O'REILLY®

Unity для разработчика

Мобильные мультиплатформенные игры

Джон Мэннинг,
Пэрис Батфилд-Эддисон

ПИТЕР®

Mobile Game Development with Unity

Build Once, Deploy Anywhere

Jon Manning and Paris Buttfield-Addison

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Unity
ДЛЯ РАЗРАБОТЧИКА

Мобильные мультиплатформенные игры

Джон Мэннинг,
Пэрис Батфилд-Эддисон

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

2018

Джон Мэннинг, Пэрис Батфилд-Эддисон

Unity для разработчика. Мобильные мультиплатформенные игры

Перевел с английского *А. Киселев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Художественный редактор	*С. Заматевская*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

ББК 32.973.2-018
УДК 004.42

Мэннинг Д., Батфилд-Эддисон П.

М97 Unity для разработчика. Мобильные мультиплатформенные игры. — СПб.: Питер, 2018. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0541-0

Добро пожаловать в мир Unity! Создавайте игры, работающие на любых мобильных устройствах, телефонах и планшетах.

Освойте игровой движок Unity, принципы создания игр, работу с графикой, системами частиц и многое другое, научитесь создавать двухмерные и трехмерные игры, узнайте о продвинутых возможностях Unity.

Советы профессиональных разработчиков помогут быстро начать работу и сразу получить красивый, качественный и интерактивный 3D- и 2D-контент.

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

ББК 32.973.2-018
УДК 004.42

ISBN 978-1491944745 англ.	Authorized Russian translation of the English edition of Mobile Game Development with Unity © 2017 Secret Lab. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same
ISBN 978-5-4461-0541-0	© Перевод на русский язык ООО Издательство «Питер», 2018
	© Издание на русском языке, оформление ООО Издательство «Питер», 2018
	© Серия «Бестселлеры O'Reilly», 2018

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 191123, Россия, город Санкт-Петербург, улица Радищева, дом 39, корпус Д, офис 415. Тел.: +78127037373.
Дата изготовления: 03.2018. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Подписано в печать 14.03.18. Формат 70x100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор». 142300, Московская область, г. Чехов, ул. Полиграфистов, 1. Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Оглавление

Предисловие

Добро пожаловать в «Unity для разработчика. Мобильные мультиплатформенные игры»! В этой книге мы проведем вас от начала и до конца через весь процесс создания двух законченных игр, а также познакомим не только с основными, но и с продвинутыми идеями и приемами использования Unity.

Книга разделена на четыре части.

Часть I знакомит с игровым движком Unity и рассматривает основные понятия, в том числе организацию игры, графику, сценарии, звуки, физику и систему частиц. Часть II проведет вас через процесс создания законченной двумерной игры с использованием Unity, в которой гномик на веревке пытается добраться до сокровищ. Часть III рассматривает построение законченной трехмерной игры, в которой вы увидите космические корабли, астероиды и многое другое. Часть IV исследует некоторые продвинутые возможности Unity, включая освещение, систему поддержки графического пользовательского интерфейса, а также способы расширения редакторов Unity, хранение ресурсов в Unity, развертывание игр и платформенно-зависимые особенности.

Если у вас появятся какие-либо предложения или вопросы, присылайте их нам на адрес электронной почты *unitybook@secretlab.com.au*.

Ресурсы, используемые в книге

Дополнительный материал (изображения, звуки, исходный код примеров, упражнения, список опечаток и т. д.) можно скачать на *http://secretlab.com.au/books/unity*.

Аудитория и подход

Эта книга адресована всем, кто хочет создавать игры, но не имеет предыдущего опыта разработок.

Unity поддерживает несколько языков программирования. В этой книге мы будем использовать C#. Мы будем полагать, что вы уже знаете, как программировать на этом относительно современном языке, но не требуем, чтобы вы знали его в совершенстве, достаточно владеть хотя бы основами.

Редактор Unity работает как в macOS, так и в Windows. Мы используем macOS, поэтому все скриншоты, которые приводятся в книге, сделаны именно в этой ОС. Но все, о чем мы будем рассказывать, в равной степени относится и к Windows, за одним маленьким исключением: сборки игр под iOS на Unity. Мы еще остановимся на этом, когда придет время, но собирать игры под iOS в Windows вы не сможе-

те. В Windows отлично собираются игры под Android, а в macOS можно собирать игры как под iOS, так и под Android.

Прежде чем заняться созданием игр, вам необходимо понять основы их проектирования, а также самого игрового движка Unity, поэтому в первой части книги мы изучим эти вопросы. Во второй и третьей частях мы рассмотрим процесс создания двумерных и трехмерных игр соответственно, а в четвертой части расскажем о других возможностях Unity, знание которых вам пригодится.

Мы будем полагать, что вы достаточно уверенно ориентируетесь в своей операционной системе и используете мобильное устройство (будь то iOS или Android).

Мы не будем касаться вопросов создания изображений или звуков для игр, но предоставляем все ресурсы, необходимые для создания двух игр, рассматриваемых в этой книге.

Типографские соглашения

В этой книге приняты следующие типографские соглашения.

Курсив

Используется для обозначения новых терминов, адресов URL, адресов электронной почты, имен файлов и расширений имен файлов.

`Моноширинный шрифт`

Используется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных, функции, базы данных, типы данных и переменных окружения, инструкции и ключевые слова.

`Моноширинный жирный`

Обозначает команды или другой текст, который должен вводиться пользователем буквально.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так выделяются советы и предложения.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Дополнительные материалы (примеры кода, упражнения, список опечаток и т. д.) можно скачать на *http://secretlab.com.au/books/unity*.

Настоящая книга поможет вам в решении ваших рабочих задач. В общем случае вы можете использовать примеры кода из этой книги в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные фрагменты кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Mobile Game Development with Unity by Jonathon Manning and Paris Buttfield-Addison (O'Reilly). Copyright 2017 Jon Manning and Paris Buttfield-Addison, 978-1-491-94474-5».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Электронная библиотека O'Reilly Safari

Safari (прежнее название Safari Books Online) — это обучающая и справочная платформа для предприятий, правительств, преподавателей и отдельных лиц, основанная на подписке.

Подписчики имеют доступ к тысячам книг, видеоматериалов, справочников, интерактивных руководств и спискам воспроизведения от более чем 250 издателей, включая O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesly Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett и Course Technology.

За подробной информацией обращайтесь по адресу *http://oreilly.com/safari*.

Как связаться с нами

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North Sebastopol, CA 95472
800-998-9938 (США или Канада)
707-829-0515 (международный и местный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на странице книги *http://bit.ly/Mobile-Game-Dev-Unity*.

Свои пожелания и вопросы технического характера отправляйте по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о книгах, обсуждения, конференции и новости вы найдете на веб-сайте издательства: *http://www.oreilly.com*.

Ищите нас в Facebook: *http://facebook.com/oreilly*.

Следуйте за нами в Твиттере: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Джон и Парис выражают благодарность потрясающим редакторам, особенно Брайану Макдональду (Brian MacDonald, *@bmac_editor*) и Рейчел Румелиотис (Rachel Roumeliotis, *@rroumeliotis*), за их помощь в создании этой книги. Спасибо вам за ваш энтузиазм! Спасибо также всем сотрудникам издательства O'Reilly Media за то, что выпускают такие книги.

Спасибо также нашим семьям, что поддерживают наше увлечение разработкой игр, а также всем в MacLab и OSCON (вы знаете, о ком это мы) за содействие и энтузиазм. Отдельное спасибо нашему техническому обозревателю, доктору Тиму Нудженту (Dr. Tim Nugent, *@the_mcjones*).

От издательства

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

Мы будем рады узнать ваше мнение!

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

ЧАСТЬ I
Основы Unity

В эту книгу включено многое из того, что вы должны знать для эффективной разработки мобильных игр с использованием игрового движка Unity. Три главы первой части познакомят вас с Unity, проведут экскурсию по приложению и расскажут, как программировать в Unity на языке C#.

1 Введение в Unity

Изучение игрового движка Unity мы начнем с основ: что такое Unity, когда он может пригодиться и как его установить. Также мы определим некоторые ограничения, действующие здесь; в конце концов, вы держите в руках книгу, рассказывающую о разработке для мобильных платформ, а не о разработке *вообще*. Такая книга была бы намного тяжелее для чтения и освоения. Мы постарались уберечь вас от такой напасти.

«Привет, книга!»

Прежде чем погрузиться в изучение Unity, уточним, что в этой книге рассказывается о создании игр для мобильных платформ.

Мобильные игры

Итак, что такое мобильная игра и чем она отличаются от других видов игр? А если точнее, как эти различия влияют на решения, которые приходится принимать при разработке и дальнейшей реализации игры?

Пятнадцать лет назад под мобильными играми подразумевалось одно из двух:

- Невероятно простая игра с минимальным набором действий, простой графикой и элементарным сценарием.
- Нечто более сложное, доступное только на специализированных мобильных игровых устройствах и созданное компаниями, имеющими доступ к дорогим комплектам библиотек для разработки мобильных игр.

Такое деление было обусловлено сложностью аппаратных средств и ограниченной доступностью дистрибутивов. Если вы хотели написать хоть сколько-нибудь сложную игру (под *сложностью* мы подразумеваем возможность одновременно выполнять на экране несколько движений), то вам требовалась существенная вычислительная мощность, которой обладали только дорогие портативные игровые консоли, такие как наладонники Nintendo. Поскольку производители консолей также владели каналами распространения игр и хотели контролировать всё и вся,

получение разрешения на создание игр для их мощного оборудования представляло определенную проблему.

Однако с течением времени оборудование стало дешевле и для разработчиков открылись новые возможности. В 2008 году компания Apple открыла свой iPhone для разработчиков, а Google в том же году выпустила свою платформу Android. Спустя годы iOS и Android превратились в очень мощные платформы, а мобильные игры стали самыми популярными видеоиграми в мире.

В наши дни под мобильными играми подразумевается одно из трех:

- Простая игра с тщательно подобранными взаимодействиями, графикой и ограниченной сложностью, потому что эти аспекты лучше всего соответствуют архитектуре игр.
- Более сложные игры, доступные для широкого круга устройств, от специализированных игровых консолей до смартфонов.
- Мобильные версии игр, первоначально создававшиеся для консолей или персональных компьютеров.

Движок Unity можно использовать для создания игр любого из этих трех видов; в этой книге основное внимание будет уделено первому. После исследования Unity и приемов его использования мы перейдем к созданию двух игр, отвечающих этим требованиям.

«Привет, Unity!»

Теперь, немного прояснив наши цели, поговорим об инструменте, *с помощью* которого мы предполагаем достигнуть их: об игровом движке Unity.

Что такое Unity?

На протяжении многих лет основное внимание разработчиков Unity было сосредоточено на *демократизации разработки игр*, чтобы любой желающий мог написать игру и сделать ее доступной самой широкой аудитории. Однако никакой программный пакет не может идеально подходить для всех ситуаций, поэтому вы должны знать, когда с успехом можно использовать Unity, а когда лучше поискать другой программный пакет.

Движок Unity особенно хорошо подходит в следующих ситуациях.

При создании игры для нескольких устройств.

> Кроссплатформенная поддержка Unity является, пожалуй, лучшей в индустрии, и если необходимо создать игру, действующую на нескольких платформах (или даже на нескольких *мобильных* платформах), Unity может оказаться лучшим выбором для этого.

Когда важна скорость разработки.

Вы можете потратить месяцы на разработку игрового движка, обладающего всеми необходимыми функциями. Или можете использовать сторонний движок, такой как Unity. Справедливости ради нужно сказать, что существуют другие движки, такие как Unreal или Cocos2D; однако это ведет нас к следующему пункту.

Когда требуется полный набор функций, но нет желания создавать собственный комплект инструментов.

Unity обладает идеальными возможностями для создания мобильных игр и поддерживает очень простые способы формирования их контента.

При этом в некоторых ситуациях Unity оказывается не так полезен. В том числе:

В играх, не требующих частой перерисовки сцены.

Unity хуже подходит для реализации игр, не требующих интенсивных операций с графикой, так как движок Unity перерисовывает каждый кадр. Это необходимо для поддержки анимации в масштабе реального времени, но требует больших затрат энергии.

Когда требуется точное управление действиями движка.

Отказавшись от приобретения лицензии на исходный код Unity (такая возможность есть, но она используется редко), вы теряете возможность контролировать поведение движка на низком уровне. Это не значит, что вы вообще теряете контроль над работой движка Unity (в большинстве случаев этого и не требуется), но кое-что окажется вам недоступно.

Как получить Unity

Unity доступен для Windows, macOS и Linux и распространяется в трех основных версиях: *Personal*, *Plus* и *Pro*.

Во время работы над книгой (середина 2017 года) поддержка Linux находилась на экспериментальном уровне.

- Версия Personal предназначена для индивидуальных разработчиков, желающих использовать Unity для создания своих игр. Версия Personal доступна бесплатно.
- Версия Plus предназначена для индивидуальных разработчиков и небольших команд. На период написания книги версия Plus была доступна за 35 долларов США в месяц.

- Версия Pro предназначена для команд, от небольших до крупных. На период написания книги версия Pro была доступна за 125 долларов США в месяц.

Доступна также версия Enterprise движка Unity, предназначенная для крупных команд, но авторы книги почти не использовали ее.

Все редакции Unity обладают практически одинаковыми возможностями. Главное отличие бесплатной версии Personal заключается в том, что она добавляет в игру экран-заставку с логотипом Unity. Бесплатная версия доступна только для индивидуальных разработчиков или для организаций, имеющих доход менее 100 000 долларов США в год, тогда как для версии Plus этот порог составляет 200 000 долларов США. Редакции Plus и Pro включают также улучшенные версии некоторых служб, например приоритетные очереди в службе Unity Cloud Build (подробнее обсуждается в разделе «Unity Cloud Build» в главе 17).

Загрузить Unity можно по адресу *https://store.unity.com*. После установки можно приступать к знакомству с Unity, чем мы и займемся в следующей главе.

2 Обзор Unity

После установки Unity будет полезно потратить некоторое время на изучение среды разработки. Пользовательский интерфейс Unity относительно простой, однако он включает довольно много отдельных элементов, на изучение которых стоит потратить какое-то время.

Редактор

После первого запуска Unity вам будет предложено ввести лицензионный ключ и войти в свою учетную запись. Если у вас нет своей учетной записи или вы не хотите входить в нее, этот шаг можно пропустить.

Если не войти в свою учетную запись, Cloud Builder и другие службы Unity будут недоступны. Мы рассмотрим службы Unity в главе 17; мы редко будем использовать их на начальном этапе, но войти в систему было бы неплохо.

Сразу после запуска откроется начальный экран Unity, где на выбор можно создать новый проект или открыть существующий (рис. 2.1).

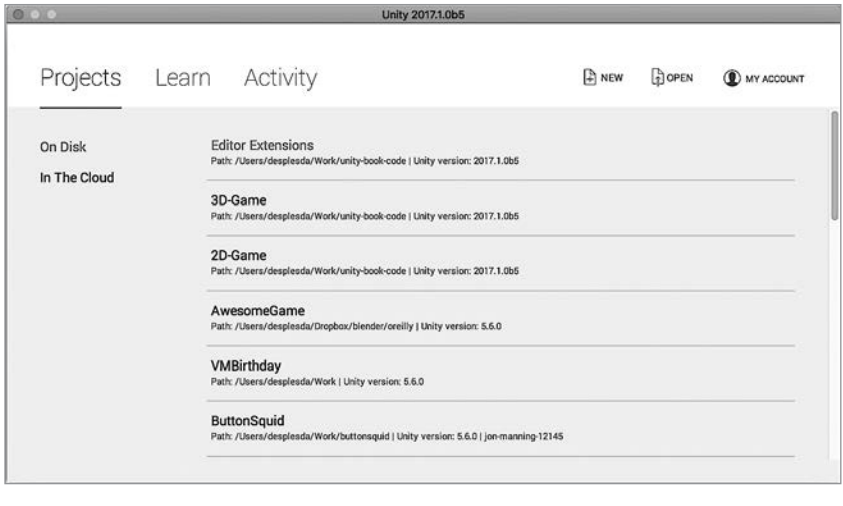

Рис. 2.1. Начальный экран Unity после входа в учетную запись

Если щелкнуть по кнопке New (Новый) справа вверху, Unity предложит ввести некоторую информацию для настройки нового проекта (рис. 2.2), включая название проекта, каталог для его хранения и тип — двумерный или трехмерный (2D или 3D).

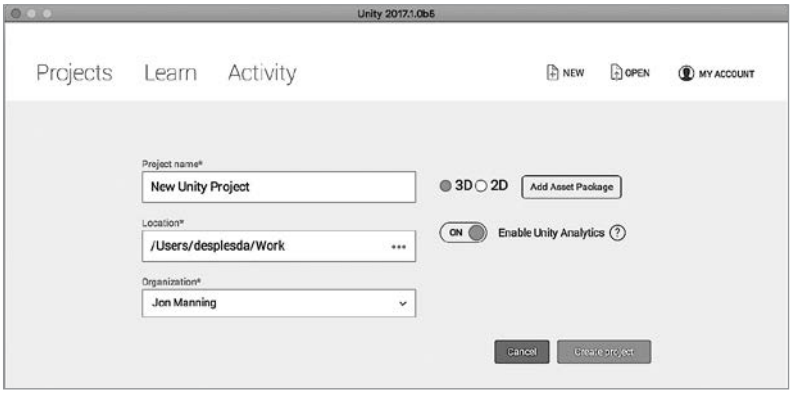

Рис. 2.2. Создание нового проекта

Выбор между 2D (двумерный) и 3D (трехмерный) не влечет больших различий. Двумерные проекты по умолчанию отображаются сбоку, а трехмерные — в трехмерной перспективе. Этот параметр проекта можно изменить в любой момент в инспекторе Editor Settings (Настройки редактора; см. раздел «Инспектор» ниже в этой главе, чтобы узнать, как получить доступ к нему).

После щелчка по кнопке Create project (Создать проект) Unity сгенерирует файлы проекта на диске и откроет проект в редакторе (рис. 2.3).

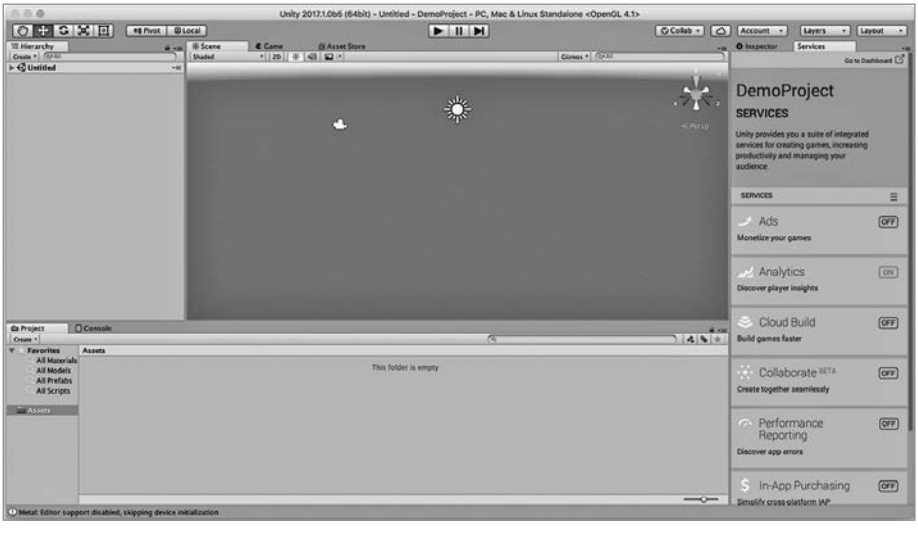

Рис. 2.3. Редактор

СТРУКТУРА ПРОЕКТА

Проект Unity — это не единственный файл, а папка, содержащая три наиболее важные подпапки: *Assets*, *ProjectSettings* и *Library*. Папка *Assets* содержит все файлы, используемые игрой: уровни, текстуры, звуковые эффекты и сценарии. Папка *Library* содержит внутренние данные для Unity, а папка *ProjectSettings* — файлы с настройками проекта.

В основном вам не придется касаться файлов в папках *Library* и *ProjectSettings*.

Кроме того, если вы пользуетесь системой управления версиями, такой как Git или Perforce, вам не нужно сохранять папку *Library* в своем репозитории, однако вы должны сохранить папки *Assets* и *ProjectSettings*, чтобы ваши коллеги смогли использовать те же ресурсы и настройки, что и вы.

Если все это звучит незнакомо, можете просто игнорировать эту информацию, но мы настоятельно рекомендуем следовать за стандартами управления исходным кодом — это может быть крайне полезно!

Интерфейс Unity отображается как набор панелей. Каждая панель имеет вкладку слева вверху, за которую панель можно переместить и тем самым изменить макет приложения. Также, перемещая панель за вкладку, ее можно превратить в самостоятельное окно. Не все панели видимы по умолчанию, и по мере разработки игры вы будете открывать новые панели через меню Window (Окно).

Если в какой-то момент вы почувствуете, что запутались в панелях, то всегда можно вернуть приложению начальный вид, выбрав в меню Window (Окно) пункт Layouts ▸ Default (Макет ▸ По умолчанию).

Режимы проигрывания и редактирования

Редактор Unity всегда находится в одном из двух режимов: Edit Mode (Режим редактирования) или Play Mode (Режим проигрывания). В режиме редактирования, который действует по умолчанию, можно создавать сцены, настраивать игровые объекты и выполнять другие действия, связанные с созданием игры. В режиме проигрывания вы можете играть в свою игру и взаимодействовать со сценой.

Чтобы перейти в режим проигрывания, щелкните по кнопке Play (Играть) в верхней части окна редактора (рис. 2.4) — Unity запустит игру; чтобы покинуть режим проигрывания, щелкните по кнопке Play (Играть) еще раз.

Находясь в режиме проигрывания, можно приостановить игру, щелкнув по пиктограмме Pause (Пауза) в центре панели управления режимом проигрывания.

Войти или выйти из режима проигрывания можно также нажатием комбинации клавиш Command-P (Ctrl-P на PC).

Рис. 2.4. Кнопки управления режимом проигрывания

Повторный щелчок по ней возобновит игру. Можно также попросить Unity выполнить переход на один кадр вперед и приостановиться, щелкнув по кнопке Step (Шаг) справа в панели.

Все изменения, произведенные в сцене, отменяются после выхода из режима проигрывания. К ним относятся любые изменения, произошедшие в результате выполнения сценария игры, а также изменения, произведенные вами в игровых объектах в режиме проигрывания. Дважды проверьте, в каком режиме находится редактор, прежде чем производить изменения!

Теперь поближе познакомимся со вкладками, открытыми по умолчанию. В этой главе мы будем ссылаться на расположения панелей, как они находятся по умолчанию. (Если вы не видите какую-то панель, убедитесь, что используете макет по умолчанию.)

Представление сцены

Представление Scene (Сцена) — панель в центре окна. В этом представлении вы будете проводить большую часть времени, потому что именно здесь отображается содержимое *сцен* игры.

Проекты в Unity поделены на сцены. Каждая сцена содержит коллекцию игровых объектов; создавая и изменяя игровые объекты, вы создаете свои игровые миры.

Сцену можно рассматривать как один уровень в игре, но сцены также помогают разделить игру на управляемые фрагменты. Например, главное меню игры обычно реализуется в виде отдельной сцены, как и каждый ее уровень.

Переключатель режимов

Представление сцены может находиться в одном из пяти разных режимов. Переключатель режимов, находящийся слева вверху в окне (рис. 2.5), управляет режимом взаимодействий с представлением сцены.

Рис. 2.5. Переключатель режимов в представлении сцены, в данном случае включен режим Translate (Перемещение)

Далее перечислены пять режимов, слева направо.

Режим захвата (Grab mode)

В этом режиме можно нажать левую кнопку мыши и сдвинуть представление.

Режим перемещения (Translation mode)

В этом режиме можно перемещать выделенные объекты.

Режим вращения (Rotation mode)

В этом режиме можно поворачивать выделенные объекты.

Режим масштабирования (Scale mode)

В этом режиме можно изменять размеры выделенных объектов.

Режим прямоугольника (Rectangle mode)

В этом режиме можно перемещать выбранные объекты и изменять их размеры, используя двумерные маркеры. Это особенно удобно при формировании двумерных сцен или пользовательского интерфейса.

Выбирать объекты можно в любом режиме, кроме режима захвата.

Переключение между режимами представления сцены можно осуществлять с помощью переключателя или клавишами Q, W, E, R и T.

Навигация

Поддерживается несколько способов навигации в представлении сцены:

- Щелкнуть по пиктограмме с изображением руки слева вверху, чтобы перейти в режим захвата, затем нажать левую кнопку мыши и сдвинуть сцену в нужном направлении.
- Удерживая нажатой клавишу Option (Alt на PC), нажать левую кнопку мыши и повернуть сцену на нужный угол.
- Выбрать объект в сцене, щелкнув по нему левой кнопкой или выбрав его в панели иерархии (о которой рассказывается в разделе «Иерархия» ниже в этой главе), поместить указатель мыши в пределы панели с представлением сцены и нажать F, чтобы передать фокус ввода выбранному объекту.
- Удерживая нажатой правую кнопку, переместить мышь, чтобы осмотреться; пока правая кнопка мыши остается нажатой, можно использовать клавиши W, A, S и D, чтобы сместить камеру вперед, влево, назад и вправо. Клавишами Q и E камеру можно смещать вверх и вниз. Если при этом еще нажать и удерживать клавишу Shift, перемещения будут происходить быстрее.

Для перехода в режим захвата вместо щелчка по пиктограмме с изображением руки можно нажать клавишу Q.

Управление маркерами

Справа от переключателя режимов находится панель управления маркерами (Handle) (рис. 2.6). Элементы управления на этой панели позволяют определить

местоположение и ориентацию маркеров для перемещения, вращения и масштабирования, появляющихся при выборе объекта.

Рис. 2.6. Панель управления маркерами; на этом изображении выбрано позиционирование в опорной точке (Pivot) и локальная ориентация (Local)

Эти две кнопки дают возможность управлять позиционированием маркеров и их ориентацией.

Позиционирование можно определить как Pivot (В опорной точке) или Center (В центре).

- Когда выбрано позиционирование Pivot (В опорной точке), маркеры появляются в опорной точке объекта. Например, опорная точка трехмерной модели человека обычно находится между ступней.
- Когда выбрано позиционирование Center (В центре), маркеры появляются в центре объекта, независимо от местонахождения его опорной точки.

Ориентацию можно определить как Local (Локальная) или Global (Глобальная).

- Когда выбрана ориентация Local (Локальная), маркеры ориентируются относительно выбранного объекта. То есть если повернуть объект так, что его *верх* будет направлен вбок, стрелка *вверх* также будет направлена вбок. Это позволит переместить объект в его «локальном» направлении вверх.
- Когда выбрана ориентация Global (Глобальная), маркеры ориентируются относительно мировой системы координат, то есть направление *вверх* всегда будет направлено вверх, без учета поворота объекта. Это может пригодиться, например, для перемещения повернутого объекта.

Иерархия

Панель Hierarchy (Иерархия), как показано на рис. 2.7 находится слева от представления сцены и отображает список всех объектов в сцене, открытой в данный момент. При работе со сложными сценами иерархия позволяет быстро находить нужные объекты по именам.

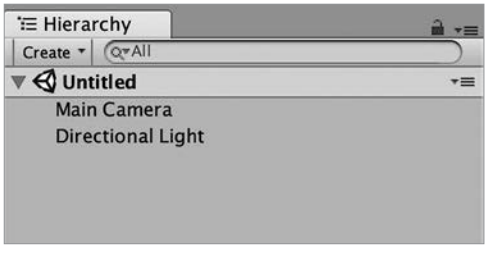

Рис. 2.7. Панель Hierarchy (Иерархия)

Иерархия, как следует из названия, позволяет также видеть отношения родитель–потомок между объектами. В Unity объекты могут содержать другие объекты; панель иерархии позволяет исследовать такие деревья объектов. Также поддерживается возможность переупорядочивать объекты в списки, перетаскивая их мышью.

В верхней части панели с иерархией находится поле поиска, в которое можно ввести имя искомого объекта. Это особенно удобно при работе со сложными сценами.

Обозреватель проекта

Обозреватель проекта (рис. 2.8) в нижней части окна редактора отображает содержимое папки *Assets* проекта. В этой панели можно работать с ресурсами игры и управлять структурой папки.

Перемещайте, переименовывайте и удаляйте файлы ресурсов только в обозревателе проекта. В этом случае Unity сможет следить за изменениями в файлах. Если для этой цели использовать другие инструменты (такие, как Finder в macOS или Проводник в Windows), Unity не сможет отследить изменения. В результате Unity может не найти нужные файлы, и ваша игра станет неработоспособной.

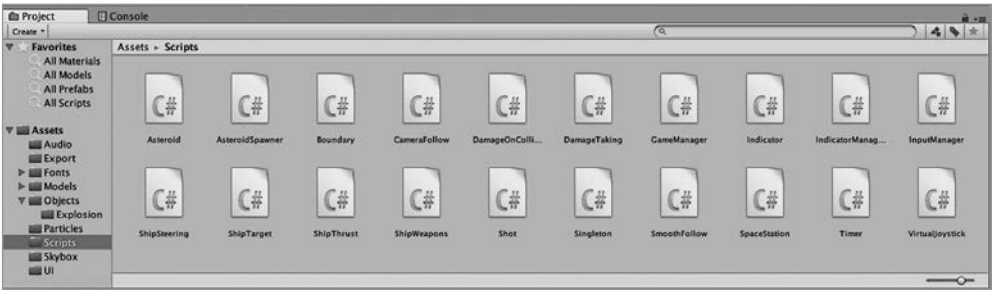

Рис. 2.8. Обозреватель проекта (здесь отображается набор ресурсов из другого проекта; во вновь созданном проекте папка Assets ничего не содержит)

Обозреватель проекта может отображать содержимое в одну или в две колонки. На рис. 2.8 показано отображение в две колонки; в левой колонке выводится список папок, а в правой — содержимое папки, выбранной в данный момент. Отображение в две колонки лучше подходит для широких экранов.

При отображении в одну колонку (рис. 2.9) все папки и их содержимое выводятся в одном списке. Такой способ отображения лучше подходит для узких экранов.

Инспектор

Представление Inspector (Инспектор), как показано на рис. 2.10, — одно из наиболее важных в редакторе, второе по важности после представления сцены. Инспек-

тор отображает информацию об объекте, выбранном в данный момент, и именно здесь вы будете осуществлять настройки своих игровых объектов. Инспектор находится справа в окне редактора; по умолчанию он находится в одной группе со вкладкой Services (Службы).

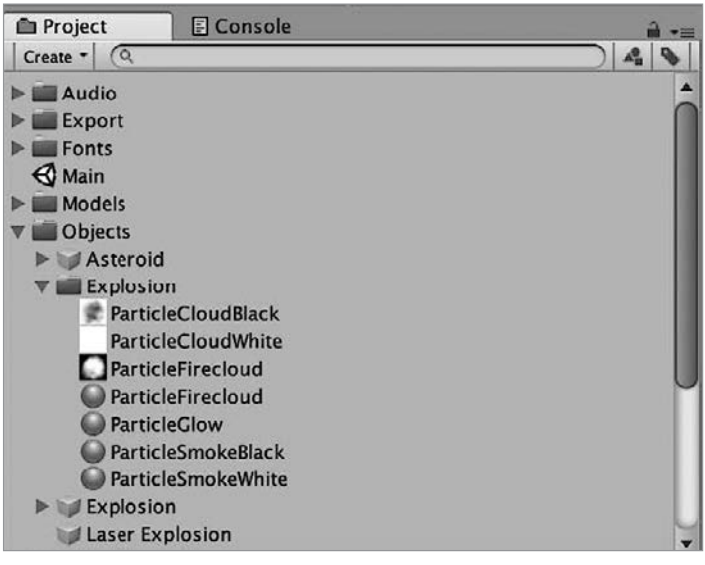

Рис. 2.9. Обозреватель проекта в режиме отображения в одну колонку

Инспектор отображает список всех компонентов, подключенных к выбранному объекту или ресурсу. Для разных компонентов отображается разная информация; когда мы приступим к разработке своих проектов во второй и третьей частях, вы увидите, насколько велико их разнообразие. Со временем мы будем все ближе и ближе знакомиться с инспектором и его содержимым.

Кроме информации о текущем выбранном объекте, инспектор отображает также настройки проекта, которые можно открыть, выбрав пункт меню Edit ▸ Project Settings (Правка ▸ Настройки проекта).

Представление игры

Представление Game (Игра), находящееся в одной группе вкладок с представлением Scene (Сцена), отображает вид с текущей активной камеры. После перехода в режим проигрывания (см. раздел «Режимы редактирования и проигрывания» выше в этой главе) автоматически активируется представление игры, где вы можете поиграть в свою игру.

Представление игры не обладает собственными интерактивными средствами — оно способно только отображать картинку с активной камеры. Это означает, что попытки взаимодействовать с представлением игры, когда редактор находится в режиме редактирования, ни к чему не приведут.

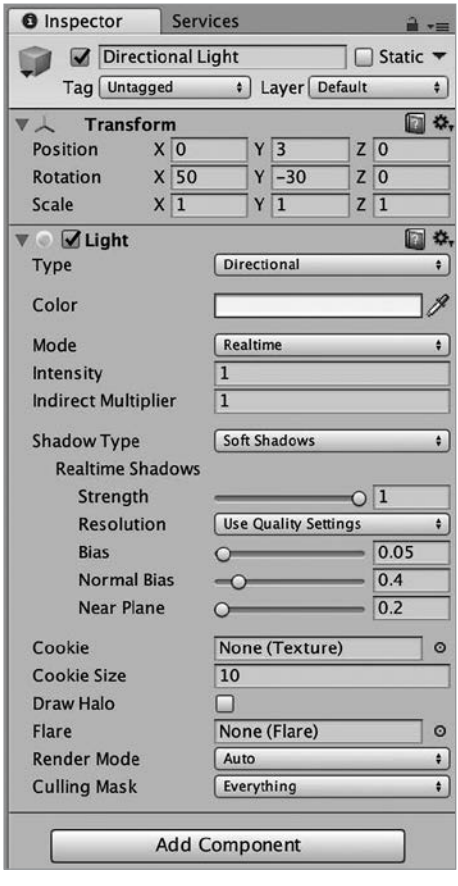

Рис. 2.10. Представление Inspector (Инспектор) отображает информацию об объекте, содержащем компонент Light

В заключение

Теперь вы знаете, как управлять представлениями в Unity, и готовы начать создавать что-то свое. В такой сложной программе всегда можно найти, что изучить; уделите время, чтобы покопаться в ней.

В следующей главе мы поговорим о приемах работы с игровыми объектами и сценариями. После этого вы будете готовы приступить к созданию своих игр.

3 Выполнение сценариев в Unity

Чтобы игра заработала, вы должны определить, что в действительности должно в ней *происходить*. Unity предоставляет все необходимое для этого, в том числе механизм отображения графики, средства получения ввода пользователя и инструменты воспроизведения звуков; вам остается только придать своей игре уникальность.

Для этого вы должны написать *сценарии* и добавить их в свои игровые объекты. В этой главе мы познакомим вас с системой поддержки сценариев в Unity, использующей язык программирования C#.

ЯЗЫКИ В UNITY

Для программирования в Unity у вас на выбор есть несколько языков. Официально Unity поддерживает два языка: C# и «JavaScript».

Мы взяли JavaScript в кавычки, потому что в действительности это не совсем тот язык JavaScript, широко известный в мире и с которым вы можете быть знакомы. Этот язык хотя и похож на JavaScript, но имеет множество отличий, которых так много, что и пользователи Unity, и некоторые разработчики Unity часто называют его UnityScript.

Мы не будем использовать JavaScript из Unity в этой книге. Во-первых, потому что примеры, демонстрируемые в справочниках по Unity, в основном написаны на C#, и нам кажется, что сами разработчики Unity предпочитают C#.

Во-вторых, C# в Unity — это тот же язык C#, который можно встретить где угодно, тогда как версия JavaScript в Unity весьма специфична для Unity. Из этого следует, что вам проще будет найти помощь, касающуюся этого языка.

Краткий курс C#

Сценарии для игр в Unity мы пишем на языке, который называется C#. В этой книге мы не будем знакомить вас с основами программирования (для этого в ней не так много места!), но мы опишем некоторые аспекты, которые важно знать и помнить.

Издательство O'Reilly выпустило отличное руководство «C# in a Nutshell», написанное Джозефом (Joseph) и Беном Албахари (Ben Albahari)[1].

В качестве краткого введения приведем фрагмент кода на C# и выделим некоторые наиболее важные элементы:

```
using UnityEngine; ❶
namespace MyGame { ❷

     [RequireComponent(typeof(SpriteRenderer))] ❸
    class Alien : MonoBehaviour { ❹
        public bool appearsPeaceful; ❺
        private int cowsAbducted;

        public void GreetHumans() {
            Debug.Log("Hello, humans!");

            if (appearsPeaceful == false) {
                cowsAbducted += 1;
            }
        }
    }
}
```

❶ Ключевое слово `using` сообщает пользователю, какие пакеты вы предполагаете использовать. Пакет `UnityEngine` содержит определения основных типов данных Unity.

❷ C# позволяет вам заключать свои типы в *пространства имен*, что помогает избегать конфликтов.

❸ В квадратных скобках определяются *атрибуты*, позволяющие добавлять дополнительную информацию о типах и методах.

❹ С помощью ключевого слова `class` вы можете объявлять свои классы и указывать суперкласс через двоеточие. Класс, наследующий `MonoBehaviour`, можно использовать как компонент сценария.

❺ Переменные, объявленные в классе, называют *полями*.

Mono и Unity

Система поддержки сценариев в Unity основана на фреймворке Mono. Mono — это открытая реализация Microsoft .NET Framework, а это означает, что кроме библиотек, поставляемых вместе с Unity, вы можете использовать полный набор библиотек .NET.

[1] Джозеф Албахари, Бен Албахари, «C# 6.0. Справочник: Полное описание языка». М.: Вильямс, 2016. — *Примеч. пер.*

Распространенным заблуждением является то, что Unity реализован поверх Mono. В действительности это не так; он просто использует Mono как механизм выполнения сценариев. Unity использует Mono для выполнения сценариев на обоих языках: C# и UnityScript (который в Unity называется «JavaScript»; см. врезку «Языки в Unity» выше).

Версии C# и .NET Framework, доступные в Unity, старее текущих версий. На момент написания книги, в начале 2017-го, в Unity использовался язык C# версии 4 и .NET Framework версии 3.5. Причина такого отставания в том, что в Unity используется своя ветка проекта Mono, которая отклонилась от основной ветви несколько лет тому назад. Это означает, что разработчики Unity могут добавлять свои специфические особенности, которые в первую очередь обусловлены спецификой работы компиляторов кода для мобильных устройств.

Unity находится на полпути к завершению обновления своих инструментов компиляции, чтобы дать пользователям возможность использовать последние версии языка C# и .NET Framework. Но пока это не случится, вам придется пользоваться немного устаревшими версиями.

По этой причине, если вы ищете в интернете примеры на C# или советы, в большинстве случаев предпочтительнее искать код, характерный для Unity. Аналогично, программируя на C# для Unity, вы будете использовать комбинацию Mono API (для всего, что обычно предоставляет большинство платформ) и Unity API (для доступа к инструментам игрового движка).

MonoDevelop

MonoDevelop — это среда разработки, включаемая в состав Unity. Главная задача MonoDevelop — быть текстовым редактором, в котором вы пишете свои сценарии; однако она обладает некоторыми полезными свойствами, которые могут упростить программирование.

Если дважды щелкнуть на любом файле сценария в проекте, Unity откроет редактор, указанный в настройках. По умолчанию таким редактором является MonoDevelop, однако вы можете указать в настройках любой другой текстовый редактор по своему усмотрению.

Unity автоматически обновит проект в MonoDevelop, включающий сценарии из вашего проекта, и скомпилирует код после возврата в Unity. Это означает, что вам достаточно просто отредактировать сценарий, сохранить изменения и вернуться в редактор Unity.

MonoDevelop обладает несколькими полезными особенностями, помогающими экономить время.

Дополнение кода

Если в MonoDevelop нажать комбинацию Ctrl-Space (на PC и Mac), то откроется всплывающее окно со списком возможных вариантов дополнения введенного кода; например, если вы уже напечатали половину имени класса, MonoDevelop

предложит дополнить его. Нажимайте клавиши со стрелками вверх и вниз, чтобы выбрать нужный вариант из списка, а затем нажмите клавишу Enter, чтобы принять этот вариант.

Рефакторинг

Если нажать комбинацию Alt-Enter (Option-Enter в Mac), MonoDevelop предложит автоматически выполнить некоторые задачи, связанные с правкой исходного кода. В число этих задач входят: добавление и удаление скобок вокруг конструкций `if`, автоматическое заполнение меток `case` в конструкциях `switch` или разбиение одной инструкции объявления переменной с присваиванием на две строки.

Сборка

Unity автоматически скомпилирует ваш код после возврата в редактор. Однако если нажать комбинацию Command-B (F7 на PC), весь код будет скомпилирован в MonoDevelop. Файлы, получившиеся при этом, не будут использоваться в игре, но так вы сможете убедиться в отсутствии ошибок компиляции, прежде чем вернуться в Unity.

Игровые объекты, компоненты и сценарии

Сцены в Unity состоят из *игровых объектов*. Сами по себе эти объекты невидимы и не обладают ничем, кроме имени. Их поведение определяется *компонентами*.

Компоненты — это строительные блоки игры, и все, что вы видите в инспекторе, это и есть компоненты. Разные компоненты отвечают за разные аспекты; например, визуализаторы мешей (Mesh Renderers) отображают трехмерные меши (сетки), а источники звука (Audio Sources) воспроизводят звуки. Сценарии, которые вы пишете, тоже являются компонентами.

Чтобы создать сценарий, нужно:

1. *Создать ресурс сценария*. Открыть меню Assets (Ресурсы) и выбрать пункт Create ▸ Script ▸ C# Script (Создать ▸ Сценарий ▸ Сценарий на C#).
2. *Присвоить имя ресурсу сценария*. В папке, выбранной в обозревателе проекта, появится новый сценарий с указанным вами именем.
3. *Дважды щелкнуть на ресурсе сценария*. Сценарий откроется в редакторе сценариев, роль которого по умолчанию играет MonoDevelop. Большинство ваших сценариев будут выглядеть примерно так:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class AlienSpaceship : MonoBehaviour { ❶

    // Используйте этот метод для инициализации
```

```
    void Start () { ❷

    }

    // Метод Update вызывается один раз перед отображением каждого кадра
    void Update () { ❸

    }
}
```

❶ Имя класса, в данном случае `AlienSpaceship`, должно совпадать с именем файла ресурса.

❷ Функция `Start` вызывается перед первым вызовом функции `Update`. Сюда вы можете поместить код, инициализирующий переменные, загружающий хранимые предпочтения или выполняющий настройку других сценариев и игровых объектов.

❸ Функция `Update` вызывается перед отображением каждого кадра. Сюда можно включить код, отвечающий на ввод пользователя, запускающий другие сценарии или перемещающий объекты, то есть управляющий происходящими событиями.

Вы можете быть знакомы с таким понятием, как *конструктор из других сред разработки*. В Unity вам не придется писать свои конструкторы для классов, наследующих `MonoBehaviour`, потому что конструирование объектов осуществляется самим движком Unity, и не всегда в тот момент, который вы считаете наиболее подходящим для этого.

Ресурсы сценариев в Unity фактически ничего не делают сами по себе — их код не запускается, пока они не будут подключены к игровому объекту (рис. 3.1). Существует два основных способа подключения сценария к игровому объекту:

1. *Перетащить мышью ресурс сценария на игровой объект*. Для этого можно использовать панель Inspector (Инспектор) или Hierarchy (Иерархия).
2. *С помощью меню* Component *(Компонент)*. Найти все сценарии, имеющиеся в проекте, можно в меню Component ▸ Scripts (Компонент ▸ Сценарии).

Поскольку сценарии, подключенные к игровым объектам, редактор Unity отображает в виде компонентов, у вас есть возможность определять в сценариях свойства, доступные для редактирования в инспекторе. Для этого достаточно определить в своем сценарии общедоступные переменные. Все переменные, объявленные общедоступными, будут отображаться в инспекторе; переменные также можно объявлять приватными.

```
public class AlienSpaceship : MonoBehaviour {
    public string shipName;

    // В инспекторе появится поле "Ship Name",
    // доступное для редактирования
}
```

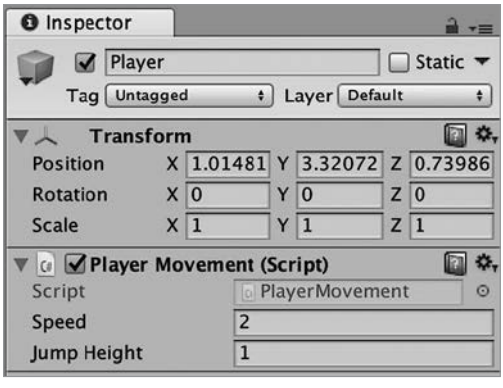

Рис. 3.1. В инспекторе с информацией об игровом объекте можно видеть, что к этому объекту подключен компонент со сценарием «PlayerMovement»

Инспектор

После подключения сценария к игровому объекту он появится в инспекторе, если выбрать этот объект. Unity автоматически отобразит все общедоступные переменные в порядке их объявления в коде.

Также в инспекторе появятся приватные переменные (объявленные со спецификатором `private`), имеющие атрибут `[SerializeField]`. Это удобно, когда требуется, чтобы поле было доступно в инспекторе, но недоступно другим сценариям.

Отображая имена переменных в инспекторе, редактор Unity заменяет первую букву в имени буквой верхнего регистра, а перед каждой последующей буквой верхнего регистра вставляет пробел. Например, переменная `shipName` будет отображаться в редакторе как «Ship Name».

Компоненты

Сценарии имеют доступ к разным компонентам, присутствующим в игровом объекте. Для этого используется метод `GetComponent`.

```
// получить компонент Animator этого объекта, если имеется
var animator = GetComponent<Animator>();
```

Также можно вызывать метод `GetComponent` других игровых объектов, чтобы получить компоненты, подключенные к ним.

Аналогично можно получать компоненты, подключенные к родительским или дочерним объектам, используя методы `GetComponentInParent` и `GetComponent InChildren`.

Важные методы

Класс `MonoBehaviours` имеет несколько методов, особенно важных для Unity. Эти методы вызываются в разные моменты в течение жизненного цикла компонента и позволяют выполнить нужные действия в нужные моменты. В этом разделе перечисляются методы в порядке их вызова.

Awake и OnEnable

Метод `Awake` вызывается сразу после создания экземпляра объекта в сцене, и это первая возможность выполнить код в своем сценарии. `Awake` вызывается только один раз на протяжении всего времени жизни экземпляра.

Метод `OnEnable`, напротив, вызывается всякий раз, когда экземпляр включается.

Start

Метод `Start` вызывается непосредственно перед первым вызовом метода `Update` объекта.

Start и Awake

Вы можете задаться вопросом, почему есть две возможности для настройки объекта: `Awake` и `Start`. Не означает ли это в конечном итоге, что для этой цели можно выбрать любой из методов наугад?

В действительности для такого разделения есть веская причина. Когда запускается новая сцена, вызываются методы `Awake` и `Start` всех присутствующих в ней объектов. Однако крайне важно, чтобы Unity удостоверился, что перед вызовом любого из методов `Start` завершатся вызовы методов `Awake` всех объектов.

Это означает, что любые операции, выполняемые в методах `Awake`, гарантированно будут завершены к моменту вызова метода `Start` любого объекта. Это удобно, например, когда в объекте A имеется поле, используемое объектом B:

```
// В файле с именем ObjectA.cs
class ObjectA : MonoBehaviour {

    // Переменная, используемая другим сценарием
    public Animator animator;

    void Awake() {
        animator = GetComponent<Animator>();
    }
}

// В файле с именем ObjectB.cs
class ObjectB : MonoBehaviour {
```

```
    // Подключение к сценарию ObjectA
    public ObjectA someObject;

    void Awake() {
        // Проверить, инициализировал ли someObject свою
        // переменную 'animator'
        bool hasAnimator = someObject.animator == null;

        // Выведет 'true' или 'false' в зависимости от очередности
        // создания объектов
        Debug.Log("Awake: " + hasAnimator.ToString());
    }

    void Start() {
        // Проверить, инициализировал ли someObject свою
        // переменную 'animator'
        bool hasAnimator = someObject.animator == null;

        // *Всегда* будет выводить 'true'
        Debug.Log("Start: " + hasAnimator.ToString());
    }
}
```

В этом примере сценарий `ObjectA` находится в объекте, который также имеет подключенный компонент `Animator`. (В данном примере этот компонент ничего не делает, и вместо него с таким же успехом можно было бы использовать любой другой компонент.) Сценарий `ObjectB` настраивается так, что его переменная `someObject` ссылается на объект, содержащий сценарий `ObjectA`.

Когда сцена запускается, сценарий `ObjectB` дважды выводит сообщение — первый раз в методе `Awake` и второй раз в методе `Start`. В обоих случаях он сравнивает поле `animator` своей переменной `someObject` со значением `null` и по результатам сравнения выводит «true» или «false».

Если вы запустите этот пример, первое сообщение, которое выведет метод `Awake` в сценарии `ObjectB`, будет «true» или «false», в зависимости от очередности вызова методов `Awake` этих двух сценариев. (Если вручную явно не определить очередность вызова, заранее невозможно предугадать, какой сценарий запустится первым.)

Однако вторым сообщением, которое выводит метод `Start` в `ObjectB`, *гарантированно* будет «true». Это объясняется тем, что в момент запуска сцены все объекты, присутствующие в ней, сначала выполнят свои методы `Awake`, и только после этого будет вызван первый метод `Start`.

Update и LateUpdate

Метод `Update` вызывается для каждого кадра, пока активны компонент и объект, к которому подключен сценарий.

Методы `Update` должны выполнять свою работу максимально быстро, потому что вызываются для каждого кадра. Если в методе `Update` организовать выполнение продолжительных операций, это замедлит работу всей игры. Для выполнения продолжительных операций следует использовать сопрограммы (coroutine), которые описывают в следующем разделе.

Unity вызывает методы `Update` во всех сценариях, имеющих его. Вслед за этим вызываются методы `LateUpdate` во всех сценариях, имеющих его. Методы `Update` и `LateUpdate` связаны похожими отношениями, что и методы `Awake` и `Start`: никакой из методов `LateUpdate` не будет вызван, пока не выполнятся все методы `Update`.

Это может пригодиться для выполнения действий, зависящих от результатов операций, которые выполняются в методах `Update` некоторых других объектов. Заранее нельзя предсказать, какой из методов `Update` выполнится первым; но код в методе `LateUpdate` гарантированно будет выполнен после завершения методов `Update` всех объектов.

Кроме метода `Update` можно также использовать метод `FixedUpdate`. В отличие от `Update`, который вызывается один раз для каждого кадра, метод `FixedUpdate` вызывается фиксированное число раз в секунду. Это может пригодиться для реализации физических явлений, когда некоторые операции должны выполняться через регулярные интервалы времени.

Сопрограммы

Большинство функций выполняют свою работу и сразу возвращают управление. Но иногда желательно продолжать выполнять некоторые действия в течение длительного времени. Например, если нужно переместить объект из одной точки в другую, такое его движение должно продолжаться на протяжении нескольких кадров.

Сопрограмма (coroutine) — это функция, которая выполняется на протяжении нескольких кадров. Чтобы создать сопрограмму, прежде нужно объявить метод, возвращающий значение типа `IEnumerator`:

```
IEnumerator MoveObject() {
}
```

и использовать в нем конструкцию `yield return` для временной приостановки, чтобы дать возможность выполниться остальной игре. Например, вот как можно организовать перемещение объекта вперед на определенное расстояние в каждом кадре[1]:

```
IEnumerator MoveObject() {
    // Бесконечный цикл
    while (true) {
```

[1] На самом деле это не лучшее решение по причинам, объясняемым в разделе «Время в сценариях» в конце этой главы, но оно наглядно демонстрирует саму идею.

```
            transform.Translate(0,1,0); // перемещать на 1 единицу по оси Y
                                        // в каждом кадре

            yield return null; // ждать наступления следующего кадра

        }
    }
```

Используя бесконечный цикл (как `while (true)` в предыдущем примере), вы *должны* вызывать конструкцию `yield return` в нем. В противном случае такой цикл не даст возможности выполниться никакому другому коду в вашей игре. Так как код игры выполняется внутри Unity, вы рискуете «подвесить» Unity, если выполнение дойдет до такого цикла. Если это случится, вам придется принудительно завершить Unity, что может привести к потере несохраненных данных.

Конструкция `yield return` временно приостанавливает выполнение функции. Unity автоматически возобновит ее работу позднее, но *когда* это произойдет, зависит от значения, возвращаемого `yield return`.

Например:

```
yield return null
```

приостановит выполнение до следующего кадра;

```
yield return new WaitForSeconds(3)
```

приостановит выполнение на три секунды;

```
yield return new WaitUntil(() => this.someVariable == true)
```

приостановит выполнение до момента, когда переменная `someVariable` получит значение `true`; вместо констант `true` и `false` также можно использовать любые выражения, возвращающие `true` или `false`.

Остановить сопрограмму можно конструкцией `yield break`:

```
// немедленно остановит сопрограмму
yield break;
```

Кроме того, сопрограммы автоматически останавливаются, когда выполнение достигает конца метода.

Создав функцию сопрограммы, ее можно запустить. Но для этого функция сопрограммы должна вызываться не напрямую, а посредством функции `StartCoroutine`:

```
StartCoroutine(MoveObject());
```

Этот вызов запустит сопрограмму, которая продолжит выполняться, пока не достигнет конструкции `yield break` или конца метода.

Кроме обычных значений в конструкции `yield return` можно также указать другую сопрограмму. В этом случае вызывающая сопрограмма будет ждать, пока не завершится вызванная ею сопрограмма.

Сопрограмму можно остановить извне. Для этого нужно сохранить ссылку на значение, возвращаемое методом `StartCoroutine`, и в нужный момент передать его в вызов метода `StopCoroutine`:

```
Coroutine myCoroutine = StartCoroutine(MyCoroutine());

// ... позднее ...

StopCoroutine(myCoroutine);
```

Создание и уничтожение объектов

Существует два способа создания объектов в течение игры. Первый — создать пустой игровой объект и подключить к нему компоненты программным способом; второй заключается в создании копии имеющегося объекта (называется *созданием экземпляра*). Второй метод пользуется большей популярностью, потому что все необходимое можно сделать одной строкой кода. Мы рассмотрим этот способ первым.

Объекты, создаваемые в режиме проигрывания, исчезают после остановки игры. Если вам нужно, чтобы они сохранялись, выполните следующие действия.

1. Выберите объекты для сохранения.
2. Скопируйте их в буфер обмена, нажав Command-C (Ctrl-C на PC) или открыв меню Edit (Правка) и выбрав пункт Copy (Копировать).
3. Покиньте режим проигрывания. Объекты исчезнут из сцены.
4. Вставьте их, нажав комбинацию Command-V (Ctrl-V на PC) или открыв меню Edit (Правка) и выбрав пункт Paste (Вставить). Объекты вновь появятся; теперь с ними можно работать в режиме редактирования как обычно.

Копирование

Под созданием экземпляра объекта в Unity подразумевается копирование существующего объекта со всеми его компонентами, дочерними объектами и *их* компонентами. Это особенно удобно, когда копируемый объект является шаблонным (prefab). Шаблонные объекты — это объекты, подготовленные заранее и хранящиеся в виде ресурсов. Это означает, что можно создать один шаблон объекта, а затем создать из него множество копий в разных сценах.

Создать копию объекта можно с помощью метода `Instantiate`:

```
public GameObject myPrefab;

void Start() {
    // Создать новую копию объекта myPrefab
    // и поместить его в ту же позицию, где находится объект this
```

```
    var newObject = (GameObject)Instantiate(myPrefab);

    newObject.transform.position = this.transform.position;
}
```

Метод `Instantiate` возвращает значение типа `Object`, не `GameObject`. Чтобы использовать его как экземпляр `GameObject`, требуется выполнить приведение типа.

Создание объекта с нуля

Другой способ создания объектов — конструирование их вручную в программном коде. Для этого прежде всего нужно с помощью ключевого слова `new` сконструировать новый экземпляр `GameObject`, а затем добавить в него компоненты вызовом метода `AddComponent`.

```
// Создать новый игровой объект; он появится в иерархии
// под именем "My New GameObject"
var newObject = new GameObject("My New GameObject");

// Добавить в него новый компонент SpriteRenderer
var renderer = newObject.AddComponent<SpriteRenderer>();

// Определить спрайт, который должен отображать новый SpriteRenderer
renderer.sprite = myAwesomeSprite;
```

Метод `AddComponent` принимает параметр типа компонента, который вы хотите добавить. Вы можете передать в этом параметре любой класс, наследующий класс `Component`, и его экземпляр будет добавлен.

Уничтожение объектов

Метод `Destroy` удаляет объект из сцены. Обратите внимание — мы не говорим *игровой объект*, мы говорим *объект*! Метод `Destroy` удаляет и игровые объекты, и компоненты.

Чтобы удалить объект из сцены, вызовите его методом `Destroy`:

```
// Удалить игровой объект, к которому подключен данный сценарий
Destroy(this.gameObject);
```

Метод `Destroy` работает и с компонентами, и с игровыми объектами.

Если в вызов `Destroy` передать ссылку `this`, которая ссылается на *компонент текущего сценария*, он удалит не игровой объект, а компонент с данным сценарием, подключенный к игровому объекту. Игровой объект останется в сцене, но уже без данного сценария.

Атрибуты

Атрибут — это фрагмент информации, который можно присоединить к классу, переменной или методу. Unity определяет несколько полезных атрибутов, влияющих на поведение классов или определяющих особенности представления в редакторе.

RequireComponent

Атрибут `RequireComponent`, когда он присоединен к классу, сообщает Unity, что для подключения сценария требуется наличие другого компонента. Это может пригодиться, когда сценарий имеет смысл, только когда к объекту подключен компонент указанного типа. Например, если ваш сценарий изменяет настройки компонента `Animator`, имеет смысл потребовать наличия компонента `Animator` перед подключением сценария.

Тип требуемого компонента передается в параметре, например:

```
[RequireComponent(typeof(Animator))]
class ClassThatRequiresAnAnimator : MonoBehaviour {
    // этот класс требует предварительного подключения компонента
    // Animator к игровому объекту GameObject
}
```

Если попытаться подключить к игровому объекту сценарий, который требует наличия некоторого компонента, отсутствующего в данный момент, Unity автоматически добавит этот компонент.

Header и Space

Атрибут `Header`, когда присоединяется к полю, заставляет Unity выводить указанную надпись над полем в инспекторе. Атрибут `Space` действует аналогично, но добавляет пустое пространство. Оба атрибута удобно использовать для визуальной организации содержимого в инспекторе.

Например, на рис. 3.2 показано, как выглядит в инспекторе следующий код:

```
public class Spaceship : MonoBehaviour {

    [Header("Spaceship Info")]

    public string name;
    public Color color;
    [Space]

    public int missileCount;
}
```

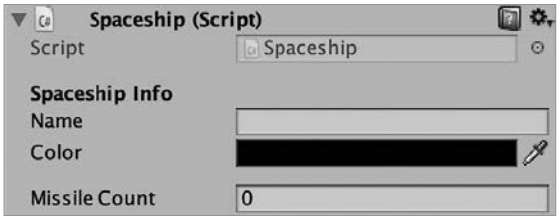

Рис. 3.2. Надписи и пустые пространства в инспекторе

SerializeField и HideInInspector

Обычно в инспекторе отображаются только общедоступные поля (со спецификатором `public`). Однако общедоступные переменные доступны также другим объектам, то есть объект теряет абсолютный контроль над своими данными. Но если объявить переменную приватной (`private`), Unity не будет отображать ее в инспекторе.

Чтобы обойти это ограничение, добавьте атрибут `SerializeField` перед приватными переменными, которые вы хотите сделать доступными в инспекторе.

Если вам требуется *обратное* (то есть скрыть общедоступную переменную в инспекторе), добавьте атрибут `HideInInspec`:

```
class Monster : MonoBehaviour {

    // Появится в инспекторе, потому что общедоступная
    // Доступна другим сценариям
    public int hitPoints;

    // Не появится в инспекторе, потому что приватная
    // Недоступна другим сценариям
    private bool isAlive;

    // Появится в инспекторе благодаря атрибуту SerializeField
    // Недоступна другим сценариям
    [SerializeField]
    private int magicPoints;

    // Не появится в инспекторе благодаря атрибуту HideInInspector
    // Доступна другим сценариям
    [HideInInspector]
    public bool isHostileToPlayer;
}
```

ExecuteInEditMode

По умолчанию код сценариев выполняется только в режиме проигрывания, то есть код в вашем методе `Update` выполнится только после запуска игры.

Однако иногда бывает удобно иметь код, выполняющийся все время. Если это ваш случай, добавьте к своему классу атрибут `ExecuteInEditMode`.

Жизненный цикл компонента в режиме редактирования протекает иначе, чем в режиме проигрывания. В режиме редактирования Unity перерисовывает себя только тогда, когда это действительно необходимо, обычно в ответ на ввод пользователя, такой как щелчки мышью. Это означает, что метод `Update` будет выполняться лишь изредка. Кроме того, сопрограммы в этом режиме действуют не так, как ожидается.

Помимо этого, вы не сможете вызвать `Destroy` в режиме редактирования, потому что Unity откладывает фактическое удаление до следующего кадра. Вместо него в режиме редактирования следует вызывать метод `DestroyImmediate`, который удаляет объект немедленно.

Например, вот сценарий, который удерживает объект всегда повернутым лицом к цели, даже в режиме редактирования:

```
[ExecuteInEditMode]
class LookAtTarget : MonoBehaviour {

    public Transform target;

    void Update() {
        // Не продолжать, если нет цели
        if (target != null) {
            return;
        }

        // Повернуть лицом к цели
        transform.LookAt(target);
    }

}
```

Если подключить этот сценарий к объекту и записать в его переменную `target` ссылку на другой объект, первый объект будет поворачиваться лицом к целевому объекту в обоих режимах, как редактирования, так и проигрывания.

Время в сценариях

Для получения информации о текущем времени в играх используется класс `Time`. В этом классе доступно несколько переменных (и мы настоятельно рекомендуем ознакомиться с его описанием в документации[1]!), но наиболее важной и часто используемой является `deltaTime`.

`Time.deltaTime` измеряет интервал времени, потребовавшийся на рисование последнего кадра. Важно понимать, что это время может сильно меняться. Эту переменную можно использовать для выполнения действий, которые должны продолжаться на протяжении нескольких кадров, но в течение определенного отрезка времени.

Выше, в разделе «Сопрограммы» в этой главе, приводился пример перемещения объекта на одну единицу расстояния в каждом кадре. Это не лучшее решение, по-

[1] *https://docs.unity3d.com/ru/current/Manual/TimeFrameManagement.html.*

тому что частота кадров в секунду может существенно изменяться. Например, если камера направлена на относительно простой участок сцены, частота кадров может быть очень высокой, а при отображении визуально более сложных сцен частота может падать до крайне низких значений.

Так как невозможно гарантировать постоянную частоту кадров, лучшее, что можно сделать, — включить в расчеты переменную `Time.deltaTime`. Этот прием проще объяснить на примере:

```
IEnumerator MoveSmoothly() {
    while (true) {

        // перемещать на 1 единицу в секунду
        var movement = 1.0f * Time.deltaTime;

        transform.Translate(0, movement, 0);

        yield return null;

    }
}
```

Журналирование в консоль

Как было показано в разделе «Awake и OnEnable» выше в этой главе, иногда удобно иметь возможность выводить информацию в консоль для нужд отладки или вывода сообщений о возникших проблемах.

Для этого можно использовать функцию `Debug.Log`. Поддерживается три уровня журналирования: информация, предупреждение и ошибка. Между ними нет никаких различий кроме того, что функции для вывода предупреждений и ошибок более заметны в коде.

Кроме `Debug.Log` можно также использовать функцию `Debug.LogFormat`, которая позволяет внедрять значения в строку, посылаемую в консоль:

```
Debug.Log("This is an info message!");
Debug.LogWarning("This is a warning message!");
Debug.LogError("This is a warning message!");

Debug.LogFormat("This is an info message! 1 + 1 = {0}", 1+1);
```

В заключение

Программирование в Unity — крайне необходимый навык, и если вы умеете писать код на языке C# и владеете инструментами, используемыми для этого, вам будет проще и интереснее создавать свои игры.

ЧАСТЬ II
Создание двумерной игры «Колодец с сокровищами»

Теперь, закончив несколько отвлеченное изучение Unity, мы воспользуемся полученными знаниями и применим их на практике. В этой и в следующей части мы создадим две законченные игры с самого начала.

В следующих нескольких главах мы создадим игру с боковой прокруткой, которая называется *Колодец с сокровищами*. В этой игре используется двумерная графика и многие возможности физического движка в Unity, а также довольно интенсивно используется подсистема пользовательского интерфейса. Это будет весело!

4 Начало создания игры

Знание интерфейса Unity — это одно, а вот создание полноценной игры — совсем другое. В этой части книги вы сможете применить все, что узнали в первой части, для создания двумерной игры. К концу этой части вы закончите создание игры *Колодец с сокровищами* (взглянув на рис. 4.1, можно увидеть, как эта игра выглядит).

Рис. 4.1. Законченная игра

Дизайн игры

Игровой процесс *Колодца с сокровищами* простейший. Игрок управляет спуском гномика в колодец на веревочке. На дне колодца находятся сокровища. Сложность в том, что колодец наполнен ловушками, которые убивают гномика, если он касается их.

Для начала нарисуем грубый эскиз, поясняющий, как должна выглядеть игра. Мы использовали для этого OmniGraffle, отличное приложение для рисования диаграмм, но в данном случае выбор инструмента не имеет большого значения — с неменьшим, а иногда и с большим успехом можно использовать карандаш и бумагу. Цель — как можно быстрее получить примерное представление о том, как будет выглядеть игра. Эскиз игры *Колодец с сокровищами* можно увидеть на рис. 4.2.

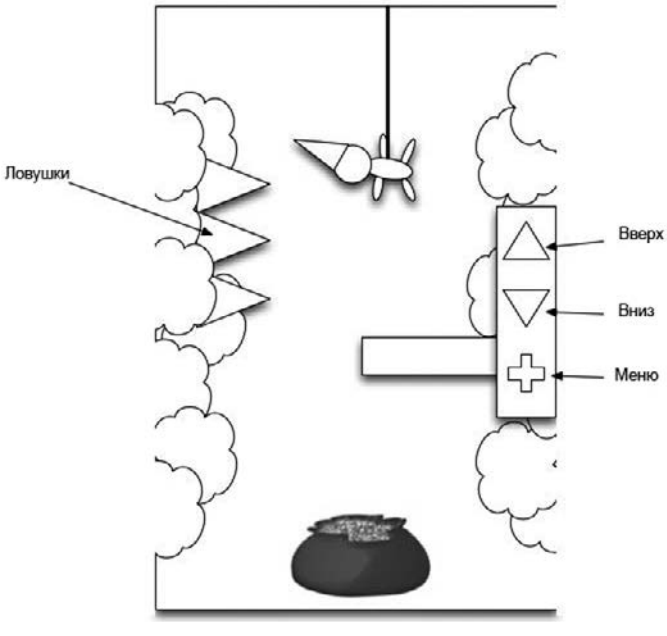

Рис. 4.2. Концептуальный эскиз игры

Определившись с основной идеей игры, приступим к созданию общей архитектуры. Сначала определим перечень видимых объектов и взаимосвязи между ними. Одновременно подумаем, как будут работать «невидимые» компоненты: как будет приниматься ввод и как внутренние средства управления будут взаимодействовать друг с другом.

Наконец, нам также нужно позаботиться о визуальных элементах игры. Мы обратились к нашему другу-художнику и попросили его нарисовать гномиков, спускающихся в колодец, который наполнен опасными ловушками. Благодаря этому мы получили представление о том, как будет выглядеть главный персонаж, и смогли

определить общий тон игры: нехитрая, немного жестокая игра мультяшного вида с жадными гномиками в главной роли. Окончательный эскиз можно увидеть на рис. 4.3.

Если у вас нет знакомого художника, попробуйте нарисовать сами! Даже если вы считаете, что не умеете рисовать, *любые* мысли о том, как будет выглядеть игра, намного лучше, чем их отсутствие.

Рис. 4.3. Концептуальное изображение главного персонажа

После подготовки предварительного дизайна можно приступать к работе над реализацией: как будет перемещаться гномик в игре, как настроить интерфейс для работы и как должны быть связаны между собой игровые объекты.

Для управления положением гномика в колодце игрок может использовать три кнопки: одна увеличивает длину веревки, другая уменьшает, а третья вызывает игровое меню. Если нажать и удерживать кнопку, удлиняющую веревку, гномик будет спускаться в колодец. Чтобы избежать ловушек, встречающихся на пути, игрок может наклонять свое устройство влево или вправо. В результате этого гномик будет смещаться влево или вправо.

Игровой процесс главным образом является результатом имитации физики двумерного мира. Гномик — это «кукла» — коллекция фрагментов, связанных сочленениями, каждый из которых моделируется независимо, как твердое тело. Это означает, что когда гномик будет связан с верхним краем колодца посредством объекта, представляющего веревку, он будет качаться в соответствии с законами физики.

Веревка сделана аналогично: это коллекция твердых тел, соединенных друг с другом сочленениями. Первое звено этой «цепочки» соединено с верхним краем колодца и со вторым звеном через вращающиеся сочленения. Второе звено соединено с третьим, третье с четвертым и так далее до последнего звена, которое соединено с лодыжкой гномика. Чтобы удлинить веревку, в верхнюю ее часть добавляются новые звенья, а чтобы укоротить — звенья сверху удаляются.

Остальное в игровом процессе обрабатывается простым механизмом определения столкновений:

- Если какая-то часть гномика касается объекта ловушки, гномик погибает и создается новый гномик. Дополнительно создается спрайт духа гномика, возносящийся вверх по колодцу.
- Если спрайты гномика касаются сокровища, его изображение изменяется, чтобы показать, что он держит сокровище.
- Если какая-то часть гномика касается верхнего края колодца (невидимый объект) и он удерживает сокровище, игрок объявляется победителем.

Кроме гномика, ловушек и сокровища в игре имеется камера со сценарием, который обновляет позицию камеры в соответствии с вертикальной позицией гномика, но не позволяет ей подняться выше верхнего края колодца или опуститься ниже дна.

Ниже перечислены шаги, которые нужно сделать, чтобы создать игру (не волнуйтесь — мы подробно разберем каждый шаг в свое время):

1. Сначала создадим гномика, используя временные схематические изображения. Мы подготовим куклу и подключим спрайты.
2. Далее мы создадим веревку. При этом мы напишем первый значительный фрагмент кода, потому что веревка будет генерироваться во время выполнения, и нам потребуется организовать возможность ее удлинения и укорочения.
3. Когда веревка будет готова, мы перейдем к созданию системы ввода. Эта система будет принимать информацию о наклоне устройства и делать ее доступной для других частей игры (в частности, для гномика). Одновременно мы займемся пользовательским интерфейсом игры и создадим кнопки для удлинения и укорочения веревки.
4. Закончив с гномиком, веревкой и системой ввода, мы займемся созданием самой игры: реализуем ловушки и сокровище и попробуем поиграть.
5. После этого останется только усовершенствовать игру: спрайты гномика заменить более сложными изображениями, а также добавить эффект частиц и звуки.

К концу этой главы функциональная сторона игры будет закончена, но художественное оформление на этом не завершится. Мы займемся им позже, в главе 7. На рис. 4.4 можно увидеть, как будет выглядеть игра.

В процессе работы над этим проектом вы добавите в игровые объекты большое количество компонентов и будете заниматься настройкой значений их свойств. Мы будем вам предлагать изменить настройки компонентов, но далеко не всех, поэтому не бойтесь самостоятельно экспериментировать с любыми другими доступными вам настройками или можете оставить все настройки со значениями по умолчанию.

А теперь приступим!

Рис. 4.4. Так будет выглядеть первая черновая версия игры

Создание проекта и импортирование ресурсов

Начнем с создания проекта в Unity и изменения некоторых настроек. Мы также импортируем ряд ресурсов, которые понадобятся нам на первых этапах; по мере разработки мы будем импортировать дополнительные ресурсы.

1. *Создание проекта.* Выберите пункт меню File ▸ New Project (Файл ▸ Новый проект) и создайте новый проект с именем GnomesWell. В диалоге New Project (Новый проект) (рис. 4.5) выберите флажок 2D, а не 3D, и убедитесь, что никакие пакеты ресурсов не выбраны для импортирования. Вы просто должны создать пустой проект.

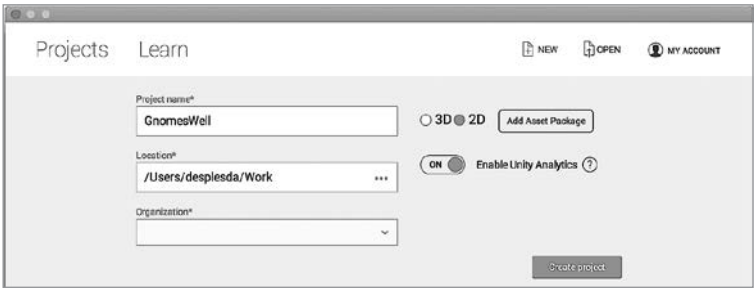

Рис. 4.5. Создание проекта

2. *Загрузка ресурсов.* Загрузите с сайта *https://www.secretlab.com.au/books/unity* пакет с изображениями, звуками и другими ресурсами, которые мы подготовили для этого проекта, и распакуйте его в подходящую папку на своем компьютере. Вы будете импортировать эти ресурсы в проект по мере продвижения.
3. *Сохранение сцены в файле Main.scene.* Сейчас вы уже можете сохранить сцену нажатием комбинации клавиш Command-S (или Ctrl-S на PC). В первый раз вам будет предложено ввести имя сцены и указать, где ее сохранить, — сохраните ее в папку *Assets*.
4. *Создание папок для проекта.* Для поддержания порядка рекомендуется создавать разные папки для разных категорий ресурсов. Unity прекрасно будет работать, даже если все ресурсы хранить в одной папке, но вам самим сложнее будет находить нужные файлы. Создайте следующие папки, щелкая правой кнопкой мыши в папке *Assets* в обозревателе проекта и выбирая в контекстном меню пункт Create ▸ Folder (Создать ▸ Папку):

Scripts

В этой папке будет храниться код на C#. (По умолчанию Unity сохраняет все файлы с программным кодом в корневой папке *Assets*; вам придется вручную переносить эти файлы в папку *Scripts*.)

Sounds

В этой папке будут храниться музыка и звуки.

Sprites

В этой папке будут храниться все изображения спрайтов. Их будет очень много, поэтому для них мы создадим несколько подпапок.

Gnome

В этой папке будут храниться шаблонные объекты (prefabs), необходимые для создания гномика, а также дополнительные объекты, связанные с гномиком, такие как веревка, эффекты частиц и дух.

Level

В этой папке будут храниться шаблонные объекты для самого уровня, включая фоновые изображения, стены, декоративные объекты и ловушки.

App Resources

В этой папке будут храниться ресурсы, используемые приложением как таковым: пиктограммы и начальный экран.

В итоге содержимое папки *Assets* должно выглядеть так, как показано на рис. 4.6.

Рис. 4.6. Папка Assets с созданными в ней папками

5. *Импортирование ресурсов прототипа гномика*. Прототип гномика — это грубая схематическая версия гномика, которую мы построим на первых порах. Позднее мы заменим ее усовершенствованными спрайтами.

Найдите папку *Prototype Gnome* в наборе загруженных ресурсов и перетащите ее в папку *Sprites* в Unity (рис. 4.7).

Теперь мы готовы начать конструирование гномика.

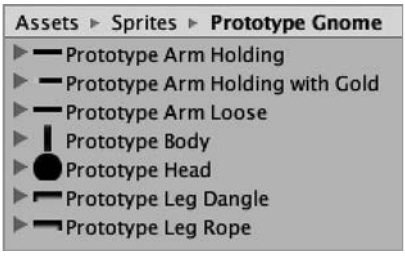

Рис. 4.7. Спрайты для создания прототипа гномика

Создание гномика

Поскольку гномик будет состоять из нескольких объектов, движущихся независимо, сначала нужно создать объект, который послужит контейнером для каждой части. Этому объекту также надо присвоить тег Player, потому что механизму определения столкновений, который используется для определения касаний гномиком ловушек, сокровища и выхода из уровня, нужно знать, что объект является специальным объектом `Player`. Чтобы создать гномика, выполните следующие действия.

1. *Создайте объект Prototype Gnome.* Создайте новый пустой игровой объект, открыв меню GameObject (Игровой объект) и выбрав пункт Create Empty (Создать пустой).

   Дайте новому объекту имя «Prototype Gnome», а затем присвойте ему тег `Player`, выбрав Player в раскрывающемся списке Tag (Тег), находящемся в верхней части инспектора.

Координаты X, Y и Z объекта Prototype Gnome в разделе Position компонента `Transform` в панели инспектора должны быть равны нулю. Если это не так, щелкните по пиктограмме с шестеренкой в правом верхнем углу компонента `Transform` и выберите в открывшемся меню пункт Reset Position (Сбросить позицию).

2. *Добавьте спрайты.* Найдите папку *Prototype Gnome*, добавленную ранее, и перетащите из нее на сцену все имеющиеся в ней спрайты, кроме *Prototype Arm Holding with Gold*, который мы пока не будем использовать.

Вы должны перетащить спрайты по одному — если выделить несколько спрайтов и попытаться перетащить их все сразу, Unity решит, что вы пытаетесь перетащить последовательность изображений, и создаст анимацию.

   Когда вы закончите, в сцене должно появиться шесть новых спрайтов: *Prototype Arm Holding*, *Prototype Arm Loose*, *Prototype Body*, *Prototype Head*, *Prototype Leg Dangle* и *Prototype Leg Rope*.

3. *Установите спрайты как дочерние объекты для объекта Prototype Gnome.* В панели иерархии выберите все только что добавленные спрайты и перетащите их на пустой объект *Prototype Gnome*. В результате иерархия должна выглядеть так, как показано на рис. 4.8.

4. *Позиционируйте спрайты.* После добавления спрайтов их нужно поместить в соответствующие позиции — руки, ноги и голову нужно присоединить к телу. В представлении сцены выберите инструмент Move (Перемещение) — щелкните на кнопке инструмента в панели или нажмите клавишу T.

С помощью инструмента Move (Перемещение) расположите спрайты так, чтобы получилась фигурка как на рис. 4.9.

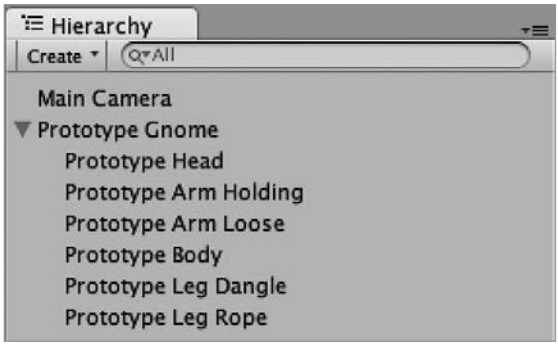

Рис. 4.8. Иерархия объекта Prototype Gnome с дочерними спрайтами

Дополнительно присвойте каждому спрайту тег `Player`, как у родительского объекта. Наконец, убедитесь, что координата Z всех объектов равна нулю. Увидеть эту координату можно в разделе Position компонента `Transform` каждого объекта в панели инспектора.

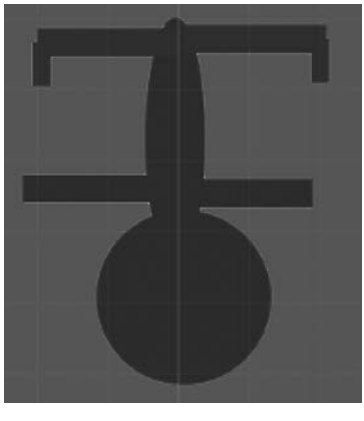

Рис. 4.9. Спрайты прототипа гномика

5. *Добавьте компоненты Rigidbody 2D в части тела.* Выберите все спрайты частей тела и щелкните по кнопке Add Component (Добавить компонент) в инспекторе. Введите `Rigidbody` в поле поиска и выберите в открывшемся списке Rigidbody 2D (как показано на рис. 4.10).

Убедитесь, что добавили компонент «Rigidbody 2D», а не обычный «Rigidbody». Обычный компонент «Rigidbody» имитирует поведение в трехмерном пространстве, а это совсем не то, что нам нужно в этой игре.

Также убедитесь, что добавили компонент «Rigidbody 2D» только в спрайты, — его не надо добавлять в родительский объект *Prototype Gnome*.

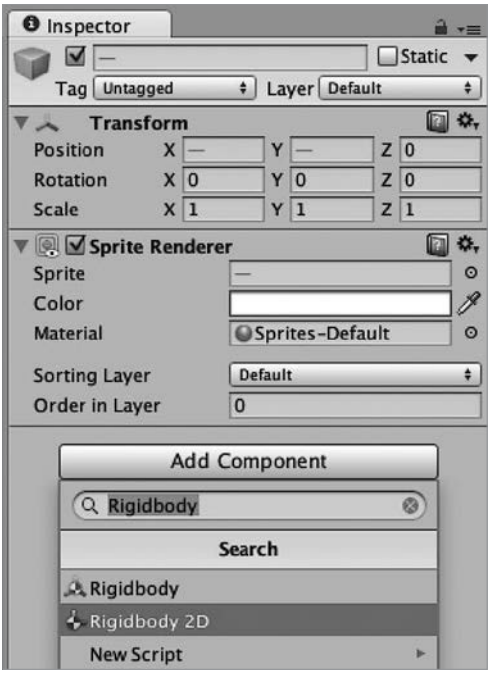

Рис. 4.10. Добавление компонентов Rigidbody 2D в спрайты

6. *Добавьте коллайдеры в части тела*. Коллайдеры определяют физическую форму объектов. Так как разные части тела имеют разную форму, в них нужно добавить коллайдеры разной формы:
   - Выберите спрайты рук и ног и добавьте в них компоненты Box Collider 2D.
   - Выберите спрайт головы и добавьте компонент Circle Collider 2D. Оставьте значение радиуса, установленное по умолчанию.
   - Выберите спрайт тела и добавьте коллайдер Circle Collider 2D. После этого перейдите в панель инспектора и уменьшите радиус коллайдера примерно до половины размера тела.

Теперь части тела гномика готовы к соединению друг с другом. Соединение частей тела будет осуществляться с помощью сочленения `HingeJoint2D`, которое позволяет объектам вращаться вокруг точки сочленения относительно друг друга. Руки, ноги и голова будут соединены с телом. Для настройки сочленений выполните следующие действия:

1. *Выберите все спрайты*, кроме *тела*. Тело не должно иметь своих сочленений — другие части тела будут соединяться с ним посредством *своих* сочленений.
2. *Добавьте компонент HingeJoint2D во все выбранные спрайты*. Для этого щелкните по кнопке Add Component (Добавить компонент) внизу в панели инспектора и выберите компонент Physics 2D ▸ Hinge Joint 2D.

3. *Настройте сочленения.* Пока спрайты остаются выделенными, настройте свойство, которое будет иметь одинаковое значение для всех частей тела: соединение со спрайтом тела *Body*.

   Перетащите *Prototype Body* из панели иерархии в поле Connected Rigid Body (Несущее твердое тело). В результате все объекты окажутся соединенными с телом. В результате настройки сочленений должны выглядеть так, как показано на рис. 4.11.

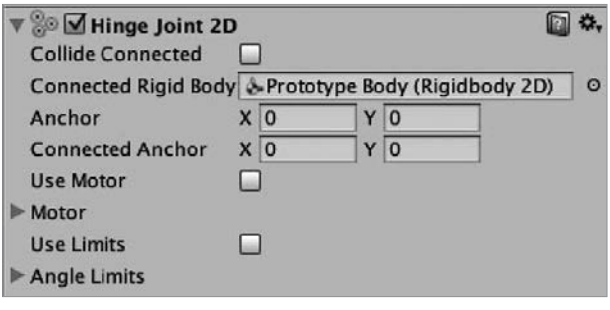

Рис. 4.11. Начальные настройки сочленений

4. *Добавьте ограничения для сочленений.* Нам не нужно, чтобы объекты могли делать полный оборот вокруг сочленений, поэтому добавим ограничители угла поворота. Это предотвратит неестественное поведение, когда, например, нога может сделать полный оборот вокруг сочленения с телом.

   Выберите руки и голову и установите флажок Use Limits (Использовать ограничения). Установите свойство Lower Angle (Нижний угол) в значение −15, а свойство Upper Angle (Верхний угол) — в значение 15.

   Затем выберите ноги и также установите флажок Use Limits (Использовать ограничения). Установите свойство Lower Angle (Нижний угол) в значение −45, а свойство Upper Angle (Верхний угол) — в значение 0.

5. *Обновите опорные точки сочленений.* Мы хотим, чтобы руки вращались относительно плечевых суставов, а ноги — относительно тазобедренных. По умолчанию сочленения обеспечивают вращение относительно центра объекта (рис. 4.12), что выглядит довольно странно.

   Чтобы исправить это, нужно изменить позиции Anchor (Точка привязки) и Connected Anchor (Несущая точка привязки) сочленения. Anchor (Точка привязки) — это точка, вокруг которой вращается тело, владеющее сочленением, а Connected Anchor (Несущая точка привязки) — это точка *на несущем теле*, вокруг которой будет вращаться сочленение. В случае с сочленениями в фигурке гномика нам нужно, чтобы обе позиции — позиции Anchor (Точка привязки) и Connected Anchor (Несущая точка привязки) — совпадали.

   Когда выбирается объект с сочленением, в представлении сцены появляются обе точки: Anchor (Точка привязки) и Connected Anchor (Несущая точка при-

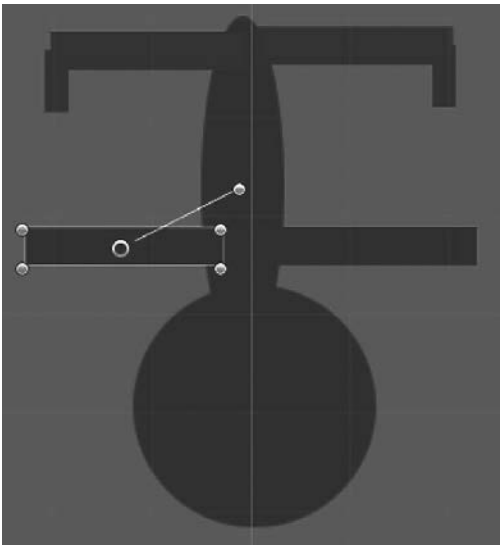

Рис. 4.12. Точки привязки сочленений находятся в неверных позициях

вязки): несущая точка привязки отображается как синяя точка, а точка привязки — как синяя окружность.

Выберите поочередно каждую часть тела с сочленением и переместите обе точки, Anchor (Точка привязки) и Connected Anchor (Несущая точка привязки), в нужную позицию. Например, выберите правую руку и сдвиньте синюю точку, соответствующую несущей точке привязки, в позицию плечевого сустава.

Сдвинуть точку привязки Anchor немного сложнее, потому что по умолчанию она находится в центре, а попытка сдвинуть центр объекта приводит к тому, что Unity сдвигает объект целиком. Чтобы сдвинуть точку привязки Anchor, сначала нужно вручную изменить ее координаты в инспекторе — это повлечет изменение местоположения точки привязки в сцене. Сдвинув точку в сторону от центра, вы сможете перетащить ее мышью в нужную позицию, как уже проделали это с несущей точкой привязки (рис. 4.13.)

Повторите эту процедуру для обеих рук (присоединив их к плечам), обеих ног (присоединив их к тазу) и головы (присоединив ее к основанию шеи).

Теперь добавим сочленение для присоединения объекта веревки. Это будет компонент `SpringJoint2D`, подключенный к правой ноге гномика. Он позволит свободно вращаться вокруг точки сочленения и ограничит расстояние, на которое тело может быть отдалено от веревки. (Веревку мы создадим в следующем разделе.) Пружинное сочленение действует подобно пружинам в реальном мире: оно обладает некоторой упругостью и может немного растягиваться.

В Unity пружинные сочленения управляются двумя основными параметрами: длиной и частотой. Под длиной понимается «предпочтительная» длина пружины,

к которой она стремится вернуться, если ее сжать или растянуть. Под частотой понимается жесткость пружины. Чем меньше частота, тем мягче пружина.

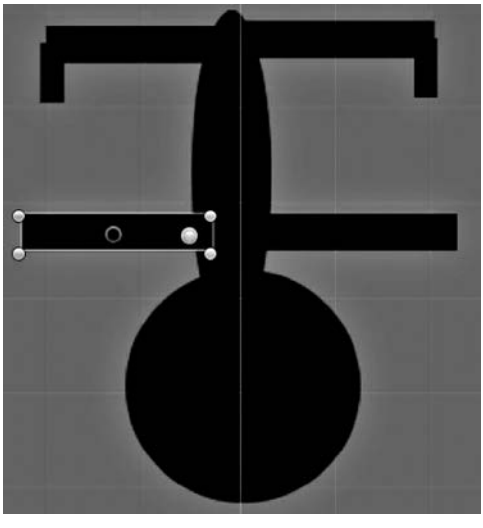

Рис. 4.13. Точки привязки левой руки в правильной позиции; обратите внимание, как точка вписалась в окружность, что указывает на совпадение позиций обеих точек привязки — Connected Anchor и Anchor

Чтобы настроить пружину для использования в веревке, выполните следующие действия.

1. *Добавьте сочленение с веревкой*. Выберите объект Prototype Leg Rope. Это спрайт ноги справа вверху.
2. *Добавьте в него пружинное сочленение*. Добавьте в него компонент `SpringJoint2D`. Перетащите его точку привязки Anchor (синяя окружность) так, чтобы она оказалась ближе к концу ноги. Не трогайте несущую точку привязки Connected Anchor (то есть вы должны перетащить синюю окружность, но не синюю точку). Требуемое положение точек привязки на фигурке гномика показано на рис. 4.14.
3. *Настройте сочленение*. Снимите флажок Auto Configure Distance (Автоматическое определение длины) и установите свойство Distance (Длина) сочленения в значение 0,01, а свойство Frequency (Частота) — в значение 5.
4. *Запустите игру.* Сразу после запуска гномик повиснет в середине экрана.

   Последний шаг — уменьшение изображения гномика, чтобы придать ему правильный размер на фоне других объектов, присутствующих в уровне.
5. *Уменьшите размер гномика*. Выберите родительский объект Gnome и измените коэффициенты масштабирования по осям X и Y, подставив значение 0,5. В результате фигурка гномика уменьшится наполовину.

Гномик готов. Теперь добавим веревку!

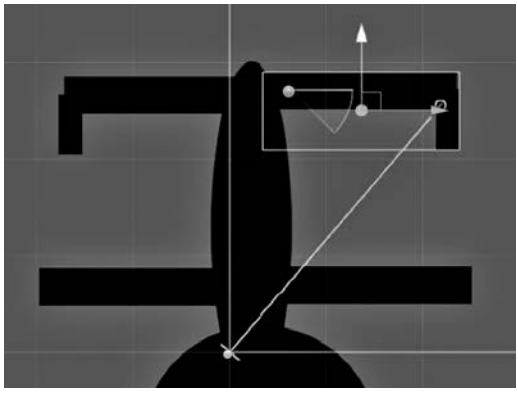

Рис. 4.14. Добавление пружинного сочленения для соединения ноги с веревкой — точка привязки Anchor сочленения находится рядом с пальцем на ноге

Веревка

Веревка — первый элемент игры, для реализации которого потребуется писать программный код. Это работает так: веревка является коллекцией игровых объектов, каждый из которых имеет твердое тело и пружинное сочленение. Каждое пружинное сочленение соединено со следующим объектом Rope Segment, который в свою очередь соединен со следующим и так далее до начала веревки вверху. Самый верхний объект Rope Segment соединен с неподвижным твердым телом. Другой конец веревки соединен с одним из компонентов гномика: объектом Leg Rope.

Прежде чем приступать к созданию веревки, нужно создать объект, который послужит шаблоном для звеньев. Затем мы создадим объект, использующий этот шаблон звена, и некоторый программный код для генерирования целой веревки. Для создания объекта звена Rope Segment выполните следующие действия.

1. *Создайте объект Rope Segment*. Создайте новый пустой игровой объект и дайте ему имя `Rope Segment`.
2. *Добавьте твердое тело в объект*. Добавьте компонент `Rigidbody2D`. Установите его свойство Mass (Масса) в значение 0,5, чтобы придать веревке немного веса.
3. *Добавьте сочленение*. Добавьте компонент `SpringJoint2D`. Установите его свойство Damping Ratio (Скорость затухания) в значение 1, а свойство Frequency (Частота) в значение 30.

Не бойтесь пробовать другие значения. Значения, упомянутые выше, позволяют, по нашему мнению, получить наиболее реалистичную имитацию веревки. Создание игр в значительной степени состоит из подбора таких значений.

4. *Создайте шаблон, используя этот объект.* Откройте папку *Gnome* в панели Assets (Ресурсы) и перетащите объект Rope Segment из панели Hierarchy (Иерархия) в панель Assets (Ресурсы). В результате в этой папке будет создан новый файл шаблона.
5. *Удалите исходный объект Rope Segment.* Он больше не нужен — мы напишем код, который будет создавать множество экземпляров объекта Rope Segment и соединять их в веревку.

Теперь создадим сам объект веревки Rope.

1. *Создайте новый пустой игровой объект и дайте ему имя Rope.*
2. *Измените пиктограмму объекта Rope.* Так как веревка не имеет визуального представления в сцене, пока игра не запущена, желательно определить для объекта отображаемую пиктограмму. Выберите вновь созданный объект Rope и щелкните по пиктограмме слева вверху в инспекторе (рис. 4.15).

   Выберите прямоугольник красного цвета с закругленными углами, и объект Rope появится в сцене в форме, напоминающей красную пилюлю (рис. 4.16).

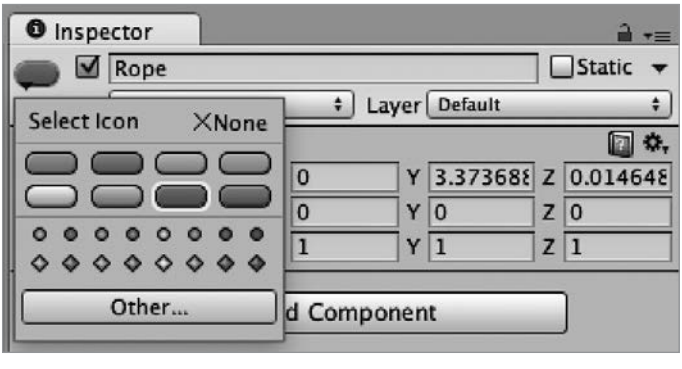

Рис. 4.15. Выбор пиктограммы для объекта Rope

3. *Добавьте твердое тело.* Щелкните по кнопке Add Component (Добавить компонент) и добавьте в объект компонент `Rigidbody2D`. Затем в инспекторе установите его свойство Body Type (Тип тела) в значение Kinematic (Кинематический). Это зафиксирует объект на месте, и он не будет падать вниз, что и требуется.
4. *Добавьте визуализатор (renderer).* Щелкните по кнопке Add Component (Добавить компонент) еще раз и добавьте компонент `LineRenderer`. Установите его свойство Width (Ширина) в значение 0,075, это придаст ему вид тонкой веревки. Остальные значения визуализатора оставьте по умолчанию.

Теперь, после настройки компонентов веревки, перейдем к созданию сценария, управляющего ими.

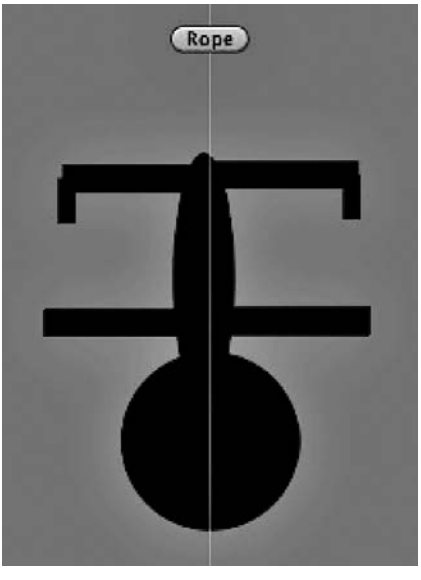

Рис. 4.16. После выбора пиктограммы объект Rope появится в сцене

Код реализации веревки

Прежде чем писать сам код, нужно добавить компонент сценария Rope. Для этого выполните следующие действия.

1. *Добавьте сценарий управления веревкой.* Требуемого сценария пока нет, но Unity создаст файл для него. Выберите объект Rope и щелкните по кнопке Add Component (Добавить компонент).

   Введите Rope; вы не увидите результатов поиска, потому что в Unity пока нет ни одного компонента с именем Rope. Но вы увидите пункт New Script (Новый сценарий), как показано на рис. 4.17. Выберите его.

   Unity предложит создать новый файл сценария. Убедитесь, что выбран язык C Sharp, а имя Rope начинается с заглавной буквы R. Щелкните по кнопке Create and Add (Создать и добавить). В ответ Unity создаст файл *Rope.cs* и подключит компонент сценария Rope к объекту Rope.

2. *Перетащите файл Rope.cs в папку Scripts.* По умолчанию Unity сохраняет новые сценарии в папке *Assets*; чтобы не нарушать порядок, перетащите его в папку *Scripts*.

3. *Добавьте код в файл Rope.cs.* Щелкните по файлу *Rope.cs* дважды либо откройте его в текстовом редакторе.

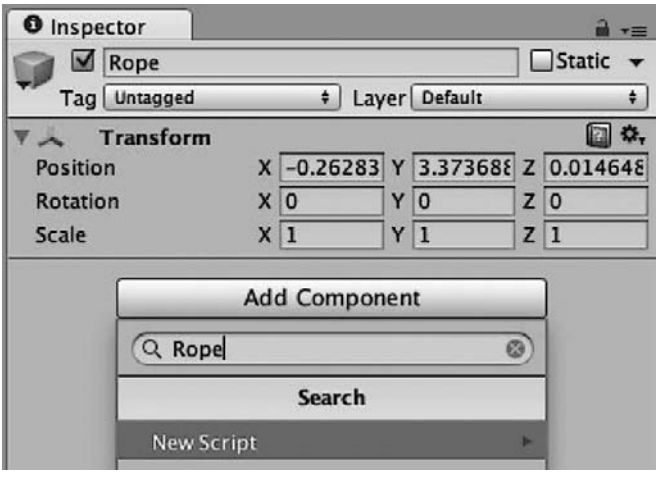

Рис. 4.17. Создание файла Rope.cs

Добавьте в него следующий код (чуть ниже мы объясним, что он делает):

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

// Веревка, состоящая из звеньев.
public class Rope : MonoBehaviour {

    // Шаблон Rope Segment для создания новых звеньев.
    public GameObject ropeSegmentPrefab;

    // Список объектов Rope Segment.
    List<GameObject> ropeSegments = new List<GameObject>();

    // Веревка удлиняется или укорачивается?
    public bool isIncreasing { get; set; }
    public bool isDecreasing { get; set; }

    // Объект твердого тела, к которому следует
    // присоединить конец веревки.
    public Rigidbody2D connectedObject;

    // Максимальная длина звена веревки
    // (если потребуется удлинить веревку больше, чем на эту величину,
    // будет создано новое звено).
    public float maxRopeSegmentLength = 1.0f;

    // Как быстро должны создаваться новые звенья веревки?
    public float ropeSpeed = 4.0f;

    // Визуализатор LineRenderer, отображающий веревку.
```

```
LineRenderer lineRenderer;

void Start() {

    // Кэшировать ссылку на визуализатор, чтобы
    // не пришлось искать его в каждом кадре.
    lineRenderer = GetComponent<LineRenderer>();

    // Сбросить состояние веревки в исходное.
    ResetLength();
}

// Удаляет все звенья и создает новое.
public void ResetLength() {

    foreach (GameObject segment in ropeSegments) {
        Destroy (segment);

    }

    ropeSegments = new List<GameObject>();

    isDecreasing = false;
    isIncreasing = false;

    CreateRopeSegment();

}

// Добавляет новое звено веревки к верхнему концу.
void CreateRopeSegment() {

    // Создать новое звено.
    GameObject segment = (GameObject)Instantiate(
    ropeSegmentPrefab,
    this.transform.position,
    Quaternion.identity);

    // Сделать звено потомком объекта this
    // и сохранить его мировые координаты
    segment.transform.SetParent(this.transform, true);

    // Получить твердое тело звена
    Rigidbody2D segmentBody = segment
        .GetComponent<Rigidbody2D>();

    // Получить длину сочленения из звена
    SpringJoint2D segmentJoint =
        segment.GetComponent<SpringJoint2D>();

    // Ошибка, если шаблон звена не имеет
    // твердого тела или пружинного сочленения - нужны оба
    if (segmentBody == null || segmentJoint == null) {
```

```
        Debug.LogError("Rope segment body prefab has no " +
                "Rigidbody2D and/or SpringJoint2D!");

            return;
        }

        // Теперь, после всех проверок, можно добавить
        // новое звено в начало списка звеньев
        ropeSegments.Insert(0, segment);

        // Если это *первое* звено, его нужно соединить
        // с ногой гномика
        if (ropeSegments.Count == 1) {
            // Соединить звено с сочленением несущего объекта
            SpringJoint2D connectedObjectJoint =
                connectedObject.GetComponent<SpringJoint2D>();

            connectedObjectJoint.connectedBody
                = segmentBody;

            connectedObjectJoint.distance = 0.1f;

            // Установить длину звена в максимальное значение
            segmentJoint.distance = maxRopeSegmentLength;
        } else {
            // Это не первое звено. Его нужно соединить
            // с предыдущим звеном

            // Получить второе звено
            GameObject nextSegment = ropeSegments[1];

            // Получить сочленение для соединения
            SpringJoint2D nextSegmentJoint =
            nextSegment.GetComponent<SpringJoint2D>();

            // Присоединить сочленение к новому звену
            nextSegmentJoint.connectedBody = segmentBody;

            // Установить начальную длину сочленения нового звена
            // равной 0 - она увеличится автоматически.
            segmentJoint.distance = 0.0f;
        }

        // Соединить новое звено с
        // опорой для веревки (то есть с объектом this)
        segmentJoint.connectedBody =
            this.GetComponent<Rigidbody2D>();
    }

    // Вызывается, когда нужно укоротить веревку,
    // и удаляет звено сверху.
    void RemoveRopeSegment() {

        // Если звеньев меньше двух, выйти.
        if (ropeSegments.Count < 2) {
```

```
            return;
        }

        // Получить верхнее звено и звено под ним.
        GameObject topSegment = ropeSegments[0];
        GameObject nextSegment = ropeSegments[1];

        // Соединить второе звено с опорой для веревки.
        SpringJoint2D nextSegmentJoint =
            nextSegment.GetComponent<SpringJoint2D>();

        nextSegmentJoint.connectedBody =
            this.GetComponent<Rigidbody2D>();

        // Удалить верхнее звено из списка и уничтожить его.
        ropeSegments.RemoveAt(0);
        Destroy (topSegment);
    }

    // При необходимости в каждом кадре длина веревки
    // удлиняется или укорачивается
    void Update() {

        // Получить верхнее звено и его сочленение.
        GameObject topSegment = ropeSegments[0];
        SpringJoint2D topSegmentJoint =
            topSegment.GetComponent<SpringJoint2D>();

        if (isIncreasing) {

            // Веревку нужно удлинить. Если длина сочленения больше
            // или равна максимальной, добавляется новое звено;
            // иначе увеличивается длина сочленения звена.

            if (topSegmentJoint.distance >=
              maxRopeSegmentLength) {
                 CreateRopeSegment();
            } else {
                topSegmentJoint.distance += ropeSpeed *
                Time.deltaTime;
            }

        }

        if (isDecreasing) {

            // Веревку нужно удлинить. Если длина сочленения
            // близка к нулю, удалить звено; иначе
            // уменьшить длину сочленения верхнего звена.

            if (topSegmentJoint.distance <= 0.005f) {
                RemoveRopeSegment();
            } else {
                topSegmentJoint.distance -= ropeSpeed *
                    Time.deltaTime;
```

```
            }

        }

        if (lineRenderer != null) {
            // Визуализатор LineRenderer рисует линию по
            // коллекции точек. Эти точки должны соответствовать
            // позициям звеньев веревки.

            // Число вершин, отображаемых визуализатором,
            // равно числу звеньев плюс одна точка
            // на верхней опоре плюс одна точка
            // на ноге гномика.
            lineRenderer.positionCount
                = ropeSegments.Count + 2;

            // Верхняя вершина всегда соответствует положению опоры.
            lineRenderer.SetPosition(0,
                this.transform.position);

            // Передать визуализатору координаты всех
            // звеньев веревки.
            for (int i = 0; i < ropeSegments.Count; i++) {
                lineRenderer.SetPosition(i+1,
                    ropeSegments[i].transform.position);
            }

            // Последняя точка соответствует точке привязки
            // несущего объекта.
            SpringJoint2D connectedObjectJoint =
                connectedObject.GetComponent<SpringJoint2D>();
            lineRenderer.SetPosition(
                ropeSegments.Count + 1,
                connectedObject.transform.
                    TransformPoint(connectedObjectJoint.anchor)
            );
        }
    }
}
```

Фрагмент кода получился большим, поэтому давайте разобьем его на части и рассмотрим их по отдельности:

```
void Start() {

    // Кэшировать ссылку на визуализатор, чтобы
    // не пришлось искать его в каждом кадре.
    lineRenderer = GetComponent<LineRenderer>();

    // Сбросить состояние веревки в исходное.
    ResetLength();

}
```

Когда объект Rope появляется в первый раз, вызывается его метод Start. Он вызывает метод ResetLength, который также вызывается, когда гномик погибает. Дополнительно в переменной lineRenderer сохраняется ссылка на компонент визуализатора:

```
// Удаляет все звенья и создает новое звено.
public void ResetLength() {

    foreach (GameObject segment in ropeSegments) {
        Destroy (segment);
    }

    ropeSegments = new List<GameObject>();

    isDecreasing = false;
    isIncreasing = false;

    CreateRopeSegment();
}
```

Метод ResetLength удаляет все звенья веревки, сбрасывает ее внутреннее состояние, очищая список ropeSegements и устанавливая свойства isDecreasing/isIncreasing, и в конце вызывает CreateRopeSegment, чтобы создать совершенно новую веревку:

```
// Добавляет новое звено веревки к верхнему концу.
void CreateRopeSegment() {

    // Создать новое звено.
    GameObject segment = (GameObject)Instantiate(
        ropeSegmentPrefab,
        this.transform.position,
        Quaternion.identity);

    // Сделать звено потомком объекта this
    // и сохранить его мировые координаты
    segment.transform.SetParent(this.transform, true);

    // Получить твердое тело звена
    Rigidbody2D segmentBody
       = segment.GetComponent<Rigidbody2D>();

    // Получить длину сочленения из звена
    SpringJoint2D segmentJoint =
        segment.GetComponent<SpringJoint2D>();

    // Ошибка, если шаблон звена не имеет
    // твердого тела или пружинного сочленения - нужны оба
    if (segmentBody == null || segmentJoint == null) {
        Debug.LogError(
           "Rope segment body prefab has no " +
           "Rigidbody2D and/or SpringJoint2D!"
```

```
            );

        return;
    }

    // Теперь, после всех проверок, можно добавить
    // новое звено в начало списка звеньев
    ropeSegments.Insert(0, segment);

    // Если это *первое* звено, его нужно соединить
    // с ногой гномика

    if (ropeSegments.Count == 1) {
        // Соединить звено с сочленением несущего объекта
        SpringJoint2D connectedObjectJoint =
            connectedObject.GetComponent<SpringJoint2D>();

        connectedObjectJoint.connectedBody =
            segmentBody;
        connectedObjectJoint.distance = 0.1f;

        // Установить длину звена на максимальное значение
        segmentJoint.distance = maxRopeSegmentLength;
    } else {
        // Это не первое звено. Его нужно соединить
        // с предыдущим звеном

        // Получить второе звено
        GameObject nextSegment = ropeSegments[1];

        // Получить сочленение для соединения
        SpringJoint2D nextSegmentJoint =
            nextSegment.GetComponent<SpringJoint2D>();

        // Присоединить сочленение к новому звену
        nextSegmentJoint.connectedBody = segmentBody;

        // Установить начальную длину сочленения нового звена
        // равной 0 - она увеличится автоматически.
        segmentJoint.distance = 0.0f;
    }

    // Соединить новое звено
    // с опорой для веревки (то есть с объектом this)
    segmentJoint.connectedBody =
        this.GetComponent<Rigidbody2D>();
}
```

`CreateRopeSegment` создает новую копию объекта Rope Segment и добавляет его на вершину цепочки, представляющей веревку. При этом он отсоединяет текущее верхнее звено веревки (если оно имеется) от опоры и присоединяет его к вновь созданному звену. Затем новое звено присоединяется к компоненту опоры `Rigidbody2D`, включенному в объект Rope.

Если новое звено единственное, то оно присоединяется к твердому телу `connectedObject`. Этой переменной присваивается ссылка на ногу гномика:

```
// Вызывается, когда нужно укоротить веревку,
// и удаляет звено сверху.
void RemoveRopeSegment() {

    // Если звеньев меньше двух, выйти.
    if (ropeSegments.Count < 2) {
        return;
    }

    // Получить верхнее звено и звено под ним.
    GameObject topSegment = ropeSegments[0];
    GameObject nextSegment = ropeSegments[1];

    // Соединить второе звено с опорой для веревки.
    SpringJoint2D nextSegmentJoint =
        nextSegment.GetComponent<SpringJoint2D>();

    nextSegmentJoint.connectedBody =
        this.GetComponent<Rigidbody2D>();

    // Удалить верхнее звено из списка и уничтожить его.
    ropeSegments.RemoveAt(0);
    Destroy (topSegment);

}
```

Метод `RemoveRopeSegment` действует с точностью до наоборот. Он удаляет верхнее звено, а следующее за ним соединяет с твердым телом объекта `Rope`. Обратите внимание, что `RemoveRopeSegment` ничего не делает, если веревка состоит из единственного звена, — это означает, что веревка не исчезнет, если укоротить ее до минимума:

```
// При необходимости в каждом кадре длина веревки
// удлиняется или укорачивается
void Update() {

    // Получить верхнее звено и его сочленение.
    GameObject topSegment = ropeSegments[0];
    SpringJoint2D topSegmentJoint =
        topSegment.GetComponent<SpringJoint2D>();

    if (isIncreasing) {

        // Веревку нужно удлинить. Если длина сочленения больше
        // или равна максимальной, добавляется новое звено;
        // иначе увеличивается длина сочленения звена.

        if (topSegmentJoint.distance >=
           maxRopeSegmentLength) {
```

```
            CreateRopeSegment();

        } else {

            topSegmentJoint.distance += ropeSpeed *
                Time.deltaTime;

        }

    }

    if (isDecreasing) {

        // Веревку нужно удлинить. Если длина сочленения
        // близка к нулю, удалить звено; иначе
        // уменьшить длину сочленения верхнего звена.

        if (topSegmentJoint.distance <= 0.005f) {
            RemoveRopeSegment();
        } else {
            topSegmentJoint.distance -= ropeSpeed *
                Time.deltaTime;
        }
    }

    if (lineRenderer != null) {
        // Визуализатор LineRenderer рисует линию по
        // коллекции точек. Эти точки должны соответствовать
        // позициям звеньев веревки.

        // Число вершин, отображаемых визуализатором,
        // равно числу звеньев плюс одна точка
        // на верхней опоре плюс одна точка
        // на ноге гномика.
        lineRenderer.positionCount =
            ropeSegments.Count + 2;

        // Верхняя вершина всегда соответствует положению опоры.
        lineRenderer.SetPosition(0,
            this.transform.position);

        // Передать визуализатору координаты всех
        // звеньев веревки.
        for (int i = 0; i < ropeSegments.Count; i++) {
            lineRenderer.SetPosition(
                i+1,
                ropeSegments[i].transform.position
             );
        }

        // Последняя точка соответствует точке привязки
        // несущего объекта.
```

```
            SpringJoint2D connectedObjectJoint =
                connectedObject.GetComponent<SpringJoint2D>();

            var lastPosition = connectedObject
                .transform
                .TransformPoint(
                    connectedObjectJoint.anchor
                );

            lineRenderer.SetPosition(
                ropeSegments.Count + 1,
                position
            );
        }
    }
```

Каждый раз, когда вызывается метод `Update` (то есть когда игра перерисовывает экран), он проверяет, не равно ли свойство `isIncreasing` или `isDecreasing` значению `true`.

Если проверка показывает, что свойство `isIncreasing` равно `true`, веревка удлиняется увеличением значения свойства `distance` пружинного сочленения верхнего звена. Если значение этого свойства оказалось больше или равно значению переменной `maxRopeSegmentLength` , создается новое звено.

Аналогично, если свойство `isDecreasing` равно `true`, свойство `distance` уменьшается. Если его значение оказалось близко к нулю, верхнее звено удаляется.

В заключение обновляется содержимое визуализатора `LineRenderer`, чтобы вершины, определяющие линию веревки, совпадали с местоположениями ее звеньев.

Настройка объекта веревки Rope

Теперь, закончив работу над кодом веревки, можно добавить в сцену объекты, использующие его. Для этого выполните следующие действия.

1. *Настройте объект Rope*. Выберите игровой объект Rope. Перетащите шаблон Rope Segment в поле Rope Segment Prefab (Шаблон Rope Segment) и объект Leg Rope в поле Connected Object (Несущий объект). Во всех остальных полях оставьте значения по умолчанию, которые определены в файле *Rope.cs*. В результате инспектор с содержимым объекта Rope должен выглядеть так, как показано на рис. 4.18.
2. *Запустите игру*. Теперь гномик повиснет под объектом Rope, а вы увидите линию, соединяющую гномика с точкой чуть выше него.

Для завершения работы с объектом Rope остался один шаг — необходимо настроить материал для использования визуализатором Line Renderer.

1. *Создайте материал*. Откройте меню Assets (Ресурсы) и выберите пункт Create ▸ Material (Создать ▸ Материал). Дайте новому материалу имя Rope.

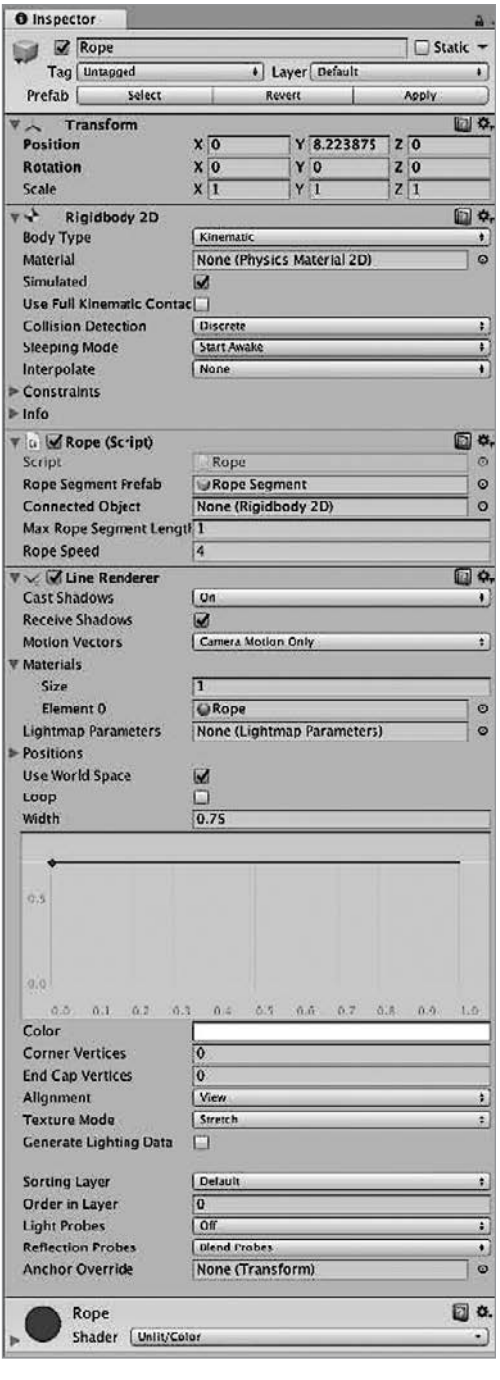

Рис. 4.18. Настроенный объект Rope

2. *Настройте материал Rope*. Выберите новый материал Rope и откройте меню Shader (Шейдер) в инспекторе. Выберите пункт Unlit ▸ Color (Неосвещенный ▸ Цвет).

   Содержимое инспектора изменится, и в нем появится раздел с параметрами для нового шейдера, включающий единственное поле цвета. Выберите в этом поле темно-коричневый цвет, щелкнув по нему и выбрав цвет из появившейся палитры.

3. *Задействуйте новый материал в объекте Rope*. Выберите объект Rope и откройте свойство Materials (Материалы). Перетащите в поле Element 0 (Элемент 0) только что созданный материал Rope.

4. *Снова запустите игру*. Теперь веревка будет коричневого цвета.

В заключение

На данный момент основная структура игры начала приобретать очертания. Мы реализовали два наиболее важных элемента игры: куклу гномика и веревку, на которой он подвешен.

В следующей главе мы приступим к созданию систем, реализующих игровой процесс с использованием этих объектов. Это будет здорово.

5 Подготовка к игре

Теперь, когда созданы гномик и веревка, пришло время настроить систему, которая позволит пользователю управлять игрой.

Мы сделаем это в два этапа: сначала добавим сценарий, который будет отклонять гномика влево и вправо при наклоне телефона. А затем добавим кнопки, удлиняющие и укорачивающие веревку.

Как только это будет сделано, мы приступим к реализации кода, управляющего ходом самой игры: сначала реализуем настройки, набор которых будет использоваться гномиком, а затем создадим объект диспетчера для слежения за некоторыми важными событиями в игре.

Ввод

Мы подошли к тому моменту, когда нам нужно получить ввод с устройства, поэтому давайте посмотрим, сможет ли Unity Editor принимать ввод. Если нет, то единственная возможность протестировать игру — собрать ее и установить на устройство, однако это может отнять много времени. Но Unity разрабатывался специально для того, чтобы дать возможность быстро тестировать изменения, а ожидание завершения сборки слишком замедлит вашу работу.

Unity Remote

Для быстрой передачи ввода в редактор разработчики Unity разместили в онлайн-магазине приложений App Store мобильное приложение под названием Unity Remote. Unity Remote соединяется с Unity Editor посредством кабеля для телефона; когда редактор проигрывает игру, на экране телефона отображается копия окна Game (Игра) и вашему сценарию посылаются обратно все касания и информация с датчиков. Это позволяет тестировать игру без необходимости ее сборки — достаточно лишь запустить приложение на телефоне, и можно играть в игру, как если бы она была установлена.

Приложение Unity Remote имеет несколько недостатков.

- Отображая игру на экране телефона, Unity довольно сильно сжимает изображение. Помимо ухудшения качества картинки передача изображения на телефон идет с некоторой задержкой и снижением частоты кадров.
- Из-за того что игра запускается на компьютере, частота кадров будет получаться иной, чем при выполнении на телефоне. Если сцена наполнена мелкими графическими элементами или на выполнение сценариев уходит значительная доля времени отображения каждого кадра, то вам не удастся получить ту же производительность, что могла бы быть при выполнении игры непосредственно на телефоне.
- Наконец, как вы понимаете, все это работает только тогда, когда телефон соединен с компьютером.

Чтобы задействовать Unity Remote, установите приложение на своем устройстве, запустите его и соедините телефон с компьютером USB-кабелем. Затем щелкните по кнопке Play (Играть). Игра появится на экране вашего устройства.

Если на экране устройства ничего не появилось, откройте меню Edit (Правка) и выберите пункт Project Settings ▸ Editor (Настройки проекта ▸ Редактор). В инспекторе откроются настройки редактора. Выберите в параметре Device (Устройство) свой телефон.

Самые последние инструкции по установке Unity Remote на ваше устройство вы найдете в онлайн-документации к Unity: *http://www.bit.ly/unity-remote-5*.

Добавление управления наклоном

Поддержка управления игрой наклоном устройства будет реализована в двух сценариях: `InputManager` (читающий информацию с датчика акселерометра) и `Swinging` (получающий входные данные от `InputManager` и применяющий боковую силу к телу — в данном случае к телу гномика).

Создание класса Singleton

Диспетчер ввода `InputManager` — это объект-одиночка. То есть в сцене всегда будет присутствовать единственный объект `InputManager`, а все другие объекты будут обращаться к нему. Позднее нам также понадобится добавлять в игру другие виды объектов-одиночек, поэтому есть смысл создать класс, который потом мы сможем повторно использовать в разных сценариях. Для создания класса `Singleton`, который будет использован для создания диспетчера ввода `InputManager`, выполните следующие действия.

1. *Создайте сценарий Singleton*. Создайте новый ресурс сценария на языке C# в папке *Scripts*, открыв меню Assets (Ресурсы) и выбрав пункт Create ▸ C# Script (Создать ▸ Сценарий C#). Дайте сценарию имя Singleton.

2. *Добавьте код в сценарий Singleton.* Откройте файл *Singleton.cs* и замените его содержимое следующим кодом:

```
using UnityEngine;
using System.Collections;

// Этот класс позволяет другим объектам ссылаться на единственный
// общий объект. Используется в классах GameManager и InputManager.

// Чтобы воспользоваться им, унаследуйте его, например:
// public class MyManager : Singleton<MyManager> { }

// После этого появится возможность обращаться к единственному
// общему экземпляру класса, например, так:
// MyManager.instance.DoSomething();

public class Singleton<T> : MonoBehaviour
  where T : MonoBehaviour {

  // Единственный экземпляр класса.
  private static T _instance;

  // Метод доступа. В первом вызове настроит свойство _instance.
  // Если требуемый объект не найден, выводит сообщение об ошибке.
  public static T instance {
    get {
      // Если свойство _instance еще не настроено ...
      if (_instance == null)
      {
        // Попытаться найти объект.
        _instance = FindObjectOfType<T>();

        // Вывести собщение в случае неудачи.
        if (_instance == null) {
          Debug.LogError("Can't find " +
            typeof(T) + "!");
        }
      }

      // Вернуть экземпляр для использования!
      return _instance;
    }
  }
}
```

Вот как действует класс `Singleton`: другие классы наследуют этот шаблонный класс и открывают доступ к статическому свойству с именем `instance`. Это свойство всегда ссылается на общий экземпляр класса. То есть когда другой сценарий обратится к свойству `InputManager.instance`, он получит единственный экземпляр `InputManager`.

При таком подходе другим сценариям не требуется хранить ссылки на экземпляр `InputManager`.

Реализация объекта-одиночки InputManager

Теперь создадим класс `InputManager`, наследующий `Singleton`.

1. *Создайте игровой объект InputManager.* Создайте новый игровой объект и дайте ему имя InputManager.
2. *Создайте и добавьте сценарий InputManager.* Выберите объект InputManager и щелкните по кнопке Add Component (Добавить компонент). Введите `InputManager` и выберите создание нового сценария. Убедитесь, что компонент сценария получил имя InputManager и для него выбран язык C Sharp.
3. *Добавьте код в сценарий InputManager.cs.* Откройте только что созданный файл *InputManager.cs* и добавьте в него следующий код:

```
using UnityEngine;
using System.Collections;

// Преобразует данные, полученные от акселерометра,
// в информацию о боковом смещении.
public class InputManager : Singleton<InputManager> {

  // Величина смещения. -1.0 = максимально влево,
  // +1.0 = максимально вправо
  private float _sidewaysMotion = 0.0f;

  // Это свойство доступно только для чтения, поэтому
  // другие сценарии не смогут изменить его.
  public float sidewaysMotion {
    get {
      return _sidewaysMotion;
    }
  }

  // Величина отклонения сохраняется в каждом кадре
  void Update () {
    Vector3 accel = Input.acceleration;

    _sidewaysMotion = accel.x;
  }
}
```

В каждом кадре объект класса `InputManager` извлекает данные из акселерометра, используя встроенный класс `Input`, и сохраняет компонент X (определяющий силу, действующую на левую и правую стороны устройства) в переменную. Эта переменная определена как общедоступное свойство только для чтения `sidewaysMotion`.

Свойства, доступные только для чтения, исключают возможность случайного их изменения другими классами.

Проще говоря, если любой другой класс захочет узнать, насколько телефон наклонен влево или вправо, ему достаточно прочитать свойство `InputManager.instance.sidewaysMotion`.

Теперь напишем код для сценария Swinging.

1. *Выберите объект Body тела гномика.*
2. *Создайте и добавьте новый сценарий на C# с именем Swinging.* Добавьте в файл сценария *Swinging.cs* следующий код:

```
using UnityEngine;
using System.Collections;

// Использует диспетчер ввода для приложения боковой силы
// к объекту. Используется для отклонения гномика в сторону.
public class Swinging : MonoBehaviour {

  // Насколько большим должно быть отклонение?
  // Больше число = больше отклонение
  public float swingSensitivity = 100.0f;

  // Вместо Update используется FixedUpdate, чтобы
  // упростить работу с физическим движком
  void FixedUpdate() {

    // Если твердое тело отсутствует (уже), удалить
    // этот компонент
    if (GetComponent<Rigidbody2D>() == null) {
      Destroy (this);
      return;
    }

    // Получить величину наклона из InputManager
    float swing = InputManager.instance.sidewaysMotion;

    // Вычислить прилагаемую силу
    Vector2 force =
      new Vector2(swing * swingSensitivity, 0);

    // Приложить силу
    GetComponent<Rigidbody2D>().AddForce(force);
  }

}
```

Код класса `Swinging` выполняется каждый раз, когда обновляется состояние физической системы. Сначала он проверяет наличие в объекте компонента `Rigidbody2D`. Если компонент отсутствует, дальнейшее выполнение немедленно прекращается. Если компонент присутствует, из экземпляра `InputManager` извлекается значение `swidewaysMotion` и на его основе конструируется вектор `Vector2`, описывающий прикладываемую силу, после чего вычисленная сила прикладывается к твердому телу объекта.

3. *Запустите игру.* Запустите Unity Remote на телефоне и попробуйте наклонять его влево и вправо; гномик должен перемещаться соответственно наклонам.

Если наклонить телефон слишком сильно, Unity Remote может перейти в альбомную ориентацию, растянув картинку по горизонтали. Предотвратить это можно, включив в телефоне запрет на смену ориентации.

Управление веревкой

Теперь добавим кнопки, удлиняющие и укорачивающие веревку. Для этого нам понадобятся две кнопки графического интерфейса Unity; нажимая и удерживая кнопку Down, пользователь будет сообщать, что веревка должна удлиниться, а когда пользователь перестает удерживать кнопку, веревка должна перестать удлиняться. Кнопка Up действует аналогично и управляет началом и концом укорачивания веревки.

1. *Добавьте кнопку.* Откройте меню GameObject (Игровой объект) и выберите пункт UI ▸ Button (Пользовательский интерфейс ▸ Кнопка). В результате будет добавлена кнопка, а также компонент Canvas для ее отображения и компонент EventSystem, обрабатывающий ее ввод. (Вам не придется беспокоиться о добавлении этих дополнительных компонентов.) Дайте игровому объекту кнопки имя Down.
2. *Привяжите кнопку к правому нижнему углу.* Выберите кнопку Down и щелкните по кнопке Anchor (Привязка), которая появится слева вверху в панели инспектора. Нажмите клавиши Shift и Alt (Option на Mac) и, удерживая их, щелкните на варианте bottom-right (правый-нижний, как показано на рис. 5.1). Этими действиями вы привяжете кнопку к правому нижнему углу экрана, после чего кнопка переместится в правый нижний угол экрана.
3. *Создайте надпись на кнопке с текстом «Down».* Объект Button имеет единственный дочерний объект с именем Text. Этот объект представляет надпись на кнопке. Выберите его, найдите в инспекторе присоединенный к объекту компонент Text. Введите в свойство Text текст «Down». Надпись на кнопке также сменится на Down.
4. *Удалите компонент Button из объекта Button.* Щелкните по пиктограмме с шестеренкой в правом верхнем углу в разделе компонента и выберите пункт Remove Component (Удалить компонент).

Это может показаться немного неожиданным, но нам просто нужно избавиться от стандартного поведения «обычной» кнопки.

Обычная кнопка генерирует событие, когда «нажимается и отпускается», то есть когда пользователь касается пальцем кнопки, а затем убирает палец. Событие посылается только в момент, когда палец убирается, что нам не подходит, — нам нужно, чтобы в момент касания кнопки генерировалось первое событие, а когда палец убирается — второе.

Поэтому мы вручную добавим необходимые компоненты, которые будут посылать сообщения объекту веревки.

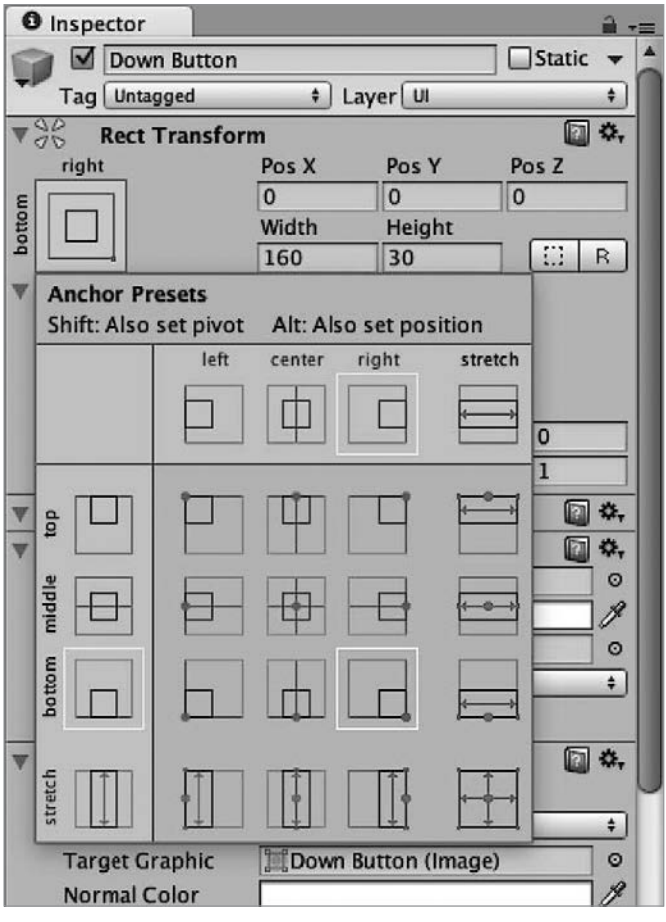

Рис. 5.1. Привязка кнопки Down к правому нижнему углу сцены; на этом скриншоте удержание клавиш Shift и Alt во время выбора варианта приведет к тому, что одновременно с привязкой произойдет установка и позиционирование опорной точки

5. *Добавьте в объект Button компонент Event Trigger.* Этот компонент следит за взаимодействиями и посылает сообщения, когда такие взаимодействия происходят.
6. *Добавьте событие Pointer Down.* Щелкните по кнопке Add New Event Type (Добавить новый тип событий) и выберите из появившегося списка пункт Pointer Down (Нажатие).
7. *Соедините свойство isIncreasing объекта Rope с событием.* Щелкните по кнопке + в списке Pointer Down, и в нем появится новый элемент (рис. 5.2).

Перетащите объект Rope с панели иерархии в появившийся элемент списка.

Замените No Function свойством Rope ▸ isIncreasing. (После выбора свойства оно отобразится в раскрывающемся меню как `Rope.isIncreasing`.) Благодаря этому нажатие кнопки будет изменять свойство `isIncreasing` веревки.

Установите флажок, который появится чуть ниже. Этот флажок сообщает, что в ответ на нажатие кнопки в свойство `isIncreasing` должно записываться значение `true`.

В результате новый элемент в событии Pointer Down должен выглядеть так, как показано на рис. 5.3.

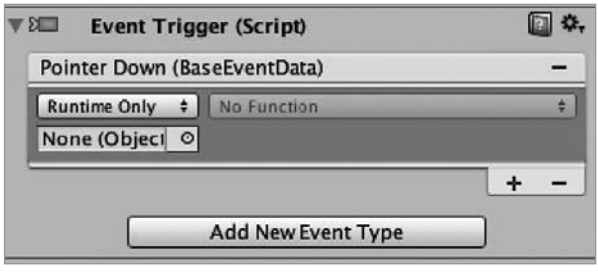

Рис. 5.2. Новое событие в списке

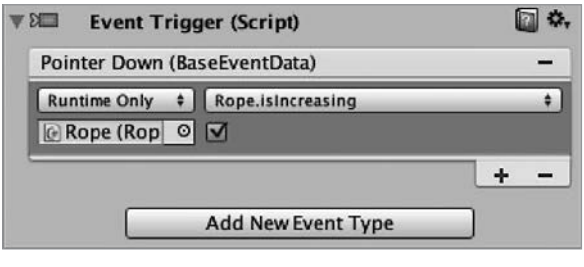

Рис. 5.3. Настроенное событие Pointer Down (Нажатие)

8. *Добавьте событие Pointer Up и заставьте его записывать false в свойство isIncreasing объекта Rope*. Когда палец убирается с кнопки, ее удлинение должно прекратиться.

   Добавьте новое событие Pointer Up в компонент Event Trigger, щелкнув по кнопке Add New Event Type (Добавить новый тип событий), и снимите флажок для свойства `isIncreasing` объекта Rope. Благодаря этому свойство `isIncreasing` будет получать значение `false` при отпускании кнопки.

   После этого раздел Event Trigger в инспекторе должен выглядеть так, как показано на рис. 5.4.

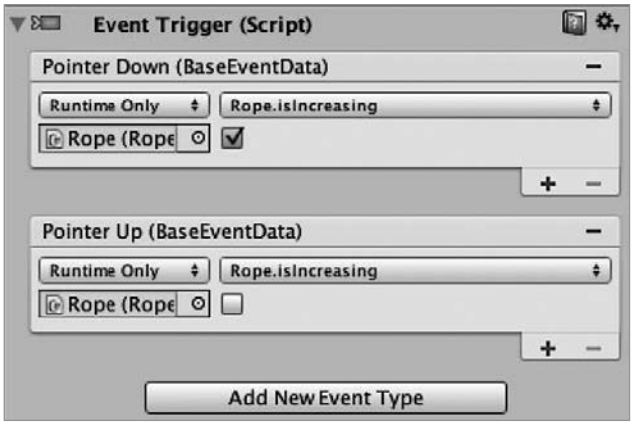

Рис. 5.4. Настройки компонента Event Trigger для кнопки Down

9. *Протестируйте кнопку Down*. Запустите игру, затем нажмите и удерживайте кнопку Down. Веревка должна начать удлиняться и прекратить, когда вы отпустите кнопку мыши. Если игра ведет себя по-другому, проверьте еще раз настройки событий для кнопки Down; событие Pointer Down должно записывать в свойство `isIncreasing` значение `true`, а событие Pointer Up должно записывать в свойство `isIncreasing` значение `false`.

10. *Добавьте кнопку Up*. Повторите тот же процесс, но для кнопки, укорачивающей веревку. Добавьте новую кнопку, расположите ее над кнопкой Down и привяжите к правому нижнему углу.

    Создайте надпись на кнопке с текстом «Up», удалите компонент Button и добавьте компонент Event Trigger (с событиями двух типов — Pointer Down и Pointer Up). События компонента Event Trigger должны воздействовать на свойство `isDecreasing` объекта Rope.

    Эти две кнопки отличаются только текстом надписи и свойствами, на которые они воздействуют. В остальном они совершенно идентичны.

11. *Протестируйте кнопку Up*. Снова запустите игру. Теперь нажатие на кнопку должно удлинять и укорачивать веревку.

    Вы можете также воспользоваться приложением Unity Remote, чтобы запустить игру на телефоне и попробовать наклонять телефон влево-вправо и одновременно удлинять или укорачивать веревку.

    Поздравляем: основа системы ввода готова!

Настройка камеры для следования за гномиком

Теперь, если удерживать нажатой кнопку Down, гномик будет опускаться на веревке все ниже и ниже и, наконец, исчезнет из поля зрения. Чтобы этого не произошло, камера должна следовать за гномиком.

Для этого создадим сценарий, который будет подключаться к камере и корректировать ее координату Y (то есть вертикальную позицию) в соответствии с положением другого объекта. Назначив этим другим объектом объект Gnome, мы обеспечим синхронное следование камеры за гномиком. Сценарий будет подключаться к камере и настраиваться на следование за телом гномика. Чтобы создать сценарий, выполните следующие действия.

1. *Добавьте сценарий* CameraFollow. Выберите объект Camera в иерархии и добавьте новый компонент сценария на C#, дав ему имя CameraFollow.
2. *Добавьте следующий код в файл сценария CameraFollow.cs:*

```
// Синхронизирует позицию камеры с позицией Y
// целевого объекта, соблюдая некоторые ограничения.
public class CameraFollow : MonoBehaviour {

  // Целевой объект, с позицией Y которого будет
  // синхронизироваться положение камеры.
  public Transform target;

  // Наивысшая точка, где может находиться камера.
  public float topLimit = 10.0f;

  // Низшая точка, где может находиться камера.
  public float bottomLimit = -10.0f;

  // Скорость следования за целевым объектом.
  public float followSpeed = 0.5f;

  // Определяет положение камеры после установки
  // позиций всех объектов
  void LateUpdate () {

    // Если целевой объект определен...
    if (target != null) {

      // Получить его позицию
      Vector3 newPosition = this.transform.position;

      // Определить, где камера должна находиться
      newPosition.y = Mathf.Lerp (newPosition.y,
        target.position.y, followSpeed);

      // Предотвратить выход позиции за граничные точки
      newPosition.y =
        Mathf.Min(newPosition.y, topLimit);
      newPosition.y =
        Mathf.Max(newPosition.y, bottomLimit);

      // Обновить местоположение
      transform.position = newPosition;
    }

  }
```

```
  // Если камера выбрана в редакторе, рисует линию от верхней
  // граничной точки до нижней.
  void OnDrawGizmosSelected() {
    Gizmos.color = Color.yellow;

    Vector3 topPoint =
      new Vector3(this.transform.position.x,
        topLimit, this.transform.position.z);
    Vector3 bottomPoint =
      new Vector3(this.transform.position.x,
        bottomLimit, this.transform.position.z);

    Gizmos.DrawLine(topPoint, bottomPoint);
  }
}
```

Сценарий `CameraFollow` использует метод `LateUpdate`, который вызывается после вызова метода `Update` всех других объектов. Метод `Update` часто используется для изменения координат объектов, а это значит, что код в `LateUpdate` будет выполняться *после* изменения позиций.

Сценарий `CameraFollow` корректирует позицию Y компонента `transform` объекта, к которому он подключен, но при этом гарантирует, что позиция не выйдет за граничные точки. Это означает, что когда веревка укоротится до минимума, камера не покажет пустое пространство над колодцем. Кроме того, сценарий использует функцию `Mathf.Lerp` для вычисления позиции камеры, близкой к позиции целевого объекта. Это создает эффект «свободного» следования за объектом — чем ближе значение параметра `followSpeed` будет к единице, тем быстрее будет двигаться камера.

Визуализация граничных точек реализуется методом `OnDrawGizmosSelected`. Этот метод используется самим редактором Unity и рисует линию от верхней до нижней граничной точки, когда выбрана камера. Если в инспекторе изменить свойство `topLimit` и/или `bottomLimit`, длина линии изменится.

3. *Настройте компонент CameraFollow.* Перетащите объект Body гномика в поле Target (Цель) (рис. 5.5) и оставьте другие свойства как есть.

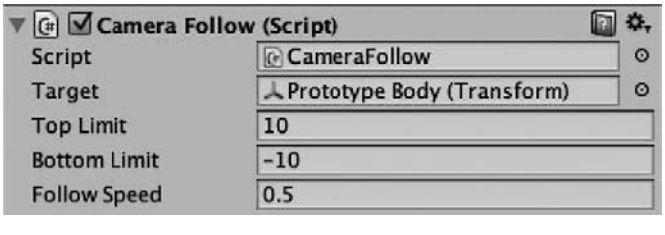

Рис. 5.5. Настройки сценария CameraFollow

4. *Протестируйте камеру.* Запустите игру и опустите гномика вниз кнопкой Down. Камера должна последовать за гномиком.

Отладка сценариев

Сейчас самое время обсудить поиск и исправление проблем в сценариях, потому что дальше код будет становиться все более сложным.

Иногда сценарии действуют не так, как задумывалось, из-за опечаток или логических ошибок. Для поиска и решения таких проблем в сценариях можно использовать средства отладки, имеющиеся в MonoDevelop: расставлять контрольные точки в коде, исследовать состояние программы и вообще иметь полный контроль над выполнением сценариев.

Для правки сценариев можно использовать любой текстовый редактор, но для их отладки нужна специализированная среда разработки, такая как MonoDevelop или Visual Studio. В этой книге мы используем MonoDevelop; если вы хотите использовать Visual Studio, то у Microsoft есть отличная документация (*http://www.bit.ly/ms-debugger-basics*).

Установка контрольных точек

Для исследования этой возможности установим контрольную точку в только что написанном сценарии Rope, чтобы получить более полное представление о его поведении. Для этого выполните следующие действия.

1. *Откройте файл сценария Rope.cs в MonoDevelop.*
2. *Найдите метод Update.* В частности, найдите строку:

```
if (topSegmentJoint.distance >= maxRopeSegmentLength) {
```

3. *Щелкните на сером поле слева от этой строки.* В результате будет добавлена контрольная точка (рис. 5.6).

```
// Every frame, increase or decrease the rope's length if neccessary
void Update() {

    // Get the top segment and its joint.
    GameObject topSegment = ropeSegments[0];
    SpringJoint2D topSegmentJoint =
        topSegment.GetComponent<SpringJoint2D>();

    if (isIncreasing) {

        // We're increasing the rope. If it's at max length,
        // add  a new segment; otherwise, increase the top
        // rope segment's length.

        if (topSegmentJoint.distance >= maxRopeSegmentLength) {
            CreateRopeSegment();
        } else {
            topSegmentJoint.distance += ropeSpeed *
                Time.deltaTime;
        }

    }
```

Рис. 5.6. Добавление контрольной точки

Далее свяжем MonoDevelop с Unity, чтобы по достижении контрольной точки MonoDevelop приостановил работу Unity и его окно всплыло вверх.

4. *Щелкните по кнопке Play (Играть) слева вверху в окне MonoDevelop* (рис. 5.7).

Рис. 5.7. Кнопка Play (Играть) слева вверху в окне MonoDevelop

5. *В появившемся окне щелкните по кнопке Attach (Подключиться)*, как показано на рис. 5.8.

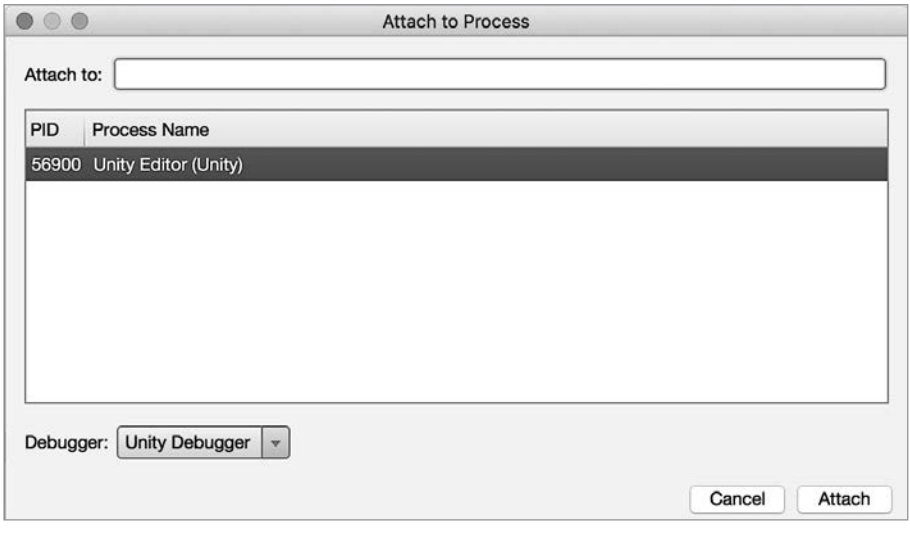

Рис. 5.8. Диалог Attach to Process (Подключение к процессу)

После этого MonoDevelop подключится к процессу Unity. Когда выполнение достигнет контрольной точки, MonoDevelop приостановит Unity и даст возможность отладить код.

Говоря, что Unity приостановится, мы не имеем в виду приостановку *игры* внутри Unity, как это происходит в случае щелчка по кнопке Pause (Пауза). *Приостановится приложение Unity целиком* и будет продолжать стоять, пока мы не потребуем от MonoDevelop возобновить выполнение. Если вам покажется, что приложение Unity зависло, не волнуйтесь.

6. *Запустите игру и щелкните по кнопке Down.*

В тот момент, когда это произойдет, Unity приостановится и всплывет окно MonoDevelop. Строка с контрольной точкой подсветится, показывая текущую точку выполнения.

Теперь вы можете изучить состояние программы. В нижней части окна MonoDevelop можно увидеть две панели: Locals (Текущие данные) и Immediate (Вмешательство). (В зависимости от ситуации могут быть открыты другие вкладки; если это так, просто щелкните на нужной вкладке, чтобы открыть ее.)

В панели Locals (Текущие данные) отображается список переменных, присутствующих в текущей области видимости.

7. Откройте переменную `topSegmentJoint` в панели Locals (Текущие данные). Внутри появится список полей, что позволит исследовать их (рис. 5.9).

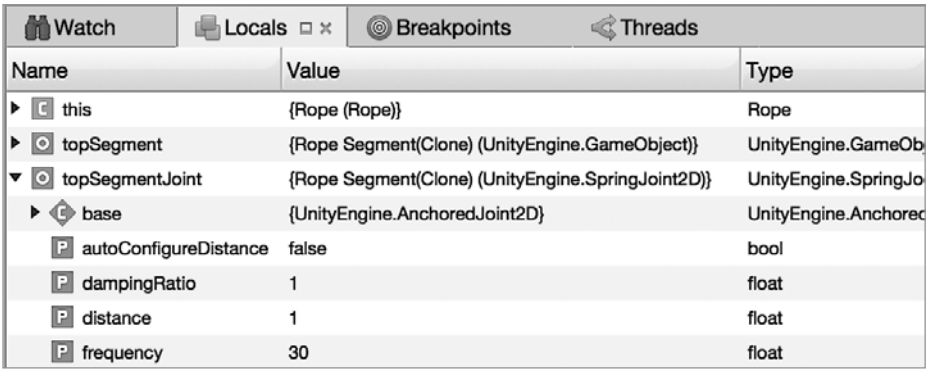

Рис. 5.9. Панель Locals (Текущие данные) отображает данные внутри topSegmentJoint

Панель Immediate (Вмешательство) позволяет вводить код на языке C# и получать результат его выполнения. Например, ту же информацию о свойстве `distance` переменной `topSegmentJoint`, что показана на рис. 5.9, можно получить, введя выражение `topSegmentJoint.distance`.

Закончив отладку кода, нужно подсказать отладчику, что можно возобновить выполнение Unity. Сделать это можно двумя способами: отсоединить отладчик от процесса или, оставив его подключенным, дать команду продолжить выполнение.

Если отсоединить отладчик от процесса, он перестанет останавливать выполнение в контрольных точках и вам придется вновь подключать его. Если оставить отладчик подключенным, следующая же контрольная точка вновь приостановит игру.

- Чтобы отсоединить отладчик, щелкните по кнопке Stop (Стоп), изображенной на рис. 5.10.

Рис. 5.10. Остановка (отключение) отладчика

- Чтобы оставить отладчик подключенным и продолжить выполнение, щелкните по кнопке Continue (Продолжить), изображенной на рис. 5.11.

Рис. 5.11. Продолжение выполнения

Код реализации поведения гномика

Настал момент заняться самим гномиком. Гномик должен хранить некоторую информацию о своем состоянии и уметь определять, что с ним происходит.

В частности, нам нужно следующее.

- Когда гномик получает повреждение, мы должны отобразить некоторый визуальный эффект с применением системы частиц (в зависимости от полученного повреждения).
- В случае гибели гномика мы должны:
  - обновить спрайты, соответствующие разным частям тела (в зависимости от повреждения), и отделить некоторые из них;
  - создать объект духа Ghost сразу после гибели, который автоматически начнет подниматься вверх;
  - изобразить фонтан крови, бьющий из тела, когда отделяется какая-то его часть; нам потребуется узнать, из какой точки должен бить фонтан для каждой части тела;
  - когда оторванная или отрезанная часть тела прекратит падение, она должна потерять все физические характеристики, чтобы не мешать игроку (нам совсем не нужно, чтобы гора погибших гномиков помешала взять сокровище).
- Когда гномик касается сокровища, мы должны заменить спрайт пустой руки спрайтом с изображением руки, *держащей* сокровище.
- Также в объекте гномика должна храниться и другая информация, например, целевой объект, за которым должна следовать камера, а также какое твердое тело должно быть связано с веревкой.
- Гномик должен хранить признак — жив он или мертв.

Обратите внимание, что эта информация не имеет ничего общего с состоянием игры в целом — гномик не знает, выиграли вы или нет, он хранит информацию только о себе самом. Мы также (позднее) создадим объект, управляющий состоянием игры и вызывающий гибель гномика.

Для реализации перечисленного нам потребуется написать сценарий, управляющий гномиком как одним целым. Также понадобится добавить по сценарию

в каждую часть тела (для управления их спрайтами и удаления после падения на дно колодца.)

Помимо этого, нам нужно добавить дополнительную информацию о точках, откуда должны бить фонтаны крови. Эти точки будут представлены игровыми объектами (потому что они хранят свои позиции в сцене); каждая часть тела будет иметь ссылку на соответствующую точку для «фонтана крови».

Начнем со сценария для части тела, а затем перейдем к сценарию, реализующему поведение гномика. Мы выбрали такой порядок потому, что главный сценарий управления гномиком должен знать о сценарии `BodyPart` для части тела, тогда как сценарий `BodyPart` ничего не должен знать о гномике.

1. *Создайте файл BodyPart.cs.* Создайте новый сценарий на языке C# с именем *BodyPart.cs*. Добавьте в него следующий код:

```
[RequireComponent (typeof(SpriteRenderer))]
public class BodyPart : MonoBehaviour {

  // Спрайт, используемый в вызове ApplyDamageSprite
  // с повреждением типа 'порез'
  public Sprite detachedSprite;

  // Спрайт, используемый в вызове ApplyDamageSprite
  // с повреждением типа 'ожог'
  public Sprite burnedSprite;

  // Представляет позицию и поворот для отображения фонтана
  // крови, бьющего из основного тела
  public Transform bloodFountainOrigin;

  // Если имеет значение true, после падения из этого объекта должны
  // быть удалены все коллизии, сочленения и твердое тело
  bool detached = false;

  // Отделяет объект this от родителя и устанавливает флаг,
  // требующий удаления физических свойств
  public void Detach() {
    detached = true;

    this.tag = "Untagged";

    transform.SetParent(null, true);

}

  // В каждом кадре, если часть тела отделена от основного тела,
  // удаляет физические характеристики после достижения дна колодца.
  // Это означает, что отделенная часть тела не будет мешать
  // гномику взять сокровище.
  public void Update() {

    // Если часть тела не отделена, ничего не делать
```

```
    if (detached == false) {
      return;
    }

    // Твердое тело прекратило падение?
    var rigidbody = GetComponent<Rigidbody2D>();

    if (rigidbody.IsSleeping()) {

      // Если да, удалить все сочленения...
      foreach (Joint2D joint in
        GetComponentsInChildren<Joint2D>()) {
        Destroy (joint);
      }

      // ...твердые тела...
      foreach (Rigidbody2D body in
        GetComponentsInChildren<Rigidbody2D>()) {
        Destroy (body);
      }

      // ...и коллайдеры.
      foreach (Collider2D collider in
        GetComponentsInChildren<Collider2D>()) {
        Destroy (collider);
      }

      // В конце удалить компонент с этим сценарием.
      Destroy (this);
    }
  }

  // Заменяет спрайт этой части тела, исходя из
  // вида полученного повреждения
  public void ApplyDamageSprite(
    Gnome.DamageType damageType) {

    Sprite spriteToUse = null;

    switch (damageType) {

    case Gnome.DamageType.Burning:
      spriteToUse = burnedSprite;

      break;

    case Gnome.DamageType.Slicing:
      spriteToUse = detachedSprite;

      break;
    }

    if (spriteToUse != null) {
```

```
        GetComponent<SpriteRenderer>().sprite =
          spriteToUse;
      }

    }

  }
```

Этот код пока не компилируется, потому что в нем используется тип `Gnome.DamageType`, который еще не определен. Мы добавим его, когда будем создавать класс `Gnome`.

Сценарий `BodyPart` обрабатывает два вида повреждений: ожог и порез. Они представлены перечислением `Gnome.DamageType`, которое мы определим чуть ниже, и будут использоваться методами, обрабатывающими повреждения в нескольких разных классах. Повреждение `Burning` (ожог), наносимое некоторыми видами ловушек, будет вызывать визуальный эффект вспышки, а повреждение `Slicing` (порез), наносимое другими ловушками, будет вызывать эффект (весьма кровавый) отсечения части тела, сопровождаемый потоками красной крови, бьющими из тела гномика.

Сам класс `BodyPart` отмечен как требующий наличия компонента `SpriteRenderer` в игровом объекте, к которому данный сценарий подключается. Так как разные типы повреждений вызывают необходимость изменения спрайта, представляющего часть тела, вполне разумно потребовать, чтобы любой объект, к которому подключается сценарий `BodyPart`, также имел визуализатор `SpriteRenderer`.

Класс имеет несколько разных свойств: `detachedSprite` определяет спрайт, который должен использоваться, когда гномик получает повреждение `Slicing` (порез), а `burnedSprite` определяет спрайт, который должен использоваться, когда гномик получает повреждение `Burning` (ожог). Кроме того, свойство `bloodFountainOrigin` типа `Transform` будет использоваться главным компонентом `Gnome` для добавления объекта, представляющего фонтан крови; оно не используется данным классом, но хранит необходимую информацию.

Дополнительно сценарий `BodyPart` определяет момент завершения падения компонента `RigidBody2D` (то есть когда в течение нескольких мгновений он остается неподвижным и на него не воздействуют никакие новые силы). В этом случае сценарий `BodyPart` удаляет все компоненты, определяющие физические свойства части тела, кроме визуализатора спрайта, фактически превращая упавшую часть тела в декорацию. Это необходимо, чтобы помешать заполнению колодца останками частей тела, которые могли бы затруднить действия игрока.

К реализации фонтана крови мы еще вернемся в разделе «Эффекты частиц» в главе 8; здесь мы ограничимся начальной подготовкой, которая позволит потом добавить все необходимое намного быстрее.

Теперь добавим сам сценарий Gnome. По сути это будет лишь заготовка для дальнейшего развития в будущем, когда мы реализуем гибель гномика, но будет хорошо, если мы подготовим его заранее.

2. *Создайте сценарий Gnome.* Создайте новый файл *Gnome.cs* сценария на языке C#.
3. *Добавьте код для компонента Gnome.* Добавьте следующий код в файл *Gnome.cs*:

```
public class Gnome : MonoBehaviour {

  // Объект, за которым должна следовать камера.
  public Transform cameraFollowTarget;

  public Rigidbody2D ropeBody;

  public Sprite armHoldingEmpty;
  public Sprite armHoldingTreasure;

  public SpriteRenderer holdingArm;

  public GameObject deathPrefab;
  public GameObject flameDeathPrefab;
  public GameObject ghostPrefab;

  public float delayBeforeRemoving = 3.0f;
  public float delayBeforeReleasingGhost = 0.25f;

  public GameObject bloodFountainPrefab;

  bool dead = false;

  bool _holdingTreasure = false;

  public bool holdingTreasure {
    get {
      return _holdingTreasure;
    }
    set {
      if (dead == true) {
        return;
      }

      _holdingTreasure = value;

      if (holdingArm != null) {
        if (_holdingTreasure) {
          holdingArm.sprite =
            armHoldingTreasure;
        } else {
          holdingArm.sprite =
            armHoldingEmpty;
        }
```

```
    }

  }
}

public enum DamageType {
  Slicing,
  Burning
}

public void ShowDamageEffect(DamageType type) {
  switch (type) {

  case DamageType.Burning:
    if (flameDeathPrefab != null) {
      Instantiate(
          flameDeathPrefab,cameraFollowTarget.position,
          cameraFollowTarget.rotation
      );
    }
    break;

  case DamageType.Slicing:
    if (deathPrefab != null) {
      Instantiate(
          deathPrefab,
          cameraFollowTarget.position,
          cameraFollowTarget.rotation
      );
    }
    break;
  }
}

public void DestroyGnome(DamageType type) {

  holdingTreasure = false;

  dead = true;

  // найти все дочерние объекты и произвольно
  // отсоединить их сочленения
  foreach (BodyPart part in
    GetComponentsInChildren<BodyPart>()) {

    switch (type) {

    case DamageType.Burning:
      // один шанс из трех получить ожог
      bool shouldBurn = Random.Range (0, 2) == 0;
      if (shouldBurn) {
        part.ApplyDamageSprite(type);
      }
```

```
      break;

    case DamageType.Slicing:
      // Отсечение части тела всегда влечет смену спрайта
      part.ApplyDamageSprite (type);

      break;
    }

    // один шанс из трех отделения от тела
    bool shouldDetach = Random.Range (0, 2) == 0;

    if (shouldDetach) {

      // Обеспечить удаление твердого тела и коллайдера
      // из этого объекта после достижения дна
      part.Detach ();

      // Если часть тела отделена и повреждение имеет
      // тип Slicing, добавить фонтан крови

      if (type == DamageType.Slicing) {

        if (part.bloodFountainOrigin != null &&
          bloodFountainPrefab != null) {

          // Присоединить фонтан крови
          // к отделившейся части тела
          GameObject fountain = Instantiate(
              bloodFountainPrefab,
              part.bloodFountainOrigin.position,
              part.bloodFountainOrigin.rotation
          ) as GameObject;

          fountain.transform.SetParent(
              this.cameraFollowTarget,
              false
          );
        }
      }

      // Отделить объект this
      var allJoints = part.GetComponentsInChildren<Joint2D>();
      foreach (Joint2D joint in allJoints) {
          Destroy (joint);
      }
    }
  }

  // Добавить компонент RemoveAfterDelay в объект this
  var remove = gameObject.AddComponent<RemoveAfterDelay>();
  remove.delay = delayBeforeRemoving;
```

```
    StartCoroutine(ReleaseGhost());
  }

  IEnumerator ReleaseGhost() {

    // Шаблон духа не определен? Выйти.
    if (ghostPrefab == null) {
      yield break;
    }

    // Ждать delayBeforeReleasingGhost секунд
    yield return new WaitForSeconds(delayBeforeReleasingGhost);

    // Добавить дух
    Instantiate(
        ghostPrefab,
        transform.position,
        Quaternion.identity
    );
  }

}
```

Добавив этот код, вы заметите пару ошибок компилятора, включая «The type or namespace name RemoveAfterDelay could not be found» («Не найдено имя RemoveAfterDelay типа или пространства имен»). Это ожидаемая ошибка, и мы решим ее чуть позже, добавив класс `RemoveAfterDelay`!

Сценарий `Gnome` прежде всего отвечает за поддержку важной информации о происходящем с гномиком и обработку ситуаций, когда гномик получает повреждения. Многие из свойств самим гномиком не используются, но используются диспетчером игры Game Manager (который мы скоро напишем) для настройки игры, когда нужно создать нового гномика.

Вот некоторые из особенностей сценария `Gnome`, которые заслуживают особого упоминания.

- Новое значение свойству `holdingTreasure` присваивается с помощью специализированного метода записи. Когда изменяется значение свойства `holdingTreasure`, требуется изменить визуальное представление гномика: если гномик взял сокровище (то есть свойство `holdingTreasure` получило значение `true`), спрайт с изображением пустой руки нужно заменить спрайтом с изображением руки, держащей сокровище. Аналогично, если свойство получает значение `false`, визуализатор должен отобразить спрайт пустой руки.
- Когда гномик получает повреждение, создается объект «эффекта повреждения». Специфика объекта зависит от конкретного вида повреждения: если это `Burning` (ожог), нужно создать эффект появления облачка дыма, а если это `Slicing` (порез), нужно воспроизвести брызги крови. Для этого мы предусмотрели метод `ShowDamageEffect`.

В этой книге мы реализуем эффект разбрызгивания крови. А эффект ожога мы оставим вам в качестве самостоятельного упражнения!

- Метод `DestroyGnome` отвечает за обработку повреждений, получаемых гномиком, и отделение компонентов `BodyPart`, на которые пришлись повреждения. Кроме того, для повреждения типа `Slicing` (порез) он создает эффект фонтана крови.

Метод создает также компонент `RemoveAfterDelay`, который мы добавим чуть ниже. Он удаляет гномика целиком из игры.

Наконец, метод запускает сопрограмму `ReleaseGhost`, которая ждет в течение определенного времени, а затем создает объект Ghost. (Мы оставляем вам создание шаблонного (prefab) объекта Ghost.)

4. *Добавьте компонент сценария* `BodyPart` *во все части тела гномика.* Для этого выберите все части тела (голову, ноги, руки и туловище) и добавьте в них компонент `BodyPart`.
5. *Добавьте контейнер для фонтанов крови.* Создайте пустой игровой объект и дайте ему имя Blood Fountains. Сделайте его потомком главного объекта Gnome (то есть не любой части тела, а самого родительского объекта).
6. *Добавьте источники фонтанов крови.* Создайте пять пустых игровых объектов и сделайте их потомками объекта Blood Fountains.

   Дайте им имена, соответствующие частям тела: Head, Leg Rope, Leg Dangle, Arm Holding, Arm Loose.

   Расположите их в местах, откуда, по вашему мнению, из разных частей тела должны бить фонтаны крови (например, переместите объект Head в область шеи); затем поверните каждый объект так, чтобы их оси Z (синяя стрелка) были направлены в ту сторону, куда должны бить фонтаны, как, например, на рис. 5.12, где выбран объект Head, а синяя стрелка направлена вниз. При такой ориентации фонтан крови будет бить вверх из шеи гномика.
7. *Свяжите источники фонтанов крови с соответствующими частями тела.* Каждый источник фонтана крови перетащите в поле Blood Fountain Origin соответствующей части тела. Например, перетащите игровой объект, соответствующий источнику фонтана крови Head, в часть тела Head (рис. 5.13). Обратите внимание, что в объекте Body нет ни одного источника — туловище не является отделяемой частью тела. Не нужно перетаскивать саму часть тела в поле! Перетаскивайте только что созданные игровые объекты.

   Спустя какое-то время туловище гномика должно исчезнуть. С этой целью создадим сценарий, удаляющий объект с задержкой. Это тоже пригодится в основной игре — огненные шары, обжигающие гномика, должны исчезать, а спустя какое-то время точно так же должен исчезать дух, появляющийся после гибели.

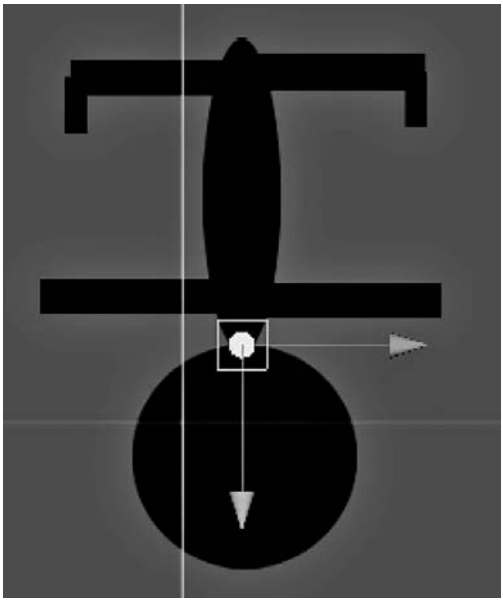

Рис. 5.12. Местоположение и ориентация фонтана крови Head

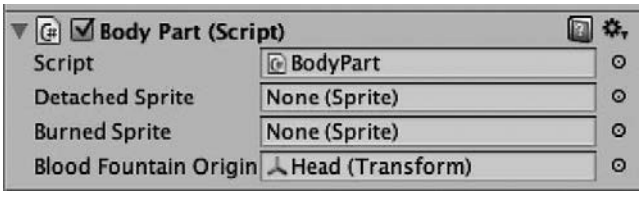

Рис. 5.13. Подключение объекта источника фонтана крови Head

1. *Создайте сценарий* `RemoveAfterDelay`. Создайте новый сценарий на C# в файле с именем *RemoveAfterDelay.cs*. Добавьте в него следующий код:

```
// Удаляет объект с заданной задержкой.
public class RemoveAfterDelay : MonoBehaviour {

  // Задержка в секундах перед удалением.
  public float delay = 1.0f;

  void Start () {
    // Запустить сопрограмму 'Remove'.
    StartCoroutine("Remove");
  }

  IEnumerator Remove() {
    // Ждать 'delay' секунд и затем уничтожить объект
```

```
        // gameObject, присоединенный к объекту this.
        yield return new WaitForSeconds(delay);
        Destroy (gameObject);

        // Нельзя использовать вызов Destroy(this) - он уничтожит сам
        // объект сценария RemoveAfterDelay.
    }
}
```

После добавления этого кода исчезнут ошибки компилятора, упоминавшиеся выше, — для компиляции класса `Gnome` необходим класс `RemoveAfterDelay`.

Класс `RemoveAfterDelay` очень прост: когда компонент появляется в сцене, он запускает сопрограмму, которая ждет заданный интервал времени, а затем удаляет объект.

2. *Присоедините компонент Gnome к гномику.* Настройте его так:
   - В поле Camera Follow Target перетащите туловище гномика.
   - В поле Rope Body — объект Leg Rope.
   - В поле Arm Holding Empty — спрайт Prototype Arm Holding.
   - В поле Arm Holding Treasure — спрайт Prototype Arm Holding with Gold.
   - В поле Holding Arm — часть тела Arm Holding.

В результате настройки сценария должны выглядеть так, как показано на рис. 5.14.

Эти свойства используются диспетчером игры Game Manager, который мы добавим чуть ниже, и обеспечивают следование камеры за правильным объектом и соединение веревки с правильной частью тела.

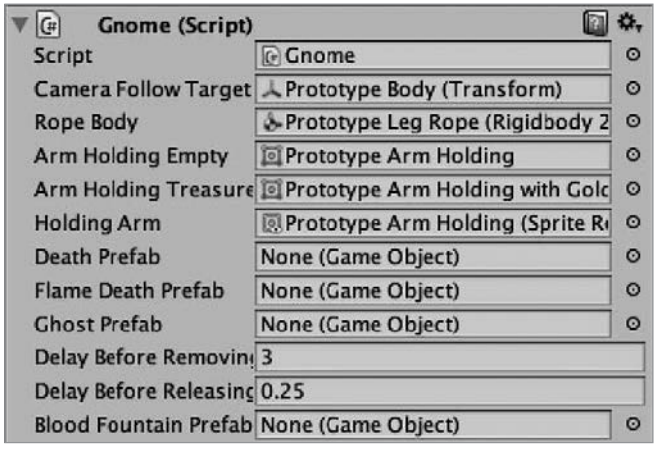

Рис. 5.14. Настройки компонента Gnome

Подготовка диспетчера игры

Диспетчер игры — это объект, отвечающий за управление игрой в целом. Он создает гномика в начале игры, обрабатывает события касания гномиком важных объектов, таких как ловушки, сокровище или событие выхода из уровня, и имеет дело со всем, что существует дольше отдельно взятого гномика.

В частности, диспетчер игры должен:

1. В момент запуска или перезапуска игры:
   a) создать экземпляр гномика;
   b) удалить старого гномика, если необходимо;
   c) поместить его в начало уровня;
   d) присоединить к нему веревку;
   e) настроить следование камеры за ним;
   f) переустановить все объекты, требующие переустановки, например сокровище.
2. Когда гномик касается сокровища:

   сообщить гномику, что он схватил сокровище, изменив его свойство `holdingTreasure`.
3. Когда гномик касается ловушки:
   a) отобразить эффект действия ловушки вызовом `ShowDamageEffect`;
   b) убить гномика вызовом `DestroyGnome`;
   c) сбросить игру в исходное состояние.
4. Когда гномик касается выхода:

   если гномик удерживает сокровище, показать экран окончания игры.

Перед тем как приступить к реализации диспетчера игры, мы должны добавить класс, от которого он зависит: класс `Resettable`.

Нам нужен универсальный способ запуска кода, когда потребуется сбросить игру в исходное состояние. Один из таких способов основан на событиях Unity — мы создадим сценарий *Resettable.cs* с полем Unity Event, который можно будет подключить к любому объекту, требующему переустановки в исходное состояние. Когда потребуется сбросить игру, диспетчер отыщет все объекты с компонентом `Resettable` и вызовет Unity Event.

При таком подходе можно настраивать отдельные объекты так, что они сами будут приводить себя в исходное состояние, без необходимости писать код для каждого из них. Например, объект сокровища Treasure, который мы добавим позже, должен изменить свой спрайт, чтобы показать, что его больше нет на дне колодца; мы добавим к нему объект `Resettable`, который восстановит спрайт, вернув исходное изображение сокровища на место.

Создайте сценарий `Resettable`. Добавьте новый сценарий на C# с именем *Resettable.cs* и поместите в него следующий код:

```
using UnityEngine.Events;

// Содержит поле UnityEvent, которое используется для
// установки объекта this в исходное состояние.
public class Resettable : MonoBehaviour {

  // В редакторе подключите это событие к методам, которые должны
  // вызываться в момент сброса игры.
  public UnityEvent onReset;

  // Вызывается диспетчером игры GameManager в момент сброса игры.
  public void Reset() {
    // Породить событие, которое вызовет все
    // подключенные методы.
    onReset.Invoke();
  }
}
```

Класс `Resettable` очень прост. Он содержит единственное свойство `UnityEvent`, которое позволяет добавлять вызовы методов и изменяемые свойства в инспекторе. Вызов метода `Reset` породит событие, в результате которого будут вызваны все подключенные методы и изменены указанные свойства.

Теперь можно создать диспетчер игры — объект `Game Manager`.

1. *Создайте объект* `Game Manager`. Создайте новый пустой игровой объект и дайте ему имя Game Manager.
2. *Создайте и добавьте в него код* `GameManager`. Добавьте новый сценарий на C# в файле с именем *GameManager.cs* и поместите в него следующий код:

```
// Управляет состоянием игры.
public class GameManager : Singleton<GameManager> {

// Местоположение, где должен появиться гномик.
public GameObject startingPoint;

// Объект веревки, опускающей и поднимающей гномика.
public Rope rope;

// Сценарий, управляющий камерой, которая должна следовать за гномиком
public CameraFollow cameraFollow;

// 'текущий' гномик (в противоположность всем погибшим)
Gnome currentGnome;

// Объект-шаблон для создания нового гномика
public GameObject gnomePrefab;

// Компонент пользовательского интерфейса с кнопками
// 'перезапустить и 'продолжить'
public RectTransform mainMenu;
```

```
// Компонент пользовательского интерфейса с кнопками
// 'вверх', 'вниз' и 'меню'
public RectTransform gameplayMenu;

// Компонент пользовательского интерфейса с экраном
// 'вы выиграли!'
public RectTransform gameOverMenu;

// Значение true в этом свойстве требует игнорировать любые повреждения
// (но показывать визуальные эффекты).
// Объявление 'get; set;' превращает поле в свойство, что
// необходимо для отображения в списке методов в инспекторе
// для Unity Events
public bool gnomeInvincible { get; set; }

// Задержка перед созданием нового гномика после гибели
public float delayAfterDeath = 1.0f;

// Звук, проигрываемый в случае гибели гномика
public AudioClip gnomeDiedSound;

// Звук, проигрываемый в случае победы в игре
public AudioClip gameOverSound;

void Start() {
  // В момент запуска игры вызвать Reset, чтобы
  // подготовить гномика.
  Reset ();
}

// Сбрасывает игру в исходное состояние.
public void Reset() {

  // Выключает меню, включает интерфейс игры
  if (gameOverMenu)
    gameOverMenu.gameObject.SetActive(false);

  if (mainMenu)
    mainMenu.gameObject.SetActive(false);

  if (gameplayMenu)
    gameplayMenu.gameObject.SetActive(true);

  // Найти все компоненты Resettable и сбросить их в исходное состояние
  var resetObjects = FindObjectsOfType<Resettable>();

  foreach (Resettable r in resetObjects) {
      r.Reset();
  }

  // Создать нового гномика
  CreateNewGnome();

  // Прервать паузу в игре
  Time.timeScale = 1.0f;
}

void CreateNewGnome() {
```

```
  // Удалить текущего гномика, если имеется
  RemoveGnome();

  // Создать новый объект Gnome и назначить его текущим
  GameObject newGnome =
    (GameObject)Instantiate(gnomePrefab,
      startingPoint.transform.position,
      Quaternion.identity);

  currentGnome = newGnome.GetComponent<Gnome>();

  // Показать веревку
  rope.gameObject.SetActive(true);

  // Привязать конец веревки к заданному
  // твердому телу в объекте Gnome (например, к его ноге)
  rope.connectedObject = currentGnome.ropeBody;

  // Установить длину веревки в начальное значение
  rope.ResetLength();

  // Сообщить объекту cameraFollow, что он должен
  // начать следить за новым объектом Gnome
  cameraFollow.target = currentGnome.cameraFollowTarget;

}

void RemoveGnome() {

  // Ничего не делать, если гномик неуязвим
  if (gnomeInvincible)
    return;

  // Скрыть веревку
  rope.gameObject.SetActive(false);

  // Запретить камере следовать за гномиком
  cameraFollow.target = null;

  // Если текущий гномик существует, исключить его из игры
  if (currentGnome != null) {

    // Этот гномик больше не удерживает сокровище
    currentGnome.holdingTreasure = false;

    // Пометить объект как исключенный из игры
    // (чтобы коллайдеры перестали сообщать о столкновениях с ним)
    currentGnome.gameObject.tag = "Untagged";

    // Найти все объекты с тегом "Player" и удалить этот тег
    foreach (Transform child in
      currentGnome.transform) {
        child.gameObject.tag = "Untagged";
    }

    // Установить признак отсутствия текущего гномика
    currentGnome = null;
  }
```

```
}

// Убивает гномика.
void KillGnome(Gnome.DamageType damageType) {

  // Если задан источник звука, проиграть звук "гибель гномика"
  var audio = GetComponent<AudioSource>();
  if (audio) {
      audio.PlayOneShot(this.gnomeDiedSound);
  }

  // Показать эффект действия ловушки
  currentGnome.ShowDamageEffect(damageType);

  // Если гномик уязвим, сбросить игру
  // и исключить гномика из игры.
  if (gnomeInvincible == false) {

    // Сообщить гномику, что он погиб
    currentGnome.DestroyGnome(damageType);

    // Удалить гномика
    RemoveGnome();

    // Сбросить игру
    StartCoroutine(ResetAfterDelay());

 }
}

// Вызывается в момент гибели гномика.
IEnumerator ResetAfterDelay() {

  // Ждать delayAfterDeath секунд, затем вызвать Reset
  yield return new WaitForSeconds(delayAfterDeath);
  Reset();
}

// Вызывается, когда гномик касается ловушки
// с ножами
public void TrapTouched() {
  KillGnome(Gnome.DamageType.Slicing);
}

// Вызывается, когда гномик касается огненной ловушки
public void FireTrapTouched() {
  KillGnome(Gnome.DamageType.Burning);
}

// вызывается, когда гномик касается сокровища.
public void TreasureCollected() {
  // Сообщить текущему гномику, что он взял сокровище.
  currentGnome.holdingTreasure = true;
}

// Вызывается, когда гномик касается выхода.
public void ExitReached() {
```

```
    // Завершить игру, если есть гномик и он держит сокровище!
    if (currentGnome != null &&
      currentGnome.holdingTreasure == true) {

      // Если задан источник звука, проиграть звук
      // "игра завершена"
      var audio = GetComponent<AudioSource>();
      if (audio) {
        audio.PlayOneShot(this.gameOverSound);
      }

      // Приостановить игру
      Time.timeScale = 0.0f;

      // Выключить меню завершения игры и включить экран
      // "игра завершена"!
      if (gameOverMenu) {
        gameOverMenu.gameObject.SetActive(true);
      }

      if (gameplayMenu) {
        gameplayMenu.gameObject.SetActive(false);
      }

    }
  }

  // Вызывается в ответ на касание кнопок Menu и Resume Game.
  public void SetPaused(bool paused) {

    // Если игра на паузе, остановить время и включить меню
    // (и выключить интерфейс игры)
    if (paused) {
      Time.timeScale = 0.0f;
      mainMenu.gameObject.SetActive(true);
      gameplayMenu.gameObject.SetActive(false);
    } else {
      // Если игра не на паузе, возобновить ход времени и
      // выключить меню (и включить интерфейс игры)
      Time.timeScale = 1.0f;
      mainMenu.gameObject.SetActive(false);
      gameplayMenu.gameObject.SetActive(true);
    }
  }

  // Вызывается в ответ на касание кнопки Restart.
  public void RestartGame() {

    // Немедленно удалить гномика (минуя этап гибели)
    Destroy(currentGnome.gameObject);
    currentGnome = null;

    // Сбросить игру в исходное состояние, чтобы создать нового гномика.
    Reset();
  }

}
```

Главная задача диспетчера игры Game Manager — создание новых гномиков и подключение других систем к правильным объектам. Когда появляется новый гномик, к его ноге нужно привязать веревку Rope и подсказать сценарию `CameraFollow`, за каким гномиком он должен следить. Кроме того, диспетчер игры отвечает за отображение меню и обработку нажатий на кнопки в меню. (Меню мы реализуем позже.)

Фрагмент кода получился большим, поэтому давайте разобьем его на части и рассмотрим их по отдельности.

Настройка и сброс игры

Метод `Start`, который вызывается, когда объект появляется в первый раз, сразу же вызывает метод `Reset`. Задача метода `Reset` — сбросить игру в исходное состояние, поэтому его вызов из метода `Start` является простейшим способом объединить «начальную настройку» и «сброс игры в исходное состояние».

Сам метод `Reset` обеспечивает настройку и отображение элементов меню, которые мы добавим позже. Всем компонентам `Resettable`, присутствующим в сцене, передается сигнал сброса, а затем вызовом метода `CreateNewGnome` создается новый гномик. В конце игра снимается с паузы (если она была поставлена на паузу).

```
void Start() {

  // В момент запуска игры вызвать Reset, чтобы
  // подготовить гномика.
  Reset ();
}

// Сбрасывает игру в исходное состояние.
public void Reset() {

  // Выключает меню, включает интерфейс игры
  if (gameOverMenu)
    gameOverMenu.gameObject.SetActive(false);

  if (mainMenu)
    mainMenu.gameObject.SetActive(false);

  if (gameplayMenu)
    gameplayMenu.gameObject.SetActive(true);

  // Найти все компоненты Resettable и сбросить их в исходное состояние
  var resetObjects = FindObjectsOfType<Resettable>();

  foreach (Resettable r in resetObjects) {
      r.Reset();
  }

  // Создать нового гномика
  CreateNewGnome();

  // Прервать паузу в игре
  Time.timeScale = 1.0f;
}
```

Создание нового гномика

Метод `CreateNewGnome` заменяет текущего гномика вновь созданным, сначала удаляя текущего гномика, если тот существует, а затем создавая нового; он также активизирует веревку и привязывает ее конец к ноге гномика (компоненту `ropeBody`). Длина веревки устанавливается в исходное значение, и, наконец, выполняется настройка камеры для сопровождения вновь созданного гномика:

```
void CreateNewGnome() {

  // Удалить текущего гномика, если имеется
  RemoveGnome();

  // Создать новый объект Gnome и назначить его текущим
  GameObject newGnome =
    (GameObject)Instantiate(gnomePrefab,
      startingPoint.transform.position,
      Quaternion.identity);

  currentGnome = newGnome.GetComponent<Gnome>();

  // Показать веревку
  rope.gameObject.SetActive(true);

  // Привязать конец веревки к заданному
  // твердому телу в объекте Gnome (например, к его ноге)
  rope.connectedObject = currentGnome.ropeBody;

  // Установить длину веревки в начальное значение
  rope.ResetLength();

  // Сообщить объекту cameraFollow, что он должен
  // начать следить за новым объектом Gnome
  cameraFollow.target = currentGnome.cameraFollowTarget;

}
```

Удаление старого гномика

Возможны две ситуации, когда необходимо отцепить гномика от веревки: когда гномик погибает и когда игрок решает начать игру сначала. В обоих случаях гномик отцепляется от веревки и исключается из игры. Он продолжает оставаться на уровне, но касания его ловушек больше не интерпретируются как сигнал начать уровень заново.

Чтобы удалить активного гномика, мы выключаем веревку и запрещаем камере сопровождать текущего гномика. Затем отмечаем гномика как не удерживающего сокровище, в результате чего возвращаются исходные спрайты, и присваиваем объекту тег «Untagged». Это делается по той простой причине, что ловушки, которые мы добавим чуть ниже, определяют касания с объектами, имеющими тег «Player»; если бы старый гномик все еще был отмечен тегом «Player», в ответ на

его касание любой ловушки диспетчер игры Game Manager перезапускал бы уровень.

```
void RemoveGnome() {

  // Ничего не делать, если гномик неуязвим
  if (gnomeInvincible)
    return;

  // Скрыть веревку
  rope.gameObject.SetActive(false);

  // Запретить камере следовать за гномиком
  cameraFollow.target = null;

  // Если текущий гномик существует, исключить его из игры
  if (currentGnome != null) {

    // Этот гномик больше не удерживает сокровище
    currentGnome.holdingTreasure = false;

    // Пометить объект как исключенный из игры
    // (чтобы коллайдеры перестали сообщать о столкновениях с ним)
    currentGnome.gameObject.tag = "Untagged";

    // Найти все объекты с тегом "Player" и удалить этот тег
    foreach (Transform child in
      currentGnome.transform) {
        child.gameObject.tag = "Untagged";
    }

    // Установить признак отсутствия текущего гномика
    currentGnome = null;
  }
}
```

Гибель гномика

Когда гномик погибает, мы должны воспроизвести соответствующие игровые эффекты. К ним относятся звуки и визуальные эффекты; кроме того, если гномик в данный момент уязвим, мы должны сообщить гномику, что он погиб, удалить гномика и с некоторой задержкой сбросить игру в исходное состояние. Все эти операции выполняет следующий код:

```
void KillGnome(Gnome.DamageType damageType) {

  // Если задан источник звука, проиграть звук "гибель гномика"
  var audio = GetComponent<AudioSource>();

  if (audio) {
      audio.PlayOneShot(this.gnomeDiedSound);
  }
```

```
    // Показать эффект действия ловушки
    currentGnome.ShowDamageEffect(damageType);

    // Если гномик уязвим, сбросить игру и
    // исключить гномика из игры.
    if (gnomeInvincible == false) {

      // Сообщить гномику, что он погиб
      currentGnome.DestroyGnome(damageType);

      // Удалить гномика
      RemoveGnome();

      // Сбросить игру
      StartCoroutine(ResetAfterDelay());

    }
  }
```

Сброс игры

Когда гномик погибает, необходимо, чтобы камера задержалась в точке гибели. Это позволит игроку увидеть, как гномик падает вниз, прежде чем камера вернется в верхнюю часть экрана.

Для этого используется сопрограмма, ожидающая некоторое количество секунд (хранится в `delayAfterDeath`) и затем вызывающая метод `Reset` для сброса игры в исходное состояние:

```
// Вызывается в момент гибели гномика.
IEnumerator ResetAfterDelay() {

  // Ждать delayAfterDeath секунд, затем вызвать Reset
  yield return new WaitForSeconds(delayAfterDeath);
  Reset();

}
```

Обработка касаний

Следующие три метода реализуют реакцию на касание гномиком разных объектов. Если гномик касается ловушки с ножами, вызывается `KillGnome` и воспроизводится эффект нанесения порезов. Если гномик касается огненной ловушки, воспроизводится эффект нанесения ожогов. Наконец, если гномик касается сокровища, воспроизводится эффект его захвата. Все это реализует следующий код:

```
// Вызывается, когда гномик касается ловушки
// с ножами
public void TrapTouched() {
  KillGnome(Gnome.DamageType.Slicing);
}
```

```
// Вызывается, когда гномик касается огненной ловушки
public void FireTrapTouched() {
  KillGnome(Gnome.DamageType.Burning);
}

// вызывается, когда гномик касается сокровища.
public void TreasureCollected() {
  // Сообщить текущему гномику, что он взял сокровище.
  currentGnome.holdingTreasure = true;
}
```

Достижение выхода

Когда гномик касается выхода на вершине уровня, нужно проверить, держит ли текущий гномик сокровище. Если держит, значит, игрок выиграл! В результате воспроизводится звук «игра завершена» (мы настроим его в разделе «Аудио» в главе 8), игра ставится на паузу установкой масштаба времени в ноль, и отображается экран Game Over (включающий кнопку сброса игры в исходное состояние):

```
// Вызывается, когда гномик касается выхода.
public void ExitReached() {
  // Завершить игру, если есть гномик и он держит сокровище!
  if (currentGnome != null &&
    currentGnome.holdingTreasure == true) {

    // Если задан источник звука, проиграть звук
    // "игра завершена"
    var audio = GetComponent<AudioSource>();
    if (audio) {
      audio.PlayOneShot(this.gameOverSound);
    }

    // Приостановить игру
    Time.timeScale = 0.0f;

    // Выключить меню завершения игры и включить экран
    // "игра завершена"!
    if (gameOverMenu) {
      gameOverMenu.gameObject.SetActive(true);
    }

    if (gameplayMenu) {
      gameplayMenu.gameObject.SetActive(false);
    }
  }
}
```

Приостановка и возобновление игры

Приостановка игры заключается в выполнении трех действий: во-первых, остановка течения времени путем установки масштаба времени в ноль. Далее, отобра-

жение главного меню и сокрытие игрового интерфейса. Чтобы возобновить игру, достаточно выполнить обратные действия — запустить ход времени, скрыть меню и отобразить игровой интерфейс:

```
// Вызывается в ответ на касание кнопок Menu и Resume Game.
public void SetPaused(bool paused) {

  // Если игра на паузе, остановить время и включить меню
  // (и выключить интерфейс игры)
  if (paused) {
    Time.timeScale = 0.0f;
    mainMenu.gameObject.SetActive(true);
    gameplayMenu.gameObject.SetActive(false);
  } else {
    // Если игра не на паузе, возобновить ход времени и
    // выключить меню (и активизировать игровое поле)
    Time.timeScale = 1.0f;
    mainMenu.gameObject.SetActive(false);
    gameplayMenu.gameObject.SetActive(true);
  }
}
```

Обработка кнопки сброса

Метод `RestartGame` вызывается, когда пользователь касается определенных кнопок в пользовательском интерфейсе. Он немедленно перезапускает игру:

```
// Вызывается в ответ на касание кнопки Restart.
public void RestartGame() {

  // Немедленно удалить гномика (минуя этап гибели)
  Destroy(currentGnome.gameObject);
  currentGnome = null;

  // Сбросить игру в исходное состояние, чтобы создать нового гномика.
  Reset();
}
```

Подготовка сцены

Теперь, написав код, его можно подключить к сцене.

1. *Создайте начальную точку.* Это объект, который диспетчер игры Game Manager будет использовать для позиционирования вновь создаваемых гномиков. Создайте новый игровой объект и дайте ему имя Start Point. Расположите его в том месте, где, по вашему мнению, должен появляться новый гномик (рядом с объектом Rope; как показано на рис. 5.15), и измените его пиктограмму, чтобы он выглядел как желтая капсула (так же, как мы настраивали пиктограмму для объекта Rope).

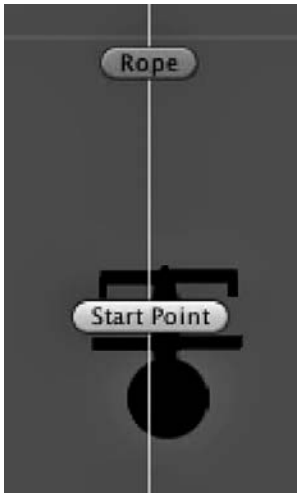

Рис. 5.15. Выбор местоположения для начальной точки

2. *Преобразуйте гномика в шаблонный объект (prefab)*. Теперь гномики будут создаваться диспетчером игры Game Manager, то есть гномика, присутствующего сейчас в сцене, нужно убрать. Но перед этим его следует преобразовать в шаблонный объект, чтобы диспетчер Game Manager смог создавать его экземпляры во время выполнения.

   Перетащите гномика в папку *Gnome* в обозревателе проекта. В результате будет создан новый шаблонный объект (рис. 5.16) — полная копия исходного объекта Gnome.

   После создания шаблона надобность объекта в сцене отпала, поэтому удалите гномика из сцены.

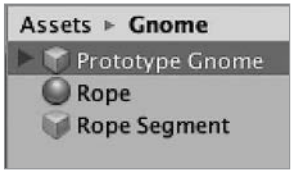

Рис. 5.16. Гномик как шаблонный объект в папке Gnome

3. *Настройте диспетчер игры Game Manager*. В диспетчере есть несколько соединений, которые мы должны настроить:
   - свяжите поле Starting Point с только что созданным объектом Start Point;
   - свяжите поле Rope с объектом Rope;
   - свяжите поле Camera Follow с объектом Main Camera;

- Свяжите поле Gnome Prefab с только что созданным шаблонным объектом Gnome.

В результате настройки диспетчера игры в инспекторе должны выглядеть так, как показано на рис. 5.17.

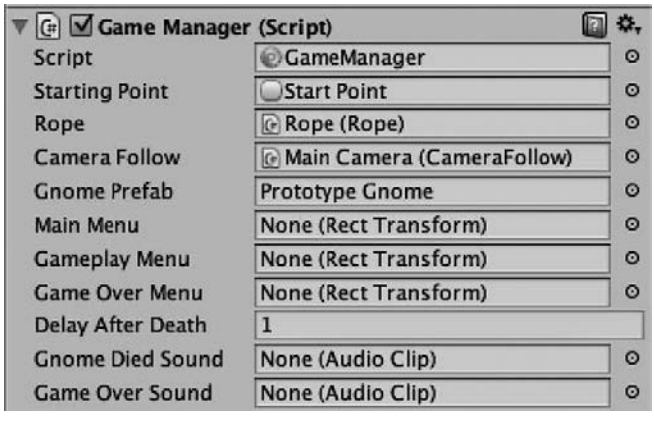

Рис. 5.17. Настройки диспетчера игры Game Manager

4. *Протестируйте игру.* Гномик будет появляться в начальной точке и соединяться с веревкой. Кроме того, по мере удлинения или укорочения веревки камера будет следовать за туловищем гномика. На данном этапе вы не сможете протестировать захват сокровища, но не волнуйтесь — мы займемся этим очень скоро!

В заключение

Теперь, закончив создание диспетчера игры Game Manager, мы готовы перейти к фактической реализации игрового процесса. В главе 6 мы начнем добавлять элементы, с которыми взаимодействует гномик: сокровище и ловушки.

6 Реализация игрового процесса с ловушками и целями

Теперь, когда основы игрового процесса настроены, можно начинать добавлять в игру такие элементы, как ловушки и сокровище. С этого момента дальнейшее развитие игры в основном будет касаться ее дизайна.

Простые ловушки

Большая часть этой игры заключается в обработке касаний разных объектов — ловушек, сокровища, точки выхода и т. д. Учитывая, насколько важно определение момента касания конкретных объектов, создадим общий сценарий, генерирующий событие Unity Event, когда любой объект с тегом «Player» касается их. Это событие будет затем по-разному настраиваться для различных объектов: ловушки будут сообщать диспетчеру игры Game Manager, что гномик получил повреждение, сокровище будет сообщать, что гномик подобрал сокровище, а точка выхода — что гномик достиг выхода.

Теперь создайте новый сценарий на C# в файле с именем *SignalOnTouch.cs* и добавьте в него следующий код:

```
using UnityEngine.Events;

// Вызывает UnityEvent, когда объект с тегом "Player" касается
// данного объекта.
[RequireComponent (typeof(Collider2D))]
public class SignalOnTouch : MonoBehaviour {

  // UnityEvent для выполнения в ответ на касание.
  // Вызываемый метод подключается в редакторе.
  public UnityEvent onTouch;

  // Если имеется значение true, при касании проигрывается звук из AudioSource.
  public bool playAudioOnTouch = true;
  // Когда обнаруживается вход в область действия триггера,
  // вызывается SendSignal.
  void OnTriggerEnter2D(Collider2D collider) {
    SendSignal (collider.gameObject);
  }
```

```
    // Когда обнаруживается касание с данным объектом,
    // вызывается SendSignal.
    void OnCollisionEnter2D(Collision2D collision) {
      SendSignal (collision.gameObject);
    }

    // Проверяет наличие тега "Player" у данного объекта и
    // вызывает UnityEvent, если такой тег имеется.
    void SendSignal(GameObject objectThatHit) {

      // Объект отмечен тегом "Player"?
      if (objectThatHit.CompareTag("Player")) {

        // Если требуется воспроизвести звук, попытаться сделать это
        if (playAudioOnTouch) {
          var audio = GetComponent<AudioSource>();

          // Если имеется аудиокомпонент
          // и родитель этого компонента активен,
          // воспроизвести звук
          if (audio &&
            audio.gameObject.activeInHierarchy)
            audio.Play();
        }

        // Вызвать событие
        onTouch.Invoke();
      }
    }

}
```

Основой класса `SignalOnTouch` является метод `SendSignal`, который вызывается методами `OnCollisionEnter2D` и `OnTriggerEnter2D`. Последние два метода вызывает движок Unity, когда объект касается коллайдера или когда объект входит в область действия триггера. Метод `SendSignal` проверяет тег объекта и, если он хранит строку «Player», генерирует событие Unity.

Теперь, имея класс `SignalOnTouch`, можно добавить первую ловушку.

1. *Импортируйте спрайты объектов.* Импортируйте содержимое папки *Sprites/Objects* в проект.
2. *Добавьте коричневые шипы.* Найдите спрайт SpikesBrown и перетащите его в сцену.
3. *Настройте объект с шипами.* Добавьте в объект с шипами компоненты PolygonCollider2D и SignalOnTouch.

   Добавьте новую функцию в событие `SignalOnTouch`. Перетащите диспетчер игры Game Manager в поле объекта и выберите функцию `GameManager.TrapTouched`, как показано на рис. 6.1.

Рис. 6.1. Настройки объекта с шипами

4. *Преобразуйте объект с шипами в шаблон.* Перетащите объект SpikesBrown из панели Hierarchy (Иерархия) в папку *Level*. В результате будет создан шаблон, с помощью которого можно будет создать несколько копий объекта.
5. *Протестируйте.* Запустите игру. Сделайте так, чтобы гномик попал на шипы. Этот гномик должен упасть и появиться заново!

Сокровище и выход

После успешного добавления ловушки, убивающей гномика, самое время добавить возможность выиграть в игре. Для этого добавим два новых элемента: сокровище и точку выхода.

Сокровище — это спрайт на дне колодца, который обнаруживает касание гномика и посылает сигнал диспетчеру игры Game Manager. Когда это происходит, диспетчер игры сообщает гномику, что тот ухватил сокровище, после чего спрайт с изображением пустой руки гномика заменяется спрайтом с изображением руки, удерживающей сокровище.

Точка выхода — это еще один спрайт, находящийся в верхней части колодца. Подобно сокровищу, он обнаруживает касание гномика и извещает об этом диспетчер игры. Если в этот момент гномик держит сокровище, игроку присуждается победа в игре.

Основную работу в этих двух объектах выполняет компонент SignalOnTouch — когда гномик достигает точки выхода, должен вызываться метод `ExitReached` диспетчера игры, а когда гномик касается сокровища, должен вызываться метод `TreasureCollected`.

Начнем с создания точки выхода, а потом добавим сокровище.

Создание точки выхода

Прежде всего, импортируем спрайты.

1. *Импортируйте спрайты Level Background.* Скопируйте папку *Sprites/Background* из загруженного пакета с ресурсами в папку *Sprites* проекта.

2. *Добавьте спрайт Top.* Поместите его чуть ниже объекта Rope. Этот спрайт будет служить точкой выхода.
3. *Настройте спрайт.* Добавьте в спрайт компонент Box Collider 2D и установите флажок Is Trigger. Щелкните по кнопке Edit Collider (Редактировать коллайдер) и измените размеры коллайдера, чтобы он был коротким и широким, как показано на рис. 6.2.

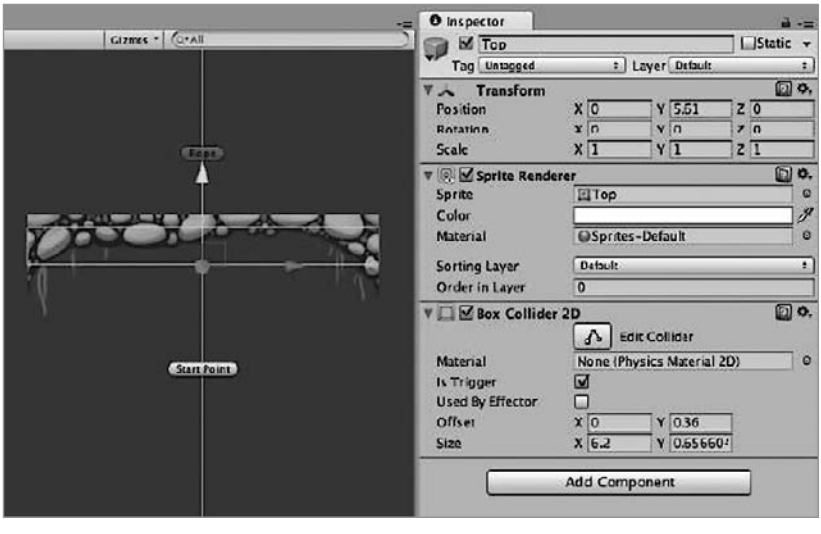

Рис. 6.2. Поместите коллайдер в верхнюю часть уровня и сделайте его широким и коротким

4. *Настройте отправку сигнала диспетчеру игры в момент касания спрайта.* Добавьте в спрайт компонент SignalOnTouch. Добавьте элемент в список событий компонента и соедините его с диспетчером игры Game Manager. Выберите функцию `GameManager.ExitReached`. Теперь касание гномика точки выхода будет вызывать метод `ExitReached` диспетчера игры Game Manager.

Теперь добавим сокровище.

Сокровище работает так: по умолчанию объект Treasure отображает спрайт с изображением сокровища. Когда гномик касается его, вызывается метод `TreasureCollected` диспетчера игры Game Manager и на месте сокровища отображается другой спрайт, показывающий, что сокровище подобрано. Когда гномик погибает, объект Treasure возвращается в исходное состояние и вновь отображает спрайт с изображением сокровища.

Так как смена спрайтов в игре будет выполняться довольно часто — вы убедитесь в этом, когда мы займемся улучшением графики, — имеет смысл создать универсальный класс смены спрайтов и задействовать его в объекте сокровища.

Создайте новый сценарий на C# с именем *SpriteSwapper.cs.* Добавьте в него следующий код:

```
// Меняет один спрайт на другой. Например, при переключении сокровища
// из состояния 'сокровище есть' в состояние 'сокровища нет'.
public class SpriteSwapper : MonoBehaviour {
  // Спрайт, который требуется отобразить.
  public Sprite spriteToUse;

  // Визуализатор спрайта, который должен использоваться
  // для отображения нового спрайта.
  public SpriteRenderer spriteRenderer;

  // Исходный спрайт. Используется в вызове ResetSprite.
  private Sprite originalSprite;

  // Меняет спрайт.
  public void SwapSprite() {

    // Если требуемый спрайт отличается от текущего...
    if (spriteToUse != spriteRenderer.sprite) {

      // Сохранить предыдущий в originalSprite
      originalSprite = spriteRenderer.sprite;

      // Передать новый спрайт визуализатору.
      spriteRenderer.sprite = spriteToUse;
    }
  }

  // Возвращает прежний спрайт.
  public void ResetSprite() {

    // Если прежний спрайт был сохранен...
    if (originalSprite != null) {
      // ...передать его визуализатору.
      spriteRenderer.sprite = originalSprite;
    }
  }
}
```

Класс `SpriteSwapper` предназначен для двух операций: когда вызывается метод `SwapSprite`, визуализатору `SpriteRenderer`, подключенному к игровому объекту, передается другой спрайт для отображения. При этом исходный спрайт сохраняется в переменной. Когда вызывается метод `ResetSprite`, визуализатору передается исходный спрайт для отображения.

Теперь можно создать и настроить объект Treasure.

1. *Добавьте спрайт с изображением сокровища.* Найдите спрайт TreasurePresent и добавьте его в сцену. Разместите его поближе ко дну, но с таким расчетом, чтобы гномик мог достать его.
2. *Добавьте коллайдер для сокровища.* Выберите спрайт с изображением сокровища и добавьте в него компонент Box Collider 2D. Установите флажок на значении Is Trigger.

3. *Добавьте и настройте сценарий смены спрайта.* Добавьте компонент SpriteSwapper. Перетащите сам спрайт с изображением сокровища в поле Sprite Renderer этого компонента. Затем найдите спрайт TreasureAbsent и перетащите его в поле Sprite To Use компонента, выполняющего смену спрайтов.
4. *Добавьте и настройте компонент отправки сигнала в ответ на касание.* Добавьте компонент SignalOnTouch. Добавьте два элемента в список On Touch:
   - первый свяжите с объектом Game Manager и выберите метод `GameManager.TreasureCollected`;
   - второй свяжите со спрайтом сокровища (то есть с объектом, который сейчас настраиваете) и выберите метод `SpriteSwapper.SwapSprite`.
5. *Добавьте и настройте компонент Resettable.* Добавьте в объект компонент Resettable. Добавьте единственный элемент в список On Touch, выберите метод `SpriteSwapper.ResetSprite` и свяжите его с объектом Treasure.

В результате настройки в панели инспектора должны выглядеть так, как показано на рис. 6.3.

6. *Протестируйте игру.* Запустите игру и коснитесь сокровища. В этот момент сокровище исчезнет; если после этого гномик погибнет, сокровище вновь появится на своем месте после создания нового гномика.

Добавление фона

В настоящее время действие игры разворачивается на унылом синем фоне, заданном в Unity по умолчанию. В этом разделе мы добавим временный фон, который заменим фоновым спрайтом, когда приступим к улучшению графического оформления игры.

1. *Добавьте фоновый прямоугольник.* Откройте меню GameObject (Игровой объект) и выберите пункт 3D Object ▸ Quad (3D Объект ▸ Прямоугольник). Дайте новому объекту имя Background.
2. *Переместите фон на задний план.* Чтобы избежать ситуации, когда прямоугольный фон будет рисоваться поверх игровых спрайтов, переместите его на задний план, подальше от камеры. Для этого присвойте координате Z прямоугольника значение 10.

Несмотря на то что мы делаем двумерную игру, Unity не перестает быть трехмерным движком. Благодаря этому можно с успехом пользоваться тем, что одни объекты могут находиться «позади» других, как в нашем случае.

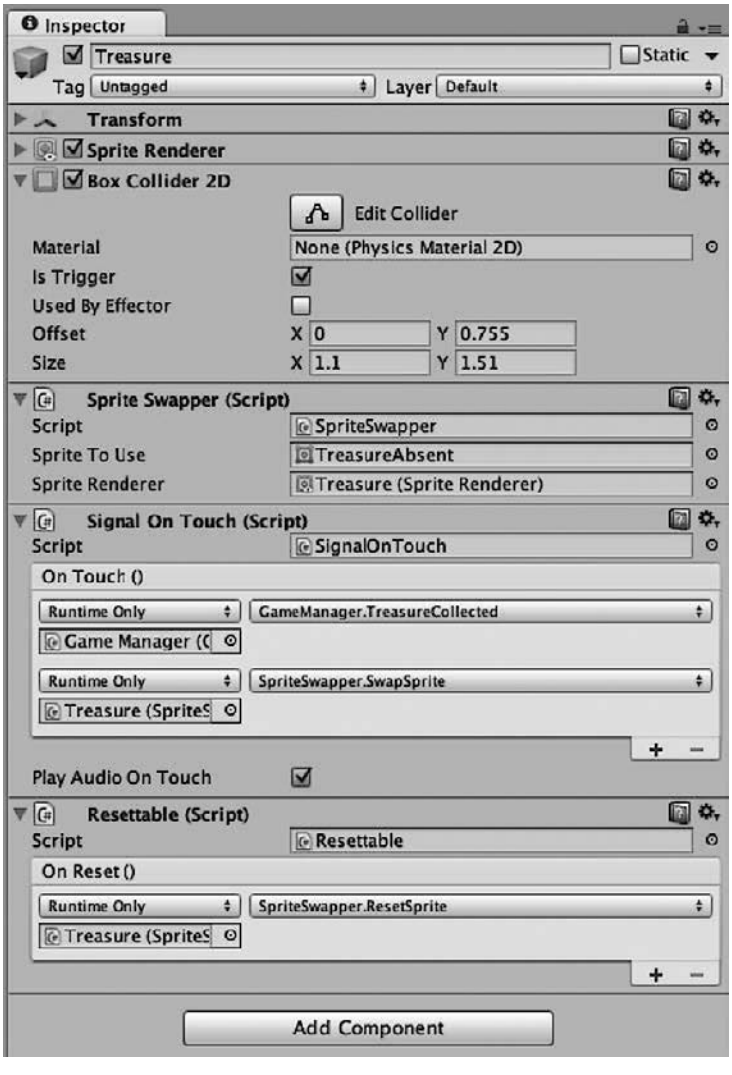

Рис. 6.3. Настройки объекта Treasure

3. *Установите ширину и высоту фонового прямоугольника.* Включите инструмент Rect (Прямоугольник), нажав клавишу T, а затем, используя маркеры, установите размеры прямоугольника. Верхняя граница фона должна быть на одном уровне со спрайтом в верхней части сцены, а нижняя — с сокровищем (рис. 6.4).

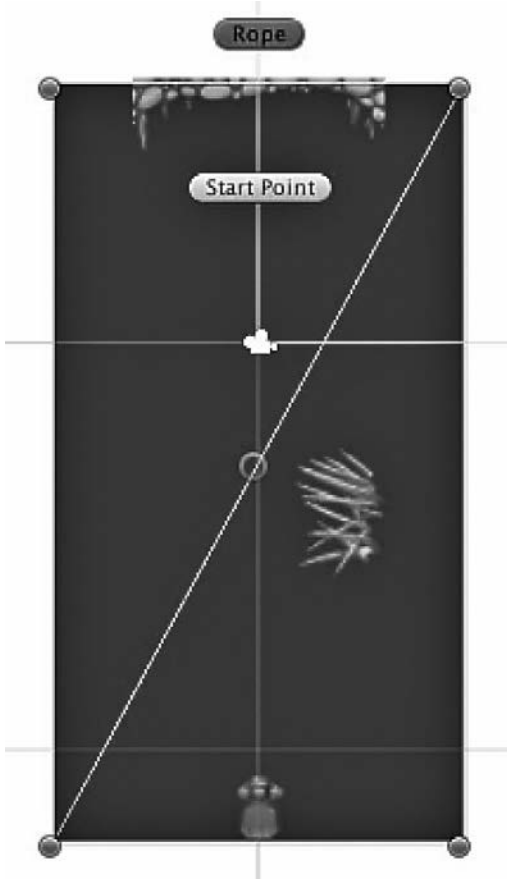

Рис. 6.4. Установка размеров фонового прямоугольника

4. *Протестируйте игру*. После запуска фон игрового поля должен окраситься в темно-серый цвет.

В заключение

Мы реализовали все основные функции поддержки игрового процесса и очень много добавили в него. На данном этапе в игровом процессе:

- имитируется поведение гномика и веревки с соблюдением законов физики;
- длиной веревки можно управлять с помощью кнопок на экране, появилась возможность опускать и поднимать гномика;
- камера настроена так, что следует за гномиком, благодаря чему тот постоянно остается в поле зрения;

- в ответ на наклоны телефона гномик отклоняется от вертикали, смещаясь влево-вправо;
- гномик может погибнуть, коснувшись ловушки, и может подобрать сокровище.

На рис. 6.5 приводится скриншот игры в ее текущем состоянии.

Рис. 6.5. Игра в конце этой главы

Несмотря на законченную функциональность, сейчас наша игра не отличается привлекательностью. Гномик изображен схематично, а уровни выглядят скудно. В главе 7 мы продолжим работу над игрой и улучшим ее визуальные элементы.

7 Доводка игры

В этой главе мы внесем много доработок в игру *Колодец с сокровищами*, и в результате она будет выглядеть так, как показано на рис. 7.1.

Рис. 7.1. Окончательный вид игры

Доводку игры мы будем вести в трех основных направлениях:

Визуальное оформление

Мы добавим новые спрайты для изображения гномика, улучшим вид фона и реализуем эффекты частиц для придания привлекательности игре.

Игровой процесс

Мы добавим новые виды ловушек, начальный экран, а также возможность делать гномика неуязвимым, что поможет протестировать игру.

Звуки

Мы также добавим в игру звуковые эффекты, воспроизводимые в ответ на действия пользователя.

Ресурсы, используемые в этой главе, можно найти в загружаемом пакете, доступном по адресу *https://www.secretlab.com.au/books/unity*.

Улучшение изображения гномика

Первое, что мы улучшим в игре, — изображение гномика. Мы заменим схематические спрайты с частями тела набором спрайтов, нарисованных художником.

Для начала скопируйте папку *GnomeParts* из исходного набора ресурсов в папку *Sprites* проекта. Эта папка содержит две вложенные папки: *Alive* содержит новые части тела живого гномика, а *Dead* содержит части тела погибшего гномика (рис. 7.2). Начнем со спрайтов, изображающих живого гномика, а изображения частей тела погибшего гномика используем позже.

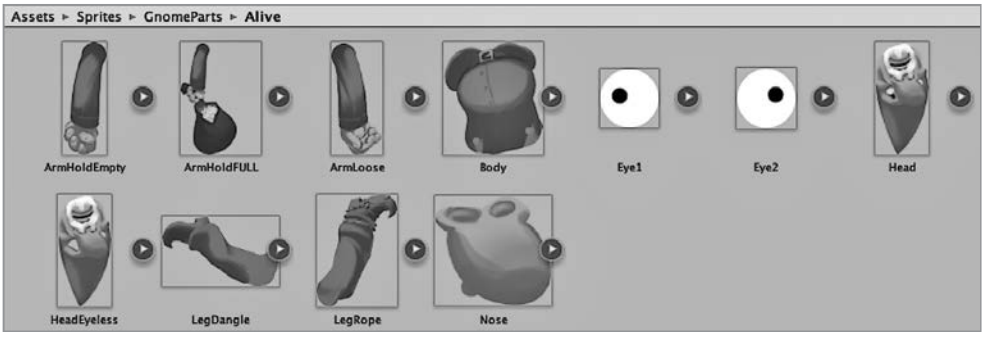

Рис. 7.2. Спрайты с частями тела гномика из папки Alive

В загружаемом пакете вы найдете намного больше ресурсов, чем мы используем, в том числе версию головы без глаз, созданную на случай, если для отображения глаз будут использоваться отдельные спрайты. Если вы захотите улучшить игру еще больше, чем описывается в этой книге, эти дополнительные ресурсы могут вам пригодиться!

Первым делом настроим спрайты для использования в объекте Gnome. В частности, нужно импортировать их как спрайты и поместить опорные точки спрайтов в правильные места. Для этого выполните следующие действия.

1. *Преобразуйте изображения в спрайты, если это еще не было сделано*. Выберите спрайты в папке *Alive* и установите для них тип текстуры Sprite (2D and UI).
2. *Откорректируйте опорные точки спрайтов*. Для каждого спрайта, кроме Body, выполните следующее:
   a) выберите спрайт;
   b) щелкните по кнопке Sprite Editor (Редактор спрайта);
   c) перетащите пиктограмму опорной точки (то есть маленькую синюю окружность) в точку, вокруг которой должна вращаться данная часть тела. Например, на рис. 7.3 показано местоположение опорной точки для спрайта ArmHoldEmpty.

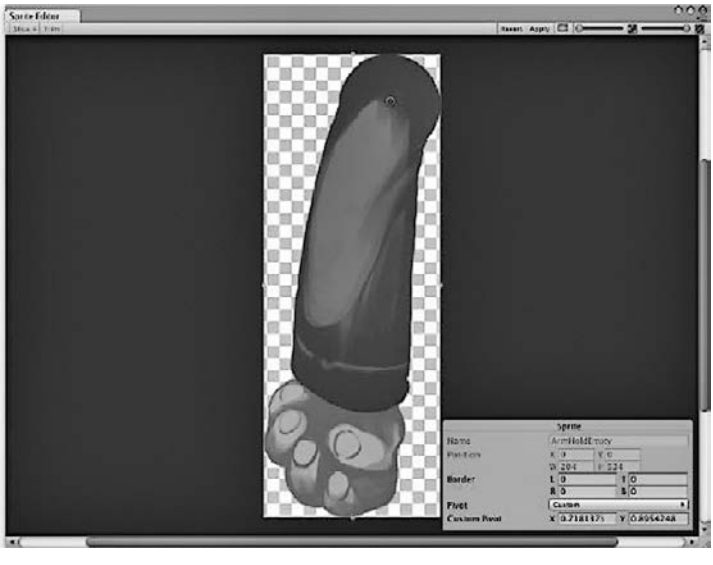

Рис. 7.3. Настройка опорной точки для спрайта ArmHoldEmpty; обратите внимание на местоположение опорной точки справа вверху

После настройки спрайтов можно задействовать их для изображения гномика. Для поддержания порядка и сохранения возможности использовать старую версию гномика мы создадим копию его шаблона и внесем необходимые изменения в новую версию. Затем мы настроим игру на использование новой, улучшенной версии гномика.

Закончив с этим, мы добавим результат в сцену и начнем заменять разные компоненты тела гномика новыми изображениями. Отличный план? Да! И вот что надо сделать.

1. *Скопируйте шаблонный объект прототипа гномика*. Найдите шаблон прототипа гномика и скопируйте его, нажав клавиши Ctrl-D (Command-D на Mac). Дайте новому объекту имя Gnome.
2. *Добавьте нового гномика в сцену*. Перетащите новый шаблон в окно Scene (Сцена), чтобы создать его экземпляр.
3. *Измените оформление*. Выбирайте части тела гномика по одной и заменяйте старые спрайты новыми. Например, выберите голову и замените ее спрайтом Head из папки *Alive*.

Когда вы закончите, гномик должен выглядеть примерно так, как показано на рис. 7.4. Если какие-то части тела находятся не совсем в правильных позициях — ничего страшного, чуть позже мы это исправим.

Рис. 7.4. Объект Gnome с новыми спрайтами

Далее мы должны настроить местоположение частей тела гномика. У новых спрайтов несколько иные форма и размеры, и мы должны правильно разместить их. Для этого выполните следующие действия.

1. *Поместите голову, руки и ноги в правильные места*. Выберите объект Head и скорректируйте его местоположение так, чтобы шея располагалась точно посередине между плечами. Проделайте то же самое с руками (присоединив их точно к плечам) и с ногами (присоединив их точно к тазу).

   Обратите внимание, что для удобства на спрайте Body имеются опорные точки розового цвета в области плеч.

Выполняя позиционирование частей тела, важно соблюдать правильное наложение спрайтов — ноги не должны отображаться поверх туловища, туловище

не должно отображаться поверх рук, голова должна отображаться поверх всех остальных спрайтов, составляющих гномика.

2. *Исправьте порядок наложения частей тела.* Выберите голову и обе руки и установите свойство Order in Layer визуализатора Sprite Renderer равным 2.

   Затем выберите туловище и установите порядок равным 1.

По окончании объект Gnome должен выглядеть так, как показано на рис. 7.5.

Рис. 7.5. Вид гномика после правильного размещения спрайтов

Изменение физических параметров

После размещения спрайтов, составляющих гномика, мы должны изменить физические параметры некоторых компонентов, а именно: придать коллайдерам правильную форму и настроить сочленения так, чтобы конечности гномика могли двигаться относительно опорных точек.

Сначала настроим коллайдеры. Поскольку спрайты не являются горизонтальными или вертикальными линиями, мы должны заменить простые прямоугольные и круговые коллайдеры *многоугольными*.

Создать многоугольный коллайдер можно двумя способами: дать Unity сгенерировать фигуру автоматически или определить ее вручную. Мы рассмотрим вариант создания вручную, потому что это более эффективно (Unity обычно генерирует сложные фигуры, отрицательно влияющие на производительность) и позволяет точнее контролировать конечный результат.

Когда многоугольный коллайдер добавляется в объект, имеющий визуализатор спрайта, Unity автоматически сгенерирует многоугольник, описывающий все непрозрачные части изображения. Чтобы определить иную форму для области обнаружения столкновений, компонент многоугольного коллайдера следует добавлять в игровой объект, не имеющий визуализатора спрайта. Самый простой способ в этом случае — создать пустой дочерний объект и добавить многоугольный коллайдер в него. Итак, выполните следующие действия.

1. *Удалите существующие коллайдеры.* Выберите все ноги и руки и удалите компонент Box Collider 2D. Затем выберите голову и удалите компонент Circle Collider 2D.
2. *Повторите следующие действия для всех рук, ног и головы:*
   a) *добавьте дочерний объект для коллайдера.* Создайте новый пустой игровой объект с именем Collider. Сделайте дочерними для объекта части тела и установите его координаты равными 0,0,0;
   b) *добавьте многоугольный коллайдер.* Выберите вновь созданный объект Collider и добавьте в него компонент Polygon Collider 2D. На экране появится фигура зеленого цвета (рис. 7.6); по умолчанию Unity создаст пятиугольник, и вам нужно скорректировать эту фигуру, чтобы она как можно точнее описывала объект;
   c) *отредактируйте фигуру многоугольного коллайдера.* Щелкните по кнопке Edit Collider (Редактировать коллайдер) (рис. 7.7), чтобы перейти в режим редактирования.

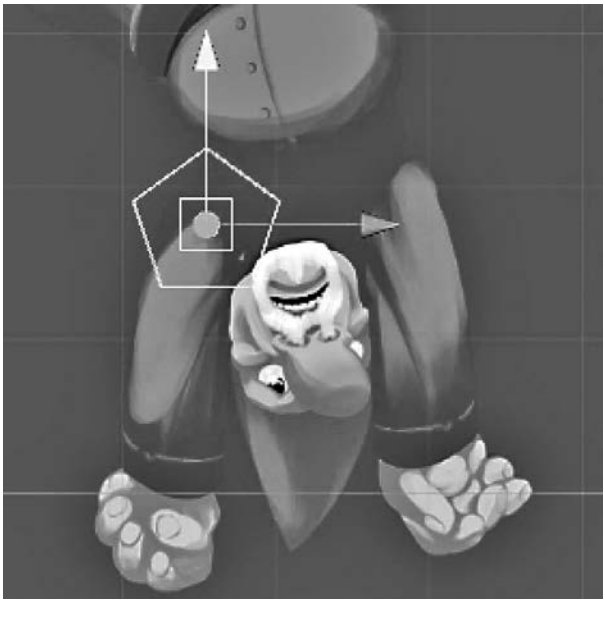

Рис. 7.6. Вновь добавленный многоугольный коллайдер Polygon Collider 2D

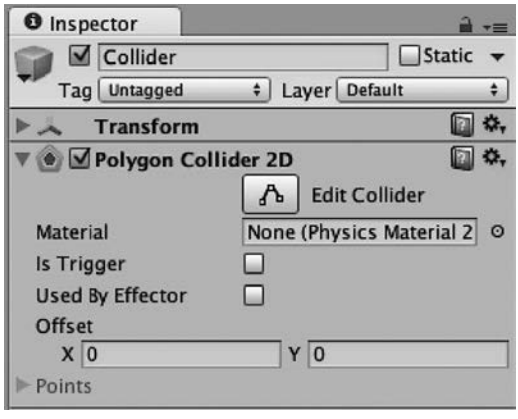

Рис. 7.7. Кнопка Edit Collider (Редактировать коллайдер)

В этом режиме можно передвигать отдельные вершины фигуры. Можно также щелкнуть на любой стороне фигуры и потянуть ее, чтобы создать новую вершину, а удерживая нажатой клавишу Ctrl (Command на Mac), можно щелчком мыши удалить любую вершину.

Двигая вершины, разместите их так, чтобы получившаяся фигура как можно точнее соответствовала части тела (рис. 7.8).

По завершении щелкните по кнопке Edit Collider (Редактировать коллайдер) еще раз.

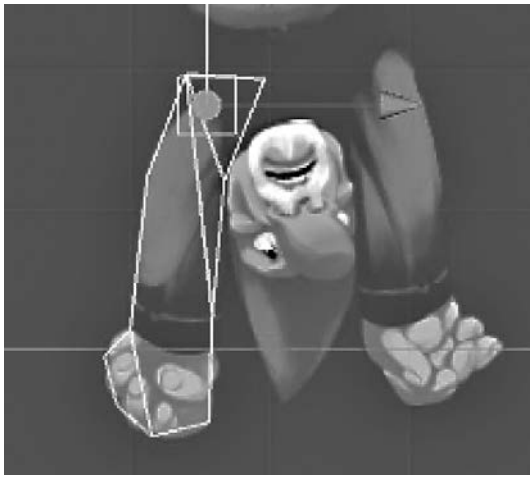

Рис. 7.8. Скорректированный коллайдер для руки гномика

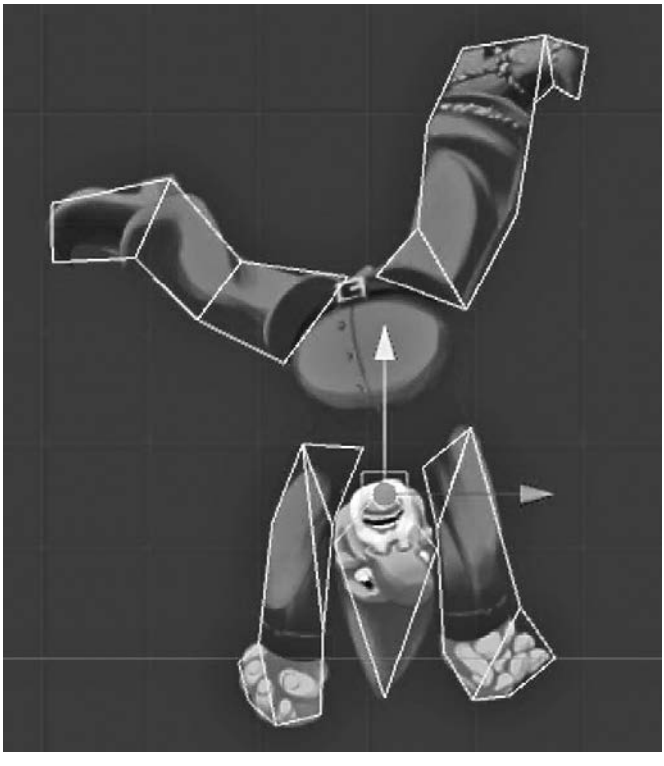

Рис. 7.9. Коллайдеры для рук, ног и головы

Вновь добавленные коллайдеры должны выглядеть примерно так, как показано на рис. 7.9.

В наборе коллайдеров требуется произвести еще одно изменение: круговой коллайдер туловища следует немного увеличить, чтобы он точнее соответствовал более крупному туловищу.

3. *Увеличьте радиус кругового коллайдера объекта Body до 1.2.*

   Цель данного этапа в том, чтобы получить коллайдеры, примерно соответствующие форме спрайтов, но без перекрытия. Это означает, что в процессе игры части тела гномика не будут перекрывать друг друга, создавая странное впечатление.

   Закончив с коллайдерами, можно заняться сочленениями. Как вы помните, голова, руки и ноги имеют шарнирные сочленения, которые присоединяются к туловищу. Теперь вы должны правильно определить опорные точки, чтобы избежать таких странностей, как вращение руки относительно локтя.

4. *Измените позиции Connected Anchor (Несущая точка привязки) и Anchor (Точка привязки) сочленений гномика.* Для каждой части тела, кроме туловища, перетащите точки привязки в опорные точки. Ноги должны вращаться относительно тазобедренных суставов, руки — относительно плечевых, а голова — вокруг шеи.

Если перетащить обе точки привязки ближе к центру спрайта, они зафиксируются в этой точке.

Не забывайте, что Leg Rope имеет два сочленения: одно служит для крепления ноги к туловищу, другое используется для крепления веревки. Сдвиньте второе сочленение в область ступни.

Также мы должны внести два изменения в сценарий Gnome. Помните, что спрайт, изображающий руку гномика, должен изменяться после касания с сокровищем? В настоящее время сценарий использует старый спрайт, не соответствующий новому изображению.

5. *Измените сценарий Gnome, использовав в нем новые спрайты.* Выберите родительский объект Gnome. Перетащите спрайт ArmHoldEmpty в поле Arm Holding Empty объекта Gnome, а спрайт ArmHoldFull — в поле Arm Holding Full.

   Теперь, когда гномик подберет сокровище, спрайт руки заменится правильным изображением. Кроме того, когда гномик бросит сокровище (что может случиться, если он коснется ловушки и погибнет), руку гномика не заменит прежнее схематическое изображение.

   Наконец, нужно немного изменить масштаб изображения гномика, чтобы его фигурка точнее соответствовала размерам игрового мира, а затем сохранить изменения в виде шаблонного объекта.

6. *Скорректируйте масштаб Gnome.* Выберите родительский объект Gnome и уменьшите масштабный коэффициент по осям X и Y с 0,5 до 0,3.

7. *Сохраните изменения в виде шаблона.* Выберите родительский объект Gnome и щелкните по кнопке Apply (Применить) в верхней части инспектора.

8. *Удалите гномика из сцены.* После создания шаблона нет необходимости продолжать хранить гномика в сцене, поэтому удалите его.

   Теперь, закончив вносить исправления в гномика, самое время изменить диспетчер игры Game Manager, чтобы он использовал вновь созданный объект.

9. *Исправьте диспетчер игры, чтобы он использовал вновь созданный шаблон.* Выберите объект Game Manager и перетащите только что созданный шаблон гномика в поле Gnome Prefab.

10. *Протестируйте игру.* Теперь в игре участвует обновленный гномик! Как это выглядит, можно увидеть на рис. 7.10.

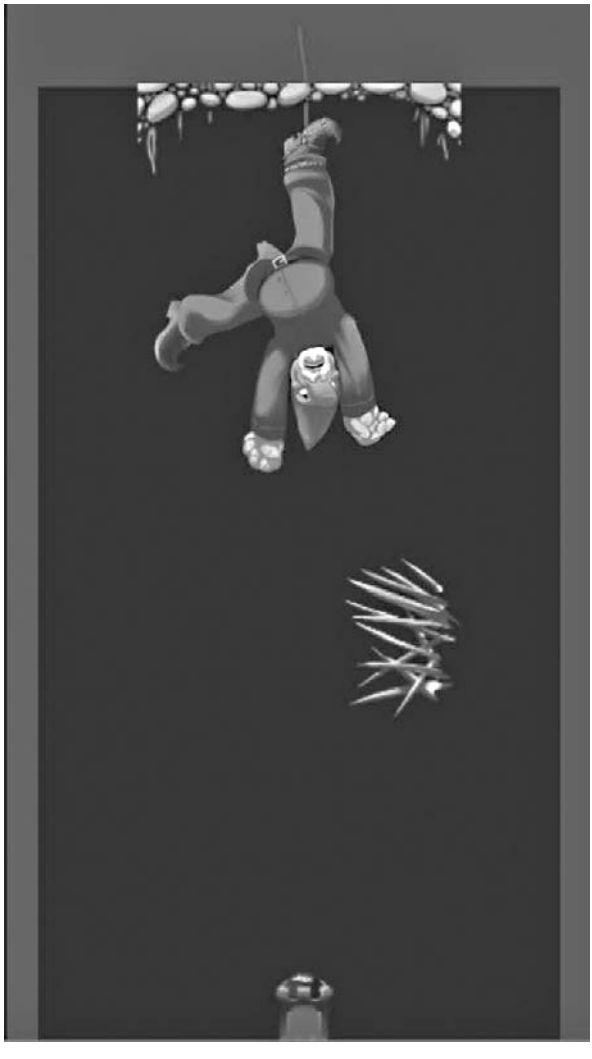

Рис. 7.10. Обновленный гномик в игре

Фон

В настоящее время фоном служит плоский серый прямоугольник, мало напоминающий стены колодца. Давайте исправим это!

Для решения этой проблемы добавим набор объектов, представляющих заднюю и боковые стены колодца. Прежде чем продолжить, добавьте в проект спрайты из папки *Background*.

Слои

Перед добавлением изображений мы должны сначала определить их порядок в сцене. При создании двумерной игры важно, но иногда бывает непросто обеспечить правильный порядок отображения спрайтов, когда одни должны появляться поверх других. К счастью, в Unity имеется встроенное решение, упрощающее эту задачу: *слои сортировки* (sorting layers).

Слой сортировки — это группа объектов, которые отображаются все вместе. Слои сортировки, как следует из названия, можно располагать в любом порядке. То есть можно сгруппировать несколько объектов в слой «Background», другие объекты — в слой «Foreground» и т. д. Кроме того, внутри слоев объекты также можно упорядочивать, чтобы одни элементы фона всегда отображались позади других.

В игре всегда присутствует хотя бы один слой — с названием «Default». И все новые объекты помещаются в этот слой, если вы не укажете иной.

Мы добавим в этот проект несколько слоев сортировки, в частности:

- слой Level Background, содержащий фоновые объекты уровня и всегда отображаемый на заднем плане;
- слой Level Foreground, содержащий объекты переднего плана, такие как стены;
- слой Level Objects, содержащий такие объекты, как ловушки.

Для создания слоев выполните следующие действия.

1. *Откройте инспектор тегов и слоев.* Откройте меню Edit (Правка) и выберите пункт Project Settings ▸ Tags & Layers (Настройки проекта ▸ Теги и Слои).
2. *Добавьте слой сортировки Level Background.* Откройте раздел Sorting Layers (Слои сортировки) и добавьте новый слой. Дайте ему имя Level Background.

   Перетащите его в начало списка (над слоем Default). Благодаря этому все объекты в данном слое будут отображаться *позади* объектов в слое Default.
3. *Добавьте слой Level Foreground.* Повторите процедуру и добавьте новый слой с названием Level Foreground. Поместите его в списке *ниже* слоя Default. Благодаря этому все объекты в данном слое будут отображаться *перед* объектами в слое Default.
4. *Добавьте слой Level Objects.* Повторите процедуру еще раз и добавьте новый слой с названием Level Objects. Поместите его в списке ниже слоя Default и выше слоя Level Foreground. В него будут помещаться ловушки и сокровище и, соответственно, отображаться позади объектов переднего плана.

Создание фона

Закончив создание слоев, можно приступать к конструированию самого фона. Фон имеет три темы оформления, а именно коричневую, синюю и красную. Каждая тема состоит из нескольких спрайтов: задняя стена, боковая стена и затененная версия боковой стены.

Чтобы вы могли скомпоновать содержимое уровня в соответствии со своими вкусами, можно предложить создать шаблоны для трех разных тем. Сначала мы создадим коричневую тему, уложив объекты и сохранив результат как шаблон; затем проделаем то же самое и создадим синюю и красную темы.

Прежде чем мы начнем, создадим объект, содержащий все объекты из фонового слоя, чтобы сохранить порядок. Для этого выполните следующие действия.

1. *Создайте объект-контейнер Level.* Создайте новый пустой игровой объект, открыв меню GameObject (Игровой объект) и выбрав пункт Create Empty (Создать пустой). Дайте новому объекту имя Level и поместите его в позицию с координатами (0,0,1).
2. *Создайте объект-контейнер Background Brown.* Создайте еще один игровой объект и дайте ему имя Background Brown. Сделайте его дочерним по отношению к объекту Level и поместите его в позицию с координатами (0,0,0). Благодаря этому позиция дочернего объекта не будет смещена относительно родительского объекта Level.
3. *Добавьте спрайты оформления фона.* Перетащите в сцену спрайт BrownBack и сделайте его дочерним по отношению к объекту Background Brown.

   Выберите новый спрайт и измените в его свойстве Sorting Layer слой Level Background. Наконец, установите его координату X в позицию 0, чтобы выровнять по центру.
4. *Добавьте задний объект боковой стены.* Перетащите в сцену спрайт BrownBackSide и сделайте его дочерним по отношению к объекту Background Brown.

   Выберите в его свойстве Sorting Layer слой Level Background и установите свойство Order In Layer в значение 1. Благодаря этому объект будет отображаться перед главным фоном, но позади всех других объектов в других слоях.

   Установите его координату X в позицию −3, чтобы сместить влево.
5. *Добавьте передний объект боковой стены.* Перетащите в сцену спрайт BrownSide и сделайте его дочерним по отношению к объекту Background Brown.

Выберите в свойстве Sorting Layer слой Level Background. Установите координату X в позицию −3,7, а координату Y в одну позицию со спрайтом BrownBackSide. Они должны находиться на одном уровне по вертикали, но передний объект боковой стены следует сместить чуть влево.

Поскольку боковые объекты по высоте вдвое меньше основного фонового изображения, создайте второй ряд из этих же боковых объектов.

Чтобы скопировать их, выберите оба спрайта, BrownBackSide и BrownSide, и создайте их копии нажатием комбинации Ctrl-D (Command-D на Mac).

Сместите вновь добавленные боковые объекты вниз, выровняв их по нижнему краю верхнего ряда. В результате получившийся фон должен выглядеть так, как показано на рис. 7.11.

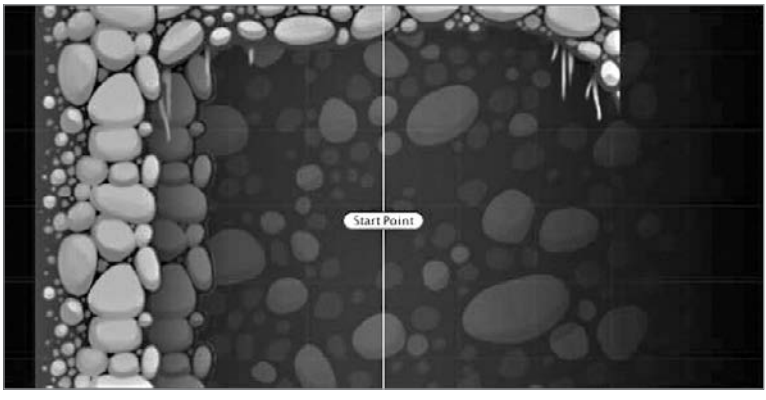

Рис. 7.11. Частично готовый фон

Закончив с настройкой боковых объектов слева, можно переходить к созданию боковых объектов справа. Для этого скопируйте имеющиеся спрайты и разместите их с правого края.

1. *Еще раз скопируйте боковые объекты*. Выберите все объекты BrownSide и BrownBackSide и нажмите комбинацию Ctrl-D (Command-D на Mac).
2. *Включите позиционирование маркеров Center (В центре)*. Если в данный момент на кнопке переключателя отображается надпись Pivot (В опорной точке), щелкните по ней, чтобы появилась надпись Center (В центре).
3. *Поверните объекты*. Используя инструмент вращения, поверните объекты справа на 180°. Удерживайте клавишу Ctrl (Command на Mac), чтобы осуществить поворот с шагом, заданным в настройках.

Не используйте инспектор для изменения величины поворота — в этом случае каждый спрайт будет повернут относительно своего центра. Нам же нужно повернуть объекты относительно общего центра.

4. *Переверните объекты по вертикали*. Для этого измените их масштаб по оси Y на −1. Если этого не сделать, освещение будет выглядеть неправильно — вверх ногами.

   В результате раздел Transform этих объектов в инспекторе должен выглядеть так, как показано на рис. 7.12.

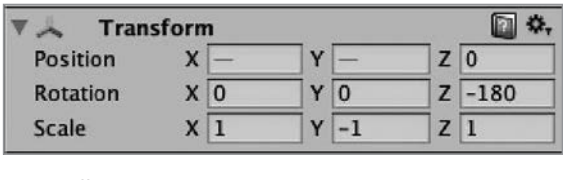

Рис. 7.12. Настройки трансформации элементов фона с правой стороны

5. Передвиньте новые объекты на правый край сцены. Именно там они и должны находиться. Результат должен быть таким, как показано на рис. 7.13.

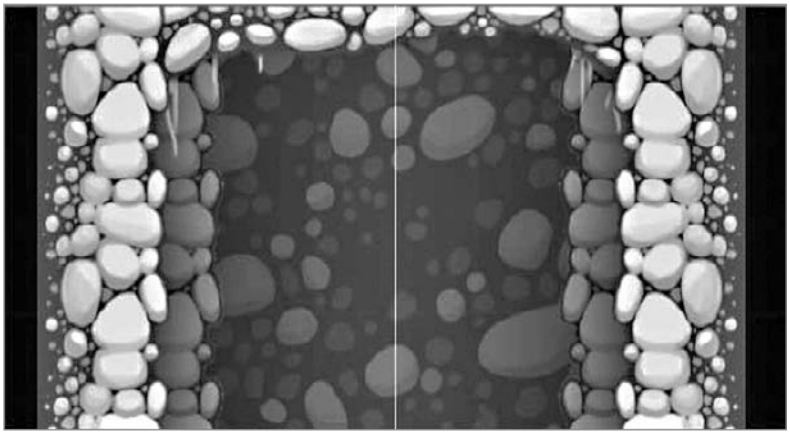

Рис. 7.13. Готовый фон

Теперь объект Background Brown готов и его можно преобразовать в шаблон. Для этого выполните следующие действия.

1. *Создайте шаблон из объекта Background Brown.* Перетащите объект Background Brown в панель обозревателя проекта, и Unity создаст шаблон. Перетащите получившийся шаблон в папку *Level*.
2. *Скопируйте объект Background Brown.* Выберите объект Background Brown и нажмите комбинацию Ctrl-D (Command-D на Mac) несколько раз. Переместите каждый из этих объектов вниз, стыкуя друг с другом, пока не получите достаточно протяженный фон.

Разные фоновые изображения

Теперь, после создания первого фона, проделаем те же действия для создания двух других фоновых тем.

1. *Создайте тему Background Blue.* Создайте новый пустой игровой объект с именем Background Blue и сделайте его дочерним по отношению к объекту Level.

   Выполните те же действия, как при создании объекта Background Brown, но на этот раз используйте спрайты BlueBack, BlueBackSide и BlueSide.

   Когда все будет сделано, не забудьте создать шаблонный объект Background Blue.
2. *Создайте тему Background Red.* И снова выполните те же действия, используя спрайты RedBack, RedBackSide и RedSide.

В итоге уровень должен выглядеть примерно так, как показано на рис. 7.14.

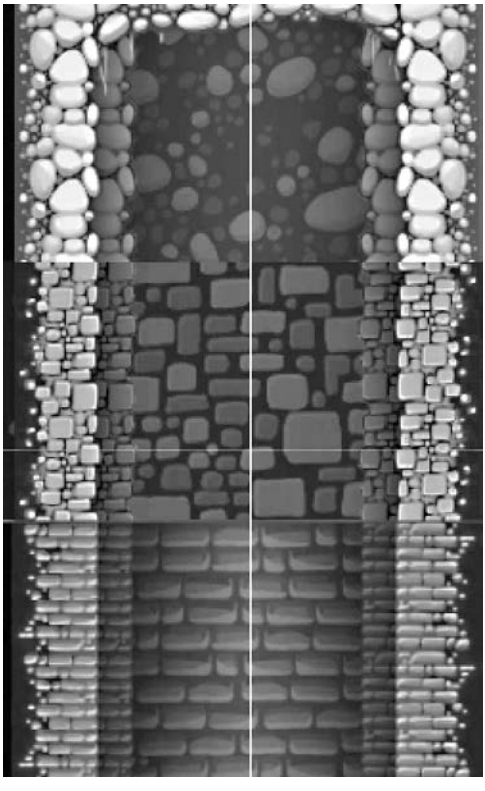

Рис. 7.14. Области фона

У такого оформления есть одна проблема: фоновые изображения одного цвета сочетаются хорошо, но границы между фоновыми изображениями разного цвета выглядят резко.

Чтобы это исправить, наложим сверху спрайты, покрывающие эту неоднородность. Эти спрайты будут находиться в слое Level Foreground и появляться повсюду в игре.

1. *Добавьте спрайт BlueBarrier*. Этот спрайт создавался специально с целью скрыть границу между коричневым и синим фоном. Поместите его в точку, где граничат коричневый и синий фон, и сделайте его дочерним по отношению к объекту Level.
2. *Добавьте спрайт RedBarrier*. Он создавался с целью скрыть границу между синим и красным фоном. Поместите его в точку, где граничат синий и красный фон, и сделайте его дочерним по отношению к объекту Level.
3. *Переместите оба спрайта в соответствующий слой сортировки*. Выберите оба спрайта, BlueBarrier и RedBarrier, и выберите в свойстве Sorting Layer слой Level Foreground.

Затем установите свойство Order In Layer в значение 1. В результате этого барьеры будут отображаться поверх боковых стен.

В итоге уровень должен выглядеть, как показано на рис. 7.15.

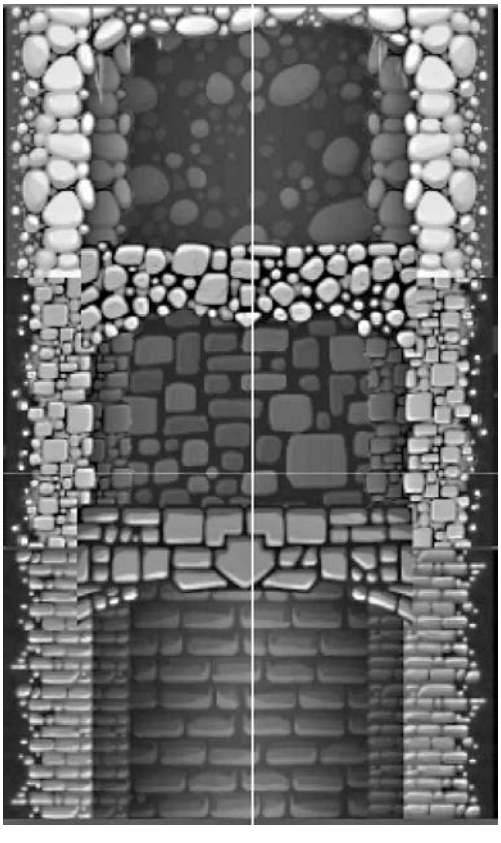

Рис. 7.15. Фон со спрайтами Barrier

Дно колодца

Осталось только добавить последнюю вещь: дно колодца. В этой игре в колодце нет воды, и его дно покрывает песок. Часть песка намыта на стены. Чтобы добавить его в сцену, выполните следующие действия.

1. *Создайте объект-контейнер для спрайтов с изображением дна*. Создайте новый пустой игровой объект с именем Well Bottom. Сделайте его дочерним по отношению к объекту Level.
2. *Добавьте спрайт, изображающий дно колодца*. Перетащите спрайт Bottom и добавьте его как дочерний объект в объект Well Bottom.

Выберите для спрайта уровень сортировки Level Background и установите его свойство Order In Layer в значение 2. Благодаря этому он будет отображаться перед спрайтами с изображениями стен, но позади любых других объектов.

Поместите спрайт на дно колодца и установите его координату X в позицию 0, чтобы выровнять по горизонтали с другими спрайтами в слое.

3. *Добавьте декоративный спрайт слева.* Перетащите в сцену спрайт SandySide и добавьте его как дочерний объект в объект Well Bottom.

   Выберите в свойстве Sorting Layer уровень сортировки Level Foreground. Установите свойство Order In Layer в значение 1, чтобы спрайт отображался поверх стен.

   Затем переместите спрайт влево и разместите его на одном уровне со стенами (на рис. 7.16 показано, как это должно выглядеть).

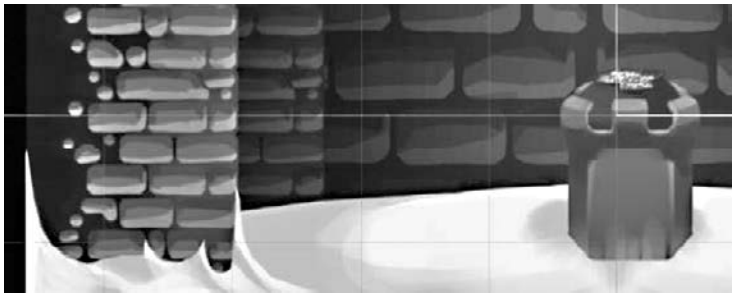

Рис. 7.16. Спрайт SandySide, размещенный на одном уровне со стеной

4. *Добавьте спрайт справа.* Скопируйте спрайт SandySide. Измените масштаб по оси X на −1, чтобы перевернуть его, а затем переместите вправо от стены.
5. *Убедитесь, что сокровище размещено правильно.* Переместите спрайт, изображающий сокровище, так, чтобы он находился в центре песчаного дна.

Результат должен выглядеть так, как показано на рис. 7.17.

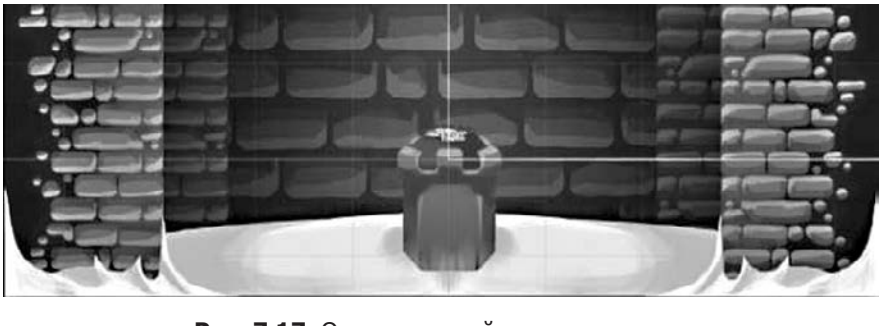

Рис. 7.17. Окончательный вид дна колодца

Настройка камеры

Чтобы вписать обновленный фон в игру, осталось только скорректировать настройки камеры. Нам потребуется внести следующие изменения: во-первых, камеру нужно настроить так, чтобы игрок мог видеть весь уровень на всю ширину, а во-вторых, необходимо изменить сценарий, ограничивающий перемещение камеры, чтобы учесть изменившиеся размеры уровня. Для настройки камеры выполните следующие действия.

1. *Измените размер камеры.* Выберите объект Main Camera и установите его свойство Ortho Size в значение 7. Благодаря этому камера сможет охватить уровень во всю ширину.
2. *Измените ограничения, влияющие на позиционирование камеры.* После расширения области, охватываемой камерой, необходимо скорректировать ограничения, влияющие на позиционирование камеры. Установите свойство Top Limit камеры в значение 11,5.

Также нужно изменить свойство Bottom Limit, но значение для него должно выбираться в зависимости от глубины созданного вами колодца.

Лучший способ определить это значение: опустить гномика как можно ниже, и если камера остановится до того, как гномик достигнет дна, уменьшите значение свойства Bottom Limit; если камера опустилась ниже дна (показав синий фон), увеличьте значение Bottom Limit.

Запишите это значение, прежде чем остановите игру, потому что оно будет сброшено в исходное при завершении игры; остановив игру, введите записанное число в поле Bottom Limit.

Пользовательский интерфейс

Теперь займемся улучшением пользовательского интерфейса игры. Создавая пользовательский интерфейс, мы использовали стандартные кнопки Unity. Их можно было бы использовать и дальше, но они плохо сочетаются с графикой игры, поэтому заменим их на более подходящие.

Кроме того, нам нужно показать экран «Конец игры», когда гномик с сокровищем поднимется вверх до точки выхода, а также экран, когда игрок ставит игру на паузу.

Для начала импортируем все спрайты, которые понадобятся в этом разделе. Импортируйте папку *Import* со спрайтами и поместите ее в папку *Sprites*.

Эти спрайты создавались в расчете на высокое разрешение, поэтому их можно использовать в самых разных ситуациях. Чтобы спрайты можно было использовать как кнопки в игре, движок Unity должен знать их отображаемые размеры. Скорректировать размеры спрайта можно с помощью его свойства Pixels Per Unit, которое управляет масштабом изображения при добавлении в компоненты пользовательского интерфейса или в визуализаторы спрайтов.

Настройте спрайты, выбрав все изображения в этой папке (кроме «You Win») и установив их свойства Pixels Per Unit в значение 2500.

Для начала обновим кнопки Up и Down, которые сейчас отображаются в правом нижнем углу экрана, поместив на них симпатичные изображения. Для этого нам понадобится удалить надписи на кнопках, а затем настроить их размеры и местоположение, подогнав под новые изображения. Итак, выполните следующие действия.

1. *Удалите надпись с кнопки Down*. Найдите объект Down Button и удалите из него дочерний объект Text.
2. *Добавьте спрайт*. Выберите объект Down Button и выберите в его свойстве Source Image спрайт Down (который находится в папке *Interface*).

   Щелкните по кнопке Set Native Size (Установить исходный размер), и кнопка скорректирует свои размеры.

   В заключение настройте местоположение кнопки, чтобы она оказалась в правом нижнем углу экрана.
3. *Обновите кнопку Up*. Повторите ту же процедуру для кнопки Up. Удалите дочерний объект Text и выберите в свойстве Source Image спрайт Up. Затем щелкните по кнопке Set Native Size (Установить исходный размер) и настройте местоположение кнопки, чтобы она находилась точно над кнопкой Down.
4. *Протестируйте игру*. Кнопки по-прежнему должны выполнять свои функции, но теперь они выглядят намного привлекательнее (рис. 7.18).

Теперь поместим кнопки в контейнер. На то есть две причины: во-первых, всегда желательно иметь хорошо организованный пользовательский интерфейс, а во-вторых, объединив кнопки в один объект, мы сможем включать и выключать их все сразу. Это очень скоро пригодится нам, когда мы приступим к реализации меню Pause (Пауза). Для этого выполните следующие действия.

1. *Создайте родительский объект-контейнер для кнопок*. Создайте новый пустой игровой объект с именем Gameplay Menu. Сделайте его дочерним по отношению к объекту Canvas.
2. *Установите размеры объекта так, чтобы он покрывал весь экран*. Установите точки привязки объекта Gameplay Menu так, чтобы он растянулся на весь экран по горизонтали и по вертикали. Для этого щелкните по кнопке Anchor (Привязка) слева вверху в панели инспектора и в раскрывшемся меню выберите вариант в правом нижнем углу (рис. 7.19).

   После этого установите свойства Left, Top, Right и Bottom в значение 0. Благодаря этому объект заполнит все пространство, занимаемое родительским объектом (которым в данном случае является объект Canvas), то есть объект заполнит весь экран.
3. *Переместите кнопки в объект Gameplay Menu*. В панели иерархии перетащите обе кнопки, Up и Down, в объект Gameplay Menu.

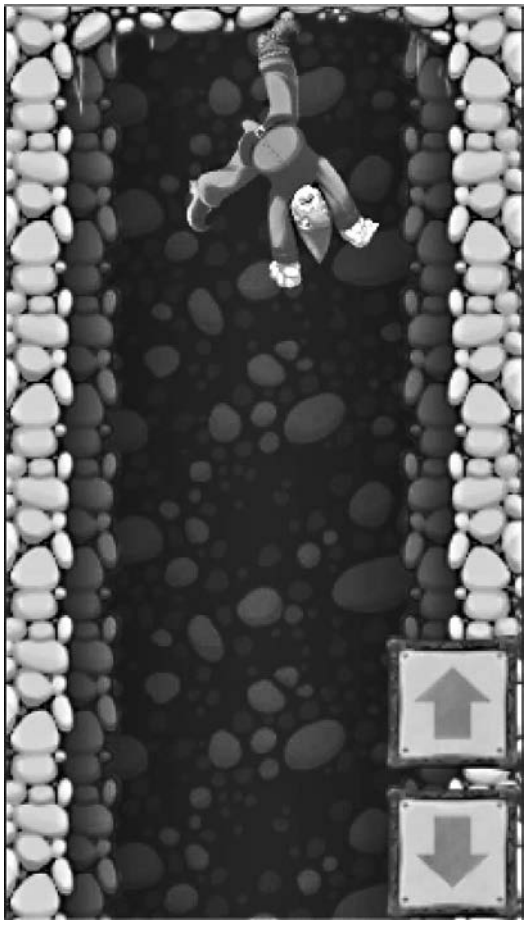

Рис. 7.18. Обновленные кнопки Up и Down

Далее создадим экран «Вы выиграли». На нем будет отображаться изображение, а также кнопка, позволяющая игроку начать новую игру. Для этого выполните следующие действия.

1. *Создайте объект-контейнер Game Over для экрана.* Создайте новый пустой игровой объект с именем Game Over и сделайте его дочерним по отношению к объекту Canvas.

   Растяните его по горизонтали и по вертикали, как мы сделали это с объектом Gameplay Menu.

2. *Добавьте изображение Game Over.* Создайте новый объект Image, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Image (Пользовательский интерфейс ▸ Изображение). Сделайте новый объект Image дочерним по отношению к созданному перед этим объекту Game Over.

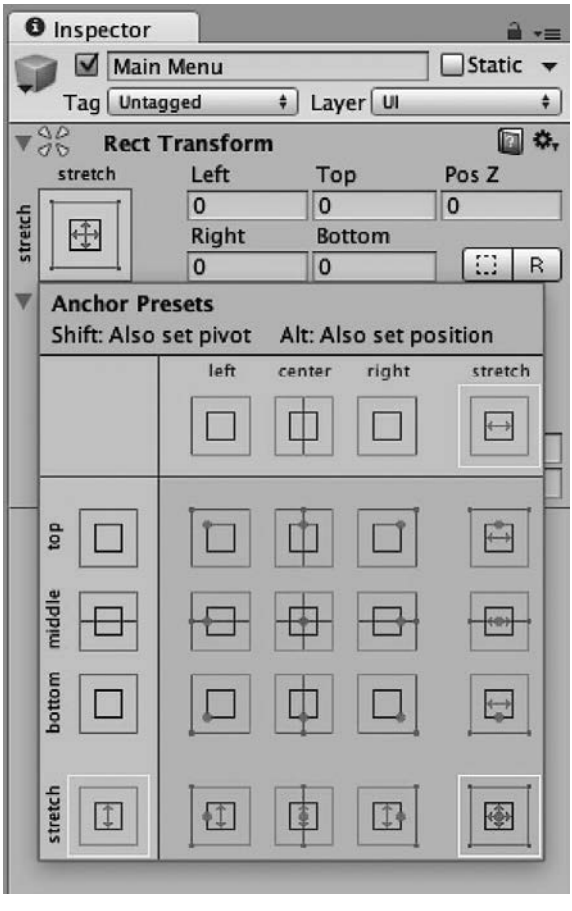

Рис. 7.19. Настройка растягивания объекта по горизонтали и по вертикали

Растяните объект Image по горизонтали и по вертикали. Установите свойства Left Margin и Right Margin равными 30, а свойство Bottom Margin — равным 60. Благодаря этому вокруг изображения появятся отступы и оно не будет перекрывать кнопку New Game, которую мы сейчас добавим.

Выберите в свойстве Source Image объекта Image спрайт You Win и включите флажок Preserve Aspect (Сохранить соотношение сторон), чтобы исключить его растягивание.

3. *Добавьте кнопку New Game*. Добавьте новый объект Button в объект Game Over, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Button (Пользовательский интерфейс ▸ Кнопка).

   Добавьте на кнопку надпись с текстом «New Game» и привяжите кнопку к нижнему краю в центре, выбрав вариант bottom-center после щелчка по кнопке Anchor (Привязка). В итоге интерфейс должен выглядеть так, как показано на рис. 7.20.

Рис. 7.20. Экран Game Over

4. *Подключите кнопку New Game к диспетчеру игры Game Manager.* Нам требуется, чтобы щелчок по кнопке вызывал сброс игры в исходное состояние. Для этого можно организовать вызов функции `RestartGame` из сценария `GameManager`.

   Щелкните по кнопке + внизу окна инспектора и перетащите диспетчер игры Game Manager в появившееся поле. Затем выберите функцию GameManager ▸ RestartGame.

Теперь мы должны связать диспетчер игры Game Manager с вновь созданными элементами пользовательского интерфейса. Сценарий `GameManager` уже предусматривает включение/выключение соответствующих элементов пользовательского интерфейса, исходя из состояния игры: когда игра запускается, он пытается активировать любой объект, указанный в переменной Gameplay Menu, и деактивировать все остальные меню. Выполните следующие действия, чтобы произвести необходимые настройки и протестировать результат.

1. *Подключите Game Manager к меню.* Выберите Game Manager и перетащите объект Gameplay Menu в поле Gameplay Menu. Затем перетащите объект Game Over в поле Game Over Menu.
2. *Протестируйте игру.* Опустите гномика до дна колодца, подберите сокровище и вернитесь в точку выхода. В результате должен появиться экран Game Over.

У нас осталось последнее меню, которое нужно настроить, — меню Pause, а также кнопка, которая приостанавливает игру. Кнопка Pause будет находиться в верхнем правом углу экрана, и когда пользователь нажмет ее, игра приостановится и появятся кнопки для возобновления и перезапуска игры.

Чтобы добавить кнопку Pause, создайте новый объект Button с именем Menu Button. Сделайте его дочерним по отношению к объекту Gameplay Menu.

- Удалите дочерний объект Text и выберите в свойстве Source Image кнопки спрайт Menu.
- Щелкните по кнопке Set Native Size (Установить исходный размер), а затем перетащите кнопку в правый верхний угол сцены.
- Привяжите кнопку к правому верхнему углу.
- В итоге новая кнопка должна выглядеть так, как показано на рис. 7.21.

Рис. 7.21. Кнопка Menu

Далее нужно связать кнопку с диспетчером игры Game Manager. Нажатие этой кнопки даст команду диспетчеру игры перейти в состояние паузы, отобразить экран Main Menu (который мы создадим далее), скрыть объект Gameplay Menu и приостановить игру.

Чтобы связать кнопку Menu с диспетчером игры Game Manager, щелкните по кнопке + внизу инспектора и перетащите Game Manager в появившееся поле.

Выберите функцию `GameManager.SetPaused`. Установите флажок SetPaused, чтобы при нажатии кнопки в вызов функции передавался аргумент `true`.

Теперь можно настроить отображение меню после приостановки игры.

1. *Создайте объект-контейнер Main Menu*. Создайте новый пустой объект с именем Main Menu. Сделайте его дочерним по отношению к объекту Canvas и растяните по горизонтали и по вертикали. Установите свойства Left Margin, Top Margin, Right Margin и Bottom Margin в значение 0.
2. *Добавьте кнопки в Main Menu*. Добавьте две кнопки с именами Restart и Resume. Сделайте их дочерними по отношению к только что созданному объекту Main Menu и задайте текст надписей на них: «Restart Game» и «Resume Game».

   В итоге объект Main Menu должен выглядеть так, как показано на рис. 7.22.

Рис. 7.22. Объект меню Main Menu

3. *Подключите кнопки к диспетчеру игры Game Manager*. Выберите кнопку Restart и настройте в ней вызов функции `GameManager.RestartGame`.

   Затем выберите кнопку Resume и настройте в ней вызов функции `GameManager.Reset`.
4. *Подключите Main Menu к диспетчеру игры Game Manager*. Диспетчер игры должен знать, какой объект отобразить, когда вызывается функция `SetPaused`. Выберите Game Manager и перетащите объект Main Menu в поле Main Menu диспетчера игры.
5. *Протестируйте игру.* Теперь у вас должна быть возможность приостановить и возобновить игру, а также перезапустить ее.

Режим неуязвимости

Идея чит-кодов в видеоиграх родилась благодаря практическим соображениям. Тестируя игру в процессе разработки, бывает достаточно утомительно проходить различные ловушки и головоломки, чтобы достичь определенной точки для тестирования. Чтобы ускорить разработку, добавляются инструменты, меняющие принцип игры: в играх-стрелялках часто есть коды, избавляющие игрока от нападений врагов, а в играх-стратегиях можно отключить «туман войны».

Эта игра не является исключением: в процессе ее создания вам не стоит беспокоиться, что придется постоянно иметь дело с каждым препятствием, когда вы запускаете игру. Для этого мы добавим инструмент, делающий гномика неуязвимым.

Он будет реализован как флажок (иногда флажки называют *переключателями*) в левом верхнем углу экрана. Когда флажок включен, гномик не будет погибать. При этом он также будет «получать повреждения», то есть будут воспроизводиться все эффекты, которые мы добавим в игру в следующей главе, что очень пригодится для тестирования.

Следуя заведенному порядку, поместим этот флажок в объект-контейнер подобно другим компонентам пользовательского интерфейса. Начнем с создания этого контейнера.

1. *Создайте объект-контейнер Debug Menu.* Создайте новый пустой игровой объект с именем Debug Menu и сделайте его дочерним по отношению к объекту Canvas. Растяните его по горизонтали и по вертикали во весь экран и установите его свойства Left Margin, Top Margin, Right Margin и Bottom Margin в значение 0.
2. *Добавьте флажок Invincible.* Создайте новый объект Toggle, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Toggle (Пользовательский интерфейс ▸ Флажок). Дайте объекту имя Invincible.

   Привяжите объект к левому верхнему углу и переместите его в верхний левый угол экрана.
3. *Настройте флажок.* Выберите объект Label — дочерний объект только что добавленного объекта Toggle — и установите белый цвет компоненту Text. Задайте текст надписи «Invincible».

   Выключите свойство Is On объекта Toggle.

   В результате флажок должен выглядеть так, как показано на рис. 7.23.
4. *Подключите флажок к диспетчеру игры Game Manager.* Добавьте новый элемент в событие Value Changed флажка Invincible, щелкнув по кнопке +. Перетащите Game Manager в появившееся поле и выберите функцию `GameManager.gnomeInvincible`. Теперь, когда состояние флажка изменится, одновременно изменится значение свойства `gnomeInvincible`.
5. *Протестируйте игру.* Запустите игру и включите флажок Invincible. Теперь гномик не должен погибать, коснувшись ловушки!

Рис. 7.23. Флажок Invincible в левом верхнем углу экрана

В заключение

Теперь игра стала выглядеть намного привлекательнее. Игровой процесс сохранился и прекрасно действует, и, помимо прочего, мы добавили собственные инструменты, упрощающие тестирование. Но предстоит сделать еще кое-что. В следующей главе мы добавим больше эффектов и усовершенствований и завершим разработку игры, создав структуру меню и звуков.

8 Последние штрихи в игре «Колодец с сокровищами»

Больше ловушек и уровней

Игра начинает обретать форму: теперь гномик, пользовательский интерфейс и фон игры выглядят намного привлекательнее. Сейчас у нас имеется только одна ловушка: коричневые шипы. Следующим нашим шагом станет создание еще двух таких же ловушек, но окрашенных для разнообразия в другие цвета.

Мы также добавим ловушку нового типа: вращающееся лезвие. Вращающееся лезвие наносит такие же травмы, как шипы, но имеет более сложное устройство — оно состоит из трех спрайтов, один из которых — анимированный.

Наконец, мы добавим препятствия, не наносящие повреждений, в форме стен и блоков, которые игрок должен будет обходить. Размещение этих объектов в сочетании с ловушками заставит игрока думать, как управлять перемещениями по уровню.

Шипы

Начнем с добавления шипов другого цвета. Сейчас у нас уже есть шаблон для существующих шипов, поэтому нам осталось только обновить спрайты и создать коллайдер. Для этого выполните следующие действия.

1. *Создайте новые шаблонные объекты для шипов.* Выберите шаблон SpikesBrown и создайте его копию, нажав комбинацию Ctrl-D (Command-D на Mac). Дайте новому объекту имя SpikesBlue.

   Создайте еще одну копию с именем SpikesRed.

2. *Обновите спрайт.* Выберите шаблон SpikesBlue и замените его спрайт изображением SpikesBlue.

3. *Обновите многоугольный коллайдер.* Поскольку многоугольный коллайдер находится в том же объекте, что и визуализатор Sprite Renderer, Unity будет использовать спрайт для определения формы коллайдера. Однако при смене спрайта форма коллайдера не обновляется автоматически; чтобы исправить эту проблему, нужно сбросить коллайдер в исходное состояние.

Щелкните по пиктограмме с изображением шестеренки в правом верхнем углу компонента Polygon Collider 2D и выберите пункт Reset (Сбросить) в появившемся меню.

4. *Обновите объект SpikesRed.* Вслед за объектом SpikesBlue выполните те же действия с объектом SpikesRed (и используйте изображение SpikesRed).

Когда все будет сделано, можете добавить в уровень несколько объектов SpikesBlue и SpikesRed.

Вращающееся лезвие

Далее добавим вращающееся лезвие. Вращающееся лезвие выступает чуть дальше, чем шипы, и имеет угрожающий вид циркулярной пилы. С точки зрения игровой логики вращающееся лезвие производит действие, идентичное шипам: когда гномик касается его, он погибает. Однако добавление в игру разных ловушек помогает усложнить процесс игры и удержать интерес игрока.

Поскольку вращающееся лезвие — это анимированный объект, мы сконструируем его из нескольких спрайтов. Кроме того, один из этих спрайтов — циркулярная пила — будет вращаться с высокой скоростью.

Чтобы добавить вращающееся лезвие в игру, перетащите спрайт SpinnerArm в сцену и выберите в его свойстве Sorting Layer слой Level Objects.

Перетащите спрайт SpinnerBladesClean и сделайте его дочерним по отношению к объекту SpinnerArm. Выберите в его свойстве Sorting Layer слой Level Objects и установите свойство Order in Layer в значение 1. Разместите его в верхней части спрайта опоры (SpinnerArm), затем установите его координату X равной 0, чтобы выровнять по центру.

Перетащите спрайт SpinnerHubcab и сделайте его дочерним по отношению к объекту SpinnerArm. Выберите в его свойстве Sorting Layer слой Level Objects и установите свойство Order in Layer в значение 2. Установите его координату X равной 0.

В результате вращающееся лезвие должно выглядеть так, как показано на рис. 8.1.

Теперь сделаем его опасным для гномика: подключим сценарий `SignalOnTouch`. Сценарий `SignalOnTouch` должен посылать сообщение, когда гномик коснется коллайдера, присоединенного к объекту; чтобы он работал, нам также понадобится добавить коллайдер. Выполните следующие действия, чтобы произвести все необходимые настройки.

1. *Добавьте коллайдер к вращающемуся лезвию.* Выберите объект SpinnerBladesClean и добавьте компонент Circle Collider 2D. Уменьшите его радиус до 2; это уменьшит область поражения и немного облегчит преодоление ловушки.

2. *Добавьте компонент SignalOnTouch.* Щелкните по кнопке Add Component (Добавить компонент) и добавьте сценарий `SignalOnTouch`.

   Щелкните по кнопке + внизу инспектора и перетащите во вновь появившееся поле объект Game Manager. Выберите функцию `GameManager.TrapTouched`.

Рис. 8.1. Сконструированное вращающееся лезвие

Далее нам нужно заставить лезвие вращаться. Для этого мы добавим объект аниматора Animator и настроим выполняемый им анимационный эффект. Анимационный эффект очень прост: достаточно просто повернуть присоединенный объект на полный оборот.

Для настройки аниматора нужно создать контроллер аниматора Animator Controller. Контроллеры аниматоров позволяют с помощью разных параметров определять, какой анимационный эффект должен воспроизводить аниматор. В этой игре мы не будем использовать продвинутые возможности контроллера, но вам будет полезно знать, что они существуют. Для настройки выполните следующие действия.

1. *Добавьте аниматор.* Выберите лезвие и добавьте новый компонент Animator.
2. *Создайте контроллер аниматора.* В папке *Level* создайте новый ресурс Animator Controller с именем Spinner.

   Здесь же, в папке *Level*, создайте новый ресурс Animation с именем Spinning.
3. *Подключите к аниматору новый контроллер аниматора.* Выберите лезвие и перетащите только что созданный ресурс Animator Controller в поле Controller.

Далее настроим сам контроллер аниматора.

1. *Откройте аниматор Animator*. Дважды щелкните по Animator Controller, чтобы открыть вкладку Animation (Анимация).
2. *Добавьте анимацию Spinning*. Перетащите анимацию Spinning в панель Animator. После этого в Animator Controller должно появиться единственное состояние анимации, помимо уже существующих элементов Entry, Exit и Any State (рис. 8.2).

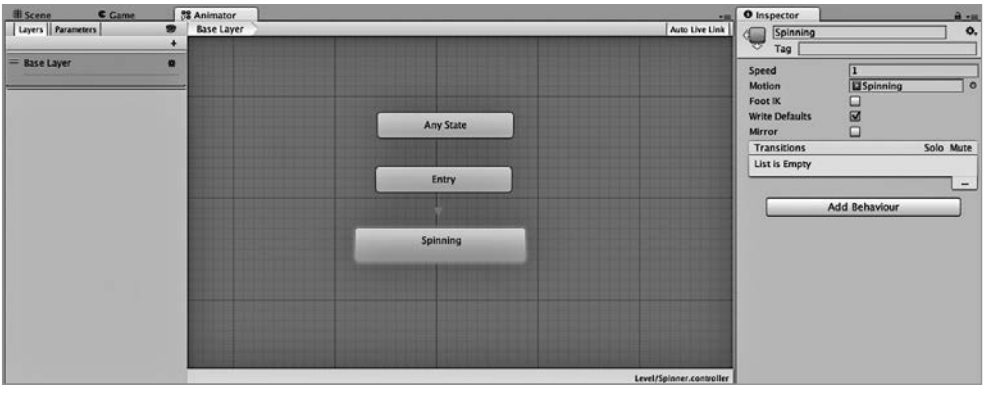

Рис. 8.2. Настройки Animator Controller для анимации Spinner

Теперь, когда аниматор Animator настроен на использование контроллера Animator Controller, который также настроен на запуск анимации Spinning, самое время настроить саму анимацию вращения.

1. *Выберите вращающееся лезвие*. Вернитесь обратно в представление сцены и выберите вращающееся лезвие.
2. *Откройте панель* Animation (Анимация). Откройте меню Window (Окно) и выберите пункт Animation (Анимация). Перетащите открывшуюся вкладку Animation (Анимация) в удобное для вас место. При желании можете даже присоединить ее к какому-нибудь разделу в интерфейсе, перетащив вкладку на панель в главном окне Unity.

   Прежде чем продолжить, убедитесь, что анимация Spinning выбрана вверху слева в панели Animation (Анимация).
3. *Выберите кривую в свойстве Rotation анимации Spinning*. Щелкните по кнопке Add Property (Добавить свойство) — откроется список анимируемых компонентов. Перейдите к элементу Transform ▸ Rotation и щелкните по кнопке + справа от списка.

По умолчанию свойства имеют два ключевых кадра — один соответствует началу анимации, а другой — концу (рис. 8.3).

Нам нужно, чтобы объект поворачивался на 360°. Для этого в начале объект должен иметь угол поворота 0°, а в конце — 360°. Чтобы добиться желаемого, нужно изменить последний ключевой кадр в анимации.

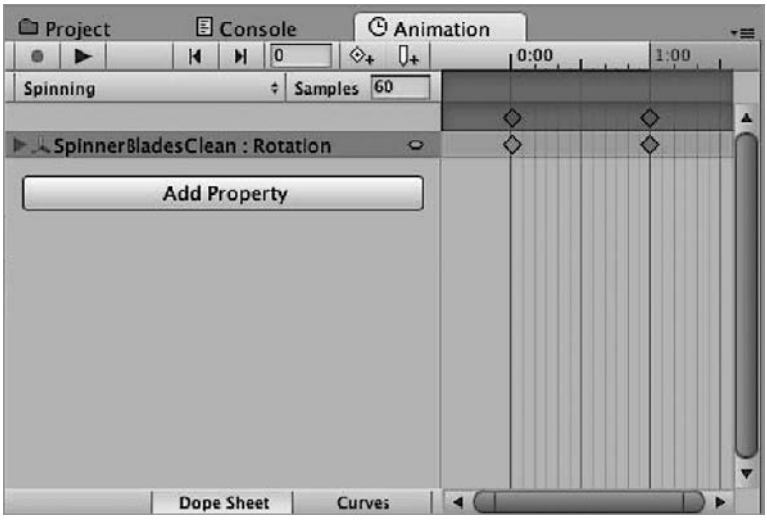

Рис. 8.3. Ключевые кадры для вновь созданной анимации

1. *Выберите самый правый ключевой кадр.*
2. *Щелкните по самому правому ромбику в панели Animation (Анимация), и анимация перескочит в эту точку на шкале времени.* После этого Unity перейдет в режим записи, то есть будет фиксировать любые изменения в анимации Spinning. Обратите также внимание, что квадратик в верхней части окна Unity станет красного цвета — это будет напоминать вам о происходящем.

   Взглянув на панель инспектора, можно заметить, что значение свойства Rotation компонента Transform также окрасилось в красный цвет.
3. *Измените угол поворота.* Установите угол поворота Z равным 360.
4. *Протестируйте анимацию.* Щелкните по кнопке Play (Играть) во вкладке Animation (Анимация) и понаблюдайте, как будет вращаться лезвие. Если оно вращается недостаточно быстро, щелкните и передвиньте последний ключевой кадр ближе к началу. Это уменьшит продолжительность анимации и заставит объект вращаться быстрее.
5. *Настройте бесконечное воспроизведение анимации в цикле.* Перейдите в панель обозревателя проекта Project и выберите созданный вами ресурс анимации Spinning. В инспекторе установите флажок Loop Time.
6. *Запустите игру.* Теперь циркулярная пила должна вращаться.

Прежде чем вращающееся лезвие будет готово к использованию, нужно уменьшить его размер, чтобы оно гармоничнее смотрелось в игровом поле.

1. *Измените масштаб вращающегося лезвия.* Выберите родительский объект SpinnerArm и установите масштаб по осям X и Y равным 0,4.

2. *Преобразуйте его в шаблон*. Перетащите объект SpinnerArm в панель обозревателя проекта Project. В результате будет создан новый шаблонный объект с именем SpinnerArm; переименуйте его в Spinner.

Теперь можно добавить вращающееся лезвие в нужное место в сцене, и гномик будет погибать, когда коснется его.

Препятствия

Кроме ловушек хорошо также добавить препятствия, не убивающие гномика. Эти препятствия будут замедлять продвижение игрока и вынуждать его думать о том, как обойти разные ловушки, добавленные вами.

Эти препятствия — самые простые из всех объектов, что вы добавляли в игру: для их создания вам потребуются только спрайт и коллайдер. Поскольку все они имеют очень простую конструкцию и похожи друг на друга, можно одновременно создать шаблоны сразу для всех препятствий. Вот что вы должны сделать для этого.

1. *Перетащите спрайты с изображениями блоков*. Добавьте в сцену спрайты BlockSquareBlue, BlockSquareRed и BlockSquareBrown. Затем добавьте спрайты BlockLongBlue, BlockLongRed и BlockLongBrown.
2. *Добавьте коллайдеры*. Выберите все шесть объектов и щелкните по кнопке Add Component (Добавить компонент) внизу инспектора. Добавьте компонент Box Collider 2D, и в каждом блоке появится зеленая граница, очерчивающая область определения столкновений.
3. *Преобразуйте их в шаблоны*. Перетащите каждый блок по отдельности в папку *Level*, чтобы создать шаблоны.

Блоки готовы, и теперь можно приступать к их размещению в уровне. Это было просто.

Эффекты частиц

Падение гномика вниз после гибели — не самый впечатляющий визуальный эффект. Чтобы создать эффект интереснее, нам придется задействовать системы частиц.

В частности, мы добавим эффекты частиц, появляющиеся в момент касания гномиком ловушки («брызги крови») и когда отделяется какая-то часть тела гномика («фонтан крови»).

Определение материала частиц

Поскольку в обоих случаях испускаемые частицы будут из одного и того же материала (кровь гномика), то начнем с создания единого материала, общего для двух систем. Выполните следующие действия, чтобы создать и настроить этот материал.

1. *Настройте текстуру Blood.* Найдите и выберите текстуру Blood. Измените ее тип со Sprite на Default и установите флажок Alpha Is Transparency (Альфа-прозрачность), как показано на рис. 8.4.
2. *Создайте материал Blood.* Создайте новый ресурс Material, открыв меню Assets (Ресурсы) и выбрав пункт Create ▸ Material (Создать ▸ Материал). Дайте материалу имя Blood и измените его шейдер на Unlit ▸ Transparent.

Затем перетащите текстуру Blood в поле Texture (Текстура). В результате настройки в инспекторе должны выглядеть так, как показано на рис. 8.5.

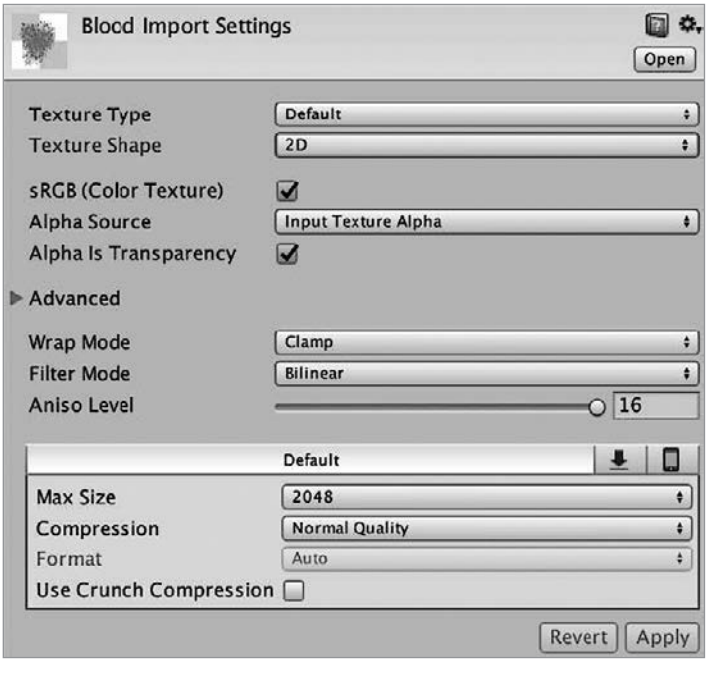

Рис. 8.4. Настройки для текстуры Blood

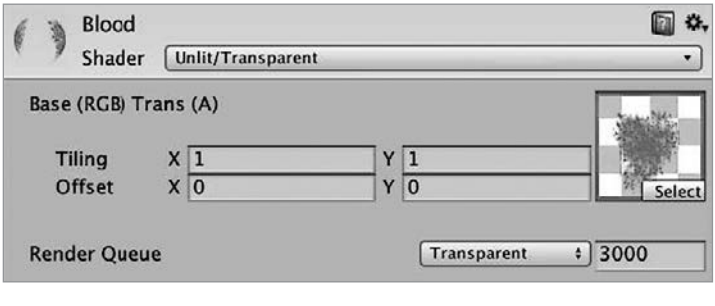

Рис. 8.5. Материал для эффекта частиц

Фонтан крови

Материал для крови готов к использованию, и теперь можно приступать к созданию эффектов частиц. Начнем с эффекта фонтана крови, который воспроизводится как эффект потока частиц, бьющего в определенном направлении и в итоге исчезающего. Вот как можно его создать.

1. *Создайте игровой объект для системы частиц.* Откройте меню GameObject (Игровой объект), откройте подменю Effects (Эффекты) и создайте новую систему частиц Particle System. Дайте новому объекту имя Blood Fountain.
2. *Настройте систему частиц.* Выберите объект и измените значения в разделе Particle System (Система частиц), как показано на рис. 8.6 и 8.7.

   Здесь есть пара параметров, о которых хотелось бы рассказать подробнее, поскольку они имеют нечисловые значения, которые нельзя просто скопировать из скриншота. В частности:

   - значение Color Over Lifetime (Цвет с течением времени) альфа-канала цвета изменяется от полностью непрозрачного (100 %) в начале до полностью прозрачного (0 %) в конце. Значение самого цвета изменяется от белого в начале до черного в конце;
   - в разделе Renderer (Визуализатор) системы частиц Particle System используется только что созданный материал Blood.
3. *Преобразуйте объект Blood Fountain в шаблон.* Перетащите объект Blood Fountain в папку *Gnome*.

Брызги крови

Теперь создадим шаблон Blood Explosion, испускающий одиночные брызги крови вместо сплошного потока.

1. *Создайте объект для системы частиц.* Создайте еще один игровой объект Particle System и дайте ему имя Blood Explosion.
2. *Настройте систему частиц.* Измените значения свойств в инспекторе, как показано на рис. 8.8.

   В этой системе частиц используется тот же материал и настройки изменения цвета с течением времени, как в эффекте Blood Fountain; единственное существенное отличие — новый эффект использует круговой эмиттер, и испускание всех частиц настроено так, что происходит одномоментно.
3. *Добавьте сценарий RemoveAfterDelay.* Чтобы не засорять сцену, эффект Blood Explosion должен удалять себя через некоторое время.

   Добавьте компонент RemoveAfterDelay в объект и установите его свойство Delay в значение 2.
4. Преобразуйте Blood Explosion в шаблон.

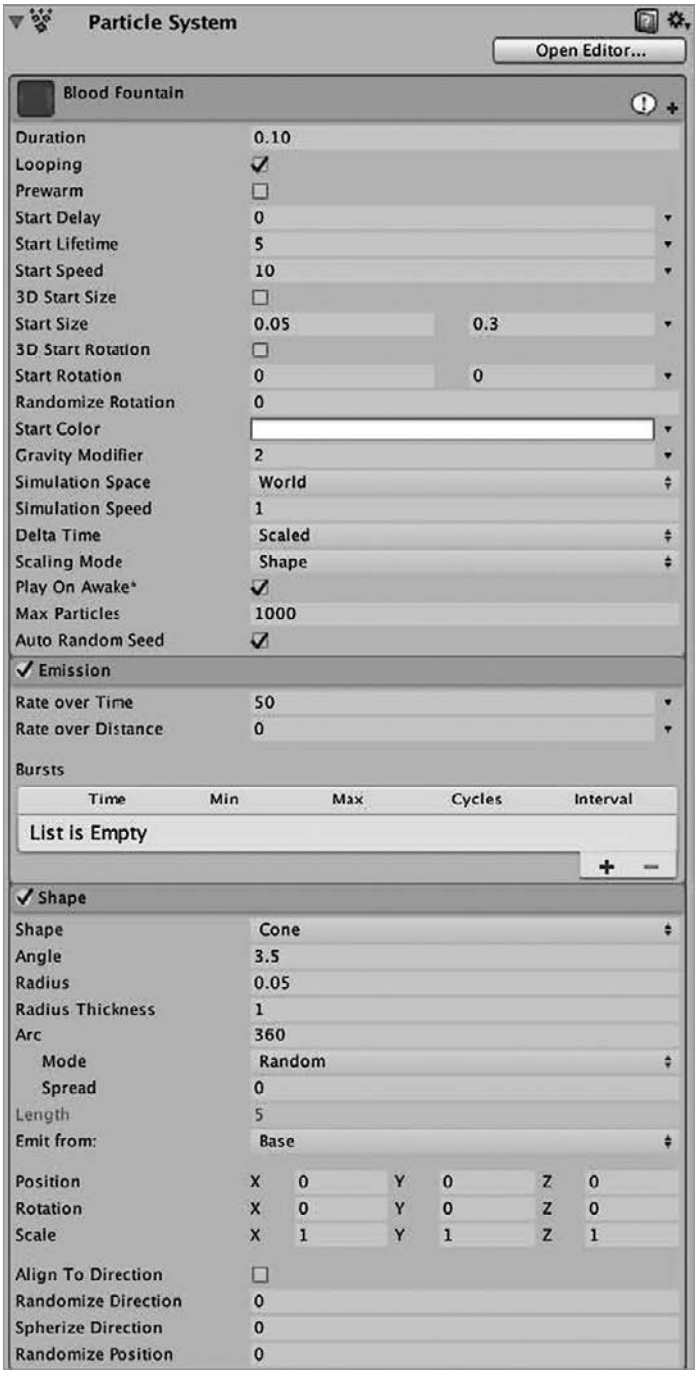

Рис. 8.6. Настройки объекта Blood Fountain

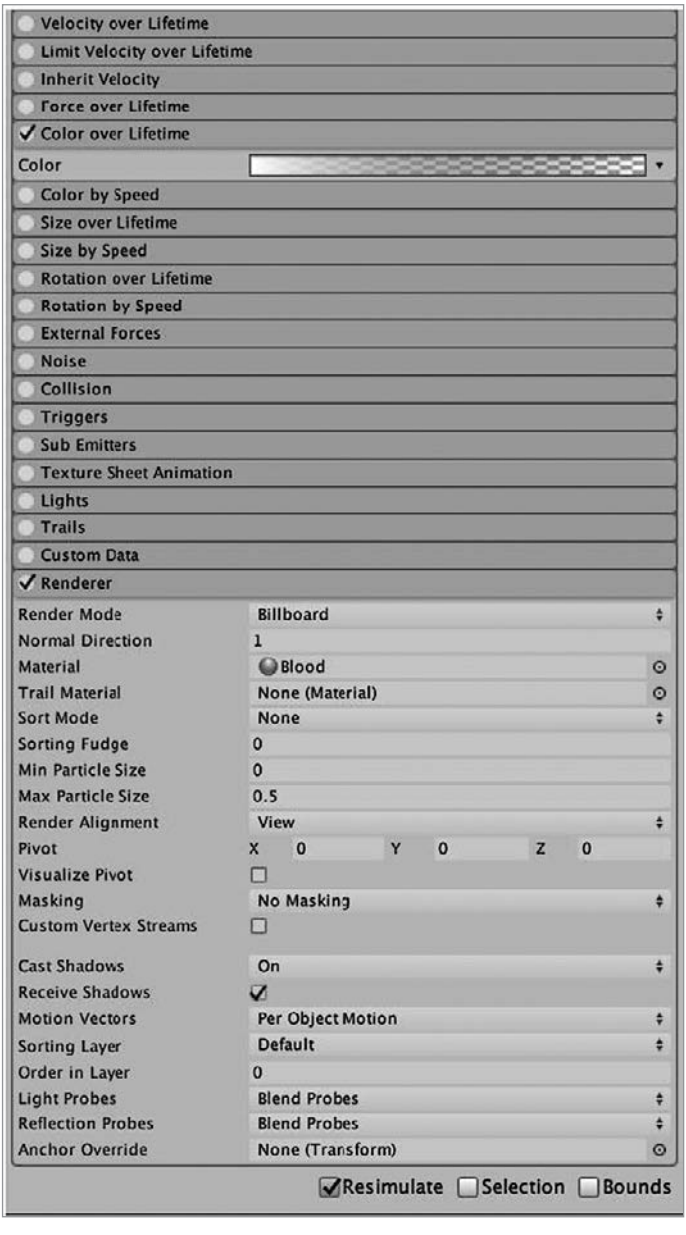

Рис. 8.7. Настройки объекта Blood Fountain (продолжение)

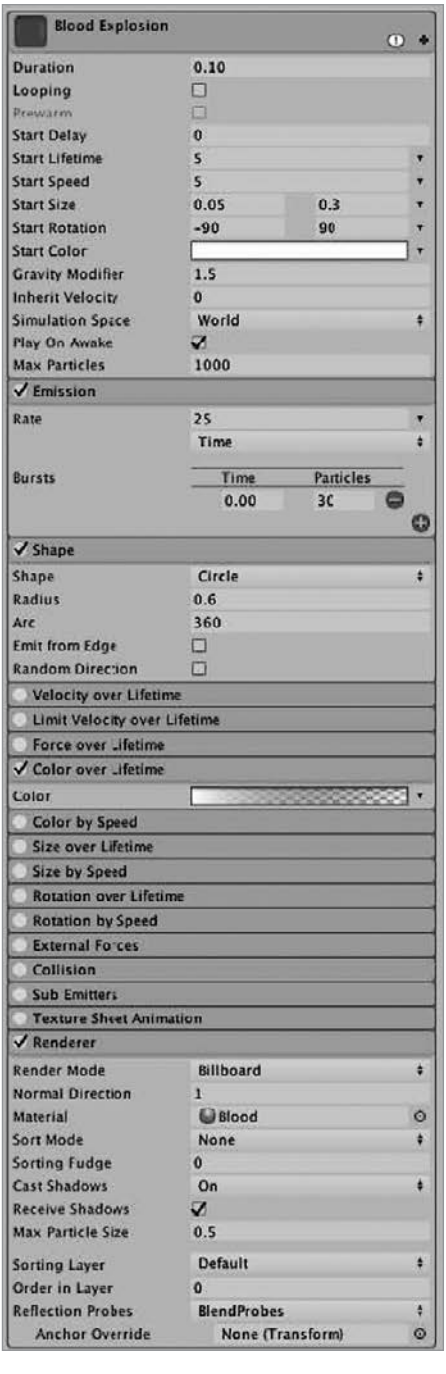

Рис. 8.8. Настройки объекта Blood Explosion

Теперь новые эффекты готовы к использованию в игре.

Использование систем частиц

Чтобы задействовать системы частиц в игре, нужно подключить их к шаблонному объекту Gnome. Вот как это можно сделать.

1. *Выберите шаблон Gnome*. Выберите правильный шаблон — нам нужен новый шаблон Gnome, а не старый Prototype Gnome.
2. *Подключите системы частиц к шаблону Gnome*. Перетащите шаблон Blood Explosion в поле Death Prefab, а шаблон Blood Fountain — в поле Blood Fountain.
3. *Протестируйте игру*. Подведите гномика к ловушке, чтобы он коснулся ее, — вы должны увидеть кровь.

Главное меню

Основная часть игры готова и усовершенствована. Теперь займемся особенностями, необходимыми в любой игре, а не только в нашей игре *Колодец с сокровищами*, а именно: нам нужен начальный экран и возможность закрыть его и перейти к игре.

Этот экран мы реализуем в виде отдельной сцены, чтобы не путать его с игрой. Так как меню — это сцена, которая проще самой игры, оно будет загружаться быстрее игры, и игрок увидит его практически сразу после запуска. Кроме того, меню начинает загрузку игры в фоновом режиме; когда игрок коснется кнопки New Game (Новая игра), после окончания загрузки игры произойдет переключение сцены. В итоге у игрока будет складываться ощущение, что игра загружается намного быстрее. Для создания и настройки начального экрана выполните следующие действия.

1. *Создайте новую сцену.* Откройте меню File (Файл) и выберите пункт New Scene (Новая сцена). Сразу же сохраните новую сцену, открыв меню File (Файл) и выбрав пункт Save Scene (Сохранить сцену). Дайте этой сцене имя Menu.
2. *Добавьте фоновое изображение.* Откройте меню GameObject (Игровой объект) и выберите пункт UI ▸ Image (Пользовательский интерфейс ▸ Изображение).

   Выберите в свойстве Source Image спрайт Main Menu Background.

   Установите точки привязки изображения так, чтобы оно растянулось на весь экран по горизонтали и по вертикали, и отцентрируйте его по горизонтали. Установите координату X в позицию 0, свойство Top Margin в значение 0, свойство Bottom Margin в значение 0 и ширину — равной 800.

   Включите флажок Preserve Aspect (Сохранить соотношение сторон), чтобы исключить его растягивание.

Теперь настройки в инспекторе должны выглядеть так, как показано на рис. 8.9, а само изображение — как на рис. 8.10.

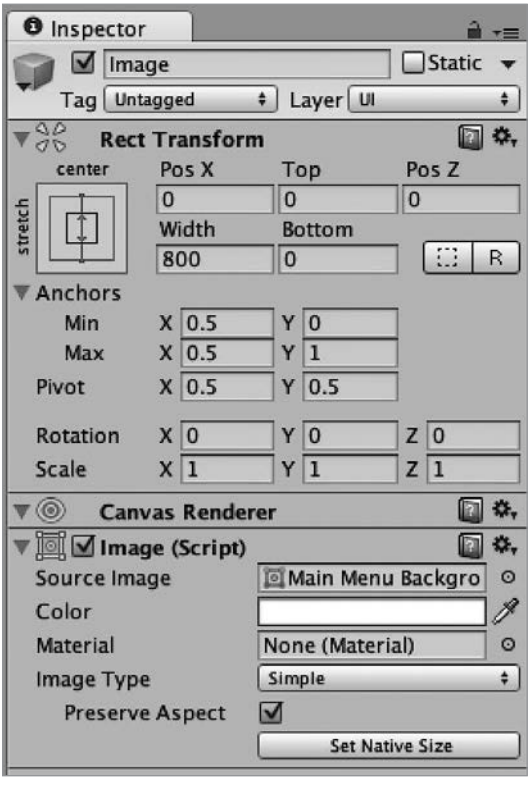

Рис. 8.9. Настройки фонового изображения для главного меню

Рис. 8.10. Фоновое изображение

3. *Добавьте кнопку New Game (Новая игра)*. Откройте меню GameObject (Игровой объект) и выберите пункт UI ▸ Button (Пользовательский интерфейс ▸ Кнопка). Дайте этому объекту имя New Game.

   Привяжите кнопку к нижнему краю в центре, выбрав вариант bottom-center после щелчка по кнопке Anchor (Привязка). Затем установите его координату X в позицию 0, координату Y в позицию 40, ширину равной 160, а высоту — равной 30.

   В компоненте Label кнопки задайте текст «New Game». В итоге кнопка должна выглядеть так, как показано на рис. 8.11.

Рис. 8.11. Меню с добавленной кнопкой

Загрузка сцены

Когда игрок нажимает кнопку New Game (Новая игра), мы должны показать заставку, сообщающую, что идет загрузка игры. Для ее создания выполните следующие действия.

1. *Создайте объект заставки.* Создайте новый пустой игровой объект с именем Loading Overlay. Сделайте его дочерним по отношению к объекту Canvas.

   Установите точки привязки объекта так, чтобы он растянулся на весь экран по горизонтали и по вертикали, и установите свойства Left Margin, Top Margin, Right Margin и Bottom Margin в значение 0. Это вынудит заставку заполнить весь экран.

2. *Добавьте компонент Image.* Пока объект Loading Overlay остается выбранным, щелкните по кнопке Add Component (Добавить компонент) и добавьте компонент Image. Объект Canvas окрасится в сплошной белый цвет.

   Измените свойство Color, выбрав черный цвет с уровнем прозрачности 50 %. Теперь заставка будет выглядеть как черная полупрозрачная обложка.

3. *Добавьте надпись.* Добавьте объект Text и сделайте его дочерним по отношению к объекту заставки Loading Overlay.

   Привяжите надпись к центру по вертикали и по горизонтали. Установите свойства Left, Top, Right и Bottom в значение 0.

   Затем увеличьте размер шрифта в компоненте Text и отцентрируйте текст по вертикали и по горизонтали. Выберите белый цвет для надписи и задайте текст «Loading...» (Загружается...).

Закончив настройку заставки, добавим код, который фактически загружает игру и переключает сцены после нажатия кнопки New Game (Новая игра). Для удобства подключим сценарий к главной камере Main Camera, но при желании его можно подключить к новому пустому игровому объекту. Выполните для этого следующие действия.

1. *Добавьте сценарий MainMenu в объект Main Camera.* Выберите объект Main Camera и добавьте новый сценарий на C# с именем `MainMenu`.

2. *Добавьте следующий код в файл MainMenu.cs*:

```
using UnityEngine.SceneManagement;

// Управляет главным меню.
public class MainMenu : MonoBehaviour {

  // Имя сцены, содержащей саму игру.
  public string sceneToLoad;

  // Компонент пользовательского интерфейса,
  // содержащий текст "Loading...".
  public RectTransform loadingOverlay;

  // Выполняет загрузку сцены в фоновом режиме. Используется
  // для управления, когда требуется переключить сцену.
  AsyncOperation sceneLoadingOperation;

  // При запуске начинает загрузку игры.
  public void Start() {
```

```
    // Скрыть заставку 'loading'
    loadingOverlay.gameObject.SetActive(false);

    // Начать загрузку сцены в фоновом режиме...
    sceneLoadingOperation =
      SceneManager.LoadSceneAsync(sceneToLoad);

    // ...но не переключаться в новую сцену,
    // пока мы не будем готовы.
    sceneLoadingOperation.allowSceneActivation = false;
  }

  // Вызывается в ответ на касание кнопки New Game.
  public void LoadScene() {

    // Сделать заставку 'Loading' видимой
    loadingOverlay.gameObject.SetActive(true);

    // Сообщить операции загрузки сцены, что требуется
    // переключить сцены по окончании загрузки.
    sceneLoadingOperation.allowSceneActivation = true;
  }

}
```

Сценарий `Main Menu` выполняет две операции: загружает сцену игры в фоновом режиме и обрабатывает нажатие кнопки New Game (Новая игра) игроком. В методе `Start` `SceneManager` обращается к диспетчеру сцены с требованием начать загрузку сцены в фоновом режиме. В ответ возвращается объект `AsyncOperation` и сохраняется в переменной с именем `sceneLoadingOperation`. Этот объект дает возможность контролировать процесс загрузки. В данном случае мы сообщаем объекту `sceneLoadingOperation`, что тот не должен активировать новую сцену по окончании загрузки. То есть, выполнив загрузку, `sceneLoadingOperation` будет ждать, пока пользователь будет готов перейти к следующему меню.

Этот переход выполняет метод `LoadScene`, который вызывается, когда пользователь касается кнопки New Game (Новая игра). Сначала он показывает заставку «loading», которую мы только что создали; затем объекту, загружающему сцену, сообщается, что он может активировать сцену после загрузки. То есть если сцена уже была загружена, она появится немедленно; если сцена еще не загрузилась, она появится по окончании загрузки.

Такая организация главного меню создает ощущение, что игра загружается быстрее. Так как главное меню требует меньше ресурсов, то оно появляется намного быстрее; после появления меню пользователь тратит какое-то время, чтобы нажать кнопку New Game (Новая игра), и в течение этого времени выполняется загрузка новой сцены. Так как пользователю не пришлось в бездействии смотреть в начальный экран с надписью «Пожалуйста, подождите», ему будет казаться, что игра загружается быстрее, чем если бы она запускалась без вывода промежуточного главного меню.

Выполните следующие действия.

1. *Настройте компонент Main Menu.* Настройте переменную Scene to Load в Main Menu (то есть укажите в ней имя главной сцены игры). Настройте переменную Loading Overlay, указав в ней созданный выше объект заставки Loading Overlay.
2. *Настройте кнопку запуска игры.* Выберите кнопку New Game (Новая игра) и настройте ее на вызов функции `MainMenu.LoadScene`.

Наконец, нам нужно настроить список сцен для включения в сборку. `Application.LoadLevel` и родственные функции могут загружать только сцены, включенные в сборку, то есть мы должны включить сцены, Main и Menu. Вот как это можно сделать.

1. *Откройте окно Build Settings (Параметры сборки).* Для этого откройте меню File (Файл) и выберите пункт File ▸ Build Settings (Файл ▸ Параметры сборки).
2. *Добавьте сцены в список Scenes In Build (Сцены в сборке).* Перетащите файлы обеих сцен, Main и Menu, из папки *Assets* в список Scenes In Build (Сцены в сборке). При этом сцена Menu должна находиться первой в списке, потому что эта сцена должна появляться первой после запуска игры.
3. *Протестируйте игру.* Запустите игру и щелкните по кнопке New Game (Новая игра). Вы закончили создание игры!

Звуки

Нам осталось доработать последний аспект игры: добавить звуковые эффекты. Без звуков игра смотрится просто как пугающий эпизод гибели гномика, и мы должны это исправить.

К счастью, код игры уже готов к добавлению звуков. Сценарий Signal On Touch воспроизведет звук в момент, когда гномик касается соответствующего коллайдера, но только если подключен источник звука. Чтобы это произошло, нужно добавить компоненты Audio Source в разные шаблонные объекты.

Сценарий Game Manager также готов воспроизводить звуки, когда гномик погибает или благополучно достигает точки выхода с сокровищем в руках, нам нужно лишь добавить компоненты Audio Source в Game Manager. Для этого выполните следующие действия.

1. *Добавьте компоненты Audio Source в шипы.* Найдите шаблон SpikesBrown и добавьте новый компонент Audio Source.

   Подключите звук Death By Static Object к новому компоненту Audio Source. Снимите флажки Loop (Бесконечно) и Play On Awake (Проигрывать автоматически).

   Повторите то же самое для шаблонов SpikesRed и SpikesBlue.

2. *Добавьте компонент Audio Source в Spinner*. Найдите шаблон Spinner и добавьте новый компонент Audio Source. Подключите звук Death by Moving Object к компоненту Audio Source. Снимите флажки Loop (Бесконечно) и Play On Awake (Проигрывать автоматически).
3. *Добавьте компонент Audio Source в Treasure*. Выберите объект Treasure на дне колодца и добавьте новый компонент Audio Source. Подключите звук Treasure Collected к компоненту Audio Source. Снимите флажки Loop (Бесконечно) и Play On Awake (Проигрывать автоматически).
4. *Добавьте компонент Audio Source в Game Manager*. Наконец, выберите объект Game Manager и добавьте новый компонент Audio Source. Оставьте поле Audio Clip пустым; вместо этого подключите звук Game Over к полю Gnome Died Sound, а звук You Win — к полю Game Over Sound.
5. *Протестируйте игру*. Теперь вы должны слышать звуковые эффекты, когда гномик погибает, подбирает сокровище и выигрывает в игре.

В заключение и задания

Вы закончили создание игры *Колодец с сокровищами*, и теперь она должна выглядеть так, как показано на рис. 8.12. Поздравляем!

Теперь вы можете добавить новые возможности, чтобы дальше продолжить изучение этой игры.

Добавление духа

В разделе «Код реализации поведения гномика» в главе 5 мы решили, что в момент гибели гномика должен создаваться новый объект. Поэтому вашим следующим шагом может стать создание шаблона, отображающего спрайт духа (мы включили его в набор ресурсов), всплывающего вверх. Подумайте также о возможности использования эффекта частиц для обозначения эфирного следа, оставляемого духом.

Добавление дополнительных ловушек

Мы включили в ресурсы две дополнительные ловушки: *качающийся нож* и *огнемет*. Качающийся нож — это большой нож, подвешенный на цепочке и качающийся влево-вправо. Вам придется использовать аниматор, чтобы заставить его двигаться. Огнемет — это объект, стреляющий огненными шарами; попадая в гномика, они должны вызывать функцию `FireTrapTouched` из сценария диспетчера игры Game Manager. Не забудьте подумать о добавлении спрайта, изображающего сожженный скелет гномика!

Добавление дополнительных уровней

В игре сейчас только один уровень, но нет никаких причин не добавлять другие.

Рис. 8.12. Законченная игра

Добавление дополнительных эффектов

Добавьте эффект частиц, витающих вокруг сокровища (используйте изображения Shiny1 и Shiny2). Добавьте также эффект искр, выбиваемых из стен, когда гномик касается их.

ЧАСТЬ III
Создание трехмерной игры «Метеоритный дождь»

В этой части мы создадим вторую игру. В отличие от игры, созданной в части II, события в этой игре будут разворачиваться в трехмерном пространстве. Мы создадим симулятор боя в космическом пространстве, в котором игрок должен защищать космическую станцию от астероидов. В ходе работы мы исследуем системы, часто используемые в других играх, такие как стрельба ракетами, повторное создание объектов и управление трехмерным видом моделей. Это будет круто.

9 Создание игры «Метеоритный дождь»

Движок Unity — не только превосходная платформа для двумерных игр, он также великолепно подходит для создания трехмерного контента. Unity изначально разрабатывался как трехмерный движок, и только потом в него были добавлены средства для работы с двумерной графикой, и как таковой, Unity в первую очередь ориентирован на создание трехмерных игр.

В этой главе вы узнаете, как в Unity сконструировать игру *Метеоритный дождь* — трехмерный космический симулятор. Игры этого вида были популярны в середине 1990-х, когда симуляторы, такие как *Star Wars: X-Wing* (1993) и *Descent: Freespace* (1998), дали игрокам возможность свободно перемещаться в открытом космосе, уничтожать врагов и взрывать пространство. Игры такого типа очень близки к симуляторам полетов, но так как никто не ожидает реалистичной реализации физики полета, разработчики игр могут позволить себе больше сосредоточиться на игровых аспектах.

Нельзя сказать, что аркадных симуляторов полета не существует, однако аркадный стиль более свойствен космическим симуляторам, чем реалистичным симуляторам полетов. Самым заметным исключением в последние годы стала игра *Kerbal Space Program*, настолько точно имитирующая физику полетов в космическом пространстве, что оказалась очень далека от типа игр, рассматриваемого в этой главе. Если вы хотите узнать больше о механике орбитальных полетов и о том, что происходит в удаленной точке орбиты, тогда эта игра для вас.

Поэтому можно сказать, что вместо термина «космический симулятор», часто используемого для обозначения таких игр, как в этой главе, лучше было бы использовать термин «симулятор боя в космосе».

Но хватит об условностях. Начнем стрельбу из лазерных пушек.

В следующих нескольких главах мы создадим игру, которая выглядит так, как показано на рис. 9.1.

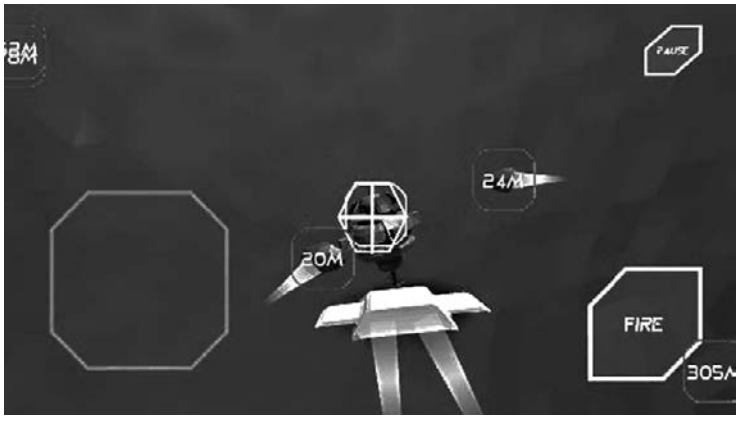

Рис. 9.1. Законченная игра

Проектирование игры

Приступая к проектированию игры, мы должны определить несколько ключевых ограничений.

- Сеанс игры не должен превышать пары минут.
- Элементы управления должны быть очень простыми и включать только элементы управления «перемещением» и «стрельбой».
- Игрок должен решать несколько краткосрочных задач, а не одну большую. То есть он должен вести бой с несколькими мелкими врагами, а не с одним большим (в противоположность игре *Колодец с сокровищами* — двумерной игре, описываемой в части II).
- Основным действием в игре должна быть стрельба в космосе из лазерных пушек. Видеоигр со стрельбой из лазерных пушек в космосе много не бывает.

Почти всегда желательно начинать размышления о высокоуровневых понятиях с их изложения на бумаге. Изложение мыслей на бумаге дает нам неструктурный подход, способствующий открытию новых идей, вписывающихся в общий план. Поэтому мы сели и набросали идею игры (рис. 9.2).

Эскиз рисовался очень быстро и получился весьма схематичным, но он позволяет увидеть некоторые дополнительные детали: астероиды летят в направлении космической станции, управление космическим кораблем осуществляется с помощью экранного джойстика, а стрельба производится с помощью кнопки «огонь». Также на эскизе видны некоторые более конкретные детали, являющиеся результатом размышлений о представлении сцены такого типа: как показать расстояние астероидов от космической станции и как игрок мог бы держать устройство.

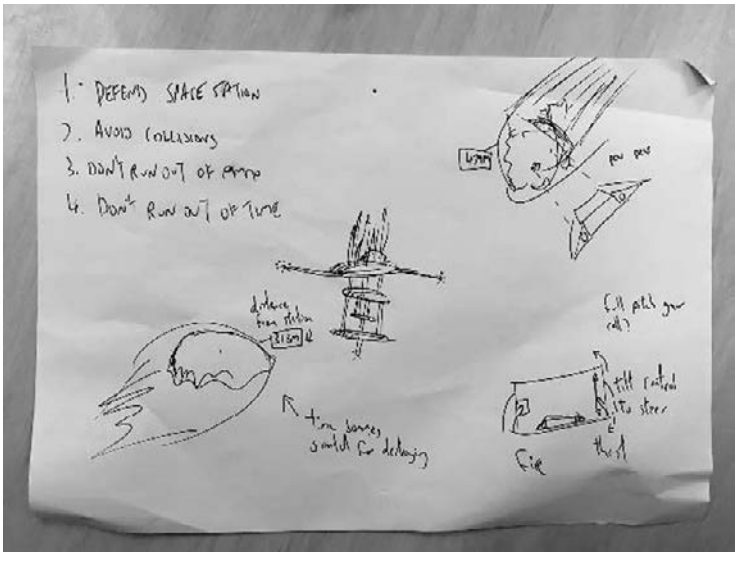

Рис. 9.2. Эскиз оригинальной идеи игры

Нарисовав этот эскиз, мы обратились к нашему другу и художнику Рексу Смилу (Rex Smeal, *https://twitter.com/RexSmeal*) с просьбой превратить грубый набросок Джона во что-то более интересное и подробное. Рисунки, созданные рукой художника, не были критически важными для проектирования игры, но они помогли нам составить более полное представление об игре. Например, мы поняли, что космической станции, которую будет защищать игрок, нужно уделить особое внимание, чтобы ее вид вызывал желание сохранить ее. Когда мы рассказали художнику о своей затее, он нарисовал набросок, изображенный на рис. 9.3; затем, после того, как мы посидели вместе и обсудили ограничения, Рекс подправил проект, и получилось то, что мы уже могли смоделировать (рис. 9.4).

Используя этот эскиз в качестве основы, мы смоделировали станцию в Blender. Работая над станцией, мы решили, что в данном случае лучше применить подход к созданию графических образов с использованием минимального количества полигонов в духе таких художников, как Хизер Пенн (Heather Penn, *https://twitter.com/heatpenn*) и Тимоти Рейнолдс (Timothy Reynolds, *http://www.turnislefthome.com*). (Нельзя сказать, что графика с минимальным количеством полигонов имеет простой вид или проста в создании, нет, но с графикой в этом стиле проще работать по тем же причинам, почему рисовать карандашом проще, чем маслом.)

Получившуюся модель станции можно увидеть на рис. 9.5. Мы также смоделировали в Blender космический корабль и астероид. Эти модели показаны на рис. 9.6 и 9.7.

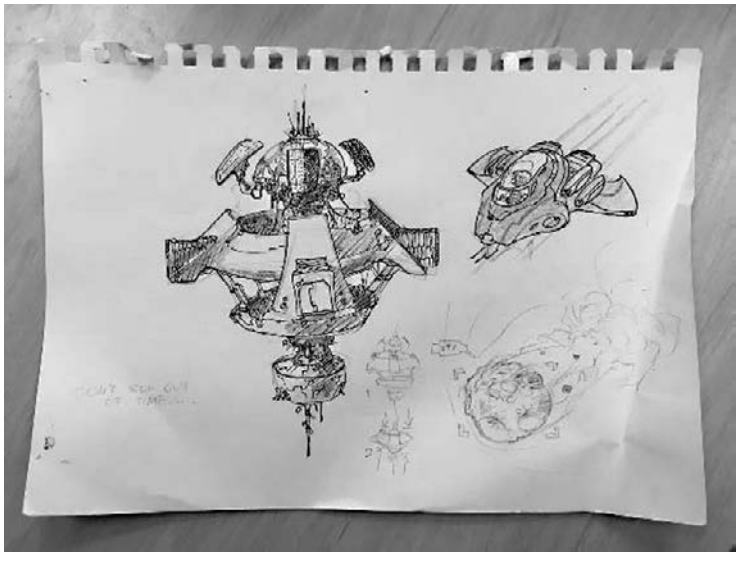

Рис. 9.3. Первоначальная идея игры, изображенная Рексом

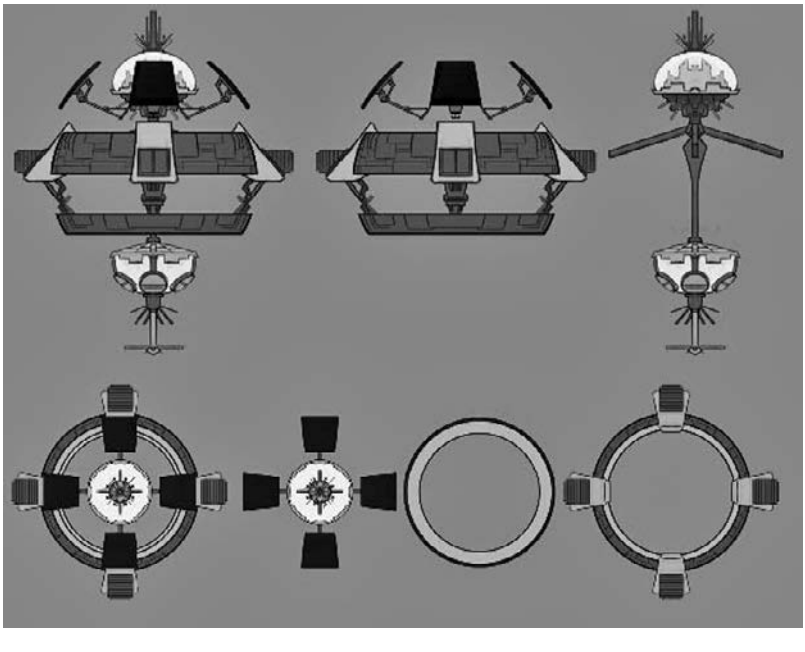

Рис. 9.4. Улучшенный эскиз космический станции, пригодный для моделирования

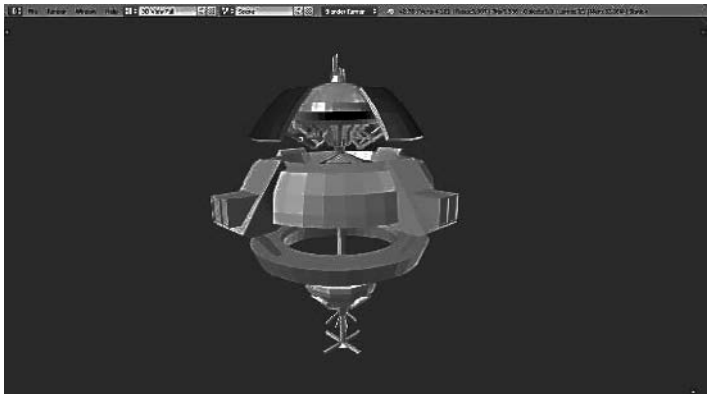

Рис. 9.5. Модель космической станции

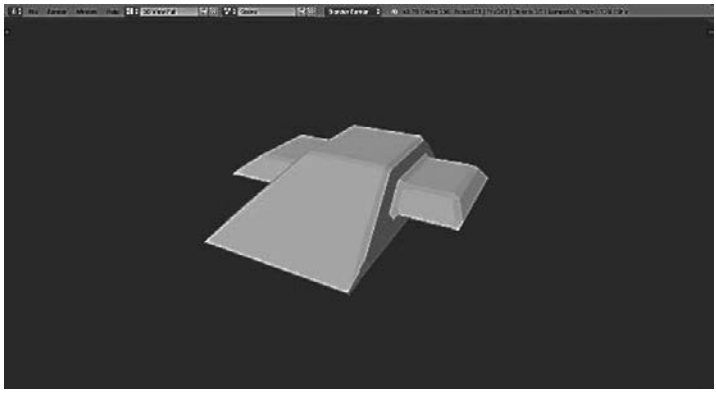

Рис. 9.6. Модель космического корабля

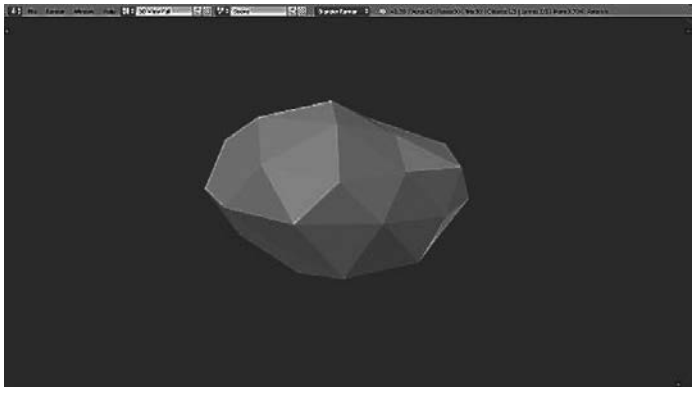

Рис. 9.7. Модель астероида

Получение ресурсов

Для создания игры вам понадобятся ресурсы, такие как звуковые эффекты, модели и текстуры. Мы упаковали все это для вас. Вам нужно просто загрузить этот пакет к себе. Файлы в пакете организованы в иерархию папок так, чтобы вам проще было отыскать нужное.

Загрузить ресурсы можно со страницы GitHub проекта: *http://www.bit.ly/rockfall-assets*.

Архитектура

Архитектура, заложенная в основу игры, очень похожа на архитектуру игры *Колодец с сокровищами*. Основные объекты игры, такие как космический корабль, управляемый игроком, и космическая станция, создаются центральным диспетчером игры; диспетчер также извещается о завершении игры, когда игрок погибает.

Пользовательский интерфейс этой игры немного сложнее, чем предыдущей. Управление игрой *Колодец с сокровищами* осуществлялось двумя кнопками и наклоном устройства; в трехмерной игре, где игрок может двигаться в любом направлении, управление наклоном малопригодно. Поэтому мы решили использовать в этой главе экранный «джойстик» — область на экране, определяющую касания и позволяющую пользователю указывать направление, двигая пальцем. Эта область будет передавать информацию о вводе общему диспетчеру ввода, использующему эту информацию для управления направлением полета.

Управление наклоном сложнее реализовать в трехмерной игре, но это не значит, что такое невозможно. Игра N.O.V.A. 3 — шутер от первого лица — поддерживает управление наклоном, позволяя игроку поворачивать своего персонажа и прицеливаться с высокой точностью. Вам стоит попробовать сыграть в эту игру, чтобы получить представление, как работает такой способ управления.

Модель полета в этой игре преднамеренно реализована немного нереалистично. Самый простой и реалистичный подход — смоделировать физический объект, определить силу тяги, толкающей его вперед, и применять физические силы для вращения корабля. Однако это усложнит управление полетом и приведет к тому, что игрок быстро потеряет интерес. Поэтому мы решили упростить физику полета: корабль всегда будет двигаться вперед с фиксированной скоростью. Кроме того, игрок не сможет вращать корабль, а любые попытки сделать это будут корректироваться (то есть, в отличие от настоящего космического пространства, пространство в игре будет поддерживать понятия «верх» и «низ»).

ПРОЕКТИРОВАНИЕ И УПРАВЛЕНИЕ

Все дизайнерские решения для игр в этой книге принимались совершенно произвольно. Даже при том, что мы решили не использовать физическую модель

полета в этой игре, это не значит, что такая модель не должна использоваться в аркадных симуляторах полета. Поиграйте в другие игры и посмотрите, какие решения приняты там. Не думайте, что все игры должны быть организованы так, как рассказали вам авторы книги. В каждой игре могут быть приняты и реализованы свои решения.

Астероиды будут реализованы как объекты, создаваемые из шаблона специальным объектом — «источником астероидов». Этот объект будет создавать астероиды и направлять их в сторону космической станции. Столкновение с астероидом будет уменьшать жизнеспособность станции; когда количество этих очков достигнет нуля, станция разрушится, и игра завершится.

Сцена

Начнем с подготовки сцены. Мы создадим новый проект Unity, а затем добавим космический корабль, способный летать в пределах сцены. Выполните следующие действия, чтобы создать основу для дальнейшей разработки игры.

1. *Создайте проект.* Создайте новый проект в Unity с именем *Rockfall* и выберите для него режим 3D (рис. 9.8).

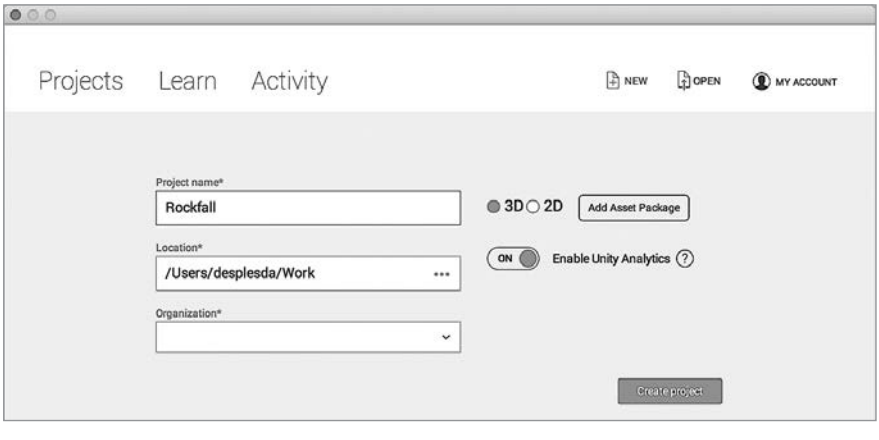

Рис. 9.8. Создание проекта

2. *Сохраните новую сцену.* После создания проекта и появления новой пустой сцены сохраните ее, открыв меню File (Файл) и выбрав пункт Save (Сохранить). Сохраните сцену с именем *Main.scene* в папке *Assets*.
3. *Импортируйте загруженные ресурсы.* Дважды щелкните на файле *.unitypackage*, загруженном в разделе «Получение ресурсов». Импортируйте в проект все ресурсы.

Теперь можно приступать к конструированию космического корабля.

Корабль

Начнем разработку игры с создания космического корабля, использовав модель из ресурсов, загруженных в разделе «Получение ресурсов».

Сам объект Ship, представляющий космический корабль, будет невидимым объектом, содержащим только сценарии; к нему будет подключено множество дочерних объектов, решающих определенные задачи отображения на экране.

1. *Создайте объект Ship.* Откройте меню GameObject (Игровой объект) и выберите пункт Create Empty (Создать пустой). В сцене появится новый игровой объект; дайте ему имя Ship.

   Теперь добавим модель корабля.

2. *Откройте папку Models и перетащите модель Ship в игровой объект Ship.* В результате в сцену будет добавлена трехмерная модель корабля, как показано на рис. 9.9. Перетащив ее в игровой объект Ship, вы сделаете ее дочерним объектом, то есть она будет перемещаться вместе с родительским игровым объектом Ship.

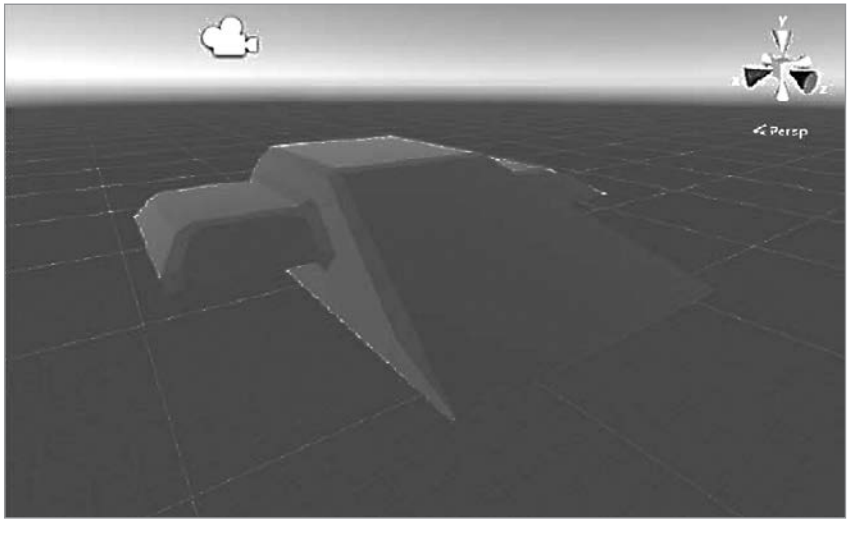

Рис. 9.9. Модель корабля в сцене

3. *Переименуйте объект модели в Graphics.*

   Далее нужно настроить объект Graphics так, чтобы он находился в одной позиции с объектом Ship.

4. *Выберите объект Graphics.* Щелкните по пиктограмме с изображением шестеренки в правом верхнем углу в разделе с настройками компонента Transform и выберите Reset Position (Сбросить позицию), как показано на рис. 9.10.

Оставьте углы поворота равными (–90, 0, 0). Это объясняется тем, что корабль моделировался в Blender, где используется система координат, отличная от Unity; в частности, направлению «вверх» в Blender соответствует ось Z, тогда как в Unity этому направлению соответствует ось Y. Чтобы исправить это, Unity автоматически поворачивает модели, созданные в Blender.

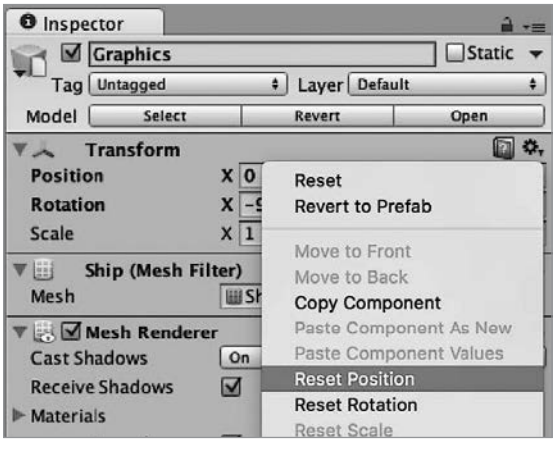

Рис. 9.10. Сброс позиции объекта Graphics

Чтобы обнаруживались столкновения корабля с другими объектами, добавим коллайдер.

5. *Добавьте в корабль компонент Box Collider.* Выберите объект Ship (то есть родителя объекта Graphics) и щелкните по кнопке Add Component (Добавить компонент) внизу инспектора. Выберите компонент Physics ▸ Box Collider.

   После добавления коллайдера установите флажок Is Trigger и установите размеры коллайдера равными (2, 1,2, 3). В результате будет создан коллайдер в форме параллелепипеда, окружающего космический корабль.

   Корабль должен двигаться вперед с постоянной скоростью. Для этого добавим сценарий, перемещающий любые объекты, к которым он будет подключен.

6. *Добавьте сценарий* `ShipThrust`. Пока выбран объект Ship, щелкните по кнопке Add Component (Добавить компонент) внизу инспектора. Создайте новый сценарий на C# в файле *ShipThrust.cs*.

   Затем откройте файл *ShipThrust.cs* и добавьте в него следующий код:

```
public class ShipThrust : MonoBehaviour {

  public float speed = 5.0f;

  // Перемещает корабль вперед с постоянной скоростью
  void Update () {
    var offset = Vector3.forward * Time.deltaTime * speed;
```

```
    this.transform.Translate(offset);
  }
}
```

Сценарий `ShipThrust` экспортирует единственный параметр `speed`, который используется в функции `Update` для перемещения объекта вперед. Величина перемещения определяется как произведение направления «вперед» в векторе на параметр скорости и на величину свойства `Time.deltaTime`, благодаря чему скорость объекта остается постоянной, независимо от числа вызовов `Update` в секунду.

Обратите внимание, что компонент сценария `ShipThrust` должен подключаться к объекту Ship, а не Graphics.

7. *Протестируйте игру.* Щелкните по кнопке Play (Играть) и посмотрите, как корабль начнет двигаться вперед.

Следование камеры

Следующий шаг — заставить камеру следовать за космическим кораблем. Сделать это можно несколькими способами: самый простой — поместить камеру в объект Ship, чтобы она двигалась одновременно с ним. Однако это не лучшее решение, потому что при таком подходе никогда не будет возникать эффекта вращения корабля относительно камеры.

Лучшее решение — оставить камеру отдельным объектом и добавить сценарий, заставляющий ее постепенно смещаться в нужную позицию. То есть когда корабль выполнит резкий поворот, потребуется некоторое время, чтобы камера компенсировала маневр, — в точности как действовал бы оператор камеры, стремящийся следовать за объектом съемки.

1. *Добавьте в главную камеру сценарий* `SmoothFollow`. Выберите главную камеру Main Camera и щелкните по кнопке Add Component (Добавить компонент). Добавьте новый сценарий на C# с именем *SmoothFollow.cs*.

   Откройте файл и добавьте в него следующий код:

```
public class SmoothFollow : MonoBehaviour
{
  // Целевой объект для следования
  public Transform target;

  // Высота камеры над целевым объектом
  public float height = 5.0f;

  // Расстояние до целевого объекта без учета высоты
  public float distance = 10.0f;

  // Насколько замедляются изменения в повороте и высоте
  public float rotationDamping;
  public float heightDamping;
```

```
  // Вызывается для каждого кадра
  void LateUpdate()
  {
    // Выйти, если цель не определена
    if (!target)
      return;

    // Вычислить желаемые местоположение и ориентацию
    var wantedRotationAngle = target.eulerAngles.y;
    var wantedHeight = target.position.y + height;

    // Выяснить текущее местоположение и ориентацию
    var currentRotationAngle = transform.eulerAngles.y;
    var currentHeight = transform.position.y;

    // Продолжить выполнять замедленный поворот вокруг оси y
    currentRotationAngle
      = Mathf.LerpAngle(currentRotationAngle,
        wantedRotationAngle,
          rotationDamping * Time.deltaTime);

    // Продолжить постепенно корректировать высоты над целью
    currentHeight = Mathf.Lerp(currentHeight,
      wantedHeight, heightDamping * Time.deltaTime);

    // Преобразовать угол в поворот
    var currentRotation
      = Quaternion.Euler(0, currentRotationAngle, 0);

    // Установить местоположение камеры в плоскости x-z
    // на расстоянии в "distance" метрах от цели
    transform.position = target.position;
    transform.position -=
      currentRotation * Vector3.forward * distance;

    // Установить местоположение камеры, используя новую высоту
    transform.position = new Vector3(transform.position.x,
      currentHeight, transform.position.z);

    // Наконец, сориентировать объектив камеры в сторону,
    // куда направляется целевой объект
    transform.rotation = Quaternion.Lerp(transform.rotation,
      target.rotation,
        rotationDamping * Time.deltaTime);

  }
}
```

Сценарий *SmoothFollow.cs*, приводящийся в этой книге, основан на коде, который генерирует Unity. Мы лишь немного адаптировали его, чтобы он лучше подходил для симулятора полетов. Если у вас появится желание увидеть оригинальную версию, вы найдете ее в пакете Utility, который можно импортировать, открыв меню Assets (Ресурсы) и выбрав пункт Import Package ▸ Utility (Импортировать пакет ▸ Utility). После импортирования вы найдете оригинальный файл *SmoothFollow.cs* в Standardard Assets ▸ Utility (Стандартные ресурсы ▸ Utility).

Сценарий `SmoothFollow` вычисляет местоположение в трехмерном пространстве, где должна находиться камера, а затем определяет точку *между* этим местоположением и местоположением, где камера находится прямо сейчас. При применении к нескольким кадрам этот сценарий создает эффект постепенного приближения камеры к требуемой точке, замедляя скорость приближения к этой точке. Кроме того, так как местоположение, где должна находиться камера, изменяется в каждом кадре, то она всегда будет немного отставать, а это именно то, чего мы хотели добиться.

2. *Настройте компонент SmoothFollow*. Перетащите объект Ship в поле Target сценария `SmoothFollow`.
3. *Протестируйте игру*. Щелкните по кнопке Play (Играть). Когда игра запустится, корабль в панели Game (Игра) больше не будет выглядеть перемещающимся, так как камера будет неотрывно следовать за ним. Убедиться в этом вы сможете, взглянув на панель сцены Scene.

Космическая станция

Процесс создания космической станции, которой угрожают летящие астероиды, идет по тому же шаблону, что и процесс создания корабля: мы должны создать пустой игровой объект и подключить к нему модель. Однако космическая станция все же немного проще космического корабля, потому что она никуда не движется: она остается на месте и подвергается ударам астероидов. Выполните следующие действия для ее создания.

1. *Создайте контейнер для космической станции*. Создайте новый пустой игровой объект с именем Space Station.
2. *Добавьте модель в качестве дочернего объекта*. Откройте папку *Models* и перетащите модель Station в игровой объект Space Station.
3. *Сбросьте местоположение объекта модели Station*. Выберите только что добавленный объект Station и щелкните правой кнопкой на компоненте Transform. Выберите пункт Reset Position (Сбросить позицию), как это делалось для модели корабля Ship.

В результате космическая станция должна выглядеть так, как показано на рис. 9.11.

После добавления модели взглянем на ее структуру и подумаем, как добавить коллайдеры. Коллайдеры необходимы для обнаружения попаданий астероидов в станцию (которые мы в итоге добавим).

Выберите объект модели и раскройте его, чтобы увидеть перечень дочерних объектов. Модель станции состоит из нескольких поверхностей (мешей); главная из них — с именем Station. Выберите ее.

Взгляните на содержимое панели инспектора. Кроме Mesh Filter и Mesh Renderer вы увидите также коллайдер Mesh Collider (рис. 9.12). Если вы не увидели его, прочитайте врезку «Модели и коллайдеры» ниже.

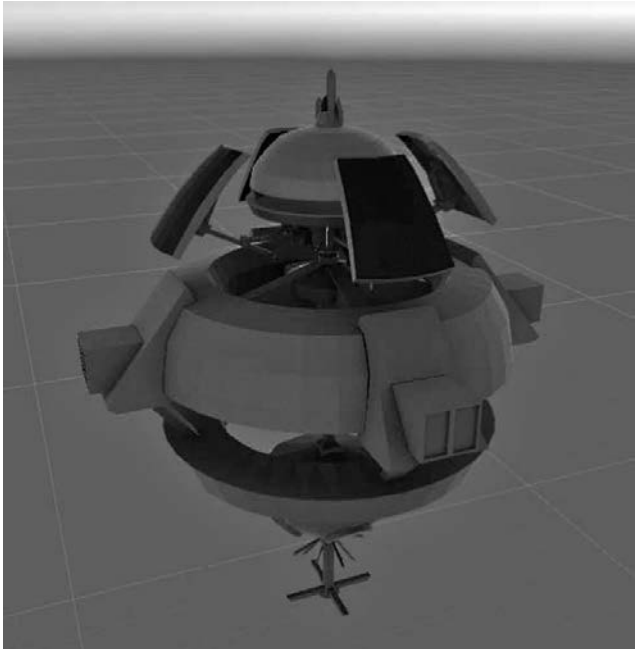

Рис. 9.11. Космическая станция

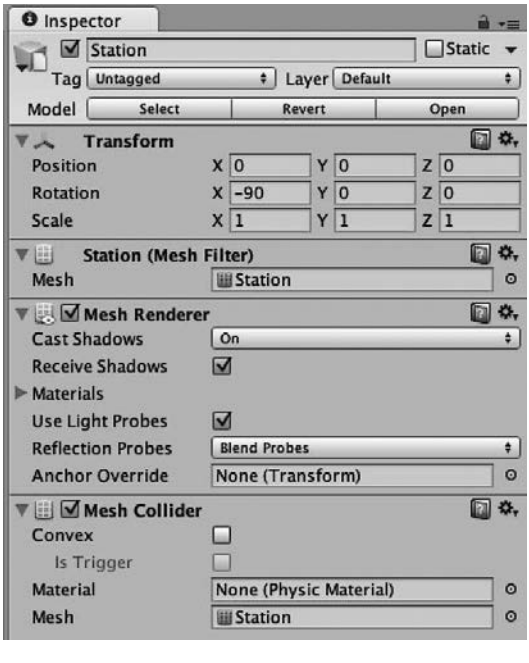

Рис. 9.12. Коллайдер космической станции

МОДЕЛИ И КОЛЛАЙДЕРЫ

При импортировании моделей Unity может автоматически создавать коллайдеры. Импортируя модель из пакета *Asset*, вы одновременно импортируете настройки, которые мы определили для нее, включая настройки, необходимые для создания коллайдеров. (Мы сделали то же самое для моделей Ship и Asteroid.)

Если вы не видите коллайдера или импортировали свою модель и хотите узнать, как настроить ее, можете посмотреть и изменить эти настройки, выбрав саму модель (то есть файл в папке *Models*) и посмотрев настройки (рис. 9.13). В частности, обратите внимание, что установлен флажок Generate Colliders (Генерировать коллайдеры).

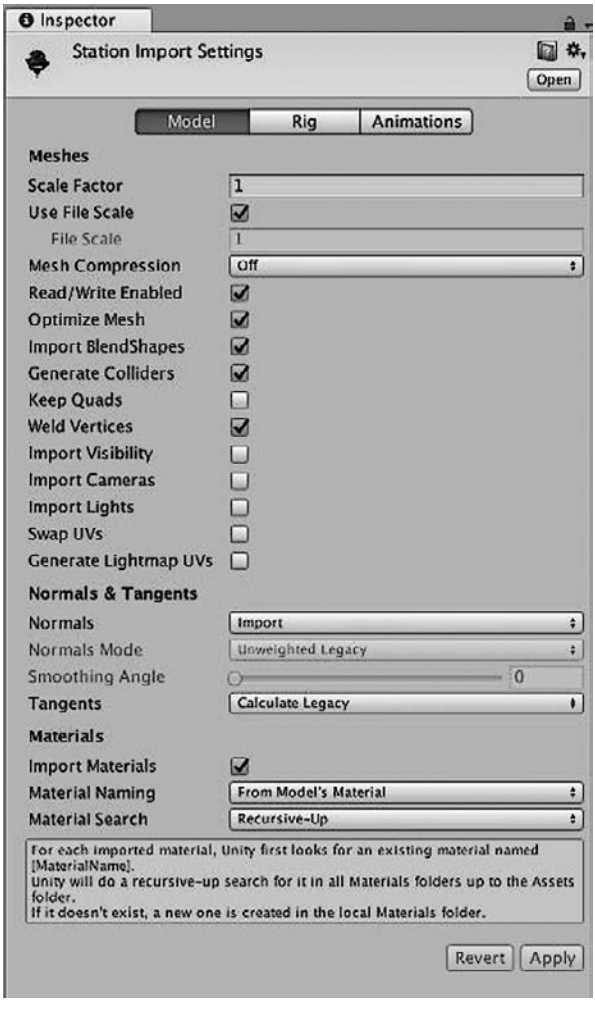

Рис. 9.13. Параметры импортирования для модели космической станции

Небо

В настоящее время в проекте по умолчанию используется скайбокс (или «небесный куб», от англ. skybox — трехмерное изображение неба и горизонта), предназначенный для игр, события в которых развиваются на поверхности планеты. Замена его на скайбокс, специально предназначенный для изображения космического пространства, совершенно необходима для создания правильного восприятия игры.

Скайбокс — это виртуальный куб, который всегда отображается позади всего, что есть в сцене, и никогда не перемещается относительно камеры. Это создает ощущение, что текстуры, натянутые на этот куб, находятся бесконечно далеко. Так и появилось название «небесный куб».

Чтобы создать иллюзию, что игрок находится внутри сферы, а не куба, текстуры должны искажаться так, чтобы не было видно швов на ребрах. Существует множество способов добиться этого. Есть даже несколько плагинов для Adobe Photoshop. Однако большая их часть предназначена для деформирования фотографий, которые могли бы быть у вас или у других. Трудно получить фотографии космического пространства, которые хорошо подходили бы для видеоигр, поэтому проще воспользоваться инструментом, создающим искусственные изображения.

Создание скайбокса

После создания или получения изображений для скайбокса можно добавить их в игру. Для этого нужно добавить материал скайбокса, а затем включить этот материал в настройки освещения сцены. Вот что для этого нужно сделать.

1. *Создайте материал Skybox.* Создайте новый материал, открыв меню Assets (Ресурсы) и выбрав пункт Create ▸ Material (Создать ▸ Материал). Дайте ему имя Skybox и поместите его в папку *Skybox*.
2. *Настройте материал.* Выберите материал и измените значение поля Shader (Шейдер) со Standard (Стандартный) на Skybox ▸ 6 Sided (Скайбокс ▸ Шестигранный). Вид инспектора изменится так, что у вас появится возможность подключить шесть текстур (рис. 9.14).

   Найдите текстуры для скайбокса в папке *Skybox*. Перетащите их по одной в соответствующие поля: текстуру Up — в поле Up, текстуру Front — в поле Front и т. д.

   В результате настройки в инспекторе должны выглядеть так, как на рис. 9.15.
3. *Подключите скайбокс к настройкам освещения.* Откройте меню Window (Окно) и выберите пункт Lighting ▸ Settings (Освещение ▸ Настройки). На экране появится панель Lighting (Освещение). В верхней части панели вы увидите поле с подписью «Skybox». Перетащите в это поле только что созданный материал Skybox (рис. 9.16).

В итоге небо заменят изображения космического пространства (как показано на рис. 9.17). Кроме того, система освещения из Unity будет использовать информацию из скайбокса, чтобы определить, как освещать объекты; если приглядеться внимательнее, то можно заметить, что космический корабль и станция приобрели зеленоватый оттенок, потому что изображения неба имеют зеленый цвет.

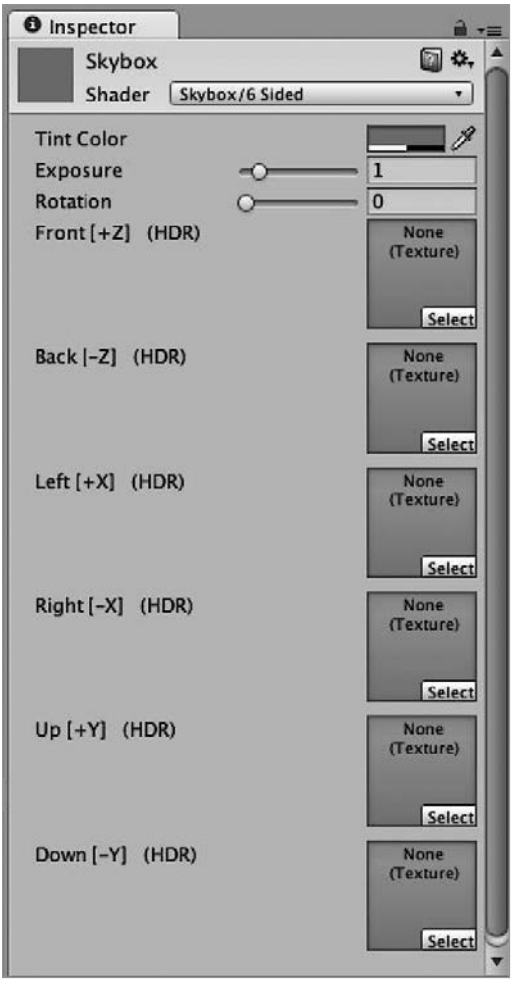

Рис. 9.14. Настройки скайбокса без текстур

Холст

В настоящее время космический корабль движется в пространстве только вперед, потому что игрок пока не имеет возможности управлять полетом. Вскоре мы добавим элементы пользовательского интерфейса для этого, но прежде нужно создать и настроить холст (canvas), на котором эти элементы будут отображаться. Для этого выполните следующие действия.

1. *Создайте холст Canvas.* Откройте меню GameObject (Игровой объект) и выберите пункт UI ▸ Canvas (Пользовательский интерфейс ▸ Холст). В результате будут созданы два объекта: Canvas и EventSystem.
2. *Настройте холст.* Выберите игровой объект Canvas, а в инспекторе в настройках подключенного компонента Canvas найдите параметр Render Mode (Режим

отображения). Выберите в нем значение Screen Space - Camera (Экранное пространство — Камера). После этого появятся новые параметры, характерные для данного режима отображения.

Перетащите главную камеру Main Camera в поле Render Camera (Камера для отображения) и установите свойство Plane Distance (Расстояние до плоскости) в значение 1 (рис. 9.18). В результате холст для отображения элементов пользовательского интерфейса будет отображаться на расстоянии в одну единицу от камеры.

В свойстве UI Scale Mode (Режим масштабирования пользовательского интерфейса) в разделе Canvas Scaler выберите Scale with Screen Size (Масштабировать с изменением размера экрана) и установите контрольное разрешение (Reference Resolution) 1024 × 768, которое соответствует разрешению экрана iPad.

Теперь холст готов, и можно начинать добавлять в него элементы пользовательского интерфейса.

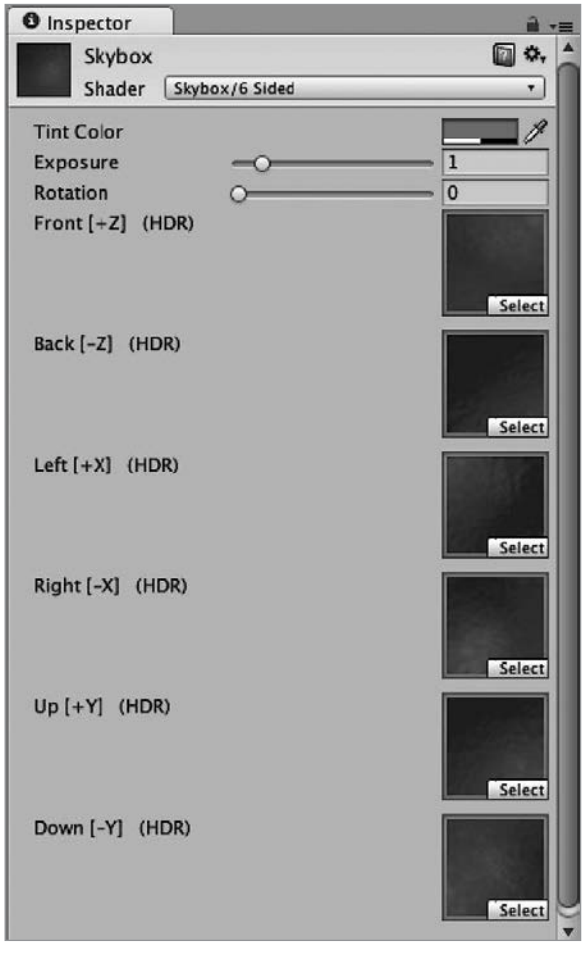

Рис. 9.15. Настройки скайбокса с подключенными текстурами

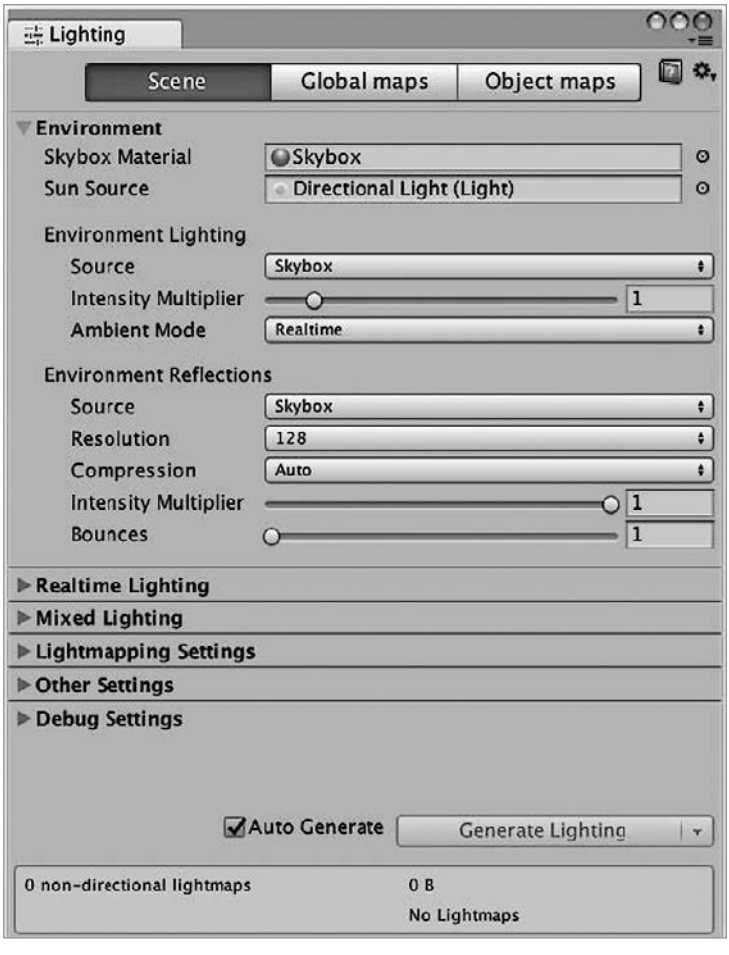

Рис. 9.16. Создание настроек освещения

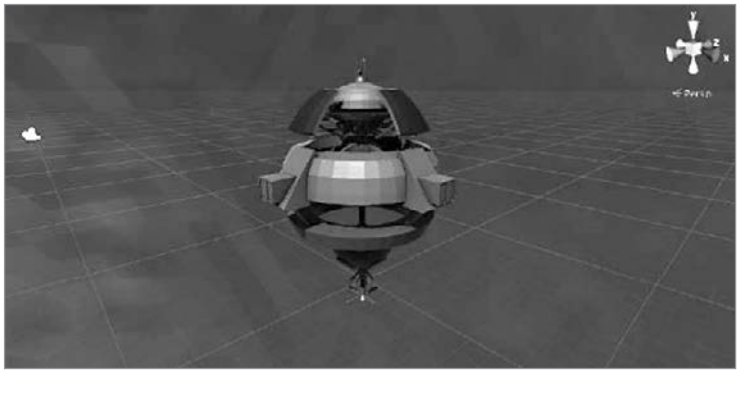

Рис. 9.17. Скайбокс в сцене

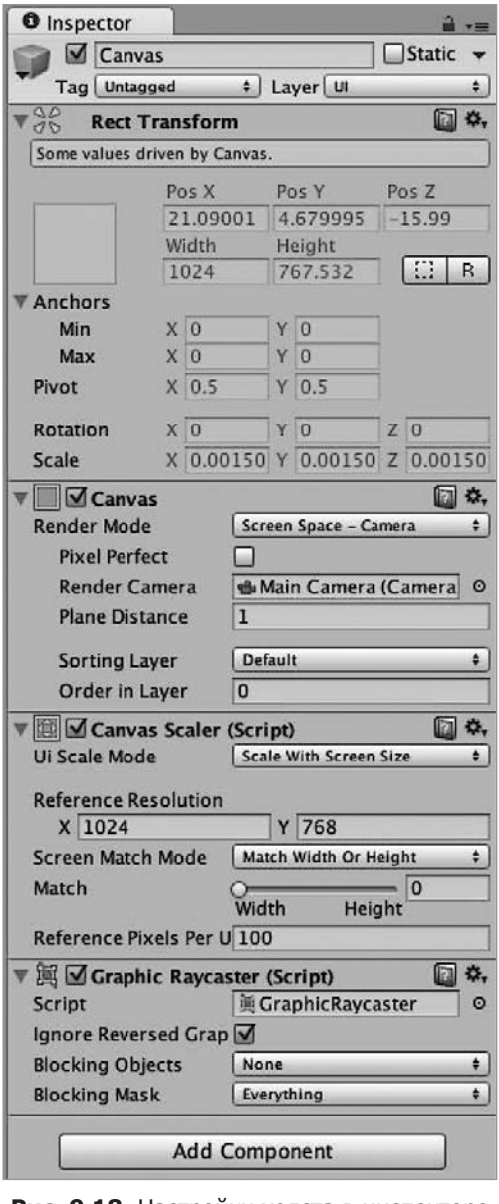

Рис. 9.18. Настройки холста в инспекторе

В заключение

Сцена готова, и теперь мы готовы к реализации систем, необходимых для игры. В следующей главе мы углубимся в темноту космоса и реализуем систему управления полетом космического корабля.

10 Ввод и управление полетом

После создания сцены пришло время добавить в нее элемент игры. В этой главе мы приступим к созданию систем управления кораблем в космическом пространстве.

Ввод

В этой игре присутствуют два устройства ввода: виртуальный джойстик, позволяющий игроку управлять направлением полета корабля, и кнопка, сигнализирующая о моментах, когда пользователь ведет стрельбу из лазерных пушек.

Не забывайте, что проверить управление игрой посредством сенсорного экрана можно только на устройстве с сенсорным экраном. Чтобы протестировать игру без сборки на устройстве, используйте приложение Unity Remote (см. врезку «Unity Remote» в главе 5).

Добавление джойстика

Начнем с создания джойстика. Джойстик состоит из двух видимых компонентов: большого квадрата «контактной панели» в левом нижнем углу холста и маленькой «площадки» в центре этого квадрата. Когда пользователь помещает палец в границы контактной панели, джойстик меняет свое местоположение так, чтобы площадка оказалась точно под пальцем и одновременно в центре панели. Когда пользователь будет двигать пальцем, вместе с ним будет двигаться площадка. Чтобы приступить к созданию системы ввода, выполните следующие действия.

1. *Создайте контактную панель.* Откройте меню GameObject (Игровой объект) и выберите пункт UI ▸ Panel (Пользовательский интерфейс ▸ Панель). Дайте новой панели имя Joystick.

   Поместите панель в левый нижний угол экрана и придайте ей квадратную форму. Привяжите ее к левому нижнему углу, выбрав вариант Bottom Left после щелчка по кнопке Anchor (Привязка). Затем установите ширину и высоту панели равными 250.

2. *Добавьте изображение в контактную панель*. Выберите в свойстве Source Image компонента Image спрайт Pad.

В итоге контактная панель должна выглядеть так, как показано на рис. 10.1.

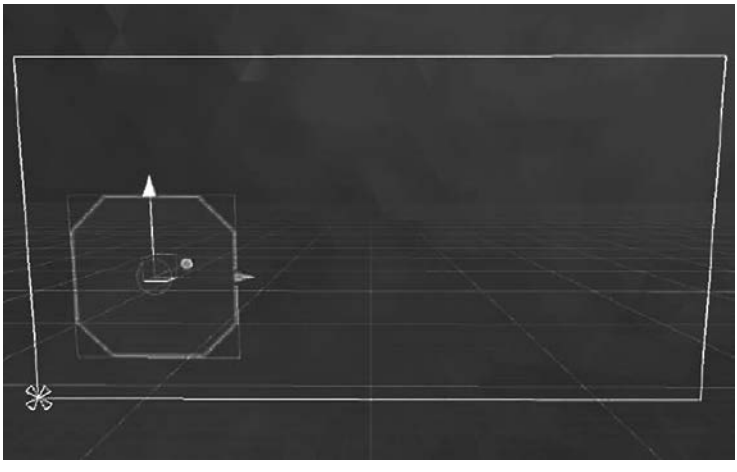

Рис. 10.1. Контактная панель джойстика

3. *Создайте площадку для пальца*. Создайте второй объект панели и дайте ему имя Thumb.

   Сделайте площадку дочерним объектом по отношению к объекту Joystick. Привяжите ее к середине в центре и установите ширину и высоту равными 80. Установите свойства Pos X и Pos Y на 0. В результате площадка окажется в центре контактной панели. Наконец, выберите в свойстве Source Image спрайт Thumb.

4. *Добавьте сценарий* `VirtualJoystick`. Выберите объект Joystick и добавьте новый сценарий на C# с именем *VirtualJoystick.cs*. Откройте файл и добавьте в него следующий код:

```
// Получить доступ к интерфейсам Event
using UnityEngine.EventSystems;

// Получить доступ к элементам пользовательского интерфейса
using UnityEngine.UI;

public class VirtualJoystick : MonoBehaviour,
  IBeginDragHandler, IDragHandler, IEndDragHandler {

  // Спрайт, перемещаемый по экрану
  public RectTransform thumb;

  // Местоположение пальца и джойстика, когда
  // происходит перемещение
  private Vector2 originalPosition;
```

```
private Vector2 originalThumbPosition;

// Расстояние, на которое сместился палец относительно
// исходного местоположения
public Vector2 delta;

void Start () {
  // В момент запуска запомнить исходные
  // координаты
  originalPosition
    = this.GetComponent<RectTransform>().localPosition;
  originalThumbPosition = thumb.localPosition;

  // Выключить площадку, сделав ее невидимой
  thumb.gameObject.SetActive(false);

  // Сбросить величину смещения в ноль
  delta = Vector2.zero;
}

// Вызывается, когда начинается перемещение
public void OnBeginDrag (PointerEventData eventData) {

  // Сделать площадку видимой
  thumb.gameObject.SetActive(true);

  // Зафиксировать мировые координаты, откуда начато перемещение
  Vector3 worldPoint = new Vector3();
  RectTransformUtility.ScreenPointToWorldPointInRectangle(
    this.transform as RectTransform,
    eventData.position,
    eventData.enterEventCamera,
    out worldPoint);

  // Поместить джойстик в эту позицию
  this.GetComponent<RectTransform>().position
    = worldPoint;

  // Поместить площадку в исходную позицию
  // относительно джойстика
  thumb.localPosition = originalThumbPosition;
}

// Вызывается в ходе перемещения
public void OnDrag (PointerEventData eventData) {

  // Определить текущие мировые координаты точки контакта пальца с экраном
  Vector3 worldPoint = new Vector3();
  RectTransformUtility.ScreenPointToWorldPointInRectangle(
    this.transform as RectTransform,
    eventData.position,
    eventData.enterEventCamera,
    out worldPoint);
```

```
      // Поместить площадку в эту точку
      thumb.position = worldPoint;

      // Вычислить смещение от исходной позиции
      var size = GetComponent<RectTransform>().rect.size;

      delta = thumb.localPosition;

      delta.x /= size.x / 2.0f;
      delta.y /= size.y / 2.0f;

      delta.x = Mathf.Clamp(delta.x, -1.0f, 1.0f);
      delta.y = Mathf.Clamp(delta.y, -1.0f, 1.0f);

    }

    // Вызывается по окончании перемещения
    public void OnEndDrag (PointerEventData eventData) {
      // Сбросить позицию джойстика
      this.GetComponent<RectTransform>().localPosition
        = originalPosition;

      // Сбросить величину смещения в ноль
      delta = Vector2.zero;

      // Скрыть площадку
      thumb.gameObject.SetActive(false);
    }
  }
```

Класс `VirtualJoystick` реализует три ключевых интерфейса C#: `IBeginDragHandler`, `IDragHandler` и `IEndDragHandler`. Когда игрок начинает двигать пальцем по экрану, продолжает и завершает движение в любой точке в пределах панели джойстика Joystick, автоматически вызываются методы `OnBeginDrag`, `OnDrag` и `OnEndDrag` соответственно. Эти методы принимают единственный параметр: объект `PointerEventData`, содержащий, кроме всего прочего, информацию о местоположении пальца на экране.

- Когда *начинается* движение пальца по экрану, контактная панель позиционируется так, чтобы ее центр оказался точно под пальцем.
- Когда движение *продолжается*, площадка движется синхронно с пальцем, при этом измеряется величина смещения площадки от центра контактной панели и сохраняется в свойстве `delta`.
- Когда движение *заканчивается* (то есть когда пользователь отрывает палец от экрана), контактная панель и площадка возвращаются в исходное местоположение, а свойство `delta` сбрасывается в ноль.

Чтобы завершить создание системы ввода, выполните следующие действия.

1. *Настройте джойстик.* Выберите объект Joystick и перетащите в его свойство Thumb объект Thumb.

2. *Протестируйте джойстик.* Запустите игру, нажмите кнопку мыши в пределах контактной панели и, удерживая ее нажатой, передвиньте в любую сторону. С началом движения контактная панель переместится один раз, а площадка будет продолжать двигаться, синхронно с перемещениями указателя. Обратите внимание на значение свойства Delta объекта Joystick — оно должно изменяться с перемещением площадки.

Диспетчер ввода

Теперь, закончив настройку джойстика, организуем передачу информации от джойстика космическому кораблю, чтобы тот мог изменять направление полета.

Мы *могли бы* напрямую связать корабль с джойстиком, но возникает одна проблема. В процессе игры корабль может быть уничтожен, и будет создан новый корабль. Чтобы такое стало возможным, корабль придется превратить в шаблон — в этом случае диспетчер игры сможет создать множество его копий. Однако шаблоны не могут ссылаться на объекты в сцене, то есть вновь созданный объект корабля не будет иметь ссылки на джойстик.

Лучшее решение — создать объект-одиночку (singleton) диспетчера ввода, всегда присутствующий в сцене и *всегда имеющий* доступ к джойстику. Так как этот объект не является экземпляром шаблона, нам не придется беспокоиться о потере ссылки при его создании. Когда будет создаваться новый корабль, он сможет использовать диспетчер ввода (доступный программно) для получения информации из джойстика.

1. *Создайте сценарий* `Singleton`. Создайте новый сценарий на C# с именем *Singleton.cs* в папке *Assets*. Откройте этот файл и введите следующий код:

```
// Этот класс позволяет другим объектам ссылаться на единственный
// общий объект. Его используют классы GameManager и InputManager.

// Чтобы воспользоваться этим классом, унаследуйте его:
// public class MyManager : Singleton<MyManager> { }

// После этого вы сможете обращаться к единственному общему
// экземпляру класса так:
// MyManager.instance.DoSomething();

public class Singleton<T> : MonoBehaviour
  where T : MonoBehaviour {

  // Единственный экземпляр этого класса.
  private static T _instance;

  // Метод доступа. При первом вызове настраивает _instance.
  // Если требуемый объект не найден,
  // выводит сообщение об ошибке в журнал.
  public static T instance {
    get {
      // Если свойство _instance еще не настроено..
      if (_instance == null)
```

```
        {
          // ...попытаться найти объект.
          _instance = FindObjectOfType<T>();

          // Записать сообщение об ошибке в случае неудачи.
          if (_instance == null) {
            Debug.LogError("Can't find "
              + typeof(T) + "!");
          }
        }

        // Вернуть экземпляр для использования!
        return _instance;
      }
    }

}
```

Этот сценарий `Singleton` идентичен сценарию Singleton, использовавшемуся в игре *Колодец с сокровищами*. Подробное описание его работы вы найдете в разделе «Создание класса Singleton» в главе 5.

2. *Создайте диспетчер ввода*. Создайте новый пустой игровой объект с именем Input Manager. Добавьте в него новый сценарий на C# с именем *InputManager.cs*. Откройте файл и добавьте следующий код:

```
public class InputManager : Singleton<InputManager> {

  // Джойстик, используемый для управления кораблем.
  public VirtualJoystick steering;

}
```

   В данный момент `InputManager` играет роль простого объекта данных: он просто хранит ссылку на `VirtualJoystick`. Позднее мы добавим в него дополнительную логику для поддержки стрельбы из текущего оружия корабля.

3. *Настройте диспетчер ввода*. Перетащите объект Joystick в поле Steering.

   Теперь, настроив диспетчер ввода, мы готовы использовать его для управления полетом корабля.

Управление полетом

В данный момент корабль может двигаться только вперед. Чтобы изменить направление движения корабля, мы должны изменить его направление «вперед». Для этого мы будем извлекать информацию из виртуального джойстика и в соответствии с ней изменять ориентацию корабля в пространстве.

В каждом кадре корабль использует направление, определяемое джойстиком, и значение, задающее скорость поворота корабля, чтобы получить новую величину поворота. Затем эта информация объединяется с *текущей* ориентацией корабля, и в результате получается его новое направление «вперед».

Однако нам совсем не нужно, чтобы игрок перевернулся и потерял направление на важные объекты, такие как космическая станция. Чтобы избежать этой проблемы, сценарий управления также применяет *дополнительное* вращение, которое медленно поворачивает корабль обратно. Это поведение немного напоминает поведение самолета, летящего в атмосфере, и более понятно игроку (хотя и в ущерб реалистичности).

1. *Добавьте сценарий* `ShipSteering`. Выберите объект Ship и добавьте новый сценарий на C# с именем *ShipSteering.cs*. Откройте файл и добавьте следующий код:

```
public class ShipSteering : MonoBehaviour {

  // Скорость поворота корабля
  public float turnRate = 6.0f;

  // Сила выравнивания корабля
  public float levelDamping = 1.0f;

  void Update () {

    // Создать новый поворот, умножив вектор направления джойстика
    // на turnRate, и ограничить величиной 90 % от половины круга.

    // Сначала получить ввод пользователя.
    var steeringInput
      = InputManager.instance.steering.delta;

    // Теперь создать вектор для вычисления поворота.
    var rotation = new Vector2();

    rotation.y = steeringInput.x;
    rotation.x = steeringInput.y;

    // Умножить на turnRate, чтобы получить величину поворота.
    rotation *= turnRate;

    // Преобразовать в радианы, умножив на 90 %
    // половины круга
    rotation.x = Mathf.Clamp(
      rotation.x, -Mathf.PI * 0.9f, Mathf.PI * 0.9f);

    // И преобразовать радианы в кватернион поворота!
    var newOrientation = Quaternion.Euler(rotation);

    // Объединить поворот с текущей ориентацией
    transform.rotation *= newOrientation;

    // Далее попытаться минимизировать поворот!
```

```
        // Сначала определить, какой была бы ориентация
        // в отсутствие вращения относительно оси Z
        var levelAngles = transform.eulerAngles;
        levelAngles.z = 0.0f;
        var levelOrientation = Quaternion.Euler(levelAngles);

        // Объединить текущую ориентацию с небольшой величиной
        // этой ориентации "без вращения"; когда это происходит
        // на протяжении нескольких кадров, объект медленно
        // выравнивается над поверхностью
        transform.rotation = Quaternion.Slerp(
          transform.rotation, levelOrientation,
          levelDamping * Time.deltaTime);

    }
}
```

Сценарий `ShipSteering` использует информацию из джойстика для вычисления нового сглаженного угла поворота и применяет его к кораблю. После этого используется небольшое усилие, вынуждающее корабль выравниваться.

2. *Протестируйте управление*. Запустите игру. Корабль начнет полет вперед; когда вы щелкнете в области джойстика и, удерживая кнопку мыши, начнете перемещать указатель, корабль изменит направление полета. Таким способом можно заставить корабль лететь в любую сторону. Обратите внимание, что после поворота (например, если резко задрать нос корабля и повернуть в сторону) корабль попытается вернуться в прежнее положение полета над воображаемой поверхностью.

Индикаторы

Так как основу этой игры составляет полет в трехмерном пространстве, очень легко потерять след разных объектов в игре. Космическая станция находится под угрозой попадания астероидов, и игрок должен знать, где находится она и где находятся астероиды.

Чтобы решить эту проблему, мы реализуем систему индикаторов на экране, подсказывающих местоположение важных объектов. Объекты, находящиеся в поле зрения камеры, будут обводиться окружностями. Индикаторы, соответствующие объектам, которые находятся за границами экрана, будут выводиться на границах, подсказывая направление поворота, чтобы увидеть их.

Создание элементов пользовательского интерфейса

Для начала создадим объект внутри холста, который будет действовать как контейнер для всех индикаторов. Затем мы создадим индикатор и превратим его в шаблон для многократного использования. Чтобы реализовать этот план, выполните следующие действия.

1. *Создайте контейнер для индикаторов.* Выберите объект Canvas и создайте новый пустой дочерний объект, открыв меню GameObject (Игровой объект) и выбрав пункт Create Empty Child (Создать пустой дочерний объект). В результате будет создан новый объект со свойством Rect Transform (которое используется для двумерных объектов, таких как элементы холста) в противоположность объекту с обычным свойством Transform (которое используется для трехмерных объектов). Установите точки привязки объекта так, чтобы он растянулся на весь экран по горизонтали и вертикали.

   Дайте новому объекту имя Indicators.

2. *Создайте прототип индикатора.* Создайте новый объект Image, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Image (Пользовательский интерфейс ▸ Изображение).

   Дайте новому объекту имя Position Indicator. Сделайте его дочерним по отношению к объекту Indicators, созданному на предыдущем шаге.

   Перетащите спрайт Indicator из папки UI в поле Source Image.

3. *Создайте текстовую надпись.* Создайте новый объект Text (открыв меню GameObject (Игровой объект) и подменю UI (Пользовательский интерфейс)). Сделайте этот объект Text дочерним по отношению к объекту спрайта Position Indicator.

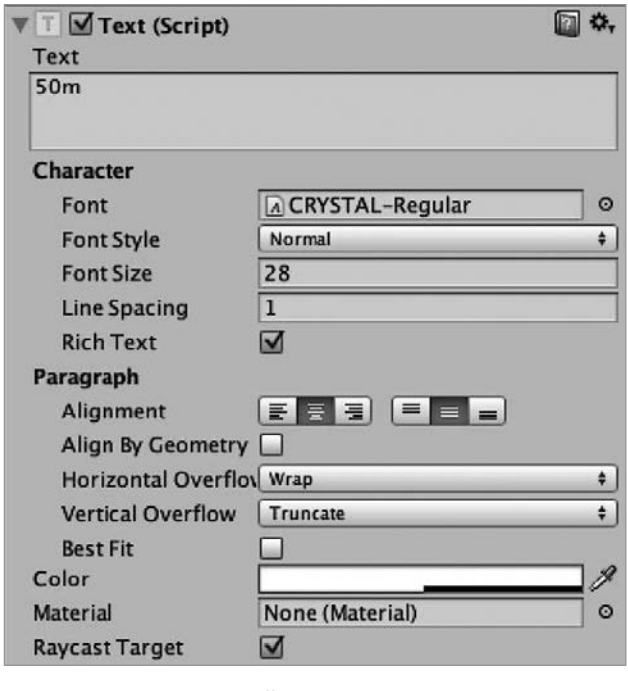

Рис. 10.2. Настройки надписи на индикаторе

Выберите белый цвет для текста и задайте выравнивание по центру, по горизонтали и по вертикали.

Задайте текст надписи «50m». (Текст будет меняться в ходе игры, но, задав текст, вы лучше будете представлять, как выглядит индикатор.)

Привяжите объект Text к середине в центре, выбрав вариант center-middle, и установите его координаты X и Y на ноль. Благодаря этому текст окажется в центре спрайта.

Наконец, мы будем использовать нестандартный шрифт для индикаторов. Найдите шрифт CRYSTAL-Regular в папке *Fonts* и перетащите его в поле Font объекта Text. Затем измените свойство Font Size (Размер шрифта) в значение 28.

В результате настройки компонента Text в инспекторе должны выглядеть так, как показано на рис. 10.2, а сам объект индикатора должен иметь вид как на рис. 10.3.

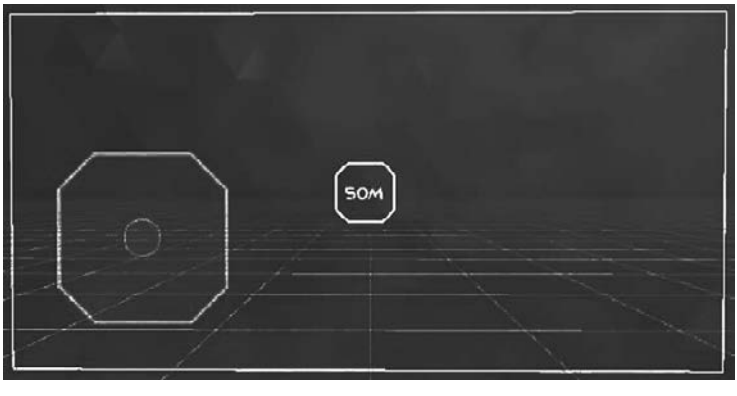

Рис. 10.3. Прототип индикатора

4. *Добавьте код.* Добавьте в объект прототипа Indicator новый сценарий на C# с именем *Indicator.cs* и введите в него следующий код:

```
// Получить доступ к классам UI
using UnityEngine.UI;

public class Indicator : MonoBehaviour {

  // Отслеживаемый объект.
  public Transform target;

  // Расстояние от 'target' до данного объекта.
  public Transform showDistanceTo;

  // Надпись для отображения расстояния.
  public Text distanceLabel;

  // Расстояние от края экрана.
```

```
public int margin = 50;

// Цвет оттенка изображения.
public Color color {
  set {
    GetComponent<Image>().color = value;
  }
  get {
    return GetComponent<Image>().color;
  }
}

// Выполняет настройку индикатора
void Start() {
  // Скрыть надпись; она будет сделана видимой
  // в методе Update, если цель target будет назначена
  distanceLabel.enabled = false;

  // На запуске дождаться ближайшего кадра перед отображением
  // для предотвращения визуальных артефактов
  GetComponent<Image>().enabled = false;

}

// Обновляет положение индикатора в каждом кадре
void Update()
{

  // Цель исчезла? Если да, значит, индикатор тоже надо убрать
  if (target == null) {
    Destroy (gameObject);
    return;
  }

  // Если цель присутствует, вычислить расстояние до нее
  // и показать в distanceLabel
  if (showDistanceTo != null) {

    // Показать надпись
    distanceLabel.enabled = true;

    // Вычислить расстояние
    var distance = (int)Vector3.Magnitude(
      showDistanceTo.position - target.position);

    // Показать расстояние в надписи
    distanceLabel.text = distance.ToString() + "m";
  } else {
    // Скрыть надпись
    distanceLabel.enabled = false;
  }


  GetComponent<Image>().enabled = true;

  //Определить экранные координаты объекта
  var viewportPoint =
```

```
        Camera.main.WorldToViewportPoint(target.position);

      // Объект позади нас?
      if (viewportPoint.z < 0) {
        // Сместить к границе экрана
        viewportPoint.z = 0;
        viewportPoint = viewportPoint.normalized;
        viewportPoint.x *= -Mathf.Infinity;
      }

      // Определить видимые координаты для индикатора
      var screenPoint =
        Camera.main.ViewportToScreenPoint(viewportPoint);

      // Ограничить краями экрана
      screenPoint.x = Mathf.Clamp(
        screenPoint.x,
        margin,
        Screen.width - margin * 2);

      screenPoint.y = Mathf.Clamp(
        screenPoint.y,
        margin,
        Screen.height - margin * 2);

      // Определить, где в области холста находится видимая координата
      var localPosition = new Vector2();
      RectTransformUtility.ScreenPointToLocalPointInRectangle(
        transform.parent.GetComponent<RectTransform>(),
        screenPoint,
        Camera.main,
        out localPosition);

      // Обновить позицию индикатора
      var rectTransform = GetComponent<RectTransform>();
      rectTransform.localPosition = localPosition;

    }
  }
```

Вот как работает индикатор.

- В методе `Update` в каждом кадре трехмерные координаты объекта, которому соответствует индикатор, преобразуются в координаты *видимой области* (viewport space).

  Координаты в видимой области представляют позицию на экране, где (0,0,0) соответствует левому нижнему углу экрана, а (1,1,0) — правому верхнему углу. Компонента Z координат видимой области представляет расстояние до камеры в мировых единицах.

  То есть мы легко можем сказать, находится ли точка с известными координатами на экране или нет, а также позади нас она или нет. Если видимые координ-

наты X и Y объекта находятся за пределами диапазона от (0,0) до (1,1), значит, объект располагается за пределами экрана, а если координата Z меньше 0, значит, объект располагается позади.

- Если объект находится позади (то есть координата Z меньше нуля), мы должны сдвинуть индикатор к границе экрана. Если этого не сделать, то индикатор, соответствующий объекту прямо сзади нас, появился бы в центре экрана, и игрок мог бы по ошибке подумать, что объект находится спереди.

  Чтобы сдвинуть индикатор к краю, видимая координата X умножается на отрицательную бесконечность; при умножении координаты X на бесконечность индикатор всегда оказывается далеко за левым или за правым краем экрана. Мы умножаем на отрицательную бесконечность, чтобы компенсировать факт нахождения за нами.

- Далее видимые координаты преобразуются в экранные координаты и затем ограничиваются заданным диапазоном, чтобы они никогда не могли выйти за пределы окна. Чтобы чуть сдвинуть индикатор и обеспечить читаемость текста, обозначающего расстояние, используется параметр отступа `margin`.

- Наконец, экранные координаты преобразуются в координаты внутри контейнера индикаторов, а затем используются для обновления позиции индикатора. После этого индикатор оказывается в нужной позиции.

Индикаторы также отвечают за удаление самих себя: в каждом кадре они проверяют свое свойство `target` на равенство `null` и, если условие выполняется, удаляют себя.

В настройке индикаторов осталось выполнить один заключительный шаг. Завершив создание прототипа индикатора, его нужно преобразовать в шаблон.

1. *Подключите надпись, отображающую расстояние*. Перетащите дочерний объект Text в поле Distance Label.

2. *Преобразуйте прототип в шаблонный объект (prefab)*. Перетащите объект Position Indicator в панель обозревателя проекта. В результате будет создан новый шаблонный объект (prefab), используя который можно создать множество индикаторов во время выполнения.

   После создания шаблона удалите прототип из сцены.

Диспетчер индикаторов

Диспетчер индикаторов — это объект-одиночка, управляющий процессом создания индикаторов. Этот объект будет использоваться любыми другими объектами, которым потребуется добавить индикатор на экран, в частности космической станцией и астероидами.

Сделав этот объект одиночкой, мы получаем возможность создать и настроить объект в сцене, не совершая сложных действий, чтобы дать возможность объектам, создаваемым из шаблонов, обращаться к диспетчеру.

1. *Создайте диспетчер индикаторов.* Создайте новый пустой объект с именем Indicator Manager.
2. *Добавьте сценарий* `IndicatorManager`. Добавьте в объект новый сценарий на C# с именем *IndicatorManager.cs* и введите в него следующий код:

```
using UnityEngine.UI;

public class IndicatorManager : Singleton<IndicatorManager> {

  // Объект, потомками которого будут все индикаторы
  public RectTransform labelContainer;

  // Шаблон для создания индикаторов
  public Indicator indicatorPrefab;

  // Этот метод будет вызываться другими объектами
  public Indicator AddIndicator(GameObject target,
    Color color, Sprite sprite = null) {

    // Создать объект индикатора
    var newIndicator = Instantiate(indicatorPrefab);

    // Связать его с целевым объектом
    newIndicator.target = target.transform;

    // Обновить его цвет
    newIndicator.color = color;

    // Если задан спрайт, установить его как изображение
    // для данного индикатора
    if (sprite != null) {
      newIndicator
        .GetComponent<Image>().sprite = sprite;
    }

    // Добавить индикатор в контейнер.
    newIndicator.transform.SetParent(labelContainer, false);

    return newIndicator;
  }

}
```

Диспетчер индикаторов предоставляет единственный метод `AddIndicator`, который создает экземпляры из шаблона `Indicator`, связывает их с целевыми объектами, настраивает цвет для окраски спрайта и добавляет их в контейнер индикаторов. При желании можно передать этому методу свой спрайт, чтобы создать индикатор с уникальным видом. (Мы сделаем это позже, когда добавим прицельную сетку.)

После ввода исходного кода сценария `IndicatorManager` можно перейти к его настройке. Диспетчер должен знать две вещи: какой шаблон использовать для создания индикаторов и какой объект должен быть их родителем.

3. *Настройте диспетчер индикаторов.* Перетащите объект-контейнер Indicators в поле Label Container, а шаблон Position Indicator — в поле Indicator Prefab.

   Теперь добавим в космическую станцию код, добавляющий индикатор при запуске.

4. *Выберите космическую станцию.*

5. *Добавьте в нее сценарий* `SpaceStation`. Добавьте в объект станции новый сценарий на C# с именем *SpaceStation.cs* и введите в него следующий код:

```
public class SpaceStation : MonoBehaviour {

  void Start () {
    IndicatorManager.instance.AddIndicator(
      gameObject,
      Color.green
    );
  }

}
```

   Этот код просто обращается к объекту-одиночке `IndicatorManager`, чтобы добавить новый индикатор зеленого цвета, отображающий расстояние до этого объекта.

6. *Запустите игру.* Теперь со станцией будет связан индикатор.

Расстояние до станции не отображается, потому что космическая станция не устанавливает переменную `showDistanceTo`. Это сделано намеренно — мы настроим эту переменную для астероидов, но не для станции. Слишком большое количество чисел на экране может запутать игрока.

В заключение

Поздравляем! Начав с нуля, вы получили управляемый космический корабль. В следующей главе мы продолжим развитие игры и добавим в нее настоящий игровой процесс.

11 Добавление оружия и прицеливания

Теперь, реализовав управление космическим кораблем, добавим в общую картину еще один игровой элемент. Сначала мы добавим оружие, а после этого реализуем механизм прицеливания.

Оружие

Каждый раз, когда игрок приводит в действие оружие, лазерные пушки космического корабля выстреливают шарами плазмы, которые летят вперед, пока не поразят какую-либо цель или не истечет время их существования. Если плазменный шар попадает в какой-то объект и этот объект может получить повреждение, информация о попадании должна передаваться такому объекту.

Реализовать это можно, создав объект с коллайдером, который движется вперед с определенной скоростью (подобно космическому кораблю). Есть несколько вариантов отображения снаряда — можно создать трехмерную модель ракеты, использовать эффект частиц или спрайт. Вы можете выбрать любое решение, но оно не влияет на фактическое поведение снаряда в игре.

В этой главе для отображения снаряда мы используем визуализатор *Trail Renderer*. Этот визуализатор создает постепенно исчезающий светящийся след, оставляемый движущимся объектом. Этот эффект особенно хорош для представления таких движущихся объектов, как качающиеся клинки или летящие снаряды.

Применение визуализатора *Trail Renderer* поможет легко создать эффект летящего снаряда: шар плазмы будет оставлять за собой красный след, становящийся со временем все тоньше и тоньше. Так как выстрелы всегда производятся только вперед, в результате получится красивый эффект удаляющегося шара плазмы.

Неграфический компонент снаряда будет реализован с применением *кинематического твердого тела*. Обычно твердые тела реагируют на силы, прикладываемые к ним: сила гравитации заставляет их терять высоту, а когда они сталкиваются с другими твердыми телами, в действие вступает первый закон Ньютона, изменяя их скорость движения. Однако для нас нежелательно, чтобы снаряды изменяли траекторию движения. Чтобы сообщить Unity, что твердое тело игнорирует любые силы, прикладываемые к нему, и в то же время может сталкиваться с другими телами, мы сделаем его *кинематическим*.

Возникает резонный вопрос: зачем вообще использовать твердые тела для имитации снарядов? В конце концов, в космическом корабле не используется ни одного твердого тела, зачем они нужны в снарядах?

Причина объясняется ограничениями физического движка в Unity. Столкновения между объектами обнаруживаются, только если хотя бы один из них обладает твердым телом; как результат, чтобы снаряд мог передать информацию о попадании объекту, с которым он столкнулся, мы должны подключить к нему компонент твердого тела и объявить его кинематическим.

Сначала мы создадим объект снаряда и настроим его свойства, управляющие обнаружением столкновений. Затем подключим к нему сценарий `Shot`, заставляющий снаряд лететь вперед с постоянной скоростью.

1. *Создайте снаряд*. Создайте новый пустой игровой объект с именем Shot.

   Добавьте в объект компонент Rigidbody твердого тела. Затем снимите флажок Use Gravity и установите флажок Is Kinematic.

   Добавьте в объект сферический коллайдер. Установите радиус сферы равным 0,5 и установите координаты центра равными (0,0,0). Флажок Is Trigger должен быть установлен.

2. *Добавьте сценарий* `Shot`. Добавьте в объект новый сценарий на C# с именем Shot. Откройте файл *Shot.cs* и добавьте в него следующий код:

```
// Перемещает объект вперед с постоянной скоростью и уничтожает
// его спустя заданный промежуток времени.
public class Shot : MonoBehaviour {

  // Скорость движения снаряда
  public float speed = 100.0f;

  // Время в секундах, через которое следует уничтожить снаряд
  public float life = 5.0f;

  void Start() {
    // Уничтожить через 'life' секунд
    Destroy(gameObject, life);
  }

  void Update () {
    // Перемещать вперед с постоянной скоростью
    transform.Translate(
      Vector3.forward * speed * Time.deltaTime);
  }
}
```

Сценарий `Shot` очень прост, он решает две задачи: гарантирует уничтожение снаряда спустя заданное время и его перемещение вперед с постоянной скоростью.

Метод `Destroy` обычно вызывается с единственным параметром — ссылкой на объект, который требуется убрать из игры. Однако он принимает второй необязательный параметр — число секунд от текущего момента, которое должно пройти

перед уничтожением объекта. В методе `Start` сценарий вызывает метод `Destroy` и передает ему переменную `life`, сообщая движку Unity, что тот должен уничтожить объект через `life` секунд.

Функция `Update` просто использует метод `Translate` компонента `transform` для перемещения объекта вперед с постоянной скоростью. В результате умножения свойства `Vector3.forward` на `speed`, а затем на `Time.deltaTime` объект перемещается вперед с постоянной скоростью в каждом кадре.

Далее добавим графическую составляющую снаряда. Как отмечалось выше, для создания визуального эффекта летящего снаряда мы будем использовать визуализатор Trail Renderer. Для определения внешнего вида следа этот визуализатор использует материал, а это значит, что мы должны создать его.

Вы можете определить любой материал, но для сохранения простоты внешнего вида игры мы используем сплошной темный красный цвет.

1. *Создайте новый материал*. Дайте ему имя Shot.
2. *Обновите шейдер*. Чтобы след отображался сплошным красным цветом, без подсветки, выберите для материала шейдер Unlit/Color.
3. *Настройте цвет*. После изменения шейдера материала в самом материале останется единственный параметр — используемый цвет. Выберите в нем красивый ярко-красный цвет.

После создания материала его можно использовать в визуализаторе Trail Renderer.

1. *Создайте графический объект в объекте Shot*. Создайте новый пустой объект с именем Graphics. Сделайте его дочерним по отношению к объекту Shot и установите в позицию (0,0,0).
2. *Создайте визуализатор Trail Renderer*. Добавьте в объект Graphics новый компонент Trail Renderer.

   Затем снимите флажки Cast Shadows, Receive Shadows и Use Light Probes.

   Далее установите свойство Time в значение 0,05, а свойство Width в значение 0,2.
3. *Настройте сужение следа от начала к концу.* Дважды щелкните на кривой вида (ниже поля Width), после чего появится контрольная точка. Перетащите ее в правый нижний угол.
4. *Примените материал Shot*. Откройте список Materials и перетащите в него только что созданный материал Shot.

В результате настройки Trail Renderer в инспекторе должны выглядеть, как показано на рис. 11.1.

Мы еще не закончили создание объекта Shot: у нас (пока) нет возможности проверить стрельбу из оружия корабля, но мы скоро добавим эту возможность.

Остался последний шаг — превратить объект Shot в шаблон.

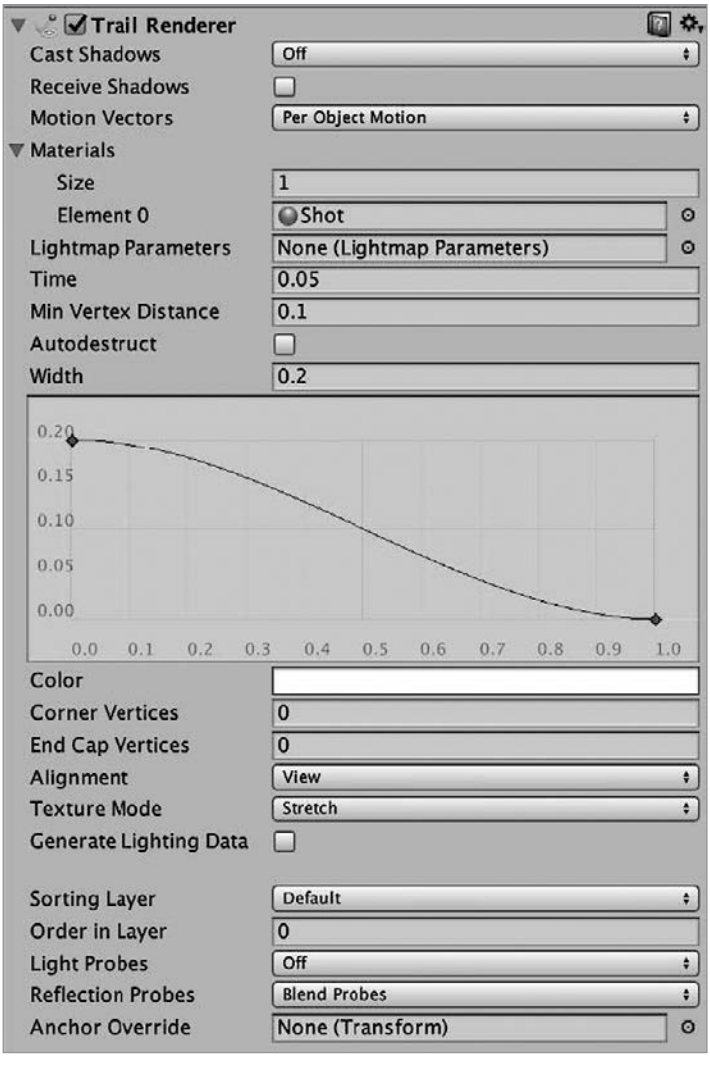

Рис. 11.1. Настройки визуализатора Trail Renderer для снаряда

1. *Перетащите объект Shot из сцены в папку Objects.* В результате этот объект превратится в шаблон.
2. *Удалите объект Shot из сцены.*

Далее мы создадим объект, реализующий стрельбу из оружия.

Оружие корабля

Чтобы игрок смог открыть огонь из лазерных пушек корабля, мы должны предусмотреть какой-то механизм создания объектов Shot. Механика стрельбы из ла-

зерных пушек немного сложнее, чем простое выбрасывание снаряда, когда игрок нажимает кнопку Fire (Огонь); нам нужно, чтобы снаряды вылетали с постоянной частотой, когда игрок нажимает и удерживает кнопку.

Также нам нужно определить, *откуда* эти снаряды будут вылетать. На концептуальном рисунке (рис. 9.3) можно видеть, что лазерные пушки размещены по бокам на крыльях, соответственно, снаряды должны вылетать с обеих сторон.

В таких случаях необходимо также решить, *как* будут выстреливаться снаряды. Можно выстреливать два снаряда сразу с обеих сторон или попеременно — то слева, то справа. В этой игре мы решили реализовать попеременную стрельбу, потому что это создает ощущение непрерывной стрельбы, — но мы не претендуем на истину в последней инстанции! Попробуйте разные подходы к организации стрельбы и посмотрите, как это скажется на поведении корабля.

Стрельбой будет управлять сценарий `ShipWeapons`. Этот сценарий использует шаблон снаряда, созданный в предыдущем разделе, а также массив объектов Transform; с началом стрельбы он начинает поочередно создавать снаряды в разных точках, соответствующих объектам Transform. При достижении конца массива с объектами `Transform` происходит возврат к началу.

1. *Добавьте сценарий* `ShipWeapons` *в объект Ship*. Выберите объект Ship, добавьте новый сценарий на C# с именем *ShipWeapons.cs* и введите в него следующий код:

```
public class ShipWeapons : MonoBehaviour {

  // Шаблон для создания снарядов
  public GameObject shotPrefab;

  // Список пушек для стрельбы
  public Transform[] firePoints;

  // Индекс в firePoints, указывающий на следующую
  // пушку
  private int firePointIndex;

  // Вызывается диспетчером ввода InputManager.
  public void Fire() {

    // Если пушки отсутствуют, выйти
    if (firePoints.Length == 0)
      return;

    // Определить следующую пушку для выстрела
    var firePointToUse = firePoints[firePointIndex];

    // Создать новый снаряд с ориентацией,
    // соответствующей пушке
    Instantiate(shotPrefab,
      firePointToUse.position,
      firePointToUse.rotation);

    // Перейти к следующей пушке
    firePointIndex++;

    // Если произошел выход за границы массива,
```

```
    // вернуться к его началу
    if (firePointIndex >= firePoints.Length)
      firePointIndex = 0;
  }

}
```

Сценарий `ShipWeapons` хранит список пушек (позиций), выстреливающих снарядами (переменная `firePoints`), а также шаблон для создания каждого снаряда (переменная `shotPrefab`). Кроме того, он хранит *индекс* следующей пушки в списке для следующего выстрела (переменная `firePointIndex`). Когда игрок нажимает кнопку Fire, снаряд вылетает из одной из пушек, а затем индекс `firePointIndex` переносится на следующую пушку.

2. *Создайте пушки (позиции для выстрелов)*. Создайте новый пустой игровой объект с именем Fire Point 1. Сделайте его дочерним по отношению к объекту Ship, а затем скопируйте нажатием комбинации Ctrl-D (Command-D на Mac.) В результате будет создан еще один пустой объект с именем; дайте ему имя Fire Point 2.

   Установите координаты для Fire Point 1 равными (−1,9, 0, 0), что соответствует левому крылу корабля.

   Установите координаты для Fire Point 2 равными (1,9, 0, 0), что соответствует правому крылу корабля.

   В итоге позиции Fire Points 1 и Fire Points 2 должны выглядеть как на рис. 11.2 и 11.3.

3. *Настройте сценарий* `ShipWeapons`. Перетащите шаблон Shot, созданный в предыдущем разделе, в поле Shot Prefab сценария `ShipWeapons`.

   Далее нам нужно добавить оба объекта Fire Point в сценарий `ShipWeapons`. Для этого можно установить размер массива Fire Points равным 2 и перетащить каждый объект по отдельности, но есть более быстрый способ.

   Выберите объект Ship и щелкните по пиктограмме замка в правом верхнем углу инспектора. Это заблокирует инспектор, и его содержимое не изменится при выборе другого объекта.

   Далее выберите оба объекта Fire Point в панели иерархии, щелкнув сначала на Fire Point 1, а затем, удерживая нажатой клавишу Ctrl (Command на Mac), на Fire Point 2.

   Потом перетащите оба объекта в поле Fire Points объекта ShipWeapons. Обязательно сбросьте перетаскиваемые объекты на надпись «Fire Points» (а не где-то под ней), иначе ничего не получится.

Этот прием работает с *любыми* переменными-массивами в сценариях. Он поможет вам избежать лишних движений мышью. Но имейте в виду, что при перетаскивании объектов из иерархии в массив их порядок следования может не сохраняться.

4. *Разблокируйте инспектор*. Закончив настройку сценария ShipWeapons, разблокируйте инспектор, щелкнув по пиктограмме с изображением замка в правом верхнем углу инспектора.

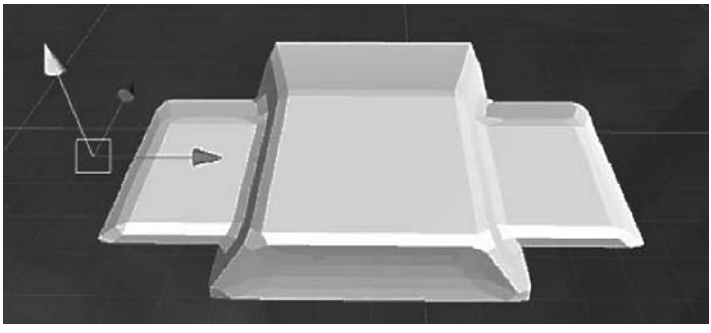

Рис. 11.2. Позиция Fire Point 1

Рис. 11.3. Позиция Fire Point 2

В этой игре космический корабль имеет только две пушки, но вообще сценарий способен обработать больше. При желании можете добавить столько пушек, сколько вы посчитаете нужным, только не забудьте сделать их дочерними по отношению к объекту Ship и добавить в список Fire Points в инспекторе.

Далее добавим в интерфейс игры кнопку Fire, которая позволит вам стрелять из пушек.

Кнопка «Fire»

Теперь добавим кнопку, коснувшись которой пользователь может открыть огонь из оружия корабля и прекратить его, отпустив палец.

В игре будет только одна кнопка Fire, но множество кораблей. Это означает, что мы не можем привязать кнопку Fire непосредственно к кораблю; вместо этого нам придется добавить диспетчер ввода, чтобы обслужить несколько экземпляров сценария `ShipWeapons`.

Вот как будет работать диспетчер ввода: так как в каждый конкретный момент времени в игре будет только один корабль, то будет присутствовать только один экземпляр `ShipWeapons`. В момент появления сценарий `ShipWeapons` будет полу-

чать ссылку на объект-одиночку диспетчера ввода и сообщать ему, что является текущим сценарием `ShipWeapons`. Диспетчер ввода запомнит это и будет использовать сценарий как часть системы стрельбы.

Наконец, кнопка Fire будет подключаться к объекту диспетчера ввода и посылать ему сигнал «огонь открыт» в момент касания ее пользователем и сигнал «огонь прекращен», когда пользователь уберет палец с кнопки. Диспетчер ввода будет передавать эти сообщения текущему сценарию `ShipWeapons`, благодаря чему тот будет вести огонь.

Альтернативное решение можно реализовать на основе метода `FindObjectOfType`. Этот метод перебирает все объекты, пытаясь найти компонент соответствующего типа, и возвращает первый найденный. Задействовав метод `FindObjectOfType`, можно избавить объект от необходимости *регистрировать* себя в роли текущего объекта, но это обходится довольно дорого: `FindObjectsOfType` работает медленно, потому что ему приходится проверять все компоненты во всех объектах в сцене. Этот прием можно применять время от времени, но не следует использовать его в каждом кадре.

Сначала реализуем поддержку регистрации текущего экземпляра `ShipWeapons` в классе `InputManager`; затем добавим код в `ShipWeapons`, который регистрирует сценарий как текущий в момент появления и аннулирует регистрацию, когда компонент удаляется (когда корабль разрушается).

Добавьте в сценарий диспетчера ввода управление сценарием `ShipWeapons`, включив следующие свойства и методы в класс `InputManager`:

```
public class InputManager : Singleton<InputManager> {

  // Джойстик, используемый для управления кораблем.
  public VirtualJoystick steering;

> // Задержка между выстрелами в секундах.
> public float fireRate = 0.2f;
>
> // Текущий сценарий ShipWeapons управления стрельбой.
> private ShipWeapons currentWeapons;
>
> // Содержит true, если в данный момент ведется огонь.
> private bool isFiring = false;
>
> // Вызывается сценарием ShipWeapons для обновления
> // переменной currentWeapons.
> public void SetWeapons(ShipWeapons weapons) {
>   this.currentWeapons = weapons;
> }
>
> // Аналогично; вызывается для сброса
> // переменной currentWeapons.
> public void RemoveWeapons(ShipWeapons weapons) {
>
>   // Если currentWeapons ссылается на данный объект 'weapons',
>   // присвоить ей null.
```

```
>   if (this.currentWeapons == weapons) {
>     this.currentWeapons = null;
>   }
> }
>
> // Вызывается, когда пользователь касается кнопки Fire.
> public void StartFiring() {
>
>   // Запустить сопрограмму ведения огня
>   StartCoroutine(FireWeapons());
> }
>
> IEnumerator FireWeapons() {
>
>   // Установить признак ведения огня
>   isFiring = true;
>
>   // Продолжать итерации, пока isFiring равна true
>   while (isFiring) {
>
>     // Если сценарий управления оружием зарегистрирован,
>     // сообщить ему о необходимости произвести выстрел!
>     if (this.currentWeapons != null) {
>       currentWeapons.Fire();
>     }
>
>     // Ждать fireRate секунд перед
>     // следующим выстрелом
>     yield return new WaitForSeconds(fireRate);
>
>   }
>
> }
>
> // Вызывается, когда пользователь убирает палец с кнопки Fire
> public void StopFiring() {
>
>   // Присвоить false, чтобы завершить цикл в
>   // FireWeapons
>   isFiring = false;
> }
>
}
```

Этот код хранит ссылку на текущий сценарий `ShipWeapons`, управляющий огнем из оружия корабля. Методы `SetWeapons` и `RemoveWeapons` вызываются сценарием `ShipWeapons`, когда он создается и уничтожается.

Метод `StartFiring` запускает новую сопрограмму, которая ведет огонь, вызывая метод `Fire` компонента `ShipWeapons`, а затем ожидая `fireRate` секунд. Поочередный вызов метода `Fire` и ожидание продолжаются, пока переменная `isFiring` хранит `true`; `isFiring` получает значение `false`, когда вызывается метод `StopFiring`. Методы `StartFiring` и `StopFiring` вызываются, когда пользователь начинает и прекращает касаться кнопки Fire, которую мы вскоре настроим.

Далее нам нужно реализовать в `ShipWeapons` взаимодействие с диспетчером ввода, добавив следующие методы в класс `ShipWeapons`:

```
public class ShipWeapons : MonoBehaviour {

  // Шаблон для создания снарядов
  public GameObject shotPrefab;

> public void Awake() {
>   // Когда данный объект запускается, сообщить
>   // диспетчеру ввода, чтобы использовать его
>   // как текущий сценарий управления оружием
>   InputManager.instance.SetWeapons(this);
> }
>
> // Вызывается при удалении объекта
> public void OnDestroy() {
>   // Ничего не делать, если вызывается не в режиме игры
>   if (Application.isPlaying == true) {
>     InputManager.instance
>       .RemoveWeapons(this);
>   }
> }

  // Список пушек для стрельбы
  public Transform[] firePoints;

  // Индекс в firePoints, указывающий на следующую
  // пушку
  private int firePointIndex;

  // Вызывается диспетчером ввода InputManager.
  public void Fire() {

    // Если пушки отсутствуют, выйти
    if (firePoints.Length == 0)
      return;

    // Определить следующую пушку для выстрела
    var firePointToUse = firePoints[firePointIndex];

    // Создать новый снаряд с ориентацией,
    // соответствующей пушке
    Instantiate(shotPrefab,
      firePointToUse.position,
      firePointToUse.rotation);

    // Перейти к следующей пушке
    firePointIndex++;

    // Если произошел выход за границы массива,
    // вернуться к его началу
    if (firePointIndex >= firePoints.Length)
      firePointIndex = 0;

  }

}
```

В момент создания космического корабля метод `Awake` из сценария `ShipWeapons` обращается к объекту-одиночке `InputManager` и регистрирует сценарий как текущий сценарий управления оружием. Когда сценарий уничтожается, что случается, когда космический корабль сталкивается с астероидом (мы добавим это событие позже), его метод `OnDestroy` аннулирует регистрацию сценария в диспетчере ввода.

Вы обратили внимание, что метод `OnDestroy` проверяет условие `Application.isPlaying == true`, прежде чем продолжить? Все потому, что когда вы останавливаете игру в редакторе, все объекты уничтожаются и, соответственно, вызываются все имеющиеся в сценариях методы `OnDestroy`. Однако при обращении к объекту-одиночке `InputManager.instance` возникает ошибка, потому что игра закончена и этот объект уже уничтожен.

Чтобы предотвратить ошибку, мы проверяем свойство `Application.isPlaying`. Оно получает значение `false` после того, как вы потребуете от редактора Unity остановить игру, и предотвращает проблемное обращение к `InputManager.instance`.

Теперь создадим кнопку Fire, предписывающую диспетчеру ввода открыть или прекратить огонь. Так как нам требуется извещать диспетчер ввода о начале и конце удержания кнопки, мы не можем использовать стандартное поведение, когда кнопка посылает сообщение только после «щелчка» (когда палец касается кнопки и затем отрывается от нее). Вместо этого для отправки отдельных сообщений Pointer Down и Pointer Up нам понадобится использовать компоненты Event Trigger.

Сначала создайте саму кнопку, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Button (Пользовательский интерфейс ▸ Кнопка). Дайте новой кнопке имя Fire Button.

Установите точки привязки и опорную точку кнопки справа внизу, щелкнув по кнопке Anchor (Привязка) слева вверху в панели инспектора, и, удерживая клавишу Alt (Option на Mac), выберите в раскрывшемся меню вариант bottom-right.

Далее установите позицию кнопки равной (−50, 50, 0). В результате кнопка разместится в правом нижнем углу холста. Установите ширину и высоту кнопки равными 160.

Выберите в свойстве Source Image компонента Image кнопки спрайт Button. Выберите в свойстве Image Type значение Sliced.

Выберите дочерний объект Text кнопки Fire Button и задайте текст надписи «Fire». Выберите в свойстве Font шрифт CRYSTAL-Regular и установите свойство Font Size в значение 28. Задайте выравнивание по центру, по горизонтали и по вертикали.

Наконец, задайте бирюзовый цвет для кнопки Fire, щелкнув на поле Color и введя в поле Hex Color значение 3DFFD0FF (рис. 11.4).

В результате кнопка должна выглядеть так, как показано на рис. 11.5.

Теперь настроим поведение кнопки.

1. *Удалите компонент Button*. Выберите объект Fire Button и щелкните по пиктограмме с изображением шестеренки в правом верхнем углу раздела с настройками компонента Button. В открывшемся меню выберите пункт Remove Component (Удалить компонент).

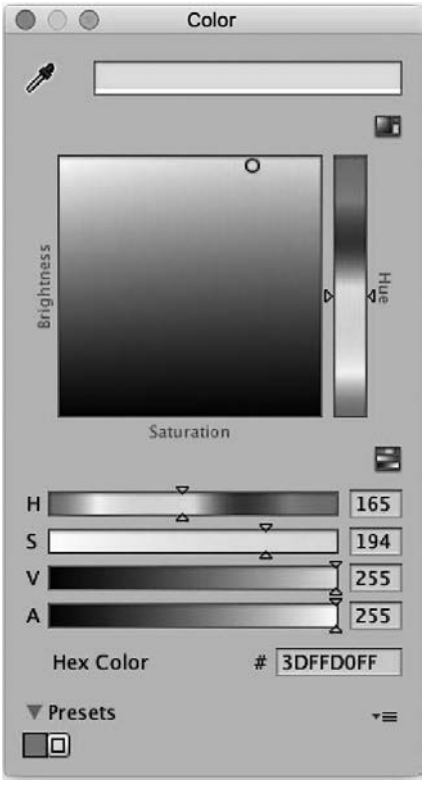

Рис. 11.4. Настройка цвета надписи на кнопке Fire

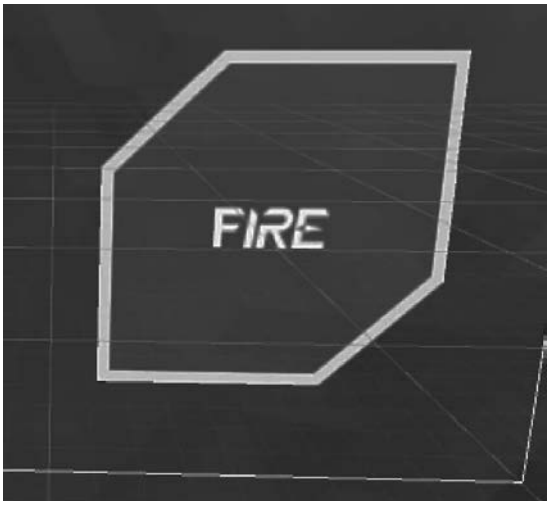

Рис. 11.5. Кнопка Fire

2. *Добавьте триггер события Event Trigger и событие Pointer Down*. Добавьте новый компонент Event Trigger, а затем щелкните по кнопке Add Event Type (Добавить тип события). Из раскрывшегося меню выберите пункт PointerDown.

   В списке появится новое событие, содержащее список объектов и методов, которые будут вызываться в ответ на касание кнопки (то есть когда пользователь коснется пальцем кнопки Fire).

   По умолчанию этот список пуст, поэтому в него нужно добавить новую цель.

3. *Настройте событие Pointer Down*. Щелкните по кнопке + внизу списка PointerDown — появится новый элемент.

   Перетащите объект Input Manager из иерархии в новое поле. Затем замените метод No Function на InputManager ▸ StartFiring.

4. *Добавьте и настройте событие Pointer Up*. Далее добавьте событие, возникающее, когда пользователь убирает палец от экрана. Щелкните еще раз по кнопке Add Event Type (Добавить тип события) и выберите пункт PointerUp.

   Настройте это событие по аналогии с событием PointerDown, но для вызова выберите метод InputManager ▸ StopFiring.

   В результате панель инспектора должна выглядеть так, как показано на рис. 11.6.

5. *Протестируйте кнопку Fire*. Запустите игру. После нажатия кнопки Fire должны начать вылетать шары плазмы!

Прицельная сетка

В настоящее время игрок не понимает, в какую точку наведены пушки. Поскольку и камера и корабль могут вращаться, правильно прицелиться может быть очень сложно. Чтобы исправить этот недостаток, используем систему индикации, созданную ранее, для отображения прицельной сетки на экране.

Мы создадим новый объект, который, как космическая станция, предписывает диспетчеру индикаторов Indicator Manager создать на экране новый индикатор, отображающий точку прицеливания. Это будет невидимый объект, дочерний по отношению к объекту корабля, находящийся на удалении перед кораблем. Это создаст эффект размещения индикатора в точке прицеливания.

Наконец, этот индикатор должен иметь особый вид, чтобы было очевидно, что он представляет точку прицеливания. В файле *Target Reticle.psd* вы найдете изображение перекрестья, прекрасно решающее эту задачу.

1. *Создайте объект Target*. Дайте этому объекту имя Target и сделайте его дочерним по отношению к объекту Ship.

2. *Установите координаты объекта Target*. Установите координаты объекта Target равными (0,0,100). В результате он разместится на некотором удалении от корабля, впереди по его курсу.

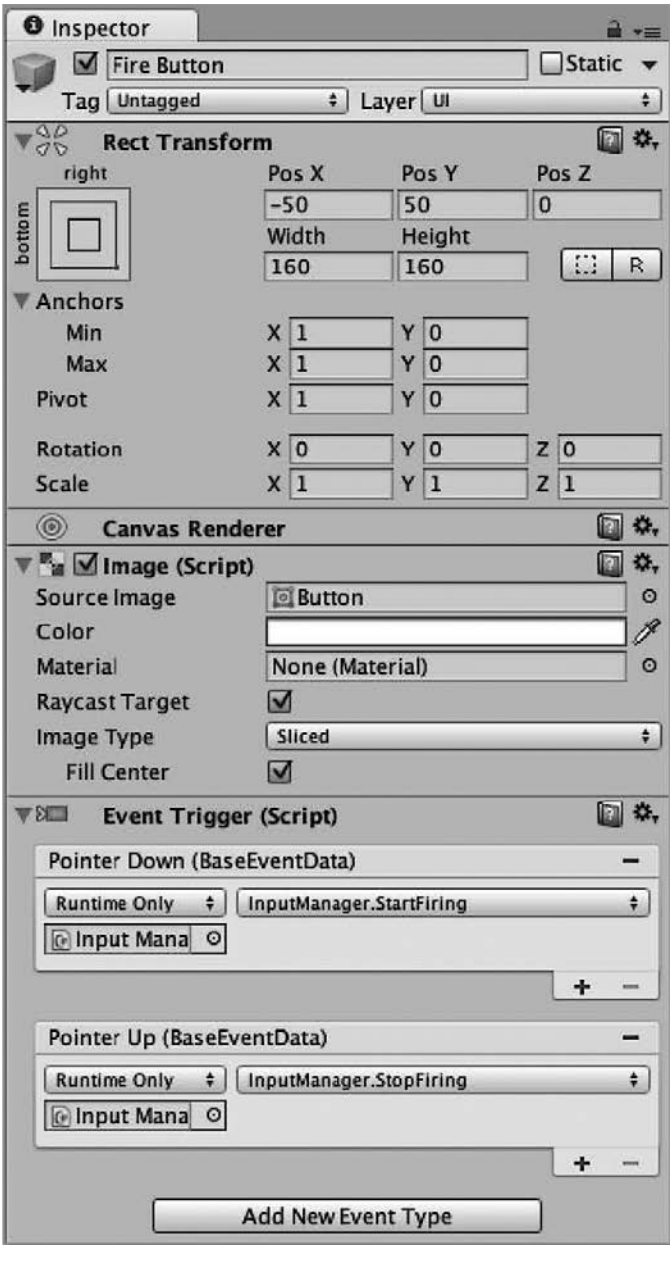

Рис. 11.6. Настройки кнопки Fire

3. *Добавьте сценарий* `ShipTarget`. Добавьте в объект Target новый сценарий на C# с именем *ShipTarget.cs* и введите в него следующий код:

```
public class ShipTarget : MonoBehaviour {

  // Спрайт для использования в качестве прицельной сетки.
  public Sprite targetImage;

  void Start () {

    // Зарегистрировать новый индикатор, соответствующий
    // данному объекту, использовать желтый цвет и
    // нестандартный спрайт.
    IndicatorManager.instance.AddIndicator(gameObject,
      Color.yellow, targetImage);
  }

}
```

Сценарий `ShipTarget` использует переменную `targetImage`, чтобы сообщить диспетчеру индикаторов Indicator Manager, какой спрайт он должен использовать для отображения индикатора на экране. Это означает необходимость настройки поля Target Image.

4. *Настройте сценарий* `ShipTarget`. Перетащите спрайт Target Reticle в поле Target Image сценария `ShipTarget`.
5. *Запустите игру*. На экране должна появиться прицельная сетка, обозначающая точку прицеливания.

В заключение

Системы управления вооружением готовы. Попробуйте поиграть с кораблем и понаблюдайте за поведением этих систем. Вы наверняка заметите, что пока в космосе нет целей; мы создали вполне реалистичную имитацию космического пространства, но в нем не хватает одного из важнейших элементов игры. Переверните страницу, и давайте исправим этот недостаток.

12 Астероиды и повреждения

Астероиды

Сейчас у нас есть корабль, летящий в космосе, индикаторы на экране, а также мы можем прицеливаться и стрелять из лазерных пушек. Единственное, чего у нас *нет*, — это объектов для стрельбы. (Космическая станция не в счет.)

Пришло время исправить это упущение. Мы создадим астероиды, не очень большие, но способные летать. Также мы добавим систему, создающую эти астероиды и направляющую их к космической станции.

Сначала создадим прототип астероида. Каждый астероид будет состоять из двух объектов: объекта высокоуровневой абстракции, содержащего коллайдер и всю необходимую логику, и дополнительного «графического» объекта, отвечающего за визуальное представление астероида на экране.

1. *Создайте объект.* Создайте новый пустой игровой объект с именем Asteroid.
2. *Добавьте в него модель астероида.* Найдите модель Asteroid в папке *Models*. Перетащите ее в только что созданный объект Asteroid и дайте новому дочернему объекту имя Graphics. Установите координаты (0,0,0) в разделе Position компонента Transform в объекте Graphics.
3. *Добавьте в объект Asteroid твердое тело и сферический коллайдер.* Не добавляйте их в объект Graphics.
4. Затем снимите флажок Use Gravity в компоненте Rigidbody и установите радиус сферического коллайдера равным 2.
5. *Добавьте сценарий Asteroid.* Добавьте в игровой объект Asteroid новый сценарий на C#, дайте ему имя *Asteroid.cs* и введите в него следующий код:

```
public class Asteroid : MonoBehaviour {

  // Скорость перемещения астероида.
  public float speed = 10.0f;

  void Start () {
    // установить скорость перемещения твердого тела
    GetComponent<Rigidbody>().velocity
      = transform.forward * speed;
```

```
    // Создать красный индикатор для данного астероида
    var indicator = IndicatorManager.instance
      .AddIndicator(gameObject, Color.red);

  }

}
```

Сценарий Asteroid очень прост: когда объект появляется в игре, на твердое тело объекта начинает действовать сила, толкающая его «вперед». Кроме того, диспетчеру индикаторов сообщается, что тот должен добавить на экран новый индикатор для этого астероида.

Вы получите предупреждение, что в переменную `indicator` записывается значение, но оно нигде не извлекается. Это нормально и не является ошибкой. Позднее мы добавим код, использующий переменную `indicator`, и устраним это предупреждение.

В результате настройки объекта Asteroid в инспекторе должны выглядеть так, как показано на рис. 12.1.

В итоге объект должен выглядеть так, как на рис. 12.2.

6. *Протестируйте поведение астероида*. Запустите игру и понаблюдайте за астероидом. Он должен двигаться вперед, а на экране появится индикатор!

Создание астероидов

Теперь, получив действующий астероид, мы можем перейти к реализации *системы создания астероидов*. Это объект, периодически создающий новые объекты астероидов и запускающий их к цели. Астероиды будут создаваться в случайных точках на поверхности невидимой сферы и настраиваться так, что их направление «вперед» будет нацелено на объект в игре. Кроме того, система создания астероидов будет использовать одну из возможностей Unity под названием Gizmos, позволяющую отображать дополнительную информацию в представлении сцены для визуализации объема космического пространства, в котором появляются астероиды.

Сначала превратим в шаблон прототип астероида, созданный в предыдущем разделе. Затем настроим систему создания астероидов Asteroid Spawner.

1. *Преобразуйте астероид в шаблон*. Перетащите объект Asteroid из панели с иерархией в панель обозревателя проекта. В результате будет создан шаблон. Затем удалите объект Asteroid из сцены.
2. *Создайте объект Asteroid Spawner*. Создайте новый пустой игровой объект с именем Asteroid Spawner. Установите его координаты равными (0,0,0).

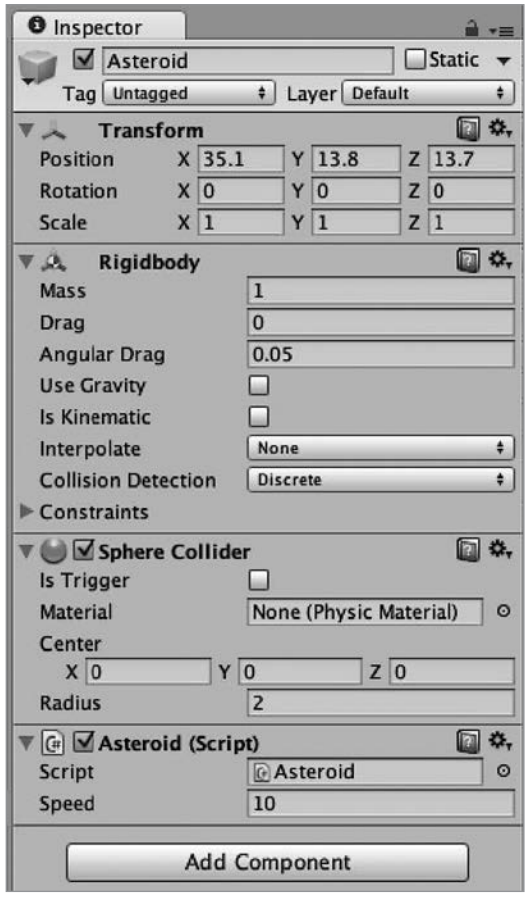

Рис. 12.1. Настройки объекта Asteroid

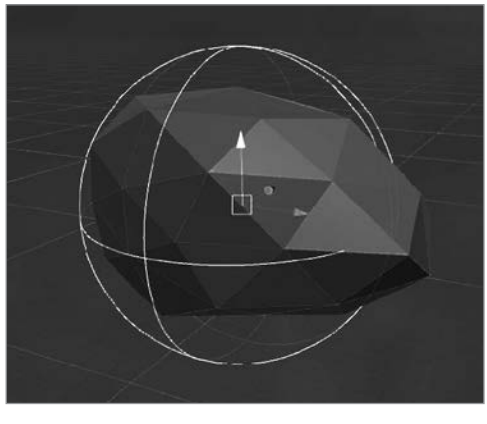

Рис. 12.2. Объект Asteroid в представлении сцены

Далее добавьте новый сценарий на C# с именем *AsteroidSpawner.cs* и введите в него следующий код:

```
public class AsteroidSpawner : MonoBehaviour {

  // Радиус сферы, на поверхности которой создаются астероиды
  public float radius = 250.0f;

  // Шаблон для создания астероидов
  public Rigidbody asteroidPrefab;

  // Ждать spawnRate ± variance секунд перед созданием нового астероида
  public float spawnRate = 5.0f;
  public float variance = 1.0f;

  // Объект, служащий целью для астероидов
  public Transform target;

  // Значение false запрещает создавать астероиды
  public bool spawnAsteroids = false;

  void Start () {
    // Запустить сопрограмму, создающую астероиды,
    // немедленно
    StartCoroutine(CreateAsteroids());
  }

  IEnumerator CreateAsteroids() {

    // Бесконечный цикл
    while (true) {

      // Определить место появления следующего астероида
      float nextSpawnTime
        = spawnRate + Random.Range(-variance, variance);

      // Ждать в течение заданного интервала времени
      yield return new WaitForSeconds(nextSpawnTime);

      // Также дождаться, пока обновится физическая подсистема
      yield return new WaitForFixedUpdate();

      // Создать астероид
      CreateNewAsteroid();
    }

  }

  void CreateNewAsteroid() {

    // Если создавать астероиды запрещено, выйти
    if (spawnAsteroids == false) {
      return;
    }

    // Выбрать случайную точку на поверхности сферы
    var asteroidPosition = Random.onUnitSphere * radius;

    // Масштабировать в соответствии с объектом
```

```
    asteroidPosition.Scale(transform.lossyScale);

    // И добавить смещение объекта, порождающего астероиды
    asteroidPosition += transform.position;

    // Создать новый астероид
    var newAsteroid = Instantiate(asteroidPrefab);

    // Поместить его в только что вычисленную точку
    newAsteroid.transform.position = asteroidPosition;

    // Направить на цель
    newAsteroid.transform.LookAt(target);
  }

  // Вызывается редактором, когда выбирается объект,
  // порождающий астероиды.
  void OnDrawGizmosSelected() {

    // Установить желтый цвет
    Gizmos.color = Color.yellow;

    // Сообщить визуализатору Gizmos, что тот должен использовать
    // текущие позицию и масштаб
    Gizmos.matrix = transform.localToWorldMatrix;

    // Нарисовать сферу, представляющую собой область создания астероидов
    Gizmos.DrawWireSphere(Vector3.zero, radius);
  }

  public void DestroyAllAsteroids() {
    // Удалить все имеющиеся в игре астероиды
    foreach (var asteroid in
      FindObjectsOfType<Asteroid>()) {
      Destroy (asteroid.gameObject);
    }
  }
}
```

Сценарий AsteroidSpawner использует сопрограмму `CreateAsteroids`, которая непрерывно создает новые объекты астероидов вызовом `CreateNewAsteroid`, ждет в течение некоторого времени и повторяет процесс.

Метод `OnDrawGizmosSelected` отвечает за отображение каркаса сферы при выборе объекта Asteroid Spawner. Эта сфера представляет собой поверхность, на которой рождаются астероиды и откуда они начинают свое движение к цели.

3. *Сплюсните Asteroid Spawner.* Установите для Asteroid Spawner масштаб по осям (1, 0,1, 1). Благодаря этому астероиды будут появляться на границе окружности, описанной вокруг цели, а не на границе сферы (рис. 12.3).
4. *Настройте сценарий AsteroidSpawner.* Перетащите недавно созданный шаблон Asteroid в поле Asteroid Prefab, а объект Space Station — в поле Target. Включите Spawn Asteroids.
5. *Протестируйте игру.* После запуска игры появятся астероиды, которые начнут перемещаться в сторону космической станции!

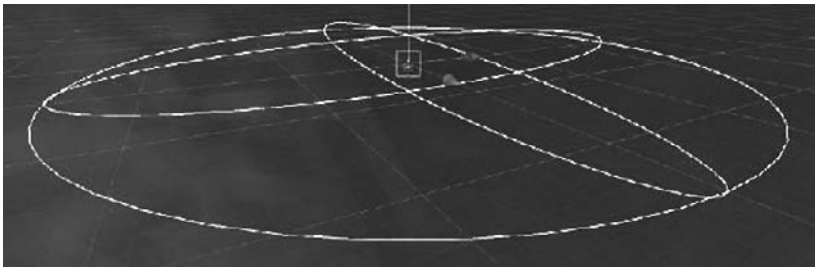

Рис. 12.3. Объект Asteroid Spawner в представлении сцены

Нанесение и получение повреждений

Теперь корабль может летать вокруг космической станции и между астероидами, летящими к ней, но выстреливаемые игроком шары плазмы фактически не причиняют никакого вреда. Нам нужно добавить возможность нанесения и получения повреждений.

Под «причинением повреждений» в этой игре понимается то, что некоторые объекты имеют «очки прочности» — число, представляющее степень их целостности. Если это число уменьшается до нуля, объект удаляется из игры.

Какие-то объекты будут способны получать повреждения, какие-то — наносить повреждения. А некоторые, например астероиды, будут способны на то и на другое — они могут получать повреждения от попадания шаров плазмы, а также могут наносить повреждения объектам, сталкивающимся с ними, таким как космическая станция.

Чтобы реализовать это, мы создадим два отдельных сценария: DamageTaking и DamageOnCollide.

- Сценарий DamageTaking контролирует очки целостности объекта, к которому он подключен, и удаляет объект из игры, когда это число достигает нуля. Сценарий DamageTaking также экспортирует метод `TakeDamage`, который будет вызываться другими объектами для нанесения повреждений.
- Сценарий DamageOnCollide выполняется, когда объект, к которому он подключен, сталкивается с любым другим объектом или входит в область действия триггера. Если объект, с которым сталкивается данный объект, имеет компонент DamageTaking, сценарий DamageOnCollide вызывает его метод `TakeDamage`.

Сценарий DamageOnCollide будет добавляться в объекты Shot и Asteroid, а сценарий DamageTaking — в объекты Space Station и Asteroid.

Сначала реализуем возможность нанесения повреждений астероидам.

1. *Добавьте в астероиды сценарий DamageTaking.* Выберите шаблон Asteroid в панели обозревателя проекта, добавьте в него новый сценарий на C# с именем *DamageTaking.cs* и введите в файл следующий код:

```
public class DamageTaking : MonoBehaviour {

  // Число очков прочности данного объекта
  public int hitPoints = 10;

  // При разрушении создать этот объект
  // в текущей позиции
  public GameObject destructionPrefab;

  // Завершить игру при разрушении данного объекта?
  public bool gameOverOnDestroyed = false;

  // Вызывается другими объектами (например, астероидами и шарами плазмы)
  // для нанесения повреждений
  public void TakeDamage(int amount) {

    // Сообщить о попадании в текущий объект
    Debug.Log(gameObject.name + " damaged!");

    // Вычесть amount из числа очков прочности
    hitPoints -= amount;

    // Очки исчерпаны?
    if (hitPoints <= 0) {

      // Зафиксировать этот факт
      Debug.Log(gameObject.name + " destroyed!");

      // Удалить себя из игры
      Destroy(gameObject);

      // Задан шаблон для создания объекта в точке разрушения?
      if (destructionPrefab != null) {

        // Создать объект в текущей позиции
        // с текущей ориентацией.
        Instantiate(destructionPrefab,
          transform.position, transform.rotation);
      }

    }

  }

}
```

Сценарий DamageTaking просто следит за количеством очков прочности объекта и предоставляет метод, который другие объекты могут вызвать для нанесения повреждений. Когда количество очков достигает нуля, объект уничтожается, и если задан шаблон для создания объекта на месте разрушения (например, взрыва, который мы добавим в разделе «Взрывы» ниже), то он создает этот объект.

2. *Настройте астероид*. Установите свойство Hit Points астероида в значение 1. Это сделает астероиды легко разрушаемыми.

   Далее придадим объектам Shot возможность наносить повреждения любым объектам, в которые они попадают.

3. *Добавьте сценарий DamageOnCollide в шары плазмы.* Выберите шаблон Shot и добавьте в него новый сценарий на C# с именем *DamageOnCollide.cs*, введите в файл следующий код:

```
public class DamageOnCollide : MonoBehaviour {

  // Объем повреждений, наносимых объекту.
  public int damage = 1;

  // Объем повреждений, наносимых себе при попадании
  // в какой-то другой объект.
  public int damageToSelf = 5;

  void HitObject(GameObject theObject) {
    // Нанести повреждение объекту, в который попал данный объект,
    // если возможно.
    var theirDamage =
      theObject.GetComponentInParent<DamageTaking>();
    if (theirDamage) {
      theirDamage.TakeDamage(damage);
    }

    // Нанести повреждение себе, если возможно
    var ourDamage =
      this.GetComponentInParent<DamageTaking>();
    if (ourDamage) {
      ourDamage.TakeDamage(damageToSelf);
    }
  }

  // Объект вошел в область действия данного триггера?
  void OnTriggerEnter(Collider collider) {
    HitObject(collider.gameObject);
  }

  // Другой объект столкнулся с текущим объектом?
  void OnCollisionEnter(Collision collision) {
    HitObject(collision.gameObject);
  }
}
```

Сценарий DamageOnCollide тоже очень прост; если он обнаруживает столкновение с другим объектом или вторжение в область коллайдера другого объекта (например, корабля), вызывается метод `HitObject`, который определяет наличие у объекта компонента `DamageTaking`. Если компонент имеется, вызывается его метод `TakeDamage`. Аналогичное действие выполняется в отношении текущего объекта; это сделано для того, чтобы при попадании в космическую станцию астероид не только наносил повреждения, но и разрушался сам.

4. *Протестируйте игру.* Попробуйте пострелять по астероидам. При попадании плазменного шара астероиды будут уничтожаться.

Далее добавим возможность разрушения космической станции.

5. *Добавьте сценарий DamageTaking в космическую станцию.* Выберите объект Space Station и добавьте в него компонент сценария DamageTaking.

Установите флажок Game Over On Destruction. Пока он не оказывает никакого влияния на игру, но позднее мы задействуем его для прекращения игры после разрушения станции.

В результате настройки объект Space Station в инспекторе должен выглядеть так, как показано на рис. 12.4.

Рис. 12.4. Настройки космической станции после добавления сценария DamageTaking

Взрывы

Когда астероид разрушается, он просто исчезает. Нас это не устраивает — было бы лучше, если бы астероиды исчезали в огне взрыва.

Один из лучших способов создать взрыв — использовать эффект частиц. Эффекты частиц великолепно подходят для ситуаций, когда требуется воспроизвести натурально выглядящий эффект, включающий элемент случайности. С их помощью можно воспроизвести дым, огонь и, конечно, взрывы.

Взрыв в этой игре будет сконструирован из *двух* эффектов частиц. Первый эффект будет имитировать начальную вспышку, а второй — постепенно исчезающее остаточное облако пыли.

Работая с эффектами частиц, важно заранее подобрать необходимые ресурсы. В частности, мы должны решить, использовать с эффектом частиц свой материал или материал по умолчанию. Материал по умолчанию — простой размытый круг — хорошо подходит для создания самых разных эффектов, но если понадобится добавить в эффект больше деталей, лучше создать свой материал.

Материал частиц по умолчанию можно использовать для создания эффекта вспышки, но чтобы воспроизвести пылевое облако, нам понадобится создать свой материал. Мы *могли бы* воссоздать облако пыли, используя большое количество мелких экземпляров частиц по умолчанию, но если в качестве отправной точки использовать изображение облака, мы можем получить более эффектный результат с меньшими усилиями.

1. *Создайте материал Dust.* Откройте меню Assets (Ресурсы) и выберите пункт Create ▸ Material (Создать ▸ Материал). Дайте материалу имя Dust.
2. *Настройте материал.* Выберите материал и измените его шейдер на Particles/Additive.

   Затем перетащите текстуру Dust в поле Particle Texture.

   Настройте полупрозрачный темно-серый цвет оттенка, щелкнув на поле Tint Color и выбрав цвет. Если вы предпочитаете конкретные числовые значения, введите (70, 70, 70, 190), как показано на рис. 12.5.

   Наконец, установите свойство Soft Particles Factor в значение 0,8.

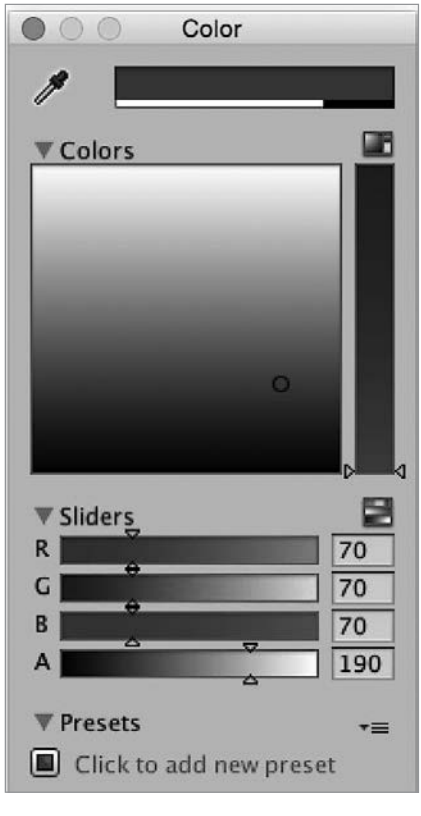

Рис. 12.5. Цвет оттенка материала Dust

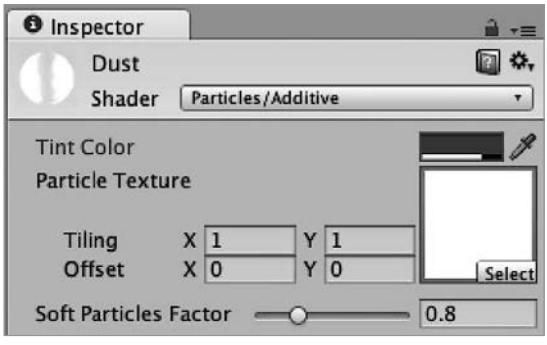

Рис. 12.6. Материал Dust

В результате настройки материала в инспекторе должны выглядеть так, как показано на рис. 12.6.

Теперь можно создать системы частиц. Сначала создадим пустой объект-контейнер для имитации взрыва, а затем создадим и настроим две системы частиц.

1. *Создайте объект Explosion*. Создайте новый пустой объект с именем Explosion.
2. *Создайте объект Fireball*. Создайте второй пустой объект и дайте ему имя Fireball. Сделайте его дочерним по отношению к объекту Explosion.
3. *Добавьте и настройте эффект частиц для вспышки*. Выберите объект Fireball и добавьте в него новый компонент Particle Effect.

Настройте эффект частиц, как показано на рис. 12.7.

Хотя большинство этих параметров числовые, не вызывающие затруднений при вводе, среди них есть несколько параметров, требующих дополнительных пояснений.

На рис. 12.8 изображен градиент, представляющий изменение цвета с течением времени.

В этом градиенте используются следующие альфа-значения (степень непрозрачности):

- 0 в точке 0 %;
- 255 в точке 12 %;
- 0 в точке 100 %.

Цвет в разных точках:

- белый в точке 0 %;
- светлый рыжевато-коричневый в точке 12 %;
- темный рыжевато-коричневый в точке 57 %;
- белый в точке 100 %.

Размер с течением времени тоже изменяется, увеличиваясь до 3 в точке 35 %, и затем постепенно уменьшается до нуля (рис. 12.9).

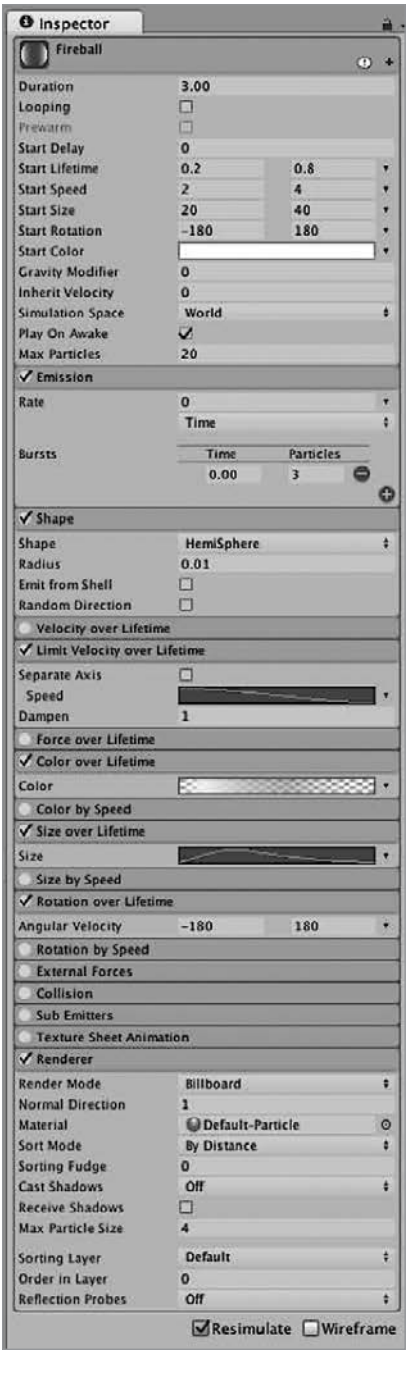

Рис. 12.7. Настройки эффекта частиц в объекте Fireball

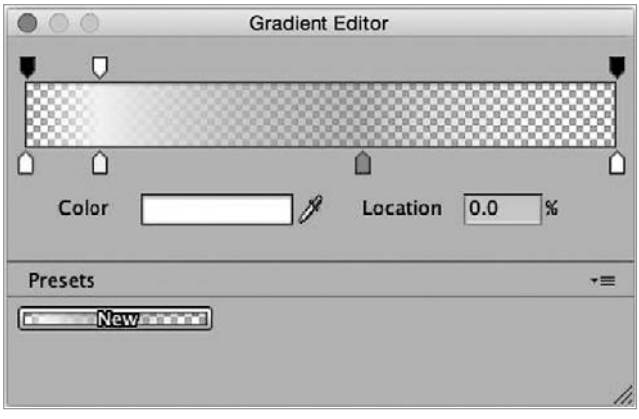

Рис. 12.8. Градиент изменения цвета начальной вспышки взрыва с течением времени

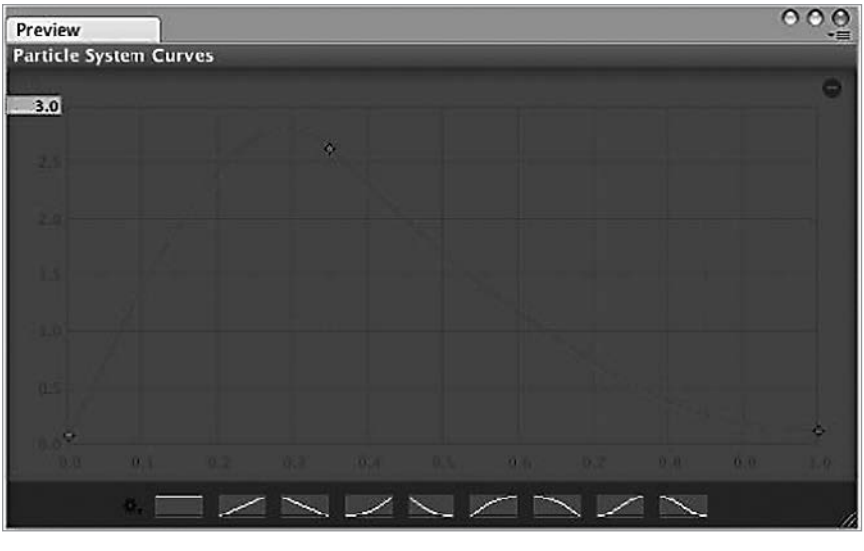

Рис. 12.9. Кривая изменения размера облака взрыва с течением времени

Объект Fireball создает первичную короткую вспышку взрыва. Второй эффект частиц, который мы добавим, — эффект Dust.

1. *Создайте объект Dust.* Создайте пустой игровой объект с именем Dust. Сделайте его дочерним по отношению к объекту Explosion.
2. *Добавьте и настройте систему частиц.* Добавьте новый компонент Particle System и настройте его, как показано на рис. 12.10.

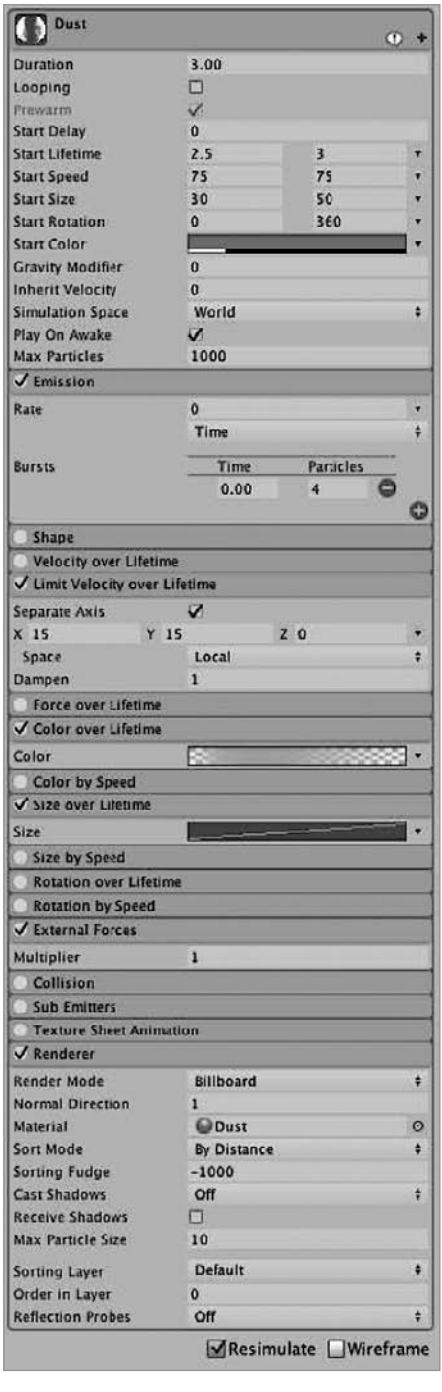

Рис. 12.10. Настройки системы частиц в объекте Dust

Некоторые параметры нельзя сразу скопировать из рис. 12.10:

- значением для поля Material визуализатора Renderer является созданный нами материал Dust. Просто перетащите его из обозревателя проекта в это поле;
- цвет в поле Start Color определяется как RGBA-значение [130, 130, 120, 45]. Щелкните на поле Start Color и введите эти числа;
- изменение размера с течением времени в поле Size Over Lifetime задается прямой линией от 0 до 100 %;
- изменение цвета с течением времени в поле Color Over Lifetime определяется так, как показано на рис. 12.11, — сам цвет постоянно остается рыжевато-коричневым, а его альфа-значение (непрозрачность) изменяется от 0 в точке 0 % до 255 в точке 14 %, а затем до 0 в точке 100 %.

Вот и всё! Теперь этот взрыв можно использовать для визуализации уничтожения астероидов.

1. *Преобразуйте объект в шаблон*. Перетащите объект Explosion в панель обозревателя проекта и удалите его из сцены.
2. *Настройте в астероидах создание эффекта взрыва при их уничтожении*. Выберите шаблон Asteroid и перетащите шаблон Explosion в поле Destruction Prefab.
3. *Протестируйте*. При попадании шаром плазмы в астероид последний должен взрываться!

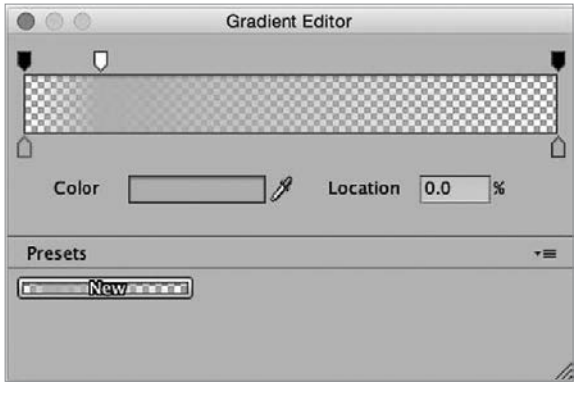

Рис. 12.11. Градиент изменения цвета частиц облака взрыва с течением времени

В заключение

Теперь, создав астероиды и модель нанесения повреждений, мы почти закончили игру. В следующей главе мы усовершенствуем ее и превратим в настоящую, эффектную игру.

13 Звуки, меню, разрушения и взрывы!

Реализация основного игрового процесса в нашем космическом шутере готова, но игра пока не закончена. Чтобы в нее можно было играть за пределами Unity, необходимо добавить меню и другие элементы управления, позволяющие пользователю взаимодействовать с игрой как с приложением. Наконец, мы усовершенствуем игру, заменив временную графику высококачественными трехмерными моделями и материалами.

Меню

На данный момент в игру можно играть, только если запускать ее из редактора Unity, щелкнув по кнопке Play (Играть). Когда игра запускается, тут же начинается игровое действие, и если космическая станция окажется разрушена, игру придется остановить и запустить снова.

Чтобы дать игроку возможность управлять игрой, нужно добавить меню. В частности, мы должны добавить самую главную кнопку: New Game (Новая игра). Мы должны дать игроку возможность начать игру сначала, когда космическая станция окажется разрушенной.

Добавление структуры меню имеет большое значение для создания ощущения законченности игры. Далее мы добавим в меню четыре компонента.

Главное меню

На этом экране будет отображаться название игры и кнопка New Game (Новая игра).

Экран паузы

На этом экране будет отображаться надпись «Paused» (Приостановлено) и кнопка для возобновления.

Экран завершения игры

На этом экране будет отображаться текст «Game Over» (Конец игры) и кнопка New Game (Новая игра).

Элементы управления игрой

На этом экране будет отображаться джойстик, индикаторы, кнопка Fire (Огонь) и все остальное, что игрок видит в процессе игры.

Все эти группы элементов пользовательского интерфейса являются взаимоисключающими: в каждый конкретный момент времени на экране может отображаться только одна группа. Сразу после запуска на экране должно появляться главное меню. После щелчка по кнопке New Game (Новая игра) это меню должно исчезнуть, а вместо него должны появиться элементы управления игрой (в дополнение к игровым объектам).

Система пользовательского интерфейса в Unity позволяет тестировать пользовательский интерфейс с помощью мыши или тачпада. Тем не менее в процессе разработки следует тестировать работу меню на настоящем сенсорном экране, например в приложении Unity Remote (см. раздел «Unity Remote» в главе 5).

Первым шагом на этом пути является объединение элементов управления игрой в один общий объект для управления им как единым целым.

1. *Создайте контейнер для элементов управления игрой.* Выберите объект Canvas и создайте новый пустой дочерний объект. Дайте этому объекту имя In-Game UI.
2. *Настройте контейнер.* Установите точки привязки объекта In-Game UI так, чтобы он растянулся на весь экран по горизонтали и по вертикали, и установите отступы Left Margin, Top Margin, Right Margin и Bottom Margin на значение 0. Благодаря этому объект заполнит весь холст.

   Далее поместим все имеющиеся элементы пользовательского интерфейса в контейнер.

3. *Сгруппируйте элементы управления игрой.* Выберите все дочерние объекты в объекте Canvas, кроме контейнера In-Game UI, и переместите их в In-Game UI.

   Теперь приступим к созданию других меню. Но прежде скроем элементы в In-Game UI, чтобы они нам не мешали.

4. *Отключите In-Game UI.* Выберите объект In-Game UI и отключите его, сняв флажок в верхнем левом углу инспектора. После этого настройки должны выглядеть так, как показано на рис. 13.1.

Главное меню

Главное меню имеет очень простую организацию — оно включает текстовую надпись с названием игры («Rockfall») и кнопку для запуска новой игры.

По аналогии с элементами управления игрой, заключим все компоненты главного меню в общий объект-контейнер в виде дочерних объектов.

1. *Создайте контейнер Main Menu.* Создайте новый пустой игровой объект и сделайте его дочерним по отношению к объекту Canvas. Дайте ему имя Main Menu.

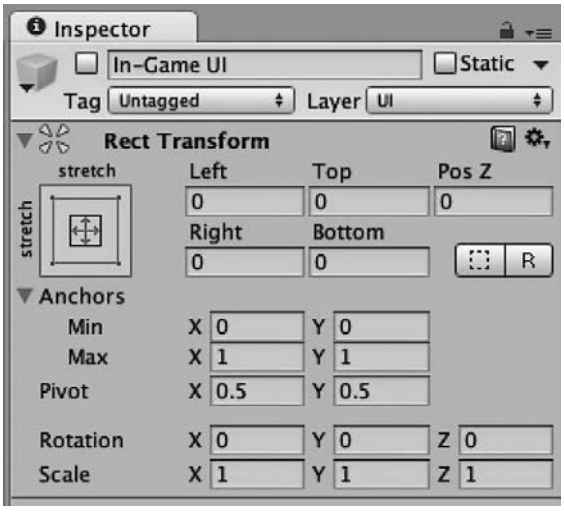

Рис. 13.1. Объект In-Game UI отключен; также обратите внимание на настройки размера и позиции объекта, которые установлены так, чтобы объект покрывал всю поверхность холста без отступов

Растяните его на весь холст по вертикали и по горизонтали. Установите все отступы равными 0.

2. *Создайте объект надписи с названием игры.* Создайте новый объект Text, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Text (Пользовательский интерфейс ▸ Текст). Сделайте его дочерним по отношению к Main Menu и дайте ему имя Title.

   Привяжите новый объект Text к верхнему краю в центре. Установите значение Pos X равным 0, а Pos Y равным −100. Установите высоту 120, а ширину — 1024.

   Далее настроим сам текст. Установите цвет текста #FFE99A (желтоватый), выровняйте текст по центру и задайте сам текст «Rockfall». Дополнительно установите флажок Best Fit. Это заставит текст изменять свои размеры в соответствии с положением границ объекта Text. Наконец, перетащите шрифт At Night в поле Font.

3. *Создайте кнопку.* Создайте новый объект Button с именем New Game. Сделайте его дочерним по отношению к объекту Main Menu.

   Привяжите кнопку к верхнему краю в центре и установите его свойства X и Y в значения [0, −300]. Установите высоту равной 330, а ширину — 80.

   Выберите в поле Source Image компонента Image кнопки спрайт Button и установите свойство Image Type в значение Sliced.

   Выберите дочерний объект Text и замените в нем текст на «New Game». В поле Font выберите шрифт CRYSTAL-Regular, установите значение Font Size равным 28 и Color равным `3DFFD0FF`.

В итоге меню должно приобрести такой вид, как показано на рис. 13.2.

Рис. 13.2. Главное меню

Прежде чем продолжить, скройте главное меню, выключив контейнер Main Menu.

Экран паузы

Экран паузы отображает надпись «Paused» (Приостановлено) и кнопку для возобновления игры. Чтобы создать его, выполните те же действия, как при создании главного меню, но со следующими изменениями:

- объекту-контейнеру присвойте имя Paused;
- в объект Title введите текст «Paused»;
- объекту кнопки присвойте имя Unpause ;
- надпись на кнопке должна содержать текст «Unpause».

В результате меню Pause должно выглядеть так, как на рис. 13.3.

Выключите контейнер Paused перед тем, как приступить к созданию последнего меню: экрана завершения игры.

Рис. 13.3. Меню Pause

Экран завершения игры

Экран завершения игры отображает текст «Game Over» (Конец игры) и содержит кнопку, запускающую новую игру. Он будет появляться после разрушения космической станции, означающего конец игры.

И снова выполните те же действия, как при создании главного меню и экрана паузы, но со следующими изменениями:

- объекту-контейнеру присвойте имя Game Over;
- в объект Title введите текст «Game Over»;
- объекту кнопки присвойте имя New Game;
- надпись на кнопке должна содержать текст «New Game».

В результате экран завершения игры должен выглядеть как на рис. 13.4.

Рис. 13.4. Меню завершения игры

Все три новых меню практически идентичны друг другу, и у вас вполне мог бы возникнуть вопрос, почему потребовалось выполнять одну и ту же работу трижды. Причина в том, что позднее может понадобиться изменить их независимо друг от друга, и, разделив их сейчас, мы избавили себя от лишней работы потом.

Теперь мы должны добавить в игру еще один компонент пользовательского интерфейса — кнопку для приостановки игры.

Добавление кнопки для приостановки игры

Кнопка приостановки будет находиться справа вверху в In-Game UI и сообщать игре, что пользователь решил приостановить ее.

Чтобы получить эту кнопку, создайте сначала новый объект Button и сделайте его дочерним по отношению к объекту-контейнеру In-Game UI. Дайте ему имя Pause.

Привяжите кнопку Pause к правому верхнему углу и установите свойства X и Y в значения [–50, –30]. Установите ширину равной 80, а высоту равной 70.

Выберите в поле Source Image компонента Image кнопки спрайт Button.

Задайте в дочернем объекте Text текст «Pause». В поле Font выберите шрифт CRYSTAL-Regular, установите Font Size в значение 28 и Color в значение 3DFFD0FF.

Вот и всё! Пользовательский интерфейс готов. Однако пока ни одна из добавленных кнопок не работает. Чтобы они заработали, добавим диспетчер игры, который будет координировать все действия.

Диспетчер игры и разрушения

Диспетчер игры, подобно диспетчеру ввода и диспетчеру индикаторов, — это объект-одиночка. Он решает две главные задачи:

- управляет состоянием игры и разными меню;
- создает корабль и станцию.

Сразу после старта игра будет находиться в незапущенном состоянии — когда корабль и станция отсутствуют в сцене, а система создания астероидов ничего не создает. Кроме того, диспетчер игры отобразит главное меню и скроет все другие меню.

Когда пользователь коснется кнопки New Game (Новая игра), на экране появится интерфейс управления игрой, корабль и станция, а система создания астероидов начнет посылать астероиды в сторону станции. Дополнительно диспетчер игры настроит некоторые важные элементы игры: сценарию `Camera Follow` будет предписано следовать за новым объектом Ship, а объекту Asteroid Spawner — посылать астероиды в направлении Space Station.

Наконец, диспетчер игры будет обслуживать состояние завершения игры. Возможно, вы помните, что в сценарии `DamageTaking` есть флажок Game Over On Destroyed. Мы настроим диспетчер игры так, что он будет заканчивать игру при разрушении объекта, к которому подключен сценарий, если установлен этот флажок. Для завершения игры достаточно просто остановить систему создания астероидов и удалить текущий корабль (а также станцию, если она к этому моменту еще не разрушена).

Прежде чем приступить к созданию диспетчера игры, нужно реализовать возможность создания множества копий корабля и станции. Для этого превратим оба этих объекта в шаблоны и определим, в каких точках пространства они будут появляться.

Чтобы превратить объекты Ship и Space Station в шаблоны, перетащите их по очереди на панель обозревателя проекта, а затем удалите из сцены.

Начальные точки

Теперь создадим два объекта-маркера, которые будут служить индикаторами точек в пространстве для создания корабля и космической станции в момент запуска новой игры. Эти индикаторы не будут видны игроку, но мы сделаем их видимыми в редакторе Unity.

1. *Создайте маркер, обозначающий начальную позицию корабля.* Создайте новый пустой игровой объект с именем Ship Start Point.

   Щелкните по пиктограмме в левом верхнем углу инспектора и выберите красную метку (рис. 13.5). После этого объект появится в представлении сцены, но останется невидимым для игрока.

   Установите маркер в позицию, где, по вашему мнению, должен появляться корабль.

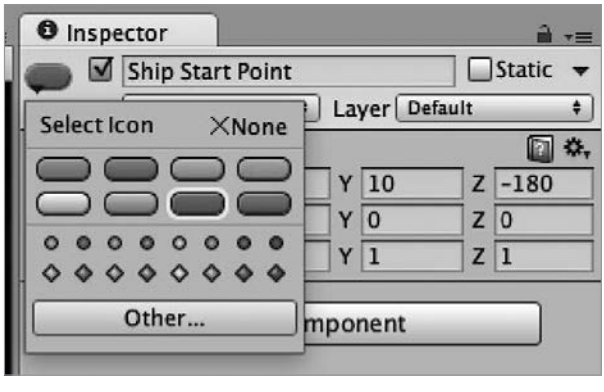

Рис. 13.5. Выбор метки для объекта-маркера, обозначающего начальную позицию появления корабля

2. *Создайте маркер, обозначающий начальную позицию космической станции.* Повторите те же действия, но на этот раз дайте объекту имя Station Start Point. Установите его в позицию, где должна появляться космическая станция.

Теперь можно создать и настроить диспетчер игры.

Создание диспетчера игры

Главная задача диспетчера игры — служить центральным хранилищем важнейшей информации об игре, такой как ссылки на текущие объекты Ship и Space Station, а также изменять состояние основных игровых объектов, когда пользователь касается кнопки или когда сценарий DamageTaking сообщает, что игра должна быть закончена.

Итак, создайте новый пустой игровой объект с именем Game Manager, добавьте в него новый сценарий на C# с именем *GameManager.cs* и введите в файл сценария следующий код:

```
public class GameManager : Singleton<GameManager> {

  // Шаблон корабля, позиция его создания
  // и текущий объект корабля
  public Gameobject shipPrefab;
  public Transform shipStartPosition;
  public GameObject currentShip {get; private set;}

  // Шаблон космической станции, позиция ее создания
  // и текущий объект станции
  public GameObject spaceStationPrefab;
  public Transform spaceStationStartPosition;
  public GameObject currentSpaceStation {get; private set;}

  // Сценарий, управляющий главной камерой
  public SmoothFollow cameraFollow;

  // Контейнеры для разных групп элементов
  // пользовательского интерфейса
  public GameObject inGameUI;
  public GameObject pausedUI;
  public GameObject gameOverUI;
  public GameObject mainMenuUI;

  // Игра находится в состоянии проигрывания?
  public bool gameIsPlaying {get; private set;}

  // Система создания астероидов
  public AsteroidSpawner asteroidSpawner;

  // Признак приостановки игры.
  public bool paused;

  // Отображает главное меню в момент запуска игры
  void Start() {
    ShowMainMenu();
  }

  // Отображает заданный контейнер с элементами пользовательского
  // интерфейса и скрывает все остальные.
  void ShowUI(GameObject newUI) {

    // Создать список всех контейнеров.
    GameObject[] allUI
      = {inGameUI, pausedUI, gameOverUI, mainMenuUI};

    // Скрыть их все.
    foreach (GameObject UIToHide in allUI) {
      UIToHide.SetActive(false);
```

```
    }

    // И затем отобразить указанный.
    newUI.SetActive(true);
  }

  public void ShowMainMenu() {
    ShowUI(mainMenuUI);

    // Когда игра запускается, она находится не в состоянии проигрывания
    gameIsPlaying = false;

    // Запретить создавать астероиды
    asteroidSpawner.spawnAsteroids = false;
  }

  // Вызывается в ответ на касание кнопки New Game
  public void StartGame() {
    // Вывести интерфейс игры
    ShowUI(inGameUI);

    // Перейти в режим игры
    gameIsPlaying = true;

    // Если корабль уже есть, удалить его
    if (currentShip != null) {
      Destroy(currentShip);
    }

    // То же для станции
    if (currentSpaceStation != null) {
      Destroy(currentSpaceStation);
    }

    // Создать новый корабль и поместить
    // его в начальную позицию
    currentShip = Instantiate(shipPrefab);
    currentShip.transform.position
      = shipStartPosition.position;
    currentShip.transform.rotation
      = shipStartPosition.rotation;

    // То же для станции
    currentSpaceStation = Instantiate(spaceStationPrefab);

    currentSpaceStation.transform.position =
      spaceStationStartPosition.position;

    currentSpaceStation.transform.rotation =
      spaceStationStartPosition.rotation;

    // Передать сценарию управления камерой ссылку на
    // новый корабль, за которым она должна следовать
    cameraFollow.target = currentShip.transform;
```

```
    // Начать создавать астероиды
    asteroidSpawner.spawnAsteroids = true;

    // Сообщить системе создания астероидов
    // позицию новой станции
    asteroidSpawner.target = currentSpaceStation.transform;

  }

  // Вызывается объектами, завершающими игру при разрушении
  public void GameOver() {

    // Показать меню завершения игры
    ShowUI(gameOverUI);

    // Выйти из режима игры
    gameIsPlaying = false;

    // Удалить корабль и станцию
    if (currentShip != null)
      Destroy (currentShip);

    if (currentSpaceStation != null)
      Destroy (currentSpaceStation);

    // Прекратить создавать астероиды
    asteroidSpawner.spawnAsteroids = false;

    // и удалить все уже созданные астероиды
    asteroidSpawner.DestroyAllAsteroids();
  }

  // Вызывается в ответ на касание кнопки Pause или Unpause
  public void SetPaused(bool paused) {

    // Переключиться между интерфейсами паузы и игры
    inGameUI.SetActive(!paused);
    pausedUI.SetActive(paused);

    // Если игра приостановлена...
    if (paused) {
      // Остановить время
      Time.timeScale = 0.0f;
    } else {
      // Возобновить ход времени
      Time.timeScale = 1.0f;
    }
  }

}
```

Сценарий Game Manager объемный, но простой. Он включает две основные функции: управления появлением меню и игрового интерфейса, а также создания

и уничтожения космической станции и корабля, когда игра начинается и заканчивается.

Давайте шаг за шагом рассмотрим его более детально.

Начальная настройка

Метод `Start` вызывается, когда диспетчер игры Game Manager впервые появляется в сцене, то есть в момент запуска игры. Этот метод просто отображает главное меню, вызывая `ShowMainMenu`.

```
// Отображает главное меню в момент запуска игры
void Start() {
  ShowMainMenu();
}
```

Чтобы отобразить любой из возможных пользовательских интерфейсов, мы используем метод `ShowUI`, который берет на себя все задачи по выводу требуемого объекта пользовательского интерфейса и сокрытия *всех* остальных. Для этого он сначала скрывает все объекты, а затем отображает требуемый:

```
// Отображает заданный контейнер с элементами пользовательского
// интерфейса и скрывает все остальные.
void ShowUI(GameObject newUI) {

  // Создать список всех контейнеров.
  GameObject[] allUI
    = {inGameUI, pausedUI, gameOverUI, mainMenuUI};

  // Скрыть все.
  foreach (GameObject UIToHide in allUI) {
    UIToHide.SetActive(false);
  }

  // И затем отобразить указанный.
  newUI.SetActive(true);
}
```

При наличии этого метода можно реализовать `ShowMainMenu`. Он лишь отображает главное меню (вызывая `ShowUI`) и устанавливает флаги, сигнализирующие, что в данный момент процесс игры остановлен, а система создания астероидов должна прекратить генерировать новые астероиды:

```
public void ShowMainMenu() {
  ShowUI(mainMenuUI);

  // Когда игра запускается, она находится не в состоянии проигрывания
  gameIsPlaying = false;

  // Запретить создавать астероиды
  asteroidSpawner.spawnAsteroids = false;
}
```

Запуск игры

Метод `StartGame`, который вызывается в ответ на нажатие кнопки New Game (Новая игра), отображает игровой интерфейс In-Game UI (и скрывает все другие), а также подготавливает сцену к новому сеансу игры, удаляя существующие корабль и космическую станцию и создавая новые. Он также передает камере ссылку на новый корабль, за которым она должна следовать, и сообщает системе создания астероидов, что она должна начать создавать астероиды и направлять их в сторону вновь созданной космической станции:

```
// Вызывается в ответ на нажатие кнопки New Game
public void StartGame() {
  // Вывести интерфейс игры
  ShowUI(inGameUI);

  // Перейти в режим игры
  gameIsPlaying = true;

  // Если корабль уже есть, удалить его
  if (currentShip != null) {
    Destroy(currentShip);
  }

  // То же для станции
  if (currentSpaceStation != null) {
    Destroy(currentSpaceStation);
  }

  // Создать новый корабль и поместить
  // его в начальную позицию
  currentShip = Instantiate(shipPrefab);
  currentShip.transform.position
    = shipStartPosition.position;
  currentShip.transform.rotation
    = shipStartPosition.rotation;

  // То же для станции
  currentSpaceStation = Instantiate(spaceStationPrefab);

  currentSpaceStation.transform.position =
    spaceStationStartPosition.position;

  currentSpaceStation.transform.rotation =
    spaceStationStartPosition.rotation;

  // Передать сценарию управления камерой ссылку на
  // новый корабль, за которым она должна следовать
  cameraFollow.target = currentShip.transform;

  // Начать создавать астероиды
  asteroidSpawner.spawnAsteroids = true;

  // Сообщить системе создания астероидов
```

```
    // позицию новой станции
    asteroidSpawner.target = currentSpaceStation.transform;

  }
```

Завершение игры

Метод `GameOver` вызывается определенными объектами, при разрушении которых игра должна завершаться. Он выводит экран завершения игры, останавливает игровой процесс и удаляет корабль и станцию. Дополнительно останавливает систему создания астероидов и удаляет все оставшиеся астероиды. Фактически этот метод возвращает игру в начальное предстартовое состояние:

```
// Вызывается объектами, завершающими игру при разрушении
public void GameOver() {

  // Показать меню завершения игры
  ShowUI(gameOverUI);

  // Пользователь не играет
  gameIsPlaying = false;

  // Удалить корабль и станцию
  if (currentShip != null)
    Destroy (currentShip);

  if (currentSpaceStation != null)
    Destroy (currentSpaceStation);

  // Прекратить создавать астероиды
  asteroidSpawner.spawnAsteroids = false;

  // и удалить все уже созданные астероиды
  asteroidSpawner.DestroyAllAsteroids();
}
```

Приостановка игры

Метод `SetPaused` вызывается в ответ на касание кнопки Pause (Приостановить) или Unpause (Возобновить). Этот метод управляет отображением экрана паузы и приостанавливает или возобновляет игровой процесс.

```
// Вызывается в ответ на касание кнопки Pause или Unpause
public void SetPaused(bool paused) {

  // Переключиться между интерфейсами паузы и игры
  inGameUI.SetActive(!paused);
  pausedUI.SetActive(paused);

  // Если игра приостановлена...
  if (paused) {
    // Остановить время
    Time.timeScale = 0.0f;
  } else {
    // Возобновить ход времени
```

```
        Time.timeScale = 1.0f;
    }
}
```

Настройка сцены

Закончив с написанием кода, можно перейти к настройке диспетчера игры Game Manager в сцене. Настройка заключается в установке связей между объектами в сцене и переменными в сценарии:

- в свойство Ship Prefab следует перетащить шаблон корабля;
- в свойство Ship Start Position следует перетащить объект-маркер, обозначающий начальную позицию корабля в сцене;
- в свойство Space Station Prefab следует перетащить шаблон космической станции;
- в свойство Station Start Position следует перетащить объект-маркер, обозначающий начальную позицию станции в сцене;
- в свойство Camera Follow следует перетащить объект Main Camera из сцены;
- в свойства In-Game UI, Main Menu UI, Paused UI и Game Over UI следует перетащить соответствующие объекты-контейнеры из сцены;
- в свойство Asteroid Spawner следует перетащить объект Asteroid Spawner из сцены;
- свойство Warning UI оставим пока как есть; мы реализуем его в следующем разделе.

В результате настройки диспетчера игры Game Manager в инспекторе должны выглядеть так, как показано на рис. 13.6.

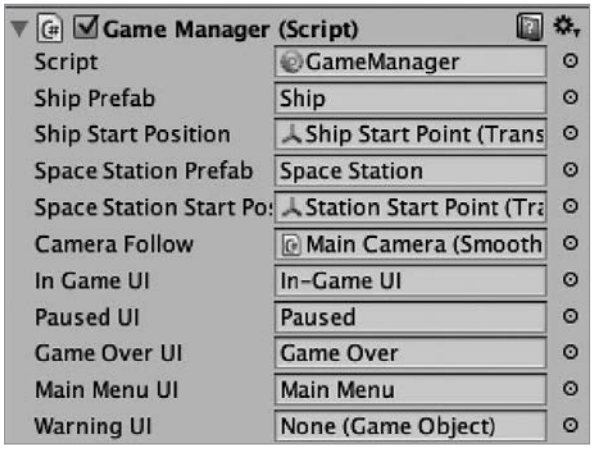

Рис. 13.6. Настройки объекта Game Manager в инспекторе

Теперь, закончив с настройкой диспетчера игры, нужно подключить к нему кнопки, присутствующие в разных меню.

1. *Подключите кнопку Pause.* Выберите кнопку Pause в контейнере In-Game UI и щелкните по кнопке + внизу раздела с настройками события Clicked. Перетащите объект Game Manager в появившееся поле и выберите функцию GameManager ▸ SetPaused. Установите появившийся после этого флажок. Теперь нажатие кнопки Pause будет приводить к вызову метода `SetPaused` в сценарии Game Manager с передачей ему логического значения `true`.
2. *Подключите кнопку Unpause.* Выберите кнопку Unpause в меню Paused. Выполните те же действия, как и для кнопки Pause, но выключите флажок после выбора функции GameManager ▸ SetPaused. Теперь нажатие кнопки будет приводить к вызову метода `SetPaused` с логическим значением `false`.
3. *Подключите кнопки New Game.* Выберите кнопку New Game в контейнере Main Menu и щелкните по кнопке + внизу раздела с настройками события Clicked. Перетащите объект Game Manager в появившееся поле и выберите функцию GameManager ▸ StartGame.

   Далее повторите те же действия для кнопки New Game в меню Game Over.

   Кнопки подключены! Но прежде чем мы закончим, необходимо еще кое-что настроить, чтобы получить полноценную игру.

   Во-первых, нужно сделать так, чтобы разрушение космической станции вызывало завершение игры. К объекту космической станции Space Station уже подключен сценарий DamageTaking; нам осталось только заставить этот сценарий вызывать функцию `GameOver` из сценария Game Manager.

4. *Добавьте в сценарий DamageTaking.cs вызов GameOver.* Откройте файл и добавьте в него следующий код:

```
public class DamageTaking : MonoBehaviour {

// Число очков прочности данного объекта
public int hitPoints = 10;

  // При разрушении создать этот объект
  // в текущей позиции
  public GameObject destructionPrefab;

  // Завершить игру при разрушении данного объекта?
  public bool gameOverOnDestroyed = false;

  // Вызывается другими объектами (например, астероидами и шарами плазмы)
  // для нанесения повреждений
  public void TakeDamage(int amount) {

    // Сообщить о попадании в текущий объект
    Debug.Log(gameObject.name + « damaged!»);

    // Вычесть amount из числа очков прочности
    hitPoints -= amount;
```

```
    // Очки исчерпаны?
    if (hitPoints <= 0) {

      // Зафиксировать этот факт
      Debug.Log(gameObject.name + " destroyed!");

      // Удалить себя из игры
      Destroy(gameObject);

      // Задан шаблон для создания объекта в точке разрушения?
      if (destructionPrefab != null) {

      // Создать объект в текущей позиции
      // с текущей ориентацией.
      Instantiate(destructionPrefab,
      transform.position, transform.rotation);
      }

>     // Если требуется завершить игру, вызвать
>     // метод GameOver класса GameManager.
>     if (gameOverOnDestroyed == true) {
>     GameManager.instance.GameOver();
>     }
    }

  }

}
```

Этот код заставит разрушившийся объект проверить свою переменную `gameOverOnDestroyed`, и если она содержит значение `true` — вызвать метод `GameOver` из сценария Game Manager, чтобы завершить игру.

Мы также должны дать астероидам возможность причинять повреждения другим объектам при столкновении с ними. Для этого добавим в них сценарий DamageOnCollide.

Чтобы астероиды могли причинять повреждения другим объектам, выберите шаблон Asteroid и добавьте в него компонент DamageOnCollide.

Далее, астероиды должны отображать на экране расстояние между ними и космической станцией. Это поможет игроку решить, какой астероид расстрелять первым. Для этого изменим сценарий Asteroid, добавив в него обращение к диспетчеру игры для получения координат космической станции, которые затем перепишем в переменную `showDistanceTo` индикатора астероида.

Чтобы астероид отображал расстояние до станции, откройте сценарий *Asteroid.cs* и добавьте следующий код в функцию `Start`:

```
public class Asteroid : MonoBehaviour {

  // Скорость перемещения астероида.
  public float speed = 10.0f;

  void Start () {
```

```
        // установить скорость перемещения твердого тела
        GetComponent<Rigidbody>().velocity
          = transform.forward * speed;

        // Создать красный индикатор для данного астероида
        var indicator =
          IndicatorManager.instance
          .AddIndicator
          (gameObject, Color.red);

>       // Запомнить координаты космической станции,
>       // управляемой диспетчером игры,
>       // для отображения расстояния от нее до астероида
>       indicator.showDistanceTo =
>       GameManager.instance.currentSpaceStation
>       .transform;
    }

}
```

Этот код настраивает индикатор на отображение расстояния от астероида до космической станции, что поможет игроку определить, какой астероид находится ближе к станции.

Вот и все!

Поиграйте в игру. Теперь вы можете летать вокруг станции, расстреливать астероиды, а астероиды могут наносить повреждения станции и даже разрушить ее. Вы тоже сможете разрушить станцию, расстреляв ее, а как только станция разрушится, игра закончится!

Границы

Осталась последняя важная деталь, которую нужно добавить в игру: мы должны предупредить игрока, если он оказался слишком далеко от космической станции. Если игрок улетит слишком далеко, мы покажем ему красную предупреждающую рамку вокруг экрана; если он не развернется, игра закончится.

Создание рамки

Сначала создадим предупреждающую рамку.

1. *Добавьте спрайт Warning.* Выберите текстуру Warning. Измените ее тип на Sprite/UI.

   Нам нужно *нарезать* спрайт, чтобы растянуть его на весь экран без искажения его формы и углов.

2. *Нарежьте спрайт.* Щелкните по кнопке Sprite Editor (Редактор спрайта), и спрайт появится в новом окне. В панели, внизу справа в окне, введите во все поля в разделе Border число 127. Это сделает углы нерастягиваемыми (рис. 13.7).

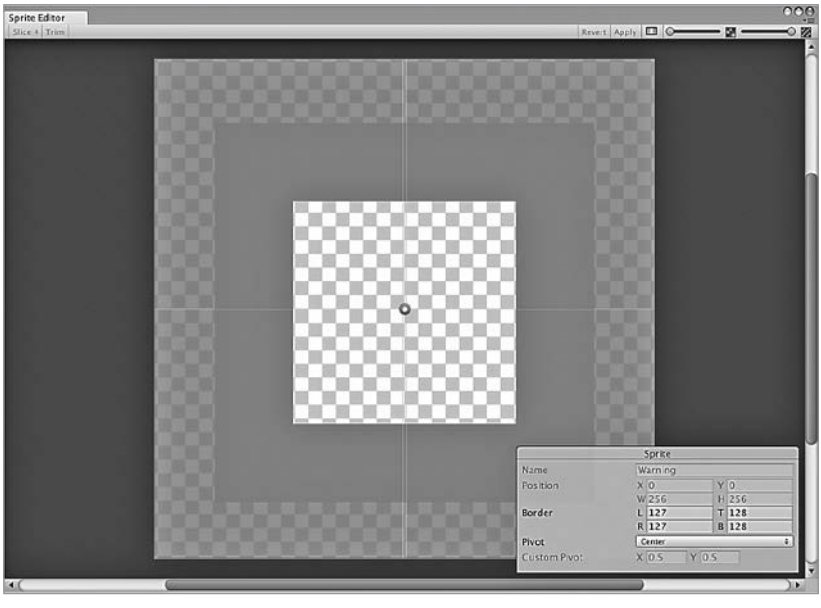

Рис. 13.7. Нарезка спрайта Warning

Щелкните по кнопке Apply (Применить).

3. *Теперь создадим предупреждающую рамку.* Это простое изображение, отображаемое в интерфейсе игры, которое будет растягиваться на весь экран.

   Чтобы получить рамку, создайте новый пустой игровой объект с именем Warning UI. Сделайте его дочерним по отношению к объекту Canvas.

   Установите точки привязки объекта так, чтобы он растянулся на весь экран по горизонтали и по вертикали, и задайте ширину всех отступов равной нулю. Благодаря этому объект растянется на весь холст.

   Добавьте в него компонент Image. Выберите в свойстве Source Image этого компонента только что созданный спрайт Warning и установите свойство Image Type в значение Sliced. Изображение растянется на весь экран.

Теперь реализуем управление рамкой.

Управление рамкой

Границы невидимы для игрока и точно так же невидимы в режиме редактирования игры. Чтобы отобразить объем пространства, в котором игрок может летать, нужно вновь воспользоваться возможностью под названием Gizmos, как мы это делали при создании Asteroid Spawner.

Для наших целей создадим две сферы, первую из которых назовем сферой *предупреждения*, а вторую — сферой *уничтожения*. Центры обеих сфер будут нахо-

диться в одной и той же точке, но радиусы их будут отличаться: радиус сферы предупреждения меньше радиуса сферы уничтожения.

- Если корабль находится внутри сферы предупреждения, значит, все в порядке и никаких предупреждений выводиться не будет.
- Если корабль окажется за границами сферы предупреждения, на экране появится предупреждение, сигнализирующее, что игрок должен развернуться и лететь обратно.
- Если корабль окажется за границами сферы уничтожения, игра закончится.

Фактическая проверка выхода корабля за пределы любой из двух сфер будет осуществляться диспетчером игры и использовать данные, хранящиеся в объекте Boundary (который мы сейчас создадим).

Сначала создадим объект Boundary и добавим код визуализации двух сфер.

1. *Создайте объект Boundary*. Создайте новый пустой объект с именем Boundary.

Добавьте в объект новый сценарий на C# с именем *Boundary.cs* и введите в него следующий код:

```
public class Boundary : MonoBehaviour {

  // Показывает предупреждающую рамку, когда игрок
  // улетает слишком далеко от центра
  public float warningRadius = 400.0f;

  // Расстояние от центра, удаление на которое вызывает завершение игры
  public float destroyRadius = 450.0f;

  public void OnDrawGizmosSelected() {
    // Желтым цветом показать сферу предупреждения
    Gizmos.color = Color.yellow;
    Gizmos.DrawWireSphere(transform.position,
      warningRadius);

    // ...а красным — сферу уничтожения
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(transform.position,
      destroyRadius);
  }
}
```

Вернувшись в редактор, вы увидите каркасы двух сфер. Желтым цветом отображается сфера предупреждения и красным — сфера уничтожения (как показано на рис. 13.8).

Сценарий Boundary фактически не выполняет никакой игровой логики. Но сценарий GameManager использует его данные для определения пересечения игроком границ.

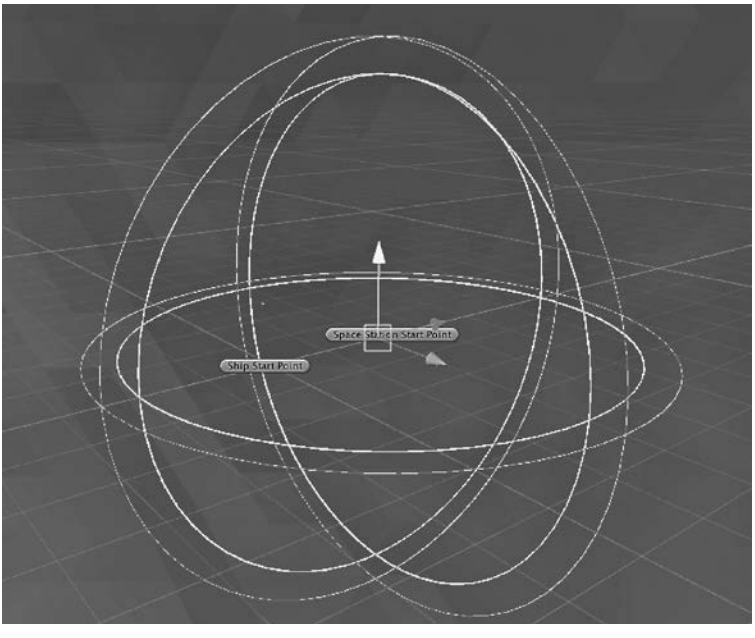

Рис. 13.8. Границы

Теперь, когда объект границ готов, нам осталось только настроить диспетчер игры Game Manager на его использование.

2. *Добавьте поля границ в сценарий GameManager и код, использующий их.* Добавьте следующий код в сценарий *GameManager.cs*:

```
public class GameManager : Singleton<GameManager> {

  // Шаблон корабля, позиция его создания
  // и текущий объект корабля
  public Gameobject shipPrefab;
  public Transform shipStartPosition;
  public GameObject currentShip {get; private set;}

  // Шаблон космической станции, позиция ее создания
  // и текущий объект станции
  public GameObject spaceStationPrefab;
  public Transform spaceStationStartPosition;
  public GameObject currentSpaceStation {get; private set;}

  // Сценарий, управляющий главной камерой
  public SmoothFollow cameraFollow;

> // Границы игры
> public Boundary boundary;

  // Контейнеры для разных групп элементов
```

```
  // пользовательского интерфейса
  public GameObject inGameUI;
  public GameObject pausedUI;
  public GameObject gameOverUI;
  public GameObject mainMenuUI;

> // Предупреждающая рамка, которая появляется,
> // когда игрок пересекает границу
> public GameObject warningUI;

// Игра находится в состоянии проигрывания?
  public bool gameIsPlaying {get; private set;}

  // Система создания астероидов
  public AsteroidSpawner asteroidSpawner;

  // Признак приостановки игры.
  public bool paused;

  // Отображает главное меню в момент запуска игры
  void Start() {
    ShowMainMenu();
  }

  // Отображает заданный контейнер с элементами пользовательского
  // интерфейса и скрывает все остальные.
  void ShowUI(GameObject newUI) {

    // Создать список всех контейнеров.
    GameObject[] allUI
      = {inGameUI, pausedUI, gameOverUI, mainMenuUI};

    // Скрыть их все.
    foreach (GameObject UIToHide in allUI) {
      UIToHide.SetActive(false);
    }

    // И затем отобразить указанный.
    newUI.SetActive(true);
  }

  public void ShowMainMenu() {
    ShowUI(mainMenuUI);

    // Когда игра запускается, она находится не в состоянии проигрывания
    gameIsPlaying = false;

    // Запретить создавать астероиды
    asteroidSpawner.spawnAsteroids = false;
  }

  // Вызывается в ответ на касание кнопки New Game
  public void StartGame() {
```

```
    // Вывести интерфейс игры
    ShowUI(inGameUI);

    // Перейти в режим игры
    gameIsPlaying = true;

    // Если корабль уже есть, удалить его
    if (currentShip != null) {
      Destroy(currentShip);
    }

    // То же для станции
    if (currentSpaceStation != null) {
      Destroy(currentSpaceStation);
    }

    // Создать новый корабль и поместить
    // его в начальную позицию
    currentShip = Instantiate(shipPrefab);
    currentShip.transform.position
      = shipStartPosition.position;
    currentShip.transform.rotation
      = shipStartPosition.rotation;

    // То же для станции
    currentSpaceStation = Instantiate(spaceStationPrefab);

    currentSpaceStation.transform.position =
      spaceStationStartPosition.position;

    currentSpaceStation.transform.rotation =
      spaceStationStartPosition.rotation;

    // Передать сценарию управления камерой ссылку на
    // новый корабль, за которым она должна следовать
    cameraFollow.target = currentShip.transform;

    // Начать создавать астероиды
    asteroidSpawner.spawnAsteroids = true;

    // Сообщить системе создания астероидов
    // позицию новой станции
    asteroidSpawner.target = currentSpaceStation.transform;

  }

  // Вызывается объектами, завершающими игру при разрушении
  public void GameOver() {

    // Показать меню завершения игры
    ShowUI(gameOverUI);

    // Выйти из режима игры
```

```
    gameIsPlaying = false;

    // Удалить корабль и станцию
    if (currentShip != null)
      Destroy (currentShip);

    if (currentSpaceStation != null)
      Destroy (currentSpaceStation);

>   // Скрыть предупреждающую рамку, если она видима
>   warningUI.SetActive(false);

    // Прекратить создавать астероиды
    asteroidSpawner.spawnAsteroids = false;

    // И удалить все уже созданные астероиды
    asteroidSpawner.DestroyAllAsteroids();
  }

  // Вызывается в ответ на нажатие кнопки Pause или Unpause
  public void SetPaused(bool paused) {

    // Переключиться между интерфейсами паузы и игры
    inGameUI.SetActive(!paused);
    pausedUI.SetActive(paused);

    // Если игра приостановлена...
    if (paused) {
      // Остановить время
      Time.timeScale = 0.0f;
    } else {
      // Возобновить ход времени
      Time.timeScale = 1.0f;
    }
  }

> public void Update() {
>
>   // Если корабля нет, выйти
>   if (currentShip == null)
>     return;
>
>   // Если корабль вышел за границу сферы уничтожения,
>   // завершить игру. Если он внутри сферы уничтожения, но
>   // за границами сферы предупреждения, показать предупреждающую
>   // рамку. Если он внутри обеих сфер, скрыть рамку.
>
>   float distance =
>     (currentShip.transform.position
>       - boundary.transform.position)
>     .magnitude;
>
>   if (distance > boundary.destroyRadius) {
>     // Корабль за пределами сферы уничтожения,
```

```
>     // завершить игру
>     GameOver();
>   } else if (distance > boundary.warningRadius) {
>     // Корабль за пределами сферы предупреждения,
>     // показать предупреждающую рамку
>     warningUI.SetActive(true);
>   } else {
>     // Корабль внутри сферы предупреждения,
>     // скрыть рамку
>     warningUI.SetActive(false);
>   }
>
> }

}
```

Этот новый код использует только что созданный класс `Boundary` для проверки выхода за пределы сферы предупреждения или сферы уничтожения. В каждом кадре он проверяет расстояние от корабля до центра сфер; если корабль покинул сферу предупреждения, отображается предупреждающая рамка, а если корабль покинул сферу уничтожения, игра завершается. Если корабль находится *внутри* сферы предупреждения, значит, все в порядке, и рамка скрывается. То есть если игрок вылетит за пределы сферы предупреждения, а затем вернется на безопасное расстояние, он сначала увидит предупреждающую рамку, а потом она скроется.

Далее нам осталось лишь подключить переменные. Диспетчеру игры Game Manager нужны ссылки на объекты Boundary и Warning UI.

3. *Настройте диспетчер игры Game Manager на использование объектов границ.* Перетащите Warning UI в поле Warning UI, а объект Boundary в поле Boundary.
4. *Поиграйте в игру*. При пересечении границы предупреждения должна появиться рамка, и если вы не повернете обратно, игра завершится!

Окончательная доводка

Поздравляем! Вы только что завершили работу над основой игрового процесса весьма сложного космического шутера. В предыдущих разделах, следуя за пояснениями, вы настроили космическое пространство, создали космический корабль, космическую станцию, астероиды и лазерные пушки, отрегулировали их физические параметры и настроили логические компоненты, связывающие все это воедино. Кроме того, вы создали пользовательский интерфейс, необходимый для игры за пределами редактора Unity.

Основа игры готова, но в ней еще есть место для улучшения визуального представления. Поскольку визуальные эффекты в игре встречаются довольно редко, игрок не получает визуальных подсказок, помогающих ощутить скорость движения. Дополнительно мы добавим в игру чуть больше красок, подключив к кораблю и астероидам визуализаторы Trail Renderer.

Космическая пыль

Если вам приходилось играть в игры, имитирующие полет в космосе, такие как *Freelancer* или *Independence War*, вы могли заметить, как во время движения мимо корабля проносятся пылинки, осколки и другие мелкие объекты.

Чтобы улучшить нашу игру, добавим маленькие пылинки, которые помогут создать ощущение глубины и перспективы во время полета корабля. Для этого создадим систему частиц, передвигающуюся вместе с кораблем и непрерывно генерирующую частицы пыли вокруг него. Важно отметить, что эти частицы не будут перемещаться относительно окружающего пространства. То есть во время полета будет создаваться ощущение, что игрок пролетает мимо них. Это создаст более полное ощущение скорости.

Для создания частиц пыли выполните следующие действия.

1. *Перетащите шаблон Ship в сцену*. Мы внесем в него некоторые изменения.
2. *Создайте дочерний объект Dust*. Создайте новый пустой игровой объект с именем Dust. Сделайте его дочерним по отношению к игровому объекту Ship, только что перенесенному в сцену.
3. *Добавьте в него компонент Particle System*. Настройте компонент, как показано на рис. 13.9.

   Важной особенностью этой системы частиц является значение World в свойстве Simulation Space и значение Sphere в свойстве Shape. Первая настройка (значение World в свойстве Simulation Space) указывает на то, что частицы не будут перемещаться вместе с кораблем. Это означает, что корабль будет пролетать мимо них.
4. *Примените изменения к шаблону.* Выберите объект Ship и щелкните по кнопке Apply (Применить) в верхней части инспектора. После этого произведенные изменения сохранятся в шаблоне. Мы еще не закончили с кораблем, поэтому пока не удаляйте его из сцены.

   Увидеть получившуюся систему частиц можно на рис. 13.10. Обратите внимание, как она создает эффект звездного поля на фоне сглаженных цветовых переходов небесного куба.

Светящиеся следы

Корабль — очень простая модель, но нет причин, по которым нельзя было бы украсить ее некоторыми специальными эффектами. Давайте добавим два светящихся следа, создающих эффект работы реактивных двигателей.

1. *Создайте новый материал для следа*. Откройте меню Assets (Ресурсы) и выберите пункт Create ▸ Material (Создать ▸ Материал). Присвойте новому материалу имя Trail и поместите его в папку *Objects*.

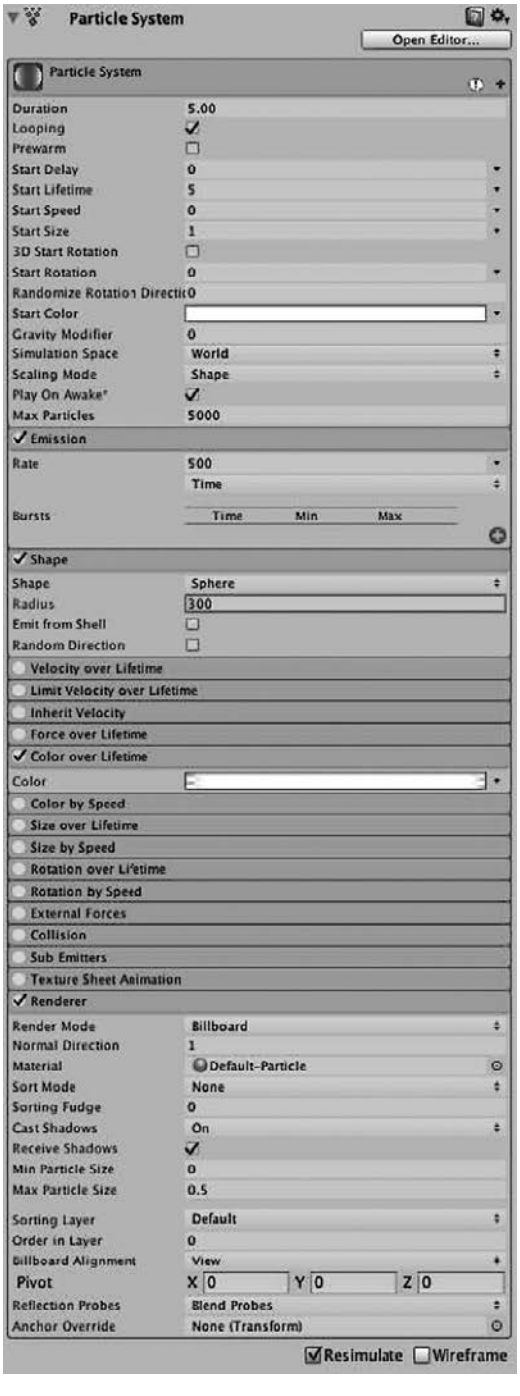

Рис. 13.9. Настройки частиц пыли

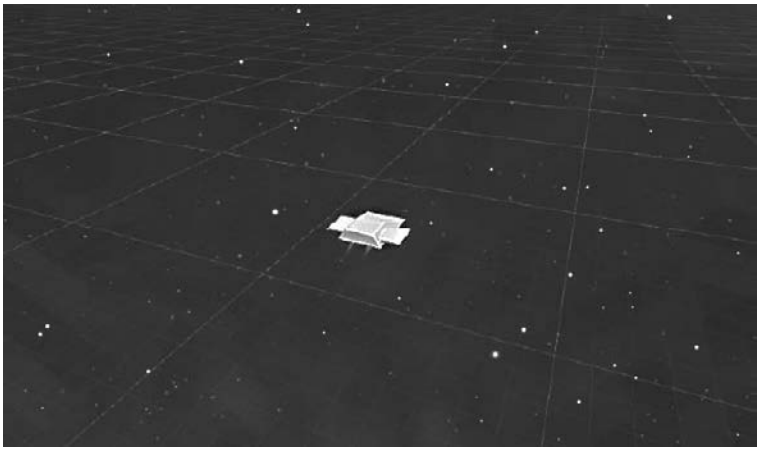

Рис. 13.10. Система частиц, имитирующая космическую пыль

2. *Задействуйте в материале Trail шейдер Additive*. Выберите материал Trail и установите в поле Shader значение Mobile ▸ Particles ▸ Additive. Этот простой шейдер всего лишь добавляет свой цвет к фону. Оставьте свойство Particle Texture пустым — оно нам не понадобится.
3. *Добавьте новый дочерний объект в шаблон Ship*. Присвойте ему имя Trail 1. Установите в позицию (–0,38, 0, –0,77).
4. *Добавьте компонент Trail Renderer*. Настройте так, как показано на рис. 13.11. Обратите внимание, что в поле Material выбран только что созданный новый материал Trail.

В градиенте визуализатора Trail Renderer используются следующие цвета:

- #000B78FF
- #061EF9FF
- #0080FCFF
- #000000FF
- #00000000

Обратите внимание, что цвета ближе к концу становятся все темнее. Так как для отображения материала Trail используется шейдер Additive, это создает эффект растворения следа.

5. *Скопируйте объект Trail 1*. Закончив настройку первого следа, скопируйте его, открыв меню Edit (Правка) и выбрав пункт Duplicate (Дублировать). Установите новую копию в позицию (0,38, 0, –0,77).

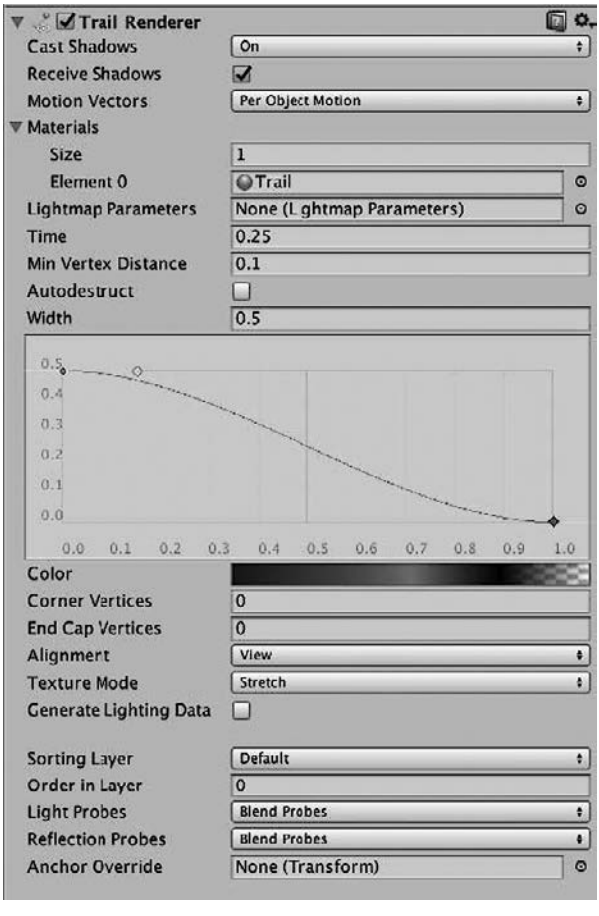

Рис. 13.11. Настройки визуализатора Trail Renderer в шаблоне корабля

Второй след имеет точно такие же координаты Z и Y, как и первый, но координата X имеет противоположный знак.

6. *Примените изменения, произведенные в шаблоне*. Выберите объект Ship и щелкните по кнопке Apply (Применить) в верхней части инспектора, затем удалите объект Ship из сцены.

Теперь можно протестировать результат! Во время полета корабля за ним будут тянуться два голубоватых светящихся следа, как показано на рис. 13.12.

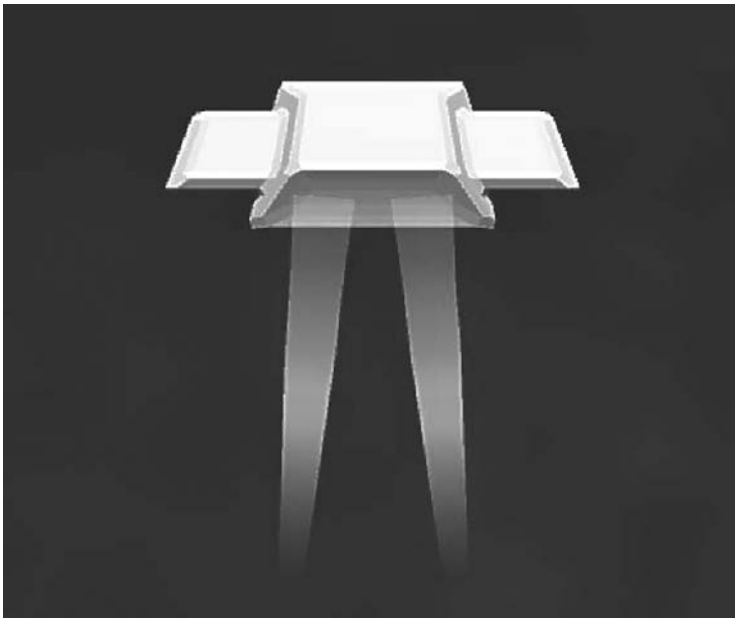

Рис. 13.12. Два следа за кораблем, имитирующих работу реактивных двигателей

Теперь применим похожий эффект к астероидам. Астероиды в игре довольно темные, и хотя есть индикаторы, помогающие игроку следить за астероидами, мы могли бы добавить чуть больше красок. С этой целью сделаем светящийся след.

1. *Добавьте астероид в сцену.* Перетащите шаблон Asteroid в сцену, чтобы получить возможность внести в него изменения.
2. *Добавьте компонент Trail Renderer в дочерний объект Graphics.* Используйте настройки, изображенные на рис. 13.13.
3. *Примените изменения к шаблону Asteroid и удалите его из сцены.*

Теперь астероиды будут оставлять за собой светящийся след. Законченную игру в действии можно увидеть на рис. 13.14.

Звуки

Нам осталось добавить в игру последний набор эффектов: звуки! Даже при том, что в настоящем космосе нет звуков, качество видеоигры существенно повысится после их добавления. Нам нужно добавить три звука: рев двигателей корабля, звук выстрелов лазерных пушек и звук взрыва астероидов. Добавим их по очереди.

В комплект ресурсов, распространяемых с книгой, мы включили несколько бесплатных звуковых эффектов, которые вы найдете в папке *Audio*.

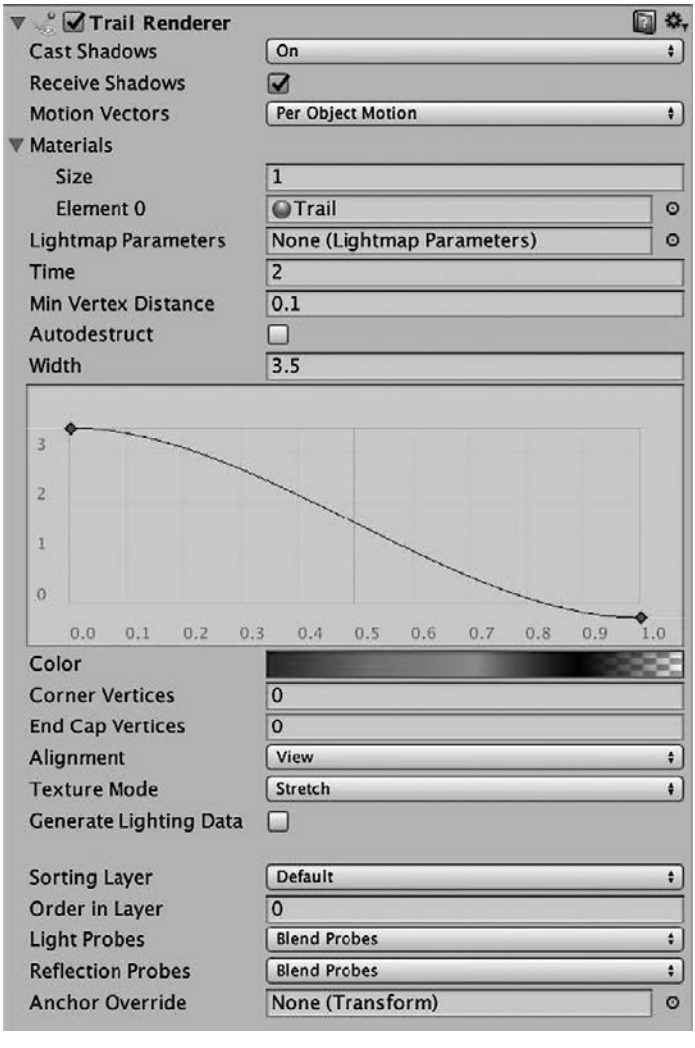

Рис. 13.13. Настройки светящегося следа астероида

Космический корабль

Сначала добавим непрерывный гул космического корабля.

1. *Добавьте шаблон Ship в сцену.* Мы внесем в него несколько изменений.
2. *Добавьте компонент Audio Source в объект Ship.* Компонент Audio Source управляет воспроизведением звуков.
3. *Установите флажок Loop.* Нам нужно, чтобы шум двигателей звучал непрерывно.

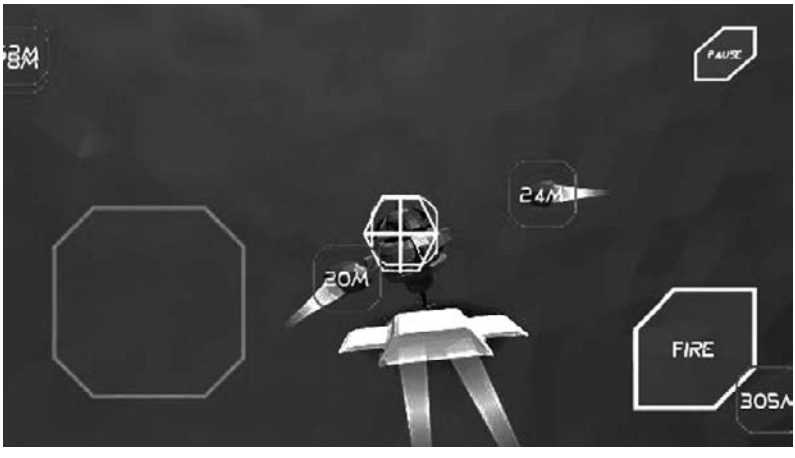

Рис. 13.14. Игра в действии

4. *Добавьте звук работающих двигателей*. Перетащите аудиоклип Engine в поле AudioClip.
5. *Сохраните изменения в шаблоне*. Вот и всё!

Добавить звуки, воспроизводимые непрерывно, очень просто, но этот шаг позволяет значительно улучшить восприятие игры с минимумом усилий с вашей стороны.

Пока не удаляйте корабль из сцены — чуть ниже мы добавим в него еще кое-что.

Выстрелы

Добавить звуки выстрелов немного сложнее. Нам нужно, чтобы звуковой эффект воспроизводился с каждым выстрелом, а это значит, что нам придется запускать воспроизведение программно.

Сначала добавим источники звука к двум пушкам.

1. *Добавьте компоненты Audio Source в объекты огневых точек*. Выберите оба объекта огневых точек — Fire Point 1 и Fire Point 2 — и добавьте компонент Audio Source.
2. *Добавьте в источники звука аудиоклип Laser*. После добавления снимите флажок Play On Awake — нам нужно, чтобы звук воспроизводился только в момент выстрела.
3. *Добавьте код, запускающий воспроизведение эффекта в момент выстрела*. Добавьте следующий код в сценарий *ShipWeapons.cs*:

```
public class ShipWeapons : MonoBehaviour {

  // Шаблон для создания снарядов
  public GameObject shotPrefab;

  public void Awake() {
    // Когда данный объект запускается, сообщить
    // диспетчеру ввода использовать его
    // как текущий сценарий управления оружием
    InputManager.instance.SetWeapons(this);
  }

  // Вызывается при удалении объекта
  public void OnDestroy() {
    // Ничего не делать, если вызывается не в режиме игры
    if (Application.isPlaying == true) {
      InputManager.instance.RemoveWeapons(this);
    }
  }

  // Список пушек для стрельбы
  public Transform[] firePoints;

  // Индекс в firePoints, указывающий на следующую
  // пушку
  private int firePointIndex;

  // Вызывается диспетчером ввода InputManager.
  public void Fire() {

    // Если пушки отсутствуют, выйти
    if (firePoints.Length == 0)
      return;

    // Определить следующую пушку для выстрела
    var firePointToUse = firePoints[firePointIndex];

    // Создать новый снаряд с ориентацией,
    // соответствующей пушке
    Instantiate(shotPrefab,
      firePointToUse.position,
      firePointToUse.rotation);

>   // Если пушка имеет компонент источника звука,
>   // воспроизвести звуковой эффект
>   var audio
>     = firePointToUse.GetComponent<AudioSource>();
>   if (audio) {
>     audio.Play();
>   }

    // Перейти к следующей пушке
    firePointIndex++;

    // Если произошел выход за границы массива,
    // вернуться к его началу
```

```
      if (firePointIndex >= firePoints.Length)
        firePointIndex = 0;

    }

}
```

Этот код проверяет наличие компонента AudioSource в огневой точке, выполняющей выстрел. Если компонент имеется, запускается воспроизведение его звукового эффекта.

4. *Сохраните изменения в шаблоне Ship и удалите его из сцены.*

Вот и всё. Теперь вы будете слышать звуковой эффект при каждом выстреле!

Взрывы

Осталось добавить последний звуковой эффект: звук взрыва. Сделать это просто: нужно всего лишь добавить источник звука Audio Source в объект взрыва Explosion и установить флажок Play On Awake. Когда взрыв появится на экране, автоматически будет воспроизведен звук взрыва Explosion.

1. *Добавьте шаблон Explosion в сцену.*
2. *Добавьте компонент Audio Source в объект Explosion.* Перетащите аудиоклип и установите флажок Play On Awake.
3. *Сохраните изменения в шаблоне и удалите его из сцены.*

Теперь вы будете слышать звуки взрывов!

В заключение

Вот мы и закончили. Поздравляем! В ваших руках законченная игра «Метеоритный дождь». Теперь вам решать, как развивать ее дальше!

Мы лишь подскажем несколько идей.

Добавьте новое вооружение

Например, ракету с самонаведением.

Добавьте врагов, атакующих игрока

Астероиды, прямиком летящие к космической станции, — слишком простая цель. Игроку практически ничего не угрожает.

Добавьте эффекты разрушения космической станции

Добавьте систему частиц, которая воспроизводит пламя и дым в точке попадания, когда астероид достигает станции. Это, конечно, нереалистично, но это не помешало нам добавить в игру и другие эффекты.

ЧАСТЬ IV
Дополнительные возможности

В этой части мы детально рассмотрим некоторые особенности Unity, начав с подробного исследования системы пользовательского интерфейса и закончив изучением возможности расширения самого редактора Unity. Мы также исследуем систему освещения и завершим экскурсию по Unity знакомством с приемами расширения инфраструктуры и обсуждением вопросов выхода и распространения своих игр.

14 Освещение и шейдеры

В этой главе мы рассмотрим освещение и материалы — самое главное (кроме текстур), что определяет внешний вид игры. В частности, мы подробнее рассмотрим стандартный шейдер (Standard), специально спроектированный, чтобы упростить создание материалов с привлекательным внешним видом. Мы также расскажем, как писать свои шейдеры, которые дадут вам широкие возможности управления внешним видом объектов в игре. Наконец, мы обсудим, как использовать глобальное освещение и карты освещенности, которые помогут создавать красивые сцены за счет реалистичного моделирования отражения света в сцене.

Материалы и шейдеры

Внешний вид объектов в Unity определяется подключенным к нему *материалом*. Материал состоит из двух компонентов: *шейдера* (shader) и данных, которые используются шейдером.

Шейдер — это очень маленькая программа, которая выполняется процессором графической карты. Все, что вы видите на экране, является результатом вычислений цвета для каждого пиксела, произведенных шейдером.

В Unity поддерживаются два основных типа шейдеров: *поверхностные* шейдеры (surface shaders) и *фрагмент-вершинные* шейдеры (fragment-vertex shaders).

Поверхностные шейдеры отвечают за вычисление цвета поверхности объекта. Цвет поверхности определяется несколькими свойствами, включая альбедо (отражающую способность), гладкость и т. п. Задача поверхностного шейдера — вычислить значение каждого свойства для каждого пиксела поверхности объекта; затем эта информация возвращается движку Unity, который объединяет ее с информацией о каждом источнике света в сцене и определяет окончательный цвет каждого пикселов.

Фрагмент-вершинный шейдер, напротив, намного проще. Шейдеры этого вида отвечают за вычисление окончательного цвета пикселов; если шейдер должен включить информацию об освещении, вам придется сделать это самостоятельно. Фрагмент-вершинные шейдеры дают возможность управления низкоуровневыми особенностями, то есть играют важную роль в создании визуальных эффектов. Они обычно намного проще поверхностных шейдеров и, как следствие, выполняются намного быстрее.

Поверхностные шейдеры фактически компилируются движком Unity во фрагмент-вершинные шейдеры, которые берут на себя вычисления освещенности для достижения максимальной реалистичности. Все, что делается в поверхностном шейдере, можно сделать во фрагмент-вершинном шейдере, но при этом придется приложить больше усилий.

Если у вас нет каких-то особых требований, предпочтительнее использовать поверхностные шейдеры. В этой главе мы рассмотрим оба вида.

Движок Unity поддерживает также третий тип шейдеров — *шейдеры с фиксированной функциональностью* (fixed-function). Вы не сможете сами писать шейдеры этого типа — они объединяют предопределенные операции. Шейдеры с фиксированной функциональностью были основным механизмом до появления поддержки пользовательских шейдеров; они не такие сложные и редко позволяют добиться высокой реалистичности, поэтому в наши дни не рекомендуются к использованию. В этой главе мы не будем говорить о шейдерах с фиксированной функциональностью, но если у вас появится желание познакомиться с ними, загляните в документацию к Unity, где найдете руководство по их использованию (*https://docs.unity3d.com/ru/current/Manual/ShaderTut1.html*).

А теперь создадим собственный поверхностный шейдер, очень похожий на стандартный, но обладающий дополнительной возможностью отображения подсветки сзади, то есть подсветки контура объекта. Пример такой подсветки можно видеть на рис. 14.1.

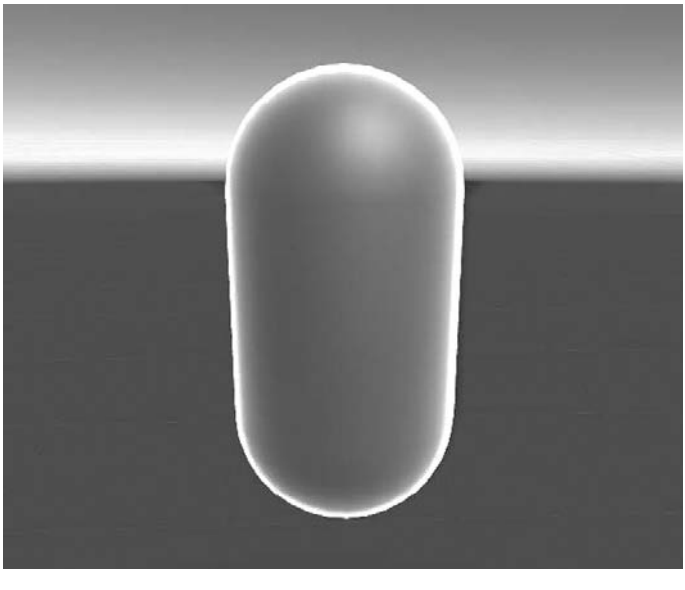

Рис. 14.1. Контурная подсветка, реализованная с применением нестандартного шейдера

Чтобы создать такой эффект, выполните следующие действия.

1. *Создайте новый проект*. Присвойте ему любое имя по своему выбору и выберите режим 3D.
2. *Создайте новый шейдер, открыв меню Assets (Ресурсы) и выбрав пункт Create ▸ Shader ▸ Surface Shader (Создать ▸ Шейдер ▸ Поверхностный шейдер)*. Присвойте новому шейдеру имя SimpleSurfaceShader.
3. *Дважды щелкните по нему*.
4. *Замените его содержимое следующим кодом*:

```
Shader "Custom/SimpleSurfaceShader" {

    Properties {

        // Цвет оттенка объекта
        _Color ("Color", Color) = (0.5,0.5,0.5,1)

        // Текстура для обертывания объекта;
        // по умолчанию используется плоская белая текстура
        _MainTex ("Albedo (RGB)", 2D) = "white" {}

        // Гладкость поверхности
        _Smoothness ("Smoothness", Range(0,1)) = 0.5

        // Насколько металлической должна выглядеть поверхность
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }

    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
            // Физически основана на стандартной модели освещения
            // и поддерживает тени от источников света всех типов
            #pragma surface surf Standard fullforwardshadows

            // Использовать модель шейдеров версии 3.0, чтобы
            // получить более реалистичное освещение
            #pragma target 3.0

            // Следующие переменные являются "униформными" - одно и то же
            // значение действует для каждого пиксела

            // Текстура, определяющая альбедо (отраженный свет)
            sampler2D _MainTex;

            // Цвет оттенка отраженного света
            fixed4 _Color;

            // Свойства гладкости и металличности
            half _Smoothness;
            half _Metallic;
```

```
            // Структура 'Input' содержит переменные, которые получают
            // уникальные значения для каждого пиксела
            struct Input {
                // Координаты в текстуре для данного пиксела
                float2 uv_MainTex;

            };

            // Эта единственная функция вычисляет свойства
            // данной поверхности
            void surf (Input IN, inout SurfaceOutputStandard o) {

                // Использует данные в IN и переменные,
                // объявленные выше, вычисляет значения
                // и сохраняет их в 'o'

                // Альбедо извлекается из текстуры и окрашивается
                // цветом оттенка
                fixed4 c =
                    tex2D (_MainTex, IN.uv_MainTex) * _Color;
                o.Albedo = c.rgb;

                // Металличность и гладкость извлекаются
                // из текущих переменных
                o.Metallic = _Metallic;
                o.Smoothness = _Smoothness;

                // Значение альфа-канала извлекается
                // из текстуры, используемой для определения альбедо
                o.Alpha = c.a;
            }
        ENDCG
    }

    // Если компьютер, выполняющий этот шейдер, не поддерживает
    // модель шейдеров 3.0, использовать встроенный
    // шейдер "Diffuse", который выглядит не так реалистично,
    // но гарантированно работает
    FallBack "Diffuse"
}
```

5. *Создайте новый материал с именем SimpleSurface.*
6. *Выберите новый материал и откройте меню Shader в верхней части инспектора.* Выберите пункт Custom ▸ SimpleSurfaceShader (Пользовательский ▸ Simple SurfaceShader).

   В инспекторе появятся настройки свойств поверхностного шейдера (рис. 14.2).
7. *Создайте новую капсулу*, выбрав пункт меню GameObject ▸ 3D Object ▸ Capsule (Игровой объект ▸ 3D Объект ▸ Капсула).
8. *Перетащите материал SimpleShader* на капсулу, и для ее отображения будет использоваться новый материал (рис. 14.3).

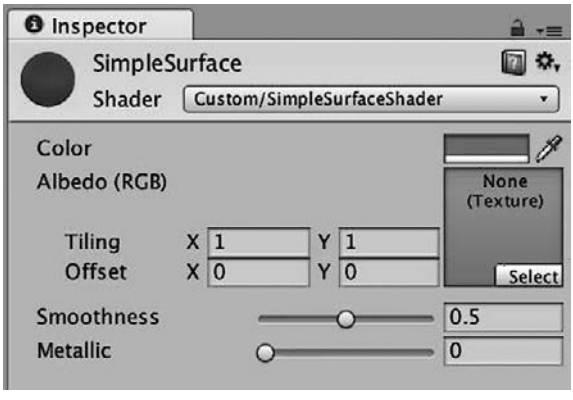

Рис. 14.2. Настройки нестандартного пользовательского шейдера

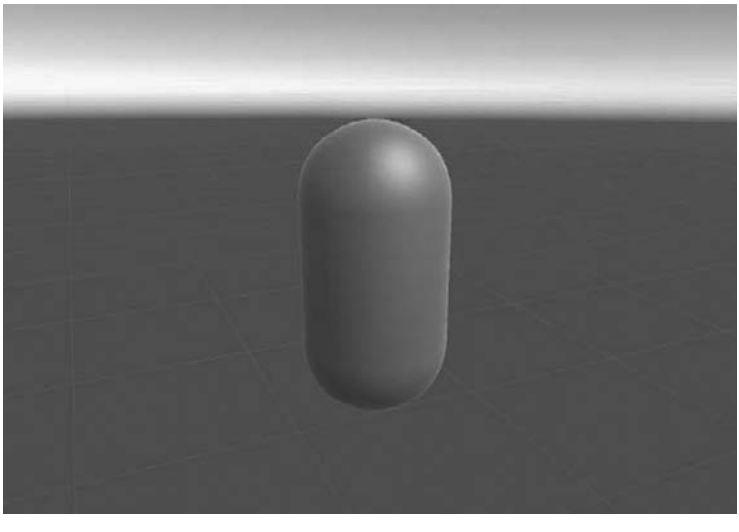

Рис. 14.3. Изображение капсулы, полученное с использованием нестандартного шейдера

Сейчас объект выглядит почти так же, как при отображении с использованием стандартного шейдера. Давайте теперь добавим освещение сзади!

Для вычисления освещения, падающего сзади, нужно знать три характеристики:

- желаемый цвет освещения;
- толщину контура, создаваемого подсветкой;
- угол между направлением визирования камеры и направлением ориентации поверхности.

Направление ориентации поверхности называется *нормалью* к поверхности. Код, который мы напишем, будет использовать этот термин.

Первые две характеристики являются *униформными* (uniform), то есть их значения применяются ко всем пикселам объекта. Третья характеристика имеет *переменный* (varying) характер, то есть ее значение зависит от направления взгляда; угол между направлением визирования камеры и нормалью к поверхности зависит от того, смотрите вы в середину цилиндра или на его край.

Переменные значения вычисляются графической картой во время выполнения для использования вашим поверхностным шейдером, тогда как униформные значения определяются свойствами материала, которые можно менять в инспекторе. То есть чтобы добавить поддержку подсветки сзади, прежде всего нужно добавить в шейдер две униформные переменные.

1. *Измените раздел Properties шейдера*, добавив следующий код:

```
Properties {

    // Цвет оттенка объекта
    _Color ("Color", Color) = (0.5,0.5,0.5,1)

    // Текстура для обертывания объекта;
    // по умолчанию используется плоская белая текстура
    _MainTex ("Albedo (RGB)", 2D) = "white" {}

    // Гладкость поверхности
    _Smoothness ("Smoothness", Range(0,1)) = 0.5

    // Насколько металлической должна выглядеть поверхность
    _Metallic ("Metallic", Range(0,1)) = 0.0

>   // Цвет света, падающего сзади
>   _RimColor ("Rim Color", Color) = (1.0,1.0,1.0,0.0)
>
>   // Толщина контура, создаваемого подсветкой
>   _RimPower ("Rim Power", Range(0.5,8.0)) = 2.0

}
```

После добавления этого кода в настройках шейдера в инспекторе появятся два новых поля. Теперь эти свойства нужно сделать доступными в коде шейдера, чтобы функция `surf` могла использовать их.

2. *Добавьте в шейдер следующий код*:

```
    // Свойства гладкости и металличности
    half _Smoothness;
    half _Metallic;

> // Цвет света, падающего сзади
> float4 _RimColor;
```

```
>
> // Толщина контура, создаваемого подсветкой, - чем ближе
> // к нулю, тем тоньше контур
> float _RimPower;
```

Далее нужно дать шейдеру возможность определять направление визирования камеры. Все переменные значения, используемые шейдером, включены в структуру `Input`, следовательно, направление визирования камеры нужно добавить туда.

В структуру можно добавить несколько полей, которые движок Unity будет заполнять автоматически. Если добавить переменную `viewDir` типа `float3`, Unity поместит в нее направление визирования камеры.

`viewDir` — не единственная переменная, которая автоматически используется движком Unity для передачи изменяющейся информации. Полный список вы найдете в документации к Unity в разделе «Разработка поверхностных шейдеров» (*https://docs.unity3d.com/ru/current/Manual/SL-SurfaceShaders.html*).

3. *Добавьте следующий код в определение структуры* `Input`*:*

```
struct Input {
    // Координаты в текстуре для данного пиксела
    float2 uv_MainTex;

>   // Направление визирования камеры для данной вершины
>   float3 viewDir;
 };
```

Теперь в настройках материала, в панели инспектора, появятся дополнительные поля (рис. 14.4).

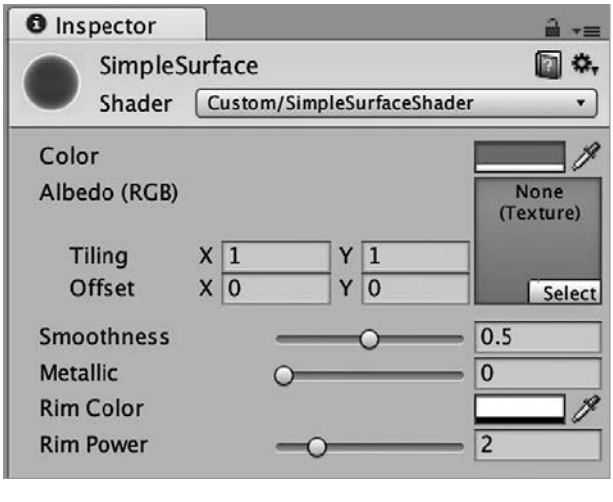

Рис. 14.4. В инспекторе появились вновь добавленные поля

Теперь у нас есть вся информация, необходимая для вычисления параметров контура, создаваемого подсветкой сзади; осталось сделать последний шаг — реализовать фактические вычисления и добавить их результаты в информацию о поверхности.

4. *Добавьте следующий код в функцию* `surf`:

```
// Эта единственная функция вычисляет свойства
// данной поверхности
void surf (Input IN, inout SurfaceOutputStandard o) {

    // Использует данные в IN и переменные,
    // объявленные выше, вычисляет значения
    // и сохраняет их в 'o'

    // Альбедо извлекается из текстуры и окрашивается
    // цветом оттенка
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    o.Albedo = c.rgb;

    // Металличность и гладкость извлекаются
    // из текущих переменных
    o.Metallic = _Metallic;
    o.Smoothness = _Smoothness;

    // Значение альфа-канала извлекается
    // из текстуры, используемой для определения альбедо
    o.Alpha = c.a;

>   // Вычислить яркость освещенности данного пиксела
>   // источником света сзади
>   half rim =
>     1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
>
>   // Использовать эту яркость для вычисления цвета контура
>   // и использовать ее для определения цвета отраженного света
>   o.Emission = _RimColor.rgb * pow (rim, _RimPower);

  }
```

5. *Сохраните шейдер и вернитесь в Unity.* Теперь капсула получит светлый контур! Результат можно видеть на рис. 14.5.

Скорректировать параметры освещенного контура можно, изменяя свойства материала. Попробуйте изменить параметр Rim Color, чтобы изменить яркость и оттенок света, и Rim Power, чтобы изменить толщину видимого контура.

Поверхностные шейдеры отлично подходят для использования на основе существующей системы вычисления освещенности и являются лучшим выбором, когда требуется, чтобы поверхности объектов реагировали на источники света, добавляемые в сцену. Однако иногда реалистичность освещенности не важна или требуется иметь очень специфический контроль над внешним видом поверхно-

сти. В таких ситуациях можно реализовать свои, полностью нестандартные, фрагмент-вершинные шейдеры.

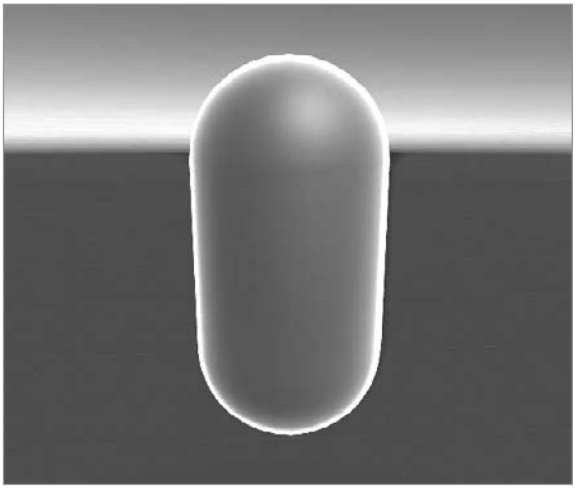

Рис. 14.5. Капсула с контуром, созданным подсветкой сзади

Фрагмент-вершинные шейдеры (без освещения)

Фрагмент-вершинные шейдеры получили такое название, потому что фактически состоят из двух шейдеров: *фрагментного* (fragment) и *вершинного* (vertex). Это две отдельные функции, управляющие внешним видом и отображением поверхности.

Вершинный шейдер — это функция, которая в процессе подготовки к отображению преобразует каждую вершину, то есть каждую точку объекта, из координат мирового пространства в координаты видимого пространства. Под мировым пространством в данном случае понимается пространство, наблюдаемое в редакторе Unity: объекты позиционируются в пространстве, и вы можете перемещать их с места на место. Однако чтобы отобразить сцену с использованием камеры, движок Unity должен сначала преобразовать позиции всех объектов, присутствующих в сцене, в координаты *видимого пространства* — пространства, в котором все мировое пространство и все его объекты позиционируются так, что центром мира является камера. Кроме того, объекты в видимом пространстве масштабируются так, что более далекие от камеры объекты имеют меньшие размеры. Вершинный шейдер отвечает также за вычисление значений переменчивых параметров, которые должны передаваться фрагментному шейдеру.

Вам практически никогда не придется писать свои вершинные шейдеры, но иногда это умение может пригодиться. Например, если понадобится исказить форму объекта, вы можете написать свой вершинный шейдер, изменяющий позицию каждой вершины.

Фрагментный шейдер — вторая половина пары. Фрагментные шейдеры отвечают за вычисление окончательного цвета каждого фрагмента — обычно пиксела — объекта. Фрагментный шейдер получает параметры с переменными значениями, вычисляемые вершинным шейдером; эти значения *интерполируются*, или смешиваются, с учетом расстояний от отображаемого фрагмента до ближайших вершин.

Поскольку фрагментный шейдер имеет полный контроль над окончательным цветом объекта, в нем можно рассчитать эффект влияния близко расположенных источников света. Если шейдер сам не выполняет таких вычислений, поверхность не будет выглядеть освещенной.

Именно по этой причине для реализации освещения рекомендуется использовать поверхностные шейдеры; расчет освещения может быть очень сложным и часто намного проще не заставлять себя думать о нем.

В действительности поверхностные шейдеры являются фрагмент-вершинными шейдерами. Движок Unity автоматически преобразует поверхностные шейдеры в низкоуровневый фрагмент-вершинный код и добавляет в него расчет освещенности.

Недостаток такого решения в том, что поверхностные шейдеры проектируются для обобщенных случаев и могут оказаться менее эффективными, чем шейдеры, написанные вручную.

Для демонстрации особенностей работы фрагмент-вершинных шейдеров создадим простой шейдер, отображающий равномерно окрашенные одноцветные объекты. Затем мы преобразуем его, чтобы он отображал градиентную заливку, в зависимости от местоположения объекта на экране.

1. *Создайте новый шейдер, открыв меню Assets (Ресурсы) и выбрав пункт Create ▸ Shader ▸ Unlit Shader (Создать ▸ Шейдер ▸ Шейдер без освещения).* Дайте новому шейдеру имя SimpleUnlitShader.
2. *Выполните двойной щелчок на нем, чтобы открыть.*
3. *Замените содержимое файла следующим кодом:*

```
Shader "Custom/SimpleUnlitShader"
{
    Properties
    {
        _Color ("Color", Color) = (1.0,1.0,1.0,1)
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM

            // Определения функций,
```

```
// используемых данным шейдером.

// Функция 'vert' будет использоваться в качестве
// вершинного шейдера.
#pragma vertex vert

// Функция 'frag' будет использоваться в качестве
// фрагментного шейдера.
#pragma fragment frag

// Подключение некоторых удобных
// утилит из Unity.
#include "UnityCG.cginc"

float4 _Color;

// Эта структура передается в вершинный шейдер
// для каждой вершины
struct appdata
{
    // Позиция вершины в мировом пространстве.
    float4 vertex : POSITION;

};

// Эта структура передается в фрагментный шейдер
// для каждого фрагмента
struct v2f
{
    // Позиция фрагмента в
    // экранных координатах
    float4 vertex : SV_POSITION;
};

// Получает вершину и преобразует ее
v2f vert (appdata v)
{
    v2f o;

    // Преобразовать вершину из мировых координат
    // в видимые умножением на матрицу,
    // переданную движком Unity (из UnityCG.cginc)
    o.vertex = UnityObjectToClipPos(v.vertex);

    // Вернуть вершину для последующей передачи
    // фрагментному шейдеру
    return o;
}

// Интерполирует информацию о ближайших вершинах
// и возвращает окончательный цвет
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col;
```

```
                // вернуть исходный цвет
                col = _Color;

                return col;
            }
            ENDCG
        }
    }
}
```

4. *Создайте новый материал, открыв меню Assets (Ресурсы) и выбрав пункт Create ▸ Material (Создать ▸ Материал).* Дайте материалу имя SimpleShader.
5. *Выберите новый материал* и измените его шейдер на Custom ▸ SimpleUnlitShader (Пользовательский ▸ SimpleUnlitShader).
6. *Создайте в сцене сферу,* открыв меню GameObject (Игровой объект) и выбрав пункт 3D Object ▸ Sphere (3D Объект ▸ Сфера). Перетащите на него материал SimpleShader.

В результате должна получиться сфера, окрашенная однотонным цветом, как показано на рис. 14.6.

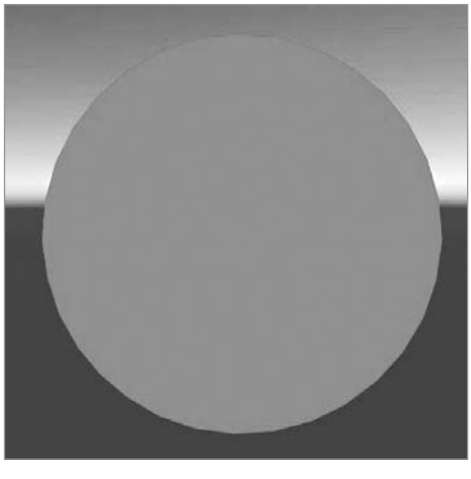

Рис. 14.6. Сфера, окрашенная в однотонный цвет

Заливка объекта однотонным цветом настолько распространенная задача, что в Unity был включен встроенный шейдер, очень похожий на тот, что мы написали выше. Вы можете найти его в меню Shader в виде пункта Unlit ▸ Color (Без освещения ▸ Цвет).

Далее попробуем использовать этот шейдер для воспроизведения анимационного эффекта. Для этого не придется писать сценарий; все необходимое мы реализуем внутри шейдера.

1. *Добавьте следующий код в функцию frag*:

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col;

    // вернуть исходный цвет
    col = _Color;

>   // Плавно изменять цвет с течением времени - от черного до _Color
>   col *= abs(_SinTime[3]);

    return col;
}
```

2. *Вернитесь в Unity* и обратите внимание, что цвет объекта изменился на черный. Так и должно быть.
3. *Щелкните по кнопке Play (Играть)* и понаблюдайте, как изменяется цвет объекта. Промежуточные этапы можно увидеть на рис. 14.7.

Рис. 14.7. Постепенное изменение цвета объекта

Как видите, фрагмент-вершинные шейдеры позволяют получить полный контроль над внешним видом объектов, и этот раздел помог вам почувствовать это. Обсуждение данной темы во всех подробностях заняло бы целую книгу, поэтому мы не будем останавливаться на ней, а если вас заинтересовала тема использования шейдеров, то обращайтесь к подробной документации Unity (*https://docs.unity3d.com/ru/current/Manual/SL-Reference.html*).

Глобальное освещение

Когда объект освещается, шейдер, отвечающий за его отображение, должен выполнить ряд сложных вычислений, определить количество света, получаемого объектом, и на основе этой величины рассчитать цвет, который видит камера. Это нормально, но некоторые величины очень сложно рассчитать во время выполнения.

Например, если сфера покоится на белой поверхности и освещается прямым солнечным светом, она должна подсвечиваться снизу светом, отраженным от белой

поверхности. Однако шейдеру известно только направление на солнце, и, как результат, он не отображает эту подсветку. Конечно, такую подсветку можно рассчитать, но это сразу превращается в очень сложную задачу, чтобы решать ее в каждом кадре.

Лучшее решение заключается в использовании *глобального освещения* (global illumination) и *карт освещенности* (lightmapping). Название «глобальное освещение» объединяет множество родственных технологий вычисления количества света, получаемого каждой поверхностью в сцене, с учетом света, отраженного объектами.

Применение методов глобального освещения позволяет получить очень правдоподобную картину, но требует значительных вычислительных ресурсов. По этой причине вычисления освещенности часто выполняют заблаговременно в редакторе Unity. Результаты также могут быть сохранены в так называемой *карте освещенности* (lightmap), которая определяет количество света, получаемое каждым фрагментом каждой поверхности в сцене.

Поскольку вычисления глобального освещения выполняются заранее, нетрудно догадаться, что данный прием применим только к неподвижным объектам (соответственно, не изменяющим направление отраженного света). Ни для какого движущегося объекта нельзя применить методы глобального освещения непосредственно; в таких случаях нужно другое решение, о котором мы расскажем чуть ниже.

Использование карт освещенности может значительно увеличить производительность создания реалистичного освещения в сцене, так как вычисления выполняются заранее, а их результаты сохраняются в текстурах. Однако чтобы использовать эти текстуры для отображения, их нужно загружать в память. Это может стать проблемой для сложных сцен, использующих много сложных текстур. Сгладить эту проблему можно за счет уменьшения разрешения карт освещенности, но при этом снизится визуальное качество освещения.

С учетом вышеизложенного углубимся в использование приемов глобального освещения, создав сцену. Сначала настроим материалы разного цвета — это поможет нам увидеть, как свет распространяется в сцене, а затем создадим несколько объектов и настроим использование системы глобального освещения.

1. *Создайте новую сцену в Unity.*
2. *Создайте новый материал с именем Green.* Оставьте выбранным шейдер Standard и измените цвет в свойстве Albedo на зеленый. Настройки материала в инспекторе должны выглядеть так, как показано на рис. 14.8.

Теперь добавим объекты в сцену.

3. *Создайте куб,* открыв меню GameObject (Игровой объект) и выбрав пункт 3D Object ▸ Cube (3D Объект ▸ Куб). Дайте объекту имя floor, установите его в позицию 0,0,0 и задайте масштаб 10,1,10.
4. *Создайте второй куб* с именем Wall 1. Установите его в позицию −1,3,0 и задайте поворот 0,45,0. Также задайте масштаб 1,5,4.

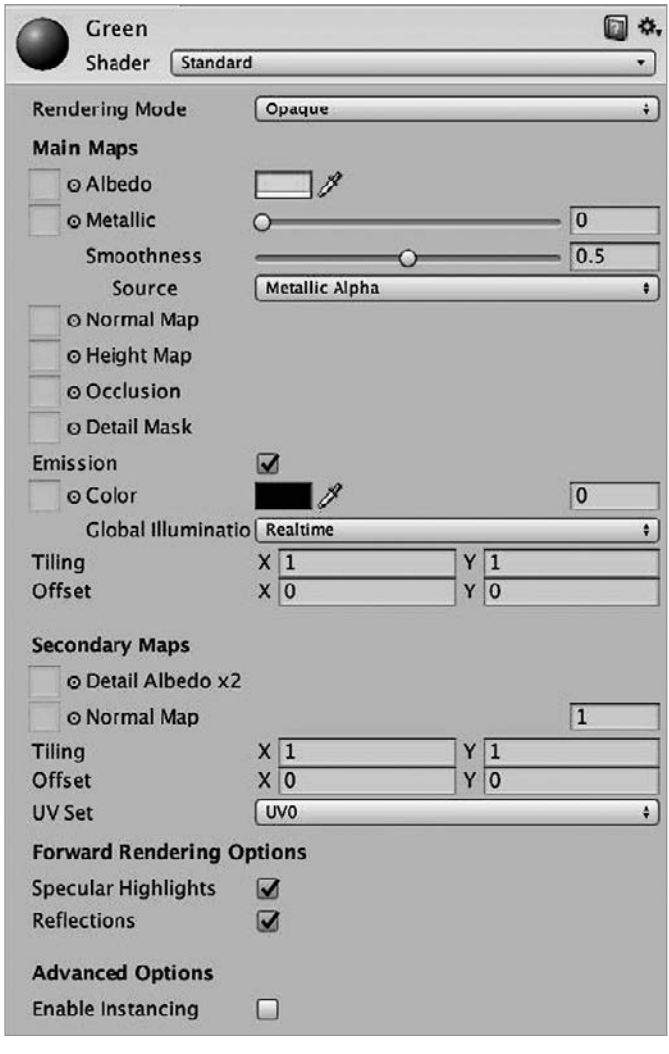

Рис. 14.8. Настройки материала Green

5. *Создайте третий куб* с именем Wall 2. Он должен находиться в позиции 2,3,0 с поворотом 0,45,0 и иметь масштаб 1,5,4.
6. *Перетащите материал Green на объект Wall 1.*

Сцена должна выглядеть так, как показано на рис. 14.9.

Теперь заставим Unity вычислить освещенность.

7. Выберите все три объекта — floor, Wall 1 и Wall 2 — и установите флажок Static (Статический) справа вверху в инспекторе (рис. 14.10).

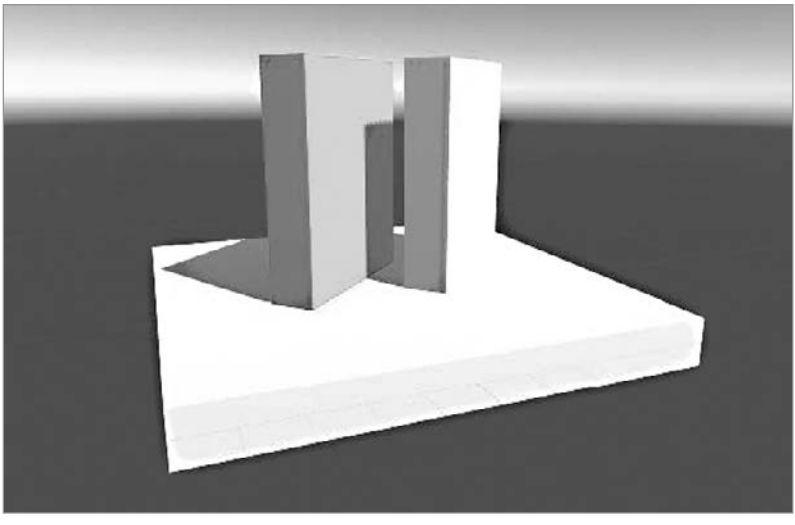

Рис. 14.9. Сцена без наложения карты освещенности

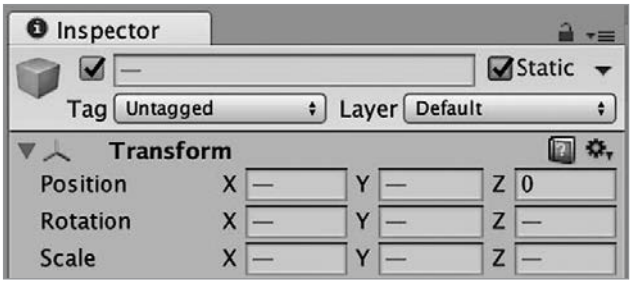

Рис. 14.10. Объявление объектов статическими

Как только объекты в сцене будут объявлены статическими, Unity немедленно приступит к вычислению информации об их освещенности. Через несколько мгновений картина освещенности немного изменится. Наиболее заметным изменением станет появление на белом кубе светового пятна, отбрасываемого зеленым кубом. Сравните рис. 14.11 и 14.12.

Этот прием позволяет использовать глобальное освещение в режиме реального времени. Он дает неплохой результат, но влечет значительное ухудшение производительности, потому что заранее делается только часть вычислений. Чтобы улучшить производительность за счет увеличенного потребления памяти, можно зафиксировать освещенность в виде карты освещенности.

8. *Выберите направленный свет и установите свойство Baking в значение* `Baked`.

Через мгновение освещенность будет вычислена и сохранена в карте освещенности.

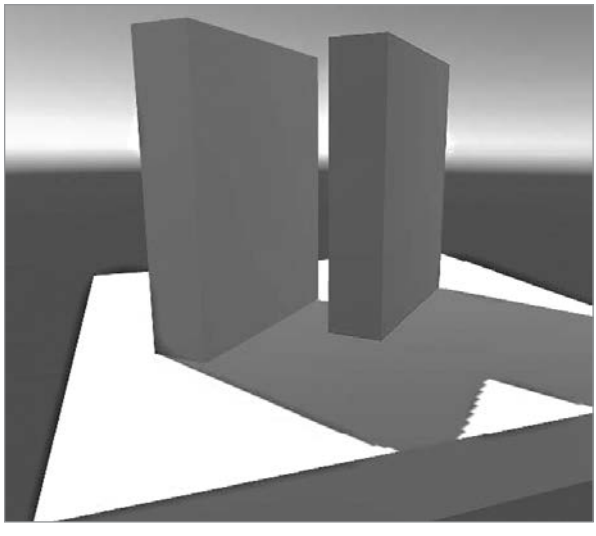

Рис. 14.11. Сцена с выключенным глобальным освещением

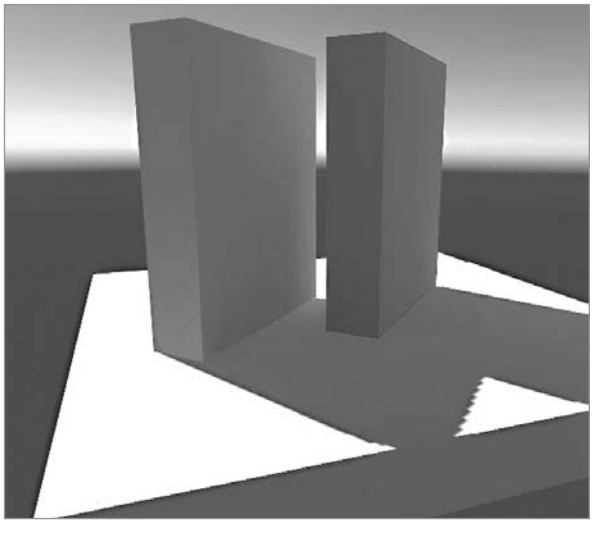

Рис. 14.12. Сцена с включенным глобальным освещением — обратите внимание на зеленое отражение на белом кубе

Прием глобального освещения дает хорошие результаты для статических объектов, но он не оказывает влияния на освещенность движущихся объектов. Чтобы исправить эту проблему, можно воспользоваться зондами освещения (light probes).

Зонды освещения

Зонд освещения (light probe) — это невидимый объект, который регистрирует свет, поступающий со всех направлений, и запоминает его. Нестатические объекты могут использовать эту информацию об освещенности и освещать себя.

Зонды освещения не работают по отдельности. Их нужно создавать группами; во время выполнения объекты, которым требуется информация об освещении, объединяют сведения, полученные с ближайших зондов, с учетом их удаленности от объекта. Это, например, дает объектам возможность отразить больше света при приближении к поверхности, отражающей свет.

Добавим нестатический объект в сцену, а затем добавим несколько зондов освещения, чтобы увидеть их влияние на освещенность.

1. *Добавьте в сцену новую капсулу*, выбрав пункт меню GameObject ▸ 3D Object ▸ Capsule (Игровой объект ▸ 3D Объект ▸ Капсула). Поместите капсулу где-нибудь поблизости от зеленого куба.

Обратите внимание, что капсула не освещается светом, отраженным от зеленого куба. Фактически она получает *слишком много* света, распространяющегося в направлении от куба, — света, падающего с неба. Этот свет должен блокироваться кубом. Вы можете видеть этот недостаток на рис. 14.13.

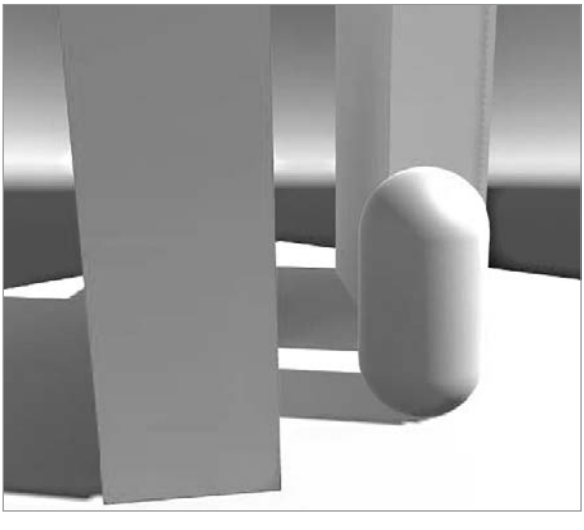

Рис. 14.13. Сцена без зондов освещения

2. *Добавьте несколько зондов освещения*, открыв меню GameObject (Игровой объект) и выбрав пункт Light ▸ Light Probe Group (Свет ▸ Группа зондов освещения).

В сцене появится несколько сфер. Каждая сфера представляет один зонд освещения и показывает освещенность в этой точке пространства.

3. *Расположите зонды* так, чтобы ни один из них не находился внутри какого-либо объекта — то есть все они должны находиться в свободном пространстве, за пределами кубов.

Чтобы изменить местоположение отдельных зондов, выберите группу и щелкните по кнопке Edit Light Probes (Редактировать зонды освещения); после этого вы сможете выбирать отдельные зонды и перемещать их.

После этого капсула должна получить дополнительный зеленый свет, отраженный зеленым кубом, — сравните рис. 14.13 и 14.14.

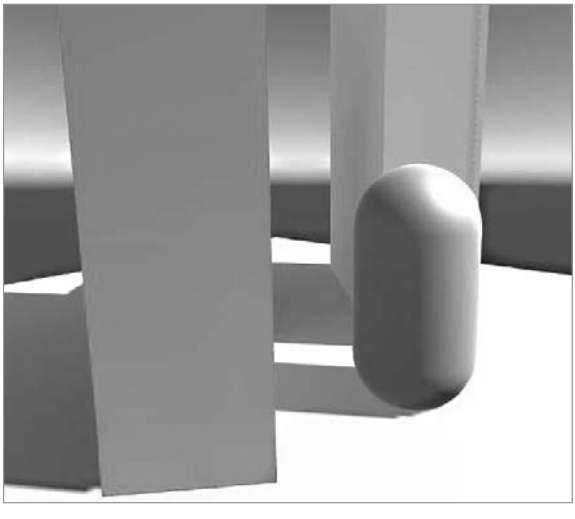

Рис. 14.14. Сцена после добавления зондов освещения — обратите внимание на зеленое отражение на капсуле

Чем больше зондов, тем больше времени будет уходить на вычисления. Кроме того, если в сцене имеются резкие перепады освещенности, зонды следует расположить плотнее в пограничных областях, чтобы объекты не выглядели неуместными.

Размышления о производительности

Прежде чем завершить эту главу, поговорим о производительности инструментов, встроенных в Unity. Настройки освещенности, используемые в игре, могут существенно влиять на производительность пользовательского устройства; то есть наряду с глобальным освещением и картами появляются и другие проблемы с производительностью, вызванные объявлением объектов статическими.

Однако производительность игр не всегда зависит от графики: время, на которое сценарии занимают центральный процессор, тоже имеет большое значение.

К счастью, в состав Unity входит несколько инструментов, которые можно с успехом использовать для повышения производительности.

Профайлер

Профайлер — это инструмент, записывающий данные в процессе выполнения игры. Из нескольких источников он собирает информацию о каждом кадре, такую как:

- методы сценариев, вызывавшиеся в каждом кадре, и время, затраченное на их выполнение;
- количество «вызовов рисования», то есть команд, посланных графическому процессору для вывода графики и потребовавшихся для отображения кадра;
- объем памяти, использованный игрой, — оперативной памяти и памяти графической карты;
- процессорное время, потребовавшееся на проигрывание звуков;
- количество активных физических тел и количество физических столкновений, возникших при обработке кадра;
- объем данных, отправленных в Сеть и полученных из Сети.

Окно профайлера разбито пополам. Верхняя половина делится на несколько горизонтальных панелей — по одной для каждого регистратора данных из перечисленных выше. Скриншот окна профайлера можно увидеть на рис. 14.15. В процессе выполнения игры каждый регистратор заполняется информацией. В нижней половине окна выводится подробная информация о конкретном кадре, выбранном в том или ином регистраторе.

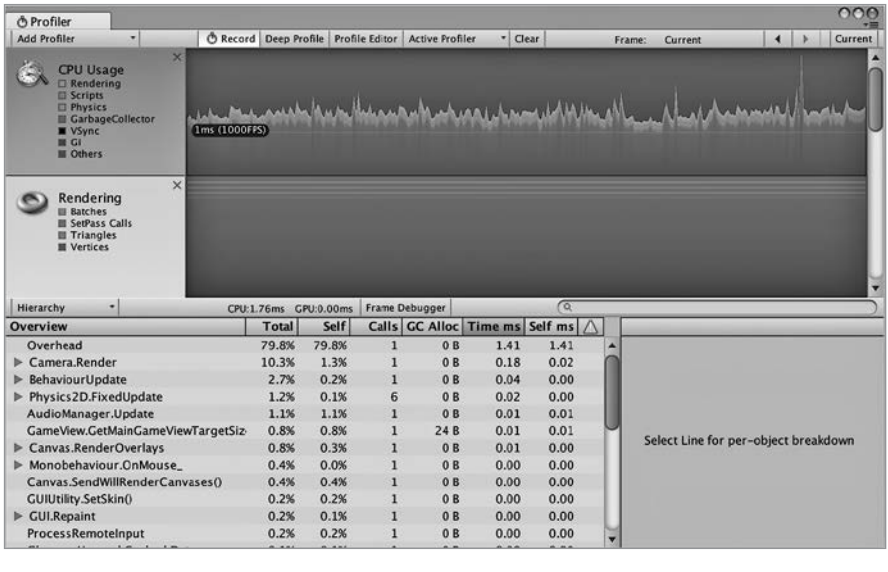

Рис. 14.15. Профайлер

Конкретные результаты, которые вы увидите в этой книге, необязательно будут совпадать с тем, что вы увидите у себя. Многое зависит от используемых аппаратных средств — компьютера, на котором выполняется Unity, мобильного устройства, на котором вы тестируете игру, а также от используемой версии Unity. Unity Technologies постоянно развивает движок, поэтому вы наверняка увидите иные результаты.

Однако сбор данных о производительности игры *всегда будет* выполняться одинаково, и вы сможете применить описываемые приемы к любой игре.

1. *Запустите профайлер, открыв меню* Window *(Окно) и выбрав пункт Profiler (Профайлер).* Также можно нажать комбинацию Command-7 на Mac или Ctrl-7 на PC. После этого появится окно профайлера.

   Чтобы начать пользоваться профайлером, достаточно просто запустить его, пока выполняется игра.

2. *Запустите игру*, щелкнув по кнопке Play (Играть) или нажав комбинацию Ctrl-P (Command-P на Mac).

   Профайлер начнет заполняться информацией. Анализировать данные намного проще, когда игра не выполняется, поэтому прежде чем продолжить, остановите игру.

3. *Спустя несколько мгновений остановите или приостановите игру.* Данные, накопленные в окне профайлера, никуда не исчезнут.

   Теперь, после остановки сбора данных, можно подробно изучить отдельные кадры.

4. *Щелкните левой кнопкой мыши на верхней панели и, удерживая кнопку мыши нажатой, протяните указатель по вертикали.* В панели появится вертикальная линия, и нижняя половина окна профайлера заполнится данными о выбранном кадре.

Для разных регистраторов отображается разная информация. Если выбран регистратор CPU Usage (Использование процессора), для него внизу по умолчанию отображается иерархия вызовов — список всех методов, вызывавшихся в кадре и отсортированных в порядке уменьшения времени, затраченного на вызов (рис. 14.16). В списке можно щелкать на треугольниках слева, чтобы раскрыть ту или иную строку и увидеть информацию о вызовах методов, соответствующих этой строке.

Мы немного подробнее остановимся на профайлере CPU, потому что знание и понимание информации, предоставляемой им, может помочь вам выявить и исправить самые разные проблемы производительности в играх.

Колонки в представлении иерархии отображают разную информацию для каждой строки.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	67.1%	67.1%	1	0 B	2.72	2.72
Overhead	24.4%	24.4%	1	0 B	0.99	0.99
▶ Camera.Render	4.3%	0.4%	1	0 B	0.17	0.02
▶ BehaviourUpdate	1.0%	0.1%	1	0 B	0.04	0.00
Profiler.FinalizeAndSendFrame	0.4%	0.4%	1	0 B	0.01	0.01
GameView.GetMainGameViewTargetSiz	0.4%	0.4%	1	24 B	0.01	0.01
▶ Canvas.RenderOverlays	0.3%	0.1%	1	0 B	0.01	0.00
AudioManager.Update	0.2%	0.2%	1	0 B	0.01	0.01
▶ Monobehaviour.OnMouse_	0.1%	0.0%	1	0 B	0.00	0.00
Canvas.SendWillRenderCanvases()	0.1%	0.1%	1	0 B	0.00	0.00
GUIUtility.SetSkin()	0.1%	0.1%	1	0 B	0.00	0.00
▶ Physics2D.FixedUpdate	0.1%	0.0%	1	0 B	0.00	0.00

Рис. 14.16. Представление иерархии профайлера CPU

Total (Всего)

В данной колонке выводится объем времени в процентах, затраченного этим методом и методами, вызываемыми им, при отображении данного кадра.

На рис. 14.16, например, видно, что на выполнение метода `Camera.Render` (внутренний метод движка Unity) затрачено 4,3 % времени, потребовавшегося на отображение кадра.

Self (Свое)

В данной колонке выводится объем времени в процентах, затраченного этим и *только* этим методом при отображении данного кадра. Эта информация помогает понять, где тратится основное время — в данном методе или в методах, которые он вызывает. Если значение в колонке Self (Свое) близко к значению в колонке Total (Всего), значит, основное время тратится в самом методе, а не в методах, которые он вызывает.

На рис. 14.16, например, видно, что сам метод `Camera.Render` потребляет только 0,4 % времени, затраченного на отображение кадра, что говорит о том, что сам метод менее затратный, но методы, которые он вызывает, тратят намного больше времени.

Calls (Вызовов)

В данной колонке выводится количество вызовов этого метода, выполненных при отображении данного кадра.

На рис. 14.16, например, видно, что метод `Camera.Render` вызывался только один раз (скорее всего, потому, что в сцене присутствует только одна камера).

GC Alloc (Распределений памяти)

В данной колонке выводится объем памяти, выделенной этим методом при отображении данного кадра. Если память выделяется слишком часто, это увеличивает вероятность запуска сборщика мусора позднее, что может привести к задержкам в игре.

На рис. 14.16, например, видно, что `GameView.GetMainGameViewTargetSize` выделил 24 байта памяти. На первый взгляд этот объем кажется незначительным,

но не забывайте, что игра стремится отобразить как можно больше кадров; с течением времени маленькие объемы, выделяемые в каждом кадре, могут сложиться в существенный объем и вызвать запуск сборщика мусора, что отрицательно скажется на производительности.

Time ms (Время мс)

В данной колонке выводится объем времени в миллисекундах, затраченного этим методом и методами, вызываемыми им. На рис. 14.16, например, видно, что вызов `Camera.Render` затратил 0,17 миллисекунды.

Self ms (Свое мс)

В данной колонке выводится объем времени в миллисекундах, затраченного этим и *только* этим методом. На рис. 14.16, например, видно, что на выполнение самого метода `Camera.Render` затрачено всего 0,02 миллисекунды; остальные 0,15 миллисекунды были затрачены на выполнение вызываемых им методов.

Warnings (Предупреждения)

В данной колонке выводится информация о проблемах, обнаруженных профайлером. Профайлер применяет некоторые виды анализа к записываемым данным и может давать ограниченное количество рекомендаций.

Получение данных с устройства

Шаги, перечисленные в предыдущем разделе, описывают работу с информацией, которую собирает редактор Unity. Однако в редакторе игра имеет иную производительность, чем на устройстве. Персональные компьютеры обычно оснащены намного более быстрыми процессорами и обладают большим объемом оперативной памяти, чем мобильные устройства. Соответственно, результаты, полученные в профайлере, будут отличаться, и оптимизация узких мест, наблюдаемых при выполнении игры под управлением редактора, может не улучшить производительность на устройстве конечного пользователя.

Чтобы решить эту проблему, профайлер можно настроить на сбор данных при выполнении игры на устройстве. Для этого выполните следующие действия.

1. *Соберите и установите игру на свое устройство, как описано в разделе «Развертывание» в главе 17.* Важно: не забудьте установить флажки Development Build (Отладочная сборка) и Autoconnect Profiler (Автоматически подключать профайлер).
2. *Убедитесь, что ваши устройство и компьютер находятся в одной сети Wi-Fi* и устройство соединено с компьютером кабелем USB.
3. *Запустите игру на устройстве.*
4. *Запустите профайлер и откройте меню Active Profiler (Активный профайлер).* Выберите свое устройство из списка.

Профайлер начнет сбор данных непосредственно с устройства.

Общие советы

Ниже перечислено, что можно сделать для повышения производительности игры.

- Старайтесь удерживать значение счетчика Verts (Вершин) в профайлере Rendering (Отображение) меньше 200 000 на кадр.
- Выбирая шейдеры для использования в игре, отдавайте предпочтение шейдерам из категорий Mobile (Мобильный) и Unlit (Без освещения). Эти шейдеры проще и на их выполнение тратится меньше времени по сравнению с другими.
- Старайтесь использовать в сцене как можно меньше разных материалов. Кроме того, попытайтесь использовать *один и тот же* материал в как можно большем количестве объектов. Благодаря этому Unity сможет рисовать такие объекты одновременно, что несомненно улучшит производительность.
- Если объект в сцене не будет перемещаться, масштабироваться и поворачиваться, установите в его настройках флажок Static (Статический) справа вверху в инспекторе. Это позволит движку задействовать множество внутренних оптимизаций.
- Уменьшите число источников света в сцене. Чем больше источников света, тем больше работы придется выполнить движку.
- Использование карт освещенности вместо вычисления параметров освещения в реальном времени намного эффективнее. Но имейте в виду, что карты освещенности не поддерживают движущиеся объекты и для их хранения требуется дополнительная память.
- При любой возможности используйте сжатые текстуры. Сжатые текстуры занимают меньше памяти, и движку требуется меньше времени для доступа к ним (так как приходится читать меньше данных).

Множество других полезных советов вы найдете в руководстве к Unity (*https://docs.unity3d.com/ru/current/Manual/OptimizingGraphicsPerformance.html*).

В заключение

Управляя освещением, можно значительно улучшить вид сцены. Даже если игра не предусматривает максимальную реалистичность, ей можно придать особую привлекательность, приложив немного усилий к настройке освещения в сцене.

Также важно постоянно следить за производительностью игры. С помощью профайлера можно во всех деталях рассмотреть происходящее в игре и скорректировать ее, используя эту информацию.

15 Создание графических интерфейсов пользователя в Unity

Игры — это программы, а все программы нуждаются в пользовательском интерфейсе. Игра должна иметь самые обычные, «неигровые» визуальные компоненты для взаимодействия с пользователем, пусть и такие простые, как кнопка, запускающая игру, или надпись, отображающая количество набранных очков.

К счастью, Unity является превосходной основой для реализации пользовательских интерфейсов. В версии Unity 4.6 появилась необычайно гибкая и мощная система пользовательского интерфейса, спроектированная для часто встречающихся в играх случаев. Например, система пользовательского интерфейса способна работать на персональном компьютере, в консоли и на мобильных платформах; позволяет масштабировать единый пользовательский интерфейс под экраны разных размеров; реагирует на ввод с клавиатуры, мыши, сенсорного экрана и игровых контроллеров; поддерживает отображение пользовательского интерфейса в обоих пространствах, видимом и мировом.

Проще говоря, это набор инструментов с впечатляющими возможностями. Даже при том, что мы уже демонстрировали некоторые приемы создания пользовательских интерфейсов в частях II и III, нам хотелось бы дополнительно рассказать о некоторых замечательных особенностях системы графического пользовательского интерфейса, чтобы подготовить вас к использованию всех ее преимуществ.

Как действует пользовательский интерфейс в Unity

По сути, графический пользовательский интерфейс в Unity мало чем отличается от других видимых объектов в сцене. Графический интерфейс — это меш (mesh), конструируемый движком в процессе выполнения, с наложенной на него текстурой; кроме того, элементы графического интерфейса содержат сценарии, вызываемые в ответ на перемещение указателя мыши, события от клавиатуры и касания, для обновления и изменения этого меша. Меши отображаются с помощью камеры.

Система графического пользовательского интерфейса в Unity состоит из нескольких элементов, взаимодействующих друг с другом. Основу графического

пользовательского интерфейса составляют несколько объектов с компонентами `RectTransform`, которые рисуют свое содержимое и откликаются на *события*. Все они находятся внутри объекта Canvas.

Canvas

Canvas — это объект, отвечающий за отображение всех элементов пользовательского интерфейса на экране. Он также служит единым пространством, в котором отображается холст.

Все элементы пользовательского интерфейса являются дочерними объектами по отношению к Canvas: если кнопку не сделать дочерним объектом холста, она не появится на экране.

Объект Canvas позволяет выбирать, как отображать элементы пользовательского интерфейса. Кроме того, подключив компонент Canvas Scaler, можно управлять масштабированием элементов пользовательского интерфейса. Подробнее о масштабировании мы расскажем в разделе «Масштабирование холста» далее в этой главе.

Объект Canvas может использоваться в одном из трех режимов — Screen Space - Overlay (Пространство экрана — Перекрытие), Screen Space - Camera (Пространство экрана — Камера) и World Space (Пространство игрового мира).

- В режиме Screen Space - Overlay (Пространство экрана — Перекрытие) все содержимое объекта Canvas отображается поверх игровой сцены. То есть сначала на экран выводятся изображения, получаемые со всех камер в сцене, а затем поверх всего этого выводится содержимое холста. Этот режим используется по умолчанию.
- В режиме Screen Space - Camera (Пространство экрана — Камера) содержимое объекта Canvas отображается в плоскости, находящейся в трехмерном пространстве на некотором удалении от заданной камеры. Когда камера перемещается, плоскость холста также изменяет свое положение в пространстве, чтобы сохранить свое положение относительно камеры. В этом режиме холст фактически превращается в трехмерный объект, то есть объекты, находящиеся между камерой и холстом, будут перекрывать изображение холста.
- В режиме World Space (Пространство игрового мира) холст действует как трехмерный объект в сцене, имеет свои координаты и угол поворота, не зависящие ни от одной из камер в сцене. Это означает, например, что можно создать объект Canvas с кнопочной панелью открывания замка двери и разместить его рядом с дверью.

Если вам доводилось играть в такие игры, как *DOOM* (2016) или *Deus Ex: Human Revolution*, вы уже имеете опыт взаимодействия с пользовательскими интерфейсами в пространстве игрового мира. В этих играх игрок взаимодействует с компьютерами, приближаясь к ним и «нажимая» кнопки, отображаемые на их экранах.

RectTransform

Unity — это трехмерный игровой движок, все объекты которого обладают компонентом Transform, определяющим их координаты, поворот и масштаб в трехмерном пространстве. Однако пользовательский интерфейс в Unity имеет двумерную природу. То есть все элементы пользовательского интерфейса являются двумерными прямоугольниками, характеризующимися местоположением, шириной и высотой.

Управление элементами пользовательского интерфейса осуществляется посредством компонента `RectTransform`. `RectTransform` представляет собой прямоугольник, в котором отображается элемент пользовательского интерфейса. Важно отметить, что если объект с компонентом `RectTransform` является дочерним по отношению к *другому* объекту с компонентом `RectTransform`, его местоположение и размеры определяются относительно родителя.

Например, объект Canvas обладает компонентом `RectTransform`, который определяет как минимум размер пользовательского интерфейса; кроме того, все элементы пользовательского интерфейса игры имеют собственные компоненты `RectTransform`. А поскольку все эти элементы являются дочерними по отношению к объекту Canvas, их компоненты `RectTransform` определяют местоположение и размеры относительно холста.

Компонент `RectTransform` холста определяет также местоположение пользовательского интерфейса в зависимости от выбранного режима — Screen Space - Overlay (Пространство экрана — Перекрытие), Screen Space - Camera (Пространство экрана — Камера) или World Space (Пространство игрового мира). Во всех режимах, кроме World Space (Пространство игрового мира), позиция холста определяется автоматически.

Эту цепочку можно продолжить, добавляя вложенные элементы. Если создать объект с компонентом `RectTransform` и добавить в него дочерние объекты (с собственными компонентами `RectTransform`), позиции этих дочерних объектов будут определяться относительно позиции родителя.

Компоненты `RectTransform` можно подключать не только к элементам пользовательского интерфейса, но и к любым другим объектам; в этом случае `RectTransform` заменит компонент `Transform` в инспекторе.

Инструмент Rect

Инструмент Rect (Прямоугольник) дает возможность перемещать объекты с компонентом `RectTransform` и изменять их размеры. Для его активации нажмите клавишу `T` или щелкните по крайней правой пиктограмме в панели переключателя режимов, находящейся вверху слева в окне Unity (рис. 15.1).

Рис. 15.1. Выбор инструмента Rect (Прямоугольник)

Когда включен инструмент Rect (Прямоугольник), вокруг выбранного объекта появляется рамка с маркерами (рис. 15.2). Двигая эти маркеры мышью, можно изменять размеры объекта и перемещать его.

Кроме того, если приблизить указатель мыши к маркеру вне прямоугольника, изображение указателя изменится, показывая переход в режим поворота. Если в этот момент нажать левую кнопку мыши и, удерживая ее, потянуть указатель, объект повернется относительно своей опорной точки. Опорная точка изображается окружностью в центре объекта; если выбранный объект обладает компонентом `RectTransform`, вы сможете переместить его, нажав левую кнопку мыши на опорной точке и потянув за нее.

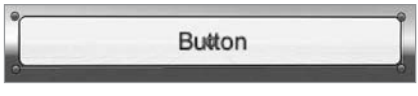

Рис. 15.2. Маркеры инструмента Rect (Прямоугольник) и опорная точка в центре

Инструмент Rect (Прямоугольник) не ограничивается элементами пользовательского интерфейса! Он способен также работать с трехмерными объектами; после выбора такого объекта прямоугольная рамка и маркеры разместятся в зависимости от положения объекта к направлению взгляда. На рис. 15.3 показано, как это выглядит.

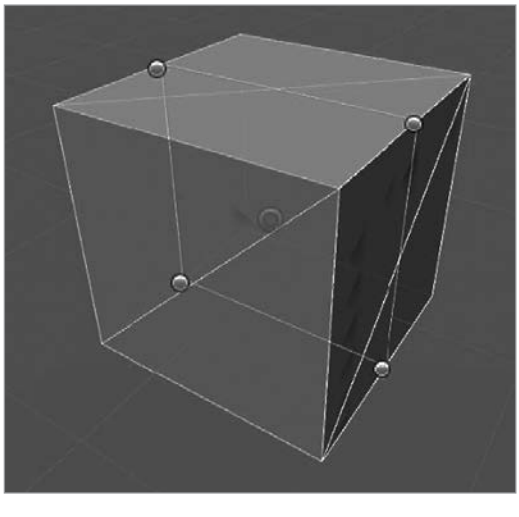

Рис. 15.3. Маркеры инструмента Rect (Прямоугольник), окружающие трехмерный куб

Привязки

Когда `RectTransform` является дочерним по отношению к другому `RectTransform`, он позиционируется с учетом своих *привязок* (anchors). Это позволяет определять отношения между размерами родительского и позицией и размерами дочернего прямоугольника. Например, дочерний объект можно привязать к нижнему краю родителя и растянуть на всю его ширину; когда размеры родителя изменятся, соответственно изменится позиция и размеры дочернего объекта.

В инспекторе, в разделе с настройками `RectTransform`, имеется пиктограмма, позволяющая выбирать предопределенные значения для привязок (рис. 15.4).

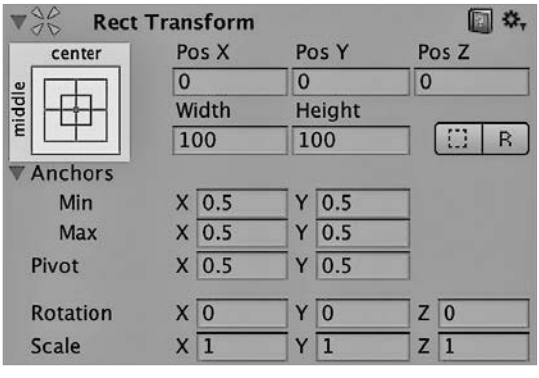

Рис. 15.4. Пиктограмма, отображающая текущие настройки привязок в компоненте RectTransform

Если щелкнуть по этой пиктограмме, откроется панель, в которой можно изменить привязки (рис. 15.5).

Выбор любых предопределенных настроек изменяет привязки в компоненте `RectTransform`. Позиция и размеры прямоугольника при этом не изменятся сразу же — они изменятся, когда изменятся размеры *родителя*.

Это очень наглядная особенность системы пользовательского интерфейса, поэтому лучший способ понять, как она работает, — поэкспериментировать с ней. Поместите игровой объект `Image` внутрь другого объекта `Image` и попробуйте менять параметры привязки дочернего объекта и изменять размеры родительского.

Элементы управления

Для использования в сценах доступно несколько элементов управления. От простых и распространенных, как, например, кнопки и поля ввода, до сложных, таких как прокручиваемые представления.

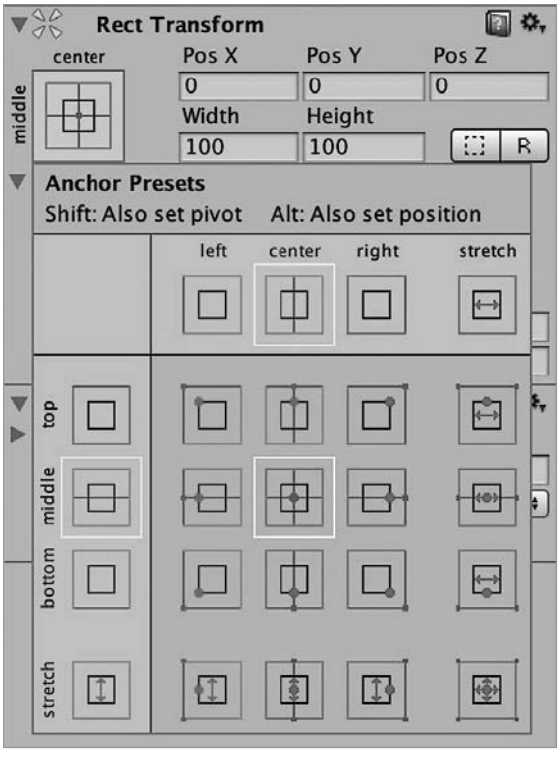

Рис. 15.5. Панель настройки привязок

В этом разделе мы поговорим о наиболее важных из них и расскажем, как ими пользоваться. Со временем постоянно добавляется поддержка новых элементов управления, поэтому их исчерпывающий список ищите в руководстве к Unity.

Элементы управления в системе поддержки пользовательского интерфейса Unity часто конструируются из нескольких игровых объектов, взаимодействующих друг с другом. Поэтому не удивляйтесь, если после добавления одного элемента управления в иерархии появится несколько объектов.

События и метод выпускания лучей

Когда пользователь касается кнопки на экране, он ожидает, что та выполнит некоторое действие, назначенное ей. Для этого система пользовательского интерфейса должна иметь возможность узнать, *какой* объект был использован.

Система, поддерживающая такую возможность, называется *системой событий*. Она имеет весьма сложное устройство: помимо поддержки ввода в графическом интерфейсе она также может использоваться как универсальное решение для определения таких событий в игре, как щелчок, касание или перетаскивание.

Система событий представлена объектом Event System, который появляется при создании объекта Canvas.

Действие системы событий основано на методе *выпускания лучей* (raycasts). Под *лучом* в данном случае понимается невидимая линия, отбрасываемая от точки касания к экрану. Этот луч продолжает распространяться вглубь сцены, пока не столкнется с поверхностью какого-нибудь объекта, благодаря чему система событий узнает, какой объект оказался «под» пальцем пользователя.

Метод выпускания лучей предназначен для использования в трехмерном пространстве, поэтому так же, как и вся остальная часть движка, система событий способна работать и с двумерными, и с трехмерными интерфейсами. Когда происходит событие, такое как касание пальцем или щелчок мышью, каждый *источник лучей* (raycaster) в сцене выпускает свой луч. Существует три типа источников, испускающих лучи: *графические источники лучей* (graphic raycasters), *двумерные физические источники* (2D physics raycasters) и *трехмерные физические источники лучей* (3D physics raycasters).

- Графические источники определяют пересечение их луча с любым компонентом Image в холсте.
- Двумерные физические источники определяют пересечение их луча с любым двумерным коллайдером в сцене.
- Трехмерные физические источники определяют пересечение их луча с любым трехмерным коллайдером в сцене.

Когда пользователь касается кнопки, компонент графического источника лучей, подключенный к объекту Canvas, испускает луч от точки контакта пальца с экраном и определяет, встретился ли на пути луча какой-нибудь компонент Image. Так как кнопки включают компонент Image, источник лучей сообщит системе событий о касании кнопки.

Двумерные и трехмерные физические источники лучей не используются в системе графического интерфейса, тем не менее их можно использовать для определения событий щелчков, касаний и перетаскивания в отношении двух- и трехмерных объектов в сцене. Например, с помощью трехмерного физического источника лучей можно определить, когда пользователь щелкнет по кубу.

Обработка событий

При создании своего пользовательского интерфейса часто бывает удобно иметь возможность добавлять нестандартное поведение в элементы интерфейса. Обычно под этим понимается обработка событий ввода, таких как щелчки и перетаскивания.

Чтобы сценарий мог откликаться на эти события, класс должен унаследовать определенные интерфейсы и реализовать их методы. Например, класс, наследующий интерфейс `IPointerClickHandler`, обязан реализовать метод с сигнатурой:

`public void OnPointerClick (PointerEventData event Data)`. Этот метод вызывается, когда система событий обнаруживает, что текущий указатель (указатель мыши или палец, коснувшийся экрана) выполнил «щелчок», то есть в пределах изображения была нажата и отпущена кнопка мыши или опущен и поднят палец.

Для демонстрации ниже приводится краткое руководство, как обработать щелчок по объекту пользовательского интерфейса.

1. *В пустой сцене создайте новый объект Canvas*, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Canvas (Пользовательский интерфейс ▸ Холст). В сцену будет добавлен объект Canvas.
2. *Создайте новый объект Image*, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Image (Пользовательский интерфейс ▸ Изображение). Объект Image будет добавлен как дочерний по отношению к Canvas.
3. *Добавьте в объект Image новый сценарий на C# с именем EventResponder.cs* и введите в файл следующий код:

```
// Необходимо для доступа к 'IPointerClickHandler' и
// 'PointerEventData'
using UnityEngine.EventSystems;

public class EventResponder : MonoBehaviour,
  IPointerClickHandler {

    public void OnPointerClick (PointerEventData eventData)
    {
        Debug.Log("Clicked!");
    }

}
```

4. *Запустите игру*. После щелчка по изображению в консоли должно появиться слово «Clicked!».

Использование системы компоновки интерфейса

Когда создается новый элемент пользовательского интерфейса, обычно он добавляется непосредственно в сцену, после чего вручную корректируются его позиция и размеры. Однако такой подход невозможно использовать в двух ситуациях:

- когда размер холста неизвестен заранее, потому что игра будет отображаться на экранах с разными размерами;
- когда предполагается добавлять и удалять элементы пользовательского интерфейса во время выполнения.

В этих случаях можно воспользоваться системой компоновки интерфейса, встроенной в Unity.

Чтобы показать, как она работает, создадим вертикальный список кнопок.

1. *Выберите объект Canvas*, щелкнув по нему в панели иерархии. (Если у вас такого объекта нет, создайте его, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Canvas (Пользовательский интерфейс ▸ Холст).)
2. *Создайте новый пустой дочерний объект*, открыв меню GameObject (Игровой объект) и выбрав пункт Create Empty Child (Создать пустой дочерний объект) или нажав комбинацию Ctrl-Alt-N (Command-Option-N на Mac).
3. *Дайте новому объекту имя List*.
4. *Создайте новый объект Button*, открыв меню GameObject (Игровой объект) и выбрав пункт UI ▸ Button (Пользовательский интерфейс ▸ Кнопка). Сделайте новую кнопку дочерним объектом по отношению к объекту List.
5. *Добавьте в объект List компонент Vertical Layout Group*, для чего выберите объект List, щелкните по кнопке Add Component (Добавить компонент) и в раскрывшемся меню выберите пункт Layout ▸ Vertical Layout Group (Компоновка ▸ Вертикальный макет). (Можно также ввести несколько первых букв из названия «vertical layout group», чтобы быстро выбрать этот объект.)

   В момент добавления компонента Vertical Layout Group в объект List вы должны заметить, что размер кнопки Button изменился и она заполнила все пространство в списке List. Состояние сцены до и после этого момента изображено на рис. 15.6 и 15.7.

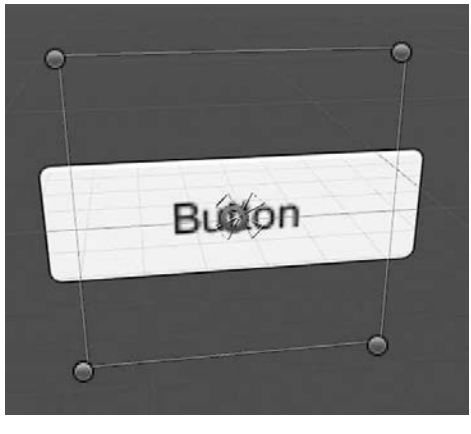

Рис. 15.6. Кнопка до добавления компонента Vertical Layout Group в список List

Теперь посмотрим, что получится, если включить в макет *несколько* кнопок.

6. *Выберите кнопку и скопируйте ее*, нажав комбинацию Ctrl-D (Command-D на Mac).

   После этого исходная кнопка и ее копия немедленно изменят свое местоположение и размеры так, что обе уложатся в объект List (рис. 15.8).

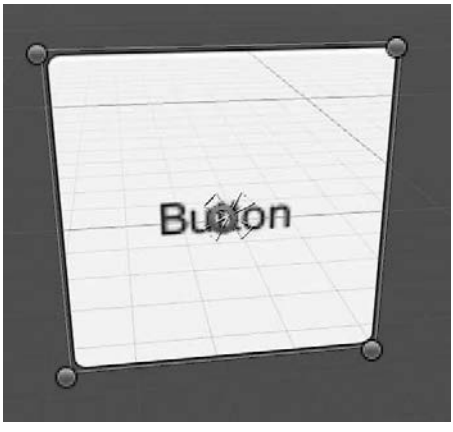

Рис. 15.7. Кнопка после добавления компонента Vertical Layout Group в список List

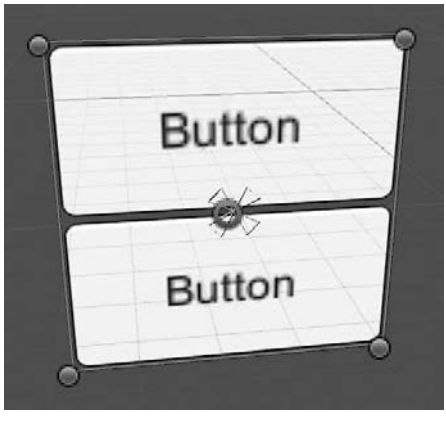

Рис. 15.8. Две кнопки, расположенные вертикально

Помимо вертикального макета Vertical Layout Group система пользовательского интерфейса поддерживает также горизонтальный макет Horizontal Layout Group, действующий точно так же, но размещающий дочерние элементы по горизонтали. Имеется также макет размещения по сетке Grid Layout Group, позволяющий размещать элементы в несколько рядов и выполняющий перенос элементов на следующий ряд, если потребуется.

Масштабирование холста

Кроме того что разные экраны, на которых будет отображаться игра, имеют разные размеры, скорее всего, они будут иметь и *разное разрешение*. Под разреше-

нием понимаются размеры отдельных пикселов; современные мобильные устройства обычно оснащаются экранами с большим разрешением.

Показательным примером могут служить экраны производства компании Retina, которыми оснащаются все модели iPhone, начиная с iPhone 4, и все iPad, начиная с третьего поколения. Эти устройства оснащаются экранами с теми же физическими размерами, что и предыдущая модель, но с удвоенным разрешением: экран iPhone 3GS имел ширину 320 пикселов, а iPhone 4 — уже 640. Содержимое, отображаемое на экране, должно сохранять свои физические размеры, но с увеличением разрешения появляется возможность добиваться более гладкого и привлекательного изображения.

Поскольку движок Unity работает с отдельными пикселами, на экранах с высоким разрешением графический пользовательский интерфейс будет отображаться в половину своего размера.

Для решения этой проблемы система графического пользовательского интерфейса в Unity включает компонент Canvas Scaler. Этот компонент предназначен для автоматического масштабирования всех элементов пользовательского интерфейса, чтобы придать им соответствующие размеры на экране, на котором в данный момент выполняется игра.

Когда вы создаете объект Canvas из меню GameObject (Игровой объект), в него автоматически добавляется компонент Canvas Scaler. Этот компонент поддерживает три режима работы: Constant Pixel Size (Постоянный размер пиксела), Scale With Screen Size (Масштабирование по экранным размерам) и Constant Physical Size (Постоянный физический размер).

Constant Pixel Size (Постоянный размер пиксела)

Действует по умолчанию. В этом режиме холст не масштабируется для экранов с разными размерами и разрешениями.

Scale With Screen Size (Масштабирование по экранным размерам)

В этом режиме обеспечивается масштабирование холста с учетом его размеров при «опорном разрешении», которое задается в инспекторе. Например, если задать опорное разрешение 640 × 480, а затем запустить игру на устройстве с разрешением 1280 × 960, то размеры каждого элемента пользовательского интерфейса увеличатся в 2 раза.

Constant Physical Size (Постоянный физический размер)

В этом режиме холст масштабируется с учетом разрешающей способности экрана (DPI, dots per inch — точек на дюйм) устройства, на котором выполняется игра, если данная информация доступна.

По своему опыту мы считаем режим Scale With Screen Size (Масштабирование по экранным размерам) наиболее подходящим для большинства ситуаций.

Переходы между экранами

Большинство пользовательских интерфейсов в играх делится на два типа: меню и интерфейс в игре. Меню — это интерфейс, с которым взаимодействует игрок, готовясь начать игру, — то есть выбирает между началом новой игры или продолжением предыдущей, настраивает параметры или просматривает список участников перед присоединением к многопользовательской игре. Интерфейс в игре накладывается сверху на игровой мир.

Интерфейс в игре обычно редко изменяет свою структуру и часто служит для отображения важной информации: количество стрел в колчане игрока, уровень здоровья и расстояние до следующей цели. Меню, напротив, меняются очень часто; главное меню обычно сильно отличается от экрана с настройками, потому что к ним предъявляются разные требования.

Поскольку пользовательский интерфейс — это всего лишь объект, отображаемый камерой, в Unity в действительности отсутствует понятие «экрана» как совокупности отображаемых элементов. Есть лишь текущая коллекция объектов на холсте. Если вам понадобится сменить один экран на другой, вы должны будете или изменить содержимое холста, который отображает камера, или направить камеру на что-то другое.

Прием изменения содержимого холста хорошо подходит для случаев, когда требуется изменить ограниченное подмножество элементов интерфейса. Например, если желательно оставить декоративное оформление, но заменить некоторые элементы пользовательского интерфейса, тогда имеет смысл не трогать камеру, а выполнить необходимые изменения в холсте. Но если требуется произвести полную замену элементов интерфейса, изменение направления визирования камеры может оказаться более эффективным решением.

При этом важно помнить, что для независимого изменения положения камеры в режиме работы холста требуется выбрать режим World Space (Пространство игрового мира); в двух других режимах — Screen Space - Overlay (Пространство экрана — Перекрытие), Screen Space - Camera (Пространство экрана — Камера) — пользовательский интерфейс всегда находится прямо перед камерой.

В заключение

Как видите, система графического пользовательского интерфейса в Unity обладает обширными возможностями. Вы можете использовать ее разными способами и в разных контекстах; кроме того, ее гибкая конструкция позволяет создавать пользовательские интерфейсы, точно соответствующие вашим потребностям.

Важно помнить, что пользовательский интерфейс является одним из важнейших компонентов игры. Он дает возможность взаимодействовать с игрой и на мобильных устройствах является основным средством управления игрой. Будьте готовы потратить много времени на доводку и усовершенствование пользовательского интерфейса.

16 Расширения редактора

При создании игр в Unity приходится работать с большим количеством игровых объектов и управлять всеми их компонентами. Инспектор уже избавляет от множества хлопот, автоматически отображая все переменные в сценариях в виде простых в использовании текстовых полей ввода, флажков и слотов для сброса ресурсов и объектов из сцены, помогая намного быстрее конструировать сцены.

Но иногда возможностей инспектора оказывается недостаточно. Unity разрабатывался так, чтобы максимально упростить создание двух- и трехмерных окружений, но разработчики Unity не в состоянии предвидеть все, что может вам потребоваться в играх.

Пользовательские расширения позволят вам получить контроль над самим редактором. Это могут быть очень маленькие вспомогательные окна, автоматизирующие типичные задачи в редакторе, или даже полностью переделанный инспектор.

Создавая игры более сложные, чем показанные в этой книге, мы обнаружили, что порой можно сэкономить массу времени, написав свои инструменты для автоматизации повторяющихся задач. Это не значит, что ваша главная задача как разработчика игр должна заключаться в создании программного обеспечения, помогающего конструировать игры, — ваша главная задача заключается в создании самих игр! Но если вы обнаружите, что какое-то действие вызывает большие сложности или его приходится повторять снова и снова, то подумайте о создании расширения, выполняющего эту работу за вас.

Эта глава ведет вас за кулисы Unity. Фактически мы будем использовать классы и код, которые использует сам редактор Unity. Как следствие, код может показаться вам немного сложнее, чем тот, что мы писали в предыдущих главах.

Существует несколько способов расширения Unity. В этой главе мы рассмотрим четыре из них, каждый из которых будет немного сложнее и производительнее предыдущего.

- Собственные мастеры, дающие возможность задать параметры и выполнить некоторое действие в сцене, например создать сложный составной объект.
- Собственные окна редактора, позволяющие создавать свои окна и вкладки, содержащие любые необходимые вам элементы управления.

- Собственные редакторы свойств (property drawers), позволяющие создавать нестандартные элементы пользовательского интерфейса в инспекторе для данных ваших собственных типов.
- Собственные редакторы, позволяющие полностью переопределить интерфейс инспектора для определенных объектов.

Чтобы приступить к работе с примерами в этой главе, создадим новый проект.

1. *Создайте новый проект* с названием «Editor Extensions». Выберите его тип 3D и сохраните в любой папке по своему выбору (рис. 16.1).

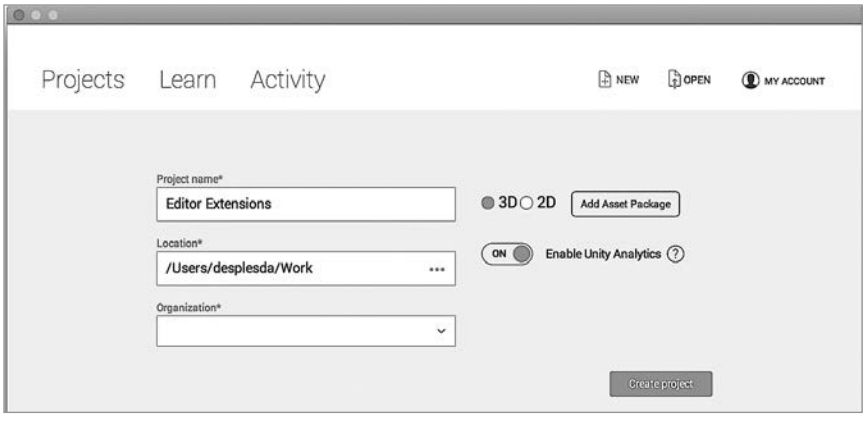

Рис. 16.1. Создание нового проекта

2. *Когда Unity загрузится, создайте новую папку внутри папки Assets.* Дайте новой папке имя *Editor*. В ней мы будем хранить сценарии для наших расширений.

Обратите внимание: папка обязательно должна иметь имя *Editor*, именно в таком написании, с первой заглавной буквой. Unity будет искать папку именно с таким именем.

Фактически эта папка может находиться где угодно — совсем необязательно помещать ее в папку *Assets*, главное, чтобы она называлась *Editor*. Это очень удобно, потому что в больших проектах может быть несколько папок *Editor*, что значительно упрощает работу с большим количеством сценариев.

Теперь вы готовы приступить к созданию сценариев своих собственных редакторов!

Создание своего мастера

Для начала создадим свой мастер. Мастер — это самый простой способ показать окно, в котором можно получить ввод пользователя и на его основе создать что-

нибудь в сцене. Типичным примером может служить создание объекта, зависящего от параметров, введенных пользователем.

Мастеры и окна редактора, о которых рассказывается в разделе «Создание собственного окна редактора» далее в этой главе, концептуально похожи: оба отображают окно с элементами управления. Но они отличаются способом их использования: элементы управления мастера создаются и управляются редактором Unity, тогда как управление элементами в окне редактора полностью возлагается на вас. Мастеры прекрасно подходят в ситуациях, когда не требуется какой-то специализированный пользовательский интерфейс, тогда как окна редактора лучше подходят для случаев, где требуется управлять отображаемым содержимым.

Лучший способ понять, чем мастеры могут помочь в повседневной практике использования Unity, — создать один такой. Мы реализуем мастер, который создает в сцене игровые объекты в форме тетраэдра — треугольной пирамидки, как показано на рис. 16.2.

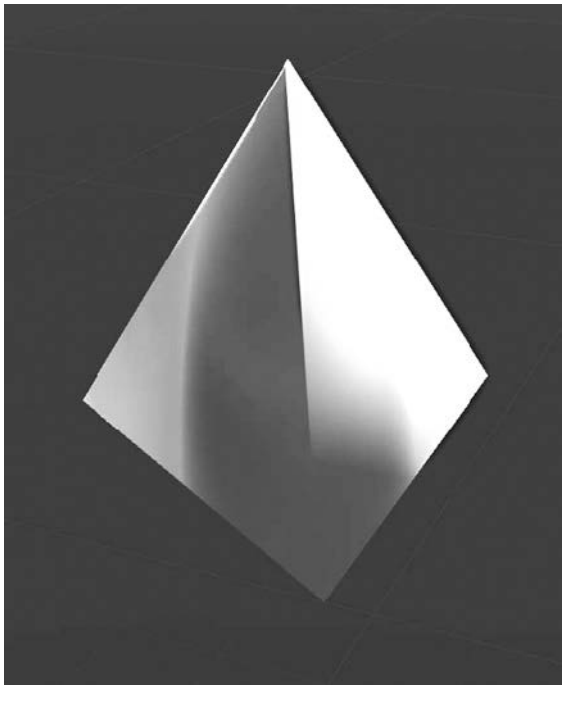

Рис. 16.2. Тетраэдр, созданный мастером

Чтобы получить подобный объект, требуется вручную создать объект `Mesh`. Обычно такие объекты импортируются из файлов, например, из файла *.blend*, как было показано в главе 9; однако их можно также создавать программно.

С помощью `Mesh` можно создать объект, отображающий меш. Для этого сначала нужно создать новый игровой объект `GameObject`, а затем подключить к нему два компонента: `MeshRenderer` и `MeshFilter`. После этого объект будет готов для использования в сцене.

Эти шаги легко автоматизировать, а это значит, что данная процедура прекрасно подходит для мастера.

1. *Создайте в папке Editor новый сценарий на C# с именем Tetrahedron.cs* и добавьте в него следующий код:

```
using UnityEditor;

public class Tetrahedron : ScriptableWizard {

}
```

   Класс `ScriptableWizard` определяет базовое поведение мастера. Мы реализуем несколько методов, переопределяющих это поведение, и получим интересный опыт.

   Сначала реализуем метод, отображающий мастер. Для этого требуется решить две задачи: добавить новый пункт в меню Unity, который можно использовать для вызова метода, а внутри этого метода обратиться к Unity, чтобы отобразить мастер.

2. *Добавьте следующий код в класс Tetrahedron*:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

    // Этому методу можно дать любое имя, главное, чтобы
    // он был объявлен статическим и имел атрибут MenuItem
    [MenuItem("GameObject/3D Object/Tetrahedron")]
    static void ShowWizard() {
      // Первый параметр - текст в заголовке окна, второй - надпись
      // на кнопке создания
      ScriptableWizard.DisplayWizard<Tetrahedron>(
        "Create Tetrahedron", "Create");
}
```

   Когда статический метод снабжается атрибутом `MenuItem`, Unity добавляет в меню приложения соответствующий пункт. В данном случае он создаст новый пункт Tetrahedron в меню GameObject ▸ 3D Object; когда пользователь выберет этот пункт, Unity вызовет метод `ShowWizard`.

В действительности необязательно было давать методу имя `ShowWizard`. С таким же успехом можно дать любое другое имя — Unity принимает во внимание только атрибут `MenuItem`.

3. *Вернитесь в Unity* и откройте меню GameObject (Игровой объект). Выберите пункт 3D Object ▸ Tetrahedron, и на экране появится пустое окно мастера (рис. 16.3).

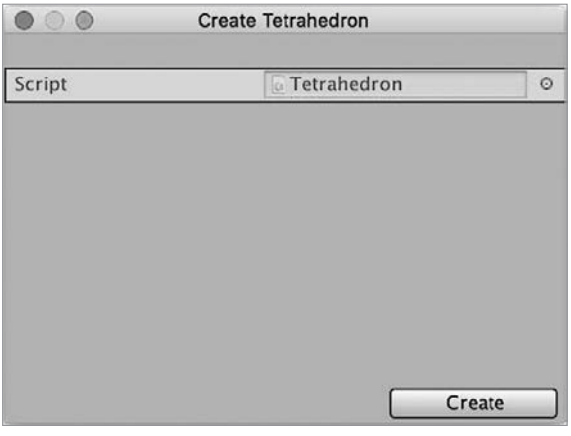

Рис. 16.3. Пустое окно мастера

Далее добавим переменную в класс мастера. После этого Unity отобразит соответствующий элемент управления в окне мастера, как это делает инспектор. Эта переменная будет иметь тип `Vector3` и представлять высоту, ширину и глубину объекта.

4. *Добавьте в класс Tetrahedron следующую переменную, представляющую размеры тетраэдра*:

```
// Эта переменная будет выглядеть в окне мастера
// так же, как в инспекторе
public Vector3 size = new Vector3(1,1,1);
```

5. *Вернитесь в Unity.* Закройте и вновь откройте мастер, и вы увидите поле, соответствующее переменной `Size` (рис. 16.4).

Теперь мастер позволяет вводить данные, но пока никак их не обрабатывает. Давайте решим эту задачу прямо сейчас!

Вызывая метод `DisplayWizard`, мы передали ему две строки. Первая — для отображения в заголовке окна, а вторая — для отображения на кнопке Create (Создать). Когда пользователь щелкнет по этой кнопке, Unity вызовет метод `OnWizardCreate` класса мастера, сообщая тем самым, что данные готовы к использованию; когда `OnWizardCreate` вернет управление, Unity закроет окно.

Теперь реализуем метод `OnWizardCreate`, который выполнит основную задачу мастера. Используя переменную `Size`, он создаст объект `Mesh` и сконструирует игровой объект, отображающий этот меш.

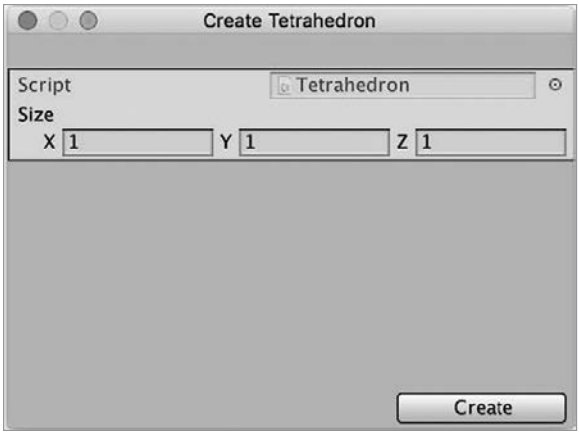

Рис. 16.4. Мастер с элементами управления содержимым переменной Size

6. *Добавьте следующий метод в класс Tetrahedron*:

```
// Вызывается, когда пользователь щелкает по кнопке Create
void OnWizardCreate() {

  // Создать меш
  var mesh = new Mesh();

  // Создать четыре точки
  Vector3 p0 = new Vector3(0,0,0);
  Vector3 p1 = new Vector3(1,0,0);
  Vector3 p2 = new Vector3(0.5f,
                           0,
                           Mathf.Sqrt(0.75f));
  Vector3 p3 = new Vector3(0.5f,
                           Mathf.Sqrt(0.75f),
                           Mathf.Sqrt(0.75f)/3);

  // Масштабировать в соответствии с размером
  p0.Scale(size);
  p1.Scale(size);
  p2.Scale(size);
  p3.Scale(size);

  // Передать список вершин
  mesh.vertices = new Vector3[] {p0,p1,p2,p3};

  // Передать список треугольников, связанных вершинами
  mesh.triangles = new int[] {
    0,1,2,
    0,2,3,
    2,1,3,
    0,3,1
```

```
    };

    // Обновить некоторые дополнительные данные в меше,
    // используя эти данные
    mesh.RecalculateNormals();
    mesh.RecalculateBounds();

    // Создать игровой объект, использующий меш
    var gameObject = new GameObject("Tetrahedron");
    var meshFilter = gameObject.AddComponent<MeshFilter>();
    meshFilter.mesh = mesh;

    var meshRenderer
      = gameObject.AddComponent<MeshRenderer>();
    meshRenderer.material
      = new Material(Shader.Find("Standard"));

}
```

Этот метод сначала создает новый объект `Mesh`, затем вычисляет координаты четырех вершин тетраэдра. После этого они масштабируются в соответствии с вектором `size`, то есть располагаются так, что оказываются в нужных координатах, чтобы образовать тетраэдр с шириной, высотой и глубиной, определяемыми вектором `size`.

Эти точки передаются в объект `Mesh` через его свойство `vertices`; после этого через свойство `triangles` передается список треугольников в виде списка целых чисел. Каждое число представляет одну из точек в `vertices`.

Например, число `0` в списке треугольников соответствует первой точке, число `1` — второй и т. д. Треугольники в списке определяются как группы по три числа; то есть, например, числа `0, 1, 2` означают, что меш будет содержать треугольник, образованный первой, второй и третьей вершинами из списка `vertices`. Тетраэдр состоит из четырех треугольников: основания и трех сторон. Соответственно, список `triangles` включает четыре группы по три числа.

Наконец, мешу дается команда пересчитать некоторую внутреннюю информацию, опираясь на данные в свойствах `vertices` и `triangles`. После этого его можно использовать в сцене. Затем создается новый объект `GameObject`, к нему подключается объект `MeshFilter`, к которому в свою очередь подключается только что созданный объект `Mesh`. Далее к игровому объекту подключается визуализатор `MeshRenderer`, который будет отображать `Mesh`. В итоге визуализатору `MeshRenderer` передается новый материал `Material`, который создается с использованием стандартного шейдера `Standard` — в точности как при создании любого другого встроенного объекта, созданного через меню GameObject (Игровой объект).

7. *Вернитесь в Unity, закройте и вновь откройте окно мастера.* Если теперь щелкнуть по кнопке Create (Создать), в сцене появится новый тетраэдр. Если вы измените переменную `Size`, мастер создаст тетраэдр с другими размерами.

Нам осталось добавить в мастер еще одну особенность. В настоящее время мастер не проверяет, насколько правдоподобные значения ввел пользователь в переменную `Size`; например, мастер должен отвергать попытку создать тетраэдр с *отрицательной* высотой.

Строго говоря, в этом нет необходимости, потому что Unity способен справиться с этой ситуацией. Однако вам будет полезно знать, как выполнить проверку ввода.

В данном примере мы реализуем мастер так, что он будет отвергать попытку создать тетраэдр, если какой-то элемент вектора `Size` — `X`, `Y` или `Z` — имеет значение меньше или равное нулю.

Для этого мы реализуем метод `OnWizardUpdate`, который вызывается всякий раз, когда пользователь изменяет любую переменную в окне мастера. Это дает нам шанс проверить значения и разрешить или запретить кнопку Create (Создать). Что особенно важно, есть возможность добавить текст, объясняющий, *почему* мастер отверг ввод.

8. *Добавьте следующий метод в класс Tetrahedron*:

```
// Вызывается, когда пользователь изменяет какое-то значение в мастере
void OnWizardUpdate() {

  // Проверить допустимость введенных значений
  if (this.size.x <= 0 ||
    this.size.y <= 0 ||
    this.size.z <= 0) {

    // Когда isValid получает значение true, разрешается
    // щелкнуть по кнопке Create
    this.isValid = false;

    // Объяснить причину
    this.errorString
      = "Size cannot be less than zero";

  } else {

    // Пользователь может щелкнуть по кнопке Create, поэтому
    // разрешим ему это и очистим сообщение об ошибке
    this.errorString = null;
    this.isValid = true;
  }
}
```

Когда свойство `isValid` получит значение `false`, кнопка Create (Создать) окажется запрещена, то есть пользователь не сможет щелкнуть по ней. Кроме того, если присвоить свойству `errorString` любое строковое значение, кроме `null`, текст из этой строки появится в окне. Вы можете использовать это обстоятельство, чтобы объяснить пользователю, в чем заключается проблема (рис. 16.5).

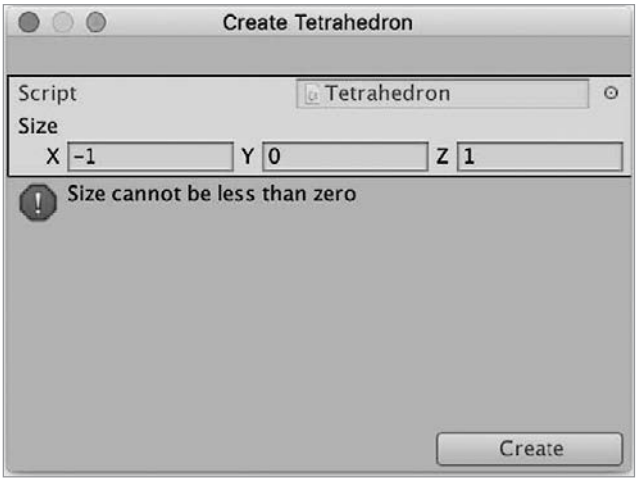

Рис. 16.5. Сообщение об ошибке в окне мастера

Мастеры позволяют сэкономить массу времени при выполнении повторяющихся или сложных задач в редакторе Unity. Они быстро реализуются в коде, потому что создание необходимого пользовательского интерфейса берет на себя редактор Unity. Но иногда требуется нечто большее, чем может дать система поддержки мастеров; далее мы посмотрим, как создать полностью свое окно редактора.

Создание собственного окна редактора

Окном в Unity называется область, которая может быть отдельным плавающим окном или вкладкой, пристыкованной к главному интерфейсу редактора Unity.

Почти все элементы интерфейса Unity являются окнами редактора.

Создавая окно редактора, вы получаете полный контроль над его содержимым. В этом заключается важное отличие от мастеров и инспектора, в которых элементы пользовательского интерфейса создаются автоматически самим редактором Unity; в окне редактора ничего не появится без явного требования с вашей стороны. Это открывает перед вами широкие возможности и дает вам в руки инструмент, позволяющий добавлять в Unity полностью новые особенности, соответствующие вашим потребностям.

В данном разделе мы создадим окно редактора, которое просто подсчитывает количество текстур в проекте. Но прежде чем перейти к главному функционалу, нам нужно научиться выводить *какую-либо* информацию в окно редактора.

Сначала создадим новое пустое окно редактора.

1. *Создайте новый сценарий с именем TextureCounter.cs* и сохраните его в папку *Editor*.
2. *Откройте файл* и замените его содержимое следующим кодом:

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class TextureCounter : EditorWindow {

  [MenuItem("Window/Texture Counter")]
  public static void Init() {
    var window = EditorWindow
      .GetWindow<TextureCounter>("Texture Counter");
    // Запретить уничтожение этого окна
    // при загрузке новой сцены
    DontDestroyOnLoad(window);
  }

  private void OnGUI() {
    // Здесь формируется пользовательский интерфейс редактора
    EditorGUILayout.LabelField("Current selected size is "
      + sizes[selectedSizeIndex]);
  }

}
```

   Этот код добавляет новый пункт в меню Window (Окно), щелчок по которому создает и отображает новое окно, использующее класс `TextureCounter`. Он также устанавливает признак, что окно не должно закрываться редактором Unity при изменении текущей сцены.

3. *Сохраните файл* и вернитесь в Unity.
4. *Откройте меню Window (Окно)*, и вы увидите пункт Texture Counter. Щелкните по нему, и на экране появится новое пустое окно!

Теперь добавим в это окно элементы управления. Но для этого нужно знать, как работает система пользовательского интерфейса редактора Unity.

Программный интерфейс редактора Unity

Система пользовательского интерфейса, используемая редактором Unity, в корне отличается от аналогичной системы, используемой в играх.

Система игрового пользовательского интерфейса (назовем ее *Unity GUI*) создает игровые объекты, представляющие такие элементы, как текстовые надписи и кнопки, и помещает их в сцену.

Система пользовательского интерфейса редактора (назовем ее *GUI немедленного режима*; причину выбора такого названия мы объясним чуть ниже) предлагает специальные функции, которые создают надписи или кнопки в определенных

местах; эти функции вызываются редактором Unity всякий раз, когда требуется перерисовать содержимое экрана.

Термин *немедленный режим* (immediate mode) основывается на том факте, что вызов этих специальных функций приводит к немедленному появлению элементов управления на экране; затем спустя какое-то время, когда экран очищается и удаляются все элементы управления, в следующем же кадре функции GUI немедленного режима вызываются вновь. Этот процесс повторяется бесконечно.

Из соображений эффективности Unity не вызывает функции пользовательского интерфейса редактора непрерывно. Очистка экрана и вызов функций происходят, только когда это действительно необходимо, например когда пользователь щелкнет кнопкой мыши, нажмет клавишу, изменит размер окна или выполнит другое действие, влияющее на содержимое экрана.

Другим важным отличием систем немедленного режима и Unity GUI является порядок работы механизма размещения элементов. В Unity GUI объекты позиционируются относительно своих родителей, с учетом настроек привязки; в GUI немедленного режима вы либо определяете прямоугольную область, описывающую местоположение и размеры элемента управления, либо используете систему автоматического размещения, которая называется `GUILayout` (мы опишем ее далее).

Отличия проще всего объяснить на примере. На следующих нескольких страницах мы исследуем основы использования системы GUI немедленного режима, а также познакомимся с доступным набором элементов управления.

Прямоугольники и система автоматического размещения

Простейшим элементом управления, который можно добавить в окно, является текстовая надпись. Далее мы добавим код, решающий некоторую задачу, а затем объясним, как он действует.

1. *Добавьте следующий код в метод OnGUI*:

```
GUI.Label(                          ❶
    new Rect(50,50,100,20),         ❷
    "This is a label!"              ❸
);
```

2. *Вернитесь в Unity и откройте наше новое окно редактора*. Теперь вы увидите в окне текст «This is a label» (рис. 16.6).

А теперь пройдемся по этому коду.

❶ Вызов метода `Label` класса `GUI` выводит текст в окно.

❷ Создает новый объект `Rect`, определяющий координаты x и y, ширину и высоту надписи. В данном случае надпись помещается на 50 пикселов ниже и правее верхнего и левого края, и для нее отводится пространство шириной 100 и высотой 20 пикселов.

❸ Фактический текст, который должен появиться на экране: «This is a label!».

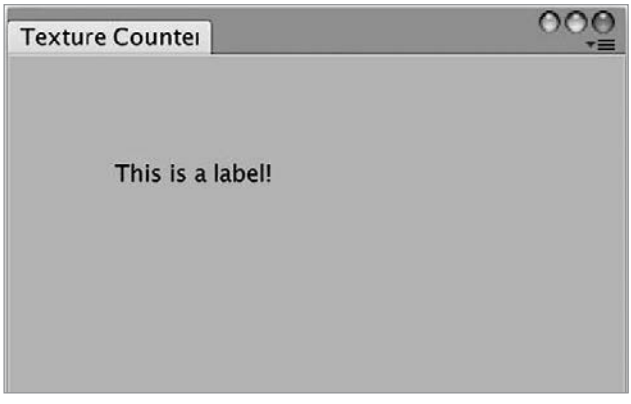

Рис. 16.6. Надпись, добавленная в окно редактора

Этот код будет выполняться всякий раз, когда редактору Unity потребуется обновить содержимое окна. Вызов метода `GUI.Label` добавляет надпись в окно.

Все вызовы функций GUI немедленного режима должны производиться только внутри метода `OnGUI`. Вызов `GUI.Label` из любого другого места может привести к проблемам.

Объект `Rect`, переданный в вызов `GUI.Label`, управляет местоположением надписи. В простых ситуациях, как в данном примере, такой способ позиционирования не вызывает затруднений, но в более сложных случаях он может стать источником затруднений.

Чтобы избежать их, GUI немедленного режима предоставляет возможность автоматического размещения элементов управления по вертикали или по горизонтали.

Например, чтобы разместить элементы управления в виде вертикального списка, нужно создать новый объект `EditorGUILayout.VerticalScope`, заключив вызов конструктора в конструкцию `using`.

3. *Замените содержимое метода OnGUI следующим кодом*:

```
using (var verticalArea
  = new EditorGUILayout.VerticalScope()) {
          GUILayout.Label("These");
          GUILayout.Label("Labels");
          GUILayout.Label("Will be shown");
          GUILayout.Label("On top of each other");
}
```

Этот пример имеет два важных отличия.

Во-первых, обратите внимание, что теперь вызывается метод `Label` класса `GUILayout`, а не `GUI`. Эта версия определяет факт вызова в контексте `VerticalScope` и размещает создаваемые надписи соответственно этому контексту.

Во-вторых, больше не требуется передавать объект `Rect`, определяющий местоположение и размеры надписей. Эти параметры определяются из контекста `VerticalScope`.

Система автоматического размещения намного удобнее в использовании и повышает вашу производительность как программиста. Поэтому в оставшейся части главы мы почти везде будем использовать систему автоматического размещения.

Единственное исключение — редакторы свойств, в которых система автоматического размещения не работает. В этом разделе мы продолжим вручную размещать элементы управления и определять их размеры.

Как действуют элементы управления

Как упоминалось выше, элементы управления в системе GUI немедленного режима создаются вызовами функций. Это обстоятельство не вызывает сложностей в случае таких простых элементов, как надписи; ситуация осложняется, когда требуется создать элемент, принимающий ввод пользователя, такой как кнопка или поле ввода.

Как с помощью таких элементов получать ввод пользователя, если они являются результатом вызовов функций? Ответ остроумен: функции, отображающие элементы управления, также *возвращают* информацию в вызывающий код.

И снова эту идею проще объяснить на примере.

Кнопки

Сначала создадим кнопку с помощью системы GUI немедленного режима.

1. *Замените содержимое метода OnGUI следующим кодом*:

```
private void OnGUI() {
  using (var verticalArea
    = new EditorGUILayout.VerticalScope()) {
      var buttonClicked = GUILayout.Button("Click me!");
      if (buttonClicked) {
        Debug.Log("The custom window's " +
                  "button was clicked!");
      }
  }
}
```

Когда вызывается метод `GUILayout.Button`, происходят два важных события. На экране появляется кнопка; а кроме того, если в этой области только что был выполнен щелчок мышью, метод вернет `true`.

Такое возможно благодаря повторяющимся вызовам `OnGUI`. Когда окно первый раз появится на экране, вызов `Button` отобразит кнопку на экране. Когда пользователь наведет указатель мыши на кнопку и щелкнет по ней, `OnGUI` будет вызван еще раз и система GUI немедленного режима нарисует кнопку в «нажатом» состоянии. Когда пользователь отпустит кнопку мыши, `OnGUI` будет вызван снова; поскольку щелчок завершился, третий вызов `Button` вернет `true`.

Этот стиль программирования можно интерпретировать так: метод `GUILayout.Button` одновременно рисует кнопку на экране и возвращает `true`, если пользователь щелкнул по ней.

2. *Вернитесь в Unity* и обратите внимание, что теперь в окне появилась кнопка. Если щелкнуть по ней, во вкладке Console (Консоль) появится текст «The custom window's button was clicked!».

Да, немного непривычно. Но ¯_(ツ)_/¯.

Текстовые поля ввода

Кнопка — один из простейших элементов управления, с помощью которых пользователь может передать информацию: либо щелкнуть по кнопке, либо нет. Однако система GUI немедленного режима поддерживает более сложные типы элементов управления. Например, текстовое поле ввода решает две задачи: отображает текст *и* позволяет пользователю править этот текст.

Чтобы показать текстовое поле ввода, нужно вызвать метод `EditorGUILayout.TextField`. Вызывая этот метод, вы должны передать ему строку для отображения в текстовом поле; и от него же можно получить текст, *введенный* пользователем.

Для нормальной работы элемента переменная, хранящая текст, не должна быть локальной. То есть следующий код работает *неправильно*:

```
private void OnGUI() {
  using (var verticalArea
    = new EditorGUILayout.VerticalScope()) {
            string textValue = "";

            textValue
              = EditorGUILayout.TextField(textValue);
  }
}
```

Метод `TextField` находится в классе `EditorGUILayout`, а не в `GUILayout`. Класс `GUILayout` также имеет метод `TextField`, но он действует совершенно иначе.

Если протестировать этот код в Unity, можно заметить, что он позволяет вводить символы в поле, но стоит оставить поле, как оно тут же очистится.

Чтобы этого не происходило, переменная, хранящая текст, должна принадлежать классу:

```
private string stringValue;
private void OnGUI() {
  using (var verticalArea
    = new EditorGUILayout.VerticalScope()) {
```

```
      this.stringValue
        = EditorGUILayout.TextField(this.stringValue);
    }
  }
```

Это решение работает, потому что содержимое `stringValue` сохраняется между вызовами `OnGUI`.

Элемент управления `TextField` отображает однострочный текст. Если вам понадобится отобразить текст, состоящий из нескольких строк, используйте `TextArea`:

```
this.stringValue = EditorGUILayout.TextArea(
  this.stringValue,
  GUILayout.Height(80)
);
```

Эти два элемента управления используют одну и ту же переменную, поэтому они будут отображать один и тот же текст. Более того, если изменить текст в одном, он автоматически изменится в другом. И это круто!

В предыдущем примере высота текстовой области была переопределена параметром `GUILayout`. Его можно добавить в *любой* элемент управления; если вам нужна кнопка повыше, просто добавьте вызов `GUILayout.Height(80)`, и вы получите кнопку высотой 80 пикселов.

Текстовые поля ввода с задержкой. Существует еще один тип текстового поля — *текстовое поле ввода с задержкой*. Оно действует как обычное поле ввода, но возвращает вводимую строку не сразу, не в процессе ввода, а только после потери фокуса ввода, то есть когда пользователь переместится в другое поле ввода или щелкнет мышью где-то еще.

Это удобно, когда требуется проверить допустимость введенной информации, но бессмысленно делать это до того, как пользователь завершит ввод.

Создается текстовое поле ввода с задержкой вызовом метода `DelayedTextField`, как показано ниже:

```
this.stringValue
  = EditorGUILayout.DelayedTextField(this.stringValue);
```

Специальные текстовые поля ввода. Кроме простого текста, поля ввода можно также использовать для ввода чисел. В частности, существует четыре очень удобных варианта элемента управления `TextField`: для ввода целых и вещественных чисел, для ввода значений типа `Vector2D` и для ввода значений типа `Vector3D`.

Например, добавив в свой класс свойства:

```
private int intValue;

private float floatValue;

private Vector2 vector2DValue;

private Vector3 vector3DValue;
```

вы сможете создавать поля для ввода данных в них:

```
this.intValue
  = EditorGUILayout.IntField("Int", this.intValue);

this.floatValue
  = EditorGUILayout.FloatField("Float", this.floatValue);

this.vector2DValue
  = EditorGUILayout.Vector2Field("Vector 2D",
                                 this.vector2DValue);
this.vector3DValue
  = EditorGUILayout.Vector3Field("Vector 3D",
                                 this.vector3DValue);
```

Обратите внимание на первый строковый параметр в вызовах методов: он определяет надпись перед полем ввода.

Ползунки

Вводить числовые значения можно не только с помощью полей ввода, но и с помощью графического ползунка. Например, вот как можно использовать `IntSlider`:

```
var minIntValue = 0;
var maxIntValue = 10;
this.intValue
  = EditorGUILayout.IntSlider(this.intValue,
                              minIntValue,
                              maxIntValue);
```

Ползунки особенно удобны в сочетании с элементом управления `IntField` или `FloatField`, использующим ту же переменную. Ползунок позволяет быстро ввести приближенное значение, но если потребуется более конкретное значение, его можно ввести в поле ввода.

Кроме обычных ползунков, имеется также ползунок для выбора минимального и максимального значений диапазона. Например, добавив в класс две переменные для хранения минимального и максимального значений:

```
private float minFloatValue;
private float maxFloatValue;
```

вы сможете создать ползунок для выбора диапазона вызовом метода `MinMaxSlider`:

```
var minLimit = 0;
var maxLimit = 10;
EditorGUILayout.MinMaxSlider(ref minFloatValue,
                             ref maxFloatValue,
                             minLimit,
                             maxLimit);
```

Обратите внимание, что этот метод не возвращает значения; он изменяет переменные `minFloatValue` и `maxFloatValue`. Кроме того, переменные `minLimit` и `maxLimit` ограничивают минимальное и максимальное значения, которые могут быть записаны в `minFloatValue` и `maxFloatValue`.

Пустое пространство

Элемент управления `Space` не имеет визуального представления. Он просто добавляет пустое пространство в пользовательский интерфейс. Его можно использовать, например, чтобы визуально отделить группы видимых элементов управления:

```
EditorGUILayout.Space();
```

Списки

До сих пор мы обсуждали элементы управления, никак не ограничивающие пользователя и позволяющие вводить любой текст или любое число. Но иногда бывают ситуации, когда желательно дать пользователю возможность выбирать только предопределенные варианты из списка.

С этой целью можно использовать раскрывающийся список. Раскрывающийся список использует массив строковых вариантов и возвращает целое число, представляющее выбранный элемент массива; когда пользователь делает выбор, изменяется номер текущего выбора.

Например, если добавить в класс следующие переменные:

```
private int selectedSizeIndex = 0;
```

в метод `OnGUI` можно добавить следующий код:

```
var sizes = new string[] {"small","medium","large"};

selectedSizeIndex
  = EditorGUILayout.Popup(selectedSizeIndex, sizes);
```

Однако порой трудно запомнить соответствие номеров, сохраняемых в `selectedSizeIndex`, и вариантов, которые они представляют. Поэтому для этой цели часто используются *перечисления (enum)*.

Перечисления лучше, потому что проверяются компилятором, — в предыдущем примере вам придется помнить, что «0» означает «small», но было бы намного проще, если бы вместо «0» можно было использовать идентификатор `Small`. Перечисления дают такую возможность!

Давайте объявим перечисление, которое определяет несколько разных типов повреждений, а также добавим переменную для хранения типа повреждения, выбранного в данный момент.

1. Добавьте следующий код в класс `TextureCounter`:

```
private enum DamageType {
  Fire,
  Frost,
  Electric,
  Shadow
}

private DamageType damageType;
```

Используя это перечисление и переменную `damageType`, мы можем создать раскрывающийся список, отображающий значения из этого перечисления.

2. Добавьте следующий код в метод `OnGUI`:

```
damageType
  = (DamageType)EditorGUILayout.EnumPopup(damageType);
```

Он отобразит раскрывающийся список, содержащий все возможные значения, представимые перечислением `DamageType`, и запишет в переменную `damageType` текущее выбранное значение.

Результат, возвращаемый методом `EnumPopup`, требуется привести к типу перечисления, потому что метод ничего не знает об используемом перечислении.

Представления с прокруткой

Если вы следовали за примерами, приводившимися в этой главе, и опробовали их у себя, то могли заметить, что элементы управления уже не умещаются в границах окна редактора. Для решения этой проблемы можно использовать *представления с прокруткой*, дающие возможность прокручивать их содержимое назад/вперед.

Представление с прокруткой должно где-то сохранять свою текущую позицию прокрутки. То есть вы должны создать переменную для этой цели, как это делалось для других элементов управления.

1. *Добавьте следующую переменную в класс TextureCounter*:

```
private Vector2 scrollPosition;
```

Создается представление с прокруткой точно так же, как вертикальный список: нужно создать новый объект `EditorGUILayout.ScrollViewScope` внутри конструкции `using`.

2. *Добавьте следующий код в метод OnGUI*:

```
using (var scrollView =
  new EditorGUILayout.ScrollViewScope(this.scrollPosition)) {
    this.scrollPosition = scrollView.scrollPosition;
```

```
        GUILayout.Label("These");
        GUILayout.Label("Labels");
        GUILayout.Label("Will be shown");
        GUILayout.Label("On top of each other");
    }
```

3. *Вернитесь в Unity* и обратите внимание, что новые надписи появились в прокручиваемой области. Возможно, вам придется изменить размер окна, чтобы увидеть этот эффект.

База данных ресурсов

В завершение нашего обсуждения окон редактора вернемся к цели создания окна `TextureCounter`: мы должны подсчитать количество текстур в проекте и вывести в окно надпись, отображающую его.

Для этого мы воспользуемся классом `AssetDatabase`. Этот класс служит воротами во все ресурсы, имеющиеся в проекте. С его помощью можно получать информацию обо всех файлах, находящихся под контролем Unity, и вносить в них изменения.

У нас недостаточно места, чтобы обсудить все возможности класса `AssetDatabase`, поэтому мы настоятельно рекомендуем вам обратиться к описанию этого класса в руководстве к Unity (*https://docs.unity3d.com/ru/current/Manual/AssetDatabase.html*).

1. *Измените метод OnGUI в классе TextureCounter, как показано ниже*:

```
private void OnGUI() {
  using (var vertical = new EditorGUILayout.VerticalScope()) {
    // Получить список всех текстур
    var paths = AssetDatabase.FindAssets("t:texture");

    // Получить длину списка
    var count = paths.Length;

    // Вывести надпись
    EditorGUILayout.LabelField("Texture Count",
      count.ToString());

  }

}
```

2. *Вернитесь в Unity и добавьте в проект несколько изображений.* Неважно, что это за изображения, — просто перетащите файлы. Если у вас не найдется изображений, посетите сайт Flickr (*https://www.flickr.com/*) и поищите по слову «cats».

После этого окно редактора покажет, сколько текстур вы добавили.

Создание собственного редактора свойства

Кроме создания собственных окон редактора также имеется возможность расширять поведение инспектора.

Роль инспектора — предоставить пользовательский интерфейс для настройки всех компонентов, подключенных к игровому объекту, выбранному в данный момент. Для каждого компонента инспектор отображает элементы управления, представляющие его переменные.

Инспектор уже знает, какие элементы управления использовать для представления распространенных типов данных, таких как строки, целые и вещественные числа. Но если вы определите свой нестандартный тип, инспектор может оказаться не в состоянии представить его правильно. Обычно это не вызывает больших сложностей, но иногда может приводить к беспорядку.

Для решения этой проблемы используются *редакторы свойств* (property drawers). Вы можете передать Unity свой код, чтобы помочь ему определить, как отображать разные типы данных.

Система автоматического размещения элементов пользовательского интерфейса не работает внутри нестандартных редакторов свойств. Поэтому вам придется размещать свои элементы вручную. Но не волнуйтесь — это не так страшно, как кажется, и на примерах мы покажем, как это делается.

Для демонстрации создадим свой класс, представляющий диапазоны значений, который затем можно будет использовать в любых сценариях. Затем мы определим свой редактор свойства для этого класса. С этой целью выполните следующие действия:

1. *Создайте новый сценарий на C#* с именем *Range.cs* и сохраните его в папку *Assets*.
2. *Добавьте следующий код в сценарий Range.cs*:

```
[System.Serializable]
public class Range {

  public float minLimit = 0;
  public float maxLimit = 10;

  public float min;
  public float max;

}
```

Атрибут `System.Serializable` отмечает данный класс как пригодный для сохранения на диск. Он также указывает редактору Unity, что значение с типом этого класса должно отображаться в инспекторе.

3. *Создайте второй сценарий на C#* с именем *RangeTest.cs* и сохраните его в папку *Assets*. Этот сценарий будет подключаться к компонентам, использующим `Range`. Добавьте в *RangeTest.cs* следующий код:

```
public class RangeTest : MonoBehaviour {

  public Range range;

}
```

4. Создайте пустой игровой объект и перетащите на него сценарий RangeTest.

   После выбора игрового объекта в инспекторе появятся простые поля со значениями (рис. 16.7).

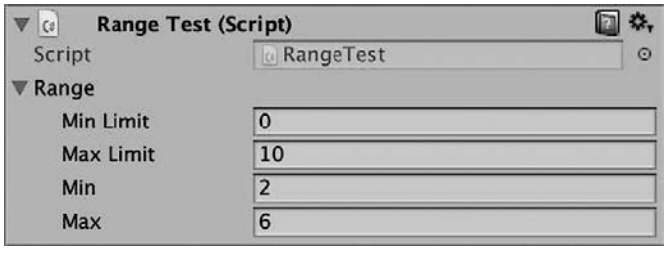

Рис. 16.7. Инспектор отображает интерфейс по умолчанию для класса Range

Чтобы исправить этот недостаток, реализуем новый класс, который заменит интерфейс, используемый редактором Unity по умолчанию.

5. *Создайте новый сценарий* с именем *RangeEditor.cs* и сохраните его в папку *Editor*.
6. *Замените содержимое RangeEditor.cs следующим кодом*:

```
using UnityEngine;
using System.Collections;

using UnityEditor;

[CustomPropertyDrawer(typeof(Range))]
public class RangeEditor : PropertyDrawer {

  // Этот редактор будет отображать два ряда элементов
  // управления друг под другом - в первом будет находиться ползунок,
  // а во втором - текстовые поля ввода для непосредственного
  // изменения значений
  const int LINE_COUNT = 2;

  public override float GetPropertyHeight (
    SerializedProperty property, GUIContent label)
  {
    // Вернуть высоту пространства в пикселах,
    // которые займут это свойство в инспекторе
```

```
  return base.GetPropertyHeight (property, label)
    * LINE_COUNT;
}

public override void OnGUI (Rect position,
  SerializedProperty property, GUIContent label)
{

  // Получить объекты, представляющие поля внутри
  // данного свойства Range
  var minProperty = property.FindPropertyRelative("min");
  var maxProperty = property.FindPropertyRelative("max");

  var minLimitProperty
    = property.FindPropertyRelative("minLimit");
  var maxLimitProperty
    = property.FindPropertyRelative("maxLimit");

  // Любые элементы управления внутри PropertyScope будут
  // правильно работать с шаблонами - значения, отличающиеся
  // от заданных в шаблоне, будут отображаться жирным; вы сможете
  // щелкнуть на значении правой кнопкой и вернуть значение из шаблона
  using (var propertyScope
    = new EditorGUI.PropertyScope(
    position, label, property)) {

      // Показать надпись; этот метод возвращает объект Rect,
      // определяющий размеры прямоугольника, необходимого для вывода
      // указанной строки
      Rect sliderRect
        = EditorGUI.PrefixLabel(position, label);

      // Сконструировать прямоугольники для всех элементов управления:
      // вычислить размер одного ряда
      var lineHeight = position.height / LINE_COUNT;

      // Высота ползунка должна совпадать с высотой ряда
      sliderRect.height = lineHeight;

      // Область для двух полей ввода имеет те же размеры,
      // что и область для ползунка, но смещена на один ряд ниже
      var valuesRect = sliderRect;
      valuesRect.y += sliderRect.height;

     // Определить прямоугольники для двух полей ввода
      var minValueRect = valuesRect;
      minValueRect.width /= 2.0f;

      var maxValueRect = valuesRect;
      maxValueRect.width /= 2.0f;
      maxValueRect.x += minValueRect.width;

      // Вывести вещественные значения
      var minValue = minProperty.floatValue;
      var maxValue = maxProperty.floatValue;

      // Начать проверку изменений - это необходимо для
```

```
        // поддержки редактирования нескольких объектов
        EditorGUI.BeginChangeCheck();

        // Показать ползунок
        EditorGUI.MinMaxSlider(
          sliderRect,
          ref minValue,
          ref maxValue,
          minLimitProperty.floatValue,
          maxLimitProperty.floatValue
        );

        // Показать поля ввода
        minValue
          = EditorGUI.FloatField(minValueRect, minValue);
        maxValue
          = EditorGUI.FloatField(maxValueRect, maxValue);

        // Значение изменилось?
        var valueWasChanged = EditorGUI.EndChangeCheck();

        if (valueWasChanged) {
          // Сохранить изменившиеся значения
          minProperty.floatValue = minValue;
          maxProperty.floatValue = maxValue;
        }
      }

    }
}
```

Получился довольно длинный фрагмент кода, поэтому разберем его по частям.

Создание класса

Прежде всего, необходимо объявить класс и подсказать редактору Unity, что он должен использоваться для создания интерфейса редактирования любого свойства `Range`, которое встретит инспектор. Для этого мы использовали атрибут `CustomPropertyDrawer`, которому в качестве параметра передается тип класса `Range`.

Кроме того, в качестве родительского класса для `RangeEditor` выбран класс `PropertyDrawer`.

```
[CustomPropertyDrawer(typeof(Range))]
public class RangeEditor : PropertyDrawer {
```

Настройка высоты свойства

Свойство занимает в инспекторе некоторое пространство по вертикали. По умолчанию для свойства отводится примерно 20 пикселов; однако для отображения свойства, определяющего диапазон, требуется больше пространства, потому что

нам нужно нарисовать ползунок для задания диапазона, а также два текстовых поля ввода под ним.

В классе `PropertyDrawer` имеется метод `GetPropertyHeight`, задача которого — вернуть высоту свойства в пикселах. Этот метод можно переопределить и вернуть другое значение высоты.

Чтобы не «зашивать в код» точное значение, которое может быть разным в разных версиях Unity, мы определили количество рядов в виде константы `LINE_COUNT` и вызвали базовую реализацию метода из родительского класса (`base`), чтобы получить высоту одного ряда, которую затем умножили на `LINE_COUNT`.

```
// Этот редактор будет отображать два ряда элементов
// управления друг под другом - в первом будет находиться ползунок,
// а во втором - текстовые поля ввода для непосредственного
// изменения значений
const int LINE_COUNT = 2;

public override float GetPropertyHeight (
  SerializedProperty property, GUIContent label)
{
  // Вернуть высоту пространства в пикселах,
  // которые займут это свойство в инспекторе
  return base.GetPropertyHeight (
    property, label) * LINE_COUNT;
}
```

Переопределение OnGUI

Теперь пришло время приступить к реализации основного метода этого класса: `OnGUI`. В редакторах свойств этот метод принимает три параметра.

- Параметр `position` типа `Rect` определяет местоположение и размеры пространства, доступного методу `OnGUI` для отображения элементов управления.
- Параметр `property` — это объект типа `SerializedProperty`, дающий возможность взаимодействовать с конкретным свойством `Range` компонента, подключенного к данному экземпляру класса.
- Параметр `label` — это объект типа `GUIContent`, представляющий некоторое графическое содержимое — обычно текст, — которое должно появиться в виде подписи для данного свойства.

```
public override void OnGUI (Rect position,
  SerializedProperty property, GUIContent label)
{
```

Извлечение свойств

Задача редактора свойства — отобразить и дать возможность изменить единственное свойство внутри компонента. При этом он не изменяет компонент непосред-

ственно — доступ осуществляется посредством промежуточного звена, параметра `property`. Благодаря такой организации Unity сможет предложить дополнительные возможности, такие как автоматическая поддержка отмены изменений.

В случае с объектом `Range` данное свойство содержит другие свойства. Переменные `min`, `max`, `minLimit` и `maxLimit` сами являются свойствами, поэтому мы должны получить доступ к ним:

```
// Получить объекты, представляющие поля внутри
// данного свойства Range
var minProperty = property.FindPropertyRelative("min");
var maxProperty = property.FindPropertyRelative("max");

var minLimitProperty
  = property.FindPropertyRelative("minLimit");
var maxLimitProperty
  = property.FindPropertyRelative("maxLimit");
```

Создание контекста свойства

Мы должны не только получить объекты, представляющие свойства, но также сообщить системе GUI немедленного режима, что отображаемые нами элементы управления имеют отношение к конкретному свойству.

Благодаря этому редактор Unity сможет при необходимости настроить отображение элементов управления; например, когда объект, которому принадлежит свойство, является измененным экземпляром шаблона, значение свойства должно отображаться жирным; кроме того, когда выполняется щелчок правой кнопкой мыши на измененном свойстве, Unity открывает меню и дает возможность вернуть значение, заданное в шаблоне.

Для поддержки всего перечисленного завернем все элементы управления в контекст `PropertyScope`:

```
using (var propertyScope
  = new EditorGUI.PropertyScope(position, label, property)) {
```

Отображение надписей

Теперь выведем надписи, используя элемент управления `PrefixLabel`. Этот элемент управления отображает текст надписи внутри прямоугольника `position` и возвращает новый объект `Rect`, представляющий свободную область за надписью, где можно отобразить другие элементы управления.

Благодаря этой особенности можно размещать элементы свойства, следуя стилю, принятому в редакторе Unity: свойства включают надпись в верхнем левом углу и поля справа; область под надписью остается незанятой:

```
Rect sliderRect = EditorGUI.PrefixLabel(position, label);
```

Вычисление прямоугольников

Теперь, когда известно, сколько пространства доступно, можно приступать к вычислению прямоугольников для размещения всех трех элементов управления: ползунка и двух текстовых полей ввода.

Для этого сначала вычислим высоту одного ряда в пикселах, разделив *общую* высоту на `LINE_COUNT`. Затем установим высоту `sliderRect` равной вычисленной высоте `lineHeight` и оставим исходное значение ширины. После этого ползунок займет все пространство верхнего ряда.

Далее вычислим прямоугольники для двух текстовых полей ввода. Они будут отображаться в одну линию под ползунком. Для этого получим прямоугольник, представляющий весь второй ряд целиком, а затем разделим его ширину пополам:

```
var lineHeight = position.height / LINE_COUNT;

// Высота ползунка должна совпадать с высотой ряда
sliderRect.height = lineHeight;

// Область для двух полей ввода имеет те же размеры,
// что и область для ползунка, но смещена на один ряд ниже
var valuesRect = sliderRect;
valuesRect.y += sliderRect.height;

// Определить прямоугольники для двух полей ввода
var minValueRect = valuesRect;
minValueRect.width /= 2.0f;

var maxValueRect = valuesRect;
maxValueRect.width /= 2.0f;
maxValueRect.x += minValueRect.w
```

Получение значений

Поскольку ползунок `MinMaxSlider` напрямую изменяет переданные ему переменные, нам нужно временно сохранить значения `minProperty` и `maxProperty` в переменных. В конечном счете мы сохраним эти значения обратно в объектах свойства, после их изменения элементами управления, отображением которых мы сейчас занимаемся:

```
var minValue = minProperty.floatValue;
var maxValue = maxProperty.floatValue;
```

Добавление проверки изменений

Прежде чем перейти к отображению элементов управления, необходимо выполнить еще одну настройку — попросить редактор Unity сообщить нам, когда изменится значение в любом элементе управления, которые мы собираемся отобразить.

Это важный шаг — если его не сделать, мы будем изменять свойства при каждом отображении элементов управления, даже если изменения не были подтверждены.

В этом не было бы ничего страшного, но если окажутся выбранными несколько объектов и все они будут иметь свойство типа `Range`, тогда отображение элементов управления для свойства `Range` одного из них приведет к присваиванию им всем одного и того же значения, даже если пользователь вообще ничего не вводил. Добавим проверку изменений, чтобы предотвратить такое непреднамеренное поведение.

```
EditorGUI.BeginChangeCheck();
```

Отображение ползунка

Теперь мы можем отобразить элементы управления. У нас есть все данные, которые они должны показывать, переменные для сохранения возвращаемых ими результатов, а также определены прямоугольники, в которых они должны отображаться.

Начнем с ползунка `MinMaxSlider`:

```
EditorGUI.MinMaxSlider(
  sliderRect,
  ref minValue,
  ref maxValue,
  minLimitProperty.floatValue,
  maxLimitProperty.floatValue
);
```

Отображение полей ввода

Далее отобразим текстовые поля ввода. Обратите внимание, что мы используем те же переменные, что передавались ползунку `MinMaxSlider`; благодаря этому изменение положений ползунков приведет к изменению значений в полях ввода, и наоборот:

```
minValue = EditorGUI.FloatField(minValueRect, minValue);
maxValue = EditorGUI.FloatField(maxValueRect, maxValue);
```

Проверка наличия изменений

Наконец, можно спросить у редактора Unity, имели ли место какие-либо изменения с момента запуска проверки изменений. Метод `EditorGUI.EndChangeCheck` вернет `true`, если такие изменения произошли:

```
var valueWasChanged = EditorGUI.EndChangeCheck();
```

Сохранение свойств

Если элемент управления *изменился*, мы должны сохранить новое значение в свойстве:

```
if (valueWasChanged) {
  // Сохранить изменившиеся значения
  minProperty.floatValue = minValue;
  maxProperty.floatValue = maxValue;
}
```

Тестирование

Вот, собственно, и всё.

Вернитесь в Unity и взгляните на окно инспектора. Вы увидите наш пользовательский интерфейс, представляющий свойство `Range` (рис. 16.8).

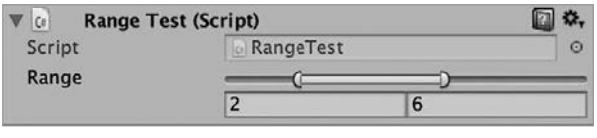

Рис. 16.8. Наш редактор свойства

Вам может потребоваться снять выделение, а затем выделить игровой объект, чтобы обновить пользовательский интерфейс в инспекторе.

Благодаря этому коду, *любое* свойство типа `Range` в *любом* сценарии получит этот пользовательский интерфейс.

Создание собственного инспектора

Последнее, что мы обсудим в этой главе, — создание полностью своих инспекторов. В дополнение к созданию собственных редакторов отдельных свойств, имеется возможность полностью изменить пользовательский интерфейс представления компонентов в инспекторе.

Мы посмотрим, как это сделать, создав для начала простой компонент, а затем создадим совершенно новый интерфейс инспектора для представления этого компонента.

Создание простого сценария

Этот простой компонент будет изменять цвет меша в момент запуска игры.

1. *Создайте новый сценарий* с именем `RuntimeColorChanger`.
2. *Добавьте в файл сценария следующий код*:

```
public class RuntimeColorChanger : MonoBehaviour {

  public Color color = Color.white;
```

```
  void Awake() {
    GetComponent<Renderer>().material.color = color;
  }
}
```

3. *Вернитесь в Unity.* Откройте меню GameObject (Игровой объект) и выберите пункт 3D Object ▸ Capsule (3D Объект ▸ Капсула).
4. *Перетащите сценарий* `RuntimeColorChanger` *в созданный объект.*
5. *Выберите в свойстве Color сценария* `RuntimeColorChanger` *красный цвет* и щелкните по кнопке Play (Играть). Капсула окрасится в красный цвет.

Создание собственного инспектора

Пока все идет хорошо: сценарий делает именно то, что нам нужно.

Теперь создадим собственный инспектор, который добавит крутую возможность: отобразит несколько кнопок с предопределенными цветами, которые позволят быстро менять значение свойства Color сценария `RuntimeColorChanger`.

1. *Создайте сценарий с именем RuntimeColorChangerEditor.cs* и сохраните его в папку *Editor*.
2. *Замените содержимое файла RuntimeColorChangerEditor.cs следующим кодом*:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic; // необходимо для словаря
using UnityEditor;

// Это - редактор для RuntimeColorChangers
[CustomEditor(typeof(RuntimeColorChanger))]
// Поддерживает одновременное редактирование нескольких объектов
[CanEditMultipleObjects]
class RuntimeColorChangerEditor : Editor {

  // Коллекция пар строка/цвет
  private Dictionary<string, Color> colorPresets;

  // Представляет свойство "color" во всех выбранных объектах
  private SerializedProperty colorProperty;

  // Вызывается при первом появлении редактора
  public void OnEnable() {

    // Подготовить список предопределенных цветов
    colorPresets = new Dictionary<string, Color>();

    colorPresets["Red"] = Color.red;
    colorPresets["Green"] = Color.green;
    colorPresets["Blue"] = Color.blue;
    colorPresets["Yellow"] = Color.yellow;
    colorPresets["White"] = Color.white;

    // Получить свойство из объекта (объектов),
    // выбранного в данный момент
```

```
    colorProperty
      = serializedObject.FindProperty("color");
  }

  // Вызывается для отображения пользовательского
  // интерфейса в инспекторе
  public override void OnInspectorGUI ()
  {
    // Гарантировать актуальность состояния serializedObject
    serializedObject.Update();

    // Начать создание вертикального списка элементов управления
    using (var area
      = new EditorGUILayout.VerticalScope()) {

      // Для каждого предопределенного цвета...
      foreach (var preset in colorPresets) {

        // Отобразить кнопку
        var clicked = GUILayout.Button(preset.Key);

        // Если был произведен щелчок, обновить свойство
        if (clicked) {
          colorProperty.colorValue = preset.Value;
        }
      }

      // В заключение вывести текстовое поле ввода, чтобы
      // дать возможность напрямую ввести код цвета
      EditorGUILayout.PropertyField(colorProperty);
    }

    // Применить любые произведенные изменения
    serializedObject.ApplyModifiedProperties();
  }
}
```

И снова рассмотрим этот длинный фрагмент кода по частям.

Подготовка класса

Первый шаг — объявление класса и его роли в системе Unity. Класс `RuntimeColorChangerEditor` наследует класс `Editor`.

Кроме того, мы присвоили классу атрибут `CustomEditor`, указывающий, что данный класс должен использоваться как редактор для любого компонента `RuntimeColorChanger`. Наконец, классу присвоен атрибут `CanEditMultipleObjects` (он, как следует из имени, указывает, что может редактировать сразу несколько объектов):

```
// Это - редактор для RuntimeColorChangers
[CustomEditor(typeof(RuntimeColorChanger))]
// Поддерживает одновременное редактирование нескольких объектов
[CanEditMultipleObjects]
class RuntimeColorChangerEditor : Editor {
```

Определение цветов и свойств

Класс должен хранить два блока информации. Первый — список предопределенных цветов, из которых мы сможем выбирать. Второй — объект, представляющий свойство `color` во всех выбранных в данный момент объектах.

Точно так же, как мы делали это в собственном редакторе свойства, мы будем представлять свойства с помощью объекта `SerializedProperty`. Благодаря этому Unity сможет предложить дополнительные возможности, такие как автоматическая поддержка отмены изменений:

```
// Коллекция пар строка/цвет
private Dictionary<string, Color> colorPresets;

// Представляет свойство "color" во всех выбранных объектах
private SerializedProperty colorProperty;
```

Настройка переменных

Когда пользователь выберет объект с компонентом `RuntimeColorChanger`, инспектор создаст для него редактор, затем вызовет метод `OnEnable`, который дает нам первую возможность выполнить какие-либо настройки. В данном редакторе мы подготавливаем словарь `colorPresets`, заполняя его предопределенными цветами.

Кроме того, мы должны получить свойство для работы. Для этого используется переменная `serializedObject`, которая настраивается редактором Unity; эта переменная представляет все объекты, выбранные в данный момент.

```
public void OnEnable() {

  // Подготовить список предопределенных цветов
  colorPresets = new Dictionary<string, Color>();

  colorPresets["Red"] = Color.red;
  colorPresets["Green"] = Color.green;
  colorPresets["Blue"] = Color.blue;
  colorPresets["Yellow"] = Color.yellow;
  colorPresets["White"] = Color.white;

  // Получить свойство из объекта (объектов),
  // выбранного в данный момент
  colorProperty = serializedObject.FindProperty("color");
}
```

Начало отображения пользовательского интерфейса

В методе `OnInspectorGUI` можно реализовать отображение своего пользовательского интерфейса инспектора. Первым делом мы просим объект `serializedObject` обновить себя до состояния, соответствующего текущему состоянию сцены, что-

бы гарантировать, что отображаемые нами элементы управления будут точно представлять сцену:

```
public override void OnInspectorGUI ()
{
  // Гарантировать актуальность состояния serializedObject
  serializedObject.Update();
```

Отображение элементов управления

Теперь мы можем отобразить элементы управления для редактирования свойства данного компонента. С помощью `VerticalScope` мы отображаем кнопки, по одной для каждого цвета, присутствующего в словаре `colorPresets`. Если пользователь щелкнет по какой-то из них, свойству `colorProperty` будет присвоено соответствующее значение цвета.

Вслед за отображением кнопок вызывается метод `PropertyField`, чтобы нарисовать поле ввода для свойства. `PropertyField` автоматически выбирает элемент управления, соответствующий типу свойства. В данном случае, поскольку `colorProperty` представляет переменную `color` в `RuntimeColorChanger`, в пользовательском интерфейсе появится элемент выбора цвета, позволяющий пользователю определить свой цвет. Так мы сохраняем возможность сделать нестандартный выбор, а также предлагаем дополнительные возможности:

```
using (var area = new EditorGUILayout.VerticalScope()) {

    // Для каждого предопределенного цвета...
    foreach (var preset in colorPresets) {

      // отобразить кнопку
      var clicked = GUILayout.Button(preset.Key);

      // Если был произведен щелчок, обновить свойство
      if (clicked) {
        colorProperty.colorValue = preset.Value;
      }
    }

    // В заключение вывести текстовое поле ввода, чтобы
    // дать возможность напрямую ввести код цвета
    EditorGUILayout.PropertyField(colorProperty);
}
```

Применение изменений

Последнее, что осталось сделать, — потребовать от выбранного объекта (или объектов, если выбрано несколько объектов) применить сделанные изменения. Для этого вызывается метод `ApplyModifiedProperties` объекта `serializedObject`.

```
// Применить любые произведенные изменения
serializedObject.ApplyModifiedProperties();
```

Тестирование

Теперь можно протестировать работу нашего собственного инспектора.

Выберите игровой объект, и вы увидите наш пользовательский интерфейс в инспекторе (рис. 16.9). Вам может потребоваться снять выделение и снова выделить капсулу.

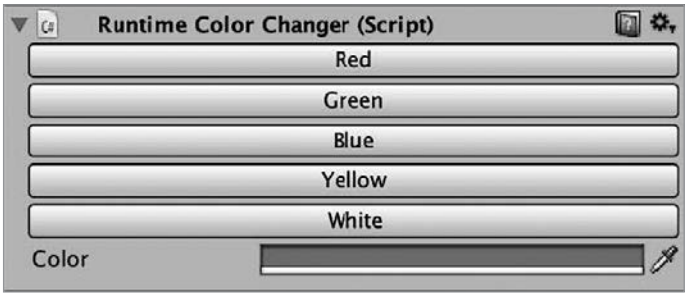

Рис. 16.9. Наш инспектор

ОТОБРАЖЕНИЕ СОДЕРЖИМОГО ПО УМОЛЧАНИЮ В ИНСПЕКТОРЕ

Иногда требуется не заменить интерфейс инспектора для какого-то компонента, а только что-нибудь добавить. В таких случаях можно использовать метод `DrawDefaultInspector` для быстрого отображения всего, что обычно отображает инспектор; после этого можно добавить дополнительные элементы управления, выше или ниже:

```
public override void OnInspectorGUI() {

    // Отобразить интерфейс инспектора по умолчанию
    DrawDefaultInspector();

    // Показать ободряющее сообщение разработчику
    var msg = "You're doing a great job! " +
      "Keep it up!";

    EditorGUILayout.HelpBox(msg, MessageType.Info);
}
```

В заключение

Собственные редакторы могут облегчить вам жизнь. Если вам постоянно приходится выполнять повторяющиеся операции или нужен другой, лучший способ отобразить данные, содержащиеся в ваших объектах, редакторы могут оказать вам действенную помощь. Тем не менее помните, что игроки никогда не увидят ваши редакторы. Они существуют только для вас как для разработчика, поэтому не увлекайтесь чересчур созданием собственных редакторов — важнее то, что они помогают создавать.

17 За рамками редактора

Ваша игра закончена, игровой процесс отлажен, и все выглядит прекрасно. Но что делать дальше?

Пришло время заглянуть за пределы редактора Unity. Проект Unity предоставляет ряд полезных служб, которые можно использовать для улучшения игры, усовершенствования способов создания игр или даже для заработка на своих играх. В этой главе мы познакомимся со всеми тремя.

Мы также обсудим сборку игр для устройств и то, как сделать их доступными во всем мире.

Инфраструктура служб Unity

Когда люди обсуждают Unity, они обычно имеют в виду редактор Unity editor — программное обеспечение, развиваемое и продаваемое компанией Unity Technologies. Однако Unity — это больше, чем редактор. Помимо этого программного обеспечения Unity поддерживает службы, целью которых является облегчение труда разработчиков. Наиболее популярные из них: интернет-магазин ресурсов Asset Store, служба сборки Unity Cloud Build и платформа для распространения игр Unity Ads.

Asset Store

Unity Asset Store — это интернет-магазин, где программисты, художники и другие создатели контента могут продавать его для дальнейшего использования в играх.

Интернет-магазин Asset Store особенно удобен для тех, кому не хватает определенных навыков; например, программисты, не умеющие (или не имеющие времени для этого) создавать художественные ресурсы, могут приобрести здесь необходимые им трехмерные модели, благодаря чему у них останется больше времени на то, что они умеют делать лучше всего, — на программирование. Это относится и к тем, кому нужны звуки, сценарии и т. д. В Asset Store можно найти самый разнообразный контент: вы можете приобрести единственную трехмерную модель автомобиля или даже полный комплект ресурсов для игры определенного типа.

Ресурсы, приобретаемые вами в Asset Store, особенно *качественные* ресурсы, легко опознаются многими как ресурсы, приобретенные в интернет-магазине. Не нужно слишком полагаться на ресурсы из Asset Store, чтобы ваша игра не выглядела похожей на другие игры.

Некоторые ресурсы в магазине особенно интересны, потому что добавляют новые возможности в Unity.

PlayMaker

PlayMaker — это инструмент визуального программирования сценариев, созданный компанией Hutong Games. Эта система визуального программирования позволяет определять поведение игровых объектов путем соединения их с предопределенными модулями кода, которые отображаются в виде блоков с исходящими из них связями.

Системы визуального программирования являются альтернативным способом создания программного кода, и их часто легче понять новичкам в программировании. Особенно хорошо они подходят для представления разных видов поведения, основанных на состояниях, — например, для наделения искусственным интеллектом врага, который движется, выбирая случайные направления, пока не увидит игрока, после чего переходит в состояние *преследования* и атакует, пока не погибнет он сам, персонаж игрока или пока не потеряет игрока из виду.

PlayMaker можно найти в Asset Store по адресу *https://www.assetstore.unity3d.com/en/#!/content/368*.

Установка PlayMaker. Так как PlayMaker реализует совершенно иной подход к определению поведения игры, имеет смысл познакомиться с ним поближе и посмотреть, как с его помощью можно определить некоторое простое поведение. Для выполнения шагов, описываемых далее, вам понадобится приобрести PlayMaker в магазине Asset Store; на момент написания этих строк — конец 2017-го — его цена составляла $65.

Действия, описываемые далее, мы выполняли в новом пустом проекте, настроенном для отображения трехмерной графики.

1. *Загрузите и импортируйте пакет.* На экране появится окно мастера установки (рис. 17.1).
2. *Щелкните по кнопке Install (Установить).* PlayMaker проверит проект, чтобы убедиться, что сможет работать в нем, и установит самую последнюю версию.

PlayMaker предупредит, если вы не пользуетесь инструментами управления версиями, такими как Git; вы можете проигнорировать это предупреждение, но вообще использование систем управления версиями — отличная идея.

Рис. 17.1. Окно мастера установки PlayMaker, появляющееся после запуска импортирования пакета

Рис. 17.2. Второе окно мастера установки

3. *Щелкните по кнопке Install (Установить)* во втором окне мастера установки (рис. 17.2), а затем, в появившемся диалоге, щелкните по кнопке I Made a Backup, Go Ahead! (Я сделал резервную копию, вперед!). Unity импортирует второй пакет.
4. *В появившемся окне щелкните по кнопке Import (Импортировать).*

В зависимости от версии Unity вам может быть предложено обновить код для совместимости с последней версией API. Вам необходимо согласиться, чтобы продолжить.

После завершения установки закройте все лишние окна. Теперь вы готовы к использованию PlayMaker.

Проигрывание с PlayMaker. Идея PlayMaker основывается на понятии *конечных автоматов*, или *КА*. Конечный автомат (Finite State Machine, FSM) — это логическая система, в которой объект может находиться в одном из нескольких *состояний*; каждое состояние может изменяться или *переходить* в предопределенное множество состояний. То есть если у вас имеются состояния *сидит*, *стоит* и *бежит*, то объект, находящийся в состоянии *стоит*, сможет перейти в состояние *сидит* или *бежит*, но не сможет из состояния *сидит* перейти сразу в состояние *бежит*. В момент смены состояния появляется возможность выполнить некоторое действие.

В этом коротком примере мы реализуем очень простое действие: создадим шар, который меняет цвет после падения на поверхность.

Сначала настроим окружение.

1. *Создайте сферу*, открыв меню GameObject (Игровой объект) и выбрав пункт 3D Object ▸ Sphere (3D Объект ▸ Сфера).

   Выберите вновь созданный объект и в компоненте Transform (в инспекторе) установите позицию (0, 15, 0).
2. *Добавьте в сферу компонент Rigidbody*.
3. *Создайте поверхность*, открыв меню GameObject (Игровой объект) и выбрав пункт 3D Object ▸ Plane (3D Объект ▸ Плоскость).

   Установите объект в позицию (0, 0, 0).
4. Наконец, *установите камеру* в позицию (0, 9, −16) с нулевым поворотом. Благодаря этому в поле зрения окажутся и шар, и плоскость.

После этих действий сцена должна выглядеть так, как показано на рис. 17.3.

Теперь начнем добавлять в сферу реакции на смену состояний.

1. *Откройте редактор PlayMaker*, выбрав пункт PlayMaker Editor в меню PlayMaker.

   На экране появится вкладка PlayMaker Editor (рис. 17.4).

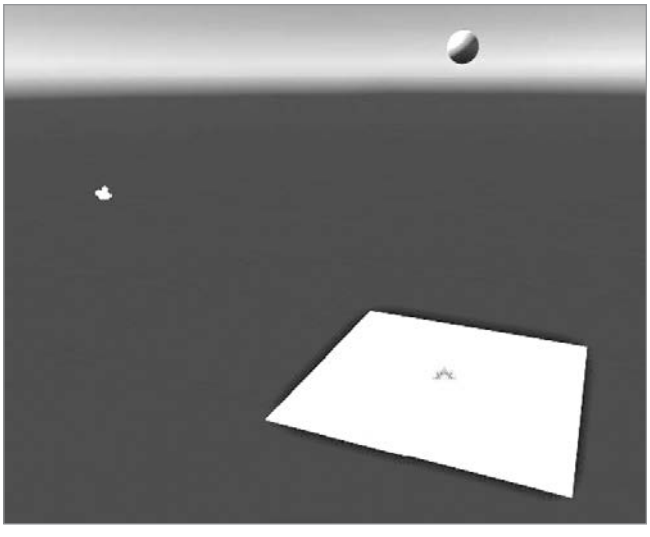

Рис. 17.3. Начальное состояние сцены для учебного примера

Возможно, будет удобнее присоединить вкладку к окну Unity. Для этого перетащите вкладку на окно, к которому вы хотели бы ее присоединить.

Рис. 17.4. Редактор PlayMaker

2. *Добавьте конечный автомат в сферу.* Выберите объект Sphere, в окне PlayMaker щелкните правой кнопкой мыши и в открывшемся меню выберите пункт Add FSM (Добавить КА).

В окне PlayMaker отображается множество разных советов, в целом весьма полезных, но занимающих слишком много места. Вы можете отключить подсказки, нажав клавишу F1 или щелкнув по кнопке Hints (Подсказки) справа внизу в окне PlayMaker.

По умолчанию конечный автомат содержит единственное состояние с названием State1. В нашем примере будут поддерживаться два состояния: `Falling` (Падение)

и `HitGround` (Поверхность достигнута). Мы просто переименуем первое состояние, а затем добавим второе.

3. *Переименуйте первое состояние в* Falling, выбрав его во вкладке State (Состояние) справа в окне PlayMaker и изменив его имя на Falling.
4. *Добавьте состояние HitGround*, щелкнув правой кнопкой в окне PlayMaker и выбрав в контекстном меню пункт Add State (Добавить состояние). Переименуйте новое состояние в HitGround.

   Теперь конечный автомат должен выглядеть так, как показано на рис. 17.5.

Рис. 17.5. Конечный автомат с состояниями

Нам нужно, чтобы смена состояний происходила, когда шар достигнет поверхности. Для этого создадим *переход* из состояния `Falling` в состояние `HitGround`, который будет инициироваться, когда объект с конечным автоматом столкнется с чем-либо.

5. *Добавьте переход*, щелкнув правой кнопкой на состоянии `Falling` и выбрав в контекстном меню пункт Add Transition ▸ System Events ▸ COLLISION ENTER (Добавить переход ▸ Системные события ▸ Контакт). Новый переход появится с предупреждающей пиктограммой, показывающей, что переход не связан с состоянием назначения (рис. 17.6).

Рис. 17.6. Конечный автомат после добавления перехода, но до соединения его с состоянием назначения

6. *Соедините переход с состоянием HitGround*, щелкнув левой кнопкой на переходе COLLISION ENTER и перетащив его на состояние `HitGround`. В результате появится стрелка, соединяющая их (рис. 17.7).
7. *Протестируйте игру*, щелкнув по кнопке Play (Играть). PlayMaker подсветит текущее состояние Falling в окне FSM, и оно будет оставаться подсвеченным, пока сфера не коснется плоскости.

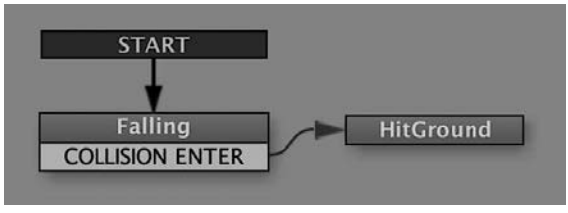

Рис. 17.7. Конечный автомат после соединения перехода с состоянием

Далее нам нужно добавить действие, выполняющееся, когда объект перейдет в состояние `HitGround`. В частности, мы должны изменить цвет материала.

1. *Добавьте в состояние HitGround действие Set Material Color (Установить цвет материала)*, перейдя на вкладку State (Состояние) и щелкнув по кнопке Action Browser (Браузер действия). Откроется окно браузера действий; прокрутите вниз до кнопки Material (Материал), щелкните по ней, а затем выберите пункт Set Material Color (Установить цвет материала), как показано на рис. 17.8. Щелкните по кнопке Add Action to State (Добавить действие в состояние), после чего выбранное действие появится во вкладке State (Состояние), как показано на рис. 17.9.
2. *Выберите конечный цвет — зеленый*. В разделе State (Состояние) выберите зеленый цвет.

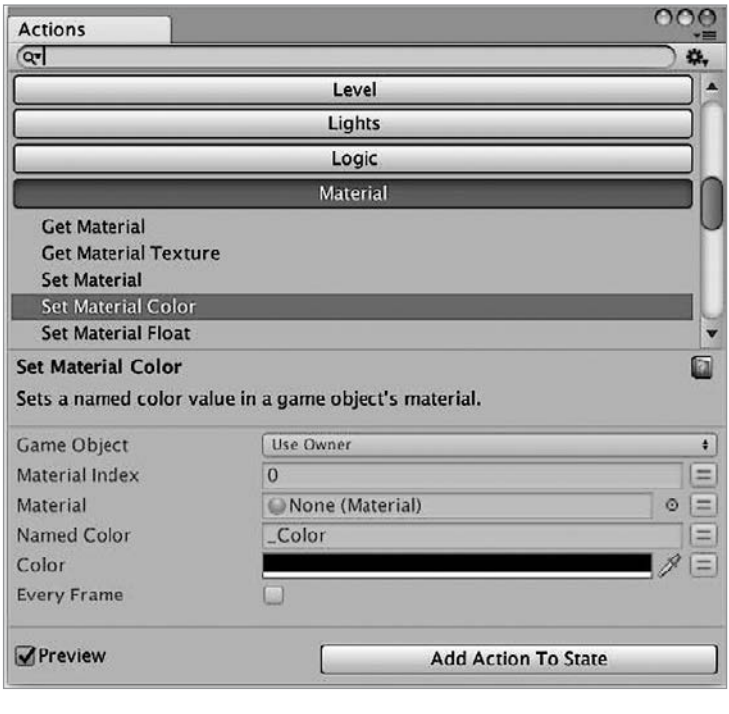

Рис. 17.8. Браузер действий

Рис. 17.9. Конечный автомат с добавленным и настроенным действием

3. *Протестируйте игру*. Когда шар коснется плоскости, он окрасится в зеленый цвет.

Amplify Shader Editor

Создание шейдеров, как рассказывалось в главе 14, обычно связано с созданием программного кода. Однако визуальная природа шейдеров предполагает даже большую их пригодность для визуального конструирования, чем код, реализующий процесс игры; вместо того чтобы писать код, перемножающий два вектора, представляющих смешиваемые цвета, намного понятнее было бы *видеть* происходящее.

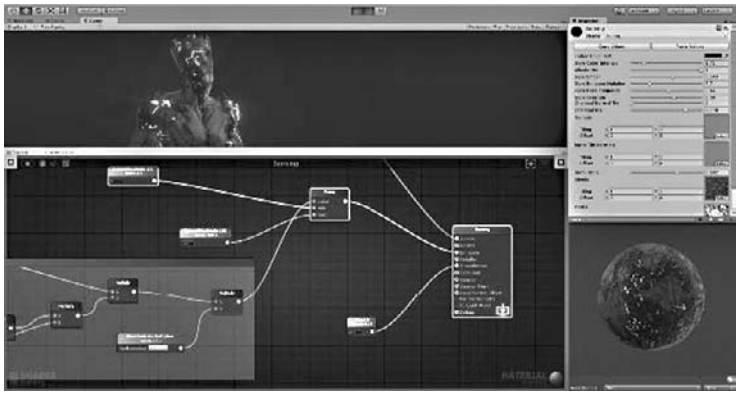

Рис. 17.10. Amplify Shader Editor

Amplify Shader Editor (рис. 17.10) — один из нескольких визуальных редакторов шейдеров для Unity. Соединяя узлы вместе, Amplify Shader Editor создает и демонстрирует получаемый материал, а также генерирует ресурсы для использования в игре. Это часто быстрее и проще, чем писать код шейдера вручную, и особенно удобно для тех, кому визуальное создание результатов кажется более понятным.

Amplify Shader Editor можно найти в Asset Store по адресу *https://www.assetstore.unity3d.com/en/#!/content/68570*.

UFPS

UFPS, или Ultimate FPS (рис. 17.11), — это простая основа для игр-шутеров от первого лица. В Unity уже имеется готовый контроллер «от первого лица», но он не включает в себя другие возможности, свойственные подобным играм, такие как ныряние, подъем по лестницам или взаимодействие с кнопками. UFPS реализует все это, а также другие возможности, характерные для шутеров, такие как управление оружием, боеприпасами и здоровьем игрока.

Несмотря на то что пакет UFPS предназначен для создания динамичных шутеров, он также хорошо подходит для игр с более медленным темпом. Примером может служить игра Gone Home (2014), созданная студией Fullbright, в которой игрок путешествует по дому и исследует находящиеся в нем объекты, документы и мебель. В ней как раз для представления сцены от первого лица используется пакет UFPS.

Рис. 17.11. Ultimate FPS

UFPS можно найти в Asset Store по адресу *https://www.assetstore.unity3d.com/en/#!/content/2943*.

Unity Cloud Build

Сборка проекта для целевой платформы — сложный процесс, требующий времени и большой вычислительной мощности. Вы *можете* выполнить все на собственном компьютере, но *не обязаны* поступать так все время, особенно в случаях с большими и сложными проектами.

Служба Unity Cloud Build загружает ваш исходный код, собирает его и создает сборку, доступную для загрузки (или извещает об ошибке, возникшей во время сборки). Если настроить в службе Cloud Build слежение за репозиторием, она будет обнаруживать изменения и автоматически выполнять сборку.

Можно, конечно, создать свой сервер сборки и не пользоваться службой Cloud Build. Но это сложный процесс, и он потребует одной из двух активаций, получаемых с лицензией. Служба Cloud Build лишает вас определенной доли контроля, но взамен дает простоту использования.

К моменту написания этих строк служба Cloud Build предоставляла свои услуги бесплатно. Если вы приобрели подписку Unity Plus, ваши сборки будут запускаться с более высоким приоритетом и, соответственно, выполняться быстрее. Если вы приобрели подписку Unity Pro, ваши сборки также будут выполняться параллельно, то есть если ваша игра поддерживает несколько платформ (например, iOS и Android), обе версии будут собираться одновременно.

Больше информации о службе Cloud Build вы найдете на веб-сайте Unity: *https://unity3d.com/ru/unity/features/cloud-build*.

Unity Ads

Служба Unity Ads интегрирует полноэкранную видеорекламу в игры. Когда игрок видит рекламу, вы получаете небольшие отчисления. Таким образом вы можете получить дополнительный источник дохода от своих игр.

В качестве варианта можно *вознаграждать игрока за просмотр видеорекламы*, например, увеличивая его сумму в игровой валюте, добавляя некоторые косметические улучшения или расширяя какие-либо возможности.

Проектирование стратегии монетизации игр — обширная тема, и для ее обсуждения может потребоваться (и печатается) целая библиотека книг. Чтобы поближе познакомиться со службой Unity Ads, загляните на страницу на веб-сайте Unity: *https://unity3d.com/ru/unity/features/ads*.

Развертывание

Когда вы будете готовы выгрузить свою игру на устройство, редактору Unity придется *собрать* ее. Для этого он должен выполнить три этапа: собрать все ресурсы игры, скомпилировать сценарии и установить собранное приложение на устрой-

ство. Первые два этапа Unity выполнит автоматически, а третий этап вы должны выполнить самостоятельно.

В этом разделе мы расскажем, как собирать игры для обеих платформ, iOS и Android. Но перед этим мы должны коснуться некоторых настроек, которые вы должны выполнить, и рассказать о различиях между разными версиями Unity.

Настройка проекта

Выполнить сборку проекта можно в любой момент. Однако для достижения лучших результатов желательно сначала убедиться в правильности настройки параметров проигрывателя для вашей игры. Параметры проигрывателя включают название игры и пиктограмму, а также параметры, управляющие ориентацией экрана, уникальную строку для идентификации вашей игры в операционной системе, куда она будет устанавливаться, и т. д.

Чтобы перейти к настройке этих параметров, откройте меню Edit (Правка) и выберите пункт Project Settings ▸ Player (Настройки проекта ▸ Проигрыватель). В инспекторе отобразится интерфейс, как показано на рис. 17.12.

Некоторые настройки являются общими для нескольких платформ. Например, название игры едва ли будет зависеть от платформы, как и пиктограмма. Подобные универсальные настройки в Unity обозначаются звездочкой (*), следующей за их именами.

Каждое приложение для обеих платформ — iOS и Android — должно определять:

- *рекламное название* игры, которое будет отображаться на начальном экране и в интернет-магазине;
- *название компании*, создавшей игру, для отображения в интернет-магазине;
- *пиктограмму* игры, также для отображения на начальном экране и в интернет-магазине;
- *экран-заставку*, который будет отображаться в момент запуска игры;
- *идентификатор пакета* игры — фрагмент теста, уникальным образом идентифицирующий игру в интернет-магазине и не видимый пользователю; этот идентификатор создается на основе вашего доменного имени (например, *oreilly.com*) с компонентами, переставленными в обратном порядке, к которому добавляется название игры (например, *com.oreilly.MyAwesomeGame*).

Для тестирования игры достаточно определить название и идентификатор. Чтобы выложить свою игру в iTunes App Store или Google Play, вы должны настроить все перечисленные элементы.

По умолчанию в качестве названия продукта используется имя проекта, а в качестве названия компании выбирается строка «DefaultCompany». Если вас это устраивает, можете оставить название продукта как есть (можно видеть в верхней части рис. 17.12).

Рис. 17.12. Настройки проигрывателя в инспекторе

Чтобы изменить идентификатор пакета, выберите целевую платформу из меню (под параметром Cursor Hotspot (Начальная позиция курсора)) и откройте раздел Other Settings (Другие настройки). Здесь настройте параметр Bundle Identifier (Идентификатор пакета), выбрав идентификатор по своему желанию (рис. 17.13).

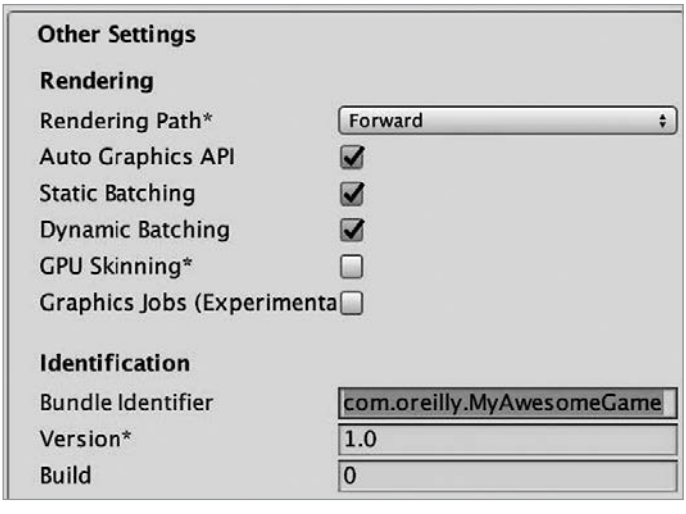

Рис. 17.13. Настройка идентификатора пакета для проекта

Если вы решите собрать игру для обеих платформ — iOS и Android, вам не придется настраивать идентификатор пакета дважды. Этот параметр является общим для всех платформ, так же как номер версии игры и некоторые другие значения.

Настройка целевой платформы

Для сборки можно выбрать только одну целевую платформу. По умолчанию Unity выбирает целевую платформу PC, Mac и Linux Standalone (Автономная для PC, Mac и Linux) и конкретную ее разновидность — в соответствии с платформой, на которой выполняется Unity. То есть если вы пользуетесь компьютером Mac, по умолчанию будет выбрана цель macOS, а для PC — Windows.

ЗАГРУЗКА МОДУЛЕЙ ДЛЯ ПЛАТФОРМЫ

Чтобы выполнить сборку для конкретной платформы, в Unity нужно предварительно установить необходимые модули. В процессе установки Unity мастер спрашивает, на какие платформы предполагается устанавливать игры; если в вашей установке отсутствуют модули, необходимые для выбранной платформы, окно Build Settings (Параметры сборки) будет выглядеть, как показано на рис. 17.14. Щелкните по кнопке в окне, чтобы загрузить и установить необходимые модули.

Рис. 17.14. Окно с параметрами сборки, где выбрана платформа, для которой отсутствуют необходимые модули

Например, для сборки игр, созданных нами в частях II и III, нужно сначала выбрать требуемую платформу. Для этого откройте окно Build Settings (Параметры сборки), выбрав в меню File (Файл) пункт Build Settings (Параметры сборки), как показано на рис. 17.15.

Рис. 17.15. Окно Build Settings (Параметры сборки)

Затем просто выберите целевую платформу и щелкните по кнопке Switch Platform (Переключить платформу) в окне внизу слева.

После переключения платформы Unity заново импортирует все ресурсы игры. Если вы создаете большой проект, это может занять много времени. Будьте готовы ждать. В целях сокращения потерь времени в состав Unity входит инструмент под названием Cache Server, который хранит копии импортированных ресурсов; за дополнительной информацией обращайтесь к документации (*https://docs.unity3d.com/ru/current/Manual/CacheServer.html*).

Начальный экран

Важно отметить различия между тарифными планами Free, Plus и Pro. Пользователи тарифного плана Free не смогут собрать игру без начального экрана, появляющегося в момент запуска игры, тогда как пользователи тарифных планов Plus и Pro могут отключить его создание.

Начальный экран имеет довольно минималистичный вид; он содержит логотип Unity со словами «Made with Unity» (Сделано с помощью Unity) и отображается в течение двух секунд, пока начальная сцена игры продолжает загружаться в фоновом режиме.

Экран-заставка может настраиваться в очень широких пределах, независимо от используемой версии Unity. Кроме логотипа Unity, в нем можно также отобразить свой логотип, изменить цвет фона, задать фоновое изображение и прозрачность, а также настроить одновременное или попеременное отображение нескольких логотипов. Для настройки экрана-заставки откройте меню Edit (Правка), выберите пункт Project Settings ▸ Player (Настройки проекта ▸ Проигрыватель) и прокрутите вниз до параметра Splash Screen (Экран-заставка) в разделе Splash Image (Изображение-заставка). В документации к Unity приводится масса информации по этой теме, поэтому за дополнительными сведениями обращайтесь к ней (*https://docs.unity3d.com/ru/current/Manual/class-PlayerSettingsSplashScreen.html*).

Сборка для выбранной платформы

Этапы сборки для iOS и Android отличаются. Поэтому в данном разделе мы рассмотрим процедуру сборки для каждой из этих систем.

Сборка для iOS

Unity упрощает сборку версии игры для iOS. В этом разделе мы рассмотрим процесс получения игры, готовой к работе на телефоне.

В настоящее время сборка для iOS возможна только в macOS или с помощью службы Unity Cloud (которая выполняет сборку на своих компьютерах Macs).

Вы можете совершенно бесплатно развертывать свои игры на своих личных устройствах. Но чтобы передать игру другим, вам придется воспользоваться службой iTunes App Store. А это означает, что вы должны будете зарегистрироваться в программе Apple Developer Program, годовая подписка на которую стоит $99. Регистрацию можно пройти по адресу *https://developer.apple.com/programs/*.

Сначала вам нужно загрузить Xcode — среду разработки для iOS.

1. *Загрузите Xcode* из Mac App Store; для этого запустите это приложение, найдите среду разработки Xcode и загрузите ее.
2. *Запустите Xcode* после завершения загрузки.

Теперь нужно настроить учетную запись в Xcode. Независимо от участия в платной программе Apple Developer Program, среда Xcode должна использовать ваш идентификатор Apple ID для регистрации вас как разработчика, чтобы добавить обязательную подпись в код, необходимую для установки игры на устройство.

1. *Откройте меню Xcode и выберите пункт Preferences (Настройки)*. Щелкните по кнопке Accounts (Учетные записи) в верхней части окна. Щелкните по кнопке + внизу слева в окне. Выберите в контекстном меню пункт Add Apple ID (Добавить Apple ID).
2. *Подключите свое устройство к порту USB компьютера*.

После настройки Xcode можно приступать к сборке игры в Unity.

1. *Вернитесь в Unity и откройте окно Build Settings (Параметры сборки)*, выбрав пункт Build Settings (Параметры сборки) в меню File (Файл).
2. *Выберите платформу iOS и щелкните по кнопке Switch Platform (Переключить платформу)*. Unity переключит целевую платформу проекта на iOS (рис. 17.16). Это может занять несколько минут.

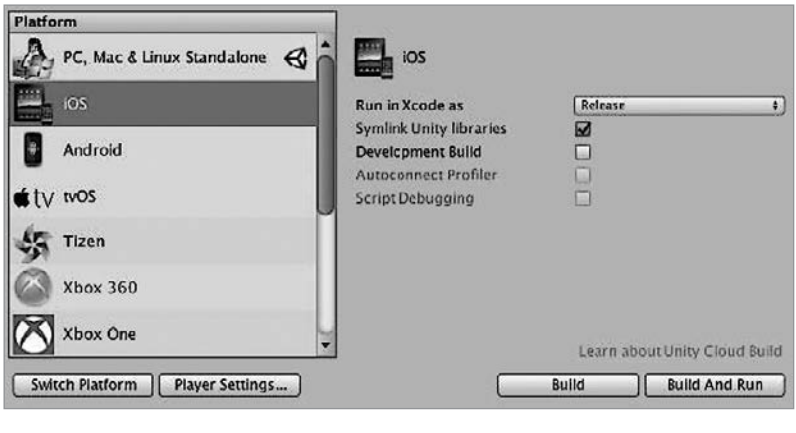

Рис. 17.16. Окно Build Settings (Параметры сборки) с выбранной целевой платформой iOS

Для экономии дискового пространства включите флажок Symink Unity Libraries (Использовать символические ссылки на библиотеки). Это предотвратит копирование в проект всего комплекта библиотек Unity и сэкономит несколько сотен мегабайт дискового пространства.

3. *Щелкните по кнопке Build and Run (Собрать и запустить)*. Unity спросит, куда сохранить проект; после выбора папки Unity соберет приложение для iOS, затем откроет проект в среде Xcode и даст ей команду собрать и запустить приложение на подключенном устройстве.

ПРОБЛЕМА С ПОДПИСЬЮ КОДА

Если вы получите сообщение об ошибке, связанной с подписью в коде, выберите проект слева вверху в окне, выберите цель Unity-iPhone, свой коллектив разработки (это может быть просто ваше имя) в раскрывающемся списке Team (Команда) и щелкните по кнопке Fix Issue (Исправить проблему), как показано на рис. 17.17. Среда разработки проанализирует ваши сертификаты и исправит проблему; после этого нажмите комбинацию Command-R, чтобы повторить попытку сборки.

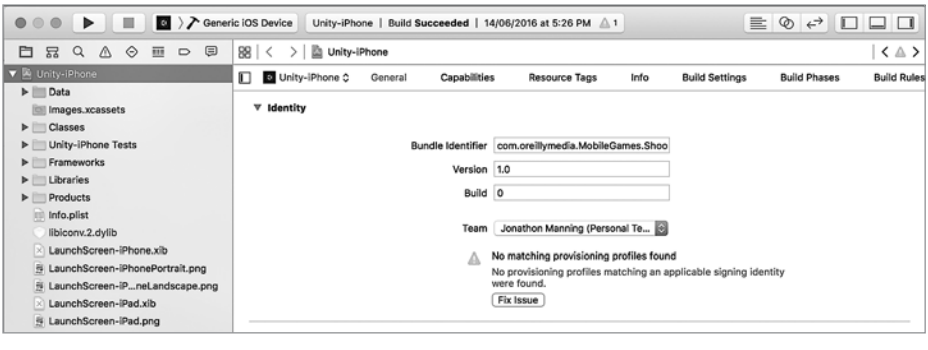

Рис. 17.17. Кнопка Fix Issue (Исправить проблему) в Xcode

Сборка для Android

Чтобы собрать игру для Android, нужно сначала установить комплект инструментов Android SDK. Он даст возможность выгрузить собранное приложение на устройство.

1. *Загрузите Android SDK* на сайте Android Developer: *http://developer.android./com/sdk*.454
2. *Установите SDK*, следуя инструкциям по адресу *http://developer.android.com/sdk/installing/index.html*.

Пользователям Windows может понадобиться загрузить дополнительный драйвер USB для поддержки взаимодействий компьютера с устройством на Android. Загрузить драйвер можно по адресу *http://developer.android.com/sdk/win-usb.html*. Пользователям macOS и Linux этого не требуется.

Теперь можно сообщить редактору Unity, куда установлен Android SDK.

1. *Откройте меню Unity и выберите пункт Preferences ▸ External Tools (Настройки ▸ Внешние инструменты)*. В открывшемся окне щелкните по кнопке Browse (Обзор) рядом с полем SDK и найдите папку установки Android Studio.
2. *Откройте окно Build Settings (Параметры сборки)*, выбрав в меню File (Файл) пункт Build Settings (Параметры сборки).
3. *Выберите платформу Android и щелкните по кнопке Switch Platform (Переключить платформу)*. Unity переключит целевую платформу проекта на Android.
4. *Установите флажок Google Android Project (Проект для Google Android)*, как показано на рис. 17.18. После этого Unity создаст проект для Android Studio.

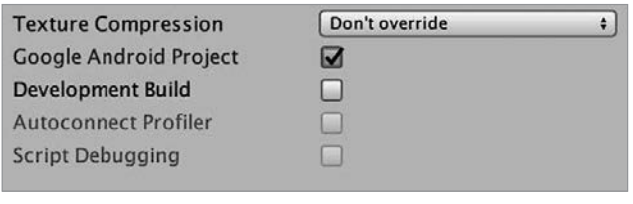

Рис. 17.18. Чтобы собрать игру для Android, нужно сгенерировать проект для Google Android

5. *Щелкните по кнопке Export (Экспортировать)*. Unity попросит указать, куда сохранить проект, и затем сгенерирует проект для Android Studio.
6. *Откройте проект в Android Studio и щелкните по кнопке Play (Играть)*. Проект скомпилируется и установится на ваш телефон.

В то время как Unity обычно не претерпевает больших изменений между выпусками версий, процессы настройки и сборки для мобильных устройств могут существенно изменяться. Поэтому чтобы не повторять документацию с описанием Unity, которая может быстро устареть, мы просто направим вас к подробным пошаговым руководствам для Unity, описывающим настройку для Android и iOS:

- «Начало разработки под Android» (*https://docs.unity3d.com/ru/current/Manual/android-GettingStarted.html*)
- «Первые шаги в iOS разработке» (*https://docs.unity3d.com/ru/current/Manual/iphone-GettingStarted.html*)

В обоих случаях — iOS и Android — для разработки требуется загрузить и установить значительный объем программного обеспечения. Мы рекомендуем выполнять установку в том месте, где у вас будет качественное подключение к интернету.

Куда идти дальше

Добро пожаловать в последний раздел книги! Если вы дошли до этого места, то, значит, совершили большое путешествие: начав с нуля, создали две законченные игры, а затем узнали, как приспособить Unity под свои нужды.

Если вы сразу перескочили в конец книги, не прочитав ее: вот так она заканчивается. Поздравляем со спойлером!

Прежде чем наши пути разойдутся, вот несколько полезных ресурсов для будущего чтения:

- Unity предлагает отличную документацию (*https://docs.unity3d.com/ru/current/Manual/UnityManual.html*), которая может служить справочным руководством по всем аспектам редактора. Документация разбита на два раздела: «Руководство Unity» (*https://docs.unity3d.com/ru/current/Manual/UnityManual.html*), где описывается редактор, а также справочник для разработчиков сценариев «Scripting Reference» (*https://docs.unity3d.com/ScriptReference/index.html*), описывающий все классы, методы и функции программного интерфейса Unity. Это чрезвычайно удобный справочник.
- Официальный форум пользователей Unity (*https://forum.unity.com/*) — центр для дискуссий в сообществе и отличное место, где вы сможете получить помощь.
- Unity Answers (*https://answers.unity.com/index.html*) — официально поддерживаемый форум, где можно задать вопрос и получить ответ. Если у вас есть какой-то конкретный вопрос, поищите сначала ответ на этом форуме.
- В Unity часто проводятся онлайн-тренинги (*https://unity3d.com/ru/learn/live-training*), в каждом из которых один из их специалистов демонстрирует какую-то возможность или даже законченный проект. Если вы не попадете на онлайн-тренинг, вы сможете просмотреть его в записи.
- Наконец, на сайте Unity можно найти множество обучающих материалов (*https://unity3d.com/ru/learn/tutorials*), от вводных руководств для начинающих до более продвинутых и специализированных инструкций.

Надеемся, вам понравилась эта книга. Если вы создадите что-то свое, пусть даже небольшое, или вам что-то не понравилось, мы будем рады любым вашим отзывам. Пишите нам по адресу *unitybook@secretlab.com.au*.

Об авторах

Доктор Джон Мэннинг (Jon Manning) и **доктор Парис Баттфилд-Аддисон (Paris Buttfield-Addison)** — соосновательи студии Secret Lab, где они создают игры и инструменты для разработчиков. Недавно они создали серию игр ABC Play School для iPad, оказали помощь в создании инди-игры Night in the Woods и разработали приложение для детей Qantas Joey Playbox.

В Secret Lab они создали фреймворк YarnSpinner для нарративных игр и написали несколько книг для издательства O'Reilly Media.

Ранее Джон и Парис работали в качестве разработчиков программного обеспечения для мобильных устройств и менеджеров по продукту в Meebo (позднее приобретенной корпорацией Google). Оба имеют степень доктора в области компьютерных технологий.

Связаться с Джоном можно в Твиттере (*@desplesda*) и на его сайте *http://www.desplesda.net*, а с Парисом — в Твиттере (*@parisba*) или на сайте *http://paris.id.au*.

У студии Secret Lab есть свой аккаунт в Твиттере (*@thesecretlab*) и сайт *http://www.secretlab.com.au*.

Выходные данные

На обложке книги «Unity для разработчика. Мобильные мультиплатформенные игры» изображены новогвинейский палочник (*Eurycantha calcarata*) и жук-дровосек (семейство *Cerambycidae*).

Новогвинейский палочник — травоядное, бескрылое насекомое, распространенное в Австралии и Океании. Мужские особи вырастают до 10–12 сантиметров в длину, а женские, более крупные, до 15 сантиметров. В отличие от большинства палочников, живущих на деревьях, представители *Eurycantha calcarata* живут на земле (обычно в тропических лесах), где кормятся ночью, а для защиты от хищников используют маскировку и каталепсию. Они собираются в группы под корой и дремлют в течение дня. В Папуа — Новой Гвинее эти насекомые пользуются большой популярностью как домашние животные, а длинные шипы на задних ногах самцов используются как рыболовные крючки.

Жуки из семейства жуков-дровосеков обладают исключительно длинными антеннами, длина которых часто превышает длину тела. Семейство составляет более 26 000 разновидностей, от дровосека-титана (самый крупный жук в мире, длина тела которого, без учета ног, составляет 32 сантиметра) до рода *Decarthia*, представители трех разновидностей которого имеют длину всего несколько миллиметров. Свое название семейство *Cerambycidae* получило от греческого мифа о Керамбе, рассказывающего о пастухе, которого нимфы превращают в жука.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся на грани исчезновения. Все они важны для нашего мира. Чтобы узнать, как помочь их сохранению, посетите страницу *animals.oreilly.com*.

Изображение для обложки нарисовала Карен Монтгомери (Karen Montgomery) по мотивам гравюр из сборника Джона Джорджа Вуда (J. G. Wood) *Insects Abroad*. Для надписей на обложке использованы шрифты URW Typewriter и Guardian Sans. Основной текст книги набран шрифтом Adobe Minion Pro; заголовки — шрифтом Adobe Myriad Condensed; фрагменты программного кода — шрифтом Ubuntu Mono, созданным Далтоном Маагом (Dalton Maag).