {
            if ( k.dMax [i] < dMin [i] )
                  return false;

            if ( dMax [i] > k.dMax [i] )
                  return false;
      }

      return true;
}
```

Рассмотрим теперь, как найти пересечение *k-DOP*'а с заданным лучом. Для этого для каждого вектора n_i найдем параметр t, соответствующий точкам пересечения луча с плоскостями $(p, n_i) + d_{i,0} = 0$ и $(p, n_i) + d_{i,1} = 0$. Минимальный из них становится точкой «входа», максимальный – точкой «выхода».

Далее среди всех точек входа выбираем наибольшую, среди всех точек выхода – наименьшую. Эти две точки и будут задавать часть луча, лежащую внутри заданного *k-DOP*'а. Если соответствующий интервал пуст, то пересечения нет (рис. 5.5).

Ниже приведен соответствующий код.

```
bool kdop :: intersects ( const ray& r, float& t1, float t2 ) const
{
      t1 = FLT_MAX;
      t2 = -FLT_MAX;
```

```
    for ( int i = 0; i < base->getCount (); i++ )
    {
        float tEnter = -( dMin [i] + glm::dot ( ray.origin,
                    base->getNormal ( i ) ) ) /
                    glm::dot ( ray.dir, base -> getNormal ( i ) );
        float tExit  = -( dMax [i] + glm::dot ( ray.origin,
                    base->getNormal ( i ) ) ) /
                    glm::dot ( ray.dir, base -> getNormal ( i ) );

        if ( tEnter > tExit )
            std::swap ( tEnter, tExit );

        if ( tEnter > t1 )
            t1 = tEnter;

        if ( tExit < t2 )
            t2 = tExit;
    }

    return t1 <= t2;
}
```

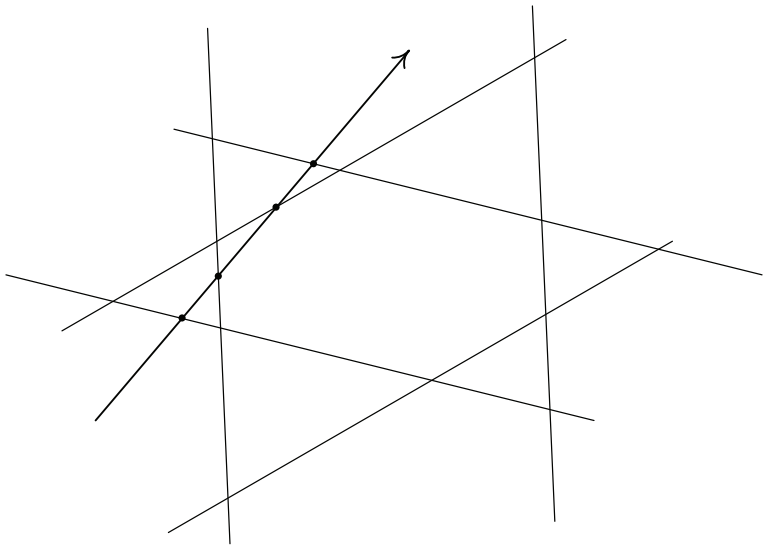

Рис. 5.5 ❖ Пересечения луча и k-DOP'а

Ориентированные ограничивающие прямоугольные параллелепипеды (OBB)

Еще одним типом ограничивающих тел являются прямоугольные параллелепипеды произвольной ориентации – *OBB* (Oriented Bounding Box). Такие ограничивающие тела могут давать хорошее приближение, но работа с ними более сложна за счет того, что у каждого *OBB* свои направления ребер.

Каждый такой OBB задается при помощи следующего уравнения:

$p = c + (e_0|e_1|e_2)y, |y_i| \leq s_i, i = 0, 1, 2.$

Здесь c – это центр OBB, e_0, e_1 и e_2 – единичные направляющие вектора для его ребер, а s_0, s_1 и s_2 – размер OBB вдоль его ребер (рис. 5.6).

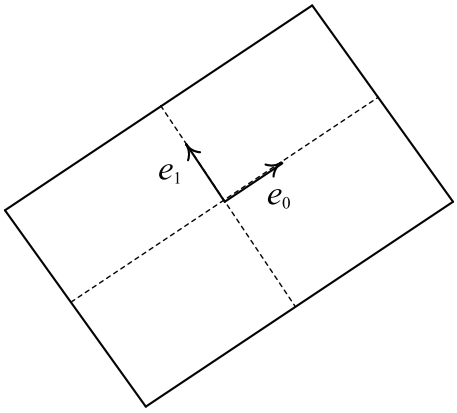

Рис. 5.6 ❖ Ориентированный ограничивающий прямоугольный параллелепипед (*OBB*)

Ниже приводится класс, который можно использовать для представления *OBB*.

```
class obb
{
public:
      glm::vec3 center;            // середина OBB
      glm::vec3 e [3];             // направления ребер и их длины

      obb ( const glm::vec3& c, const glm::vec3& e0,
            const glm::vec3& e1, const glm::vec3& e2 );
      obb ( const glm::vec3 * v, int n );

      bool intersect ( const obb& other ) const;
      bool intersect ( const ray& r, float& t1, float& t2 ) const;
};
```

Построение *OBB* по заданному набору вершин более сложно, чем для ранее рассмотренных ограничивающих тел. В качестве центра OBB, как и ранее, берется среднее арифметическое всех вершин:

$$c = \frac{1}{n}\sum_{i=0}^{n-1} v_i.$$

Однако нахождение векторов e_0, e_1 и e_2 заметно сложнее – для этого строится *матрица ковариации* (covariance matrix) M по следующей формуле:

$$M = \frac{1}{n}\sum_{i=0}^{n-1}(v_i - c)(v_i - c)^T.$$

Легко убедиться в том, что матрица M симметрична, поэтому у нее есть три попарно ортогональных собственных вектора e_0, e_1 и e_2, которые и используются как направления для ребер, далее мы будем считать их единичными. Тогда мы легко можем найти недостающие коэффициенты s_0, s_1 и s_2:

$s_i = \max|(e_i, v_j - c)|$.

Для нахождения собственных векторов матрицы 3×3 можно воспользоваться библиотекой `eig3`.

Рассмотрим теперь нахождение пересечения *OBB* с различными объектами. Наше рассмотрение мы начнем с плоскости, задаваемой уравнением $(p, n) + d = 0$. Для этого мы спроектируем OBB на прямую с направляющим вектором n (рис. 5.7).

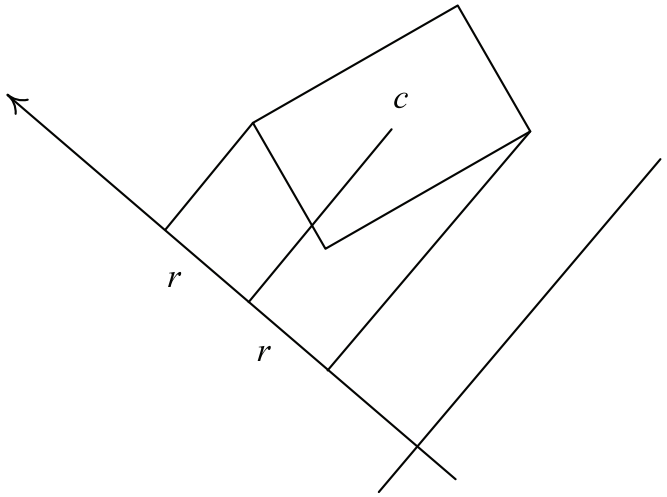

Рис. 5.7 ❖ Проекция *OBB* на нормаль к плоскости

Подобная проекция обычно определяется проекцией центра $\acute{c}$ и радиусом r, задаваемым следующей формулой:

$$r = \sum_{i=0}^{2} s_i^2 |(e_i, n)|.$$

Если $r < |(\acute{c}, n) + d|$, то *OBB* не пересекается данной плоскостью и лежит в том же полупространстве, что и его центр.

Нахождение пересечения *OBB* с лучом практически аналогично нахождению пересечения луча с *k-DOP*'ом (у нас есть всего три пары плоскостей, нормалями к которым выступают векторы e_0, e_1 и e_2).

Гораздо более интересной является задача определения пересечения двух *OBB*. В этом случае используется так называемая *теорема о разделяющей плоскости* (separating axis theorem). Согласно этой теореме два выпуклых объекта не пересекаются тогда и только тогда, когда существует такая прямая, что их проекции на эту прямую не пересекаются (рис. 5.8).

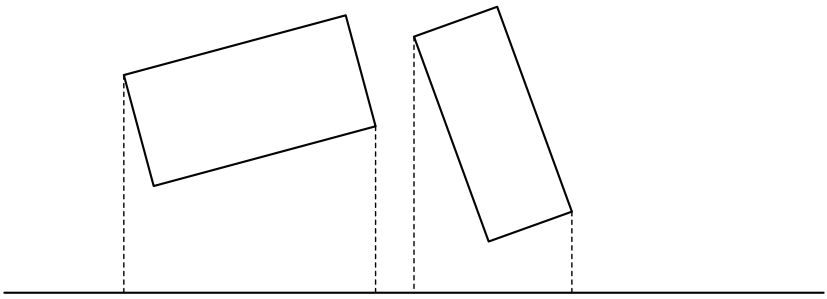

Рис. 5.8 ❖ Проекции двух *OBB* на прямую

Для проверки на пересечение двух *OBB* нам достаточно проверить всего 15 прямых (направлений) – по три направления дает каждый *OBB*, и еще девять направлений получаем в результате векторных произведений ребер этих *OBB*.

Поскольку мы будем работать только с проекциями OBB напрямую, то нам нужна не сама прямая, а всего лишь ее направляющий вектор l. Тогда, зная этот вектор, мы легко можем найти расстояние между проекциями центров c_1 и c_2 и проекции размеров OBB на эту прямую:

$$r_1 = \sum_{i=0}^{2} \left|s_i^1\right| \cdot \left|(e_i^1, l)\right|;$$

$$r_s = \sum_{i=0}^{2} \left|s_i^2\right| \cdot \left|(e_i^2, l)\right|.$$

Тогда проекции OBB на соответствующую прямую будут пересекаться, только если $|(c_1 - c_2, l)| \leq r_1 + r_2$.

Обычно сначала выполняется проверка на направления, параллельные ребрам OBB, поскольку в этом случае мы получаем более простые формулы. Пусть, например, проверка выполняется для направления $l = e_i^1$. Тогда для r_1 и r_2 мы получаем следующие уравнения:

$$r_1 = \left|s_i^1\right|;$$

$$r_s = \sum_{i=0}^{2} \left|s_j^2\right| \cdot \left|(e_j^2, e_i^1)\right|.$$

Построив матрицу R, такую что $r_{ij} = |(e_i^1, e_j^2)|$, мы можем переписать выражения для r_2 в следующем виде:

$$r_2 = \sum_{j=0}^{2} \left|s_j^2\right| \cdot r_{ij}.$$

Если эти шесть проверок не позволили дать ответ на вопрос о пересечении, то необходимо выполнить оставшиеся 9 проверок на направления вида $[e_i^1, e_i^2]$.

В данном случае вектор l будет перпендикулярен векторам e_i^1 и e_j^2. Кроме того, мы можем воспользоваться следующим равенством:

$|(l, e_k^1)| = |([e_i^1, e_j^2], e_k^1)| = |([e_i^1, e_k^1], e_j^2)|$.

Но $[e_i^1, e_k^2]$ будет давать либо нулевой вектор (в случае $i = k$), либо вектор e_m^1, где $m = 3 - i - k$. Таким образом, мы можем записать

$$|(l, e_k^1)| = r_{3-i-k,j}|([l, e_k^1]| = r_{3-i-k,j}.$$

В результате мы упрощаем все требуемые проверки.

Иерархические структуры

Хотя использование ограничивающих тел и снижает стоимость проверок, тем не менее для случая, когда у нас имеется большое количество объектов, нужно что-то еще. По аналогии с использованием иерархических структур в традиционных задачах программирования, в этом случае также можно построить соответствующие иерархические структуры данных, способные кардинально снизить стоимость выполнения запросов. Одной из простейших таких структур данных является иерархия (дерево) ограничивающих тел.

Иерархия ограничивающих тел

Пусть у нас есть множество различных объектов (фигур), над которыми мы хотим выполнять различные запросы, например нахождение ближайшего пересечения с заданным лучом.

Нашим первым шагом будет построение ограничивающего тела, например *AABB*, вокруг всех объектов нашего множества. Далее мы можем проверить наш луч на пересечение с этим общим *AABB*.

Если пересечения нет, то луч автоматически не может пересекать ни один из наших объектов, содержащихся внутри *AABB*. Однако в случае, когда луч пересекает *AABB*, все не так просто.

Поэтому давайте разобьем все исходное множество объектов на несколько групп, так чтобы близко расположенные объекты попали в одну и ту же группу (рис. 5.9).

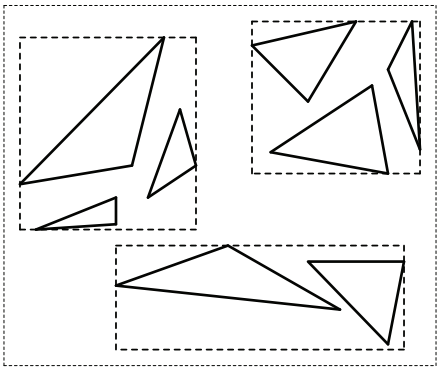

Рис. 5.9 ❖ Разбиение объектов на отдельные группы

Для каждой из этих групп построим свое ограничивающее тело. Теперь мы можем проверить луч на пересечение с ограничивающим телом для каждой такой группы. Если луч не пересекает ограничивающее тело группы, то он не может

и пересекать содержащиеся внутри объекты, и мы можем сразу же отбросить соответствующую группу.

Если же луч пересекает ограничивающее тело одной из групп, то мы опять разобьем все объекты этой группы на подгруппы и построим для каждой такой подгруппы свое ограничивающее тело.

Ясно, что подобный процесс можно продолжать довольно долго – обычно до тех пор, пока в подгруппе не останется меньше заданного числа объектов. В этом случае мы просто выполняем проверку для каждого из этих объектов.

В результате мы получили иерархическую структуру (дерево), состоящую из объектов и ограничивающих тел. Каждый узел такой структуры содержит ограничивающее тело и список дочерних узлов/объектов (явно хранить полный список содержащихся объектов имеет смысл только для листьев полученного дерева, внутренние узлы обычно хранят список дочерних узлов).

Узел подобной структуры данных может быть описан при помощи следующего класса (параметром шаблона является класс объектов, хранящихся внутри, каждый такой объект должен иметь метод `getBounds`, возвращающий его ограничивающее тело).

```
template <class Shape>
class BvhNode
{
protected:
      bbox  box;        // Ограничивающий прямоугольник
      bool  leaf;       // Является ли листом дерева
      union
      {
            std::vector<BvhNode *> children;
            std::vector<Shape *>   shapes;
      } data;

public:
};
```

Такая иерархическая структура данных получила название *иерархии ограничивающих тел* (*BVH*, Bounding Volume Hierarchy). Она позволяет выполнять запросы за $O(\log n)$.

Тетрарные и восьмеричные деревья

Еще одним распространенным типом иерархических структур являются тетрарные и восьмеричные деревья. Для них, как и ранее, вначале строится ограничивающее тело (*AABB*), описанное вокруг всех объектов исходного множества. Далее это тело разбивается на 8 частей (4 для тетрарного дерева), и для каждой из этих частей строится список всех объектов, полностью или частично попадающих в получившуюся часть (рис. 5.10).

Таким образом, для корня дерева строится восемь дочерних узлов. Для каждого из них по списку объектов мы строим ограничивающее тело (оно может оказаться больше исходной части ограничивающего тела для родительского узла). Если количество объектов превышает некоторое заранее заданное значение, то мы опять разбиваем ограничивающее тело на восемь частей, для каждой из этих частей строим список объектов и т. д. В результате мы получаем восьмеричное дерево.

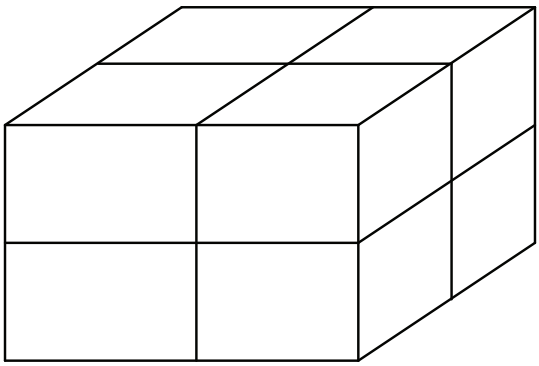

Рис. 5.10 ❖ Разбиение *AABB* для построения восьмеричного дерева

Если рассматриваемая задача двухмерна (или существенно двухмерна, например как объекты, размещенные на карте), то обычно вместо восьмеричного дерева используется тетрарное дерево. В этом случае *AABB* узла разбивается на четыре равные части.

Такие деревья позволяют получить логарифмическую скорость поиска. Также они легки в построении. Однако то, что на каждом шаге выполняется разбиение по каждой из осей координат, не всегда оказывается удобным. Так, в случае когда размер *AABB* по одной оси заметно меньше размера по остальным осям, это приведет к сильному измельчению. Вдоль данной оси и у нас окажется много узлов с одинаковыми списками объектов. Этого недостатка лишены *kD*-деревья, описываемые далее.

kD-деревья

Еще одной иерархической структурой, часто используемой для ускорения поиска геометрических данных, являются *kD-деревья* (в двухмерном случае для них иногда используется термин *xy*-дерево).

Как и ранее, каждый узел такого дерева содержит *AABB* для содержащихся в нем объектов. Однако на каждом шаге *AABB* делится всего на две части плоскостью, параллельной одной из координатных плоскостей.

Одним из простейших способов выбора разбивающей плоскости является чередование направлений, вдоль которых происходит разбиение. Тогда мы фактически приходим к аналогу восьмеричного дерева, но при этом более просто устроенному.

После того как *AABB* текущего узла разбивается на две части, составляются списки объектов, хотя бы частично попадающих в каждую из частей. Далее по каждому из этих списков строится свой узел дерева, который будет дочерним узлом для текущего узла.

Иногда встречается вариант *kD*-дерева, в котором все объекты разбиваются не на два класса, а на три – два класса образуют объекты, строго лежащие по одну из сторон от разбивающей плоскости. Третий класс образуют объекты, пересекающие разбивающую плоскость. Этот прием позволяет уменьшить размер ограничивающих тел для дочерних узлов.

Для представления узлов подобного дерева мы будем использовать объекты следующего класса:

```
template <class Shape>
class kDNode
{
protected:
      bbox             box;         // AABB для данного узла
      plane            p;           // разбивающая плоскость
      kDNode           * pos;       // узел в положительном подпр-ве
      kDNode           * neg;       // узел в отрицательном подпр-ве
      bool             isLeaf;
      std::list<Shape *> shapes;    // список фигур (для листа)
public:

      kDNode ( const bbox& b ) : box ( b )
      {
            pos    = nullptr;
            neg    = nullptr;
            isLeaf = true;
      }

      kDNode ( const std::list<Shape *>& lst, const bbox& b ) :
               shapes ( lst ), box ( b )
      {
            pos    = nullptr;
            neg    = nullptr;
            isLeaf = true;
      }

      static kDNode * buildKdTree ( const std::list<Shape *>& sh );

private:
      static plane splitBox    ( const bbox& b ) const;   // выбрать разбивающую плоскость

      static void  splitShapes ( const std::list<Shape *>& src,  // создать списки фигур
                                 std::list<Shape *>& p,          // для положительной
                                 std::list<Shape *>& n );        // и отрицательной областей

                                       // построить AABB для списка фигур
      static bbox  buildBox ( const std::list<Shape *>& lst ) const;

                                       // найти список всех фигур, пересекающих данный AABB
      void findShapes ( const bbox& area,
                        std::unordered_set<Shape *>& result ) const;

      Shape * findIntersection ( const ray& r ) const;  // вернуть первое пересечение с лучом
};
```

Одной из наиболее распространенных стратегий для выбора разбивающей плоскости (кроме уже упомянутого чередования направлений) является выбор самого длинного ребра *AABB*. После этого данное ребро просто делится на две равные части плоскостью, перпендикулярной ему. Подобный подход позволяет

корректно обрабатывать случаи сильно вытянутых (или, наоборот, сильно сжатых по одному направлению) ограничивающих тел. Ниже приводится метод splitBox, возвращающий разбивающую плоскость для такого подхода.

```
template <class Shape>
plane kDNode::splitBox ( const bbox& b ) const
{
      glm::vec3      n ( 1, 0, 0 );
      glm::vec3      size   = b.getSize   ();
      glm::vec3      center = b.getCenter ();
      int            iMax   = 0;
      float          vMax   = size.x;

      if ( size.y > vMax )
      {
            iMax = 1;
            vMax = size.y;
      }

      if ( size.z > vMax )
      {
            iMax = 2;
            vMax = size.z;
      }

      if ( iMax == 1 )
            n = glm::vec3 ( 0, 1, 0 );
      else
      if ( iMax == 2 )
            n = glm::vec3 ( 0, 0, 1 );

      float d = -glm::dot ( center, n );

      return plane ( n, d );
}
```

Важным преимуществом *kD*-деревьев является то, что у каждого узла есть всего два потомка, и они легко упорядочиваются вдоль любого заданного направления. Это позволяет нам проверять узлы в определенном порядке (например, при нахождении ближайшего пересечения с заданным лучом).

Ниже приводится текст метода buildKdTree, служащий для построения *kD*-дерева по заданному списку объектов.

```
template <class Shape>
kDNode * kDNode::buildKdTree ( const std::list<Shape *>& lst )
{
      bbox b ( buildBox ( lst ) );

      if ( lst.size () < MAX_CHILDS )    // для небольшого числа узлов разбиения не производим
            return new kDNode ( lst, b );

      plane              p ( splitBox ( b) );
      std::list<Shape *> ps;
```

```
      std::list<Shape *> ng;

      splitShapes ( lst, ps, ng );

      if ( ps.empty () || ng.empty () )
            return new kDNode ( lst, b );

      kDNode * root = new kDNode ( b );

      root->p      = p;
      root->pos    = buildKdTree ( ps );
      root->neg    = buildKdTree ( ng );
      root->isLeaf = false;

      return root;
}
```

С помощью *kD*-деревьев можно легко находить объекты, удовлетворяющие заданным пространственным критериям, например пересекающие заданное ограничивающее тело.

```
template <class Shape>
void kDNode::findShapes ( const bbox& area,
                          std::unordered_set<Shape *>& result ) const
{
      if ( !box.intersects ( area ) )
            return;

      if ( pos != nullptr )
            pos->findShapes ( area, result );

      if ( neg != nullptr )
            neg->findShapes ( area, result );

      for ( auto it : shapes )
            if ( area.intersects ( (*it) ->getBounds () ) )
                 result.insert ( *it );
}
```

Также несложно реализовать и метод, который будет находить ближайшее пересечение луча с объектами (но для этого требуется, чтобы каждый объект класса Shape умел проверять себя на пересечение с лучом).

```
Shape * kDNode::findIntersection ( const ray& r ) const
{
      if ( !box.intersects ( area ) )             // пересечения вообще нет
            return nullptr;

      Shape * ptr = nullptr;

      if ( glm::dot ( p.n, r.getDir () ) >= 0 )   // найдем ближайшего к началу луча потомка
      {           // 1-й положительный, 2-й отрицательный
            if ( pos != nullptr )
```

```
                ptr = pos->findIntersection ( r );

          if ( ptr != nullptr )
                return ptr;

          if ( neg != nullptr )
                ptr = neg->findIntersection ( r );

          if ( ptr != nullptr )
                return ptr;
    }
    else        // 1-й отрицательный, 2-й положительный
    {
          if ( neg != nullptr )
                ptr = neg->findIntersection ( r );

          if ( ptr != nullptr )
                return ptr;

          if ( pos != nullptr )
                ptr = pos->findIntersection ( r );

          if ( ptr != nullptr )
                return ptr;
    }

    float      tMin = FLT_MAX;
    float      t;

    for ( auto it : shapes )
          if ( (*it) ->getBounds ().intersects ( r ) )
                if ( (*it)->findIntersection ( r, t ) )
                      if ( t < tMin )
                      {
                            tMin = t;
                            ptr  = *it;
                      }

    return ptr;
}
```

В качестве одного из способов для выбора разбивающей плоскости может выступать *SAH* (Surface Area Heuristic). Этот критерий выбора пришел из трассировки лучей (глава 10) и пытается оптимизировать дерево именно с точки зрения трассировки лучей. Однако его использование позволяет строить очень хорошие деревья не только для трассировки лучей.

Схема для использования *SAH* довольно проста. Мы рассматриваем набор плоскостей (обычно идущих с постоянным шагом вдоль одной оси). Для каждой такой плоскости вычисляется значение специальной функции. Выбирается та плоскость, для которой это значение будет минимальным. Сама эта функция вычисляется по следующей формуле:

$$f(\pi) = C_t + C_i \frac{S_L(\pi) \cdot N_L(\pi) + S_R(\pi) \cdot N_R(\pi)}{S_{parent}}.$$

В этой формуле были использованы следующие обозначения:

C_t – стоимость трассировки луча внутри узла;

C_i – стоимость пересечения объекта лучом;

$S_L(\pi)$, $S_R(\pi)$ – площадь поверхности левого и правого дочерних узлов, образующихся при разбиении текущего узла плоскостью π;

$N_L(\pi)$, $N_R(\pi)$ – число объектов в левом и правом дочерних узлах при разбиении плоскостью π;

S_{parent} – площадь поверхности текущего (разбиваемого) узла.

BSP-деревья

Есть еще один тип деревьев для ускорения поиска, получивший название *бинарных деревьев разбиения пространства* (BSP, Binary Space Partitioning). Они очень похожи на *kD*-деревья, но есть и заметные отличия. В частности, *BSP*-деревья строятся не по произвольному набору объектов, а по набору граней.

Именно *BSP*-деревья лежали в основе технологии таких легендарных игр, как Doom и Quake. Многие игровые движки до сих пор используют *BSP*-деревья.

BSP-дерево позволяет осуществить упорядочение граней, при этом для гарантии корректности такого упорядочивания некоторые грани будут разбиваться на части. Основой для подобного упорядочивания является способность одной грани закрывать другую.

Давайте рассмотрим грани *A*, *B*, *C* и *D* (рис. 5.11). Проведем через грань *A* плоскость π и разделим все грани на несколько классов по положению к этой плоскости:

- грань лежит в положительном полупространстве (*B*);
- грань лежит в отрицательном полупространстве (*C*);
- грань лежит сразу в обоих полупространствах и, следовательно, пересекает нашу плоскость (*D*);
- грань лежит в разбивающей плоскости (*A*).

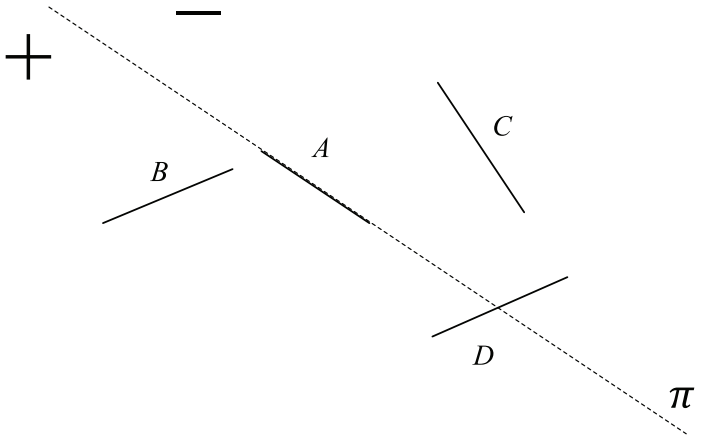

Рис. 5.11 ❖ Разбиение граней плоскостью на два множества

Если грань пересекает плоскость, то мы разобьем эту грань плоскостью на части (D_+ и D_-), каждая из которых лежит в одном полупространстве.

Таким образом, в результате проведения плоскости через грань A мы разбили все оставшиеся грани на два множества – лежащие в положительном полупространстве (B и D_+) и лежащие в отрицательном полупространстве (C и D_-).

Если у нас есть несколько граней, лежащих в плоскости разбиения, то можно отнести их к любому из этих двух классов или же вынести в отдельный класс.

Для каждого из получившихся таким образом множеств граней мы повторяем подобную процедуру до тех пор, пока не получим в результате разбиение на множества, состоящие из одной грани.

Получившаяся в результате структура может быть описана в виде бинарного дерева, узлом которого является грань, через которую проведена разбивающая плоскость. Таким образом, для граней с рис. 5.11 мы можем прийти к дереву, изображенному на рис. 5.12.

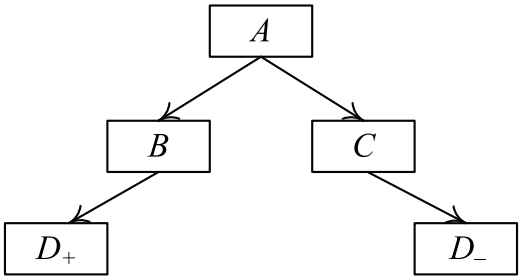

Рис. 5.12 ❖ Возможное дерево для граней с рис. 5.11

Удобно для каждого узла такого дерева хранить ограничивающее тело, построенное по всем граням из его поддеревьев. При этом для корня дерева ограничивающее тело строится по всему исходному множеству граней.

В результате мы приходим к следующему представлению узла дерева в виде класса:

```
class BspNode
{
      bbox    box;      // AABB для соответствующего поддерева (узла)
      plane   p;        // разбивающая плоскость для данного узла
      polygon * poly;   // полигон, по которому была построена плоскость
      BspNode * pos;
      BspNode * neg;
public:
      BspNode ( polygon * p, const bbox& b ) : box ( b ), poly ( p )
      {
            p   = p -> getPlane ();
            pos = nullptr;
            neg = nullptr;
      }

      static BspNode * buildTree    ( const std::list<polygon *>& polys );
      static polygon * bestSplitter ( const std::list<polygon *>& polys );
```

```
      static void      splitPolys   ( const std::list<polygon *>& polys,
                                      std::list<polygon *>& positive,
                                      std::list<polygon *>& negative );
};
```

Процедура построения *BSP*-дерева по заданному списку граней очень напоминает процедуру для построения *kD*-дерева:

```
BspNode * BspNode::buildTree ( const std::list<polygon *>& polys )
{
      polygon * splitter = bestSplitter ( polys );
      bbox    box        = buildBox ( polys );
      BspNode * root     = new BspNode ( splitter, box );

            // строим списки полигонов в положительном и отрицательном полупространствах
      std::list<polygon *> pos, neg;

      splitPolys ( root -> p, polys, pos, neg );

      if ( pos.empty () || neg.empty () )
            return root;

            // по каждому из них строим свое дерево и добавляем их как дочерние в текущий узел
      root->pos = buildTree ( pos );
      root->neg = buildTree ( neg );

      return root;
}
```

В этом коде для выбора грани, которая будет использована для построения разбивающей плоскости, используется функция `bestSplitter`. В простейшем случае эта функция просто возвращает первую грань из переданного списка. Более сложный вариант заключается в проведении анализа возможных вариантов. Есть два основных показателя, которые мы хотим минимизировать:

- разница в высоте левого и правого поддеревьев;
- число разбиваемых граней.

Обычно каждому из этих критериев дается некоторый вес. Проводить полный анализ всех возможных кандидатов обычно оказывается слишком дорого. Поэтому чаще выбирается несколько случайных граней из переданного набора, и среди них выбирается грань, дающая минимальное значение.

R-деревья

По аналогии с часто используемыми в различных базах данных *B*-деревьями для геометрических данных нередко применяются *R*-деревья (R от слова rectangle – прямоугольник). Это сбалансированное дерево обладает целым рядом полезных свойств. Так, для *R*-дерева порядка (*m*, *M*) справедливы следующие свойства:

- каждый лист дерева содержит от *m* до *M* записей. Запись состоит из самого объекта и его *AABB*;
- каждый внутренний узел (кроме корня дерева) содержит от *m* до *M* дочерних узлов вместе с их *AABB*;
- корень дерева содержит не менее двух узлов;

- все листья дерева расположены на одной и той же высоте от корня дерева;
- для каждого узла дерева его *AABB* содержит все объекты из соответствующего ему поддерева.

Узел такого дерева может быть представлен при помощи следующего класса:

```
template <class Shape>
class RTreeNode
{
      struct Record
      {
            bbox  box;          // AABB для всего поддерева
            union               // дочерний узел или объект
            {
                  RTreeNode * node;
                  Shape     * shape;
            };
      };

      bbox      box;            // AABB всего поддерева
      int       count;          // число дочерних узлов в данном узле
      Record  * recs;           // дочерние узлы или объекты
      bool      isLeaf;         // это лист или нет
      RTreeNode * parent;       // указатель на родительский узел
};
```

Процедура поиска объектов в *R*-дереве полностью аналогична поиску в ранее рассмотренных деревьях. Однако есть заметное отличие в процедуре вставки объекта в дерево.

Все дерево строится путем добавления объектов в него одного за другим. Во внутренние узлы дерева мы вставляем объект в тот дочерний узел, для которого увеличение объема его *AABB* будем минимальным.

Таким образом, мы опускаемся до листа дерева и вставляем объект в него. При этом может произойти переполнение листа (число объектов в нем станет больше, чем *M*). В этом случае лист расщепляется на два, и один узел добавляется в родительский узел. Это, в свою очередь, может привести к расщеплению родительского узла – и т. д. вплоть до корня.

```
void insertObject ( Shape * obj )
{
      RTreeNode * node = leafForShape ( obj->getBounds () );
      Record    * rec  = new Record ( obj );

      node->insert ( rec );
}

void insert ( Record * rec )
{
      if ( count < M )                  // переполнения нет
      {
            addRecord ( rec );
            box.merge ( rec -> box );
            updateParent ();
```

```
            return;
      }

      std::vector<Record>  r1, r2;      // разбиваем мн-во записей пополам

      split ( r1, r2 );

      if ( parent == nullptr )          // корень дерева
      {                                 // создаем два новых узла дерева
            RTreeNode * newNode1 = new RTreeNode;
            RTreeNode * newNode2 = new RTreeNode;

            newNode1->setRecs ( r1 );
            newNode2->setRecs ( r2 );

            Record * rec1 = new Record ( newNode1 );
            Record * rec2 = new Record ( newNode2 );

            std::vector<Record> root = { rec1, rec2 };   // корень состоит из {r1, r2}

            setRecs ( root );
      }
      else                              // обычный узел
{
      RTreeNode * newNode1 = new RTreeNode;

            setRecs      ( r1 );
            addRecord    ( rec );
            updateParent ();
            newNode1->setRecs ( r2 );   // создаем новый узел {r2}

            Record * rec1 = new Record ( newNode1 );

            parent->insert ( rec1 );    // добавляем его родителю
      }
}
```

Для разбиения списка всех *AABB* на два новых списка могут использоваться различные алгоритмы. Один из них сперва находит такую пару *AABB* b^+ и b^-, которую было бы крайне нежелательно помещать вместе в один список. Обычно это такая пара *AABB*, которая дает наибольшее значение выражения $V(b_1 \cup b_2) - V(b_1) - V(b_2)$, здесь через $V(b)$ обозначен объем *AABB* b (рис. 5.13).

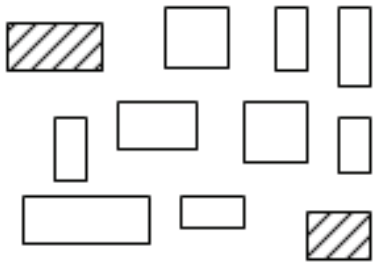

Рис. 5.13 ❖ Пример выбора b^+ и b^-

Эти два *AABB* берутся за основу нашего разбиения. Далее для каждого из оставшихся *AABB* b_i мы выбираем, к какой именно из двух групп его будет лучше добавить. Поскольку добавление *AABB* к группе может привести к увеличению объема *AABB* для всей группы, то выбирается та группа, для которой это увеличение объема будет минимальным.

Также есть отличие в процедуре удаления объекта из *R*-дерева. Проблема заключается в том, что при удалении объекта из листа дерева может возникнуть ситуация, когда в листе остается меньше *m* объектов. В этом случае используется простейшая стратегия – весь этот лист полностью удаляется. Затем все оставшиеся в нем объекты просто заново добавляются в *R*-дерево.

Равномерное разбиение пространства

Для ускорения поиска геометрических объектов, помимо иерархических структур данных, можно также использовать и более простую структуру – *равномерное разбиение пространства* (uniform grid) (рис. 5.14).

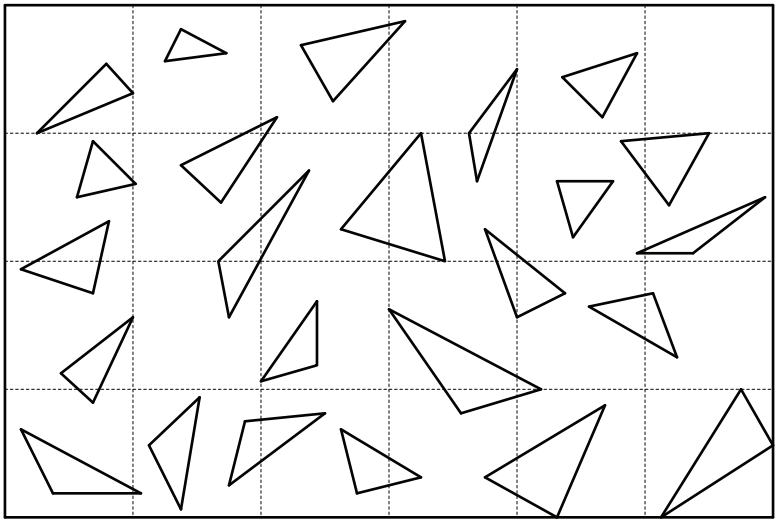

Рис. 5.14 ❖ Равномерное разбиение пространства

В этом случае, как и ранее, мы начинаем с того, что для заданного множества объектов строим ограничивающее тело (*AABB*). Далее это тело разбивается на $n_1 \times n_2 \times n_3$ одинаковых прямоугольных параллелепипедов. Для каждого из них составляется список всех объектов, полностью или частично попавших внутрь. Вся соответствующая структура данных может быть представлена при помощи следующего класса:

```
template<class Shape>
class RegularGrid
{
public:
      struct Node
      {
```

```
            std::list<Shape *> shapes;
      };

protected:
      int     n1, n2, n3;     // размер сетки в узлах
      bbox    box;            // AABB для всех объектов
      float   h1, h2, h3;     // размеры отдельных клеток по x,y и z
      Node  * nodes;          // массив из n1*n2*n3 клеток

      RegularGrid  ( int n1, int n2, int n3,
                     const std::list<Shape *>& shapes);
      ~RegularGrid ()
      {
            delete [] nodes;
      }

      Node * getNode ( int i, int j, int k ) const
      {
            return (i*n1 + j)*n2 + k;
      }

      bbox getNodeBox ( int i, int j, int k ) const
      {
            glm::vec3 start ( box.getMinPoint () +
                              glm::vec3 ( i*h1, j*h2, k*h3 ) );

            return bbox ( start, start + glm::vec3 ( h1, h2, h3 ) );
      }

                        // найти узел (клетку), содержащий точку
      void  nodeForPoint ( const glm::vec3& p, int& i, int&j , int&k ) const
      {
            i = (int)((p.x - box.getMinPoint ().x) / h1);
            j = (int)((p.y - box.getMinPoint ().y) / h2);
            k = (int)((p.z - box.getMinPoint ().z) / h3);
      }
                        // строим AABB для списка объектов
      static bbox  buildBox ( const std::list<Shape *>& polys );

                        // строим мн-во всех объектов, пересекающих заданный AABB
      std::unordered_set<Shape *> findShapes ( const bbox& area ) const;

                        // найти ближайшее пересечение с лучом
      Shape * findIntersection ( const ray& r, float& t ) const;
};
```

Пусть у нас есть подобная структура и необходимо найти все объекты, попавшие в заданный *AABB*. Для начала мы найдем те клетки разбиения, которые имеют непустое пересечение с заданным *AABB*. Далее для каждой такой клетки мы проверим все объекты из ее списка на пересечение с заданной *AABB*.

```
std::unordered_set<Shape *> RegularGrid::findShapes (
            const bbox& area ) const
```

```
{
      std::unordered_set<Shape *> result;
      int                         i1, j1, k1, i2, j2, k2;

      nodeForPoint ( area.getMinPoint (), i1, j1, k1 );
      nodeForPoint ( area.getMaxPoint (), i2, j2, k3 );

      for ( int i = i1; i <= i2; i++ )
            for ( int j = j1; j <= j2; j++ )
                  for ( int k = k1; k <= k2; k++ )
                        for ( auto it : getNode ( i, j, k )->shapes )
                              if ( area.intersects ( (*it)->getBounds () ) )
                                    result.insert ( *it );

      return result;
}
```

Глава 6

Цвет и его представление. Работа с цветом

Одним из фундаментальных понятий, которые нам нужно рассмотреть, является понятие *цвета*. Это понятие очень тесно связано с другим важным понятием – *светом*.

Из курса физики мы знаем, что свет – это поток электромагнитных волн. Электромагнитные волны имеют целый набор различных характеристик, но нас будет интересовать в первую очередь длина волны λ.

Воспринимаемый человеческим глазом цвет *монохромного* (т. е. состоящего из волн только одной длины) света зависит именно от длины волны. При этом человеческий глаз способен воспринимать не все электромагнитные волны, а только те, для которых длина волны лежит в диапазоне от примерно 500 до 700 нанометров (1 нм = 10^{-9} м).

Однако только небольшое число источников света (например, лазеры) способно давать монохромный свет. Обычно же свет состоит из множества волн с самыми разными длинами. Для описания такого света чаще всего используется *спектральная плотность излучения* (SPD, Spectral Power Distribution).

Эта функция, далее обозначаемая как $I(\lambda)$, описывает плотность потока излучения, приходящуюся на заданную длину волны λ. Если у нас есть диапазон длин волн $[\lambda_1, \lambda_2]$, то мы можем получить мощность, приходящуюся на волны с длинами из этого диапазона, при помощи следующего интеграла:

$$I = \int_{\lambda_1}^{\lambda_2} I(\lambda) d\lambda.$$

С помощью спектральной плотности можно описать любой световой поток. Часто поток задается как таблица значений $I(\lambda)$, взятых с заданным шагом (например, 10 нм). На рис. 6.1 приведена спектральная плотность для стандарта дневного света – так называемого источника D_{65}.

Многие формулы для расчета освещенности объектов формулируются именно в терминах спектральной плотности. Однако для точного описания цвета спектральная плотность плохо подходит. Дело в том, что для произвольного ощущения цвета, воспринимаемого человеком, существует бесконечное количество различных спектральных плотностей (называемых *метамерами*), дающих именно это самое ощущение цвета (т. е. не различимых человеческим глазом).

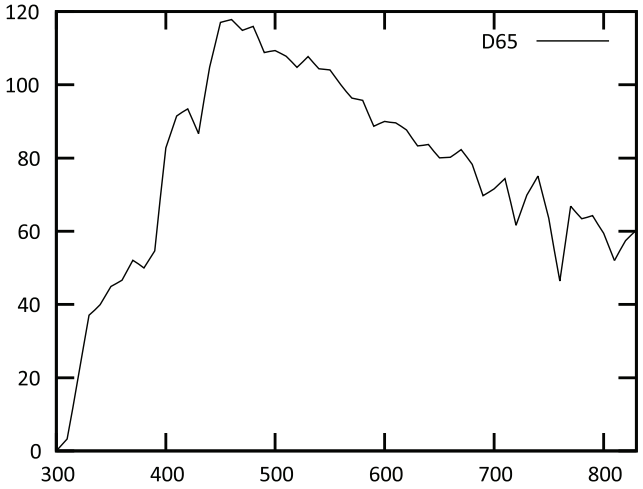

Рис. 6.1 ❖ Спектральная плотность источника D_{65}

Таким образом, цвет – это то, как человек воспринимает свет, т. е. это понятие напрямую связано со зрительной системой человека. Глаз человека можно рассматривать как фотокамеру, состоящую из объектива (хрусталик) и светочувствительного элемента – сетчатки (рис. 6.2).

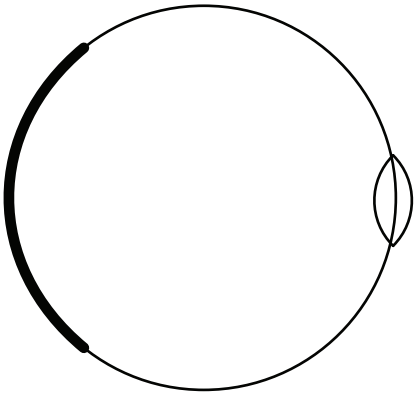

Рис. 6.2 ❖ Схематическое строение глаза человека

Сетчатка глаза состоит из двух типов светочувствительных клеток – палочек и колбочек. Палочки обладают очень высокой светочувствительностью, но при этом почти не различают длины волны. Расположены палочки в основном по краям сетчатки. Именно палочки отвечают за зрение в темноте, поэтому если вы хотите что- то разглядеть в условиях плохого освещения, то лучше смотреть периферическим зрением, чтобы изображение попало именно на палочки.

Другой тип светочувствительных клеток – колбочки – обладает гораздо меньшей светочувствительностью, и они делятся на три типа в зависимости от того, к каким длинам волн они чувствительны. Выделяют колбочки, чувствительные к коротким волнам (S, 420–440 нм), средним (M, 430–540 нм) и длинным (L, 560–

580 нм). Эти длины волн примерно соответствуют красному, зеленому и синему цветам. На рис. 6.3 приведены графики чувствительности для каждого из этих трех типов колбочек.

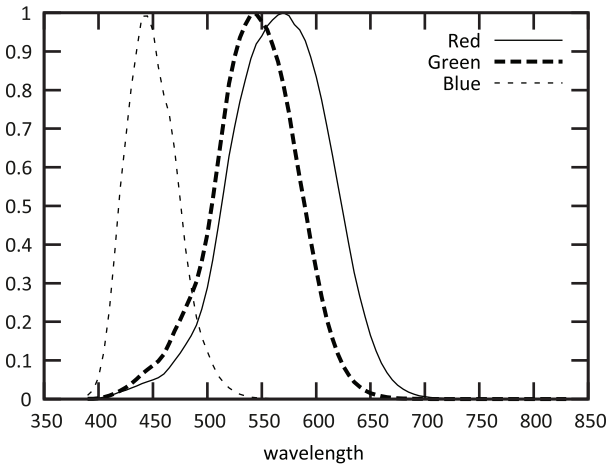

Рис. 6.3 ❖ Нормализованные светочувствительности для трех типов колбочек

Именно колбочки отвечают за восприятие человеком цвета. Это привело к следующей теории – любой цвет можно представить в виде взвешенной суммы трех базовых цветов (primaries) – красного (R, Red), зеленого (G, Green) и синего (B. Blue).

Однако в 1920-х годах независимо друг от друга исследователи Дэвид Райт и Джон Гилд провели серию экспериментов по разложению ряда цветов (в том числе и чистых спектральных цветов) на суммы красного, зеленого и синего цветов.

Для экспериментов были отобраны наблюдатели с нормальным зрением. Им показывались две цветные точки (соответствующие углу обзора 2°). При этом одна из этих точек была точкой цвета, который нужно было разложить, а другая получалась как сумма трех базовых цветов. Наблюдатель мог управлять пропорциями трех базовых цветов. Его задачей было добиться того, чтобы эти точки стали неразличимыми.

Однако в ходе экспериментов выяснилось, что многие цвета (в том числе и чистые спектральные цвета, отличные от трех базовых) нельзя разложить в сумму трех базовых цветов.

Но при этом выяснилось, что если к цвету, который не получается разложить, добавить определенный цвет из базовых, то получившийся цвет легко раскладывается по трем базовым. Фактически это означает, что любой заданный цвет может быть разложен, но вклад некоторых цветов может быть отрицательным.

На рис. 6.4 приводятся такие коэффициенты для разложения чистых спектральных цветов. При этом в качестве базовых цветов выбраны красный (700 нм), зеленый (546.1 нм) и синий (435.8 нм). Обратите внимание, что некоторые из получающихся коэффициентов действительно отрицательны.

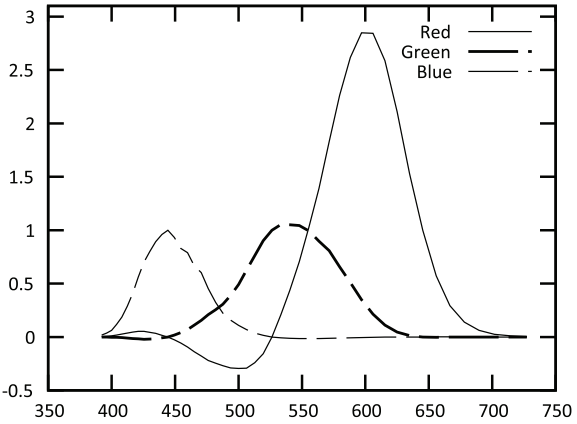

Рис. 6.4 ❖ Коэффициенты разложения чистых спектральных цветов

Цветовая модель CIE XYZ

Поскольку использование отрицательных весов для разложения цвета неудобно, то в 1931 году была разработана новая модель для представления цвета, основанная на работах Д. Гилда и Дж. Райта.

Созданная модель получила название *CIE XYZ*. Она позволяет описать каждый цвет при помощи трех неотрицательных чисел – *X*, *Y* и *Z*. При этом гарантируется, что если два световых потока воспринимаются человеком как имеющие один и тот же цвет, то они будут иметь одинаковые цветовые координаты *XYZ*.

Данная модель является независимой от какого-либо устройства (device-independent). В ней каждый из трех коэффициентов получается путем интегрирования спектральной плотности с соответствующей весовой функцией. Каждая из этих трех весовых функций ($\overline{x}(\lambda)$, $\overline{y}(\lambda)$ и $\overline{z}(\lambda)$) задана таблично с шагом 10 нм. Таким образом, если у нас есть спектральная плотность $I(\lambda)$, то координаты этого цвета определяются следующим образом:

$$X = \int I(\lambda)\overline{x}(\lambda)d\lambda;$$
$$Y = \int I(\lambda)\overline{y}(\lambda)d\lambda;$$
$$Z = \int I(\lambda)\overline{z}(\lambda)d\lambda.$$

На рис. 6.5 приведены графики весовых функций для модели *CIE XYZ*. Обратите внимание, что весовая функция $\overline{y}(\lambda)$ соответствует общей чувствительности глаза к свету.

Цветовая модель *CIE XYZ* является *аддитивной* – при сложении световых потоков соответствующие координаты цветов тоже складываются. В 1964 году была построена цветовая модель CIE XYZ 1964, основанная на угле обзора 10°.

В цветовой модели *CIE XYZ* каждая из трех координат несет в себе информацию и о яркости, и о самом цвете (*chroma*). В ряде случаев желательно разделить информацию о яркости и о цвете без учета яркости. Для этого вводят так называемые *хроматические координаты x* и *y* при помощи следующих уравнений:

$$x = \frac{X}{X+Y+Z};$$
$$y = \frac{Y}{X+Y+Z}.$$

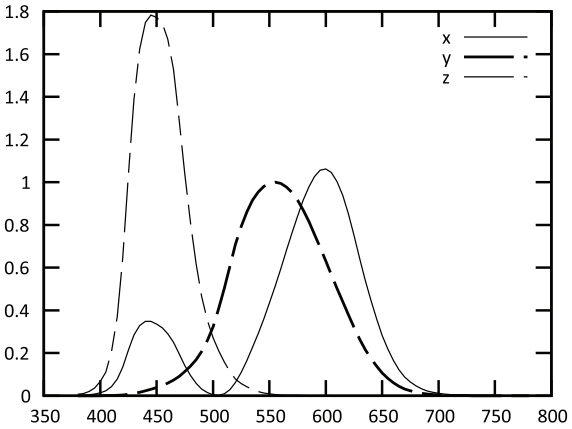

Рис. 6.5 ❖ Графики весовых функций для модели *CIE XYZ*

В результате мы получаем цветовую модель *CIE xyY*. Ниже приводятся формулы для обратных преобразований из *CIE xyY* в *CIE XYZ*:

$$X = \frac{Y}{y}x;$$
$$Z = \frac{Y}{y}(1-x-y).$$

Можно на плоскости *Oxy* построить множества хроматических координат всех различаемых глазом человека цветов. В результате мы получим так называемую *хроматическую диаграмму*, показанную на рис. 6.6.

Дуга на этой диаграмме соответствует чистым спектральным цветам, а отрезок – цветам, получающимся при смешении красного и синего (line of purples).

Хроматическая диаграмма обладает важным свойством: если на ней взять две точки *A* и *B*, то всевозможным взвешенным суммам цветов, соответствующих этим точкам, будет соответствовать отрезок *AB*.

Точно так же если мы возьмем три точки на диаграмме – *A*, *B* и *C*, то всем цветам, представимым в виде взвешенной суммы этих трех, будет соответствовать треугольник *ABC*. Из этого следует, что какое бы конечное число цветов мы ни взяли, они не могут дать все возможные цвета путем своего смешивания.

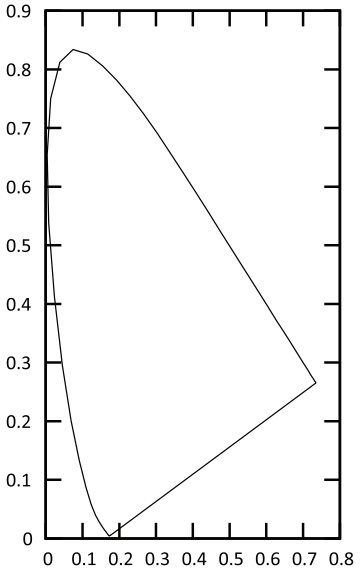

Рис. 6.6 ❖ Хроматическая диаграмма

Цветовая модель RGB

Кроме уже рассмотренных цветовых моделей (пространств) *CIE XYZ* и *CIE xyY*, существует на самом деле очень много других цветовых моделей, которые оказываются более удобными в тех или иных случаях. Большинство из этих цветовых моделей основано на *CIE XYZ*.

Одной из наиболее простых и распространенных цветовых моделей является модель *RGB*. В этой модели мы раскладываем каждый цвет по трем базовым цветам – красному (R), зеленому (G) и синему (B). Хотя далеко не все цвета могут быть представлены таким образом, именно эта модель используется практически во всех светоизлучающих устройствах для отображения изображений (дисплеи, мониторы, экраны телефонов и т. п.).

В подобных устройствах изображение представлено в виде прямоугольной матрицы светоизлучающих элементов (пикселов, от picture element). Каждый такой пиксел может излучать заданные пропорции базовых цветов, тем самым формируя заданный цвет. Именно таким образом работают различные LED- и OLED-дисплеи. Управляя яркостью каждого из базовых цветов, мы можем получать необходимые цвета (среди представимых цветов). При этом человек, смотря фильм на подобном дисплее, обычно даже и не замечает, что это устройство не в состоянии показать все возможные цвета.

Также подобная модель используется и в различных цифровых камерах, которые тоже дают по три цветовых веса для каждого пиксела.

В модели *RGB* каждый цвет представляется в виде тройки (r, g, b), где каждый элемент задает вклад соответствующего базового цвета. Множество всех возможных цветов, представимых этой моделью, обычно представляют в виде единичного куба (рис. 6.7).

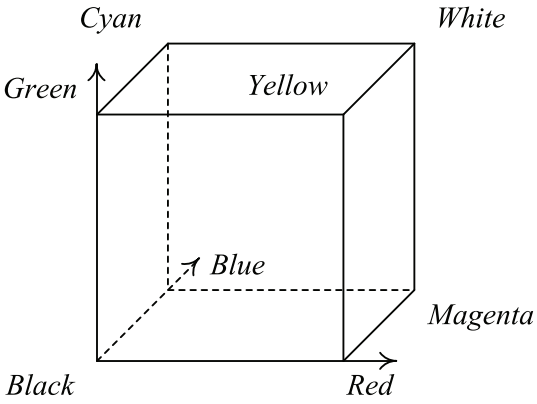

Рис. 6.7 ❖ Цветовое пространство RGB

Эта модель, как и *CIE XYZ*, является аддитивной. Обратите внимание, что в общем случае эта модель зависит от конкретного устройства – одна и та же картинка, показанная, например, на экранах мобильных телефонов от разных производителей, обычно выглядит по-разному. Это связано с тем, что разные производители могут использовать свой набор базовых цветов (например, цвет красного у одного производителя может чуть отличаться от цвета красного у другого производителя).

Однако для телевидения высокой четкости (HDTV) были явно заданы базовые цвета и тем самым установлены законы перехода между используемой моделью *RGB* и *CIE XYZ* (*Rec 709*):

$$\begin{pmatrix} R_{709} \\ G_{709} \\ B_{709} \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.537150 & -0498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix};$$

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R_{709} \\ G_{709} \\ B_{709} \end{pmatrix}.$$

Цветовые модели CMY CMYK

Если вы сталкивались с цветной печатью, то помните, что там используются цветные чернила отнюдь не красного, зеленого и синего цветов. А иногда к трем базовым цветам еще добавляется черный.

Дело в том, что модель *RGB* рассчитана на светоизлучающие устройства – фактически с каждым пикселом связаны три источника света – красного, зеленого и синего цветов.

Но при цветной печати имеет место не излучение света, а его поглощение. На лист белой бумаги наносятся чернила различных цветов. Белый свет падает на такой лист и частично отражается, частично поглощается нанесенными чернилами (рис. 6.8).

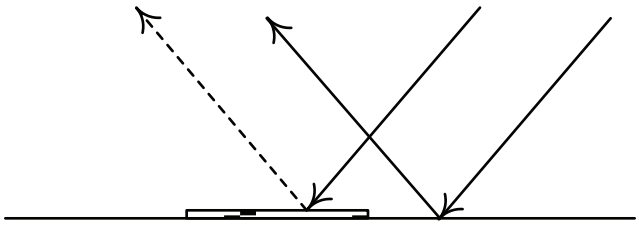

Рис. 6.8 ❖ Отражение и поглощение падающего на бумагу света

Тем самым если мы видим в результате белый цвет, то это значит, что весь падающий свет равномерно отразился от бумаги. Если мы видим какой-то другой цвет, то это значит, что волны с определенными длинами были поглощены нанесенными на бумагу чернилами. Тем самым мы видим только то, что было отражено. Фактически происходит вычитание некоторого цвета из белого цвета.

Если мы нанесем на бумагу сразу два красителя, то каждый из них поглотит свою часть белого цвета, и мы увидим некоторый новый цвет.

Поэтому цветовая модель, используемая для цветной печати, фактически является *субтрактивной*, т. е. основанной не на сложении цветов, а на их вычитании. Соответственно, базовые цвета в этой модели будут получаться путем вычитания базовых цветов модели *RGB* из белого цвета.

Вычитая красный цвет из белого, мы получаем голубой (C, Cyan), вычитая зеленый, мы получаем малиновый (M, Magenta), и, вычитая синий, мы получаем желтый (Y, Yellow). Соответственно, цветовая модель, основанная на этих трех цветах, называется *CMY*. Правила перевода между моделями *CMY* и *RGB* очень просты и основаны на следующем равенстве:

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} + \begin{pmatrix} c \\ m \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Однако в чистой модели *CMY* оттенки серого (т. е. цвета вида (x, x, x)) представляются при помощи равного количества каждого из трех базовых чернил. С одной стороны, это очень дорого (цветные чернила действительно очень дорого стоят), с другой – так как тяжело обеспечить точное наложение точек каждого из трех базовых цветов друг на друга, то в результате мы можем получить черную точку с радужной каймой вокруг нее.

Поэтому именно для таких случаев к трем базовым цветам добавляется четвертый цвет – черный (K, black). В результате получается цветовая модель CMYK. Ниже приводятся формулы для перевода цвета из модели *CMY* (c, m, y) в модель *CMYK* (c', m', y', k):

$k = \min(c, m, y)$;
$c' = c - k$;
$m' = m - k$;
$y' = y - k$.

Обратите внимание, что на самом деле формулы преобразования цветов, приведенные выше, являются приближенными. Это связано с тем, что они никак не учитывают взаимодействие чернил друг с другом (т. е. кривые поглощения для базовых чернил на самом деле перекрываются).

Точный перевод цвета между *RGB* и *CMY/CMYK* на самом деле очень сложен, необходимо учитывать не только свойства чернил, но даже и свойства и тип используемой для печати бумаги. Различные приложения, например Adobe Photoshop, содержат большое количество готовых цветовых профилей для различных принтеров и позволяют точно выполнять перевод цвета, так чтобы на разных цветных принтерах цвета выглядели одинаково.

Цветовая модель HSV

Хотя рассмотренные цветовые модели довольно удобны, но они очень мало соответствуют тому, как художники воспринимают цвет. Именно поэтому и была в 1978 году предложена цветовая модель *HSV* (Hue, Saturation, Value). Эта модель основана на следующих интуитивных свойствах цвета: *тоне* (hue), *насыщенности* (saturation) и *яркости* (value).

Тон цвета отражает «чистый» оттенок цвета, отделенный от яркости, и измеряется углом от 0° до 360°. Насыщенность – это насколько цвет близок к чистому цвету той же яркости, измеряется значениями от 0 до 1. То, что в модели *HSV* обозначено как яркость, на самом деле не является яркостью цвета, но довольно близко к ней. Эта величина также измеряется числом от 0 до 1. Все множество представимых цветов в этой модели является перевернутой шестигранной пирамидой, приведенной на рис. 6.9.

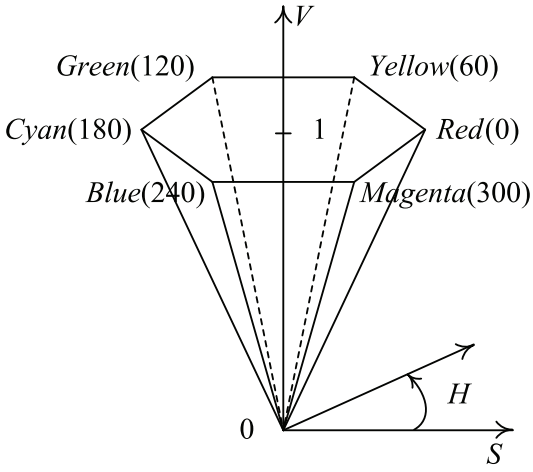

Рис. 6.9 ❖ Цветовое пространство *HSV*

Обратите внимание, что в основании пирамиды ($V = 0$) тон и насыщенность не определены. Для перевода из *RGB* в *HSV* используются следующие формулы:

$$V = M;$$

$$S = \frac{d}{M};$$

$$H = 60 \cdot \begin{cases} \frac{g-b}{d}, M = r \\ \frac{b-r}{d}, M = g. \\ \frac{r-g}{d}, M = b \end{cases}$$

Здесь используются следующие величины:

$M = \max(r, g, b)$;
$m = \min(r, g, b)$;
$d = M - m$.

Ниже приводятся функции на C++, предназначенные для перевода цвета между моделями *RGB* и *HSV* (для хранения цвета используется трехмерные векторы).

```
glm::vec3 rgbToHsv ( const glm::vec3& rgb )
{
      int iMax = 0;                 // Индекс максимального значения
      int iMin = 0;                 // Индекс минимального значения

      for ( int i = 0; i < 3; i++ )
            if ( rgb [i] > rgb [iMax] )
                  iMax = i;
            else
            if ( rgb [i] < rgb [iMin] )
                  iMin = i;

      float mx = rgb [iMax];
      float mn = rgb [iMin];
      float d  = mx - mn;

      if ( mx < EPS )         // Тон не определен
            return glm::vec3 ( 0 );

      float v = mx;
      float s = d / mx;
      float h;

      if ( iMax == 0 )
            h = 60 * (rgb.g - rgb.b) / d;
      else
      if ( iMax == 1 )
            h = 60 * (rgb.b - rgb.r) / d + 120;
      else
            h = 60 * (rgb.r - rgb.g) + 240;

      return glm::vec3 ( h, s, v );
```

```
}

glm::vec3 hsvToRgb ( const glm::vec3& hsv )
{
      float v = hsv.z;

      if ( hsv.y < 0.001 )
            return glm::vec3 ( v );

      float h  = hsv.x * 6.0;
      float fi = floor ( h );
      int   i  = int ( fi );
      float f  = h - fi;
      float p  = hsv.z * (1.0 - hsv.y);
      float q  = hsv.z * (1.0 - hsv.y * f);
      float t  = hsv.z * (1.0 - hsv.y * (1.0 - f));

      if ( i == 0 )
            return glm::vec3 ( v, t, p );
      else
      if ( i == 1 )
            return glm::vec3 ( q, v, p );
      else
      if ( i == 2 )
            return glm::vec3 ( p, v, t );
      else
      if ( i == 3 )
            return glm::vec3 ( p, q, v );
      else
      if ( i == 4 )
            return glm::vec3 ( t, p, v );
      else
            return glm::vec3 ( v, p, q );
}
```

Поскольку цветовая модель *HSV* является более интуитивной, она часто используется для задания цвета в различных программах. На рис. 6.10 приводится диалог выбора цвета. Обратите внимание, что цветовой круг полностью соответствует шестиугольнику с цветами в модели *HSV*.

Кроме того, эту модель очень удобно использовать для нахождения точек заданного цвета и выполнения различных преобразований над цветом.

В качестве примера рассмотрим преобразование цветов из фильма «Город грехов». В терминах модели HSV у всех цветов, кроме красного (тон близок либо к 0, либо 360), сильно уменьшается насыщенность. А у красного насыщенность, наоборот, увеличивается. Ниже приводится код, реализующий данный эффект.

```
glm::vec3 sinCityEffect ( const glm::vec3& color )
{
      glm::vec3   hsv = rgbToHsv ( color );

      if ( hsv.x < DELTA || hsv.x > 360-DELTA )   // Это красный цвет ?
            hsv.y *= 1.5f;
```

```
        else
                hsv.y /= 1.5f;

        return hsvToRgb ( hsv );
}
```

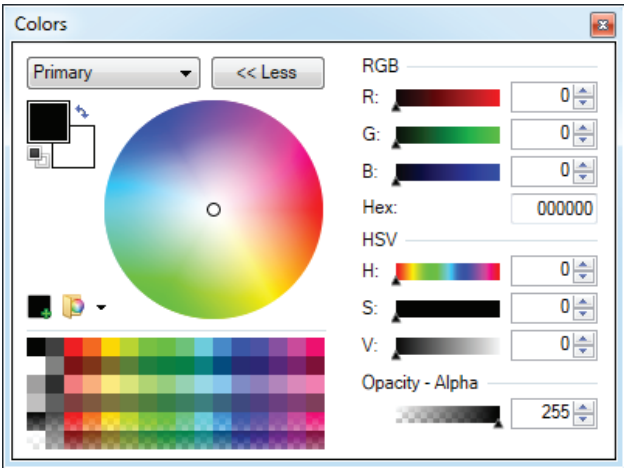

Рис. 6.10 ❖ Диалог выбора цвета из программы Paint.net

Цветовое пространство HSL

В цветовой модели *HSV* есть лишь один цвет, для которого не определены насыщенность и тон. Однако вполне логично отнести также и белый цвет к цветам, не обладающим определенными значениями насыщенности и тона.

Есть цветовая модель *HSL* (Hue, Saturation, Lightness), использующая именно этот подход. Ее цветовое пространство показано на рис. 6.11. Как видно, белому и черному цветам (вершины) действительно не соответствуют определенные значения насыщенности и тона.

Для преобразования цвета из модели *RGB* в модель *HSL* используются следующие формулы:

$$L = \frac{M+m}{2};$$

$$S = \begin{cases} 0, d = 0 \\ \dfrac{d}{1-|2L-1|}, d > 0 \end{cases};$$

$$H = 60 \cdot \begin{cases} \dfrac{g-b}{d}, M = r \\ \dfrac{b-r}{d}, M = g \\ \dfrac{r-g}{d}, M = b \end{cases}.$$

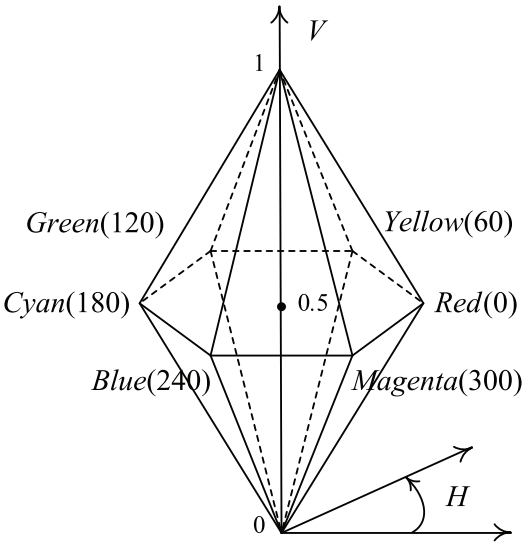

Рис. 6.11 ❖ Цветовое пространство *HSL*

Здесь величины *M*, *m* и *d* совпадают с аналогичными величинами из модели *HSV*. Ниже приводятся функции на C++, переводящие цвет из *RGB* в *HSL* и обратно.

```
glm::vec3 rgbToHsl ( const glm::vec3& rgb )
{
  int iMax = 0;           // Индекс максимального значения
  int iMin = 0;           // Индекс минимального значения

  for ( int i = 0; i < 3; i++ )
    if ( rgb [i] > rgb [iMax] )
      iMax = i;
    else
    if ( rgb [i] < rgb [iMin] )
      iMin = i;

  float mx = rgb [iMax];
  float mn = rgb [iMin];
  float d  = mx - mn;

  if ( mx < 0.001 )       // Тон не определен
    return vec3 ( 0 );

  float l = (mx + mn) * 0.5;
  float s = d / (1 - fabs(2*l - 1));
  float h;

  if ( iMax == 0 )
    h = 60 * (g - b) / d;
  else
  if ( iMax == 1 )
    h = 60 * (b - r) / d + 120;
```

```
  else
    h = 60 * (r - g) + 240;

  return glm::vec3 ( h, s, l );
}

glm::vec3 hslToRgb ( const glm::vec3& hsl )
{
  if ( hsl.y < EPS )      // S равно нулю
    return glm::vec3 ( hsl.z );

  float v2  = (hsl.z <= 0.5) ? hsl.z * (1.0 + hsl.y) :
              (hsl.z + hsl.y - hsl.y * hsl.z);
  float v1 = 2 * hsl.z - v1;

  float      h = hsv.x / 360.0;
  glm::vec3  t ( h + 1.0/3.0, h, h - 1.0/3.0 );

  if ( 6 * t.x < 1 )
    t.x = v1 + (v2-v1)*6.0*t.x;
  else
  if ( 2 * t.x < 1 )
    t.x = v2;
  else
  if ( 3 * t.x < 2 )
    t.x = v1 + (v2-v1)*((2.0/3.0)-t.x)*6.0;
  else
    t.x = v1;

  if ( 6 * t.y < 1 )
    t.y = v1 + (v2-v1)*6.0*t.y;
  else
  if ( 2 * t.x < 1 )
    t.y = v2;
  else
  if ( 3 * t.y < 2 )
    t.y = v1 + (v2-v1)*((2.0/3.0)-t.y)*6.0;
  else
    t.y = v1;

  if ( 6 * t.z < 1 )
    t.z = v1 + (v2-v1)*6.0*t.z;
  else
  if ( 2 * t.z < 1 )
    t.z = v2;
  else
  if ( 3 * t.z < 2 )
    t.z = v1 + (v2-v1)*((2.0/3.0)-t.z)*6.0;
  else
    t.z = v1;

  return t;
}
```

Обратите внимание, что параметр L в этой модели также лишь примерно соответствует яркости цвета.

Гамма-коррекция

Введенная ранее модель *CIE XYZ*, как и модель *RGB*, фактически определяет световую энергию (*luminance*). Однако человеческий глаз реагирует на световую энергию нелинейным образом – удвоение энергии не будет соответствовать вдвое более яркому воспринимаемому глазом цвету.

CIE определяет воспринимаемую человеком яркость как *светлоту* (lightness) и использует для нее следующую формулу:

$$L^* = \begin{cases} 116\left(Y/Y_n\right)^{1/3} - 16, & Y/Y_n > 0.008856 \\ 903.3\,Y/Y_n, & Y/Y_n \le 0.008856 \end{cases}.$$

В этой формуле через Y_n обозначена эталонная яркость (обычно соответствующая белому цвету). Сама функция имеет вид кубического корня с линейным отрезком около нуля и принимает значения от 0 до 100. Ее график приведен на рис. 6.12.

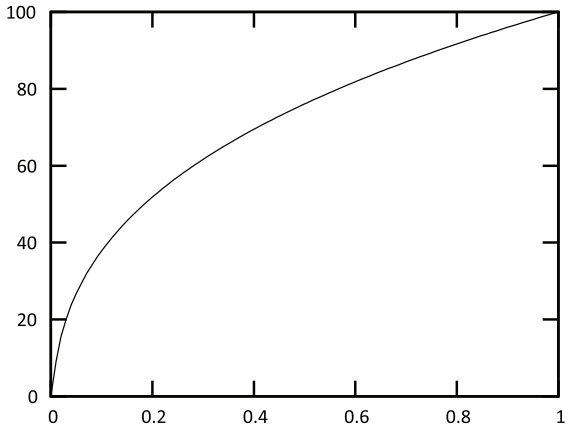

Рис. 6.12 ❖ График зависимости L^* от Y

Можно считать, что человеческое зрение обладает нелинейной функцией переноса вида степенной функции.

Оказывается, что различные как цифровые, так и аналоговые устройства (камеры, мониторы и т. п.) также обладают нелинейной функцией переноса степенного вида

$f(x) = x^{\gamma}$.

Само это преобразование получило название γ-коррекции.

Стандарт для телевидения высокой четкости *Rec 709* определяет эту функцию следующим образом:

$$E'_{709} = \begin{cases} 4.5E, E \le 0.018 \\ 1.099E^{0.45} - 0.99, E > 0.018 \end{cases}.$$

На рис. 6.13 приведен график этого преобразования.

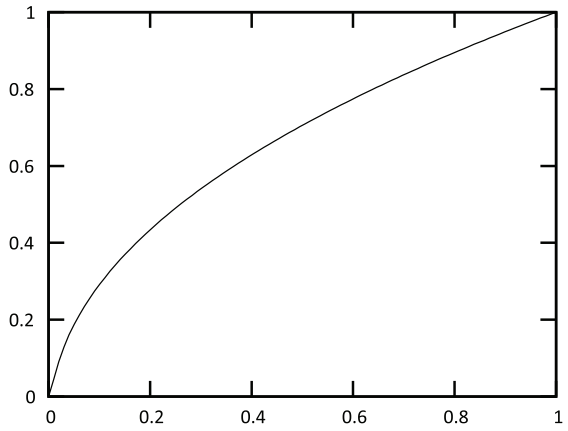

Рис. 6.13 ❖ График E'_{709}

Результат применения этой функции к компонентам *RGB* (называемым линейными) обозначается как *E′G′B′*. Стандартный монитор обращает это преобразование, переводя нелинейный цвет в линейный по следующим формулам:

$$E = \begin{cases} \dfrac{E'_{709}}{4.5}, E_{709} \le 0.081 \\ \left(\dfrac{E_{709} + 0.099}{1.099}\right)^{\frac{1}{0.45}}, E_{709} > 0.081 \end{cases}.$$

Цветовые пространства Y′uv и YCbCr

Есть еще целый ряд различных цветовых моделей, явно разделяющих яркость цвета от его хроматических координат. Одной из таких моделей является *Yuv*, используемая в цветном телевидении и для хранения видео.

При появлении цветного телевидения возникла проблема с тем, чтобы сделать его совместимым с уже существующим черно-белым. Хотелось, чтобы старые (черно-белые) телевизоры могли показывать цветные видеосигналы (естественно, без цвета).

С этой целью было использовано цветовое пространство *Y′uv*, в котором яркость *Y′* передавалась по тому же каналу, что и для черно-белого телевидения. Хроматические компоненты *u* и *v* передавались отдельно.

Полученную в результате модель *Y′uv* использовали в видеостандарте PAL (*Phase-Alternation Law*), NTSC (*National Television System Committee*) и SECAM (*Séquentiel Couleur Avec Mémoire*). Также эта модель применялась для хранения изображений и видео.

Ниже приводятся формулы для перевода цвета из *RGB* в *Y′uv*, обратите внимание, что *Y* – это яркость цвета:

$Y' = 0.299R' + 0.587G' + 0.114B'$;
$u = 0.492(B' - Y')$;
$v = 0.877(R' - Y')$.

Несложно из этих формул получить обратное преобразование:

$R' = Y' + 1.140v$;
$G' = Y' - 0.395u - 0.581v$;
$B' = Y' + 2.032u$.

Кроме этой модели, также есть очень похожая модель *YCbCr*, которая является просто перемасштабированной версией *Y′uv*. Для нее есть два варианта, соответствующих стандартам *Rec 603* и *Rec 709*. Ниже приводятся формулы для *Rec 709*:

$Y'_{709} = 0.213R' + 0.715G' + 0.07B'$;
$C_b = -0.117R' - 0.394G' + 0.511B' + 128$;
$C_r = 0.511R' - 0.464G' - 0.047B' + 128$.

Цветовые пространства L*u*v* и L*a*b*

Все рассмотренные выше цветовые модели не обладают очень важным свойством – одно и то же изменение в координатах цвета (например, (ΔX, ΔY, ΔZ)) воспринимается неодинаково в зависимости от того, к какому именно цвету оно применяется.

В то же время в ряде случаев желательно получить однородность, т. е. когда одно и то же изменение воспринимается совершенно одинаково вне зависимости от того, к какому именно цвету оно применяется.

Для этих целей обычно используются цветовые модели $L^*u^*v^*$ и $L^*a^*b^*$. Ниже приводятся формулы для преобразования цвета из *CIE XYZ* в эти модели. Обратите внимание, что нам нужен некоторый базовый (обычно белый) цвет (X_n, Y_n, Z_n).

Для обеих моделей сначала рассчитывается L^* по формуле, после чего вычисляются коэффициенты u' и v' по следующим формулам:

$$u' = \frac{4X}{X + 15Y + 3Z};$$
$$v' = \frac{9Y}{X + 15Y + 3Z}.$$

После этого по ним вычисляются u^* и v^*:

$u^* = 13L^*(u' - u'_n)$;
$v^* = 13L^*(v' - v'_n)$.

Цветовая модель $L^*a^*b^*$ задается следующими формулами:

$$a^* = 500\left[\left(\frac{X}{X_n}\right)^{\frac{1}{3}} - \left(\frac{Y}{Y_n}\right)^{\frac{1}{3}}\right];$$

$$b^* = 200\left[\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - \left(\frac{Z}{ZY_n}\right)^{\frac{1}{3}}\right].$$

Цветовое пространство sRGB

В реальных цифровых устройствах, таких как дисплеи и цифровые камеры, широкое распространение получило цветовое пространство *sRGB*. Данное пространство было совместно создано компаниями Microsoft и HP. Сейчас его поддерживают W3C, Exif, Intel, Pantone, Corel и многие другие. Данное пространство аппаратно поддерживается многими современными GPU, позволяя программистам опираться на эту аппаратную поддержку.

Формат *sRGB* использует те же самые базовые цвета – красный, зеленый и синий, что и HDTV. Первым шагом в переводе цвета из *CIE XYZ* в *sRGB* является получение линейных компонент цвета (R_l, G_l, B_l) по следующей формуле:

$$\begin{pmatrix} R_l \\ G_l \\ B_l \end{pmatrix} = \begin{pmatrix} 3.2406 & -1.5373 & -0.4986 \\ -0.9689 & 1.8758 & 0.0415 \\ 0.0557 & -0.2040 & 1.0570 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.$$

Далее каждая из этих компонент подвергается нелинейному преобразованию, задаваемому следующей формулой:

$$c_s(x) = \begin{cases} 12.92x, x \le 0.0031308 \\ 1.055x^{1/2.4}, x \ge 0.0031308 \end{cases}.$$

Для обратного преобразования используется следующая обратная функция:

$$c_s^{-1}(x) = \begin{cases} \dfrac{x}{12.92}, x \le 0.04095 \\ \left(\dfrac{x+0.055}{1.055}\right)^{2.4}, x > 0.04045 \end{cases}.$$

Соглашения по дальнейшему использованию цветов

Всюду далее мы будем рассматривать цвет как трехмерный или четырехмерный вектор для обозначения цвета (об использовании четырехмерных векторов для обозначения цвета будет подробнее рассказано в главе, посвященной OpenGL). Соответственно, будет использоваться модель *RGB*.

Там, где речь идет о расчете цветов, будут использоваться векторы из вещественных чисел (например, `glm::vec3`).

Однако когда речь заходит о сохранении цветов в файл, обычно для каждой цветовой компоненты используются беззнаковые 8-битовые целые числа (`uint8_t`). Тем самым для хранения цвета необходимо 24 или 32 бита, что полностью помещается в тип `uint32_t`. Обычно в файлах, содержащих изображения, используется 24 или 32 бита на один пиксел.

Глава 7

Растеризация и растровые алгоритмы

Встречающиеся изображения можно разделить на *векторные* и *растровые*. Между этими двумя способами представления изображений есть довольно значительные различия.

При векторном представлении изображение задается как набор базовых примитивов, таких как отрезки, дуги и т. п. Параметры, задающие данные примитивы, могут быть заданы с высокой точностью. Примером векторных изображений являются изображения в формате svg (Scalable Vector Graphics).

Изображения в таком формате обычно занимают мало места и легко подвергаются различным преобразованиям. Ниже приводится пример фрагмента svg-изображения, где было оставлено лишь описание геометрии:

```
<g id="shape1-1" v:mID="1" v:groupContext="shape"
   transform="translate(144,-468)">
      <title>Sheet.1</title>
            <rect x="0" y="756" width="72" height="36" class="st1"/>
</g>
<g id="shape2-3" v:mID="2" v:groupContext="shape"
   transform="translate(144,-531)">
      <title>Sheet.2</title>
      <ellipse cx="33.75" cy="780.75" rx="33.75" ry="11.25" class="st1"/>
</g>
<g id="shape3-5" v:mID="3" v:groupContext="shape"
   transform="translate(-464.299,-57.4884) rotate(-58.5704)">
      <title>Sheet.3</title>
      <path d="M0 792 L94.93 792" class="st2"/>
</g>
<g id="shape4-8" v:mID="4" v:groupContext="shape"
   transform="translate(-214.2,908.1) rotate(-143.13)">
      <title>Sheet.4</title>
      <path d="M0 792 L90 792" class="st2"/>
</g>
```

Соответствующее этому изображение приведено на рис. 7.1.

Собственно первые графические дисплеи были именно векторными – изображение строилось из отрезков и дуг, хранение параметров требовало мало очень

дорогой в то время памяти. Луч электронов просто обегал все заданные примитивы на экране раз за разом, формируя тем самым изображение.

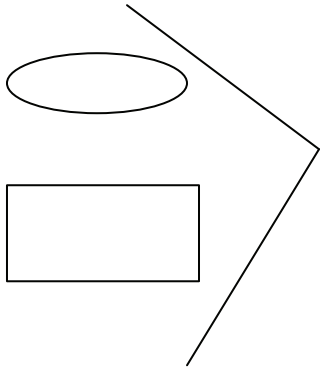

Рис. 7.1 ❖ Векторное изображение, соответствующее приведенному выше svg-файлу

Однако подобное представление несет в себе и определенные проблемы (например, резко падает частота кадров для сложных изображений), именно поэтому практически все современные графические устройства являются растровыми.

При *растровом* представлении изображение задается в виде прямоугольной матрицы *пикселов* (pixel, picture element). Каждый пиксел не зависит от остальных и обладает своим цветом. Все пикселы расположены в узлах *растровой сетки* (решетки) с целочисленными координатами (рис. 7.2).

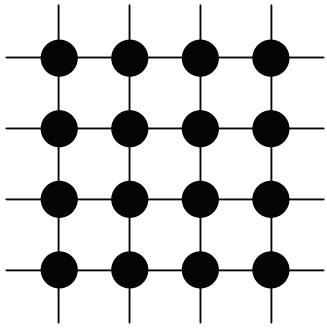

Рис. 7.2 ❖ Растровая сетка

Этот способ удобен для хранения и отображения изображений, однако при их построении мы обычно имеем дело с векторным представлением. Фактически у нас есть набор геометрических примитивов, и нам нужно их *растеризовать*, т. е. перевести во множество пикселов на растровой сетке.

Для растеризации примитивов существует большое количество различных алгоритмов, некоторые из которых рассматриваются в этой главе. Как правило, каждый современный GPU имеет аппаратную поддержку растеризации основных типов примитивов – отрезков и треугольников.

Класс TgaImage и его использование

В этой главе мы будем приводить примеры функций, реализующих те или иные растровые алгоритмы. Для демонстрации их работы мы будем использовать специальный класс TgaImage, описание которого приводится ниже.

```
class TgaImage
{
protected:
      uint8_t * data;
      int       width;
      int       height;

public:
      TgaImage  ( int w, int h );
      ~TgaImage ();

      int getWidth () const
      {
            return width;
      }

      int getHeight () const
      {
            return height;
      }

      uint8_t * getData ()
      {
            return data;
      }

      bool writeToFile ( const char * filename );
      void clear       ();

      uint32_t getPixel ( int x, int y ) const
      {
            int   pos = ((height-1-y) * width + x) * 3;

            return (uint32_t) data [pos+2] + (((uint32_t)data [pos+1]) << 8)
                  + (((uint32_t)data [pos]) << 16);
      }

      void     putPixel ( int x, int y, uint32_t color )
      {
            int   pos = ((height-1-y) * width + x) * 3;

            data [pos+2] = (uint8_t)(color & 0xFF);
            data [pos+1] = (uint8_t)((color >> 8) & 0xFF);
            data [pos]   = (uint8_t)((color >> 16) & 0xFF);
      }

      uint32_t    rgbToInt ( uint32_t red, uint32_t green,
                             uint32_t blue ) const
      {
            return red | (green << 8) | (blue << 16);
      }

      void setRgbData ( const uint8_t * ptr );
};
```

Данный класс представляет собой «виртуальную» растровую сетку заданного размера. В любой пиксел этой сетки мы можем записать любое значение цвета. Полученную сетку всегда можно сохранить в файл в виде изображения в формате tga.

Ниже приводится простой пример использования данного класса. Мы создаем растровую сетку (изображение заданного размера), очищаем ее, записываем несколько пикселов и сохраняем в файл.

```
TgaImage image ( 100,100 );

uint32_t c1 = image.rgbToInt ( 0xFF, 0, 0 );
uint32_t c2 = image.rgbToInt ( 0, 0xFF, 0 );
uint32_t c3 = image.rgbToInt ( 0, 0, 0xFF );

image.putPixel ( 10, 10, c1 );
image.putPixel ( 20, 10, c3 );

for ( int i = 15; i < 50; i++ )
      image.putPixel ( i, i, c2 );

printf ( "put: %06x get: %06x\n", c1, image.getPixel ( 10, 10 ) );

image.writeToFile ( «test.tga» );
```

Понятие связности растровой сетки. 4- и 8-связность

Стандартные растеризуемые объекты обладают таким важным геометрическим свойством, как *связность*. Связность означает, что для любых двух точек объекта всегда найдется непрерывная линия, соединяющая эти две точки и полностью лежащая в пределах нашего объекта.

Желательно, чтобы получившееся при растеризации растровое множество пикселов также обладало этим свойством. Для этого мы должны сначала определить, что значит связность на растровой сетке.

Для растровой сетки существует два типа связности – *4-связность* и *8-связность*. Пикселы (x_0, y_0) и (x_1, y_1) называются 4-связными, если $|x_1 - x_0| + |y_1 - y_0| \leq 1$. Таким образом, два 4-связных пиксела могут отличаться либо в x-координате, либо в y-координате. На рис. 7.3. слева приведен пример двух 4-связных пикселов.

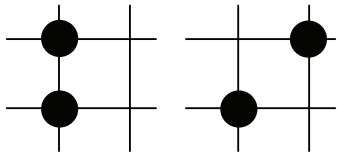

Рис. 7.3 ❖ Связность на растровой сетке

Аналогично два пиксела (x_0, y_0) и (x_1, y_1) называются 8-связными, если $|x_1 - x_0| \leq 1$ и $|y_1 - y_0| \leq 1$ (рис. 7.3 справа).

Обратите внимание, что любые два 4-связных пиксела всегда являются 8-связными. Обратное в общем случае не верно – пикселы (1, 1) и (0, 0) являются 8-связными, но не 4-связными.

Растровое представление объекта (т. е. множество пикселов на растровой сетке) называется 4-связным, если для любых двух принадлежащих ему пикселов (x_0, y_0) и (x_1, y_1) найдется такая последовательность пикселов, начинающаяся с пиксела (x_0, y_0) и заканчивающаяся пикселом (x_1, y_1), что каждые два последовательных пиксела будут являться 4-связными. Аналогично можно ввести понятие 8-связного множества пикселов.

Таким образом, можно говорить об алгоритмах растеризации, дающих 4-связные и 8-связные растровые представления объектов. Так, на рис. 7.4 слева приводится 4-связная растровая развертка отрезка, а справа – 8-связная развертка этого же отрезка.

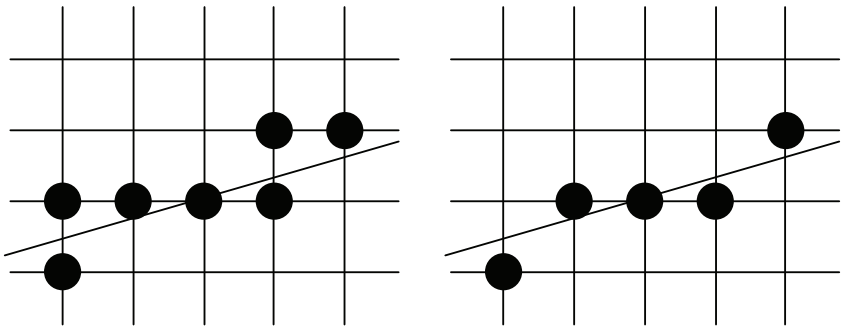

Рис. 7.4 ❖ 4- и 8-связные развертки отрезка

Построение растрового представления отрезка. Алгоритм Брезенхейма

Самым известным алгоритмом растеризации отрезка является *алгоритм Брезенхейма*. Этот алгоритм был изначально предложен в 1962 году для управления графопостроителями (плоттерами). Алгоритм оказался крайне прост и эффективен, он работает только с целыми числами и не использует операции умножения и деления. Было показано, что он дает растровое представление отрезка, минимизирующее расстояние от пикселов до исходного отрезка (суммарную ошибку).

Ниже мы покажем, как можно прийти к этому алгоритму за несколько простых шагов. Мы будем рассматривать построение 8-связной развертки отрезка, но в конце раздела будет приведен код для построения 4-связной развертки.

Мы будем строить растровую развертку отрезка AB, где на координаты точек $A(x_a, a_a)$ и $B(x_b, y_b)$ наложено следующее простое ограничение: $0 \le y_b - y_a \le x_b - x_a$. На рис. 7.5 приведен пример желаемой растровой развертки отрезка.

После того как мы придем к эффективной реализации алгоритма, мы рассмотрим, каким образом можно снять наложенные нами ограничения на точки A и B.

Самый простой способ растровой развертки отрезка заключается в использовании уравнения прямой, проходящей через точки A и B. Это уравнение имеет следующий вид:

$$y = kx + b;$$
$$k = \frac{y_b - y_a}{x_b - x_a};$$
$$b = y_b - kx_a.$$

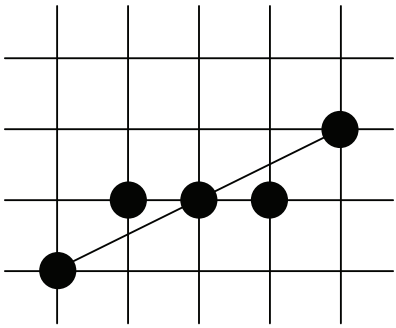

Рис. 7.5 ❖ Требуемая растровая развертка отрезка

Далее мы можем просто перебрать все пикселы от x_a до x_b (с шагом 1) и для каждого из них найти соответствующее ему значение y из уравнения прямой. Дальше это значение округляется до ближайшего целого числа. В результате мы получаем следующую функцию:

```
void drawLine1 ( TgaImage& image, int xa, int ya, int xb, int yb, uint32_t color )
{
      const float k = (float)(yb-ya)/(float)(xb - xa);
      const float b = ya - k * xa;

      for ( int x = xa; x <= xb; x++ )
            image.putPixel ( x, round ( k*x + b ), color );
}
```

Очевидно, что данный код крайне неэффективен – мы используем числа с плавающей точкой (`float`), для каждой точки выполняется умножение. Можно оптимизировать данный код, перейдя от явного вычисления значения y для каждого пиксела к использованию рекуррентных соотношений для вычисления следующего пиксела по предыдущему:

$$x_i = x_{i-1} + 1;$$
$$y_i = y_{i-1} + k.$$

Этому подходу соответствует следующая функция:

```
void drawLine2 ( TgaImage& image, int xa, int ya, int xb, int yb, uint32_t color )
{
      const float k = (float)(yb-ya)/(float)(xb - xa);
      const float b = ya - k*xa;
      float       y = ya;

      for ( int x = xa; x <= xb; x++, y += k )
            image.putPixel ( x, round ( y ), color );
}
```

В качестве следующего шага оптимизации мы можем избавиться от вызова функции round на каждом шаге. Для этого мы будем отдельно отслеживать целую (округленную) часть y_i и дробную часть c_i:

$x_i = x_{i-1} + 1;$
$c_i = \{kx_i + b\}.$

При этом мы будем использовать следующее правило для округления: если $c_i \leq 1/2$, то мы округляем $kx_i + b$ вниз, т. е. $y_i = [kx_i + b]$. В противном случае мы округляем вверх, т. е. $y_i = [kx_i + b] + 1$ (рис. 7.6).

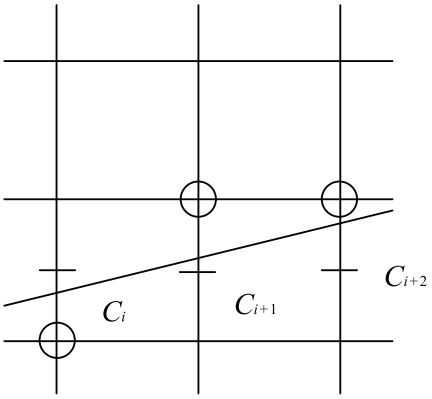

Рис. 7.6 ❖ Округление при построении отрезка

Поскольку мы хотим использовать рекуррентные формулы для y_i и c_i, то нам необходимо их явно выписать.

Здесь следует явно рассмотреть оба случая – округление вверх и округление вниз. При округлении вниз справедливы следующие формулы:

$x_i = x_{i-1} + 1;$
$c_i = c_{i-1} + k;$
$y_i = y_{i-1}.$

В случае округления вверх необходимо использовать формулы, приведенные ниже:

$x_i = x_{i-1} + 1;$
$c_i = c_{i-1} + k - 1;$
$y_i = y_{i-1} + 1.$

Осталось только заметить, что $x_0 = x_a$, $y_0 = y_a$, $c_0 = 0$, и мы приходим к следующему коду:

```
void drawLine3 ( TgaImage& image, int xa, int ya, int xb, int yb, uint32_t color )
{
      const float k = (float)(yb-ya)/(float)(xb - xa);
      float y = ya;
      float c = 0;
      int   y = ya;
```

```
      image.putPixel ( xa, ya, color );

      for ( int x = xa+1; x <= xb; x++ )
      {
            if ( (c += k) > 0.5f )
            {
                  c--;
                  y++;
            }

            image.putPixel ( x, y, color );
      }
}
```

Обратите внимание, что мы постоянно сравниваем c_i со значением 1/2. Однако с точки зрения процессора гораздо удобнее сравнивать с нулем. Для этого давайте вместо величины c_i введем новую величину $d_i = 2c_i - 1$. Тогда $d_0 = -1$, и мы будем сравнивать d_i с нулем.

```
void drawLine4 ( TgaImage& image, int xa, int ya, int xb, int yb, uint32_t color )
{
      const float k = (float)(yb-ya)/(float)(xb - xa);
      float y = ya;
      float d = 2*k - 1;
      int   y = ya;

      image.putPixel ( xa, ya, color );

      for ( int x = xa+1; x <= xb; x++ )
      {
            if ( d > 0 )
            {
                  d += 2*k + 2;
                  y++;
            }
            else
                  d += 2*k;

            image.putPixel ( x, y, color );
      }
}
```

Мы по-прежнему используем вещественные числа и операции над ними, несмотря на то что все входные величины (как и результирующие координаты пикселов) являются целочисленными. Можно легко избавиться от вещественных чисел, если заметить, что все используемые вещественные числа имеют вид $\frac{n}{x_b - x_a}$, где n – некоторое целое число.

Теперь нам осталось только умножить уравнение для d_i на $x_b - x_a$, и мы придем к версии алгоритма, вообще не использующей вещественных чисел (равно как и операций умножения и деления).

```
void drawLine5 ( TgaImage& image, int xa, int ya, int xb, int yb, uint32_t color )
{
  const int dx = xb - xa;
  const int dy = yb - ya;
  const int d1 = dy << 1;
  const int d2 = (dy - dx) << 1;
  int       d  = (dy<<1) - dx;
  int       y  = ya;

  image.putPixel ( xa, ya, color );
  for ( int x = xa+1; x <= xb; x++ )
  {
    if ( d > 0 )
    {
      d += d2;
      y++;
    }
    else
      d += d1;

    image.putPixel ( x, y, color );
  }
}
```

Завершающим шагом будет обобщение алгоритма, которое позволит нам отказаться от ранее наложенных ограничений на координаты концов отрезка. Для этого заметим, что случай $0 \leq -(y_b - y_a) \leq x_b - x_a$ полностью симметричен ранее рассмотренному – нам нужно только поменять приращение для y.

Случай $0 \leq y_b - y_a > x_b - x_a$ также легко сводится к ранее рассмотренному просто путем перестановки местами переменных x и y.

Таким образом, мы приходим к следующему коду на C++, который позволяет строить растровые развертки любых отрезков (здесь нет специальных оптимизаций для вертикальных и горизонтальных отрезков, которые обычно добавляются).

```
void drawLine ( TgaImage& image, int xa, int ya, int xb, int yb, uint32_t color )
{
      const int d  = (dy<<1) - dx;
      const int dx = abs(xb - xa);
      const int dy = abs(yb - ya);
      const int sx = xb >= xa ? 1 : -1;    // знак of xb-xa
      const int sy = yb >= ya ? 1 : -1;    // знак of yb-ya

      if ( dy <= dx )
      {
            int d  = (dy<<1) - dx;
            int d1 = dy << 1;
            int d2 = (dy - dx) << 1;
            int x  = xa + sx;
            int y  = ya;

            image.putPixel ( xa, ya, color );
```

```
            for ( int i = 1; i <= dx; i++, x += sx )
            {
                  if ( d > 0 )
                  {
                        d += d2;
                        y += sy;
                  }
                  else
                        d += d1;

                  image.putPixel ( x, y, color );
            }
      }
      else
      {
            int d  = (dx<<1) - dy;
            int d1 = dx << 1;
            int d2 = (dx - dy) << 1;
            int x  = xa;
            int y  = ya + sy;

            image.putPixel ( xa, ya, color );
            for ( int i = 1; i <= dy; i++, y += sy )
            {
                  if ( d > 0 )
                  {
                        d += d2;
                        x += sx;
                  }
                  else
                        d += d1;

                  image.putPixel ( x, y, color );
            }
      }
}
```

Алгоритм Брезенхейма для окружности

Использованный прием для построения отрезка можно адаптировать и на случай построения растровой развертки окружности. С использованием симметричности нам достаточно построить растровую развертку лишь 1/8 части окружности (рис. 7.7).

Итак, пусть у нас есть окружность радиуса R с центром в начале координат. Тогда нам достаточно построить развертку ее части, соответствующей значениям x от нуля до $R/\sqrt{2}$. Важным свойством этой части окружности является то, что угловой коэффициент на ней всегда лежит на отрезке [–1, 0]. Ниже приводится функция `drawCirclePoints`, которая, получив координаты одной точки на заданной части окружности, строит сразу все 8 точек, используя симметрию.

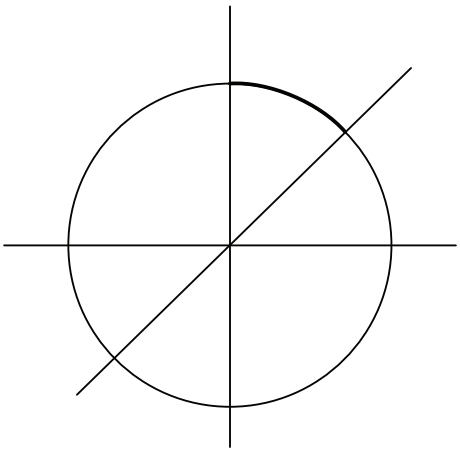

Рис. 7.7 ❖ Симметрия
для построения окружности НУМЕРАЦИЯ

```
void drawCirclePoints ( TgaImage& image, int xc, int yc, int x, int y, uint32_t color )
{
  image.putPixel ( xc + x, yc + y, color );
  image.putPixel ( xc + y, yc + x, color );
  image.putPixel ( xc + y, yc - x, color );
  image.putPixel ( xc + x, yc - y, color );
  image.putPixel ( xc - x, yc - y, color );
  image.putPixel ( xc - y, yc - x, color );
  image.putPixel ( xc - y, yc + x, color );
  image.putPixel ( xc - x, yc + y, color );
}
```

Нам понадобится способ, как для произвольной точки (x, y) проверить, лежит ли она внутри нашей окружности или же вне нее. Для этого мы будем использовать следующую функцию:

$$F(x, y) = x^2 + y^2 - R^2.$$

На самой окружности эта функция принимает нулевое значение, внутри окружности – отрицательные, вне – положительные (рис. 7.8).

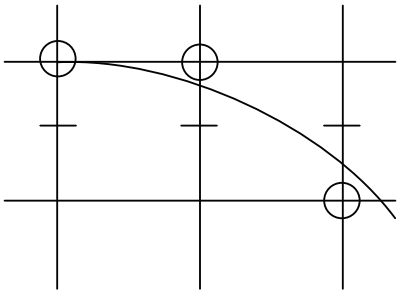

Рис. 7.8 ❖ Построение окружности

Строить последовательность пикселов мы будем, как и ранее, используя рекуррентные формулы. Нашей стартовой точкой будет точка $x_0 = 0, y_0 = R$. Формула для перехода к следующему значению x крайне проста:

$$x_i = x_{i-1} + 1.$$

Для y_i у нас есть всего два возможных значения – y_{i-1} или $y_{i-1} - 1$. Чтобы определить, какое именно из этих значений мы будем использовать, построим точку, лежащую точно посередине между этими вариантами, – $\left(x_{i-1}+1, y_{i-1}-\frac{1}{2}\right)$. После этого мы проверим, лежит она внутри окружности или вне нее. Это легко можно сделать, введя следующую переменную:

$$d_i = F\left(x_i+1, y_i-\tfrac{1}{2}\right) = (x_i+1)^2 + \left(y_i-\tfrac{1}{2}\right)^2 - R^2.$$

Если точка $\left(x_{i-1}+1, y_{i-1}-\frac{1}{2}\right)$ лежит внутри окружности (т. е. $d_i < 0$, то мы полагаем $y_i = y_{i-1}$. В противном случае $y_i = y_{i-1} - 1$.

Выведем рекуррентные формулы для d_i. В случае $d_i < 0$ мы имеем:

$$d_i - d_{i-1} = 2x_i + 3.$$

Если же $d_i \geq 0$ (и $y_i = y_{i-1}$), то мы имеем

$$d_i - d_{i-1} = 2(x_i - y_i) + 5.$$

Нам осталось лишь найти начальное значение для d_i:

$$d_0 = F\left(0+1, R-\tfrac{1}{2}\right) = \frac{5}{4} - R.$$

В результате мы приходим к следующему коду:

```
void drawCircle1 ( TgaImage& image, int xc, int yc, int r, uint32_t color )
{
    int   x = 0;
    int   y = r;
    float d = 1.25f - r;

    drawCirclePoints ( image, xc, yc, x, y, color );

    while ( y > x )
    {
        if ( d < 0 )
        {
            d += 2*x + 3;
            x++;
        }
        else
        {
            d += 2*(x-y)+5;
            x++;
            y--;
```

```
        }
        drawCirclePoints ( image, xc, yc, x, y, color );
    }
}
```

Обратите внимание, что дробная часть d_0 равна ¼ и при этом d_i всегда изменяется на целое число. Таким образом, дробная часть d_i всегда будет равна ¼, и мы можем просто ее игнорировать. Тогда d_i становится просто целым числом, и мы получаем приведенный ниже код.

```
void drawCircle2 ( TgaImage& image, int xc, int yc, int r, uint32_t color )
{
    int x      = 0;
    int y      = r;
    int d      = 1 - r;
    int delta1 = 3;
    int delta2 = -2*r+5;

    drawCirclePoints ( image, xc, yc, x, y, color );

    while ( y > x )
    {
        if ( d < 0 )
        {
            d      += delta1;
            delta1 += 2;
            delta2 += 2;
            x++;
        }
        else
        {
            d      += delta2;
            delta1 += 2;
            delta2 += 4;
            x++;
            y--;
        }

        drawCirclePoints ( image, xc, yc, x, y, color );
    }
}
```

Заполнение треугольника

Обычно трехмерные объекты в компьютерной графике представлены в виде наборов треугольников (triangle mesh). Именно поэтому задача растеризации треугольника встречается крайне часто, и современные GPU имеют аппаратную поддержку растеризации треугольников.

Здесь мы рассмотрим, каким образом можно осуществить растеризацию треугольника без помощи GPU и какие проблемы при этом возникают.

Простейшим подходом к растеризации треугольника является следующий: давайте просто растеризуем все его ребра. В результате для каждой присутствующей

в растеризации *y*-координаты мы получим отрезок значений *x*, которые должны быть заполнены пикселами.

Проблема этого подхода заключается в том, что в результате такой растеризации треугольника мы можем получить пикселы, вообще не лежащие внутри треугольника (рис. 7.9), что крайне нежелательно.

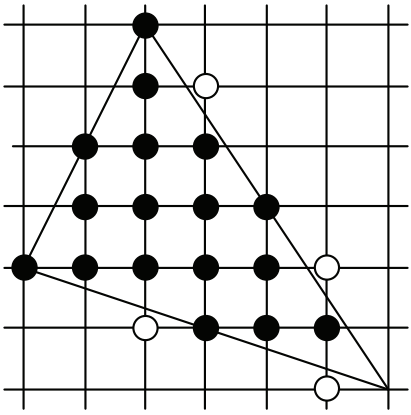

Рис. 7.9 ❖ Получение пикселов, не лежащих внутри треугольника

Другая возникающая проблема заключается в том, что часто данный треугольник имеет общие ребра с другими треугольниками (рис. 7.10), и желательно, чтобы при растеризации такой группы треугольников каждый выводимый пиксел был выведен ровно один раз (т. е. необходимо избежать дублирования пикселов на общих ребрах).

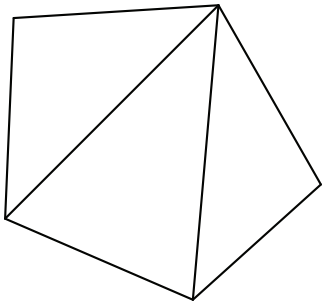

Рис. 7.10 ❖ Общие ребра при выводе группы треугольников

Для решения этой проблемы обычно используют следующее правило (top-left rule): пиксел выводится только тогда, когда его центр лежит либо строго внутри треугольника, либо на верхнем или левом ребре треугольника. Под верхним ребром понимается горизонтальное ребро, находящееся выше всех остальных ребер (рис. 7.11). Обратите внимание, что у треугольника может быть два левых ребра.

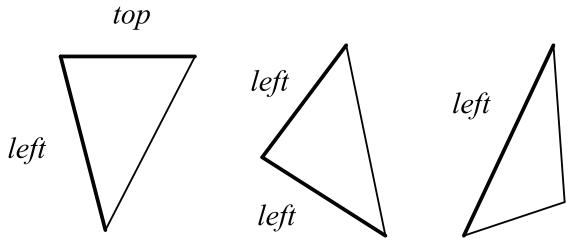

Рис. 7.11 ❖ Верхнее и левые ребра треугольника

На рис. 7.12 приведены примеры растеризации треугольников с использованием этого правила. Также на рис. 7.13 приведены результаты растеризации двух треугольников и прямоугольника, составленного из них. Обратите внимание, что при растеризации нет повторяющихся или пропущенных пикселов.

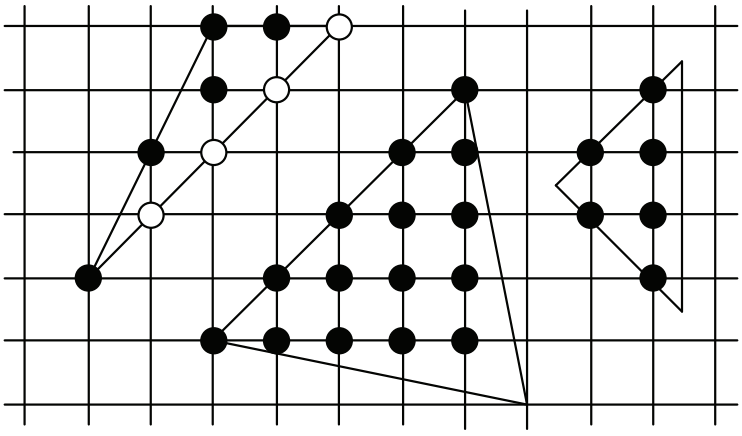

Рис. 7.12 ❖ Растеризация треугольника с использованием top-left rule

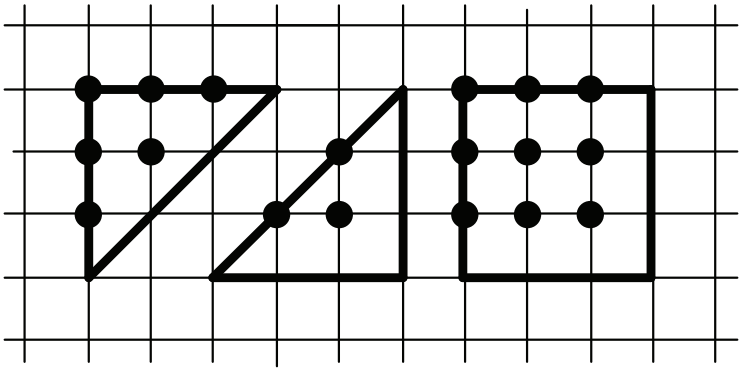

Рис. 7.13 ❖ Растеризация прямоугольника, состоящего из двух треугольников

Для алгоритма растеризации треугольника мы воспользуемся очень простым приемом. Сначала мы найдем *AABB* для нашего треугольника (по его вершинам). Затем мы просто проверим все попадающие в этот *AABB* пикселы на попадание внутрь треугольника (или на верхнее или левые ребра).

По произвольному ребру треугольника мы можем построить уравнение прямой, проходящей через него (вида $Ax + By + C = 0$). Обратите внимание, что это уравнение всегда может быть построено таким образом, что все его коэффициенты будут целыми числами.

Эта прямая разбивает всю плоскость на две полуплоскости – положительную ($Ax + By + C > 0$) и отрицательную ($Ax + By + C < 0$). Будем считать, что наш треугольник всегда лежит в положительной полуплоскости. Если это не так, то мы просто умножим уравнение прямой на –1. Для проверки того, что треугольник лежит в положительной полуплоскости, нам достаточно подставить третью вершину треугольника в уравнение и проверить знак.

Ниже приводится функция, которая по заданным вершинам треугольника строит такое уравнение прямой, проходящей через первые две вершины, что третья вершина всегда лежит в положительной полуплоскости.

```
inline void buildPlane ( const point& p1, const point& p2,
                         const point& p3,
                         int& a, int& b, int&c )
{
      a = p2.y - p1.y;                // найдем уравнение прямой по точкам p1 и p2
      b = p1.x - p2.x;
      c = -a*p1.x - b*p1.y;

      if ( a*p3.x + b*p3.y + c < 0 )   // проверим, лежит ли p3 в положительной полуплоскости
      {
            a = -a;
            b = -b;
            c = -c;
      }
}
```

Таким образом, мы строим все три уравнения прямых, проходящих через ребра треугольника, и для каждого такого уравнения определяем, соответствует ли эта прямая верхнему или левому ребру.

Далее мы проверяем каждый пиксел из *AABB* нашего треугольника на попадание внутрь. Для этого мы просто подставляем его координаты в каждое из трех уравнений прямых и проверяем знак получившегося значения. Для левого или верхнего ребра мы проверяем, что значение больше или равно нулю, во всех остальных случаях – строго больше нуля. Если хотя бы для одного из трех уравнений условие не выполнено, то пиксел отбрасывается.

Поскольку мы проверяем все пикселы подряд, то для вычисления функций вида $f(x, y) = Ax + By + C$ мы можем использовать рекуррентные формулы. Соответствующий код приводится ниже.

Если считать, что вершины треугольника задаются с направлением обхода против часовой стрелки, то левое ребро – это ребро, идущее вниз. Верхнее ребро – это ребро, идущее влево.

```
inline static bool isEdgeLeft ( const point& p1, const point& p2 )
{
      return p1.y < p2.y;
}

inline static bool isEdgeTop ( const point& p1, const point& p2 )
{
      return (p1.y == p2.y) && (p1.x >= p2.x);
}

void rasterizeTriangle( TgaImage& image, point p [], uint32_t color )
{
      int xMin = p[0].x;
      int yMin = p[0].y;
      int xMax = p[0].x;
      int yMax = p[0].y;
      int a [3], b[3], c[3];

            // найдем AABB для треугольника
      for ( int i = 1; i < 3; i++ )
      {
            if ( p [i].x < xMin )
                  xMin = p[i].x;
            else
            if ( p [i].x > xMax )
                  xMax = p[i].x;

            if ( p [i].y < yMin )
                  yMin = p[i].y;
            else
            if ( p [i].y > yMax )
                  yMax = p[i].y;
      }

      buildPlane ( p[0], p[1], p[2], a[0], b[0], c[0] );   // построим уравнения ребер
      buildPlane ( p[0], p[2], p[1], a[1], b[1], c[1] );
      buildPlane ( p[1], p[2], p[0], a[2], b[2], c[2] );

      bool      edgeInside [3];

      edgeInside[0] = isEdgeTop(p [0], p [1]) || isEdgeLeft( p [0], p [1] );
      edgeInside[1] = isEdgeTop(p [2], p [0]) || isEdgeLeft( p [2], p [0] );
      edgeInside[2] = isEdgeTop(p [1], p [2]) || isEdgeLeft( p [1], p [2] );

      int d0 = a[0]*xMin + b[0]*yMin + c[0];   // найдем значения функций в левом нижнем углу
      int d1 = a[1]*xMin + b[1]*yMin + c[1];
      int d2 = a[2]*xMin + b[2]*yMin + c[2];

      for ( int y = yMin; y <= yMax; y++ )     // проверяем все точки из AABB
      {
```

```
            int f0 = d0,
                f1 = d1,
                f2 = d2;

            d0 += b[0];
            d1 += b[1];
            d2 += b[2];

            for ( int x = xMin; x <= xMax;
                x++, f0 += a[0], f1 += a[1], f2 += a[2] )
            {
                if ( f0 < 0 || (f0 == 0 && !edgeInside [0]) )
                    continue;

                if ( f1 < 0 || (f1 == 0 && !edgeInside [1]) )
                    continue;

                if ( f2 < 0 || (f2 == 0 && !edgeInside [2]) )
                    continue;

                image.putPixel( x, y, color );
            }
        }
}
```

Заполнение области, заданной цветом границы

Еще одним примером заполнения является так называемый *flood fill*. В этом случае у нас есть область на растровой сетке, заданная цветом своей границы. При этом область должна быть связной, но она может быть невыпуклой и содержать дырки внутри себя (рис. 7.14).

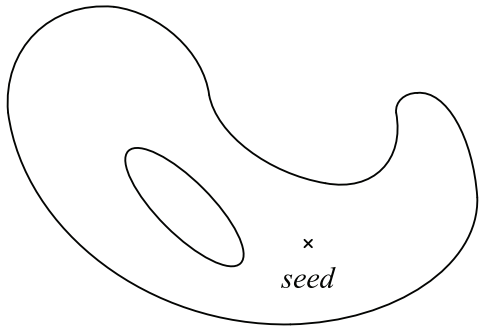

Рис. 7.14 ❖ Пример заполняемой области

Наша задача заключается в том, чтобы по заданной точке, лежащей внутри области, закрасить всю область заданным цветом.

Для решения подобной задачи можно предложить крайне простое решение, занимающее всего несколько строчек кода:

```
void boundaryFill ( TgaImage& image, int x, int y, uint32_t borderColor, uint32_t fillColor )
{
  uint32_t c = image.getPixel ( x, y );

  if ( (c != borderColor) && (c != fillColor) )
  {
    image.putPixel     ( x, y, fillColor );

    boundaryFill ( image, x - 1, y, borderColor, fillColor );
    boundaryFill ( image, x + 1, y, borderColor, fillColor );
    boundaryFill ( image, x, y - 1, borderColor, fillColor );
    boundaryFill ( image, x, y + 1, borderColor, fillColor );
  }
}
```

Несмотря на свою простоту, этот алгоритм может заполнять крайне сложные области. Однако он делает это довольно неэффективно и приводит к очень глубокой рекурсии. Поэтому на практике для подобных задач обычно используют другой подход.

Основным способом повышения эффективности является использование *когерентности* (coherence) (мы столкнемся с этим в главе 8). В данном случае когерентность означает, что если пиксел (x, y) принадлежит нашей области (и должен быть закрашен), то, скорее всего, и соседние с ним пикселы также принадлежат ей.

Мы будем использовать когерентность следующим образом: для каждого пиксела (x, y) мы будем искать такой горизонтальный отрезок (x_l, y) – (x_r, y), который полностью содержится в нашей области. Найдя такой отрезок, мы заполняем его и ищем следующий отрезок ниже (рис. 7.15).

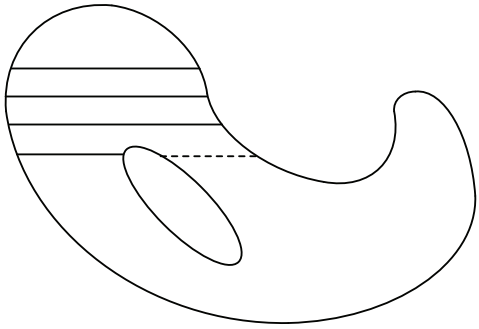

Рис. 7.15 ❖ Заполнение области горизонтальными отрезками

Обратите внимание, что мы не можем всегда идти только вниз. Иногда нам придется идти и вверх тоже. На самом деле нам достаточно просто проверять пикселы над только что заполненным отрезком и при необходимости обрабатывать их рекурсивно (рис. 7.16).

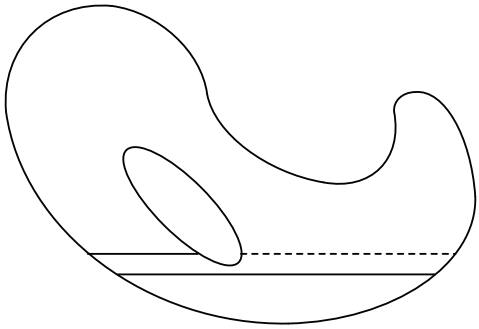

Рис. 7.16 ❖ В этом случае нам нужно не только идти вниз, но и подняться наверх

В результате мы получаем следующую функцию.

```
int lineFill ( TgaImage& image, int xx, int y, int dir,
               int prevXl, int prevXr, uint32_t borderColor,
               uint32_t fillColor )
{
      int         xl = xx;
      int         xr = xx;
      uint32_t  c;

      do
            c = image.getPixel ( --xl, y );
      while ( (c != borderColor) && (c != fillColor) );

      do
            c = image.getPixel ( ++xr, y );
      while ( (c != borderColor) && (c != fillColor) );

      xl++;
      xr--;

      for ( int x = xl; x <= xr; x++ )           // Заполняем отрезок
            image.putPixel ( x, y, fillColor );

      if ( y + dir >= 0 && y + dir < image.getHeight () )
            for ( int x = xl; x <= xr; x++ )
            {
                  c = image.getPixel ( x, y + dir );
                  if ( (c != borderColor) && (c != fillColor) )
                        x = lineFill ( image, x, y + dir, dir, xl, xr,
                                       borderColor, fillColor );
            }

      if ( y - dir >= 0 && y - dir <= image.getHeight () )
            for ( int x = xl; x < prevXl; x++ )
            {
                  c = image.getPixel ( x, y - dir );
```

```
                if ( (c != borderColor) && (c != fillColor) )
                      x = lineFill ( image, x, y - dir, -dir, xl, xr,
                                     borderColor, fillColor );
          }

     for ( int x = prevXr; x < xr; x++ )
     {
          c = image.getPixel ( x, y - dir );
          if ( (c != borderColor) && (c != fillColor) )
                x = lineFill ( image, x, y - dir, -dir, xl, xr,
                               borderColor, fillColor );
     }

     return xr;
}
```

Данная функция также является рекурсивной, но рекурсия в этом случае не будет столь глубокой, как в ранее рассмотренном варианте. Для заполнения всей области служит функция `boundaryFill`.

```
void boundaryFill ( int x, int y, uint32_t borderColor, uint32_t fillColor )
{
     lineFill ( x, y, 1, x, x, borderColor, fillColor );
}
```

Глава 8

Удаление невидимых линий и поверхностей

При рендеринге различных трехмерных моделей (на двухмерную плоскость, т. е. экран) мы постоянно сталкиваемся с тем, что при проектировании одни объекты могут закрывать собой другие объекты или их части. Иногда сложный объект сам может закрывать собой отдельные свои части.

Соответственно, возникает задача – как для заданного способа проектирования определить, что именно будет в результате видно, а что будет закрыто.

Далее в этой главе мы будем рассматривать перспективное проектирование с центром в точке c. Саму плоскость, на которую будет осуществляться проектирование, мы будем далее называть *картинной плоскостью*. Однако на самом деле все изложенное также относится и к случаю параллельного проектирования.

Произвольный луч, выпущенный из центра проектирования и пересекающий картинную плоскость, называется *проектором*. Пусть наш проектор пересекает сразу два различных объекта в точках p_1 и p_2. Тогда обе эти точки будут проектироваться в одну и ту же точку картинной плоскости (экрана). Мы будем говорить, что точка p_1 *загораживает* (закрывает) собой точку p_2, если расстояние от центра проектирования до p_1 меньше, чем расстояние от центра проектирования до p_2 (рис. 8.1).

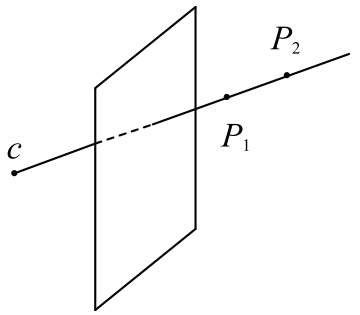

Рис. 8.1 ❖ Две точки, лежащие на одном проекторе

Соответственно, наша задача будет заключаться в определении того, что будет видно в каждой точке картинной плоскости. Несмотря на простоту формулировки, эта задача является достаточно сложной и может требовать большого количества вычислительных ресурсов.

Если посмотреть на классические игры жанра FPS (такие как Wolfenstein 3D, Doom, Quake), то алгоритм определения видимости играл в них ключевую роль.

Существует большое количество различных методов для определения видимости, включая и методы, использующие аппаратную поддержку. Ниже мы рассмотрим основные такие методы и подходы.

Сами по себе эти методы различаются по следующим основным параметрам:

- способу представления объектов;
- способу визуализации объектов;
- пространству, в котором осуществляется анализ видимости;
- виду (точности) получаемого результата.

В качестве способа представления объектов, помимо полигонального представления (т. е. в виде набора треугольников), могут выступать также явное или неявное аналитическое, параметрическое, воксельное (объемное) представления. Почти все рассматриваемые методы работают с полигональным представлением, являющимся наиболее распространенным.

В качестве способа визуализации обычно выступает *каркасное* представление (wireframe, в виде набора ребер) и *сплошное* представление (solid, в виде закрашенных граней) (рис. 8.2).

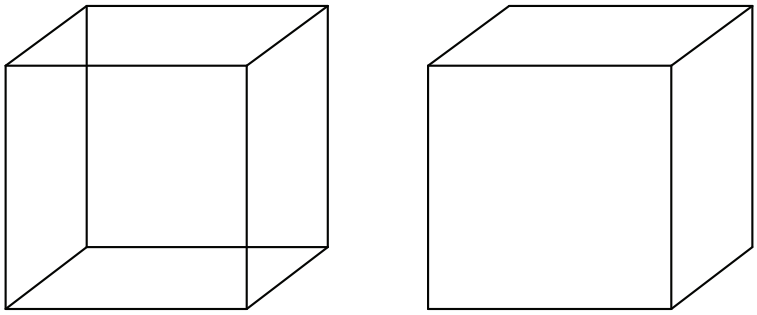

Рис. 8.2 ❖ Каркасное и сплошное представления

При каркасном представлении возникает задача определения того, какие части каких ребер будут видны, а какие будут закрыты гранями объекта (хотя сами грани мы не выводим). При сплошном способе визуализации мы определяем видимость уже не для ребер, а для граней или частей объектов.

Сам анализ видимости объектов можно проводить в исходном трехмерном пространстве или же на картинной плоскости. Соответственно, методы делятся на:

- методы, работающие непосредственно в пространстве объектов ($\mathbb{R}^3$);
- методы, работающие в картинной плоскости ($\mathbb{R}^2$), т. е. работающие с проекциями объектов.

Получаемый результат представляет собой либо набор видимых областей или отрезков, либо информацию о видимом (ближайшем) объекте для каждого пиксела картинной плоскости.

В первом случае говорится о *непрерывном* (continuous) результате, который никак не привязан к растровым свойствам картинной плоскости. В простейшем случае такие методы сравнивают каждый объект (грань) с каждым другим объектом (гранью) и имеют сложность $O(n^2)$, где n – общее число объектов (граней). Кроме того, подобные методы обычно бывают довольно сложными.

Методы второго класса (point-sampling) дают приближенное решение задачи определения видимости, поскольку определяют видимость только в узлах растровой решетки. Подобные методы обычно довольно просты и, как правило, имеют сложность $O(np)$, где через p обозначено количество пикселов на экране. При этом следует иметь в виду, что этим методам обычно свойственны *ошибки дискретизации* (aliasing artifacts).

Также есть большое число гибридных методов, сочетающих в себе разные подходы, например выполняющих часть расчетов точно, а часть – дискретно. Большинство используемых в настоящее время методов определения видимости сочетает в себе сразу несколько различных базовых алгоритмов.

Методы удаления невидимых линий и поверхностей тесно связаны с сортировкой. В некоторых случаях она присутствует в явном виде, а в некоторых скрыта. Многие методы фактически отличаются именно способом проведения сортировки.

Для оптимизации методов определения видимости широко используются различные виды пространственных индексов, например равномерные сетки или *kD*-деревья.

Далее мы рассмотрим несколько базовых понятий, играющих важную роль в различных методах определения видимости.

Лицевые и нелицевые грани

Пусть у нас в пространстве есть замкнутый многогранник. Тогда для каждой его грани мы можем определить вектор *внешней нормали* (обычно единичный) n.

Рассмотрим произвольную грань такого многогранника и выберем на ней некоторую точку. В этой точке можно определить вектор l, задающий направление на центр проектирования c. Обычно это просто вектор, направленный от выбранной точки P до центра проектирования.

Теперь рассмотрим угол между этим вектором l и вектором внешней нормали n. Если этот угол острый, то соответствующую грань мы будем называть *лицевой* (front facing). В противном случае грань называется *нелицевой* (back facing) (рис. 8.3).

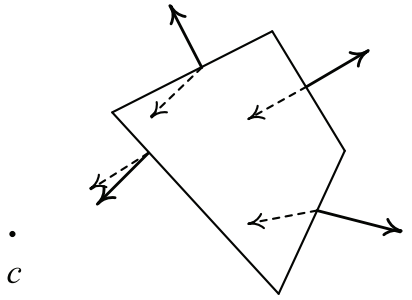

Рис. 8.3 ❖ Лицевая и нелицевая грани

На самом деле нет необходимости явно находить угол между векторами l и n – достаточно просто найти знак их скалярного произведения. Если $(n, l) > 0$, то угол острый и грань является лицевой. В противном случае грань является нелицевой.

Может показаться, что в определении лицевой грани есть некорректность – выбор точки P на грани. На самом деле легко можно показать, что тип грани не за-

висит от выбора точки на ней. Само понятие лицевой грани можно ввести другим, полностью эквивалентным образом.

Для этого проведем через грань плоскость и проверим, лежит ли центр проектирования в том же полупространстве, куда показывает вектор внешней нормали (рис. 8.4). Если это так, то эта грань является лицевой. В противном случае грань является нелицевой.

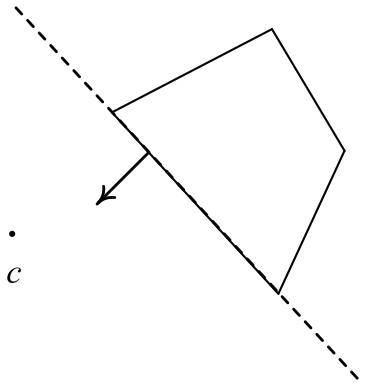

Рис. 8.4 ❖ Альтернативное определение лицевой грани

Если мы выпустим произвольный луч из центра проектирования *c*, то (по аналогии с теоремой Жордана) число его пересечений с лицевыми гранями будет равно числу его пересечений с нелицевыми гранями. Таким образом, лицевые грани составляют примерно половину от общего числа граней.

С точки зрения удаления невидимых граней важным является то, что ни одна нелицевая грань никогда не может быть видна при проектировании. В случае когда вся сцена состоит из всего одного выпуклого и замкнутого тела, то удаление нелицевых граней полностью решает задачу определения видимости – все лицевые грани (и только они) будут полностью видны.

В общем случае нелицевые грани видны по-прежнему не будут, но часть лицевых граней будет закрыта полностью или частично другими лицевыми гранями (рис. 8.5).

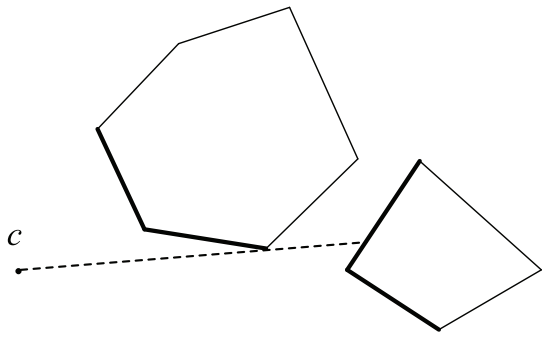

Рис. 8.5 ❖ Лицевые грани одного объекта могут при проектировании закрывать собой другие лицевые грани (этого же или другого объекта)

Сложность по глубине

Выпустим луч из центра проектирования через произвольную точку на картинной плоскости. Число лицевых граней, которые пересечет этот луч, называется *сложностью по глубине* (depth complexity), соответствующей точке картинной плоскости (рис. 8.6).

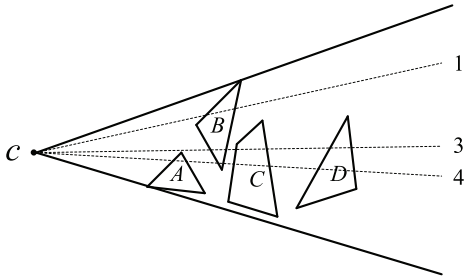

Рис. 8.6 ❖ Сложность по глубине

Простейшим случаем является тот случай, когда луч пересекает не более одной лицевой грани – в этом случае именно пересекаемая грань, и только она, будет видна в данной точке картинной плоскости.

Чем выше сложность по глубине, тем более сложным и дорогим является определение видимости. Так, если у нас сложность по глубине равна 10, то это значит, что на данную точку проектируется сразу 10 лицевых граней. При этом видна в этой точке будет всего одна грань, а остальные будут не видны.

Можно ввести понятие сложности по глубине не для одной точки, а для всей картинной плоскости. Для этого мы просто усредняем сложность по глубине (или берем ее максимальное значение). В результате мы получаем оценку сложности трехмерной сцены для выбранного способа проектирования.

Если рассмотреть современные трехмерные игры, например FPS (First Person Shooter), то, как легко можно заметить, почти все они отличаются очень высоким уровнем сложности по глубине.

Загораживание

Крайне важным понятием для определения видимости является понятие *загораживателя* (occluder). Под загораживателем понимается грань (или целый объект), закрывающая от наблюдателя в центре проектирования какие-то другие грани или объекты (рис. 8.7).

Так, на рис. 8.7 объект *A* выступает в роли загораживателя и полностью закрывает собой объекты *B* и *C*. Обратите внимание, что объект *B* также выступает в роли загораживателя, так как он полностью закрывает собой объект *C*.

Если у нас есть не один загораживатель, а несколько, то зачастую вместе они могут закрывать даже те объекты, которые они не могут закрыть по отдельности. Так, на рис. 8.8 объекты *A* и *B* вместе полностью закрывают объекты *C* и *D*. Для обозначения этого свойства обычно используется термин *occlude fusion*.

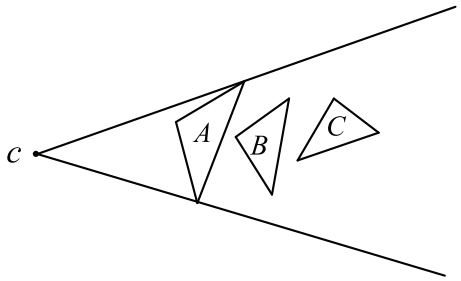

Рис. 8.7 ❖ Загораживатель

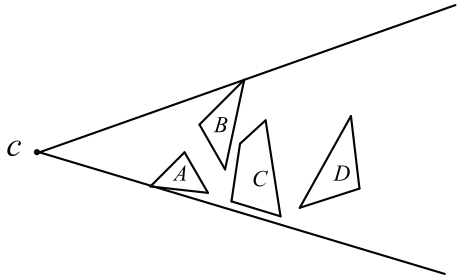

Рис. 8.8 ❖ Ни объект *A*, ни объект *B* сами по себе не могут закрыть объекты *C* и *D*. При этом вместе они легко справляются с этой задачей

Распространенным приемом в современных играх является выбор некоторого набора загораживателей (на этапе подготовки геометрии). Далее они используются для того, чтобы отбрасывать заведомо невидимые объекты.

Следует иметь в виду, что использование загораживателей дает лишь консервативную оценку видимости: если объект не был отброшен при помощи заданного набора загораживателей, то это не значит, что он обязательно будет виден.

Когерентность

Когда мы рассматривали заполнение плоских фигур, то столкнулись с таким понятием, как *когерентность* (coherence). В задачах определения видимости когерентность также играет большую роль, позволяя заметно повысить эффективность используемых алгоритмов. Под когерентностью обычно понимается то, что близко расположенные объекты, как правило, обладают схожими свойствами.

Обычно выделяют следующие виды когерентности:

- когерентность в пространстве объектов (object coherence);
- когерентность между гранями;
- когерентность ребер;
- когерентность в пространстве картинной плоскости (между пикселами);
- временная когерентность.

Под когерентностью в пространстве объектов понимается то, что близко расположенные объекты, скорее всего, видны (или не видны) одновременно.

Когерентность граней означает, что соседние грани одного и того же объекта, как правило, видны или не видны одновременно.

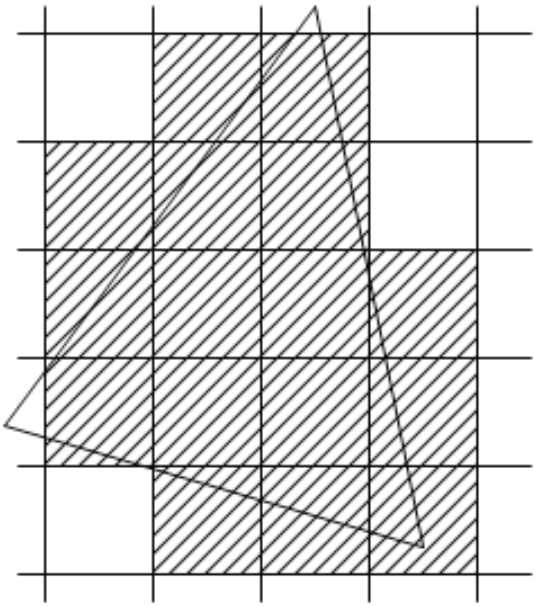

Рис. 8.9 ❖ Когерентность пикселов

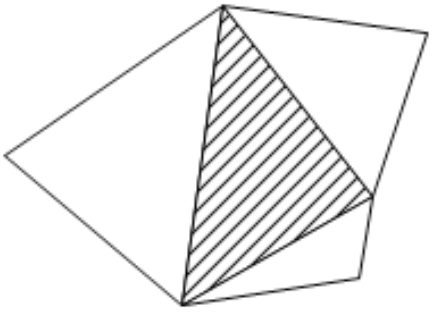

Рис. 8.10 ❖ Когерентность между гранями

Когерентность ребер означает, что близко расположенные ребра обычно принадлежат одной грани и видны (или не видны) одновременно.

Когерентность в пространстве картинной плоскости означает, что соседние пикселы на картинной плоскости, скорее всего, соответствуют одной и той же грани (одному и тому же объекту) и видны (или не видны) одновременно.

Временная когерентность означает, что видимость граней от кадра к кадру анимации меняется очень слабо. Таким образом, если грань видна в предыдущем кадре, то она, скорее всего, будет также видна и в текущем кадре.

Обратите внимание, что когерентность – это не строгий закон, а скорее общая тенденция. Тем не менее аккуратное использование этой тенденции позволяет заметно повысить быстродействие алгоритмов определения видимости.

Удаление невидимых линий. Алгоритм Робертса

Для правильного построения каркасных изображений важно уметь отбрасывать невидимые (т. е. закрытые гранями) части ребер. Первым алгоритмом для решения этой задачи был алгоритм Робертса.

В данном алгоритме все грани считаются выпуклыми многоугольниками. В самом начале определяются лицевые и нелицевые грани. Любое ребро, для которого обе проходящие через него грани являются нелицевыми, сразу же отбрасывается, так как оно никогда не может быть видно.

Дальше каждое из оставшихся ребер проверяется на закрывание каждой из лицевых граней. При проверке на закрывание ребра (или части ребра) и грани возможны следующие случаи:

- грань не закрывает ребро (даже частично) (рис. 8.11а);
- грань полностью закрывает ребро (рис. 8.11б);
- грань частично закрывает ребро (рис. 8.11в).

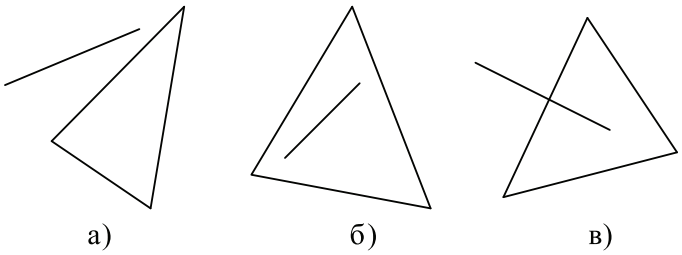

Рис. 8.11 ❖ Возможные случаи закрывания гранью ребра

В первом случае мы просто переходим к следующей грани. Во втором случае ребро отбрасывается, и мы переходим к следующему ребру. В третьем случае мы оставляем только ту часть (части) ребра, которая не закрыта гранью.

Проверка на закрывание отрезка гранью сводится к нахождению точек пересечения прямой, проходящей через проекцию отрезка с ребрами проекции грани.

Если есть пересечение прямой с ребрами грани, то необходима проверка, что лежит ближе к центру проектирования – отрезок или грань. Это сводится к проверке, лежит ли отрезок в том же полупространстве относительно плоскости, проходящей через грань, что и центр проектирования.

В процессе работы ребра могут разбиваться на части. В конце работы алгоритма мы получим окончательный список видимых ребер.

При общем числе граней n сложность данного алгоритма составляет $O(n^2)$. Можно заметно повысить эффективность этого алгоритма, если использовать равномерное разбиение картинной плоскости. При этом каждой клетке такого разбиения сопоставляется список всех пересекающих ее граней. В этом случае сложность алгоритма можно свести к $O(n)$.

Понятие количественной невидимости. Алгоритм Аппеля

Рассмотрим произвольное гладкое замкнутое выпуклое тело в пространстве. Тогда для произвольной точки P на границе этого объекта будет определен вектор внешней нормали n.

Назовем эту точку лицевой, если $(n, l) \geq 0$, где l – это вектор из этой точки к центру проектирования ($l = P - c$). В случае когда $(n, l) < 0$, эту точку назовем нелицевой.

Если $(n, l) > 0$, то в силу гладкости поверхности существует некоторая окрестность точки P, такая что все ее точки также будут лицевыми. Аналогично, если $(n, l) < 0$, то существует окрестность этой точки, состоящая целиком из нелицевых точек.

Для точки, в которой $(n, l) = 0$, подобной окрестности может и не существовать – рассмотрим, например, в качестве тела обычную сферу. Тогда все точки, для которых $(n, l) = 0$, образуют окружность, которая разбивает нашу сферу на две части – лицевую и нелицевую (рис. 8.12).

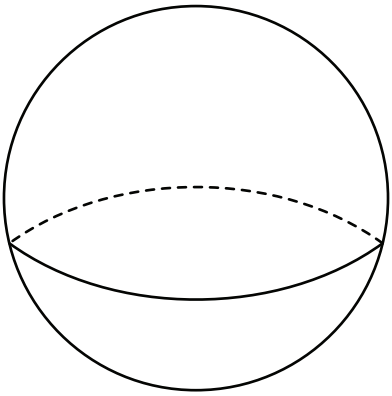

Рис. 8.12 ❖ Разбиение сферы на лицевую и нелицевую части

Подобные точки (т. е. точки, для которых $(n, l) = 0$) называются *нерегулярными* точками проектирования. В общем случае множество всех нерегулярных точек образует на поверхности объекта некоторую кривую, называемую *контурной линией*.

Эта линия разбивает всю поверхность на части, каждая из которых однозначно проектируется на картинную плоскость. Одна из этих частей будет полностью видимой, а остальные – полностью невидимыми.

Обобщим понятие контурной линии на случай нескольких невыпуклых тел. В этом случае у нас будет уже несколько контурных линий, разбивающих границы тел на отдельные области. Каждая из этих областей будет взаимно однозначно проектироваться на картинную плоскость и состоять либо только из лицевых точек, либо только из нелицевых точек.

Несмотря на то что ни одна такая часть не может закрывать себя при проектировании, возможен случай, когда одна такая часть будет закрывать (полностью или частично) другую часть.

Для учета подобного закрывания введем числовую характеристику видимости – *количественную невидимость* точки. Количественная невидимость – это число лицевых точек, закрывающих ее при заданном проектировании. Точка будет видна тогда и только тогда, когда ее количественная невидимость равна нулю.

Количественная невидимость – это кусочно-постоянная функция, определенная на границе тел. При этом она может изменять свое значение лишь в тех точ-

ках, проекции которых на картинную плоскость попадают на проекции контурных линий.

В результате проекции контурных линий разбивают всю картинную плоскость на области, каждая из которых является проекцией части определенного объекта.

Можно строго доказать, что при проектировании гладких поверхностей возможны два основных типа особых точек, все остальные типы легко могут быть приведены к этим двум.

Первым типом являются *линии складки*, являющиеся регулярными кривыми, проектирующимися на картинную плоскость без особенностей (рис. 8.13а).

Вторым типом являются так называемые *точки сборки* (рис. 8.13б). Они лежат на линиях складки и представляют собой особые точки проектирования контурных линий на картинную плоскость.

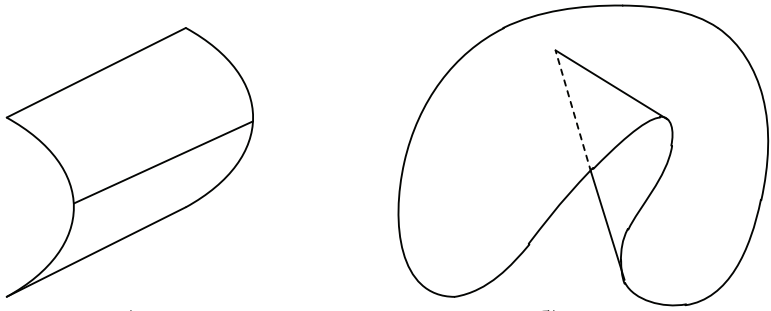

Рис. 8.13 ❖ Особенности проектирования контурных линий

Рассмотрим теперь, как меняется количественная невидимость вдоль самой контурной линии. Можно показать, что это может быть только в двух случаях – при прохождении позади другой контурной линии и в точках сборки.

В первом случае происходит закрывание другим участком поверхности, а во втором – закрывание поверхности самой себя.

Таким образом, мы приходим к следующему алгоритму – на границе проектируемых тел выделяются контурные линии *C*. Каждая из этих линий разбивается на части в тех точках, где она закрывается при проектировании другой контурной линией и в точках сборки.

В результате мы получаем множество линий, вдоль которых количественная невидимость постоянна. В случае если объекты не являются замкнутыми и гладкими, ко множеству линий *C* добавляются граничные линии и линии излома (где происходит разрыв нормали).

Вариант этого алгоритма для работы с полигональными объектами называется алгоритмом Аппеля, и мы его сейчас рассмотрим.

Количественная невидимость точки в данном случае – это число лицевых граней, закрывающих ее при проектировании. Контурная линия – это множество тех ребер, для которых одна из прилегающих граней является лицевой, а другая – нелицевой.

Для невыпуклого многогранника на рис. 8.14 контурной линией будет ломаная *ABCIJDEKLGA*.

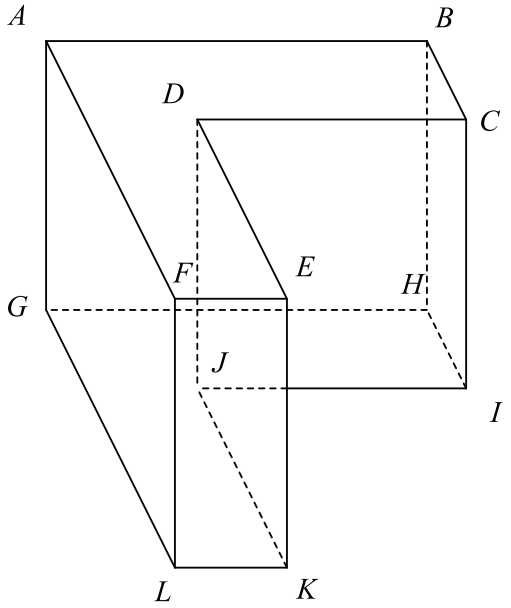

Рис. 8.14 ❖ Пример многогранника для алгоритма Аппеля

Для определения видимости ребер мы сначала берем произвольную вершину и непосредственно (т. е. «в лоб») определяем ее количественную невидимость.

Далее мы идем вдоль ребер, входящих в эту вершину. При движении вдоль очередного ребра выполняется проверка на прохождение позади контурной линии, при этом количественная невидимость изменяется на единицу.

При анализе изменения количественной невидимости вдоль ребра, выходящего из вершины, принадлежащей контурному ребру, необходимо проверить, не закрывается ли это ребро одной из граней, выходящих из этой вершины (это является аналогом точек сборки). Так, на рис. 8.14 грань *DEKJ* закрывает ребро *DJ*.

Обычно если общее число ребер равно n, то число ребер, входящих в контурную линию, равно $O(\sqrt{n})$. Таким образом, алгоритм Аппеля является более эффективным, чем алгоритм Робертса.

Для повышения эффективности алгоритма Аппеля также можно использовать различные пространственные индексы, например регулярное разбиение картинной плоскости.

Удаление невидимых граней. Метод трассировки лучей

В большинстве случаев сейчас приходится сталкиваться не с задачей удаления невидимых ребер, а с задачей удаления невидимых граней (поверхностей) (HSR, Hidden Surface Removal). В этом случае мы выводим не отдельные ребра, а закрашиваем целые грани. Соответственно, нашей задачей является определение видимых частей граней. Обычно задача формулируется следующим образом: для каждого пиксела картинной плоскости найти видимую в этом пикселе (т. е. ближайшую к нему вдоль направления проектирования) грань.

Одним из наиболее простых методов для определения видимости в каждом пикселе является метод *трассировки лучей* (ray casting). В главе 10 мы рассмотрим адаптацию этого метода, который сможет не только определять видимость, но еще и строить изображения с очень высоким качеством (ray tracing).

В этом алгоритме через каждый пиксел картинной плоскости и центр проектирования мы выпустим луч и найдем ближайшее его пересечение с объектами сцены (рис. 8.15). Оно и определяет видимость в данном пикселе.

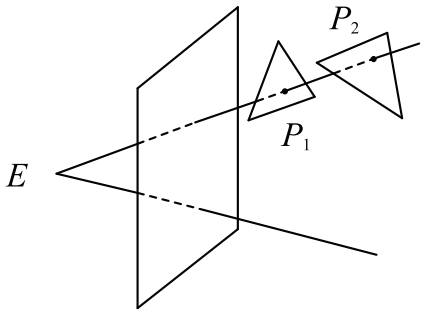

Рис. 8.15 ❖ Метод трассировки лучей

Работу этого алгоритма можно представить следующим образом:

```
for all pixels:
    for all objects:
        compare distance
```

В случае когда у нас всего n объектов и p пикселов на экране, сложность алгоритма составит $O(np)$. В главе 10 мы рассмотрим, как при помощи различных структур можно повысить быстродействие этого алгоритма, доведя его до $O(p \cdot \log n)$.

Метод трассировки лучей может работать не только с полигональными данными, но и с целым рядом других объектов – единственным требованием для его применимости является возможность нахождения пересечения луча с объектами. Однако для задачи реального времени этот метод малоприменим из-за своей невысокой скорости.

Метод буфера глубины (z-буфера)

Еще одним классическим методом удаления невидимых поверхностей является метод *буфера глубины* (также иногда называемый методом z-буфера). Термин z используется потому, что при каноническом проектировании для определения того, какая из двух точек, p_1 или p_2, лежит ближе к центру проектирования, нам достаточно просто сравнить их z-координаты (рис. 8.16). Поэтому z-координату часто называют *глубиной* (depth).

В методе трассировки лучей мы для каждого пиксела выпускали луч и сравнивали между собой пересечения этого луча с объектами.

В методе z-буфера мы сразу для всех пикселов ищем минимальное значение z, перебирая все полигоны. Каждый полигон растеризуется, при растеризации для каждого получающегося фрагмента (пиксела) путем билинейной интерполяции находится соответствующее ему значение z-координаты.

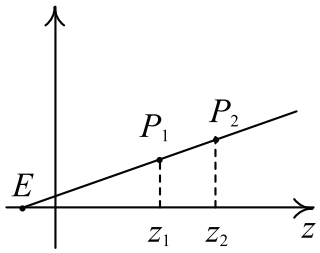

Рис. 8.16 ❖ Сравнение точек на загораживание эквивалентно сравнению их z-координат

Нахождение минимального z выполняется сразу для всех пикселов – для каждого фрагмента его z-координата сравнивается с текущей минимальной z-координатой для соответствующего этому фрагменту пиксела. Фактически данный алгоритм можно представить при помощи следующего кода:

```
for all objects
     for all covered pixels:
         compare z
```

Поскольку мы ищем минимальное значение сразу для всех пикселов, то нам нужно где-то хранить текущее минимальное значение для каждого пиксела. С этой целью вводится вспомогательный буфер (двухмерный массив), в котором для каждого пиксела хранится соответствующее ему значение z-координаты. Сами значения обычно представляются в виде 16/24/32-битовых беззнаковых целых чисел (хотя многие современные GPU поддерживают хранение глубины в виде чисел с плавающей точкой). Сам этот буфер называется *буфером глубины* (или z-буфером).

Перед началом рендеринга в этот буфер для каждого пиксела заносится максимально представимое число (например, для 16-битового буфера глубины таким значением будет `0xFFFF`).

Далее при растеризации каждого полигона глубина для каждого полученного фрагмента сравнивается с соответствующим фрагменту значением в z-буфере. Если у фрагмента значение z будет больше или равно значению в буфере, то этот фрагмент отбрасывается.

В противном случае значение глубины для этого фрагмента записывается в буфер глубины, а соответствующее этому фрагменту значение цвета записывается в буфер цвета.

Обратите внимание, что операция, выполняемая для каждого пиксела, крайне проста и легко реализуется аппаратно. Обработка отдельных фрагментов легко может осуществляться параллельно. Общие затраты метода составляют $O(np)$.

Каждый современный GPU содержит в себе аппаратную поддержку z-буфера и позволяет с большой скоростью определять видимость для большого числа полигонов. Однако поскольку для метода z-буфера требуется вывод всех полигонов (в произвольном порядке), то для рендеринга больших сцен (состоящих из многих миллионов полигонов) такой метод непосредственно не подходит.

В этом случае желательно сперва сократить число обрабатываемых граней, отбросив заведомо не видимые. После этого оставшиеся грани можно подавать на вход аппаратного z-буфера.

Метод иерархического z-буфера

Одним из первых методов, ориентированных на работу с очень большим числом граней, был метод *иерархического z-буфера*. Важным свойством этого метода было использование практически всех видов когерентности.

Ключевым понятием в этом алгоритме является понятие видимости полигона относительно *z*-буфера. Полигон будет называться *видимым*, если при его растеризации хотя бы для одного из его фрагментов глубина будет меньше соответствующего значения в буфере глубины (рис. 8.17). Можно ввести процедуру для проверки видимости полигона, которая не будет модифицировать сам буфер глубины.

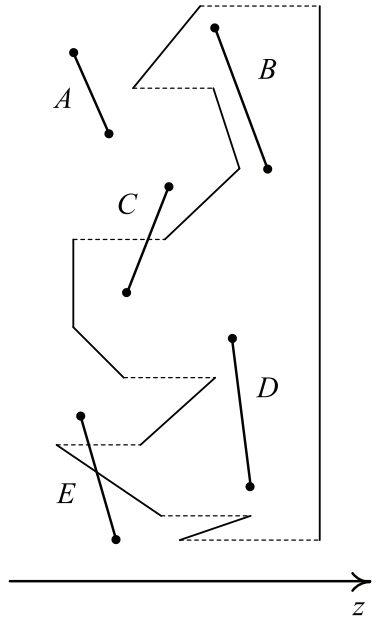

Рис. 8.17 ❖ Видимость полигона относительно заданного буфера глубины

Так, на рис. 8.17 грани *A*, *C* и *E* являются видимыми, остальные грани невидимы. Если полигон не является видимым, то он называется невидимым (относительно заданного буфера глубины). Очевидно, что все невидимые полигоны можно просто отбросить – они не вносят никакого вклада в получаемое изображение.

Теперь пусть у нас есть некоторый *AABB*. Тогда если все его лицевые грани невидимы относительно буфера глубины, то и сам *AABB* мы также назовем невидимым. Очевидно, что если *AABB*, описанный вокруг группы граней, невидим, то и все эти грани также невидимы. Так, на рис. 8.18 *AABB B* и *C* являются невидимыми.

Тем самым понятие видимости относительно заданного буфера глубины позволяет легко отбрасывать целые группы заведомо невидимых граней. Чтобы более эффективно использовать это свойство, давайте организуем все полигоны сцены в восьмеричное дерево. С каждым узлом этого дерева мы свяжем *AABB*, который и будем проверять на видимость.

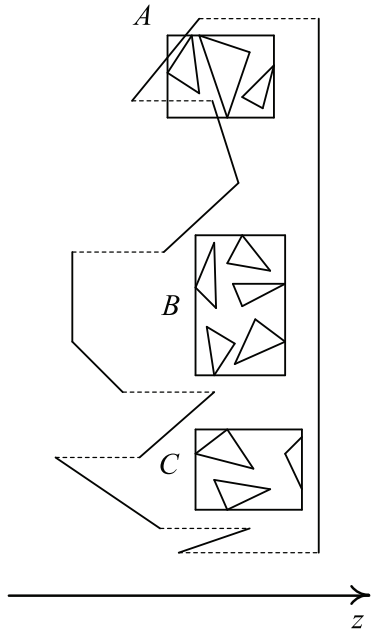

Рис. 8.18 ❖ Видимые и невидимые *AABB*

Тогда процедура вывода полигонов начинается с корня дерева и носит рекурсивный характер. Для вывода листа дерева мы просто выводим все содержащиеся в нем полигоны, которые еще не были выведены, используя стандартный буфер глубины. Обратите внимание, что поскольку одна и та же грань может попасть сразу в несколько листьев дерева, то нам необходимо отслеживать уже выведенные грани. Для этого достаточно для каждой грани просто хранить номер кадра, когда грань последний раз была выведена.

Если узел дерева не является листом, то мы по очереди проверяем все восемь его дочерних узлов, начиная с самого ближнего и заканчивая самым дальним, на видимость относительно буфера глубины. Если *AABB* дочернего узла невидим, то этот узел может быть сразу же отброшен. В противном случае мы рекурсивно обрабатываем его таким же образом.

В результате мы используем когерентность в пространстве объектов. Очевидно, что чем ранее видимые полигоны будут выведены, тем больше полигонов удастся сразу же отбросить. Поэтому также используется когерентность по времени – перед рекурсивным обходом дерева мы сначала выводим (и помечаем их как уже выведенные) все грани, которые были видны на предыдущем кадре.

Таким образом, мы используем когерентность по времени – скорее всего, эти грани также будут видны, и мы сможем быстрее отбросить заведомо невидимые грани.

Данный алгоритм тратит очень много времени, проверяя видимость отдельных полигонов. Поэтому следующая оптимизация направлена на удешевление этой проверки. Для этого вместо одного буфера глубины мы построим целый набор буферов глубины разных размеров, которые мы и будем использовать для проверки на видимость.

Вначале мы разбиваем весь исходный буфер глубины на группы 2×2 значения и из каждой такой группы выбираем максимальное значение. В результате мы получаем новый буфер глубины, ширина и высота которого будут вдвое меньше, чем у исходного буфера.

Далее мы снова разбиваем уже получившийся на прошлом шаге буфер на группы 2×2 значения и снова выбираем из каждой группы максимальное значение. В результате мы получаем буфер вдвое меньшего размера, чем предыдущий.

Мы повторяем эту процедуру до тех пор, пока не придем к буферу, состоящему всего из одного значения. В результате мы получаем набор буферов, где каждый следующий буфер вдвое (по ширине и высоте) меньше предыдущего. Последний буфер будет содержать всего одно значение – максимальное значение глубины для всего исходного буфера (рис. 8.19).

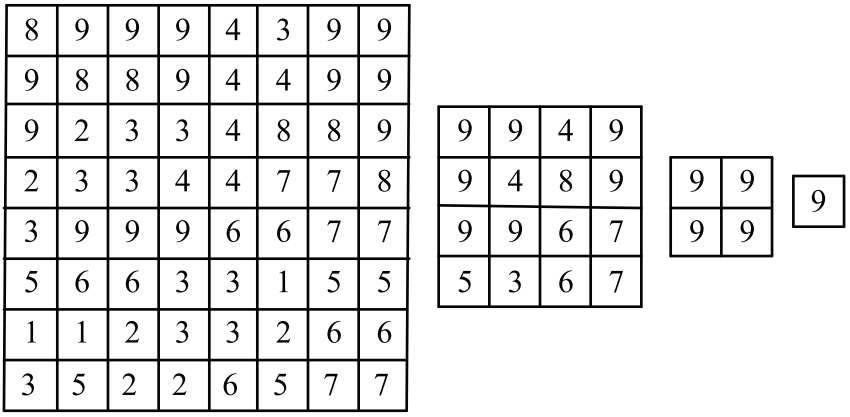

Рис. 8.19 ❖ Пирамида буферов глубины

Весь полученный набор буферов называется *z*-пирамидой. В основании этой пирамиды лежит исходный буфер глубины, а вершиной является последний буфер, состоящий всего из одного значения.

Тогда проверку полигона на видимость мы начнем с вершины этой пирамиды – там, где эта проверка будет дешевле всего. Если *z*-координаты всех вершин полигона больше или равны значениям в вершине пирамиды, то весь соответствующий полигон невидим.

В противном случае мы проверяем полигон на буфере глубины размером 2×2. Для этого выполняется грубая растеризация полигона и проверяется видимость для получившихся фрагментов. Если нам опять не удалось установить его невидимость, то мы переходим к следующему уровню пирамиды и т. д., пока либо не установим невидимость полигона, либо не дойдем до основания пирамиды.

Использование *z*-пирамиды позволяет применять также когерентность в пространстве картинной плоскости и увеличить быстродействие алгоритма, так как чем выше уровень пирамиды, тем дешевле проверка.

Понятно, что изменение *z*-пирамиды является довольно дорогой операцией, поскольку нужно провести изменение вверх по пирамиде, иногда до самой ее вершины. Поэтому обычно эти изменения не сразу проводятся через всю пирамиду, а лишь после вывода целой группы граней.

На данный момент нет полной аппаратной реализации данного метода, но большинство современных GPU использует для ускорения работы *z*-пирамиду.

Алгоритмы, основанные на упорядочивании. Алгоритм художника

Поскольку в основе каждого алгоритма определения видимости лежит сортировка, то можно попробовать явно выполнить сортировку граней, а потом просто вывести отсортированные в нужном порядке грани.

Если у нас есть две грани *A* и *B* и грань *A* может закрывать грань *B* при выводе, то мы сначала должны вывести грань *B* и только потом грань *A*. В случае если нам удастся таким образом упорядочить грани, что это условие окажется выполненным для любых двух граней, то мы получим корректное изображение.

Обратите внимание, что далеко не всегда можно упорядочить грани таким образом. Так, на рис. 8.20 приведены три грани, для которых вывод в любом порядке даст неверное изображение.

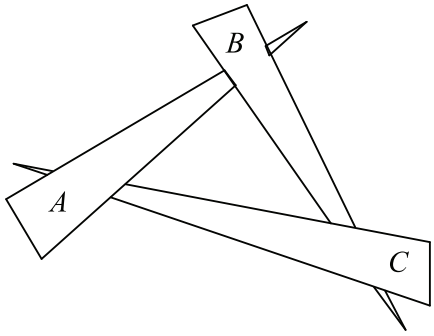

Рис. 8.20 ❖ Неупорядочиваемые грани

Две выпуклые грани можно упорядочить между собой всегда, но для невыпуклых граней это уже неверно (рис. 8.21).

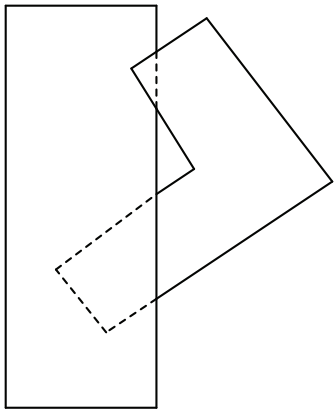

Рис. 8.21 ❖ Две неупорядочиваемые невыпуклые грани

Простейшим методом, основанным на таком подходе, является *алгоритм художника* (painters algorithm). Свое название он получил, поскольку использует тот же подход, что и художник, – сначала выводятся более дальние грани, затем (поверх них) более близкие. Фактически алгоритм художника сортирует грани по z-координате, а потом выводит их в полученном порядке.

Следует иметь в виду, что грань обычно соответствуют не одному значению z, а целому диапазону. Соответственно, сравнение двух граней по z-координате не всегда дает единственный (и верный) ответ.

Обычно на грани выбирается одна точка – первая, средняя, ближайшая к камере, – и грани сортируются по z-координате этой точки. Однако на рис. 8.22 приведен пример, когда подобная сортировка дает неверный результат.

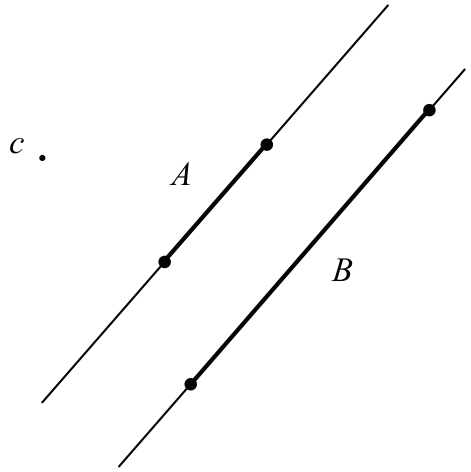

Рис. 8.22 ❖ Неверное сравнение граней по z-координате

Если при использовании алгоритма художника мы хотим получать корректные результаты, то нужен точный способ определения того, какая из двух граней может закрывать другую при заданном проектировании. Кроме того, как показано на рис. 8.20, в некоторых случаях нам придется разрезать грани на части, для того чтобы требуемое упорядочивание могло быть построено.

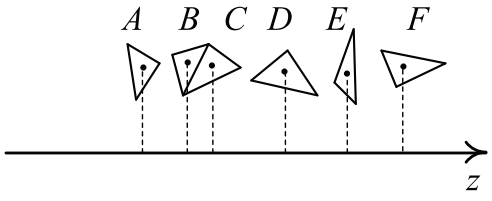

Рис. 8.23 ❖ Пример упорядочивания граней

Тем не менее для многих достаточно простых случаев (например, построения графика функции двух переменных) подобный подход отлично работает.

Использование BSP-деревьев для определения видимости

Существует очень простой и точный критерий для определения того, какая из двух граней, *A* или *B*, должна быть выведена раньше. Пусть у нас есть некоторая плоскость, такая что эти грани лежат по разные стороны от этой плоскости (рис. 8.24).

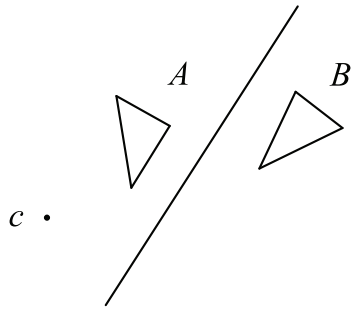

Рис. 8.24 ❖ Определение порядка вывода граней при помощи разбивающей плоскости

Тогда грань, не лежащая в том же полупространстве, что и центр проектирования *c* (*B*), никогда не сможет закрыть собой грань, лежащую в том же полупространстве, что и центр проектирования (*A*). Это же утверждение остается в силе, если вместо двух граней мы рассмотрим два объекта или две группы объектов, разделенные плоскостью.

Тем самым мы приходим к довольно простому алгоритму. Выбираем некоторую плоскость и разбиваем все множество граней на две группы, порядок которых строго определен положением центра проектирования относительно разбивающей плоскости.

Далее для каждой из этих двух групп проводим свою разбивающую плоскость и разбиваем каждую из них на две группы (каждая группа разбивается независимо от другой). Этот процесс разбиения групп мы проводим до тех пор, пока порядок вывода граней внутри каждой из получившихся групп не станет ясным. Простейшим случаем этого являются группы, состоящие из не более чем одной грани. В результате этой процедуры мы можем построить верное разбиение (рис. 8.25).

Так, на рис. 8.25 приведен набор граней *A*, *B*, *C*, *D*, *E*, *F* и *G* вместе с разбивающими их плоскостями, в результате чего мы получаем набор групп по одной грани, которые легко упорядочиваются.

Сначала первая плоскость разбивает все множество граней на подмножества (*A*, *B*, *C*) и (*D*, *E*, *F*, *G*). При этом сперва должны быть выведены грани *A*, *B* и *C* и только потом грани *D*, *E*, *F* и *G*. Далее следующая плоскость разбивает множество граней *A*, *B* и *C* на два множества – *A* и (*B*, *C*). При этом вперед выводится грань *A* и только потом грани *B* и *C*. Далее для граней *B* и *C* строится разбивающая их плоскость и задается их порядок вывода – сперва грань *B*, и после нее грань *C*.

Описанный процесс построения очень напоминает процесс быстрой сортировки Хоара. С другой стороны, результат этого процесса можно представить в виде бинарного дерева. При этом внутренние узлы дерева соответствуют разбивающим плоскостям. Соответствующее рис. 8.25 дерево представлено на рис. 8.26.

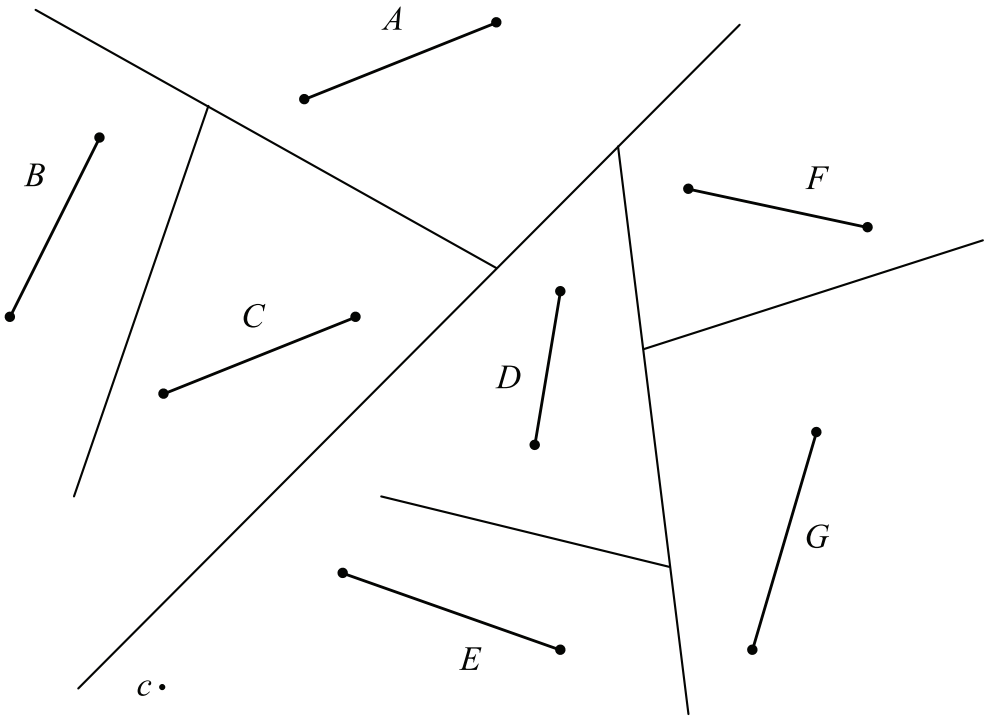

Рис. 8.25 ❖ Пример построения разбиения множества граней на отдельные группы

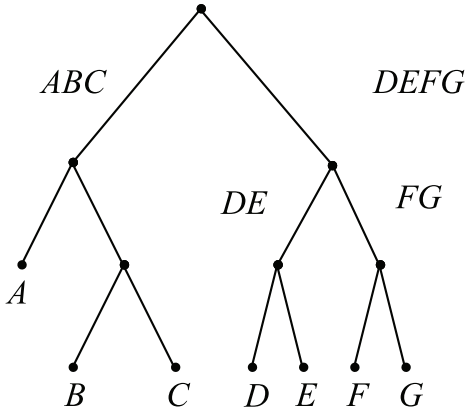

Рис. 8.26 ❖ Бинарное дерево разбиения для набора граней с рис. 8.25

Фактически мы получили уже знакомое нам BSP-дерево. Оказывается, его можно также с успехом использовать для построения правильного упорядочивания граней.

Обратите внимание, что сам процесс построения дерева никак не привязан к центру проектирования. Можно один раз построить дерево и потом использо-

вать его для самых различных положений центра проектирования – центр проектирования только влияет на порядок обхода дерева, но не само дерево. Кроме того, при помощи BSP-дерева мы можем сортировать грани как в порядке приближения к центру (back-to-front), так и в порядке удаления (front-to-back).

Важным свойством BSP-дерева является то, что оно разбивает все пространство на набор выпуклых многогранников.

Именно BSP-деревья лежат в основе таких легендарных игр, как Doom и Quake, – определение видимости в этих играх (и не только оно) было основано на использовании BSP-деревьев.

Метод порталов

Еще одним часто используемым в различных играх алгоритмом удаления невидимых граней является метод порталов. Обычно этот метод с успехом применяется для определения видимости в различных архитектурных сценах.

В этом методе мы добавляем к сцене дополнительные многоугольники, называемые *порталами*. Их задача – разбить всю сцену на набор достаточно простых частей (рис. 8.27). В первых реализациях этого метода требовалось разбиение на выпуклые многогранники (поскольку видимость граней выпуклого многогранника устанавливается тривиально), однако сейчас это требование уже не используется (для точного определения видимости внутри части применяется традиционный метод буфера глубины).

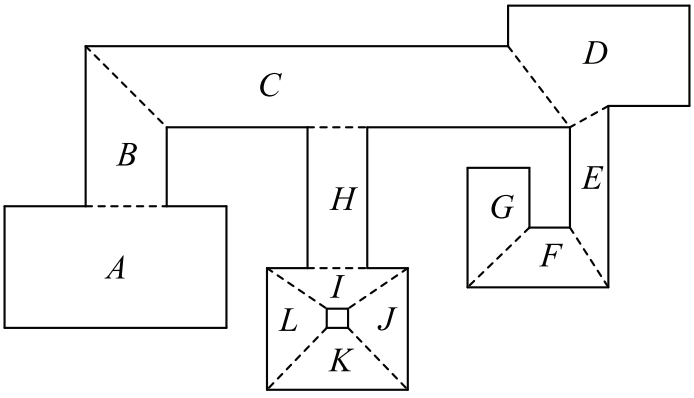

Рис. 8.27 ❖ Пример сцены для метода порталов

В результате вся сцена может быть представлена в виде графа (рис. 8.28). Узлы графа соответствуют частям (комнатам), а его ребра – порталам.

Рассмотрим теперь, каким образом осуществляется рендеринг сцены, заданной в виде такого графа. При этом нам необходимо задать камеру – виртуального наблюдателя. У камеры есть, помимо положения *c*, еще и область видимости (viewing frustum) – это *пирамида видимости*, вершиной является точка *c* (рис. 8.29). Кроме того, нам нужно определить, в какой именно «комнате» находится камера.

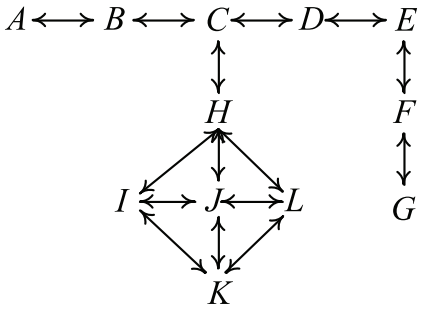

Рис. 8.28 ❖ Граф для сцены с рис. 8.27

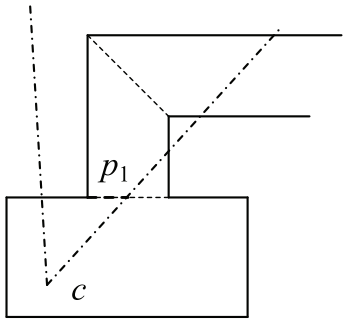

Рис. 8.29 ❖ Область видимости

Мы выводим все грани из текущей комнаты (используя, например, метод буфера глубины для точного определения видимости для этих граней). Далее для каждого портала *p*, хотя бы частично попавшего в пирамиду видимости, мы сначала обрезаем этот портал по текущей пирамиде видимости. После этого мы по центру проектирования *c* и обрезанному порталу строим новую пирамиду видимости (рис. 8.30).

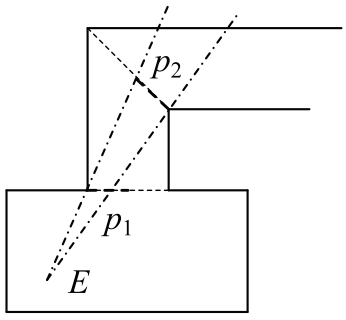

Рис. 8.30 ❖ Пирамида видимости по усеченному порталу

В качестве следующего шага мы выводим все грани из той комнаты, в которую ведет этот портал (отбрасывая при этом все многоугольники, не попавшие в эту

пирамиду видимости). Далее для каждого портала, полностью или частично попавшего в новую пирамиду видимости, мы обрезаем его по пирамиде видимости и строим новую пирамиду видимости.

После этого процесс продолжается дальше. Его можно представить при помощи следующего фрагмента кода:

```
def renderScene ( viewFrustum, room ):
  for poly in room.polygons:
    if not poly.isFrontFacing:
      continue

    clippedPoly = viewFrustum.clip ( poly )
    if not clippedPoly.isEmpty ():
      draw ( clippedPoly )

  for portals in room.portals:
    if not poly.isFrontFacing:
      continue

    clippedPortal = viewFrustum.clip ( portal )
    if not clippedPortal.isEmpty ()
      newFrustum = buildFrustum ( viewFrustum.origin,
                                  clippedPortal )
      newFrustum.addClipPlane ( portal.plane )
      renderScene ( newFrustum,
                    portal.adjacentRoom ( room ) )
```

Большим плюсом метода порталов является то, что он позволяет обрабатывать только те части сцены, которые могут быть видны. Все остальные части отбрасываются, т. е. быстродействие алгоритма зависит не от общего числа граней, а от числа видимых граней.

Метод порталов очень хорошо подходит для рендеринга геометрии, представляющей собой внутренность помещений, лабиринтов и т. п. Однако он практически не применим для рендеринга обычных ландшафтных сцен, так как их очень тяжело эффективно разбить на отдельные, соединенные порталами части.

Множества потенциально видимых граней (PVS). Расчет PVS при помощи порталов

Если мы имеем дело с неизменяющейся архитектурной геометрией (что очень часто встречается в различных играх и других приложениях), то можно заранее, на этапе подготовки геометрии, определить видимость. Именно этот подход был активно использован в серии игр Quake от idSoftware.

В этих играх вначале по граням строилось BSP-дерево. Оно разбивает всю сцену на набор выпуклых многогранников. Границы этих многогранников состоят как из исходных граней, так и из новых граней (рис. 8.31).

При этом новым граням (например, *IF*) не соответствуют никакие видимые многоугольники – они являются «окнами» в соседние многогранники. Тем самым эти новые грани являются просто порталами.

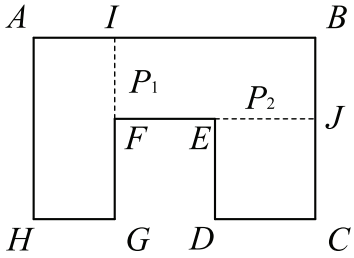

Рис. 8.31 ❖ Пример разбиения сцены на выпуклые многогранники

Пусть у нас есть сцена, разбитая множеством порталов на набор выпуклых «комнат». Тогда множеством потенциально видимых граней (PVS, Potentially Visible Set) для заданной комнаты называется множество всех комнат, которые можно увидеть из нее для всевозможных положений и ориентаций наблюдателя (камеры).

Ключевым понятием для построения PVS при помощи порталов является понятие антитени (anti-penumbrae). Рассмотрим, что это такое. Пусть у нас два портала – p_1 и p_2 (рис. 8.32).

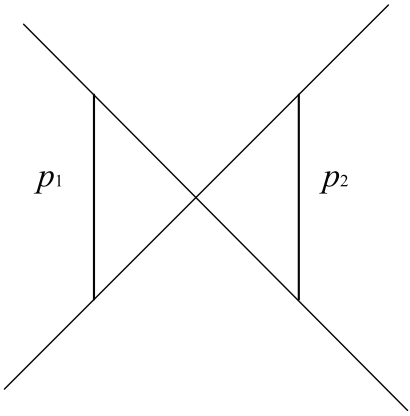

Рис. 8.32 ❖ Антитень для двух порталов

Рассмотрим множество плоскостей, проходящих через вершину одного портала и ребро другого портала таким образом, что эти порталы лежат по разные стороны такой плоскости. Тогда антитень – это пересечение всех полупространств, задаваемых этими плоскостями, которые не содержат портал p_2.

Пусть у нас есть три комнаты r_1, r_2 и r_3, соединенные порталами p_1 и p_2 (рис. 8.33). Представим себе, что вся комната r_1 залита светом. Тогда и вся комната r_2, соединенная с r_1 при помощи портала p_1, также будет залита светом. Антитень, построенная по порталам p_1 и p_2,определяет часть комнаты r_3, которая будет также залита светом.

Если соответствующая антитень не пуста, то комната r_3 будет видна из r_1, и мы добавляем в PVS r_1 комнату r_3. Заметим, что r_2 всегда входит в PVS r_1.

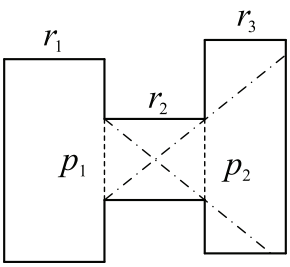

Рис. 8.33 ❖ Тестовая сцена с двумя порталами

Если антитень, построенная по порталам p_1 и p_2, не пуста, то мы проверяем все порталы из r_3 на пересечение с этой антитенью. Если портал p_3, ведущий в какую-то комнату r_4, хотя бы частично попадает в нашу антитень, то мы добавляем r_4 в PVS r_1. После этого мы обрезаем портал p_3 по антитени и строим новую антитень по порталу p_1 и обрезанному порталу p_3.

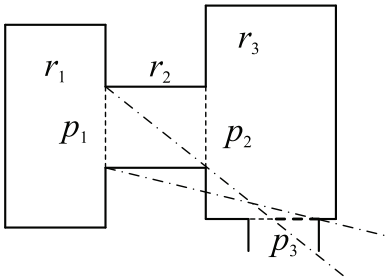

Рис. 8.34 ❖ Построение антитени по порталу p_1 и усеченному порталу p_3

Далее мы проверяем все порталы r_4 на пересечение с новой антитенью и т. д. Таким образом, мы строим список всех комнат, которые могут быть видны из r_1. Эту же операцию мы проводим и для всех остальных комнат. В результате получаем для каждой комнаты список всех видимых из нее комнат. Само построение PVS очень ресурсоемко и занимает много времени, но это делается всего один раз и далее используется во время рендеринга для отбрасывания заведомо невидимых комнат.

Подобный список можно представить в виде битового массива, где бит 1 означает, что соответствующая комната видна. Понятно, что подобный массив будет содержать огромное количество нулей, поэтому он может быть очень эффективно сжат при помощи RLE-кодирования.

Глава 9

Отражение и преломление света. Модели освещения

Освещение играет крайне важную роль в получении правдоподобно выглядящих изображений. При этом чем больше тонкостей освещения мы в состоянии передать, тем естественнее и правдоподобнее получается готовое изображение.

Ключевым моментом в реализации освещения является понимание того, как именно происходит взаимодействие света с объектами. В этой главе мы достаточно подробно опишем этот процесс и дадим большое количество различных моделей освещения. При этом нам придется столкнуться с целым рядом элементов из курса физики, которые необходимы для получения реалистических моделей освещения.

Немного физики

Будем далее считать, что некоторая плоскость является границей между двумя различными (но однородными) материалами. Возьмем произвольную точку P на этой плоскости. Будем считать, что в этой точке нам известны единичный вектор внешней нормали n, единичный вектор направления на точечный источник света l и единичный вектор на наблюдателя v (рис. 9.1).

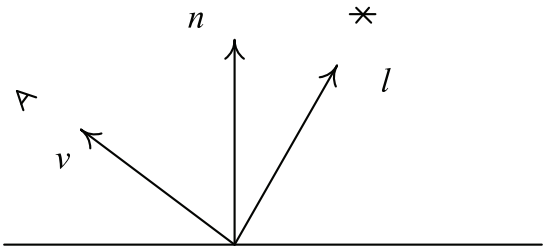

Рис. 9.1 ❖ Базовая сцена для расчета освещенности

Пусть свет падает вдоль вектора l в точку P. Часть этого света *отразится* и/или рассеется. Если материал по ту сторону плоскости является прозрачным (напри-

мер, это стекло), то также произойдет *преломление* – часть света пройдет сквозь границу раздела материалов.

Известно, что в однородной среде свет распространяется прямолинейно и с постоянной скоростью. При этом эта скорость не может превосходить скорость света в вакууме *c*. Отношение скорости света в вакууме к скорости распространения света в материале называется *коэффициентом преломления* (IOR, Index Of Refraction) этого материала и обозначается как η.

Таблица 9.1. Коэффициенты преломления некоторых распространенных материалов

Материал	Коэффициент преломления
Вода	1.33
Стекло	1.5–1.8
Алмаз	2.4
Воздух	1.0003
Золото	0.47
Кварц	1.544
Лед	1.309
Сталь	2.5

Обратите внимание, что коэффициент преломления всегда больше или равен единице. Коэффициент преломления также зависит от длины волны λ. Именно этим объясняется то, что белый свет раскладывается стеклянной призмой на радугу. Далее мы будем игнорировать эту зависимость.

Мы будем считать, что среда, из которой идет луч света (куда направлен вектор нормали), имеет коэффициент преломления η_i, а вторая среда – коэффициент η_t.

Простейшим случаем отражения света является идеальное *зеркальное отражение*. Оно определяется следующим образом: *угол падения* θ_i *равен углу отражения* θ_r, и отраженный вектор *r* лежит в той же плоскости, что вектор падения *i* и вектор нормали *n* (рис. 9.2).

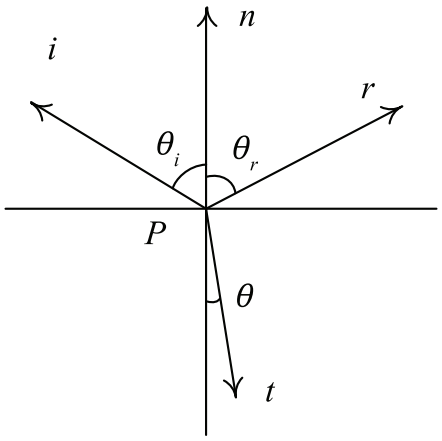

Рис. 9.2 ❖ Идеальное отражение и преломление света

Так как векторы i, r и n лежат в одной плоскости, то они линейно зависимы и один из них может быть представлен в виде суммы двух других с некоторыми коэффициентами:

$$r = \alpha i + \beta n.$$

Так как все векторы единичные, то можно выразить косинусы углов через их скалярные произведения: $\cos\theta_i = (-i, n)$ и $\cos\theta_r = (r, n)$. Из равенства углов следует равенство скалярных произведений. Таким образом, мы получаем первое уравнение для α и β. Второе уравнение можно получить из условия единичности отраженного вектора r. Решив эту систему уравнений, мы получаем такое уравнение для единичного отраженного вектора:

$$r = i - 2(i, n)n.$$

Следующим случаем является *идеальное преломление*. Оно описывается *законом Снеллиуса*. Согласно этому закону, вектор направления отраженного луча t лежит в той же плоскости, что векторы i и n. Кроме того, угол падения θ_i связан с *углом преломления* θ_t следующим образом:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t.$$

Как и ранее, представляем вектор t через векторы i и n. Далее можно возвести уравнение для углов в квадрат и выразить квадрат синуса через квадрат косинуса. Потом мы заменяем косинусы на скалярные произведения и получаем квадратичное уравнение. Еще одним уравнением будет условие единичности t. Мы не будем приводить здесь все выкладки, а сразу выпишем готовый результат:

$$t = \eta \cdot i + \left(\eta\cos\theta_i - \sqrt{1 + \eta^2(\cos^2\theta_i - 1)}\right)\cdot n.$$

Здесь через η обозначен *относительный коэффициент преломления* (одного материала относительно другого):

$$\eta = \frac{\eta_i}{\eta_t}.$$

Обратите внимание, что выражение под корнем $(1 + \eta^2(\cos^2\theta_i - 1))$ может быть отрицательным. Это соответствует случаю так называемого *полного внутреннего отражения*, когда вообще не происходит преломления света – он полностью отражается.

Для описания доли энергии отраженного и преломленного света используется *коэффициент Френеля* F, задающий, какая доля от падающего света отразится от границы раздела сред. Соответственно, в случае преломления доля преломленного света будет $1 - F$.

Для диэлектриков коэффициент Френеля очень мал и обычно лежит в диапазоне [0.02, 0.05]. Есть довольно простая формула, позволяющая связать коэффициент Френеля (для случая перпендикулярного падения, т. е. $\theta = 0$) с коэффициентом преломления η:

$$F(0) = \left(\frac{\eta - 1}{\eta + 1}\right)^2.$$

Коэффициент Френеля зависит как от угла падения θ, так и от длины волны света λ. Более того, формулы для коэффициента Френеля для проводников и диэлектриков различаются. Поэтому обычно используют приближенную (и более простую) формулу для его расчета. Очень часто для этого используют *формулу Шлика* (Schlick), приводимую ниже:

$$F(\theta) = F(0) + (1 - F(0))(1 - \cos\theta)^5.$$

Чаще всего коэффициент Френеля считается для трех базовых цветов (RGB), соответственно, мы получаем также трехмерный вектор, зависящий от угла падения θ и отражения для случая нормального падения $F(0)$ (который обычно просто задается цветом).

Модель Ламберта (идеальное диффузное освещение)

Как легко можно заметить, большинство материалов, которые мы наблюдаем, отражает свет совсем иначе – падающий свет рассеивается по всей верхней полусфере (но не обязательно равномерно).

Построение аккуратной модели подобного рассеивания довольно сложно, поэтому мы начнем с описания ряда приближенных (упрощенных) моделей. Простейшей из подобных моделей является модель Ламберта.

Созданная более века назад диффузная модель считает, что падающий в точку P свет равномерно рассеивается во всех направлениях верхней полусферы (рис. 9.3).

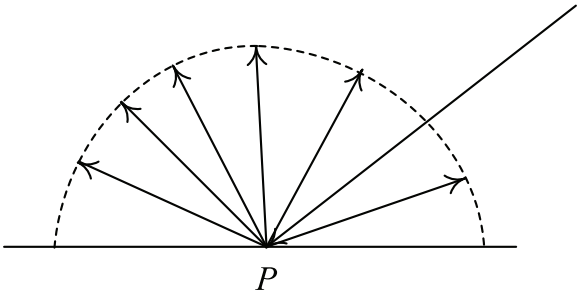

Рис. 9.3 ❖ Идеальное диффузное рассеивание

Тогда видимая в точке P освещенность вообще не будет зависеть от направления на наблюдателя/камеру. Единственное, от чего будет зависеть то, насколько яркой мы увидим точку P, – это падающая в эту точку световая мощность на единицу площади.

Давайте рассмотрим область S на плоскости, освещаемую падающим на нее лучом света. Через $S_{регр}$ мы обозначим площадь поперечного сечения луча, освещающего S. Из курса школьной геометрии мы знаем, что эти площади связаны следующим соотношением:

$$S_{регр} = S \cdot \cos\theta.$$

Отсюда мы сразу получаем, что мощность на единицу площади будет прямо пропорциональна косинусу угла падения θ (рис. 9.4).

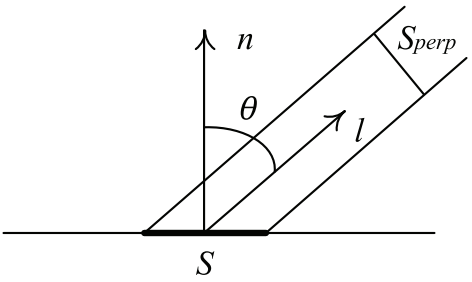

Рис. 9.4 ❖ Удельная мощность падающего света

Давайте через I_l обозначим RGB-вектор, задающий цвет и мощность источника света, через C – цвет поверхности в точке P (тоже как RGB-вектор) и через I видимую освещенность в этой точке. Тогда мы можем записать следующее уравнение для идеальной диффузной освещенности:

$$I = I_l \cdot C \cdot \cos\theta.$$

Считая векторы l и n единичными, мы можем переписать это уравнение через скалярное произведение:

$$I = I_l \cdot C \cdot \max(0, (n, l)).$$

В этой формуле используется max, для того чтобы не получалось отрицательных цветов, когда источник светит снизу (а такое может быть в реальных сценах).

К сожалению, использование этой формулы приводит к тому, что неосвещенные участки оказываются полностью черными. Чтобы этого избежать, часто в эту формулу добавляют еще один член – *фоновое* (ambient) освещение: некоторый свет, падающий равномерно со всех сторон и ни от чего зависящий. Тогда итоговая освещенность может быть представлена в виде суммы двух членов с весами k_a и k_d соответственно:

$$I = k_a I_a C + I_l C \cdot \max(0, (n, l)).$$

Обратите внимание, что это приблизительная формула, не претендующая на физическую корректность.

Модель Фонга

Одним из наиболее заметных недостатков модели Ламберта является полное отсутствие бликов на гладких поверхностях. Есть различные способы учесть блики, большинство из них просто добавляет к ранее рассмотренной формуле еще один член, отвечающий за блики (со своим весом k_s).

Одной из самых первых моделей освещения, поддерживающих блики, была модель Фонга, задаваемая следующей формулой:

$$I = k_a I_a C + I_l \cdot C \cdot \max(0, (n, l)) + k_s C_s I_l \cdot \max(0, (r, l))^p.$$

Здесь через C_s обозначен цвет блика, через r – вектор v, отраженный относительно вектора нормали n, и через p – некоторый коэффициент, задающий сте-

пень неровности поверхности (чем он больше, тем более гладкой является поверхность) (рис. 9.5).

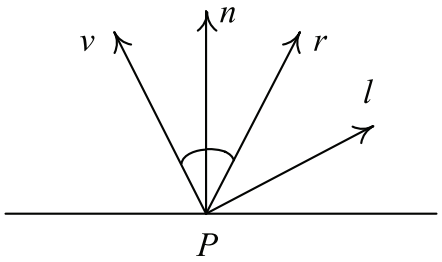

Рис. 9.5 ❖ Векторы для модели Фонга

Отраженный вектор r задается следующей формулой:

$$r = 2n \cdot (n, l) - l.$$

Цвет блика C_s для диэлектриков является просто белым цветом, а для металлов он совпадает с цветом поверхности C.

На рис. 9.6 приведены изображения объекта, построенные при помощи модели освещения Фонга с разными значениями параметра p.

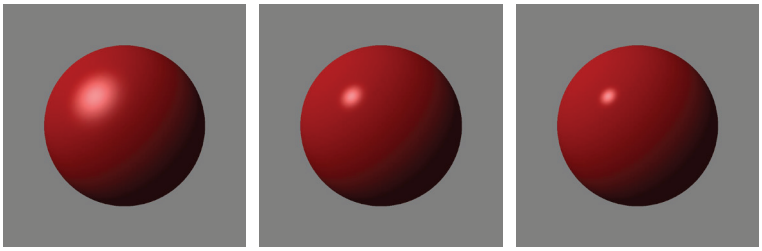

Рис. 9.6 ❖ Модель освещения Фонга
(степень равна 10, 50 и 90 соответственно)

Модель Блинна–Фонга

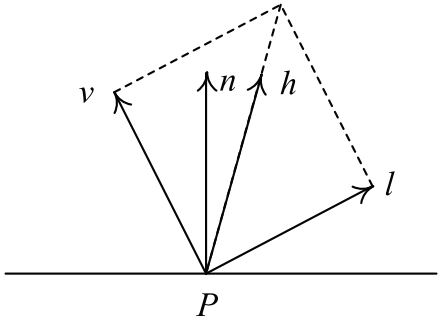

Рис. 9.7 ❖ Векторы для модели Блинна–Фонга

Еще одной моделью освещения, поддерживающей блики, является модель Блинна–Фонга, задаваемая следующим уравнением:

$$I = k_a I_a C + I_l \cdot C \cdot \max(0, (n, l)) + k_s C_s I_l \cdot \max(0, (h, n))^p.$$

Здесь через h обозначен бисектор векторов l и v (т. е. их нормированная сумма):

$$h = \frac{l + v}{\|l + v\|}.$$

Обратите внимание, что модели Фонга и Блинна–Фонга не являются эквивалентными и одна из них не может быть приведена к другой путем простого преобразования параметра p.

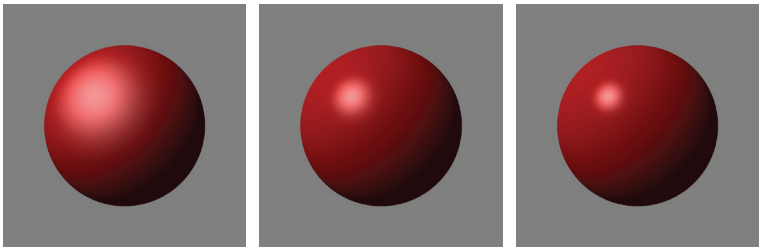

Рис. 9.8 ❖ Модель освещения Блинна–Фонга для различных значений параметра p (10, 50 и 90 соответственно)

Изотропная модель Уорда

Еще одной моделью, поддерживающей блики, является изотропная модель Уорда (Ward). Она задается следующим уравнением:

$$I = k_a I_a C + I_l C \max(0, (n, l)) + k_s C_s I_l \exp(-k \cdot \tan^2\theta).$$

В этом уравнении коэффициент k отвечает за гладкость поверхности, а θ – угол между векторами n и h. Это уравнение можно переписать, выразив тангенс угла через скалярное произведение:

$$I = k_a I_a C + I_l C \max(0, (n,l)) + k_s C_s I_l \exp\left(-k \cdot \frac{1-(n,h)^2}{(n,h)^2}\right).$$

Модель Миннаэрта

Для моделирования освещения Луны Миннаэртом (Minnaert) была предложена следующая модель освещенности (обратите внимание, эта модель не поддерживает блики):

$$I = k_a I_a C + k_d C I_l ((n, l)^{1+k} (1 - (n, v))^{1-k}.$$

На рис. 9.9 приведено сравнение модели Миннаэрта для различных значений k с моделью Ламберта.

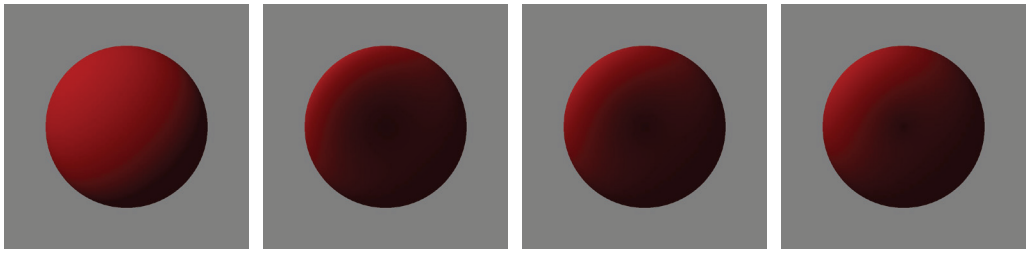

Рис. 9.9 ❖ Сравнение модели Ламберта (слева) с моделью Миннаэрта (k = 0.3, 0.5, 0.7)

Модель Ломмеля–Зилиджера

Кроме модели Миннаэрта, из астрономии также пришла еще одна модель освещения – Ломмеля–Зилиджера (Lommel–Seeliger). Эта модель использовалась для моделирования освещения астероидов и задается следующей формулой:

$$I = k_a I_a C + k_d I_l C \frac{\max((n,l),0)}{\max((n,l),0)+\max((n,v),0)}.$$

На рис. 9.10 приведено сравнение освещения модели Ламберта и Ломмеля–Зилиджера.

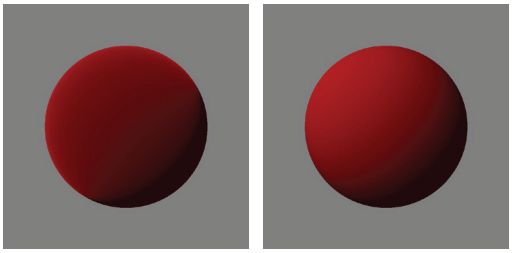

Рис. 9.10 ❖ Сравнение модели освещения Ломмеля–Зилиджера и Ламберта

Модель Страусса

В ранее рассмотренных моделях освещения многие параметры не являются интуитивно-понятными и зачастую не имеют прямого физического соответствия.

Предложенная в 1990 году модель Страусса использует всего 4 параметра, каждый из которых изменяется от нуля до единицы. Этими параметрами являются:

- C – цвета материала в виде RGB-вектора;
- s – гладкость материала. Ноль соответствует очень неровной поверхности, единица – идеальному зеркалу;
- m – «металличность» поверхности. Единица соответствует металлу;
- t – прозрачность поверхности.

В модели Страусса освещенность разбивается на диффузную D и бликовую S составляющие. Данная модель освещения описывается следующими формулами:

$I = I_l(D + S)$;

$D = C \cdot \max(0, (n, l))(1 - ms)(1 - s^2)(1 - t)$;

$S = C_s R(-v,h)^{\frac{3}{1-s}}$;

$R = \min(1, 2 + j(r + k))$;

$r = (1 - t) - (1 - s^3)(1 - t)$;

$j = fresnel((n, l)) \cdot shadow((n, l)) \cdot shadow((n, v))$;

$C_s = C_{white} + m(1 - fresnel((n, l))(C - C_{white})$.

Функции *fresnel* и *shadow* вводятся следующим образом:

$$fresnel(x) = \frac{\frac{1}{(x-k_f)^2} - \frac{1}{k_f^2}}{\frac{1}{(1-k_f)^2} - \frac{1}{k_f^2}};$$

$$shadow(x) = \frac{\frac{1}{(1-k_s)^2} - \frac{1}{(x-k_s)^2}}{\frac{1}{(1-k_s)^2} - \frac{1}{k_s^2}}.$$

Обычно используются такие значения коэффициентов:

$k = 0.1$;
$k_f = 1.12$;
$k_s = 1.01$.

Значением C_{white} является белый цвет, т. е. вектор $(1 \quad 1 \quad 1)^T$.

Простейшая анизотропная модель

Все рассмотренные ранее модели освещения являются *изотропными*. Это значит, что если мы повернем поверхность вокруг прямой, параллельной нормали и проходящей через точку P (рис. 9.11), то видимый в этой точке цвет не изменится.

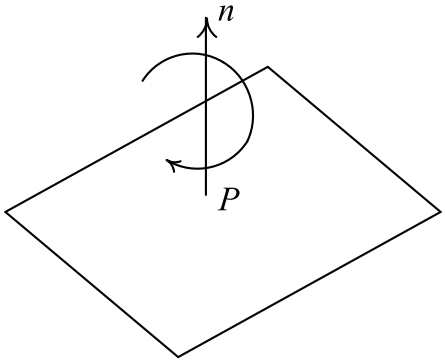

Рис. 9.11 ❖ Понятие изотропности

В то же время есть целый ряд материалов, для которых это свойство не выполнено. К ним относятся полированный металл, поверхность компакт-диска, воло-

сы. Все подобные материалы называются *анизотропными*, и для них нужны свои модели освещенности.

Для того чтобы в такой модели можно было отслеживать поворот, необходимо, кроме вектора нормали n (который при повороте не изменяется), ввести некоторый дополнительный вектор, который будет «привязан» к поверхности и поворачиваться вместе с ней.

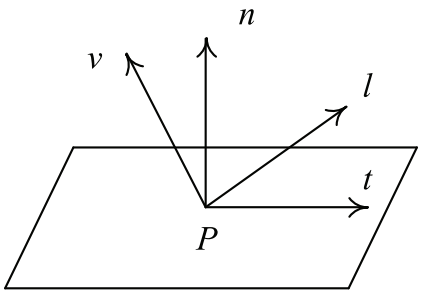

Рис. 9.12 ❖ Нормаль и касательный вектор

Обычно в качестве такого вектора выступает так называемый *касательный вектор* (tangent) t, перпендикулярный нормали (и, следовательно, лежащий в плоскости грани). Тогда освещенность точки будет зависеть не только от вектора нормали n, но и от касательного вектора t. По этим двум векторам можно построить третий вектор $b = [n, t]$, называемый *бинормалью*. Вместе векторы n, t и b образуют *касательный базис* (или базис касательного пространства).

Простейшая анизотропная модель освещения считает поверхность состоящей из бесконечно тонких нитей. Соответственно, вектор t в этом случае просто является касательным вектором к нити, проходящей через данную точку (рис. 9.13).

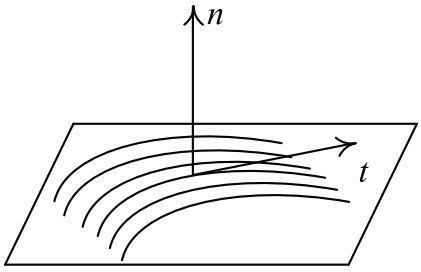

Рис. 9.13 ❖ Представление поверхности через бесконечно тонкие нити

Соответственно, вся модель освещения такой поверхности строится на основе модели освещения бесконечно тонкой нити. В каждой точке такой нити у нас определен касательный вектор t (рис. 9.14).

Взяв точку P на такой нити, мы не можем однозначно определить вектор нормали n к ней – в качестве этого вектора можно взять любой единичный вектор, перпендикулярный касательному вектору t.

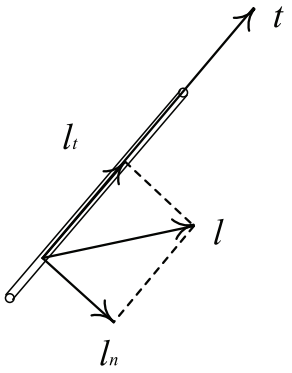

Рис. 9.14 ❖ Векторы для освещения нити

Будем считать, что освещенность в точке P определяется тем (перпендикулярным n) вектором n, который дает максимальную яркость. Исходя из этого, мы раздельно найдем диффузную и бликовую яркости в нашей точке.

Для этого мы разложим вектор на источник света l на две части – l_t и l_n. Первая из этих частей – l_t – параллельна вектору t, а вторая – перпендикулярна ему:

$$l = l_t + l_n.$$

Так как $l_t \parallel t$, то этот вектор отличается от t просто скалярным множителем. Таким образом, считая вектор t единичным, мы получаем:

$$l_t = (l, t)t;$$
$$l_n = l - (l, t)t.$$

В этом случае $(n, l) = (l_n, n)$, и максимальное значение будет достигаться, когда вектор n параллелен вектору l_n. Найдем явное выражение для этого вектора n:

$$n = \frac{l_n}{\|l_n\|} = \frac{l-(l,t)t}{\|l-(l,t)t\|}.$$

Нужная нам норма вектора l_n легко может быть найдена из следующего скалярного произведения:

$$\|l - (l, t)t\|^2 = (l, l) - 2(l, t)^2 + (t, t)(l, t)^2 = 1 - (l, t)^2.$$

Тогда мы получим следующее выражение для (l, n):

$$(l,n) = \frac{l-(l,t)t}{\sqrt{1-(l,t)^2}} = \sqrt{1-(l,t)^2}.$$

Таким образом, диффузная составляющая будет задаваться следующей формулой:

$$I = k_d C I_l \sqrt{1-(l,t)^2}.$$

Аналогично можно получить выражения для (n, h):

$$\max(n,h) = (1-(h,t))^{p/2}.$$

В результате мы приходим к следующей формуле для освещения анизотропной поверхности:

$$I = k_a C I_a + k_d C I_l \sqrt{1-(l,t)^2} + k_s C_s I_l (1-(h,t))^{p/2}.$$

На рис. 9.15 приведены изображения сфер с различными значениями p.

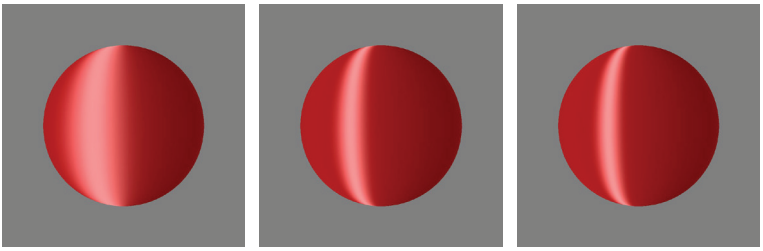

Рис. 9.15 ❖ Изображения сфер с анизотропным освещением (со степенью 10, 50 и 70 соответственно)

Анизотропная модель Уорда

Еще одной распространенной моделью для освещения анизотропных поверхностей является анизотропная модель Уорда (Ward), являющаяся расширением соответствующей изотропной модели. Она задается следующей формулой:

$$I = k_d I_l C + k_s C_s I_l \cdot \exp(-k_t (h, t)^2 - k_b (h, b)^2).$$

Двулучевая функция отражательной способности (BRDF)

Все ранее рассмотренные модели освещения были эмпирическими – они не были основаны на строгих расчетах распространения световой энергии. В этом разделе мы введем как необходимые понятия для учета распределения энергии, так и функции, описывающей, каким именно образом световая энергия, падающая в точку, распределяется по различным направлениям.

Как и ранее, мы будем считать, что у нас есть некоторая точка P и в этой точке задан единичный вектор нормали n. Также мы будем считать, что есть некоторая бесконечно малая площадка dA, содержащая эту точку и перпендикулярная вектору нормали (рис. 9.16).

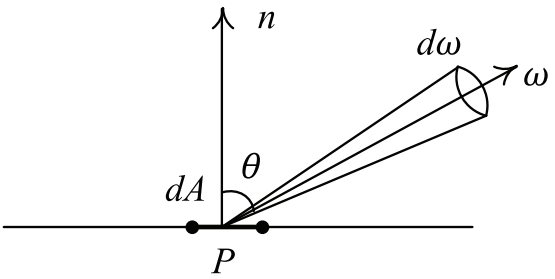

Рис. 9.16 ❖ Падение света на бесконечно малую площадку dA

Падающий на эту площадку свет описывается при помощи мощности (измеряемой в ваттах, W). Однако нас интересует не столько падающая мощность, сколько мощность, приходящаяся на единицу площади, измеряемая в W/m^2. Именно она будет определять то, насколько яркой мы будем видеть точку P.

Рассмотрим теперь некоторое направление ω и бесконечно малый *телесный угол* $d\omega$. Под телесным углом можно понимать некоторую область на единичной сфере, построенной вокруг точки P. Площадь этой области и будет равна величине нашего телесного угла. Обычный (плоский) угол измеряется в радианах, телесный угол измеряется в *стерадианах* (*sr*).

Через этот телесный угол $d\omega$ на нашу площадку dA падает некоторый световой поток $d\Phi$ (измеряемый в ваттах). Введем теперь понятие *энергетической яркости* (radiance) L как отношение падающей мощности $d\Phi$ к величине телесного угла $d\omega$ и площади проекции dA на плоскость, перпендикулярную направлению ω (обозначаемую далее как dA_{proj}):

$$L = \frac{d\Phi}{d\omega \cdot dA_{proj}}.$$

При этом площадь dA и площадь ее проекции dA_{proj} связаны между собой следующим соотношением (через θ обозначен угол между нормалью n и направлением ω):

$$dA_{proj} = dA \cdot \cos\theta.$$

Таким образом, энергетическая яркость может быть выражена следующей формулой:

$$L = \frac{d\Phi}{d\omega \cdot dA \cdot \cos\theta}.$$

Она имеет размерность $\frac{W}{sr \cdot m^2}$ и описывает плотность светового потока, падающего на единицу площади через единичный телесный угол. С ее помощью можно на самом деле описывать как падающий, так и отраженный свет.

Тем самым если у нас есть площадка dA и телесный угол $d\omega$, то мощность, падающая на единицу площади (*облучательность*, irradiance), будет равна:

$E = L \cdot \cos\theta \cdot d\omega$.

Рассмотрим теперь два направления – ω_i (откуда свет падает) и ω_o (куда он уходит). Также будем считать, что есть бесконечно малый телесный угол $d\omega_i$, через который на площадку падает свет. Плотность световой мощности, рассеиваемой в направлении ω_o, обозначим через dL_o (обратите внимание, что это тоже будет бесконечно малая величина).

Тогда мы можем ввести *двулучевую функцию отражательной способности* (BRDF, Bidirectional Reflection Distribution Function) как отношение dL_o и $dE_i = L_i \cdot \cos\theta_i \cdot d\omega_i$:

$$f_r(\omega_i, \omega_r) = \frac{dL_o}{dE_i} = \frac{dL_o}{L_i \cdot \cos\theta_i \cdot d\omega_i}.$$

Обратите внимание, что это не безразмерная величина – она имеет размерность $1/sr^2$. Эта функция всегда принимает неотрицательные значения, но она может принимать значения, большие единицы (из-за деления на $\cos\theta_i$).

BRDF фактически описывает, как именно падающий свет рассеивается в различные стороны. И строгое описание свойств поверхности заключается в задании *BRDF* для этой поверхности.

Простейшим вариантом *BRDF* является константа – падающий свет равномерно рассеивается во все стороны (идеальная диффузная поверхность, константа π возникает из условия того, чтобы вся рассеиваемая энергия равнялась падающей):

$$f_r(\omega_i, \omega_o) = \frac{C_{diff}}{\pi}.$$

Случаю идеального зеркального отражения соответствует следующая функция (через $\delta(x)$ обозначена дельта-функция Дирака, а через $R(\omega, n)$ – функция, дающая отраженное направление):

$$f_r(\omega_i, \omega_o) = \frac{\delta(\omega_i - R(\omega_o, n))}{\cos\theta_i}.$$

Если мы знаем $L_i(\omega)$ – падающий свет со всех сторон на точку P и *BRDF* в этой точке, то мы можем описать световую энергию, отражаемую в заданном направлении ω_o через следующий интеграл:

$$L_o(\omega) = \int_{S^+} f_r(\omega_o, \omega_i) L_i(\omega_i) \cos\theta_i d\omega_i.$$

Физически корректные модели освещения

Сейчас получили большое распространение физически корректные модели освещения (PBR, Physically Based Rendering). Однако прежде чем давать определение физически корректных моделей освещения, давайте более подробно рассмотрим, за счет чего именно возникают диффузная и бликовая части освещения.

Когда свет падает на поверхность, то его часть, определяемая коэффициентом Френеля F, отразится. Поскольку в жизни нет идеальных поверхностей, то реальная поверхность будет иметь микроскопические отклонения от идеальной поверхности – *микрорельеф*.

Свет будет отражаться от этого микрорельефа, в результате чего мы получим не один отраженный луч, а некоторое распределение отраженного света (рис. 9.17). Чем более неровной является поверхность, тем сильнее будут расходиться отраженные лучи.

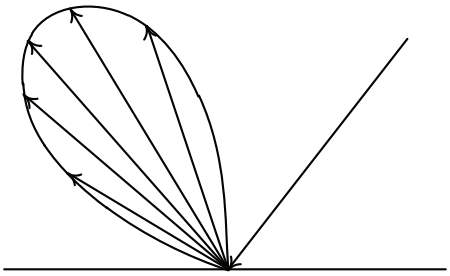

Рис. 9.17 ❖ Распределение отраженного света

Рассмотрим теперь, что происходит с оставшейся световой энергией. Она преломится. Что именно с ней будет, зависит от того, является материал проводником или диэлектриком.

Если материал является проводником (металл), то весь преломленный свет полностью поглотится – диффузной компоненты вообще не будет.

Для диэлектриков преломленный свет внутри материала рассеивается. Часть рассеянного света выходит наружу. При этом, кроме рассеивания света, внутри материала происходит и его поглощение материалом (рис. 9.18).

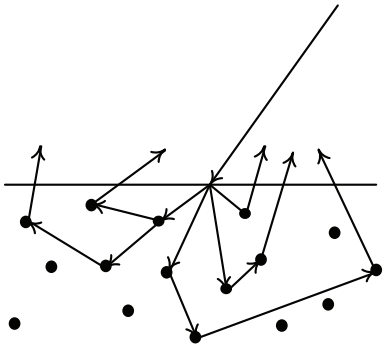

Рис. 9.18 ❖ Рассеивание света в материале

Если имеет место достаточно сильное поглощение, то выходящий наружу рассеянный свет будет выходить очень близко к точке падения исходного света. Поэтому для таких материалов обычно считают, что рассеянный свет выходит наружу прямо в точке падения.

Есть материалы, для которых степень поглощения преломленного света невысока. К таким материалам относится, например, человеческая кожа. В этом случае мы уже не можем считать, что преломленный свет выходит точно в точке падения.

Именно такой преломленный и рассеянный свет и формирует диффузное освещение. При этом направление такого света очень мало зависит от направления падающего света.

Для того чтобы мы могли считать заданную *BRDF* физически корректной, она должна удовлетворять двум базовым требованиям – *принципу взаимности Гельмгольца* (Helmholtz reciprocity) и *закону сохранения энергии*.

Первое из этих требований говорит о том, что если мы обратим направление движения света (т. е. поменяем местами источник и приемник света), то ничего не изменится – *BRDF* симметрична:

$$f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i).$$

Второе требование заключается в том, что поверхность не может отражать больше световой энергии, чем на нее падает. Вся падающая световая энергия может быть выражена в виде интеграла по верхней полусфере S^+. В результате мы приходим к следующему неравенству:

$$\int_{S^+} f_r(\omega_i, \omega_o)\cos\theta_i d\omega_i \leq 1.$$

Для ранее рассмотренных моделей Фонга и Блинна–Фонга не выполнен закон сохранения энергии. Для его выполнения необходимо ввести специальный нормирующий коэффициент для бликовой части. С учетом этой поправки физически корректные версии моделей Фонга и Блинна–Фонга выглядят следующим образом:

$$I = k_d I_l \frac{C}{\pi} + k_s I_l C_s \frac{p+1}{2\pi} \max((r,v),0)^p;$$

$$I = k_d I_l \frac{C}{\pi} + k_s I_l C_s \frac{p+2}{4\pi(2-2^{-p/2})} \max((h,n),0)^p.$$

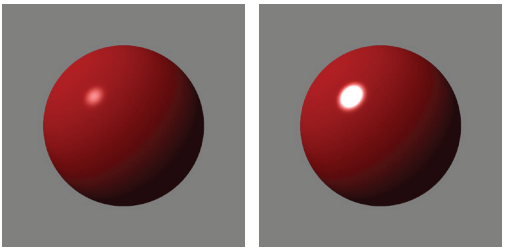

Рис. 9.19 ❖ Сравнение обычной и нормализованной моделей Фонга (p = 70)

Далее мы рассмотрим некоторые физически корректные модели освещения.

Модель Орена–Найара

Большинство физически корректных моделей освещения основано на *микрофасетной модели* поверхности. Согласно этой модели, вся поверхность состоит из микроскопических граней. При этом все эти грани считаются случайно ориентированными, но при этом задается какой-то закон распределения для нормалей h этих граней (рис. 9.20) (NDF, *Normal Distribution Function*). Также считается, что каждая грань удовлетворяет какой-то очень простой модели освещения.

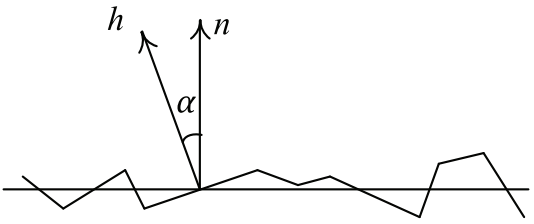

Рис. 9.20 ❖ Микрофасетная модель

При этом в произвольной точке P поверхности у нас есть нормаль n к самой поверхности (точнее, к ее средней линии) и нормаль h к конкретной микрограни. Вектор h является случайной величиной, закон распределения которой $D(\alpha)$ является функцией угла α между векторами h и n.

Модель Орена–Найара (Oren–Nayar), впервые опубликованная в 1992 г., является расширением модели Ламберта. Согласно этой модели, каждая микрограни является идеальным диффузным отражателем.

Но при этом мы учитываем то, что направление к наблюдателю *v* влияет на видимую освещенность (в отличие от классической модели Ламберта). Как показано на рис. 9.21, при одном и том же положении источника света освещенность зависит от положения наблюдателя. В случае когда наблюдатель находится справа, то он видит в основном освещенные грани. А в случае когда он находится слева – неосвещенные.

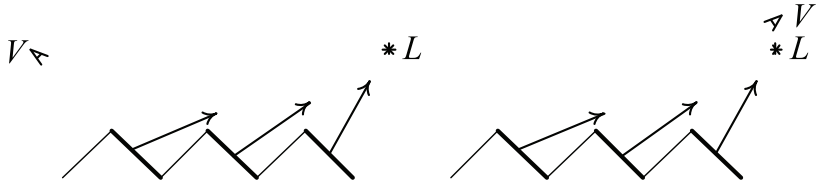

Рис. 9.21 ❖ Влияние положения наблюдателя на видимую освещенность. В одном случае (слева) мы видим неосвещенные грани, в другом (справа) – освещенные

В модели Орена–Найара распределение микрограней (NDF) считается гауссовым:

$$D(\alpha) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\alpha^2}{2\sigma^2}}.$$

Константа перед экспонентной в этой формуле выбирается из условия нормировки:

$$\int D(\alpha) d\alpha = 1.$$

Параметр σ соответствует неровности поверхности. Значение $\sigma = 0$ соответствует идеально гладкой поверхности (зеркалу).

Полная модель Орена–Найара довольно сложна (например, она учитывает переотражение света между отдельными микрогранями), поэтому обычно используют ее упрощенный вид, приводимый ниже:

$$I = \frac{C}{\pi} \max(\cos\theta, 0) \cdot (A + B \cdot \max(0, \cos(\varphi_v + \varphi_l)) \cdot \sin\alpha \cdot \tan\beta).$$

Здесь использованы следующие параметры (параметр σ отвечает за степень неровности поверхности):

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33};$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09};$$

$$\alpha = \max(\theta_l, \theta_v);$$

$$\beta = \min(\theta_l, \theta_v).$$

В этих формулах углы θ_l и θ_v – это углы между нормалью *n* и векторами *l* и *v*. Также угол $\varphi_l - \varphi_v$ – это угол между проекциями векторов *l* и *v* на касательную плоскость (рис. 9.22).

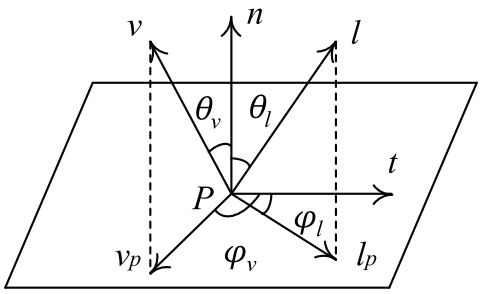

Рис. 9.22 ❖ Используемые в модели Орена–Найара углы

Довольно часто в этой формуле используется косинус не суммы, а разности углов φ_l и φ_v. На самом деле дело в том, что там применяется не вектор направления на источник света, а вектор направления, в котором свет падает.

Для того чтобы найти нужный нам $\cos(\varphi_l - \varphi_v)$, мы сначала спроектируем векторы l и v на касательную плоскость, получив при этом векторы v_p и l_p:

$v_p = v - n(n, v)$;
$l_p = l - n(n, l)$.

Далее можно легко найти их длины и косинус угла между ними:

$$\|v_p\|^2 = \|v\|^2 - 2(n,v)^2 + n^2(n,v)^2 = 1-(n,v)^2;$$

$$\|l_p\|^2 = 1-(n,l)^2;$$

$$\cos(\varphi_v - \varphi_l) = \frac{(v_p, l_p)}{\|v_p\| \cdot \|l_p\|} = \frac{(v,l)-(n,v)(n,l)}{\sqrt{1-(n,v)^2}\cdot\sqrt{1-(n,l)^2}}.$$

Легко заметить, что оба угла θ_l и θ_v лежат на отрезке $\left[0, \frac{\pi}{2}\right]$. Соответственно, поскольку и синус, и тангенс на этом отрезке строго неотрицательны и монотонны, мы получаем:

$\cos\alpha = \cos(\max(\theta_l, \theta_v)) = \min(\cos\theta_l, \cos\theta_v)$;
$\cos\beta = \cos(\min(\theta_l, \theta_v)) = \min(\cos\theta_l, \cos\theta_v)$.

Выразив $\sin\alpha$ и $\tan\beta$ через косинусы углов, мы получаем следующую формулу:

$$\sin\alpha\cdot\tan\beta = \frac{\sqrt{1-\cos^2\alpha}\cdot\sqrt{1-\cos^2\beta}}{\cos\beta}.$$

Эта формула легко может быть переписана таким образом:

$$\sin\alpha\cdot\tan\beta = \frac{\sqrt{1-\cos^2\theta_l}\cdot\sqrt{1-\cos^2\theta_v}}{\cos\beta}.$$

На рис. 9.23 приведено сравнение модели Орена–Найра для различных значений параметра σ с моделью Ламберта.

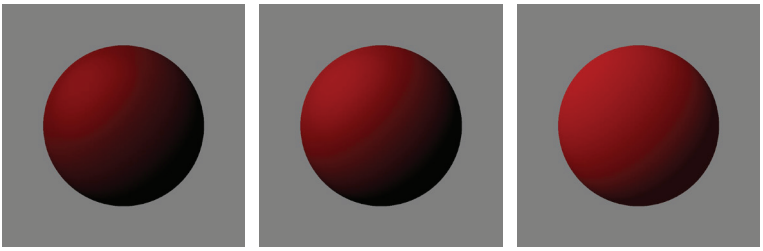

Рис. 9.23 ❖ Сравнение модели Ламберта с моделью Орена–Найара (со степенью неровности 1 и 0.5 соответственно)

Модель Кука–Торранса

Одной из самых известных и широко распространенных моделей освещения для получения качественных бликов является модель Кука–Торранса (Cook–Torrance).

Как и ранее рассмотренная модель Орена–Найара, эта модель является физически корректной и также основана на микрофасетной модели поверхности. Эта модель использует коэффициент Френеля для точного учета количества отраженного света, также использует специальный геометрический член для учета возможного закрывания микрогранями друг друга.

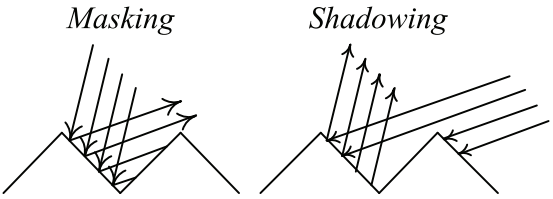

Рис. 9.24 ❖ Случаи закрывания микрогранями друг друга

Геометрический член G отвечает за два типа закрывания микрогранями друг друга – закрывание от источника света (*shadowing*, рис. 9.24 справа) и закрывание от наблюдателя (*masking*, рис. 9.24 слева). Данный член в модели Кука–Торранса задается следующей формулой:

$$G_{Cook-Torrance} = \min\left(1, \frac{2(n,h)(n,v)}{(v,h)}, \frac{2(n,h)(n,l)}{(v,h)}\right).$$

Для задания неровности поверхности в этой модели используется распределение Бэкмена $D_B(\alpha)$. Параметр m отвечает за неровность поверхности:

$$D_{Beckman}(\alpha) = \frac{1}{m^2 \cos^4 \alpha} \exp\left(-\left(\frac{\tan \alpha}{m}\right)^2\right).$$

Используя скалярное произведение, мы можем переписать эту формулу следующим образом:

$$D_{Beckman}(\alpha) = \frac{1}{m^2 (n,h)^4} \exp\left(-\frac{1-(n,h)^2}{m^2 (n,h)^2}\right).$$

Итоговая формула освещенности для модели Кука–Торранса выглядит следующим образом:

$$I = I_l \frac{F(\theta) D_B(\alpha) G}{4(n,l)(n,v)}.$$

Кроме распределения Бэкмена, также иногда используется GGX-распределение (распределение Троубриджа–Рейтца, Trowbridge–Reitz):

$$D_{GGX}(\alpha) = \frac{\sigma^2}{\pi\left(\left((n,h)^2(\sigma^2-1)+1\right)\right)^2}.$$

Существует также анизотропный вариант распределения GGX, приводимый ниже:

$$D_{GGX-aniso}(\alpha) = \frac{1}{\pi\sigma_x\sigma_y}\frac{1}{\left(\frac{(t,h)^2}{\sigma_x^2}+\frac{(b,h)^2}{\sigma_y^2}+(n,h)^2\right)^2}.$$

Кроме того, существуют и другие варианты геометрического члена G:

$$G_{Implicit}(n,l,v) = (n,l)(n,v);$$

$$G_{smith}(l,v,h) = G_1(l)G_1(v);$$

$$G_{1,Schlick}(v) = \frac{(n,v)}{(n,v)(1-k)+k}, k = \frac{2\sigma}{\pi};$$

$$G_{Neumann}(l,v,h) = \frac{(n,l)(n,v)}{\max\left((n,l),(n,v)\right)};$$

$$G_{Kelem}(l,v,h) = \frac{(n,l)(n,v)}{(v,h)^2};$$

$$G_{GGX}(v) = \frac{2(n,v)}{(n,v)+\sqrt{\sigma^2+(1-\sigma^2)(n,v)^2}}.$$

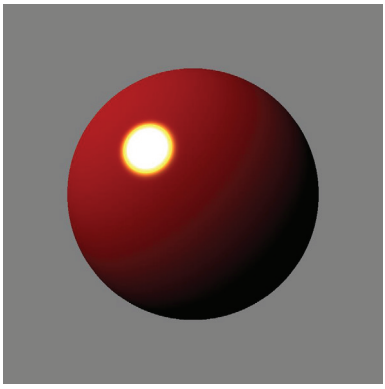

Рис. 9.25 ❖ Изображения сферы с использованием модели Кука–Торранса

Диффузная модель Диснея

Компанией Disney для мультфильма *Wreck It Ralph* была предложена модель освещения с новым диффузным членом. Соответствующая BRDF задавалась следующей формулой:

$$f_r = \frac{C}{\pi}\left(1+(F_{d90}-1)(1-\cos\theta_l)^5\right)\left(1+(F_{D90}-1)(1-\cos\theta_v)^5\right).$$

Здесь C – это цвет поверхности в точке, θ_l – угол между нормалью и вектором l, θ_v – угол между нормалью и вектором v. Коэффициент F_{D90} задается следующей формулой:

$$F_{D90} = 0.5 + 2\cos^2\theta_d \cdot \alpha.$$

В этой формуле θ_d – это угол между векторами l и h, а α – это степень неровности поверхности.

Обратите внимание, что для этой модели освещения не выполнен закон сохранения энергии – это было сознательным решением, направленным на удобство художников и моделлеров. Вскоре компания Frostbite предложила модификацию этой модели с выполнением закона сохранения энергии:

$$f_r = \frac{C}{\pi}\left(1+(F_{d90}-1)(1-\cos\theta_l)^5\right)\left(1+(F_{D90}-1)(1-\cos\theta_v)^5\right)\left(1+\alpha\left(\frac{1}{1.51}-1\right)\right).$$

Также была слегка изменена формула, по которой шел расчет коэффициента F_{D90}:

$$F_{D90} = 0.5\alpha + 2\cos^2\theta_d \cdot \alpha.$$

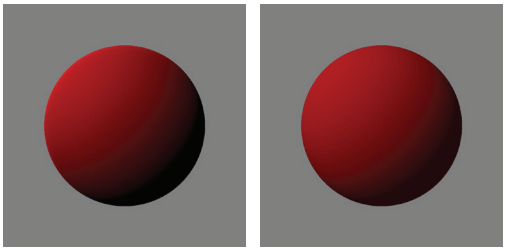

Рис. 9.26 ❖ Сравнение диффузной модели Диснея и модели Ламберта

Модель Ашихмина–Ширли

Еще одной физически корректной моделью является модель Ашихмина–Ширли (Ashikhmin–Shirley). Эта модель ориентирована на анизотропные материалы. Она использует четыре простых и понятных параметра и является более простой для расчетов, чем модель Кука–Торранса.

Эта модель использует функцию распределения $D(\alpha)$ следующего вида:

$$D(\alpha) = \sqrt{(e_t+1)(e_b+1)}\cdot\max(0,(n,h))^{e_t\cos^2\varphi_h + e_b\sin^2\varphi_h}.$$

Данную формулу можно переписать, используя скалярные произведения следующим образом:

$$D(\alpha) = \sqrt{(e_t + 1)(e_b + 1)} \cdot \max(0, (n,h))^{\frac{e_t(h,t)^2 + e_b(h,b)^2}{1-(n,h)^2}}.$$

Сама модель Ашихмина–Ширли состоит из диффузной и бликовой частей и записывается следующим образом:

$$I = \max(0, (n,l)) \cdot I_l \cdot (I_d + I_s).$$

Соответственно, диффузная и бликовая части задаются приводимыми ниже формулами:

$$I_s = \frac{D(\alpha)}{8\pi} \cdot \frac{F(\theta, C_s)}{(v,h) \cdot \max((n,l)(n,v))};$$

$$I_d = \frac{28}{23\pi} \cdot C_d (1 - C_s) \cdot \left(1 - \left(1 - \frac{(n,v)}{2}\right)^5\right) \cdot \left(1 - \left(1 - \frac{(n,l)}{2}\right)^5\right).$$

Данная модель использует два цвета – C_d (диффузный) и C_s (бликовый), а также два коэффициента – e_t и e_b. При этом случай $e_t = e_b$ соответствует изотропному материалу.

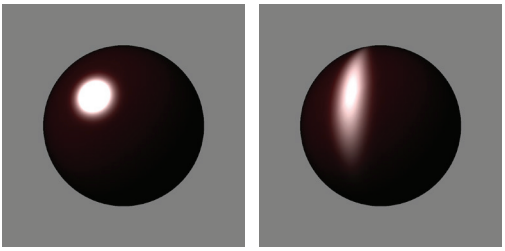

Рис. 9.27 ❖ Изображения сфер, выведенных при помощи модели Ашихмина–Ширли

Image-based lighting

Очень часто мы сталкиваемся с тем, что свет в точку падает не из небольшого числа направлений, а со всех сторон. Например, это справедливо для точек на ландшафте, когда свет падает со всего неба, т. е. со всей верхней полусферы (рис. 9.28).

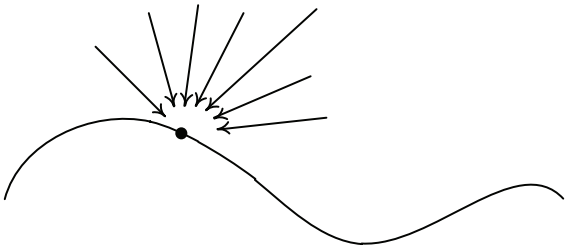

Рис. 9.28 ❖ Освещение точки на ландшафте

Для расчета освещения точки в этом случае необходимо вычислять интеграл по всей верхней полусфере от произведения *BRDF* и функции распределения падающего света. Однако точно считать подобный интеграл в реальном времени пока еще не возможно, поэтому приходится прибегать к различным упрощениям и приближениям.

В целом ряде случаев оказывается очень удобным задавать освещение, падающее со всех сторон при помощи специальных, как правило, заранее подготовленных, изображений. Так, такое изображение может задавать свет, падающий в точку для каждого возможного направления.

Одним из простейших приемов подобного рода является расстановка в пространстве в узлах регулярной сетки так называемых *приемников* (probe) (рис. 9.29). Каждый из них представляет из себя куб со стандартной ориентацией в пространстве, и для каждой его грани мы храним всего один цвет.

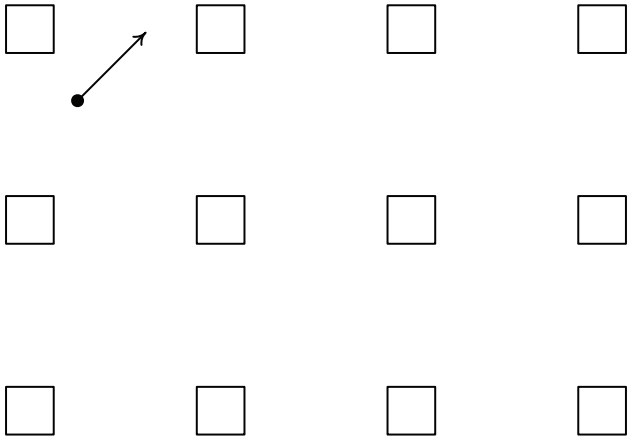

Рис. 9.29 ❖ Сетка из probe

Тогда если нас интересует свет, падающий из заданного направления, мы просто раскладываем это направление по нормалям к граням ближайшего куба. После этого мы используем получившиеся веса для смешивания цветов с соответствующих граней куба.

Цвета для набора этих приемников (ambient cubes) обычно считают заранее, на этапе подготовки геометрии сцены. На этапе рендеринга мы просто находим ближайший к интересующей нас точке куб и по нужному нам направлению определяем цвет.

Вместо приемников, содержащих всего по одному цвету для каждой его грани, можно использовать для каждой из сторон полноценные изображения. Тогда мы фактически получаем функцию, которая для каждого направления дает приходящий из него свет.

Тогда можно для вычисления интеграла использовать следующую приближенную формулу:

$$\int_{S^+} L_i(l) f_r(l,v)\cos\theta_l dl \approx \frac{1}{N}\sum_{k=1}^{N}\frac{L_i(l_k) f_r(l_k,v)\cos\theta_k}{p(l_k,v)} \approx \left(\frac{1}{N}\sum_{k=1}^{N} L_i(l_k)\right)\cdot\left(\frac{1}{N}\sum_{k=1}^{N}\frac{f_r(l_k,v)\cos\theta_k}{p(l_k,v)}\right).$$

Первый член – это просто усреднение падающего освещения. По заранее заданным шести изображениям с граней куба мы можем на этапе подготовки геометрии провести необходимое усреднение.

Второй множитель – это просто функция от направления и угла θ_v и также может быть заранее рассчитана и помещена в специальную двухмерную таблицу (текстуру). Пример использования этого подхода будет нами рассмотрен в главе 19.

Сферические гармоники и их использование

Для точного расчета освещения нам необходимо знать, как свет со всех сторон падает на заданную точку. Такой падающий свет может быть представлен как функция направления, т. е. функция, определенная на единичной сфере S.

Рассмотрим теперь множество всех функций, определенных на S, таких что их квадрат будет интегрируем на S. Это множество функций обозначается как $L_2(S)$.

На самом деле это бесконечномерное векторное пространство (так, при сложении и умножении на константу мы снова получаем функцию, квадрат которой будет интегрируем на S).

В этом пространстве мы можем ввести скалярное произведение следующим образом (соответствующий интеграл всегда существует в силу интегрируемости квадратов функций):

$$(f,g)=\int_S f(\omega)g(\omega)d\omega=\int_0^{2\pi}\int_0^{\pi} f(\theta,\varphi)g(\theta,\varphi)\sin\theta d\theta d\varphi.$$

Через скалярное произведение мы можем ввести длину (норму):

$$\|f\|=\sqrt{(f,f)}.$$

Рассмотрим теперь некоторый ортонормированный базис $\{e_i(\omega)\}$ в пространстве $L_2(S)$. Будем считать, что любую функцию $f\in L_2(S)$ можно разложить по этому базису, т. е. для нее существуют такие коэффициенты f_i, что справедливо разложение:

$$f(\omega)=\sum_{i=0}^{\infty} f_i e_i(\omega).$$

Так как базис $\{e_i(\omega)\}$ ортонормирован, то, как и в конечномерном случае, справедливо

$$f_i=(f,e_i)=\int_S f(\omega)e_i(\omega)d\omega.$$

Если у нас есть две функции $f(\omega)$ и $g(\omega)$ и каждая из них разложена по этому базису, то для вычисления их скалярного произведения нам уже не нужно считать интеграл:

$$(f,g)=\sum_{i=0}^{\infty} f_i g_i.$$

Рассмотрим теперь в пространстве $L_2(S)$ линейный оператор. Можно показать, что произвольный линейный оператор в этом пространстве задается при помощи своего ядра – функции $k(\omega_1, \omega_2)$ – следующим образом:

$$(Af)(\omega)=\int k(\omega,\omega')f(\omega')d\omega'.$$

Рассмотрим теперь, как оператор A действует на векторы нашего базиса:

$$(Ae_i)(\omega)=\int k(\omega,\omega')e_i(\omega')d\omega=\sum_{j=0}^{\infty} a_{ij}e_j(\omega).$$

Тем самым оператор A задается бесконечной матрицей коэффициентов $\{a_{ij}\}$.

Зафиксируем первые n векторов из нашего базиса. Тогда можно показать, что вектор $\sum_{k=0}^{n-1} f_k e_k(\omega)$ дает наилучшее приближение к f по первым n базисным векторам $e_1, e_2, \ldots, e_{n-1}$.

Выпишем теперь все ранее рассмотренные операции, используя при этом только первые n базисных векторов. В результате мы получим приближенные оценки для ряда величин:

$$(f, g) \approx \sum_{k=0}^{n-1} f_k g_k;$$

$$(Af)(\omega) \approx \sum_{i=0}^{n-1} f_i \sum_{j=0}^{n-1} a_{ij} e_j(\omega).$$

Легко заметить, что уравнение, дающее световую энергию, уходящую в заданную сторону, на самом деле просто является линейным оператором, примененным к распределению падающей световой энергии $L_i(\omega)$.

Все, что нам теперь осталось, – это найти подходящий ортонормированный базис в $L_2(S)$. Для функций, определенных на единичном отрезке, в качестве ортонормированного базиса обычно используется базис Фурье. На самом деле обычный ряд Фурье – это просто разложение функции по ортонормированному базису, состоящему из синусов и косинусов. Для функций, определенных на поверхности единичной сферы, в качестве такого базиса обычно используются так называемые *сферические гармоники*.

Сферические гармоники – это функции $y_l^m(\theta, \varphi)$, определенные следующим образом:

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2} K_l^m \cos(m\varphi) P_l^m(\cos\theta), m > 0 \\ \sqrt{2} K_l^m \sin(-m\varphi) P_l^m(\cos\theta), m < 0 \\ K_l^0 P_l^0(\cos\theta), m = 0 \end{cases}.$$

Здесь индекс l задает полосу (band), и для каждой полосы l индекс m пробегает значения от $-l$ до l. Параметр K_l^m – это нормировочный коэффициент (для того чтобы норма каждой гармоники была равна единице):

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}.$$

Функция $P_l^m(x)$ – это *присоединенные (ассоциированные) многочлены Лежандра*. Они задаются следующими уравнениями:

$$P_l^m(x)(l-m) = x(2l-1)P_{l-1}^m(x) - (l+m-1)P_{l-2}^m(x);$$

$$P_m^m(x) = (-1)^m(2m-1)!!(1-x^2)^{\frac{m}{2}};$$

$$P_{m+1}^m(x) = x(2m+1)P_m^m(x).$$

Ниже приводится нескольких первых полос:

$$P_0^0(x) = 1;$$

$$P_1^0(x) = x;$$

$$P_1^1(x) = -\sqrt{1-x^2};$$

$$P_2^0(x) = \frac{1}{2}(3x^2 - 1);$$

$$P_2^1(x) = -3x\sqrt{1-x^2};$$

$$P_2^2(x) = 3(1-x^2).$$

На рис. 9.30 приводятся графики этих многочленов.

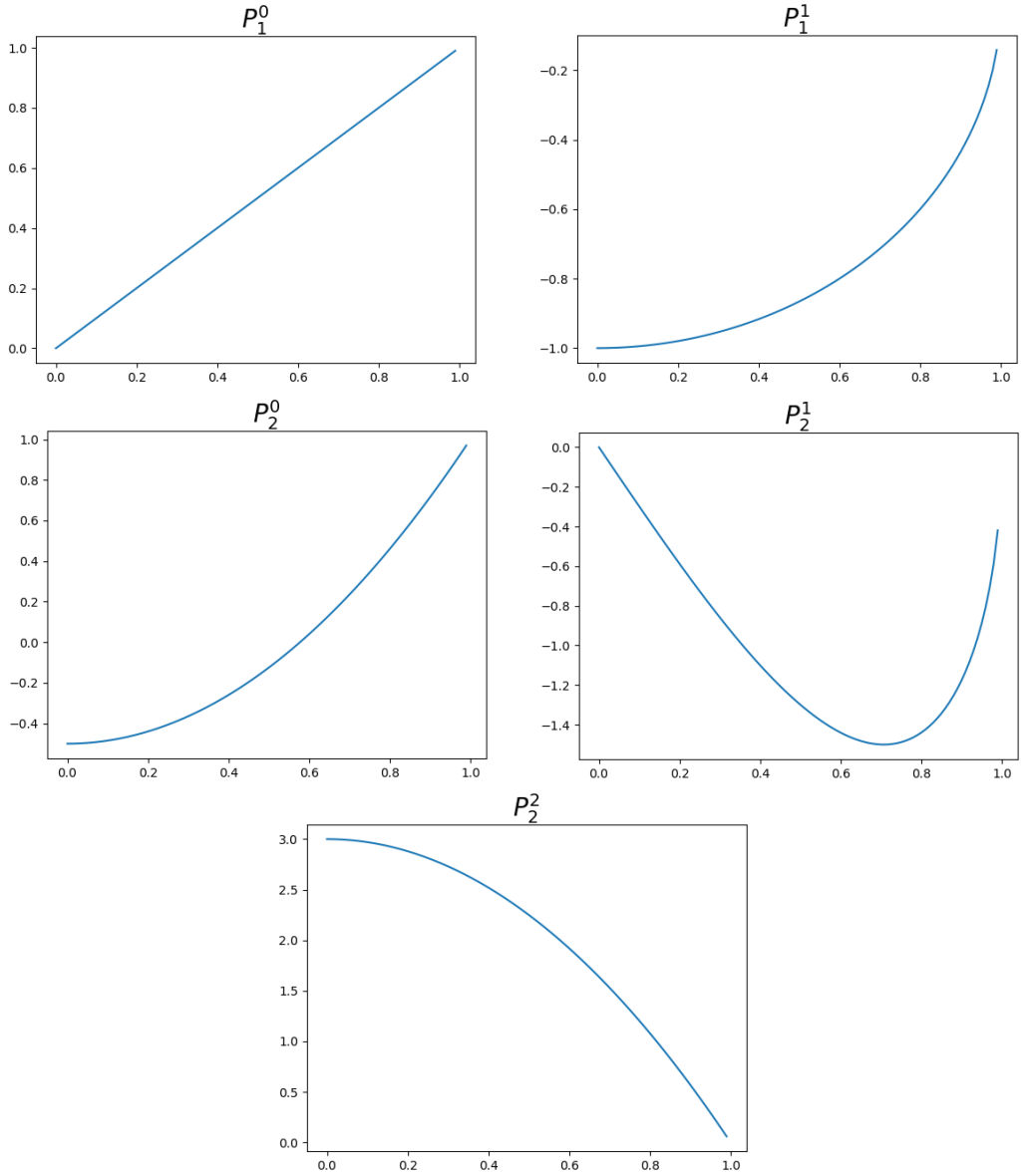

Рис. 9.30 ❖ Графики присоединенных многочленов Лежандра

Так как сферические гармоники определяются через $P_l^m(\cos\theta)$, то ниже приводятся первые два полосы как функции от θ:

$$P_0^0(\cos\theta) = 1;$$
$$P_1^0(\cos\theta) = \cos\theta;$$
$$P_1^1(\cos\theta) = -\sin\theta;$$
$$P_2^0(\cos\theta) = \frac{1}{2}(3\cos^2\theta - 1);$$
$$P_2^1(\cos\theta) = -3\sin\theta\cos\theta;$$
$$P_2^2(\cos\theta) = 3\sin^2\theta.$$

Выпишем теперь соответствующие им сферические гармоники.

$$y_0^0(\theta,\varphi) = \sqrt{\frac{1}{4\pi}};$$
$$y_1^{-1}(\theta,\varphi) = \sqrt{\frac{3}{4\pi}}\cdot\sin\varphi\cdot\sin\theta = \sqrt{\frac{3}{4\pi}}\cdot x;$$
$$y_1^0(\theta,\varphi) = \sqrt{\frac{3}{4\pi}}\cdot\cos\theta = \sqrt{\frac{3}{4\pi}}\cdot z;$$
$$y_1^1(\theta,\varphi) = \sqrt{\frac{3}{4\pi}}\cdot\cos\varphi\cdot\sin\theta = \sqrt{\frac{3}{4\pi}}\cdot y.$$

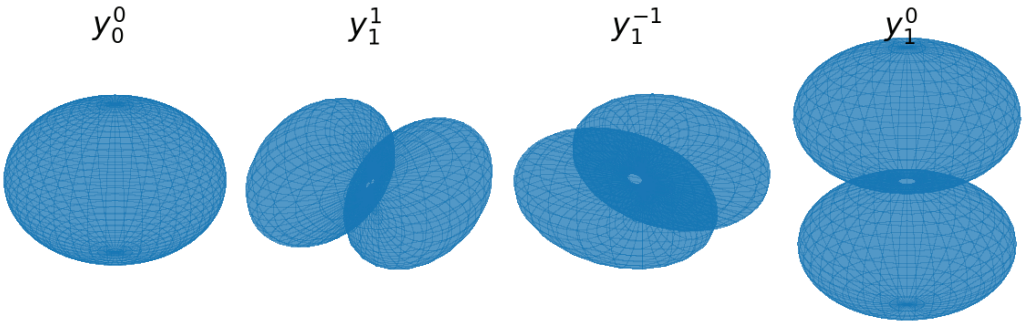

Рис. 9.31 ❖ График гармоник y_0^0, y_1^1, y_1^{-1} и y_1^0

Таким образом, мы получаем набор функций $y_l^m(\theta, \varphi)$, определенных на единичной сфере. Можно показать, что эти функции образуют ортонормированный базис в $L_2(S)$ и любая функция f из $L_2(S)$ может быть разложена в следующий ряд:

$$f(\theta,\varphi) = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} f_l^m y_l^m(\theta,\varphi).$$

Здесь коэффициенты f_l^m определяются таким образом:

$$f_l^m = \int_S f(\theta,\varphi)y_l^m(\theta,\varphi)dS = \int_0^{2\pi}\int_0^{\pi} f(\theta,\varphi)y_l^m(\theta,\varphi)\sin\theta\, d\theta d\varphi.$$

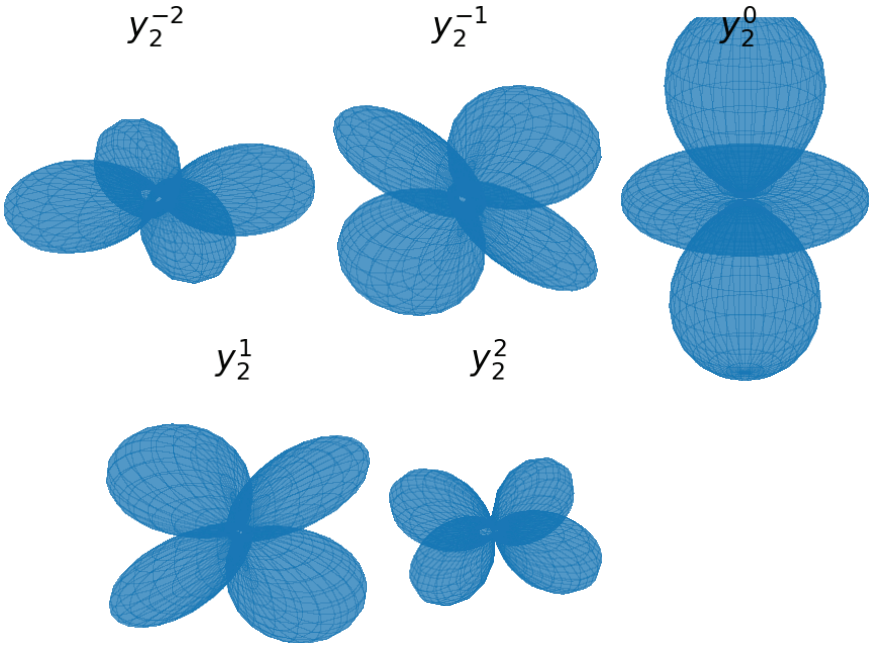

Рис. 9.32 ❖ Графики гармоник второго порядка

Имея ортонормированный базис в $L_2(S)$, можно строить приближенные представления функций направления (как для падающего, так и для отраженного света), используя разложения по гармоникам первых нескольких порядков. Обычно ограничиваются гармониками 0-го и 1-го порядка, что дает нам 4 коэффициента разложения.

Precomputed Radiance Transfer

Представьте себе, что у нас есть некоторый диффузный объект, освещаемый со всех сторон бесконечно удаленными источниками света (например, небом) $L_i(\omega)$.

Тогда освещенность произвольной точки P не зависит от направления на наблюдателя и является только функцией от падающего света $L_i(\omega)$, поэтому может быть выражена следующим образом:

$$L_o = \int_S L_i(\omega) V(\omega) \max(0, \cos\theta) dS.$$

Здесь через V обозначена функция видимости – падает ли свет из направления ω в эту точку или же он закрывается самим объектом.

Данный интеграл может быть записан как скалярное произведение двух функций из $L_2(S) - L_1(\omega)$ и $V(\omega)\max(0, \cos\theta)$.

Разложим каждую из этих функций в ряд по сферическим гармоникам до заданного порядка:

$$L_i(\omega) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} a_l^m y_l^m(\omega);$$

$$V(\omega)\max(0,\cos\theta) = \sum_{l=0}^{n-1}\sum_{m=-l}^{l} t_l^m y_l^m(\omega).$$

Тогда функция $L_o(\omega)$ может быть приближенно записана как следующее скалярное произведение:

$$L_o = \sum_{l=0}^{n-1}\sum_{m=-l}^{l} a_l^m t_l^m.$$

Данная формула описывает только прямое (непосредственное) освещение. Можно найти коэффициенты $\{t_l^m\}$, отвечающие также за взаимное переотражение света. Обычно это делается следующим образом: в качестве распределения для падающего света берется одна гармоника и при помощи трассировки лучей (см. главу 10) считается освещенность точки. Далее берется следующая гармоника, считается освещенность для нее и т. д. В результате мы получаем, как каждая из используемых гармоник освещает нашу точку с учетом переотражения света, загораживания света и т. п. Процесс расчета этих коэффициентов довольно медленный, но он обычно выполняется на этапе подготовки геометрии.

На этапе рендеринга мы просто берем падающее освещение $L_i(\omega)$(заданное в виде кубической текстуры), раскладываем его по сферическим гармоникам и, используя веса, полученные на этапе подготовки данных, сразу же получаем освещенность точки с учетом всех требуемых эффектов.

Подобный подход можно обобщить и на случай недиффузных поверхностей – тогда мы, как и ранее, берем очередную гармонику в качестве падающего освещения и рассчитываем отраженный свет $L_o(\omega)$. Только, в отличие от диффузного случая, когда отраженный свет не зависел от направления, здесь отраженный свет будет функцией направления, и мы раскладываем его по сферическим гармоникам.

В результате мы просто получаем матрицу коэффициентов, которая переводит разложение падающего света по гармоникам в разложение отраженного света по гармоникам. И на этапе рендеринга мы просто раскладываем падающий свет по гармоникам, при помощи построенной матрицы находим разложение отраженного света. Зная разложение отраженного света, можно легко найти отраженный свет для любого требуемого направления.

Подобный подход позволяет получить реалистическое и мягкое освещение любой заданной геометрии и носит название *precomputed radiance transfer* (PRT). Обычно подобная схема используется для рендеринга ландшафтных сцен в зависимости от времени дня.

При этом каждому времени дня соответствует свое падающее освещение (заданное своим разложением по гармоникам). И мы можем легко и быстро найти освещение от падающего со всех сторон света.

Использование PRT в играх Far Cry 3 и FarCry 4

Для получения красивого и реалистичного освещения в играх FarCry 3 и FarCry 4 от компании Ubisoft использовали PRT. На этапе подготовки геометрии в пространстве размещались приемники (probe). Для каждого из них при помощи трассировки лучей вычислялись коэффициенты для PRT. Для FraCry 3 также вычислялось видимая часть неба.

Однако поскольку пробники размещались прямо в пространстве, а не на поверхности, то для них нет какого-либо вектора нормали, который нужен для расчета PRT. Вместо этого в пространстве было зафиксировано четыре единичных вектора, и каждый из них играл роль нормали, и для него находились соответствующие коэффициенты разложения (рис. 9.33).

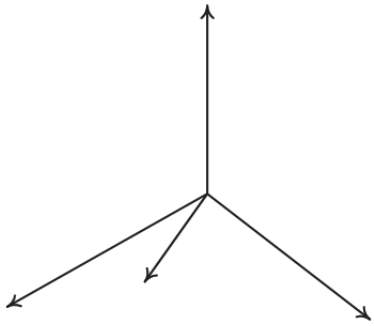

Рис. 9.33 ❖ Векторы, играющие роль нормали

Для хранения найденного освещения использовались сферические гармоники до второго порядка, т. е. произвольная функция на сфере задавалась всего четырьмя коэффициентами. Так как PRT считалось сразу для четырех векторов нормали, то в результате получалась матрица 4×4. Всего хранилось по три такие матрицы – по одной для каждого цветового канала (RGB).

Освещение от неба и солнца задавалось при помощи специальных текстур, определяющих в результате функцию, заданную на верхней полусфере. На этапе рендеринга это падающее от неба и солнца освещение раскладывалось по сферическим гармоникам. Далее, зная коэффициенты для PRT, находились коэффициенты для разложения отраженного света по гармоникам.

В результате для каждого пробника у нас получалось 4 цвета – по одному для каждого из базовых направлений. Полученные значения цветов заносились в трехмерную текстуру (см. главу 14). При рендеринге GPU извлекал соответствующие четыре цвета. Далее конкретная нормаль раскладывалась по заданным векторам, и полученные коэффициенты разложения использовались для смешивания четырех цветов из текстуры. Таким образом, для всех объектов мы получали красивое и естественно выглядящее освещение от Солнца и всего неба.

Глава 10

Трассировка лучей

Одним из самых простых и эффективных способов построения высококачественных изображений является метод *трассировки лучей* (ray tracing). Этот метод фактически моделирует то, каким образом возникает изображение в глазу наблюдателя. Он позволяет строить изображения с точным учетом таких явлений, как преломление и отражение.

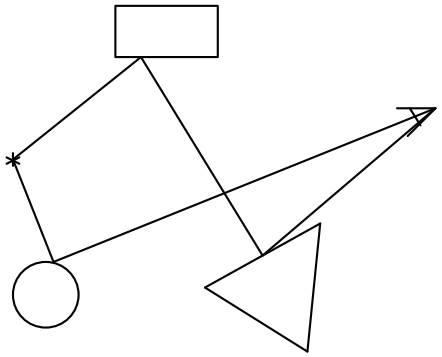

Рис. 10.1 ❖ Прямая трассировка лучей

Рассмотрим простейшую сцену, изображенную на рис. 10.1. Из источника света выходят лучи, вдоль которых распространяется световая энергия. При попадании на границу объекта происходит частично отражение/рассеивание и частично преломление. В результате чего возникают новые лучи света.

Каждый из этих (вторичных) лучей также может попасть на границу раздела сред, приводя к возникновению третичных лучей, и т. д. Часть из этих лучей, в конце концов, попадает в глаз наблюдателя, формируя там изображение сцены.

Описанный процесс называется *прямой трассировкой лучей* (forward ray tracing). Проблема этого подхода заключается в том, что из-за малого размера диафрагмы камеры лишь очень малая часть от всех трассируемых лучей (именно так называется описанный выше процесс отслеживания распространения света в сцене) попадает в глаз наблюдателя.

Поэтому для получения качественного изображения этим методом нужно оттрассировать просто огромное количество лучей, что делает этот метод малопригодным для получения реалистических изображений. Хотя есть способы борьбы с этим.

На практике обычно используется *обратная трассировка лучей* (backward ray tracing). В ней мы отслеживаем лишь те лучи, которые, в конце концов, попадают в глаз наблюдателя. Очевидно, что каждый такой луч проходит через глаз наблюдателя и какой-то пиксел на картинной плоскости (экране). Классической работой по обратной трассировке лучей является статья Уиттеда **https://dl.acm.org/citation.cfm?id=358882**.

Таким образом, мы можем поступить следующим образом: выпустим из глаза наблюдателя (камеры) луч через каждый пиксел экрана и отследим, какая яркость (энергия) приходит вдоль этого луча. Для этого мы отследим распространение луча в направлении, противоположном направлению распространения световой энергии (отсюда и термин – обратная трассировка).

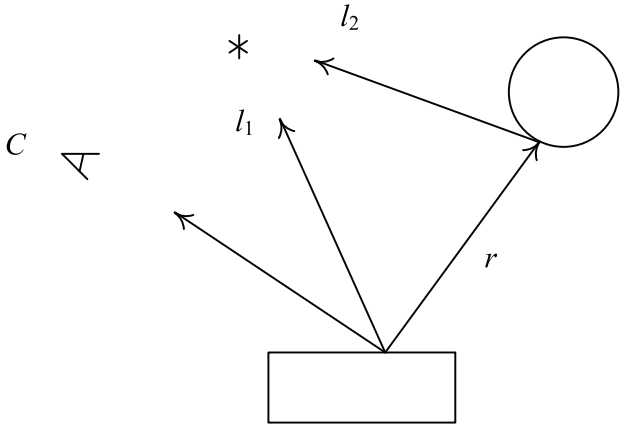

Рис. 10.2 ❖ Обратная трассировка лучей

Мы отслеживаем луч из глаза наблюдателя до его ближайшего пересечения с каким-либо объектом сцены. Далее мы оцениваем световую энергию, падающую в точку пересечения, и, зная *BRDF*, определяем, какая ее доля уйдет по отрассированному нами лучу.

В общем случае, для того чтобы определить световую освещенность, падающую в заданную точку, мы должны найти интеграл по всей верхней полусфере (или даже по всей сфере в случае прозрачного материала). Поскольку это тяжело реализуемо с практической точки зрения, то обычно выделяют несколько ключевых направлений. Вдоль этих направлений выпускаются лучи, определяющие приходящую в точку световую энергию.

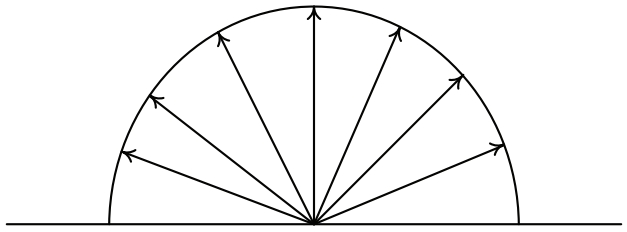

Рис. 10.3 ❖ Определение падающей в точку энергии

Обычно накладывается ряд ограничений, часть из которых мы в дальнейшем снимем. Основным подобным ограничением является то, что мы считаем, что свет, непосредственно падающий от источника света (первичное освещение), вносит гораздо больший вклад, чем вторичное освещение. Также мы считаем, что все источники света являются точечными (или направленными).

В результате мы из точки выпускаем лучи ко всем источникам света и проверяем, не закрывает ли какой-то объект данный источник света. При этом мы считаем, что абсолютно не важно, прозрачен закрывающий объект или нет, – он все равно закрывает нужную нам точку от заданного источника света.

Подобная «дискриминация» прозрачных объектов связана с тем, что для прозрачных объектов все довольно сложно. В качестве примера представьте себе стеклянный шар, закрывающий от источника света точку. Этот шар на самом деле будет работать как линза, и в результате в каких-то «закрытых» им точках света будет очень мало, а в каких-то окажется, наоборот, очень много.

Поскольку в рамках обратной трассировки точно рассчитать это оказывается крайне сложно, то мы просто будем считать, что каждый объект всегда полностью закрывает источник света. Иногда прозрачные объекты можно игнорировать, но это некорректно. Далее мы более подробно рассмотрим, как можно учитывать подобные случаи.

Также мы выпускаем отраженный луч и (если объект прозрачен) преломленный луч. Мы трассируем эти лучи, для того чтобы определить долю световой энергии, проходящей вдоль них.

Фактически мы получаем рекурсивный алгоритм, так как для расчета света, проходящего вдоль отраженного и преломленного лучей, нам опять необходимо выполнить трассировку лучей.

В силу такой рекурсивной природы алгоритма необходимо ввести некоторый критерий прекращения рекурсии. Обычно для этого одновременно используются сразу два таких критерия – по глубине рекурсии и по вкладу луча в готовое изображение.

Первый критерий вводит ограничение на максимальную глубину рекурсии и не выпускает отраженные и преломленные лучи, если максимальная глубина рекурсии уже достигнута.

Второй критерий связан с тем, что вклад отраженного луча в итоговое значение цвета пиксела уменьшается с увеличением глубины рекурсии. Поэтому в ходе трассировки лучей отслеживается, какой максимальный вклад может внести луч, и если этот вклад меньше заданной погрешности, то луч дальше не трассируется.

Можно показать, что на самом деле существуют такие конфигурации, когда каждый из этих критериев приводит к заметной ошибке. Однако на практике оба этих критерия отлично работают, и мы будем использовать их оба.

При написании трассировщика лучей мы будем использовать следующее уравнение освещенности:

$$I = I_a k_a C + k_d C \sum I_i \max(0,(n,l_i)) + k_s \sum I_i f_r(n,l_i,v) + k_r I_r F_r(\theta_r) + k_t I_t (1 - F_r(\theta_r)).$$

Через f_r обозначена *BRDF* поверхности в точке попадания луча, через l_i и I_i – единичное направление на i-й источник света и его яркость/цвет, C – цвет поверхности в точке, n – нормаль к поверхности в точке.

Через I_r и I_t обозначена световая энергия, проходящая вдоль отраженного и преломленного лучей. Коэффициенты k_a, k_d, k_s, k_r и k_t характеризуют поверхность в точке (и зависят от этой точки).

Введем специальный класс SurfaceData для хранения параметров, необходимых для расчета освещенности в точке. Его описание приводится ниже.

```
struct SurfaceData
{
      glm::vec3   pos;               // точка, для которой задаются свойства
      glm::vec3   n;                 // внешняя нормаль в точке
      glm::vec3   color;             // цвет в точке
      glm::vec3   emission;          // собственная светимость в точке
      float       ka, kd, ks, kr, kt;
      float       p;                 // степень для бликовой модели
      Medium      medium;            // материал
};
```

Здесь через структуру Medium описывается однородный материал, в котором трассируется луч – он характеризуется коэффициентом преломления и степенью поглощения энергии.

```
struct      Medium      // основные свойства среды
{
      float nRefr;      // коэффициент преломления
      float extinct;    // коэффициент поглощения
};
```

Также нам понадобится класс LightSource для представления источников света, его описание приводится ниже. Основным методом данного класса является метод shadow, который определяет вектор направления на источник света и долю световой энергии (или 0, если источник света закрыт для заданной точки).

```
class LightSource
{
public:
      glm::vec3      color;

      LightSource ( const glm::vec3& c ) : color ( c ) {}
      virtual ~LightSource () {}

                        // виден ли источник и ослабление света
      virtual float shadow ( const glm::vec3& pt,  glm::vec3& l ) const = 0;
};
```

Ключевым классом, хранящим в себе всею геометрию сцены, включая источники света, является класс Scene, описание которого приводится ниже.

```
class Scene
{
      std::vector<LightSource *>  lights;         // источники света
      std::vector<Object *>       objects;        // объекты
      glm::vec3                   ambient = glm::vec3 ( 0.2 );

public:
      Scene ();
      virtual ~Scene ();

      virtual void addObject ( Object * object )
```

```
    {
        objects.push_back ( object );
    }

    virtual void addLight  ( LightSource * light )
    {
        lights.push_back ( light );
    }

                    // цвет для луча, не попавшего никуда
    virtual glm::vec3 getBackground ( const ray& r ) const
    {
        return glm::vec3 ( 0, 0.5, 0.5 );
    }

                    // найти ближайшее пересечение луча
    virtual Object * intersect ( const ray& r, HitData& hit ) const;

                    // выполнить трассировку луча
    glm::vec3 trace ( const Medium& curMedium, float weight, int depth,
                    const ray& r ) const;

                    // найти цвет в точке
    glm::vec3 shade ( const Medium& curMedium, float weight, int depth,
                    const glm::vec3& p, const glm::vec3& v,
                    HitData& hit ) const;

    static Scene * scene;
    static int        level;            // для статистики
    static int        rayCount;
};
```

Объекты, которые можно будет трассировать, будут представлены экземплярами классов, унаследованных от класса `Object`.

```
class Object
{
public:
    SurfaceData defSurface;

    Object () {}
    virtual ~Object () {}

            // проверить на пересечение с лучом
            // в hit кешируются внутренние параметры
    virtual bool intersect ( const ray& r, HitData& hit ) const = 0;

            // найти свойства поверхности в заданной точке
    virtual void getSurface ( HitData& hit,
                              SurfaceData& surface ) const = 0;
};
```

Как видно из описания, основным методом является метод `intersect`, задача которого – проверить луч на пересечение с объектом. В случае если пересече-

ние имеет место, то данные о пересечении заносятся в структуру HitData. В этой структуре содержится информация о пересечении. Кроме того, различные классы могут хранить в HitData какую-то дополнительную информацию (зависящую от конкретного класса объекта), которая будет использоваться потом для получения свойств поверхности в точке пересечения.

За это отвечает метод getSurface. Его задача – по HitData вернуть параметры поверхности, используемые для расчета освещенности. Все эти параметры хранятся в структуре SurfaceData.

```
struct HitData          // информация о пересечении с кешированными данными
{
      glm::vec3         pos;              // точка пересечения
      float             t;                // параметр вдоль луча
      const Object  * object;             // пересекаемый объект
      bool              invertNormal;     // нужно ли перевернуть нормаль
      float             cache [13];       // кешированные значения для getSurface
};
```

Ключевым методом для осуществления трассировки лучей является метод trace, осуществляющий трассировку заданного луча. Ниже приводится его реализация.

```
glm::vec3 Scene :: trace ( const Medium& curMedium, float weight,
                           int depth, const ray& r ) const
{
      Object    * obj = nullptr;
      glm::vec3   color;
      HitData     hit;

      rayCount ++;

      if ( ( obj = intersect ( r, hit ) ) != nullptr )   // есть пересечение
      {
            color = shade ( curMedium, weight, depth + 1,
                            r.pointAt ( hit.t ), r.getDir (), hit );

            if ( curMedium.extinct > EPS )
                  color *= expf ( -hit.t * curMedium.extinct );
      }
      else                      // пересечения нет, берем фон
            color = getBackground ( r );

      return color;
}
```

Для определения световой энергии, покидающей точку в заданном направлении, используется метод shade. Его реализация также приводится ниже.

```
glm::vec3 Scene :: shade ( const Medium& curMedium, float weight, int depth,
                           const glm::vec3& p, const glm::vec3& v,
                           HitData& hit ) const
{
      SurfaceData    txt;
```

```
glm::vec3      l;                        // вектор на источник света
bool           entering = true;          // мы входим в объект или выходим из него
float          shadow;

hit.object -> getSurface ( hit, txt );   // получаем свойства поверхности в точке

if ( hit.invertNormal )
      txt.n = -txt.n;

float            vn = glm::dot ( v, txt.n );

if ( vn  > 0 )                           // обеспечиваем  ( -v, n ) > 0
{
      txt.n    = -txt.n;
      vn       = -vn;
      entering = false;
}

glm::vec3      color ( ambient * txt.color * txt.ka );   // фоновое освещение
for ( auto it : lights )
      if ( ( shadow = it -> shadow ( hit.pos, l ) ) > EPS )
      {
            float      ln = glm::dot ( l, txt.n );

            if ( ln > EPS )               // источник света виден
            {
                  if ( txt.kd > EPS )     // считаем диффузную освещенность
                        color += it -> color * txt.color *
                                 (txt.kd * shadow * ln);

                  if ( txt.ks > EPS )     // считаем бликовое освещение
                  {
                        glm::vec3   h = glm::normalize ( l - v );

                        color += it -> color * (txt.ks * shadow *
                                 pow ( std::max ( 0.0f,
                                 glm::dot ( txt.n, h ) ), txt.p ));
                  }
            }
      }

float      rWeight = weight * txt.kr;      // вклад отраженного луча
float      tWeight = weight * txt.kt;      // вклад преломленного луча
if ( rWeight > EPS && depth < MAX_DEPTH )  // отраженный луч
{
      glm::vec3   r  = v - txt.n * (2 * vn);

      color  += txt.kr * trace ( curMedium, rWeight, depth + 1,
                                 ray ( hit.pos, r ) );
}
if ( tWeight > EPS && depth < MAX_DEPTH )  // преломленный луч
{
```

```
            float eta  = curMedium.nRefr / (entering ? txt.medium.nRefr : 1);
            float ci   = -vn;
            float ctSq = 1 + eta*eta*( ci*ci - 1 );
            if ( ctSq > EPS )                    // неполное внутреннее отражение
            {
                  glm::vec3   t = v*eta + txt.n*( eta*ci - sqrtf ( ctSq ) );

                  if ( entering )
                        color += txt.kt * trace ( txt.medium, tWeight,
                                 depth + 1, ray ( hit.pos, t ) );
                  else
                        color += txt.kt * trace ( air, tWeight, depth + 1,
                                 ray ( hit.pos, t ) );
            }
      }

      return color;
}
```

Для построения итогового изображения мы будем использовать класс `Camera`, моделирующий наблюдателя, и уже знакомый класс `TgaImage` для хранения получающегося изображения.

```
class Camera
{
      glm::vec3   pos;               // положение камеры
      glm::vec3   view, right, up;   // базис камеры
      int         width, height;     // ширина и высота в пикселах
      float       fovx, fovy;        // углы обзора по x и по y
      TgaImage  * image;

public:
      Camera ( const glm::vec3& p, const glm::vec3& v, const glm::vec3& u,
               float fx, int w, int h );
      ~Camera ();

      int      getWidth () const
      {
            return width;
      }

      int      getHeight () const
      {
            return height;
      }

      float      getAspect () const
      {
            return (float) width / (float) height;
      }

      bool traceFrame            ( const char * fileName ) const;
      bool traceFrameDistributed ( const char * fileName, int n1,
                                   int n2 ) const;
};
```

Также нам понадобятся некоторые базовые объекты, из которых мы будем строить сцены для рендеринга. Простейшим таким объектом будет обычная сфера.

```
class Sphere : public Object
{
      glm::vec3   center;
      float       radius, radiusSq;

public:
      Sphere ( const glm::vec3& c, float r );

                  // проверить на пересечение, параметры пересечения занести в hit
      virtual bool intersect ( const ray& r, HitData& hit ) const;

                  // получить свойства поверхности в точке
      virtual void getSurface ( HitData& hit, SurfaceData& surface ) const;
};
```

Еще одним используемым классом будет обычная треугольная грань, реализация соответствующего класса приводится ниже.

```
class Tri : public Object
{
      glm::vec3   p [3];;
      glm::vec3   normal;

public:
      Tri ( const glm::vec3& a, const glm::vec3& b, const glm::vec3& c );

      virtual bool intersect ( const ray& r, HitData& hit ) const;

      virtual void getSurface ( HitData& hit, SurfaceData& surface ) const;
};
```

В качестве последнего класса объектов, который мы рассмотрим в этой главе, будет плоскость.

```
class Plane : public Object
{
      glm::vec3   n;            // уравнение плоскости(p,n) + d = 0
      float       d;

public:
      Plane ( const glm::vec3& nn, float dist ) : Object (), n ( nn ),
                                                  d ( dist ) {}

      virtual bool intersect ( const ray& r, HitData& hit ) const;

      virtual void getSurface ( HitData& hit, SurfaceData& surface ) const;
};
```

Ниже приводится изображение, получаемое при помощи трассировщика.

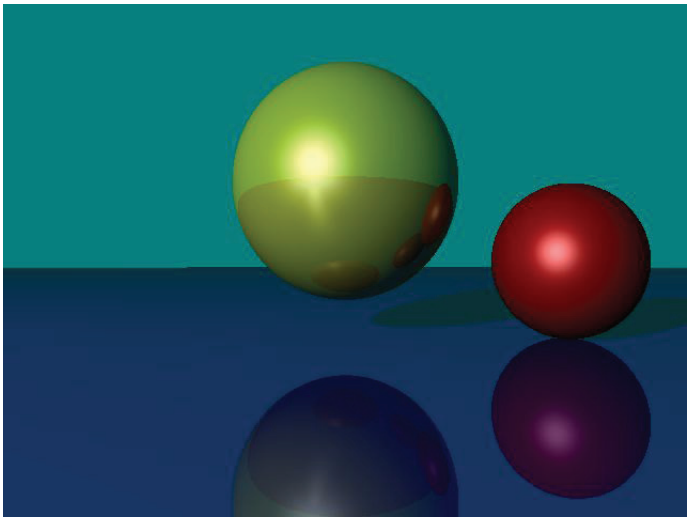

Рис. 10.4 ❖ Изображение,
получаемое при помощи трассировщика лучей

На следующем рисунке приведен результат трассировки набора шаров с изменяющимися параметрами материала, включая прозрачность. Здесь используется модель освещения Кука–Торранса. Соответствующий код здесь не приводится, но он есть в репозитории **https://github.com/steps3d/graphics-book.git**.

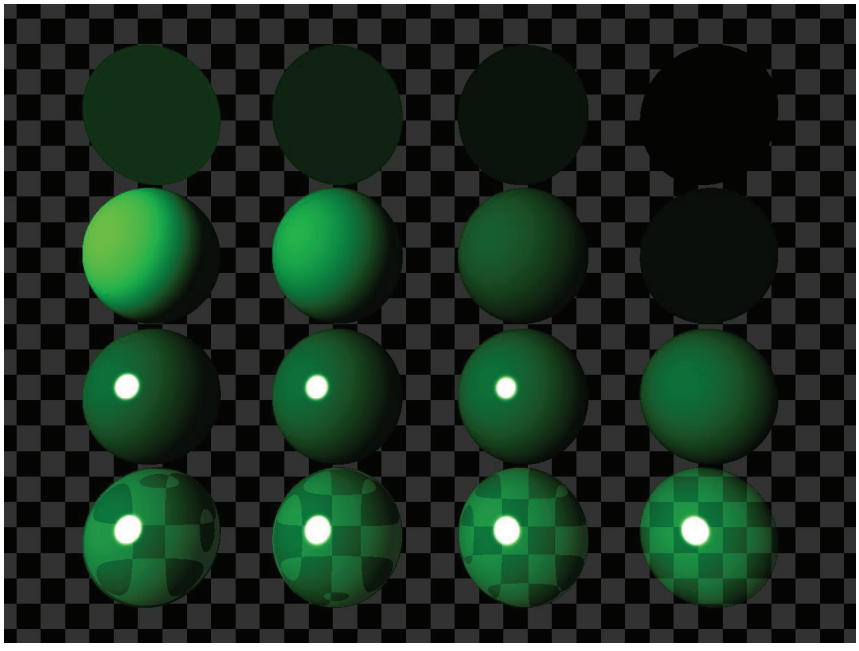

Рис. 10.5 ❖ Набор сфер

Constructive Solid Geometry

Одним из красивых и в то же время простых способов создания сложной геометрии на базе простых объектов является CSG (Constructive Solid Geometry). Этот подход использует операции объединения, пересечения и вычитания объектов для создания сложных объектов из простых.

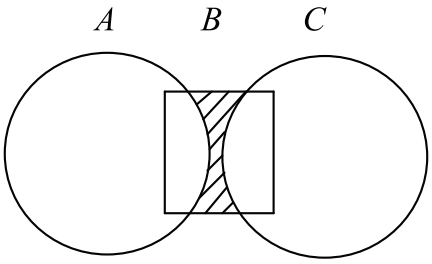

Рис. 10.6 ❖ Пример CSG-объекта

Так, на рис. 10.6 приведен пример создания сложного объекта из простых. Для получения нужного объекта (двояковогнутой линзы) мы вычитаем из цилиндра (*B*) две сферы (*A* и *C*). Тем самым результирующий объект может быть записан при помощи следующей формулы:

$D = B - A - C$.

Любой CSG-объект можно легко представить в виде бинарного дерева, так, приведенной выше формуле соответствует следующее дерево (рис. 10.7).

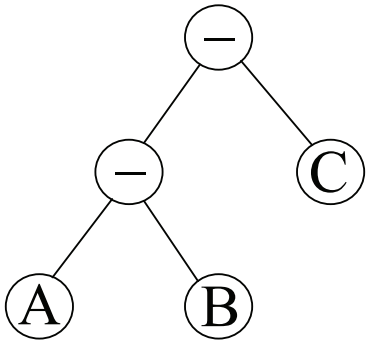

Рис. 10.7 ❖ Бинарное дерево для объекта с рис. 10.3

Если объекты заданы в полигональном виде, т. е. виде граней, то можно получить точное представление результата CSG-операций, но это довольно затруднительно.

Однако трассировку таких объектов можно реализовать довольно просто. Для этого достаточно добавить новый метод в базовый класс `Object`. Этот метод должен будет для заданного луча возвращать не только ближайшую точку пересечения, но и полное пересечение луча с объектов. В нашем случае таким пересечением будет набор отрезков вдоль луча.

Каждый такой отрезок включает в себя два значения параметра *t* – точку входа и точку выхода. Кроме того, для каждой такой точки пересечения нужно задать указатели на соответствующие этим точкам пересечения объекты (они могут не совпадать) и признак инвертирования нормали.

```
struct RaySegment                  // отрезок луча, пересекающий объект
{                                  // от входа (0) до выхода (1)
      float          t      [2];   // параметр вдоль луча, t[0] <= t[1]
      const Object * obj    [2];   // объекты для каждого конца
      bool           invert [2];   // нужно ли инвертировать нормаль
      HitData        hit    [2];

      float start () const
      {
            return t [0];
      }

      float end () const
      {
            return t [1];
      }

      bool empty () const
      {
            return start () >= end ();
      }

      bool intersects ( const RaySegment& s ) const
      {
            return std::max ( start (), s.start () ) <
                   std::min ( end (), s.end () );
      }

      void copyPoint ( const RaySegment& s, int from, int to, bool inv )
      {
            assert ( from >= 0 && from < 2 );
            assert ( to   >= 0 && to   < 2 );

            t      [to] = s.t      [from];
            obj    [to] = s.obj    [from];
            invert [to] = s.invert [from];
            hit    [to] = s.hit    [from];

            if ( inv )
                  invert [to] = !invert [to];
      }

      void intersect ( const RaySegment& s );
      void merge     ( const RaySegment& s );
};
```

Для того чтобы понять, зачем нам нужны дополнительный указатель на объект и булевый флаг, давайте рассмотрим объект с рис. 10.8. Этот объект задается

формулой $A - B$, где A и B – это сферы. В одной точке происходит пересечение с объектом A, и мы берем нормаль от объекта A.

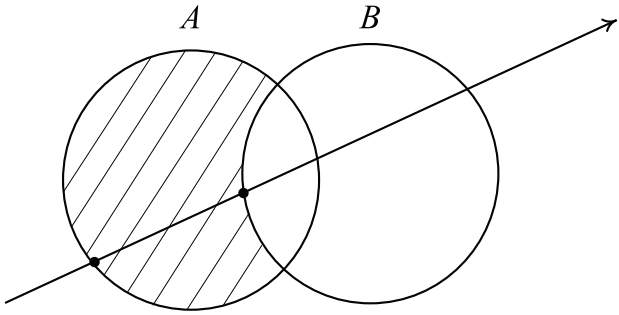

Рис. 10.8 ❖ Пересечение луча с разностью двух сфер

Но для второй точки мы берем нормаль уже от объекта B и инвертируем ее (т. е. умножаем на –1). Инвертирование нормали связано с тем, что объект B мы *вычитаем*. При вычитании всегда происходит переворачивание нормали, если точка пересечения принадлежит вычитаемому объекту.

Тогда полное пересечение объекта с лучом всегда будет просто отсортированным множеством структур `RaySegment`. Для хранения таких структур мы будем использовать класс `std::vector<RaySegment>`.

Пусть мы знаем полное пересечение объектов A и B с лучом r. Тогда справедливы следующие формулы:

$(A \cup B) \cap r = (A \cap r) \cup (B \cap r)$;
$(A \cap B) \cap r = (A \cap r) \cap (B \cap r)$;
$(A - B) \cap r = (A \cap r) - (B \cap r)$.

Тем самым нам достаточно просто научиться объединять, пересекать и вычитать множества одномерных отрезков друг из друга. Обратите внимание, что при этом может происходить слияние, разбиение и пропадание отдельных отрезков. Ниже приводится код для реализации одной из этих трех операций – операции пересечения.

```
void csgAnd ( const std::vector<RaySegment>& a,
              const std::vector<RaySegment>& b,
              std::vector<RaySegment>& c )
{
      auto ia = a.begin ();
      auto ib = b.begin ();

      c.clear ();

      if ( a.empty() || b.empty() )
            return;

      for ( ; ; )
      {
            if ( ib != b.end () )
```

```
            while ( ia != a.end () && ia->end () < ib->start () )
                  ia++;

      if ( ia != a.end () )
            while ( ib != b.end () && ib->end () < ia->start () )
                  ib++;

      if ( ia == a.end () || ib == b.end () )
            return;

      assert ( ia -> intersects ( *ib ) );

      RaySegment s = *ia++;

      while ( ia != b.end () && s.intersects ( *ib ) )
      {
            RaySegment s1 = s;

            s1.intersect ( *ib++ );

            if ( !s1.empty () )
                  c.push_back  ( s1 );
      }
   }
}
```

Теперь создадим новый класс для работы с CSG. Его главным методом будет метод, возвращающий список всех пересечений луча с объектом, – `allIntersections`. Ниже приводится его описание.

```
class CsgObject : public Object
{
public:
   CsgObject () : Object () {}

            // возвращает все пересечения луча, false - если их нет
   virtual bool allIntersections ( const ray& r,
                              std::vector<RaySegment>& res ) const = 0;

   virtual bool intersect ( const ray& r, HitData& hit ) const
   {
      std::vector<RaySegment> out;

      if ( !allIntersections ( r, out ) )
            return false;

      if ( out.empty () )
            return false;

      if ( out [0].t [0] > EPS )
      {
            hit = out [0].hit [0];

            return true;
      }

      hit = out [0].hit [1];

      return true;
   }
};
```

После этого мы можем ввести класс CompositeObject, служащий для представления результатов CSG-операций над базовыми объектами, следующим образом:

```
class CompositeObject : public CsgObject
{
public:
      enum Operation
      {
            OR,
            AND,
            SUB
      };

protected:
      CsgObject * op1;                  // 1-й операнд
      CsgObject * op2;                  // 2-й операнд
      Operation   op;

public:
      CompositeObject  ( CsgObject * a, CsgObject * b, Operation oper );
      ~CompositeObject ();

                  // найти все пересечения луча или вернуть false
      virtual bool allIntersections ( const ray& r,
                                      std::vector<RaySegment>& res ) const;

                  // проверить луч на пересечение
      virtual bool intersect ( const ray& r, HitData& hit ) const;

                  // найти параметры поверхности в точке
      virtual void getSurface ( HitData& hit, SurfaceData& surface ) const;
};
```

На рис. 10.9 приводится изображение, полученное путем трассировки разности двух сфер.

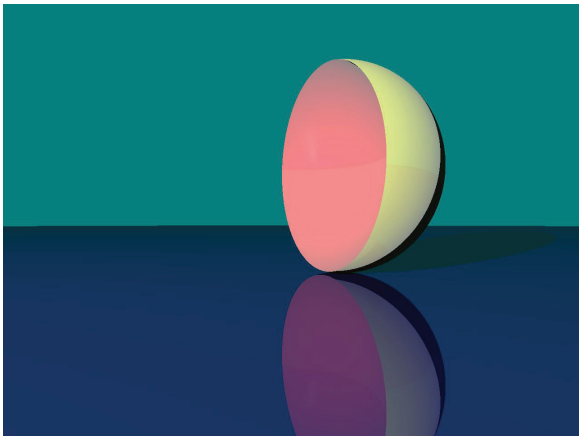

Рис. 10.9 ❖ Результат трассировки CSG-объекта

Ниже приводится код, строящий соответствующий CSG-объект.

```
Scene           * scene = new Scene;
CsgSphere       * s1    = new CsgSphere ( glm::vec3 ( -1, 1, 10 ), 2 );
CsgSphere       * s2    = new CsgSphere ( glm::vec3 (  0, 1,  9.5 ), 2.1 );
CompositeObject * obj   = new CompositeObject ( s1, s2, CompositeObject::SUB );
Plane           * p1    = new Plane  ( glm::vec3 ( 0, 1, 0 ), 1 );
PointLight      * l1    = new PointLight ( glm::vec3 ( 10, 5, -10 ), glm::vec3 ( 1 ) );
Camera            camera ( glm::vec3 ( 0 ), glm::vec3 ( 0, 0, 1 ),
                           glm::vec3 ( 0, 1, 0 ), 60, 640*2, 480*2 );
```

Распределенная трассировка лучей

К сожалению, трассировке лучей, как и всем алгоритмам, основанным на регулярной сетке, свойственны так называемые *проблемы дискретизации* (aliasing artifacts).

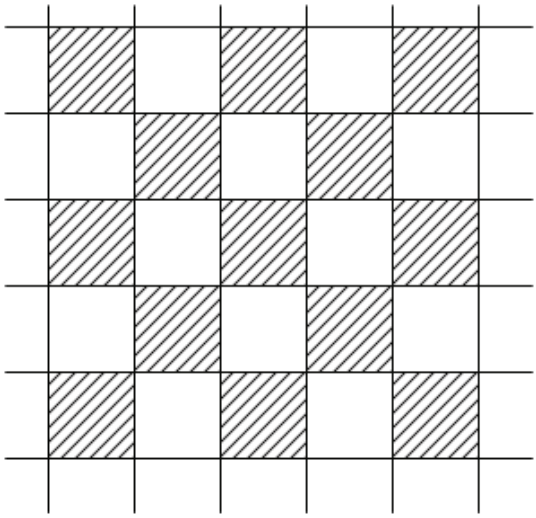

Рис. 10.10 ❖ Клеточный шаблон

В качестве простейшего примера подобных погрешностей может выступать плоскость с текстурой в виде шахматной доски из чередующихся черных и белых квадратов (рис. 10.10). Если мы эту плоскость поместим достаточно далеко от камеры (так, чтобы проекция одной клетки стала заметно меньше одного пиксела), то цвет пиксела будет определяться только тем, в какую именно клетку – черную или белую – попадет луч. А это будет определяться крайне небольшими численными погрешностями. В результате мы получим просто случайный набор черных и белых точек.

Это связано с тем, что мы трактуем каждый пиксел как бесконечно малую точку, в то время как на самом деле это некоторая малая прямоугольная область S на картинной плоскости. Тогда если через $I(x, y)$ обозначить цвет, получаемый при трассировке луча, проходящего через точку (x, y) на картинной плоскости, то соответствующий пикселу цвет будет задаваться следующей формулой:

$$I = \int_S I(x,y)dxdy.$$

Понятно, что подобный интеграл аналитически не взять. Кроме того, подинтегральная функция слишком сложна и нерегулярна для использования обычных методов интегрирования.

Поэтому для нахождения подобных интегралов обычно используется метод Монте-Карло. Данный метод основан на применении случайных величин и приближенного нахождения математического ожидания (среднего значения) заданной случайной величины. Фактически мы просто заменяем нахождение сложного интеграла на нахождение среднего значения специальной случайной величины.

Итак, мы в области S, соответствующей пикселу, выбираем случайную точку ξ с равномерным распределением в S, после чего трассируем луч через соответствующую точку и находим проходящую вдоль этого энергию.

Выпустив несколько лучей, мы просто усредняем проходящую вдоль них световую энергию. Подобный подход получил название *распределенной трассировки лучей* (distributed ray tracing).

Теперь мы рассмотрим, каким именно образом следует строить равномерно распределенные случайные величины. Обычно используются так называемые генераторы псевдослучайных чисел – алгоритмы, генерирующие последовательности чисел, обладающих требуемыми статистическими свойствами (но при этом эти числа строго детерминированы и на самом деле не являются случайными).

Простейшим способом получения таких чисел является использование стандартной функции `rand`, генерирующей псевдослучайные целые числа от нуля до `RAND_MAX`. Таким образом, для получения псевдослучайных чисел в [0, 1] мы можем использовать следующую функцию:

```
float randUniform ()
{
      return rand () / (float) RAND_MAX;
}

float randUniform ( float a, float b )
{
      return a + randUniform () * (b - a);
}
```

На основании этой функции можно получить функции для получения других случайных чисел. Ниже приводится функция для получения псевдослучайных точек на поверхности единичной сферы.

```
glm::vec3   randOnSphere ()
{
      float t = randUniform ( 0, 2*M_PI );
      float z = randUniform ( -1, 1 );
      float r = sqrt ( 1 - z*z );

      return glm::vec3 ( r * cos ( t ), r * sin ( t ), z );

}
```

Самый простой способ генерации необходимого числа случайных точек заключается просто в многократном вызове функции, генерирующей случайные точки. К сожалению, этот метод не всегда генерирует достаточно хорошие распределения.

Поэтому на практике обычно используется другой подход – в области S вводится равномерная сетка, и после этого узлы этой сетки подвергаются случайным «шевелениям» (jitter). Ниже приводится код метода `Camera::traceFrameDistributed`.

```
bool Camera :: traceFrameDistributed ( const char * fileName, int n1, int n2 ) const
{
      auto  start      = std::chrono::steady_clock::now ();
      float tfx        = tanf ( fovx * M_PI / 180 / 2 );      // tan (fovx/2)
      float tfy        = tanf ( fovy * M_PI / 180 / 2 );      // tan (fovy/2)
      float hx         = tfx * 2 / (float)width;
      float hy         = tfx * 2 / (float)width;
      float hxSub      = hx / n1;
      float hySub      = hy / n2;
      int   numSamples = n1 * n2;

      for ( int i = 0; i < width; i++ )
            for ( int j = 0; j < height; j++ )
            {
                  float       x = (i - width/2)  * hx;
                  float       y = -(j - height/2) * hy;
                  float       x1 = x - 0.5f * hx;
                  float       y1 = y - 0.5f * hy;
                  glm::vec3   c ( 0 );

                  for ( int iSub = 0; iSub < n1; iSub++ )
                        for ( int jSub = 0; jSub < n2; jSub++ )
                        {
                              float xt = x1 + hxSub*(iSub + randUniform() );
                              float yt = y1 + hySub*(jSub + randUniform() );

                              ray         r ( pos, glm::normalize (view +
                                              xt*right + yt*up) );

                              c += Scene::scene -> trace ( air, 1, 1, r );
                        }

                  c /= numSamples;

                  uint32_t red   = (uint32_t)(255.0f*std::min(1.0f, c.x ));
                  uint32_t green = (uint32_t)(255.0f*std::min(1.0f, c.y ));
                  uint32_t blue  = (uint32_t)(255.0f*std::min(1.0f, c.z ));

                  image -> putPixel ( i, j, image -> rgbToInt ( red, green, blue ) );
            }
      image -> writeToFile ( fileName );

      return true;
}
```

Реализация спецэффектов при помощи распределенной трассировки лучей

Используя метод Монте-Карло, можно отказаться от ряда ограничений, которые мы ранее накладывали. Одним из таких ограничений было то, что мы рассматривали только точечные источники света.

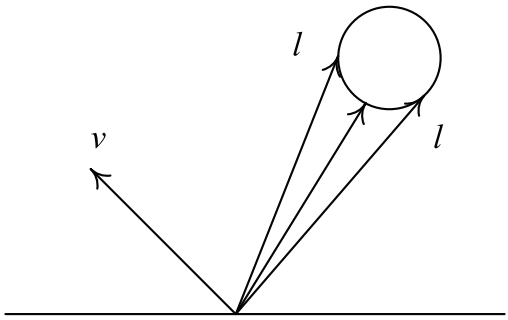

Рис. 10.11 ❖ Неточечный источник света

Пусть у нас есть неточечный источник света (рис. 10.11). Тогда для получения доли световой энергии, отраженной из точки P в заданном направлении v, нам нужно найти интеграл по видимой из точки поверхности источника:

$$I = \int_S f_r \max(0,(n,l))dl.$$

Именно это и позволяет получать для неточечных источников света мягкие полутени – когда лишь некоторые лучи к источнику света оказываются перекрытыми геометрией сцены (рис. 10.12).

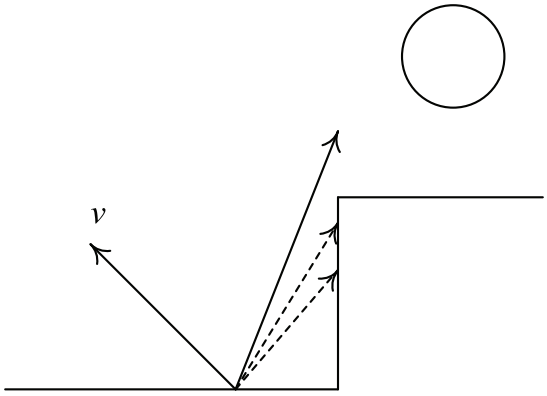

Рис. 10.12 ❖ Мягкие тени

Для реализации этого эффекта выберем на поверхности источника света набор случайных точек. Для каждой из этих точек построим свой вектор l к источнику света и определим даваемую им световую энергию (с учетом возможного закрывания источника света). После этого просто определим среднее значение по всем этим точкам – оно и будет освещенностью данной точки поверхности.

Следующим, легкореализуемым эффектом являются *нечеткие отражения*. Ранее мы строили лишь один отраженный луч с направляющим вектором r. На самом деле в качестве вектора r может выступать любой единичный вектор из верхней полусферы (т. е. такой, что $(r, n) > 0$). И вся световая энергия, приходящая вдоль соответствующего луча, получает вес, задаваемый *BRDF*. Тем самым мы фактически интегрируем I_r по всем подходящим векторам r.

Как и ранее, мы выбираем несколько случайных единичных векторов *r*, строим соответствующие отраженные лучи и определяем приходящую вдоль них световую энергию. Далее для каждого луча энергия умножается на *BRDF* и усредняется. В результате мы действительно получим нечеткие отражения.

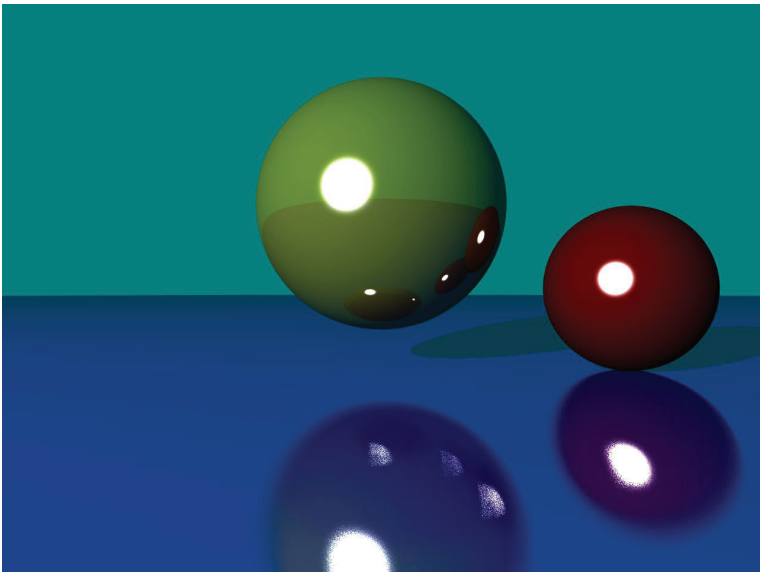

Рис. 10.13 ❖ Пример изображения с учетом нечеткого отражения

Ниже приводится реализация расчета нечеткого отражения, использованная для получения приведенного выше изображения.

```
if ( rWeight > EPS && depth < MAX_DEPTH )
{                              // направление идеального отражения
      glm::vec3   r  = v - txt.n * (2 * vn);
      glm::vec3   c ( 0 );     // средний цвет
                               // выбираем число лучей в зависимости от неровности поверхности
      int         numSamples = (int)std::max ( 1.0f, std::min ( 10.0f,
                                          txt.glossiness * 20.0f ) );
      float       sum        = 0;
                               // трассируем лучи для моделирования нечеткого отражения
      for ( int i = 0; i < numSamples; i++ )
      {
            glm::vec3   rNew = glm::normalize ( r + 0.3f * txt.glossiness *
                                          getRandomVector () );
            float       w    = powf ( glm::dot ( r, rNew ), 1 +
                                    txt.glossiness * 15.0f );

            c   += w * trace ( curMedium, rWeight, depth + 1, ray ( hit.pos, rNew ) );
            sum += w;
      }

      color += txt.kr * (c / sum);
}
```

Последним из рассматриваемых эффектов будет эффект *глубины резкости* (depth of field). Если рассматривать любой реально существующий оптический прибор (например, цифровую камеру), то он может одновременно удерживать в фокусе только те объекты, которые находятся на определенном расстоянии.

Человеческий глаз не является исключением. Мы просто не замечаем этого, поскольку как только мы перемещаем свое внимание на какой-то объект, глаз автоматически подстраивается под него, так что этот объект сразу же оказывается в фокусе.

Та модель камеры, которую мы использовали ранее, соответствует объективу нулевого диаметра – только в этом случае можно одновременно удерживать все объекты в фокусе.

Давайте рассмотрим, из-за чего возникает эффект глубины резкости.

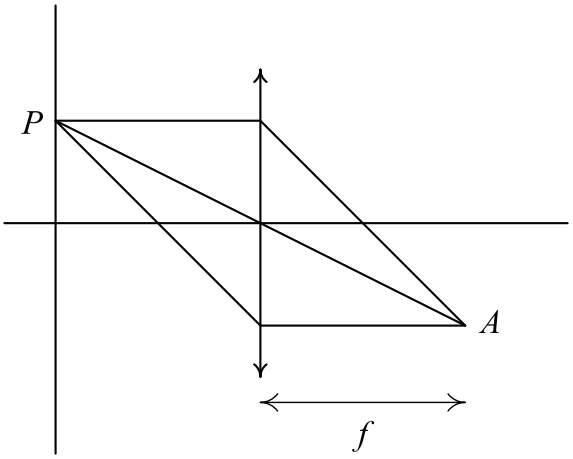

Рис. 10.14 ❖ Физика глубины резкости

Пусть у нас есть некоторая точка P (рис. 10.14). Выберем случайную точку R на линзе и проведем через эти точки луч. В силу законов геометрической оптики этот луч в точке R преломится и пройдет через точку A. Если у нас точка P имеет координаты $(-d, y)$, а точка R – $(0, y_0)$, то тогда точка A будет иметь координаты $\left(f, -y\frac{f}{d}\right)$. При этом величины f и d связаны следующим соотношением (здесь через F обозначено фокусное расстояние линзы):

$$\frac{1}{F}=\frac{1}{f}+\frac{1}{d}.$$

Отсюда мы получаем направляющий вектор v для выходящего из R луча:

$$v=\left(f-y\frac{f}{d}-y_0\right).$$

Соответственно, для моделирования глубины резкости мы выбираем несколько случайных точек на линзе и выпустим луч через каждую из них. Оттрассировав получающиеся при этом лучи, мы, как и ранее, усредняем приносимую ими световую энергию.

Описанный выше алгоритм позволяет получать изображения очень высокого качества с учетом отражения и преломления света. Однако есть некоторые эффекты, которые обратная трассировка лучей не может правильно обработать. К из числу относится так называемая *каустика*: если у нас есть стеклянный объект (например, шар), то проходящий через него свет формирует определенный рисунок на плоскости позади этого объекта.

Можно модифицировать алгоритм трассировки лучей, для того чтобы корректно обрабатывать подобные явления, и мы это сейчас рассмотрим.

Фотонные карты

Для точного учета каустики и диффузного рассеивания света существует модификация метода трассировки лучей с использованием *фотонных карт* (photon maps). Получающийся при этом алгоритм является двухпроходным. На первом проходе мы используем прямую трассировку лучей для трассировки большого количества лучей (фотонов), выпускаемых из источников света.

Каждый такой фотон трассируется до попадания на диффузную поверхность. После чего мы запоминаем координаты точки попадания, направляющий вектор луча и энергию фотона. Для того чтобы мы потом могли легко находить эти точки, на втором проходе они обычно организуются в пространственный индекс, например *kD*-дерево.

Исходя из мощности источника света, определяется число выпускаемых из него фотонов. При этом мощность источника поровну делится между всеми выпускаемыми из него фотонами. Если у нас есть точечный источник света, то в качестве направления выпущенного фотона используется просто равномерно распределенный единичный случайный вектор.

Далее каждый из выпущенных фотонов трассируется до его попадания на диффузную поверхность. При попадании на недиффузную поверхность происходит отражение, преломление или поглощение.

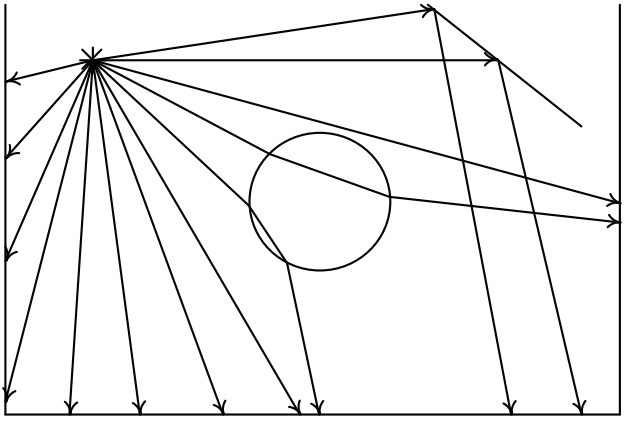

Рис. 10.15 ❖ Построение фотонной карты

Вместо того чтобы «разбивать фотон на части», обычно используется метод «русской рулетки». Мы просто отражаем, преломляем или поглощаем данный фотон

с определенной вероятностью. В результате не происходит увеличения общего числа фотонов. За счет этого мы стремимся к получению везде одинаковой точности – там, где больше освещено, будет больше фотонов, чем там, где освещено слабее.

Чтобы показать, как это работает, сначала мы рассмотрим более простой случай. Пусть у нас нет цветов (только оттенки серого) и у поверхности в точке попадания диффузный коэффициент равен d, а бликовый – s. При этом мы считаем, что выполняется закон сохранения энергии, т. е. $d + s \leq 1$.

Тогда для определения дальнейшей судьбы фотона мы разыгрываем случайную величину ξ, равномерно распределенную на отрезке [0, 1]. В случае $\xi \in [0, d]$ мы будем считать, что имеет место диффузное отражение. В случае $\xi \in [d, s + d)$ мы будем считать, что имеет место бликовое отражение. Иначе мы считаем, что имеет место поглощение фотона. Обратите внимание, что в этом упрощенном случае не происходит изменение энергии самого фотона.

Теперь давайте рассмотрим общий случай – коэффициенты диффузного и бликового отражения являются RGB-векторами (d_r, d_g, d_b) и (s_r, s_g, s_b) соответственно. Мы тогда можем принимать решения, исходя из различных критериев. Так, одним из вариантов является следующий:

$$P_r = \max(d_r + s_r, d_g + s_g, d_b + s_b).$$

Здесь P_r – это вероятность отражения (диффузного или бликового). Тогда вероятность поглощения фотона будет равна $P_0 = 1 - P_r$. Вероятность диффузного отражения задается следующей формулой:

$$P_d = \frac{d_r + d_g + d_b}{d_r + d_g + d_b + s_r + s_g + s_b} P_r.$$

Аналогично, вероятность бликового отражения будет равна:

$$P_s = \frac{s_r + s_g + s_b}{d_r + d_g + d_b + s_r + s_g + s_b} P_r.$$

Тогда для определения судьбы фотона, попавшего на поверхность, мы разыгрываем случайную величину ξ и в случае $\xi \in [0, P_d)$ считаем, что имеет место диффузное отражение. В случае $\xi \in [P_d, P_s + P_d)$ – бликовое, и в случае $\xi \in [P_d + P_s, 1]$ – поглощение.

Однако в этом случае необходимо откорректировать энергию фотона по следующей формуле:

$$P_{refl,r} = P_{inc,r} \frac{s_r}{P_s};$$

$$P_{refl,g} = P_{inc,g} \frac{s_g}{P_s};$$

$$P_{refl,b} = P_{inc,b} \frac{s_b}{P_s}.$$

Здесь через P_{inc} обозначена мощность падающего фотона, а через P_{refl} – отраженного. Аналогичным образом обрабатывается и преломление, нечеткое отражение и т. п.

Проводя подобную трассировку фотонов, мы строим так называемую фотонную карту – множество точек, где фотон попал на диффузную поверхность. Для каждой такой точки мы запоминаем ее координаты, мощность (RGB) и направление, откуда пришел фотон.

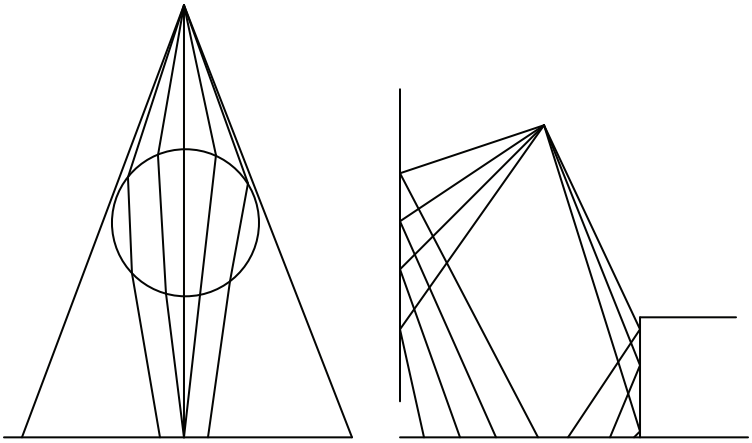

Рис. 10.16 ❖ Фотонные карты

Фактически мы рассматриваем каждый фотон, попавший на диффузную поверхность, как миниатюрный источник вторичного света. При выполнении второго прохода трассировки (обратная трассировка) мы будем искать фотоны рядом с точкой пересечения луча и учитывать их вклад в освещенность точки.

Обычно считается сразу две фотонные карты – одна для каустики (рис. 10.16 слева) и одна для всех остальных случаев (рис. 10.16 справа). Использование для каустики отдельной фотонной карты обычно связано с тем, что каустика требует большей точности и, следовательно, большего числа фотонов.

Второй проход заключается в обратной трассировке лучей, однако для получения падающего света мы используем построенные на первом проходе фотонные карты.

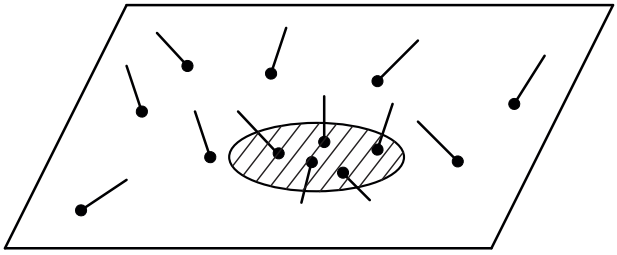

Рис. 10.17 ❖ Поиск фотонов в фотонной карте рядом с заданной точкой на поверхности

Для каждой точки пересечения луча с объектом мы находим ближайшие к ней точки попадания фотонов на поверхность. Как правило, мы просто ищем все точки попадания фотонов на расстоянии, не большем некоторого r.

Тогда отраженный в заданном направлении свет может быть представлен в виде следующей суммы (через $\Delta\Phi_i$ обозначена мощность i-го фотона):

$$L_r(x,\omega) \approx \frac{1}{\pi r^2}\sum_{i=1}^{N} f_r(x,\omega,\omega_i)\Delta\Phi_i(x,\omega_i).$$

Monte-Carlo path tracing

Еще одним способом получения фотореалистических изображений с учетом широкого класса различных эффектов является Monte-Carlo path tracing. Этот подход пытается честно посчитать соответствующий интеграл по сфере, учитывая при этом такие явления, как диффузное рассеивание света.

При этом сам базовый код для получения изображений крайне прост – простейший трассировщик путей `smallpt` состоит всего из 99 строк на C++.

Этот подход использует метод Монте-Карло для точного нахождения освещенности с учетом вторичных переотражений света. Учитываются такие эффекты, как диффузное переотражение света. Недостатком этого метода является очень высокая стоимость получения изображения – для получения качественных изображений требуются десятки, а иногда и сотни лучей на один пиксел, вплоть до многих тысяч лучей на пиксел.

Ниже приводится текст функции `tracePath` для простейшего трассировщика пути, обрабатывающего для простоты только диффузные поверхности.

```
glm::vec3 Scene::tracePath ( const ray& r, int depth ) const
{
      HitData     hit;
      SurfaceData surf;

      if ( !intersect ( r, hit ) )
            return getBackground ( r );

      if ( ++depth > MAX_DEPTH )
            return glm::vec3 ( 0 );

      hit.object -> getSurface ( hit, surf );

      glm::vec3   c = surf.color;
      glm::vec3   n = surf.n;
      float       p = std::max ( c.x, std::max ( c.y, c.z ) );

      if ( randUniform () < p )              // сохраняем луч
            c *= 1.0f / p;
      else
            return surf.emission;

      glm::vec3 dir = getRandomVector ();

                              // если луч не в верхней полусфере, инвертируем его направление
      if ( glm::dot ( n, dir ) < 0 )
            dir = -dir;

      glm::vec3   brdf = c / (float)M_PI;    // диффузная BRDF

      return surf.emission + brdf * 2.0f * M_PI * std::max (
                              glm::dot ( n, dir ), 0.0f ) *
                              tracePath ( ray ( hit.pos, dir ), depth );
}
```

Данная функция легко может быть изменена, чтобы добавить поддержку других типов поверхностей – тогда разыгрывается, какой именно луч мы выпускаем – случайный, отраженный или преломленный.

Для того чтобы получилось достаточно реалистическое изображение, нужно оттрассировать очень большое число лучей для каждого пиксела. На рис. 10.18 приводятся результаты рендеринга сцены с числом лучей на пиксел 25.

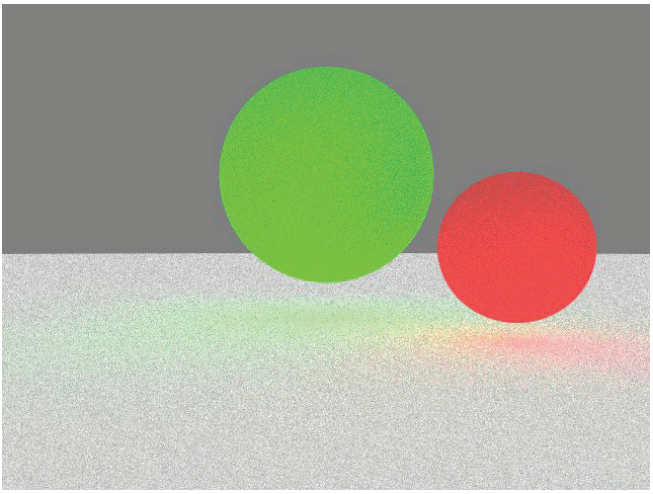

Рис. 10.18 ❖ Результат работы path tracing

Глава 11

Взаимодействие с оконной системой. Библиотеки freeglut и GLFW

В следующей главе мы начнем рассмотрение современного OpenGL (мы будем рассматривать версию 3.3 и выше). Библиотека OpenGL предназначена для рендеринга двух- и трехмерной графики. Она не содержит функций для создания окон, чтения пользовательского ввода от мыши и клавиатуры и других взаимодействий с оконной системой.

Поэтому если мы хотим писать полноценные приложения с использованием OpenGL, то нам нужна некоторая, желательно кроссплатформенная, библиотека, которая отвечала бы за все взаимодействие с оконной системой.

В этой главе мы рассмотрим две такие библиотеки – `freeglut` и `GLFW`. Кроме того, мы напишем простую обертку над `freeglut`, которую дальше будем постоянно использовать при написании приложений. Также в конце главы приводится пример работы с современным OpenGL при помощи библиотеки `Qt 5`.

Основы работы оконной системы

Существуют различные типы оконных систем, однако их все объединяют некоторые общие принципы и идеи.

В соответствии с этими принципами окно – это *объект*, который умеет реагировать на определенные поступающие *сообщения*. Конкретные типы сообщений и их параметры меняются в зависимости от оконной системы, однако есть некоторые стандартные типы сообщений, общие для всех оконных систем.

Самым главным таким сообщением является запрос окну, чтобы оно нарисовало в заданной области экрана свое содержимое. При этом сама оконная система устанавливает такую область отсечения, чтобы окно не могло ничего вывести на экран вне пределов своей видимой области.

Каждый раз, когда открывается ранее закрытая часть окна (или же все окно целиком), окну посылается запрос на вывод своего содержимого (при этом область отсечения становится равной открывшейся части окна).

Если необходимо изменить что-то в окне, то обычно просто посылается сообщение, чтобы оно перерисовало себя (или свою часть).

Обычно каждый раз, когда изменяется размер окна (а также когда окно создается), ему посылается сообщение об изменении его размера. При этом это сообщение всегда приходит до соответствующего ему запроса на перерисовку содержимого окна.

Также каждый ввод от пользователя, такой как перемещение мыши, нажатие или отпускание клавиш мыши или клавиатуры, также приводит к посылке окну соответствующих сообщений.

Все сообщения передаются окну не напрямую, а через *очередь сообщений* – при посылке окну сообщения оно просто добавляется в конец этой очереди. В каждом приложении есть *цикл обработки сообщений* – в цикле из этой очереди извлекается очередное сообщение и отправляется соответствующему окну для обработки.

Конкретная работа с оконной системой, используя стандартный API для этого (например, WinAPI), довольно громоздка и сильно зависит от конкретного API. Далее мы рассмотрим библиотеки `freeglut` и `GLFW`. Это кроссплатформенные библиотеки, задачей которых является создание окон для рендеринга в них при помощи OpenGL и обработка сообщений для этих окон.

Обе библиотеки представляют собой набор C-функций, имя каждой такой функции начинается со специального префикса (`glut` для библиотеки `freeglut`, `glfw` для библиотеки `GLFW`). Каждая из этих библиотек позволяет создавать окна для рендеринга. При этом создается *контекст* OpenGL, необходимый для дальнейшего использования OpenGL.

И можно явно задать требования к создаваемому контексту, что мы и будем дальше делать.

Вместо использования одной большой функции для обработки всех сообщений обе библиотеки позволяют задать для каждого окна свой обработчик для каждого из базовых типов сообщений.

Работа с библиотекой freeglut

Библиотека `freeglut` является opensource-версией библиотеки `GLUT`, на протяжении многих лет использующейся для создания небольших примеров и приложений для OpenGL. Она может быть скачана по адресу **http://freeglut.sourceforge.net**. Она полностью совместима с `GLUT`, но предоставляет целый ряд дополнительных возможностей. Она позволяет писать кроссплатформенный код, который будет легко компилироваться и работать на разных операционных системах (Windows, Linux и macOS).

Для использования этой библиотеки необходимо подключить заголовочный файл `freeglut.h` (а также соответствующий файл библиотеки).

```
#include      <GL/freeglut.h>
```

Инициализация

Следующим шагом является инициализация библиотеки. Ниже приводится пример этой инициализации.

```
glutInit ( &argc, argv );
```

Обратите внимание, что в функцию `glutInit` передается массив аргументов, заданных при запуске приложения, и адрес переменной, содержащей число этих ар-

гументов. Обычно argc и argv – это стандартные аргументы функции main. Передача этих аргументов связана с тем, что исторически можно передать в командной строке опции для GLUT. Тогда вызов glutInit обрабатывает эти опции и убирает их из списка аргументов.

Следующий шаг – необходимо задать, какие именно буферы OpenGL нам понадобятся. Подробнее о буферах в следующей главе, здесь мы просто перечислим соответствующие буферам константы:

- GLUT_RGB, GLUT_RGBA – обозначают буфер цвета;
- GLUT_DEPTH – буфер глубины;
- GLUT_STENCIL – буфер трафарета;
- GLUT_DOUBLE – задает использование *двойной буферизации*.

Каждая из этих констант представляет собой битовый флаг, эти флаги объединяются при помощи побитовой операции ИЛИ ("|").

Двойная буферизация необходима для получения *немерцающей* (flicker-free) анимации. Для того чтобы понять, зачем это нужно, давайте рассмотрим следующую ситуацию: нам нужно построить анимацию движения белого круга на черном фоне.

Без двойной буферизации мы рисуем непосредственно в тот самый буфер, который виден на экране. Соответственно, мы вначале очищаем буфер, заполняя его черным цветом, а потом поверх черного выводим белый круг.

Но при этом будет один короткий момент времени, когда белого круга еще не будет, а на его месте будет черный фон. И глаз человека в состоянии это заметить – то, что мы видим то белый круг, то снова черный фон на его месте. Создается эффект мерцания, неприятный для глаза.

Поэтому, чтобы избежать этого мерцания, обычно используют сразу два буфера – один отображается на экране, а в другой идет вывод. Когда изображение будет полностью готово, буферы меняются местами. Таким образом, мы всегда видим уже готовое изображение, и мерцания нет.

Ниже приводится пример инициализации с использованием двойной буферизации и буферов цвета и глубины:

```
glutInitDisplayMode ( GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE );
```

Поскольку в этой книге мы будем использовать OpenGL версии не ниже 3.3, то мы должны явно задать версию OpenGL, контекст для которой мы хотим получить.

Также мы должны задать тип контекста и профиль. Ниже приводится пример задания контекста для версии OpenGL 3.3 с удалением устаревшего функционала (GLUT_FORWARD_COMPATIBLE) и с поддержкой отладочной информации (GLUT_DEBUG):

```
glutInitContextVersion ( 3, 3 );      // требуем контекст для OpenGL 3.3
glutInitContextProfile ( GLUT_CORE_PROFILE );
glutInitContextFlags   ( GLUT_FORWARD_COMPATIBLE | GLUT_DEBUG );
```

Создание окна

Для создания окна необходимо задать координаты (в пикселах) его верхнего левого угла, его высоту и ширину (тоже в пикселах) и его заголовок.

По историческим причинам в GLUT (и, соответственно, в freeglut) задание положения и размеров окна разнесено в отдельные функции. Обратите внимание, что

как функция задания координат верхнего левого угла окна (glutInitWindowPos), так и функция задания размера окна (glutInitWindowSize) просто устанавливают значения соответствующих глобальных переменных. Соответственно, эти значения могут многократно переиспользоваться – при создании окна нужные значения просто берутся из этих глобальных переменных.

Ниже приводится пример создания окна с левым верхним углом в точке (300, 200) и размером 800×600.

```
glutInitWindowPosition ( 300, 200 );
glutInitWindowSize     ( 800, 600 );
glutCreateWindow       ( "Test window" );
```

Функция glutCreateWindow создает новое окно, создает для него контекст OpenGL, делает это окно текущим и возвращает его идентификатор (целое число).

Для того чтобы сделать окно с заданным идентификатором текущим, используется функция glutSetWindow:

```
void glutSetWindow ( int window );
```

Можно перевести текущее окно в состояние full-screen при помощи вызова glutFullScreen. Аналогично, full-screen окно можно снова сделать обычным при помощи вызова glutLeaveFullScreen.

```
void glutFullScreen      ();
void glutLeaveFullScreen ();
```

Поскольку контекст OpenGL создается при вызове glutCreateWindow, то команды OpenGL можно вызывать только после вызова glutCreateWindow.

Обработка сообщений

Самым первым сообщением, которое обычно устанавливается, является обработчик изменения размеров окна. Сам такой обработчик имеет следующий вид:

```
void reshape ( int width, int height )
{
      // обрабатываем изменение размера окна
}
```

Для того чтобы установить обработчик изменения размеров окна, для текущего окна используется функция glutReshapeFunc. Обратите внимание, что обработчик устанавливается только для текущего окна.

```
glutReshapeFunc ( reshape );
```

Следующим крайне важным обработчиком, который всегда нужно устанавливать, является обработчик запроса на перерисовку содержимого окна. Он будет вызываться каждый раз, когда нужно обновить содержимое окна. Этот обработчик имеет следующий вид:

```
void display ()
{
            // очищаем буферы цвета и глубины
      glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
      . . .
```

```
            // смена переднего и заднего буферов
      glutSwapBuffers ();
}
```

Обратите внимание, что всегда, когда изображение будет готово, необходимо вызвать функцию `glutSwapBuffers`, для того чтобы буферы, используемые при двойной буферизации, поменялись местами. Эту функцию можно вызывать даже в том случае, когда двойная буферизация не используется.

Также обратите внимание, что эту функцию может вызывать только `freeglut`. Если вам необходимо обновить содержимое окна, то следует вызвать функцию `glutPostRedisplay`. Этот вызов пошлет сообщение о перерисовке данному окну, которое приведет к последующему вызову обработчика перерисовки окна.

Для задания обработчика вывода содержимого для текущего окна служит функция `glutDisplayFunc`:

```
glutDisplayFunc ( display );
```

Следующим важным и часто используемым обработчиком является обработчик нажатий и отпусканий клавиш на клавиатуре. Библиотека `freeglut` (как и `GLUT`) различает клавиши – есть клавиши, нажатие которых генерирует ASCII-код (например, пробел, **Enter**, символьные клавиши), и есть клавиши, которые не генерируют ASCII-кодов (например, **Alt**, **Shift**, **F1**).

Соответственно, в `freeglut` для обработки нажатий и отпусканий клавиш используются четыре различных обработчика: два обработчика (нажатий и отпусканий) для клавиш с ASCII-кодами и два (нажатия и отпускания) – для клавиш без ASCII-кодов.

Первые два обработчика получают на вход ASCII-символ, соответствующий клавише, и координаты курсора мыши в пикселах в момент нажатия/отпускания клавиши.

```
void charPressed ( unsigned char ch, int x, int y )
{
      if ( ch == 'q' || ch == 'Q' || ch == 27 )     // выход по Esc или Q
            exit ( 0 );
}

void charReleased ( unsigned char ch, int x, int y )
{
      printf ( "Key %c released\n", ch );
}
```

Для установки этих обработчиков служат функции `glutKeyboardFunc` и `glutKeyboardUpFunc`:

```
glutKeyboardFunc   ( charPressed  );
glutKeyboardUpFunc ( charReleased );
```

Обработчики нажатий и отпусканий клавиш, не генерирующих ASCII-кодов (они называются специальными), для идентификации клавиш используют специальные константы. В число таких констант входят `GLUT_SHIFT`, `GLUT_KEY_F1`, `GLUT_INS` и т. д. Также каждый обработчик получает координаты курсора мыши в момент нажатия или отпускания соответствующей клавиши. Сами обработчики имеют следующий вид:

```
void specialKeyPressed ( int key, int x, int y )
{
      if ( key == GLUT_SHIFT )
            printf ( "SHIFT pressed\n");
}

void specialKeyReleased ( int key, int x, int y )
{
      if ( key == GLUT_SHIFT )
            printf ( "SHIFT released\n" );
}
```

Для задания этих обработчиков служат функции `glutSpecialFunc` и `glutSpecialUpFunc`:

```
glutSpecialFunc   ( specialKeyPressed  );
glutSpecialUpFunc ( specialKeyReleased );
```

Следующая группа обработчиков связана с мышью. Событиями, генерируемыми мышью, являются нажатия и отпускания кнопок мыши, перемещение мыши и поворот колесика мыши. При этом `freeglut` различает перемещение мыши при нажатой (любой) кнопке мыши и перемещение, когда ни одна из кнопок не является нажатой (так называемое *пассивное* перемещение).

Нажатия и отпускания кнопок мыши обрабатываются при помощи обработчика следующего вида:

```
void mouseButton ( int button, int state, int x, int y )
{
      if ( button == GLUT_RIGHT_BUTTON )
            if ( state == GLUT_DOWN )
                  printf ( "Right button down\n" );
            else
                  printf ( "Right button up\n" );
}
```

Параметр *button* задает кнопку мыши (`GLUT_LEFT_BUTTON`, `GLUT_RIGHT_BUTTON` и `GLUT_MIDDLE_BUTTON`). Второй параметр (*state*) сообщает, имеем ли мы дело с нажатием кнопки (`GLUT_DOWN`) или же с ее отпусканием (`GLUT_UP`). Параметры *x* и *y*, как и ранее, задают координаты курсора мыши. Задается этот обработчик при помощи вызова `glutMouseFunc`:

```
glutMouseFunc ( mouseButton );
```

Обработчики для активного и пассивного перемещений мыши выглядят одинаково и получают на вход только координаты курсора мыши:

```
void mouseMove ( int x, int y )
{
      . . .
}
```

Для задания этих обработчиков служат функции `glutMotionFunc` и `glutPassiveMotionFunc` соответственно.

```
glutMotionFunc        ( mouseMove );
glutPassiveMotionFunc ( mouseMove );
```

Также freeglut позволяет установить обработчик событий, связанных с вращением колесика мыши. Сам такой обработчик имеет следующий вид:

```
void mouseWheel ( int wheel, int dir, int x, int y )
{
     . . .
}
```

Параметры *x* и *y*, как и ранее, задают координаты курсора мыши в пикселах в момент поворота колесика. Параметр *wheel* задает номер колесика, вызвавшего событие, и параметр *dir* задает направление, в котором был совершен поворот. Для установки данного обработчика для текущего окна служит функция glutMouseWheelFunc:

```
glutMouseWheelFunc ( mouseWheel );
```

Еще один полезный и часто используемый обработчик устанавливается при помощи функции glutIdleFunc. Этот обработчик не получает никаких параметров и будет вызываться всякий раз, когда очередь сообщений оказывается пуста. Это происходит на самом деле очень часто и хорошо подходит для задания анимации – обычно в этом обработчике мы запрашиваем время, прошедшее с момента старта программы, и, исходя из этого времени, определяем значения анимируемых параметров. После чего вызывается glutPostRedisplay для запроса перерисовки окна.

```
void idle ()
{
          // получаем число время в секундах с запуска программы
     float t = 0.001f * glutGet ( GLUT_ELAPSED_TIME );

          // вычисляем параметры анимации
     . . .
          // запрашиваем перерисовку окна
     glutPostRedisplay ();
}
```

Несмотря на то что в библиотеке freeglut очень много полезных функций, мы рассмотрим здесь еще только две – glutMainLoop и glutLeaveMainLoop.

Первая из них (glutMainLoop) запускает бесконечный цикл обработки сообщений. Вторая (glutLeaveMainLoop) прерывает этот цикл и дает приложению шанс выполнить необходимую работу перед завершением приложения.

Заворачиваем freeglut в класс C++

Для удобства работы и сокращения размера кода мы в примерах к этой книге будем использовать специальный класс GlutWindow, инкапсулирующий всю работу по созданию окна, рендерингу и обработке сообщений.

В этом классе всем рассматриваемым обработчикам соответствуют виртуальные функции, которые можно переопределять в унаследованных классах. Ниже приводится пример приложения, использующего такой класс.

```
class TestWindow : public GlutWindow
{
```

```
public:
      TestWindow () : GlutWindow ( 100, 200, 500, 500, "Test" ) {}

      virtual     void  redisplay ()
      {
            glClear ( GL_COLOR_BUFFER_BIT );
      }
};

int main ( int argc, char * argv [] )
{
      GlutWindow::init ( argc, argv );

      TestWindow win;

      return win.run ();
}
```

Также можно запрашивать GLUT о значениях ряда параметров состояния при помощи вызова функции glutGet. Наиболее полезным таким параметром является время в миллисекундах, прошедшее с момента запуска программы. Оно возвращается при помощи вызова glutGet(GLUT_ELAPSED_TIME).

Работа с библиотекой GLFW

Еще одной библиотекой, предназначенной для создания окон и обработки ввода, является библиотека GLFW (она может быть скачана с сайта **http://www.glfw.org**).

Для использования этой библиотеки необходимо подключить заголовочный файл glfw3.h и соответствующий библиотечный файл.

```
#include <GLFW/glfw3.h>
```

Обратите внимание, что этот файл сам включает заголовочный файл gl.h для использования OpenGL. Также он определяет все необходимые определения из файла windows.h (для платформы Windows). Поэтому если нужно явно включить файл windows.h, то это нужно сделать до включения файла glfw3.h.

Инициализация и обработка ошибок

Для инициализации библиотеки GLFW служит функция glfwInit. В случае ошибки она возвращает значение GL_FALSE.

```
if ( !glfwInit () )
      exit ( 1 );
```

При завершении работы с GLFW следует вызвать функцию glfwTerminate, она закроет все открытые окна и освободит выделенные ресурсы.

Библиотека GLFW позволяет устанавливать функцию-обработчик, которая будет вызываться при возникновении различных ошибок. Эта функция получает на вход целочисленный код ошибки и строку с описанием ошибки. Ниже приводится пример такой функции.

```
void error ( int error, const char * description )
{
      fputs ( description, stderr );
}
```

Для установки этого обработчика служит вызов функции glfwSetErrorCallback:

```
glfwSetErrorCallback ( error );
```

Создание окна

Перед созданием окна при помощи функции glfwWindowHint можно задать ряд параметров для создаваемого окна. Ниже приводится пример для задания основных свойств по аналогии с библиотекой freeglut.

```
glfwWindowHint ( GLFW_RESIZABLE, 1 );     // можно менять размер окна мышью
glfwWindowHint ( GLFW_DOUBLEBUFFER, 1 );  // поддержка двойной буферизации
glfwWindowHint ( GLFW_DEPTH_BITS, 24 );   // 24-битовый буфер глубины
                                          // заказываем OpenGL 3.3, forward-compatible,
                                          // core profile
glfwWindowHint ( GLFW_CLEINT_API, GLFW_OPENGL_API );
glfwWindowHint ( GLFW_CONTEXT_VERSIONM_MAJOR, 3 );
glfwWindowHint ( GLFW_CONTEXT_VERSIONM_MINOR, 3 );
glfwWindowHint ( GLFW_OPENGL_FORWARD_COMPAT,  1 );
glfwWindowHint ( GLFW_OPENGL_PROFILE,         GLFW_CORE_PROFILE );
```

Непосредственно для создания окна служит функция glfwCreateWindow, возвращающая указатель на структуру GLFWwindow, используемую для идентификации окна.

```
GLFWwindow * window = glfwCreateWindow ( 640, 480, "Simple example",
                                         NULL, NULL );

if ( !window )
{
      glfwTerminate ();
      exit          ( 1 );
}
```

Для явного уничтожения окна служит функция glfwDestroyWindow:

```
void glfwDestroyWindow ( GLFWwindow * win );
```

При попытке пользователя закрыть окна у данного окна просто выставляется флаг о том, что данное окно должно быть закрыто. При помощи функции glfwWindowShouldClose можно проверить состояние этого флага у заданного окна.

```
int glfwWindowShouldClose ( GLFWwindow * win );
```

Также можно явно выставить данный флаг окну при помощи следующей функции.

```
void glfwSetWindowShouldClose ( GLFWwindow * win, int value );
```

При создании окна автоматически создается требуемый контекст OpenGL для него. Но перед тем, как можно будет вызывать команды OpenGL, нужно сделать контекст окна текущим при помощи функции glfwMakeContextCurrent.

```
void glfwMakeContextCurrent ( GLFWwinodw * win );
```

При помощи функции glfwSetWindowMonitor можно перевести окно в полноэкранный режим или вернуть из полноэкранного в обычный режим.

```
void glfwSetWindowMonitor ( GLFWwindow * win, GLFWmonitor * monitor,
                            int x, int y, int width, int height,
                            int refreshRate );
```

При задании параметра *monitor* не равным NULL окно становится полноэкранным с заданным разрешением. Если этот параметр равен NULL, то окно возвращается в обычный режим. Для получения адреса основного монитора можно использовать следующую функцию:

```
GLFWmonitor * glfwGetPrimaryMonitor ();
```

Обработка сообщений

Библиотека GLFW поддерживает все те же базовые обработчики, что и freeglut. При этом при задании обработчика мы явно задаем, для какого именно окна мы устанавливаем данный обработчик. Точно так же сам обработчик тоже получает на вход указатель на окно, для которого он был вызван. Ниже мы рассмотрим основные обработчики в GLFW.

Обработчик изменения размеров окна выглядит следующим образом:

```
void reshape ( GLFWwindow * win, int width, int height )
{
     . . .
}
```

Для установки этого обработчика служит функция glfwSetFramebufferSizeCallback:

```
glfwSetFramebufferSizeCallback ( win, reshape );
```

Обработчика запроса на вывод содержимого окна нет – в цикле сообщений в любой момент времени можно явно выполнять рендеринг в окно. Для переключения буферов при двойной буферизации служит функция glfwSwapBuffers.

```
void glfwSwapBuffers ( GLFWwindow * win );
```

Цикл обработки сообщений в библиотеке GLFW явный (в отличие от freeglut). Основной используемой в нем функцией является glfwPollEvents, которая извлекает из очереди все сообщения и передает их на обработку. Если в очереди нет ни одного сообщения, то управление из этой функции немедленно возвращается. Ниже приводится пример цикла обработки сообщений.

```
while ( !glfwWindowShouldClose ( window ) )
{
     display         ( window );            // выполнить требуемый рендеринг
     glfwSwapBuffers ( window );
     glfwPollEvents  ();
}
```

Как и в библиотеке freeglut, все сообщения от клавиатуры разделены на два класса. Первый класс связан с физическими клавишами и использует для идентификации клавиш специальные константы. Второй класс связан с генерацией ASCII-кодов.

Обработчик нажатий и отпусканий физических клавиш имеет следующий вид:

```
void key ( GLFWwindow * window, int key, int scancode, int action,
           int mods )
{
      if ( key == GLFW_KEY_ESCAPE && action == GLFW_PRESS )
            glfwSetWindowShouldClose ( window, GL_TRUE );
}
```

Параметр *key* задает конкретную клавишу на клавиатуре (например, GLFW_KEY_SPACE или GLFW_KEY_ESCAPE). Параметр action задает, была ли клавиша нажата (GLFW_PRESS), отпущена (GLFW_RELEASE) или же имеет место автоматическая генерация нажатий при удерживаемой клавише (GLFW_REPEAT).

Параметр *scancode* задает аппаратный код клавиши, но его значения зависят от используемой клавиатуры.

Параметр *mode* – это просто набор битовых флагов, задающий состояние клавиш-модификаторов на момент возникновения события (GLFW_MOD_SHIFT, GLFW_MOD_CONTROL и GLFW_MOD_ALT).

Данный обработчик устанавливается при помощи следующего вызова:

```
glfwSetKeyCallback ( window, key );
```

Обработчик символов (в UNICODE) имеет следующий вид:

```
void charCallback ( GLFWwindow * win, unsigned int ch )
{
      . . .
}
```

Для его установки используется функция glfwSetCharCallback:

```
glfwSetCharCallback ( win, charCallback );
```

Библиотека GLFW не различает активных и пассивных перемещений мыши – для всех перемещений мыши используется один обработчик следующего вида:

```
void moouseMove ( GLFWwindow * win, double x, double y )
{
      . . .
}
```

Для установки этого обработчика используется функция glfwSetCursorPosCallback.

Обработчик нажатий и отпусканий клавши мыши устанавливается при помощи функции glfwSetMouseButtonCallback и имеет следующий вид:

```
void mouseKey ( GLFWwindow * win, int button, int action, int mode )
{
      double x, y;

      glfwGetCursorPos ( win, &x, &y );     // получаем координаты курсора мыши

      if ( button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS )
            printf ( "Right button pressed\n" );
}
```

Также можно установить обработчик для колесика мыши при помощи функции glfwSetScrollCalback. Этот обработчик имеет следующий вид:

```
void scroll ( GLFWwindow * win, double xOffs, double yOffs )
{
      // обрабатываем вращение колесика

}
```

Также в библиотеке GLFW можно легко получить время в секундах, прошедшее с момента запуска программы при помощи функции glfwGetTime.

```
double time = glfwGetTime ();
```

Ниже приводится пример простейшего приложения с использованием библиотеки GLFW.

```
#include <GLFW/glfw3.h>
#include <stdlib.h>
#include <stdio.h>

void error ( int error, const char * description )
{
      fputs ( description, stderr );
}

void key ( GLFWwindow * window, int key, int scancode, int action, int mods )
{
      if ( key == GLFW_KEY_ESCAPE && action == GLFW_PRESS )
            glfwSetWindowShouldClose ( window, GL_TRUE );
}

void display ( GLFWwindow * window )
{
      int width, height;

      glfwGetFramebufferSize ( window, &width, &height );

      float ratio = width / (float) height;

      glViewport(0, 0, width, height);
      glClear(GL_COLOR_BUFFER_BIT);
}

int main ()
{
      glfwSetErrorCallback ( error );

      if ( !glfwInit() )
            exit ( 1 );

      GLFWwindow * window = glfwCreateWindow ( 640, 480, "Simple example",
                                               NULL, NULL );

      if ( !window )
      {
```

```
            glfwTerminate ();
            exit ( 1 );
      }

      glfwMakeContextCurrent ( window );
      glfwSwapInterval       ( 1 );
      glfwSetKeyCallback     ( window, key );

      while ( !glfwWindowShouldClose ( window ) )
      {
            display ( window );            // выполнить требуемый рендеринг
            glfwSwapBuffers ( window );
            glfwPollEvents  ();
      }

      glfwDestroyWindow ( window );
      glfwTerminate     ();

      return 0;
}
```

Пример работы с OpenGL при помощи библиотеки Qt 5

Библиотека Qt традиционно поддерживает работу с OpenGL. В версии Qt 5 появился новый класс виджета для работы с OpenGL и ряд вспомогательных классов для работы с шейдерами и т. п. Ниже приводится простейший пример, создающий контекст для OpenGL 3.3. Для начала нужно создать правильный класс, унаследовав его от класса QOpenGLQWidget:

```
#include <QOpenGLWidget>
#include <QOpenGLFunctions>

class MyGlWidget : public QOpenGLWidget, protected QOpenGLFunctions
{
      Q_OBJECT

public:
    MyGlWidget(QWidget *parent = 0) : QOpenGLWidget(parent) {}

protected:
    void initializeGL ()
    {
            initializeOpenGLFunctions ();
    }

    void resizeGL (int w, int h )
    {

    }

    void paintGL ()
    {
        glClear ( GL_COLOR_BUFFER_BIT );
    }
};
```

В основном приложении нужно не просто создать данный виджет, но и задать для него формат (версия OpenGL и т. д.).

```
int main ( int argc, char **argv )
{
    QApplication   app ( argc, argv );
    QSurfaceFormat format;
    MyGlWidget     widget;


    format.setRenderableType    ( QSurfaceFormat::OpenGL);
    format.setDepthBufferSize   ( 24 );
    format.setStencilBufferSize ( 8 );
    format.setVersion           ( 3, 3 );
    format.setProfile           ( QSurfaceFormat::CoreProfile );

    widget.setFormat ( format );
    widget.show      ();

    return app.exec();
}
```

Глава 12

Основы современного OpenGL

На данный момент OpenGL – это современный кроссплатформенный и мультивендорный API для работы с двух- и трехмерной графикой. Он существует уже более 20 лет (первая версия вышла в январе 1992 г.) и поддерживается практически на всех платформах – от мобильных устройств (где используется OpenGL ES) до мощных рабочих станций.

На OpenGL был написан целый ряд легендарных игр, таких как Quake III, Quake 4, Doom 3, Doom 4, Rage. Библиотеки для работы с OpenGL существуют практически для всех языков программирования, включая Java, C#, Python и многие другие.

В основе OpenGL лежит библиотека IRIS GL от компании SGI Inc. За время своего существования OpenGL прошел большой путь от использования фиксированного конвейера до работы с современными программируемыми GPU.

Первые версии OpenGL (1.x) были рассчитаны на работу с фиксированным конвейером и просто добавляли новые возможности к нему. Версия 2.0 принесла поддержку вершинных и фрагментных *шейдеров* – программ, выполняющихся непосредственно на GPU и отвечающих за обработку вершин и фрагментов. Но при этом была сохранена полная совместимость с версией 1.x, шейдеры были необязательными. Фактически весь старый код продолжает работать, и только в нужных местах можно было добавить шейдеры.

Версии 3.x привнесли очень важное изменение – некоторые функции и возможности (связанные с фиксированным конвейером) были объявлены *устаревшими* (deprecated). Несмотря на то что можно было получить контекст, в котором эти функции по-прежнему поддерживались, фокус сместился на использование новых, более эффективных функций. Также в основу был положен только программируемый конвейер рендеринга, и шейдеры стали обязательной частью приложений – без них вывод геометрии стал просто невозможен.

На момент написания книги последней версией является OpenGL 4.6. Большинство примеров в этой книге рассчитано на OpenGL 3.3, однако некоторые примеры (например, использующие аппаратную тесселляцию) потребуют более поздней версии OpenGL.

В этой книге мы сознательно не будем рассматривать весь устаревший функционал, а сразу перейдем к изучению OpenGL 3.3. При этом мы также будем считать, что окно и соответствующий контекст рендеринга уже были созданы (см. предыдущую главу).

Все взаимодействие с GPU в OpenGL происходит через драйвер. Поэтому, для того чтобы полноценно использовать все возможности OpenGL, следует установить свежий драйвер для своего GPU. Многие функции OpenGL на самом деле реализуются именно в драйвере. Без драйвера вы получите устаревший OpenGL, лишенный огромного количества важных функций.

Основные концепции OpenGL. Графический конвейер

Весь OpenGL основан на нескольких очень простых идеях, и крайне важно их хорошо понимать. Ключевой идеей является то, что OpenGL основан на модели *клиент–сервер*. Приложение – это клиент, а в роли сервера выступает драйвер и GPU. Сейчас обычно и клиент, и сервер находятся на одном и том же компьютере, но на самом деле это необязательно.

Все запросы от клиента поступают в очередь, откуда они со временем извлекаются сервером и выполняются. Аналогично ответ от сервера также помещается в очередь, откуда он со временем будет извлечен клиентом.

Фактически GPU работает *асинхронно* – мы передаем запрос на сервер (например, на вывод группы треугольников) и сразу же получаем управление обратно. При этом наш запрос был просто помещен в очередь, и вполне возможно, что GPU даже не начал его обрабатывать. Однако в OpenGL предусмотрены функции для явной синхронизации, но важно понимать, что их вызов фактически останавливает конвейер и очень негативно сказывается на быстродействии.

Также OpenGL можно рассматривать как *конечный автомат* (FSM, Finite State Machine) – у него есть свое состояние, и это состояние может изменяться только при помощи команд (функций) OpenGL. Само по себе состояние никогда не изменяется. Все состояние делится на состояние клиента и состояние сервера.

Для работы с OpenGL необходимо создать специальный *контекст*, связанный с окном, куда будет производиться рендеринг. Серверное состояние хранится в этом контексте. Обратите внимание, что сам контекст привязан к текущей нити (thread), поэтому если приложение выполняется сразу на нескольких нитях, то созданный контекст будет валиден только на той нити, где он был создан.

OpenGL представляет собой программный интерфейс (API, Application Programming Interface) к GPU. OpenGL задумывался как простой и не зависящий от аппаратного обеспечения API, который может быть реализован на самых разных платформах.

Именно поэтому в него не включены команды для работы с окнами и получения пользовательского ввода. Также в OpenGL нет команд для описания различных трехмерных моделей – это все делается поверх OpenGL, на основании тех команд, которые OpenGL предоставляет. Сам OpenGL умеет лишь выводить примитивы основных типов – точки, отрезки и треугольники.

Весь API OpenGL основан на языке C и представляет собой набор функций (также называемых командами) и констант. Все эти функции описываются в заголовочном файле `gl.h`. Поэтому программы, использующие OpenGL, включают в себя следующую строку:

```
#include <GL/gl.h>
```

Поскольку в языке C нет пространств имен (namespace), то, для того чтобы избежать возможных конфликтов по именам, все функции OpenGL и все его константы начинаются со специальных префиксов.

Все функции OpenGL обязательно начинаются с префикса `gl`, все константы – с префикса `GL_`. Обратите внимание, что этот же подход использовался и в библиотеках `freeglut` и `GLFW`.

Кроме того, в OpenGL есть некоторые функции, для которых существует несколько их вариантов, отличающихся числом и типом аргументов. Чтобы их различать, к концу имени каждой такой функции добавляется число аргументов (цифра от 1 до 4) и суффикс, задающий тип аргументов. Также к концу имени может быть добавлен специальный суффикс "v", обозначающий, что аргументы передаются не по одному, а в виде массива, указатель на который функция получает на вход.

```
glVertexAttrib2f ( 0, 1.2f, 3.4f );     // индекс и два аргумента типа float
glVertexAttrib3i ( 1, 0, 2, 4 );        // индекс и три аргумента типа int
```

OpenGL вводит для совместимости свои типы (их имена всегда начинаются с префикса `GL`). В табл. 12.1 приведены вводимые в OpenGL типы данных и соответствующие им префиксы, обратите внимание, что не для всех типов эти префиксы существуют.

Таблица 12.1. Типы OpenGL

Тип	Число бит	Описание	Константа типа	Префикс
`GLboolean`	1+	Логическое значение, `GL_TRUE` или `GL_FALSE`		
`GLbyte`	8	Целое число со знаком	`GL_BYTE`	`b`
`GLubyte`	8	Целое число без знака	`GL_UNSIGNED_BYTE`	`ub`
`GLshort`	16	Целое число со знаком	`GL_SHORT`	`s`
`GLushort`	16	Целое число без знака	`GL_UNSIGNED_SHORT`	`us`
`GLint`	32	Целое число со знаком	`GL_INT`	`i`
`GLuint`	32	Целое число без знака	`GL_UNSIGNED_INT`	`ui`
`GLint64`	64	Целое число со знаком		`i64`
`GLuint64`	64	Целое число без знака		`ui64`
`GLsizei`	32	Неотрицательное целое число для хранения размеров		
`GLenum`	32	Целое число для хранения констант		
`GLintptr`	Число бит, достаточное для хранения адреса	Целое число со знаком		
`GLsizeiptr`	Число бит, достаточное для хранения адреса	Целое число без знака		
GLsync	Число бит, достаточное для хранения адреса	Объект синхронизации		
`GLbitfield`	32	Набор битовых полей		
`GLhalf`	16	Число с плавающей точкой	`GL_HALF_FLOAT`	
`GLfloat`	32	Число с плавающей точкой	`GL_FLOAT`	`f`
`GLclampf`	32	Число с плавающей точкой в диапазоне [0, 1]		
`GLdouble`	64	Число с плавающей точкой	`GL_DOUBLE`	
`GLclampd`	64	Число с плавающей точкой в диапазоне [0, 1]		

Рендеринг геометрии осуществляется в так называемый *фреймбуфер* (framebuffer, буфер кадра). Он содержит информацию для каждого пиксела окна и состоит из набора различных буферов – буфера цвета, буфера глубины (*z*-буфера), буфера трафарета (рис. 12.1).

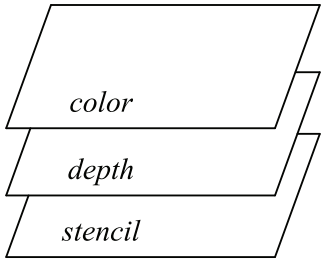

Рис. 12.1 ❖ Буферы в OpenGL

Кроме того, OpenGL позволяет создавать свои фреймбуферы, которые никак не привязаны к окну, тем не менее в них тоже можно осуществлять рендеринг. Такой фреймбуфер может содержать несколько различных буферов цвета (в том числе и различных форматов), и рендеринг идет сразу во все эти буферы.

Фактически каждый такой буфер содержит для каждого пиксела некоторый объем памяти, в который можно осуществлять запись и чтение.

Буфер цвета содержит для каждого пиксела соответствующий ему цвет обычно в трехкомпонентном (RGB) или четырехкомпонентном (RGBA) формате. При этом четвертая компонента – *альфа* – представляет из себя *непрозрачность* (opacity). Обратите внимание, что это никак не связано с оконной системой и не делает соответствующее окно прозрачным.

Буфер глубины для каждого пиксела хранит соответствующее ему значение глубины. Обычно значения хранятся в виде беззнаковых 16/24/32-битовых целых чисел. Однако многие современные GPU позволяют создавать буферы глубины, где все значения хранятся в виде 32-битовых чисел с плавающей точкой (`float`).

Буфер трафарета позволяет задать для каждого пиксела несколько бит (обычно не более 8) и выполнять над ними некоторые простые операции. Более подробно работа с буфером трафарета будет рассмотрена в соответствующем разделе.

Еще одним крайне важным понятием является так называемый *конвейер рендеринга* (rendering pipeline, рис. 12.2). Он состоит из набора отдельных шагов/стадий (stage), через которые проходят данные при их рендеринге.

Как видно по рис. 12.2, на вход поступают вершины, заданные своими координатами, и информация о том, как из этих вершин собирать отдельные примитивы (связность). В программируемом конвейере (а в OpenGL 3.3 используется именно он) все вершины поступают на вход *вершинного шейдера* (vertex shader) – специальной программы, выполняющейся на GPU.

Вершинный шейдер обрабатывает каждую вершину независимо от всех остальных вершин и не имеет доступа к информации о связности (т. е. как соединять эти вершины в примитивы) и о других вершинах.

На следующей стадии выполняется *сборка примитивов* из выданных вершинным шейдером вершин и информации о связности.

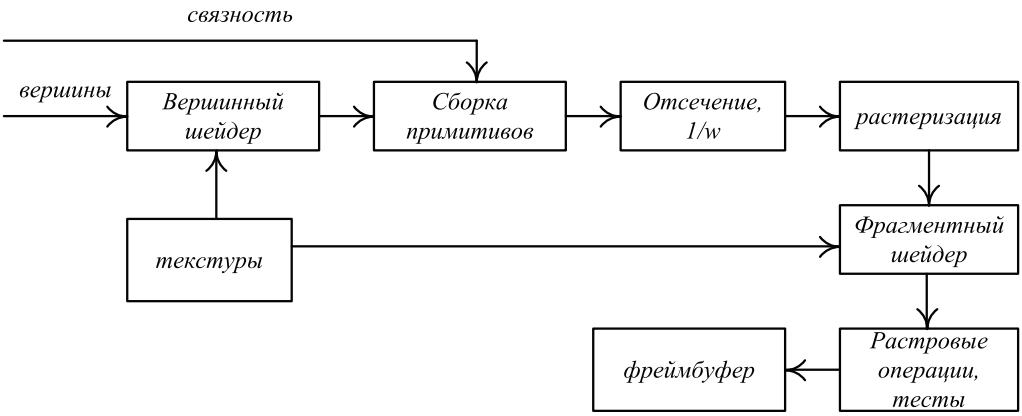

Рис. 12.2 ❖ Конвейер рендеринга в OpenGL 3.3

Далее происходит *отсечение геометрии* по границе области видимости и *перспективное деление* (т. е. деление координат вершин на однородную координату *w*).

Потом происходит *растеризация* примитива, превращающего его в набор отдельных *фрагментов*. Фрагмент – это то, что может в конце стать пикселом, если он не будет отброшен. После этого каждый фрагмент обрабатывается *фрагментным шейдером*, и для него выполняются различные тесты (например, тест глубины) и растровые операции. Далее мы подробнее рассмотрим каждый из этих шагов.

Внутри OpenGL есть специальная переменная, хранящая в себе ошибки. Изначально она устанавливается в `GL_NO_ERROR`. При возникновении ошибки соответствующий код ошибки записывается в эту переменную и может быть извлечен при помощи вызова функции `glGetError`. До тех пор, пока он не будет извлечен, никакой другой код ошибки не может быть запомнен (т. е. новый код ошибки в этом случае просто отбрасывается). При вызове `glGetError` текущий код ошибки возвращается, а значение соответствующей переменной снова устанавливается в `GL_NO_ERROR`.

Расширения OpenGL

OpenGL является расширяемым API – различные производители GPU могут давать разработчикам доступ к функциональности, еще не вошедшей в базовый состав OpenGL. Для этого используется механизм *расширений* (extension).

Расширения позволяют добавлять в OpenGL новые команды (функции) и новые константы. Каждое расширение имеет свое имя, состоящее из нескольких частей, например `GL_ARB_occlusion_query`.

Первая часть имени задает, относится ли данное расширение ко всем платформам (`GL`) или же к какой-то конкретной платформе (`WGL`, `GLX`). Далее идет часть, задающая «владельца» расширения. Обычно это один из производителей GPU (`AMD`, `NV`, `INTEL`). Также может использоваться `EXT` для обозначения расширения, общего для нескольких производителей GPU, или `ARB` для обозначения расширения, одобренного OpenGL Architecture Review Board.

Последняя часть имени задает, что именно вводит данное расширение (например, `occlusion_query`). Подобная схема именования гарантирует, что не будет возникать конфликтов между расширениями от различных производителей GPU и организаций.

После того как был создан контекст OpenGL, можно в любой момент получить список всех поддерживаемых расширений. Давайте рассмотрим, как это делается. Ниже приводится фрагмент кода, который печатает список всех поддерживаемых расширений, обратите внимание на использование суффиксов `v` и `i`.

```
GLint num;
glGetIntegerv ( GL_NUM_EXTENSIONS, &num );   // получить число поддерживаемых расширений

for ( int i = 0; i < num; i++ )              // для каждого расширения по номеру получить имя
      printf ( "%s\n", glGetStringi ( GL_EXTENSIONS, i ) );
```

Обычно новые версии OpenGL получаются путем переноса части расширений в базовый набор возможностей. В частности, это позволяет получить доступ к последним версиям OpenGL, притом что по умолчанию на платформе может поддерживаться очень старая версия (так, под Windows обычно поддерживается только OpenGL 1.5, но мы можем получить доступ к любой нужной нам версии именно через механизм расширений).

Существует стандартное место на официальном сайте OpenGL – **http://www.opengl.org/registry**, где можно скачать файлы, содержащие описания всех существующих расширений. Файл `glext.h` содержит описания расширений, общих для всех платформ, файл `wglext.h` содержит описания расширений, специфичных для платформы Windows, и `glxext.h` содержит описания расширений для X Window.

Каждое расширение может ввести как новые команды (т. е. функции), так и новые константы. С константами все очень просто – они явно задаются в соответствующем заголовочном файле. С функциями ситуация несколько сложнее. Для каждой новой функции в соответствующем заголовочном файле описывается тип указателя на соответствующую функцию.

```
typedef void (GLAPIENTRY * PFNGLUSEPROGRAMPROC) (GLuint program);
```

Это связано с тем, что для каждой платформы есть механизм, позволяющий по имени функции получить ее адрес. Поэтому можно просто ввести переменную с именем нужной нам функции и типом указателя на нее и присвоить ей адрес этой функции. Тогда в соответствии с правилами языка C мы можем обращаться к этой переменной так, как если бы это была просто функция.

```
PFNGLUSEPROGRAMPROC glUseProgram;
```

Делать это каждый раз для всех вводимых функций довольно тяжело. Поэтому обычно используют готовые библиотеки, где все это уже сделано. В данной книге мы в этих целях будем использовать библиотеку `GLEW`, доступную для скачивания на сайте **http://glew.sourceforge.net**.

Для использования этой библиотеки необходимо сначала подключить соответствующие заголовочные файлы, как показано ниже:

```
#include <GL/glew.h>          // общие для всех платформ расширения
#ifdef _WIN32
```

```
   #include <GL/wglew.h>      // расширения только для Windows
#else
   #include <GL/glxew.h>      // расширения только для X Window
#endif
```

После этого необходимо проинициализировать данную библиотеку из нашего приложения. Мы будем делать это следующим образом (обратите внимание, что это можно делать только после того, как будет создан контекст OpenGL):

```
glewExperimental = GL_TRUE;
glewInit   ();
glGetError ();      // вызов glewInit может привести к ошибке, игнорируем ее
```

Все, что нужно для использования библиотеки GLEW, уже включено в класс GlutWindow, рассмотренный в предыдущей главе.

Кроме того, GLEW вводит все функции, вводимые различными расширениями, также GLEW предоставляет некоторое количество своих функций и макросов, позволяющих легко проверять поддержку того или иного расширения.

```
if ( !GLEW_ARB_tessellation_shader)          // проверяем поддержку GL_ARB_tessellation_shader
      printf ( "Tessellation not supported\n");

if ( !glewIsSuported ( "GL_ARB_debug_output" ) )   // проверяем поддержку GL_ARB_debug_output
      printf ( "Debug output is not supported\n" );
```

Системы и преобразования координат в OpenGL

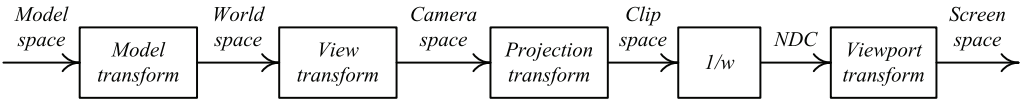

Рис. 12.3 ❖ Системы координат в OpenGL

В процессе рендеринга координаты вершины подвергаются ряду преобразований перехода из одной системы координат в другую. На рис. 12.3 приведены основные преобразования координат.

Если у нас есть сцена, состоящая из нескольких объектов, то обычно каждый такой объект задается в некоторой своей системе координат. Например, вы хотите использовать в сцене уже готовую модель, созданную в каком-либо пакете трехмерного моделирования. Тогда эта модель будет задана в своей системе координат. Эта система координат обычно называется *системой координат модели* (объекта) (model space).

Далее, у вас есть некоторая общая для всей сцены система координат, куда вы помещаете отдельные модели (объекты). Эта система координат называется *мировой* (world space). Преобразование из системы координат модели в мировую систему координат осуществляется при помощи *модельной матрицы*.

Обратите внимание, что в сцене одна и та же модель может быть использована несколько раз. При этом отдельные экземпляры этой модели будут иметь общую систему координат модели, но разные модельные матрицы.

Следующим преобразованием координат будет преобразование в *систему координат камеры* (camera/eye space) при помощи *видовой матрицы* (view matrix). Часто вместо отдельных видовой и модельной матриц используется *модельно-видовая матрица* (modelview matrix), равная их произведению.

Далее осуществляется *преобразование проектирования* при помощи *матрицы проектирования* (projection matrix). После этого мы получаем координаты в *системе координат отсечения* (clip space).

Отсечение примитивов

Для того чтобы не выполнять растеризацию заведомо невидимых граней и гарантировать безопасность перспективного деления, все примитивы после перевода координат их вершин в систему координат отсечения отсекаются по 6 плоскостям. Фактически все, что лежит вне приводимой ниже области, автоматически обрезается:

$$\begin{cases} -w \le x \le w \\ -w \le y \le w \\ -w \le z \le w \end{cases}.$$

Данная область в случае перспективного проектирования представляет из себя усеченную прямоугольную пирамиду (рис. 12.4).

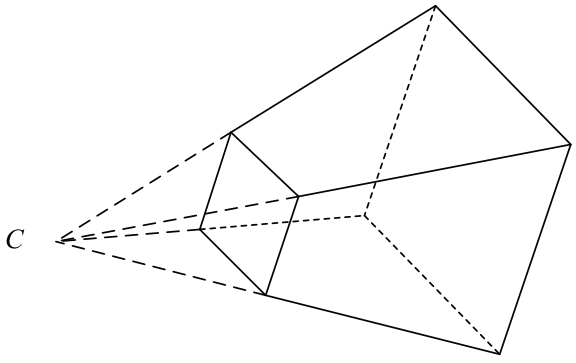

Рис. 12.4 ❖ Пирамида видимости

После выполнения перспективного деления (на w) мы получаем так называемые *нормализованные координаты устройства* (NDC, Normalized Device Coordinates), которые лежат в $[-1, 1]^3$.

Заключительное преобразование переводит спроектированные координаты из $[-1, 1]^3$ в координаты в окне (screen space).

Вершинный шейдер

Основная часть преобразования координат (вплоть до проектирования) выполняется *вершинным шейдером* (vertex shader). Вершинный шейдер – это выполняемая на GPU программа, которая получает на вход *атрибуты* вершины и должна выдать некоторый набор значений, которые будут проинтерполированы в ходе растери-

зации. Обязательным выходным значением являются однородные координаты в пространстве отсечения (через встроенную в GLSL переменную `gl_Position`).

Помимо собственно атрибутов вершины, вершинному шейдеру также доступны так называемые `uniform`-переменные – специальные, задаваемые пользователем значения, которые остаются неизменными на протяжении всего примитива (или группы примитивов). В качестве примеров таких значений выступают, например, матрицы для преобразования координат. Значения для этих переменных задаются в приложении, шейдеру они доступны только на чтение.

Кроме того, шейдер может также читать значения из различных текстур и использовать полученные значения для дальнейших расчетов.

Шейдеры пишутся на специальном высокоуровневом языке GLSL (OpenGL Shading Language), который основан на языке C. В этот язык были добавлены специальные типы данных (векторы, матрицы и семплеры), а также набор встроенных функций и переменных. Также из языка были удалены некоторые конструкции, не подходящие для шейдеров. К их числу относятся указатели и операции с ними, строки, битовые поля, рекурсия. Кроме того, в язык были добавлены новые ключевые слова, связанные со спецификой шейдеров.

Ниже приводится пример простого вершинного шейдера.

```
#version 330 core                       // задаем версию OpenGL 3.3

uniform mat4 proj;                      // матрица проектирования
uniform mat4 mv;                        // модельно-видовая матрица
                                        // входные атрибуты:
layout(location = 0) in vec3 pos;       //   координаты вершины
layout(location = 2) in vec2 txCoords;  //   текстурные координаты
                                        // выходные значения:
out vec2 texCoords;                     // текстурные координаты

void main(void)
{
      texCoords = txCoords;   // копируем текстурные координаты на выход
                              // преобразуем координаты при помощи произведения
                              // модельно-видовой матрицы и матрицы проектирования
      gl_Position  = proj * mv * vec4 ( pos, 1.0 );
}
```

Подробное описание GLSL содержится в приложении А.

Растеризация и система координат экрана

Преобразование координат из нормализованных координат экрана $(x_{ndc} \quad y_{ndc} \quad z_{ndc})^T$ в экранные (оконные) координаты $(x_w \quad y_w \quad z_w)^T$ осуществляется по следующей формуле:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{w}{2} x_{ndc} + \left(x_{tl} + \frac{w}{w} \right) \\ \frac{h}{2} y_{ndc} + \left(y_{tl} + \frac{h}{2} \right) \\ \frac{f-n}{2} z_{ndc} + \frac{f+n}{2} \end{pmatrix}.$$

В этой формуле через (x_{tl}, y_{tl}) обозначены координаты в пикселах верхнего левого угла области вывода (рис. 12.5), через w и h – ширина и высота области вывода в пикселах, через n и f – диапазон изменения глубины.

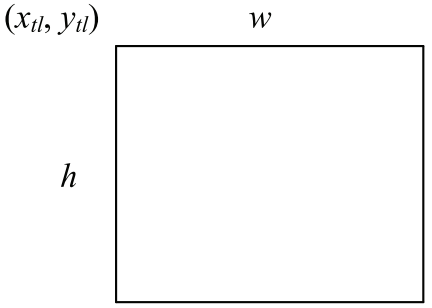

Рис. 12.5 ❖ Координаты экрана

Для задания области вывода внутри текущего окна служит следующая команда OpenGL:

```
void glViewport ( GLint x, GLint y, GLsizei w, GLsizei h );
```

По умолчанию параметры n и f равны 0 и 1 соответственно. Для задания другого диапазона (обратите внимание, что переданные параметры будут отсечены по отрезку [0, 1]) служит команда `glDepthRangef`.

```
void glDepthRangef ( GLclampf f, GLclampf n );
```

OpenGL дает возможность при растеризации сразу отбрасывать грани заданного типа (лицевые или нелицевые). В OpenGL нет понятия нормали к грани – значение нормали обычно задается только в вершинах. Поэтому для определения того, является ли грань лицевой, OpenGL использует другой подход. При этом для определения того, какие именно грани являются лицевыми, используется направление обхода проекций вершин (рис. 12.6).

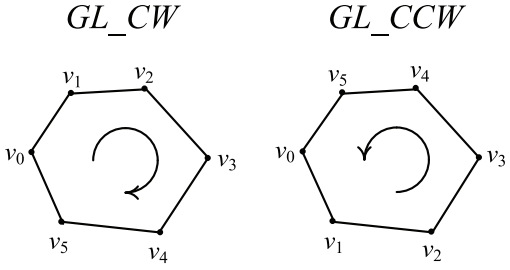

Рис. 12.6 ❖ Возможные варианты направления обхода

Для того чтобы определить направление обхода проекций вершин, нам достаточно просто найти площадь соответствующего многоугольника со знаком. Знак будет определять, имеет ли место направление обхода по часовой стрелке (`GL_CW`) или против часовой стрелки (`GL_CCW`).

Какое именно направление обхода соответствует лицевым граням, задается при помощи команды glFrontFace:

```
void glFrontFace ( GLenum dir );
```

Параметр *dir* задает ориентацию для лицевых граней (GL_CW или GL_CCW). По умолчанию лицевым граням соответствует ориентация GL_CCW.

При помощи команды glCullFace можно задать, какие именно грани нужно отбрасывать. Допустимыми значениями параметра *mode* являются GL_FRONT (лицевые), GL_BACK (нелицевые) и GL_FRONT_AND_BACK (и лицевые, и нелицевые).

```
void glCullFace ( GLenum mode );
```

И наконец, для того чтобы грани заданного типа действительно отбрасывались, нужно включить их отбрасывание при помощи вызова функции glEnable с параметром GL_CULL_FACE. Аналогично, для того чтобы выключить автоматическое отбрасывание граней, вызывается glDisable с тем же параметром.

```
void glEnable  ( GLenum cap );
void glDisable ( GLenum cap );
```

На самом деле многие возможности OpenGL можно включать и выключать при помощи команд glEnable и glDisable.

Если грань не была отброшена, то она растеризуется. Результатом растеризации является множество *фрагментов*. Каждый фрагмент обладает координатами соответствующего ему пиксела, но, кроме этих координат, он содержит еще и целый ряд дополнительных атрибутов. Они получаются путем интерполяции выходных значений вершинного шейдера для вершин соответствующего многоугольника.

Фрагментный шейдер

Получившиеся при растеризации фрагменты попадают на вход *фрагментного шейдера* (fragment shader). Фрагментный шейдер получает на вход значения, получаемые путем интерполяции выходных значений вершинного шейдера. Эти значения интерполируются вдоль всего примитива во время его растеризации, и каждый фрагмент получает соответствующие ему значения.

Также фрагментный шейдер получает на вход uniform-переменные, заданные пользователем, и текстуры.

Задачей фрагментного шейдера является вычисление по входным значениям цвета для данного фрагмента. В случае когда рендеринг осуществляется сразу в несколько буферов цвета (MRT, Multiple Render Targets), фрагментный шейдер выдает сразу несколько выходных значений. Обратите внимание, что не обязательно выходные значения должны быть именно цветами.

Ниже приводится простой пример фрагментного шейдера, получающего *текстурные координаты* от вершинного шейдера и использующего их для чтения цвета из текстуры, который и становится возвращаемым значением.

```
#version 330 core             // задаем версию OpenGL 3.3

in vec2            texCoords; // входные текстурные координаты от вершинного шейдера
uniform samepler2D tex;       // текстура, из которой мы будем читать цвета
out vec4           color;     // выходное значение
```

```
void main ()
{
                          // читаем значение из текстуры и записываем его в выходную переменную
      color = texture ( tex, texCoords );
}
```

Операции с фрагментами

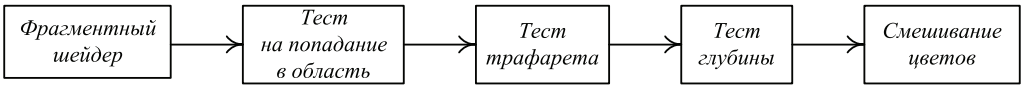

Рис. 12.7 ❖ Операции с фрагментами

После фрагментного шейдера фрагмент проходит через ряд тестов, после чего накладывается на фреймбуфер (рис. 12.7). Если хотя бы для одного разрешенного теста условие не выполнено, то фрагмент отбрасывается. Далее идет смешивание цветов. На этом шаге можно не просто записать цвета фрагмента в буфер цвета, но и наложить этот цвет на ранее существующий во фреймбуфере цвет. Основные тесты будут рассмотрены далее в этой главе.

Работа с буферами

Обычно, перед тем как выводить что-то в буфер, этот буфер необходимо очистить. Для этого служит команда `glClear`, позволяющая за один вызов очистить сразу несколько буферов.

```
void glClear ( GLbitfield mask );
```

Параметр *mask* является битовой маской, где отдельные биты задают, какие именно буферы следует очистить. Стандартным буферам соответствуют следующие константы: `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT` и `GL_STENCIL_BUFFER_BIT`.

Для задания значений, которыми будут заполняться соответствующие буферы, служат следующие функции:

```
void glClearColor   ( GLclampf red, GLclampf green,
                      GLclampf blue, GLclampf alpha );
void glClearDepth   ( GLfloat depth );
void glClearStencil ( GLint s );
```

Кроме того, для каждого буфера можно задать маску, управляющую записью в соответствующий буфер. Для этого служат следующие команды:

```
void glColorMask   ( GLboolean red, GLboolean green,
                     GLboolean blue, GLboolean alpha );
void glDepthMask   ( GLboolean depth );
void glStencilMask ( GLuint mask );
```

В первых двух командах аргументы являются логическими значениями – `GL_TRUE` разрешает запись в соответствующую компоненту, а `GL_FALSE` – запрещает. В функции `glStencilMask` параметр *mask* определяет, в какие биты соответствующего значения в буфере трафарета разрешена запись.

Атрибуты вершин. Вершинные массивы, VBO, VAO

Вся геометрия в OpenGL задается при помощи *вершин* (vertex). Каждая вершина имеет свои координаты, кроме координат, у вершины могут быть и другие *атрибуты* – нормаль, цвет, текстурные координаты и т. п.

Каждый атрибут имеет свой тип, но при этом OpenGL не придает какого-то смысла атрибутам – для него это просто набор абстрактных значений, которые заданы для каждой вершины выводимого объекта.

Каждый атрибут вершины имеет свой индекс, который используется для связи передаваемых пользователем данных из приложения с входными переменными вершинного шейдера. Для этого служит описатель `layout`, ниже приводится пример такого задания. Первый атрибут имеет индекс 0 и тип `vec3` – трехмерный вектор. Второй атрибут имеет индекс 2 и тип `vec2` – двухмерный вектор.

```
                                        // входные атрибуты:
layout(location = 0) in vec3 pos;       //   координаты вершины
layout(location = 2) in vec2 txCoords;  //   текстурные координаты
```

Каждая реализация OpenGL может иметь свои ограничения на максимальное количество атрибутов вершины, ниже приводится код, получающий это максимальное количество для текущей реализации.

```
int maxVertexAttribs;
glGetIntegerv ( GL_MAX_VERTEX_ATTRIBS, &maxVertexAttribs );
```

Есть два способа, при помощи которых можно задать данные в вершине. Мы можем использовать вершинный массив, хранящий данные сразу для группы вершин, или можем использовать постоянное значение атрибута для всех вершин примитива.

Каждый атрибут вершины может быть или скаляром, или 2/3/4-мерным вектором одного из базовых типов.

Для задания постоянного значения для какого-либо атрибута служит команда `glVertexAttrib`. Ниже приводятся примеры ее использования. Обратите внимание на использование суффиксов числа аргументов и их типа в имени функции. Мы не будем приводить все варианты этой команды – их слишком много, – скажем только, что поддерживается от одной до четырех компонент одного из следующих базовых типов: `GLfloat`, `GLshort`, `GLdouble`, `GLuint` и `GLint`.

```
glVertexAttrib2f ( 0, 0.0f, 0.5f );   // задает атрибут 0 равным (0,0.5)

GLshort vals[] = { 1, 4, 7 };
glVertexAttrib3sv ( 1, vals );        // задаем атрибут 1 равным (1, 4, 7)
```

Вершинные массивы, задание атрибутов при помощи вершинных массивов

Вторым способом задания значений для вершин является использование *вершинных массивов* (vertex array). Вершинный массив представляет из себя массив значений для одного или нескольких атрибутов, хранящийся в памяти приложения (клиента).

При таком способе задания каждая вершина получает свое значение атрибута – они будут последовательно читаться из массива или нескольких массивов.

Для задания атрибутов при помощи вершинных массивов служат функции `glVertexAttrib*Pointer`. Ниже приводятся соответствующие описания.

```
void glVertexAttribPointer  ( GLuint index, GLint size, GLenum type,
                              GLboolean normalized, GLsizei stride,
                              const void * pointer );
void glVertexAttribIPointer ( GLuint index, GLint size, GLenum type,
                              GLsizei stride, const void * pointer );
void glVertexAttribLPointer ( GLuint index, GLint size, GLenum type,
                              GLuint stride, const void * pointer );
```

Параметр *index* задает индекс атрибута (используемый в директиве `layout` в вершинном шейдере). Параметр *size* задает число компонент атрибута и принимает значение от 1 до 4.

Параметр *type* задает тип значений компонент и принимает одно из следующих значений: `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_USNIGNED_SHORT`, `GL_INT` и `GL_UNSIGNED_INT` для `glVertexAttribIPointer` и `glVertexAttribPointer`.

Кроме того, `glVertexAttribPointer` также поддерживает следующие значения: `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_DOUBLE`, `GL_FIXED`, `GL_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_2_10_10_10_REV` и `GL_UNSIGNED_INT_10F_11F_11F_REV`. Единственным допустимым значением для `glVertexAttribLPointer` является `GL_DOUBLE`.

Параметр *normalized* задает, нужно *нормализовывать* целочисленные значения или нет. Под нормализацией подразумевается перемасштабирование числа, при котором весь диапазон изменения для соответствующего типа отображается в [−1, 1] для типов со знаком и в [0, 1] для типов без знака. Так, например, для значений типа `GL_UNSIGNED_BYTE` нормализация означает фактически просто деление на 255.

В одном вершинном массиве можно хранить значения не только одного конкретного атрибута, но и значения сразу нескольких атрибутов. В этом случае сначала идет набор значений для первой вершины, затем набор значений для второй вершины, потом набор значений для третьей вершины и т. д. Параметр *stride* задает расстояние в байтах между двумя последовательными наборами значений. И наконец, параметр *pointer* содержит указатель на область памяти, содержащую необходимые значения. Пример такого задания будет приведен далее.

В случае если у вершины есть несколько атрибутов, то возможны два базовых способа организации их хранения в вершинных массивах. В первом случае мы помещаем все атрибуты в один массив – такой подход называется массив структур. В роли структуры в этом случае выступает набор атрибутов для одной вершины.

Во втором случае мы храним каждый атрибут в своем отдельном вершинном массиве, этот подход называется структура массивов.

Для выбора того, откуда следует брать значение для того или иного атрибута – постоянное значение (задаваемое при помощи `glVertexAttrib*`) или значения из вершинного массива, – служит команда `glEnableVertexArray` (использовать значения из массива) и `glDisableVertexArray` (использовать постоянное значение).

```
void glEnableVertexArray  ( GLuint attribIndex );
void glDisableVertexArray ( GLuint attribIndex );
```

Обычно при рендеринге используются одни и те же данные для вершин. Но при использовании вершинных массивов данные приходится постоянно копировать из памяти приложения в память GPU, что крайне неэффективно.

Было бы гораздо удобнее, если бы можно было с самого начала один раз скопировать данные в память GPU и потом их использовать без копирования. Именно такую возможность и предоставляют *вершинные буферы* (VBO, Vertex Buffer Object).

Вершинный буфер позволяет разместить массив данных прямо в памяти GPU и использовать ее для рендеринга. Каждый такой буфер имеет свой тип (*target*), определяющий его использование. Наиболее часто встречающимися типами являются `GL_ARRAY_BUFFER` (буфер, который содержит сами вершины) и `GL_ELEMENT_ARRAY_BUFFER` (буфер, который содержит массив целочисленных индексов вершин).

Каждый вершинный буфер (и не только он) является объектом OpenGL, и он идентифицируется при помощи целого числа без знака (`GLuint`). При этом значение 0 зарезервировано и не соответствует ни одному буферу.

Для создания вершинного буфера нужно выделить для него свободный идентификатор при помощи команды `glGenBuffers`:

```
void glGenBuffers ( GLsizei n, GLuint * buffers );
```

Данная команда выделяет *n* идентификаторов и записывает их в массив *buffers*.

Обычно в командах OpenGL идентификаторы не задаются явно – вместо этого нужный буфер делается *текущим* для соответствующего типа буфера. После этого данный буфер будет оставаться текущим, пока текущим не станет другой буфер или текущим не станет буфер 0, означающий отсутствие текущего буфера для заданного типа. И когда OpenGL понадобятся соответствующие данные, он просто возьмет их из текущего буфера заданного типа.

Для того чтобы буфер с идентификатором *id* (значение 0 является допустимым для *id* и обозначает отсутствие текущего буфера для заданного типа) сделать текущим для типа *target*, служит следующая команда:

```
void glBindBuffer ( GLenum target, GLuint id );
```

Параметр *target* принимает одно из следующих значений: `GL_ARRAY_BUFFER`, `GL_ATOMIC_COUNTER_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_DISPATCH_INDIRECT_BUFFER`, `GL_DRAW_INDIRECT_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_IXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_QUERY_BUFFER`, `GL_SHADER_STORAGE_BUFFER`, `GL_TEXTURE_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER` и `GL_UNIFORM_BUFFER`.

Обратите внимание, что вызов `glGenBuffers` не создает объекта внутри OpenGL – он только выделяет под него идентификатор. Само создание объекта происходит в тот момент, когда буфер делается текущим первый раз.

Следующим шагом является наполнение уже выбранного буфера данными. Самым простым способом сделать это является использование команды `glBufferData`.

```
void glBufferData ( GLenum target, GLsizeiptr size,
                    const void * data, GLenum usage );
```

Параметр *target* задает тип буфера, параметр *size* задает размер выделяемой памяти под буфер в байтах. Параметр *data* указывает на область памяти, откуда надо взять данные для инициализации буфера. Параметр *usage* содержит информацию для OpenGL о предполагаемом использовании данного буфера.

Возможными значениями для *usage* являются GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ и GL_DYNAMIC_COPY.

В этих константах STREAM обозначает, что данные один раз задаются и будут использованы всего несколько раз. Значение STATIC обозначает, что данные задаются один раз и будут многократно использованы. И значение DYNAMIC обозначает, что данные будут постоянно изменяться и постоянно использоваться.

Значение DRAW обозначает, что эти данные будут использоваться в качестве источника данных для рендеринга. Значение READ обозначает, что данные будут читаться приложением, и COPY обозначает, что данные будут читаться и использоваться как источник данных для рендеринга.

Обратите внимание, что значение параметра *usage* – это просто информация, для того чтобы OpenGL мог более эффективно использовать данный буфер. Допустимо задать одно применение в glBufferData, а затем использовать буфер совсем другим образом – просто это может быть не так эффективно.

Если параметр *data* не равен NULL, то выделяется память на GPU, и в нее копируются заданные этим указателем данные. В противном случае ранее выделенная под этот буфер память GPU освобождается, и ее можно в дальнейшем снова выделить.

После того как память GPU была выделена, можно выполнить копирование части данных при помощи команды glBufferSubData (обратите внимание, что эта команда только копирует данные, а не выделяет память).

```
void glBufferSubData ( GLenum target, GLintptr offset,
                       GLsizeiptr size, const void * data );
```

Для того чтобы использовать вместо вершинного массива данные из вершинного буфера, мы делаем соответствующий буфер текущим и, как и ранее, вызываем glVertexAttrib*Pointer. Только в этом случае меняется смысл параметра *pointer* – он задает не адрес, а смещение начала заданного атрибута для вершины внутри текущего буфера.

Ниже приводится пример создания вершинного буфера, содержащего трехмерные координаты и двухмерные текстурные координаты, и задание соответствующих атрибутов вершин.

```
float buf [] =                  // данные в памяти клиента
{
      0, 0, 0, 0, 0,            // (x, y, z), (s, t)
      0, 1, 0, 0, 1,
      0, 1, 1, 1, 1,
      1, 0, 0, 1, 0
};

GLuint buffer;                  // идентификатор буфера
glGenBuffers ( 1, &buffer );  // выделяем идентификатор под буфер
glBindBuffer ( GL_ARRAY_BUFFER, buffer );   // делаем буфер текущим
                                            // копируем данные в буфер
glBufferData ( GL_ARRAY_BUFFER, sizeof ( buf ), buf, GL_STATIC_DRAW );
                                // задаем первый атрибут
glVertexAttribPointer ( 0,          // индекс
```

```
                        3,          // число компонент
                        GL_FLOAT,   // тип одной компоненты
                        GL_FLOAT,   // нормализация не нужна, так как тип float
                                    // stride равен размеру одной вершины
                        5 * sizeof ( float ),
                                    // координаты начинаются сразу же
                          (const void *) 0                        );
                              // задаем второй атрибут
glVertexAttribPointer ( 0,          // индекс
                        2,          // число компонент
                        GL_FLOAT,   // тип одной компоненты
                        GL_FLOAT,   // нормализация не нужна, так как тип float
                                    // stride равен размеру одной вершины
                        5 * sizeof ( float ),
                                    // координаты начинаются после первых трех значений
                        (const void *) (3*sizeof(float))
                      );
```

Если вершинный буфер стал не нужен, то его следует удалить при помощи вызова команды `glDeleteBuffers`:

```
void glDeleteBuffers ( GLsizei n, const GLuint * buffers );
```

Помимо команд `glBufferData` и `glBufferSubData`, есть и другой способ задания данных для вершинного буфера – отобразить его память приложения (клиента). При этом мы получаем обычный указатель в память приложения, соответствующий всему буферу или заданной его части. Далее мы можем работать с памятью по этому указателю как с обычной памятью. В конце работы мы закрываем это отображение, после чего соответствующие данные становятся доступными на GPU. Обратите внимание, что после закрытия отображения нельзя обращаться к данным по этому указателю.

Для управления отображением содержимого вершинного буфера в память приложения используются команды `glMapBufferRange` и `glUnmapBuffer`.

```
void * glMapBufferRange ( GLenum target, GLintptr offset,
                          GLsizeiptr length, GLbitfield access );
void   glUnmapBuffer    ( GLenum target );
```

Параметр *access* является набором битовых флагов, определяющим способ доступа к буферу. Допустимыми флагами являются `GL_MAP_READ_BIT` (чтение со стороны CPU) или `GL_MAP_WRITE_BIT` (чтение со стороны GPU).

Следующий пример демонстрирует использование отображения буфера. Вместо того чтобы выделить массив в памяти приложения, заполнить его данными и потом скопировать их в память GPU, мы просто отображаем вершинный буфер в память приложения и сразу же заполняем данные по этому указателю. Дополнительного выделения памяти в этом случае не нужно. Сам этот пример заполняет буфер вершинами, задающими тор.

```
float     d1       = 1.0f / (float) n1;
float     d2       = 1.0f / (float) n2;
float     deltaPhi = d1 *  M_PI;
float     deltaPsi = d2 * 2.0f * M_PI;
int       index    = 0;
```

```
glm::vec3 * ptr = (glm::vec3 *)
                glMapBufferRange ( GL_ARRAY_BUFFER, 0,
                       sizeof (float) * (n1+1)*(n2+1), GL_MAP_WRITE_BIT );

for ( int i = 0; i <= n1; i++ )
{
      float phi    = i * deltaPhi;
      float sinPhi = sin ( phi );
      float cosPhi = cos ( phi );

      for ( int j = 0; j <= n2; j++ )
      {
            float     psi = j * deltaPsi;
            glm::vec3 n ( sinPhi * cos( psi ), sinPhi * sin( psi ), cosPhi );

            ptr [index] = org + r * n;
            index++;
      }
}
glUnmapBuffer ( GL_ARRAY_BUFFER );
```

Обычно вершины объекта содержат много различных атрибутов, которые, в свою очередь, лежат в различных вершинных буферах. При этом перед рендерингом такого объекта чаще всего необходимо вызывать функции `glBindBuffer`, `glVertexAttrib*Pointer` и `glEnableVertexAttribArray`.

Нередко подобных объектов бывает много, и каждый раз вызывать все необходимые функции для настройки вершинных атрибутов очень неэффективно и неудобно. С этой целью в OpenGL был добавлен специальный *объект состояния вершинных буферов* (VAO, Vertex Array Object). Такой объект хранит в себе все состояние, требуемое для задания вершины. Для переключения на другое состояние мы просто выбираем другой VAO.

Объекты этого типа также идентифицируются при помощи беззнаковых целых чисел, но при этом 0 соответствует всегда существующему объекту, который является VAO по умолчанию.

Для создания новых объектов этого типа служит команда `glGenVertexArrays`:

```
void glGenVertexArrays ( GLsizie n, GLuint * arrays );
```

После создания объекта его можно сделать текущим при помощи вызова команды `glBindVertexArray`.

```
void glBindVertexArray ( GLuint array );
```

Для уничтожения этих объектов служит команда `glDeleteVertexArrays`:

```
void glDeleteVertexArrays ( GLsizei n, const GLuint * arrays );
```

После того как создан объект состояния (VAO), он запоминает все состояние, связанное с использованием вершинных атрибутов и буферов (`glBindBufer`, `glVertexAttri b*Pointer`, `glEnable/DisableVertexArray`).

Таким образом, для того чтобы быстро переключиться с одной конфигурации буферов и атрибутов на другую, достаточно просто сделать текущим другой объект состояния.

Если у нас есть два вершинных буфера, то можно скопировать данные из одного буфера в другой при помощи следующей команды:

```
void glCopyBufferSubData ( GLenum readTarget, GLenum writeTarget,
                           GLintptr readOffs, GLintptr writeOffs,
                           GLsizeiptr size );
```

Вывод примитивов

Вся геометрия в OpenGL задается в виде различных примитивов – точек, отрезков, многоугольников. На рис. 12.8 приведены поддерживаемые типы примитивов, обратите внимание, что выводимые многоугольники должны всегда быть выпуклыми.

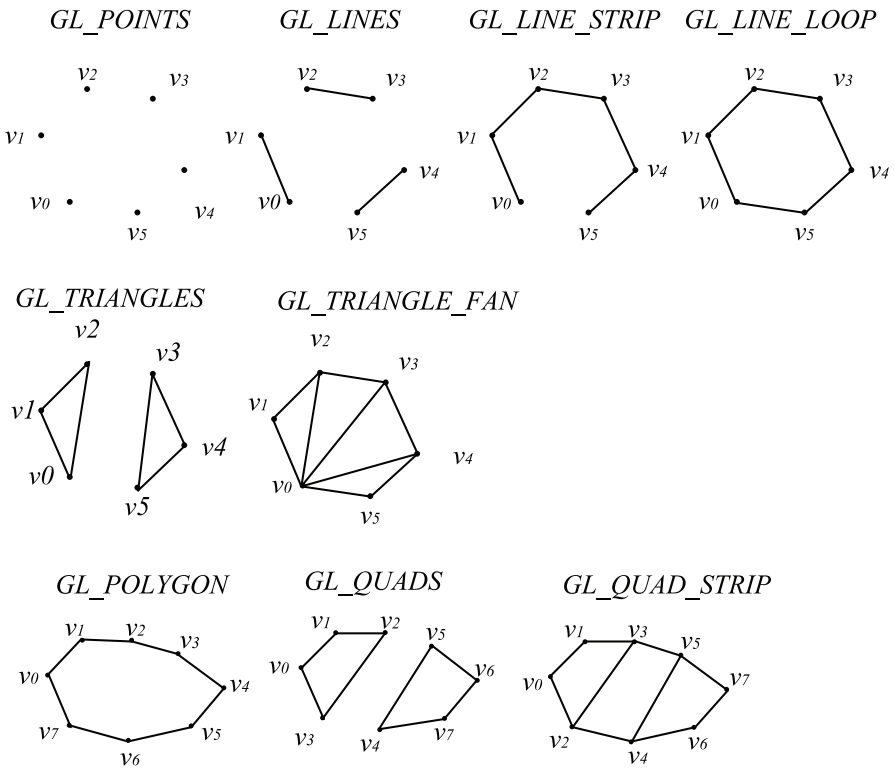

Рис. 12.8 ❖ Поддерживаемые типы примитивов

Для многих базовых типов геометрии – например, треугольников – есть несколько различных примитивов, например `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` и `GL_TRIANGLE_FAN`.

Это связано с тем, что дополнительные типы примитивов позволяют выводить одну и ту же геометрию, используя для этого меньшее число вершин, что обычно является более эффективным. Поэтому если геометрию можно организовать так, что для ее вывода можно использовать дополнительные типы (такие как `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`), то это лучше сделать.

Наиболее распространенным типом геометрии являются треугольники GL_TRIANGLES. При его использовании все входные вершины разбиваются на группы по три, и каждая такая группа из трех вершин рассматривается как отдельный треугольник. При этом *n* входным вершинам соответствует *n*/3 треугольника. Лишние вершины просто игнорируются.

GL_TRIANGLE_STRIP и GL_TRIANGLE_FAN выводят наборы связанных треугольников, при этом *n* входным вершинам соответствует *n* – 2 треугольника.

Для вывода отрезков и ломаных используются примитивы GL_LINES, GL_LINE_STRIP и GL_LINE_LOOP. Для вывода независимых отрезков служит примитив GL_LINES, при этом *n* вершинам соответствует *n*/2 отрезков. GL_LINE_STRIP и GL_LINE_LOOP выводят незамкнутую и замкнутую ломаные соответственно.

При помощи примитива GL_POINTS можно выводить не только точки, но и *точечные спрайты*. Под точечным спрайтом понимается выровненный по границам экрана квадрат, заданный своим центром и своим размером. На полученный квадрат можно наложить текстуру, что делает точечные спрайты очень удобными для рендеринга систем частиц. При этом центр спрайта задается вершиной, а его размер вычисляется в вершинном шейдере.

В OpenGL есть два способа задания примитивов, обычный и *индексированный*. В первом случае мы просто передаем массив вершин на вход команды glDrawArrays. Из данных вершин будут строиться примитивы заданного типа.

```
void glDrawArrays ( GLenum mode, GLint first, GLsizei count );
```

Первый параметр (*mode*) задает тип выводимых примитивов – то, как именно надо интерпретировать массив вершин. Выводятся *count* вершин, начиная с вершины с номером *first*. Лишние вершины (например, если для режима GL_TRIANGLES мы зададим 5 вершин) просто отбрасываются.

```
glBindVertexArray ( vao );
glDrawArrays      ( GL_TRIANGLES, 0, NUM_VERTICES );
glBindVertexArray ( 0 );
```

При втором способе задания геометрии нужен не только массив вершин (вершинный буфер типа GL_ARRAY_BUFFER), но еще и массив целочисленных индексов. Массив индексов задает номера вершин в том порядке, в котором из них надо собирать примитивы. Это более гибкий способ, обычно он требует меньшего числа вершин, поскольку вместо повторения вершины можно просто повторить ее индекс. Для вывода примитивов таким способом служат команды glDrawElements и glDrawRangeElements.

```
void glDrawElements      ( GLenum mode, GLsizei count,
                           GLenum type, const void * indices );
void glDrawRangeElements ( GLenum mode, GLuint start,
                           GLuint end, GLsizei count,
                           GLenum type, const void * indices );
```

В этих командах параметр *mode*, как и ранее, задает тип выводимых примитивов. Параметр *count* задает число выводимых индексов. Параметры *start* и *end* задают минимальный и максимальный используемые индексы в массиве индексов. Их задание может помочь драйверу провести оптимизацию рендеринга.

Параметр *type* задает, какой именно целочисленный тип используется для задания индексов. Возможными значениями являются GL_UNSIGNED_BYTE, GL_UNSIGNED_

SHORT и GL_UNSIGNED_INT. С точки зрения быстродействия лучше использовать минимальный (по размеру) тип для задания индексов, т. е. если все индексы не более 255, то лучше всего использовать тип GL_UNSIGNED_BYTE.

Параметр *indices* может быть как адресом массива индексов в памяти приложения, так и смещением внутри вершинного буфера с типом GL_ELEMENT_ARRAY_BUFFER (если есть текущий буфер такого типа, то этот параметр трактуется именно как смещение внутри буфера).

При выводе группы связанных примитивов, например типа GL_TRIANGLE_STRIP, очень полезным оказывается так называемый *перезапуск примитива* (primitive restart). Если перезапуск примитива разрешен, то максимальное значение для типа индекса (так, для GL_UNSIGNED_BYTE максимальным значением будет 255) трактуется особым образом – не как индекс вершины, а как признак того, что одна группа примитивов закончилась и начинается следующая группа примитивов того же типа (никак не связанная с предыдущей).

В качестве примера давайте рассмотрим случай, когда мы хотим за один раз вывести сразу два примитива типа GL_TRIANGLE_STRIP с индексами (0, 1, 2, 3) и (4, 5, 6, 7, 8, 9). В данном случае для задания индексов мы можем использовать значения типа GL_UNSIGNED_BYTE, максимальным значением для данного типа будет 255. Таким образом, для вывода сразу обоих примитивов за одну команду нам следует использовать следующий массив индексов: (0, 1, 2, 3, 255, 4, 5, 6, 7, 8, 9).

Для того чтобы включить перезапуск примитива (изначально он выключен и максимальное значение индекса трактуется просто как номер вершины), необходимо вызвать glEnable с аргументом GL_PRIMITIVE_RESTART.

Провоцирующая вершина

При рендеринге примитивов выходные значения вершинного шейдера интерполируются вдоль всего примитива для получения значений для фрагментов. Однако можно описать выходное значение вершинного шейдера при помощи описателя **flat**. В этом случае интерполяции не происходит, а соответствующее значение для каждого фрагмента будет взято от определенной вершины исходного примитива, называемой *провоцирующей* (provoking vertex). OpenGL поддерживает два разных варианта выбора провоцирующей вершины – GL_FIRST_VERTEX_CONVENTION и GL_LAST_VERTEX_CONVENTION. В табл. 12.2 приводятся реальные номера вершин для каждого из этих режимов.

```
flat in int matIndex;
```

Таблица 12.2. Провоцирующая вершина

Тип примитива	GL_FIRST_VERTEX_CONVENTION	GL_LAST_VERTEX_CONVENTION
GL_POINTS	i	i
GL_LINES	$2i - 1$	$2i$
GL_LINE_LOOP	i	$i + 1$, если i меньше числа вершин i иначе
GL_LINE_STRIP	i	$i + 1$
GL_TRIANGLES	$3i - 2$	$3i$
GL_TRIANGLE_STRIP	i	$i + 3$
GL_TRIANGLE_FAN	$i + 1$	$i + 2$

Для задания используемого режима выбора провоцирующей вершины служит следующая команда:

```
void glProvokingVertex ( GLenum provokeMode );
```

Буфер трафарета и работа с ним

Одним из буферов, с которым умеет работать OpenGL, является *буфер трафарета* (stencil buffer). Этот буфер для каждого пиксела хранит несколько (обычно не больше 8) бит, которые интерпретируются как целые числа без знака.

Можно включить тест трафарета, в котором для каждого фрагмента будет выполняться сравнение по маске соответствующего фрагменту значения в буфере трафарета с некоторым заданным пользователем значением (reference value). Если тест не пройден, то фрагмент просто отбрасывается.

Кроме того, в зависимости от результатов теста глубины и теста трафарета можно выполнить заданную операцию над соответствующим значением в буфере трафарета.

Для задания операции сравнения в тесте трафарета используется функция `glStencilFunc`:

```
void glStencilFunc ( GLenum func, GLint ref, GLuint mask );
```

Параметр *ref* задает базовое значение, с которым будет сравниваться значение из буфера трафарета. Параметр *mask* задает маску, которая будет накладываться как на значение из буфера трафарета, так и на *ref* перед сравнением. И наконец, параметр *func* задает используемую операцию сравнения, возможные значения для этого параметра приведены в табл. 12.3.

Таблица 12.3. Поддерживаемые OpenGL операции сравнения

Операция	Описание
GL_NEVER	Тест никогда не выполнен
GL_ALWAYS	Тест всегда выполнен
GL_EQUAL	Тест выполнен, если значения равны
GL_NOTEQUAL	Тест выполнен, если значения не равны
GL_LESS	Тест выполнен, если *(ref & mask) < (stencil & mask)*
GL_LEQUAL	Тест выполнен, если *(ref & mask) <= (stencil & mask)*
GL_GREATER	Тест выполнен, если *(ref & mask) > (stencil & mask)*
GL_GEQUAL	*Тест выполнен, если (ref & mask) >= (stencil & mask)*

При помощи команды `glStencilOp` можно установить действия, выполняемые над значениями в буфере трафарета. OpenGL выделяет три разных случая:

- тест трафарета не пройден;
- тест трафарета пройден, тест глубины не пройден;
- тесты трафарета и глубины пройдены.

Для каждого из этих трех случаев можно задать свое действие.

```
void glStencilOp ( GLenum sfail, GLenum dfail, GLenum dpass );
```

Параметр *sfail* задает операцию в случае, когда тест трафарета не пройден, параметр *dfail* задает операцию в случае, когда тест трафарета пройден, а тест глу-

бины нет. И наконец, параметр *dpass* задает операцию в случае, когда оба теста успешно пройдены. В табл. 12.4 приведены допустимые значения для всех этих параметров.

Таблица 12.4. Допустимые операции над значением в буфере трафарета

Операция	Описание
GL_KEEP	Значение в буфере трафарета не изменяется
GL_ZERO	Значение становится равным нулю
GL_REPLACE	Значение заменяется на параметр *ref* из функции glStencilFunc
GL_INCR	Значение увеличивается на единицу с отсечением
GL_INCR_WRAP	Значение увеличивается на единицу по модулю
GL_DECR	Значение уменьшается на единицу с отсечением
GL_DECR_WRAP	Значение уменьшается на единицу по модулю
GL_INVERT	Все биты значения инвертируются

Разница между GL_INCR и GL_INCR_WRAP (как и между GL_DECR и GL_DECR_WRAP) заключается в том, что при увеличении максимального значения (если под пиксел отведено 8 бит, то максимальным значением будет 255) в первом случае мы опять получим 255 (т. е. получающееся значение будет приведено к отрезку [0, 255]). Во втором случае мы получим 0, т. е. получающееся значение будет взято по модулю 256 (т. е. взят остаток от деления на 256).

Если буфера трафарета нет, то тест трафарета всегда проходит. По умолчанию тест трафарета выключен. Для его включения необходимо вызвать glEnable с параметром GL_STENCIL_TEST.

Тест глубины

Тест глубины является еще одним тестом, который в OpenGL можно настроить. По умолчанию он выключен, для его включения следует вызвать glEnable с параметром GL_DEPTH_TEST.

Кроме того, можно явно задать, какую именно операцию сравнения следует использовать в этом тесте при помощи функции glDepthFunc. В качестве допустимых значений для нее выступают те же самые значения из табл. 12.3.

```
void glDepthFunc ( GLenum func );
```

Полупрозрачность. Смешивание цветов

OpenGL поддерживает полупрозрачность – возможно задать, что выводимый фрагмент должен накладываться на уже существующий пиксел в буфере цвета определенным образом с учетом цвета этого пиксела. Для работы с полупрозрачностью ее нужно сначала включить при помощи вызова glEnable с аргументом GL_BLEND.

Поддерживается несколько режимов такого наложения, конкретный режим задается при помощи функции glBlendFunc.

```
void glBlendFunc ( GLenum sFunc, GLenum dFunc );
```

Параметры *sFunc* и *dFunc* определяют, каким образом будут вычисляться коэффициенты (s_r, s_g, s_b, s_a) и (d_r, d_g, d_b, d_a), с помощью которых осуществляется смешивание цветов – исходный цвет (цвет фрагмента, source) и цвет приемника (цвет в буфере цвета, destination). Смешивание цветов осуществляется по следующей формуле:

$$C_{out} = s \cdot C_{source} + d \cdot C_{dest}.$$

В следующей таблице заданы законы для вычисления коэффициентов s и d по входным цветам C_{source} и C_{dest}.

Таблица 12.5. Допустимые способы вычисления коэффициентов

Операция	Весовой коэффициент
GL_ZERO	(0, 0, 0, 0)
GL_ONE	(1, 1, 1, 1)
GL_SRC_COLOR	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_SRC_COLOR	(1 – R_s, 1 – G_s, 1 – B_s, 1 – A_s)
GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	(1 – A_s, 1 – A_s, 1 – A_s, 1 – A_s)
GL_DST_COLOR	(R_d, G_d, B_d, A_d)
GL_ONE_DST_COLOR	(1 – R_d, 1 – G_d, 1 – B_d, 1 – A_d)
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	(1 – A_d, 1 – A_d, 1 – A_d, 1 – A_d)

Классическому пониманию полупрозрачности соответствует следующий вызов `glBlendFunc`:

```
glBlendFunc ( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
```

Обратите внимание, что в общем случае результат вывода полупрозрачных граней зависит от того, в каком именно порядке они выводились. Поэтому обычно эти грани сортируют и выводят, начиная с самой дальней и заканчивая самой ближней (back-to-front).

Текстуры и работа с ними

Одним из распространенных способов улучшения качества изображения является наложение на объекты *текстур*. В самом простейшем случае текстура представляет собой двухмерный массив цветовых значений. Для обозначения этих значений по аналогии с пикселами используется термин *тексел* (texel, от texture element).

Такой массив (фактически просто цветное изображение, картинка) накладывается на выводимую грань – при растеризации грани для каждого фрагмента путем интерполяции определяются его *текстурные координаты*, идентифицирующие тексел внутри текстуры. Далее во фрагментном шейдере по этим текстурным координатам из текстуры извлекается соответствующий им цвет. На рис. 12.9 приведены изображения одного и того же объекта без текстуры (с постоянным цветом для всех пикселов) и с текстурой. Обратите внимание, насколько выразительнее является объект с текстурой.

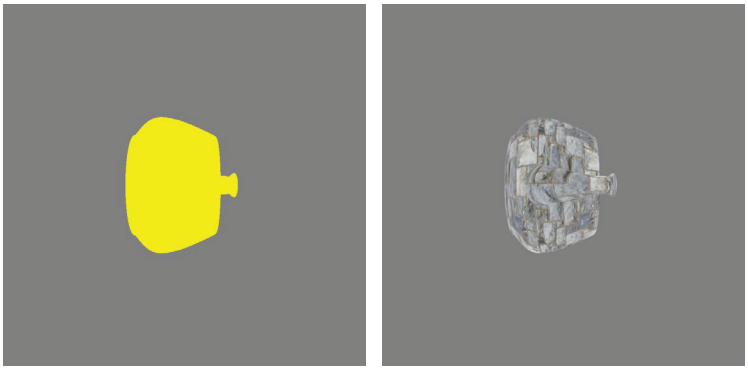

Рис. 12.9 ❖ Объект с текстурой и без

Использование текстур позволяет заметно улучшить внешний вид объектов без усложнения используемой геометрии. Кроме того, текстуры могут хранить не только цвета, но и самые различные данные, которые можно организовать в виде массивов.

OpenGL поддерживает работу с текстурами, выделяя при этом различные типы текстур. Простейший тип – это двухмерная текстура. Такая текстура представляет собой двухмерный массив текселов, и для получения цвета из него нужны две текстурные координаты (s, t).

Также поддерживаются одномерные и трехмерные текстуры. Первые представляют собой одномерный массив текселов и используют для доступа к ним всего одну текстурную координату – s. Вторые представляют собой трехмерный массив и требуют трех текстурных координат (s, t, r).

Используемые при этом текстурные координаты являются *нормализованными*, т. е. они изменяются от 0 до 1 независимо от фактического размера массива. Это позволяет легко менять используемые текстуры без изменения самих текстурных координат.

Каждую из этих текстур можно рассматривать как функцию, определенную на единичном отрезке ($[0, 1]$), единичном квадрате ($[0, 1]^2$) и единичном кубе ($[0, 1]^3$).

Есть и еще один тип текстур – *кубические текстуры* (cubemap), которые можно рассматривать как функции, заданные на поверхности единичной сферы. Функция, определенная на поверхности сферы, фактически является функцией от направления и оказывается очень полезной в целом ряде случаев.

К сожалению, наложение двухмерной текстуры на поверхность сферы невозможно без сильных искажений. Поэтому используется другой подход. Функция от направления может рассматриваться как функция не на поверхности сферы, а как функция, заданная на поверхности единичного куба (рис. 12.10). В этом случае удается довольно просто наложить набор двухмерных текстур на его поверхность без искажений.

Рассмотрим куб $[-1, 1]^3$ с центром в начале координат и ребрами, параллельными координатным осям. Тогда произвольному ненулевому направлению v мы можем сопоставить однозначным образом некоторую точку P на поверхности одной из граней этого куба. Для определения этой точки мы просто выпускаем луч из начала координат с направляющим вектором v, и точка P будет просто пересечением этого луча с поверхностью куба.

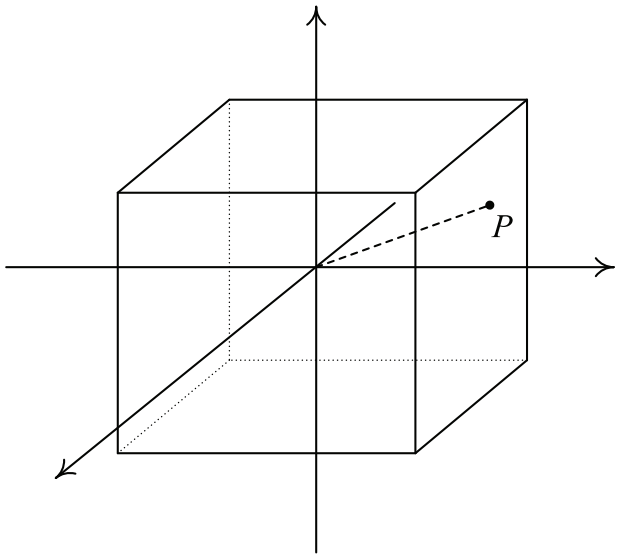

Рис. 12.10 ❖ Кубическая текстура как функция, заданная на поверхности куба

Тем самым устанавливается взаимно однозначное соответствие между гранями куба и всевозможными направлениями. При этом задать текстуру на поверхности куба очень просто – нужно просто задать шесть изображений: по одному на каждую грань куба.

Соответствующий тип текстур аппаратно поддерживается GPU, и такие текстуры называются кубическими. В качестве текстурных координат для таких текстур выступает трехмерный вектор (s, t, r). Однако обратите внимание, что длина этого вектора не играет никакой роли – текстурные координаты (s, t, r) и ($2s$, $2t$, $2r$) соответствуют одной и той же точке на поверхности куба (но ($-s$, $-t$, $-r$) соответствует уже другой точке на поверхности куба).

Кроме этих типов текстур, есть и другие типы, о которых мы расскажем позже.

Текстуры также являются объектами OpenGL и идентифицируются при помощи беззнаковых целых чисел. Как и для вершинных буферов, значение 0 является зарезервированным и не соответствует ни одной текстуре.

Для того чтобы выделить n идентификаторов текстур, служит функция `glGenTextures`, полностью аналогичная функции `glGenBuffers`:

```
void glGenTextures ( GLsizei n, GLuint * textures );
```

Для освобождения идентификатора (и уничтожения соответствующего объекта с освобождением всех ресурсов) служит функция `glDeleteTexteres`:

```
void glDeleteTextures ( GLsizei n, const GLuint * textures );
```

После того как идентификатор под текстуру был выделен, никакого объекта внутри OpenGL еще создано не было. Он создается при первом выборе текстуры как текущей для одного из типов текстуры при помощи `glBindTexture`.

```
void glBindTexture ( GLenum target, GLuint texture );
```

Как и при работе с вершинными буферами, для каждого типа текстуры можно выбрать свою «текущую текстуру» или сделать так, что текущей текстуры для данного типа не будет (т. е. выбрать текстуру с идентификатором 0).

Параметр *target* задает тип текстуры. Рассмотренным выше типам текстур соответствуют значения `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` и `GL_TEXTURE_CUBE_MAP`.

Следующим шагом, после того как созданная текстура стала текущей для заданного типа, является загрузка изображения в нее. Для этого используются такие функции:

```
void glTexImage1D ( GLenum target, GLint level,
                    GLenum internalFormat, GLsizei width,
                    GLint border, GLenum format, GLenum type,
                    const void * data );
void glTexImage2D ( GLenum target, GLint level,
                    GLenum internalFormat, GLsizei width, GLsizei height,
                    GLint border, GLenum format, GLenum type,
                    const void * data );
void glTexImage3D ( GLenum target, GLint level,
                    GLenum internalFormat, GLsizei width,
                    GLsizei height, GLsizei depth,
                    GLint border, GLenum format, GLenum type,
                    const void * data );
```

Параметр *target* (как и для `glBindTexture`) задает тип текстуры. Параметр *level* задает, какой именно уровень в mipmap-пирамиде мы будем загружать (об этом позже). Параметр *internalFormat* задает внутренний формат для хранения текстуры в памяти GPU. Наиболее распространенными форматами, не указывающими явно размер одного тексела, являются `GL_RGBA`, `GL_RGB` и `GL_ALPHA`. Некоторые другие поддерживаемые внутренние форматы будут рассмотрены в других главах.

Параметры *width*, *height* и *depth* задают размер загружаемого изображения в текселах по каждому измерению. В старых версиях OpenGL было требование, чтобы эти величины обязательно были степенями двух. Сейчас это ограничение уже снято. Тем не менее есть ограничение на максимальный размер текстуры по каждому измерению.

Параметр *border* унаследован от ранних версий OpenGL, сейчас не используется и должен быть равен 0.

Последние три параметра задают сами данные загружаемых текселов. Формат, в котором задаются данные для текселов, не обязан совпадать с внутренним форматом – при необходимости OpenGL сам произведет преобразование типов. Последний параметр, *data*, является указателем на область памяти приложения, содержащей начальные данные для всех загружаемых текселов. Каждый тексел задается несколькими компонентами (от 1 до 4 в зависимости от формата). Формат, в котором заданы текселы, определяется параметром *format*, а параметр *type* задает тип, значения которого используются для задания текселов.

Тем самым параметр *format* определяет, в каком именно формате (число и порядок компонентов) мы задаем данные в памяти приложения, а параметр *internalFormat* – в каком именно формате они будут храниться в памяти GPU. Наиболее распространенными форматами являются `GL_RGBA`, `GL_RGB` и `GL_ALPHA`. Наиболее распространенным типом является `GL_UNSIGNED_BYTE`.

Ниже приводится пример кода, строящего двухмерную текстуру 64×64 в виде шахматных клеток размером 8×8.

```
#define WIDTH  64
#defihe HEIGHT 64

GLuint buildCheckerTexture ()
{
      GLubyte image [WIDTH][HEIGHT][4];
      GLuint  id;

      for ( int i = 0; i < HEIGHT; i++ )
            for ( int j = 0; j < WIDTH; j++ )
            {
                  bool xf = (i & 8) == 0;
                  bool yf = (j & 8) == 0;
                  int  c  = xf == yf ? 255 : 0;

                  image [i][j][0] = (GLubyte) c;
                  image [i][j][1] = (GLubyte) c;
                  image [i][j][2] = (GLubyte) c;
                  image [i][j][3] = (GLubyte) 255;
            }


      glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
      glGenTextures ( 1, &id );
      glBindTexture ( GL_TEXTURE_2D, id );

      glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
      glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
      glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
      glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
      glTexImage2D    ( GL_TEXTURE_2D, 0, GL_RGBA, WIDTH, HEIGHT,
                        0, GL_RGBA, GL_UNSIGNED_BYTE, image );

      return id;
}
```

В этом примере используются две функции OpenGL, которые мы еще не рассмотрели, – `glPixelStorei` и `glTexParameteri`.

Обычно мы считаем, что данные текселов упакованы плотно, т. е. сразу за компонентами одного тексела идут компоненты следующего тексела, одна строка текселов сразу же идет за предыдущей строкой и т. д. Но на самом деле это не всегда так – иногда данные могут быть заданы с использованием выравнивания. Поэтому OpenGL позволяет явно задать выравнивание данных, используемых для задания текстуры. Именно для этого и нужна команда `glPixelStorei`.

```
void glPixelStorei ( GLenum pname, GLint param );
```

Здесь параметр *pname* определяет, какой именно параметр для выравнивания мы будем задавать, а *param* задает значение для этого параметра. Наиболее важным параметром является `GL_UNPACK_ALIGNMENT`. Он задает выравнивание строк, по

умолчанию оно равно 4 – т. е. смещение строк от начала изображения должно быть кратно 4.

При помощи команд `glTexParameter*` можно задать такие важные свойства для чтения из текстуры, как режим отсечения текстурных координат и фильтрование.

Рассмотрим эти свойства на примере выборки (чтения) из двухмерной текстуры. Для чтения из этой текстуры нужно задать текстурные координаты – два числа из отрезка [0, 1].

Но что произойдет, если мы зададим в качестве текстурных координат числа, выходящие за этот отрезок? Есть несколько стандартных вариантов поведения в этом случае, какой именно из них будет иметь место, определяется режимом *отсечения текстурных координат* (`GL_TEXTURE_WRAP_*`). Режим задается независимо для каждой из координат. Простейшими вариантами являются приведение значения к ближайшему допустимому (`GL_CLAMP_TO_EDGE`) и взятие его дробной части (`GL_WRAP`). Соответственно, в первом случае значение 1.2 будет приведено к 1, а во втором – к 0.2.

Более сложным является понятие *фильтрования* текстуры. Дело в том, что при выводе грани с наложенной на нее текстурой может происходить как сжатие этой текстуры, так и ее растяжение. В OpenGL можно задать различные способы фильтрования как для сжатия текстуры (`GL_TEXTURE_MIN_FILTER`), так и для ее растяжения (`GL_TEXTURE_MAG_FILTER`).

При чтении из текстуры, с одной стороны, текстура – это просто набор текселов, а с другой – для выборки из текстуры мы используем не целочисленные индексы, а непрерывно изменяющиеся вещественные числа. Таким образом, возможна ситуация, когда точка, заданная текстурными координатами, не попадает точно в тексел, а лежит сразу между четырьмя ближайшими текселами. Задача фильтрации заключается в определении того, что именно нужно вернуть при чтении из текстуры в этом случае.

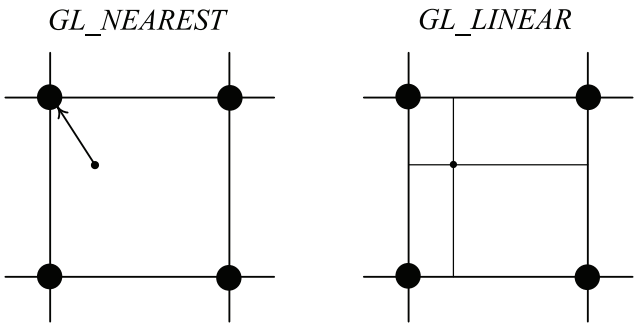

Рис. 12.11 ❖ Режимы фильтрации текстур

Самым простым фильтром является `GL_NEAREST` (точечная фильтрация). При использовании данного фильтра просто выбирается тот тексел, который ближе всех лежит к точке с заданными текстурными координатами (рис. 12.11, слева).

Этот способ фильтрации является самым быстрым, но он дает не самое качественное изображение – при сильном увеличении каждый тексел текстуры просто заменяется прямоугольником из значений одного цвета, т. е. мы получаем «блочный» характер изображения.

Поэтому для улучшения качества часто используют линейную фильтрацию (GL_LINEAR) – в этом случае определяются четыре тексела, между которыми лежит наша точка. После этого производится билинейная интерполяция между этими четырьмя значениями, исходя из положения точки относительно этих 4 текселов.

В результате, когда точка с текстурными координатами точно попадает в тексел, мы получаем значение этого тексела. Но чем сильнее она отходит от него, тем больше будет влияние остальных трех текселов. Обратите внимание, что данный способ фильтрации является более дорогим, так как требуется прочесть все четыре значения из текстуры.

При сжатии текстуры мы можем столкнуться с *артефактами дискретизации* (aliasing artifacts). Давайте, как и ранее в главе 10, возьмем текстуру с черно-белыми клетками и сожмем ее. При достаточно сильном сжатии (когда сразу несколько клеток проектируется в один пиксел экрана) мы опять сталкиваемся с той же проблемой, что и при трассировке лучей. Мы получаем фактически просто мешанину черных и белых точек, т. е. просто мусор.

В следующей книге мы рассмотрим математику, лежащую в основе этого явления. Пока лишь достаточно сказать, что проблема лежит в мелких (т. е. высокочастотных) деталях изображения. Если мы сгладим изображение достаточно сильно (т. е. подавим эти высокие частоты), то подобные артефакты пропадут.

Проблема в том, что степень сглаживания зависит от требуемой степени сжатия. Делать полноценное сглаживание на этапе выполнения очень дорого. Поэтому обычно применяют так называемое *пирамидальное фильтрование* (mipmapping).

Основная идея пирамидального фильтрования заключается в том, чтобы заранее подготовить вместо одного изображения набор заранее сглаженных изображений. Тогда на этапе выполнения мы, исходя из требуемой степени сжатия, выбираем для чтения изображения с ближайшей степенью сжатия.

Рассмотрим это более подробно. Пусть у нас есть изображение размером $2^n \times 2^n$. Разобьем его на блоки 2×2 тексела. Для каждого такого блока найдем среднее значение цвета и из этих средних значений построим новое изображение размером $2^{n-1} \times 2^{n-1}$.

Потом получившееся новое изображение опять разобьем на блоки 2×2, найдем среднее значение и построим еще одно изображение (размером $2^{n-2} \times 2^{n-2}$).

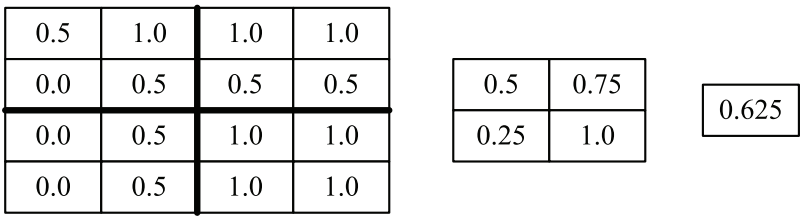

Рис. 12.12 ❖ Построение пирамиды

Этот процесс мы будем продолжать до тех пор, пока не придем к изображению, состоящему из единственного тексела. Значением этого тексела будет среднее значение цвета по всему исходному изображению.

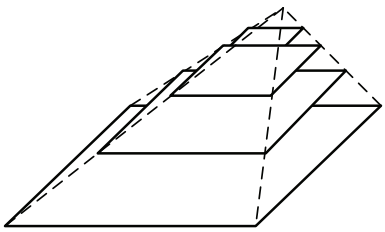

Рис. 12.13 ❖ Пирамида сжатых изображений

Если мы все эти изображения сложим одно на другое, то можем представить это как пирамиду (рис. 12.13). В основании пирамиды лежит исходное изображение. Размер каждого следующего изображения равен ¼ от размера предыдущего изображения. Таким образом, мы можем получить оценку на то, во сколько раз больше памяти нужно для хранения всей этой пирамиды, по сравнению с исходным изображением:

$$1+\frac{1}{4}+\left(\frac{1}{4}\right)^2+\ldots=\frac{1-\left(\frac{1}{4}\right)^n}{1-\frac{1}{4}}<\frac{1}{1-\frac{1}{4}}=\frac{4}{3}.$$

Таким образом, общий размер требуемой памяти возрастает не более чем на треть.

Каждый слой этой пирамиды соответствует сжатию в 2 раза предыдущего слоя (по каждому измерению). На самом деле описанный процесс просто является сглаживанием, т. е. он убирает высокие частоты.

Пирамидальное фильтрование позволяет дешево и просто избавиться почти от всех артефактов путем незначительного увеличения занимаемой памяти.

При этом для определения степени сжатия/растяжения текстуры используется следующий подход. Пусть u, v и w – это текстурные координаты, а x и y – это оконные координаты. Тогда вводятся следующие величины:

$$\rho=\max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2+\left(\frac{\partial v}{\partial x}\right)^2+\left(\frac{\partial w}{\partial x}\right)^2},\sqrt{\left(\frac{\partial u}{\partial y}\right)^2+\left(\frac{\partial v}{\partial y}\right)^2+\left(\frac{\partial w}{\partial y}\right)^2}\right\};$$

$$\lambda_{base}(x,y)=\log_2(\rho(x,y));$$

$$\lambda'(x,y)=\lambda_{base}(x,y)+clamp(bias_{texobj}+bias_{shader});$$

$$\lambda=\begin{cases}lod_{\max}, \lambda'>lod_{\max}\\ \lambda', lod_{\min}\le\lambda'\le lod_{\max}\\ lod_{\min}, \lambda'<lod_{\min}\end{cases}.$$

Здесь параметр $bias_{texobj}$ – это значение свойства `GL_TEXTURE_LOD_BIAS`, $bias_{shader}$ – это параметр `bias`, используемый в шейдере при чтении из текстуры. Указанные здесь производные заменяются их разностными аналогами.

Фактически с первой версии OpenGL пирамидальное фильтрование аппаратно поддерживается. Поэтому все, что нужно для его использования, – это задать все слои пирамиды и установить соответствующий режим фильтрации. Все эти параметры вычисляются на GPU.

Для каждой текстуры можно задать не одно изображение, а набор отдельных слоев пирамиды. Параметр *level* в функциях glTexImage* как раз и задает номер слоя, в который нужно загрузить соответствующие данные. Таким образом, мы загружаем не только слой 0, но и все остальные слои пирамиды.

OpenGL поддерживает и другой способ задания слоев для пирамиды. Если у нас есть только базовое изображение (слой 0), то можно вызвать функцию glGenerateMipmap. Она сама построит все остальные слои и загрузит их в текстуру.

```
void glGenerateMipmap ( GLenum target );
```

Режим фильтрации (как и ряд других режимов, связанных с чтением из текстуры) задается при помощи команд glTexParameter*:

```
void glTexParameteri  ( GLenum target, GLenum pname, GLint     param );
void glTexParameteriv ( GLenum target, GLenum pname, GLint   * params );
void glTexParameterf  ( GLenum target, GLenum pname, GLfloat   param );
void glTexParameterfv ( GLenum target, GLenum pname, GLfloat * params );
```

Параметр *target*, как и ранее, задает тип текстуры. Параметр *pname* определяет, какое именно свойство текстуры будет задаваться. И наконец, параметр *param*/*params* задает значение/значения для соответствующего свойства.

Наиболее важными для нас значениями *pname* будут GL_TEXTURE_COMPARE_FUNC, GL_TEXTURE_COMPARE_MODE, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T и GL_TEXTURE_WRAP_R.

Для использования пирамидального фильтрования при чтении из текстуры служат следующие режимы фильтрования: GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR и GL_LINEAR_MIPMAP_LINEAR.

Режим GL_NEAREST_MIPMAP_NEAREST заключается в том, что выбирается ближайший к требуемой степени сжатия слой пирамиды и на этом слое выбирается ближайший тексел.

Режим GL_LINEAR_MIPMAP_NEAREST выбирает ближайший слой в пирамиде, но выборка на этом слое использует линейное фильтрование.

Режим GL_NEAREST_MIPMAP_LINEAR выбирает два ближайших слоя (так что требуемая степень сжатия заключена между степенями сжатия для этих слоев), на каждом из них выбирает по ближайшему текселу, но возвращает результат линейной интерполяции между прочтенными текселами.

И наконец, режим GL_LINEAR_MIPMAP_LINEAR выбирает два ближайших слоя. На каждом из них делается билинейная выборка и возвращается результат линейной интерполяции между прочтенными значениями. Поскольку этот режим использует и линейную интерполяцию, и билинейную интерполяцию, то его называют трилинейным. Он обеспечивает наибольшее качество, но при этом является и наиболее затратным (поскольку требует прочтения сразу восьми значений из текстуры).

Как видно из рассказанного выше, у нас есть текущая текстура для каждого типа, и мы можем задавать ее свойства. На самом деле все несколько сложнее – есть набор *текстурных блоков* (texture unit). В каждом текстурном блоке мы можем выбрать свою текущую текстуру для каждого типа, и она будет текущей для заданного блока.

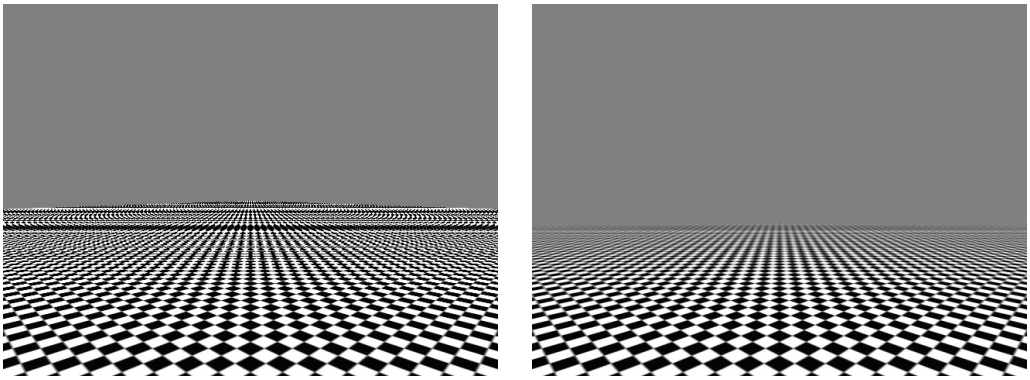

Рис. 12.14 ❖ Отличие фильтрации GL_NEAREST (слева) от GL_LINEAR_MIPMAP_LINEAR (справа)

Таким образом, можно сразу иметь много текущих текстур одного типа – просто они должны быть текущими в разных текстурных блоках. Именно такой подход используется при задании текстур для шейдеров – для каждой используемой в шейдере текстуры мы просто задаем номер текстурного блока, в котором соответствующая текстура является текущей.

Из всех текстурных блоков один является текущим – все команды по работе с текстурами будут относиться именно к этому блоку. Для переключения между текстурными блоками служит следующая команда:

```
void glActiveTexture ( GLenum unit );
```

Параметр *unit* задает текущий текстурный блок и принимает значения следующего вида: GL_TEXTURE0, GL_TEXTURE1, GL_TEXTURE2 и т. д. Для выбора *i*-го текстурного блока как текущего можно использовать следующий вызов:

```
glActiveTexture ( GL_TEXTURE0 + i );
```

Чаще всего встречаются текстуры, у которых при хранении в памяти GPU используется 8 бит на каждую компоненту и при чтении из такой текстуры происходит *нормализация*, т. е. прочитанное целочисленное беззнаковое значение переводится в число с плавающей точкой, лежащее в отрезке [0, 1]. Такие текстуры называются беззнаковыми нормализованными текстурами.

Кроме таких текстур, OpenGL умеет поддерживать и другие типы текстур. В их число входят текстуры, компоненты которых являются 16- и 32-битовыми числами с плавающей точкой. Эти числа могут и не лежать на отрезке [0, 1], в частности принимать отрицательные значения. В табл. 12.6 приводятся наиболее распространенные внутренние форматы (т. е. форматы для хранения в памяти GPU) поддерживаемых OpenGL текстур с компонентами, заданными в виде чисел с плавающей точкой.

Таблица 12.6. Форматы текстур, заданные с плавающей точкой

Внутренний формат	Число компонент	Число бит на компоненту
GL_R16F	1	16
GL_RG16F	2	16
GL_RGB16F	3	16

Таблица 12.6 (окончание)

Внутренний формат	Число компонент	Число бит на компоненту
GL_RGBA16F	4	16
GL_R32F	1	32
GL_RG32F	2	32
GL_RGB32F	3	32
GL_RGBA32F	4	32
GL_R11F_G11F_B10F	3	10/11

На рис. 12.15 приведено устройство форматов с плавающей точкой в OpenGL.

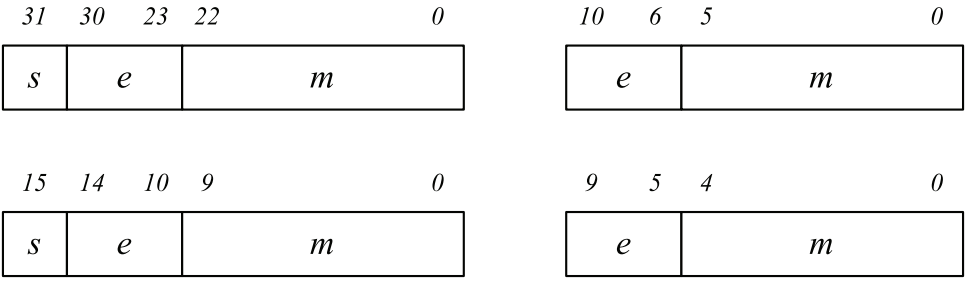

Рис. 12.15 ❖ Форматы значений с плавающей точкой в OpenGL

Для работы с этими текстурами в GLSL используются типы `sampler*` и функции `texture` и `textureProj`.

Кроме этого, OpenGL поддерживает также целочисленные ненормализованные текстуры – чтение из этих текстур возвращает именно те целые числа (со знаком или нет), которые хранятся в текстуре. Для чтения из таких текстур в шейдерах используются типы `isampler*` и `usampler*`. В табл. 12.7 приведены основные целочисленные форматы текстур.

Таблица 12.7. Форматы текстур с целочисленными значениями

Внутренний формат	Число компонент	Знаковость	Число бит на компоненту
GL_R8I	1	Да	8
GL_R8UI	1	Нет	8
GL_R16I	1	Да	16
GL_R16UI	1	Нет	16
GL_R32I	1	Да	32
GL_R32UI	1	Нет	32
GL_RG8I	2	Да	8
GL_RG8UI	2	Нет	8
GL_RG16I	2	Да	16
GL_RG16U	2	Нет	16
GL_RG32I	2	Да	32
GL_RG32UI	2	Нет	32
GL_RGB8I	3	Да	8

Таблица 12.7 (окончание)

Внутренний формат	Число компонент	Знаковость	Число бит на компоненту
GL_RGB8UI	3	Нет	8
GL_RGB16I	3	Да	16
GL_RGB16UI	3	Нет	16
GL_RGB32I	3	Да	32
GL_RGB32UI	3	Нет	32
GL_RGBA8I	4	Да	8
GL_RGBA8UI	4	Нет	8
GL_RGBA16I	4	Да	16
GL_RGBA16UI	4	Нет	16
GL_RGBA32I	4	Да	32
GL_RGBA32UI	4	Нет	32

Кроме этих форматов, также есть специальные форматы со значениями глубины или глубины-трафарета. Этими форматами являются GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, GL_DEPTH_COMPONENT32, GL_DEPTH_COMPONENT32F, GL_DEPTH24_STENCIL8 и GL_DEPTH32F_STENCIL8. Обратите внимание, что форматы GL_DEPTH_COMPONENT32F и GL_DEPTH32F_STENCIL8 хранят значения глубины в виде 32-битовых чисел с плавающей точкой.

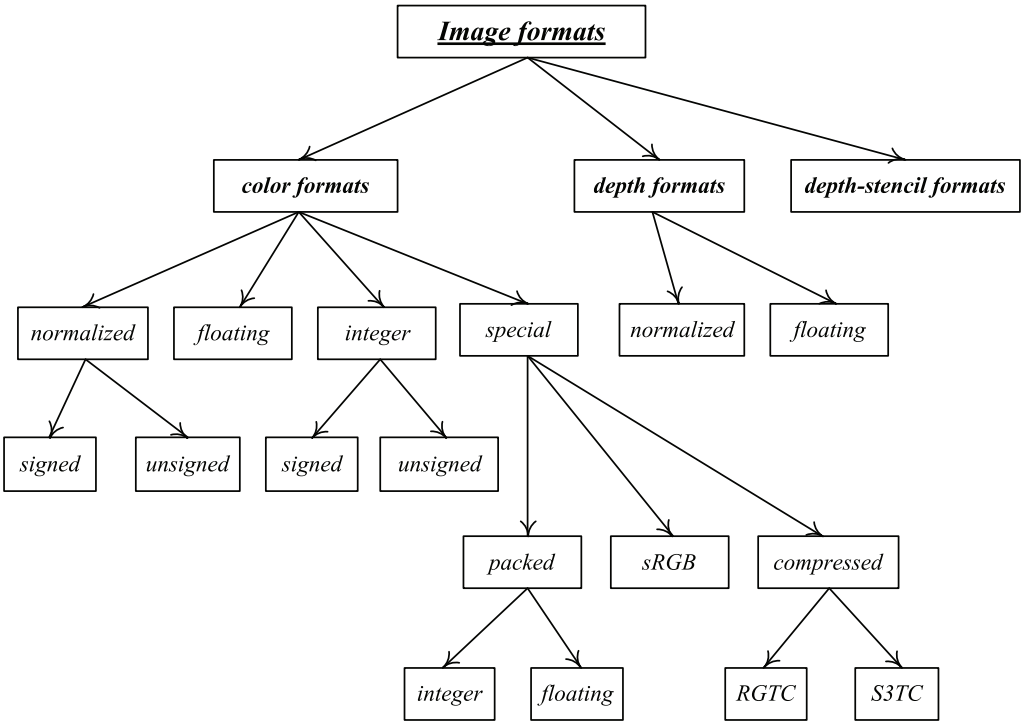

Рис. 12.16 ❖ Классификация форматов текстур в OpenGL

Работа с текстурами

Обратите внимание, что все ранее рассмотренные функции для работы с текстурами никак не связаны с тем, как именно текстура хранится в файле. Когда мы загружаем текселы для текстуры, то мы просто передаем на вход массив текселов требуемого размера и формата. Обычно при этом каждая компонента задается всего одним байтом. Тем самым для задания текстуры формата `GL_RGB` нам нужно по три байта на каждый тексел.

Однако когда изображения хранятся в файлах, то они хранятся обычно в специальных форматах. Кроме того, обычно при этом используются различные формы сжатия изображений. При этом используемый способ сжатия изображений может быть довольно сложным, так, например, широко распространенный формат `jpeg` использует преобразование Фурье.

Поэтому если мы хотим загружать изображения из файлов в текстуры, то нам нужен способ, позволяющий по содержимому файла получить требуемый нам массив текселов. При этом желательно уметь это делать для целого ряда распространенных форматов файлов. Все это может оказаться довольно сложным. Поэтому мы в примерах к данной книге будем использовать модифицированную библиотеку SOIL, поддерживающую целый ряд популярных форматов файлов. Ниже приводится пример создания текстуры в OpenGL при помощи этой библиотеки.

```
tex = SOIL_load_OGL_texture (
            fileName,
            SOIL_LOAD_AUTO,
            SOIL_CREATE_NEW_ID,
            SOIL_FLAG_POWER_OF_TWO       | SOIL_FLAG_MIPMAPS
            | SOIL_FLAG_DDS_LOAD_DIRECT );
```

Работа с шейдерами

Для обозначения программ, выполняемых на GPU, обычно используется термин *шейдер* (shader). Мы пока просто рассмотрим два основных типа шейдеров – вершинный и фрагментный.

В OpenGL шейдер – это тоже объект, и, как и остальные объекты, он идентифицируется беззнаковым целым числом. При этом, как и для текстур и буферов, значение 0 является зарезервированным и не соответствует ни одному шейдеру.

Для создания шейдера служит функция `glCreateShader`, получающая на вход тип создаваемого шейдера – вершинный (`GL_VERTEX_SHADER`) или фрагментный (`GL_FRAGMENT_SHADER`). Для уничтожения шейдеров служит функция `glDeleteShader`.

```
GLuint glCreateShader ( GLenum shaderType );
GLvoid glDeleteShader ( GLuint shader );
```

После того как соответствующий объект был создан, в него нужно загрузить исходный текст шейдера на языке GLSL и откомпилировать его. Для этого служат следующие функции:

```
void glShaderSource  ( GLuint shader, GLsizei count,
                       const char ** strings,
                       const GLint *lengths );
void glCompileShader ( GLuint shader );
```

Параметр *shader* задает сам шейдер, созданный функцией glCreateShader. Параметр *count* задает количество элементов (строк) в массивах *strings* и *lengths*. При этом *strings* является указателем на массив строк с текстом шейдера, а *lengths* задает массив длин этих строк.

Длины строк нужны на самом деле только тогда, когда сами строки не завершаются нулевым байтом. В противном случае вместо параметра *lengths* можно просто передать NULL. Обратите внимание, что строки, передаваемые через параметр *strings*, вообще говоря, никак не связаны с разбиением исходного кода на строки в текстовом редакторе. Поэтому мы будем передавать весь текст как одну строку, завершенную нулевым байтом.

После задания исходного кода шейдера его следует откомпилировать. В результате этой операции для шейдера устанавливается статус компиляции, равный GL_TRUE, если компиляция прошла без ошибок, и GL_FALSE в противном случае. Этот статус можно получить при помощи вызова функции glGetShaderiv с параметром GL_COMPILE_STATUS.

В случае ошибок компиляции все ошибки записываются в лог шейдера. Длину этого лога можно получить при помощи вызова glGetShaderiv с параметром GL_INFO_LOG_LENGTH. Сам текст лога получается при вызове glGetShaderInfoLog:

```
void glGetShaderInfoLog ( GLuint shader, GLsizei maxLength,
                          GLsizei * length, char * buf );
```

Ниже приводится функция, которая, получив тип шейдера и его исходный код, создает соответствующий шейдер, загружает в него исходный код и компилирует. В случае успешной компиляции возвращается идентификатор шейдера. При возникновении ошибок лог выводится на печать, шейдер уничтожается, и возвращается 0.

```
GLuint loadShader ( const char * source, GLenum type )
{
      GLuint shader = glCreateShader ( type );
      GLint status, length;

      if ( shader == 0 )
            return 0;

      glShaderSource  ( shader, 1, &source, NULL );   // передаем текст как одну строку
      glCompileShader ( shader );
      glGetShaderiv   ( shader, GL_COMPILE_STATUS, &status );

      if ( status != GL_TRUE )                        // в случае ошибки печатаем лог
      {                                               // получаем длину лога

            glGetShaderiv ( shader, GL_INFO_LOG_LENGTH, &length );

            char * buf = (char *) malloc ( length );  // выделяем память под лог

            glGetShaderInfoLog ( shader, length, NULL, buf );   // получаем текст лога

            printf ( "Compile failure.\n%s\n", buf );           // печатаем его
            free   ( buf );
```

```
            glDeleteShader ( shader );

            return 0;
      }

      return shader;
}
```

Таким образом, создаются вершинный и фрагментный шейдеры. Но на самом деле они не являются независимыми объектами – входы фрагментного шейдера должны соответствовать выходам вершинного. Поэтому в OpenGL вводится дополнительный объект – *программа*.

Программа объединяет в себе совместно используемые шейдеры. При этом выполняются все необходимые проверки на соответствие шейдеров друг другу. Кроме того, вся работа с `uniform`-переменными также идет не через отдельные шейдеры, а через программу.

Программа также идентифицируется беззнаковым целым числом. Для создания и уничтожения программ служат следующие функции:

```
GLuint glCreateProgram ();
void   glDeleteProgram ( GLuint program );
```

После того как программа была создана, к ней необходимо присоединить шейдеры и выполнить линковку. Для этого служат следующие функции:

```
void glAttachShader ( GLuint program, GLuint shader );
void glLinkProgram  ( GLuint program );
```

В результате линковки у программы выставляется соответствующий статус, который можно получить. В случае ошибок также в лог программы добавляются сообщения об ошибках. Ниже приводится фрагмент кода, создающий программу, подключающий вершинный и фрагментный шейдеры и выполняющий линковку. В случае ошибок печатается лог.

```
                           // загружаем вершинный шейдер из файла simple.vsh
GLuint vertexShader = loadShaderFromFile("simple.vsh", GL_VERTEX_SHADER);

if ( !vertexShader )
      return 1;

                           // загружаем фрагментный шейдер из файла simple.fsh
GLuint fragmentShader = loadShaderFromFile("simple.fsh", GL_FRAGMENT_SHADER);

if ( !fragmentShader )
      return 1;
                           // создаем программу и подключаем к ней
                           // загруженные шейдеры, линкуем программу
program = glCreateProgram ();

glAttachShader ( program, vertexShader   );
glAttachShader ( program, fragmentShader );
glLinkProgram  ( program );
glGetProgramiv ( program, GL_LINK_STATUS, &status );
```

```
if ( status != GL_TRUE )    // в случае ошибки печатаем лог
{
      int length;

      glGetProgramiv ( program, GL_INFO_LOG_LENGTH, &length );

      char * buf = (char *) malloc ( length );

      glGetProgramInfoLog ( program, length, NULL, buf );

      printf ( "Link failure.\n%s\n", buf );
      free   ( buf );

      return 0;
}
```

Теперь давайте рассмотрим работу с uniform-переменными. Под uniform-переменными понимаются некоторые значения, постоянные вдоль одного или нескольких примитивов. Примерами таких значений являются матрицы преобразования координат, координаты источника света, текстуры и многое другое. Вся эти величины описываются при помощи ключевого слова uniform.

```
uniform mat4 proj;
uniform mat4 mv;
uniform vec3 light;
```

При этом все шейдеры внутри одной программы имеют общее пространство uniform-переменных – эти переменные привязываются не к конкретному шейдеру, а ко всей программе. Соответственно, если одно и то же имя такой переменной используется сразу и в вершинном, и в фрагментном шейдерах, то тип обязан совпадать (иначе будет ошибка линковки).

Каждая uniform-переменная в программе имеет свой уникальный индекс, однозначно ее идентифицирующий. Для получения индекса переменной по ее имени служит следующая функция:

```
GLint glGetUniformLocation ( GLuint program, const char * name );
```

Эта функция возвращает индекс uniform-переменной или –1 в случае, когда uniform-переменной с таким именем нет.

Для задания значений для uniform-переменных служат функции glUniform* (для скаляров и векторов) и glUniformMatrix* (для матриц). Всего существует много вариантов этих функций для различных типов значений и числа компонент. Ниже мы приведем только несколько из них.

```
void glUniform1i     ( GLint location, GLint value );
void glUniform2f     ( GLint location, GLfloat x, GLfloat y );
void glUniform4fv    ( GLint location, const GLfloat * values );
void glUniformMatrix ( GLint location, GLsizei count,
                       GLboolean transpose, const GLfloat * values );
```

В функциях для задания матриц параметр *count* служит для того, чтобы можно было задать сразу несколько матриц (когда соответствующая переменная является массивом), параметр *transpose* задает, нужно ли транспонировать матрицу перед записью.

В этих функциях не указано, к какой программе они относятся – все они изменяют значения uniform-переменных для текущей программы, которая задается при помощи функции glUseProgram. Обратите внимание, что при выводе примитивов используется именно текущая программа. Команда glUseProgram делает заданную программу текущей, т. е. выводимые примитивы будут выводиться при помощи этой программы. Можно в качестве идентификатора программы передать 0, это значит, что текущей программы нет (и вывод примитивов невозможен).

```
void glUseProgram ( GLuint program );
```

Обратите внимание на способ, при помощи которого текстуры передаются в шейдер. Фактически каждая uniform-переменная, соответствующая текстуре, рассматривается как целочисленная переменная, содержащая номер текстурного блока, в котором выбрана в качестве текущей заданная текстура.

Готовое приложение

Сейчас мы соберем все, что у нас уже было описано в этой главе, и построим нашу первую программу на OpenGL. Она будет основана на классе GlutWindow, мы просто переопределим необходимые методы. Ниже приводится описание соответствующего класса.

```
class TestWindow : public GlutWindow
{
      GLuint      program;
      GLuint      vertexShader, fragmentShader;
      GLuint      vao;
      GLuint      vbo;
      int         status;

public:
      TestWindow () : GlutWindow  ( 200, 200, 400, 400, "Test" )
      {
            createProgram ();

            glUseProgram ( program );

            createBuffers ();

            setAttrPtr ( "pos", 3, VERTEX_SIZE, (void *) 0 );
      }

      void  createBuffers ()
      {
            glGenVertexArrays ( 1, &vao );
            glBindVertexArray ( vao );
            glGenBuffers      ( 1, &vbo );
            glBindBuffer      ( GL_ARRAY_BUFFER, vbo );
            glBufferData      ( GL_ARRAY_BUFFER, NUM_VERTICES * VERTEX_SIZE,
                                vertices, GL_STATIC_DRAW );
      }
```

```
      void   createProgram      ();
      GLuint loadShaderFromFile ( const char * fileName, GLenum type );
      GLuint loadShader         ( const char * source, GLenum type );

      void   setAttrPtr ( const char * name, int numComponents,
                          GLsizei stride, void * ptr,
                          GLenum type = GL_FLOAT, bool normalized = false )
      {
            int   loc = glGetAttribLocation ( program, name );

            if ( loc < 0 )
                  exit ( 1 );

            glVertexAttribPointer ( loc, numComponents, type,
                                    normalized ? GL_TRUE : GL_FALSE,
                                    stride, (const GLvoid*) ptr );
            glEnableVertexAttribArray ( loc );
      }

      virtual     void  redisplay ()
      {
            glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

            glBindVertexArray ( vao );
            glDrawArrays      ( GL_TRIANGLES, 0, NUM_VERTICES );
            glBindVertexArray ( 0 );
      }

      virtual void keyTyped ( unsigned char key, int modifiers, int, int )
      {
            if ( key == 27 || key == 'q' || key == 'Q' )
                  exit ( 0 );
      }
};
```

Кроме того, нам понадобятся вершинный и фрагментный шейдеры. Ниже приводится простейший вершинный шейдер – он просто получает на вход трехмерные координаты вершины, достраивает до четырехмерного вектора и передает их через встроенную переменную.

```
layout(location = 0) in vec3 pos;

void main (void)
{
      gl_Position = vec4 (pos, 1.0);
}
```

Задача фрагментного шейдера еще проще – нужно просто выдать некоторый цвет, в нашем случае желтый.

```
out vec4     color;      // выходное значение

void main (void)
{                        // выдаем желтый цвет
      color = vec4 ( 0.0, 1.0, 0.0, 1.0 );
}
```

Вспомогательные классы и работа с ними

Для удобства работы мы завернем большинство необходимых объектов OpenGL в классы языка C++. Это заметно упростит код и сделает его понимание проще. Ниже мы рассмотрим основные классы, которые будут использоваться далее.

Для работы с текстурами следует использовать специальный класс `Texture`. Методы этого класса позволяют создавать как пустые текстуры заданного типа, так и готовые текстуры по содержимому файла (или файлов). В результате сразу создается готовая к употреблению текстура OpenGL. Обратите внимание, что загружать текстуры можно только после создания контекста OpenGL.

Наиболее важными методами этого класса являются методы `load2D`, `load3D` и `loadCubemap`. Для привязывания текстуры к заданному текстурному блоку служит метод `bind`, метод `buildMipmaps` строит все уровни для пирамидального фильтрования (по умолчанию для текстуры всегда строятся все уровни пирамиды и используется пирамидальное фильтрование). Ниже приводится фрагмент кода для загрузки и использования текстуры при помощи этого класса.

```
class MeshWindow : public GlutRotateWindow
{
      Program     program;
      BasicMesh * mesh;
      Texture     tex;
      glm::vec3   eye;

public:
      MeshWindow () : GlutRotateWindow (200, 200, 400, 400, "Textured mesh" )
      {
            std::string texName = "../../Textures/Fieldstone.dds";

            if ( !tex.load2D ( texName.c_str () ) )
                  exit ( "Error loading texture %s\n", texName.c_str () );
```

Самым важным для использования является класс `Program`. Данный класс представляет собой программу, собранную из отдельных шейдеров на GLSL. Конструктор данного класса (как и конструкторы других используемых классов) практически ничего не делает и поэтому может вызываться до создания контекста OpenGL.

Для загрузки шейдеров в классе `Program` используется метод `loadProgram`. Этот метод получает на вход имя файла, содержащего исходный текст всех шейдеров. Начало каждого шейдера обозначено при помощи строки, начинающейся с «--», далее идет тип шейдера. Это позволяет разместить все связанные друг с другом шейдеры вместе. Ниже приводится пример такого шейдера.

```
-- vertex
#version 330 core

layout(location = 0) in vec3 pos;

void main(void)
{
      gl_Position = vec4 ( pos, 1.0 );
}
```

```
-- fragment
#version 330 core

out vec4    color;            // выходное значение

void main ()
{
      color = vec4 ( 1, 1, 0, 1 );
}
```

Если метод `loadProgram` возвращает значение `false`, то при загрузке, компиляции или линковке были ошибки. В этом случае полный лог с ошибками доступен при помощи метода `getLog`.

Для того чтобы сделать программу текущей, служит метод `bind`. Метод `unbind` выбирает в качестве текущей программы программу 0 (что значит, что в данный момент нет текущей программы). Метод `setAttrPtr` служит для задания параметров атрибутов вершин.

Также данный класс содержит методы для задания `uniform`-переменных. Для задания `uniform`-матриц служат перегруженные методы `setUniformMatrix`. Для задания `uniform`-векторов служат перегруженные методы `setUniformVector`. Для задания целочисленных `uniform`-переменных и `uniform`-переменных с плавающей точкой служат методы `setUniformInt` и `setUniformFloat`. Для задания текстур служат методы `setTexture` –позволяют для имени sampler-переменной задать соответствующий ей текстурный блок.

Объекты VAO представлены при помощи класса `VertexArray`. Метод `create` служит для создания соответствующего объекта, метод `bind` делает его текущим, метод `unbind` делает текущим VAO с идентификатором 0.

Для представления VBO (вершинных и индексных буферов) используется класс `VertexBuffer`. Методы `create` и `bind` создают буфер и делают его текущим для заданного типа. Для задания данных в буфере служит метод `setData`.

Ниже приводится пример простейшей программы с использованием рассмотренных выше классов. Он просто выводит на экран желтый треугольник.

```
#define     NUM_VERTICES      3        // задаем вершины - число и размер вершины в байтах
#define     VERTEX_SIZE       (3*sizeof(float))

static const float vertices [] =       // сами координаты вершин
{
       -1.0f, -1.0f, 0.0f,
        0.0f,  1.0f, 0.0f,
        1.0f, -1.0f, 0.0f
};

class TestWindow : public GlutWindow   // главное окно
{
      Program           program;
      VertexArray       vao;
      VertexBuffer      buf;

public:
      TestWindow () : GlutWindow ( 100, 200, 500, 500, "Test" )
      {
                                       // загружаем программу
```

```
        if ( !program.loadProgram ( "simple.glsl" ) )
              exit ( "Error building program: %s\n",
                     program.getLog ().c_str () );

        program.bind ();           // делаем программу текущей
        vao.create   ();           // создаем VAO
        vao.bind     ();           // делаем созданный VAO текущим
        buf.create   ();           // создаем вершинный буфер
        buf.bind     ( GL_ARRAY_BUFFER );
                                   // задаем данные в вершинном буфере
        buf.setData  ( NUM_VERTICES * VERTEX_SIZE, vertices,
                       GL_STATIC_DRAW );
                                   // задаем свойства атрибута в программе
        program.setAttrPtr ( "pos", 3, VERTEX_SIZE, (void *) 0 );

        buf.unbind     ();
        vao.unbind     ();
        program.unbind ();
    }

    virtual     void  redisplay ()
    {
        glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        program.bind ();
        vao.bind     ();

        glDrawArrays ( GL_TRIANGLES, 0, NUM_VERTICES );

        vao.unbind     ();
        program.unbind ();
    }
};

int main ( int argc, char * argv [] )
{
    GlutWindow::init ( argc, argv );

    TestWindow win;

    return win.run ();
}
```

Для представления сложных объектов, заданных наборами треугольников (mesh), мы будем использовать класс `BasicMesh`. Каждая вершина в этом классе задана при помощи структуры `BasicVertex` и содержит трехмерные координаты, текстурные координаты, векторы нормали, касательный вектор и бинормаль. Ниже приводится фрагмент из соответствующего заголовочного файла.

```
struct  BasicVertex
{
    glm::vec3   pos;          // координаты в пространстве
    glm::vec2   tex;          // текстурные координаты
    glm::vec3   n;            // нормаль
    glm::vec3   t, b;         // касательный вектор и бинормаль
};
```

```
class BasicMesh
{
      int          numVertices;  // число вершин
      int          numTriangles; // число треугольников
      VertexArray  vao;          // все привязки для данного меша
      VertexBuffer vertBuf;      // данные в вершинах
      VertexBuffer indBuf;       // индексный буфер
      std::string  name;

public:
      BasicMesh(BasicVertex * vertices, const int * indices, int nv, int nt);

      void  render ();           // вывести данный меш
                                 // вывести меш в режиме дублирования геометрии primCount раз
      void  renderInstanced ( int primCount );
```

Также есть набор функций, строящий такие меши для базовых объектов – сферы, тора, куба и т. п. Кроме того, функция `loadMesh` загружает меш из большого числа различных форматов (при помощи библиотеки `assimp`).

```
BasicMesh * createSphere ( const glm::vec3& org, float radius, int n1, int n2 );
BasicMesh * createQuad   ( const glm::vec3& org,
                             const glm::vec3& dir1, const glm::vec3& dir2 );
BasicMesh * createBox    ( const glm::vec3& pos, const glm::vec3& size,
                             const glm::mat4 * mat = nullptr, bool invertNormal = false );
BasicMesh * createTorus  ( float r1, float r2, int n1, int n2 );
BasicMesh * createKnot   ( float r1, float r2, int n1, int n2 );
BasicMesh * loadMesh     ( const char * fileName, float scale = 1.0f );
```

Глава 13

Простейшие эффекты

В этой главе мы рассмотрим реализацию при помощи OpenGL ряда простейших визуальных эффектов. Эти эффекты не требуют сложных шейдеров и довольно просты для понимания и реализации.

Отражение относительно плоскости

Одним из самых простых эффектов является отражение в плоском зеркале. Для начала мы рассмотрим случай, когда в качестве такого зеркала выступает бесконечная плоскость (рис. 13.1).

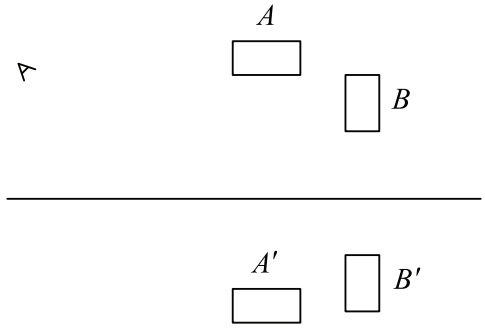

Рис. 13.1 ❖ Отражение объектов плоским зеркалом

Самым простым вариантом было явное отражение всех объектов относительно заданной плоскости и вывод получившихся отраженных объектов. Однако это не самый оптимальный вариант. На практике обычно поступают иначе. Вместо того чтобы выводить отраженные объекты при помощи стандартной камеры, мы просто отражаем саму камеру и выводим неизмененные объекты при помощи этой отраженной камеры (рис. 13.2).

Поскольку камера задается своим положением и тремя ортонормированными векторами, то для ее отражения достаточно просто отразить положения камеры и три базовых вектора. Обратите внимание, что при отражении правая тройка векторов переходит в левую, и наоборот. В используемом нами классе `Camera` уже есть метод `reflect`, служащий для отражения камеры относительно заданной плоскости.

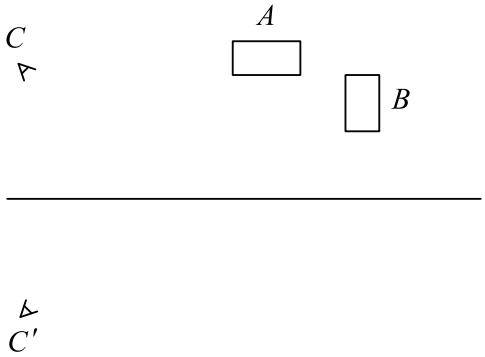

Рис. 13.2 ❖ Рендеринг сцены при помощи отраженной камеры

Теперь давайте рассмотрим более общий случай, когда отражение производится относительно не всей плоскости, а только некоторой грани, лежащей в этой плоскости.

Общая схема с отражением камеры по-прежнему работает, однако мы можем столкнуться со следующей проблемой. Возможна ситуация, когда отражение вылезает за границы грани, относительно которой мы производим отражение. То есть мы получаем «висящее в воздухе» отражение.

Для того чтобы избежать подобной ситуации, обычно используется отсечение при помощи буфера трафарета. Перед выводом отражения мы сначала осуществляем рендеринг самой отражающей грани *только* в буфер трафарета.

Поскольку мы не хотим, чтобы при этом были изменены буферы цвета и глубины, то мы запрещаем запись в эти буферы при помощи команд `glColorMask` и `glDepthMask`. Также мы включаем тест трафарета, так чтобы он был всегда пройден (режим `GL_ALWAYS`) и при выполнении теста трафарета производилась запись 1 в буфер трафарета. Тест глубины мы при этом не выключаем.

```
glColorMask   ( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
glDepthNask   ( GL_FALSE );
glStencilFunc ( GL_ALWAYS, 1, 0xFF );
glStencilOp   ( GL_KEEP, GL_KEEP, GL_REPLACE );
```

После этого мы выводим нашу зеркальную грань. При этом во все пикселы, соответствующие этой грани, будет записано новое значение трафарета, равное 1. Далее мы снова разрешаем запись в буфера цвета и глубины. Но тест трафарета мы настраиваем уже по-другому. На этот раз тест будет пройден только для тех пикселов, для которых в буфере трафарета записана 1. Сам буфер трафарета при этом меняться не будет.

```
glColorMask   ( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
glDepthNask   ( GL_TRUE );
glStencilFunc ( GL_EQUAL, 1, 0xFF );
glStencilOp   ( GL_KEEP, GL_KEEP, GL_KEEP );
```

Теперь мы можем смело вывести сцену при помощи отраженной камеры – все части отражения, которые не попадают в наше зеркало, будут отброшены тестом трафарета.

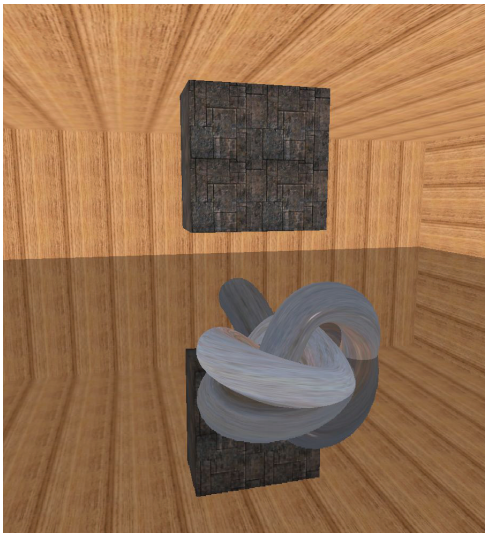

Рис. 13.3 ❖ Пример отражения

Имитация отражения окружающей среды и преломления

В ряде случаев стоит задача *имитации* отражения объектом сложной формы. Существенным здесь является именно то, что речь идет об имитации. Дело в том, что точный расчет отражения требует трассировки лучей, что в задачах реального времени обычно не приемлемо.

Поэтому для имитации отражения часто используют кубические текстуры. В такую кубическую текстуру заносится (тем или иным способом) вид из некоторой точки на всю сцену. Зачастую такой вид считается заранее, и он вообще не учитывает возможных изменений в сцене.

После этого, когда нам нужно найти отражение в заданной точке P при заданном векторе v к камере и нормали n, мы поступаем следующим образом. По этим векторам мы строим отраженный вектор r и по нему читаем значение из нашей кубической текстуры (рис. 13.4).

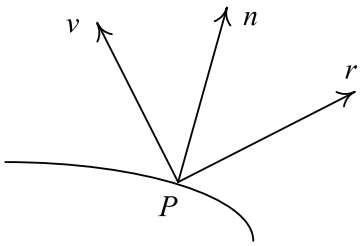

Рис. 13.4 ❖ Пример расчета отражения

Строго говоря, этот прием будет давать точное отражение только в том случае, когда точка P совпадает с той точкой, для которой строилась используемая кубическая текстура. Однако обычно считается, что текстура несильно зависит от

точки и пользователь все равно не заметит неточность отражения. Этот прием используется в большом числе приложений и игр и называется *имитация отражения окружающей среды* (environment mapping).

Для реализации данного метода мы в вершинном шейдере, используя координаты камеры и координаты текущей вершины, строим вектор *v*. По нему при помощи вектора нормали строится отраженный вектор *r*, который передается во фрагментный шейдер. Обратите внимание на использование стандартной функции `reflect` для нахождения отраженного вектора.

```
#version 330 core

layout ( location = 0 ) in vec3 pos;      // координаты вершины
layout ( location = 2 ) in vec3 normal;   // нормаль в вершине

out vec3 n;                          // выходная нормаль
out vec3 v;                          // направление к камере

uniform mat4 proj;                   // матрица проектирования
uniform mat4 mv;                     // модельно-видовая матрица
uniform mat3 nm;                     // матрица преобразования направлений
uniform vec3 eye;

void main(void)
{
      vec4  p = mv * vec4 ( pos, 1.0 );

      gl_Position  = proj * p;
      n            = normalize ( nm * normal );
      v            = normalize ( eye - p.xyz );
}
```

Задача фрагментного шейдера крайне проста – получив отраженный вектор *r*, нужно прочесть соответствующее значение цвета из кубической текстуры и вернуть его как цвет в точке. Обратите внимание на нормирование векторов *n* и *v* во фрагментном шейдере. Это нужно потому, что в результате интерполяции единичных векторов мы можем получить векторы, не являющиеся уже единичными.

```
#version 330 core

uniform samplerCube      image;

in  vec3 n;       // проинтерполированное значение нормали
in  vec3 v;       // проинтерполированное направление на камеру
out vec4 color;

void main(void)
{                 // вычисляем отраженный вектор и читаем по нему цвет
      color = texture ( image, reflect ( normalize(v), normalize(n) ) );
}
```

Рис. 13.5 ❖ Результат применения имитации отражения окружающей среды

Этот же прием можно с успехом применить и для имитации преломления. К сожалению, просчитать полностью преломление с учетом как точки входа луча, так и точки выхода без трассировки невозможно. Поэтому обычно рассматривается преломление только в точке входа луча – по аналогии с отраженным вектором строится преломленный вектор *t*, и по нему извлекается значение цвета из кубической текстуры.

```
#version 330 core

layout ( location = 0 ) in vec3 position;
layout ( location = 1 ) in vec2 texCoord;
layout ( location = 2 ) in vec3 normal;
layout ( location = 3 ) in vec3 tangent;
layout ( location = 4 ) in vec3 binormal;

uniform mat4 proj;      // матрица проектирования
uniform mat4 mv;        // модельно-видовая матрица
uniform mat3 nm;        // матрица для преобразования направлений
uniform vec3 eye;       // положение камеры

out vec3  n;
out vec3  v;

void main(void)
{
      vec4  p = mv * vec4 ( position, 1.0 );

      gl_Position  = proj * p;
      n            = normalize ( nm * normal );
      v            = normalize ( eye - p.xyz );
}
```

Ниже приводится и фрагментный шейдер для моделирования преломления.

```
#version 330 core

in  vec3 n;
in  vec3 v;
out vec4 color;
```

```
uniform samplerCube cubeMap;

void main(void)
{
      const float eta = 0.9;

      vec3  v2 = normalize ( v );
      vec3  n2 = normalize ( n );
      vec3  t  = refract ( -v2, n2, eta );

      color = texture( cubeMap, t );
}
```

Можно, используя кубическую текстуру, найти сразу два вектора – один для отражения, другой для преломления. После чего по этим векторам находятся два цвета и смешиваются при помощи коэффициента Френеля.

```
#version 330 core

in    vec3 n;
in    vec3 v;
out   vec4 color;

uniform samplerCube cubeMap;

void main(void)
{
      const float eta = 0.9;

      vec3  v2 = normalize ( v );
      vec3  n2 = normalize ( n );
      vec3  r  = reflect ( v2, n2 );
      vec3  t  = refract ( -v2, n2, eta );
      float f  = pow ( max ( 0, dot ( v2, n2 ) ), 5.0 );

      color = texture( cubeMap, r )*(1.0-f) + texture( cubeMap, t )*f;
}
```

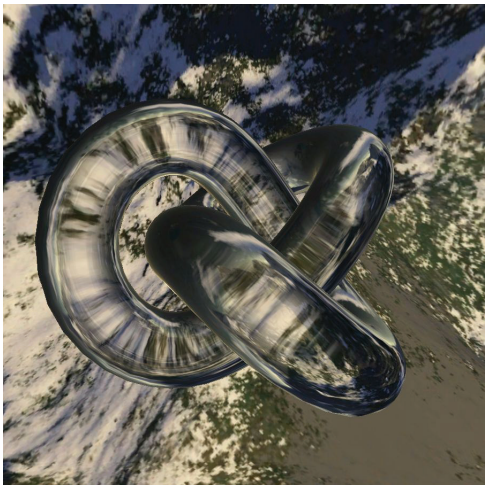

Рис. 13.6 ❖ Одновременное отражение и преломление

Можно повысить реалистичность преломления, если учесть тот факт, что коэффициент преломления η зависит от длины волны. Именно этим и объясняется разложение стеклянной призмой белого цвета. Достаточно будет вместо одного коэффициента преломления использовать три – по одному для каждого из трех базовых цветов.

Каждый из этих коэффициентов дает свой преломленный луч. Используя эти лучи, мы извлекаем из кубической текстуры сразу три цвета. От каждого из этих цветов мы берем только одну цветовую компоненту.

```
-- vertex
#version 330 core

layout ( location = 0 ) in vec3 position;
layout ( location = 1 ) in vec2 texCoord;
layout ( location = 2 ) in vec3 normal;
layout ( location = 3 ) in vec3 tangent;
layout ( location = 4 ) in vec3 binormal;

uniform mat4 proj;
uniform mat4 mv;
uniform mat3 nm;
uniform vec3 eye;

out vec3  n;
out vec3  v;

void main(void)
{
      vec4  p = mv * vec4 ( position, 1.0 );

      gl_Position  = proj * p;
      n            = normalize ( nm * normal );
      v            = normalize ( eye - p.xyz );
}

-- fragment

#version 330 core

in   vec3 n;
in   vec3 v;
out  vec4 color;

uniform samplerCube cubeMap;

void main(void)
{
      const vec3 eta = vec3 ( 0.9, 0.92, 0.94 );

      vec3 v2 = normalize ( v );
      vec3 n2 = normalize ( n );
      vec3 r  = reflect ( v2, n2 );
```

```
    vec3 tRed   = refract ( -v2, n2, eta.r );
    vec3 tGreen = refract ( -v2, n2, eta.g );
    vec3 tBlue  = refract ( -v2, n2, eta.b );
    vec3 refractionColorRed   = texture ( cubeMap, tRed   ).rgb;
    vec3 refractionColorGreen = texture ( cubeMap, tGreen ).rgb;
    vec3 refractionColorBlue  = texture ( cubeMap, tBlue  ).rgb;
    vec3 refractionColor      = vec3 ( refractionColorRed.r,
                                       refractionColorGreen.g,
                                       refractionColorBlue.b );

    float f  = pow ( max ( 0, dot ( v2, n2 ) ), 5.0 );

    color = texture( cubeMap, r )*(1.0-f) +
            vec4 ( refractionColor, 1.0 )*f;
}
```

Точечные спрайты. Системы частиц

В большом числе графических эффектов возникает необходимость использования текстурированных прямоугольников, параллельных плоскости экрана. В англоязычной литературе для них используют термин *billboard*. Мы уже рассматривали, как можно получать четырехугольники из точек при помощи геометрических шейдеров. Однако в этом разделе мы рассмотрим более ограниченный, но и заметно более простой способ получения таких прямоугольников под названием *точечные спрайты* (point sprite).

OpenGL может при выводе набора точек (`GL_POINTS`) на самом деле вместо каждой точки вывести квадрат с центром в этой точке и заданным в вершинном шейдере размером. При растеризации таких квадратов каждый получающийся фрагмент автоматически получает соответствующие ему текстурные координаты через встроенную в GLSL переменную `gl_PointCoord` (рис. 13.7).

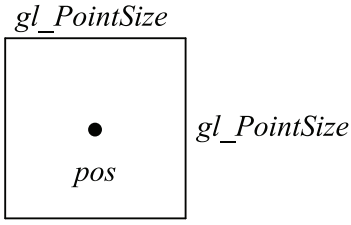

Рис. 13.7 ❖ Точечный спрайт

Для того чтобы вершинный шейдер мог задавать размер для точки (в пикселах) через запись во встроенную переменную `gl_PointSize`, необходимо разрешить соответствующую возможность при помощи следующего вызова:

```
glEnable ( GL_PROGRAM_POINT_SIZE );
```

Одним из наиболее частых применений точечных спрайтов является рендеринг *систем частиц* (particle system). Под системой частиц обычно понимается совокупность большого числа крайне простых объектов. Движение (а также из-

менение всех остальных параметров) каждого такого объекта описывается очень простым законом. При рендеринге эти объекты обычно выводятся как прямоугольники, параллельные экрану, или как точечные спрайты. Системы частиц используются для реализации большого числа различных эффектов, таких как огонь, дым, взрывы, дождь и т. п.

Как правило, каждая такая частица (объект) обычно описывается своими координатами, скоростью, цветом, размером. Обычно время жизни частицы ограничено, поэтому частица содержит оставшееся ей время жизни. Из соображений быстродействия удобно все параметры всех частиц представлять как массив структур, размещенный в вершинном буфере (VBO). При этом, когда частица «умирает», она либо помечается как мертвая, либо сразу же переиспользуется. Таким образом, в процессе анимации и рендеринга системы частиц не происходит выделений и освобождений памяти.

Сама анимация частиц может осуществляться как на CPU, так и на GPU. В последнем случае обычно используют уже рассмотренное преобразование обратной связи или же вычислительные шейдеры (рассматриваемые в главе 20). Предпочтительнее использовать анимацию на стороне GPU, так как в этом случае пропадает необходимость в постоянной пересылке данных с CPU на GPU.

В качестве примера мы рассмотрим систему частиц, моделирующую огонь. Время жизни каждой частицы ограничено – она начинает жизнь в нижней точке и постепенно поднимается вверх, меняя при этом свой цвет, увеличиваясь в размерах и становясь более прозрачной.

Как только время жизни частицы заканчивается, она снова возрождается в нижней точке огня. Для того чтобы все частицы были равномерно распределены во времени, у каждой частицы есть своя фаза. Ниже приводится вершинный и фрагментны шейдеры, реализующие эффект огня – цвет и размер частицы плавно изменяются с течением времени. По истечении срока своей жизни частица умирает и вновь возрождается.

```
--vertex

#version 330 core

layout ( location = 0 ) in vec4 position;

uniform mat4 proj;
uniform mat4 mv;
uniform mat3 nm;
uniform vec3 eye;
uniform float time;

out float phase;

const vec3  velocity = vec3 ( 0.0, 0.0, 2.0 );
const float period   = 4.0;

void main()
{
      vec3  pos = position.xyz;
```

```
      phase        = fract ( position.w + time / period );
      gl_Position  = proj * mv * vec4 ( pos + phase * velocity, 1.0 );
      gl_PointSize = 8.0 + 20.0 * phase;
}

--fragment

#version 330 core

uniform sampler2D sprite;
uniform sampler2D colorMap;

in float phase;
out vec4 color;

void main ()
{
      vec4  c1 = texture ( sprite, gl_PointCoord );
      vec4  c2 = texture ( colorMap, vec2 ( 1.0 - phase, 1.0 ) );

      color = (1.0 - phase) * vec4 ( c1.rgb * c2.rgb, 1.0 );
}
```

Ниже также приводится конструктор соответствующего класса окна и его метод redisplay на C++.

```
FireWindow () : GlutWindow ( 200, 100, 900, 900, "Fire" )
{
      if ( !program.loadProgram ( "fire.glsl" ) )
            exit ( "Error building program: %s\n",
                   program.getLog ().c_str());

      if ( !sprite.load2D ( "Fire.bmp" ) )
            exit ( "Error loading texture\n" );

      if ( !colorMap.load2D ( "Flame.tga" ) )
            exit ( "Error loading texture\n" );

      for ( int i = 0; i < NUM_VERTICES; i++ )
            vertices [i] = glm::vec4 ( randUniform (-1,1),
                                       randUniform (-1,1),
                                       0.0, randUniform () );

      vao.create  ();
      vao.bind    ();
      buf.create  ();
      buf.bind    ( GL_ARRAY_BUFFER );
      buf.setData ( NUM_VERTICES * VERTEX_SIZE, vertices, GL_STATIC_DRAW );

      program.setAttrPtr ( 0, 4, VERTEX_SIZE, (void *) 0 );

      buf.unbind     ();
      vao.unbind     ();
```

```
    program.bind ();
    program.setUniformVector ( "eye",      eye );
    program.setTexture       ( "sprite",   0   );
    program.setTexture       ( "colorMap", 1   );
}
void redisplay ()
{
    glm::mat4      mv = glm::mat4 ( 1 );
    glm::mat3      nm = glm::mat3 ( 1 );

    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    sprite.bind              ( 0 );
    colorMap.bind            ( 1 );
    program.bind             ();
    program.setUniformMatrix ( "mv",  mv );
    program.setUniformFloat  ( "time", getTime () );
    vao.bind                 ();

    glEnable     ( GL_PROGRAM_POINT_SIZE );
    glEnable     ( GL_BLEND );
    glDisable    ( GL_DEPTH_TEST );
    glBlendFunc  ( GL_ONE, GL_ONE );
    glDrawArrays ( GL_POINTS, 0, NUM_VERTICES );
    glDisable    ( GL_BLEND );

    vao.unbind      ();
    program.unbind  ();
    sprite.unbind   ();
    colorMap.unbind ();
}
```

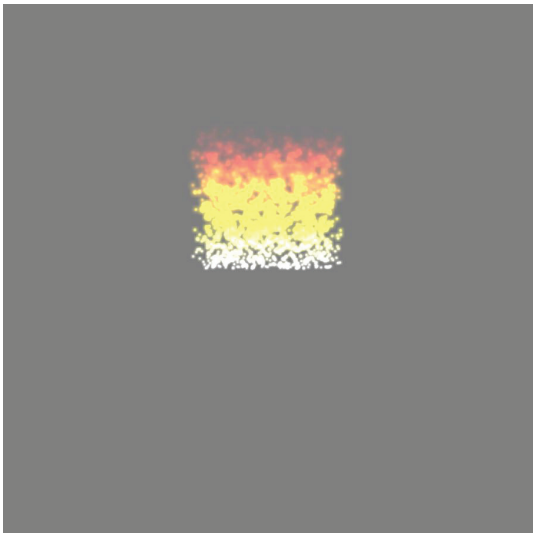

Рис. 13.8 ❖ Изображения огня через систему частиц

Проективное текстурирование

Проективное текстурирование довольно часто возникает в целом ряде случаев. Проще всего его можно понять следующим образом: представьте, что есть проектор, проектирующий некоторую картинку на всю сцену (рис. 13.9).

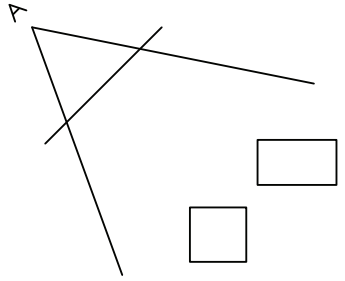

Рис. 13.9 ❖ Проектирование картинки на сцену

При использовании проективного текстурирования мы, как обычно, читаем данные из двухмерной текстуры. Однако в этом случае мы используем трехмерные текстурные координаты (s, t, r). Возвращаемый тексел соответствует двухмерным координатам $(s/r, t/r)$. Обычно в этом случае используется встроенная функция `textureProj`.

```
vec4 textureProj ( sampler2D sampler, vec3 coord [, float bias] );
```

Если у нас есть точка p на поверхности объекта, то для получения проектируемого на эту точку тексела координаты этой точки нужно умножить на следующие три матрицы.

Первая матрица (V) переводит координаты точки в систему координат, связанную с проектором. Для этого обычно используется функция `glm::lookAt`. После ее применения мы получаем координаты точки в системе координат проектора.

Вторая матрица (P) – это проектирующая матрица со своим углом обзора, соотношением сторон, ближней и дальней плоскостями отсечения. Для ее получения обычно используется функция `glm::persective`.

Однако после применения этих трех матриц и деления новые текстурные координаты будут лежать в диапазоне [–1, 1]. Но для индексации в текстуру нам нужно перевести их в диапазон [0, 1]. Именно для этого и используется третья матрица (B), задаваемая следующей формулой:

$$B = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1.0 \end{pmatrix}.$$

Обычно все эти матрицы вычисляются на CPU, и их произведение передается в шейдер как `uniform`-переменные. Ниже приводятся вершинный и фрагментный шейдеры для простого примера, демонстрирующего использование перспективного текстурирования.

```
-- vertex
#version 330 core

layout(location = 0) in vec3 pos;
layout(location = 1 ) in vec2 tex;
layout(location = 2) in vec3 normal;

uniform mat4 proj;
uniform mat4 mv;
uniform mat4 projMat;

out vec2 tx;
out vec4 projCoord;

void main(void)
{
      tx          = tex;
      projCoord   = projMat * mv * vec4 ( pos, 1.0 );
      gl_Position = proj    * mv * vec4 ( pos, 1.0 );
}

-- fragment
#version 330 core

in vec2 tx;
in vec4 projCoord;

out vec4 color;

uniform sampler2D image;
uniform sampler2D projMap;

void main ()
{
      vec4 c1 = texture     ( image, tx );
      vec4 c2 = textureProj ( projMap, projCoord );

      color = 0.5*c1 + 0.5*c2;
}
```

Ниже приводятся конструктор и метод redisplay соответствующего класса окна.

```
TestWindow() : GlutRotateWindow (200, 100, 900, 900,
                                 "Projective texturing")
{
      mesh = loadMesh ( "../../Models/teapot.3ds", 0.1f );

      if ( mesh == NULL )
            exit ("Error loading mesh" );

      std::string texName = "../../Textures/Fieldstone.dds";

      if ( !tex.load2D ( texName.c_str () ) )
```

```
            exit ( "Error loading texture %s\n", texName.c_str () );

      if ( !projMap.load2D ( "../../Textures/flower.png" ) )
            exit ( "Error loading textures\n" );

      if ( !program.loadProgram ( "projective.glsl" ) )
            exit ( "Error building program: %s\n",
                   program.getLog ().c_str());

      projMap.bind    ();
      projMap.setWrapAll ( GL_CLAMP_TO_BORDER );
      projMap.unbind  ();

      program.bind ();

      eye      = glm::vec3 ( 10, 0, 0 );
      lightDir = glm::vec3 ( 1, 1, -1 );
      projPos  = glm::vec3 ( 0, 0, 5 );

      program.setUniformFloat  ( "kd", kd );
      program.setUniformFloat  ( "ka", ka );
      program.setUniformVector ( "lightDir", lightDir );
      program.setUniformVector ( "eye",      eye );
      program.setUniformVector ( "projPos",  projPos );
      program.setTexture       ( "image",    0 );
      program.setTexture       ( "projMap",  1 );
}

void redisplay ()
{
      glm::mat4   mv = getRotation ();
      glm::mat4   bias = glm::mat4( glm::vec4(0.5f,0.0f,0.0f,0.0f),
                                    glm::vec4(0.0f,0.5f,0.0f,0.0f),
                                    glm::vec4(0.0f,0.0f,0.5f,0.0f),
                                    glm::vec4(0.5f,0.5f,0.5f,1.0f) );
      glm::mat4   v = glm::lookAt      ( projPos, glm::vec3 ( 0, 0, 0 ),
                                         glm::vec3 ( 0, 1, 0 ) );
      glm::mat4   p = glm::perspective ( toRadians ( 30.0f ),
(float) getWidth () / (float) getHeight (), 0.01f, 20.0f );

      projMat = bias * p * v;

      glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

      tex.bind     ( 0 );
      projMap.bind ( 1 );
      program.bind ();
      program.setUniformMatrix ( "mv", mv );
      program.setUniformMatrix ( "projMat", projMat );

      mesh -> render ();

      program.unbind ();
}
```

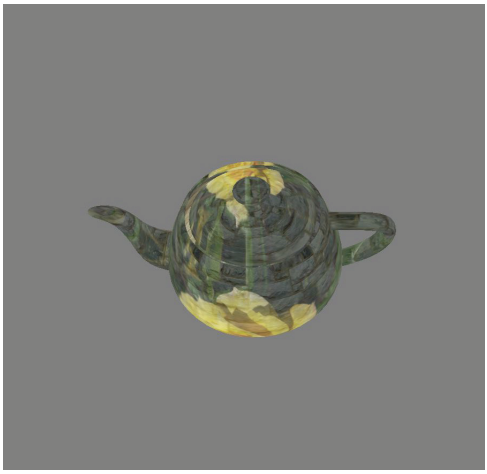

Рис. 13.10 ❖ Проективное текстурирование

Реализация основных моделей освещения

Давайте рассмотрим, как можно при помощи шейдеров реализовать основные модели освещения из главы 9. В каждой из этих моделей у нас фигурировали векторы (такие как n, l, v и другие), на основе которых и производится расчет освещенности.

Мы будем рассматривать попиксельные реализации этих моделей – окончательный расчет освещенности производится во фрагментном шейдере отдельно для каждого фрагмента. Тем не менее из соображений быстродействия обычно сами эти векторы вычисляются в вершинном шейдере. После этого при растеризации примитива происходит билинейная интерполяция этих векторов вдоль выводимого примитива.

Во фрагментном шейдере мы получаем результат интерполяции векторов-направлений, используемых для расчета освещенности. Однако, хотя вершинный шейдер на выход выдает нормированные векторы, в результате интерполяции они могут перестать быть единичными (рис. 13.11), и им потребуется нормировка.

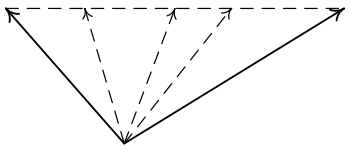

Рис. 13.11 ❖ В результате линейной интерполяции единичных векторов могут получиться неединичные векторы

Для преобразования направлений (таких как l, n и v) в касательное пространство используют матрицу 3×3, являющуюся транспонированной обратной верхней левой 3×3 подматрицей матрицы преобразования исходного объекта. Для ее получения мы будем использовать библиотеку `GLM`, ниже приводится соответствующий фрагмент кода.

```
glm::mat3 normalMatrix ( const glm::mat4& mv ) const
{
      return glm::inverseTranspose ( glm::mat3 ( mv ) );
}
```

Самым простым методом для реализации является метод Ламберта. Ниже приводятся вершинный и фрагментный шейдеры для него.

```
-- vertex

#version 330 core

uniform mat4 proj;      // матрица проектирования
uniform mat4 mv;        // модельно-видовая матрица
uniform mat3 nm;        // матрица преобразования направлений
uniform vec3 lightDir;  // направления падения света

layout(location = 0) in vec3 pos;
layout(location = 2) in vec3 normal;

out vec3  n;            // выдаем вектор нормали
out vec3  l;            // и направление на источник света

void main(void)
{
                  // преобразуем точку модельно-видовой матрицей
      vec4  p = mv * vec4 ( pos, 1.0 );
                  // вычисляем итоговые координаты
      gl_Position  = proj * p;
                  // нормаль преобразуется при помощи специальной матрицы
      n            = normalize ( nm * normal );
                  // считаем, что у нас бесконечно удаленный источник света
      l            = normalize ( lightDir );
}

-- fragment

#version 330 core

in    vec3 n;
in    vec3 l;

out vec4 color;

void main(void)
{
      vec3  n2   = normalize ( n );      // нормируем интерполированную нормаль
      vec3  l2   = normalize ( l );      // нормируем вектор на источник света
      float diff = max ( dot ( n2, l2 ), 0.0 );
      vec4  clr  = vec4 ( 0.7, 0.1, 0.1, 1.0 );
      float ka   = 0.2;
      float kd   = 0.8;

      color = (ka + kd*diff) * clr;
}
```

Здесь используется бесконечно удаленный источник света, задаваемый направлением падающего света `lightDir`. Для точечного источника света вместо на-

правления задается его положение `light`, и тогда вектор *l* для текущей точки `p` вычисляется следующим образом:

```
light = normalize ( light - p );
```

Для более сложных моделей освещения вершинный шейдер вычисляет большее число выходных векторов, по которым и производится расчет освещения. Все параметры освещения передаются как `uniform`-переменные. В ходе дальнейшего изложения мы столкнемся с реализациями некоторых других моделей освещения. Исходный код для целого ряда моделей освещения находится в исходном коде к книге на `github`.

Построение теней при помощи теневых карт

Одним из важных эффектов, связанных с освещением, является аккуратный расчет теней. Он заключается в том, что для каждого обрабатываемого фрагмента мы должны определить, освещается он заданным источником света или нет.

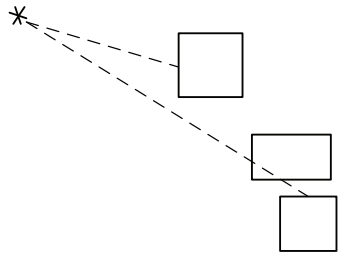

Рис. 13.12 ❖ Определение теней

Если мы поместим камеру в точку, в которой находится источник света, то можно заметить, что задача определения освещаемых участков сцены на самом деле является просто уже рассмотренной ранее задачей определения видимости (относительно источника света).

Тем самым для ее решения мы можем использовать стандартный метод буфера глубины. Для этого мы помещаем камеру в положение источника света и осуществляем рендеринг всей сцены. При этом нам на самом деле достаточно осуществлять рендеринг только в буфер глубины, выключив при этом запись в буфер цвета при помощи команды `glColorMask`.

После такого рендеринга у нас в буфере глубины будут храниться преобразованные значения глубины для всех видимых из положения источника света фрагментов. Поскольку для повторного рендеринга всей сцены (уже из нормального положения камеры) нам понадобится буфер глубины, то нам необходимо как-то сохранить текущие значения в буфере глубины.

Обычно для этого создается текстура формата глубины (чаще `GL_DEPTH_COMPONENT`), в которой и будут храниться значения из буфера глубины. Чаще всего рендеринг из положения источника света осуществляется в специальный объект-фреймбуфер. Тогда эта текстура просто подключается к нему как подключение глубины.

Рассмотрим теперь, как можно проверить видимость произвольного фрагмента при помощи этой текстуры со значениями глубины. Для этого обратите внима-

ние, что глубина, записанная в текстуру глубины, просто получается умножением координат на матрицу преобразования, используемого при рендеринге из положения источника света (и последующего перспективного деления). Координата *z* и будет тем самым значением глубины (преобразованным в отрезок [0, 1]).

Соответственно, для того чтобы проверить освещенность фрагмента, мы просто умножаем его координаты на соответствующую матрицу преобразования и выполняем перспективное деление. После этого получаемая *z*-координата и будет соответствовать глубине фрагмента по отношению к источнику света. Нам нужно просто сравнить ее с соответствующим значением из текстуры со значениями глубины (*теневой карты*, shadow map).

OpenGL предоставляет для этого аппаратную поддержку. Вместо того чтобы проводить это сравнение самому, его можно получить автоматически. Для этого достаточно просто установить необходимый режим для чтения из теневой карты. Ниже показывается, как это можно сделать.

```
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                  GL_COMPARE_REF_TO_TEXTURE );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC,
                  GL_LEQUAL );
```

Первая из этих команд задает то, что результатом чтения из текстуры (который получит соответствующий шейдер в результате вызова функции `texture*`) будет не само значение, прочитанное из текстуры. Вместо этого значения мы получим результат сравнения его с текстурной координатой *r* (т. е. *z*). Вторая команда задает, какая именно операция сравнения будет использована. В этом примере задается, что будет возвращено значение 1.0, если *r* меньше или равно значению, прочитанному из текстуры. В противном случае будет возвращен ноль.

Ниже приводится код, создающий объект-фреймбуфер с подключенной к нему теневой картой и осуществляющий рендеринг сцены с учетом теней.

```
ShadowWindow () : GlutRotateWindow ( 100, 100, 800, 800, «Shadow Mapping» )
{
            // создаем фреймбуфер 512x512 для рендеринга из источника света
      fb.create ( 512, 512, 0 );
      fb.bind   ();
      fb.attachDepthTexture ( GL_TEXTURE_2D,
                  shadowMap = fb.createColorTexture (
                  GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT,
                  GL_CLAMP_TO_BORDER,
                  FrameBuffer::filterNearest ) );

      if ( !fb.isOk () )
            exit ( "Error with fb\n" );

      fb.unbind ();

      if ( !program.loadProgram ( "decal.glsl" ) )
            exit ( "Error loading shader: %s\n", program.getLog().c_str() );

      program.bind ();
      program.setTexture ( "image", 0 );
      program.unbind ();
```

```
      if ( !program2.loadProgram ( "shadow.glsl" ) )
            exit ( "Error loading shader: %s\n", program2.getLog().c_str() );

      program2.bind ();
      program2.setTexture ( "image",     0 );
      program2.setTexture ( "shadowMap", 1 );
      program2.unbind ();

      mesh1 = createQuad  ( glm::vec3 ( -6, -6, -2 ),
                            glm::vec3 ( 12, 0, 0 ),
                            glm::vec3 ( 0, 12, 0 ) );
      mesh2 = createKnot ( 0.5, 2, 90, 50 );

      decalMap.load2D ( "../../Textures/wood.png" );
      stoneMap.load2D ( "../../Textures/brick.tga" );

      light = glm::vec3 ( 7, 0, 7 );
}
void redisplay ()
{
      renderToShadowMap ();

      glm::mat4   mv = getRotation ();

      glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

      program2.bind             ();
      program2.setUniformMatrix ( "mv",    mv   );
      program2.setUniformMatrix ( "nm",  normalMatrix ( mv ) );
      program2.setUniformMatrix ( "shadowMat",  shadowMat );
      program2.setUniformVector ( "light", light );

      decalMap.bind   ( 0 );
      shadowMap->bind ( 1 );
      mesh1->render   ();

      stoneMap.bind   ( 0 );
      shadowMap->bind ( 1 );
      mesh2->render   ();

      program2.unbind ();
}
      // осуществляет рендеринг сцены из положения источника света light
      // в теневую карту shadowMap, подключенную к фреймбуферу fb
void renderToShadowMap ()
{
      glm::mat4 proj   = glm::perspective ( glm::radians(90.0f),
            (float)shadowMap->getWidth()/(float)shadowMap->getHeight(),
            0.01f, 20.0f );
      glm::mat4 mv     = glm::lookAt      ( light, glm::vec3 ( 0, 0, 0 ),
            glm::vec3 ( 0, 0, 1 ) );
```

```
    shadowMat = proj * mv;

    program.bind ();
    program.setUniformMatrix ( "proj",  shadowMat );
    program.setUniformMatrix ( "mv",    getRotation () );
    program.setUniformMatrix ( "nm",    normalMatrix ( mv ) );
    program.setUniformVector ( "light", light );

    fb.bind ();

    glClear ( GL_DEPTH_BUFFER_BIT );

    decalMap.bind   ();
    mesh1->render   ();
    decalMap.unbind ();
    stoneMap.bind   ();
    mesh2->render   ();
    stoneMap.unbind ();
    fb.unbind       ();
    program.unbind  ();

    glFinish ();
}
```

Для чтения из текстуры можно избежать явного деления. Для этого следует вместо функции `texture` использовать функцию `textureProj`, которая сама выполняет требуемое деление. Также нам необходимо для каждого фрагмента перевести его координаты в систему координат источника света. Достаточно сделать это в вершинном шейдере и потом просто проинтерполировать полученные значения при растеризации треугольников.

Кроме того, нам нужно перевести текстурные координаты из отрезка [–1, 1] в отрезок [0, 1] подобно тому, как мы это делали при проективном текстурировании. Ниже приводятся вершинный и фрагментный шейдеры, использующие уже готовую теневую карту для расчета теней.

```
-- vertex
#version 330 core

uniform mat4 proj;
uniform mat4 mv;
uniform mat3 nm;
uniform vec3 light;
uniform mat4 shadowMat;

layout (location = 0) in vec3 pos;
layout (location = 1) in vec2 texCoord;
layout (location = 2) in vec3 normal;

out vec3  n;
out vec3  l;
out vec2  tex;
out vec4  shadowPos;
```

```
void main(void)
{
      vec4  p = mv * vec4 ( pos, 1.0 );

      gl_Position  = proj * p;
      n            = normalize ( nm * normal );
      l            = normalize ( light - p.xyz );
      tex          = texCoord;
      shadowPos    = shadowMat * p;
}

-- fragment
#version 330 core

in vec3 n;
in vec3 l;
in vec2 tex;
in vec4 shadowPos;

out vec4 color;

uniform sampler2D image;
uniform sampler2D shadowMap;

const float bias = -0.0001;

void main(void)
{
      vec3  n2   = normalize ( n );
      vec3  l2   = normalize ( l );
      float diff = max ( dot ( n2, l2 ), 0.0 );
      vec4  clr  = texture ( image, tex );
      float ka   = 0.2;
      float kd   = 0.8;

      vec3 p = shadowPos.xyz / shadowPos.w;

      p = p * 0.5 + vec3 ( 0.5 );  // переводим в отрезок [0,1]

      float closestDepth = texture ( shadowMap, p.xy ).r;
      float currentDepth = p.z;
                                  // выполняем проверку с заданной точностью bias
      float shadow       = currentDepth + bias > closestDepth ? 1.0 : 0.0;

      color = ka*clr + kd*diff * clr * (1.0 - shadow);
}
```

Обратите внимание, что при недостаточном разрешении теневой карты на изображения мы получаем «блочную» тень. В ней каждый блок соответствует одному текселу теневой карты. В следующей главе мы рассмотрим получение мягких теней при помощи теневых карт. Это позволит сильно уменьшить подобные артефакты.

Освещение с учетом микрорельефа (bump mapping)

Bump mapping (или normal mapping) – это один из способов имитации микрорельефа без усложнения используемой геометрии. Он заключается в том, что мы задаем микрорельеф грани при помощи специальной текстуры. Наиболее прямолинейный подход состоит в задании микрорельефа при помощи *карты высот* (height map). В такой карте мы задаем смещение вдоль вектора нормали.

Рис. 13.13 ❖ Освещенная грань без учета микрорельефа (слева) и с ним (справа)

Далее внутри фрагментного шейдера мы можем через конечные разности получить частные производные от карты высоты и по ним найти новый вектор нормали – свой вектор для каждого фрагмента, с учетом микрорельефа.

Однако обычно используют более эффективный подход. Вместо задания карты высот, по которой во фрагментном шейдере находится нормаль для каждого фрагмента, мы в текстуре сразу зададим нужное значение вектора нормали. Однако у обычных текстур значения компонент являются нормализованными величинами и лежат на отрезке [0, 1], а компоненты единичного вектора нормали лежат на [–1, 1]. Поэтому используется просто преобразование, сопоставляющее каждому единичному вектору нормали n некоторый допустимый цвет:

$$c = \frac{n+1}{2}.$$

Обратите внимание, что в касательном пространстве вектор нормали равен $(0 \quad 0 \quad 0)^T$. При его кодировании в виде цвета мы получаем вектор $(0.5 \quad 0.5 \quad 1)^T$. Поэтому все текстуры с нормалями имеют заметный синеватый оттенок в силу того, что наибольшей компонентой является z и ей соответствует синий цвет.

При этом хранение в текстуре готовых векторов нормали несет в себе следующий вопрос: в какой именно системе координат мы будем хранить эти векторы?

Проще всего хранить эти векторы сразу в мировой системе координат (или системе координат камеры) – там, где мы проводим расчет освещения. Однако это

несет в себе следующую проблему: любой поворот объекта делает эту текстуру неверной. Поэтому крайне желательно выбрать такую систему координат, которая была бы жестко привязана к грани и при повороте грани поворачивалась бы вместе с ней.

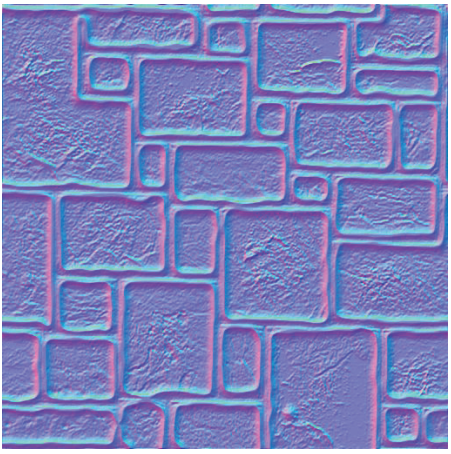

Рис. 13.14 ❖ Пример карты нормалей

Обычно в качестве такой системы координат выступает *касательная система* координат (tangent space). Она жестко привязана к грани и задается тремя векторами – касательным вектором t, бинормалью b и нормалью n. Поэтому часто для этой системы координат используют обозначение *TBN*.

Нередко готовые модели, загружаемые из файлов, уже содержат эти векторы для каждой вершины. Однако бывают случаи, когда у нас есть набор граней, для каждой вершины которого заданы только вектор нормали и текстурные координаты. Тогда можно использовать следующий приблизительный способ построения *TBN*-базиса.

Пусть у нас есть треугольная грань $v_0v_1v_2$ с заданными текстурными координатами (p, q) в каждой вершине. Если вектор нормали n не задан, то его можно легко найти по следующей формуле:

$$n = \frac{[v_2 - v_0, v_1 - v_0]}{\left\|[v_2 - v_0, v_1 - v_0]\right\|}.$$

Мы хотим найти такие векторы t' и b', чтобы была справедлива следующая система уравнений:

$$\begin{cases} v_1 - v_0 = (p_1 - p_0)t' + (q_1 - q_0)b' \\ v_2 - v_0 = (p_2 - p_0)t' + (q_2 - q_0)b' \end{cases}.$$

Решение этой системы уравнений можно представить в следующем виде:

$$\begin{pmatrix} t'_x & t'_y & t'_z \\ b'_x & b'_y & b'_z \end{pmatrix} = \frac{1}{(p_1 - p_0)(q_2 - q_0) - (p_2 - p_0)(q_1 - q_0)} \cdot A \cdot B.$$

Матрицы A и B задаются следующим образом:

$$A = \begin{pmatrix} q_2 - q_0 & -(q_1 - q_0) \\ -(p_2 - p_0) & p_1 - p_0 \end{pmatrix};$$

$$B = \begin{pmatrix} v_{1x} - v_{0x} & v_{1y} - v_{0y} & v_{1z} - v_{0z} \\ v_{2x} - v_{0x} & v_{2y} - v_{0y} & v_{2z} - v_{0z} \end{pmatrix}.$$

После этого мы получаем векторы t' и b' и применяем к ним процесс Грамма–Шмидта для их ортогонализации:

$$t = t' - (n, t')n;$$

$$b = b' - (n, b')b' - (t', b')t' \frac{1}{\| t' \|^2}.$$

После этого нам остается просто пронормировать полученные векторы t и b. Для такого расчета можно использовать приводимый ниже фрагмент кода.

```
void  computeTangents ( BasicVertex& v0, const BasicVertex& v1,
                        const BasicVertex& v2 )
{
      glm::vec3   e0(v1.pos.x-v0.pos.x, v1.tex.x-v0.tex.x, v1.tex.y-v0.tex.y);
      glm::vec3   e1(v2.pos.x-v0.pos.x, v2.tex.x-v0.tex.x, v2.tex.y-v0.tex.y);
      glm::vec3   cp ( glm::cross ( e0, e1 ) );

      if ( fabs ( cp.x ) > EPS )
      {
            v0.t.x = -cp.y / cp.x;
            v0.b.x = -cp.z / cp.x;
      }
      else
      {
            v0.t.x = 0;
            v0.b.x = 0;
      }

      e0.x = v1.pos.y - v0.pos.y;
      e1.x = v2.pos.y - v0.pos.y;
      cp   = glm::cross ( e0, e1 );

      if ( fabs ( cp.x ) > EPS )
      {
            v0.t.y = -cp.y / cp.x;
            v0.b.y = -cp.z / cp.x;
      }
      else
      {
            v0.t.y = 0;
            v0.b.y = 0;
      }

      e0.x = v1.pos.z - v0.pos.z;
```

```
      e1.x = v2.pos.z - v0.pos.z;
      cp   = cross ( e0, e1 );

      if ( fabs ( cp.x ) > EPS )
      {
            v0.t.z = -cp.y / cp.x;
            v0.b.z = -cp.z / cp.x;
      }
      else
      {
            v0.t.z = 0;
            v0.b.z = 0;
      }

      if ( glm::dot ( glm::cross ( v0.t, v0.b ), v0.n ) < 0 )
            v0.t = -v0.t;
}
```

Весь расчет освещения мы будем проводить в касательном пространстве. Тогда в вершинном шейдере мы не только вычисляем необходимые для расчета освещенности векторы (l, v и h), но и переводим их в касательное пространство. Именно в этом пространстве они и будут использованы во фрагментном шейдере. Для этого в каждой вершине нам нужен базис касательного пространства – векторы t, n и b. Ниже приводится простейший вершинный шейдер для расчета освещенности по модели Фонга.

```
#version 330 core

uniform mat4 proj;      // матрица проектирования
uniform mat4 mv;        // модельно-видовая матрица
uniform mat3 nm;        // матрица для преобразования направлений
uniform vec3 eye;       // координаты камеры
uniform vec3 light;     // координаты источника света

layout ( location = 0 ) in vec3 position;    // координаты вершины
layout ( location = 1 ) in vec2 texCoord;    // текстурные координаты
layout ( location = 2 ) in vec3 normal;      // нормаль в вершине
layout ( location = 3 ) in vec3 tangent;     // касательный вектор
layout ( location = 4 ) in vec3 binormal;    // бинормаль в вершине

out vec3  n;
out vec3  v;
out vec3  l;
out vec3  h;
out vec2  tex;

void main(void)
{
      vec4  p = mv * vec4 ( position, 1.0 );
      vec3  n = normalize ( nm * normal );
      vec3  t = normalize ( nm * tangent );
      vec3  b = normalize ( nm * binormal );
```

```
    gl_Position  = proj * p;
    n            = normalize ( nm * normal );
    v            = normalize ( eye - p.xyz );
    l            = normalize ( light - p.xyz );
    h            = normalize ( l + v );
                      // переводим векторы v, l и h в касательное пространство
    v            = vec3 ( dot ( v, t ), dot ( v, b ), dot ( v, n ) );
    l            = vec3 ( dot ( l, t ), dot ( l, b ), dot ( l, n ) );
    h            = vec3 ( dot ( h, t ), dot ( h, b ), dot ( h, n ) );
    tex          = texCoord * vec2 ( 1.0, 6.0 );
}
```

Во фрагментном шейдере, как и ранее, мы нормируем полученные на вход значения векторов *l*, *v* и *h*. Но сам вектор нормали мы берем из текстуры, где он будет задан в касательном пространстве. Значение нормали, извлеченное из соответствующей текстуры, подвергается преобразованию, переводящему ее компоненты из [0, 1] обратно в [–1, 1].

Кроме того, полученное значение также нормируется, поскольку в результате текстурной фильтрации мы можем получить неединичный вектор. Ниже приводится фрагментный шейдер, соответствующий модели освещения Фонга.

```
#version 330 core

in  vec3 n;
in  vec3 v;
in vec3 l;
in  vec3 h;
in  vec2 tex;
out vec4 color;

uniform sampler2D colorMap;
uniform sampler2D bumpMap;

const vec4  specColor = vec4 ( 1.0 );
const float specPower = 70.0;
const float ka        = 0.2;
const float kd        = 0.8;
const float ks        = 0.5;

void main(void)
{
                      // читаем нормаль и переводим из цвета в [-1,1]^3
    vec3  n    = 2.0 * (texture ( bumpMap, tex ).rgb - vec3 ( 0.5 ));
                      // нормируем интерполированные входные векторы
    vec3  v2   = normalize ( v );
    vec3  n2   = normalize ( n );
    vec3  l2   = normalize ( l );
    vec3  h2   = normalize ( h );
                      // берем цвет поверхности из текстуры
    vec4  clr  = texture ( colorMap, tex );
                      // вычисляем диффузную и бликовую компоненты
    vec4  diff = clr * max ( dot ( n2, l2 ), 0.1 );
    vec4  spec = specColor * pow (max (dot (n2, h2), 0.0), specPower);

    color = (ka + kd*diff)*clr + ks*vec4(spec);
}
```

Имитация отражения окружающей среды с учетом карт нормалей

Карты нормалей можно использовать не только для расчетов освещенности. Поскольку при имитации отражения объектом окружающей среды для расчета отраженного вектора также используется вектор нормали, то эту нормаль можно брать из карты нормалей. В результате мы получаем отражение окружающей среды с учетом карты нормалей (EMBM, Environment Mapped Bump Mapping).

Рис. 13.15 ❖ Пример отражения с учетом карты нормалей

Отличие от ранее рассмотренных шейдеров заключается в том, что для построения отраженного вектора мы берем вектор нормали из карты нормалей. Ниже приводятся соответствующие вершинный и фрагментный шейдеры.

```
-- vertex
#version 330 core

layout ( location = 0 ) in vec3 position;
layout ( location = 1 ) in vec2 texCoord;
layout ( location = 2 ) in vec3 normal;
layout ( location = 3 ) in vec3 tangent;
layout ( location = 4 ) in vec3 binormal;

uniform mat4 proj;
uniform mat4 mv;
uniform mat3 nm;
uniform vec3 eye;

out vec3  n;
out vec3  v;
out vec3  l;
out vec3  h;
out vec2  tex;
```

```
void main(void)
{
      vec4  p = mv * vec4 ( position, 1.0 );
      vec3  n = normalize ( nm * normal );
      vec3  t = normalize ( nm * tangent );
      vec3  b = normalize ( nm * binormal );

      gl_Position  = proj * p;
                        // поворачиваем нормаль
      n            = normalize ( nm * normal );
                        // находим и поворачиваем вектор v
      v            = normalize ( eye - p.xyz );
                        // переводим вектор v в касательное пространство
      v            = vec3 ( dot ( v, t ), dot ( v, b ), dot ( v, n ) );
      tex          = texCoord;
}

-- fragment
#version 330 core

In  vec3 n;
in  vec3 v;
in  vec2 tex;
out vec4 color;

uniform sampler2D   bumpMap;
uniform samplerCube cubeMap;

void main(void)
{
                        // читаем нормаль и переводим из цвета в [-1,1]^3
      vec3  n  = 2.0 * (texture ( bumpMap, tex ).rgb - vec3 ( 0.5 ));
      vec3  v2 = normalize ( v );
      vec3  n2 = normalize ( n );
      vec3  r  = reflect   ( v2, n2 );

      color = texture ( cubeMap, r );
}
```

Вывод текста при помощи поля расстояний

Довольно часто возникает задача рендеринга текста. В современных операционных системах шрифты обычно хранятся в векторном виде (например, в формате ***ttf***). При таком представлении каждый символ задается при помощи сплайнов, ограничивающих закрашиваемую область. Такой подход позволяет очень легко масштабировать шрифты и применять к нему различные эффекты. Однако, к сожалению, подобное представление очень плохо подходит для использования в OpenGL.

Поэтому обычно по такому шрифту строится изображение, содержащее блоки, соответствующие каждому символу (рис. 13.16). Шрифты в таком виде очень удобно выводить при помощи OpenGL, но любое преобразование (масштабирование, поворот и т. п.) часто приводит к заметному падению качества получаемых символов.

Рис. 13.16 ❖ Пример текстуры со шрифтом заданного размера

Есть очень простой и красивый способ, позволяющий легко справиться со всеми этими проблемами без заметного усложнения рендеринга. Он основан на использовании так называемых *полей расстояний* (distance field, signed distance field). Для получения такого поля мы берем исходную текстуру с символами шрифта и по ней строим новую текстуру из оттенков серого цвета. В этой новой текстуре для каждого тексела хранится расстояние от него до ближайшего закрашенного (т. е. принадлежащего символу) пиксела. Это и есть поле расстояний.

Для построения текстуры с исходным шрифтом можно использовать бесплатную утилиту `BMFont` (которую можно скачать по адресу **http://www.angelcode.com/products/bmfont/**). Для расчета поля нормалей по получившемуся изображению можно использовать утилиту `ImageMagick`. Ниже приводится команда для построения изображения с полем расстояний по входному изображению.

```
magick convert font_0.tga -filter Jinc -resize 400% -threshold 30% ( +clone -negate
-morphology Distance Euclidean -level 50%,-50% ) -morphology Distance Euclidean -compose Plus
-composite -level 45%,55% -resize 25% out.png
```

Ниже приводится пример текстуры со шрифтом в рис. 13.16 с рассчитанным полем расстояний. Обратите внимание, что `BMFont` плотно упаковывает символы в текстуру, поэтому вместе с изображением строится и специальный текстовый файл, содержащий свойства шрифта и задающий для каждого символа его положение в текстуре (рис. 13.17).

Рис. 13.17 ❖ Пример поля расстояний для шрифта

Для рендеринга символа по такой текстуре мы включаем для нее билинейную фильтрацию и просто проводим отсечение – если прочитанное расстояние из текстуры менее заданного, то мы считаем, что данный фрагмент принадлежит символу. Для повышения качества обычно вместо точного отсечения по одному значению используется функция `smoothstep`. Ниже приводится соответствующий фрагментный шейдер.

```
#version 330 core

in  vec2 texCoord;
out vec4 color;

uniform sampler2D fontMap;    // текстура с полем расстояний

void main ()
{
      float distance    = texture    ( fontMap, texCoord ).r;
      float smoothWidth = fwidth     ( distance );
      float alpha       = smoothstep ( 0.5 - smoothWidth,
                                       0.5 + smoothWidth, distance );
      color = vec4 ( vec3 ( alpha ), alpha );
}
```

Рендеринг меха

Рендеринг меха (или шерсти) является довольно сложным из-за того, что это фактически объем, заполненный волосками. Рендеринг каждого отдельного волоска является сильно затратным и в реальном времени используется достаточно редко. Здесь мы рассмотрим довольно простой способ, позволяющий получить реалистично выглядящее изображение объектов, покрытых мехом.

Пусть нам нужно вывести шар, покрытый мехом. Давайте выведем ряд концентрических сфер с общим центром и увеличивающимся радиусом (рис. 13.18). Эти сферы мы будем выводить в режиме альфа-смешивания и с полупрозрачной текстурой, приведенной на рис. 13.19.

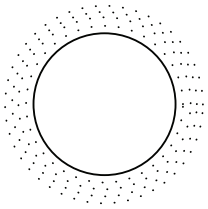

Рис. 13.18 ❖ Концентрические сферы, используемые для вывода «мехового» шара

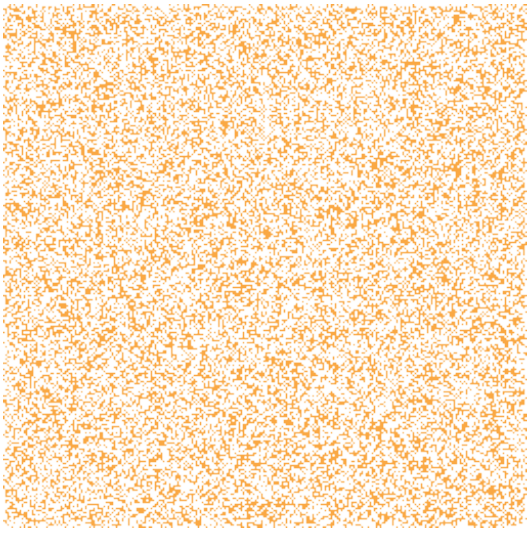

Рис. 13.19 ❖ «Меховая» текстура

Как видно из приведенного рисунка, данная текстура состоит из набора отдельных точек, соответствующих отдельным волоскам. Если изменение радиуса у соседних сфер достаточно мало, то близко расположенные точки на соседних сферах просто сливаются и формируют изображение волосков (рис. 13.20).

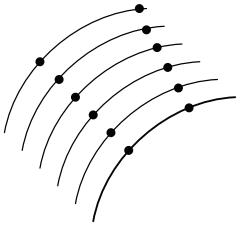

Рис. 13.20 ❖ Формирование отдельных волосков

Таким образом, мы легко получаем пространство, заполненное отдельными волосками. Мы можем легко добавить анимацию этим волоскам, просто сдвигая текстурные координаты в зависимости от номера выводимого слоя (сферы).

Для получения реалистически выглядящего меха нам осталось только добавить правдоподобное освещение. Модель освещения для отдельных волосков была нами рассмотрена в главе 9. Для того чтобы мы могли ее применять, нам нужно точно знать касательный вектор *t* в каждой точке.

В простейшем случае этот касательный вектор просто будет совпадать с нормалью к сфере. В случае сдвига текстурных координат соответствующим образом будет отклоняться и касательный вектор от вектора нормали. Ниже приводятся вершинный и фрагментный шейдеры для примера рендеринга меха.

За счет изменения текстурных координат можно управлять вектором *t*, т. е. направлением отдельных волосков меха. Используя это, очень легко реализовать эффект анимированного меха, когда волоски меха колышутся с течением времени.

```
-- vertex
#version 330 core

layout ( location = 0 ) in vec3 position;
layout ( location = 1 ) in vec2 texCoord;
layout ( location = 2 ) in vec3 normal;
layout ( location = 3 ) in vec3 tangent;
layout ( location = 4 ) in vec3 binormal;

uniform mat4 mv;
uniform mat4 proj;
uniform mat3 nm;

uniform float tl;      // параметр, управляющий анимацией меха
uniform vec3  light;   // положение источника света
uniform vec3  eye;     // положение камеры
uniform float time;    // текущее время (для анимации)

out vec3 lt;
out vec3 ht;
out vec3 furDir;
out vec2 tex;

void main ()
{
      vec3  pp = position + 0.5 * tl * normal;     // смещаем вершину вдоль нормали

      float d1 = 0.01 * sin ( 0.731*time ) * tl;   // вычисляем параметры для анимации меха
      float d2 = 0.03 * sin ( 0.231*time ) * tl;

      vec3  p = (mv * vec4 ( position, 1.0 ) ).xyz;
```

```
      vec3  l = normalize ( light - p );  // направление на источник света
      vec3  v = normalize ( eye   - p );  // направление на наблюдателя
      vec3  h = l + v;
      vec3  n = nm * normal;              // повернутая нормаль
      vec3  t = nm * tangent;             // повернутый касательный вектор
      vec3  b = nm * binormal;            // повернутая бинормаль

                  // переводим векторы в касательное пространство
      lt     = vec3 ( dot ( l, t ), dot ( l, b ), dot ( l, n ) );
      ht     = vec3 ( dot ( h, t ), dot ( h, b ), dot ( h, n ) );
                  // вектор смещения для направления волосков
      furDir = vec3 ( d1, d2, 1.0 );
      tex    = 2.25*texCoord + vec2 ( d1, d2 );

      gl_Position     = proj * mv * vec4 ( pp, 1.0 );
}
-- fragment
#version 330 core

uniform sampler2D furMap;                 // текстура с мехом

uniform float tl;
uniform vec4  light;
uniform vec4  eye;
uniform float time;

in vec2 tex;
in vec3 lt;
in vec3 ht;
in vec3 furDir;

out vec4 color;

void main ()
{
      vec4 c = texture ( furMap, 0.25*tex * vec2 ( 6.0, 12.0 ) );

      if ( c.a < 0.02 )
            discard;

      vec3  t    = normalize ( furDir );
      float tl   = dot       ( normalize ( lt ), t );
      float th   = dot       ( normalize ( ht ), t );
      float spec = pow       ( 1.0 - th*th, 70.0 );

      color = vec4 ( c.rgb*sqrt(1.0-tl*tl) + c.rgb*vec3(spec),  c.a*0.1 );
}
```

На рис. 13.16 приведен пример такого освещенного «мохнатого» тора.

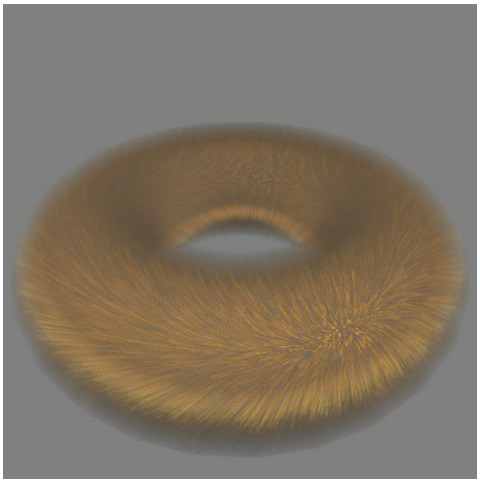

Рис. 13.21 ❖ Мохнатый тор

Ниже приводится код функции redisplay, используемой для вывода этого тора.

```
void redisplay ()
{
      glm::mat4   mv = getRotation  ();
      glm::mat3   nm = normalMatrix ( mv );
      float       shadowMin = 0.2;
      float       shadowMax = 0.5;

      glClear     ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
      glEnable    ( GL_BLEND );
      glBlendFunc ( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );

      furMap.bind              ();
      program.bind             ();
      program.setUniformMatrix ( "mv",  mv );
      program.setUniformMatrix ( "nm",  nm );
      program.setUniformFloat  ( "time", 4*getTime () );

      for ( int i = 0; i < numLayers; i++ )     // вывод отдельных слоев меха
      {
            float t      = (float) i / 16.0;
            float shadow = 0.7*(shadowMin * (1 - t) + shadowMax * t);

            program.setUniformFloat ( "tl", t );

            mesh -> render ();
      }

      program.unbind ();
      furMap.unbind  ();

      glDisable ( GL_BLEND );
}
```

Physically Based Rendering (PBR)

Давайте рассмотрим реализацию PBR. Мы будем описывать поверхность объекта при помощи следующего набора текстур:

- `baseColor` – цвет поверхности (или блика для металлов);
- `metallness` – является ли в этой точке материал металлом или диэлектриком (0 – диэлектрик, 1 – металл);
- `roughness` – неровность поверхности (изменяется от 0 до 1);
- `normal` – карта нормалей, задает нормали для каждой точки.

Каждая из этих величин задается при помощи текстур. Различие между металлами и диэлектриками связано с их разной обработкой. У металлов нет диффузной составляющей, поэтому в `baseColor` хранится цвет блика – значение $F(0)$ в формуле Шлика для коэффициента Френеля. Диффузный цвет для металлов считается равным нулю. Для диэлектриков обычно считается, что $F(0)$ всегда равен 0.04, а в `baseColor` хранится именно диффузный цвет.

Значения `roughness` используются для задания неровности поверхности в модели освещения Кука–Торранса, при этом коэффициент Френеля определяет вклад бликового и диффузного освещений. В `normal` хранится карта нормалей для задания микрорельефа поверхности. При этом `metallness` и `roughness` являются обычными скалярными значениями и могут лежать в одной текстуре.

Рис. 13.22 ❖ Пример рендеринга с использованием PBR

Ниже приводится соответствующий фрагментный шейдер. Обратите внимание на то, что определены различные версии геометрического члена и функции распределения – можно использовать любые их комбинации. Также к изображению применяется гамма-коррекция – подробнее об этом будет рассказано в следующей главе.

```
#version 330 core

in  vec2 tx;
in  vec3 l;
```

```
in  vec3 h;
in  vec3 v;
in  vec3 t;
in  vec3 b;
in  vec3 n;
out vec4 color;

uniform sampler2D albedoMap;
uniform sampler2D metallnessMap;
uniform sampler2D normalMap;
uniform sampler2D roughnessMap;
uniform vec3      lightColor;

const float gamma = 2.2;
const float pi    = 3.1415926;
const float FDiel = 0.04;            // Коэффициент Френеля для диэлектриков

vec3 fresnel ( in vec3 f0, in float product )
{
    return mix ( f0, vec3 (1.0), pow(1.0 - product, 5.0) );
}

float D_beckmann ( in float roughness, in float NdH )
{
    float m    = roughness * roughness;
    float m2   = m * m;
    float NdH2 = NdH * NdH;

    return exp( (NdH2 - 1.0) / (m2 * NdH2) ) / (pi * m2 * NdH2 * NdH2);
}

float D_GGX ( in float roughness, in float NdH )
{
    float m  = roughness * roughness;
    float m2 = m * m;
    float d  = (NdH * m2 - NdH) * NdH + 1.0;

    return m2 / (pi * d * d);
}

float G_schlick ( in float roughness, in float nv, in float nl )
{
    float k = roughness * roughness * 0.5;
    float V = nv * (1.0 - k) + k;
    float L = nl * (1.0 - k) + k;

    return 0.25 / (V * L);
}

float G_neumann ( in float nl, in float nv )
{
      return nl * nv / max ( nl, nv );
}

float G_klemen ( in float nl, in float nv, in float vh )
{
      return nl * nv / (vh * vh );
}
```

```
float G_default ( in float nl, in float nh, in float nv, in float vh )
{
      return min ( 1.0, min ( 2.0*nh*nv/vh, 2.0*nh*nl/vh ) );
}

vec3 cookTorrance ( in float NdL, in float NdV, in float NdH,
  in vec3 f0, in float roughness )
{
    float D = D_GGX     ( roughness, NdH );
    float G = G_schlick ( roughness, NdV, NdL );

      return f0 * D * G;
}

void main ()
{
      vec3 base        = texture ( albedoMap,     tx).xyz;
      vec3 n           = texture ( normalMap,     tx).xyz * 2.0 - vec3 (1.0);
      float roughness  = texture ( roughnessMap,  tx).x;
      float metallness = texture ( metallnessMap, tx ).x;

      base = pow ( base, vec3 ( gamma ) );

      vec3  n2   = normalize ( n );
      vec3  l2   = normalize ( l );
      vec3  h2   = normalize ( h );
      vec3  v2   = normalize ( v );
      float nv   = max ( 0.0, dot ( n2, v2 ));
      float nl   = max ( 0.0, dot ( n2, l2 ));
      float nh   = max ( 0.0, dot ( n2, h2 ));
      float hl   = max ( 0.0, dot ( h2, l2 ));
      float hv   = max ( 0.0, dot ( h2, v2 ));

      vec3 F0          = mix ( vec3(FDiel), base, metallness );
      vec3 specfresnel = fresnel ( F0, hv );
      vec3 spec        = cookTorrance ( nl, nv, nh, specfresnel, roughness )*
                                    nl / max ( 0.001, 4.0 * nl * nv );
      vec3 diff        = (vec3(1.0) - specfresnel) * nl / pi;

      color = pow ( vec4 ( ( diff * mix ( base, vec3(0.0), metallness) +
                           spec ) * lightColor, 1.0 ), vec4 ( 1.0 / gamma ) );
}
```

Сейчас существует много готовых моделей, которые созданы специально для PBR, и к ним идет набор текстур, задающий все рассмотренные выше параметры – диффузный/бликовый цвет, металличность, неровность и карту нормалей.

Приложение

Язык GLSL

Все шейдеры в современном OpenGL пишутся на специальном языке, получившем название GLSL (OpenGL Shading Language). Это высокоуровневый язык, созданный специально для написания шейдеров, основанных на языке C99. В этом приложении будут рассмотрены основы GLSL и встроенные функции для него.

Основы языка

Как уже говорилось, в основу языка GLSL лег язык C99 с некоторыми изменениями. Из языка были убраны некоторые возможности, ненужные для написания шейдеров (или не поддерживаемые различными GPU). В число таких возможностей вошли указатели, объединения (union), битовые поля, исключения, рекурсия.

Еще в язык были добавлены новые типы данных и описатели, соответствующие специфике графического конвейера и решаемых при помощи шейдеров задач. Также были добавлены многочисленные специальные функции, часто используемые при написании шейдеров.

Есть различные версии GLSL. Мы рассмотрим далее версию 3.30. Каждый шейдер для этой версии GLSL должен начинаться с конструкции #version, задающей используемую версию GLSL. Ниже приводится данная конструкция, задающая версию 3.30.

```
#version 330 core
```

В табл. А.1 приведены вводимые в язык векторные и матричные типы данных. Они являются встроенными в язык и могут использоваться при написании шейдеров.

Таблица А.1. Встроенные векторные и матричные типы для языка GLSL

Тип	Описание
`vec2`, `vec3`, `vec4`	Двух-, трех- и четырехмерные векторы из компонент типа `float`
`ivec2`, `ivec3`, `ivec4`	Двух-, трех- и четырехмерные векторы из компонент типа `int`
`uvec2`, `uvec3`, `uvec4`	Двух-, трех- и четырехмерные векторы из компонент типа `uint`
`bvec2`, `bvec3`, `bvec4`	Двух-, трех- и четырехмерные векторы из компонент типа `bool`
`mat2`, `mat3`, `mat4`	Двух-, трех- и четырехмерные квадратные матрицы из компонент типа `float`
`mat2x3`, `mat2x4`, `mat3x2`, `mat3x4`, `mat4x2`, `mat4x2`	Матрицы из компонент типа `float`

Все переменные в GLSL должны быть описаны так же, как и в языке C. При этом они тоже могут быть инициализированы, в том числе и при помощи конструкторов, как показано ниже.

```
int   index;
float base = 0.1;
vec3  v ( 1.0 );           // (1.0, 1.0, 1.0)
vec2  t ( 0.0, 1.5 );      // (0.0, 1.5)
```

В GLSL действуют очень строгие правила преобразования типов, многие привычные для C/C++ неявные преобразования типов были убраны из языка. Поэтому в ряде случаев нужно осуществлять явное преобразование типа при помощи конструкторов, как показано ниже.

```
float      f = float ( index );
```

При этом если в конструкторе вектора было передано всего одно скалярное значение, то оно будет использовано для инициализации сразу всех компонент этого вектора. Если было передано несколько скалярных или векторных значений, то эти значения будут назначаться слева направо, лишние значения будут просто отброшены. Обратите внимание, что в этом случае всего должно быть передано не менее требуемого числа значений.

```
bool  b = true;
float f = float(b);
vec4  v1 = v1 ( 1.0 );       // (1.0, 1.0, 1.0, 1.0)
vec3  v2 ( 1.0, 2.0, 3.0 );  // (1.0, 2.0, 3.0)
vec2  v3 ( v1 );             // (1.0, 1.0)
vec4  v4 ( v2, 1.0 );        // (1.0, 2.0, 3.0, 1.0)
```

Конструкторы матричных типов обладают большей гибкостью. Для них действуют следующие правила:

- если передано всего одно скалярное значение, то оно используется для инициализации главной диагонали матрицы, а все остальные элементы матрицы получают нулевые значения;
- матрица может быть построена из нескольких векторов, выполняющих роль столбцов матрицы;
- матрица может быть построена из набора скалярных значений, по одному на каждый элемент матрицы слева направо, сама матрица при этом хранится по столбцам (column-major).

```
mat3  m  ( v2, v2, vec3 ( 3.0 ) );
mat2  m2 ( 1.0, 0.0, 0.0, 2.0 );
```

В языке поддерживаются два способа обращения к компонентам вектора – по имени (при помощи оператора ".") или как к элементу массива (при помощи оператора "[]"). Поскольку вектор может иметь разный смысл (например, координаты в пространстве, текстурные координаты, цвет), то поддерживается сразу три набора имен для компонент вектора – {*x*, *y*, *z*, *w*}, {*r*, *g*, *b*, *a*} и {*s*, *t*, *p*, *q*}.

При этом можно обращаться сразу к нескольким компонентам вектора, например `v.xz`, можно менять компоненты местами и повторять одни и те же компоненты несколько раз. Однако при этом нельзя смешивать разные наборы имен, поэтому запись `v.zb` недопустима.

```
vec3  t = v.xyz;
vec2  s = v.zy;
vec4  c = v.bgra;
```

Обращение при помощи оператора [] рассматривает вектор просто как массив компонент с нумерацией, начинающейся с нуля. Тогда запись v1[0] соответствует v.x, а v[3] соответствует v.w.

Сами матрицы можно рассматривать как массивы векторов-столбцов. Таким образом, m[1] – это второй столбец матрицы m, а m[2][2] – это элемент в правом нижнем углу m.

К описателям переменных можно добавлять описатель const, делающий соответствующие переменные неизменяемыми (т. е. они всегда будут иметь значение, данное при инициализации).

```
const float x = 0.0;
const vec3  v = vec2 ( 1.2, 3.4 );
```

В GLSL поддерживается объединение типов в структуры, как и в языке C. После описания структуры автоматически определяется конструктор с тем же именем, что и сама структура. Порядок аргументов конструктора совпадает с тем порядком, в котором были описаны поля структуры.

```
struct Light
{
      vec3  pos;
      vec4  color;
      float radius;
};

Light light = Light ( vec3 ( 0.0 ), vec4 ( 1.0, 0.0, 0.0, 0.0 ), 1.0 );
```

Данные можно объединять в массивы, индексация начинается с нуля.

```
float f [3] = float [] ( 1.0, 2.0, 3.0 );
vec2  g [2] = vec2  [] ( vec2 ( 1.0 ), vec2 ( 2.0 ) );
```

Язык GLSL поддерживает все стандартные операторы языка C, кроме операторов, связанных с работой с указателями («&» и «->»). Обратите внимание, что в силу отсутствия автоматического преобразования типов бинарные операторы (например, оператор «+») применяются к паре значений одного типа.

Ряд операторов, таких как «*», может работать над векторными и матричными значениями. При этих векторных типах операция применяется поэлементно. Однако умножение для матричных типов – это именно умножение матриц, а не поэлементное перемножение. Также допустимо умножение матрицы на вектор (в случае соответствия необходимых размеров).

```
vec3  w = m * v2;
```

Большинство операторов (кроме == и !=), будучи примененными к векторным значениям, дает в качестве результата также вектор. При этом операторы == и != возвращают результат сравнения значений, т. е. скалярное булево значение.

В GLSL есть большое количество встроенных функций, и также можно вводить свои функции. Однако поскольку указателей и ссылок в языке нет, то аргументы функции могут снабжаться специальными описателями in, out и inout. Эти описатели задают, как именно значения для соответствующих аргументов будут передаваться в/из функции. Так, в следующем примере аргумент x передается только как входной аргумент, а y – только как выходной.

```
void foo ( in float x, out float y )
{
     y = 2.0 * x + 1.0;
}
```

Описатель `in` обозначает параметр функции, передаваемый по значению, его изменения внутри функции не выходят за пределы функции. Описатель `out` обозначает выходное значение функции (передается по ссылке), и описатель `inout` обозначает передаваемый по ссылке параметр, который передается в функцию и может быть изменен внутри функции.

Язык GLSL поддерживает все основные операторы языка C. Кроме того, во фрагментном шейдере можно использовать специальный оператор `discard`, приводящий к отбрасыванию текущего фрагмента.

```
if ( color.a < 0.1 )
     discard;
```

Входные и выходные значения для шейдера описываются вне функций (как глобальные переменные) с описателями `in` и `out` соответственно. Для входных значений вершинного шейдера (т. е. атрибутов вершины) можно задать положение (location) при помощи описателя layout. Аналогичным образом можно задать индекс выходного значения для фрагментного шейдера.

```
in vec3 pos;
layout ( location = 1 ) in vec2 texCoord;
out vec3 w;
```

Выходные значения, заданные в вершинном шейдере, в ходе растеризации примитива интерполируются вдоль него. В GLSL для описания способа интерполяции используются описатели `smooth` (интерполяция вдоль примитива с учетом перспективы), `noperspective` (просто линейная интерполяция без учета перспективы) и `flat` (интерполяции фактически нет, соответствующее значение берется от провоцирующей – `provoking` – вершины).

```
smooth out vec3 p;
flat   out vec3 n;
```

В GLSL есть препроцессор и поддерживаются стандартные команды для препроцессора. В число предопределенных макросов входят `__LINE__`, `__FILE__`, `__VERSION__` и `GL_ES`. Также были добавлены новые команды для препроцессора `#version` и `#extension`.

Первая из них задает требуемую версию GLSL, а вторая служит для включения поддержки нужных расширений OpenGL. Следующий пример требует поддержки расширения `GL_NV_shadow_sampers_cube`:

```
#extension GL_NV_shadow_sampers_cube : require
```

К выходным переменным вершинного шейдера может быть также применен специальный описатель – `invariant`. Он обычно используется при многопроходном рендеринге, когда очень важно, чтобы различные варианты вершинного шейдера всегда давали один и тот же результат.

Дело в том, что в процессе оптимизации шейдера может возникнуть ситуация, когда математически эквивалентные фрагменты кода в разных шейдерах будут

давать различающиеся (правда, очень близкие) результаты. Чтобы гарантировать всегда, что этого не произойдет, используйте при описании соответствующих переменных описатель `invariant`.

```
invariant out vec3 position;
```

Можно уточнять свойства величин при помощи описателя `layout`. Наиболее распространенным примером является явное задание положения для атрибутов вершины прямо в шейдере. В этом случае нам не нужно запрашивать положения атрибутов во время выполнения – мы их уже знаем.

```
layout (location = 2) in vec3 normal;
```

Аналогично часто используется задание положения для выходных переменных фрагментного шейдера (при рендеринге сразу в несколько текстур).

```
layout (location = 1) out vec4 packedColor;
```

Значения, передаваемые из программы в шейдер (кроме атрибутов вершин), описываются при помощи описателя `uniform`. Обратите внимание, что все шейдеры, образующие одну программу для GPU, имеют общее пространство имен для uniform-переменных. Это значит, что если у вас описана одна uniform-переменная в одном шейдере (например, вершинном), то все описания переменной с этим именем в других шейдерах должны совпадать, иначе возникнет ошибка. Все uniform-переменные доступны шейдеру только для чтения.

Можно именовать uniform-переменные в именованные uniform-блоки. При этом для доступа к переменной из uniform-блока не нужно указывать имя самого блока. Для uniform-блоков при помощи описателя `layout` можно задавать размещение переменных в памяти. Значения для переменных из uniform-блока задаются при помощи буфера, привязанного к цели типа `GL_UNIFORM_BUFFER` и индексу.

```
layout (std140) uniform TransformBlock
{
      mat4  modelView;
      mat4  projection;
};

uniform LightBlock
{
      vec4  pos;
      vec3  color;
      float radius;
};
```

Кроме uniform-блоков, общий размер которых ограничен (обычно не более 64 Кб), GLSL также поддерживает еще один тип блоков – shader storage. Данные для этих блоков тоже задаются при помощи буфера (типа GL_SHADER_STORAGE_BUFFER). Однако максимальный размер такого буфера намного больше и, что самое главное, значения из этого буфера можно не только читать, но и писать.

```
struct NodeType
{
      vec3  pos;
      vec3  normal;
};
```

```
layout (binding = 0, std430) buffer Buffer
{
      int               someVariable;
      mat2        someMatrix;
      NodeType    node [];
};
```

Шейдер каждого типа имеет свои входные и выходные переменные. Так, для вершинного шейдера встроенными переменными являются целочисленная переменная gL_VertexID (индекс текущей вершины для индексированного рендеринга) и gl_InstanceID (номер выводимого экземпляра – instance при выводе с дублированием геометрии).

Стандартные выходные переменные вершинного шейдера образуют блок, приводимый ниже.

```
out gl_PerVertex
{
      vec4  gl_Position;        // координаты вершины в пр-ве отсечения
      float gl_PointSize;       // размер точки при выводе точек (GL_POINTS)
      float gl_ClipDistance[];  // массив расстояний от вершины до задаваемых пользователем
                                // плоскостей отсечения
};
```

Для тесселляционного управляющего шейдера встроенными входными переменными являются gl_PatchVerticesIn, gl_PrimitiveID и gl_InvocationID.

```
in int gl_PatchVerticesIn;      // число вершин входного примитива
in int gl_PrimitiveID;          // индекс текущего примитива в этой команде рендеринга
in int gl_InvocationID;         // индекс вызова TCS в этом примитиве
```

Также тесселляционный управляющий шейдер принимает на вход стандартные выходные переменные выходного шейдера (блок gl_PerVertex).

```
in gl_PerVertex
{
      vec4  gl_Position;
      float gl_PointSize;
      float gl_ClipDistance [];
} gl_in [gl_MaxPatchVertices];
```

Тесселляционный управляющий шейдер имеет следующие встроенные выходные переменные (описатель patch задает, что значение задается на весь примитив -patch):

```
patch out float gl_TessLevelOuter [4];
patch out float gl_TessLevelInner [2];
```

Также тесселляционный управляющий шейдер может использовать следующие стандартные (но необязательные) выходные переменные:

```
out gl_PerVertex
{
      vec4  gl_Position;
      float gl_PointSize;
      float gl_ClipDistance [];
} gl_out;
```

Тесселляционный вычислительный шейдер получает на вход следующие встроенные переменные:

```
in vec3 gl_TessCoord;
in int  gl_PatchVerticesIn;
in int  gl_PrimitiveID;
```

В переменной `gl_TessCoord` содержатся барицентрические координаты текущей вершины внутри примитива. В переменной `gl_PatchVerticesIn` содержится число вершин примитива. В переменной `gl_PrimitiveID` содержится индекс текущего примитива среди всех примитивов, выводимых текущей командой `glDraw*`.

Также тесселляционный вычислительный шейдер имеет доступ к уровням тесселляции, выданным тесселляционным управляющим шейдером.

```
patch in float gl_TessLevelOuter [4];
patch in float gl_TessLevelInner [2];
```

Кроме того, тесселляционный вычислительный шейдер также имеет доступ к блоку `gl_PerVertex`, выданному управляющим шейдером:

```
out gl_PerVertex
{
      vec4  gl_Position;
      float gl_PointSize;
      float gl_ClipDistance [];
} gl_in [gl_MaxPatchVertices];
```

В качестве стандартных выходных переменных в тесселляционном вычислительном шейдере используется массив стандартных данных – блок `gl_PerVertex`.

Стандартными входными переменными в геометрическом шейдере выступает массив `gl_in` структур `gl_PerVertex`, `gl_PrimitiveIDIn` и `gl_InvocationID`. Целочисленная переменная `gl_PrimitiveIDIn` – это индекс (номер) входного примитива, если исходить из числа примитивов, обработанных геометрическим шейдером в начале текущей команды `glDraw*`. Переменная `gl_InvocationID` содержит номер текущего экземпляра в режиме дублирования геометрии.

Стандартными выходными переменными для геометрического шейдера являются поля блока `gl_PerVertex` – `gl_Position`, `gl_PointSize` и `gl_ClipDistance`.

Стандартные входные переменные фрагментного шейдера приведены ниже.

```
in vec4      gl_FragCoord;
in bool      gl_FrontFacing;
in vec2      gl_PointCoord;
```

В переменной `gl_FragCoord` хранятся координаты фрагмента в системе координат окна. Компоненты *x* и *y* задают оконные координаты фрагмента, а *x* содержит значение *z*, которое будет записано в буфер глубины, если шейдер не произведет запись в переменную `gl_FragDepth`. В поле *w* хранится значение $1/w_{clip}$.

Переменная `gl_FrontFacing` содержит информацию о том, принадлежит ли данный фрагмент лицевому примитиву (грани). Для фрагментов примитивов, которые не являются гранями, значение `gl_FrontFacing` всегда будет равно `true`.

В переменной `gl_PointCoord` хранятся текстурные координаты данного фрагмента внутри текущей выводимой точки (при выводе примитивов типа `GL_POINTS`).

Стандартной выходной переменной для фрагментного шейдера является `gl_FragDepth`. Запись в эту переменную изменяет глубину текущего фрагмента, делая ее равной записываемому значению. Обратите внимание, что если хотя бы в одном месте фрагментного шейдера выполняется запись в `gl_FragDepth`, то вы должны гарантировать, что в данном шейдере всегда будет производиться запись в эту переменную.

Также в GLSL поддерживается большое количество встроенных функций. Ниже приводятся таблицы, содержащие описания этих функций по классам. Многие функции могут работать с различными типами данных, например, получив векторный аргумент, вернуть вектор значений. В этих случаях тип обозначается как `T`.

Таблица А.2. Преобразование углов и тригонометрические функции

Функция	Описание
`T radians (T degrees)`	Переводит градусы в радианы, т. е. возвращает $\frac{\pi}{180}$ *degress*
`T degrees ( T radians)`	Переводит радианы в градусы, т. е. возвращает $\frac{180}{\pi}$ *radians*
`T sin ( T angle)`	Вычисляет синус угла, угол задается в радианах
`T cos ( T angle )`	Вычисляет косинус угла, угол задается в радианах
`T tan ( T angle )`	Вычисляет тангенс угла, угол задается в радианах
`T asin ( T x )`	Вычисляет арксинус. Возвращает число, синус которого равен *x*. Возвращаемые значения лежат в диапазоне $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. Результат не определен, если $\|x\| > 1$
`T acos ( T x )`	Вычисляет арккосинус. Возвращает число, косинус которого равен *x*. Возвращаемые значения лежат в диапазоне $[0, \pi]$. Результат не определен, если $\|x\| > 1$
`T atan ( T y, T x )`	Вычисляет арктангенс. Возвращает число, тангенс которого равен *y*/*x*. Возвращаемые значения лежат в диапазоне $[-\pi, \pi]$. Результат не определен, если *x* и *y* равны нулю одновременно
`T atan ( T y_over_x )`	Вычисляет арктангенс. Возвращает число, синус которого равен *y_over_x*. Возвращаемые значения лежат в диапазоне $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$
`T sinh ( T x )`	Возвращает значение гиперболического синуса $(e^x - e^{-x})/2$
`T cosh ( T x )`	Возвращает значение гиперболического косинуса $(e^x + e^{-x})/2$
`T tanh ( T x )`	Возвращает значение гиперболического тангенса
`T asinh ( T x )`	Обратная функция к гиперболическому синусу
`T acosh (T x )`	Обратная функция к гиперболическому косинусу
`T atanh ( T x )`	Обратная функция к гиперболическому тангенсу

Таблица А.3. Базовые функции

Функция	Описание
`T abs (T x )`	Возвращает *x*, если $x \geq 0$, иначе возвращает $-x$
`T sign ( T x )`	Возвращает знак числа – 1, если $x > 0$, 0, если $x = 0$, и –1, если $x < 0$
`T floor ( T x )`	Возвращает значение, равное ближайшему целому числу, меньшему или равному *x*
`T ceil ( T x )`	Возвращает значение, равное ближайшему целому числу, большему или равному *x*
`T fract ( T x )`	Возвращает $x - floor(x)$

Таблица А.3 (окончание)

Функция	Описание
`T trunc ( T x )`	Возвращает значение, равное ближайшему целому к *x*, чье значение по модулю не больше |*x*|
`T round * T x )`	Возвращает значение, равное ближайшему целому к *x*. Дробная часть, равная 0.5, будет округлена в сторону, выбранную реализацией, возможно, в ту сторону, округление в которую будет быстрее. Это включает в себя вариант, что `round(x)` всегда возвращает то же самое значение, что и `roundEven(x)`
`T roundEven ( T x )`	Возвращает значение, равное ближайшему целому к *x*. Дробная часть, равная 0.5, будет округлена к ближайшему четному целому числа (соответственно, 3.5 и 4.5 будут оба округлены к 4)
`T mod ( T x, T y )`	Возвращает $x - y * floor(x/y)$
`T modf ( T y, out T x )`	Возвращает дробную часть *x* и устанавливает *i* равным целой части (в виде числа с плавающей точкой). Знак обоих возвращаемых значений будет совпадать со знаком *x*
`T min ( T x, T y )`	Возвращает минимальное из двух чисел
`T max ( T x, T y )`	Возвращает наибольшее из двух чисел
`T clamp ( T x, T minVal, T maxVal )`	Возвращает min(max(*x*, *minValue*), *maxValue*). Результат не определен, если *minValue* > *maxValue*
`T mix ( T x, T y, T a )`	Возвращает результат смешивания значений *x* и *y*, т. е. $x \cdot (1 - a) + y \cdot a$
`T step ( T1 edge, T x )`	Возвращает 0.0, если *x* < *edge*, и 1.0 в противном случае
`T smoothStep ( T edge0, T edge1, T x )`	Возвращает 0.0, если *x* < *edge*0, и 1.0, если *x* > *edge*1. При *edge*0 < *x* < *edge*1 возвращается значение из (0, 1), получаемое при помощи интерполяции Эрмита. Результат не определен, если *edge*1 $\leq$ *edge*0
`T2 isnan ( T x )`	Возвращает `true`, если *x* содержит NaN (Not-a-Number), и `false` в противном случае
`T2 isinf ( T x )`	Возвращает `true`, если *x* содержит INF (или –INF), и `false` в противном случае
`TI floatBitsToInt ( T x )` `TU floatBitsToUint ( T x )`	Возвращает целое, знаковое или беззнаковое число, представляющее число с плавающей точкой. При этом сохраняется битовое представление числа с плавающей точкой. Фактически просто трактует биты числа с плавающей точкой как целое число и возвращает его
`T intBitsToFloat ( TI x )` `T uintBitsToFloat ( TU x )`	Возвращает число с плавающей точкой, соответствующее знаковому или беззнаковому целочисленному представлению. Фактически просто трактует биты целого числа как число с плавающей точкой

Таблица А.4. Трансцендентные функции

Функция	Описание
`T pow (T x, T y )`	Вычисляет x^y. Результат не определен, если $x < 0$ или $x = 0, y \leq 0$
`T exp ( T x )`	Вычисляет e^x
`T log ( T x )`	Вычисляет $\log x$. Результат не определен, если $x \leq 0$
`T log2 ( T x )`	Вычисляет двоичный логарифм аргумента. Результат не определен, если $x \leq 0$
`T exp2 ( T x )`	Возвращает 2^x
`T sqrt ( T x )`	Вычисляет $\sqrt{x}$. Результат не определен, если $x < 0$
`T inversesqrt * T x )`	Вычисляет $1/\sqrt{x}$. Результат не определен, если $x \leq 0$

Таблица А.5. Векторные функции

Функция	Описание
`float length ( T x )`	Возвращает длину вектора *x*
`float distance ( T p0, T p1 )`	Возвращает расстояние между *p*0 и *p*1, т. е. `length(p0-p1)`

Таблица А.5 (окончание)

Функция	Описание
`float dot ( T x, T y )`	Возвращает скалярное произведение *x* и *y*
`vec3 cross ( vec3 x, vec3 y )`	Возвращает векторное произведение *x* и *y*
`T normalize ( T x )`	Возвращает вектор того же направления, что *x*, но имеющий единичную длину. Фактически просто возвращает вектор, деленный на его длину
`T faceforward ( T n , T I, T nRef )`	Если dot(*i*, *nRef*) < 0, то возвращает *n*, иначе возвращает –*n*
`T reflect ( T i, T n )`	Возвращает отраженный вектор по формуле $r = i - 2(i, n)n$. Вектор *n* должен быть нормирован
`T refract ( T I, T n, float eta`	Возвращает преломленный вектор. В случае полного внутреннего отражения возвращает нулевой вектор

Таблица А.6. Матричные функции

Функция	Описание
`T matrixCompMult ( T x, T y )`	Выполняет покомпонентное перемножение матриц
`T1 outerProduct ( T2 c, T3 r )`	Вычисляет внешнее произведение двух векторов. Произвольный элемент такого произведения равен произведению элементов из *c* и *r*, стоящих в соответствующих позициях
`T1 transpose ( T2 m )`	Возвращает транспонированную матрицу. Сама матрица *m* при этом не изменяется
`T inverse ( T m )`	Возвращает обратную матрицу к квадратной матрице *m*
`float determinant ( T m )`	Возвращает определитель квадратной матрицы *m*

Функции сравнения выполняют покомпонентное сравнение векторных значений и обычно возвращают вектор булевых значений, где каждая компонента соответствует сравнению соответствующих компонент входных векторов. В следующей таблице через `TB` обозначен вектор булевых значений, а через `T` – просто какой-то векторный тип, значения которого сравниваются.

Таблица А.7. Функции сравнения

Функция	Описание
`TB lessThan ( T x, T y )`	Возвращает результат покомпонентного сравнения $x < y$
`TB lessThanEqual ( T x, T y )`	Возвращает результат покомпонентного сравнения $x \leq y$
`TB greaterThan ( T x, T y )`	Возвращает результат покомпонентного сравнения $x > y$
`TB greaterThanEqual ( T x, T y )`	Возвращает результат покомпонентного сравнения $x \geq y$
`TB equal ( T x, T y )`	Возвращает результат покомпонентного сравнения $x = y$
`TB notEqual ( T x, T y )`	Возвращает результат покомпонентного сравнения $x \neq y$
`bool any ( TB x )`	Возвращает `true`, если хотя бы одна из компонент *x* равна `true`
`bool all ( TB x )`	Возвращает `true`, если все компоненты *x* равны `true`
`TB not ( TB x )`	Возвращает результат покомпонентного отрицания

Шейдеры могут обращаться к текстурам на всех стадиях конвейера. Однако неявное вычисление уровня детализации доступно только во фрагментном шейдере. Для представления текстур внутри шейдера используются специальные типы данных – `sampler`.

Тип семплера включает в себя как топологию текстуры (`1D`, `2D`, `3D`, `Cube`, `1DArray`, `2DArray`, `CubeArray`, `Shadow`), так и тип данных, которые из нее читаются. По умолча-

нию подразумевается чтение значений с плавающей точкой. В этом случае мы указываем только топологию текстуры, например `sampler2D`. Если, допустим, у нас трехмерная текстура с беззнаковыми целочисленными компонентами (т. е. при чтении из нее мы будем получать вектор из беззнаковых целых чисел), то в этом случае имя семплера начинается с префикса "`u`" и сам семплер имеет тип `usampler3D`. Для целочисленных знаковых компонент используется префикс "`i`".

В следующей таблице через `gsamper*` обозначена текстура заданной топологии, без уточнения, какого типа ее компоненты (т. е. `gsampler1D` обозначает `sampler1D`, `isampler1D` и `usampler1D`).

Таблица А.8. Функции для работы с текстурами

Функция	Описание

`int textureSize ( gsampler1D sampler, int lod )`
`ivec2 textureSize ( gsampler2D sampler, int lod )`
`ivec3 textureSize ( gsampler3D sampler, int lod )`
`ivec2 textureSize ( gsamplerCube sampler, int lod )`
`int textureSize ( sampler1DShadow sampler, int lod )`
`ivec2 textureSize ( sampler2DShadow sampler, int lod )`
`ivec2 textureSize ( samplerCubeShadow sampler, int lod )`
`ivec2 textureSize ( gsampler2DRect sampler )`
`ivec2 textureSize ( gsampler2DRectShadow sampler )`
`ivec2 textureSize ( gsampler1DArray sampler, int lod )`
`ivec3 textureSize ( gsampler2DArray sampler, int lod )`
`ivec2 textureSize ( sampler1DArrayShadow sampler, int lod )`
`ivec3 textureSize ( sampler2DArrayShadow sampler, int lod )`
`int textureSize ( gsamplerBuffer sampler )`
`ivec2 textureSize ( gsampler2DMS sampler, int lod )`
`ivec2 textureSize ( gsampler2DMSArray sampler )`
	Возвращает размеры уровня `lod` заданной текстуры. При этом возвращаются ширина, высота и глубина текстуры в этом порядке.
	Для текстур-массивов последней возвращаемой компонентой является число слоев в массиве

`gvec4 texture ( gsampler1D sampler, float p [, float bias])`
`gvec4 texture ( gsampler2D sampler, vec2 p [,float bias])`
`gvec4 texture ( gsampler3D sampler, vec3 p [,float bias])`
`gvec4 texture ( gsamplerCube sampler, vec3 [,float bias])`
`float texture ( sampler1DShadow sampler, vec3 p [,float bias] )`
`float texture ( sampler2DShadow sampler, vec3 p [,float bias] )`
`float texture ( samplerCubeShadow sampler, vec4 p [, float bias] )`
`gvec4 texture ( gsampler1DArray sampler, vec2 p [,float bias] )`
`gvec4 texture ( gsampler2DArray sampler, vec3 p [,float bias] )`
`float texture ( sampler1DArrayShadow sampler, vec3 p, [,float bias])`
`float texture ( sampler2DArrayShadow sampler, vec4 p )`
`gvec4 texture ( gsampler2DRect sampler, vec2 p )`
`float texture ( sampler2DRectShadow sampler, vec3 p )`
	Осуществляет чтение из текстуры с использованием текстурных координат `p`. Для текстур типа Shadow последняя компонента используется как значение, с которым производится сравнение. Для массивов номер слоя задается последней компонентой для обычных текстур и предпоследней для Shadow-текстур

`gvec4 textureOffset ( gsampler1D sampler, float p, int offset [,float bias])`
`gvec4 textureOffset ( gsampler2D sampler, vec2 p, ivec2 offset [,float bias])`
`gvec4 textureOffset ( gsampler3D sampler, vec3t p,`
`ivec3 offset [,float bias])`
`gvec4 textureOffset ( gsampler2DRect sampler, vec2 p, ivec2 offset )`
`float textureOffset ( sampler2DRectShadow sampler, vec3 p, ivec2 offset );`
`float textureOffset ( sampler1DShadow sampler, vec3 p,`
`int offset [,float bias])`
`float textureOffset ( sampler2DShadow sampler,`
`vec3, ivec2 offset [,float bias])`
`gvec4 textureOffset ( gsampler1DArray sampler, vec2 p,`
`int offset [,float bias])`
`gvec4 textureOffset ( gsampler12Array sampler, vec3 p,`
`ivec2 offset [,float bias])`
`float textureOffset ( sampler1DArrayShadow sampler,`
`vec3 p, int offset [,float bias])`
	Выполнить чтение из текстуры, только перед чтением `offset` добавляется к координатам тексела. При этом `offset` должен быть константным выражением и есть ограничения на диапазон поддерживаемых значений для `offset`.
	Обратите внимание, что `offset` не прибавляется к номеру слоя в массивах текстур

Таблица А.8 (продолжение)

Функция	Описание

`gvec4 texelFetch ( gsampler1D sampler, int p, int lod )`
`gvec4 texelFetch ( gsampler2D sampler, ivec2 p, int lod )`
`gvec4 texelFetch ( gsampler3D sampler, ivec3 p, int lod )`
`gvec4 texelFetch ( gsampler2DRect sampler, ivec2 p )`
`gvec4 texelFetch ( gsampler1DArray sampler, ivec2 p, int lod )`
`gvec4 texelFetch ( gsampler2DArray sampler, ivec3 p, int lod )`
`gvecr texelFetch ( gsamplerBuffer sampler, int p )`
`gvec4 texelFetch ( gsampler2DMS sampler, ivec2 p, int sample )`
`gvec4 texelFetch ( gsampler2DMSArray sampler, ivec3 p, int sample )`
	Используются целочисленные текстурные координаты `p` для чтения из заданной текстуры.
	Для массивов текстур номер слоя задается последней компонентой

`gvec4 texelFtechOffset ( gsampler1D sampler, int p, int lod, int offset )`
`gvec4 texelFetchOffset ( gsampler2D sampler, ivec2 p, int lod, ivec2 offset )`
`gvec4 texelFetchOffset ( gsampler3D sampler, ivec3 p, int lod, ivec3 offset )`
`gvec4 texelFetchOffset ( gsampler2DRect sampler, ivec2 p, ivec2 offset )`
`gvec4 texelFetchOffset ( gsampler1DArray sampler,`
`ivec2 p, int lod, int offset )`
`gvec4 texelFetchOffset ( gsampler2DArray sampler,`
`ivec3 p, int lod, ivec2 offset )`
	Прочесть тексел, прибавляя смещение `offset` к текстурным координатам

`gvec4 textureProj ( gsampler1D sampler, vec2 p [, float bias])`
`gvec4 textureProj ( gsampler1D sampler, vec4 p [, float bias])`
`gvec4 textureProj ( gsampler2D sampler, vec3 p [, float bias])`
`gvec4 textureProj ( gsampler2D sampler, vec4 p [, float bias])`
`gvec4 textureProj ( gsampler3D sampler, vec4 p [, float bias])`
`float textureProj ( gsampler1DShadow sampler, vec4 p [, float bias])`
`float textureProj ( gsampler2DShadow sampler, vec4 p [, float bias])`
`gvec4 textureProj ( gsampler2DRect sampler, vec3 p )`
`gvec4 textureProj ( gsampler2DRect sampler, vec4 p )`
`float textureProj ( gsampler2DRectShadow sampler, ve4 )`
	Выполнить чтение из текстуры с проектированием. Текстурные координаты берутся из всех компонент `p`, кроме последней, и делятся на последнюю компоненту `p`.
	Для Shadow-текстур третья компонента используется в качестве значения для сравнения. После вычисления координат идет обычное обращение к текстуре, как и для `texture`

`gvec4 textureProjOffset ( gsampler1D sampler, vec2 p,`
`int offset [,float bias])`
`gvec4 textureProjOffset ( gsampler1D sampler, vec4 p,`
`int offset [,float bias])`
`gvec4 textureProjOffset ( gsampler2D sampler, vec3 p,`
`ivec2 offset [,float bias])`
`gvec4 textureProjOffset ( gsampler2D sampler, vec4 p,`
`ivec2 offset [,float bias])`
`gvec4 textureProjOffset ( gsampler3D sampler, vec4 p,`
`ivec3 offset [, float bias])`
`gvec4 textureProjOffset ( gsampler2DRect sampler, vec3 p, ivec2 offset )`
`gvec4 textureProjOffset ( gsampler2DRect sampler, vec4 p, ivec2 offset )`
`float textureProjOffset ( sampler2DRectShadow sampler, vec4 p, ivec2 offset )`
`float textureProjOffset ( sampler1DShadow sampler, vec4 p,`
`int offset [, float bias])`
`float textureProjOffset ( sampler2DShadow sampler, vec4 p,`
`ivec2 offset [, float bias])`
	Выполнить перспективное чтение из текстуры (как для `textureProj`), но сместив координаты для чтения на `offset` (как в `textureOffset`)

`gvec4 textureLod ( gsampler1D sampler, float p, float lod)`
`gvec4 textureLod ( gsampler2D sampler, vec2 p, float lod )`
`gvec4 textureLod ( gsampler3D sampler, vec3 p, float lod )`
`gvec4 textureLod ( gsamplerCube sampler, vec3 p, float lod )`
`float textureLod ( sampler1DShadow sampler, vec3 p, float lod )`
`float textureLod ( sampler1DShadow sampler, vec3 p, float lod )`
`float textureLod ( sampler2DShadow sampler, vec3 p, float lod )`
`gvec4 textureLod ( gsampler1DArray sampler, vec2 p, float lod )`
`gvec4 textureLod ( gsampler2DArray sampler, vec3 p, float lod )`
`float textureLod ( sampler1DArrayShadow sampler, vec3 p, float lod )`
	Выполнить чтение из текстуры при явном задании уровня детализации λ_{base}

Таблица А.8 (продолжение)

Функция	Описание

`gvec4 textureLodOffset ( gsampler1D sampler, float p, float lod, int offset )`
`gvec4 textureLodOffset ( sampler2D sampler, vec2 p, float lod, ivec2 offset )`
`gvec4 textureLodOffset ( sampler3D sampler, vec3 p, float lod, ivec3 offset )`
`float textureLodOffset ( sampler1DShadow sampler, vec3 p, float lod, int offset )`
`float textureLodOffset ( sampler2DShadow sampler, vec3 p, float lod, ivec2 offset )`
`gvec4 textureLodOffset ( gsampler1DArray sampler, vec2 p, float lod, int offset )`
`gvec4 textureLodOffset ( gsampler2DArray sampler, vec3 p, float lod, ivec2 offset )`
`float textureLodOffset ( sampler1DArrayShadow sampler, vec3 p, float lod, int offset )`
	Выполнить чтение из текстуры с явным заданием уровня детализации и смещения

`gvec4 textureProjLod ( gsampler1D sampler, vec2 p, float lod )`
`gvec4 textureProjLod ( gsampler1D sampler, vec4 p, float lod )`
`gvec4 textureProjLod ( gsampler2D sampler, vec3 p, float lod )`
`gvec4 textureProjLod ( gsampler2D sampler, vec4 p, float lod )`
`gvec4 textureProjLod ( gsampler3D sampler, vec4 p, float lod )`
`float textureLodProj ( sampler1DShadow sampler, vec4 p, float lod )`
`float textureLodProj ( sampler2DShadow sampler, vec4 p, float lod )`
	Выполнить проективное чтение из текстуры с явным заданием уровня детализации

`gvec4 textureProjLodOffset ( gsampler1D sampler, vec2 p, float lod, int offset )`
`gvec4 textureProjLodOffset ( gsampler1D sampler, vec4 p, float lod, int offset )`
`gvec4 textureProjLodOffset ( gsampler2D sampler, vec3 p, float lod, ivec2 offset )`
`gvec4 textureProjLodOffset ( gsampler2D sampler, vec4 p, float lod, ivec2 offset )`
`gvec4 textureProjLodOffset ( gsampler3D sampler, vec4 p, float lod, ivec3 offset )`
`float textureProjLodOffset ( sampler1DShadow sampler, vec4 p, float lod, int offset )`
`float textyreProjLodOffset ( sampler2DShadow sampler, vec4 p, float lod, ivec2 offset )`
	Выполнить перспективное чтение из текстуры с явным заданием уровня детализации и смещения

`gvec4 textureGrad ( gsampler1D sampler, float p, float dpdx, float dpdy )`
`gvec4 textureGrad ( gsampler2D sampler,vec2 p, vec2 dpdx, vec2 dpdy )`
`gvec4 textureGrad ( gsampler3D sampler, vec3 p, vec3 dpdx, vec3 dpdy )`
`gvec4 textureGrad ( gsamplerCube sampler, vec3 p, vec3 dpdx, vec3 dpdy )`
`gvec4 textureGrad ( gsampler2DRect sampler, vec2 p, vec2 dpdx, vec2 dpdy )`
`float textureGrad ( sampler2DRectShadow sampler, vec3 p, vec2 dpdx, vec2 dpdy )`
`float textureGrad ( sampler1DShadow sampler, vec3 p, float dpdx, float dpdy )`
`float textureGrad ( sampler2DShadow sampler, vec3 p, vec2 dpdx, vec2 dpdy )`
`float textureGrad ( samplerCubeShadow sampler, vec3 p, vec2 dpdx, vec2 dpdy )`
`gvec4 textureGrad ( gsampler1DArray sampler, vec2 p, float dpdx, float dpdy )`
`gvec4 textureGrad ( gsampler2DArray sampler, vec3 p, vec2 dpdx, vec2 dpdy )`
`float textureGrad ( sampler1DShadow sampler, vec3 p, float dpdx, float dpdy )`
`float textureGrad ( sampler2DShadow sampler, vec4 p, vec2 dpdx, vec2 dpdy )`
	Выполнить чтение из текстуры с явным заданием градиентов текстурных координат (производных, используемых при вычислении уровня детализации)

Таблица А.8 (окончание)

Функция	Описание

`gvec4 textureGradOffset ( gsampler1D sampler, float p, float dpdx, float dpdy, int offset )`
`gvec4 textureGradOffset ( gsampler2D sampler, vec2 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`gvec4 textureGradOffset ( gsampler3D sampler, vec3 p, vec3 dpdx, vec3 dpdy, ivec3 offset )`
`gvec4 textureGradOffset ( gsampler2DRect sampler, vec2 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`float textureGradOffset ( sampler2DRectShadow sampler, vec3 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`float textureGradOffset ( sampler1DShadow sampler, vec3 p, float dpdx, float dpdy, int offset )`
`float textureGradOffset ( sampler2DShadow sampler, vec3 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`gvec4 textureGradOffset ( gsampler1DArray sampler, vec2 p, float dpdx, float dpdy, int offset )`
`gvec4 textureGradOffset ( gsampler2DArray sampler, vec3 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`float textureGradOffset ( sampler1DArrayShadow sampler, vec3 p, float dpdx, float dpdy, int offset )`
`float textureGradOffset ( sampler2DArrayShadow sampler, vec4 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
	Выполнить чтение из текстуры с явным заданием градиентов и смещения

`gvec4 textureProjGrad ( gsampler1D sampler, vec2 p, float dpdx, float dpdy )`
`gvec4 textureProjGrad ( gsampler1D sampler, vec4 p, float dpdx, float dpdy )`
`gvec4 textureProjGrad ( gsampler2D sampler, vec3 p, vec2 dpdx, vec2 dpdy )`
`gvec4 textureProjGrad ( gsampler2D sampler, vec4 p, vec2 dpdx, vec2 dpdy )`
`gvec4 textureProjGrad ( gsampler3D sampler, vec4 p, vec3 dpdx, vec3 dpdy )`
`gvec4 textureProjGrad ( gsampler2DRect sampler, vec3 p, vec2 dpdx, vec2 dpdy )`
`gvec4 textureProjGrad ( gsampler2DRect sampler, vec4 p, vec2 dpdx, vec2 dpdy )`
`float textureProjGrad ( sampler2DRectShadow sampler, vec4 p, vec2 dpdx, vec2 dpdy )`
`float textureProjGrad ( sampler1DShadow sampler, vec4 p, float dpdx, float dpdy )`
`float textureProjGrad ( sampler2DShadow sampler, vec4 p, vec2 dpdx, vec2 dpdy )`
	Выполнить перспективное текстурирование с явным заданием градиентов

`gvec4 textureProjGradOffset ( gsampler1D sampler, vec2 p, float dpdx, float dpdy, int offset )`
`gvec4 textureProjGradOffset ( gsampler1D sampler, vec4 p, float dpdx, float dpdy, int offset )`
`gvec4 textureProjGradOffset ( gsampler2D sampler, vec3 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`gvec4 textureProjGradOffset ( gsampler2D sampler, vec4 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`gvec4 textureProjGradOffset ( gsampler2DRect sampler, vec3 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`gvec4 textureProjGradOffset ( gsampler2DRect sampler, vec4 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`float textureProjGradOffset ( sampler2DRectShadow sampler, vec4 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`
`gvec4 textureProjGradOffset ( gsampler3D sampler, vec4 p, vec3 dpdx, vec3 dpdy, ivec3 offset )`
`float textureProjGradOffset ( sampler1DShadow sampler, vec4 p, float dpdx, float dpdy, int offset )`
`float textureProjGradOffset ( sampler2DShadow sampler, vec4 p, vec2 dpdx, vec2 dpdy, ivec2 offset )`	

Также есть специальные функции, доступные только во фрагментном шейдере. Для того чтобы вычислять уровень детализации при пирамидальном фильтровании, необходимо знать производные. OpenGL позволяет приближенно находить производные переменных по экранным координатам, используя конечные разности. Соответствующие функции приведены в табл. А.9.

Таблица А.9. Функции для вычисления разностных производных

Функция	Описание
`T dFdx ( T p )`	Возвращает производную *p* по экранной координате *x*
`T dFdy ( T p )`	Возвращает производную *p* по экранной координате *y*
`T Fwidth ( T p )`	Возвращает сумму модулей производных по *x* и *y*, т. е. *abs*(*dFdx*(*p*)) + *abs*(*dFdy*(*p*))

Также есть специальные функции, доступные только в геометрическом шейдере. Они приведены в табл. А.10.

Таблица А.10. Специальные функции, доступные только в геометрическом шейдере

Фукнция	Описание
`void EmitVertex ()`	Использовать текущие значения выходных переменных как новую вершину текущего примитива. После вызова значения всех выходных переменных становятся неопределенными
`void EndPrimitive ()d`	Завершает текущий примитив и начинает новый

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Боресков Алексей Викторович

**Программирование компьютерной графики.
Современный OpenGL**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 30,23. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**