Мультиплатформенная разработка на C#

Unity в действии

Джозеф Хокинг

2-е международное издание

MANNING

ПИТЕР

Unity in Action

Multiplatform Game Development in C#

Joseph Hocking

2nd Edition

MANNING

SHELTER ISLAND

Unity
В ДЕЙСТВИИ
Мультиплатформенная разработка на C#

Джозеф Хокинг

2-е международное
издание

ПИТЕР®

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

2019

ББК 32.973.2-018
УДК 004.42
Х70

Хокинг Джозеф

Х70 Unity в действии. Мультиплатформенная разработка на C#. 2-е межд. изд. — СПб.: Питер, 2019. — 352 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0816-9

Второе издание знаменитого бестселлера «Unity в действии» было полностью переработано, чтобы познакомить вас с новыми подходами и идеями, позволяющими максимально эффективно использовать Unity для разработки игр. Больше внимания уделено проектированию двумерных игр, фундаментальные концепции которых читатель может применить на практике и построить сложный двумерный платформер. Эту книгу можно смело назвать введением в Unity для профессиональных программистов. Джозеф Хокинг дает людям, имеющим опыт разработки, всю необходимую информацию, которая поможет быстро освоить новый инструмент и приступить к созданию новых игр. А учиться лучше всего на конкретных проектах и практических заданиях.

Unity зачастую представляют как набор компонентов, не требующих программирования, что в корне неверно. Для создания успешной игры необходимо многое: великолепная работа художника, программистские навыки, интересная история, увлекательный геймплей, дружная и слаженная работа команды разработчиков. А еще нельзя забывать про безупречную визуализацию и качественную работу на всех платформах — от игровых консолей до мобильных телефонов. Unity объединяет мощный движок, возможности профессионального программирования и творчества дизайнеров, позволяя воплотить в жизнь самые невероятные и амбициозные проекты.

Осваивайте Unity и быстрее приступайте к созданию собственных игр!

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

ББК 32.973.2-018
УДК 004.42

ISBN 978-1617294969 англ.
ISBN 978-5-4461-0816-9

Оглавление

Предисловие

Созданием игр я занялся в 1982 году. Это было непросто: интернета в то время не было, доступные ресурсы ограничивались небольшим количеством по большей части ужасных книг и журналов с интересными, но запутанными фрагментами кода, а игровых движков попросту не существовало! Написание кода для игр было гонкой с огромным количеством препятствий.

Как я завидую тебе, читатель, держащий в руках эту глубокоинформативную книгу! Инструмент под названием Unity многим дал возможность заняться программированием игр. В данном случае достигнут идеальный баланс между мощностью профессионального игрового движка и его доступностью, так ценимой начинающими.

Но доступность в данном случае возникает только при правильном обучении. Мне довелось поработать в цирковой труппе, которой руководил фокусник. Он был столь любезен, что взял меня на работу и помог стать хорошим артистом. «Стоя на сцене, — говорил он, — ты даешь обещание. Ты обещаешь, что не будешь понапрасну тратить время зрителей».

В этой книге мне больше всего нравится практическая часть. Джо Хокинг не тратит ваше время понапрасну и быстро переходит к написанию кода, причем не каких-то бессмысленных фрагментов, а кода, который вы можете понять и использовать в своих целях. Ведь он знает, что вы не просто хотите прочитать эту книгу и проверить, как работают приведенные им примеры, — вашей целью является программирование *собственных игр*.

И благодаря его указаниям вы научитесь это делать быстрее, чем можно было ожидать. Следуйте тропой, которую проложил для вас Джо, но как только ощутите в себе силы, не колеблясь уходите с нее и прокладывайте собственную дорогу. Сразу переходите к наиболее интересным вам темам, экспериментируйте, будьте смелым и храбрым! Почувствовав, что заблудились, вы в любой момент сможете вернуться к тексту книги.

Впрочем, довольно вступительных слов — вас с нетерпением ждет карьера разработчика игр! Запомните этот день, ведь именно он изменит вашу жизнь. Именно сегодня вы начали создавать игры.

Из первого издания
Джесси Шелл (Jesse Schell),
руководитель фирмы Schell Games,
автор книги Art of Game Design
www.amazon.com/Art-Game-Design-Lenses-Second/dp/1466598646/

Введение

Программированием игр я занимаюсь уже много лет, но с Unity познакомился относительно недавно. Разработку игр я осваивал во времена, когда такого инструмента попросту не существовало; его первая версия появилась только в 2005 году. С самого начала это было многообещающее средство разработки, но всеобщее признание оно получило только после выхода нескольких версий. В частности, такие платформы, как iOS и Android (называемые «мобильными»), появились не так давно, и именно они повлияли на рост популярности Unity.

Изначально я смотрел на Unity как на диковинку, интересный инструмент разработки, за развитием которого имеет смысл понаблюдать. В то время я писал игры для настольных компьютеров и сайтов, выполняя проекты для широкого круга клиентов. Я пользовался такими инструментами, как Blitz3D и Flash, великолепно подходящими для программирования, но ограниченными в ряде других аспектов. Но по мере их устаревания я стал искать более совершенные средства разработки игр.

Я начал экспериментировать с Unity версии 3 и полностью переключился на этот инструмент, будучи разработчиком в компании *Synapse Games*. Изначально я занимался интернет-играми, но в итоге перешел на разработку игр для мобильных устройств. А затем все вернулось на круги своя, так как инструмент Unity позволял выполнять развертывание одного и того же базового кода как на мобильных устройствах, так и в интернете!

Я всегда понимал важность передачи знаний, поэтому последние годы занялся преподаванием такой темы, как разработка игр. Во многом это обусловлено примерами, которые подавали мне мои наставники и учителя. Кстати, возможно, об одном из них вы слышали, потому что это потрясающий человек — Рэнди Пауш (Randy Pausch), незадолго до своей кончины выступивший с *Последней общественной лекцией*. Я преподавал в нескольких школах и всегда хотел написать книгу, посвященную разработке игр.

Эта книга во многом напоминает другую, о которой я мечтал во времена, когда только начал осваивать Unity. К многочисленным достоинствам Unity можно отнести огромное количество обучающих ресурсов, но, как правило, они представляют собой наборы отдельных фрагментов (ссылки на сценарии или не связанные друг с другом уроки), соответственно, поиск нужного материала затруднен. Мне же всегда хотелось получить книгу, объединяющую все необходимые знания, представленные в четкой и логически связанной манере, поэтому я написал такую книгу для вас. Моя целевая аудитория — это люди, уже имеющие навыки программирования, но не обладающие опытом работы с Unity и, возможно, никогда не занимавшиеся разработкой игр. Выбор проектов отражает мой опыт наработки навыков и уверенности в себе, который я в свое время получил, выполняя различные заказы в достаточно быстром темпе.

Приступая к изучению разработки игр с помощью Unity, вы начинаете захватывающее приключение. Для меня этот процесс был связан с необходимостью не вешать нос перед лицом многочисленных препятствий. У вас же есть преимущество в виде единого логически согласованного ресурса — этой книги!

Благодарности

Я хотел бы поблагодарить издательство Manning Publications, предоставившее мне возможность написать эту книгу. В этом мне помогли редакторы, с которыми я работал, в том числе Робин де Йон (Robin de Jongh) и особенно Дэн Махари (Dan Maharry). Именно взаимодействие с ними сделало книгу намного лучше. Мои искренние благодарности и многим другим людям, сотрудничавшим со мной в процессе написания и издания.

От пристального внимания рецензентов на всем протяжении работы над книгой она только выиграла. Спасибо Алексу Лукасу (Alex Lucas), Крейгу Хоффману (Craig Hoffman), Дэну Кэйсенджару (Dan Kacenjar), Джошуа Фредерику (Joshua Frederick), Люке Кампобассо (Luca Campobasso), Марку Элстону (Mark Elston), Филиппу Таффету (Philip Taffet), Рене ван ден Бергу (René van den Berg), Серджио Арбео Родригесу (Sergio Arbeo Rodríguez), Шайло Моррису (Shiloh Morris), Виктору М. Пересу (Victor M. Perez), Кристоферу Хаупту (Christopher Haupt), Клаудио Казейро (Claudio Caseiro), Давиду Торрибиа Иниго (David Torribia Iñigo), Дину Цальтасу (Dean Tsaltas), Эрику Вильямсу (Eric Williams), Ники Бакнеру (Nickie Buckner), Робину Дьюсону (Robin Dewson), Сергею Евсикову и Тане Вильке (Tanya Wilke). Отдельная благодарность техническому редактору Скотту Шоси (Scott Chaussee) и техническому корректору Кристоферу Хаупту (Christopher Haupt), а также Рене ван ден Бергу (René van den Berg) и Шайло Моррису (Shiloh Morris), сыгравшим свою роль в выходе второй редакции книги. Также хотелось бы поблагодарить Джесси Шелл (Jesse Schell) за предисловие к книге.

Затем я хотел бы выразить свою признательность людям, помощь которых сделала мой опыт работы с Unity крайне плодотворным. Разумеется, этот список начинается с создавшей Unity (игровой движок) компании Unity Technologies. Чувство глубокой благодарности я испытываю и к сообществу gamedev.stackexchange.com. Этот сайт контроля качества я посещал практически ежедневно, учась у других и отвечая на чужие вопросы. Самый же сильный толчок к работе с Unity я получил от своего руководителя в фирме *Synapse Games* Алекса Рива (Alex Reeve). Мои коллеги показали мне множество приемов и методов, которые я постоянно применяю при написании кода.

Наконец, я хотел бы поблагодарить мою жену Виргинию за ту поддержку, которую она мне оказывала во время написания книги. До начала работы я понятия не имел, насколько подобные проекты переворачивают твою жизнь, влияя на все ее аспекты. Спасибо ей за ее любовь и помощь.

О книге

Второе издание книги *Unity в действии* посвящено программированию игр с помощью Unity. Эту книгу можно считать введением в Unity для опытных программистов. Цель ее крайне проста: научить людей, имеющих опыт программирования, но ни разу не сталкивавшихся с Unity, разрабатывать игры с помощью этого инструмента.

Учить разработке лучше всего на примерах проектов, заставляя обучающихся выполнять практические задания, и именно такой подход используется в данном случае. Темы представлены как этапы построения отдельных игр, и я настоятельно рекомендую вам в процессе знакомства с книгой заняться разработкой этих игр при помощи Unity. Мы рассмотрим ряд проектов, каждому из которых посвящено несколько глав. Бывают книги, целиком посвященные одному крупному проекту, но такой подход исключает возможность чтения с середины, если информация в первых главах покажется вам неинтересной.

В этой книге более строго, чем в большинстве других изданий (особенно предназначенных для начинающих), изложен материал, касающийся программирования. Приложение Unity зачастую представляют как набор компонентов, не требующих программирования, что в корне неверно, так как не дает знаний, без которых невозможно производство коммерчески успешных продуктов. Если вы пока не имеете навыков программирования, советую сначала их приобрести и только после этого приступать к чтению.

Выбор языка программирования не имеет особого значения; все примеры в книге написаны на C#, но они легко переводятся на другие языки. Первая половина книги в изрядной степени посвящена знакомству с новыми понятиями, и первые шаги по разработке игры с помощью Unity намеренно описаны со всей возможной тщательностью, но затем повествование ускоряется, давая читателям возможность выполнять проекты в различных игровых жанрах. Завершает книгу описание развертывания игр на различных платформах, но в целом мы не будем делать упор на этом аспекте, так как Unity не зависит от платформы.

Что касается прочих аспектов разработки игр, излишне широкий охват различных художественных дисциплин привел бы к сокращению объема представленного в книге конкретного материала по Unity и в значительной степени относился бы к внешним по отношению к Unity программам (например, программам создания анимации). Поэтому обсуждение художественных дисциплин сводится к тем аспектам, которые имеют непосредственное отношение к Unity или должны быть известны всем разработчикам игр. Впрочем, одно из приложений посвящено моделированию нестандартных объектов.

Структура книги

Глава 1 знакомит с Unity — межплатформенной средой разработки игр. Вы освоите базовую систему компонентов, лежащую в основе Unity, а также научитесь писать и выполнять базовые сценарии.

В главе 2 мы перейдем к написанию программы, демонстрирующей движение в трехмерном пространстве, попутно рассмотрев такие темы, как ввод с помощью мыши и клавиатуры. Детально объясняется определение положения объектов в трехмерном пространстве и операции их поворота.

В главе 3 мы превратим демонстрационную программу в шутер от первого лица, продемонстрировав метод испускания луча и основы искусственного интеллекта. Испускание луча (мы создаем в сцене линию и смотрим, с чем она пересечется) требуется во всех вариантах игр.

Глава 4 посвящена импорту и созданию игровых ресурсов. Это единственная глава в книге, в которой код не играет центральной роли, так как каждому проекту требуются (базовые) модели и текстуры.

Глава 5 научит вас создавать в Unity двумерные игры. Хотя изначально этот инструмент предназначался исключительно для создания трехмерной графики, сейчас в нем прекрасно поддерживается двумерная графика.

В главе 6 продолжается объяснение принципов создания двумерных игр на примере платформера. В частности, мы реализуем элементы управления, имитацию физической среды и анимацию для персонажа.

Глава 7 знакомит с новейшей GUI-функциональностью в Unity. Пользовательский интерфейс требуется всем играм, а последние версии Unity могут похвастаться улучшенной системой создания пользовательского интерфейса.

В главе 8 мы создадим еще одну программу, демонстрирующую движение в трехмерном пространстве, однако на этот раз с точки зрения стороннего наблюдателя. Реализация элементов управления третьим лицом даст вам представление о ключевых математических операциях в трехмерном пространстве, кроме того, вы научитесь работать с анимированными персонажами.

Глава 9 покажет способы реализации интерактивных устройств и элементов в игре. У игрока будет ряд способов применения этих устройств, в том числе прямым касанием, прикосновением к пусковым устройствам внутри игры или нажатием кнопки контроллера.

Глава 10 учит взаимодействию со Всемирной паутиной. Вы узнаете, как отправить и получить данные с помощью стандартных технологий, таких как HTTP-запросы на получение с сервера XML-данных.

В главе 11 вы научитесь добавлять в игры звук. В Unity замечательно поддерживаются как короткие звуковые эффекты, так и долгие музыкальные фонограммы; оба варианта звукового сопровождения критически важны почти для всех видеоигр.

В главе 12 мы соберем воедино фрагменты из различных глав, чтобы получить в итоге одну игру. Кроме того, вы научитесь программировать элементы управления, манипуляция которыми осуществляется с помощью мыши, и сохранять игру.

Глава 13 демонстрирует процесс создания итогового приложения с его последующим развертыванием на различных платформах, таких как настольные компьютеры, интернет, мобильные устройства и даже виртуальная реальность. Unity обладает поразительной независимостью от конкретной платформы, позволяя создавать любые варианты игр!

Затем идут три приложения с дополнительной информацией о навигации по сцене, внешних инструментах и пакете Blender.

Условные обозначения, требования и доступные для скачивания данные

Весь код в этой книге, как в листингах, так и в представленных фрагментах, набран `вот таким моноширинным шрифтом`, позволяющим отличить код от остального текста. Большинство листингов снабжено примечаниями, отмечающими ключевые понятия, кроме того, в тексте периодически встречаются маркированные списки с дополнительными сведениями по поводу кода. Код отформатирован при помощи переносов строки и аккуратных отступов в соответствии с шириной страницы.

Единственной программой, которая вам потребуется, является Unity; в книге используется версия Unity 2017.1, которая являлась самой последней на момент написания этого текста. В некоторых главах время от времени обсуждаются другие программы, но это всегда дополнительная информация, не оказывающая решающего влияния на изучение основного материала.

ВНИМАНИЕ В Unity-проектах сохраняется информация о том, в какой версии программы они были созданы, и при попытке открыть их в другой версии выводят предупреждение. Если, открыв относящиеся к этой книге материалы, скачанные из интернета, вы увидите такое предупреждение, просто щелкните на кнопке Continue.

Встречающиеся в книге фрагменты кода в общем случае демонстрируют добавления и изменения, вносимые в существующие файлы; если это не первое появление файла с кодом, не следует заменять файл соответствующим листингом. Можно просто скачать рабочий проект целиком, но обучение пойдет намного быстрее, если вы будете набирать все листинги вручную, используя примеры из проекта только для сравнения. Весь код доступен для скачивания на сайте издателя по адресу www.manning.com/books/unity-in-action-second-edition, а также на сайте GitHub (https://github.com/jhocking/uia-2e).

Форум для обсуждения книги

На странице https://forums.manning.com/forums/unity-in-action-second-edition вы найдете информацию о том, как попасть на форум после регистрации, на какую помощь вы можете рассчитывать, а также о правилах поведения на форуме.

Издательство Manning взяло на себя обязательство по предоставлению места, где читатели могут конструктивно пообщаться как друг с другом, так и с автором книги. Но оно не может гарантировать присутствия на форуме автора, участие которого в обсуждениях является добровольным (и неоплачиваемым). Мы надеемся, что вы

будете задавать автору по-настоящему трудные вопросы, чтобы его интерес к общению не угас! Как форум, так и архивы предшествующих обсуждений будут доступны на сайте издательства до тех пор, пока книга находится в продаже.

Об авторе

Джозеф Хокинг живет в Чикаго и занимается разработкой программного обеспечения для интерактивных сред. Он работает в фирме InContext Solutions. Первое издание данной книги было написано в период его работы в компании Synapse Games. Кроме того, он преподавал разработку игр в Университете Иллинойса, в Чикагском институте искусств и в колледже Колумбия. Его сайт www.newarteest.com.

Иллюстрация на обложке

Книгу *Unity в действии, 2-е издание* украшает иллюстрация, подписанная как «Платье церемониймейстера Великого господина». Великим господином называли султана Османской империи. Иллюстрация взята из четырехтомника «*Коллекция костюмов разных народов, античных и современных*» Томаса Джефферис, опубликованного в Лондоне между 1757 и 1772 годами. На титульной странице указано, что это выполненные вручную каллиграфические цветные гравюры, обработанные гуммиарабиком. Томас Джефферис (1719–1771) носил звание «Географ короля Георга III». Этот английский картограф был ведущим поставщиком карт того времени. Он гравировал и печатал множество карт для нужд правительства и других официальных органов. Выпустил он и множество коммерческих карт и атласов, в частности карт Северной Америки. Он интересовался и одеждой народов, населяющих разные земли, и собрал блестящую коллекцию различных платьев, описав ее в четырех томах.

В конце XVIII века очарование чужих земель и путешествий ради удовольствия было относительно новым явлением, и подобные коллекции были весьма популярны, так как позволяли получить представление о том, как выглядят жители других стран. Разнообразие собранных Джефферисом рисунков свидетельствует о яркой индивидуальности и уникальности народов мира, живших более 200 лет назад. С тех пор стиль одежды сильно изменился, исчезли характеризовавшие различные области и страны различия. Сейчас трудно распознать по одежде даже жителей различных континентов. Если взглянуть на это с оптимистической точки зрения, культурное и внешнее многообразие было принесено в жертву более насыщенной личной жизни, более неординарной и интересной интеллектуальной и технической деятельности.

Сейчас, когда одну техническую книгу сложно отличить от другой, издательство Manning украшает обложки своих книг изображениями, порожденными богатой вариативностью жизненного уклада народов двухвековой давности, продлевая жизнь рисункам Джефферис.

Часть I
ПЕРВЫЕ ШАГИ

Настало время знакомства с Unity. Ничего страшного, если вы ничего не знаете о нем! Я подробно опишу Unity и научу основам программирования игр с его помощью. В финале вы получите инструкцию по разработке простой игры. Первый проект познакомит вас с различными техниками, давая представление о том, как выглядит рабочий процесс.

Итак, начнем!

1 Знакомство с Unity

- ✓ Почему следует выбрать Unity.
- ✓ Как работает редактор Unity.
- ✓ Как выглядит программирование в Unity.
- ✓ В чем разница языков C# и JavaScript.

Возможно, вы, как и я, мысленно давным-давно разрабатываете видеоигру. Но перейти из лагеря игроков в лагерь создателей игр непросто. За последние годы появилось множество инструментов разработки игр, и мы обсудим один из самых новых и мощных представителей этого семейства. Приложение Unity — это профессиональный игровой движок, с помощью которого создаются видеоигры для различных платформ. Им ежедневно пользуются опытные разработчики, и вместе с тем это один из наиболее доступных инструментов для новичков. Еще недавно решение научиться программированию игр (особенно трехмерных) наталкивалось на множество серьезных препятствий, но приложение Unity сильно облегчило жизнь новичкам.

Раз вы читаете эту книгу, значит, интересуетесь компьютерными технологиями и либо разрабатывали игры с помощью других инструментов, либо создавали программное обеспечение других типов, например приложения для рабочего стола или веб-сайты. Создание видеоигр в своей основе не отличается от написания любого другого ПО; по большей части различия проявляются в количественной плоскости. К примеру, игра намного интерактивнее большинства веб-сайтов, а значит, потребуется код другого типа. Но в обоих случаях будут применяться сходные навыки и процессы. Тем, кто преодолел первое препятствие на пути к карьере разработчика игр, то есть изучил основы создания ПО, нужно сделать следующий шаг: выбрать инструмент и приобрести специализированные навыки программирования. В этом смысле приложение Unity представляет собой замечательную среду разработки.

ПРЕДУПРЕЖДЕНИЕ ПО ПОВОДУ ТЕРМИНОЛОГИИ

Эта книга посвящена программированию в Unity и будет интересна в основном кодерам. Существует множество ресурсов, где обсуждаются другие аспекты разработки игр, как в целом, так и в приложении Unity, но в нашем случае основной упор делается именно на программирование.

Хотелось бы подчеркнуть, что в контексте создания игр слово *разработчик* имеет несколько непривычный для нас смысл. Например, в области веб-разработки это синоним слова *программист*, в то время как в данном случае *разработчиком* называется любой человек, работающий над созданием игры, а *программистом* называется разработчик, выполняющий конкретные обязанности. Поэтому, к примеру, художники и дизайнеры будут также называться разработчиками, но в книге рассматривается только работа программистов.

Первым делом зайдите на сайт www.unity3d.com и скачайте среду разработки. Я пользовался версией Unity 2017.1, то есть наиболее новой на момент написания книги. Адрес сайта подчеркивает, что изначально приложение Unity предназначалось для создания трехмерных игр. Их поддержка по-прежнему остается приоритетной, но теперь в Unity замечательно разрабатываются и двумерные игры. В платной версии программы доступны расширенные функциональные возможности, но базовая версия распространяется бесплатно. Все приведенные в книге примеры прекрасно работают в бесплатной версии, так что покупать Unity Pro не потребуется. Платная версия отличается от бесплатной уже упоминавшимися расширенными функциональными возможностями (рассмотрение которых выходит за рамки данной книги) и коммерческими условиями лицензирования.

1.1. Достоинства Unity

Внимательно рассмотрим данное в начале главы определение: приложение Unity — это профессиональный игровой движок, позволяющий создавать видеоигры для различных платформ. Это достаточно точный ответ на вопрос «Что такое Unity?». Но что конкретно он означает? И чем примечателен инструмент Unity?

1.1.1. Преимущества Unity

Любой игровой движок предоставляет множество функциональных возможностей, которые задействуются в различных играх. Реализованная на конкретном движке игра получает все функциональные возможности, к которым добавляются ее собственные игровые ресурсы и код игрового сценария. Приложение Unity предлагает моделирование физических сред, карты нормалей, прегражденние окружающего света в экранном пространстве (Screen Space Ambient Occlusion, SSAO), динамические тени... список можно продолжать долго. Подобные наборы функциональных возможностей есть во многих игровых движках, но Unity обладает двумя основными преимуществами над другими передовыми инструментами разработки игр. Это крайне производительный визуальный рабочий процесс и сильная межплатформенная поддержка.

Визуальный рабочий процесс — достаточно уникальная вещь, выделяющая Unity из большинства сред разработки игр. Альтернативные инструменты разработки зачастую представляют собой набор разрозненных фрагментов, требующих контроля, а в некоторых случаях библиотеки, для работы с которой нужно настраивать собственную

интегрированную среду разработки (Integrated Development Environment, IDE), цепочку сборки и прочее в этом роде. В Unity же рабочий процесс привязан к тщательно продуманному визуальному редактору. Именно в нем вы будете компоновать сцены будущей игры, связывая игровые ресурсы и код в интерактивные объекты. Он позволяет быстро и рационально создавать профессиональные игры, обеспечивая невиданную продуктивность разработчиков и предоставляя в их распоряжение исчерпывающий список самых современных технологий в области видеоигр.

ПРИМЕЧАНИЕ В других инструментах, оснащенных центральным визуальным редактором, возможность написания сценариев зачастую поддерживается ограниченно и недостаточно гибко, но приложение Unity лишено этого недостатка. Все, что создается для Unity, в конечном счете проходит через визуальный редактор, но основной интерфейс при этом включает в себя множество связанных проектов с запускаемым в игровом движке пользовательским кодом. Это своего рода аналог связывания классов в настройках проекта для таких интегрированных сред разработки, как Visual Studio или Eclipse. Поэтому опытным программистам не стоит пренебрегать Unity, считая это приложение чисто визуальным инструментом создания игр с ограниченной возможностью программирования!

Редактор особенно удобен для процессов с последовательным улучшением, например циклов создания прототипов или тестирования. Даже после запуска игры остается возможность модифицировать в нем объекты и двигать элементы сцены. Настраивать можно и сам редактор. Для этого применяются сценарии, добавляющие к интерфейсу новые функциональные особенности и элементы меню.

Дополнением к производительности, которую обеспечивает редактор, служит сильная межплатформенная поддержка набора инструментов Unity. В данном случае это словосочетание подразумевает не только места развертывания (игру можно развернуть на персональном компьютере, в интернете, на мобильном устройстве или на консоли), но и инструменты разработки (игры создаются на машинах, работающих под управлением как Windows, так и Mac OS). Эта независимость от платформы явилась результатом того, что изначально приложение Unity предназначалось исключительно для компьютеров Mac, а позднее было перенесено на машины с операционными системами семейства Windows. Первая версия появилась в 2005 году, а к настоящему моменту вышли уже пять основных версий (с множеством небольших, но частых обновлений). Изначально разработка и развертка поддерживались только для машин Mac, но через несколько месяцев вышло обновление, позволяющее работать и на машинах с Windows. В следующих версиях добавлялись все новые платформы развертывания, например межплатформенный веб-плеер в 2006-м, iPhone в 2008-м, Android в 2010-м и даже такие игровые консоли, как Xbox и PlayStation. Позднее появилась возможность развертки в WebGL — новом фреймворке для трехмерной графики в веб-браузерах. Немногие игровые движки поддерживают такое количество целевых платформ развертывания, и ни в одном из них развертка на разных платформах не осуществляется настолько просто.

Дополнением к этим основным достоинствам идет и третье, менее бросающееся в глаза преимущество в виде модульной системы компонентов, которая используется для конструирования игровых объектов. «*Компоненты*» в такой системе представляют собой комбинируемые пакеты функциональных элементов, поэтому объекты создаются

как наборы компонентов, а не как жесткая иерархия классов. В результате получается альтернативный (и обычно более гибкий) подход к объектно-ориентированному программированию, в котором игровые объекты создаются путем объединения, а не наследования. Оба подхода схематично показаны на рис. 1.1.

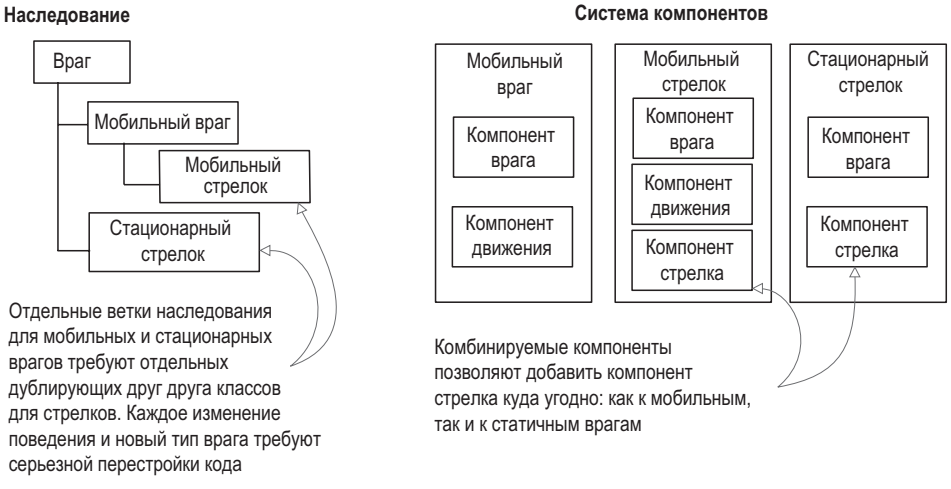


Рис. 1.1. Сравнение наследования с компонентной системой

В компонентной системе объект существует в горизонтальной иерархии, поэтому объекты состоят из наборов компонентов, а не из иерархической структуры с наследованием, в которой разные объекты оказываются на разных ветках дерева. Такая компоновка облегчает создание прототипов, потому что взять нужный набор компонентов куда быстрее и проще, чем перестраивать цепочку наследования при изменении каждого объекта.

Разумеется, ничто не мешает написать код, реализующий вашу собственную компонентную систему, но в Unity уже существует вполне надежный вариант такой системы, органично встроенный в визуальный редактор. Эта система дает возможность не только управлять компонентами программным образом, но и соединять и разрывать связи между ними в редакторе. Разумеется, возможности не ограничиваются составлением объектов из готовых деталей; в своем коде вы можете воспользоваться наследованием и всеми наработанными на его базе шаблонами проектирования.

1.1.2. Недостатки, о которых нужно знать

Многочисленные преимущества приложения Unity превращают его в замечательное средство разработки игр, но было бы упущением не упомянуть о его недостатках. В частности, затруднения может вызвать нетипичное сочетание визуального редактора со сложным кодом, несмотря на всю его эффективность в рамках компонентной системы Unity. В сложных сценах из виду могут потеряться присоединенные компоненты. Разумеется, существует функция поиска, но она могла бы быть и более надежной; порой все равно возникают ситуации, когда для поиска связанных сценариев приходится

вручную просматривать все элементы сцены. Такое случается нечасто, тем не менее подобной кропотливой и трудоемкой работы хотелось бы совсем избежать.

Опытных программистов может обескуражить и то, что Unity не поддерживает ссылки на внешние библиотеки кода. Все библиотеки, которые вы планируете задействовать, следует вручную копировать в проект, вместо того чтобы дать ссылку на одну папку общего доступа. Отсутствие единой папки с библиотеками затрудняет коллективное использование функционала разными проектами. Это неудобство можно обойти, рационально воспользовавшись системами контроля версий, но готовое решение данной проблемы в Unity отсутствует.

ПРИМЕЧАНИЕ Раньше существенным недостатком была сложность работы с системами контроля версий (такими, как Subversion, Git и Mercurial), но в более поздних версиях Unity все прекрасно работает. В Сети можно наткнуться на устаревшие сведения, в которых утверждается, что Unity не работает с системами контроля версий, но на более новых ресурсах есть указания, какие папки и файлы проекта следует поместить в репозиторий. Начать лучше всего с чтения документации https://docs.unity3d.com/ru/current/Manual/ExternalVersionControlSystemSupport.html. Или обратите внимание на файл .gitignore в хранилище GitHub (http://mng.bz/g7nl).

Третий недостаток связан с шаблонами создания экземпляров (prefabs). Эта концепция детально объясняется в главе 3, а пока достаточно понимать, что шаблоны экземпляров предлагают гибкий подход к визуальному созданию интерактивных объектов. Эта крайне мощная концепция существует исключительно в Unity (и, естественно, она связана с компонентной системой Unity), но редактирование таких шаблонов порой оказывается на удивление труднореализуемым. Учитывая их практичность и важность для работы в Unity, я надеюсь, что в будущих версиях способ их редактирования будет усовершенствован.

1.1.3. Примеры игр, созданных в Unity

Итак, вы познакомились с сильными и слабыми сторонами Unity, но, возможно, пока до конца не уверены в том, что этот инструмент позволяет получать первоклассные результаты. Зайдите в галерею Unity на странице http://unity3d.com/ru/showcase/gallery и полюбуйтесь постоянно обновляемым списком сотен игр и симуляций, созданных этим инструментом. Я, в свою очередь, приведу небольшой список игр, демонстрирующих разные жанры и платформы развертывания.

Игры для рабочего стола (WINDOWS, MAC, LINUX)

Редактор работает на одной платформе с приложением, поэтому зачастую проще всего развернуть игру на машине с операционной системой семейства Windows или Mac. Примеры игр для рабочего стола в различных жанрах:

- *Guns of Icarus Online* (рис. 1.2) — игра от первого лица в жанре многопользовательского симулятора боевого дирижабля, созданная независимым разработчиком Muse Games.
- *Gone Home* (рис. 1.3) — квест от первого лица, разработанный независимой студией Fullbright Company.

Рис. 1.2. Guns of Icarus Online

Рис. 1.3. Gone Home

Игры для мобильных устройств (IOS, ANDROID)

Приложение Unity также позволяет развертывать игры на мобильных платформах, таких как iOS (смартфонах iPhone и планшетах iPad) и Android (смартфонах и планшетах). Примеры мобильных игр в различных жанрах:

- *Lara Croft GO* (рис. 1.4) — трехмерная головоломка, разработанная компанией Square Enix.
- *INKS* (рис. 1.5) — двумерная головоломка от студии State of Play.

Рис. 1.4. Lara Croft GO

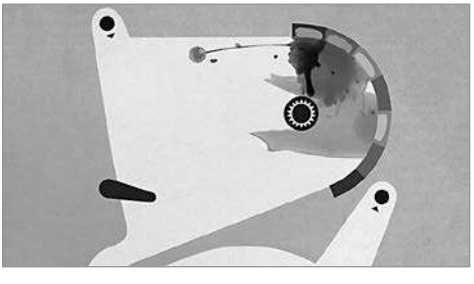


Рис. 1.5. INKS

- *Tyrant Unleashed* (рис. 1.6) — коллекционная карточная игра, созданная студией Synapse Games.

Рис. 1.6. Tyrant Unleashed

Игры для консолей (PlayStation, Xbox, Switch)

С помощью Unity можно разворачивать игры на игровых консолях, хотя для этого требуется лицензия от Sony, Microsoft или Nintendo. Межплатформенная развертка позволяет избежать лицензионных ограничений, поэтому консольные игры часто доступны и на обычных компьютерах. Примеры таких игр в различных жанрах:

- *Yooka-Laylee* (рис. 1.7) — трехмерная игра в жанре платформер от британской студии Playtonic Games.
- *Shadow Tactics* (рис. 1.8) — симулятор военной тактики в режиме реального времени, созданный немецкой студией Mimimi Productions.

Рис. 1.7. Yooka-Laylee

Рис. 1.8. Shadow Tactics

Как видно из этих примеров, Unity позволяет создавать игры хорошего товарного качества. Но, несмотря на значительные преимущества Unity над другими инструментами разработки, новички могут недооценивать важность программирования. Зачастую Unity описывают как набор готовых программных компонентов, для использования которых программирование вообще не требуется. Но это неверная точка зрения, не дающая представления о том, как создать коммерчески успешный продукт. Нет, интересный прототип можно создать и из предустановленных компонентов, пользуясь одной только мышью, но перейти от прототипа к игре, готовой увидеть свет, без программирования не удастся.

1.2. Работа с Unity

Выше уже много было сказано о том, как выгоден встроенный в Unity визуальный редактор с точки зрения производительности, пришло время познакомиться с его интерфейсом и узнать, как он работает. Если на вашем компьютере еще нет приложения, скачайте его со страницы http://unity3d.com/ru/get-unity и установите (обязательно установите флажок Example Project, если установщик его сбросит). После этого запустите Unity, и мы приступим к изучению интерфейса.

Для наглядности откройте входящий в комплект поставки проект; при установке новой копии этот проект должен предлагаться автоматически, но можно выбрать в меню File команду Open Project и открыть его вручную. Он находится в пользовательской папке общего доступа, адрес которой в операционных системах семейства Windows выглядит

примерно так: C:\Users\Public\Documents\Unity Projects\, а в Mac OS — примерно так: Users/Shared/Unity/. Если заодно потребуется открыть пример сцены, дважды щелкните на файле Car (на рис. 1.9 видно, что такие файлы в Unity обозначаются символом куба). В диспетчере файлов, расположенном в нижней части редактора, значок этого файла будет находиться по адресу SampleScenes/Scenes/. Вы должны получить экран, показанный на рис. 1.9.

Рис. 1.9. Части интерфейса Unity

Интерфейс Unity состоит из следующих частей: вкладка Scene, вкладка Game, панель инструментов, вкладка Hierarchy, панель Inspector, вкладки Project и Console. Каждая часть имеет собственное предназначение, но все они играют важную роль в цикле создания игры:

- Просмотр файлов выполняется на вкладке Project.
- Объекты трехмерной сцены просматриваются на вкладке Scene.
- Панель инструментов предоставляет элементы управления сценой.
- На вкладке Hierarchy путем перетаскивания можно менять связь между объектами.
- Панель Inspector отображает сведения о выделенных объектах и показывает связанный с ними код.
- Тестирование результатов осуществляется на вкладке Game, а сообщения об ошибках появляются на вкладке Console.

Это компоновка по умолчанию; все доступные представления помещены на вкладки, их можно перемещать, фиксировать в разных частях экрана, можно менять их размер. Чуть позже вы поэкспериментируете с выбором компоновки, а пока мы знакомимся с назначением каждого элемента интерфейса, вариант по умолчанию является оптимальным.

1.2.1. Вкладка Scene, вкладка Game и панель инструментов

Наиболее заметная часть интерфейса — расположенная в центре вкладка Scene. Именно здесь мы можем видеть, как выглядит мир игры, и перемещать объекты по сцене. Сеточные объекты в сцены, как им и положено, представлены в виде сеток. Можно наблюдать и ряд других объектов, представленных различными значками и цветными линиями. Это камеры, источники света, источники звука, области столкновений и т. п. Разумеется, эта картинка отличается от того, что будет показываться в процессе игры, — рассматривать сцену можно, не ограничиваясь игровым представлением.

ОПРЕДЕЛЕНИЕ *Сеточным объектом* (mesh object) называется объект, видимый в трехмерном пространстве. Он создается из набора соединенных друг с другом линий и форм, формирующих *сетку*.

Игровое представление отображается на вкладке Game, расположенной рядом с вкладкой Scene (переход с вкладки на вкладку осуществляется в верхнем левом углу области отображения). Интерфейс содержит и другие элементы, сконструированные подобным образом; для изменения отображаемого ими содержимого достаточно перейти на другую вкладку. После запуска игры начинает отображаться игровое представление. При этом переходить на вкладку Game не нужно, переключение выполняется автоматически.

СОВЕТ В режиме воспроизведения игры можно вернуться на вкладку Scene для просмотра объектов. Это крайне полезная возможность, позволяющая понять, как игра выглядит изнутри и что в ней происходит. В большинстве игровых движков подобный инструмент отладки отсутствует.

Для запуска игры достаточно нажать кнопку Play над вкладкой Scene. Вся верхняя часть интерфейса занята так называемой панелью инструментов (Toolbar), и кнопка Play находится как раз в центре этой панели. На рис. 1.10 из всего интерфейса редактора для наглядности оставлена только панель инструментов с расположенными под ней вкладками Scene/Game.

На левой стороне панели инструментов находятся кнопки навигации и преобразования объектов. Они позволяют рассматривать сцену с разных сторон и перемещать объекты. Попрактикуйтесь в их применении, так как основные действия, выполняемые в визуальном редакторе Unity, — это просмотр сцен и перемещение объектов (они настолько важны, что им будет посвящен отдельный раздел). Правую сторону панели инструментов занимают раскрывающиеся меню с перечнем компоновок и слоев. Я уже упоминал о гибкости и возможностях настройки интерфейса Unity. Поэтому в нем и существует меню Layouts, позволяющее переходить от одного варианта компоновки к другому. Меню Layers относится к расширенным функциональным возможностям, которые мы пока рассматривать не будем (о слоях пойдет речь в следующих главах).

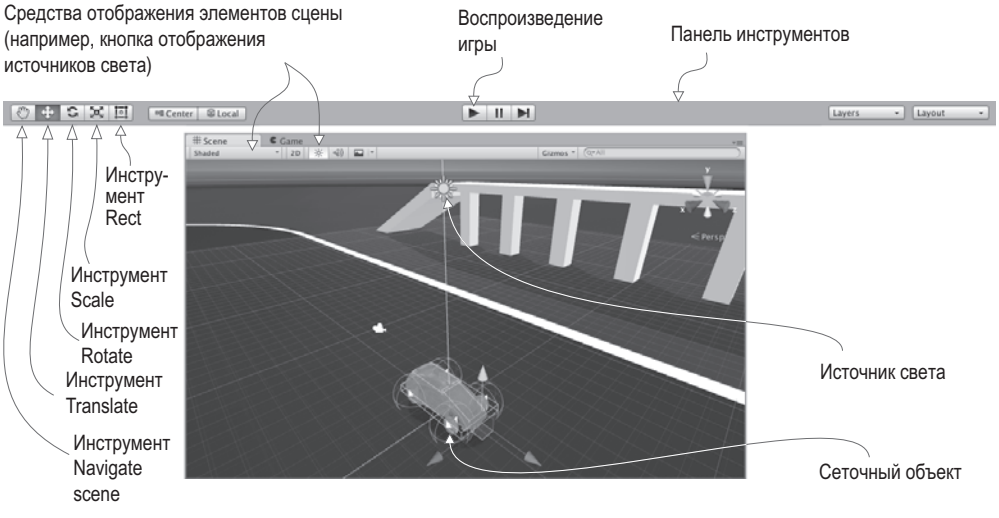


Рис. 1.10. Усеченный вариант редактора, демонстрирующий только панель инструментов и вкладки Scene и Game

1.2.2. Работа с мышью и клавиатурой

Навигация в сценах осуществляется с помощью мыши и набора клавиш-модификаторов, влияющих на результат манипуляций с мышью. Главные операции — это перемещение (move), облет (orbit) и масштабирование (zoom). Действия, необходимые для совершения каждой из этих операций, зависят от типа мыши, и их описание вы найдете в приложении А. Но в основном они сводятся к щелчкам и перетаскиванию при нажатых и удерживаемых клавишах Ctrl и Alt (или Option на компьютерах Mac). Потренируйтесь в манипулировании объектами сцены, чтобы понять, как выполняются перемещение, облет и масштабирование.

СОВЕТ Хотя работать в Unity можно и с двухкнопочной мышью, рекомендую приобрести мышь с тремя кнопками (не сомневайтесь, в Mac OS X она тоже работает).

Преобразование объектов также осуществляется посредством трех вышеупомянутых операций. Более того, каждому типу навигации соответствует собственное преобразование: перенос (translate), поворот (rotate) и изменение размеров (scale). Рисунок 1.11 демонстрирует эти преобразования на примере куба.

Выделенный объект сцены можно двигать (или, если брать более точный термин, *переносить*), вращать и указывать его размер. Если рассмотреть процесс навигации в сцене с этой точки зрения, перемещение будет соответствовать переносу камеры, облет — повороту камеры, а масштабирование — изменению размеров камеры. Переход между этими операциями осуществляется не только кнопками панели инструментов, но и нажатием клавиш W, E и R. При входе в режим преобразования у выделенного объекта появляются цветные стрелки или окружности. Это габаритный контейнер преобразования (transform gizmo), перетаскивание которого меняет вид объекта.

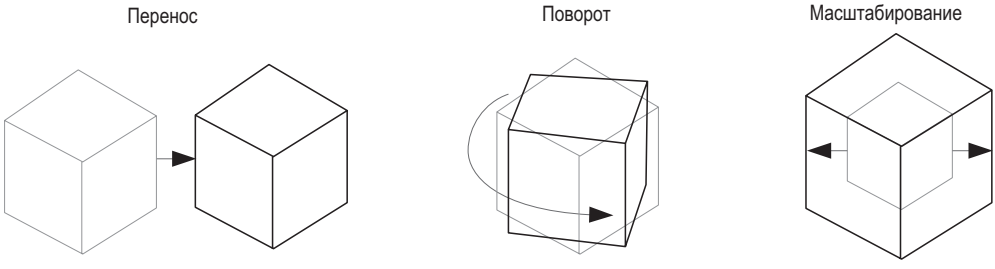


Рис. 1.11. Три варианта преобразования: перенос, поворот и масштабирование (более светлые линии обозначают исходное состояние объекта)

Рядом с кнопками преобразований находится кнопка инструмента Rect, позволяющая перейти к работе с двумерной графикой и объединяющая в себе операции переноса, поворота и масштабирования. В трехмерном пространстве за каждую из этих операций отвечает свой инструмент, но в двумерном пространстве они объединены, так как у нас становится меньше на одно измерение. В Unity существует также набор клавиатурных комбинаций для ускорения выполнения различных операций. Они перечислены в приложении А. А теперь посмотрим на остальные фрагменты интерфейса.

1.2.3. Вкладка Hierarchy и панель Inspector

На левой стороне экрана примостилась вкладка Hierarchy, в то время как правую заняла панель Inspector (рис. 1.12). Вкладка Hierarchy содержит список всех объектов сцены в виде древовидной структуры, ветки которой вложены друг в друга в соответствии с иерархическими связями между этими объектами. По сути, эта вкладка позволяет выделять объекты по именам, избавляя от необходимости искать в сцене нужный объект, чтобы выделить его щелчком. Иерархические связи объединяют объекты в группы. Визуально это оформлено в виде папок и дает возможность за одну операцию переместить целую группу объектов.

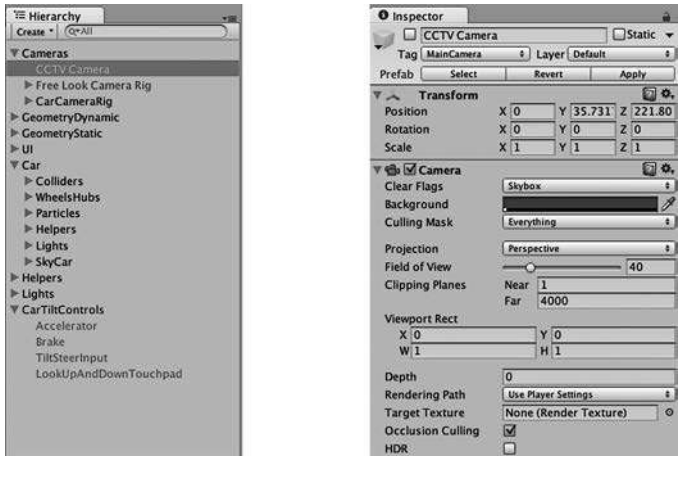


Рис. 1.12. Вкладка Hierarchy и панель Inspector

На панели Inspector отображаются данные выделенного объекта. Выделите любой объект и обратите внимание, как изменится вид панели Inspector. Отображаемая информация по большей части представляет собой список компонентов, причем вы можете как добавлять к объектам новые компоненты, так и удалять существующие. Все игровые объекты содержат по крайней мере один компонент — Transform, поэтому на панели Inspector всегда будут отображаться хотя бы сведения о положении и ориентации выделенного объекта. У многих объектов вы увидите целые списки компонентов, в том числе связанные с этими объектами сценарии.

1.2.4. Вкладки Project и Console

Нижнюю часть экрана занимают показанные на рис. 1.13 вкладки Project и Console. В данном случае мы видим пример такой же организации элементов интерфейса, как и у вкладок Scene и View, что легко позволяет переходить от одного представления к другому. На вкладке Project отображаются все ресурсы проекта (графические фрагменты, код и т. п.). В левой части этой вкладки находится список папок проекта; при выделении папки справа появляются находящиеся в ней файлы. По сути, это такой же список, как и на вкладке Hierarchy, но если эта вкладка показывает перечень объектов сцены, то на вкладке Project представлены файлы, не включенные ни в одну конкретную сцену (в том числе и сами файлы сцен — сохраненная сцена появляется на вкладке Project!).

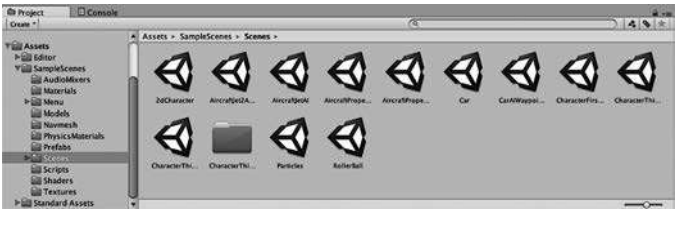


Рис. 1.13. Вкладки Project и Console

СОВЕТ Вкладка Project дублирует содержимое папки Assets на диске, но в общем случае удалять или перемещать файлы непосредственно в этой папке не рекомендуется. Выполняйте эти операции на вкладке Project, а о синхронизации с папкой Assets позаботится Unity.

Вкладка Console представляет собой место вывода связанных с кодом сообщений. Иногда это намеренно вставленные в программу сообщения отладчика, иногда Unity посылает сообщения об ошибке, обнаружив неполадку в написанном сценарии.

1.3. Подготовка к программированию в Unity

Посмотрим, как в Unity выглядит процесс программирования. Компоновка игровых ресурсов происходит в визуальном редакторе, но для обеспечения интерактивности игры требуется управляющий ресурсами код. В приложении Unity поддерживается ряд языков программирования, в частности JavaScript и C#. Каждый вариант имеет свои достоинства и недостатки, но в этой книге все примеры даны на языке C#.

ПОЧЕМУ C#, А НЕ JAVASCRIPT?

Весь код в книге написан на языке C#, так как он имеет ряд преимуществ над языком JavaScript и куда меньше недостатков, особенно с точки зрения профессионального разработчика (именно этим языком я пользуюсь для работы).

Одним из преимуществ является строгая типизация языка C#, отсутствующая в JavaScript. Среди опытных программистов существуют разные точки зрения по поводу того, является ли динамическая проверка типов оптимальным подходом, например, к веб-разработке, но при написании программ для определенных игровых платформ (таких, как iOS) зачастую выгодна, а порой даже требуется статическая типизация. В Unity даже добавлена директива `#pragma`, принудительно обеспечивающая статическую проверку типов в JavaScript. Технически такое вполне допустимо, но при этом нарушается один из основных принципов функционирования JavaScript. Поэтому лучше изначально выбрать язык со строгой типизацией.

Это всего лишь один пример того, чем отличается язык JavaScript в Unity. Во многом он напоминает JavaScript в веб-браузерах, но в его функционировании есть ряд зависящих от контекста отличий. Многие разработчики называют версию для Unity именем UnityScript, которое указывает на сходство, но одновременно и на отличие от JavaScript. Именно это состояние «аналогичный, но отличающийся» становится проблемой как при попытках применить общие знания языка JavaScript в контексте Unity, так и при попытках применять на стороне знания, полученные в процессе работы в Unity.

По этой причине в Unity понемногу прекращается поддержка JavaScript/UnityScript. Вот перевод соответствующей статьи из блога: https://devtribe.ru/p/unity/unityscripts-is-depricating. Оригинал статьи находится по адресу http://mng.bz/B9au.

Попробуем написать и запустить код. Откройте Unity и щелкните на ссылке New в стандартном окне. Или выберите в меню File команду New Project, если приложение Unity уже запущено. Откроется окно диалога New Project, предназначенное для создания нового проекта. Укажите имя проекта и место, куда бы вы хотели его сохранить. Проект Unity представляет собой обычную папку с файлами различных ресурсов и настроек, поэтому его можно сохранять где угодно. Щелкните на кнопке Create Project, и Unity ненадолго исчезнет, чтобы создать папку проекта.

ВНИМАНИЕ Проекты Unity запоминают, в какой версии программы они создавались, и при попытке открыть их в другой версии появляется предупреждение. Иногда на это можно не обращать внимания (к примеру, если такое предупреждение появится при открытии файлов с примерами к данной книге, просто игнорируйте его), но бывают и ситуации, когда перед открытием проекта имеет смысл сделать его резервную копию.

ВНИМАНИЕ При открытии файлов с примерами к книге может появиться и вот такое сообщение: Rebuilding Library because the asset database could not be found! Оно связано с папкой Library открываемого проекта; эта папка содержит файлы, генерируемые Unity и используемые в процессе работы, но редактор не считает нужным загружать их по умолчанию.

Когда Unity появится снова, вы увидите пустую сцену. Давайте посмотрим, как в Unity происходит запуск программ.

1.3.1. Запуск кода: компоненты сценария

Для запуска кода в Unity первым делом нужно связать файлы с кодом с каким-либо объектом сцены. Это часть компонентной системы, о которой говорилось выше; игровые объекты создаются как наборы компонентов, и в каждый такой набор может входить исполняемый сценарий.

ПРИМЕЧАНИЕ В терминологии Unity файлы с кодом принято называть сценариями, используя для определения этого слова значение, чаще всего применяемое к ситуации, когда в браузере запускается JavaScript: код выполняется внутри игрового движка Unity, а не превращается после компиляции в самостоятельный исполняемый файл. Но тут легко запутаться, так как многие определяют данный термин по-другому. К примеру, сценариями иногда называют автономные служебные программы. Сценарии в Unity больше напоминают индивидуальные классы ООП. Присоединенные к объектам сцены сценарии являются экземплярами объектов.

По этому описанию вы, наверное, уже догадались, что сценарии в Unity представляют собой компоненты. Но следует отметить, что в эту группу попадают не все сценарии, а только наследующие от класса `MonoBehaviour`, базового класса компонентов-сценариев. Этот класс определяет способ присоединения компонентов к игровым объектам. Наследование от этого класса дает пару автоматически запускаемых методов (показанных в листинге 1.1), которые вы можете переопределить. Это метод `Start()`, вызываемый при активации объекта (она, как правило, наступает после загрузки содержащего объект уровня), и вызываемый в каждом кадре метод `Update()`. Соответственно, код будет запущен, если вставить его в эти предустановленные методы.

ОПРЕДЕЛЕНИЕ *Кадром* (frame) называется один прогон зацикленного игрового кода. Практически все видеоигры (не только в Unity, но и вообще) строятся вокруг основного игрового цикла. То есть код этого цикла выполняется до тех пор, пока запущена игра. На каждой итерации цикла происходит прорисовка экрана, откуда, собственно, и возник термин *кадр* (по аналогии с набором статичных кадров в фильме).

Листинг 1.1. Шаблон кода для базового компонента сценария

```
using UnityEngine;  ← Добавляет пространства имен для классов Unity и Mono.
using System.Collections;
using System.Collections.Generic;

public class HelloWorld : MonoBehaviour {  ← Синтаксис для задания наследования.
    void Start() {
        // однократное действие  ← Сюда добавляется однократно выполняемый код.
    }

    void Update() {
        // действие, выполняемое в каждом кадре  ← Сюда добавляется код,
    }                                                запускаемый в каждом кадре.
}
```

Именно так выглядит файл в момент создания нового сценария на языке C#: минимальный шаблонный код, определяющий корректный компонент Unity. Существует шаблон сценария, скрытый в недрах Unity. При создании нового сценария приложение копирует этот шаблон и переименовывает класс в соответствии с именем файла (в моем случае это HelloWorld.cs). Добавляются и пустые оболочки методов `Start()` и `Update()`, так как именно из этих методов чаще всего вызывается пользовательский код.

Чтобы создать сценарий, откройте меню Assets, наведите указатель мыши на строчку Create и выберите на открывшейся дополнительной панели команду C# Script. Обратите внимание, что как в меню Assets, так и в меню GameObjects есть варианты команды

Create, но это разные вещи. Альтернативным способом доступа к нужному нам меню является щелчок правой кнопкой мыши в произвольной точке вкладки Project. Введите имя нового сценария, например HelloWorld. Чуть позже, на рис. 1.15 вы увидите, что можно перетащить файл сценария мышью на произвольный объект сцены. Дважды щелкните на значке сценария, и он автоматически откроется в программе MonoDevelop, о которой мы поговорим ниже.

1.3.2. Программа MonoDevelop, межплатформенная IDE

Программирование осуществляется не внутри Unity, код существует в виде отдельных файлов, местоположение которых вы сообщаете приложению. Файлы сценариев могут создаваться в самом приложении, но в любом случае потребуется текстовый редактор или IDE, где для изначально пустых файлов будет писаться код. В комплекте с Unity поставляется приложение MonoDevelop как межплатформенная интегрированная среда разработки (IDE) для языка C# с открытым исходным кодом (оно показано на рис. 1.14). Более подробную информацию можно получить на сайте www.monodevelop.com, но пользоваться при этом следует версией, идущей в комплекте с Unity, а не скачанным с этого сайта приложением, так как в базовую программу был внесен ряд изменений для лучшей интеграции с Unity.

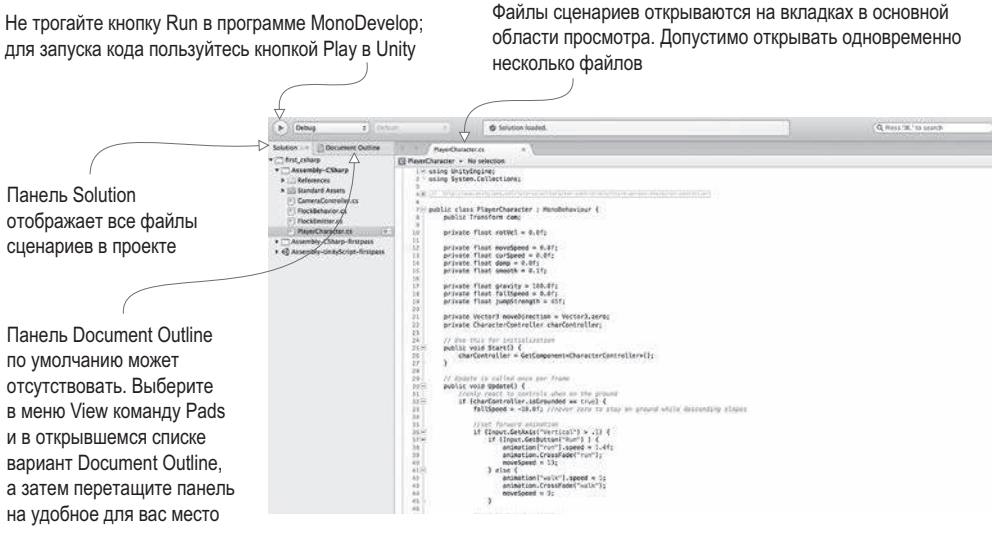


Рис. 1.14. Фрагменты интерфейса программы MonoDevelop

ПРИМЕЧАНИЕ Программа MonoDevelop объединяет файлы в группы, называемые *решениями* (solution). Приложение Unity автоматически генерирует решение, содержащее все файлы сценариев, поэтому, как правило, об этом можно вообще не думать.

Так как язык C# изначально разрабатывался компанией Microsoft, возможно, вам интересно, можно ли для программирования в Unity пользоваться Visual Studio. Да, можно. Для этого существуют инструменты, отвечающие за поддержку (в частности, за

корректную работу отладки и точек останова). Чтобы убедиться в наличии этих инструментов, проверьте, содержит ли меню Debug команду Attach Unity Debugger. В случае его отсутствия запустите установщик Visual Studio, внесите изменения в установленную версию и найдите модуль разработки игр Unity.

Я, как правило, использую программу MonoDevelop, но если вы уже привыкли программировать в Visual Studio, просто продолжайте работать с этой средой. Вы легко сможете выполнять все практические задания из книги (после вводной главы упоминать о IDE мы больше не будем). Однако привязка рабочего процесса к Windows аннулирует одно из самых важных преимуществ Unity. Язык C# разработан инженерами компании Microsoft, и поэтому изначально применялся только в Windows на платформе .NET Framework, однако он давно превратился в открытый стандарт языка и для него появился важный межплатформенный фреймворк: Mono. В Unity фреймворк Mono используется как основа для программирования, и именно среда MonoDevelop обеспечивает межплатформенные возможности всего процесса разработки.

ВНИМАНИЕ Интегрированная среда разработки MonoDevelop шла в комплекте с Unity 2017.1, но, согласно статье в блоге Unity http://mng.bz/9HR8, начиная с версии Unity 2018.1, ситуация изменится.

Все время помните, что в программах MonoDevelop и Visual Studio код только пишется, но не запускается. В данном случае IDE — это всего лишь хорошо оснащенный текстовый редактор, а воспроизводить код следует нажатием кнопки Play в программе Unity.

1.3.3. Вывод на консоль: Hello World!

Итак, в нашем проекте появился пустой сценарий, но пока отсутствует объект, к которому этот сценарий можно было бы присоединить. Вспомните рис. 1.1, демонстрирующий работу системы компонентов; любой сценарий — это тоже компонент, поэтому его нужно добавить к какому-то объекту.

Выберите в меню GameObject команду Create Empty, и на вкладке Hierarchy появится пустой объект с именем GameObject. Перетащите значок сценария с вкладки Project на вкладку Hierarchy и положите на строчку GameObject. Как показано на рис. 1.15, если присоединение сценария является допустимой операцией, компонент подсвечивается. Если вы отпустите кнопку мыши, сценарий будет связан с объектом GameObject. Чтобы удостовериться в успешном исходе операции, выделите объект и посмотрите на панель Inspector. Там должны появиться два компонента: Transform — базовый компонент положения/ориентации/масштаба, присутствующий у всех объектов, который невозможно удалить, а сразу под ним — сценарий.

ПРИМЕЧАНИЕ В конце концов вы привыкнете перетаскивать объекты и класть их на другие объекты. Этим способом в Unity создается огромное количество различных связей, а не только привязка сценариев к объектам.

Вид панели Inspector после связывания сценария с объектом показан на рис. 1.16. Сценарий уже фигурирует в качестве компонента и при воспроизведении сцены начнет исполняться. Но никаких внешних проявлений вы пока не заметите, так как код сценария еще не написан. Давайте устраним этот пробел.

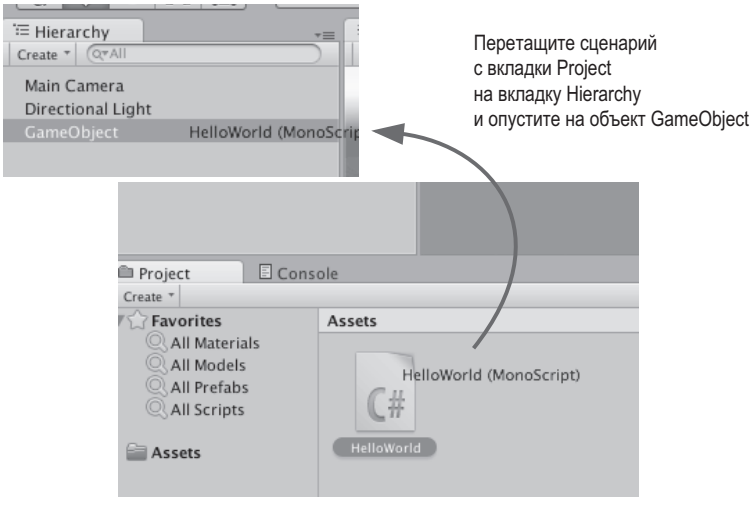


Рис. 1.15. Связывание сценария с объектом GameObject

Рис. 1.16. Связанный сценарий на панели Inspector

Откройте сценарий в Visual Studio, чтобы отредактировать листинг 1.1. Знакомство с новой средой программирования всегда начинается с вывода на экран текста «Hello World!». Поэтому добавьте в метод `Start()` строчку, показанную в следующем листинге.

Листинг 1.2. Сообщение для вывода на консоль

```
void Start() {
    Debug.Log("Hello World!"); ◄— Добавляем команду регистрации.
}
```

Команда `Debug.Log()` выводит сообщение на вкладку Console. Строка с этой командой вставляется в метод `Start()`, потому что, как упоминалось выше, данный метод однократно вызывается после активации объекта. Другими словами, после нажатия кнопки Play в редакторе метод `Start()` будет вызван всего один раз. Добавив в сценарий команду регистрации, обязательно сохраните его и нажмите кнопку Play в программе Unity. Откройте вкладку Console. Там появится сообщение «Hello World!». Поздравляю, вы

написали свой первый сценарий для Unity! В следующих главах код будет значительно сложнее, но сейчас вы сделали важный первый шаг.

ЭТАПЫ НАПИСАНИЯ СЦЕНАРИЯ HELLO WORLD!

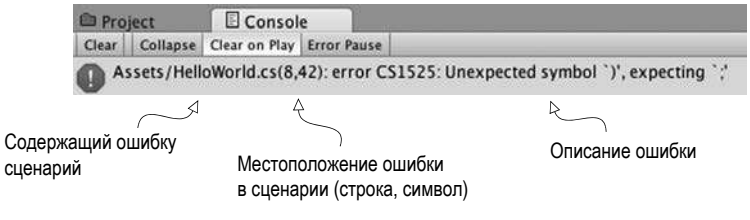
Кратко перечислим действия, выполнявшиеся при чтении последних страниц:

- Создание нового проекта.
- Создание нового сценария на языке C#.
- Создание пустого объекта GameObject.
- Перетаскивание сценария на этот объект.
- Добавление к сценарию команды регистрации.
- Нажатие кнопки Play!

Теперь сцену можно сохранить; появится файл с расширением .unity и со значком Unity. Этот файл представляет собой снимок всего, что есть в игре в данный момент, что позволяет в дальнейшем легко загрузить сцену в программу. В данном случае в сохранении нет особого смысла (у нас всего один пустой объект `GameObject`). Но если вы не сохраните сцену и захотите вернуться к ней в будущем, она окажется пустой.

ОШИБКИ В СЦЕНАРИИ

Чтобы посмотреть, каким образом Unity указывает на ошибки, специально сделайте опечатку в сценарии HelloWorld. Даже лишняя скобка приведет к появлению на вкладке Console сообщения об ошибке:



Заключение

- Unity — это мультиплатформенный инструмент разработки.
- Визуальный редактор Unity состоит из набора фрагментов, работающих в связке друг с другом.
- Сценарии присоединяются к объектам в виде компонентов.
- Код сценариев пишется в программе MonoDevelop или Visual Studio.

2 Создание 3D-ролика

- ✓ Знакомство с трехмерным координатным пространством.
- ✓ Размещение в сцене игрока.
- ✓ Создание сценария перемещения объектов.
- ✓ Реализация элементов управления персонажем в игре от первого лица.

Глава 1 завершилась традиционным способом знакомства с новыми средствами программирования — написанием программы «Hello World!». Пришло время погрузиться в более сложный Unity-проект, содержащий как средства взаимодействия с пользователем, так и графику. Вы поместите в сцену набор объектов и напишете код, позволяющий игроку перемещаться в этой сцене. По сути, это будет аналог игры Doom без монстров (примерно такой, как на рис. 2.1). Визуальный редактор Unity позволяет новым пользователям сразу приступить к сборке трехмерного прототипа без предварительного написания шаблонного кода (для таких вещей, как инициализация трехмерного представления или установка цикла визуализации).

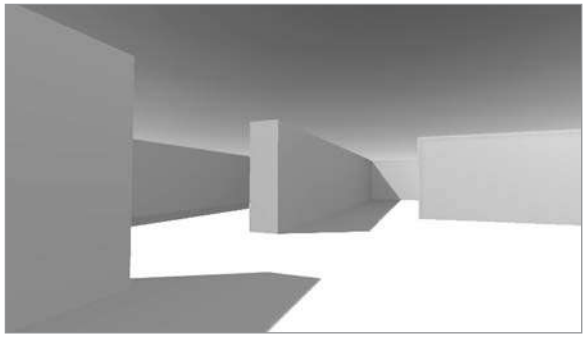


Рис. 2.1. Снимок экрана трехмерной демоверсии (по сути, это Doom без монстров)

Хотя на данном этапе высок соблазн немедленно заняться созданием сцены, особенно с учетом крайней простоты проекта, имеет смысл немного притормозить и продумать порядок своих действий. Сейчас это особенно важно, так как вы пока не знакомы с процессом.

ПРИМЕЧАНИЕ Проект этого (и всех разделов) можно загрузить с веб-сайта книги. Откройте проект в Unity, затем откройте Scene, чтобы запустить его. Пока вы учитесь, я рекомендую вам самостоятельно набирать весь код и использовать загруженный образец кода для справки. Адрес веб-сайта: www.manning.com/books/unity-in-action-second-edition.

2.1. Подготовка...

Инструмент Unity дает новичкам возможность сразу приступить к работе, но перед тем, как заняться созданием сцены, оговорим пару аспектов. Даже при наличии такого гибкого инструмента, как Unity, следует четко представлять себе, что именно вы хотите получить в результате своих действий. Кроме того, нужно хорошо представлять себе, как функционирует трехмерная система координат, в противном случае вы запутаетесь при первой же попытке вставки в сцену объекта.

2.1.1. Планирование проекта

Перед тем как приступить к программированию, всегда нужно остановиться и ответить на вопрос «Что я собираюсь построить?». Проектирование игр — весьма обширная тема, которой посвящено множество толстых книг. К счастью, в данном случае для разработки базового учебного проекта достаточно мысленно представлять структуру будущей сцены. Первые проекты будут несложными, чтобы лишние детали не мешали вам изучать принципы программирования, а задумываться о разработке более высокоуровневого проекта можно (и нужно!) только после того, как вы освоите основы создания игр.

Вашим первым проектом станет создание сцены из шутера от первого лица — мы будем обозначать этот тип игр аббревиатурой FPS (First-Person Shooter). Вы построите комнату, по которой можно перемещаться, при этом игроки будут наблюдать окружающий мир с точки зрения игрового персонажа. Управлять этим персонажем игрок сможет посредством мыши и клавиатуры. Все интересные, но сложные элементы готовой игры мы пока отбросим, чтобы сконцентрироваться на основной задаче — перемещениях в трехмерном пространстве. Рисунок 2.2 иллюстрирует сценарий проекта, по сути, представляющий собой придуманный мной перечень действий:

1. Разработка комнаты: создание пола, внешних и внутренних стен.
2. Размещение источников света и камеры.
3. Создание объекта-игрока (в том числе и присоединение камеры к его верхней части).
4. Написание сценариев перемещения: повороты при помощи мыши и перемещения при помощи клавиатуры.

Пусть столь внушительный сценарий вас не пугает! Кажется, что вам предстоит большая работа, но Unity делает ее очень простой. Разделы, посвященные сценариям перемещения, так велики только потому, что мы подробно рассматриваем каждую

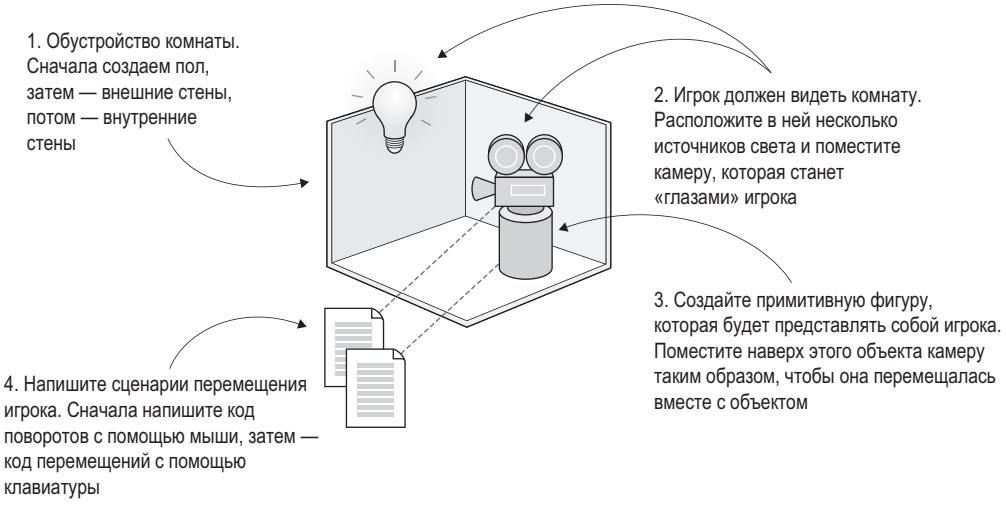


Рис. 2.2. Сценарий трехмерного демонстрационного ролика

деталь, чтобы полностью понять принципы программирования. Мы начинаем с игры от первого лица, чтобы снизить требования к художественному оформлению, — в играх от первого лица себя игрок не видит, поэтому вполне допустимо придать персонажу форму цилиндра, на верхней грани которого закреплена камера! Теперь осталось понять, как функционируют трехмерные координаты, и вы сможете легко вставлять объекты в сцены в визуальном редакторе.

2.1.2. Трехмерное координатное пространство

Сформулированный нами для начала простой план включает в себя три аспекта: игровое пространство, средства наблюдения, элементы управления. Для их реализации требуется понимание того, каким образом в трехмерных симуляциях задаются положение и перемещение объектов. Те, кто раньше никогда не сталкивался с трехмерной графикой, вполне могут этого не знать.

Все сводится к числам, указывающим положение точки в пространстве. Сопоставление этих чисел с пространством происходит через оси системы координат. На уроках математики в школе вы, скорее всего, видели показанные на рис. 2.3 оси *X* и *Y* и даже пользовались ими для присваивания координат различным точкам на листе бумаги. Это так называемая прямоугольная, или декартова, система координат.

Две оси дают вам двумерные координаты. Это случай, когда все точки лежат в одной плоскости. Трехмерное пространство задается уже тремя координатными осями. Так как ось *X* располагается на странице горизонтально, а ось *Y* — вертикально, третья ось должна как бы «протыкать» страницу, располагаясь перпендикулярно осям *X* и *Y*. Рисунок 2.4 демонстрирует оси *X*, *Y* и *Z* для трехмерного координатного пространства. У всех элементов, обладающих определенным положением в сцене (у игрока, у стен и т. п.), будут координаты *XYZ*.

На вкладке Scene в Unity вы видите значок трех осей, а на панели Inspector можно указать три числа, задающих положение объекта. Координаты в трехмерном пространстве вы

будете использовать не только при написании кода, определяющего местоположение объектов, но и для указания смещений как значений сдвига вдоль каждой из осей.

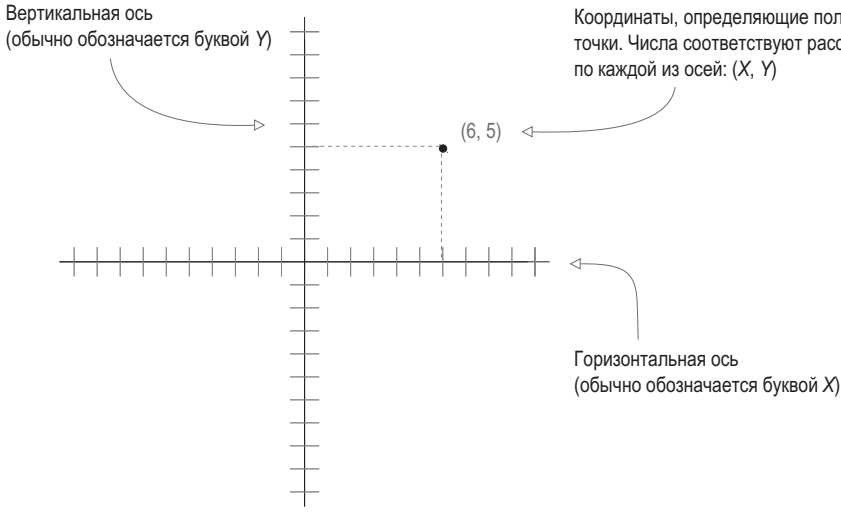


Рис. 2.3. Координаты по осям X и Y определяют положение точки на плоскости

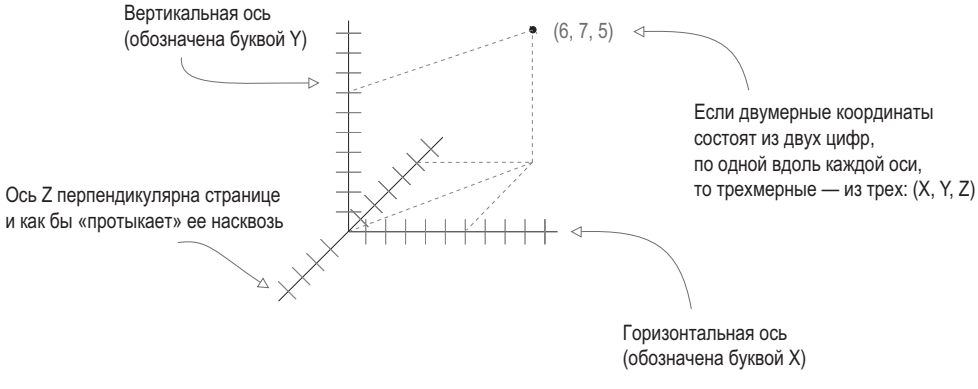
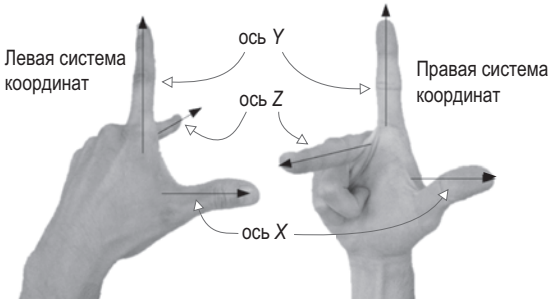


Рис. 2.4. Координаты по осям X, Y и Z определяют точку в трехмерном пространстве

ПРАВАЯ И ЛЕВАЯ СИСТЕМЫ КООРДИНАТ

Положительное и отрицательное направления каждой оси выбираются произвольным образом, в обоих случаях система координат прекрасно работает. Главное, используя инструмент для обработки трехмерной графики (инструмент анимации, инструмент разработки и т. п.), всегда выбирать одну и ту же систему координат.

Впрочем, практически всегда ось *X* указывает вправо, а ось *Y* — вверх; разница между инструментами в основном состоит в том, что где-то ось *Z* выходит из страницы, а где-то входит в нее. Эти два варианта называют «правой» и «левой» системами координат. Как показано на рисунке на с. 38, если большой палец расположить вдоль оси *X*, а указательный — вдоль оси *Y*, средний палец задаст направление оси *Z*.



У левой и правой руки ось Z ориентирована в разных направлениях

В Unity, как и во многих других приложениях для работы с компьютерной графикой, используется левая система координат. Однако существует множество инструментов, в которых применяется правая система координат (например, OpenGL). Помните об этом, чтобы не растеряться, столкнувшись с другими направлениями координатных осей.

Теперь, когда у вас есть не только план действий, но и представление о том, как с помощью координат задать положение объекта в трехмерном пространстве, приступим к работе над сценой.

2.2. Начало проекта: размещение объектов

Итак, начнем с создания объектов и размещения их в сцене. Первыми будут статические объекты — пол и стены. Затем выберем место для источников света и камеры. Последним создадим игрока — это будет объект, к которому вы добавите сценарии, перемещающие его по сцене. Рисунок 2.5 демонстрирует вид редактора после завершения работы.

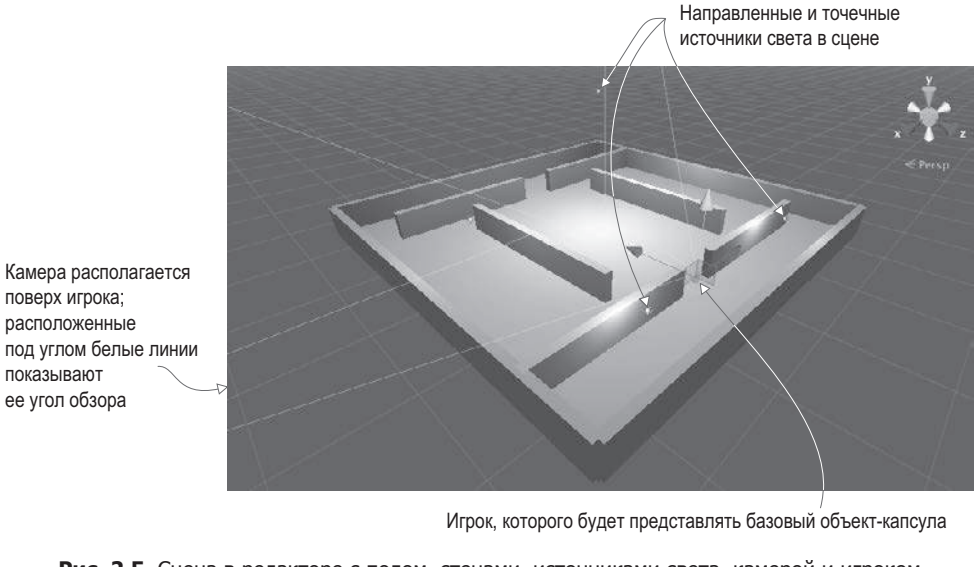


Рис. 2.5. Сцена в редакторе с полом, стенами, источниками света, камерой и игроком

В главе 1 демонстрировался способ создания нового проекта в Unity. Именно это мы сейчас и проделаем. Напоминаю, что нужно выбрать в меню File команду New Project и в появившемся окне указать имя проекта. После этого сразу же сохраните пустую сцену, так как изначально файл сцены у нового проекта отсутствует. Начнем мы с создания наиболее очевидных объектов.

2.2.1. Декорации: пол, внешние и внутренние стены

В расположенном в верхней части экрана меню GameObject наведите указатель мыши на строчку 3D Object, чтобы открыть дополнительное меню. Выберите в нем вариант Cube, так как для нашей сцены требуется куб (позднее вы поработаете и с другими фигурами, такими как Sphere и Capsule). Отредактируйте положение и масштаб появившегося в сцене куба, а также его имя таким образом, чтобы получить пол; значения, которые следует присвоить параметрам этого объекта на панели Inspector, показаны на рис. 2.6 (для превращения куба в пол его нужно растянуть).

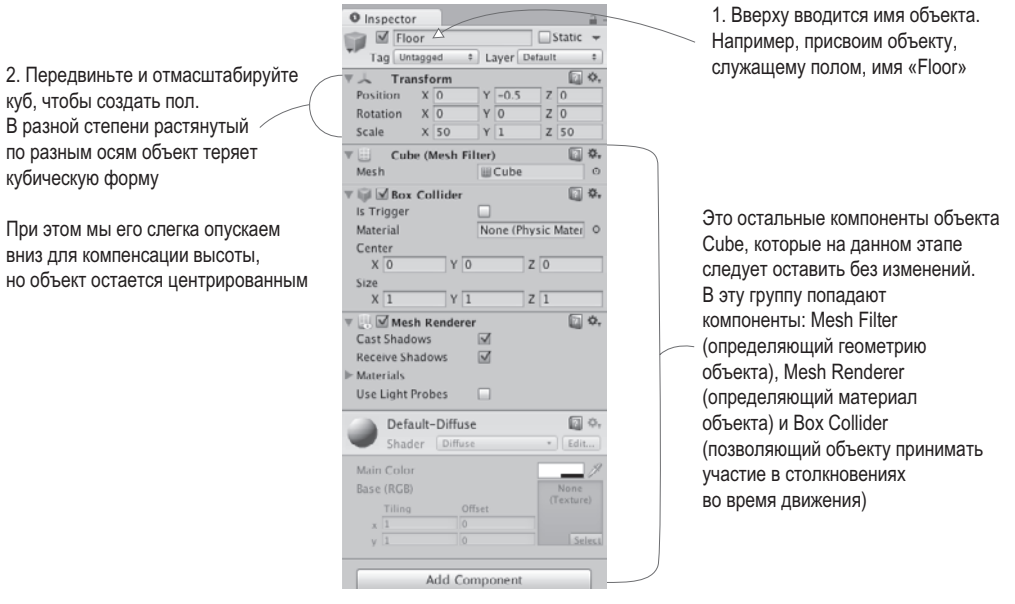


Рис. 2.6. Панель Inspector для пола

ПРИМЕЧАНИЕ Единицы измерения положения могут выбираться произвольным образом; главное, чтобы они использовались для всех элементов сцены. Чаще всего в качестве единицы измерения фигурирует метр. Сам я предпочитаю пользоваться именно метрами, но бывают случаи, когда я выбираю футы. Хотя мне доводилось видеть людей, которые считали, что размеры в сцене измеряются в дюймах!

Повторите описанную последовательность для создания внешних стен комнаты. Можно каждый раз задействовать новый куб, а можно копировать и вставлять существующий объект, указывая стандартные сокращения. Двигайте, поворачивайте

и масштабируйте стены, чтобы получить показанный на рис. 2.5 периметр. Экспериментируйте с различными значениями (например, 1, 4, 50 для полей Scale) или воспользуйтесь инструментами преобразований, с которыми вы познакомились в разделе 1.2.2 (напоминаю, что перемещения и повороты в трехмерном пространстве называют преобразованиями).

СОВЕТ Не забудьте про средства навигации, позволяющие рассматривать сцену под разными углами и менять ее масштаб, например, имитируя взгляд с высоты птичьего полета. При этом нажатие клавиши F вернет вас к просмотру выделенного в данный момент объекта.

Рис. 2.7. Панель Hierarchy, показывающая, что стены и пол являются потомками пустого объекта

Точные значения преобразований для стен будут зависеть от того, каким образом вы повернете и отмасштабируете исходные объекты Cube, подогнав их размеры и положение, а также от способа их связывания на вкладке Hierarchy. К примеру, на рис. 2.7 демонстрируется ситуация, когда все стены являются потомками пустого корневого объекта. Содержимое вкладки Hierarchy в этом случае имеет упорядоченный вид. Если вы предпочитаете просто скопировать рабочие значения, скачайте пример проекта и возьмите все данные оттуда.

СОВЕТ Связи между объектами устанавливаются простым перетаскиванием объектов друг на друга на вкладке Hierarchy. Объект, к которому присоединены другие объекты, называется предком (parent); объекты, присоединенные к другим объектам, называются потомками (children). Перемещение (поворот или масштабирование) родительского объекта сопровождается аналогичным преобразованием всех его потомков.

СОВЕТ Для систематизации объектов сцены подобным образом применяются пустые игровые объекты. Связывание видимых объектов с корневым позволяет сворачивать списки объектов на вкладке Hierarchy. Но помните, что перед этой операцией следует расположить пустой корневой объект в точке с координатами `0`, `0`, `0`, чтобы в дальнейшем избежать проблем с позиционированием.

ЧТО ТАКОЕ GAMEOBJECT?

Все объекты сцены представляют собой экземпляры класса `GameObject` аналогично тому, как все компоненты сценариев наследуют от класса `MonoBehaviour`. Этот факт становится более наглядным, если пустому объекту присвоить имя `GameObject`. Впрочем, даже если этот объект будет называться Floor, Camera или Player, суть дела не изменится.

На самом деле `GameObject` представляет собой всего лишь контейнер для набора компонентов. Его основным назначением является обеспечение некоего объекта, к которому можно присоединять класс `MonoBehaviour`. Как все это будет выглядеть в сцене, зависит от добавленных к объекту `GameObject` компонентов. К примеру, куб получается добавлением компонента `Cube`, сфера — добавлением компонента `Sphere`, и т. п.

Завершив работу над внешними стенами, приступайте к созданию внутренних. Расположите их по своему вкусу. Вам нужны коридоры и препятствия, среди которых будет происходить движение. В итоге в сцене появится комната, но без источников света игрок ничего в ней не увидит. Поэтому давайте осветим нашу комнату.

2.2.2. Источники света и камеры

Как правило, трехмерные сцены освещаются направленным источником света, к которому добавляется набор точечных осветителей. Начать имеет смысл с направленного источника. Он может по умолчанию присутствовать в сцене, но если его нет, наведите указатель мыши на строку Light в меню GameObject и в открывшемся дополнительном меню выберите вариант Directional Light.

ТИПЫ ОСВЕТИТЕЛЕЙ

Существуют различные типы осветителей, разными способами проецирующие световые лучи. Три основных типа: точечный источник, прожектор и направленный источник.

Все лучи точечного источника (point light) начинаются в одной точке и распространяются во всех направлениях. В реальном мире таким осветителем является лампочка. Яркость света увеличивается по мере приближения к источнику за счет концентрации лучей.

Лучи прожектора (spot light) также исходят из одной точки, но распространяются в пределах ограниченного конуса. В текущем проекте мы с прожекторами работать не будем, но осветители данного типа повсеместно используются для подсвечивания отдельных частей уровня.

Лучи направленного источника света (directional light) распространяются равномерно и параллельно друг другу, одинаково освещая все элементы сцены. Это аналог солнца.

Испускаемый направленным осветителем свет не зависит от местоположения источника, значение имеет только его ориентация, поэтому его можно поместить в произвольную точку сцены. Я рекомендую установить его над комнатой, чтобы он выглядел как солнце и не мешал вам при работе с остальными фрагментами сцены. Поверните источник света и посмотрите, как это повлияет на освещенность комнаты; для получения нужного эффекта я рекомендую слегка повернуть его относительно осей *X* и *Y*. На панели Inspector вы найдете параметр Intensity (рис. 2.8). Как следует из его названия, он управляет яркостью света. Если бы данный направленный осветитель был в сцене единственным, его яркость имело бы смысл увеличить, но, так как мы добавим несколько точечных источников света, его можно оставить тусклым. Например, присвойте параметру Intensity значение 0.6.

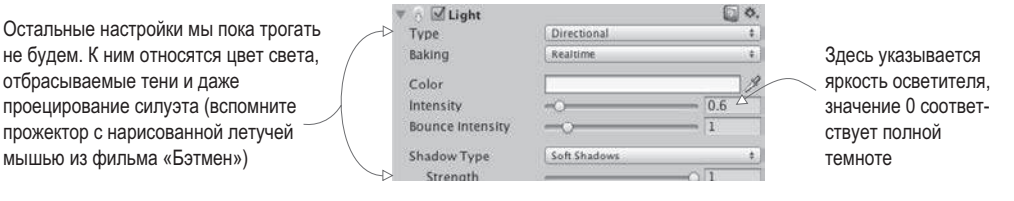


Рис. 2.8. Настройки направленного источника света на панели Inspector

Теперь командой уже знакомого вам меню создайте несколько точечных осветителей, поместив их в темные места комнаты. Вам нужно, чтобы все было как следует освещено. Источников должно быть не слишком много (иначе пострадает производительность), вполне достаточно по одному осветителю в каждом углу (я предлагаю поднять их к верхней кромке стен) и одного, расположенного высоко над сценой (например, со значением 18 координаты *Y*). Это придаст освещению комнаты некое разнообразие.

Обратите внимание, что у точечных источников света на панели Inspector появляется дополнительный параметр Range (рис. 2.9). Он управляет дальностью распространения лучей. Если направленный источник света равномерно освещает всю сцену, то яркость точечного источника уменьшается по мере удаления от него. Поэтому для стоящего на полу источника этот параметр может быть равен 18, а для поднятого к верхней кромке стены его необходимо увеличить до 40, иначе он не сможет осветить всю комнату.

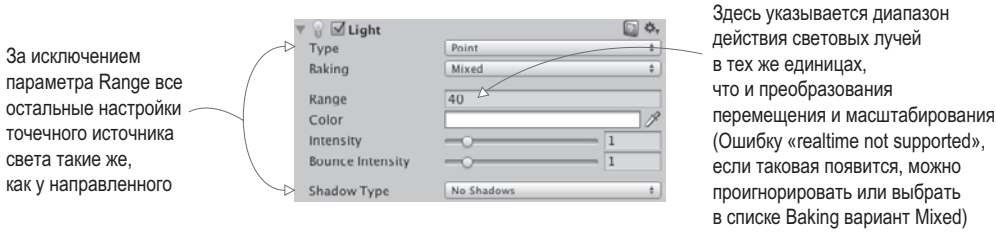


Рис. 2.9. Настройки точечного источника света на панели Inspector

Чтобы игрок мог наблюдать за происходящим, требуется еще один тип объекта — камера. Впрочем, пустая сцена содержит основную камеру, которой мы можем воспользоваться. Если вам когда-нибудь потребуется создать дополнительные камеры (например, для разделения экрана в многопользовательских играх), выберите команду Camera в меню GameObject. Нашу же камеру мы расположим поверх игрока, чтобы наблюдать сцену его глазами.

2.2.3. Коллайдер и точка наблюдения игрока

В этом проекте игрока будет представлять обычный примитив. В меню GameObject (напоминаю, что для открытия этого дополнительного меню нужно навести указатель мыши на строку 3D Object) выберите вариант Capsule. Появится цилиндрическая фигура со скругленными концами — это и есть наш игрок. Сместите объект вверх, сделав его координату *Y* равной 1.1 (половина высоты объекта плюс еще немного, чтобы избежать перекрывания с полом). Теперь наш игрок может произвольным образом перемещаться вдоль осей *X* и *Z* при условии, что он остается внутри комнаты и не касается стен. Присвойте объекту имя Player.

На панели Inspector вы увидите, что этому объекту назначен капсульный коллайдер. Это очевидный вариант для объекта Capsule, точно так же, как объект Cube по умолчанию обладает коллайдером Box. Но так как наша капсула будет представлять игрока, ее компоненты должны слегка отличаться от компонентов большинства объектов. Капсульный коллайдер мы удалим. Для этого нужно щелкнуть на значке с изображением шестерни справа от имени компонента, как показано на рис. 2.10. Откроется меню, в котором в числе прочих вы найдете и команду Remove Component. Коллайдер выглядит как окружающая объект зеленая сетка, поэтому после удаления компонента вы обнаружите, что она исчезла.

Вместо капсульного коллайдера мы назначим объекту контроллер персонажа. В нижней части панели Inspector вы найдете кнопку Add Component. Щелчок на ней открывает меню с перечнем типов доступных компонентов. В разделе Physics и находится нужная

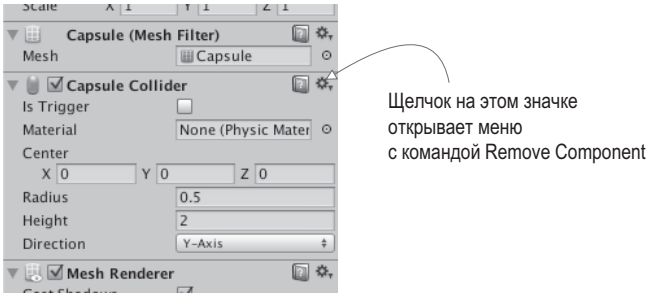


Рис. 2.10. Удаление компонента на панели Inspector

нам строка Character Controller. Как несложно догадаться, именно этот компонент позволит объекту вести себя как персонаж.

Для завершения настройки игрока осталось сделать всего один шаг — присоединить к нему камеру. Как уже упоминалось в разделе, посвященном созданию пола и стен, вы можете перетаскивать объекты и размещать их друг на друге на вкладке Hierarchy. Проделайте эту операцию, перетащив камеру на капсулу, чтобы присоединить ее к игроку. Затем расположите ее таким образом, чтобы она соответствовала глазам игрока (я предлагаю указать координаты 0, 0.5, 0). При необходимости верните координатам преобразования поворота значения 0, 0, 0 (если вы вращали капсулу, они будут различаться).

Итак, в сцене присутствуют все необходимые объекты. Осталось написать код перемещения игрока.

2.3. Двигаем объекты: сценарий, активирующий преобразования

Чтобы заставить игрока перемещаться по сцене, нам потребуются сценарии движения, которые будут присоединены к игроку. Напоминаю, что компонентами называются модульные фрагменты функциональности, добавляемые к объектам, поэтому сценарии тоже можно считать своего рода компонентами. Именно они будут реагировать на клавиатурный ввод и манипуляции мышью, но для начала мы заставим игрока поворачиваться на месте. Это продемонстрирует вам, как применять преобразования в коде. Надеюсь, вы помните, что преобразования бывают трех видов — перемещение, поворот и масштабирование; вращение объекта на месте соответствует преобразованию поворота. Но вы должны знать об этой задаче еще кое-что кроме того, что она «сводится к изменению ориентации объекта».

2.3.1. Схема программирования движения

Анимация объекта (например, его вращение) сводится к смещению его в каждом кадре на небольшое расстояние и к последующему многократному воспроизведению всех кадров. Само по себе преобразование происходит мгновенно, в отличие от движения, растянутого во времени. Но последовательное применение набора преобразований вызывает визуальное перемещение объекта, совсем как набор рисунков в кинеографе. Этот принцип проиллюстрирован на рис. 2.11.

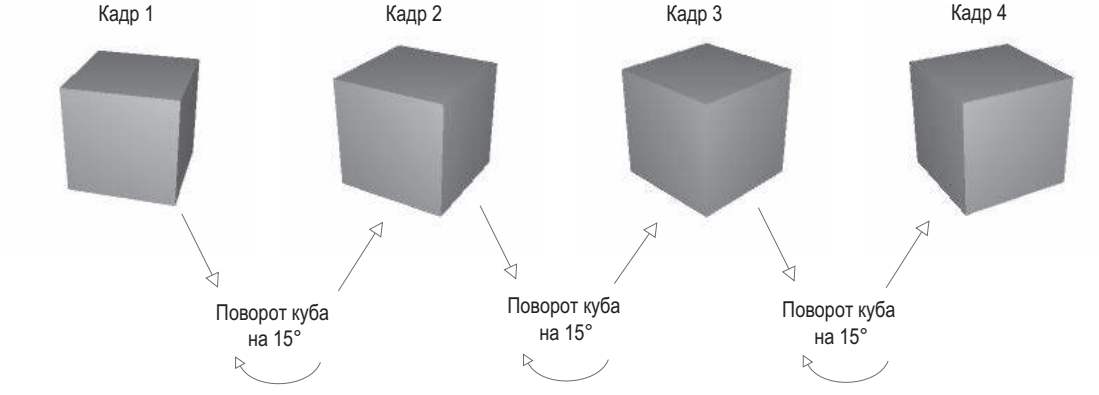


Рис. 2.11. Возникновение движения: циклический процесс преобразования статичных изображений

Напомню вам, что компоненты сценария снабжены запускающимся в каждом кадре методом `Update()`. Поэтому, чтобы заставить куб вращаться вокруг своей оси, следует добавить в метод `Update()` код, поворачивающий его на небольшой угол. Этот код будет запускаться в каждом кадре снова и снова. Не правда ли, все очень просто?

2.3.2. Написание кода

Применим рассмотренную концепцию на практике. Создайте новый сценарий C# (напоминаю, что команда находится в дополнительном меню Create, доступ к которому осуществляется через меню Assets), присвойте ему имя Spin и напишите код из следующего листинга (не забудьте после ввода листинга сохранить файл!).

Листинг 2.1. Приводим объект во вращение

```
using UnityEngine;
using System.Collections;

public class Spin : MonoBehaviour {
    public float speed = 3.0f;

    void Update() {
        transform.Rotate(0, speed, 0);
    }
}
```

Объявление общедоступной переменной для скорости вращения.

Поместите сюда команду Rotate, чтобы она запускалась в каждом кадре.

Для добавления компонента сценария к игроку перетащите сценарий с вкладки Project на строку Player на вкладке Hierarchy. Щелкните на кнопке Play, и вы увидите, как все придет во вращение — вы написали код, заставляющий объект двигаться! В основном этот код состоит из шаблона нового сценария, к которому добавлены две строки. Посмотрим, что именно они делают.

Прежде всего, появилась переменная для скорости, добавленная в верхнюю часть определения класса. Превращение скорости вращения в переменную произошло по

двум причинам. Первая — это соблюдение стандартного правила программирования: «никаких магических чисел». Вторая же причина связана со способом отображения общедоступных переменных в Unity. Познакомимся с ним.

СОВЕТ Общедоступные переменные появляются на панели Inspector, что позволяет редактировать значения уже добавленных к игровому объекту компонентов. Это называется «сериализацией» значения, так как Unity сохраняет модифицированное состояние переменной.

Рисунок 2.12 показывает вид компонента сценария на панели Inspector. Можно ввести новое значение, и сценарий будет использовать его вместо указанного в коде. Таким способом удобно редактировать параметры компонентов различных объектов непосредственно в визуальном редакторе, вместо того чтобы заниматься правкой кода.

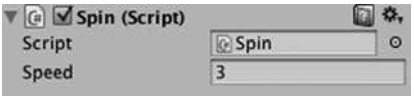


Рис. 2.12. На панели Inspector присутствует объявленная в сценарии общедоступная переменная

Вторая строка, на которую следует обратить внимание в листинге 2.1, касается метода `Rotate()`. Он вставлен в метод `Update()`, а значит, выполняется в каждом кадре. Так как метод `Rotate()` принадлежит классу `Transform`, он вызывается при помощи точечной нотации через компонент преобразования объекта (как и в большинстве объектно-ориентированных языков, если вы введете `transform`, это будет восприниматься как `this.transform`). В рассматриваемом случае преобразование сводится к повороту на `speed` градусов в каждом кадре, что в итоге даст нам плавное вращение. Но почему параметры метода `Rotate()` указаны как `(0, speed, 0)`, а не, допустим, `(speed, 0, 0)`?

Вспомните, что в трехмерном пространстве есть три оси: *X*, *Y* и *Z*. Интуитивно понятно, каким образом они связаны с местоположением объекта и его перемещениями, но, кроме того, их применяют для описания преобразования поворота. Сходным образом описывают вращение в воздухоплавании, поэтому работающие с трехмерной графикой программисты зачастую пользуются терминами из механики полета: тангаж (pitch), рысканье (yaw) и крен (roll). Значение этих терминов иллюстрирует рис. 2.13: тангаж означает вращение вокруг оси *X*, рысканье — вращение вокруг оси *Y*, крен — вращение вокруг оси *Z*.

Возможность описывать вращение вокруг осей *X*, *Y* и *Z* означает три параметра для метода `Rotate()`. Так как нам требуется только вращение вокруг своей оси без наклонов вперед и назад, меняться должна только координата *Y*, координатам же *X* и *Z* мы просто присвоим значение 0. Надеюсь, вы уже поняли, что получится, если изменить параметры на `(speed, 0, 0)` и запустить воспроизведение сцены; попробуйте реализовать это на практике!

Существует еще одна тонкость, связанная с вращением и осями координат, реализованная в виде необязательного четвертого параметра метода `Rotate()`.

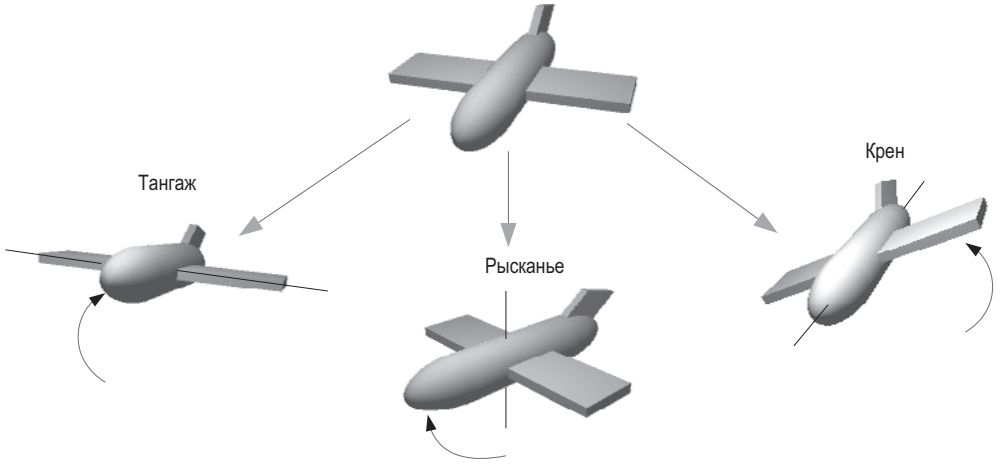


Рис. 2.13. Тангаж, рысканье и крен самолета

2.3.3. Локальные и глобальные координаты

По умолчанию метод `Rotate()` работает с так называемыми локальными координатами, хотя вы также можете использовать глобальную систему координат. Вы сообщаете методу, какой именно системой координат хотите воспользоваться, при помощи четвертого, необязательного, параметра, который может принимать два значения: `Space.Self` и `Space.World`. Например:

```
Rotate(0, speed, 0, Space.World)
```

Вернитесь к описанию трехмерного координатного пространства и хорошенько подумайте над следующими вопросами: где находится точка (0, 0, 0)? как выбирается положительное направление оси *X*? может ли перемещаться сама координатная система?

Оказывается, у каждого объекта есть собственная точка начала координат и собственные положительные направления осей. И эта координатная система перемещается вместе с объектом. В таких случаях говорят о локальных координатах. При этом трехмерная сцена как целое обладает собственной точкой начала координат и собственными направлениями осей, причем такая координатная система всегда статична. Это так называемые глобальные координаты. Соответственно, указывая вариант системы координат для метода `Rotate()`, вы сообщаете, относительно каких осей *X*, *Y* и *Z* нужно выполнить преобразование поворота (рис. 2.14).

Новичкам в области трехмерной графики эта концепция может показаться сложной для понимания. Различные оси показаны на рис. 2.14 (обратите внимание, что, к примеру, понятие «лево» для самолета отличается от понятия «лево» для сцены в целом), но разницу между локальными и глобальными координатами проще всего понять на примере.

Для начала выделите игрока и слегка его наклоните (например, повернув относительно оси *X* на 30°). Это приведет к отключению используемых по умолчанию локальных координат, и повороты относительно глобальных и локальных координат

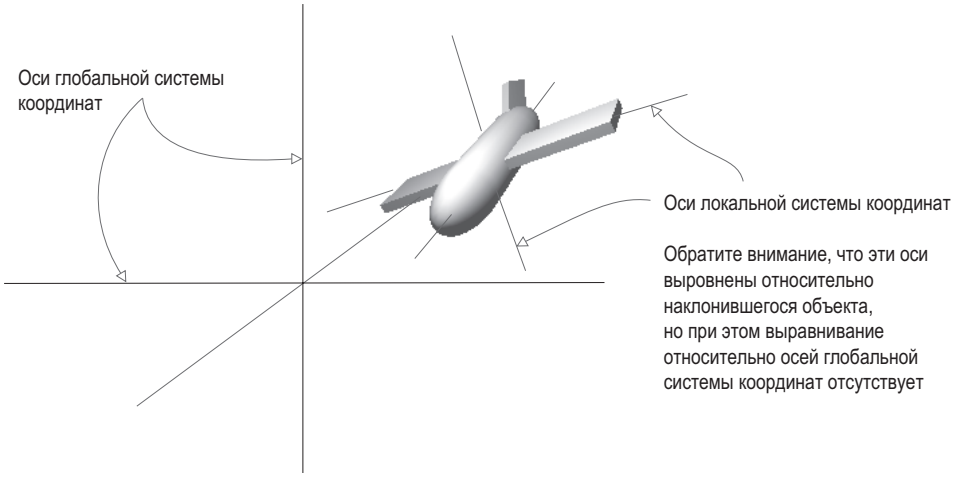


Рис. 2.14. Оси локальной и глобальной систем координат

начнут выглядеть по-разному. Теперь попробуйте запустить сценарий `Spin` как с добавленным параметром `Space.World`, так и без него; если вам сложно понять, что именно происходит, удалите назначенный игроку компонент вращения и начните вращать расположенный перед игроком наклоненный куб. Вы увидите, что оси вращения зависят от выбранной системы координат.

2.4. Компонент сценария для осмотра сцены: MouseLook

Теперь нужно заставить преобразование поворота реагировать на ввод с помощью мыши. В данном случае подразумевается вращение объекта, к которому присоединен сценарий, то есть игрока. Задача будет решаться в несколько этапов путем постепенного добавления персонажу двигательных возможностей. Сначала мы заставим игрока поворачиваться только из стороны в сторону, а затем — только вверх и вниз. В конечном счете игрок научится смотреть во всех направлениях (поворачиваясь одновременно в горизонтальной и вертикальной плоскостях). Такое поведение называют *слежением за мышью* (mouse-look).

Так как у нас предполагается три типа поведения при повороте (по горизонтали, по вертикали и комбинированный), начнем мы с написания фреймворка, поддерживающего все эти типы. Создайте новый сценарий на C# с именем MouseLook и добавьте в него код из следующего листинга.

Листинг 2.2. Фреймворк MouseLook с перечислением для преобразования поворота

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
```

Объявляем структуру данных enum, которая будет сопоставлять имена с параметрами.

```
      MouseXAndY = 0,
      MouseX = 1,
      MouseY = 2
  }
  public RotationAxes axes = RotationAxes.MouseXAndY;

  void Update() {
    if (axes == RotationAxes.MouseX) {
        // это поворот в горизонтальной плоскости
    }
    else if (axes == RotationAxes.MouseY) {
        // это поворот в вертикальной плоскости
    }
    else {
        // это комбинированный поворот
    }
  }
}
```

Объявляем общедоступную переменную, которая появится в редакторе Unity.

Сюда поместим код для вращения только по горизонтали.

Сюда поместим код для вращения только по вертикали.

Сюда поместим код для комбинированного вращения.

Обратите внимание, что именно перечисление позволяет сценарию MouseLook выбирать, каким образом — в горизонтальной или вертикальной плоскости — будет выполняться поворот. Определив структуру данных enum, вы получите возможность задавать значения по имени, вместо того чтобы указывать числа и пытаться запомнить, что означает каждое из них. (Означает ли ноль вращение по горизонтали? Или такому повороту соответствует единица?) Если затем объявить общедоступную переменную, типизированную как перечисление, она появится на панели Inspector в виде раскрывающегося меню, удобного для выбора параметров (рис. 2.15).

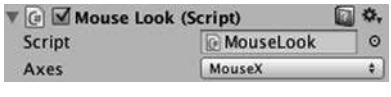


Рис. 2.15. Панель Inspector отображает сопоставленные перечислениям общедоступные переменные в виде раскрывающихся меню

Удалите компонент `Spin` (тем же способом, каким ранее удалялся капсульный коллайдер) и вместо него присоедините к игроку новый сценарий. В процессе работы над кодом для перехода от одной ветки кода к другой пользуйтесь меню Axes. По очереди выбирая горизонтальное/вертикальное вращение, вы сможете написать код для каждой ветки условной конструкции.

2.4.1. Горизонтальное вращение, следящее за указателем мыши

Первая и наиболее простая ветка соответствует вращению в горизонтальной плоскости. Для начала напишем уже знакомую вам команду из листинга 2.1, заставляющую объект поворачиваться. Не забудьте объявить общедоступную переменную для скорости вращения; сделайте это после объявления осей, но до метода `Update()`, назвав новую переменную `sensitivityHor`, так как слово «speed» имеет слишком общий смысл и не позволит отличать варианты поворота друг от друга. Увеличьте значение этой

переменной до 9, так как совсем скоро она начнет масштабироваться. Исправленный код приведен в следующем листинге.

Листинг 2.3. Поворот по горизонтали, пока не связанный с движениями указателя

```
...
public RotationAxes axes = RotationAxes.MouseXAndY;

public float sensitivityHor = 9.0f;

void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, sensitivityHor, 0);
    }
...
```

Курсивом выделен код, который уже присутствует в сценарии; здесь он приведен для удобства.

Объявляем переменную для скорости вращения.

Сюда мы поместим команду Rotate, чтобы она запускалась в каждом кадре.

Если запустить сценарий, сцена, как и раньше, начнет вращаться, но намного быстрее, потому что теперь скорость вращения вокруг оси *Y* равна 9, а не 3. Теперь нам нужно сделать так, чтобы преобразование возникало в ответ на движения указателя мыши, поэтому создадим новый метод: `Input.GetAxis()`. Класс `Input` обладает множеством методов для обработки информации, поступающей с устройств ввода (таких, как мышь). В частности, метод `GetAxis()` возвращает числа, связанные с движениями указателя мыши (положительные или отрицательные в зависимости от направления движения). Он принимает в качестве параметра имя нужной оси. А горизонтальная ось у нас называется Mouse X.

Если умножить скорость вращения на координату оси, объект начнет поворачиваться вслед за указателем мыши. Скорость масштабируется в соответствии с перемещениями указателя, уменьшаясь до нуля и даже меняя направление. Новый вид команды `Rotate` показан в следующем листинге.

Листинг 2.4. Команда Rotate, реагирующая на движения указателя мыши

```
...
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
...
```

Метод GetAxis() получает данные, вводимые с помощью мыши.

Щелкните на кнопке Play и подвигайте мышь в разные стороны. Объект начнет поворачиваться вправо и влево вслед за указателем. Видите, как здорово! Теперь нужно заставить игрока вращаться еще и в вертикальной плоскости.

2.4.2. Поворот по вертикали с ограничениями

Вращение в горизонтальной плоскости было реализовано у нас с помощью метода `Rotate()`, но для поворота по вертикали мы воспользуемся другим способом. Дело в том, что указанный метод при всем его удобстве в осуществлении преобразований недостаточно гибок. Он применим только для неограниченного приращения угла поворота, что в нашей ситуации подходит только для вращения в горизонтальной плоскости. В случае же поворота по вертикали требуется задать предел наклона вниз и вверх. Код этого преобразования для сценария `MouseLook` представлен в следующем листинге.

Листинг 2.5. Поворот в вертикальной плоскости для сценария MouseLook

```
...
public float sensitivityHor = 9.0f;
public float sensitivityVert = 9.0f;

public float minimumVert = -45.0f;
public float maximumVert = 45.0f;

private float _rotationX = 0;

void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
        _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
        _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

        float rotationY = transform.localEulerAngles.y;

        transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
...
```

Объявляем переменные, задающие поворот в вертикальной плоскости.

Объявляем закрытую переменную для угла поворота по вертикали.

Увеличиваем угол поворота по вертикали в соответствии с перемещениями указателя мыши.

Создаем новый вектор из сохраненных значений поворота.

Фиксируем угол поворота по вертикали в диапазоне, заданном минимальным и максимальным значениями.

Сохраняем одинаковый угол поворота вокруг оси Y (то есть вращение в горизонтальной плоскости отсутствует).

Выбираем в меню Axes компонента `MouseLook` вращение по вертикали и воспроизводим новый сценарий. Теперь сцена вместо поворотов из стороны в сторону будет вслед за движениями указателя мыши наклоняться вверх и вниз. При этом углы поворота ограничиваются указанными вами пределами.

Этот код знакомит вас с рядом новых понятий, которые следует объяснить более подробно. Во-первых, теперь мы не пользуемся методом `Rotate()`, поэтому нам требуется еще одна переменная (она называется `_rotationX`, так как вертикальное вращение происходит вокруг оси *X*), предназначенная для сохранения угла поворота. Метод `Rotate()` просто увеличивает угол поворота, в то время как на этот раз мы его задаем в явном виде. Другими словами, если в предыдущем листинге мы просили «добавить 5 к текущему значению угла», то теперь мы «присваиваем углу поворота значение 30». Разумеется, нам и в этом случае нужно увеличивать угол поворота, именно поэтому в коде присутствует оператор `-=`. То есть мы вычитаем значение из угла поворота, а не присваиваем это значение углу. Так как методом `Rotate()` мы не пользуемся, манипулировать углом поворота можно разными способами, а не только увеличивая его. Этот параметр умножается на `Input.GetAxis()`, совсем как в коде для вращения в горизонтальной плоскости, просто на этот раз мы выясняем значение `Mouse Y`, ведь нас интересует вертикальная ось.

Следующая строка также посвящена манипуляциям углом поворота. Метод `Mathf.Clamp()` позволяет ограничить этот параметр максимальным и минимальным значениями. Эти предельные значения задаются объявленными общедоступными

переменными, которые гарантируют поворот сцены вверх и вниз всего на 45 градусов. Метод `Clamp()` работает не только с преобразованиями поворота. Он, в принципе, используется для сохранения числового значения переменной в заданных пределах. В экспериментальных целях превратите строку с методом `Clamp()` в комментарий; вы увидите, как объект начнет свободно вращаться в вертикальной плоскости! Понятно, что нам совсем не обязательно смотреть на сцену, перевернутую с ног на голову; отсюда и появляются ограничения.

Так как угловое свойство преобразования выражается переменной `Vector3`, мы должны создать новую переменную с углом поворота, которая будет передаваться в конструктор. Метод `Rotate()` этот процесс автоматизировал, увеличив угол поворота и затем создав новый вектор.

ОПРЕДЕЛЕНИЕ *Вектором* (vector) называется набор чисел, сохраняемых как единое целое. К примеру, `Vector3` состоит из трех чисел (помеченных как x, y, z).

ВНИМАНИЕ Мы создаем новый вектор Vector3, вместо того чтобы поменять значения у существующего, так как в случае преобразований эти значения предназначены только для чтения. Это крайне распространенная ошибка.

УГЛЫ ЭЙЛЕРА И КВАТЕРНИОНЫ

Возможно, вам интересно, почему свойство называется `localEulerAngles`, а не `localRotation`. Для ответа на этот вопрос вам нужно познакомиться с концепцией, называемой *кватернионами* (quaternions).

Кватернионы представляют собой математическую конструкцию для представления вращения объектов. Они отличаются от углов Эйлера, то есть используемого нами подхода с осями *X*, *Y*, *Z*. Помните обсуждение тангажа, рысканья и крена? В этом случае вращения были представлены с помощью углов Эйлера. Кватернионы... совсем другие. Объяснить их природу сложно, так как для этого нужно углубиться в запутанные дебри высшей математики, включающие в себя движение через четыре измерения. Если вам требуется детальное объяснение, попробуйте прочесть документ http://wat.gamedev.ru/articles/quaternions.

Несколько проще объяснить, почему кватернионы используются для представления вращений: они более равномерно реализуют интерполяцию между углами поворота (то есть переход от одного промежуточного значения к другому).

Свойство `localRotation` выражено через кватернионы, а не через углы Эйлера. При этом в Unity существует и свойство, выражаемое через углы Эйлера, благодаря которому понять, как именно происходит процесс поворота, становится проще; преобразование одного варианта значений в другой и обратно делается автоматически. Инструмент Unity выполняет все сложные вычисления в фоновом режиме, избавляя вас от необходимости проделывать их вручную.

Осталось написать код для последнего вида вращения, происходящего одновременно в горизонтальной и вертикальной плоскостях.

2.4.3. Одновременные горизонтальное и вертикальное вращения

В последнем фрагменте кода метод `Rotate()` также не применяется, так как нам снова требуется ограничить диапазон углов поворота по вертикали. Это означает, что горизонтальный поворот тоже нужно вычислять вручную. Напоминаю, что метод `Rotate()` автоматизировал процесс приращения угла поворота.

Листинг 2.6. Сценарий MouseLook, поворачивающий объект одновременно по горизонтали и по вертикали

```
...
else {
    _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
    _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

    float delta = Input.GetAxis("Mouse X") * sensitivityHor;
    float rotationY = transform.localEulerAngles.y + delta;

    transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
}
...
```

Значение delta — это величина изменения угла поворота.

Приращение угла поворота через значение delta.

Первая пара строк, посвященная переменной `_rotationX`, полностью аналогична таким же строкам из предыдущего сценария. Просто помните, что вращение в вертикальной плоскости происходит вокруг оси *X* объекта. А так как вращение в горизонтальной плоскости больше не реализуется через метод `Rotate()`, появились строки с переменными `delta` и `rotationY`. *Дельта* — это распространенное математическое обозначение «величины изменения», поэтому наше вычисление переменной `delta` дает величину, на которую следует поменять угол поворота. Затем она добавляется к текущему значению этого угла для придания объекту желаемой ориентации.

В конце оба угла наклона — по вертикали и по горизонтали — используются для создания нового вектора, который назначается угловому свойству компонента `Transform`.

ОТКЛЮЧАЕМ ВЛИЯНИЕ СРЕДЫ НА ИГРОКА

Хотя для данного проекта это не имеет особого значения, в большинстве современных игр от первого лица на все элементы сцены действует модель физической среды. Она заставляет объекты отскакивать и опрокидываться; такое поведение подходит для большинства объектов, но вращение нашего игрока должно контролироваться исключительно мышью, не попадая под влияние смоделированной физической среды.

По этой причине в сценариях, где присутствует ввод с помощью мыши, к компоненту `Rigidbody` игрока обычно добавляют свойство `freezeRotation`. Поместите в сценарий `MouseLook` вот такой метод `Start()`:

```
...
void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true;
}
...
```

Проверяем, существует ли этот компонент.

`Rigidbody` — это дополнительный компонент, которым может обладать объект. Моделируемая физическая среда влияет на этот компонент и управляет объектами, к которым он присоединен.

На случай, если вы запутались и не знаете, куда вставить очередной фрагмент кода, привожу полный текст сценария. Кроме того, вы можете скачать пример проекта.

Листинг 2.7. Окончательный вид сценария MouseLook

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
    MouseXAndY = 0,
    MouseX = 1,
    MouseY = 2
  }
  public RotationAxes axes = RotationAxes.MouseXAndY;

  public float sensitivityHor = 9.0f;
  public float sensitivityVert = 9.0f;

  public float minimumVert = -45.0f;
  public float maximumVert = 45.0f;

  private float _rotationX = 0;

  void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true;
  }

  void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
      _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
      _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

      float rotationY = transform.localEulerAngles.y;

      transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
    else {
      _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
      _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

      float delta = Input.GetAxis("Mouse X") * sensitivityHor;
      float rotationY = transform.localEulerAngles.y + delta;

      transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
  }
}
```

После запуска этого сценария вы получите возможность смотреть во всех направлениях, просто перемещая указатель мыши. Великолепно! Но вы все еще стоите на месте, как орудийная башня. В следующем разделе игрок начнет перемещаться по сцене.

2.5. Компонент для клавиатурного ввода

Возможность смотреть по сторонам в ответ на перемещения указателя мыши относится к важным фрагментам элементов управления персонажем в играх от первого лица, но это только половина дела. Кроме этого, игрок должен перемещаться по сцене в ответ на клавиатурный ввод. Поэтому в дополнение к компоненту для элемента управления мышью напишем компонент для клавиатурного ввода; создайте новый компонент C# `script` с именем `FPSInput` и присоедините его к игроку (в дополнение к сценарию `MouseLook`). При этом мы на время ограничим компонент `MouseLook` только горизонтальным вращением.

СОВЕТ Рассматриваемые нами элементы управления клавиатурным вводом и вводом с помощью мыши помещены в отдельные сценарии. Вы вовсе не обязаны структурировать код подобным образом и вполне можете поместить все в единый сценарий «элементов управления игроком». Но, как правило, модульные системы (такие, как в Unity) стремятся к максимальной гибкости, и, соответственно, эффективнее всего моделировать функциональность как набор отдельных небольших компонентов.

Код, который мы писали, влиял только на ориентацию объекта, теперь же мы будем менять только его местоположение. Как показано в листинге 2.8, мы берем код вращения и вводим его в сценарий `FPSInput`, заменяя метод `Rotate()` методом `Translate()`. После щелчка на кнопке Play сцена начнет скользить вверх, а не вращаться, как раньше. Попытайтесь поменять значения параметров и посмотреть, как изменится движение (например, после того, как вы поменяете местами первое и второе числа); после некоторого количества экспериментов можно будет перейти к добавлению клавиатурного ввода.

Листинг 2.8. Код вращения из первого листинга с парой небольших изменений

```
using UnityEngine;
using System.Collections;

public class FPSInput : MonoBehaviour {
  public float speed = 6.0f;

  void Update() {
    transform.Translate(0, speed, 0);
  }
}
```

Необязательный элемент на случай, если вы захотите увеличить скорость.

Меняем метод Rotate() на метод Translate().

2.5.1. Реакция на нажатие клавиш

Код перемещения в ответ на нажатия клавиш (показанный в листинге 2.9) аналогичен коду вращения в ответ на движение указателя мыши. Мы снова используем метод `GetAxis()`, причем аналогичным образом.

Листинг 2.9. Изменение положения в ответ на нажатие клавиш

```
...
void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  transform.Translate(deltaX, 0, deltaZ);
}
```

"Horizontal" и "Vertical" — это дополнительные имена для сопоставления с клавиатурой.

Как и раньше, значения, задаваемые методом `GetAxis()`, умножаются на скорость, давая в итоге величину смещения. Но если раньше в ответ на запрос оси мы получали ответ «Mouse имя оси», то теперь это значение Horizontal или Vertical. Это абстрактные представления для параметров ввода в Unity; если в меню Edit навести указатель мыши на строку Project Settings и выбрать в открывшемся меню команду Input, откроется список абстрактных имен ввода и соответствующих им элементов управления. Клавиши с указывающими влево/вправо стрелками и с буквами A/D соответствуют имени Horizontal, в то время как клавиши со стрелками, указывающими вверх/вниз, и с буквами W/S соответствуют имени Vertical.

Обратите внимание, что значения перемещения меняются по координатам *X* и *Z*. Вы могли заметить в процессе экспериментов с методом `Translate()`, что изменение координаты *X* соответствует движению из стороны в сторону, в то время как изменение координаты *Z* означает движение вперед и назад.

Вставив в сценарий этот новый код перемещения, вы получите возможность двигать объект по сцене нажатием клавиш со стрелками или клавиш с буквами WASD. Это стандартная ситуация для большинства игр от первого лица. Сценарий, управляющий перемещениями, практически готов, осталось только внести в него некоторые изменения.

2.5.2. Независимая от скорости работы компьютера скорость перемещений

Пока вы запускаете код только на собственном компьютере, этот аспект незаметен, но на разных машинах игрок будет перемещаться с разной скоростью. Ведь более мощные компьютеры быстрее обрабатывают код и графику. В настоящее время скорость перемещения вашего игрока привязана к скорости работы компьютера. Это называется зависимостью от кадровой частоты (frame rate dependent), так как код движения зависит от частоты кадров игры.

Например, представьте, что вы запускаете деморолик на двух компьютерах, один из которых отображает на мониторе 30 кадров в секунду, а второй 60 кадров. Это означает, что на втором компьютере метод `Update()` будет вызываться в два раза чаще, двигая объект с одной и той же скоростью 6 при каждом вызове. В итоге при частоте 30 кадров в секунду скорость движения составит 180 единиц в секунду, в то время как частота 60 кадров в секунду обеспечит скорость перемещения в 360 единиц в секунду. Такая вариативность неприемлема для большинства игр.

Решить эту проблему позволяет такое редактирование кода, в результате которого мы получим *независимость от кадровой частоты* (frame rate independent). То есть скорость перемещения перестанет зависеть от частоты кадров игры. Это обеспечивается сменой скорости в соответствии с частотой кадров. Она должна уменьшаться или увеличиваться в зависимости от того, насколько быстро работает компьютер. Достигнуть такого результата нам поможет умножение скорости на переменную `deltaTime`, как показано в следующем листинге.

Листинг 2.10. Движение с независимостью от кадровой частоты благодаря переменной deltaTime

```
...
void Update() {
```

```
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  transform.Translate(deltaX * Time.deltaTime, 0, deltaZ * Time.deltaTime);
}
...
```

Это простое изменение. Класс `Time` обладает рядом свойств и методов, позволяющих регулировать время. К таким свойствам относится и `deltaTime`. Так как мы знаем, что дельта указывает на величину изменения, переменная `deltaTime` означает величину изменения во времени. А конкретно — это время между кадрами. Его величина зависит от частоты кадров (например, при частоте 30 кадров в секунду `deltaTime` составляет 1/30 секунды). Поэтому умножение скорости на эту переменную приведет к масштабированию скорости на различных компьютерах.

Теперь движение нашего персонажа одинаково на всех машинах. Но сценарий еще не завершен, ведь, перемещаясь по комнате, мы можем проходить сквозь стены.

2.5.3. Компонент CharacterController для распознавания столкновений

Назначенные объекту напрямую преобразования не затрагивают такой аспект, как распознавание столкновений. В результате персонаж начинает ходить сквозь стены. Поэтому нам требуется компонент `CharacterController`. Именно он обеспечит естественность перемещений нашего персонажа. Напомню, что в момент настройки игрока мы присоединяли этот компонент, поэтому теперь остается воспользоваться им в коде сценария `FPSInput` (листинг 2.11).

Листинг 2.11. Перемещение компонента CharacterController вместо компонента Transform

```
...
private CharacterController _charController;

void Start() {
_charController = GetComponent<CharacterController>();
}

void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  Vector3 movement = new Vector3(deltaX, 0, deltaZ);
  movement = Vector3.ClampMagnitude(movement, speed);

  movement *= Time.deltaTime;
  movement = transform.TransformDirection(movement);
  _charController.Move(movement);
}
...
```

Переменная для ссылки на компонент CharacterController.

Доступ к другим компонентам, присоединенным к этому же объекту.

Ограничим движение по диагонали той же скоростью, что и движение параллельно осям.

Преобразуем вектор движения от локальных к глобальным координатам.

Заставим этот вектор перемещать компонент CharacterController.

В этом фрагменте кода появляется несколько новых концепций. Прежде всего, это переменная для ссылки на компонент `CharacterController`. Она просто создает локальную ссылку на объект (объект в коде — не путайте его с объектами сцены); ссылаться

на этот экземпляр компонента могут разные сценарии. Изначально эта переменная пуста, поэтому, прежде чем ею пользоваться, следует указать объект, на который вы будете ссылаться с ее помощью. Именно здесь появляется метод `GetComponent()`; он возвращает другие компоненты, присоединенные к тому же объекту `GameObject`. Вместо передачи параметра в скобках мы воспользовались синтаксисом C# и определили тип в угловых скобках `<>`.

Теперь, когда у нас появилась ссылка на компонент `CharacterController`, можно вызвать для этого контроллера метод `Move()`. Мы передаем этому методу вектор тем же способом, которым код вращения в зависимости от указателя мыши использовал вектор для значений поворота. А аналогично тому, как мы ограничивали значения поворота, мы задействуем метод `Vector3.ClampMagnitude()`, чтобы ограничить модуль вектора скоростью движения; мы берем именно этот метод, потому что в противном случае движение по диагонали происходило бы быстрее движения вдоль координатных осей (нарисуйте катеты и гипотенузу прямоугольного треугольника).

Но вектор движения обладает еще одной особенностью, связанной с выбором системы координат. Вы уже сталкивались с ней при обсуждении преобразования поворота. Например, мы создадим вектор, перемещающий объект влево. Но может оказаться, что у игрока это направление не совпадает с направлением координатных осей. То есть мы говорим о локальном, а не о глобальном пространстве. Поэтому методу `Move()` следует передавать вектор движения, определенный в глобальном пространстве, а значит, нам требуется преобразование от локальных к глобальным координатам. Математически это крайне сложная операция, но, к счастью для нас, об этом позаботится Unity. Нам же достаточно вызвать метод `TransformDirection()`, чтобы, как следует из его названия, преобразовать направление.

ОПРЕДЕЛЕНИЕ Слово «transform», используемое в качестве глагола, означает преобразование от одного координатного пространства к другому (если вы не помните, что такое координатное пространство, перечитайте раздел 2.3.3). Не следует путать его с аналогичным существительным, которое означает как компонент Transform, так и изменение положения объектов в сцене. В данном случае значения терминов перекрываются, так как в основе лежит одна и та же концепция.

Запустите код. Если это еще не сделано, установите для компонента `MouseLook` вращение одновременно по горизонтали и вертикали. Вы можете без ограничений просматривать пространство вокруг себя и летать, нажимая соответствующие клавиши. Это здорово, если вам требуется игрок, умеющий летать. Но как сделать, чтобы он перемещался исключительно по земле?

2.5.4. Ходить, а не летать

Теперь, когда распознавание столкновений работает, в сценарий можно добавить силу тяжести, чтобы игрок стоял на полу. Объявим переменную `gravity` и воспользуемся ее значением для оси *Y*, как показано в следующем листинге.

Листинг 2.12. Добавление силы тяжести в код движения

```
...
public float gravity = -9.8f;
...
```

```
void Update() {
  ...
  movement = Vector3.ClampMagnitude(movement, speed);
  movement.y = gravity;  ◄——  Используем значение переменной gravity вместо нуля.
  ...
```

Теперь на игрока действует постоянная сила, тянущая его вниз. К сожалению, она не всегда направлена строго вниз, так как изображающий игрока объект может наклоняться вниз и вверх, следуя за движениями указателя мыши. К счастью, у нас есть все, чтобы исправить этот недочет, достаточно слегка поменять настройки компонента по отношению к игроку. Первым делом ограничьте компонент `MouseLook` только горизонтальным вращением. Затем добавьте этот компонент к камере и ограничьте его только вертикальным вращением. В результате на перемещения указателя мыши у вас будут реагировать два разных объекта!

Так как игрок теперь поворачивается только в горизонтальной плоскости, решилась проблема с наклоном вектора силы тяжести. При этом объект камеры является дочерним по отношению к игроку (помните, как мы установили эту связь на вкладке Hierarchy?), поэтому, обладая способностью поворачиваться в вертикальной плоскости независимо от игрока, она вращается по горизонтали вслед за ним.

СОВЕРШЕНСТВОВАНИЕ ГОТОВОГО СЦЕНАРИЯ

Воспользуйтесь методом `RequireComponent()`, чтобы проверить, присоединены ли остальные требуемые сценарию компоненты. Иногда такие компоненты являются необязательными (то есть можно сказать, что «если этот компонент присоединен, тогда...»), но бывают ситуации, когда без каких-то компонентов сценарий работать не будет. Поэтому добавьте в верхнюю часть сценария этот метод, чтобы обеспечить выполнение зависимостей, указав требуемый компонент в качестве параметра.

Кроме того, можно вставить в верхнюю часть сценария метод `AddComponentMenu()`, который добавит сценарий в меню компонентов в редакторе Unity. Достаточно указать имя элемента меню, и вы получите возможность выбирать данный сценарий в списке, открываемом щелчком на кнопке Add Component в нижней части панели Inspector. Очень удобно!

Код добавления обоих методов выглядит примерно так:

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
```

Листинг 2.13 демонстрирует полностью готовый сценарий. В дополнение к слегка скорректированной настройке компонентов игрока мы добавили игроку возможность ходить по комнате. Несмотря на наличие переменной `gravity`, вы легко можете заставить игрока летать, введя в поле Gravity на панели Inspector значение 0.

Листинг 2.13. Готовый сценарий FPSInput

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
```

```
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
  public float speed = 6.0f;
  public float gravity = -9.8f;

  private CharacterController _charController;

  void Start() {
    _charController = GetComponent<CharacterController>();
  }

  void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    Vector3 movement = new Vector3(deltaX, 0, deltaZ);
    movement = Vector3.ClampMagnitude(movement, speed);

    movement.y = gravity;

    movement *= Time.deltaTime;
    movement = transform.TransformDirection(movement);
    _charController.Move(movement);
  }
}
```

Поздравляю вас с созданием 3D-проекта! В этой главе вы получили большое количество новой информации и теперь знаете, как создать код перемещения в Unity. Но как бы ни радовал нас первый демонстрационный ролик, до готовой игры ему еще очень далеко. В плане проекта этот пункт описан как создание базовой сцены в шутере от первого лица, но какой же это шутер, если мы лишены возможности стрелять? Поэтому похвалите себя за успешно завершенный фрагмент проекта и готовьтесь к следующему шагу.

Заключение

- Трехмерное координатное пространство определяется осями *X*, *Y* и *Z*.
- Сцену создают помещенные в комнату объекты и источники света.
- Игрока в сцене от первого лица, по сути, представляет камера.
- Код движения в каждом кадре циклически повторяет небольшие преобразования.
- Элементы управления персонажем в игре от первого лица состоят из указателя мыши, отвечающего за вращение, и клавиатуры, отвечающей за перемещения.

3 Добавляем в игру врагов и снаряды

- ✓ Как игрок и его враги получают возможность целиться и стрелять.
- ✓ Попадания и реакция на них.
- ✓ Заставляем врагов перемещаться.
- ✓ Порождение новых объектов сцены.

Ролик, демонстрирующий перемещения объекта, который вы создали в предыдущей главе, выглядит здорово, но до полноценной игры ему далеко. Давайте попробуем превратить его в шутер от первого лица. Для этого нам не хватает возможности стрелять, а также врагов, которых требуется поразить. Начнем мы с написания сценариев, превращающих нашего игрока в стрелка. После этого мы заполним сцену врагами, добавив в числе прочего код, позволяющий бесцельно перемещаться по сцене и реагировать на попадание. В заключение мы дадим врагам возможность отстреливаться, кидая в игрока огненные шары. Написанные в главе 2 сценарии при этом редактироваться не будут; мы просто добавим в проект новые сценарии с дополнительными функциональными возможностями.

Я выбрал для этого проекта шутер от первого лица по двум причинам. Во-первых, подобные игры популярны, ведь людям нравится стрелять. Во-вторых, я учел набор приемов, с которыми вы можете ознакомиться в процессе работы. Создание такой игры дает возможность изучить несколько фундаментальных концепций трехмерного моделирования. Например, вы узнаете, что такое *бросание луча* (raycasting). Подробно особенности этого приема я объясню чуть позже, а пока достаточно информации о том, что он позволяет решать множество различных задач в трехмерном моделировании и применяется во множестве ситуаций, из которых интуитивно наиболее понятной является реализация стрельбы.

Создание блуждающих целей, которые требуется поразить, дает нам замечательный повод исследовать код контролируемых компьютером персонажей, а также техники

отправки сообщений и порождения объектов. Более того, поведение блуждающих целей — это еще один аспект, в моделировании которого нам пригодится прием испускания луча, так что мы познакомимся с еще одним вариантом его применения. Также широко применим демонстрируемый в рамках этого проекта подход к рассылке сообщений. В следующих главах вы встретите другие варианты применения указанных приемов. Впрочем, даже в рамках одной главы они используются в различных ситуациях.

По сути дела, мы добавляем в проект по одной функциональной возможности за раз, причем на каждом этапе в игру можно играть, но всегда остается ощущение, что каких-то элементов не хватает. Вот план нашей будущей работы:

1. Написать код, позволяющий игроку стрелять.
2. Создать статичные цели, реагирующие на попадание.
3. Заставить цели перемещаться по сцене.
4. Вызвать автоматическое появление новых блуждающих целей.
5. Дать возможность целям/врагам кидать в игрока огненные шары.

ПРИМЕЧАНИЕ Проект, которым мы будем заниматься в этой главе, предполагает наличие демонстрационного ролика с перемещающимся по сцене персонажем. Этот ролик создавался в главе 2, но если вы ее пропустили, скачайте соответствующие файлы с сайта книги.

3.1. Стрельба путем бросания лучей

Первой дополнительной функциональной возможностью, которую мы добавим в наш демонстрационный ролик, станет стрельба. Умение оглядываться по сторонам и перемещаться в шутере от первого лица является, без сомнения, решающим, но игра не начнется, пока игроки не смогут влиять на окружающее пространство и показывать свое мастерство. Стрельба в трехмерных играх реализуется несколькими способами, наиболее важным из которых является бросание лучей.

3.1.1. Что такое бросание лучей?

Название приема подразумевает, что вам предстоит бросать лучи. Но что именно в данном случае подразумевается под словом *луч*?

ОПРЕДЕЛЕНИЕ *Лучом* (ray) в сцене называется воображаемая, невидимая линия, начинающаяся в некоторой точке и распространяющаяся в определенном направлении.

Прием бросания луча иллюстрирует рис. 3.1. Вы формируете луч и затем определяете, с чем он пересекается. Подумайте, что происходит, когда вы стреляете из пистолета: пуля вылетает из точки, в которой находится пистолет, и летит по прямой вперед, пока не столкнется с каким-нибудь препятствием. Луч можно сравнить с путем пули, а бросание луча аналогично выстрелу из пистолета и наблюдению за тем, куда попадет пуля.

Думаю, очевидно, что реализация метода бросания лучей зачастую требует сложных математических расчетов. Трудно не только просчитать пересечение линии с трехмерной плоскостью, но и сделать это для всех сеточных объектов в сцене (напоминаю, что

сеточным объектом называется трехмерный видимый объект, сконструированный из множества соединенных друг с другом линий и фигур). К счастью, Unity берет на себя все сложные математические расчеты, оставляя вам заботу о задачах более высокого уровня, например о месте и причине появления лучей.

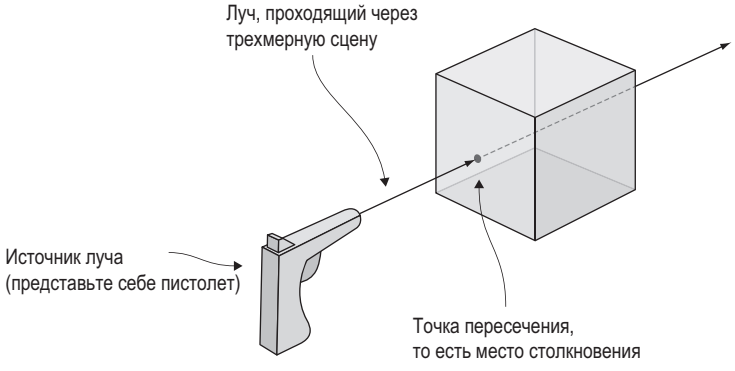


Рис. 3.1. Луч представляет собой воображаемую линию, и при бросании луча выясняется, с чем этот луч пересекается

В текущем проекте ответом на второй вопрос (почему возникает луч) является имитация выпускания пули. В шутерах от первого лица луч, как правило, начинается из места, где располагается камера, и распространяется по центру ее поля зрения. Иными словами, вы проверяете наличие объекта непосредственно перед камерой — в Unity есть соответствующие команды. Рассмотрим их более подробно.

3.1.2. Имитация стрельбы командой ScreenPointToRay

Итак, мы реализуем стрельбу проецированием луча, начинающегося в месте нахождения камеры и распространяющегося по центральной линии ее поля зрения. Проецирование луча по центральной линии поля зрения камеры представляет собой частный случай действия, которое называется *выбором с помощью мыши*.

ОПРЕДЕЛЕНИЕ *Выбор с помощью мыши* (mouse picking) означает действие по выбору в трехмерной сцене точки, непосредственно попадающей под указатель мыши.

Эта операция в Unity осуществляется методом `ScreenPointToRay()`. Происходящее иллюстрирует рис. 3.2. Метод создает луч, начинающийся с камеры, и проецирует его по линии, проходящей через указанные экранные координаты. Обычно при выборе с помощью мыши используются координаты указателя, но в шутерах от первого лица эту роль играет центр экрана. Появившийся луч передается методу `Physics.Raycast()`, который и выполняет его «бросание».

Напишем код, использующий только что рассмотренные нами методы. Создайте в редакторе Unity новый компонент `C# script` и присоедините его к камере (а не к объекту, представляющему игрока). Добавьте в него код следующего листинга.

В листинге следует обратить внимание на несколько моментов. Во-первых, компонент `Camera` вызывается в методе `Start()` совсем как компонент `CharacterController` в предыдущей главе. Остальная часть кода помещена в метод `Update()`, так как положение мыши нам требуется проверять снова и снова. Метод `Input.GetMouseButtonDown()` в зависимости от того, нажимается ли кнопка мыши, возвращает значения `true` и `false`. Помещение этой команды в условную инструкцию означает, что указанный код выполняется только после щелчка кнопкой мыши. Мы хотим, чтобы выстрел возникал по щелчку мыши, именно поэтому условная инструкция проверяет ее состояние.

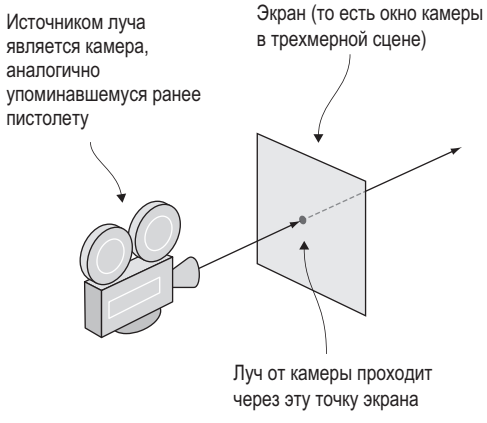


Рис. 3.2. Метод ScreenPointToRay() проецирует луч от камеры через указанные экранные координаты

Листинг 3.1. Сценарий RayShooter, присоединяемый к камере

```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent();
  }

  void Update() {
    if (Input.GetMouseButtonDown(0)) {
      Vector3 point = new Vector3(
        _camera.pixelWidth/2, _camera.pixelHeight/2, 0);
      Ray ray = _camera.ScreenPointToRay(point);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) {
        Debug.Log("Hit " + hit.point);
      }
    }
  }
}
```

Доступ к другим компонентам, присоединенным к этому же объекту.

Реакция на нажатие кнопки мыши.

Середина экрана — это половина его ширины и высоты.

Создание в этой точке луча методом ScreenPointToRay().

Загружаем координаты точки, в которую попал луч.

Испущенный луч заполняет информацией переменную, на которую имеется ссылка.

Вектор создается, чтобы определить для луча экранные координаты (напоминаю, что вектором называются несколько связанных друг с другом чисел, хранимых как единое целое). Параметры pixelWidth и pixelHeight дают нам размер экрана. Определить его центр можно, разделив эти значения пополам. Хотя координаты экрана являются двумерными, то есть у нас есть только размеры по вертикали и горизонтали, а глубина отсутствует, вектор `Vector3` все равно создается, так как метод `ScreenPointToRay()` требует данных этого типа (возможно, потому, что расчет луча включает в себя операции с трехмерными векторами). Вызванный для этого набора координат метод `ScreenPointToRay()` дает нам объект `Ray` (программный, а не игровой объект; эти два объекта часто путают).

Затем луч передается в метод `Raycast()`, причем это не единственный передаваемый в этот метод объект. Есть также структура данных `RaycastHit`; она представляет собой набор информации о пересечении луча, в том числе о точке, в которой возник луч, и об объекте, с которым он столкнулся. Используемый в данном случае синтаксис языка C# гарантирует, что структура данных, с которой работает команда, является тем же объектом, существующим вне команды, в противоположность ситуациям, когда в разных областях действия функции используются разные копии объекта.

В конце мы вызываем метод `Physics.Raycast()`, который проверяет место пересечения рассматриваемого луча, собирает информацию об этом пересечении и возвращает значение `true` в случае столкновения луча с препятствием. Так как возвращаемое значение принадлежит типу `Boolean`, метод можно поместить в инструкцию проверки условия, совсем как метод `Input.GetMouseButtonDown()` чуть раньше.

Пока что на пересечения код реагирует консольным сообщением с координатами точки, в которой луч столкнулся с препятствием (значения *X*, *Y*, *Z* мы обсуждали в главе 2). Но понять, в каком именно месте это произошло, или показать, где находится центр экрана (то есть место, через которое проходит луч), практически нереально. Поэтому давайте добавим в сцену визуальные индикаторы.

3.1.3. Добавление визуальных индикаторов для прицеливания и попаданий

Теперь нам нужно добавить в сцену два вида визуальных индикаторов: точку прицеливания в середине экрана и метку в месте столкновения луча с препятствием. Второе в случае шутера от первого лица лучше показывать как дырки от пуль, но мы пока будем обозначать это место сферой (и воспользуемся *сопрограммой*, через секунду убирающей сферу). Рисунок 3.3 демонстрирует то, что вы увидите в сцене.

ОПРЕДЕЛЕНИЕ *Сопрограммы* (coroutines) в Unity выполняются параллельно программе в течение некоторого времени; этим они отличаются от большинства функций, заставляющих программу ждать окончания своей работы.

Начнем с добавления индикаторов в точки попадания. Готовый сценарий показан в листинге 3.2. Подвигайтесь по сцене, стреляя; индикаторы в виде сфер выглядят довольно забавно!

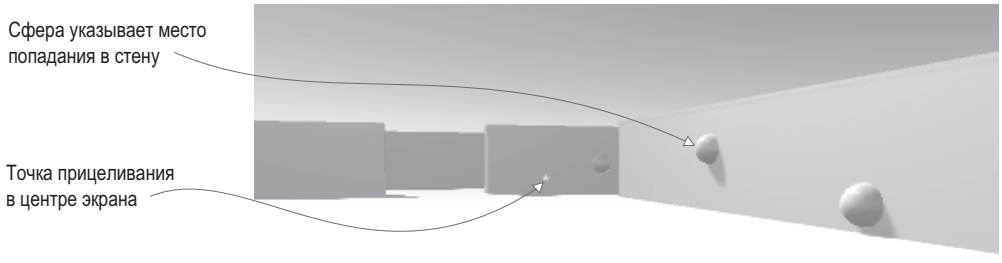


Рис. 3.3. Многократные выстрелы после добавления индикаторов прицеливания и попаданий

Листинг 3.2. Сценарий RayShooter после добавления индикаторных сфер

```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent();
  }

  void Update() {
    if (Input.GetMouseButtonDown(0)) {
      Vector3 point = new Vector3(
        _camera.pixelWidth/2, _camera.pixelHeight/2, 0);
      Ray ray = _camera.ScreenPointToRay(point);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) {
        StartCoroutine(SphereIndicator(hit.point));
      }
    }
  }

  private IEnumerator SphereIndicator(Vector3 pos) {
    GameObject sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    sphere.transform.position = pos;

    yield return new WaitForSeconds(1);

    Destroy(sphere);
  }
}
```

Эта функция по большей части содержит знакомый нам код бросания луча из листинга 3.1.

Запуск сопрограммы в ответ на попадание.

Сопрограммы пользуются функциями IEnumerator.

Ключевое слово yield указывает сопрограмме, когда следует остановиться.

Удаляем этот GameObject и очищаем память.

Из нового у нас появился метод `SphereIndicator()` и однострочная модификация существующего метода `Update()`. Новый метод создает сферу в указанной точке сцены и через секунду удаляет ее. Вызов метода `SphereIndicator()` внутри кода испускания луча гарантирует появление визуальных индикаторов точно в местах попадания.

Данная функция определена с помощью интерфейса IEnumerator, связанного с концепцией сопрограмм.

С технической точки зрения сопрограммы не асинхронны (асинхронные операции не останавливают выполнение остальной части кода; представьте, к примеру, скачивание изображения в сценарии веб-сайта), но продуманное применение перечислений в Unity заставляет сопрограммы вести себя аналогично асинхронным функциям. Секретным компонентом сопрограммы является ключевое слово `yield`, временно прерывающее ее работу, возвращающее управление основной программе и в следующем кадре возобновляющее сопрограмму с прерванной точки. В результате создается впечатление, что сопрограммы работают в фоновом режиме.

Как следует из ее имени, функция `StartCoroutine()` запускает сопрограмму. После этого она начинает работать до завершения выполнения, периодически делая паузы. Обратите внимание на небольшую, но важную деталь. Переданный в `StartCoroutine()` метод имеет набор скобок, следующий за именем. Такой синтаксис означает, что вы не передаете имя функции, а вызываете ее. И эта функция работает, пока не встретится команда `yield`. После этого ее выполнение на время прервется.

Функция `SphereIndicator()` создает сферу в определенной точке, останавливается после инструкции `yield` и после возобновления сопрограммы удаляет сферу. Продолжительность паузы контролируется значением, которое возвращается в момент появления инструкции `yield`. В сопрограммах допустимо несколько типов возвращаемых значений, но проще всего в явном виде вернуть время ожидания. Возвращая `WaitForSeconds(1)`, мы заставляем сопрограмму остановить работу на одну секунду. Создание сферы, секундная остановка и разрушение сферы — такая последовательность дает нам временный визуальный индикатор.

Код для таких индикаторов был дан в листинге 3.2. Но нам требуется также точка прицеливания в центре экрана. Она создается в следующем листинге.

Листинг 3.3. Визуальный индикатор для точки прицеливания

```
...
void Start() {
  _camera = GetComponent();

  Cursor.lockState = CursorLockMode.Locked;
  Cursor.visible = false;
}

void OnGUI() {
  int size = 12;
  float posX = _camera.pixelWidth/2 - size/4;
  float posY = _camera.pixelHeight/2 - size/2;
  GUI.Label(new Rect(posX, posY, size, size), "*");
}
...
```

Скрываем указатель мыши в центре экрана.

Команда GUI.Label() отображает на экране символ.

Еще один новый метод, добавленный в класс `RayShooter`, называется `OnGUI()`. В Unity возможна как базовая, так и усовершенствованная система пользовательского интерфейса (UI). Так как базовая система обладает множеством ограничений, в следующих

главах мы построим более гибкий, усовершенствованный пользовательский интерфейс, но пока нам проще отображать точку в центре экрана средствами базового интерфейса. Любой сценарий класса `MonoBehaviour` автоматически реагирует как на методы `Start()` и `Update()`, так и на функцию `OnGUI()`. Эта функция запускается в каждом кадре после визуализации трехмерной сцены, прорисовывая поверх сцены дополнительные элементы (представьте себе бумажные этикетки, наклеенные на нарисованный пейзаж).

ОПРЕДЕЛЕНИЕ *Визуализацией* (rendering) называется работа компьютера по прорисовыванию пикселов трехмерной сцены. Хотя сцена задается с помощью координат *X*, *Y* и *Z*, монитор отображает двумерную сетку цветных пикселов. Соответственно, для показа трехмерной сцены компьютер должен рассчитать цвет всех пикселов двумерной сетки; работа этого алгоритма и называется визуализацией.

Код внутри функции `OnGUI()` определяет двумерные координаты для отображения (слегка смещенные с учетом размера метки) и затем вызывает метод `GUI.Label()`. Этот метод отображает текстовую метку; так как переданная ему строка состоит из символа звездочки (`*`), именно он появляется в центре экрана. С ним прицеливаться в нашей будущей игре станет намного проще! Кроме того, листинг 3.3 добавляет в метод `Start()` настройки указателя мыши, а именно возможность управлять его видимостью и возможность блокировки. В принципе, сценарий прекрасно работает и без этих настроек, но они делают элементы управления более удобными в использовании. Указатель мыши все время будет располагаться в центре экрана, а чтобы не заслонять обзор, мы сделаем его невидимым. Он будет появляться только при нажатии клавиши Esc.

ВНИМАНИЕ Помните, что вы в любой момент можете снять блокировку с указателя мыши, нажав клавишу Esc. При заблокированном указателе щелкнуть на кнопке Play и остановить игру невозможно.

Итак, код, управляющий поведением игрока, готов. Фактически мы завершили работу над взаимодействиями игрока со сценой, но у нас все еще отсутствуют цели.

3.2. Создаем активные цели

Возможность стрелять — это замечательно, но в данный момент стрелять нашему игроку не во что. Поэтому нам нужно создать целевой объект и присоединить к нему сценарий, программирующий реакцию на попадание. Точнее, мы слегка отредактируем код стрельбы, чтобы получать уведомления о поражении цели, а присоединенный к целевому объекту сценарий будет запускаться в ответ на такое уведомление.

3.2.1. Определяем точку попадания

Первым делом нам нужен объект, который послужит мишенью. Создайте куб (выбрав в меню GameObject команду 3D Object, а затем вариант Cube) и поменяйте его размер по вертикали, введя в поле *Y* для преобразования Scale значение 2. В полях *X* и *Z* оставьте значение 1. Поместите новый объект в точку с координатами 0, 1, 0, чтобы он стоял на полу в центре комнаты, и присвойте ему имя `Enemy`. Создайте сценарий с именем ReactiveTarget и присоедините его к объекту. Скоро мы напишем для этого сценария код, но пока оставьте его как есть; мы создали этот файл, так как без него код из следующего

листинга компилироваться не будет. Вернитесь к сценарию `RayShooter.cs` и отредактируйте код испускания луча в соответствии с показанным листингом. Запустите новый вариант кода и попробуйте выстрелить в цель — вместо сферического индикатора на консоли появится отладочное сообщение.

Листинг 3.4. Распознавание попаданий в цель

```
...
if (Physics.Raycast(ray, out hit)) {
  GameObject hitObject = hit.transform.gameObject;
  ReactiveTarget target = hitObject.GetComponent<ReactiveTarget>();
  if (target != null) {
    Debug.Log("Target hit");
  } else {
    StartCoroutine(SphereIndicator(hit.point));
  }
}
...
```

Получаем объект, в который попал луч.

Проверяем наличие у этого объекта компонента ReactiveTarget.

Обратите внимание, что вы получаете объект, с которым пересекся луч, совсем так же, как получали координаты для сферических индикаторов. С технической точки зрения вам возвращается вовсе не игровой объект, а попавший под удар компонент `Transform`. Далее вы получаете доступ к объекту `gameObject` как к свойству класса `transform`.

Затем к этому объекту применяется метод `GetComponent()`, проверяющий, является ли он активной целью (то есть присоединен ли к нему сценарий `ReactiveTarget`). Как вы уже видели раньше, этот метод возвращает компоненты определенного типа, присоединенные к объекту `GameObject`. Если таковые отсутствуют, не возвращается ничего. Поэтому остается проверить, было ли возвращено значение null, и написать два варианта кода, которые будут запускаться в зависимости от результата проверки.

Если пораженным объектом оказывается активная цель, вместо запуска сопрограммы для сферических индикаторов код отображает отладочное сообщение. Теперь нам нужно проинформировать целевой объект о попадании, чтобы он смог отреагировать на это событие.

3.2.2. Уведомляем цель о попадании

В коде нужно поменять всего одну строку, как показано в следующем листинге.

Листинг 3.5. Отправка сообщения целевому объекту

```
...
if (target != null) {
  target.ReactToHit();
} else {
  StartCoroutine(SphereIndicator(hit.point));
}
...
```

Вызов метода для мишени вместо генерации отладочного сообщения.

Теперь отвечающий за стрельбу код вызывает связанный с мишенью метод, который нам нужно написать. Введите в сценарий `ReactiveTarget` код следующего листинга. При попадании мишень будет опрокидываться и исчезать, как показано на рис. 3.4.

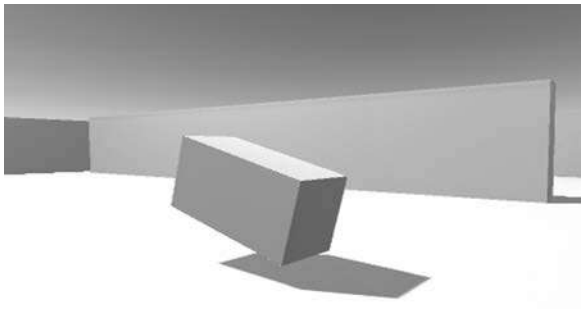


Рис. 3.4. Целевой объект, падающий после попадания

Листинг 3.6. Сценарий ReactiveTarget, реализующий смерть врага при попадании

```
using UnityEngine;
using System.Collections;

public class ReactiveTarget : MonoBehaviour {

  public void ReactToHit() {
    StartCoroutine(Die());
  }

  private IEnumerator Die() {
    this.transform.Rotate(-75, 0, 0);

    yield return new WaitForSeconds(1.5f);

    Destroy(this.gameObject);
  }
}
```

Метод, вызванный сценарием стрельбы.

Опрокидываем врага, ждем 1,5 секунды и уничтожаем его.

Объект может уничтожать сам себя точно так же, как любой другой объект.

Большая часть кода должна быть вам уже знакома по предыдущим сценариям, поэтому рассматривать его мы будем совсем кратко. Первым делом мы определяем метод `ReactToHit()`, так как именно он вызывается в сценарии стрельбы. Этот метод запускает сопрограмму, аналогичную коду для сферических индикаторов, но на этот раз она призвана манипулировать объектом этого же сценария, а не создавать отдельный объект. Такие выражения, как `this.gameObject`, относятся к объекту `GameObject`, к которому присоединен данный сценарий (ключевое слово `this` указывать необязательно, то есть можно ссылаться просто на `gameObject`).

Первая строка сопрограммы заставляет мишень опрокинуться. Как обсуждалось в главе 2, преобразование вращения можно определить как угол поворота относительно каждой из трех осей, *X*, *Y* и *Z*. Так как объект не должен поворачиваться из стороны в сторону, оставьте координатам *Y* и *Z* значение 0, а угол поворота назначьте координате *X*.

ПРИМЕЧАНИЕ Преобразование происходит мгновенно, но, возможно, вы предпочитаете видеть, как опрокидываются мишени. После более детального знакомства с методами моделирования, возможно, вы захотите воспользоваться анимацией по начальной и конечной точкам (tweening), позволяющей смоделировать плавное движение объектов.

Во второй строке метода фигурирует ключевое слово `yield`, останавливающее выполнение сопрограммы и возвращающее количество секунд, через которое она возобновит свою работу. В последней строке функции игровой объект уничтожается. Функция `Destroy(this.gameObject)` вызывается после времени ожидания точно так же, как раньше мы вызывали код `Destroy(sphere)`.

ВНИМАНИЕ Аргумент функции `Destroy()` в данном случае должен выглядеть как `this.gameObject`, а не просто `this`! Не путайте эти два случая; само по себе ключевое слово `this` относится к компоненту данного сценария, в то время как `this.gameObject` означает объект, к которому присоединен данный сценарий.

Теперь наша мишень реагирует на попадание! Но больше она ничего не делает. Давайте добавим ей дополнительные модели поведения, чтобы превратить ее в настоящего врага.

3.3. Базовый искусственный интеллект для перемещения по сцене

Статичная цель не очень интересна, поэтому давайте напишем код, который заставит врагов перемещаться по сцене. Такой код является одним из простейших примеров искусственного интеллекта (Artificial Intelligence, AI). Этот термин относится к сущностям, поведение которых контролируется компьютером. В данном случае такой сущностью служит враг в нашей игре, но в реальной жизни это может быть, например, робот.

3.3.1. Диаграмма работы базового искусственного интеллекта

Существует множество подходов к реализации искусственного интеллекта (это одна из основных областей исследований ученых, работающих в области теории вычислительных машин и систем), но для наших целей подойдет достаточно простой вариант. По мере накопления опыта и усложнения создаваемых вами игр вы, скорее всего, захотите познакомиться с другими способами реализации AI. Но пока мы ограничимся процессом, показанным на рис. 3.5.

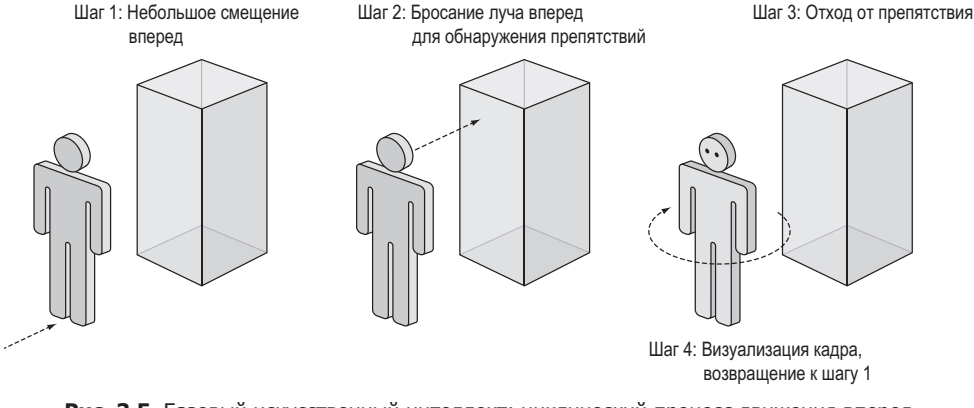


Рис. 3.5. Базовый искусственный интеллект: циклический процесс движения вперед с обходом препятствий

В каждом кадре код искусственного интеллекта сканирует окружающее пространство, чтобы определить свое дальнейшее поведение. Если на пути появляется препятствие, враг меняет направление движения. При этом независимо от поворотов он неуклонно двигается вперед. В итоге враг будет в разных направлениях ходить по комнате, непрерывно перемещаясь вперед и поворачиваясь, чтобы не сталкиваться со стенами.

Сам код будет иметь знакомый вам вид, так как движение врагов вызывают те же команды, что и движение игрока. Кроме того, в коде вы снова увидите метод бросания луча, но уже в другом контексте.

3.3.2. «Поиск» препятствий методом бросания лучей

Как вы узнали во введении к данной главе, бросание луча представляет собой прием, позволяющий решать различные задачи трехмерного моделирования. Первым делом на ум приходит имитация выстрелов, но, кроме того, этот прием позволяет сканировать окружающее пространство. А так как в данном случае нам требуется решить именно эту задачу, код испускания лучей попадет в наш код для AI.

Раньше мы создавали луч, который брал начало из камеры, так как именно она служит глазами игрока. На этот раз луч будет начинаться в месте расположения врага. В первом случае луч проходил через центр экрана, теперь же он будет распространяться перед персонажем, как показано на рис. 3.6. Затем аналогично тому, как код стрельбы использовал информацию из структуры `RaycastHit`, чтобы определить, поражена ли какая-нибудь цель и где она находится, код AI задействует эту же информацию, чтобы определить наличие препятствия по ходу движения и расстояние до этого препятствия.

Рис. 3.6. Обнаружение препятствий методом бросания луча

Основным различием между лучом, бросаемым в случае выстрела, и лучом в коде AI является радиус распознаваемого пространства. При стрельбе мы обходились бесконечно тонким лучом, в то время как у луча для AI будет большое сечение. Соответственно, мы воспользуемся методом `SphereCast()` вместо метода `Raycast()`. Все дело в том, что

пули имеют маленький размер, в то время как, проверяя наличие препятствий перед персонажем, мы должны учитывать ширину самого персонажа.

Создайте сценарий с именем `WanderingAI` и присоедините его к целевому объекту (вместе со сценарием `ReactiveTarget`). Введите в него код из следующего листинга. Запустите код, и вы увидите, как враг перемещается по комнате; при этом вы можете выстрелить в него, и он среагирует на попадание так же, как и раньше.

Листинг 3.7. Базовый сценарий WanderingAI

```
using UnityEngine;
using System.Collections;

ublic class WanderingAI : MonoBehaviour {
  public float speed = 3.0f;
  public float obstacleRange = 5.0f;

  void Update() {
    transform.Translate(0, 0, speed * Time.deltaTime);

    Ray ray = new Ray(transform.position, transform.forward);
    RaycastHit hit;
    if (Physics.SphereCast(ray, 0.75f, out hit)) {
      if (hit.distance < obstacleRange) {
        float angle = Random.Range(-110, 110);
        transform.Rotate(0, angle, 0);
      }
    }
  }
}
```

Значения для скорости движения и расстояния, с которого начинается реакция на препятствие.

Непрерывно движемся вперед в каждом кадре, несмотря на повороты.

Бросаем луч с описанной вокруг него окружностью.

Луч находится в том же положении и нацеливается в том же направлении, что и персонаж.

Поворот с наполовину случайным выбором нового направления.

В листинге появилась пара новых переменных. Одна — для скорости движения врага, а вторая — для расстояния, на котором враг начинает реагировать на препятствие. Затем мы добавили внутрь метода `Update()` метод `Translate()`, обеспечив непрерывное движение вперед (в том числе воспользовавшись переменной `deltaTime` для движения, не зависящего от частоты кадров). Кроме того, в метод `Update()` помещен код бросания луча, во многом напоминающий написанный нами ранее сценарий поражения целей. В данном случае прием бросания луча применяется для осмотра сцены, а не для стрельбы. Луч создается на базе положения врага и направления его движения, а не на базе камеры.

Как уже упоминалось, для расчета траектории луча применяется метод `Physics.SphereCast()`, в качестве параметра принимающий радиус окружности, в пределах которой будут распознаваться пересечения. Во всех же прочих отношениях он аналогичен методу `Physics.Raycast()`. Сходство наблюдается в способе получения информации о столкновении луча, способе проверки пересечений и применении свойства `distance`, гарантирующего, что враг среагирует только тогда, когда приблизится к препятствию.

Как только враг окажется перед стеной, код поменяет направление его движения наполовину случайным образом. Я использую словосочетание «наполовину случайным», так как значения ограничены минимумом и максимумом, имеющими смысл в данной ситуации. Мы прибегаем к методу `Random.Range()`, которым Unity позволяет получить

случайное значение в указанном диапазоне. В нашем случае ограничения немного превышают величины поворота влево и вправо, что дает персонажу возможность развернуться на достаточный угол, чтобы избежать препятствий.

3.3.3. Слежение за состоянием персонажа

Текущее поведение врага имеет один недостаток. Движение вперед продолжается даже после попадания в него пули. Ведь метод `Translate()` запускается в каждом кадре вне зависимости от обстоятельств. Внесем в код небольшие изменения, позволяющие следить за тем, жив персонаж или мертв. Говоря техническим языком, мы хотим отслеживать «живое» состояние персонажа. Код, по-разному реагирующий на разные состояния, представляет собой паттерн, распространенный во многих областях программирования, а не только в AI. Более сложные реализации этого паттерна называются *конечными автоматами*.

ОПРЕДЕЛЕНИЕ *Конечным автоматом* (Finite State Machine, FSM) называется структура кода, в которой отслеживается текущее состояние объекта, существуют четко определенные переходы между состояниями и код ведет себя по-разному в зависимости от состояния.

Разумеется, речь о настоящей реализации конечного автомата не идет, но нет ничего странного в том, что при обсуждении искусственного интеллекта упоминаются основы FSM. Конечный автомат обладает множеством состояний для различных вариантов поведения сложного искусственного интеллекта, в случае же базового искусственного интеллекта достаточно отследить, жив персонаж или уже нет. В следующем листинге в начальную часть сценария добавляется логическая переменная `_alive`, значение которой будет периодически проверяться в коде. Благодаря этим проверкам код движения запускается только для живого персонажа.

Листинг 3.8. Сценарий WanderingAI после добавления «живого» состояния

```
...
private bool _alive; ◄── Логическая переменная для слежения за состоянием персонажа.

void Start() {
  _alive = true; ◄── Инициализация этой переменной.
}

void Update() {
  if (_alive) { ◄── Движение начинается только в случае живого персонажа.
    transform.Translate(0, 0, speed * Time.deltaTime);
    ...
  }
}

public void SetAlive(bool alive) {
  _alive = alive;
}
...
```

Открытый метод, позволяющий внешнему коду воздействовать на «живое» состояние.

Теперь сценарий `ReactiveTarget` может сообщить сценарию `WanderingAI`, в каком состоянии находится враг.

Листинг 3.9. Сценарий ReactiveTarget сообщает сценарию WanderingAI, когда наступает смерть

```
...
public void ReactToHit() {
  WanderingAI behavior = GetComponent();
  if (behavior != null) {
    behavior.SetAlive(false);
  }
  StartCoroutine(Die());
}
...
```

Проверяем, присоединен ли к персонажу сценарий WanderingAI; он может и отсутствовать.

СТРУКТУРА КОДА ДЛЯ AI

Приведенный в этой главе код для AI помещен в один класс, так что изучить и понять его достаточно просто. Такая структура кода совершенно нормальна для простого искусственного интеллекта, поэтому не бойтесь, что вы сделали что-то не так и что на самом деле требуется более сложная структура. Для усложненных реализаций искусственного интеллекта (например, в игре с широким диапазоном интеллектуальных персонажей) облегчить разработку AI поможет более надежная структура кода.

Как упоминалось в главе 1 при сравнении компонентной структуры с наследованием, иногда фрагменты кода для AI имеет смысл помещать в отдельные сценарии. Это позволит сочетать и комбинировать компоненты, генерируя уникальное поведение для каждого персонажа. Подумайте, в чем ваши персонажи одинаковы, а чем различаются. Именно эти различия помогут вам при разработке архитектуры кода. К примеру, если в игре есть враги, сломя голову несущиеся на персонажа, и враги, тихо подкрадывающиеся в тени, имеет смысл создать для них разные компоненты перемещения и написать отдельные сценарии для перемещения бегом и перемещения крадучись.

Точная структура кода для AI зависит от особенностей конкретной игры; единственного «правильного» варианта в данном случае не существует. Средства Unity позволят вам легко спроектировать гибкую архитектуру.

3.4. Увеличение количества врагов

В настоящее время в сцене присутствует всего один враг, и после его смерти сцена становится пустой. Давайте заставим игру порождать врагов, сделав так, чтобы после смерти существующего врага сразу появлялся новый. В Unity это легко делается с помощью механизма *шаблонов экземпляров* (prefabs).

3.4.1. Что такое шаблон экземпляров?

Шаблоны экземпляров предлагают гибкий подход к визуальному определению интерактивных объектов. В двух словах, это полностью сформированный игровой объект (с уже присоединенными и настроенными компонентами), существующий не внутри конкретной сцены, а в виде ресурса, который может быть скопирован в любую сцену. Копирование может осуществляться вручную, чтобы гарантировать идентичность объектов-врагов (или других объектов) в каждой сцене. Но куда важнее то, что эти шаблоны могут порождаться кодом; поместить копии объектов в сцену можно не только вручную в визуальном редакторе, но и с помощью команд сценария.

ОПРЕДЕЛЕНИЕ *Ресурсом* (asset) называется любой файл, отображаемый на вкладке Project; это могут быть двумерные изображения, трехмерные модели, файлы с кодом, сцены и т. п. Термин «ресурс» вскользь упоминался в главе 1, но до текущего момента мы не акцентировали на нем внимание.

Термин *экземпляр* (instance) относится также к создаваемым на основе класса объектам кода. Попытайтесь не запутаться в терминологии; словосочетание «шаблон экземпляра» относится к игровому объекту, существующему вне сцены, в то время как экземпляром называется помещенная в сцену копия объекта.

3.4.2. Создание шаблона врага

Конструирование шаблона начинается с создания объекта. Так как мы планируем получить шаблон из объекта-врага, этот шаг уже сделан. Теперь нужно перетащить строку с названием этого объекта с вкладки Hierarchy на вкладку Project, как показано на рис. 3.7. Объект автоматически сохранится в качестве шаблона. На панели Hierarchy его имя будет выделено синим цветом, означающим, что теперь он связан с шаблоном экземпляров. Редактирование этого шаблона (например, добавление компонентов) осуществляется посредством редактирования объекта сцены, после чего в меню GameObject выбирается команда Apply Changes To Prefab. Но сейчас данный объект в сцене уже не нужен (мы собираемся порождать новые шаблоны, а не пользоваться уже имеющимся экземпляром), поэтому его следует удалить.

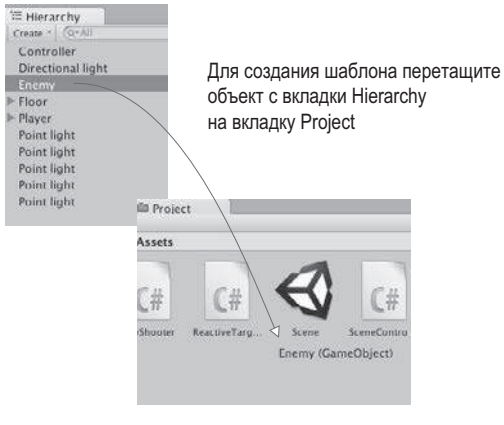


Рис. 3.7. Процесс получения шаблона экземпляров

ВНИМАНИЕ Интерфейс для работы с шаблонами экземпляров не слишкомудобен, а соотношения между шаблонами и их экземплярами в сценах порой достаточно нестабильны. К примеру, зачастую требуется перетащить шаблон в сцену для последующего редактирования, а после завершения этого процесса удалить объект. В главе 1 я упоминал об этом как о недостатке Unity и надеюсь, что в следующих версиях последовательность действий будет усовершенствована.

Теперь у нас есть шаблон для заполнения сцены врагами, поэтому давайте напишем код, создающий его экземпляры.

3.4.3. Экземпляры невидимого компонента SceneController

Хотя сам по себе шаблон в сцене отсутствует, нам нужен некий объект, к которому будет присоединяться код, порождающий врагов. Поэтому мы создадим пустой игровой объект и добавим к нему сценарий, при этом сам объект останется невидимым.

СОВЕТ Использование пустого объекта `GameObject` для присоединения к нему компонентов сценариев является распространенной практикой при разработке в Unity. Этот трюк применяется для решения абстрактных задач и нереализуем при работе с конкретными объектами сцены. Unity-сценарии присоединяются только к видимым объектам, но не для каждой задачи это имеет смысл.

Выберите в меню GameObject команду Create Empty и присвойте новому объекту имя Controller. Убедитесь, что он находится в точке с координатами 0, 0, 0 (с технической точки зрения местоположение этого объекта не имеет значения, так как его все равно не видно, но, поместив его в начало координат, вы облегчите себе жизнь, если в будущем решите использовать его в цепочке наследования). Создайте сценарий `SceneController`, показанный в следующем листинге.

Листинг 3.10. Сценарий SceneController, порождающий экземпляры врагов

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
  [SerializeField] private GameObject enemyPrefab;
  private GameObject _enemy;

  Void Update() {

    if (_enemy == null) {
      _enemy = Instantiate(enemyPrefab) as GameObject;
      _enemy.transform.position = new Vector3(0, 1, 0);
      float angle = Random.Range(0, 360);
      _enemy.transform.Rotate(0, angle, 0);
    }
  }
}
```

Сериализованная переменная для связи с объектом-шаблоном.

Закрытая переменная для слежения за экземпляром врага в сцене.

Порождаем нового врага, только если враги в сцене отсутствуют.

Метод, копирующий объект-шаблон.

Присоедините этот сценарий к объекту-контроллеру, и на панели Inspector появится поле для шаблона врага с именем Enemy Prefab. Оно работает аналогично общедоступным переменным, но есть и важное отличие.

ВНИМАНИЕ Для ссылки на объекты в редакторе Unity я рекомендую закрытые переменные с атрибутом `SerializeField`, так как нам нужно отобразить поле новой переменной на панели Inspector, но при этом не хотелось бы, чтобы ее значение могли менять другие сценарии. Как объяснялось в главе 2, открытые переменные отображаются на панели Inspector по умолчанию (другими словами, их сериализацией занимается Unity), поэтому в большинстве руководств и примеров для всех сериализованных значений фигурируют общедоступные переменные. Но эти переменные могут быть модифицированы другими сценариями (ведь они являются общедоступными); в большинстве же случаев редактирование значений должно разрешаться только через панель Inspector.

Перетащите шаблон врага с вкладки Project на пустое поле переменной; при наведении на него указателя мыши вы должны увидеть, как поле подсвечивается, демонстрируя допустимость присоединения объекта (рис. 3.8). После присоединения к шаблону врага сценария `SceneController` воспроизведите сцену, чтобы посмотреть, как работает код. Враг, как и раньше, будет возникать в центре комнаты, но если вы его застрелите, на его месте появится новый враг. Это намного лучше, чем единственный враг, который умирает навсегда!

СОВЕТ Перетаскивание объектов на поля переменных панели Inspector — весьма удобный прием, используемый в самых разных сценариях. В данном случае мы связали шаблон со сценарием, но можно связывать между собой объекты сцены и даже отдельные компоненты (так как код в этом конкретном компоненте должен вызывать открытые методы). В следующих главах мы еще не раз воспользуемся этим приемом.

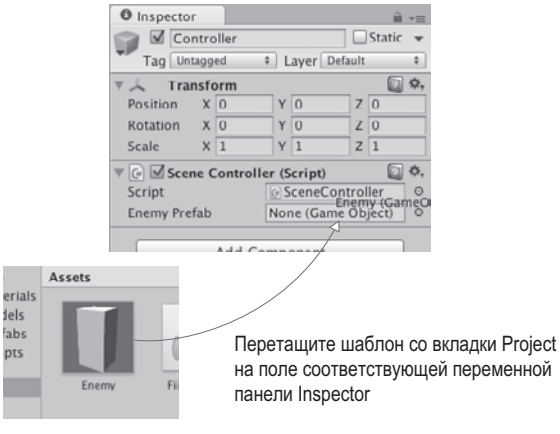


Рис. 3.8. Процедура соединения шаблона со сценарием

Центральной частью сценария является метод `Instantiate()`, поэтому обратите внимание на содержащую его строку. Созданные экземпляры шаблона появляются в сцене. По умолчанию метод `Instantiate()` возвращает новый объект обобщенного типа `Object`, который сам по себе практически бесполезен, поэтому его следует обрабатывать как объект `GameObject`. В языке C# приведение типов осуществляется при помощи ключевого слова `as` (указывается исходный объект, ключевое слово `as` и желаемый новый тип).

Полученный экземпляр сохраняется в закрытой переменной `_enemy` типа `GameObject` (снова напоминаю о разнице между шаблоном экземпляра и самим экземпляром; переменная `enemyPrefab` хранит в себе шаблон, в то время как переменная `_enemy` — экземпляр этого шаблона). Инструкция `if`, проверяющая сохраненный объект, гарантирует, что метод `Instantiate()` будет вызван только при пустой переменной `_enemy` (или на языке кода — при равенстве этой переменной значению `null`). Изначально эта переменная пуста, соответственно, код создания экземпляра запускается в самом начале сеанса. Возвращенный методом `Instantiate()` объект затем сохраняется в переменной `_enemy`, блокируя повторный запуск кода создания экземпляров.

После попадания объект-враг разрушается, переменная `_enemy` становится пустой, что приводит к вызову метода `Instantiate()`. Благодаря этому враг всегда присутствует в сцене.

РАЗРУШЕНИЕ ИГРОВЫХ ОБЪЕКТОВ И УПРАВЛЕНИЕ ПАМЯТЬЮ

Тот факт, что существующие ссылки при разрушении объекта начинают указывать на значение `null`, является до некоторой степени неожиданным. В языках программирования с автоматическим управлением памятью, к которым относится C#, мы, как правило, не можем напрямую удалять объекты; можно обнулить все ведущие на них ссылки, после чего удаление объекта произойдет автоматически. Это верно и в Unity, но фоновый способ обработки объектов `GameObject` выглядит в Unity так, как если бы удаление совершалось напрямую.

Для отображения объектов в Unity ссылки на все эти объекты должны присутствовать в графе сцены. Соответственно, даже после удаления всех ссылок на конкретный игровой объект в коде на него все равно будет ссылаться граф сцены, что сделает автоматическое удаление невозможным.

Поэтому в Unity существует метод `Destroy()`, заставляющий игровой движок удалять объект из графа сцены. Как часть этой фоновой функциональности в Unity также перегружается оператор == и начинает возвращать значение `true` при проверке на наличие значения `null`. Технически объект все еще находится в памяти, но при этом он может больше не существовать, поэтому Unity показывает его равенство значению `null`. Это можно проверить, вызвав для уничтоженного объекта метод `GetInstanceID()`.

Впрочем, разработчики Unity обдумывают замену этого поведения более стандартным вариантом управления памятью. Если подобное произойдет, придется поменять и код порождения врагов, указав вместо проверки (`_enemy==null`) новый параметр, например (`_enemy.isDestroyed`). Следите за новостями в социальной сети Facebook по адресу https://www.facebook.com/unity3d/posts/10152271098591773.

(Если большая часть данного примечания осталась вам непонятной, просто не обращайте на него внимания; это было лирическое отступление для пользователей, заинтересованных в непонятных деталях.)

3.5. Стрельба путем создания экземпляров

Хорошо, добавим врагам еще немного способностей. Пойдем по тому же пути, что и при работе над игроком. Первым делом мы научили врагов двигаться, теперь дадим им возможность стрелять! Как я упоминал, знакомя вас с приемом испускания луча, этот прием является всего лишь одним из подходов к реализации стрельбы. Другой вариант реализации — создание экземпляров из шаблона. Воспользуемся им, чтобы заставить врагов отстреливаться. Результат, который нужно получить, показан на рис. 3.9.

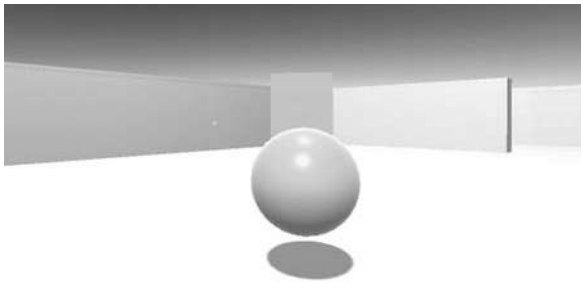


Рис. 3.9. Враг бросает в игрока «огненный шар»

3.5.1. Шаблон снаряда

Если раньше мы стреляли без применения реальных снарядов, то теперь они нам понадобятся. Стрельба приемом испускания лучей, по сути, мгновенна, и попадание регистрируется в момент щелчка кнопкой мыши, враги же будут бросать «огненные шары», летающие по воздуху. Разумеется, хотя они двигаются довольно быстро, у игрока будет возможность уклониться. Фиксировать попадания мы будем не методом испускания лучей, а методом распознавания столкновений (это уже знакомый вам метод, не дающий игроку проходить сквозь стены).

Код будет порождать огненные шары тем же способом, каким порождаются враги, — созданием экземпляров из шаблона. Как вы знаете из предыдущего раздела, первым шагом в подобных случаях является создание объекта, который послужит основой для шаблона. Так как нам требуется огненный шар, выберите в меню GameObject команду 3D Object и в дополнительном меню команду Sphere. Присвойте появившейся сфере имя Fireball. Теперь создайте новый сценарий с таким же именем и присоедините его к объекту. Чуть позже мы напишем для него код, а пока оставьте вариант, предлагаемый по умолчанию, так как первым делом мы завершим работу над снарядом. Чтобы он напоминал огненный шар, ему нужно присвоить ярко-оранжевый цвет. Такие свойства поверхностей, как цвет, контролируются при помощи *материалов*.

ОПРЕДЕЛЕНИЕ *Материалом* (material) называется пакет информации, определяющий свойства поверхности любого трехмерного объекта, к которому этот пакет присоединен. К свойствам поверхности относятся цвет, блеск и даже небольшая шероховатость.

Выберите в меню Assets команду Create и в дополнительном меню команду Material. Присвойте новому материалу имя Flame. Выберите его на вкладке Project, чтобы увидеть его свойства на панели Inspector. Как показано на рис. 3.10, щелкните на цветовой ячейке с именем Albedo (это технический термин, означающий основной цвет поверхности). В отдельном окне откроется палитра цветов; переместите расположенный справа ползунок и точку в основной области таким образом, чтобы выбрать оранжевый цвет.

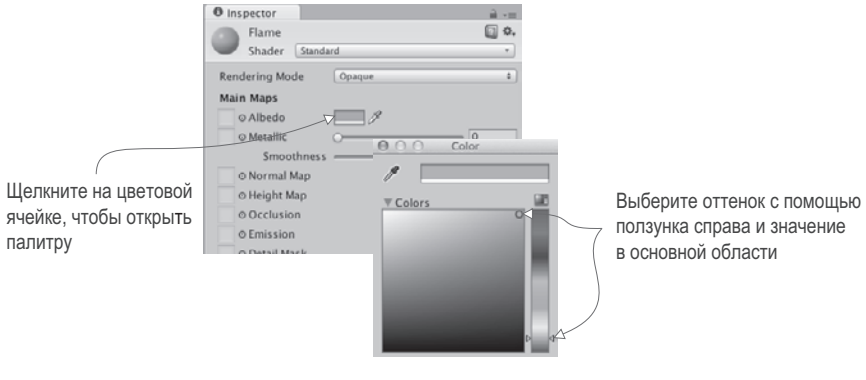


Рис. 3.10. Выбор цвета материала

Еще мы увеличим яркость материала, чтобы придать ему большее сходство с пламенем. Отредактируйте параметр Emission (он тоже находится в списке атрибутов панели Inspector). По умолчанию он равен `0`, присвойте ему значение `0.3`, чтобы сделать материал более ярким.

Теперь можно превратить наш огненный шар в шаблон, перетащив его со вкладки Hierarchy на вкладку Project, как мы делали, создавая шаблон врага. Теперь у нас есть основа для наших снарядов! Осталось написать код стрельбы.

3.5.2. Стрельба и столкновение с целью

Отредактируем шаблон врага, наделив его способностью кидать огненные шары. Чтобы код распознавал игрока, потребуется новый сценарий (аналогично тому, как сценарий `ReactiveTarget` требовался для распознавания мишеней), поэтому создадим сценарий с именем `PlayerCharacter` и присоединим его к игроку. А теперь откроем сценарий `WanderingAI` и введем в него код следующего листинга.

Листинг 3.11. Сценарий WanderingAI с возможностью кидать огненные шары

```
...
[SerializeField] private GameObject fireballPrefab;
private GameObject _fireball;
...
if (Physics.SphereCast(ray, 0.75f, out hit)) {
  GameObject hitObject = hit.transform.gameObject;
  if (hitObject.GetComponent<PlayerCharacter>()) {
    if (_fireball == null) {
      _fireball = Instantiate(fireballPrefab) as GameObject;
      _fireball.transform.position =
      transform.TransformPoint(Vector3.forward * 1.5f);
      _fireball.transform.rotation = transform.rotation;
    }
  }
  else if (hit.distance < obstacleRange) {
    float angle = Random.Range(-110, 110);
    transform.Rotate(0, angle, 0);
  }
}
...
```

Эти два поля добавляются перед любыми методами, как и в сценарии SceneController.

Игрок распознается тем же способом, что и мишень в сценарии RayShooter.

Та же самая логика с пустым игровым объектом, что и в сценарии SceneController.

Поместим огненный шар перед врагом и нацелим в направлении его движения.

Метод Instantiate() работает так же, как и в сценарии SceneController.

Думаю, вы заметили, что все примечания к этому листингу ссылаются на сходные (или аналогичные) фрагменты предыдущих сценариев. Фактически в предыдущих листингах вы уже видели весь код, необходимый для бросания огненных шаров; мы просто смешали эти фрагменты друг с другом в соответствии с новым контекстом.

Как и в листинге `SceneController`, в верхнюю часть сценария нужно добавить два поля `GameObject`: сериализованную переменную, с которой будет связываться шаблон, и закрытую переменную для слежения за экземпляром, созданным кодом этого шаблона. После испускания луча идет проверка попадания в объект `PlayerCharacter`; это работает аналогично тому, как код стрельбы проверял наличие у пораженного объекта компонента `ReactiveTarget`. Код, создающий экземпляр огненного шара

при отсутствии таких шаров в сцене, работает как код создания экземпляров врага. Отличаются только размещение и ориентация объектов; на этот раз мы помещаем экземпляр, полученный из шаблона, перед врагом и нацеливаем в направлении его движения.

Как только новый код окажется на своем месте, на панели Inspector появится новое поле Fireball Prefab, точно так же как в свое время для компонента SceneController появилось поле Enemy Prefab. Щелкните на шаблоне врага на вкладке Project, и на панели Inspector появятся компоненты этого объекта, как если бы вы выделили его в сцене.

Хотя упомянутое ранее неудобство интерфейса часто проявляется при редактировании шаблонов, именно интерфейс дает нам возможность легко редактировать компоненты объекта, чем мы сейчас и воспользуемся. Перетащите шаблон огненного шара со вкладки Project на поле Fireball Prefab панели Inspector, как показано на рис. 3.11.

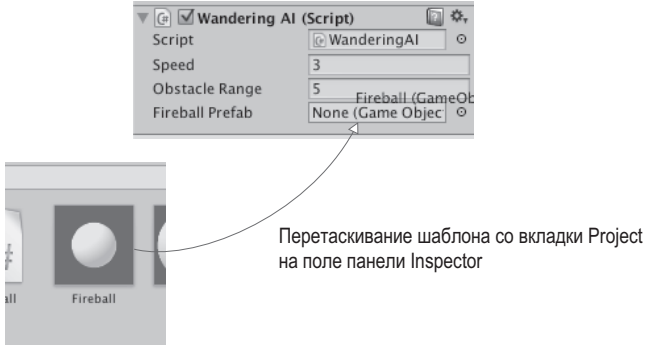


Рис. 3.11. Соединение огненного шара со сценарием

Теперь, как только игрок окажется непосредственно перед врагом, в него полетит огненный шар... Попробуйте выполнить воспроизведение — перед врагом появится огненная сфера, но никуда не полетит, потому что мы пока не написали соответствующий сценарий. Давайте сделаем это сейчас. Код сценария `Fireball` приведен в следующем листинге.

Листинг 3.12. Сценарий Fireball, реагирующий на столкновения

```
using UnityEngine;
using System.Collections;

public class Fireball : MonoBehaviour {
  public float speed = 10.0f;
  public int damage = 1;

  void Update() {
    transform.Translate(0, 0, speed * Time.deltaTime);
  }

  void OnTriggerEnter(Collider other) {
    PlayerCharacter player = other.GetComponent<PlayerCharacter>();
```

Эта функция вызывается, когда с триггером сталкивается другой объект.

```
    if (player != null) {
      Debug.Log("Player hit");
    }
    Destroy(this.gameObject);
  }
}
```

Проверяем, является ли этот другой объект объектом PlayerCharacter.

Существенным новшеством в этом коде является метод `OnTriggerEnter()`. Он автоматически вызывается при столкновении объекта, например, со стеной или с игроком. Пока что наш код работает не совсем корректно; после его запуска огненный шар благодаря строке `Translate()` полетит вперед, но триггер не сработает, а вместо этого будет запрошен новый шар путем разрушения уже существующего. Требуется внести в компоненты огненного шара еще пару дополнений. Прежде всего, нужно превратить коллайдер в триггер. Для этого установите флажок Is Trigger в разделе Sphere Collider.

СОВЕТ После превращения в триггер компонент `Collider` продолжит реагировать на соприкосновение/перекрытие с другими объектами, но уже не будет препятствовать физическому пересечению с этими объектами.

Огненному шару также требуется компонент `Rigidbody`, относящийся к системе моделирования законов физики. Именно он позволит этой системе зарегистрировать триггеры столкновений для рассматриваемого объекта. Щелкните на кнопке Add Component на панели Inspector и по очереди выберите команды Physics и Rigidbody. После добавления указанного компонента сбросьте флажок Use Gravity, как показано на рис. 3.12, чтобы на огненный шар перестала действовать сила тяжести.

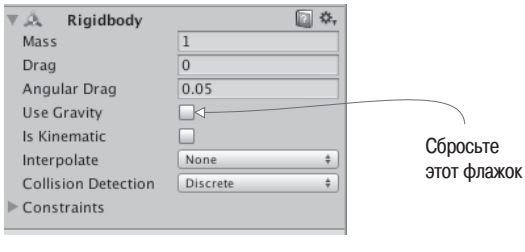


Рис. 3.12. Отключение гравитации для компонента Rigidbody

Воспроизведите сцену, и вы увидите, как огненные шары разрушаются после столкновения с каким-либо объектом. Так как код генерации этих шаров запускается сразу после исчезновения очередного экземпляра, обстрел игрока теперь становится непрерывным. Осталось добавить всего одну деталь: заставить игрока реагировать на попадание.

3.5.3. Повреждение игрока

Ранее мы создали сценарий `PlayerCharacter`, но оставили его пустым. Теперь введите в него из следующего листинга код реакции на попадание.

Листинг 3.13. Игрок может получать повреждения

```
using UnityEngine;
using System.Collections;

public class PlayerCharacter : MonoBehaviour {
  private int _health;

  void Start() {
    _health = 5;  <-- Инициализация переменной health.
  }

  public void Hurt(int damage) {
     _health -= damage;          <-- Уменьшение здоровья игрока.
    Debug.Log("Health: " + _health);
  }
}
```

В листинге определяется поле для здоровья игрока, и по команде этот показатель уменьшается. В следующих главах вы научитесь отображать на экране текстовую информацию, пока же сведения о состоянии игрока можно вывести в виде отладочного сообщения.

Теперь нужно вернуться к сценарию `Fireball`, чтобы вызвать для игрока метод `Hurt()`. Вставьте вместо отладочной строчки в сценарии `Fireball` строку `player.Hurt(damage)`, которая будет сообщать игроку о попадании. Всё. Последний фрагмент кода встал на свое место!

Ну что же, это была достаточно насыщенная глава с большим количеством новой информации. В предыдущей и этой главах вы реализовали большую часть функциональности, необходимой в шутере от первого лица.

Заключение

- Луч — это спроецированная в сцену воображаемая линия.
- Как для стрельбы, так и для обнаружения объектов применяется метод бросания лучей.
- Для моделирования хаотичных перемещений персонажа по сцене используется базовый искусственный интеллект.
- Новые объекты генерируются путем создания экземпляров из существующего шаблона.
- Сопрограммы позволяют растянуть функцию во времени.

4 Работа с графикой

- ✓ Основные сведения о графических ресурсах.
- ✓ Процесс создания геометрической модели сцены.
- ✓ Использование в Unity двумерных изображений.
- ✓ Импорт собственных трехмерных моделей.
- ✓ Моделирование эффектов частиц.

Пока что мы рассматривали в основном принципы функционирования игры, не обращая особого внимания на то, как она выглядит. И это не случайно, ведь книга посвящена программированию игр в Unity. Тем не менее важно знать, каким образом создается и улучшается визуальная картинка. В следующей главе мы вернемся к основной теме — написанию кода для различных частей игры, — а пока поговорим о такой вещи, как графика, чтобы ваши проекты не превращались в набор одинаковых с виду примитивов, плавно перемещающихся по сцене.

Все, что составляет визуальное содержимое игры, называется *графическими ресурсами* (art assets). Но что именно скрывается за этим термином?

4.1. Основные сведения о графических ресурсах

Графическим ресурсом называется отдельная единица визуальной информации (обычно — файл), используемая игрой. Это всеобъемлющее собирательное название для всего визуального содержимого; к графическим ресурсам относятся файлы изображений, трехмерные модели и т. п. На самом деле это всего лишь частный случай ресурса, который, как вы знаете, представляет собой любой используемый игрой файл (например, сценарий). Все они в Unity находятся в основной папке Assets. В табл. 4.1 перечислены и описаны пять основных видов графических ресурсов, применяемых при создании игр.

Таблица 4.1. Типы игровых ресурсов

Тип	Определение типа
Двумерное изображение	Плоские изображения. В реальном мире им соответствуют картины и фотографии
Трехмерная модель	Виртуальные трехмерные объекты (почти синоним для «сеточных объектов»). В реальном мире им соответствуют скульптуры
Материал	Пакет информации, определяющий свойства поверхности любого объекта, к которому присоединен материал. К таким свойствам относят цвет, блеск и даже небольшие шероховатости
Анимация	Пакет информации, определяющий движение связанного с ним объекта. Существуют как заблаговременно созданные детализированные последовательности перемещений, так и код, вычисляющий положения объектов «на лету»
Система частиц	Механизм создания большого количества небольших движущихся объектов и управления ими. Позволяет создавать различные визуальные эффекты, такие как огонь, дым и водяные брызги

Создание графики для новой игры в общем случае начинается с двумерных изображений или трехмерных моделей, так как эти ресурсы формируют базу для всего остального. Как несложно догадаться, двумерные изображения служат основой для двумерной графики, в то время как трехмерные модели — для трехмерной. Точнее говоря, двумерные изображения представляют собой плоские картинки. Даже если у вас нет представления о графике, используемой в играх, с ними вы, скорее всего, сталкивались, например, на сайтах. При этом трехмерные модели могут оказаться совершенно незнакомым новичкам понятием, поэтому я дам им определение.

ОПРЕДЕЛЕНИЕ *Модель* (model) — это трехмерный виртуальный объект. В главе 1 вы столкнулись с термином «сеточный объект»; трехмерная модель — это практически синоним. Оба термина часто используются в одном и том же значении, но термин «сеточный объект» относится исключительно к геометрии трехмерных объектов (соединенные друг с другом линии и фигуры), в то время как термин «модель» имеет более общий смысл и часто подразумевает и другие атрибуты объекта.

Следующие два типа ресурсов в нашем списке — это материалы и анимация. В отличие от двумерных изображений и трехмерных моделей, они не имеют смысла сами по себе, поэтому новичкам бывает непросто понять, что это такое. Изображения и модели легко представимы благодаря аналогам из реального мира. Первым соответствуют картины, вторым — скульптуры. Материалы и анимация не имеют прямых ассоциаций с реальными объектами. Это абстрактные пакеты информации, накладываемые на трехмерные модели. Например, базовый смысл материалов уже был рассмотрен в главе 3.

ОПРЕДЕЛЕНИЕ *Материалом* (material) называется пакет информации, определяющий свойства поверхности любого трехмерного объекта, к которому он присоединяется. К таким свойствам относятся, к примеру, цвет, блеск и даже небольшая шероховатость.

Если продолжить аналогию с искусством, материал можно представить как вещество (глина, бронза, мрамор и т. п.), из которого создается скульптура. Подобным же образом

анимация представляет собой присоединяемый к видимому объекту абстрактный слой информации.

ОПРЕДЕЛЕНИЕ *Анимацией* (animation) называется пакет информации, определяющий движение связанного с ним объекта. Так как эти движения можно задать независимо от объекта, их можно использовать, сочетая и комбинируя с различными объектами.

В качестве конкретного примера рассмотрим перемещающийся по сцене персонаж. Его положение в каждый момент определяется кодом игры (например, сценариями движения, которые вы писали в главе 2). Но подробные движения ног, ступающих по земле, размахивающих рук и поворачивающихся бедер представляют собой воспроизводимые в цикле анимационные последовательности, которые и являются графическим ресурсом.

Чтобы понять, как связаны друг с другом анимация и трехмерные модели, представьте себе кукольный театр: трехмерная модель будет куклой, аниматор — кукольником, заставляющим ее двигаться, а анимация — записью движений куклы. Определенные таким способом движения записываются заранее и обычно происходят в небольшом масштабе, не меняя общего положения объекта. Это отличается от крупномасштабных перемещений, которые мы программировали в предыдущих главах.

Последний графический ресурс, упоминавшийся в табл. 4.1, — это система частиц.

ОПРЕДЕЛЕНИЕ *Системой частиц* (particle system) называется механизм создания большого количества движущихся объектов и управления ими. Обычно эти объекты имеют маленький размер — именно поэтому они называются частицами, хотя это не является обязательным условием.

Системы частиц служат для создания таких визуальных эффектов, как огонь, дым или водяные брызги. Роль частиц (то есть отдельных объектов, контролируемых системой) может играть любой сеточный объект, но для большинства эффектов достаточно квадрата, воспроизводящего изображение (например, искры пламени или клуба дыма).

В основном создание графики для игры выполняется во внешних программах, никак не связанных с Unity. В Unity можно генерировать только материалы и системы частиц. Список внешних программ вы найдете в приложении Б; для создания трехмерных моделей и анимации применяются самые разные графические редакторы. Полученные во внешней программе трехмерные модели затем сохраняются как графические ресурсы, то есть импортируются в Unity. Я пользуюсь программой Blender, о которой рассказывается в приложении В. Ее можно скачать с сайта www.blender.org. Мой выбор обусловлен тем, что это программа с открытым исходным кодом, бесплатно доступная всем желающим.

ПРИМЕЧАНИЕ Проект, который можно скачать для этой главы, включает в себя папку с именем scratch. Хотя эта папка помещена в проект Unity, она не является его частью; я поместил в эту папку дополнительные внешние файлы.

Работая над проектом этой главы, вы познакомитесь с примерами большинства типов графических ресурсов (анимация пока относится к слишком сложным темам, поэтому

она рассматривается в следующих главах). Вам же предстоит построить сцену, в которой фигурируют двумерные изображения, трехмерные модели, материалы и система частиц. В некоторых случаях вы будете пользоваться уже готовыми графическими ресурсами, импортируя их в Unity, но будут и ситуации (особенно с системой частиц), когда графический ресурс придется создавать с нуля.

В этой главе мы только слегка затронем тему создания игровой графики. Все-таки эта книга посвящена в основном программированию в Unity, поэтому подробное рассмотрение вопросов, связанных с графикой, уменьшит количество информации по главной теме. Создание игровой графики — огромная предметная область, для детального описания которой потребуется не одна книга. В большинстве случаев программисты работают в паре со специалистами по графике. Принимая во внимание сказанное, человек, занимающийся программированием игр, должен понимать, как Unity работает с графическими ресурсами, и, возможно, даже уметь создавать их грубые заменители; их еще называют *программистской графикой* (programmer art) и позднее (в готовой игре) заменяют нужными графическими ресурсами.

ПРИМЕЧАНИЕ Проекты из предыдущей главы для выполнения заданий вам не потребуются. Воспользуйтесь сценариями движения из главы 2, чтобы получить возможность перемещаться по сцене; при необходимости можно взять модель игрока и сценарии из скачанного с сайта проекта. К концу этой главы вы создадите движущиеся объекты, напоминающие полученные в предыдущих главах.

4.2. Создание геометрической модели сцены

Разговор о моделировании сцен мы начнем с рассмотрения процесса создания геометрической модели сцены (whiteboxing). Обычно это первый шаг моделирования уровня с помощью компьютера (следующий за разработкой этого уровня на бумаге). Как следует из английского термина, объекты сцены создаются из набора примитивов, то есть белых ящиков (white boxes). В списке различных ресурсов пустые декорации соответствуют базовому виду трехмерной модели и являются основой для отображения двумерных картинок. Вспомните сцену, которую мы создали из примитивов в главе 2. Это и есть геометрическая модель (просто на тот момент вы еще не знали этого термина). В некоторых разделах вы найдете отсылки к вещам, которые мы делали в начале главы 2, но на этот раз я буду касаться их совсем коротко, попутно сократив обсуждение новой терминологии.

ПРИМЕЧАНИЕ В английском языке также используется термин grayboxing (gray boxes — серые ящики). Он также означает геометрическую модель сцены,но я предпочитаю слово whiteboxing, потому что узнал его раньше. Реальный цвет примитивов может различаться, точно так же как светокопии, называемые «синьками» (blueprints), далеко не всегда имели синий цвет.

4.2.1. Назначение геометрической модели

Составление сцены из примитивов практикуется по двум причинам. Во-первых, это позволяет быстро получить «набросок», который со временем будет постепенно совершенствоваться. Эта деятельность близко связана с проектированием игровых уровней.

ОПРЕДЕЛЕНИЕ *Проектирование уровней* (level design) — это дисциплина, касающаяся планирования и создания в игре сцен (или уровней). Проектировщик уровней — это тот, кто занимается проектированием уровней.

По мере увеличения количества разработчиков в группе и сужения специализации каждого отдельного члена группы создание первой версии каждого игрового уровня в виде геометрической модели отдается на откуп проектировщикам уровней. Затем эта заготовка передается специалистам по графике для визуальной доработки. Но даже в маленькой группе, где за проектирование уровней и работу с графикой может отвечать один и тот же человек, такая последовательность действий является оптимальной. В конце концов, нужно с чего-то начинать, а геометрическая модель становится хорошей основой для создания визуальных эффектов.

Второй причиной использования геометрических моделей является возможность быстро привести сцену в подходящее для игры состояние. Она может быть не окончена (более того, уровень, на котором построена только геометрическая модель, очень далек от завершения), но это уже функциональная версия, поддерживающая игровой процесс. Как минимум, игрок в состоянии перемещаться по сцене (вспомните демонстрационный ролик из главы 2). Можно провести тестирование и убедиться, что игровой уровень построен корректно (например, комнаты имеют подходящий для игры размер), и только после этого тратить время и энергию на его проработку. Если окажется, что чего-то не хватает (скажем, вы поняли, что требуется больше места), несложно внести изменения и провести повторное тестирование на стадии геометрической модели.

Более того, возможность поиграть даже на стадии конструирования уровня хорошо поднимает моральный дух. Не сбрасывайте это преимущество со счетов. Создание богатой в визуальном отношении сцены может занять долгое время, и вы почувствуете усталость от того, что никак не можете воспользоваться плодами своего труда. Геометрическая модель сразу дает вам готовый (хотя и примитивный) игровой уровень и возможность играть в постоянно совершенствующуюся игру.

Итак, теперь, когда вы понимаете, почему разработки всех уровней начинаются с геометрических моделей, давайте приступим к созданию игры!

4.2.2. Рисуем план уровня

Созданию игрового уровня на компьютере предшествует его проектирование на бумаге. Мы не будем подробно обсуждать тему проектирования уровней; достаточно сделанного в главе 2 примечания по поводу проектирования игр. Проектирование уровней (представляющее собой разновидность проектирования игр) — это обширная область знаний, достойная отдельной книги. Мы же нарисуем базовый план уровня, чтобы обозначить цель, к которой нужно стремиться.

Рисунок 4.1 демонстрирует вид сверху простого помещения, состоящего из четырех комнат и центрального коридора. Это все, что нам на данный момент нужно: набор отдельных областей и внутренние стены, которые требуется установить на свои места. План реальной игры будет содержать больше деталей, например, таких, как враги и фрагменты обстановки. Попрактиковаться в создании геометрической модели сцены можно как на примере этого плана, так и взяв за основу собственные идеи. Конкретное

расположение комнат в этом упражнении не имеет значения. Важно только наличие у вас нарисованного плана, что позволит перейти к следующему этапу.

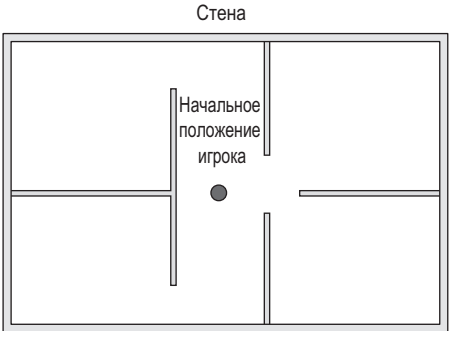


Рис. 4.1. План игрового уровня: четыре комнаты и центральный коридор

4.2.3. Расставляем примитивы в соответствии с планом

Построение геометрической модели сцены в соответствии с имеющимся планом включает в себя позиционирование и масштабирование множества параллелепипедов, которые будут играть роль стен. Как описано в разделе 2.2.1, выберите в меню GameObject команду 3D Object, а затем команду Cube, чтобы получить куб для дальнейших преобразований.

ПРИМЕЧАНИЕ При желании вместо кубов вы можете воспользоваться объектом `QuadsBox`, скачанным вместе с проектом. Он представляет собой куб, созданный из шести частей, что дает дополнительную гибкость при назначении материала. Выбор в данном случае зависит только от вашего желания; лично я не стал пользоваться объектом `QuadsBox`, так как вся геометрия со временем все равно заменяется графикой.

Первым объектом сцены станет пол; на панели Inspector поменяйте имя примитива и его положение по координате *Y* на –0.5, как показано на рис. 4.2. Это делается для компенсации высоты объекта. Затем растяните куб по осям *X* и *Z*.

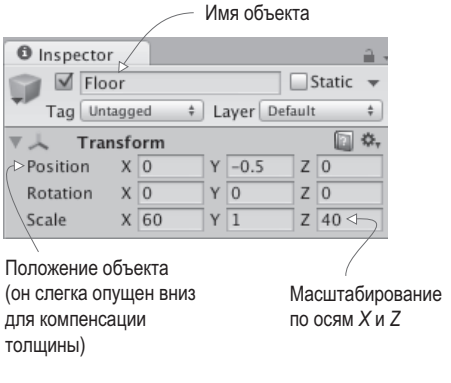


Рис. 4.2. Параметры куба на панели Inspector после перемещения и масштабирования

Повторите эти шаги, чтобы получить стены. Скорее всего, вы захотите привести в порядок вкладку Hierarchy, сделав стены потомками общего базового объекта (напоминаю, что такой объект нужно поместить в точку с координатами 0, 0, 0, а затем на вкладке Hierarchy перетащить на него остальные объекты), но это действие не является обязательным. И не забудьте расположить в сцене несколько простых источников света, чтобы видеть окружающее пространство. В главе 2 вы уже узнали, что для создания источника света нужно выбрать его тип в дополнительном меню, которое появляется после выбора команды Light в меню GameObject. Примерный вид вашего уровня после этих операций показан на рис. 4.3.

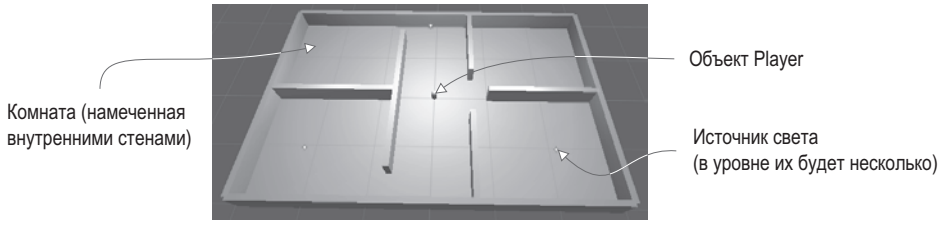


Рис. 4.3. Геометрическая модель игрового уровня, построенного по плану, представленному на рис. 4.1

Заставьте объект, представляющий игрока или камеру, двигаться по сцене (для создания игрока воспользуйтесь контроллером персонажа и сценариями движения — эта тема подробно объяснялась в главе 2). Теперь можно походить по сделанной из примитивов сцене, чтобы протестировать получившийся уровень! Все очень просто, но пока у вас есть только чистая геометрия. Декорируем ее с помощью двумерных изображений.

ЭКСПОРТ ГЕОМЕТРИЧЕСКИХ МОДЕЛЕЙ В ДРУГИЕ ПРОГРАММЫ

Большая часть работы по визуальному оформлению сцены выполняется во внешних приложениях для работы с трехмерной графикой, например в программе Blender. Удобнее всего это делать, загрузив во внешнюю программу свою геометрическую модель. Хотя по умолчанию возможность экспорта скомпонованных примитивов в Unity отсутствует, существуют сценарии, позволяющие добавить в редактор такую функциональность. Большинство из них дают возможность выделить в сцене всю геометрию и щелкнуть на кнопке Export (я упоминал их в главе 1, в разделе, посвященном настройкам редактора). Эти сценарии обычно экспортируют геометрию как OBJ-файл (этот тип файлов обсуждается чуть позже). На сайте Unity3D щелкните на кнопке поиска и введите запрос obj exporter. Или можете посмотреть пример такого сценария на странице http://wiki.unity3d.com/index.php?title=ObjExporter.

4.3. Наложение текстур

Пока что наш уровень представляет собой грубый набросок. Он уже доступен для игры, но очевидно, что над внешним видом сцены требуется еще долго работать. Следующим шагом по совершенствованию уровня будет наложение текстур.

ОПРЕДЕЛЕНИЕ *Текстурой* (texture) называется двумерное изображение, применяемое для улучшения качества трехмерной графики. Это обобщенное определение термина; различные способы использования текстур тут не учитываются. Впрочем, каким бы способом изображение ни использовалось, оно все равно будет называться текстурой.

В трехмерной графике текстуры имеют разное применение, но наиболее простым является отображение их на поверхности трехмерных моделей. Позднее вы узнаете, как это происходит в случае сложных моделей, сейчас же мы, по сути, покроем стены обоями, как показано на рис. 4.4.

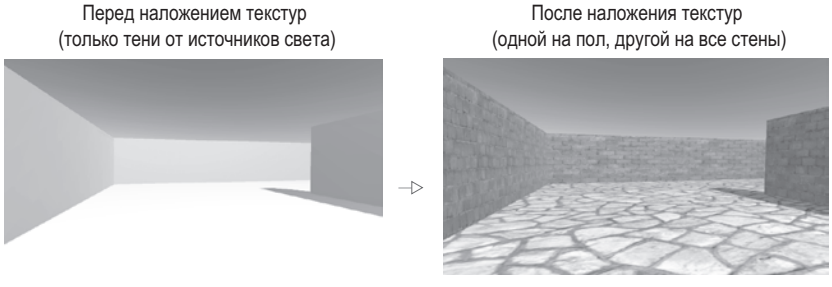


Рис. 4.4. Внешний вид игрового уровня до и после наложения текстур

Как видно из рисунка, текстура превращает откровенно нереалистичную цифровую конструкцию в кирпичную стену. Другие варианты применения текстур включают в себя маски, позволяющие вырезать фигуры, и карты нормалей для имитации рельефа. Рекомендую вам самостоятельно почитать дополнительные материалы о работе с текстурами.

4.3.1. Выбор формата файла

Для сохранения двумерных изображений существует множество форматов. Какой из них лучше выбрать? Форматы, поддерживаемые Unity, перечислены в табл. 4.2.

ОПРЕДЕЛЕНИЕ *Альфа-канал* (alpha channel) служит для хранения информации о прозрачности изображения. Видимые цвета поступают по трем «каналам»: красному, зеленому и синему. Альфа — это дополнительный невидимый канал, управляющий прозрачностью изображения.

Хотя Unity допускает импорт и использование в качестве текстур изображений всех перечисленных в табл. 4.2 форматов, форматы значительно различаются по количеству поддерживаемых функций. Для файлов, импортируемых в качестве текстур, особенно важны два фактора: как именно сжимается изображение и есть ли у него альфа-канал. С альфа-каналом все просто: так как он часто используется в трехмерной графике, лучше, чтобы у изображения он был. Объяснить важность разных аспектов сжатия чуть сложнее. Впрочем, объяснение можно свести к фразе «сжатие с потерями — это плохо». Изображения, используемые без сжатия и допускающие сжатие без потерь, сохраняют свое качество, в то время как при сжатии с потерями качество падает по мере уменьшения размера файла.

Таблица 4.2. Форматы файлов двумерных изображений, поддерживаемые Unity

Формат	Достоинства и недостатки
PNG	Повсеместно используется в интернете. Сжатие без потерь; есть альфа-канал
JPG	Повсеместно используется в интернете. Сжатие с потерями; нет альфа-канала
GIF	Повсеместно используется в интернете. Сжатие с потерями; нет альфа-канала. (Технически потери возникают не из-за сжатия, а в результате преобразования к 8-битному изображению. Но конечный результат все равно один и тот же)
BMP	Формат, по умолчанию используемый в Windows. Применяется без сжатия; нет альфа-канала
TGA	Повсеместно применяется в трехмерной графике; во всех прочих областях малоизвестен. Используется без сжатия, но возможно и сжатие без потерь; есть альфа-канал
TIFF	Повсеместно применяется в цифровой фотографии и издательском деле. Используется без сжатия, но возможно и сжатие без потерь; нет альфа-канала
PICT	Формат по умолчанию на старых компьютерах Mac. Сжатие с потерями; нет альфа-канала
PSD	Собственный формат Photoshop. Используется без сжатия; есть альфа-канал. Основным достоинством является возможность непосредственной работы с файлами в Photoshop

С учетом этих соображений я рекомендовал бы использовать в качестве текстур в Unity файлы формата PNG или TGA. Формат Targas (TGA) был любимым вариантом для создания текстур, пока в интернете не получил распространение формат PNG. В наше время PNG с технологической точки зрения является практически эквивалентом формата TGA, но получил большее распространение благодаря применению в качестве текстур в интернете. К числу рекомендуемых форматов относится также PSD, потому что удобно работать с одним и тем же файлом как в Photoshop, так и в Unity. Хотя я предпочитаю хранить рабочие файлы отдельно от «готовых» экспортированных в Unity вариантов (аналогичных взглядов я придерживаюсь касательно хранения трехмерных моделей, но об этом речь пойдет позже).

В результате все изображения, представленные в примере проекта, имеют формат PNG, и я рекомендую вам работать именно с ним. А теперь пришло время импортировать в Unity несколько изображений и применить их к объектам сцены.

4.3.2. Импорт файла изображения

Начнем с создания/подготовки наших будущих текстур. Все изображения, используемые в качестве текстур, обычно являются бесшовными, что позволяет многократно повторять их на больших поверхностях.

ОПРЕДЕЛЕНИЕ *Бесшовное изображение* (tileable image) представляет собой рисунок, края которого совпадают друг с другом. Именно это позволяет повторять его на поверхности без видимых швов в местах соединения. Концепция назначения текстур в трехмерном моделирования сходна с использованием фоновых рисунков на веб-страницах.

Получить бесшовное изображение можно различными способами: например, обработать фотографию или нарисовать собственный вариант картинки. Учебные пособия и объяснения можно найти в различных книгах и на сайтах, но сейчас мы не будем тратить на это время. Вместо этого мы воспользуемся изображениями с сайтов, предлагающих наборы графики для трехмерного моделирования. Например, показанные на рис. 4.5 текстуры я скачал с сайта www.textures.com. Именно их я собираюсь назначить полу и стенам; вы можете выбрать собственные картинки, подходящие, по вашему мнению, для этой цели.

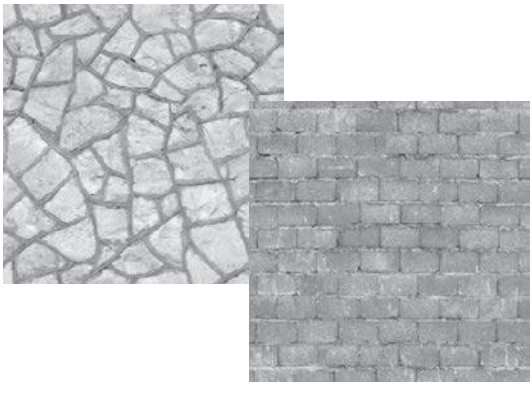


Рис. 4.5. Бесшовные текстуры камня и кирпичей, скачанные с сайта textures.com

Скачайте выбранные вами изображения и подготовьте их к использованию в качестве текстур. С технической точки зрения ничто не мешает задействовать их сразу, но в исходном виде они далеки от идеала. Разумеется, они являются бесшовными (именно поэтому мы их и скачали), но рисунок имеет некорректный размер, а файл — не тот формат, который нам нужен. Размер текстуры должен выражаться в степенях двойки. Графические процессоры показывают максимальную эффективность при обработке изображений, размер которых выражается числом 2^N: 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 (следующее в этом ряду — число 4096, но это слишком большое изображение, чтобы использовать его в качестве текстуры). В графическом редакторе (это может быть Photoshop, GIMP или любой другой вариант из перечисленных в приложении Б) отмасштабируйте скачанные изображения до размера 256 × 256 и сохраните их в формате PNG.

Теперь перетащите эти файлы из папки на вашем компьютере на вкладку Project в Unity, как показано на рис. 4.6. Это действие позволит скопировать их в Unity-проект, после чего их можно будет использовать в трехмерной сцене. Если перетаскивать файлы вам по каким-то причинам неудобно, щелкните правой кнопкой мыши на вкладке Project и выберите в появившемся меню команду Import New Asset, чтобы открыть окно выбора файлов.

СОВЕТ По мере усложнения проектов имеет смысл распределить ресурсы по отдельным папкам; на вкладке Project создайте папки для сценариев и текстур и перетащите в них соответствующие ресурсы.

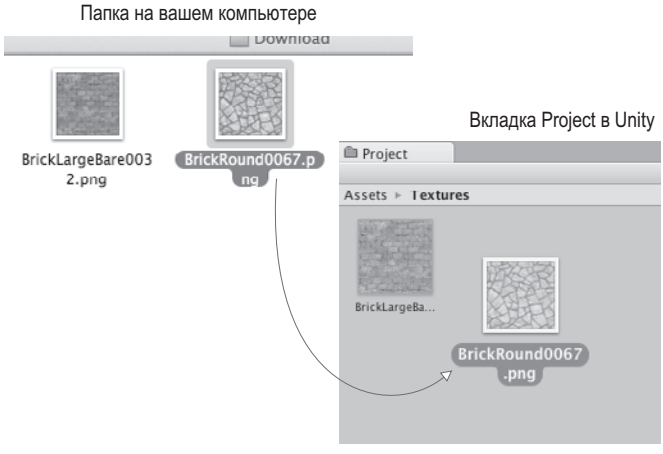


Рис. 4.6. Перетащите изображения на вкладку Project, чтобы импортировать их в Unity

ВНИМАНИЕ В Unity есть несколько ключевых слов, совпадающих с именами папок. Они инициируют обработку содержимого этих папок специальным образом. Это ключевые слова Resources, Plugins, Editor и Gizmos. Зачем нужны эти папки, вы узнаете позже, а пока просто избегайте этих слов, выбирая имена для своих папок.

Теперь изображения импортированы в Unity как текстуры и готовы к использованию. Как же назначить их объектам сцены?

4.3.3. Назначение текстуры

С технической точки зрения наложить текстуру непосредственно на геометрию невозможно. Текстуры должны входить в состав материалов, которые, собственно, и назначаются объектам. Как объяснялось во введении, материалом называется пакет информации, описывающий свойства поверхности; эта информация может включать в себя и отображаемую текстуру. Подобный подход имеет смысл, так как позволяет использовать одну и ту же текстуру для разных материалов. Но так как обычно все текстуры фигурируют в составе разных материалов, для удобства в Unity можно просто поместить текстуру на объект, в результате новый материал создается автоматически. Если вы перетащите текстуру с вкладки Project на объект сцены, как показано на рис. 4.7, Unity создаст новый материал и назначит его объекту. Попытайтесь таким образом получить материал для пола.

Кроме этого удобного метода автоматического создания материалов существует еще и «корректный» способ через подменю, которое появляется после выбора в меню Assets команды Create; новый ресурс появляется на вкладке Project. Остается только выделить полученный новый материал, чтобы его свойства отобразились на панели Inspector, и, как показано на рис. 4.8, перетащить текстуру на ячейку с именем Albedo (это технический термин для базового цвета). Перетащите полученный материал со вкладки Project на объект сцены. Попробуйте проделать все описанное с текстурой для стены: создайте новый материал, перетащите в него текстуру и назначьте его стене. Вы увидите, что на поверхности пола и стен появились изображения камня и кирпичей, но они выглядят

растянутыми и размытыми. Как получилось, что единственное изображение оказалось растянутым на весь пол? Мы же хотели, чтобы оно повторялось на поверхности несколько раз. Такой эффект дает свойство Tiling: выделите материал на вкладке Project и измените числа в полях Tiling на панели Inspector (существуют отдельные значения для координат *X* и *Y*, отвечающие за количество повторений в каждом направлении). Проверьте, что вы задаете повторение основной, а не вторичной карты (данный материал поддерживает вторичную карту текстуры для усовершенствованных эффектов). По умолчанию число повторений равно 1 (то есть единственная текстура растягивается на всю поверхность); присвойте этому параметру, к примеру, значение 8 и посмотрите, как изменится вид пола. Подберите и для второго материала кратность, обеспечивающую оптимальный вид. Итак, пол и стены нашей комнаты обзавелись текстурами! Но вы можете назначить текстуру и небу; давайте посмотрим, как это делается.

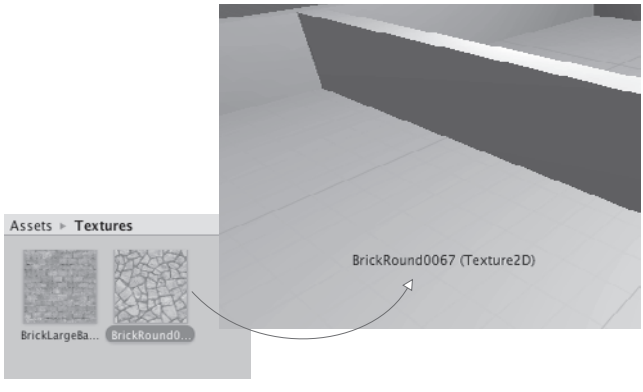


Рис. 4.7. Одним из способов наложения текстур является их перетаскивание со вкладки Project на объекты сцены

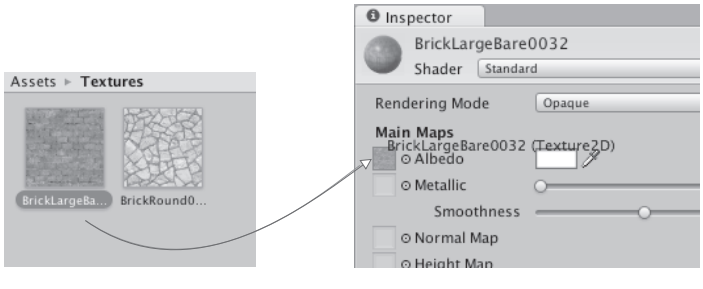


Рис. 4.8. Выделите материал для просмотра его свойств на панели Inspector и перетащите структуру на ячейку одного из свойств

4.4. Создание неба с помощью текстур

Текстуры камня и кирпича придали стенам и полу намного более естественный вид. Но небо пока выглядит пустым и ненатуральным, мы же хотим придать сцене реалистичность. Чаще всего эта задача решается при помощи специальных текстур с изображениями неба.

4.4.1. Что такое скайбокс?

По умолчанию камера показывает фоновый темно-синий цвет. Он заполняет все пустое пространство сцены (например, пространство над стенами). Но в качестве фона можно визуализировать и изображение неба. В этом нам поможет скайбокс.

ОПРЕДЕЛЕНИЕ *Скайбоксом* (skybox) называется окружающий камеру куб, на грани которого находится изображение неба. В каком бы направлении ни смотрела камера, она будет отображать небо.

Корректная реализация скайбокса — дело непростое; принцип его работы иллюстрирует рис. 4.9. Существует ряд приемов, позволяющих отобразить грани куба как удаленный фон. К счастью, все детали реализации в Unity уже учтены.

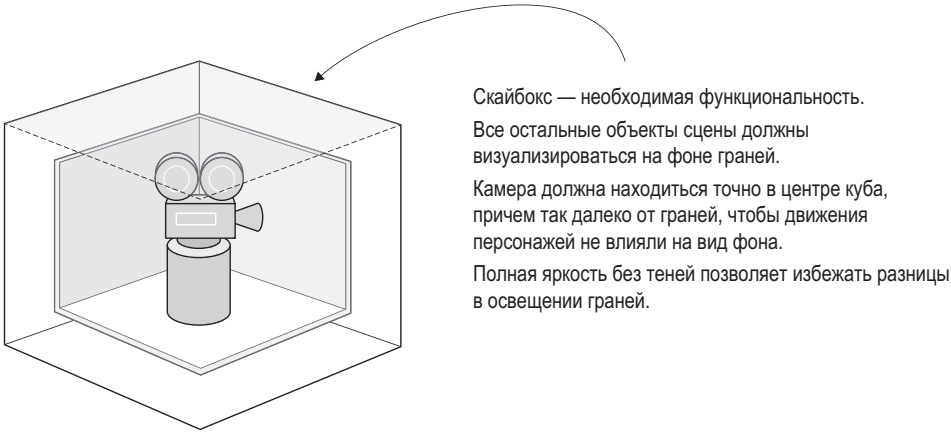


Рис. 4.9. Схема скайбокса

Новые сцены создаются с уже готовым скайбоксом. Именно поэтому вместо равномерного темно-синего фона цвет неба постепенно меняется от светлого к темно-синему. Если открыть окно диалога с параметрами освещенности (выбрав в меню Window команду Lighting), первым вы увидите параметр Skybox со значением Default. Этот параметр находится в свитке Environment Lighting; окно диалога разделено на свитки, связанные с усовершенствованной системой освещения в Unity. Впрочем, пока нас интересует только самый первый параметр.

Текстуры для скайбокса, как и текстуры кирпича и камня, можно найти на различных сайтах. Воспользуйтесь поисковым запросом текстуры для скайбокса (skybox textures). Например, я нашел несколько прекрасных вариантов на сайте www.93i.de, в том числе набор TropicalSunnyDay. После добавления к скайбоксу текстуры неба сцена начнет выглядеть так, как показано на рис. 4.10.

Как и прочие текстуры, изображения для скайбокса сначала назначаются материалу и только потом используются в сцене. Давайте попробуем создать для скайбокса новый материал.

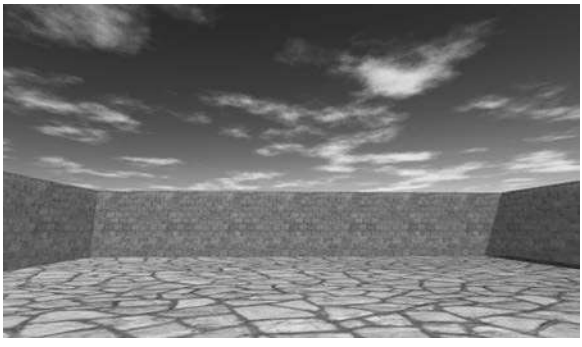


Рис. 4.10. Сцена с фоновым изображением неба

4.4.2. Создание нового материала для скайбокса

Сначала создайте новый материал (как обычно, щелкнув правой кнопкой мыши и выбрав команду Create или выбрав эту же команду в меню Assets).Его параметры отобразятся на панели Inspector. Первым делом нам нужно поменять шейдер материала. В верхней части списка настроек находится меню Shader, показанное на рис. 4.11. В разделе 4.3 мы не обращали на него внимания, так как шейдер, предлагаемый по умолчанию, подходит большинству стандартных текстур, но скайбокс требует другого варианта.

ОПРЕДЕЛЕНИЕ *Шейдером* (shader) называется короткая программа с инструкциями, описывающими способ рисования поверхности. В ней указываются, в частности, используемые текстуры. Компьютер задействует эти инструкции для вычисления пикселов в процессе визуализации изображения. В наиболее распространенном шейдере цвет материала затемняется в соответствии с освещенностью. Шейдеры применяются для всех видов визуальных эффектов.

Каждый материал имеет шейдер, который определяет его вид (можно представить материал как экземпляр шейдера). Новому материалу по умолчанию назначается шейдер Standard. Он отображает цвет материала (включая назначенную текстуру), одновременно применяя к поверхности основные настройки теней и освещенности. Для скайбоксов используется другой шейдер. Щелкните на этом меню, чтобы открыть выпадающий список с перечнем доступных шейдеров. Выделите строку Skybox и выберите в появившемся дополнительном меню вариант 6 Sided, как показано на рис. 4.11.

Теперь в настройках материала появилось шесть ячеек для текстур (вместо одной маленькой ячейки Albedo, которую мы видели у стандартного шейдера). Эти шесть текстур соответствуют шести сторонам куба. Они должны совпадать друг с другом в местах стыка, чтобы картинка получилась бесшовной. Например, рис. 4.12 демонстрирует изображения для солнечного скайбокса.

Импортируйте в Unity изображения для скайбокса тем же способом, которым импортировалась текстура кирпича: перетащите файлы на вкладку Project или щелкните правой кнопкой мыши на вкладке Project и выберите команду Import New Asset. Впрочем, в данном случае есть одно небольшое отличие; щелчком выделите импортированную текстуру,

чтобы увидеть ее свойства на панели Inspector, и поменяйте значение параметра Wrap Mode с Repeat на Clamp (рис. 4.13); не забудьте после этого щелкнуть на кнопке Apply. Обычно текстуры укладываются на поверхность как плитки, а чтобы результат такой укладки выглядел бесшовным, противоположные края изображений накладываются друг на друга. Но в случае неба подобная операция может привести к появлению небольших линий, поэтому значение Clamp (аналогичное знакомой вам по главе 2 функции `Clamp()`) очертит границы текстуры и уберет результат их наложения.

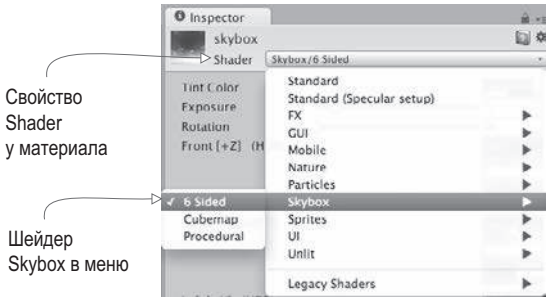


Рис. 4.11. Выпадающее меню доступных раскрасок

Изображения для скайбокса с сайта 93i.de: верхнее, нижнее, фронтальное, заднее, левое, правое

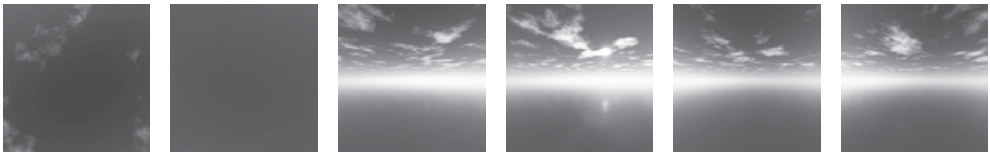


Рис. 4.12. Шесть сторон скайбокса

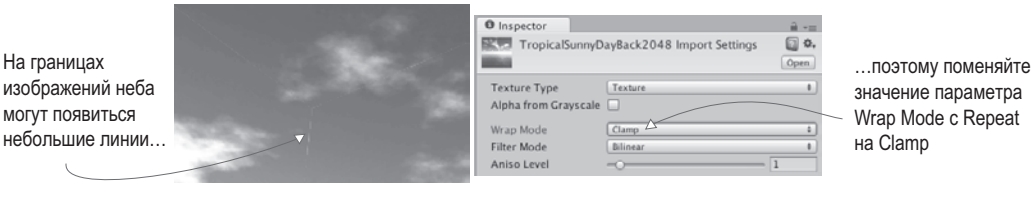


Рис. 4.13. Избавление от линий путем редактирования параметра Wrap Mode

Теперь можно перетащить изображения на ячейки для текстур. Имена изображений должны совпадать с именами ячеек (например, left или front). Как только все текстуры окажутся на своих местах, можно использовать материал для скайбокса. Снова откройте окно с параметрами освещенности и перетащите новый материал на ячейку Skybox или щелкните на маленьком кружке с точкой в центре, расположенном справа от ячейки, чтобы открыть окно выбора материала.

СОВЕТ По умолчанию Unity отображает скайбокс (или, по крайней мере, его основной цвет) на вкладке Scene редактора. Если это мешает редактированию объектов, видимость скайбокса можно отключить. В верхней части вкладки Scene располагаются кнопки, управляющие видимостью различных элементов; щелчок на крайней правой кнопке, которая называется Effects, открывает меню, через которое можно отключить видимость скайбокса.

Итак, вы узнали, как смоделировать настоящее небо! Скайбокс предлагает вам элегантный способ создать иллюзию бескрайнего пространства вокруг игрока. Следующим шагом по совершенствованию внешнего вида сцены станет получение более сложных трехмерных моделей.

4.5. Собственные трехмерные модели

В предыдущем разделе мы накладывали текстуры на большие плоские стены и пол. А что делать с более детализированными объектами? Предположим, мы хотим обставить комнаты мебелью. Для решения этой задачи нам потребуется внешнее приложение для работы с трехмерной графикой. Вспомните определение, которое было дано в начале этой главы: трехмерные модели — это помещенные в игру сеточные объекты (то есть трехмерные фигуры). В этом разделе мы импортируем в игру сетку, имеющую форму скамейки.

Для моделирования трехмерных объектов широко применяются такие приложения, как Maya от Autodesk и 3ds Max. Но это дорогие коммерческие инструменты, поэтому мы воспользуемся приложением с открытым исходным кодом, которое называется Blender. Скачанный с сайта проект включает в себя файл с расширением .blend, которым вы можете воспользоваться; рис. 4.14 демонстрирует модель скамейки в программе Blender. На случай, если вы захотите научиться моделировать такие объекты собственными руками, в приложение В включено упражнение по созданию скамейки.

Модель состоит как из определяющей ее форму
трехмерной сетки, так и из наложенной на эту сетку текстуры

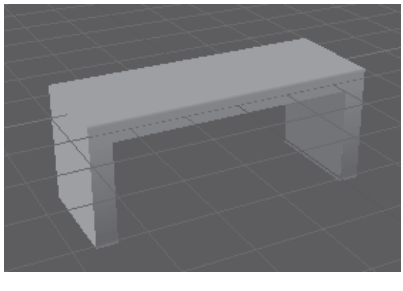


Рис. 4.14. Модель скамейки в программе Blender

Кроме моделей, созданных лично вами или сотрудничающим с вами художником, можно скачивать подходящие варианты со специальных сайтов. К примеру, существует такой замечательный ресурс, как Asset Store: https://www.assetstore.unity3d.com.

4.5.1. Выбор формата файла

Созданную в приложении Blender модель требуется экспортировать. Как и в случае с двумерными изображениями, существует множество разных форматов экспорта, каждый из которых обладает своими достоинствами и недостатками. Поддерживаемые в Unity форматы трехмерных файлов перечислены в табл. 4.3.

Таблица 4.3. Форматы файлов трехмерных моделей, поддерживаемые в Unity

Формат	Достоинства и недостатки
FBX	Сетки и анимация; рекомендуемый формат
Collada (DAE)	Сетки и анимация; еще один хороший вариант, если формат FBX недоступен
OBJ	Только сетки; это текстовый формат, который иногда используется для трансляции в интернет
3DS	Только сетки; достаточно старый и примитивный формат
DXF	Только сетки; достаточно старый и примитивный формат
Maya	Работает через FBX; требует установки этого приложения
3ds Max	Работает через FBX; требует установки этого приложения
Blender	Работает через FBX; требует установки этого приложения

Выбор варианта сводится к поддержке анимации. Так как единственными удовлетворяющими этому условию вариантами являются Collada и FBX, выбирать приходится между ними. Когда есть такая возможность (не все инструменты для работы с трехмерной графикой экспортируют данные в этом формате), лучше всего пользоваться форматом FBX, в противном случае подойдет и формат Collada. К счастью, приложение Blender допускает экспорт файлов в формате FBX.

Обратите внимание, что в нижней части табл. 4.3 перечислено несколько приложений для работы с 3D-графикой. Инструмент Unity позволяет прямо переносить их файлы в ваши проекты, что сначала кажется удобным, но эта функциональность имеет несколько подводных камней. Во-первых, Unity не загружает непосредственно сами файлы. Модель загружается в фоновом режиме, затем загружается экспортированный файл. Но так как модель в любом случае экспортируется в формате FBX или Collada, лучше делать это в явном виде. Во-вторых, для подобной операции у вас должно быть установлено соответствующее приложение. Если вы планируете организовать доступ к файлам с разных компьютеров (например, для группы разработчиков), это обстоятельство становится большой проблемой. Я не рекомендую загружать файлы из приложения Blender (или Maya, или еще откуда-то) напрямую в Unity.

4.5.2. Экспорт и импорт модели

Итак, пришло время экспортировать модель из Blender и импортировать ее в Unity. Первым делом откройте файл со скамейкой в приложении Blender и выберите в меню File команду Export4FBX. Сохраненный файл импортируйте в Unity тем же способом, каким осуществлялся импорт изображений. Перетащите FBX-файл на вкладку Project или щелкните на этой вкладке правой кнопкой мыши и выберите команду Import New Asset. Трехмерная модель, готовая к вставке в сцену, будет скопирована в Unity-проект.

ПРИМЕЧАНИЕ В доступный для скачивания пример проекта включен файл с расширением.blend, чтобы вы могли попрактиковаться в экспорте FBX-файлов из приложения Blender; даже если вы не хотите ничего моделировать самостоятельно, зачастую требуется конвертировать скачанные файлы в доступный для Unity формат. Если вы предпочитаете пропустить все шаги, связанные с приложением Blender, задействуйте имеющийся FBX-файл.

При импорте моделей желательно сразу же поменять несколько параметров. Unity масштабирует импортируемые модели до очень маленьких размеров (на рис. 4.15 показано, что вы увидите на панели Inspector, выделив такую модель), поэтому введите в поле Scale Factor значение 100, чтобы частично скомпенсировать параметр File Scale, равный 0,01. Можно также установить флажок Generate Colliders (Генерировать коллайдеры), но это необязательно; просто без коллайдера персонажи смогли бы проходить сквозь скамейку. Затем перейдите на вкладку Animation в параметрах импорта и сбросьте флажок Import Animation (Импорт анимации) — ведь эту модель мы анимировать не будем.

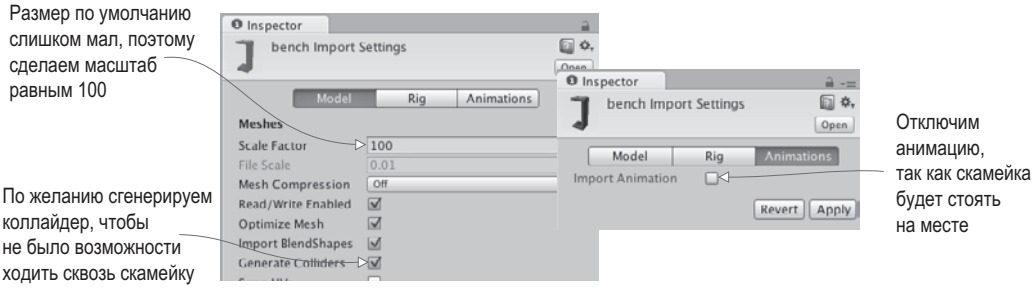


Рис. 4.15. Редактирование параметров импорта трехмерной модели

Это обеспечит нам корректный импорт сетки. Теперь что касается текстуры. При импорте FBX-файла инструмент Unity заодно создал материал для скамейки. По умолчанию он пустой (как любой новый материал), поэтому назначьте ему текстуру (показанную на рис. 4.16) тем же способом, каким вы назначали текстуру кирпича стене: перетащите изображение на вкладку Project, чтобы импортировать его в Unity, а затем — на ячейку текстуры в настройках материала скамейки. Изначально изображение будет выглядеть странновато, его разные части окажутся на разных частях скамейки, поэтому текстурные координаты были отредактированы для приведения изображения в соответствие с сеткой.

Рис. 4.16. Двумерное изображение, служащее текстурой для скамейки

ОПРЕДЕЛЕНИЕ *Текстурными координатами* (texture coordinates) называется дополнительный набор значений для каждой вершины, проецирующий полигоны на области текстуры. Представьте, что трехмерная модель — это ящик, а текстура — оберточная бумага. Текстурные координаты будут указывать, на какую из сторон ящика должен накладываться каждый фрагмент бумаги.

ПРИМЕЧАНИЕ Даже если вы не собираетесь вникать в тонкости моделирования скамейки, имеет смысл ознакомиться с информацией о текстурных координатах (см. приложение В). Понимание этой концепции (как и связанных с ней понятий UV-координат и проецирования) пригодится при программировании игр.

Новые материалы часто имеют слишком сильный блеск, поэтому имеет смысл уменьшить параметр Smoothness до нуля (чем более гладкой является поверхность, тем сильнее она блестит). После этого скамейку можно поместить в сцену. Перетащите модель с вкладки Project в одну из комнат. Как только вы поставите объект на место, вы увидите нечто, напоминающее рис. 4.17. Поздравляю, вы создали для игрового уровня текстурированную модель!

ПРИМЕЧАНИЕ В этой главе мы не будем этим заниматься, но, как правило, геометрическая модель сцены позднее заменяется моделями, созданными во внешних программах. Новая модель может выглядеть почти идентично старой, но предоставлять большую гибкость при назначении текстуре координат U и V.

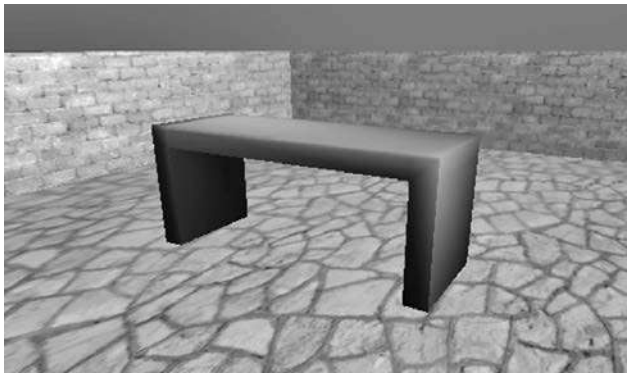


Рис. 4.17. Импортированная в уровень скамейка

СИСТЕМА АНИМАЦИИ ПЕРСОНАЖЕЙ MECANIM

Созданная нами модель статична. Но вы можете анимировать ее в приложении Blender и воспроизвести эту анимацию в Unity. Процесс создания анимации длинный и сложный, а эта книга посвящена совсем другой теме, поэтому здесь мы его обсуждать не будем. Как уже упоминалось, существует множество ресурсов для детального знакомства с трехмерной анимацией. Просто имейте в виду — это крайне обширная тема. Именно поэтому анимацией при разработке игр занимается отдельный специалист. В Unity встроена сложная система управления анимацией моделей. Она называется Mecanim. Это новая усовершенствованная система, недавно добавленная в Unity взамен старой. Впрочем, старая версия пока доступна, хотя и может быть удалена при

следующих обновлениях Unity. В этой главе мы ничего анимировать не будем, а вот в главе 7 вам предстоит заняться воспроизведением анимации для модели персонажа.

4.6. Системы частиц

Кроме двумерных изображений и трехмерных моделей у нас остался еще один тип визуального содержимого — системы частиц. Данное в начале этой главы определение поясняет, что системами частиц называются механизмы создания большого количества движущихся объектов и управления ими. Системы частиц применяются при создании таких эффектов, как огонь, дым или водяные брызги. Например, огонь на рис. 4.18 получен именно при помощи системы частиц.

Если большинство других ресурсов создается во внешних приложениях и импортируется в проект, системы частиц генерируются непосредственно в Unity благодаря гибким и мощным инструментам для создания различных эффектов.

ПРИМЕЧАНИЕ Почти как в случае с системой анимации Mecanim, в Unity соседствовали старая и более новая системы частиц. Последняя называлась Shuriken. В настоящее время старая система полностью вышла из употребления, так что отдельное имя новой системе уже не требуется.

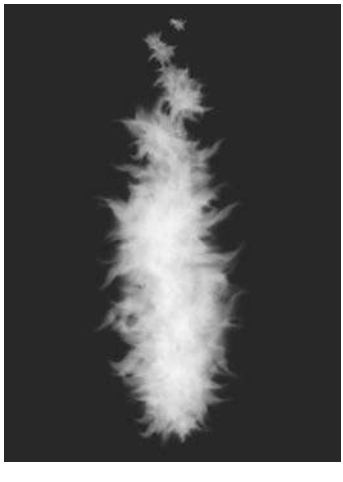


Рис. 4.18. Эффект огня, полученный при помощи системы частиц

Для начала создайте систему частиц и выполните воспроизведение, чтобы посмотреть, как эффект выглядит по умолчанию. В меню GameObject выберите команду Particle System, и вы увидите, как из нового объекта полетят вверх белые пушинки. Для прекращения этого процесса достаточно снять с объекта выделение. При выделении системы частиц в нижнем правом углу экрана появляется панель воспроизведения эффекта, на которой, в частности, можно посмотреть, сколько времени работает система (рис. 4.19).

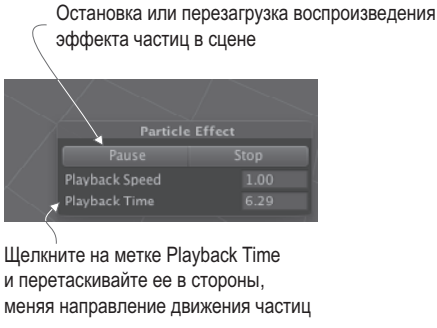


Рис. 4.19. Панель воспроизведения системы частиц

В принципе, эффект уже хорошо выглядит, но давайте рассмотрим список параметров, которыми вы можете пользоваться для его дополнительной настройки.

4.6.1. Редактирование параметров эффекта

Рисунок 4.20 демонстрирует полный список вариантов настройки системы частиц. Мы не будем рассматривать каждый параметр этого списка, а остановимся только на том, что требуется для создания эффекта пламени. Как только вы поймете принцип работы, остальное окажется достаточно очевидным. Каждая из надписей скрывает целый информационный свиток. Изначально раскрыт только первый из них; все остальные свернуты. Чтобы развернуть свиток, щелкните на его названии.

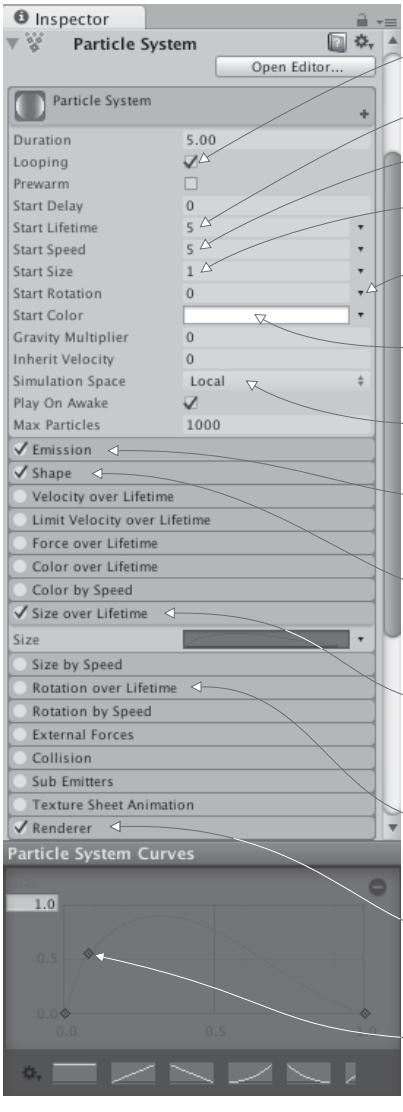
СОВЕТ Изрядное количество параметров редактируется с помощью кривой, отображаемой в нижней части панели Inspector. Она показывает изменение параметра во времени: левая сторона графика соответствует первому появлению частицы, правая — моменту ее исчезновения. Нижняя часть соответствует значению 0, а верхняя — максимально возможному значению. Меняйте форму кривой перетаскиванием точек, вставляя новые точки двойным щелчком или щелчком правой кнопки мыши.

Отредактируйте параметры системы частиц в соответствии с рис. 4.20, чтобы придать ей сходство с факелом.

4.6.2. Новая текстура для пламени

Теперь наша система частиц больше напоминает факел, но нужно придать частицам вид пламени, а не шариков. Для этого нам потребуется импортировать в Unity еще одно изображение. На рис. 4.21 показана нарисованная мной картинка: я поставил оранжевую точку и воспользовался инструментом Smudge для имитации языков пламени (затем я проделал то же самое с желтым цветом). Вне зависимости от того, возьмете вы изображение из примера проекта, нарисуете собственный вариант или скачаете подходящую картинку из интернета, первым делом его нужно импортировать в Unity. Как я уже объяснял, перетащите картинку на панель Project или выберите в меню Assets команду Import New Asset.

Системе частиц, как и трехмерной модели, невозможно назначить текстуру напрямую; поэтому добавьте эту текстуру к материалу и назначьте его системе частиц. Создайте новый материал и перетащите текстуру со вкладки Project на ячейку Albedo на панели



Looping: система частиц работает без остановки; оставьте установленный по умолчанию флажок

Lifetime: время жизни каждой частицы; уменьшите до 3

Speed: скорость перемещения частицы; уменьшите до 1

Size: размер частиц; оставьте значение по умолчанию

Rotation: ориентация частицы; щелкните на стрелке и выберите в меню вариант Between Constants, укажите значения 0 и 180

Color: цвет частиц. Нам требуется темно-оранжевый, например, со значениями RGB 182, 101, 58

Локальное пространство эффекта прекрасно подходит для статичных систем частиц, но для движущейся системы лучше выбрать значение World

Emission: скорость генерации новых частиц; оставьте значение по умолчанию

Shape: форма области, испускающей частицы. По умолчанию это широкий конус, нам же для получения компактного факела нужен куб (выберите вариант Box, все значения 0.2)

Size over Lifetime: в процессе движения частица увеличивается и уменьшается. По умолчанию эта настройка отключена. Включите ее и сформируйте кривую, которая быстро увеличивается от 0, а затем медленно уменьшается обратно до 0 (как и показано на рисунке)

Rotation over Lifetime: в процессе движения частица вращается. По умолчанию эта настройка отключена. Включите ее, выберите вариант Random Between и присвойте значения –80 и 80, чтобы частицы вращались в разных направлениях

Renderer: задает вид каждой частицы. Вы можете выбрать даже значение Mesh, но оставьте вариант Billboard и перетащите на ячейку новый материал (его мы сделаем позже)

Точки к кривой добавляются двойным щелчком или щелчком правой кнопкой мыши и выбором команды Add Key

Рис. 4.20. Панель Inspector отображает варианты настройки системы частиц (в данном случае — огня)

Рис. 4.21. Изображение, используемое для частиц пламени

Inspector. Это свяжет текстуру с материалом, после чего его можно будет добавить к системе частиц. Эта процедура показана на рис. 4.22. Выделите систему частиц, раскройте свиток Renderer в нижней части списка настроек и перетащите материал на ячейку Material.

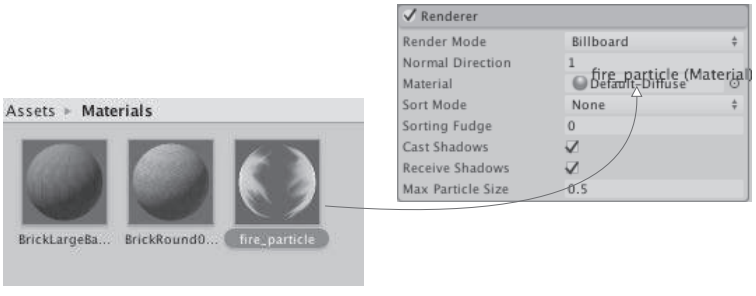


Рис. 4.22. Назначение материала системе частиц

Как и в случае материала для скайбокса, вам нужно выбрать другой вариант шейдера. Щелчком откройте меню Shader в верхней части настроек материала, чтобы увидеть список доступных раскрасок. Вместо используемого по умолчанию стандартного варианта материал для частиц нуждается в одном из вариантов шейдера, перечисленных в дополнительном меню Particles. Как показано на рис. 4.23, на этот раз мы выбираем вариант Additive (Soft). Это заставит частицы опалесцировать и освещать сцену, как настоящий огонь.

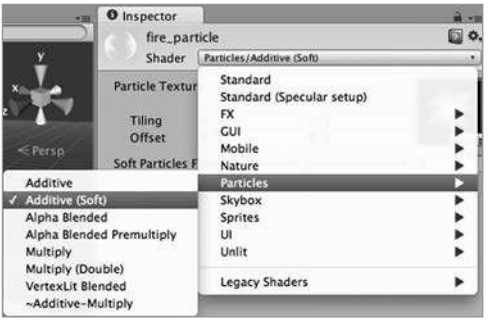


Рис. 4.23. Выбор шейдера материала для частиц пламени

ОПРЕДЕЛЕНИЕ Вариант шейдера Additive добавляет цвет частицы к цвету фона за ней, а не заменяет пикселы. Благодаря этому пикселы становятся ярче, а черный цвет делается невидимым. Противоположный эффект оказывает вариант Multiply, делающий все вокруг темнее; эти шейдеры производят тот же самый эффект, что и эффекты слоя Additive и Multiply в приложении Photoshop.

После назначения системе частиц материала, имитирующего огонь, мы получили нужный эффект (см. рис. 4.18). Факел выглядит достаточно убедительно, причем он в состоянии работать не только в статичном положении. Чтобы убедиться в этом, давайте присоединим его к движущемуся объекту.

4.6.3. Присоединение эффектов частиц к трехмерным объектам

Создайте сферу (напоминаю, что в меню GameObject нужно выбрать команду 3D Object4Sphere). Создайте новый сценарий с именем BackAndForth, показанный в следующем листинге, и присоедините его к сфере.

Листинг 4.1. Движение объекта взад и вперед по прямой

```
using UnityEngine;
using System.Collections;

public class BackAndForth : MonoBehaviour {
  public float speed = 3.0f;
  public float maxZ = 16.0f;  <-- Объект движется между этими точками.
  public float minZ = -16.0f;

  private int _direction = 1;  <-- В каком направлении объект движется в данный момент?

  void Update() {
    transform.Translate(0, 0, _direction * speed * Time.deltaTime);

    bool bounced = false;
    if (transform.position.z > maxZ || transform.position.z < minZ) {
      _direction = -_direction;  <-- Меняем направление на противоположное.
      bounced = true;
    }
    if (bounced) {
      transform.Translate(0, 0, _direction * speed * Time.deltaTime);
    }
  }
}
```

Делаем дополнительное движение в этом кадре, если объект поменял направление.

Запустите этот сценарий, и сфера начнет двигаться взад и вперед по центральному коридору уровня. Теперь можно сделать систему частиц дочерней по отношению к сфере, и огонь начнет перемещаться вместе с ней. Точно так же, как вы поступали со стенами, на вкладке Hierarchy перетащите объект Particle system на объект Sphere.

ВНИМАНИЕ Обычно после того, как объект становится потомком другого объекта, его положение требуется обнулить. К примеру, нам нужно поместить систему частиц в точку 0, 0, 0 (относительно ее предка). В Unity сохраняется положение, которое объект имел до формирования иерархической связи.

Теперь система частиц двигается вместе со сферой, но пламя при этом не отклоняется, что выглядит неестественно. Это связано с тем, что по умолчанию частицы перемещаются корректно только в локальном пространстве собственной системы. Для завершения модели горящей сферы найдите в настройках системы частиц параметр Simulation Space (он находится в верхнем свитке, показанном на рис. 4.20) и измените его значение с Local на World.

ПРИМЕЧАНИЕ Наш объект перемещается по прямой, но обычно в играх используются более замысловатые траектории. В Unity поддерживаются сложные варианты навигации и пути; прочитать об этом можно на странице https://docs.unity3d.com/ru/current/Manual/Navigation.html.

Уверен, что вам не терпится реализовать собственные идеи и добавить в игру новые объекты. Займитесь этим сейчас — можете создать больше графических ресурсов или проверить свои способности, добавив в сцену разработанный в главе 3 механизм стрельбы. В следующей главе мы перейдем к другому игровому жанру и начнем создание новой игры. Впрочем, несмотря на подобные переходы, вся информация из первых четырех глав вполне применима и может оказаться вам полезной.

Заключение

- Термин «графические ресурсы» означает совокупность всей задействованной в проекте графики.
- Создание графической модели сцены является для проектировщика уровней первым шагом по разметке игрового пространства.
- Текстуры и двумерные изображения отображаются на поверхности трехмерных моделей.
- Трехмерные модели создаются вне Unity и импортируются в виде FBX-файлов.
- Системы частиц позволяют создавать многочисленные визуальные эффекты (огонь, дым, воду и т. п.).

Часть II
ОСВАИВАЕМСЯ

Вы создали первый прототип игры и готовы расширить арсенал рабочих инструментов на примерах других игровых жанров. К настоящему моменту приемы работы в Unity должны быть полностью освоены: вы уверенно создаете сценарии с указанными функциями, перетаскиваете объекты на ячейки панели Inspector и т. п. Я больше не буду подробно останавливаться на деталях интерфейса, так как вы уже не нуждаетесь в повторении основ.

Давайте создадим несколько дополнительных проектов, чтобы проиллюстрировать новые приемы разработки игр в Unity.

5 Двумерная игра Memory

- ✓ Отображение двумерной графики в Unity.
- ✓ Создание интерактивных объектов.
- ✓ Программная загрузка изображений.
- ✓ Поддержка и отображение состояния с помощью текстового пользовательского интерфейса.
- ✓ Загрузка уровней и перезапуск игры.

До сих пор вы имели дело только с трехмерной графикой. Но в Unity можно использовать и двумерную графику, и в этой главе вы увидите, как это делается, на примере создания двумерной игры. Это классическая детская игра Memory: карты выкладываются рубашкой вверх, переворачиваются щелчком и считается количество совпадений. В процессе вы познакомитесь с основами разработки двумерных игр в Unity.

Изначально Unity предназначался для создания трехмерных игр, но у него есть и другие варианты применения. Начиная с версии 4.3, выпущенной в конце 2013 года, в Unity появилась возможность отображения двумерной графики, хотя и раньше с помощью этого инструмента разрабатывали двумерные игры (особенно мобильные, в создании которых помогла кросс-платформенная природа Unity). Изначально для эмуляции двумерной графики в трехмерных сценах требовался сторонний фреймворк (например, 2D Toolkit от Unikron Software). В конечном счете основной редактор и игровой движок поменяли, встроив в него двумерную графику. Именно с этой функциональностью и знакомит данная глава.

Рабочий процесс при создании двумерной и трехмерной графики в Unity примерно одинаков и включает в себя импорт графических ресурсов, перетаскивание их в сцену и написание сценариев, которые затем будут присоединены к объектам. Основной вид ресурсов, необходимых для создания двумерной графики, называется *спрайтом*.

ОПРЕДЕЛЕНИЕ *Спрайтом* (sprite) называется отображаемое непосредственно на экране двумерное изображение, в то время как такие же изображения, отображаемые на поверхности трехмерных моделей, называются *текстурами*.

Импорт спрайтов по большей части напоминает импорт текстур (см. главу 4). С технической точки зрения спрайты представляют собой объекты в трехмерном пространстве, обладающие плоской поверхностью и ориентированные относительно оси *Z*. Они повернуты в одном направлении, поэтому камеру можно нацелить прямо на них, при этом игроки смогут наблюдать только перемещения вдоль осей *X* и *Y* (то есть в плоскости).

Координатные оси обсуждались в главе 2. Надеюсь, вы помните, что для получения третьего измерения добавляется ось *Z*, перпендикулярная уже знакомым вам осям *X* и *Y*. Именно этими осями представлены два измерения, с которыми мы будем работать.

5.1. Подготовка к работе с двумерной графикой

Мы собираемся воспроизвести классическую игру Memory. Опишем ее для тех, кто не знаком с правилами. Набор парных карт разложен рубашкой вверх. Их расположение игроку неизвестно. Он переворачивает карты по две, пытаясь найти совпадающие. Не совпавшие карты переворачиваются назад, а игрок делает следующую попытку.

Макет игры показан на рис. 5.1; сравните его с планом, который мы составляли в главе 2.

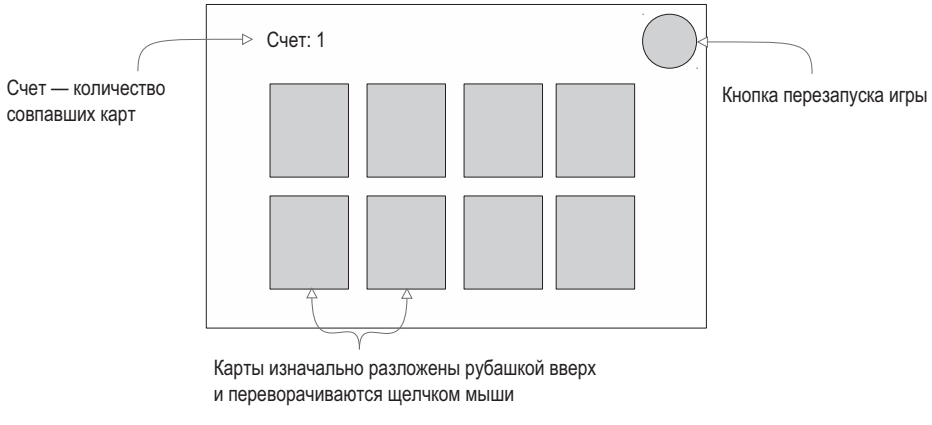


Рис. 5.1. Макет игры Memory

На этот раз макет показывает то, что видит играющий (в то время как макет трехмерной сцены описывает пространство вокруг игрока и местоположение камеры, обеспечивающей возможность смотреть по сторонам). Теперь, когда стало ясно, что мы собираемся создать, время приступить к делу!

5.1.1. Подготовка проекта

Первым делом нужно отобразить всю графику. Как и при создании трехмерного демонстрационного ролика, начнем с помещения в сцену игровых объектов, а затем напишем программы, определяющие их функциональность.

То есть нужно создать все объекты с рис. 5.1: рубашку карт, их лицевую сторону, в одном углу игрового пространства — табло, на котором будет отображаться количество набранных очков, а в другом — кнопку перезагрузки. Еще нужен фон. Весь необходимый арсенал показан на рис. 5.2.

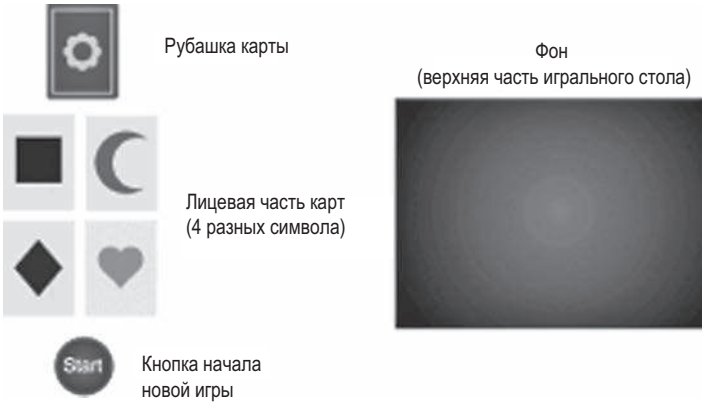


Рис. 5.2. Графические ресурсы для игры Memory

СОВЕТ Готовый проект со всеми графическими ресурсами можно скачать со страницы https://www.manning.com/books/unity-in-action-second-edition посвященного этой книге сайта. Скопируйте данные там изображения и используйте их в своем проекте.

Соберите все требуемые изображения и создайте в Unity новый проект. В нижней части окна есть пара кнопок (рис. 5.3) для переключения между 2D- и 3D-режимами. В предыдущих главах вы работали только с трехмерной графикой, а так как именно этот режим создания проектов используется по умолчанию, мы не обращали на данную настройку внимания. Сейчас же нужно переключиться в режим работы с двумерной графикой.

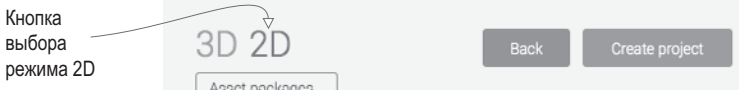


Рис. 5.3. Эти кнопки позволяют создавать как двумерные, так и трехмерные проекты

РЕЖИМ 2D ДЛЯ РЕДАКТОРА И ВКЛАДКИ SCENE

Выбирая двумерный или трехмерный режим при создании нового проекта, мы влияем на пару настроек Unity, которые позднее всегда можно поменять. Это режим работы самого редактора и режим отображения вкладки Scene. Последний определяет способ показа сцены в Unity; кнопка перехода от одного режима к другому находится в верхней части вкладки Scene.

Кнопка на вкладке Scene, меняющая способ отображения

Для изменения режима работы редактора откройте меню Edit, наведите указатель мыши на строчку Project Settings и выберите в дополнительном меню вариант Editor. Среди различных настроек на панели Inspector появится раскрывающийся список Default Behavior Mode, в котором можно выбрать вариант 3D или 2D.

Список Default Behavior Mode, открываемый командой Edit > Project Settings > Editor

В двумерном режиме все изображения импортируются как спрайты; в главе 4 вы видели, что в обычном режиме они превращаются в текстуры. Кроме того, в режиме 2D сцены лишены настроек освещения. Оно просто не требуется. Если когда-нибудь вы захотите получить трехмерную сцену без освещения, удалите автоматически создаваемый источник света и отключите скайбокс (щелкните на маленьком кружочке рядом с полем выбора файла и выберите в списке вариант None).

Итак, мы создали новый проект, выбрав для него режим 2D. Пришло время добавить в сцену изображения.

5.1.2. Отображение спрайтов

Выполните импорт всех изображений, перетащив их на вкладку Project; убедитесь, что изображения импортированы как спрайты, а не как текстуры. Для этого нужно выделить любую картинку и посмотреть настройки ее импорта на панели Inspector, а именно поле Texture Type. Затем перетащите спрайт table_top (фоновое изображение) со вкладки Project в пустую сцену. Сохраните сцену. Как и в случае с сеточными объектами, у выделенного спрайта на панели Inspector окажется компонент Transform; присвойте его полям значения `0, 0, 5`, чтобы поменять положение фонового изображения.

СОВЕТ Еще одна настройка импорта, на которую следует обратить внимание: Pixels To Units. Так как к трехмерному движку Unity двумерную графику добавили относительно недавно, одна единица Unity далеко не всегда соответствует одному пикселу изображения. Можно выбрать вариант 1:1, но я рекомендую оставить значение по умолчанию 100:1 (физический движок при отображении 1:1 работает не совсем корректно, а вариант по умолчанию обеспечивает лучшую совместимость с остальным кодом).

СОЗДАНИЕ АТЛАСОВ

В нашем проекте будут использоваться только одиночные изображения, но в одно изображение можно объединить и набор спрайтов. Когда таким способом комбинируются кадры двумерной анимации, итоговый набор называется *листом спрайтов*. Но для подобной структуры существует еще один, более общий термин — *атлас* (atlas).

Анимированные спрайты часто встречаются в двумерных играх и будут реализованы в следующей главе. Набор кадров можно импортировать в виде отдельных изображений, но в играх все кадры анимации обычно существуют в виде листа спрайтов. По сути, все кадры выглядят как сетка на одном большом изображении.

Атласы применяются не только для объединения кадров анимации, но и для статичных изображений. Дело в том, что они оптимизируют производительность: 1) уменьшая количество пустого пространства в изображениях путем их плотной упаковки, 2) уменьшая количество вызовов отрисовки (draw calls) видеокарты (каждое новое загруженное изображение означает дополнительную нагрузку на видеокарту).

Для создания атласов спрайтов применяются внешние инструменты, например TexturePacker (см. приложение Б), и это прекрасно работает. Но в Unity существует упаковщик спрайтов, автоматически соединяющий друг с другом наборы изображений. По умолчанию он отключен и включается в настройках редактора, для доступа к которым нужно выбрать в меню Edit команду Project Settings, а затем Editor. После этого следует указать имя в поле Packing Tag. Оно находится среди настроек импорта спрайта. Помеченные одним тегом спрайты Unity упакует в атлас. Более подробно эта тема освещена на странице: https://docs.unity3d.com/ru/current/Manual/SpritePacker.html.

Несложно понять, почему мы приравняли к `0` координаты *X* и *Y*. Ведь спрайт должен заполнить весь экран, поэтому его нужно расположить в центре. А вот присвоение значения `5` координате *Z* может показаться странным. Разве для двумерной графики имеют значение какие-либо координаты, кроме *X* и *Y*? Дело в том, что эти координаты определяют положение плоского объекта на экране, а координата *Z* вступает в игру, когда объекты нужно положить друг на друга. Чем меньше ее значение, тем ближе к камере находится спрайт (рис. 5.4). Соответственно, у спрайта, служащего фоном, значение *Z* должно быть максимальным. Мы присвоили ему положительную координату *Z*, у а всех остальных объектов она будет иметь нулевое или отрицательное значение.

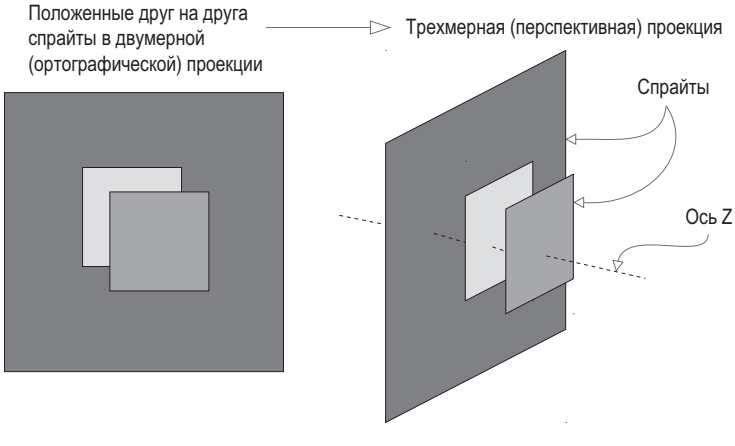


Рис. 5.4. Расположение спрайтов относительно оси Z

Положение остальных спрайтов будет указываться с точностью до двух знаков после запятой из-за упоминавшегося выше параметра Pixels To Units. Соотношение 100:1 означает, что 100 пикселов изображения соответствуют 1 измерительной единице в Unity; или, если посмотреть с другой стороны, 1 пиксел равен 0,01 единицы. Но перед тем, как мы начнем добавлять в сцену новые спрайты, нужно настроить камеру.

5.1.3. Переключение камеры в двумерный режим

Отредактируем настройки основной камеры. Может показаться, что, переключив вкладку Scene в режим работы с двумерной графикой, вы увидите, как будет выглядеть готовая игра. Но на самом деле это не так.

ВНИМАНИЕ Режим вкладки Scene не имеет никакого отношения к тому, что показывает камера запущенной игры.

Дело в том, что настройки игровой камеры не зависят от режима вкладки Scene. Зачастую это выгодно: например, вы можете вернуть вкладку Scene в трехмерный режим для работы над какими-то эффектами. Но именно поэтому то, что вы видите в редакторе Unity, вовсе не эквивалентно тому, что вы можете увидеть в игре, и новички об этом часто забывают.

Самая важная настройка камеры называется Projection. Скорее всего, наша камера уже обладает нужной проекцией, ведь мы создали новый проект в двумерном режиме. Но важно знать, как проверяются параметры камеры. Выделите камеру на вкладке Hierarchy, чтобы ее настройки появились на панели Inspector, как показано на рис. 5.5. Для работы с трехмерной графикой параметр Projection должен иметь значение Perspective, двумерная же графика требует значения Orthographic.

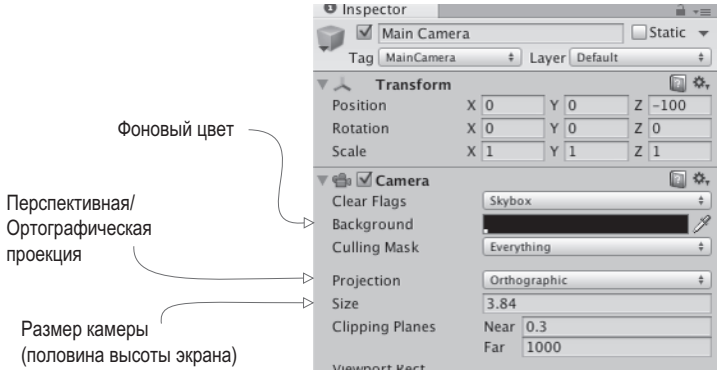


Рис. 5.5. Настройки камеры при работе с двумерной графикой

ОПРЕДЕЛЕНИЕ *Ортографическим* называется изображение с камеры, не имеющей видимой перспективы. Этим ортографическая камера отличается от перспективной, в которой ближе расположенные объекты выглядят больше и уменьшаются с увеличением расстояния до камеры.

В данном случае важный параметр Projection в редактировании не нуждается, чего не скажешь о прочих настройках камеры. Обратите внимание на параметр Size; он определяет

размер поля зрения камеры от центра экрана до его верхнего края. Другими словами, этому параметру присваивается значение, равное половине желаемого размера экрана в пикселах. Если позднее указать такое же разрешение для развертываемой игры, вы получите графику пиксел в пиксел.

ОПРЕДЕЛЕНИЕ *Пиксел в пиксел* (pixel-perfect) означает, что один экранный пиксел соответствует одному пикселу изображения (в противном случае видеокарта может слегка размыть изображение, растягивая его под размер экрана).

Предположим, нужно попасть пиксел в пиксел при разрешении экрана 1024 × 768. Это означает, что высота камеры должна составить 384 пиксела. Поделив это число на 100 (с учетом соотношения пикселов и единиц измерения), получим размер камеры, равный 3,84. То есть вычисления ведутся по формуле `РАЗМЕР ЭКРАНА / 2 /100f` (`f` указывает на значение типа `float`, а не типа `int`). Так как наше фоновое изображение имеет размер 1024 × 768 (для проверки выделите этот ресурс), очевидно, что для камеры идеально подойдет значение 3,84.

Еще на панели Inspector следует поменять фоновый цвет камеры и ее положение по координате *Z*. Как уже упоминалось при рассмотрении спрайтов, чем больше значение координаты *Z*, тем дальше расположен объект. Поэтому отодвинем камеру подальше в область отрицательных значений *Z*, например `0`, `0`, `-100`. В качестве фонового цвета, наверное, лучше всего выбрать черный; по умолчанию фон имеет синий цвет, и, если экран окажется шире фонового изображения (что вполне вероятно), синие полоски по сторонам будут выглядеть странно. Щелкните на образце цвета Background и выберите черный цвет.

Сохраните сцену под именем Scene и нажмите кнопку Play; вы увидите игровое поле, заполненное спрайтом, изображающим поверхность стола. Думаю, вы согласитесь, что путь до этой точки был не совсем очевидным (напомню, что причиной этого является трехмерный игровой движок Unity, в который только недавно добавили возможность работы с двумерной графикой). Итак, пустая поверхность стола готова. Можно выкладывать на нее карты.

5.2. Создание карт и превращение их в интерактивные объекты

Итак, мы импортировали и подготовили к работе все нужные изображения, можно приступать к созданию основных игровых объектов — карт. В игре Memory они выкладываются рубашкой вверх и переворачиваются на короткий момент после щелчка на паре карт. Подобной функциональностью обладают объекты, состоящие из положенных друг на друга спрайтов. Для них потребуется написать код, заставляющий карты переворачиваться по щелчку мыши.

5.2.1. Объект из спрайтов

Перетащите в сцену одну карту. Используйте изображение лицевой стороны, так как поверх будет добавлена другая карта, скрывающая рисунок первой. С технической точки зрения положение этого объекта пока не имеет значения, но в конечном счете карта должна будет оказаться в определенной точке, поэтому присвойте полям Position

значения `-3`, `1`, `0`. Перетащите в сцену спрайт `card_back` и сделайте его потомком предыдущего спрайта. Напоминаю, что для этого на вкладке Hierarchy нужно перетащить дочерний объект на родительский. Установите для нового спрайта положение `0`, `0`, `-0.1`. Это положение относительно предка, то есть фактически мы поместили рубашку карты в точку с теми же координатами *X* и *Y*, но чуть ближе по координате *Z*.

СОВЕТ Вместо инструментов Move, Rotate и Scale, которыми мы пользовались в трехмерном режиме, теперь доступен единый инструмент манипуляции *Rect Tool*. Он автоматически активируется после перехода в двумерный режим. Кроме того, его можно включить нажатием крайней правой кнопки на навигационной панели в верхнем левом углу Unity. Когда этот инструмент активен, перетаскивание объектов приводит к выполнению всех трех операций (перемещения/поворота/масштабирования) в двух измерениях.

Рубашка карты положена на свое место, как показано на рис. 5.6, и все готово для создания интерактивного объекта, реагирующего на щелчок мыши.

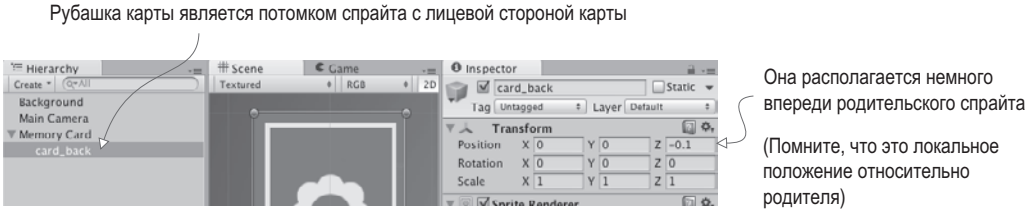


Рис. 5.6. Иерархическое связывание и местоположение спрайта с рубашкой карты

5.2.2. Код ввода с помощью мыши

Чтобы карта реагировала на щелчки мыши, требуется такой компонент, как коллайдер. У новых спрайтов он отсутствует, поэтому щелкать на них бесполезно. Мы присоединим коллайдер к корневому объекту, а не к рубашке карты, соответственно, к щелчкам будет восприимчива лицевая сторона. Выделите корневой объект на вкладке Hierarchy (щелкнув на карте в сцене, вы выделите рубашку, так как она располагается сверху) и щелкните на кнопке Add Component в нижней части панели Inspector. Выделите строку Physics 2D (строка Physics относится к физике трехмерных игр) и выберите вариант Box collider.

Теперь нужен сценарий, который заставит карту реагировать на действия игрока. Создайте сценарий с именем `MemoryCard.cs` и присоедините его к корневой карте (а не к рубашке). Вот как выглядит код, заставляющий карту сгенерировать отладочное сообщение в ответ на щелчок мыши.

Листинг 5.1. Генерация отладочных сообщений по щелчку

```
using UnityEngine;
using System.Collections;

public class MemoryCard : MonoBehaviour {
    public void OnMouseDown() {
        Debug.Log("testing 1 2 3");
    }
}
```

Эта функция вызывается после щелчка на объекте.

Пока мы только генерируем текстовое сообщение, выводимое на консоль.

СОВЕТ Если у вас пока нет такой привычки, приучайте себя распределять ресурсы по папкам; создайте для сценариев отдельную папку и перетащите туда все сценарии. Это можно сделать непосредственно на вкладке Project. Избегайте имен, на которые реагирует Unity: Resources, Plugins, Editor и Gizmos. О назначении этих папок мы поговорим позднее, а пока просто помните, что пользовательские папки так называть нельзя.

Мы можем щелкать на карте! Подобно методу `Update()`, функция `OnMouseDown()` также происходит от класса `MonoBehaviour`, именно она вызывает реакцию на щелчок мыши. Запустите воспроизведение игры и убедитесь, что на консоли стало появляться сообщение. Но вывод на консоль был сделан только с целью тестирования, мы же хотим *открыть* карту.

5.2.3. Открытие карты по щелчку

Отредактируйте код в соответствии со следующим листингом (запустить его пока невозможно, но это не повод для беспокойства).

Листинг 5.2. Сценарий, скрывающий рубашку после щелчка на карте

```
using UnityEngine;
using System.Collections;

public class MemoryCard : MonoBehaviour {
    [SerializeField] private GameObject cardBack;
  public void OnMouseDown() {
      if (cardBack.activeSelf) {
          cardBack.SetActive(false);
      }
  }
}
```

Переменная, которая появляется на панели Inspector.

Код деактивации запускается только в случае, когда объект активен/видим.

Делаем объект неактивным/невидимым.

В этом сценарии есть два ключевых дополнения: ссылка на объект сцены и деактивирующий данный объект метод `SetActive()`. Из предыдущих глав вы уже знаете, как сделать ссылку: переменная помечается как сериализованная, а затем на эту переменную на панели Inspector перетаскивается объект со вкладки Hierarchy. После этого код начнет влиять на объект сцены.

Метод `SetActive`, которому было передано значение `false`, деактивирует любой объект `GameObject`, делая его невидимым. Так что если теперь перетащить объект `card_back` на переменную этого сценария на панели Inspector, в процессе игры карта начнет исчезать после щелчка на ней. Исчезнувшая рубашка открывает лицевую сторону карты; как видите, мы решили еще одну важную задачу! Но на столе пока всего одна карта, давайте исправим этот недостаток.

5.3. Отображение набора карт

Мы написали программу, которая отображает сначала рубашку карты, а после щелчка на ней — лицевую сторону. Но карта пока только одна, а для игры требуется целый набор с разными изображениями. Давайте разложим карты. Для этого мы воспользуемся

парой уже знакомых вам по предыдущим главам концепций, а также познакомим вас с новыми понятиями. В главе 3 вы научились, во-первых, применять невидимый компонент `SceneController`, во-вторых, создавать экземпляры объекта. Именно этот компонент позволит сопоставить различным картам разные изображения.

5.3.1. Программная загрузка изображений

В игре существует четыре варианта изображений для карт. Все восемь карт на столе (по две для каждого символа) мы получим клонированием одного объекта, поэтому изначально рисунок на них будет одинаковым. Менять его придется при помощи сценария, загружая изображения программно.

Напишем простой тестовый код (позже мы его заменим), чтобы продемонстрировать процесс программной загрузки изображений. Первым делом добавьте в сценарий `MemoryCard` код из следующего листинга.

Листинг 5.3. Тестовый код, демонстрирующий изменение картинки спрайта

```
...
[SerializeField] private Sprite image;  <-- Ссылка на загружаемый ресурс Sprite.
void Start() {
    GetComponent<SpriteRenderer>().sprite = image;  <-- Сопоставляем спрайт компоненту SpriteRenderer.
}
...
```

После сохранения этого сценария на панели Inspector появилась новая переменная, которая была указана как сериализованная. Перетащите на ячейку Image какой-либо спрайт со вкладки Project (выберите одно из лицевых изображений, кроме того, которое уже есть в сцене). Запустите воспроизведение сцены, и вы увидите на карте новое изображение.

Чтобы понять смысл кода, вспомним, что представляет собой компонент `SpriteRenderer`. На рис. 5.7 мы видим, что рубашка карты имеет всего два компонента: стандартный компонент `Transform`, присутствующий у всех объектов сцены, и новый компонент `SpriteRenderer`. Он делает из объекта спрайт и определяет, какой ресурс будет отображен на этом спрайте. Обратите внимание, что первое свойство этого компонента называется `Sprite` и связано с одним из спрайтов на вкладке Project; этим свойством можно управлять программно, что, собственно, и происходит внутри нашего сценария.

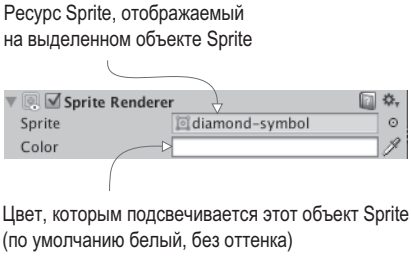


Рис. 5.7. Компонент SpriteRenderer, присоединенный к объекту-спрайту

Работая с компонентом `CharacterController` и нашими сценариями в предыдущих главах, вы видели, что метод `GetComponent()` возвращает другие компоненты для одного и того же объекта, поэтому воспользуемся им для ссылки на объект `SpriteRenderer`. Свойству `Sprite` объекта `SpriteRenderer` можно присвоить любой ресурс, поэтому в коде ему присваивается объявленная в верхней части переменная `Sprite` (в редакторе мы заполнили ее одним из спрайтов).

Как видите, это не слишком сложно! Но это всего одно изображение, а у нас их четыре. Поэтому удалите новый фрагмент кода из листинга 5.3 (он был нужен только для демонстрации), чтобы подготовиться к следующему разделу.

5.3.2. Выбор изображения из компонента SceneController

В главе 3 мы создавали невидимый объект для управления генерацией объектов. Воспользуемся этим приемом еще раз, но теперь невидимый объект будет контролировать более абстрактные, не связанные с объектами сцены свойства. Для начала создайте пустой объект `GameObject` (для этого в меню GameObject нужно выбрать команду Create Empty). На вкладке Project создайте сценарий `SceneController.cs` и перетащите этот ресурс на контроллер `GameObject`. Первым делом добавьте содержимое следующего листинга (листинг 5.4) в сценарий `MemoryCard` вместо того, что было в листинге 5.3.

Листинг 5.4. Новые открытые методы в сценарии MemoryCard.cs

```
...
[SerializeField] private SceneController controller;

private int _id;
public int id {
    get {return _id;}
}

public void SetCard(int id, Sprite image) {
   _id = id;
   GetComponent<SpriteRenderer>().sprite = image;
}
...
```

Добавленная функция чтения (идиома из таких языков, как C# и Java).

Открытый метод, которым могут пользоваться другие сценарии для передачи указанному объекту новых спрайтов.

Строка SpriteRenderer, совсем как в удаленном примере кода.

В отличие от предыдущего листинга, изображения спрайта задаются методом `SetCard()` вместо метода `Start()`. Это открытый метод, принимающий спрайт в качестве параметра, поэтому его можно вызывать из других сценариев и устанавливать изображение для объекта. Обратите внимание, что метод `SetCard()` принимает в качестве параметра ID и код его сохраняет. Пока этот параметр не нужен, но скоро мы напишем код сравнения карт, и сравнение будет производиться именно на основе значения ID.

ПРИМЕЧАНИЕ Возможно, раньше вы пользовались языком программирования, в котором отсутствовали такие понятия, как *метод чтения* (getter) и *устанавливающий метод* (setter). Это функции, которые запускаются при попытке получить доступ к связанному с ними свойству (например, получить значение `card.id`). Они применяются для разных целей, но в рассматриваемом случае свойство `id` предназначено только для чтения, поэтому вы можете только прочитать его значение, но не задать его.

Наконец, обратите внимание, что в коде есть переменная для контроллера; когда `SceneController` начнет клонировать карты для заполнения сцены, картам потребуется ссылка на этот контроллер для вызова его открытых методов. Как всегда, когда код ссылается на объекты сцены, нужно перетащить объект-контроллер из редактора Unity на ячейку переменной на панели Inspector. Достаточно сделать это для одной карты, все появившиеся позже копии получат эту ссылку автоматически.

Теперь, когда в сценарии `MemoryCard` появится новый код, введите в сценарий `SceneController` содержимое следующего листинга.

Листинг 5.5. Первый проход компонента SceneController в игре Memory

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
    [SerializeField] private MemoryCard originalCard;
    [SerializeField] private Sprite[] images;

   void Start() {
       int id = Random.Range(0, images.Length);
         originalCard.SetCard(id, images[id]);
   }
}
```

Ссылка для карты в сцене.

Массив для ссылок на ресурсы-спрайты.

Вызов открытого метода, добавленного в сценарий MemoryCard.

Этот короткий фрагмент демонстрирует принцип управления картами со стороны контроллера `SceneController`. Большая часть представленного кода уже должна быть вам знакома (например, перетаскивание объекта-карты в редакторе Unity на ячейку переменной на панели Inspector), но с массивами изображений вы раньше не сталкивались. Как показано на рис. 5.8, на панели Inspector можно задать набор элементов. Введите 4 в поле для размера массива, а затем перетащите спрайты с изображениями карт на ячейки элементов массива. Эти спрайты станут доступными в массиве, как и все остальные ссылки на объекты.

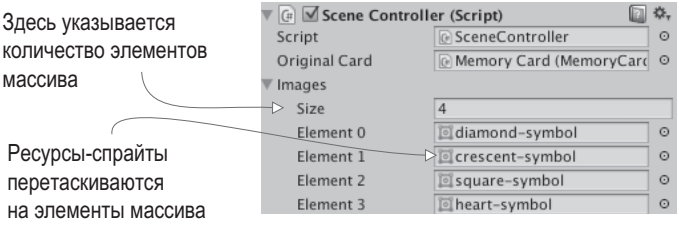


Рис. 5.8. Заполнение массива спрайтов

Метод `Random.Range()` применялся в главе 3, надеюсь, вы помните, как с ним работать. Точные граничные значения в данном случае не важны, но имейте в виду, что минимальное значение входит в диапазон и может быть возвращено, в то время как возвращаемое значение всегда меньше максимального.

Нажмите кнопку Play для воспроизведения нового кода. Вы увидите, что при каждом запуске сцены меняется изображение на открываемой карте. Пришло время выложить остальные карты.

5.3.3. Экземпляры карт

У компонента `SceneController` ссылка на объект-карту уже есть, поэтому остается воспользоваться методом `Instantiate()` (см. следующий листинг) и получить набор карт. Аналогичным способом мы генерировали объекты в главе 3.

Листинг 5.6. Восьмикратное копирование карты и выкладывание карт на стол

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
  public const int gridRows = 2;
  public const int gridCols = 4;
  public const float offsetX = 2f;
  public const float offsetY = 2.5f;

  [SerializeField] private MemoryCard originalCard;
  [SerializeField] private Sprite[] images;

  void Start() {
    Vector3 startPos = originalCard.transform.position;

    for (int i = 0; i < gridCols; i++) {
      for (int j = 0; j < gridRows; j++) {
        MemoryCard card;
        if (i == 0 && j == 0) {
          card = originalCard;
        } else {
          card = Instantiate(originalCard) as MemoryCard;
        }

        int id = Random.Range(0, images.Length);
        card.SetCard(id, images[id]);

        float posX = (offsetX * i) + startPos.x;
        float posY = -(offsetY * j) + startPos.y;
        card.transform.position = new Vector3(posX, posY, startPos.z);

      }
    }
  }
}
```

Значения, указывающие количество ячеек сетки и их расстояние друг от друга.

Положение первой карты; положение остальных карт отсчитывается от этой точки.

Вложенные циклы, задающие как столбцы, так и строки сетки.

Ссылка на контейнер для исходной карты или ее копий.

В двумерной графике требуется только смещение по X и Y; значение Z не меняется.

Этот сценарий намного длиннее предыдущих, но объяснять тут особо нечего, потому что большинство дополнений представляют собой обычные объявления переменных и математические вычисления. Самым странным фрагментом кода является, вероятно, выражение, начинающее условный оператор `if (i == 0 && j == 0)`. Оно заставляет

или выбрать исходный объект-карту для первой ячейки сетки, или клонировать этот объект для остальных ячеек. Исходная карта в сцене уже есть, поэтому, копируя ее на каждом шаге цикла, мы получим слишком много карт. Карты раскладываются путем смещения в соответствии с количеством шагов цикла.

СОВЕТ Двумерные объекты, как и трехмерные, можно двигать по экрану, меняя значение `transform.position`, причем это значение может расти внутри метода `Update()`. Когда мы занимались перемещениями объекта-игрока, вы видели, что при прямом редактировании параметра `transform.position` нельзя добавить распознавание столкновений. Для получения двумерных объектов с распознаванием столкновений нужно добавить к ним компоненты группы `Physics2D` и отредактировать, например, параметр `rigidbody2D.velocity`.

Запустите код, и в сцене появится сетка из восьми карт, показанная на рис. 5.9. Теперь осталось разбить карты на пары.

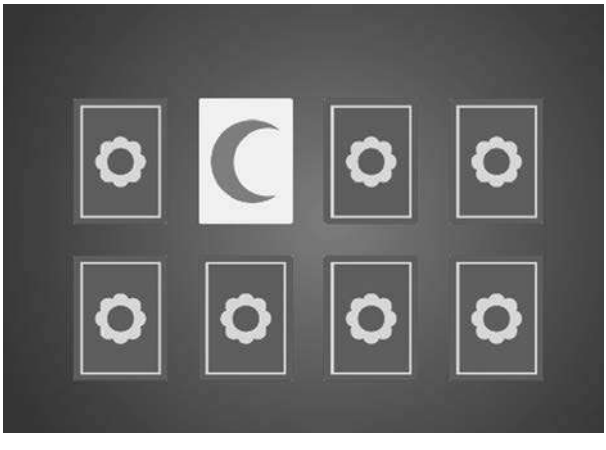


Рис. 5.9. Сетка из восьми карт, которые открываются щелчком мыши

5.3.4. Тасуем карты

Сейчас карты не зависят друг от друга. Создадим из их идентификаторов (чисел от 0 до 3, сопоставленных каждой паре карт) массив и перемешаем его элементы. На основе этого массива и будет осуществляться раскладка карт.

Листинг 5.7. Раскладка карт из перемешанного списка

```
...
void Start() {
   Vector3 startPos = originalCard.transform.position;

   int[] numbers = {0, 0, 1, 1, 2, 2, 3, 3};
   numbers = ShuffleArray(numbers);

   for (int i = 0; i < gridCols; i++) {
```

Большая часть листинга — это контекст, показывающий, куда вставлять дополнения.

Объявляем целочисленный массив с парами идентификаторов для всех четырех спрайтов с изображениями карт.

Вызываем функцию, перемешивающую элементы массива.

```
      for (int j = 0; j < gridRows; j++) {
        MemoryCard card;
        if (i == 0 && j == 0) {
          card = originalCard;
        } else {
          card = Instantiate(originalCard) as MemoryCard;
        }
        int index = j * gridCols + i;
        int id = numbers[index];
        card.SetCard(id, images[id]);

        float posX = (offsetX * i) + startPos.x;
        float posY = -(offsetY * j) + startPos.y;
        card.transform.position = new Vector3(posX, posY, startPos.z);
      }
   }
}

private int[] ShuffleArray(int[] numbers) {
   int[] newArray = numbers.Clone() as int[];
   for (int i = 0; i < newArray.Length; i++ ) {
      int tmp = newArray[i];
      int r = Random.Range(i, newArray.Length);
      newArray[i] = newArray[r];
      newArray[r] = tmp;
   }
   return newArray;
}
...
```

Идентификаторы получаем из перемешанного списка, а не из случайных чисел.

Реализация алгоритма тасования Кнута.

Нажмите кнопку Play, и сетка заполнится картами из набора, содержащего ровно по две карты каждого вида. Массив карт пропущен через алгоритм тасования *Кнута* (еще его называют алгоритмом *Фишера — Йетса*). Это простой, но эффективный способ перемешать элементы массива. Алгоритм циклически просматривает элементы массива и каждый элемент меняет местами с другим, случайным образом выбранным элементом.

Щелчком мыши можно открыть все карты, но в игре Memory карты должны открываться парами, поэтому требуется дополнительный код.

5.4. Совпадения и подсчет очков

До полнофункциональной игры Memory не хватает проверки совпадений. Карты уже положены на стол и открываются по щелчку, но они никак не связаны друг с другом. Нам же нужно, чтобы каждая открытая пара проверялась на совпадение.

Эта абстрактная логическая схема — проверка совпадений и соответствующая реакция — требует, чтобы карты посылали уведомление о щелчке сценарию `SceneController`. Добавьте в этот сценарий следующий код.

Листинг 5.8. Сценарий SceneController, следящий за открываемыми картами

```
...
private MemoryCard _firstRevealed;
private MemoryCard _secondRevealed;

public bool canReveal {
    get {return _secondRevealed == null;}
}
...
public void CardRevealed(MemoryCard card) {
    // изначально пусто
}
...
```

Функция чтения, возвращающая значение false, если вторая карта уже открыта.

Метод `CardRevealed()` мы напишем чуть позже; пока он вставлен в код как вспомогательный пустой метод, на который можно ссылаться в сценарии `MemoryCard.cs` без появления ошибок компиляции. Обратите внимание, что мы снова применяем функцию чтения, на этот раз чтобы определить, открыта ли вторая карта; возможность открыть карту дается игроку только при отсутствии в сцене двух открытых карт.

Еще раз отредактируем сценарий `MemoryCard.cs`, добавив туда вызов метода (пока пустого), который будет информировать компонент `SceneController` о щелчке на карте. Внесите в код сценария `MemoryCard.cs` изменения в соответствии со следующим листингом.

Листинг 5.9. Отредактированный сценарий MemoryCard.cs для открытия карт

Проверка свойства контроллера canReveal, гарантирующая, что одновременно можно открыть всего две карты.

```
...
public void OnMouseDown() {
   if (cardBack.activeSelf && controller.canReveal) {
     cardBack.SetActive(false);
     controller.CardRevealed(this);
   }
}

public void Unreveal() {
   cardBack.SetActive(true);
}
...
```

Уведомляем контроллер об открытии этой карты.

Открытый метод, позволяющий компоненту SceneController снова скрыть карту (вернув на место спрайт card_back).

Если поместить в метод `CardRevealed()` оператор отладки для проверки взаимодействия объектов, после щелчка на любой карте будет появляться тестовое сообщение. Но для начала разберемся с одной парой открытых карт.

5.4.1. Сохранение и сравнение открытых карт

Объект-карта был передан в метод `CardRevealed()`, поэтому мы начнем фиксировать открытые карты.

Листинг 5.10. Учет открытых карт в сценарии SceneController

```
...
public void CardRevealed(MemoryCard card) {
  if (_firstRevealed == null) {
    _firstRevealed = card;
  } else {
    _secondRevealed = card;
    Debug.Log("Match? " + (_firstRevealed.id == _secondRevealed.id));
  }
}
...
```

Сохраняем карты в одну из двух переменных, в зависимости от того, какая из них свободна.

Сравнение идентификаторов двух открытых карт.

Этот код сохраняет открытые карты в одну из двух переменных. Если первая переменная свободна, заполняется именно она; если ей уже присвоен другой объект-карта, заполняется вторая переменная, а затем проверяется, совпадают ли идентификаторы этих объектов. В зависимости от результата оператор отладки выводит на консоль значение `true` или `false`.

Реакция на совпадения пока отсутствует, код только проверяет, совпадают карты или нет. Поэтому давайте запрограммируем ответ.

5.4.2. Убираем несовпавшие карты

Воспользуемся сопрограммой, ведь если карты не совпали, выполнение программы следует прервать, чтобы у игрока появилась возможность рассмотреть открытые карты. Сопрограммы детально рассматривались в главе 3; коротко говоря, именно сопрограмма позволит сделать паузу в процедуре проверки совпадений. Добавьте в сценарий `SceneController` следующий код.

Листинг 5.11. Сценарий SceneController, подсчитывающий очки или скрывающий карты при отсутствии совпадения

```
...
private int _score = 0;
...
public void CardRevealed(MemoryCard card) {
  if (_firstRevealed == null) {
    _firstRevealed = card;
  } else {
    _secondRevealed = card;
    StartCoroutine(CheckMatch());
  }
}

private IEnumerator CheckMatch() {
  if (_firstRevealed.id == _secondRevealed.id) {
    _score++;
    Debug.Log("Score: " + _score);
  }
  else {
    yield return new WaitForSeconds(.5f);

    _firstRevealed.Unreveal();
```

Добавляем в список в верхней части сценария SceneController.

Единственная отредактированная строка в функции – вызывает сопрограмму после открытия двух карт.

Увеличиваем счет на единицу при совпадении идентификаторов открытых карт.

Закрываем несовпадающие карты.

```
    _secondRevealed.Unreveal();
  }

  _firstRevealed = null;
  _secondRevealed = null;
}
...
```

Очищаем переменные вне зависимости от того, было ли совпадение.

Сначала мы добавили переменную для слежения `_score`; затем после открытия второй карты в метод `CheckMatch()` была загружена сопрограмма с двумя вариантами кода — выбор варианта зависит от того, произошло ли совпадение. При совпадении выполнение кода не прерывается; команда `yield` просто пропускается. В противном случае происходит остановка на полсекунды, а потом для обеих карт вызывается скрывающий их метод `Unreveal()`. В конце, вне зависимости от совпадения, переменные, в которых хранятся карты, обнуляются, давая игроку возможность открыть следующие карты.

В процессе игры карты, которые не совпали, будут некоторое время демонстрироваться игроку. Количество совпадений выводится в виде отладочных сообщений, нужно же сделать так, чтобы оно выводилось на экран.

5.4.3. Текстовое отображение счета

Добавим в игру пользовательский интерфейс. Это позволит, во-первых, выводить на экран информацию для игрока. Во-вторых, даст игроку способ ввода сведений. Кнопки UI будут обсуждаться в главе 7.

ОПРЕДЕЛЕНИЕ Аббревиатура *UI* (User Interface) означает пользовательский интерфейс. Другой термин — *GUI* (от graphical user interface — графический интерфейс пользователя) — означает визуальную часть интерфейса, то есть текст и кнопки. Зачастую, когда речь заходит о UI, подразумевается GUI.

В Unity есть разные способы отображения текста. Можно, например, создать трехмерный текстовый объект. Это специальный сеточный компонент, поэтому требуется пустой объект, к которому его можно будет присоединить. Выберите в меню GameObject команду Create Empty. Затем щелкните на кнопке Add Component и выберите в разделе Mesh вариант Text Mesh.

ПРИМЕЧАНИЕ Может показаться странным, что в двумерную игру добавляется трехмерный текст. Но не следует забывать, что с технической точки зрения мы работаем с трехмерной сценой, которая демонстрируется через ортографическую камеру и поэтому выглядит плоской. Так что в игру можно добавлять трехмерные объекты — просто они будут отображаться как плоские.

СОВЕТ Я использую стандартный компонент Text Mesh, но для ваших собственных проектов имеет смысл воспользоваться шрифтовыми ресурсами TextMesh Pro. Это улучшенная текстовая система, недавно добавленная в Unity.

Поместите текстовый объект в точку с координатами `-4.75`, `3.65`, `-10`, то есть сдвиньте на 475 пикселов влево и на 365 пикселов вверх, чтобы он оказался в верхнем левом углу стола, и приблизьте к камере, чтобы он отображался поверх других игровых объектов. В нижней части панели Inspector находится параметр Font; щелкните на маленьком кружке,

чтобы вызвать окно выбора файлов, и щелчком выделите единственный доступный шрифт Arial. В поле Text введите слово `Score:`. Корректное позиционирование требует, чтобы параметр Anchor имел значение Upper Left (он контролирует, каким образом растягиваются вводимые буквы), поэтому отредактируйте его, если нужно. По умолчанию текст получится размытым, но это легко исправить, введя параметры с рис. 5.10.

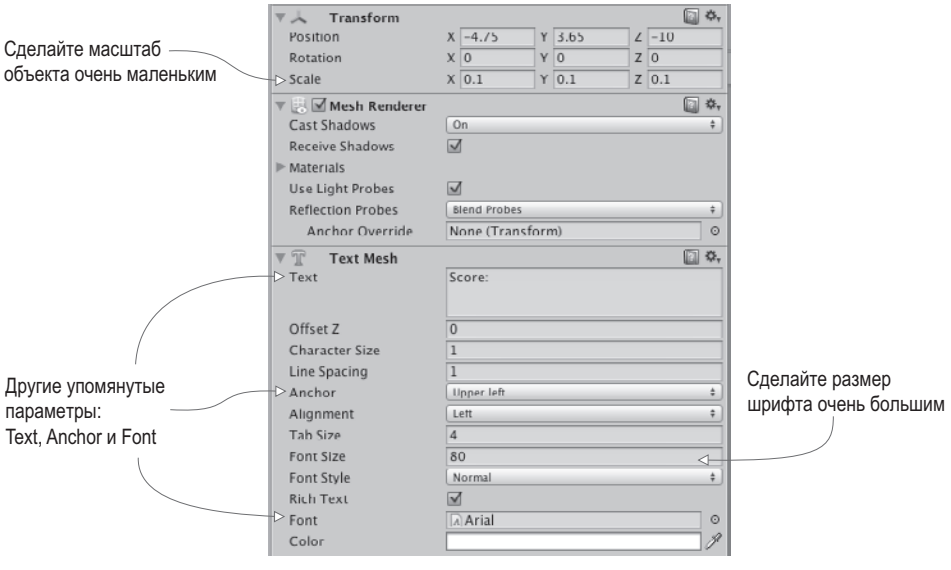


Рис. 5.10. Параметры текстового объекта на панели Inspector, позволяющие получить четкий и резкий текст

Можно импортировать в проект новый шрифт формата TrueType и использовать его, но для наших целей подойдет вариант по умолчанию. Странно, что для получения четкой и резкой надписи шрифтом по умолчанию требуется редактировать его размер. Сначала мы присвоили параметру Font Size компонента TextMesh очень большое значение (у меня это `80`). Затем сделали масштаб этого компонента очень маленьким (например, `0.1`, `0.1`, `0.1`). Увеличение размера шрифта добавило к отображаемому тексту множество пикселов, а масштабирование сгруппировало эти пикселы на меньшем пространстве.

Для дальнейшей манипуляции текстовым объектом нужно внести в код изменения из следующего листинга.

Листинг 5.12. Отображения счета на текстовом объекте

```
...
[SerializeField] private TextMesh scoreLabel;
...
private IEnumerator CheckMatch() {
  if (_firstRevealed.id == _secondRevealed.id) {
    _score++;
    scoreLabel.text = "Score: " + _score;
  }
...
```

Отображаемый текст — это задаваемое свойство текстовых объектов.

Как видите, текст — это свойство объекта, которому можно назначить новую строку. Перетащите текст из сцены на только что добавленную к компоненту `SceneController` переменную и нажмите кнопку Play. Теперь в процессе игры при каждом новом совпадении будет расти счет. Ура, игра работает!

5.5. Кнопка Restart

Наша игра Memory стала полнофункциональной. В нее уже можно играть. Отсутствует только один элемент, который рано или поздно потребуется. Дело в том, что сейчас можно сыграть всего одну партию; после этого нужно выйти из игры и загрузить ее снова. Поэтому давайте добавим элемент управления, позволяющий начинать новую игру без перезагрузки.

Для этого нужно решить две задачи: создать кнопку и сделать так, чтобы щелчок на ней вызывал перезагрузку. Итоговый вид игрового поля показан на рис. 5.11.

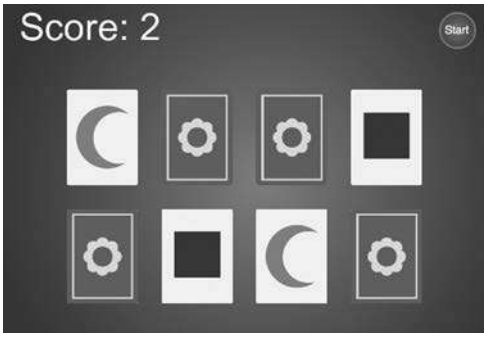


Рис. 5.11. Игра Memory вместе с кнопкой Start

С подобными задачами приходится сталкиваться не только в двумерных играх; пользовательский интерфейс с кнопками и возможность перезагрузки требуются во всех играх. Пример решения таких задач и завершит эту главу.

5.5.1. Метод SendMessage для компонента UIButton

Первым делом добавим в сцену спрайт с изображением кнопки, перетащив его со вкладки Project. Поместите его в точку с координатами `4.5`, `3.25`, `-10`; чтобы кнопка оказалась в верхнем правом углу (то есть сместилась на 450 пикселов вправо и на 325 пикселов вверх) и подвинулась ближе к камере, чтобы отобразиться поверх остальных игровых объектов. Этот объект должен реагировать на щелчки мыши, поэтому добавьте к нему коллайдер (щелкните на кнопке Add Component и выберите в разделе Physics 2D строку Box Collider).

ПРИМЕЧАНИЕ Как упоминалось в предыдущем разделе, в Unity есть разные способы создания элементов UI, к которым относится и появившаяся в последних версиях усовершенствованная система пользовательских интерфейсов. Для начала создадим кнопку из стандартных экранных объектов. А с усовершенствованной UI-функциональностью вы познакомитесь в главе 7; мы рассмотрим пользовательские интерфейсы, идеально встроенные в систему, как для двумерных, так и для трехмерных игр.

Создайте сценарий `UIButton.cs`, добавьте в него код из следующего листинга и назначьте его объекту, изображающему кнопку.

Листинг 5.13. Создание кнопки, допускающей повторное использование

```
using UnityEngine;
using System.Collections;

public class UIButton : MonoBehaviour {
  [SerializeField] private GameObject targetObject;
  [SerializeField] private string targetMessage;
  public Color highlightColor = Color.cyan;

  public void OnMouseEnter() {
    SpriteRenderer sprite = GetComponent<SpriteRenderer>();
    if (sprite != null) {
      sprite.color = highlightColor;
    }
  }
  public void OnMouseExit() {
    SpriteRenderer sprite = GetComponent<SpriteRenderer>();
    if (sprite != null) {
      sprite.color = Color.white;
    }
  }

  public void OnMouseDown() {
    transform.localScale = new Vector3(1.1f, 1.1f, 1.1f);
  }
  public void OnMouseUp() {
    transform.localScale = Vector3.one;
    if (targetObject != null) {
      targetObject.SendMessage(targetMessage);
    }
  }
}
```

Ссылка на целевой объект для информирования о щелчках.

Меняем цвет кнопки при наведении указателя мыши.

В момент щелчка размер кнопки слегка увеличивается.

После щелчка на кнопке отправляем сообщение целевому объекту.

Основная часть кода выполняется внутри функций вида `OnMouseЧтоТо`. Этот набор функций, подобно методам `Start()` и `Update()`, автоматически доступен всем компонентам сценариев в Unity. С функцией `MouseDown` вы встречались в разделе 5.2.2. У объектов с коллайдером все эти функции отвечают за взаимодействия с мышью. Например, `MouseOver` и `MouseExit` — это пара событий, возникающих при наведении указателя на объект: первое событие — появление указателя над объектом, а второе — его уход с объекта. Аналогично, функции `MouseDown` и `MouseUp` — это пара событий, возникающих в момент щелчка. Первое событие генерируется при физическом нажатии кнопки, а второе — когда кнопку отпускают.

Легко заметить, что при наведении указателя мыши на спрайт кнопки цвет спрайта слегка меняется, а в момент щелчка он меняет еще и размер. В обоих случаях эти изменения (цвета или масштаба) возникают после начала взаимодействия с мышью, а затем свойство возвращается в исходное состояние (к белому цвету или к масштабу 1). Масштабирование реализуется через стандартный компонент `transform`,

присутствующий у всех объектов GameObjects. А вот для изменения цвета применен компонент SpriteRenderer, присущий таким объектам, как спрайты; спрайту присваивается цвет, определенный в редакторе Unity через общедоступную переменную.

В момент отпускания кнопки мыши происходит не только возвращение масштаба к значению 1, но и вызов метода SendMessage(). Этот метод вызывает функцию с указанным именем во всех компонентах рассматриваемого объекта GameObject. Как целевой объект для рассылки, так и отправляемое сообщение определены через сериализованные переменные. Такой подход позволяет использовать для всех видов кнопок единый компонент UIButton. Достаточно на панели Inspector сопоставить цель в виде различных кнопок разным объектам.

В таком сильно типизированном языке, как C#, для взаимодействия с целевым объектом нужно знать его тип (к примеру, для вызова открытого метода объекта целевой-объект.SendMessage()). Но сценарии для элементов пользовательского интерфейса могут содержать целевые объекты разных типов, поэтому метод SendMessage() позволяет отправлять таким объектам сообщения без знания типа.

ВНИМАНИЕ Метод SendMessage() не так рационально использует ресурсы процессора, как открытые методы, вызываемые для известных типов (имеются в виду конструкции object.SendMessage("Метод") вместо component.Method()), поэтому пользуйтесь им только в случаях, когда это сильно упростит понимание кода и работу с ним. Как правило, это ситуация, когда сообщение адресуется множеству объектов различных типов; в этом случае отсутствие гибкости наследования или даже интерфейсов будет мешать процессу разработки игры и затруднять эксперименты.

Теперь нужно выполнить связывание общедоступных переменных кнопки на панели Inspector. Выберите цвет выделения (хотя заданный по умолчанию бирюзовый вполне подходит к синей кнопке). Кроме того, перетащите объект SceneController на ячейку целевого объекта и введите в поле для сообщения слово Restart.

Если теперь запустить воспроизведение игры, в верхнем правом углу игрового поля появится кнопка перезагрузки, меняющая цвет при наведении на нее указателя мыши и слегка выступающая в момент щелчка на ней. Но после щелчка вы увидите сообщение об ошибке; на консоли появится информация о том, что сообщение Restart уходит «в никуда». Ведь у нас пока отсутствует метод Restart() в сценарии SceneController. Давайте исправим этот недостаток.

5.5.2. Вызов метода LoadScene

Связанный с кнопкой метод SendMessage() пытается вызвать метод Restart() в сценарии SceneController, поэтому добавьте туда код следующего листинга.

Листинг 5.14. Код перезагрузки уровня в сценарии SceneController

```
...
using UnityEngine.SceneManagement;  <-- Добавляем код управления сценой SceneManagement.
...
public void Restart() {
   SceneManager.LoadScene("Scene");  <-- Эта команда загружает ресурс scene.
}
...
```

Единственное, что делает метод `Restart()`, — вызывает метод `LoadScene()`, вызывающий, в свою очередь, загрузку такого ресурса, как сохраненная сцена (то есть файл, появившийся после команды `Save Scene`). Передайте в метод имя сцены, которую вы хотите загрузить; лично я сохранил ее под именем `Scene`, если же вы использовали другое имя, укажите его.

Нажмите кнопку Play и посмотрите, что получилось. Откройте несколько карт и добейтесь нескольких совпадений; если после этого щелкнуть на кнопке Reset, игра начнется сначала, со скрытых карт и нулевого счета. Именно то, что и требовалось!

Как следует из названия метода `LoadScene()`, он может загружать различные сцены. Но что происходит в момент загрузки и почему игра при этом перезагружается? Все содержимое текущего уровня (все объекты сцены и присоединенные к ним сценарии) стирается из памяти, и загружается новая сцена. В нашем случае это сцена, сохраненная перед началом игры, поэтому все удаляется из памяти и загружается с нуля.

СОВЕТ Объекты, которые вы не хотите удалять из памяти при перезагрузке уровня, можно пометить. В Unity есть метод `DontDestroyOnLoad()`, сохраняющий объекты в разных сценах; в следующих главах он будет фигурировать как часть архитектуры кода.

Мы успешно создали еще одну игру! Разумеется, о «готовности» можно говорить только в относительном смысле — всегда остается простор для добавления новых функциональных возможностей. Но мы реализовали все намеченное в начальном плане. Многие концепции двумерной графики применимы и к трехмерным играм, особенно связанные с проверкой состояния игры и с загрузкой уровней. Пришло время изучить новую тему и выполнить новый проект.

Заключение

- Двумерная графика в Unity отображается ортогональной камерой.
- Для отображения пиксел в пиксел размер камеры должен быть равен половине высоты экрана.
- Чтобы спрайты реагировали на щелчок мыши, им следует назначить двумерный коллайдер.
- Новые изображения спрайтов можно загружать программно.
- Текст для пользовательского интерфейса можно создавать в виде трехмерных текстовых объектов.
- Загрузка игровых уровней позволяет перезагрузить сцену.

6 Базовый двумерный платформер

- ✓ Непрерывное движение спрайтов.
- ✓ Анимация на базе листов спрайтов.
- ✓ Физические явления в двумерном пространстве (столкновения, гравитация).
- ✓ Управление камерой в сайд-скроллерах.

Продолжим знакомство с двумерной функциональностью Unity на примере еще одной игры. Фундаментальные концепции мы рассмотрели в предыдущей главе, теперь воспользуемся ими, чтобы построить более сложную игру, а именно: создадим основную функциональность двумерного *платформера*. Это распространенный тип двумерных игр, самым известным представителем которого является классическая игра *Super Mario Brothers*: персонаж, показываемый сбоку, бежит и прыгает по платформам на фоне перемещающейся сцены.

Результат, который мы хотим получить, показан на рис. 6.1.

Во время работы над этим проектом вы познакомитесь с такими концепциями, как перемещение игрока влево и вправо, воспроизведение анимации спрайтов и добавление возможности прыгать. Рассмотрим мы и вещи, характерные именно для платформеров, например односторонние и движущиеся платформы. Переход от шаблона к полноценной игре по большей части сводится к многократному повторению всех этих вещей.

Первым делом нам нужен новый двумерный проект. Мы уже создавали такие проекты в предыдущей главе. Выберите команду New в окне, которое появляется после запуска Unity, или команду New Project в меню File. В открывшемся окне выберите вариант 2D. Создайте для нового проекта две папки с именами Sprites и Scripts. В них мы будем складывать различные ресурсы. Еще можно настроить камеру, как мы это делали в предыдущей главе, но в данном случае достаточно уменьшить значение поля Size до 4. В этом проекте точной настройки не требуется, но для окончательной версии игры нужно скорректировать размер поля зрения камеры.

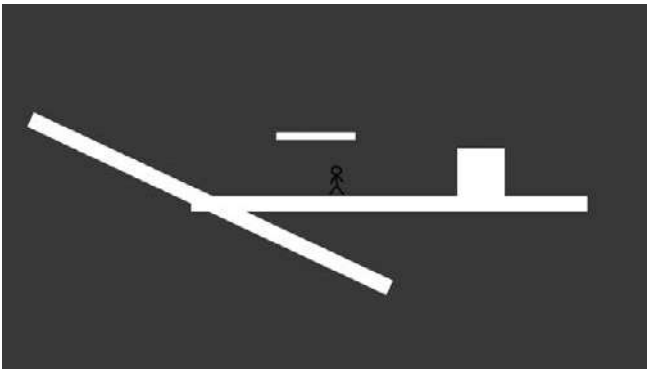


Рис. 6.1. Итоговый вид игры, которую мы создадим в этой главе

СОВЕТ Чтобы значок камеры в центре экрана не мешал работе, его нужно скрыть. В верхней части вкладки Scene откройте меню Gizmos и щелкните на значке Icon в строке Camera.

Сохраните пустую сцену, чтобы создать соответствующий набор ресурсов (в процессе работы не забывайте периодически щелкать на кнопке Save). Теперь давайте приступим к созданию графических ресурсов.

6.1. Создание графических ресурсов

Перед началом программирования функциональности нашего платформера в проект следует импортировать несколько изображений и вставить эти спрайты в сцену (надеюсь, вы помните, что в двумерных играх изображения называются *спрайтами*, а в трехмерных — *текстурами*). Так как мы просто знакомимся с принципом создания платформеров, в игре будет присутствовать управляемый игроком персонаж, перемещающийся по практически пустой сцене. Соответственно, достаточно пары спрайтов для платформ и для персонажа. Рассмотрим их по очереди, так как с этими простыми картинками связан ряд неочевидных моментов.

6.1.1. Стены и пол

Нам нужно пустое белое изображение. Пример проекта для этой главы содержит файл blank.png; скопируйте его оттуда в свой проект и поместите в папку Sprites. В параметрах импорта на панели Inspector убедитесь, что это действительно спрайт, а не текстура. Для двумерных проектов тип изображения выбирается автоматически, но проверить в любом случае имеет смысл.

По сути, мы будем строить геометрическую модель сцены, как это делалось в главе 4, просто на этот раз в режиме 2D. Поэтому основой послужат не сетки, а спрайты, но мы точно так же разместим в сцене пол и стены, формируя пространство, по которому будет перемещаться персонаж.

Чтобы получить пол, перетащите в сцену спрайт `blank`, как показано на рис. 6.2. Расположите его в точке с координатами `0.15`, `-1.27`, `0`, а в поля масштаба введите

значения `50`, `2`, `1`. Присвойте спрайту имя Floor. Перетащите в сцену еще один пустой спрайт, введите в поля масштаба значения `6`, `6`, `1` и поставьте на пол справа, указав координаты `2`, `-0.63`, `0`. Присвойте ему имя Block.

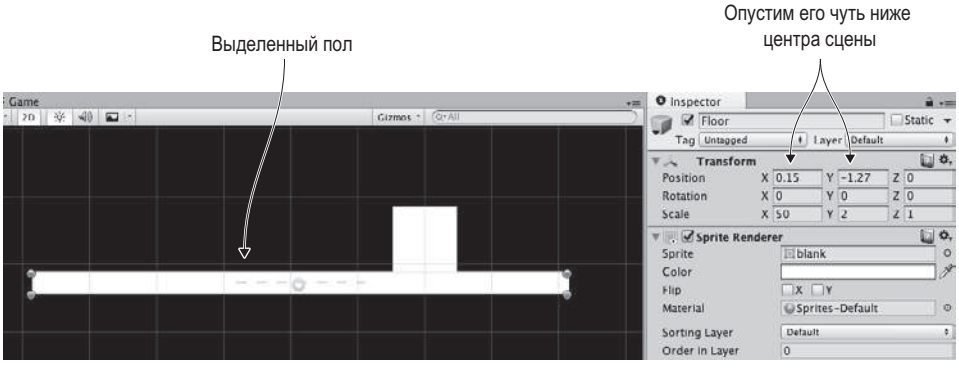


Рис. 6.2. Выбираем положение пола

Как видите, все очень просто. Пол и блок уже на месте, теперь нужно добавить в сцену персонаж.

6.1.2. Импорт листов спрайтов

Теперь нам требуется спрайт, изображающий игрока, поэтому скопируйте из папки с примером проекта файл stickman.png. На этот раз это не одно изображение, а объединенный набор спрайтов. Он показан на рис. 6.3 и представляет собой кадры двух вариантов анимации: бездействия и цикла ходьбы. Вдаваться в детали процесса анимации мы не будем, скажем только, что термины *бездействие* (idle) и *цикл* (cycle) повсеместно используются разработчиками игр. Бездействие относится к небольшим движениям стоящей на месте фигурки, а цикл представляет собой непрерывно повторяющуюся анимацию.

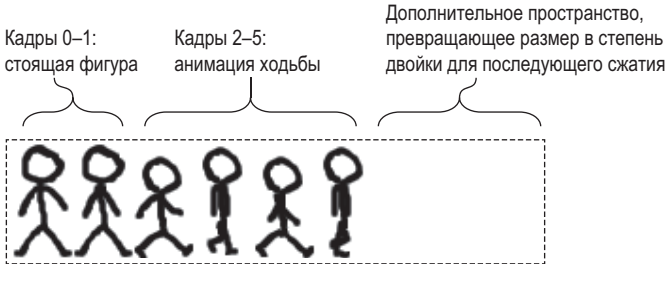


Рис. 6.3. Лист спрайтов Stickman — 6 кадров подряд

В предыдущей главе упоминалось, что изображение может представлять собой набор соединенных друг с другом спрайтов. В этом случае оно называется *листом спрайтов* (sprite sheet) и состоит из кадров анимации. В Unity листы спрайтов фигурируют на

вкладке Project как единый ресурс, но щелчок на расположенной сбоку стрелке позволяет увидеть все входящие в их состав изображения, как показано на рис. 6.4.

Рис. 6.4. Нарезка листа спрайтов на кадры

Перетащите изображение stickman.png в папку Sprites, чтобы выполнить его импорт, и внесите изменения в настройки импорта на панели Inspector. В меню Sprite Mode выберите вариант Multiple, а затем щелкните на кнопке Sprite Editor, чтобы открыть окно редактора. Откройте меню Slice в верхнем левом углу редактора и выберите в раскрывающемся списке Type вариант Grid By Cell Size (как показано на рис. 6.4). В поля Pixel Size введите значения `32` и `64` соответственно (это размер одного кадра в листе спрайтов) и щелкните на кнопке Slice. Лист распадется на отдельные кадры. Щелкните на кнопке Apply, чтобы сохранить сделанные изменения, и закройте редактор кадров.

Лист спрайтов превратился в набор кадров. Щелкните на стрелке, чтобы их увидеть. Перетащите один спрайт (например, самый первый) в сцену, поставьте на пол в центре и присвойте ему имя Player. В нашей игре появился персонаж!

6.2. Смещение персонажа вправо и влево

Теперь, когда в сцене появилась вся необходимая графика, можно приступить к программированию перемещений персонажа. Первым делом следует добавить персонажу пару компонентов. Как упоминалось в предыдущих главах, сымитировать объект, подчиняющийся законам физики, можно только при наличии у этого объекта компонента `Rigidbody`. Наш персонаж должен подчиняться законам физики (например, испытывать действие силы тяжести). Кроме того, потребуется компонент `Collider`, определяющий границы объекта при распознавании столкновений. Важно понимать разницу между этими компонентами: `Collider` определяет форму, на которую будут действовать законы физики, а `Rigidbody` указывает симулятору физики, на какие объекты он должен действовать.

ПРИМЕЧАНИЕ Эти компоненты существуют по отдельности (хотя они тесно связаны), потому что многие объекты, для которых не требуется имитация физической среды, *принимают участие* в столкновениях с объектами, *которые должны* подчиняться законам физики.

Есть еще один тонкий момент, о котором не следует забывать. В Unity для двумерных игр существует отдельная система имитации физической среды. Соответственно, в этой главе будут использоваться компоненты из раздела Physics 2D, а не из раздела Physics.

Выделите игрока и щелкните на кнопке Add Component на панели Inspector. Выберите в появившемся меню команду Physics 2D > Rigidbody 2D, как показано на рис. 6.5. Еще раз щелкните на кнопке Add Component и выберите уже команду Physics 2D > Box Collider 2D. Теперь слегка отредактируем компонент `Rigidbody`. Выберите в меню Collision Detection вариант Continuous. Затем установите флажок Freeze Rotation Z (обычно симулятор физической среды пытается поворачивать объекты в процессе перемещения, но персонаж нашей игры ведет себя немного по-другому), а в поле Gravity Scale введите значение 0 (позднее мы вернем исходное значение, но пока сила тяжести нам не требуется).

Осталось написать сценарий, который будет контролировать перемещения персонажа.

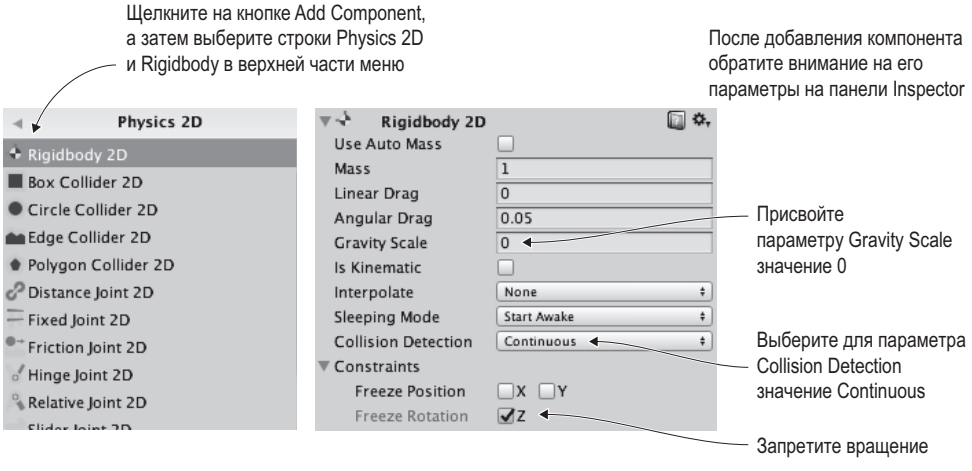


Рис. 6.5. Добавление и настройка компонента Rigidbody 2D

6.2.1. Элементы управления клавиатурой

Первым делом заставим персонажа перемещаться из стороны в сторону. Вертикальные перемещения также играют в платформерах важную роль, но ими мы займемся позднее. Создайте в папке Scripts сценарий с именем PlatformerPlayer и перетащите его на объект `Player`.

Добавьте в него код следующего листинга.

Листинг 6.1. Сценарий PlatformerPlayer для перемещений при помощи клавиш со стрелками

```
using UnityEngine;
using System.Collections;

public class PlatformerPlayer : MonoBehaviour {
  public float speed = 250.0f;
```

```
  private Rigidbody2D _body;

  void Start() {
    _body = GetComponent<Rigidbody2D>();
  }

  void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed * Time.deltaTime;
    Vector2 movement = new Vector2(deltaX, _body.velocity.y);
    _body.velocity = movement;
  }
}
```

Нужно, чтобы к объекту GameObject был прикреплен этот второй компонент.

Задаем только горизонтальное движение; сохраняем заданное вертикальное смещение.

Нажмите кнопку Play и посмотрите, как персонаж реагирует на кнопки со стрелками. Основное отличие от кода движения из предыдущих глав состоит в том, что новый код действует на компонент `Rigidbody2D`, а не на контроллер персонажа. Компонент `Character Controller` используется в трехмерном режиме, а для двумерных игр применяется компонент `Rigidbody`. Обратите внимание: движение применяется к переменной `velocity`, то есть к скорости этого компонента, а не к его положению.

СОВЕТ По умолчанию при нажатии кнопок со стрелками в Unity объект не сразу начинает двигаться с указанной скоростью, а постепенно разгоняется до нее. Но это слишком вялое движение для платформера. Для более быстрого управления увеличьте значения параметров Sensitivity и Gravity для горизонтального ввода до 6. Доступ к этим настройкам открывается командой Project Settings > Input из меню Edit. Они находятся в свитке Horizontal на панели Inspector.

Программирование движения по горизонтали практически завершено! Осталось научиться распознавать столкновения.

6.2.2. Столкновения со стенами

В настоящий момент персонаж спокойно проходит сквозь блок, ведь ни у пола, ни у блока нет коллайдера и сквозному движения ничто не мешает. Чтобы устранить эту недоработку, добавьте к объектам `Floor` и `Block` компонент `Box Collider 2D`. По очереди выделите каждый объект, щелкните на кнопке Add Component на панели Inspector и выберите в разделе Physics 2D строку Box Collider 2D.

Готово! Нажмите кнопку Play и убедитесь, что теперь персонаж останавливается перед блоком. Как и в случае перемещения игрока в главе 2, при попытках непосредственно редактировать положение перемещаемого объекта распознавание столкновений невозможно. Встроенный в Unity модуль распознавания столкновений работает, когда мы начинаем двигать добавленные объекту компоненты имитации физической среды. Другими словами, при изменениях переменной `Transform.position` распознавание столкновений игнорируется, поэтому в сценарии движения мы работаем с переменной `Rigidbody2D.velocity`.

К менее примитивным объектам добавить коллайдер чуть сложнее, но не намного. Даже если фигура не похожа на прямоугольник, можно взять прямоугольный коллайдер и окружить им форму, изображающую препятствие. Существуют и другие варианты

коллайдеров, в том числе и в виде определяемых пользователем наборов многоугольников. Принцип их применения к объектам сложной формы показан на рис. 6.6.

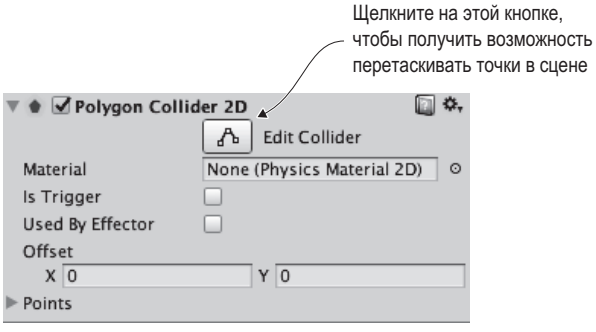


Рис. 6.6. Переход в режим редактирования формы полигонального коллайдера с помощью кнопки Edit Collider

С распознаванием столкновений мы разобрались, теперь давайте заставим персонажа двигаться в процессе перемещения.

6.3. Анимация спрайтов

После импорта изображение stickman.png было превращено в набор кадров для последующей анимации. Пришло время *запустить* эту анимацию, чтобы персонаж перестал скользить, а начал бегать в разные стороны.

6.3.1. Система анимации Mecanim

В главе 4 я уже упоминал, что система анимации в Unity называется *Mecanim*. Она позволяет визуально настроить для персонажа комплексную сеть анимационных клипов и обойтись минимумом кода при управлении этими клипами. Чаще всего ее применяют для трехмерных персонажей (эта тема будет детально рассмотрена в следующих главах), но работает она и с двумерными персонажами.

В основе системы анимации лежат два вида ресурсов: анима*ционные* клипы и анима*торные* контроллеры. Клипы представляют собой отдельные циклы анимации, в то время как контроллер — это сеть, определяющая, когда будет воспроизводиться каждый клип. Это диаграмма *конечного автомата* (state machine), а состояния на ней соответствуют различным анимационным клипам. Контроллер перемещается между состояниями, реагируя на окружающую среду, и в каждом состоянии воспроизводится свой клип.

Оба ресурса — анимационный клип и аниматорные контроллер — Unity создает автоматически в момент перетаскивания двумерной анимации в сцену. Разверните кадры нашего спрайта, как показано на рис. 6.7, выделите кадры 0–1, перетащите их в сцену и укажите в открывшемся окне имя stickman_idle.

На вкладке с ресурсами появился клип stickman_idle и контроллер `stickman_0`; переименуйте контроллер в stickman. Вы только что создали анимацию для стоящего на месте

персонажа! Кроме того, в сцене появился объект `stickman_0`, но он нам не потребуется, поэтому просто удалите его.

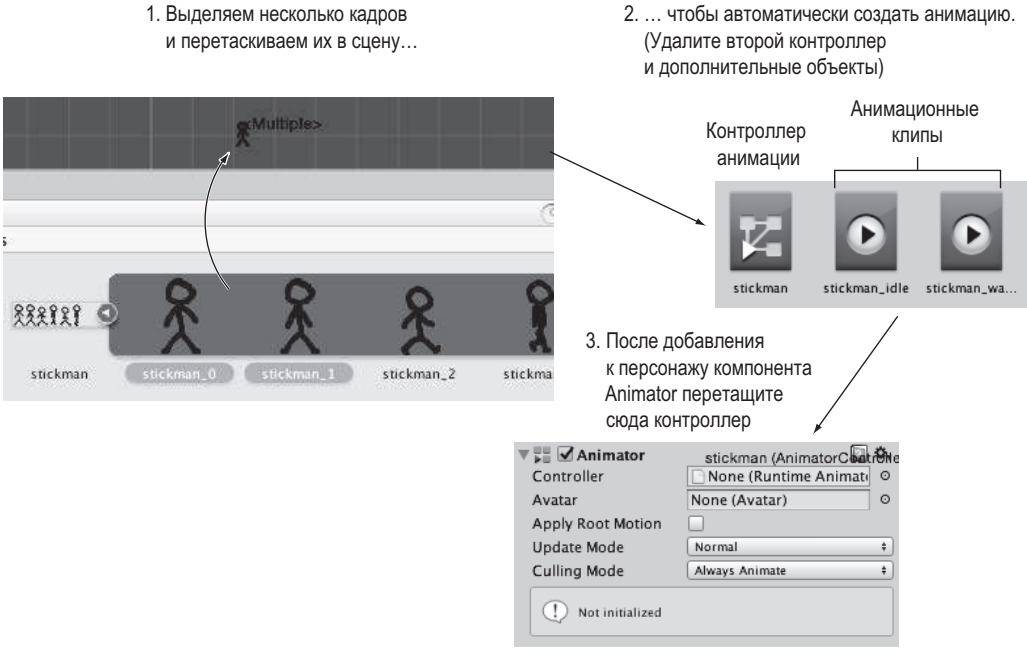


Рис. 6.7. Переход от кадров в составе листа спрайтов к компоненту Animator у персонажа

Повторим этот процесс, чтобы получить анимацию ходьбы. Выделите кадры 2–5, перетащите их в сцену и присвойте анимации имя stickman_walk. На этот раз удалите как появившийся в сцене объект `stickman_2`, так и контроллер на вкладке ресурсов; для управления обоими клипами достаточно одного контроллера, поэтому сохраните первый и удалите `stickman_2`.

Теперь нужно добавить к персонажу контроллер. Выделите в сцене объект `Player` и добавьте компонент `Animator` из раздела Miscellaneous. Как показано на рис. 6.7, опустите контроллер `stickman` на поле для контроллера на панели Inspector. Убедитесь, что объект `Player` все еще выделен, и выберите команду Animator в меню Window, чтобы открыть одноименное окно, показанное на рис. 6.8.

Анимационные клипы отображаются в окне Animator в виде блоков, называемых *состояниями* (states). Работающий контроллер совершает переходы между состояниями. Нашему контроллеру уже сопоставлен клип с состоянием покоя, но нужно добавить еще и клип с состоянием ходьбы; перетащите клип stickman_walk со вкладки Assets в окно Animator.

По умолчанию клип со статичным состоянием воспроизводится слишком быстро. Поэтому выделите блок для этого состояния и на панели справа введите в поле Speed значение `0.2`. Теперь осталось только написать запускающий анимацию код.

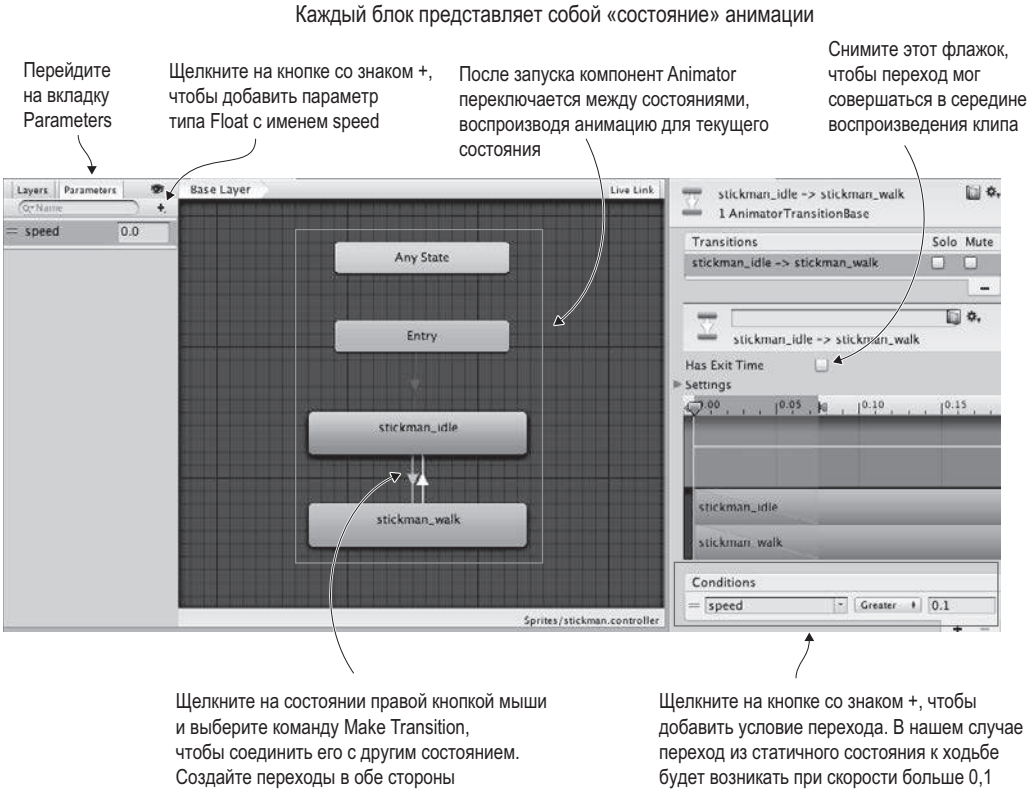


Рис. 6.8. Окно Animator, демонстрирующее анимационные состояния и переходы

6.3.2. Программный запуск анимационных клипов

Итак, мы настроили анимационные состояния в контроллере Animator, и теперь можем, переключаясь между ними, воспроизводить различные клипы. В предыдущем разделе упоминалось, что конечный автомат переходит из одного состояния в другое, реагируя на определенные условия. В Unity они называются *параметрами*. Рассмотрим процесс их добавления. Все необходимые элементы управления показаны на рис. 6.8. Нужно перейти на вкладку Parameters и щелкнуть на кнопке +, чтобы открыть меню с типами параметров. Добавьте параметр типа float и присвойте ему имя speed.

Теперь нужно организовать переход между состояниями анимации на базе этого параметра. Щелкните правой кнопкой мыши на блоке статичного состояния и выберите команду Make Transition; появится стрелка, исходящая из этого блока. Щелкните на блоке walk, чтобы завершить соединение. Переходы совершаются только в одну сторону, поэтому проделайте все процедуру еще раз, чтобы соединить блоки в обратном направлении.

Выделите стрелку перехода из статичного состояния (это можно сделать щелчком на самой стрелке), снимите флажок Has Exit Time и щелкните на расположенной ниже кнопке +, чтобы добавить условие (это тоже показано на рис. 6.8). Создайте условие

`speed Greater 0.2` (скорость больше, чем 0.2). Теперь выделите стрелку перехода из состояния ходьбы в состояние покоя. Снова снимите флажок Has Exit Time и добавьте условие `speed Less 0.2` (скорость меньше, чем 0.2).

Остается только добавить сценарий движения, который будет управлять аниматором.

Листинг 6.2. Запуск анимации во время перемещения

```
...
private Animator _anim;
...
void Start() {
  _body = GetComponent<Rigidbody2D>();
  _anim = GetComponent<Animator>();
}

void Update() {
  ...
  _anim.SetFloat("speed", Mathf.Abs(deltaX));
  if (!Mathf.Approximately(deltaX, 0)) {
    transform.localScale = new Vector3(Mathf.Sign(deltaX), 1, 1);
  }
}
...
```

Строка из ранее написанного кода, помогающая понять, куда следует вставлять новый код.

Скорость больше нуля даже при отрицательных значениях переменной velocity.

Числа типа float не всегда полностью совпадают, поэтому сравним их методом Approximately().

В процессе движения масштабируем положительную или отрицательную 1 для поворота направо или налево.

Вы не поверите, но так выглядит практически любой код управления анимационными клипами! Практически всю работу берет на себя система Mecanim, и вам остается добавить совсем немного. Запустите воспроизведение игры, подвигайтесь в разные стороны, и вы увидите, что спрайт персонажа ожил. Игра начинает выглядеть все интереснее. Пора переходить к следующему шагу!

6.4. Прыжки

Персонаж бегает по сцене взад и вперед, но не может перемещаться по вертикали. Но в платформерах персонажи умеют падать со ступенек и запрыгивать на платформы, поэтому давайте добавим эту способность и в нашу игру.

6.4.1. Падение под действием силы тяжести

На первый взгляд это кажется нелогичным, но прежде чем научить персонажа прыгать, следует добавить в сцену силу тяжести, которая будет мешать прыжкам. Возможно, вы помните, что чуть раньше мы присвоили переменной `Gravity Scale` связанного с персонажем компонента `Rigidbody` значение `0`, после чего он утратил способность падать под действием силы тяжести. Вернем этой переменной значение `1`: выделите объект Player и в разделе Rigidbody на панели Inspector введите `1` в поле Gravity Scale.

Теперь на персонажа действует сила тяжести, но пол не дает ему упасть (при условии, что ранее объекту `Floor` был назначен компонент `Box Collider`). Но если выйти за границу пола, персонаж упадет в пропасть. По умолчанию влияние силы тяжести не очень велико и имеет смысл его увеличить. Для имитации физической среды существуют глобальные настройки этого параметра, доступ к редактированию которых

осуществляется через меню Edit. Выберите в этом меню команду Project Settings > Physics 2D. Как показано на рис. 6.9, в верхней части списка элементов управления находится параметр Gravity Y; присвойте ему значение `-40`.

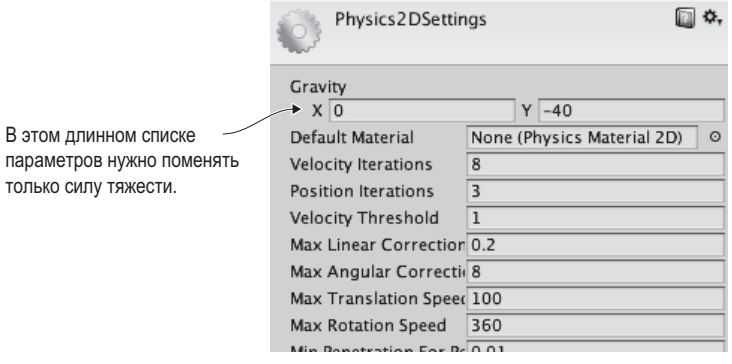


Рис. 6.9. Интенсивность силы тяжести в настройках имитации физической среды

Возможно, вы уже заметили небольшую проблему: падающий персонаж прилипает к краю пола. Чтобы обнаружить это поведение, выйдите за край платформы и сразу нажмите кнопку, заставляющую персонаж пойти в противоположном направлении. Происходящее выглядит неестественно. К счастью, это легко исправить средствами Unity — достаточно добавить к объектам `Block` и `Floor` компонент Physics 2D > `Platform Effector 2D`. Этот эффектор заставляет объекты сцены вести себя как платформы в платформенных играх. Редактируемые параметры показаны на рис. 6.10. Установите флажок Used By Effector в разделе с параметрами прямоугольного коллайдера и снимите флажок Use One Way в разделе с параметрами эффектора (этот флажок пригодится для других платформ, но сейчас он не нужен).

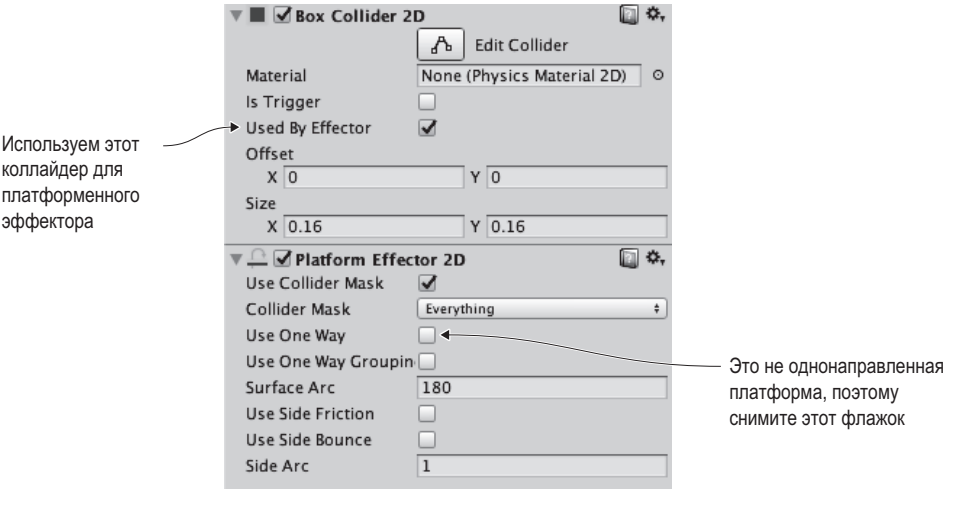


Рис. 6.10. Настройки коллайдера и эффектора на панели Inspector

Теперь движение персонажа вниз выглядит удовлетворительно, осталось разобраться с движением вверх.

6.4.2. Толчок вверх

Давайте научим персонаж прыгать. Для этого при нажатии кнопки Jump персонаж должно толкать вверх. Скорость перемещения по горизонтали редактируется в коде напрямую, а с движением по вертикали мы поступим по-другому, чтобы не мешать действию силы тяжести. Дело в том, что на объект могут влиять несколько сил одновременно, поэтому добавим к нему силу, направленную вверх. Вставьте в сценарий движения этот код.

Листинг 6.3. Прыжок при нажатии клавиши Пробел

```
...
public float jumpForce = 12.0f;
...
_body.velocity = movement;

if (Input.GetKeyDown(KeyCode.Space)) {
  _body.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
}
...
```

Строка из ранее написанного кода, помогающая понять, куда следует вставлять новый код.

Сила добавляется только при нажатии клавиши Пробел.

Обратите внимание на команду `AddForce()`. Восходящая сила добавляется к компоненту `Rigidbody` в *импульсном* режиме; импульс — это внезапный толчок в отличие от непрерывно действующей силы. В коде такой толчок возникает после нажатия клавиши Пробел. При этом на персонаж продолжает действовать сила тяжести, благодаря чему после прыжка он описывает красивую дугу.

Но, возможно, вы уже обратили внимание на один недостаток, устранением которого мы сейчас и займемся.

6.4.3. Распознавание поверхности

У элемента управления прыжком есть один маленький недостаток: возможность прыгать в полете (когда персонаж уже подпрыгнул или когда он падает). Нажатие клавиши Пробел даже в этом случае даст импульс вверх, хотя этого делать нельзя. Элемент управления прыжком должен срабатывать только для стоящего на платформе персонажа. А значит, нужно научиться определять, стоит ли персонаж на поверхности.

Листинг 6.4. Распознавание поверхностей

```
...
private BoxCollider2D _box;
...
_box = GetComponent<BoxCollider2D>();
...
_body.velocity = movement;

Vector3 max = _box.bounds.max;
Vector3 min = _box.bounds.min;
```

Получаем этот компонент, чтобы использовать его как проверочную область для коллайдера персонажа.

```
Vector2 corner1 = new Vector2(max.x, min.y - .1f);
Vector2 corner2 = new Vector2(min.x, min.y - .2f);
Collider2D hit = Physics2D.OverlapArea(corner1, corner2);

bool grounded = false;
if (hit != null) {
  grounded = true;
}

if (grounded && Input.GetKeyDown(KeyCode.Space)) {
...
```

Ниже проверяем значение минимальной Y-координаты коллайдера.

Если под персонажем обнаружен коллайдер...

...добавляем в условие для прыжка переменную grounded.

Этот код лишает персонаж способности к прыжкам в воздухе. Он проверяет, есть ли под персонажем коллайдер, и учитывает результат проверки в условии для совершения прыжка. Первым делом определяются границы примитива столкновений нашего персонажа, а затем в области с такой же шириной непосредственно под ногами персонажа ищутся перекрывающиеся коллайдеры. Результат проверки сохраняется в переменную `grounded` и используется в условном операторе.

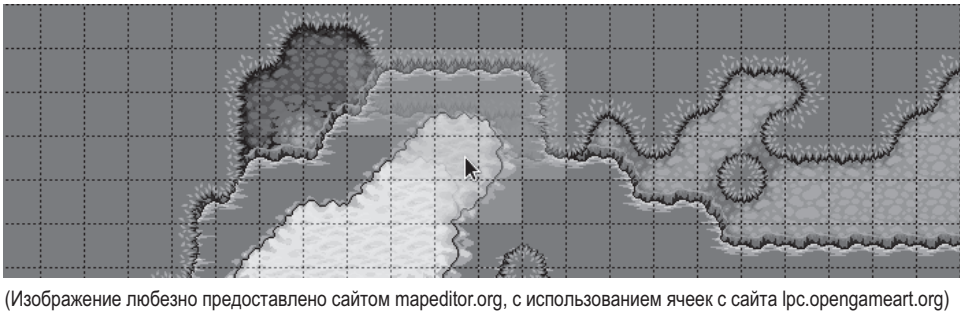
6.5. Дополнительные возможности для платформера

Итак, мы разобрались с нюансами реализации перемещений, ходьбы и прыжков персонажа. Дополним демоверсию нашей игры новыми функциональными возможностями окружающей среды.

ПРОЕКТИРОВАНИЕ УРОВНЕЙ ПРИ ПОМОЩИ ИНСТРУМЕНТА TILEMAP

В рамках проекта пол и платформы созданы из пустых белых прямоугольников. Для окончательного варианта игры требуется более проработанная графика, но вряд ли компьютер сможет обработать изображение размером с целый уровень. Чаще всего эту проблему решают с помощью инструмента tilemap. Он позволяет конструировать изображения по принципу координатной сетки. Вот как это выглядит.

Едва заметные линии сетки показывают границы ячеек игрового поля; на реальной карте сетка отсутствует



(Изображение любезно предоставлено сайтом mapeditor.org, с использованием ячеек с сайта lpc.opengameart.org)

Обратите внимание: карта целиком составлена из маленьких ячеек. Именно это позволяет накрывать изображением весь экран, не прибегая к изображениям слишком больших размеров.

Официальный инструмент tilemap добавлен в последние версии приложения Unity. Можно использовать и внешние библиотеки, например Tiled2Unity, систему, которая импортирует карты, созданные в популярном (и бесплатном) редакторе Tiled.

Дополнительную информацию (на английском языке) можно найти на сайтах: http://mng.bz/318f и www.seanba.com/tiled2unity.

6.5.1. Наклонные и односторонние платформы

Сейчас в нашей игре есть только обычный пол и блок, через который можно перепрыгнуть. Но существуют и другие варианты платформ. Воспользуемся ими, чтобы разнообразить сцену. Начнем с наклонного участка. Создайте копию объекта `Floor`, присвойте ей имя Slope, введите в поля преобразования вращения значения `0`, `0`, `-25` и сдвиньте ее влево, присвоив координатам преобразования перемещения значения `-3.47`, `-1.27, 0`.

Запустите воспроизведение сцены. В процессе движения персонаж корректно скользит вверх и вниз. Но из-за силы тяжести он начинает скользить вниз и в статичном состоянии. Для решения этой проблемы следует отключить имитацию силы тяжести в случаях, когда персонаж (a) стоит на платформе, (b) находится в состоянии покоя. К счастью, мы уже научили его распознавать поверхности и можем воспользоваться этим в новом фрагменте кода. Нужно добавить всего одну строку.

Листинг 6.5. Отключение имитации силы тяжести при нахождении на поверхности

```
...
_body.gravityScale = grounded && deltaX == 0 ? 0 : 1;
if (grounded && Input.GetKeyDown(KeyCode.Space)) {
...
```

Остановка при нахождении на поверхности и в статичном состоянии.

Строка из ранее написанного кода, помогающая понять, куда следует вставлять новый код.

После редактирования кода движения персонаж стал корректно вести себя на наклонных поверхностях. Теперь добавим так называемую одностороннюю платформу. Персонаж может не только стоять на ней, но и прыгать сквозь нее. При попытке прыжка сквозь обычную платформу персонаж ударится об нее головой.

Односторонние платформы часто встречаются в играх, поэтому в Unity есть средства их реализации. Если помните, ранее при добавлении компонента `Platform Effector` мы снимали флажок Use One Way. А вот теперь он нам потребуется! Еще раз создайте копию объекта `Floor`, присвойте ей имя Platform, введите в поля для масштаба значения `10`, `1`, `1` и поместите полученный объект над полом, указав для него координаты `-1.68`, `0.11, 0`. И обязательно установите флажок Use One Way в разделе Platform Effector 2D.

Персонаж сможет пролететь сквозь платформу при прыжке вверх, но движению вниз она воспрепятствует. Нужно устранить только один недостаток. Его иллюстрирует рис. 6.11. В Unity спрайт платформы может отобразиться поверх персонажа (чтобы было проще это увидеть, присвойте переменной `jumpForce` значение `7`. Можно отредактировать координату *Z* персонажа, как это было сделано в предыдущей главе, но на этот раз поступим по-другому. У компонента, отвечающего за визуализацию спрайтов,

существует порядок сортировки, определяющий, какой спрайт будет демонстрироваться сверху. Выделите объект Player и присвойте параметру Order in Layer в разделе Sprite Renderer значение 1.

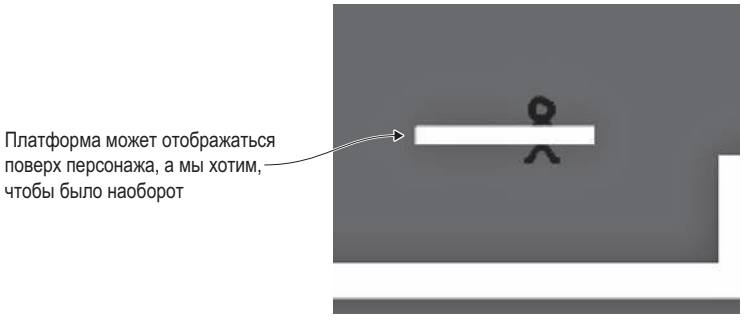


Рис. 6.11. Спрайт Platform, перекрывающий спрайт Player

Теперь в сцене есть наклонный фрагмент пола и однонаправленная платформа. Но существует еще один необычный вариант, который реализуется немного сложнее уже знакомых вам платформ. Давайте посмотрим, как это делается.

6.5.2. Движущиеся платформы

Третий вариант необычного пола, распространенный в платформенных играх, — это движущиеся платформы. Чтобы добавить в нашу игру такую платформу, потребуется, во-первых, написать новый сценарий, контролирующий ее перемещения, во-вторых, отредактировать сценарий движения персонажа. В новом сценарии платформа будет перемещаться между двумя точками: начальной и конечной. Создайте сценарий C# с именем `MovingPlatform` и добавьте в него следующий код.

Листинг 6.6. Сценарий MovingPlatform, заставляющий пол двигаться взад и вперед

```
using UnityEngine;
using System.Collections;

public class MovingPlatform : MonoBehaviour {
  public Vector3 finishPos = Vector3.zero;
  public float speed = 0.5f;

  private Vector3 _startPos;
  private float _trackPercent = 0;
  private int _direction = 1;

  void Start() {
    _startPos = transform.position;
  }

  void Update() {
    _trackPercent += _direction * speed * Time.deltaTime;
    float x = (finishPos.x - _startPos.x) * _trackPercent + _startPos.x;
```

Целевое положение.

Насколько далеко наше положение от старта до финиша.

Направление движения в данный момент.

Положение в сцене — это точка, от которой начинается движение.

```
    float y = (finishPos.y - _startPos.y) * _trackPercent + _startPos.y;
    transform.position = new Vector3(x, y, _startPos.z);

    if ((_direction == 1 && _trackPercent > .9f) ||
    (_direction == -1 && _trackPercent < .1f)) {
      _direction *= -1;
    }
  }
}
```

Меняем направление как в начале, так и в конце.

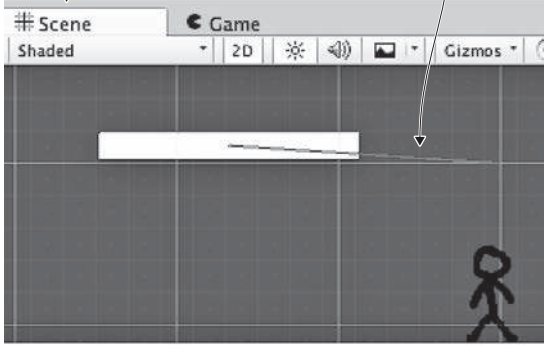
DRAWING CUSTOM GIZMOS

Большая часть кода, который пишется в процессе чтения книги, программирует происходящее в играх. Но сценарии Unity позволяют влиять и на *встроенный редактор*. Многие не знают, что в Unity можно добавлять дополнительные меню и окна. Более того, с их помощью создаются вспомогательные объекты на вкладке Scene, которые называются *габаритными контейнерами* (gizmos).

С габаритными контейнерами вы уже сталкивались. Помните зеленые параллелепипеды, обозначающие коллайдеры? Они встроены в Unity, но никто не мешает вам написать сценарий для собственного габаритного контейнера. К примеру, создадим на вкладке Scene линию, соответствующую траектории перемещения платформы. Она показана на рисунке.

Габаритные контейнеры отображаются только на вкладке Scene, облегчая процесс редактирования. На вкладке Game их не видно

Это пользовательский габаритный контейнер, показывающий траекторию движения платформы



Код, рисующий такую линию, очень прост. В верхнюю часть кода, влияющего на интерфейс Unity, нужно добавить строку `using UnityEditor;` (потому что большинство функций редактора находится именно в этом пространстве имен), но в данном случае этого не требуется. Просто добавьте этот метод в сценарий `MovingPlatform`:

```
...
void OnDrawGizmos() {
  Gizmos.color = Color.red;
  Gizmos.DrawLine(transform.position, finishPos);
}
...
```

Про этот код важно понимать следующее. Во-первых, все действие совершается внутри метода `OnDrawGizmos()`. Это системное имя, как и имена методов `Start` и `Update`. Во-вторых, мы добавляем в этот метод две строки кода: первая задает цвет линии, а вторая заставляет Unity нарисовать линию от места, где находится платформа, до целевой точки.

Перетащите этот сценарий на платформу. Запустите воспроизведение сцены, и вы увидите, что теперь платформа двигается из стороны в сторону! Осталось отредактировать сценарий, управляющий перемещениями персонажа, чтобы связать его с движущейся платформой. Внесите следующие изменения.

Листинг 6.7. Обработка движущейся платформы в сценарии PlatformerPlayer.cs

```
...
_body.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
}

MovingPlatform platform = null;
if (hit != null) {
  platform = hit.GetComponent<MovingPlatform>();
}
if (platform != null) {
  transform.parent = platform.transform;
} else {
  transform.parent = null;
}

_anim.SetFloat("speed", Mathf.Abs(deltaX));
...
```

Проверяем, может ли двигаться платформа, находящаяся над персонажем.

Выполняем связывание с платформой или очищаем переменную transform.parent.

Строка из ранее написанного кода, помогающая понять, куда следует вставлять новый код.

Теперь запрыгнувший на платформу персонаж будет перемещаться вместе с ней. По сути, все свелось к превращению персонажа в дочерний по отношению к платформе объект. Надеюсь, вы помните, что после формирования иерархической связи дочерний объект начинает перемещаться вслед за родительским. Метод `GetComponent()` в листинге 6.7 проверяет, является ли распознанная поверхность движущейся платформой. В случае положительного результата проверки она становится предком по отношению к персонажу; в противном случае персонаж отсоединяется от любого предка.

Но возникает одна проблема. Персонаж наследует от платформы и преобразование масштабирования, что влияет на его размер. Ситуацию позволяет исправить обратное преобразование (уменьшение масштаба персонажа, компенсирующее спровоцированное платформой увеличение размера).

Листинг 6.8. Коррекция масштабов персонажа

```
...
  _anim.SetFloat("speed", Mathf.Abs(deltaX));

  Vector3 pScale = Vector3.one;
  if (platform != null) {
    pScale = platform.transform.localScale;
  }
  if (deltaX != 0) {
    transform.localScale = new Vector3(
```

При нахождении вне движущейся платформы масштаб по умолчанию равен 1.

```
    Mathf.Sign(deltaX) / pScale.x, 1/pScale.y, 1);
  }
}
...
```

Замещаем существующее масштабирование новым кодом.

Математически компенсация масштабирования выглядит очень просто: игроку присваивается 1, деленная на масштаб платформы. И когда чуть позже масштаб игрока умножается на масштаб платформы, в итоге получается 1. Единственный сложный момент в данном случае состоит в учете знака, определяющего направление движения; возможно, вы помните, что в зависимости от него персонаж поворачивался из стороны в сторону.

Итак, мы добавили в игру движущуюся платформу. Остался один завершающий штрих, и демонстрационная версия будет полностью готова...

6.5.3. Управление камерой

Последнее, что мы добавим в наш двумерный платформер, это движение камеры. Создайте сценарий с именем `FollowCam`, перетащите его на камеру и введите в него следующий код.

Листинг 6.9. Сценарий FollowCam, перемещающий камеру вместе с персонажем

```
using UnityEngine;
using System.Collections;

public class FollowCam : MonoBehaviour {
  public Transform target;

  void LateUpdate() {
    transform.position = new Vector3(
    target.position.x, target.position.y, transform.position.z);
  }
}
```

Сохраняем координату Z, меняя значения X и Y.

Перетащите объект `Player` на поле Target в разделе Follow Cam (Script) панели Inspector. Запустите воспроизведение сцены. Теперь камера перемещается таким образом, чтобы персонаж все время оставался в центре экрана. Обратите внимание, что в коде с камерой связывается положение целевого объекта, а мы превратили персонаж в целевой объект. Причем метод называется `LateUpdate`, а не просто `Update`; это еще одно системное имя, принятое в Unity. Метод `LateUpdate` тоже исполняет каждый кадр, но только после того, как со всеми кадрами поработает метод `Update`.

Камера, постоянно перемещающаяся вслед за персонажем, немного раздражает. В большинстве платформенных игр камера ведет себя достаточно деликатно, выделяя в процессе движения персонажа различные фрагменты уровня. Вообще говоря, управление камерой в платформерах на удивление сложная тема; воспользуйтесь поиском по ключевым словам «platform game camera» или «камера в платформерах» и посмотрите все результаты. Мы же ограничимся тем, что сделаем ее движение плавным и менее раздражающим; вот отредактированная версия кода.

Листинг 6.10. Сглаживание движения камеры

```
...
public float smoothTime = 0.2f;

private Vector3 _velocity = Vector3.zero;
...
void LateUpdate() {
  Vector3 targetPosition = new Vector3(
  target.position.x, target.position.y, transform.position.z);

  transform.position = Vector3.SmoothDamp(transform.position,
  targetPosition, ref _velocity, smoothTime);
}
...
```

Сохраняем координату Z, меняя значения X и Y.

Плавный переход от текущей к целевой позиции.

Основное дополнение сводится к вызову функции `SmoothDamp`; все остальные (например, добавление переменных `time` и `velocity`) только поддерживают эту функцию. В Unity эта функция обеспечивает плавный переход от одного значения к другому. В рассматриваемом случае переход совершается от текущего положения камеры к целевому.

Теперь камера плавно следует за персонажем. Вы запрограммировали движение персонажа, несколько видов платформ и элемент управления камерой. Работа над проектом окончена!

Заключение

- Листы спрайтов часто используются как основа для двумерной анимации.
- Персонажи игр ведут себя не так, как реальные объекты, поэтому для них нужно особым образом настраивать параметры физической среды.
- Для управления объектами с компонентом `Rigidbody` им назначаются различные силы или напрямую задается их скорость.
- Для конструирования уровней в двумерных играх часто применяется инструмент Tilemaps.
- Простой сценарий позволяет заставить камеру плавно следовать за персонажем.

7 Двумерный GUI для трехмерной игры

- ✓ Сравнение старой (до версии Unity 4.6) и новой систем GUI.
- ✓ Создание холста для интерфейса.
- ✓ Позиционирование элементов UI с помощью точек привязки.
- ✓ Добавление к UI интерактивных элементов (кнопок, ползунков и т. п.).
- ✓ Рассылка и прием уведомлений о UI-событиях.

В этой главе мы поработаем над двумерным интерфейсом для трехмерной игры. При создании демонстрационного ролика от первого лица внимание концентрировалось исключительно на виртуальной сцене. Но любая игра требует не только места, где происходит действие, но и средств отображения абстрактных взаимодействий и информации. Без этого невозможно представить себе ни двумерную, ни трехмерную игру, ни шутер от первого лица, ни головоломку. Соответственно, все техники, которые будут показаны в этой главе на примере трехмерной игры, применимы и к двумерным играм.

Эти средства отображения абстрактных взаимодействий называют пользовательским интерфейсом (UI) или, точнее, графическим интерфейсом пользователя (GUI). Хотя с технической точки зрения аббревиатура GUI относится к визуальной части интерфейса, например к тексту и кнопкам (рис. 7.1), а аббревиатура UI — к физическим средствам управления, таким как клавиатура или джойстик, люди, говоря про «пользовательский интерфейс», как правило, подразумевают и графическую часть.

Хотя UI требуется любому программному обеспечению, потому что иначе пользователь просто не сможет контролировать работу приложения, GUI в играх зачастую функционирует несколько по-другому. Например, GUI веб-сайта, по сути, представляет собой *сам сайт* (если говорить о визуальном представлении). В игре же текст и кнопки зачастую накладываются на игровое пространство с помощью так называемого проекционного дисплея.

Рис. 7.1. Графический интерфейс, который мы создадим для игры

ОПРЕДЕЛЕНИЕ *Проекционный дисплей* (Heads-Up Display, HUD) накладывает графику поверх игровой сцены. Его прообразом был индикатор на лобовом стекле, позволяющий пилотам видеть навигационную и специальную информацию, не отвлекаясь на приборную панель, который использовался изначально в военной авиации.

В этой главе мы добавим в игру проекционный дисплей, используя новейшие инструменты Unity. Как упоминалось в главе 5, создавать элементы пользовательского интерфейса в Unity можно разными способами. Пришло время познакомиться с новой UI-системой, которая впервые появилась в версии Unity 4.6. Будет упоминаться и старая система, на примере которой я продемонстрирую преимущества нововведений.

Инструменты создания UI в Unity будут рассматриваться на примере шутера от первого лица, которым мы занимались в главе 3. Перед нами стоят следующие задачи:

1. Спланировать интерфейс.
2. Расположить на экране UI-элементы.
3. Запрограммировать взаимодействия с UI-элементами.
4. Запрограммировать GUI-реакции на события в игре.
5. Запрограммировать реакции сцены на действия с GUI.

Скопируйте проект из главы 3 и откройте его в Unity. Все необходимые ресурсы, как обычно, находятся в доступном для скачивания примере проекта. Надеюсь, вы готовы к построению пользовательского интерфейса для нашей игры.

ПРИМЕЧАНИЕ По большому счету не имеет значения, какой именно проект послужит рабочей основой для материала данной главы. Ведь мы просто добавим графический интерфейс поверх существующего демонстрационного ролика. Все упражнения даются на примере шутера от первого лица, созданного в главе 3. Можно скачать готовую версию этого проекта или воспользоваться любым другим демонстрационным роликом игры.

7.1. Перед тем как писать код...

Чтобы приступить к работе над элементами HUD, важно понимать, как функционирует UI-система. В Unity существуют разные подходы к построению проекционного дисплея, и желательно выбрать наиболее подходящий для наших целей. После этого

остается в общих чертах спланировать UI и подготовить необходимые графические ресурсы.

7.1.1. IMGUI или усовершенствованный 2D-интерфейс?

Уже в первой версии Unity появилась система GUI непосредственного режима (Immediate Mode GUI, IMGUI).

ОПРЕДЕЛЕНИЕ Словосочетание *непосредственный режим* (immediate mode) означает явное выполнение команд рисования в каждом кадре, в отличие от *режима удержания* (retained mode), в котором все визуальные элементы задаются однократно, после чего система сама решает, что именно следует рисовать в каждом кадре.

Система IMGUI позволяет легко поместить на экран интерактивную кнопку. Код этой операции показан в листинге 7.1. Остается только присоединить сценарий из этого листинга к любому объекту сцены. Можно вспомнить еще один пример UI непосредственного режима. Это курсор прицеливания, который мы создавали в главе 3. В основе такой системы GUI лежит исключительно код, работа в редакторе Unity не требуется.

Листинг 7.1. Реализация кнопки в системе IMGUI

```
using UnityEngine;
using System.Collections;

public class BasicUI : MonoBehaviour {
   void OnGUI() {
      if (GUI.Button(new Rect(10, 10, 40, 20), "Test")) {
        Debug.Log("Test button");
      }
   }
}
```

Функция вызывается в каждом кадре после того, как все остальное уже визуализировано.

Параметры: положение по X, положение по Y, ширина, высота, текстовая подпись.

Основа листинга — метод `OnGUI()`. Все представители класса `MonoBehaviour` автоматически отвечают как на методы `Start()` и `Update()`, так и на метод `OnGUI()`. Он запускается в каждом кадре после визуализации трехмерной сцены, предоставляя место для команд рисования GUI. В рассматриваемом случае код рисует кнопку; обратите внимание, что команда рисования выполняется в каждом кадре (то есть в непосредственном режиме). Эта команда фигурирует внутри условного оператора, срабатывающего при щелчке на кнопке.

Так как GUI непосредственного режима позволяет с минимальными усилиями добавить на экран несколько кнопок, мы будем пользоваться им и в следующих главах. Но это практически единственный легко генерируемый элемент, поэтому в новейших версиях Unity появилась новая система интерфейса, основанная на компонуемой в редакторе двумерной графике. Ее настройка требует некоторых усилий, но, скорее всего, в готовых играх вы предпочтете пользоваться именно этой системой, дающей более проработанные элементы интерфейса.

Новая система UI функционирует в режиме удержания, поэтому вся графика задается один раз и рисуется в каждом кадре без постоянных повторных определений

компоновки. Размещение графических элементов UI при этом выполняется в редакторе Unity, что дает два преимущества перед UI непосредственного режима. Во-первых, вид будущего интерфейса можно оценить в процессе выкладывания его элементов, во-вторых, для такой системы можно использовать собственную графику.

Для работы с этой системой мы импортируем в редактор изображения и перетащим объекты в сцену. Теперь осталось определить будущий вид нашего пользовательского интерфейса.

7.1.2. Выбор компоновки

В большинстве игр проекционный дисплей состоит из нескольких многократно повторяющихся UI-элементов. Поэтому понять процесс создания UI несложно. Мы добавим в углы экрана индикатор счета и кнопку настроек, разместив их поверх основного изображения сцены, как показано на рис. 7.2. Кнопка будет вызывать всплывающее окно с текстовым полем и ползунком.

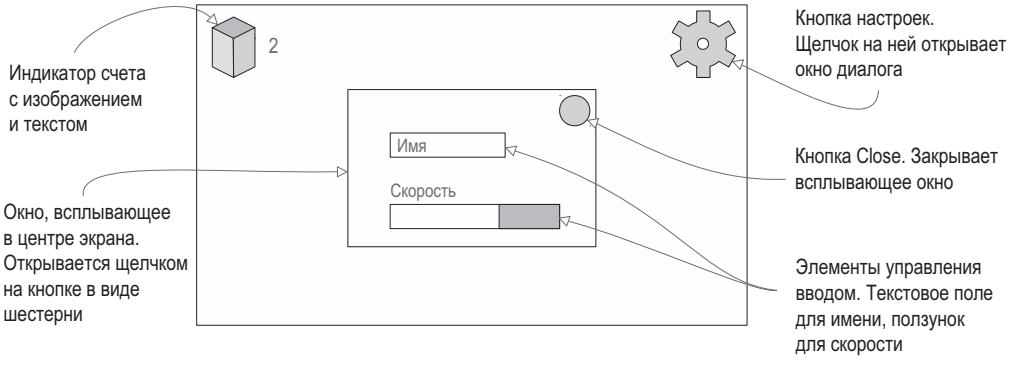


Рис. 7.2. План будущего GUI

В рассматриваемом примере элементы управления вводом служат для указания имени игрока и выбора скорости движения, но они могут управлять любыми относящимися к вашей игре настройками.

Как видите, макет крайне прост! Теперь нужно импортировать необходимые изображения.

7.1.3. Импорт изображений для элементов UI

Нашему пользовательскому интерфейсу требуется графика для отображения таких элементов, как, к примеру, кнопки. Воспользуемся двумерными изображениями, поэтому все сведется к уже знакомой вам по главе 5 процедуре:

1. Импорт изображений (если нужно, вручную определите их как спрайты).
2. Перетаскивание спрайтов в сцену.

Первым делом перетащите все изображения на вкладку Project, чтобы импортировать их в редактор, а затем на панели Inspector убедитесь, что у каждого из них параметр Texture Type имеет значение Sprite (2D And UI).

ВНИМАНИЕ По умолчанию параметр Texture Type в трехмерных проектах принимает значение Texture, а в двумерных — значение Sprite. Если требуются спрайты для трехмерного проекта, этот параметр нужно редактировать вручную.

Необходимые изображения (рис. 7.3) есть в скачанном примере проекта. Проверьте, что все импортированные ресурсы являются спрайтами; скорее всего, это не так и потребуется отредактировать у них параметр Texture Type.

Это изображения кнопок, индикатора счета и всплывающего окна, то есть элементов интерфейса, который мы собираемся создать. Теперь, когда они добавлены в проект, расположим графику на экране.

close

Это изображение станет кнопкой закрытия всплывающего окна

enemy

Это изображение станет индикатором счета в верхнем левом углу

gear

Это изображение станет кнопкой настроек в верхнем правом углу

popup

Это изображение после масштабирования послужит фоном всплывающего окна

Рис. 7.3. Изображения для текущего проекта

7.2. Настройка GUI

В качестве графических ресурсов мы взяли уже знакомую по главе 5 разновидность двумерных спрайтов, но на этот раз они будут использоваться немного по-другому. В Unity есть специальные инструменты, которые превращают изображения в отображаемые поверх трехмерной сцены элементы HUD. Существуют и особые приемы размещения UI-элементов, обусловленные тем, что на разных экранах фрагменты интерфейса зачастую отображаются по-разному.

7.2.1. Холст для интерфейса

Один из наиболее фундаментальных и при этом неочевидных аспектов функционирования системы UI сводится к необходимости связывать все изображения с объектом `Canvas`.

СОВЕТ Холст (canvas) — это разновидность объектов, которую Unity визуализирует как игровой пользовательский интерфейс.

Откройте меню GameObject с перечнем доступных объектов; в категории UI выберите вариант Canvas. В сцене появится холст (для наглядности присвойте ему имя HUD Canvas). Этот объект растянут на весь экран и имеет огромный в сравнении с трехмерной сценой размер, так как масштабирует один пиксел экрана в одну единицу измерения сцены.

ВНИМАНИЕ Вместе с холстом автоматически создается объект `EventSystem`. Он требуется для UI-взаимодействий, в остальных случаях его можно просто игнорировать.

Переключитесь в режим 2D, как показано на рис. 7.4, и дважды щелкните на имени холста на вкладке Hierarchy, чтобы увидеть этот объект целиком. Двумерный режим включается автоматически для всех 2D-проектов, но в случае 3D-проекта переход между UI и основной сценой осуществляется при помощи переключателя. Для возвращения к просмотру трехмерной сцены отключите 2D-режим и дважды щелкните на строчке Building, чтобы развернуть этот объект на весь экран.

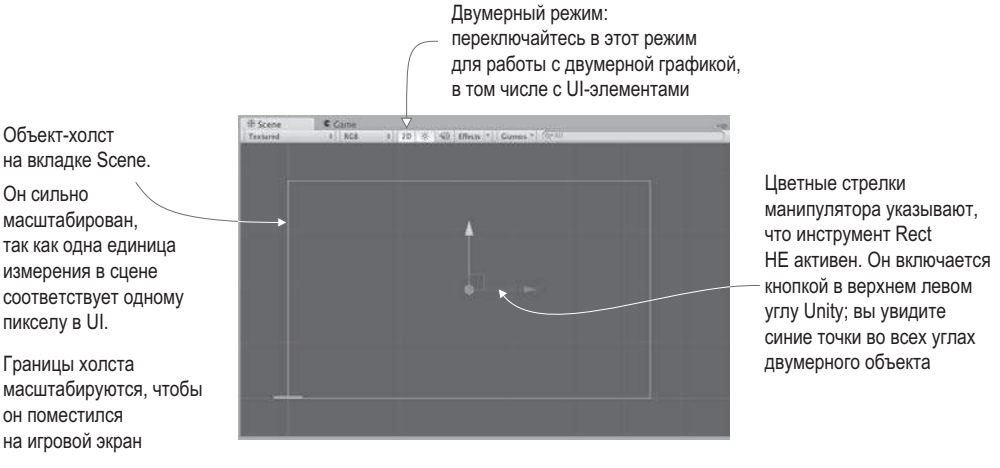


Рис. 7.4. Пустой холст на вкладке Scene

СОВЕТ Хочу еще раз привести совет из главы 4: в верхней части вкладки Scene располагаются кнопки, управляющие видимостью различных элементов, поэтому найдите кнопку Effects для отключения скайбокса.

У холста есть доступные для редактирования параметры. Во-первых, параметр Render Mode, которому следует оставить значение по умолчанию. Для него возможны следующие значения:

- Screen Space—Overlay — элементы UI визуализируются как наложенная поверх вида с камеры двумерная графика (установлено по умолчанию).
- Screen Space—Camera — элементы UI также визуализируются поверх вида с камеры, но могут вращаться, создавая эффекты перспективы.
- World Space — холст помещается в сцену, делая элементы UI частью трехмерной сцены.

Последние два режима применяются для создания специальных эффектов, но имеют несколько более сложную реализацию по сравнению с режимом по умолчанию.

Еще одна важная настройка — флажок Pixel Perfect. После его установки в процессе визуализации начинает корректироваться положение изображений с целью придать им четкий и контрастный вид (в отличие от размывания, возникающего при размещении между пикселов). Установите этот флажок, чтобы окончательно подготовить холст к размещению спрайтов.

7.2.2. Кнопки, изображения и текстовые подписи

Холст задает область, которая будет отображаться как пользовательский интерфейс, но туда следует добавить спрайты с изображениями отдельных элементов. В соответствии с макетом с рис. 7.2 в верхнем левом углу находится изображение блока/противника и рядом текст, отображающий набранные очки, а в верхнем правом углу — кнопка в виде шестерни. В разделе UI меню GameObject есть команды, позволяющие создать изображение (image), текст (text) или кнопку (button). Создайте по одному элементу каждого вида.

СОВЕТ Как и при моделировании текстовых объектов в главе 5, для ваших собственных проектов имеет смысл использовать шрифтовые ресурсы TextMesh Pro. Это внешняя система, призванная улучшить вид текста в Unity.

Для корректного отображения элементы интерфейса должны быть потомками холста. В Unity эта иерархическая связь возникает автоматически, но напомню, что подобные зависимости можно формировать и вручную, перетаскивая объекты на вкладке Hierarchy (рис. 7.5).

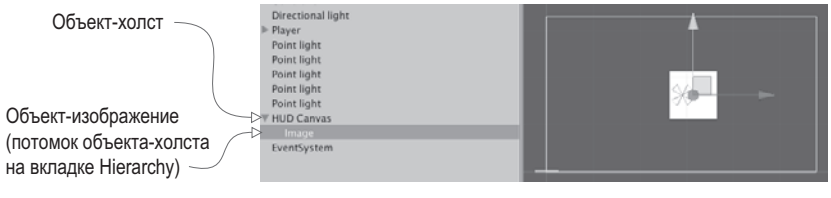


Рис. 7.5. Холст и связанное с ним изображение на вкладке Hierarchy

Между объектами на холсте также можно сформировать иерархические связи для удобства их размещения. Например, если перетащить текст на изображение, надпись будет перемещаться вместе с картинкой. Более того, у кнопки, созданной с параметрами по умолчанию, есть дочерний по отношению к ней текстовый объект; в данном случае мы не будем ничего на ней писать, поэтому просто удалите его.

Перетащите элементы интерфейса на предназначенные им места. В следующем разделе мы зададим их точное положение, пока же это не имеет значения. Указателем мыши перетащите объект-изображение в верхний левый угол холста, а кнопку — в верхний правый.

СОВЕТ В главе 5 упоминалось, что в двумерном режиме активируется инструмент Rect. Я описывал его как средство, совмещающее все три преобразования: перемещение, поворот и масштабирование. В трехмерном режиме эти операции выполняются отдельными инструментами, но при переходе к работе с двумерными объектами объединяются, так как у нас пропадает одно измерение. В 2D-режиме этот инструмент выбирается автоматически, кроме того, можно активировать его нажатием кнопки в верхнем левом углу Unity.

Пока оба элемента пусты. Если выделить объект UI, на панели Inspector, в верхней части свитка Image вы увидите поле Source Image. Перетащите на это поле спрайты (ни в коем случае не текстуры!) со вкладки Project, как показано на рис. 7.6. Назначьте

спрайт с изображением противника объекту `Image`, а спрайт с изображением шестерни объекту `Button`. Чтобы спрайты приобрели корректный размер, щелкните на кнопке Set Native Size.

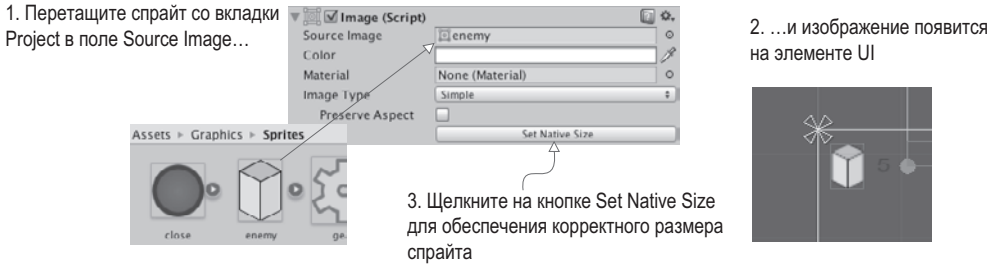


Рис. 7.6. Назначение двумерных спрайтов свойству Image элементов интерфейса

В результате картинки с изображениями противника и кнопки настроек приобретут нужный нам вид. Многочисленные параметры текстового объекта также отображаются на панели Inspector. Первым делом введите какое-нибудь число в большое поле Text; позднее это значение будет переопределено, пока же введенная информация выглядит в редакторе как отображение набранных очков. Увеличьте размер текста, присвоив параметру Font Size значение 24 и выбрав в раскрывающемся списке Font Style вариант Bold. При этом горизонтально эта текстовая подпись должна быть выровнена по левому краю, а вертикально — по центру, как показано на рис. 7.7. Остальным параметрам пока оставим значения по умолчанию.

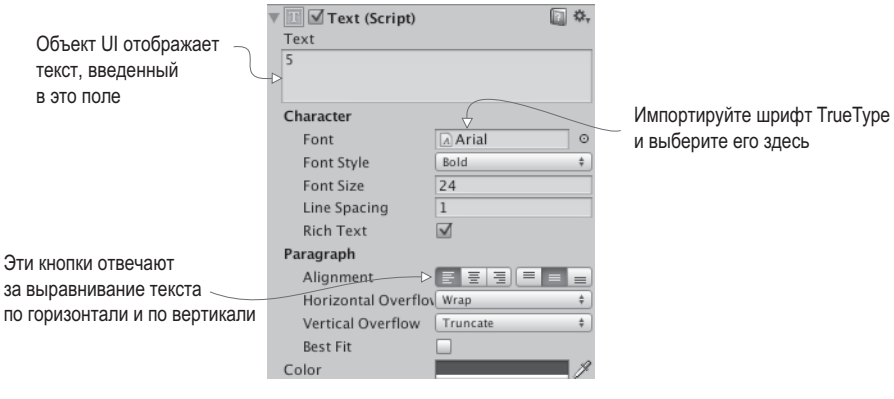


Рис. 7.7. Настройки текстового элемента UI

ПРИМЕЧАНИЕ Кроме содержимого поля Text и выравнивания чаще всего редактируется такое свойство, как шрифт. В Unity можно импортировать шрифт TrueType и выбрать его на панели Inspector.

Мы успешно назначили элементам интерфейса спрайты и указали параметры текста. Теперь нажмите кнопку Play, чтобы посмотреть, как выглядит проекционный дисплей поверх игровой сцены. Как показано на рис. 7.8, холст в редакторе Unity обозначает границы экрана, а элементы UI появляются на этом экране в заданных точках.

Холст, показанный в редакторе

Во время игры проекционный дисплей перекрывает основную игровую сцену

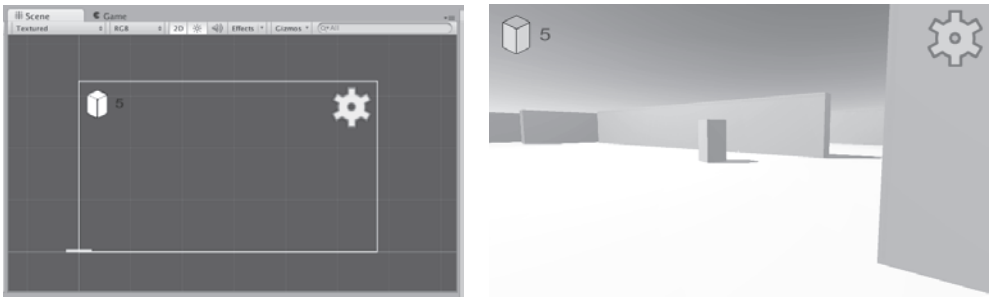


Рис. 7.8. Вид GUI в редакторе (слева) и в процессе игры (справа)

Видите, как здорово: вы научились отображать поверх трехмерной игровой сцены проекционный дисплей с элементами интерфейса! Осталось правильно расположить эти элементы относительно холста.

7.2.3. Управление положением элементов интерфейса

У всех объектов UI существует точка привязки, отображаемая в редакторе в виде крестика (рис. 7.9). Это гибкий способ позиционирования элементов интерфейса.

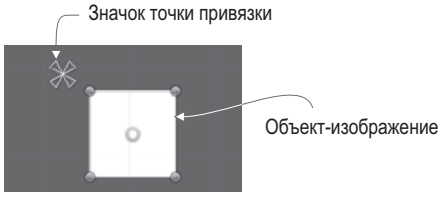


Рис. 7.9. Точка привязки изображения

ОПРЕДЕЛЕНИЕ *Привязкой* (anchor) объекта называется точка его присоединения к холсту или экрану. Именно относительно этой точки указывается положение объекта.

Положение задается в виде «50 пикселов по оси X». Но возникает вопрос: от какой точки отсчитывать 50 пикселов? На помощь приходит точка привязки. В то время как объект остается статичным относительно этой точки, сама она перемещается относительно холста. Точку привязки можно задать, например, как «центр экрана», и она будет оставаться в центре, когда экран меняет свой размер. Аналогично привязка к правой стороне экрана позволит объекту оставаться справа даже при изменении размеров (к примеру, при игре на другом мониторе).

Проще всего понять эти вещи на примере. Выделите объект `Image`. Настройки привязки на панели Inspector находятся в верхней части раздела Rect transform (рис. 7.10). По умолчанию привязка элементов интерфейса имеет значение Center, но нам в данном случае требуется значение Top Left; рис. 7.10 демонстрирует процесс редактирования данного параметра в окне Anchor Presets.

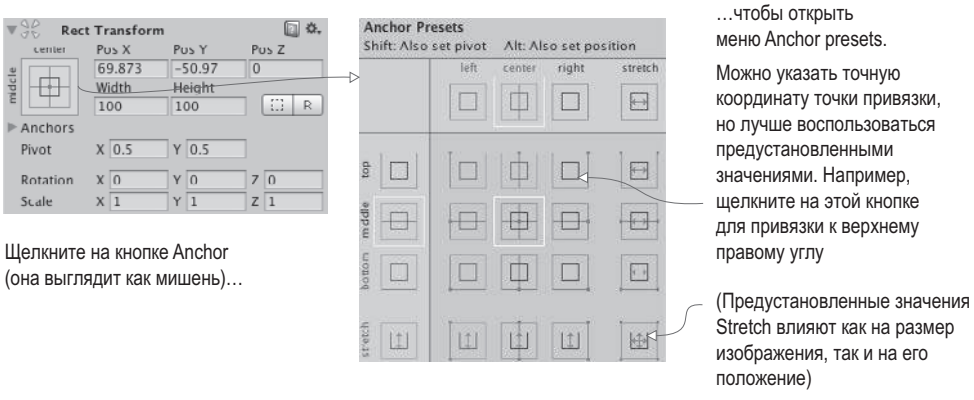


Рис. 7.10. Редактирование параметров привязки

Заодно поменяем привязку кнопки настроек. Щелчком на верхнем правом квадрате в меню Anchor Preset установите для нее значение Top Right. Попробуйте менять размер экрана, перетаскивая его боковые края влево и вправо. Благодаря привязкам при изменении размеров холста объекты UI остаются на своих местах. Как показано на рис. 7.11, элементы интерфейса перемещаются вместе с краями экрана.

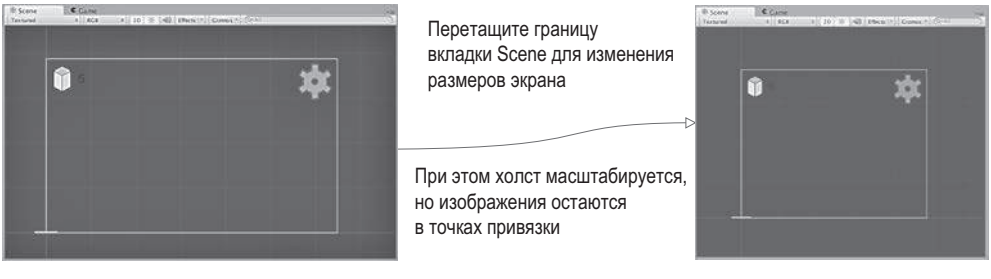


Рис. 7.11. При изменении размеров экрана привязки остаются на месте

СОВЕТ Точки привязки позволяют подстраиваться не только под изменение положения, но и под изменение размера. В этой главе данная функциональность не требуется, но каждый угол изображения можно связать с одним из углов экрана. На рис. 7.11 изображения сохраняют свой размер, но привязку можно отредактировать таким образом, что при изменении размеров экрана изображения будут растягиваться вместе с ним.

Итак, все визуальные элементы на своих местах, пришло время обеспечить их интерактивность.

7.3. Программирование интерактивного UI

Для взаимодействия с пользовательским интерфейсом нужен указатель мыши. Если помните, в этой игре его настройки редактируются в методе `Start()` сценария `RayShooter`. Изначально мы блокировали и скрывали его. Такое поведение прекрасно подходит для элементов управления в шутере от первого лица, но не позволяет работать

с элементами интерфейса. Удалите эти строчки из сценария `RayShooter.cs`, чтобы появилась возможность щелкать на элементах проекционного дисплея.

Кроме того, в сценарий `RayShooter.cs` нужно добавить строки, блокирующие в процессе взаимодействия с GUI возможность стрелять. Вот новая версия кода.

Листинг 7.2. Добавление взаимодействий с GUI в код сценария RayShooter.cs

```
using UnityEngine.EventSystems;
...
void Update() {
   if (Input.GetMouseButtonDown(0) &&
!EventSystem.current.IsPointerOverGameObject()) {
   Vector3 point = new Vector3(
      camera.pixelWidth/2, camera.pixelHeight/2, 0);
   ...
```

Подключаем библиотеку для UI-системы.

Курсивом выделен код, который уже был в сценарии и теперь приведен для справки.

Проверяем, что GUI не используется.

Теперь в процессе игры можно щелкать на кнопках, хотя пока это не дает результатов. Меняется только оттенок кнопки при наведении на нее указателя мыши и при щелчке. Это изменение цвета задано по умолчанию, и для каждой кнопки его можно отредактировать, но пока мы этого делать не будем. За скорость возвращения кнопки к обычному цвету отвечает параметр Fade Duration в свитке Button. Попробуйте уменьшить его до 0.01 и посмотрите, что получится.

СОВЕТ Иногда элементы взаимодействия с UI мешают игре. Помните автоматически появившийся вместе с холстом объект `EventSystem`? В числе прочего он контролирует элементы интерфейса, по умолчанию используя для взаимодействия с GUI кнопки со стрелками. Имеет смысл отключить работу с этими кнопками для компонента EventSystem: выделите его на вкладке Hierarchy и на панели Inspector снимите флажок Send Navigation Event.

Щелчок на кнопке пока ни к чему не приводит, так как она не связана с каким-либо кодом. Давайте исправим этот недостаток.

7.3.1. Программирование невидимого объекта UIController

Как правило, программирование взаимодействия с элементами интерфейса сводится к стандартной, общей для всех элементов последовательности:

1. В сцене создается UI-объект (в нашем случае это созданная в предыдущем разделе кнопка).
2. Пишется сценарий, который будет вызываться при обращении к этому элементу интерфейса.
3. Сценарий присоединяется к объекту в сцене.
4. Элементы интерфейса (например, кнопки) связываются с объектом, к которому присоединен этот сценарий.

Кнопка у нас уже есть, осталось создать контроллер, который будет с ней связываться. Создайте сценарий `UIController` и перетащите его на объект-контроллер в сцене.

Листинг 7.3. Сценарий UIController, предназначенный для программирования кнопок

```
using UnityEngine;
using UnityEngine.UI;   <-- Импортируем фреймворк для работы с кодом UI.
using System.Collections;

public class UIController : MonoBehaviour {
   [SerializeField] private Text scoreLabel;   <-- Объект сцены Reference Text для задания свойства text.

   void Update() {
      scoreLabel.text = Time.realtimeSinceStartup.ToString();
   }

   public void OnOpenSettings() {   <-- Метод, вызываемый кнопкой настроек.
      Debug.Log("open settings");
   }
}
```

СОВЕТ Казалось бы, зачем нужны два объекта, `SceneController` и `UIController`, ведь наша сцена настолько проста, что с управлением ее объектами и элементами интерфейса вполне мог бы справиться один контроллер? Но по мере усложнения сцены вы убедитесь, что лучше иметь отдельные модули управления, взаимодействующие друг с другом косвенным образом. Этот принцип применим не только к играм, но и к программному обеспечению в целом и в среде разработчиков ПО называется *разделением ответственности* (separation of concerns).

Перетащим объекты на ячейки компонентов, чтобы связать их друг с другом. Перетащите на поле Score Label объекта `UIController` текстовый объект, предназначенный для отображения счета. Текст данной подписи начнет определять код сценария `UIController`. В настоящее время отображается таймер, который мы добавили, чтобы протестировать, как все работает; чуть позже мы заменим его на набранные игроком очки.

Снабдим кнопку элементом `OnClick`, чтобы добавить ее к объекту-контроллеру. Выделите кнопку и найдите в нижней части панели Inspector поле OnClick; изначально оно пустое, но, как показано на рис. 7.12, можно щелкнуть на кнопке + и добавить туда элемент. Каждый элемент определяет одну функцию, вызываемую щелчком на кнопке; в листинге присутствует как ячейка для объекта, так и меню для вызываемой функции. Перетащите на ячейку объект-контроллер, выделите в меню строку UIController и выберите в дополнительном меню вариант OnOpenSettings().

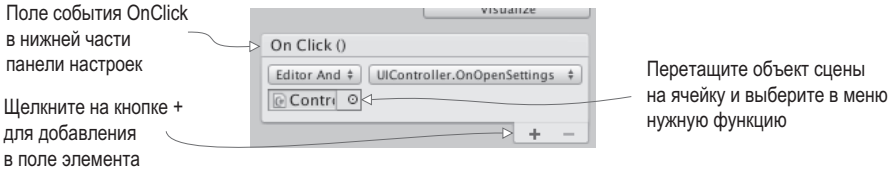


Рис. 7.12. Поле OnClick в нижней части панели с настройками кнопки

РЕАКЦИЯ НА ОСТАЛЬНЫЕ СОБЫТИЯ МЫШИ

Наша кнопка пока реагирует только на событие `OnClick`, в то время как элементы интерфейса могут отвечать на разные варианты взаимодействий. Для программирования взаимодействий, отличных от заданных по умолчанию, применяется компонент `EventTrigger`.

Добавьте к кнопке новый компонент и найдите раздел Event в меню этого компонента. Выберите вариант EventTrigger. Хотя событие `OnClick` кнопки отвечает только на полноценный щелчок (кнопка мыши нажимается, а затем отпускается), попробуем запрограммировать реакцию только на нажатие кнопки мыши. Последовательность действий будет такой же, как и для события `OnClick`, просто на этот раз мы укажем реакцию на другое событие. Первым делом добавьте в сценарий `UIController` еще один метод:

```
...
public void OnPointerDown() {
  Debug.Log("pointer down");
}
...
```

Щелкните на кнопке Add New Event Type, чтобы добавить к компоненту `EventTrigger` новый тип. Выберите вариант Pointer Down. Появится пустое поле для этого события, полностью аналогичное полю для события `OnClick`. Щелкните на кнопке +, чтобы добавить листинг события, и перетащите на этот элемент объект-контроллер, после чего выберите в меню вариант OnPointerDown(). Все готово!

Запустите игру и щелкните на кнопке для вывода на консоль отладочных сообщений. В данном случае мы выводим информацию, не имеющую к игре отношения, просто чтобы протестировать работу кнопки. По щелчку должно появляться всплывающее окно с настройками. Давайте его сформируем.

7.3.2. Всплывающее окно

Кнопка, входящая в состав нашего интерфейса, должна открывать окно диалога. Но окно сначала нужно создать. Его роль сыграет новый объект Image с присоединенными к нему элементами управления (кнопками и ползунками). Первым делом получим новое изображение, поэтому выберите в меню GameObject команду UI, а затем команду Image. Как и раньше, на панели Inspector вы найдете ячейку Source Image. Перетащите на нее спрайт с именем `popup`.

По умолчанию спрайт растягивается на всю поверхность объекта-изображения; именно так вели себя спрайты, предназначенные для отображения счета и кнопки настроек. Вы щелкали на кнопке Set Native Size, чтобы подогнать объект под размер картинки. Это стандартное поведение объектов-изображений, но у всплывающего окна другое назначение.

Как показано на рис. 7.13, у компонента `image` есть параметр Image Type. По умолчанию он имеет значение Simple, которое раньше нас вполне устраивало. Но для всплывающего окна присвойте параметру Image Type значение Sliced.

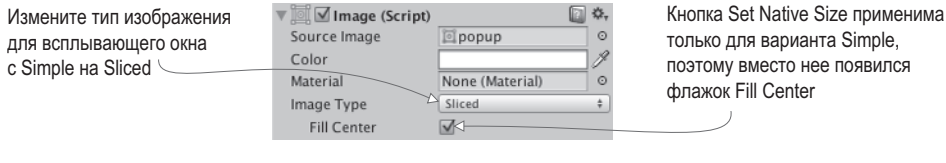


Рис. 7.13. Параметр Image Type компонента image

ОПРЕДЕЛЕНИЕ *Фрагментированное изображение* (sliced image) разбито на девять частей, которые масштабируются по-разному. Масштабирование краев отдельно от середины гарантирует сохранение четких и резких границ при любом изменении размеров. В других инструментах разработки имена таких изображений часто содержат цифру 9 (например, 9-slice, 9-patch, scale-9), что подчеркивает факт разделения на девять частей.

Переход к фрагментированному изображению может завершиться сообщением об ошибке, информирующим, что у изображения отсутствуют границы. Ведь спрайт popup пока не разделен на части. Выделите его на вкладке Project и на панели Inspector нажмите кнопку Sprite Editor, как показано на рис. 7.14. Откроется окно диалога Sprite Editor.

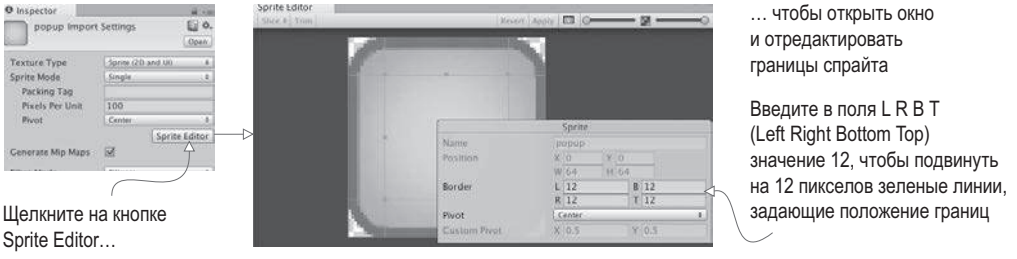


Рис. 7.14. Кнопка Sprite Editor на панели Inspector и открываемое ею окно

В окне Sprite Editor вы увидите зеленые линии, указывающие способ разбивки изображения. Изначально границ у спрайта нет (то есть величина всех границ равна 0). Увеличьте ширину границ со всех сторон, чтобы получить показанный на рис. 7.14 результат. Все четыре параметра (Left, Right, Bottom и Top) имеют значение 12 пикселов, поэтому пересекающиеся зеленые линии поделят изображение на девять частей. Закройте окно редактора и примените сделанные изменения.

Теперь, когда спрайт разделен на девять частей, параметр Image Type без проблем примет значение Sliced (а внизу появится флажок Fill Center). Перетащите находящийся в любом из углов изображения манипулятор синего цвета, чтобы выполнить масштабирование (если вы не видите манипуляторов, активируйте описанный в главе 5 инструмент Rect). Боковые фрагменты при этом сохранят свои размеры, в то время как центральная часть поменяет масштаб.

Это свойство боковых фрагментов сохраняет резкость границ изображения при любом изменении размеров, что идеально подходит для элементов интерфейса — окна могут иметь разные размеры, но выглядеть при этом должны одинаково. Укажите для ширины окна значение 250, а для высоты — 200, придав ему такой же вид, как на рис. 7.15 (заодно убедитесь, что оно находится в точке с координатами `0`, `0`, `0`).

СОВЕТ Способ наложения элементов интерфейса друг на друга определяется порядком их следования на вкладке Hierarchy. Расположите всплывающее окно поверх остальных элементов (разумеется, сохранив его связь с холстом). Теперь подвигайте окно по сцене и посмотрите, каким образом перекрываются изображения. Перетащите окно в самый низ иерархического списка дочерних элементов холста, чтобы оно отображалось поверх всего остального.

Всплывающее окно готово, можно писать для него код. Создайте сценарий `SettingsPopup` и перетащите его на наше окно.

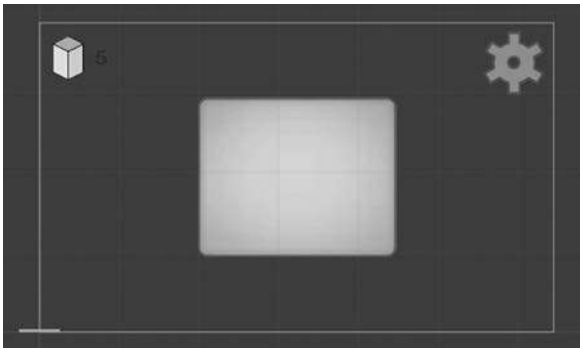


Рис. 7.15. Фрагментированное изображение отмасштабировано до размеров всплывающего окна

Листинг 7.4. Сценарий SettingsPopup для всплывающего окна

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour {
   public void Open() {
      gameObject.SetActive(true);  <-- Активируем этот объект, чтобы открыть окно.
   }
   public void Close() {
      gameObject.SetActive(false);  <-- Деактивируем объект, чтобы закрыть окно.
   }
}
```

Теперь откройте сценарий `UIController.cs` и добавьте в него содержимое следующего листинга.

Листинг 7.5. Возможность работы с окном для сценария UIController

```
...
[SerializeField] private SettingsPopup settingsPopup;
void Start() {
   settingsPopup.Close();  <-- Закрываем всплывающее окно в момент начала игры.
}
...
public void OnOpenSettings() {
   settingsPopup.Open();  <-- Заменяем отладочный текст методом всплывающего окна.
}
...
```

Этот код добавляет ячейку для всплывающего окна, поэтому свяжите это окно с объектом `UIController`. После этого оно начнет закрываться в начале игры и открываться при щелчке на кнопке настроек.

Способа закрыть его вручную пока не существует. Нужно добавить соответствующую кнопку. Последовательность действий будет практически такой же, как при создании предыдущей кнопки: выберите в меню GameObject команду UI> Button, поместите новую кнопку в верхний правый угол всплывающего окна, перетащите спрайт close на ячейку Source Image данного элемента интерфейса и щелкните на кнопке Set Native Size, чтобы

изображение приобрело нужный размер. Но сейчас, в отличие от предыдущего случая, нам требуется текстовая подпись, поэтому выделите дочерний компонент text и введите в текстовое поле слово Close, сделав его цвет белым. На вкладке Hierarchy перетащите эту кнопку на всплывающее окно, чтобы сформировать иерархическую связь. Как заключительный штрих присвойте параметру Fade Duration значение `0.01` и сделайте чуть темнее параметр Normal Color, присвоив ему значения `110`, `110`, `110`, `255`.

Чтобы кнопка закрывала окно, ей нужно сопоставить событие `OnClick`. Щелкните на кнопке + поля OnClick, перетащите всплывающее окно на ячейку объекта и выберите в списке функций вариант Close(). Запустите воспроизведение игры и убедитесь, что кнопка действительно закрывает всплывающее окно.

Итак, мы добавили к элементам проекционного дисплея всплывающее окно. Пока оно пустое, значит, нужно добавить к нему элементы управления. Именно этим мы и займемся в следующем разделе.

7.3.3. Задание значений с помощью ползунка и поля ввода

Процедура добавления элементов управления к всплывающему окну настроек состоит из двух этапов, почти как знакомая вам процедура создания кнопок. Вы генерируете присоединенные к холсту элементы интерфейса и связываете их со сценарием. Создадим ползунок, текстовое поле и текстовую подпись к ползунку. Выберите в меню GameObject команду UI > Text, чтобы получить текстовый объект, затем — команду UI > InputField, чтобы получить текстовое поле, а потом — UI > Slider для получения ползунка (рис. 7.16).

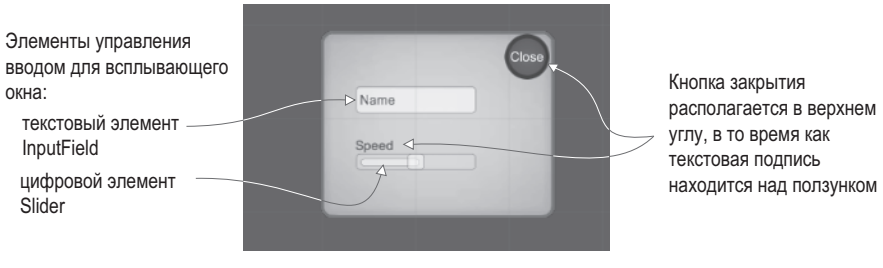


Рис. 7.16. Элементы управления вводом, добавленные к всплывающему окну

Сделайте все три объекта потомками всплывающего окна, перетащив их на вкладке Hierarchy, а затем расположите их в соответствии с рисунком, выровняв по центру всплывающего окна. Присвойте текстовому элементу значение Speed, формируя подпись к ползунку. Поле предназначено для ввода текста и по умолчанию, пока игрок ничего не напечатал, там отображается слово Text; замените его словом Name. Параметрам Content Type и Line Type можно оставить значения по умолчанию. Кроме того, можно ограничить вводимую информацию только буквами или только целыми числами, изменив параметр Content Type, а также разрешить ввод нескольких строк с помощью параметра Line Type.

ВНИМАНИЕ Пользоваться ползунком невозможно, если его перекрывает текстовая подпись. На вкладке Hierarchy текстовый объект должен располагаться над ползунком, чтобы обеспечить расположение ползунка поверх подписи.

ВНИМАНИЕ В рассматриваемом случае имеет смысл оставить полю ввода размер по умолчанию. Но если вы решите сделать его меньше, уменьшайте только значение Width. При значениях параметра Height меньше 30 поле становится слишком маленьким для отображения текста.

Параметры ползунка находятся в нижней части панели Inspector. Минимальное значение задает параметр Min Value, по умолчанию равный `0`; его мы менять не будем. Параметр же Max Value по умолчанию имеет значение `1`, но в нашем примере мы присвоим ему значение `2`. Параметрам Value и Whole Numbers оставим значения по умолчанию; первый устанавливает начальное положение ползунка, а второй ограничивает допустимые значения целыми числами (в данном случае такое ограничение не требуется).

Параметры всех объектов заданы, осталось написать управляющий их поведением код; добавьте в сценарий `SettingsPopup.cs` методы из следующего листинга.

Листинг 7.6. Методы для элементов управления вводом всплывающего окна

```
...
public void OnSubmitName(string name) {
   Debug.Log(name);
}
public void OnSpeedValue(float speed) {
   Debug.Log("Speed: " + speed);
}
...
```

Этот метод срабатывает в момент начала ввода данных в текстовое поле.

Этот метод срабатывает при изменении положения ползунка.

Замечательно! Наши элементы управления обзавелись методами. Среди настроек поля ввода появилось поле End Edit; перечисленные в этом поле события наступают после завершения пользовательского ввода. Добавьте к этому полю элемент, щелкнув на кнопке со значком +, перетащите всплывающее окно на ячейку для объекта и выберите в списке функций вариант `OnSubmitName()`.

ВНИМАНИЕ Функция `OnSubmitName()` присутствует как в верхнем списке Dynamic String, так и в нижнем Static Parameters. Но выбор варианта из нижнего списка дает возможность отправлять всего одну, заранее заданную строку. Нам же нужно, чтобы пересылалось любое введенное в поле значение, то есть *динамическая строка*, поэтому выберите функцию `OnSubmitName()` из нижнего списка Dynamic String.

Повторите эту последовательность действий для ползунка: найдите поле событий в нижней части панели Inspector (в данном случае оно называется OnValueChanged), щелкните на кнопке со значком + для добавления элемента, перетащите на ячейку объекта всплывающее окно и выберите в списке динамических функций вариант OnSpeedValue().

Теперь оба элемента управления вводом связаны с кодом сценария для всплывающего окна. Запустите воспроизведение игры и понаблюдайте, что происходит на консоли в процессе перемещения ползунка или при нажатии клавиши Enter после ввода имени в текстовое поле.

СОХРАНЕНИЕ НАСТРОЕК МЕЖДУ ИГРАМИ

В Unity есть несколько методов сохранения постоянных данных, простейшим из которых является `PlayerPrefs`. Это абстрагированный вариант, работающий на всех платформах и с разными файловыми системами (то есть не нужно беспокоиться о деталях реализации), позволяющий сохранять небольшие фрагменты информации. При большом объеме данных метод `PlayerPrefs` не помогает (в следующих главах для сохранения в процессе игры мы воспользуемся другими методами), но для сохранения настроек он подходит просто идеально.

Метод `PlayerPrefs` предоставляет простые команды чтения и задания именованных значений (его работа во многом напоминает хеш-таблицу или словарь). Например, для сохранения выбранной скорости достаточно добавить в метод `OnSpeedValue()` сценария `SettingsPopup` строку `PlayerPrefs.SetFloat("speed", speed);`. Метод сохранит десятичное значение скорости в переменную `speed`.

Аналогичным образом происходит инициализация ползунка с использованием сохраненного значения. Добавьте в сценарий `SettingsPopup` следующий код:

```
using UnityEngine.UI;
...
[SerializeField] private Slider speedSlider;
void Start() {
  speedSlider.value = PlayerPrefs.GetFloat("speed", 1);
}
...
```

Обратите внимание, что команде `get` предоставляется как значение переменной `speed`, так и значение по умолчанию на случай, если сохраненная скорость отсутствует.

Наши элементы управления генерируют отладочные сообщения, но пока никак не влияют на происходящее в игре. Программированию взаимодействий элементов проекционного дисплея с игрой посвящен последний раздел нашей главы.

7.4. Обновление игры в ответ на события

До текущего момента проекционный дисплей и игра существовали отдельно друг от друга, в то время как они должны взаимодействовать. Мы уже программировали взаимодействия между объектами с помощью ссылок в сценариях, но в данном случае этот вариант не подходит, так как приводит к формированию сильной связи между сценой и элементами проекционного дисплея. Нам же нужны практически независимые друг от друга фрагменты, дающие возможность свободно редактировать игру, не беспокоясь о сохранности проекционного дисплея.

Создадим систему широковещательной рассылки сообщений, которая будет информировать пользовательский интерфейс о том, что происходит в сцене. Принцип ее работы иллюстрирует рис. 7.17. Сценарии могут подписываться на слушание события, сообщение о котором рассылает конкретный код. Давайте посмотрим на практике, как все это работает.

СОВЕТ В языке C# существует встроенная система обработки событий. Почему мы ею не пользуемся? Дело в том, что эта система работает с нацеленными сообщениями, в то время как нам требуется широковещательная рассылка. В нацеленной системе код должен заранее знать источник сообщения, в то время как рассылка работает с сообщениями из произвольного источника.

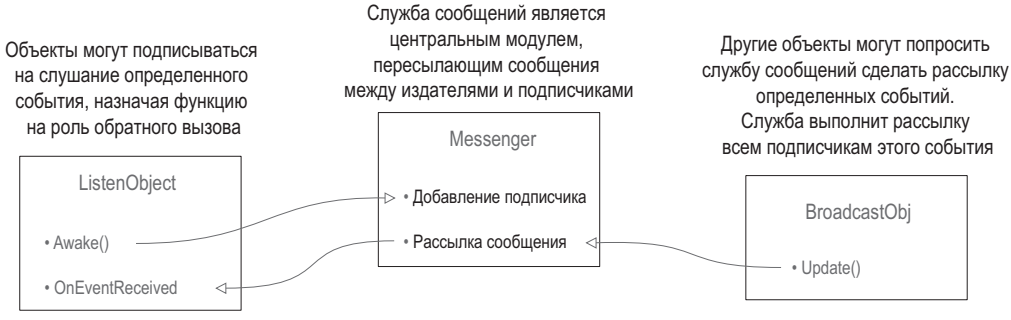


Рис. 7.17. Схема широковещательной рассылки сообщений

7.4.1. Интегрирование системы сообщений

Для оповещения пользовательского интерфейса о происходящем в сцене создадим систему широковещательной рассылки сообщений. В Unity нет встроенной функции, подходящей для выполнения такой задачи, но можно скачать нужный сценарий. Вики-сообщество Unify представляет собой репозиторий бесплатного кода от различных разработчиков. Их система для службы сообщений дает несвязанный способ взаимодействия с остальной частью программы посредством событий. Рассылающий сообщения код может ничего не знать о подписчиках, что позволяет легко менять или добавлять взаимодействующие объекты.

Создайте сценарий с именем Messenger и скопируйте в него одноименный код со страницы Unify: http://wiki.unity3d.com/index.php/CSharpMessenger_Extended. Затем понадобится еще один сценарий с именем GameEvent. Вот его код:

Листинг 7.7. Сценарий GameEvent, который будет использоваться вместе со сценарием Messenger

```
public static class GameEvent {
   public const string ENEMY_HIT = "ENEMY_HIT";
   public const string SPEED_CHANGED = "SPEED_CHANGED";
}
```

Этот сценарий задает константу для пары сообщений о событиях, что позволяет систематизировать сообщения, одновременно избавляя от необходимости вводить строку сообщения в разных местах.

Система оповещений о событиях готова, давайте ею воспользуемся. Начнем с сообщений от сцены к элементам проекционного дисплея, а затем запрограммируем взаимодействие в противоположном направлении.

7.4.2. Рассылка и слушание сообщений от сцены

До текущего момента вместо набранных игроком очков отображался таймер, который применялся для тестирования функциональности текстового дисплея. Но там должно отображаться количество пораженных игроком противников, поэтому давайте отредактируем код сценария `UIController`. Первым делом удалите метод

`Update()`, так как именно там находился тестовый код. В момент смерти противника генерируется событие. Следующий листинг заставляет сценарий `UIController` слушать это событие.

Листинг 7.8. Добавление подписчиков на событие в сценарий UIController

```
...
private int _score;

void Awake() {
   Messenger.AddListener(GameEvent.ENEMY_HIT, OnEnemyHit);
}
void OnDestroy() {
   Messenger.RemoveListener(GameEvent.ENEMY_HIT, OnEnemyHit);
}

void Start() {
   _score = 0;
   scoreLabel.text = _score.ToString();

   settingsPopup.Close();
}

private void OnEnemyHit() {
   _score += 1;
   scoreLabel.text = _score.ToString();
}
...
```

Объявляем, какой метод отвечает на событие ENEMY_HIT.

При разрушении объекта удаляем подписчика, чтобы избежать ошибок.

Присваиваем переменной score начальное значение 0.

Увеличиваем переменную score на 1 в ответ на данное событие.

Первым делом обратите внимание на методы `Awake()` и `OnDestroy()`. Как и методы `Start()` и `Update()`, все члены класса `MonoBehaviour` автоматически реагируют на активацию или удаление объекта. Подписчик добавляется в методе `Awake()` и удаляется в методе `OnDestroy()`. Будучи частью системы широковещательной рассылки сообщений, при получении сообщения он вызывает метод `OnEnemyHit()`, который увеличивает переменную `score` на 1 и выводит новое значение на текстовый дисплей.

Подписчик события задан в коде пользовательского интерфейса, поэтому каждое поражение противника должно сопровождаться рассылкой сообщения. Код реакции на смерть противника находится в сценарии `RayShooter.cs`, поэтому добавьте туда код отправки сообщения из следующего листинга.

Листинг 7.9. Рассылка сообщения о событии в сценарии RayShooter

```
...
if (target != null) {
   target.ReactToHit();
   Messenger.Broadcast(GameEvent.ENEMY_HIT);
} else {
...
```

К реакции на попадания добавлена рассылка сообщения.

Запустите игру и убедитесь, что теперь текстовый дисплей отображает количество поверженных противников. При каждом попадании счетчик должен увеличиваться на единицу. Мы успешно запрограммировали рассылку сообщений от трехмерной

игры к двумерному интерфейсу, и теперь требуется создать рассылку в обратном направлении.

7.4.3. Рассылка и слушание сообщений проекционного дисплея

В предыдущем разделе сообщение о событии посылалось сценой и принималось элементом проекционного дисплея. Сходным образом элементы пользовательского интерфейса могут посылать сообщения, которые будут слышать как игроки, так и противники. В результате на настройки игры начнут влиять параметры, которые игрок указывает во всплывающем окне.

Откройте сценарий `WanderingAI.cs` и добавьте туда следующий код.

Листинг 7.10. Добавление подписчика события в сценарий WanderingAI

```
...
public const float baseSpeed = 3.0f; ◄──┐
...                                     │ Базовая скорость, которая регулируется
void Awake() {                          │ положением ползунка.
   Messenger<float>.AddListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
void OnDestroy() {
   Messenger<float>.RemoveListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
...
private void OnSpeedChanged(float value) { ◄──┐
   speed = baseSpeed * value;                 │ Метод, объявленный в подписчике
}                                             │ для события SPEED_CHANGED.
...
```

Методы `Awake()` и `OnDestroy()` в данном случае тоже выполняют добавление и удаление подписчика, но на этот раз у них есть еще и значение. Оно указывает скорость перемещения противников.

СОВЕТ В предыдущем разделе фигурировало обобщенное событие, в то время как система рассылки позволяет передавать не только сообщения, но и значение. Для этого к подписчику достаточно добавить определение типа; обратите внимание на дополнение `<float>` в команде задания подписчика.

Внесем аналогичные изменения в сценарий `FPSInput.cs`, чтобы получить возможность влиять на скорость перемещения игрока. Содержимое следующего листинга отличается от листинга 7.10 только значением переменной `baseSpeed` у игрока.

Листинг 7.11. Добавление подписчика события в сценарий FPSInput

```
...
public const float baseSpeed = 6.0f; ◄── Это значение отличается от указанного в листинге 7.10.
...
void Awake() {
   Messenger<float>.AddListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
void OnDestroy() {
   Messenger<float>.RemoveListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
```

```
...
private void OnSpeedChanged(float value) {
   speed = baseSpeed * value;
}
...
```

Напоследок сделаем рассылку значений скорости из сценария `SettingsPopup` как ответ на изменение положения ползунка. Код этой рассылки приведен в следующем листинге.

Листинг 7.12. Рассылка сообщения сценарием SettingsPopup

```
public void OnSpeedValue(float speed) {
   Messenger<float>.Broadcast(GameEvent.SPEED_CHANGED, speed);
...
```

Значение, заданное положением ползунка, рассылается как событие <float>.

Теперь при изменении положения ползунка скорости противника и игрока будут меняться. Нажмите кнопку Play и убедитесь сами!

УПРАЖНЕНИЕ: ИЗМЕНЕНИЕ СКОРОСТИ ГЕНЕРИРУЕМЫХ ПРОТИВНИКОВ

Пока что корректируется только скорость уже присутствующего в сцене противника. Новые противники создаются без учета настроек скорости. Попробуйте самостоятельно задать скорость их перемещения. Подсказка: добавьте подписчика `SPEED_CHANGED` в сценарий `SceneController`, так как именно этот сценарий отвечает за появление новых противников.

Теперь вы умеете создавать графические интерфейсы с помощью инструментов Unity. Это пригодится при работе над остальными проектами, несмотря на то что мы начнем знакомиться с другими игровыми жанрами.

Заключение

- В Unity поддерживается как система GUI непосредственного режима, так и более новая система, основой которой являются двумерные спрайты.
- Применение двумерных спрайтов для создания GUI требует наличия в сцене такого объекта, как холст.
- Элементы пользовательского интерфейса можно привязывать к различным точкам настраиваемого холста.
- Свойство Active включает и выключает элементы пользовательского интерфейса.
- Несвязанная система передачи сообщений является замечательным способом обмена информацией между интерфейсом и сценой.

8 Игра от третьего лица: перемещения и анимация игрока

- ✓ Добавление теней в реальном времени.
- ✓ Облет камеры вокруг цели.
- ✓ Плавное изменение вращения с помощью алгоритма линейной интерполяции.
- ✓ Распознавание поверхности при прыжках, а также с учетом краев и склонов.
- ✓ Назначение анимации антропоморфному персонажу и управление ею.

В этой главе мы запрограммируем еще одну трехмерную игру, но на этот раз в другом жанре. В главе 2 был создан демонстрационный ролик для игры от первого лица. Давайте сделаем еще один демонстрационный ролик, но на этот раз сфокусируемся на перемещениях персонажа. Основное отличие состоит в положении камеры относительно игрока: в игре от первого лица игрок смотрит на сцену глазами персонажа, в то время как в игре от третьего лица камера находится *в стороне* от него. Скорее всего, такое представление знакомо вам по серии приключенческих игр *Legend of Zelda* или по более поздней серии игр *Uncharted* (виды сцены в игре от первого и от третьего лица сравниваются на рис. 8.3).

Эта глава посвящена одному из самых интересных в визуальном отношении проектов книги. Схема будущей сцены показана на рис. 8.1. Сравните ее с показанной на рис. 2.2 схемой игры от первого лица, с которой началось ваше знакомство с созданием игр в Unity.

Как видите, конструкция комнаты не изменилась, сценарии также применяются сходным способом. Коренным изменениям подверглись только вид игрока и положение камеры. Еще раз упомяну аспекты, превращающие происходящее в игру от третьего лица. Это расположение камеры в стороне от персонажа и ее нацеленность в сторону персонажа. Вместо примитивной капсулы теперь потребуется модель человека, так как игроки могут себя видеть.

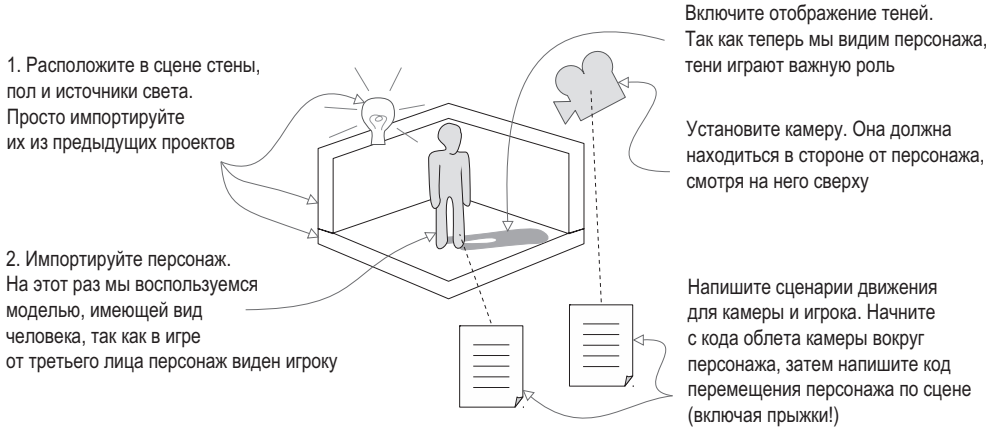


Рис. 8.1. Макет демонстрационного ролика игры от третьего лица

В главе 4 упоминались такие типы графических ресурсов, как трехмерные модели и анимация. Термин *трехмерная модель* — практически синоним сеточного объекта. Это статическая форма, определенная вершинами и полигонами (то есть сеточная геометрия). В случае антропоморфного персонажа этой геометрии придается форма головы, рук, ног и т. д. (рис. 8.2).

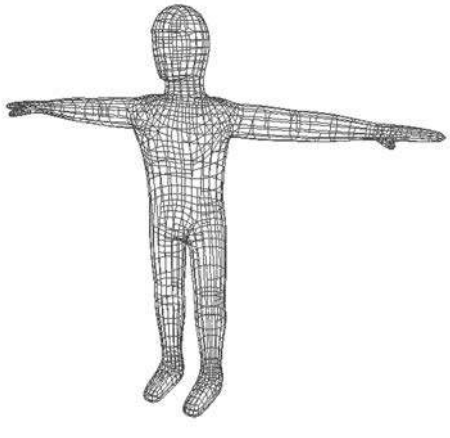


Рис. 8.2. Каркасное представление модели, с которой мы будем работать в этой главе

Как обычно, сфокусируемся на последнем пункте нашего плана: программировании объектов. Вот будущая последовательность действий:

1. Импорт модели персонажа в сцену.
2. Настройка элементов управления камеры и ее нацеливание на персонаж.
3. Написание сценария, позволяющего персонажу бегать по поверхности.

4. Добавление в сценарий движения возможности прыгать.
5. Воспроизведение анимации на модели на основе ее движений.

Скопируйте проект из главы 2, чтобы воспользоваться им как основой. Также можно открыть новый проект Unity (убедитесь, что вы создаете проект 3D, а не 2D, как в главе 5) и скопировать в него файл сцены из главы 2. Кроме того, понадобится содержимое папки scratch из материалов к данной главе. Именно в ней находится модель персонажа.

ПРИМЕЧАНИЕ Новый проект будет разрабатываться в интерьерах сцены из главы 2. Стены и источники света останутся теми же, замене подлежат только персонаж и все сценарии. Если вам нужны эти материалы, скачайте примеры файлов к данной главе.

Если вы начали с готового проекта из главы 2 (я имею в виду демонстрационный ролик, а не более позднюю версию), первым делом следует разорвать связь камеры с игроком. Это делается простым перетаскиванием на вкладке Hierarchy. Затем удалите объект `player`. Если бы камера оставалась его дочерним объектом, она при этом тоже исчезла бы. Нам же нужно избавиться только от капсулы, которая выполняла функцию игрока, сохранив камеру. Если же вы случайно удалили камеру, создайте новую, выбрав в меню GameObject команду Camera.

Заодно удалите все сценарии (для этого потребуется как избавить камеру от компонента `script`, так и удалить все файлы сценариев на вкладке Project). В сцене должны остаться только стены, пол и источники света.

8.1. Корректировка положения камеры

Перед тем как приступить к написанию кода, управляющего перемещениями персонажа, следует поместить персонаж в сцену и настроить камеру на отслеживание его положения. Импортируем в проект антропоморфную модель без лица и расположим камеру сверху таким образом, чтобы она смотрела на эту модель под углом. На рис. 8.3 сравнивается сцена в игре от первого лица с тем, что мы увидим в игре от третьего лица (во втором случае в сцене присутствуют крупные блоки, которые будут добавлены в проект в этой главе).

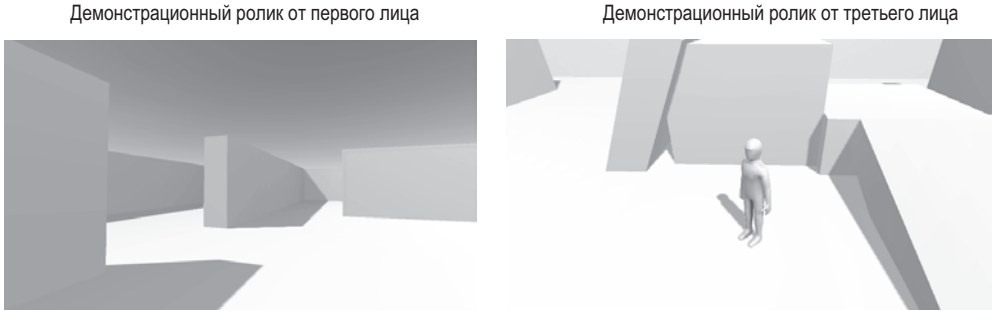


Рис. 8.3. Сравнительный вид сцены от первого и от третьего лица

Сцена полностью готова, осталось добавить в нее модель персонажа.

8.1.1. Импорт персонажа

В папке scratch, доступной для скачивания вместе с остальными материалами к этой главе, содержится как сама модель, так и текстура. В главе 4 мы говорили о том, что файл FBX — это модель, а файл TGA — это текстура. Импортируйте в проект файл FBX, перетащив его на вкладку Project или щелкнув на этой вкладке правой кнопкой мыши и выбрав в открывшемся меню команду Import New Asset. Теперь обратите внимание на панель Inspector, где нужно скорректировать параметры импорта модели. Редактированием анимации займемся чуть позже, а пока ограничимся парой параметров на вкладке Model. Первым делом присвойте параметру Scale Factor значение `10` (частично компенсируя слишком маленькое значение параметра File Scale), чтобы получить модель корректного размера.

Ниже располагается параметр Normals (рис. 8.4). Он контролирует вид света и теней на модели, используя для этого такое математическое понятие, как нормали.

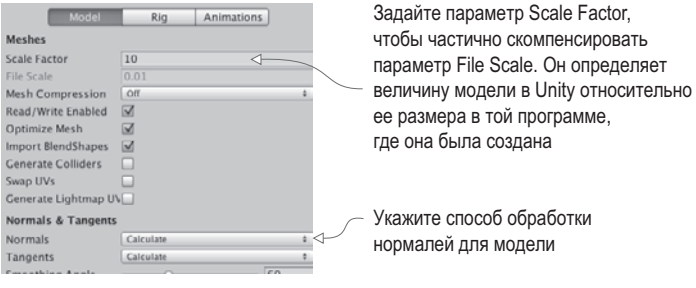


Рис. 8.4. Параметры импорта для модели персонажа

ОПРЕДЕЛЕНИЕ *Нормалями* (normals) называются перпендикулярные полигонам векторы направления, указывающие лицевую сторону полигонов. Именно это направление используется для вычисления освещенности.

По умолчанию параметр Normals имеет значение Import, в котором используются нормали, задаваемые геометрией импортированной сетки. Но нормали нашей модели определены некорректно, что приводит к странной реакции на источники света. Поэтому присвоим данному параметру значение Calculate, заставив Unity вычислять вектор для лицевой стороны каждого полигона.

Завершив редактирование этих двух параметров, щелкните на кнопке Apply на панели Inspector. Затем импортируйте в проект файл TGA и сделайте это изображение текстурой. Для этого выделите назначенный персонажу материал и перетащите изображение текстуры на ячейку Albedo на панели Inspector. Цвет модели практически не изменится (эта текстура по большей части имеет белый цвет), но на текстуре нарисованы тени, улучшающие внешний вид модели.

Перетащите модель персонажа со вкладки Project в сцену. Введите в поля Position значения `0`, `1.1`, `0`, чтобы персонаж оказался в центре помещения. Первый шаг к созданию игры от третьего лица сделан!

ПРИМЕЧАНИЕ Руки персонажа вытянуты в стороны, хотя естественнее было бы опустить их вниз. Это так называемая *T-образная поза*, по умолчанию придаваемая всем подлежащим анимации персонажам.

8.1.2. Добавление теней

Рассмотрим такой важный аспект, как отбрасываемая персонажем тень. Наличие теней считается само собой разумеющимся, но в виртуальном мире свои законы. К счастью, Unity в состоянии позаботиться о таких вещах, и источник света, по умолчанию присутствующий в любой новой сцене, приводит к формированию теней. Выделите на вкладке Hierarchy строчку Directional Light и обратите внимание на параметр Shadow Type на панели Inspector. Он, как показано на рис. 8.5, имеет значение Soft Shadows, при этом среди доступных значений есть и вариант No Shadows (без теней).

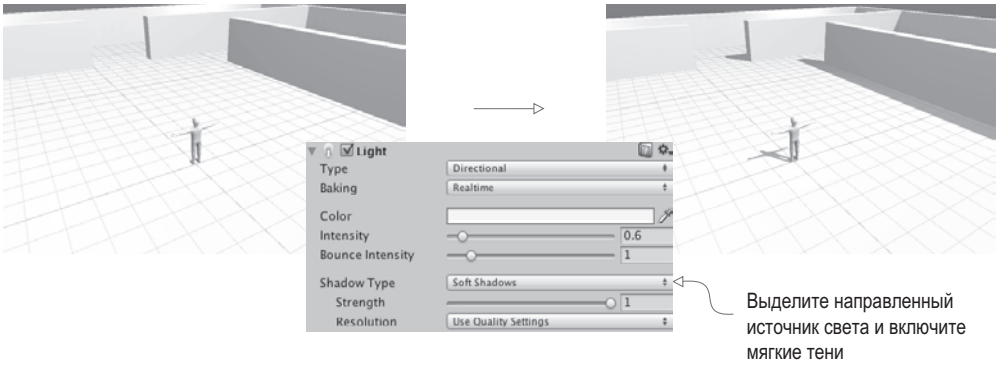


Рис. 8.5. Вид без теней от направленного источника света и с тенями

В рамках данного проекта настройка теней завершена, но, разумеется, в целом создание теней в играх представляет собой куда более сложный процесс. Расчет теней в сцене является крайне затратной по времени операцией, поэтому для получения нужного эффекта зачастую идут на хитрости и различными способами имитируют нужные детали. Тени от персонажа относятся к так называемым теням, *визуализируемым в реальном времени* (real-time shadow), так как они вычисляются в процессе игры и двигаются вместе с объектом. Можно создать идеальное реалистичное освещение, при котором все объекты будут отбрасывать тени и формировать тени на своих поверхностях в реальном времени, но обычно с целью ускорения расчетов накладывают ограничения как на вид теней, так и на количество источников света, приводящих к их формированию. Обратите внимание, что в нашей сцене тени формирует только направленный осветитель.

Также распространен способ создания теней на основе *карт освещения*.

ОПРЕДЕЛЕНИЕ *Картами освещения* (lightmaps) называются текстуры, которые назначаются геометрии игровых уровней и в которых «запечено» изображение теней.

ОПРЕДЕЛЕНИЕ Рисование теней на текстуре модели называется *запеканием теней* (baking the shadows).

Эти изображения генерируются заранее (а не во время игры), поэтому они могут быть крайне детализированными и реалистичными. Разумеется, они полностью статичны и используются с неподвижной геометрией, так как совершенно неприменимы к динамическим объектам, например персонажам. Карты освещения генерируются автоматически. Компьютер вычисляет, как имеющиеся в сцене источники света будут освещать уровень, оставляя в углах небольшое затемнение. Система визуализации карт освещения в Unity называется Enlighten. Используйте это название как ключевое поисковое слово в справочниках по Unity.

К счастью, нам не приходится делать выбор между визуализацией теней в реальном времени и картами освещения на уровне сцены в целом. У источников света есть свойство Culling Mask, позволяющее визуализировать в реальном времени тени только для определенных объектов, сымитировав детализированные тени от остальных объектов сцены с помощью карт освещения. Кроме того, хотя основной персонаж обычно в обязательном порядке отбрасывает тень, далеко не всегда нужно, чтобы на нем формировались тени от окружающих объектов. Поэтому для всех сеточных объектов можно включать функции отбрасывания и восприятия теней, как показано на рис. 8.6.

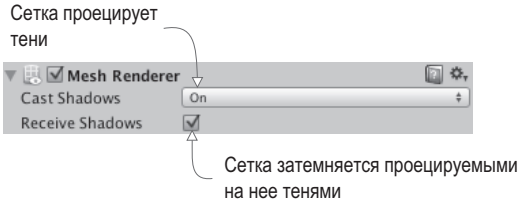


Рис. 8.6. Параметры Cast Shadows и Receive Shadows на панели Inspector

ОПРЕДЕЛЕНИЕ Термин *отбор* (culling) в общем случае означает исключение ненужных вещей. В статьях о компьютерной графике он появляется в разных контекстах. В нашем случае свойство Culling mask определяет набор объектов, которые не будут отбрасывать тени.

Итак, вы познакомились с основами процедуры добавления теней. Освещение сцен и формирование теней — это огромная тема (в книгах, посвященных редактированию уровней, им зачастую посвящается не одна глава), но мы ограничимся визуализацией теней в реальном времени от одного источника света. И займемся настройкой камеры.

8.1.3. Облет камеры вокруг персонажа

В демонстрационном ролике от первого лица мы связывали камеру с объектом-персонажем на вкладке Hierarchy, обеспечивая их совместное вращение. В игре же от третьего лица персонаж будет поворачиваться независимо от камеры. Поэтому связывать камеру с персонажем на вкладке Hierarchy мы на этот раз не будем. Вместо этого напишем код, который будет менять положение камеры вместе с положением персонажа, поворачивая ее независимо от последнего.

Первым делом выберем положение камеры относительно персонажа. Я ввел в поля position значения `0`, `3.5`, `-3.75`, расположив камеру чуть выше персонажа и за его спиной

(в полях rotation при этом должны быть значения `0`, `0`, `0`). После этого нужно создать сценарий `OrbitCamera` и добавить в него код следующего листинга. Присоедините сценарий к камере в виде нового компонента, а затем перетащите объект `player` на ячейку Target. Запустите воспроизведение сцены, чтобы посмотреть, как работает код камеры.

Листинг 8.1. Сценарий вращения нацеленной на объект камеры вокруг объекта

```
using UnityEngine;
using System.Collections;

public class OrbitCamera : MonoBehaviour {
   [SerializeField] private Transform target;

   public float rotSpeed = 1.5f;
   private float _rotY;
   private Vector3 _offset;

   void Start() {
       _rotY = transform.eulerAngles.y;
       _offset = target.position - transform.position;
   }

   void LateUpdate() {
       float horInput = Input.GetAxis("Horizontal");
       if (horInput != 0) {
            _rotY += horInput * rotSpeed;
       } else {
            _rotY += Input.GetAxis("Mouse X") * rotSpeed * 3;
       }

       Quaternion rotation = Quaternion.Euler(0, _rotY, 0);
       transform.position = target.position - (rotation * _offset);
       transform.LookAt(target);
   }
}
```

Сериализованная ссылка на объект, вокруг которого производится облет.

Сохраняем начальное смещение между камерой и целью.

Медленный поворот камеры при помощи клавиш со стрелками...

... или быстрый поворот с помощью мыши.

Где бы ни находилась камера, она всегда смотрит на цель.

Поддерживаем начальное смещение, сдвигаемое в соответствии с поворотом камеры.

В листинге обратите внимание на переменную для целевого объекта. Код должен знать, вокруг какого объекта будет вращаться камера, поэтому данная переменная была сериализована, чтобы появиться в редакторе Unity и дать возможность связать с ней объект `player`. Следующая пара переменных связана с углами поворота и используется тем же способом, что и в коде управления камерой в главе 2. Еще код содержит объявление переменной `_offset`; в методе `Start()` ей присваивается разница в положении камеры и целевого объекта. Она позволяет в процессе выполнения кода сценария сохранять относительное положение камеры. Другими словами, камера все время остается на одном и том же расстоянии от целевого объекта, в какую бы сторону она ни поворачивалась. Остальная часть кода помещена в метод `LateUpdate()`.

СОВЕТ Метод `LateUpdate()` также относится к классу `MonoBehaviour` и, подобно методу `Update()`, запускается в каждом кадре. Как понятно из его имени, он вызывается для всех объектов после того, как с ними поработал метод `Update()`. В результате обновление камеры происходит только после перемещения целевого объекта.

Во-первых, код увеличивает значение поворота в зависимости от элементов управления вводом. У нас есть элементы двух типов: клавиши с горизонтальными стрелками и горизонтальные перемещения указателя мыши, — поэтому для переключения между ними используется условный оператор. Код проверяет, нажимаются ли клавиши со стрелками; при положительном результате проверки применяется этот тип ввода, в противном случае проверяется указатель мыши. Независимая проверка двух вариантов ввода позволяет в каждом случае задавать собственную скорость вращения.

Во-вторых, код задает положение камеры на основе положения целевого объекта и угла поворота. Скорее всего, самый непонятный фрагмент кода — это строка `transform.position`, содержащая математические вычисления, с которыми вы пока не сталкивались. Умножение вектора `position` на кватернион (обратите внимание, что угол поворота преобразован в кватернион методом `Quaternion.Euler`) дает новое положение, смещенное в соответствии с углом поворота. Этот новый вектор положения затем прибавляется к смещению от положения персонажа, что дает нам положение камеры. Этапы и подробный механизм вычислений этой компактной строчки кода иллюстрирует рис. 8.7.

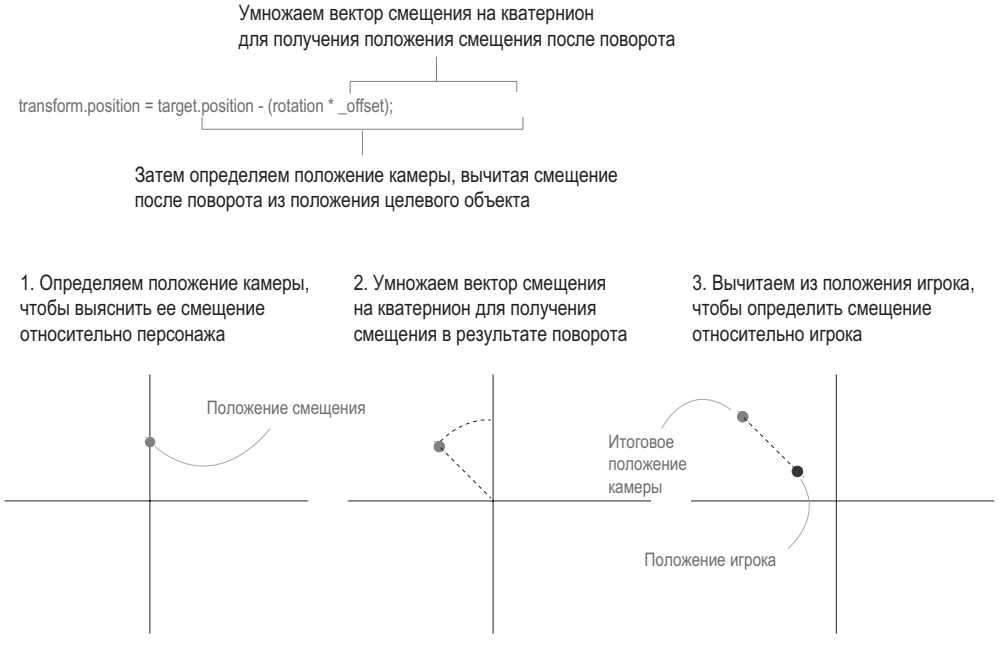


Рис. 8.7. Этапы вычисления положения камеры

ПРИМЕЧАНИЕ У знатоков математики может возникнуть вопрос: «Почему не воспользоваться преобразованиями от одной координатной системы к другой, о которых шла речь в главе 2?» Разумеется, мы можем преобразовать положение точки смещения, воспользовавшись углами Эйлера, но для этого сначала нужно перейти к другой системе координат. Намного проще без этого обойтись.

Завершает код метод `LookAt()`, направляющий камеру на целевой объект; он предназначен для нацеливания одного объекта (не обязательно камеры) на другой. Рассчитанное ранее значение поворота используется для размещения камеры под нужным углом к целевому объекту, но при этом камера уже не поворачивается, меняется только ее положение. То есть без заключительной строки с методом `LookAt` камера будет летать вокруг персонажа, при этом смотря в разные стороны. Превратите эту строку в комментарий и посмотрите, что получится.

Итак, мы написали сценарий, перемещающий камеру вокруг персонажа; пришла очередь кода, перемещающего персонаж по сцене.

8.2. Элементы управления движением, связанные с камерой

Теперь, когда мы импортировали в Unity модель персонажа и написали код, управляющий видом с камеры, нужны элементы управления персонажем. Напишем код для связанных с камерой элементов управления, которые будут перемещать персонаж по сцене при нажатии клавиш со стрелками, а также поворачивать его лицом в нужную сторону.

ЧТО ОЗНАЧАЕТ «СВЯЗАННЫЕ С КАМЕРОЙ»?

Принцип «связи с камерой» неочевиден, но понять его крайне важно. Во многом он напоминает уже знакомую ситуацию с локальными и глобальными координатами, когда «лево» с точки зрения объекта может не совпадать с «лево» в общем смысле. Аналогично обстоит дело с выражением «персонаж бежит налево». Подразумеваться может как левая сторона с точки зрения персонажа, так и левая сторона экрана.

В игре от первого лица камера помещена внутрь персонажа и перемещается вместе с ним, поэтому понятие «лево» с точки зрения камеры и персонажа одно и то же. В игре от третьего лица камера и персонаж существуют отдельно друг от друга, поэтому, к примеру, если камера смотрит персонажу в лицо, лево с точки зрения камеры и с точки зрения персонажа находится с противоположных сторон. Именно поэтому настройки элементов управления нужно выбирать заранее.

Тут возможны разные варианты, но, как правило, в играх от третьего лица элементы управления настраиваются относительно камеры. При нажатии клавиши с левой стрелкой персонаж бежит по экрану налево. Многочисленные эксперименты с другими схемами элементов управления показали, что интуитивно понятным является именно вариант, когда «лево» означает «левую сторону экрана» (что далеко не всегда совпадает с левой стороной персонажа).

Реализация связанных с камерой элементов управления состоит из двух этапов: сначала персонаж ориентируется в соответствии с направлением элементов управления, а затем мы двигаем его вперед. Давайте напишем соответствующий код.

8.2.1. Поворот персонажа лицом в направлении движения

Напишем код, разворачивающий персонаж в направлении клавиш со стрелками. Создайте сценарий `RelativeMovement` (листинг 8.2), перетащите его на объект `player` и свяжите камеру со свойством Target компонента `Script` (таким же образом, как вы связывали персонаж со свойством Target сценария камеры). После этого при нажатии управляющих клавиш персонаж будет поворачиваться в разные стороны, выбирая

направление относительно камеры, а при его вращении с помощью мыши останется статичным.

Листинг 8.2. Поворот персонажа относительно камеры

```
using UnityEngine;
using System.Collections;

public class RelativeMovement : MonoBehaviour {
   [SerializeField] private Transform target;

   void Update() {
      Vector3 movement = Vector3.zero;

      float horInput = Input.GetAxis("Horizontal");
      float vertInput = Input.GetAxis("Vertical");
      if (horInput != 0 || vertInput != 0) {
          movement.x = horInput;
          movement.z = vertInput;

          Quaternion tmp = target.rotation;
          target.eulerAngles = new Vector3(0, target.eulerAngles.y, 0);
          movement = target.TransformDirection(movement);
          target.rotation = tmp;

          transform.rotation = Quaternion.LookRotation(movement);
      }
   }
}
```

Сценарию нужна ссылка на объект, относительно которого происходит перемещение.

Начинаем с вектора (0, 0, 0), постепенно добавляя компоненты движения.

Движение обрабатывается только при нажатии клавиш со стрелками.

Сохраняем начальную ориентацию, чтобы вернуться к ней после завершения работы с целевым объектом.

Преобразуем направление движения из локальных в глобальные координаты.

Метод LookRotation() вычисляет кватернион, смотрящий в этом направлении.

Как и в листинге 8.1, наш код начинается с сериализованной переменной для целевого объекта. Если в предыдущем случае требовалась ссылка на объект, вокруг которого должен совершаться облет, то теперь нужна ссылка на объект, относительно которого будет выполняться перемещение. После чего переходим к методу `Update()`. Его первая строчка объявляет, что `Vector3` имеет значение `0`, `0`, `0`. Важно взять нулевой вектор и затем присвоить ему значения, а не просто создать вектор с вычисленными значениями перемещения. Дело в том, что перемещение по вертикали и по горизонтали вычисляется на разных этапах, но при этом их значения должны быть частями одного и того же вектора.

Затем проверяем элементы управления вводом, как это делалось в предыдущих сценариях. Именно здесь вектору движения присваиваются значения координат X и Z, определяющие горизонтальные перемещения по сцене. Если ни одна клавиша не нажата, метод `Input.GetAxis()` возвращает значение `0`, а при нажатии клавиш со стрелками его значение меняется от `1` до `-1`. Присваивая это значение вектору движения, мы задаем перемещение в положительном или отрицательном направлении оси (в случае оси X это перемещение влево и вправо, в случае оси Z — вперед и назад).

В следующих строках вектор движения корректируется относительно камеры, а именно: метод `TransformDirection()` выполняет преобразование локальных координат в глобальные. Он уже применялся для этой цели в главе 2, но на этот раз мы совершаем переход для системы координат целевого объекта, а не персонажа. Код до и после

строки с методом `TransformDirection()` выравнивает систему координат нужным нам образом: первым делом сохраняется ориентация целевого объекта, чтобы потом к ней можно было вернуться, а затем преобразование поворота корректируется таким образом, чтобы оно совершалось только относительно оси Y, а не всех трех осей. После чего мы выполняем преобразование и восстанавливаем ориентацию целевого объекта.

Весь этот код вычисляет направление движения в виде вектора. Последняя строка назначает полученный результат к персонажу, преобразуя `Vector3` в `Quaternion` методом `Quaternion.LookDirection()` и присваивая это значение. Попробуйте запустить игру и посмотреть, что получится!

ПЛАВНОЕ ВРАЩЕНИЕ С ПОМОЩЬЮ АЛГОРИТМА ЛИНЕЙНОЙ ИНТЕРПОЛЯЦИИ

Сейчас персонаж меняет ориентацию скачками, мгновенно поворачиваясь лицом в направлении, заданном клавишей. Но плавное движение смотрится куда лучше. Здесь на помощь приходит линейный алгоритм интерполяции. Добавьте в сценарий переменную:

```
public float rotSpeed = 15.0f;
```

Затем замените строчку `transform.rotation...` в конце листинга 7.2 следующим кодом:

```
        ...
        Quaternion direction = Quaternion.LookRotation(movement);
        transform.rotation = Quaternion.Lerp(transform.rotation, direction,
rotSpeed * Time.deltaTime);
    }
  }
}
```

В результате вместо привязки непосредственно к значению, возвращаемому методом `LookRotation()`, это значение начнет использоваться в качестве целевого положения, в которое объект будет постепенно поворачиваться. Метод `Quaternion.Lerp()` выполняет плавный поворот из текущего положения в целевое (третий параметр метода контролирует скорость вращения).

Кстати, плавный переход от одного значения к другому называется *интерполяцией*; интерполировать можно значения любых типов. Термин Lerp расшифровывается как linear interpolation — линейная интерполяция. В Unity есть методы `Lerp` как для векторов, так и для значений типа `float` (то есть интерполировать можно положение объектов, цвета и другие параметры). Для кватернионов имеется родственный метод `Slerp` (сферической линейной интерполяции). Для поворотов на маленькие углы преобразования `Slerp` зачастую подходят лучше, чем `Lerp`.

Пока персонаж умеет только поворачиваться на месте; в следующем разделе мы напишем код, который заставит его перемещаться по сцене.

ПРИМЕЧАНИЕ Так как поворотом налево и направо управляют клавиши, отвечающие за облет камеры, поворот персонажа в сторону будет сопровождаться медленным вращением. Такое дублирование функций элементов управления является желательным поведением для данного проекта.

8.2.2. Движение вперед в выбранном направлении

В главе 2 вы узнали, что для перемещения персонажа по сцене к нему нужно добавить соответствующий контроллер. Выделите объект `player` и выберите в меню Components

команду Physics > Character Controller. На панели Inspector уменьшите параметр Radius до `0,4`, остальным параметрам оставьте значения по умолчанию, так как они вполне подходят для нашей модели персонажа.

Следующий листинг содержит код, который нужно добавить в сценарий `RelativeMovement`.

Листинг 8.3. Код, меняющий положение персонажа

```
using UnityEngine;
using System.Collections;

 [RequireComponent(typeof(CharacterController))]
public class RelativeMovement : MonoBehaviour {
...
public float moveSpeed = 6.0f;

private CharacterController _charController;

void Start() {
   _charController = GetComponent<CharacterController>();
}

   void Update() {
      ...
        movement.x = horInput * moveSpeed;
        movement.z = vertInput * moveSpeed;
        movement = Vector3.ClampMagnitude(movement, moveSpeed);
        ...
      }

      movement *= Time.deltaTime;
      _charController.Move(movement);
   }
}
```

Окружающие строки показывают контекст размещения метода RequireComponent().

Шаблон, знакомый по предыдущим главам. Используется для доступа к другим компонентам.

Переписываем существующие строки X и Z, чтобы добавить скорость движения.

Ограничиваем движение по диагонали той же скоростью, что и движение вдоль оси.

Перемещения всегда нужно умножать на deltaTime, чтобы они были независимыми от частоты кадров.

Запустив воспроизведение игры, вы увидите, как персонаж перемещается по сцене (в Т-образной позе). Практически весь код этого листинга вы уже видели, поэтому ограничусь кратким обзором.

Во-первых, в верхней части кода появился метод `RequireComponent()`. В главе 2 я объяснял, что этот метод заставляет Unity проверять наличие у объекта `GameObject` компонента именно того типа, который был передан в команду. Эта строка добавляется по желанию; но без этого компонента в сценарии появятся ошибки.

Затем мы объявляем переменную для перемещений, после чего сценарий получает ссылку на контроллер персонажа. Как вы помните, метод `GetComponent()` возвращает остальные присоединенные к этому объекту компоненты, и если объект не указывается явным образом, подразумевается вариант `this.GetComponent()` (то есть объект, с которым работает сценарий).

Величины перемещений берутся из данных, которые предоставляют элементы управления вводом. В предыдущем листинге такой прием уже использовался, но сейчас мы

учитываем еще и скорость перемещения. Мы умножаем обе оси, вдоль которых происходит движение, на его скорость, а затем метод `Vector3.ClampMagnitude()` ограничивает модуль вектора этой скоростью. Если так не сделать, у движения по диагонали будет больший вектор, чем у движения вдоль осей (нарисуйте катеты и гипотенузу прямоугольного треугольника).

Напоследок мы умножаем значения перемещения на параметр deltaTime, чтобы сделать данное преобразование независимым от частоты кадров (это обеспечивает перемещение персонажа с одной и той же скоростью на разных компьютерах с разной частотой кадров). Полученные значения передаются методу `CharacterController.Move()`, который и приводит персонаж в движение.

Этот код обеспечивает перемещения в горизонтальном направлении, нам же нужно, чтобы персонаж мог перемещаться еще и по вертикали.

8.3. Прыжки

В предыдущем разделе мы написали код перемещения персонажа по поверхности. Но в начале главы упоминалась возможность прыжков. Пришла пора ее добавить, тем более что в большинстве игр от третьего лица есть соответствующий элемент управления. И даже при его отсутствии перемещение по вертикали происходит практически всегда, так как персонаж может сорваться вниз, например, пытаясь преодолеть пропасть. Код, который мы добавим, будет обрабатывать как прыжки, так и падения. Точнее, он будет включать в себя силу тяжести, все время тянущую персонаж вниз, а в момент прыжка его будет толкать вверх.

Но сначала добавим в сцену пару платформ. Ведь пока персонажу некуда запрыгивать и неоткуда падать! Создайте несколько кубов, поменяйте их размер по собственному вкусу и расположите в сцене. Лично я добавил два куба со следующими параметрами: Position `5`, `0.75`, `5` Scale `4`, `1.5`, `4`; и Position `1`, `1.5`, `5.5`, Scale `4`, `3`, `4`. Их вид показан на рис. 8.8.

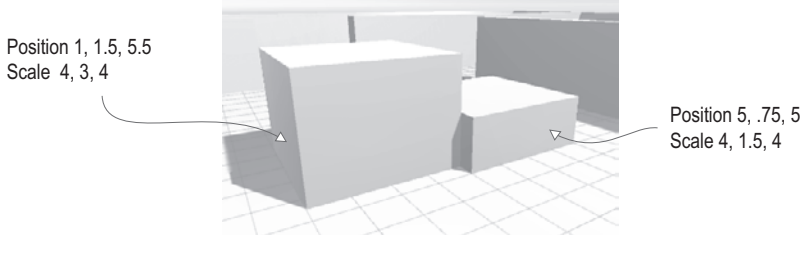


Рис. 8.8. Пара платформ, добавленных в сцену

8.3.1. Вертикальная скорость и ускорение

В начале работы над сценарием `RelativeMovement` я упоминал, что величины перемещений вычисляются по отдельности и все время добавляются к вектору перемещения. Следующий листинг добавит к этому вектору движение по вертикали.

Листинг 8.4. Добавление движения по вертикали в сценарий RelativeMovement

```
...
public float jumpSpeed = 15.0f;
public float gravity = -9.8f;
public float terminalVelocity = -10.0f;
public float minFall = -1.5f;

private float _vertSpeed;
...
void Start() {
   _vertSpeed = minFall;
   ...
}

   void Update() {
      ...
      if (_charController.isGrounded) {
          if (Input.GetButtonDown("Jump")) {
              _vertSpeed = jumpSpeed;
          } else {
              _vertSpeed = minFall;
          }
      } else {
          _vertSpeed += gravity * 5 * Time.deltaTime;
          if (_vertSpeed < terminalVelocity) {
              _vertSpeed = terminalVelocity;
          }
      }
      movement.y = _vertSpeed;

      movement *= Time.deltaTime;
      _charController.Move(movement);
   }
}
```

Инициализируем переменную вертикальной скорости, присваивая ей минимальную скорость падения в начале существующей функции.

Свойство isGrounded компонента CharacterController проверяет, соприкасается ли контроллер с поверхностью.

Реакция на кнопку Jump при нахождении на поверхности.

Если персонаж не стоит на поверхности, применяем гравитацию, пока не будет достигнута предельная скорость.

Конец листинга 8.3, чтобы вы могли понять, куда вставлять новый код.

Как обычно, первым делом добавляем в верхнюю часть сценария новые переменные для различных значений скорости и их корректной инициализации. Оставляем без изменений код до большого оператора `if`, задающего перемещения по горизонтали, а потом добавляем еще один большой оператор `if`, задающий перемещения по вертикали. Этот новый фрагмент кода проверяет, стоит ли персонаж на поверхности, так как именно от этого зависит изменение вертикальной скорости. Компонент `CharacterController` обладает свойством `isGrounded`, предназначенным для проверки соприкосновения персонажа с поверхностью. Оно получает значение `true`, если нижняя часть контроллера персонажа в последнем кадре сталкивается с любым объектом.

Если персонаж стоит на поверхности, значение вертикальной скорости (это закрытая переменная `_vertSpeed`) становится нулевым. Раз персонаж не падает, очевидно, что его вертикальная скорость равна 0. Если после этого персонаж спрыгнет с края платформы, получится вполне натуральное падение, так как его скорость начнет увеличиваться от нуля.

ПРИМЕЧАНИЕ На самом деле в качестве начального значения вертикальной скорости мы присваиваем не 0, а `minFall` — небольшое движение вниз, которое в процессе перемещений по горизонтали слегка придавливает персонажа к поверхности. Небольшая, направленная вниз сила нужна для перемещений вверх и вниз по пересеченной местности.

При нажатии кнопки, отвечающей за прыжок, возникает новая ситуация. Вертикальная скорость должна увеличиться. Оператор `if` проверяет результат метода `GetButtonDown()` — новой функции ввода, во многом аналогичной методу `GetAxis()`. Этот метод также возвращает состояние элемента управления вводом. Клавиша, отвечающая за прыжок, выбирается на панели Inspector, как и клавиши перемещения вдоль горизонтальной и вертикальной осей. Для доступа к нужному набору настроек выберите в меню Edit команду Project Settings > Input (по умолчанию эта настройка имеет значение Space, то есть прыжок совершается нажатием клавиши Space).

Когда персонаж не стоит на поверхности, скорость движения вверх должна непрерывно уменьшаться под действием силы тяжести. Обратите внимание, что код не присваивает значение скорости, а производит ее декремент; по сути, мы получаем направленное вниз ускорение, благодаря которому падение становится реалистичным. Прыжки также происходят по дуге, так как скорость перемещения персонажа вверх постепенно уменьшается до 0, и он начинает падать.

При этом код гарантирует, что скорость движения вниз не превысит предельного значения. Обратите внимание, что для этого используется оператор «меньше, чем», а не «больше, чем», так как скорость движения вниз имеет отрицательное значение. После большого оператора `if` рассчитанная вертикальная скорость присваивается оси Y вектора движения.

Это все, что требуется для моделирования реалистичного перемещения по вертикали! Благодаря направленному вниз постоянному ускорению, когда персонаж не касается поверхности, и правильной корректировке скорости при его расположении на поверхности, код прекрасно имитирует падение. Но все это работает только при корректном распознавании поверхности. И здесь есть одна тонкость, о которой мы поговорим в следующем разделе.

8.3.2. Распознавание поверхностей с учетом краев и склонов

В предыдущем разделе вы узнали, что свойство `isGrounded` компонента `CharacterController` показывает, сталкивалась ли в последнем кадре нижняя часть контроллера персонажа с каким-либо объектом. В большинстве случаев оно дает прекрасные результаты, но, возможно, вы уже заметили, что, пытаясь сойти с края платформы, персонаж как бы зависает в воздухе. Дело в том, что область столкновений персонажа — это окружающая его капсула (ее можно увидеть, выделив объект), и в начале движения за пределы платформы нижняя часть этой капсулы остается в контакте с поверхностью, как показано на рис. 8.9.

В текущем варианте распознавания поверхности вышеописанная проблема возникает и при попадании персонажа на наклонную плоскость. Создайте наклонный блок, опирающийся на одну из платформ. У меня это был куб, для которого значения Position составили `-1.5`, `1.5`, `5`, Rotation — `0`, `0`, `-25`, а Scale — `1`, `4`, `4`.

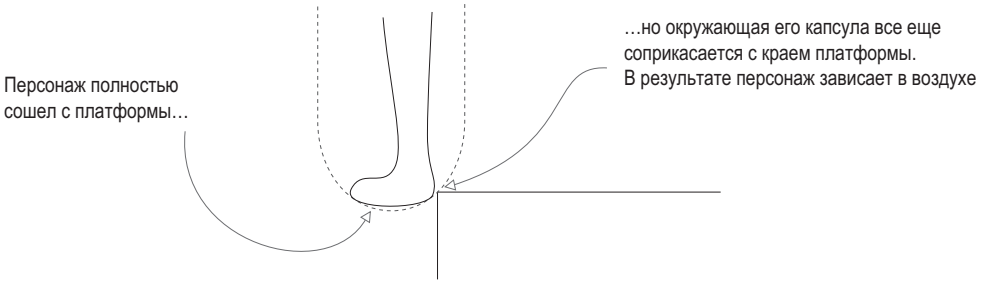


Рис. 8.9. Схема контакта капсулы контроллера и края платформы

Если попытаться запрыгнуть на такой склон с поверхности, персонажу ничто не помешает после первого прыжка совершить второй и оказаться наверху. Ведь склон соприкасается с фрагментом капсулы, а любые касания нижней части наш код рассматривает как наличие твердой точки опоры. Но это нереалистичное поведение; после первого прыжка персонаж должен скатиться по склону вниз.

ПРИМЕЧАНИЕ Скатываться вниз персонаж должен только с крутых склонов. Пологие склоны, например неровности почвы, следует пробегать, не обращая на них внимания. Чтобы получить такой вариант рельефа для тестирования, создайте куб и присвойте параметру Position значения `5.25`, `0.25`, `0.25`, параметру Rotation значения `0`, `90`, `75`, а параметру Scale значения `1`, `6`, `3`.

У всех этих проблем одна и та же причина: то, как нижняя часть персонажа распознает столкновения, не позволяет однозначно определить положение персонажа по отношению к поверхности. Давайте вместо этого попробуем распознать поверхность методом бросания лучей. В главе 3 наш искусственный интеллект таким способом определял наличие перед ним препятствий; используем этот подход, чтобы определить тип поверхности под персонажем. Луч следует бросать вниз из точки, в которой он находится. Столкновение, зарегистрированное непосредственно под ногами персонажа, будет означать, что он стоит на поверхности.

В результате может возникнуть ситуация, когда с точки зрения луча поверхность под ногами персонажа отсутствует, в то время как контроллер с ней сталкивается. На рис. 8.9 капсула соприкасалась с платформой, хотя персонаж уже стоял за ее пределами. Рисунок 8.10 добавляет в эту схему метод бросания лучей, демонстрируя другой вариант развития событий: луч не сталкивается с платформой, в то время как капсула касается ее края. Код должен каким-то образом обработать эту ситуацию.

В данном случае код должен заставить персонаж соскользнуть с уступа. При этом он будет падать (так как не стоит на поверхности), но начнет отталкиваться от точки касания (так как капсулу нужно отодвинуть от платформы, с которой она соприкасается). Соответственно, код распознает столкновение с помощью контроллера персонажа и отреагирует на него небольшим толчком в сторону.

Следующий листинг добавляет в вертикальное движение все упомянутые выше аспекты.

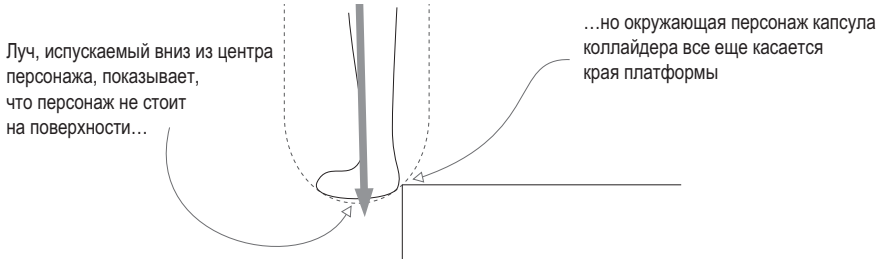


Рис. 8.10. Схема бросания лучей вниз при сходе с края платформы

Листинг 8.5. Распознавание поверхности методом бросания лучей

```
...
private ControllerColliderHit _contact;
...
        bool hitGround = false;
        RaycastHit hit;
        if (_vertSpeed < 0 &&
            Physics.Raycast(transform.position, Vector3.down, out hit)) {
            float check =
            (_charController.height + _charController.radius) / 1.9f;
        hitGround = hit.distance <= check;
      }

      if (hitGround) {
          if (Input.GetButtonDown("Jump")) {
              _vertSpeed = jumpSpeed;
          } else {
              _vertSpeed = minFall;
          }
      } else {
          _vertSpeed += gravity * 5 * Time.deltaTime;
          if (_vertSpeed < terminalVelocity) {
              _vertSpeed = terminalVelocity;
          }

          if (_charController.isGrounded) {
              if (Vector3.Dot(movement, _contact.normal) < 0) {
                  movement = _contact.normal * moveSpeed;
              } else {
                  movement += _contact.normal * moveSpeed;
              }
          }
      }
      movement.y = _vertSpeed;

      movement *= Time.deltaTime;
      _charController.Move(movement);
   }

   void OnControllerColliderHit(ControllerColliderHit hit) {
      _contact = hit;
   }
}
```

Нужно для хранения данных о столкновении между функциями.

Проверяем, падает ли персонаж.

Расстояние, с которым производится сравнение (слегка выходит за нижнюю часть капсулы).

Вместо проверки свойства isGrounded, смотрим на результат метода бросания лучей.

Метод бросания лучей не обнаруживает поверхности, но капсула с ней соприкасается.

Реакция меняется в зависимости от того, смотрит ли персонаж в сторону точки контакта.

При распознавании столкновения данные этого столкновения сохраняются в методе обратного вызова.

Этот листинг содержит практически тот же самый код, что и предыдущий; новый код перемешан с существующим сценарием движения, и крупные фрагменты предыдущего листинга включены с целью задания контекста. Первая строка добавляет новую переменную в верхнюю часть сценария `RelativeMovement`. Эта переменная используется для хранения данных о столкновении между функциями.

Следующие несколько строк посвящены методу бросания лучей. Этот код следует вставить после кода движения по горизонтали, но перед оператором `if`, который задает вертикальное движение. С вызовом метода `Physics.Raycast()` вы уже сталкивались, но на этот раз воспользуемся другим набором параметров. Луч испускается из той же самой точки (местоположения персонажа), но направлен не вперед, а вниз. Нас интересует расстояние, пройденное до момента столкновения; если оно совпадает с расстоянием до ступней персонажа, значит, персонаж стоит на поверхности и свойству `hitGround` можно присвоить значение `true`.

ВНИМАНИЕ Не совсем понятно, как именно определяется расстояние, поэтому рассмотрим процесс расчета более подробно. Мы берем высоту контроллера персонажа (то есть его рост без скругленных углов) и добавляем к ней скругленные углы. Полученное значение делится пополам, так как луч испускается из центра персонажа и проходит до его стоп. Но проверять следует чуть более длинную дистанцию, чтобы учесть небольшие неточности в методе бросания лучей, поэтому высота персонажа делится на 1,9, а не на 2, в итоге луч проходит несколько большее расстояние.

В операторе `if` для движения по вертикали свойство `isGrounded` заменили свойством `hitGround`. Большая часть кода, управляющая перемещениями в вертикальном направлении, осталась без изменений, мы просто добавили код, обрабатывающий ситуацию, когда персонаж уже сошел с поверхности, но контроллер все еще соприкасается с ней (то есть момент схода с края платформы). Здесь появился еще один условный оператор со свойством `isGrounded`, но обратите внимание, что он вложен в инструкцию проверки свойства `hitGround`. То есть свойство `isGrounded` проверяется только при условии, что свойство `hitGround` показало отсутствие поверхности.

Данные о столкновении включают в себя свойство `normal` (еще раз напомню, что нормалью называется вектор, определяющий лицевую сторону полигона), указывающее направление ухода от точки столкновения. Но есть один тонкий момент. Мы хотим, чтобы небольшой импульс в сторону от точки контакта обрабатывался в зависимости от направления движения персонажа. Если предшествующее движение по горизонтали совершалось в сторону платформы, его следует заменить, чтобы остановить смещение персонажа в некорректном направлении. Если же лицом он был повернут в сторону от края, предшествующее движение по горизонтали нужно добавить, чтобы сохранить толкающий вперед импульс. Направление вектора движения относительно точки столкновения определяется при помощи скалярного произведения.

ОПРЕДЕЛЕНИЕ *Скалярным произведением* называется математическая операция между двумя векторами. Результат такого произведения варьируется между -1 и 1, где 1 означает, что векторы смотрят в одном направлении, в то время как -1 указывает на диаметрально противоположные направления. Не следует путать скалярное произведение с другой, часто применяемой к векторам математической операцией — векторным произведением.

Объект `Vector3` обладает методом `Dot()`, вычисляющим скалярное произведение двух векторов. Скалярное произведение вектора движения и нормали плоскости

столкновения будет отрицательным, если эти векторы смотрят в разные стороны, и положительным при их одинаковом направлении.

В конце листинга 8.5 появляется новый метод. Выше мы проверяли нормаль плоскости столкновения, но откуда бралась информация об этой нормали? Оказывается, о столкновениях с контроллером персонажа сообщалось через функцию обратного вызова `OnControllerColliderHit()`, предоставляемую классом `MonoBehaviour`; для программирования реакции на данные о столкновении в каком-либо фрагменте сценария следует сохранить эти данные во внешнюю переменную. Именно этим занимается наш метод: сохраняет данные о столкновении в переменную `_contact`, предоставляя возможность воспользоваться этими данными в методе `Update()`.

Итак, мы скорректировали поведение персонажа на краях платформ и на наклонной плоскости. Запустите игру и посмотрите, что происходит при сходе с края и при попытке запрыгнуть на крутой склон. Демонстрационный ролик почти готов. Персонаж нужным образом перемещается по сцене, осталось придать ему более естественную позу.

8.4. Анимация персонажа

Кроме более сложной формы, которая определяется сеточной геометрией, антропоморфный персонаж нуждается еще и в анимации. В главе 4 мы говорили о том, что анимация — это пакет информации, определяющий движение связанного с ним трехмерного объекта. В качестве примера был рассмотрен персонаж, ходящий по сцене. Именно эту ситуацию нам предстоит воспроизвести! Персонажу следует назначить анимацию, которая заставит его руки и ноги двигаться взад и вперед, как показано на рис. 8.11.

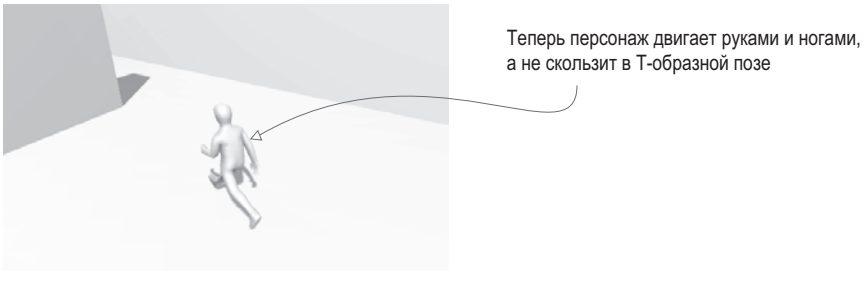


Рис. 8.11. Персонаж, двигающийся по сцене в процессе воспроизведения анимации

В качестве аналогии трехмерной анимации можно вспомнить кукольный театр: трехмерные модели играют роль кукол, аниматор представляет кукловода, а анимация — это запись перемещений кукол. Существуют разные способы ее создания; в большинстве современных игр для персонажей используется техника *скелетной анимации*.

ОПРЕДЕЛЕНИЕ *Скелетной анимацией* (skeletal animation) называется процедура связывания с моделью набора костей, которые и осуществляют движение. При перемещении кости перемещается и связанная с ней поверхность модели.

Принцип скелетной анимации интуитивно понятен. Представьте, что у персонажа появился скелет (как показано на рис. 8.12). Но этот «скелет» — абстракция, применимая в любой ситуации, когда требуется гибкая и деформирующаяся модель с четко определенной структурой, программирующей ее перемещения (представьте, например, извивающееся щупальце осьминога). Сами кости перемещаются как недеформируемые объекты, в то время как окружающая их поверхность модели может сгибаться и менять свою форму.

Результат с рис. 8.11 достигается в несколько этапов: для начала определяются анимационные клипы в импортированном файле, затем настраивается контроллер для воспроизведения этих клипов, и, наконец, этот контроллер вставляется в код. Анимация персонажа будет воспроизводиться в соответствии с написанными сценариями движения.

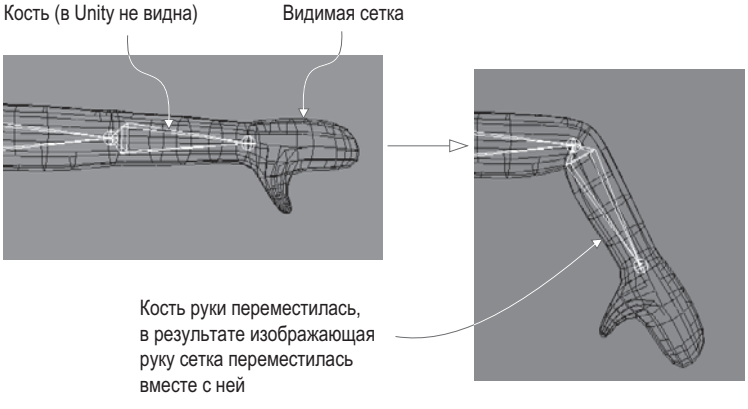


Рис. 8.12. Скелетная анимация антропоморфного персонажа

Разумеется, первым делом нужно включить систему анимации. Выделите модель персонажа на вкладке Project и на панели Inspector перейдите на вкладку Animations, где убедитесь в наличии флажка Import Animation. Затем перейдите на вкладку Rig и выберите в раскрывающемся списке Animation Type вариант Humanoid (ведь у нас антропоморфный персонаж). Обратите внимание на вариант Legacy; до появления системы Mecanim настройки Generic и Humanoid были объединены друг с другом.

СИСТЕМА АНИМАЦИИ MECANIM

В Unity встроена сложная система управления анимацией моделей. Она называется Mecanim. Разговор о ней был начат в главе 6, и я пообещал, что в следующих главах мы рассмотрим ее более подробно. Поэтому в данной главе будут встречаться повторы уже знакомой вам информации, но уже на примере трехмерной, а не двумерной анимации.

Имя Mecanim указывает, что это более новая, усовершенствованная система анимации, добавленная как замена старой версии. Последняя до сих пор доступна через настройку Legacy, но есть вероятность, что в следующих версиях Unity ее уже не будет.

Анимация, с которой мы будем работать, включена в тот же самый файл FBX, что и модель персонажа. Но система Mecanim дает возможность применять анимацию из других файлов FBX.

Например, для всех противников можно использовать один набор анимационных клипов. Такой подход имеет ряд преимуществ, так как дает возможность хранить данные структурированно (модели сохраняются в одну папку, а анимационные ролики — в другую) и экономить время за счет одновременной анимации.

Щелкните на кнопке Apply в нижней части панели Inspector, чтобы применить выбранные настройки к импортированной модели. Теперь можно продолжить работу над определением анимационных клипов.

ВНИМАНИЕ На консоли может появиться предостережение (не ошибка) с текстом «conversion warning: spine3 is between humanoid transforms». На него можно не обращать внимания; оно сообщает, что в импортированной модели больше костей, чем ожидала система Mecanim.

8.4.1. Анимационные клипы для импортированной модели

Первым шагом по созданию анимации для персонажа будет задание отдельных клипов. У персонажа, напоминающего человека, разные движения совершаются в разное время. Наш персонаж будет то бегать по сцене, то запрыгивать на платформы, то просто стоять с опущенными руками. Каждое такое движение представляет собой независимый «клип».

Импортированная анимация зачастую выглядит как один длинный клип, который можно превратить в набор отдельных анимационных роликов. Это делается на вкладке Animations панели Inspector. Там находится список Clips, показанный на рис. 8.13; в нем перечислены все анимационные клипы, изначально входившие в состав единого импортированного клипа. Обратите внимание на кнопки + и – в нижней части списка; они отвечают за добавление и удаление клипов. Для нашего персонажа потребуется четыре клипа, поэтому добавляйте и удаляйте их по мере необходимости.

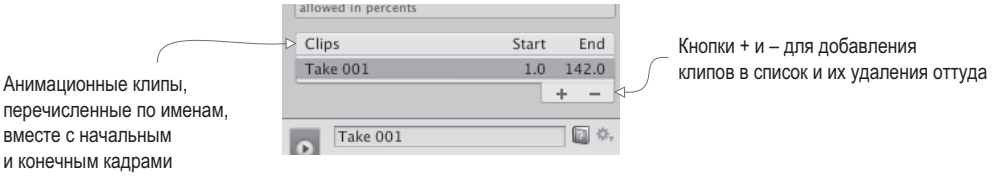


Рис. 8.13. Список Clips на вкладке Animation

Информация о выделенном клипе появляется ниже, как показано на рис. 8.14. Первым идет поле с именем клипа, причем туда можно ввести новое имя. Присвойте нашему первому клипу имя idle. Теперь нужно задать параметры Start и End, определяющие первый и последний кадр выбранного клипа. Именно это позволит вырезать фрагмент из длинной импортированной анимации. Для клипа idle введите в поле Start значение 3, а в поле End — значение 141. Теперь перейдем к настройкам цикла.

ОПРЕДЕЛЕНИЕ *Цикл* (loop) означает запись, которая воспроизводится снова и снова. Зацикленный анимационный клип запускается сразу же после завершения воспроизведения.

Наш клип должен воспроизводиться в цикле, поэтому установите флажки Loop Time и Loop Pose. Кстати, зеленый цвет индикаторной точки означает, что позы персонажа в начале и в конце клипа совпадают, что указывает на корректность циклического воспроизведения. В случае отдельных несовпадений индикатор становится желтым, красный же цвет указывает на совершенно разные позы.

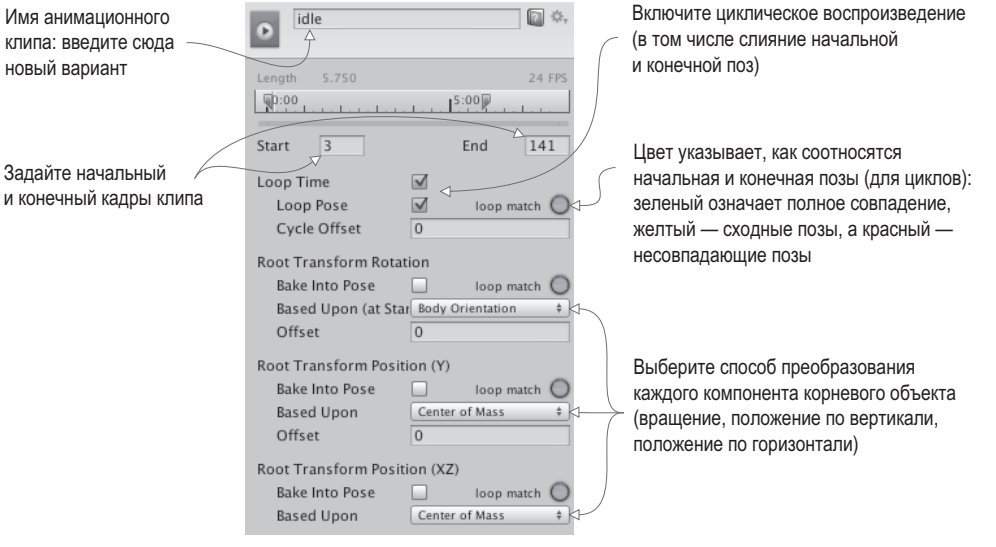


Рис. 8.14. Информация о выбранном анимационном клипе

Ниже находится набор параметров, связанных с преобразованиями корневого элемента. Слово *корневой* в случае скелетной анимации имеет то же самое значение, что и в случае иерархических цепочек: корневым называется объект, с которым связаны все остальные объекты. Соответственно, корень анимации можно представить как основу персонажа, а все остальное двигается относительно этой основы. Для настройки этого элемента предоставляется несколько параметров; возможно, в процессе работы над собственными вариантами анимации вы захотите поэкспериментировать, присваивая им различные значения. Для наших же целей установите значения Body Orientation, Center Of Mass и Center Of Mass.

Щелчок на кнопке Apply добавит к персонажу анимационный клип idle. Выполните указанные операции еще два раза: клип walk должен начинаться в кадре 144 и заканчиваться в кадре 169, а клип run начинаться в кадре 171 и заканчиваться в кадре 190. Все остальные параметры сделайте такими же, как у клипа idle, так как это тоже зацикленная анимация.

Четвертым станет анимационный клип jump, параметры которого будут несколько отличаться. Во-первых, на этот раз требуется не цикл, а неподвижная поза, поэтому снимите флажок Loop Time. Введите в поля Start и End значения `190.5` и `191` соответственно; в данном случае поза задана в одном кадре, но Unity требуются различные значения параметров Start и End. Из-за введенных значений анимация в расположенном

снизу окне предварительного просмотра будет выглядеть не совсем корректно, но на положении персонажа во время игры это не скажется.

Щелкните на кнопке Apply, чтобы подтвердить создание новых анимационных клипов, и переходите к следующему этапу: созданию контроллера анимации.

8.4.2. Контроллер для анимационных клипов

Теперь для персонажа нужно создать контроллер-аниматор. На этом этапе мы определяем состояния анимации и создаем между ними переходы. В каждом состоянии воспроизводится свой набор клипов, а сценарии вызывают переключение контроллера от одного состояния к другому.

Применение дополнительного средства может показаться странным. Зачем между кодом и воспроизведением анимации нужна абстракция в виде контроллера? Скорее всего, раньше вы работали с системами, в которых анимация воспроизводилась непосредственно из кода; кстати, именно такой была старая система Legacy, где применялись вызовы вида `Play("idle")`. Но у непрямого способа есть преимущество. Если раньше воспроизводить анимацию можно было только для той модели, с которой она была связана, теперь мы можем назначать одну и ту же анимацию разным моделям. Сейчас эта возможность нам не потребуется, поэтому просто запомните, что она существует. Получить анимацию можно разными способами. Например, заказать нужные ролики у аниматора или приобрести их в интернет-магазине (например, в Asset Store для Unity).

Давайте создадим новый контроллер-аниматор (в меню Assets выберите команду Create> Animator Controller — не перепутайте с командой Animation, которая создает ресурсы совсем другого типа). На вкладке Project появится значок, украшенный забавной сеткой линий (рис. 8.15); присвойте ему имя player. Выделив персонаж, вы увидите, что у него появился новый компонент Animator; он есть у любой допускающей анимацию модели наряду с компонентом Transform и любыми другими добавленными вами компонентами. Поле Controller этого компонента предназначено для привязки контроллера-аниматора. Перетащите на это поле только что созданный контроллер player (обязательно снимите флажок Apply Root Motion).

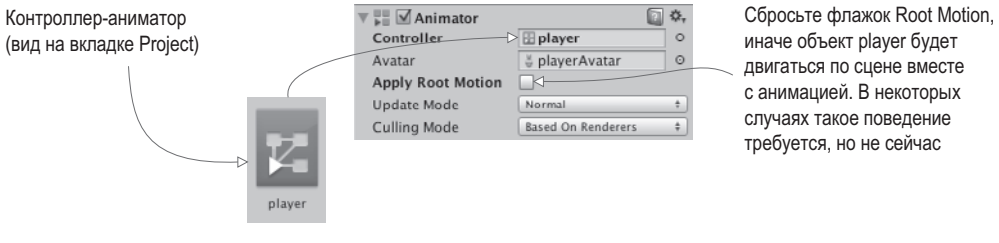


Рис. 8.15. Контроллер-аниматор и компонент Animator

Контроллер-аниматор представляет собой дерево связанных узлов (именно этим объясняется рисунок на значке данного ресурса), просмотр и управление которыми осуществляются на вкладке Animator. Эта вкладка, показанная на рис. 8.16,

аналогично вкладке Scene или Project дает еще одно представление сцены, но по умолчанию она закрыта. Выберите в меню Window команду Animator (не перепутайте с окном Animation). На открывшейся вкладке вы увидите сеть узлов, принадлежащую выделенному в данный момент контроллеру (или контроллеру-аниматору выделенного персонажа).

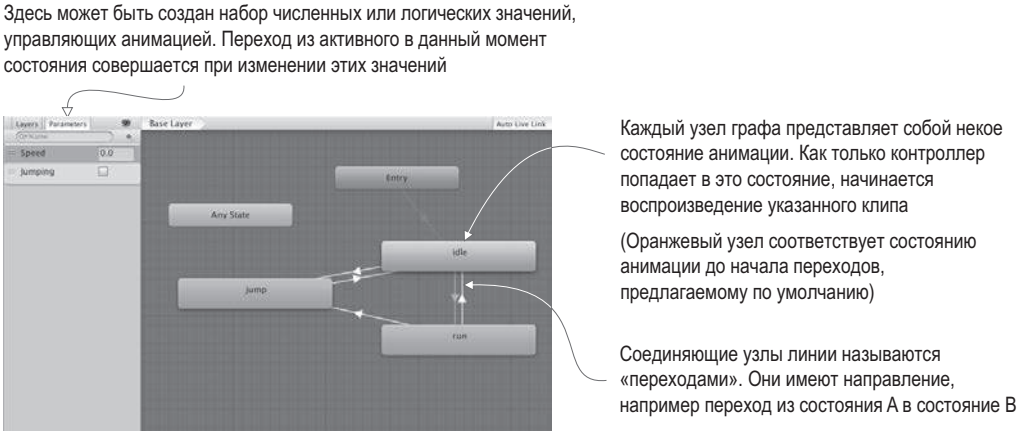


Рис. 8.16. Вид нашего контроллера-аниматора на вкладке Animator

СОВЕТ Напоминаю, что вкладки внутри редактора Unity можно перемещать, располагая в нужных вам местах и формируя интерфейс по собственному вкусу. Мне нравится, когда вкладка Animator находится рядом с окнами Scene и Game.

Изначально мы видим только два существующих по умолчанию узла: Entry и Any State. Узел Any State использоваться не будет. Создадим новые узлы, перетаскивая анимационные клипы. На вкладке Project щелкните на стрелке, расположенной на значке модели персонажа, чтобы посмотреть содержимое этого ресурса, как показано на рис. 8.17. В числе прочего вы увидите и определенные ранее анимационные клипы. Перетащите их на вкладку Animator. Клип walk пока не трогайте (он пригодится для других проектов), ограничьтесь клипами idle, run и jump.

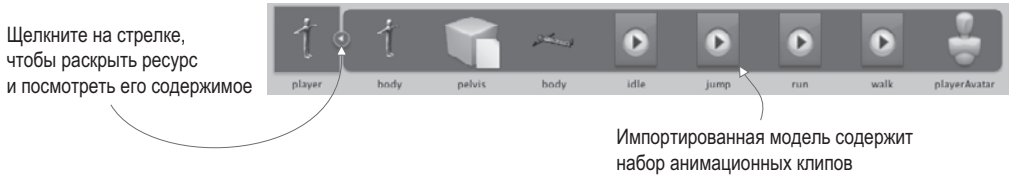


Рис. 8.17. Раскрытый ресурс-модель на вкладке Project

Щелкните правой кнопкой мыши на узле Idle и выберите команду Set As Layer Default State. Узел станет оранжевым, в то время как все остальные останутся серыми; состояние анимации по умолчанию — это место, в котором начинается сеть узлов, до того, как игра

внесла свои коррективы. Узлы нужно соединить друг с другом, обеспечив переходы между состояниями анимации; щелкните на узле правой кнопкой мыши, выберите команду Make Transition и начните перетаскивать стрелку в направлении другого узла. Связь образуется щелчком на этом узле. Соедините узлы в соответствии с образцом с рис. 8.16 (в большинстве случаев требуется переход туда и обратно, исключением является переход от узла jump к узлу run). Эти линии переходов, определяющие связи состояний анимации друг с другом, во время игры будут контролировать переходы из одного состояния в другое.

Переходы обусловлены набором управляющих значений, которые пока отсутствуют. В верхнем левом углу рис. 8.16 видна вкладка Parameters; перейдите на нее, чтобы увидеть панель добавления параметров с кнопкой +. Добавьте параметр типа float с именем Speed и параметр типа Boolean с именем Jumping. Наш код будет менять эти значения, причем они послужат триггерами переходов между состояниями анимации.

Выделите линию перехода для доступа к ее параметрам на панели Inspector, как показано на рис. 8.18.

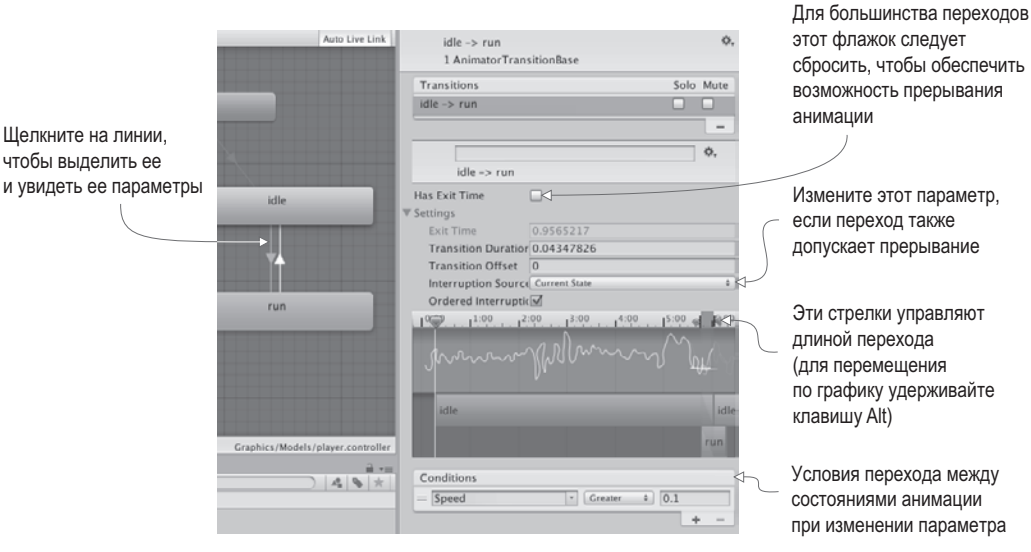


Рис. 8.18. Параметры перехода на панели Inspector

Именно здесь мы указываем, каким образом должны меняться состояния анимации при изменении параметров. Например, выделите переход Idle-Run, чтобы отредактировать условия его возникновения. В свитке Conditions выберите Speed, Greater и 0.1. Снимите флажок Has Exit Time (он заставляет клип воспроизводиться на протяжении всей игры, вместо того чтобы завершиться сразу же после перехода). Затем щелкните на стрелке рядом с названием раздела Settings, чтобы увидеть меню целиком; другие переходы должны быть в состоянии прерывать этот, поэтому выберите в раскрывающемся списке Interruption Source вариант Current State. Повторите эту операцию для всех переходов, перечисленных в табл. 8.1.

Таблица 8.1. Условия всех переходов для данного контроллера анимации

Переход	Условие	Прерывание
Idle-Run	Speed greater чем 0.1	Current State
Run-Idle	Speed less чем 0.1	None
Idle-Jump	Jumping = true	None
Run-Jump	Jumping = true	None
Jump-Idle	Jumping = false	None

В дополнение к параметрам, имеющим вид меню, предоставляется еще и визуальный интерфейс, расположенный непосредственно над списком Condition, как показано на рис. 8.18. График позволяет визуально корректировать продолжительность перехода. Заданное по умолчанию время перехода прекрасно подходит для переходов из Idle в Run и обратно, а вот переходы в состояние Jump и из него должны совершаться быстрее. Окрашенная область графика показывает, сколько времени займет переход; одновременное нажатие клавиши Alt и левой кнопки мыши даст возможность панорамирования графика, а клавиша Alt с правой кнопкой мыши позволят выполнить масштабирование (аналогичные способы управления применяются для представления Scene). Воспользуйтесь кнопками со стрелками над окрашенной областью, чтобы установить для всех трех связанных с прыжками переходов время менее 4 миллисекунд.

Напоследок усовершенствуем анимационную сеть, по очереди выделив узлы и скорректировав порядок переходов. На панели Inspector будет отображаться список всех переходов к выделенному узлу и от него; элементы этого списка можно перетаскивать (манипуляторы управления находятся слева от их значков), меняя их порядок. Поместите переход Jump наверх списка для обоих узлов, Idle и Run, дав ему приоритет над остальными. Если анимация кажется слишком медленной, поменяйте скорость воспроизведения (воспроизведение выглядит лучше всего при скорости 1.5).

Итак, мы настроили контроллер анимации. Теперь можно управлять анимационными клипами через сценарий движения.

8.4.3. Код, управляющий контроллером-аниматором

В заключение добавим методы в сценарий `RelativeMovement`. Как уже упоминалось, большая часть настройки анимационных состояний выполнена внутри контроллера; осталось написать небольшой код, управляющий богатой и гибкой системой анимации.

Листинг 8.6. Задание значений в компоненте Animator

```
...
private Animator _animator;
...
_animator = GetComponent<Animator>();  <-- Добавляем в метод Start().
...

      _animator.SetFloat("Speed", movement.sqrMagnitude);

      if (hitGround) {                                   Сразу после оператора if для
          if (Input.GetButtonDown("Jump")) {             горизонтального движения.
```

```
            _vertSpeed = jumpSpeed;
        } else {
            _vertSpeed = minFall;
            _animator.SetBool("Jumping", false);
        }
    } else {
        _vertSpeed += gravity * 5 * Time.deltaTime;
        if (_vertSpeed < terminalVelocity) {
            _vertSpeed = terminalVelocity;
        }
        if (_contact != null ) {  <-- Не активируйте это значение в начале уровня.
            _animator.SetBool("Jumping", true);
        }

        if (_charController.isGrounded) {
            if (Vector3.Dot(movement, _contact.normal) < 0) {
                movement = _contact.normal * moveSpeed;
            } else {
                movement += _contact.normal * moveSpeed;
            }
        }
    }
...
```

Большая часть кода снова повторяет предыдущие листинги; код анимации имеет множество пересечений с существующим сценарием движения. Выберите строки с переменной `_animator`, чтобы найти фрагменты, предназначенные для вставки в код.

Сценарию нужна ссылка на компонент Animator, после чего код присваивает контроллеру-аниматору значения (типа float или Boolean). Условие (`_contact != null`) перед заданием логической переменной, отвечающей за прыжок, мешает контроллеру запустить клип с прыжком сразу после начала действия. С технической точки зрения падение персонажа длится долю секунды, но информация о столкновении появится только после его первого соприкосновения с поверхностью.

Итак, все готово! Мы создали демонстрационный ролик движения от третьего лица с элементами управления, связанными с камерой и анимированным персонажем.

Заключение

- Вид от третьего лица означает, что камера перемещается вокруг персонажа, а не вместе с ним.
- Вид сцены улучшает имитация теней, которая выполняется путем визуализации в реальном времени или с помощью карт освещенности.
- Действие элементов управления можно определить относительно камеры, а не персонажа.
- Улучшить распознавание поверхностей в Unity можно, бросив вниз луч.
- Сложная анимация, настраиваемая в Unity с помощью контроллера-аниматора, позволяет моделировать антропоморфные персонажи.

9 Интерактивные устройства и элементы

- ✓ Программирование дверей, которые может открывать персонаж (срабатывают при нажатии клавиши или при столкновении).
- ✓ Включение имитации физической среды для моделирования разлетающихся коробок.
- ✓ Элементы, которые игрок сможет сохранять как инвентарь.
- ✓ Код управления состоянием игры, например данными об инвентаре.
- ✓ Подготовка и использование инвентаря.

В этой главе мы детально рассмотрим реализацию функциональных элементов. Вы уже получили подробную информацию о различных элементах готовой игры: перемещениях, противниках, пользовательском интерфейсе и т. п. Но взаимодействие до этого момента осуществлялось только с противниками и было достаточно ограниченным. В этой главе вы научитесь создавать функциональные устройства, например двери. Также мы обсудим сбор элементов. Этот процесс подразумевает как взаимодействие с объектами игрового уровня, так и слежение за состоянием игры. В играх часто приходится учитывать такие вещи, как статистика игрока, прохождение по этапам и т. п. Игровой инвентарь является примером состояния, поэтому мы напишем код, отслеживающий собранные игроком предметы. К концу главы в проекте появится динамическое пространство, похожее на настоящую игру!

Начнем с исследования устройств (таких, как двери), срабатывающих в ответ на нажатие клавиш. После этого будет написан код, распознающий столкновение персонажа с объектом игрового уровня и обеспечивающий такие взаимодействия, как перемещение предметов или сбор инвентаря. Затем для управления данными о собранных предметах инвентаря мы разработаем архитектуру кода в стиле MVC (Model-View-Controller — Модель-Представление-Контроллер). Завершит главу программирование интерфейса, позволяющего пользоваться игровым инвентарем, к примеру открывать дверь при помощи найденного ранее ключа.

ВНИМАНИЕ Предыдущие главы относительно независимы друг от друга, поэтому с технической точки зрения они не требовали проектов из более ранних глав. Теперь же ряд листингов будет получен редактированием сценариев из главы 8. Если вы перешли сразу к этой главе, скачайте пример проекта для главы 8 в качестве рабочей основы.

Пример проекта будет содержать вышеупомянутые устройства и элементы, случайным образом распределенные по игровому уровню. Для полноценной игры требуется тщательно планировать размещение элементов на игровом уровне, но мы пока только тестируем данную функциональность. Впрочем, случайное расположение объектов не помешает получить представление о порядке внедрения различных элементов.

Как обычно, в процессе объяснений мы будем шаг за шагом писать код. Но если вы хотите сразу получить готовую версию, скачайте с сайта пример проекта.

9.1. Двери и другие устройства

Игровые уровни, как правило, состоят из статичных стен и предметов обстановки, к которым добавляются функциональные устройства. Я имею в виду объекты, с которыми может взаимодействовать и которыми может пользоваться персонаж. Это может быть, например, включающийся свет или приходящий во вращение вентилятор. Многообразие устройств ограничено только вашим воображением, но практически все они пользуются одинаковым кодом, позволяющим персонажу активировать их. В этой главе мы рассмотрим пару примеров, что даст вам возможность адаптировать данный код к устройствам других видов.

9.1.1. Открывающиеся и закрывающиеся двери

Первый вид устройства, который нам предстоит запрограммировать, — это открывающаяся и закрывающаяся дверь. Начнем мы с управления дверью путем нажатия клавиши. В игру можно поместить множество устройств с разными способами управления. Из них дверь распространена шире всего, а управление с клавиатуры — наиболее очевидный подход, поэтому мы начнем именно с них.

В стенах есть промежутки, в которые можно поместить объект, блокирующий проход. Я создал куб, в поля Position которого ввел значения `2.5`, `1.5`, `17`, а в поля Scale — значения `5`, `3`, `0.5`. Полученный результат показан на рис. 9.1.

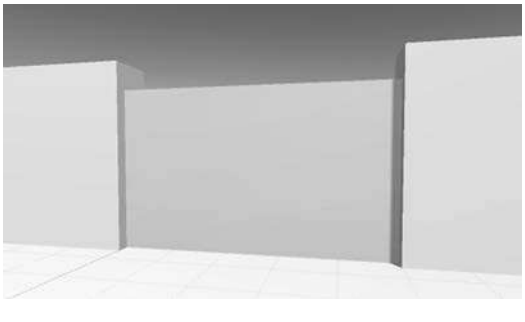


Рис. 9.1. Объект Door, перекрывающий дверной проем

Создайте сценарий с именем `DoorOpenDevice` и назначьте его объекту `door`. Добавьте в него код из следующего листинга, чтобы заставить объект функционировать как дверь.

Листинг 9.1. Сценарий, по команде закрывающий и открывающий дверь

```
using UnityEngine;
using System.Collections;

public class DoorOpenDevice : MonoBehaviour {
  [SerializeField] private Vector3 dPos;

  private bool _open;

  public void Operate() {
    if (_open) {
      Vector3 pos = transform.position - dPos;
      transform.position = pos;
    } else {
      Vector3 pos = transform.position + dPos;
      transform.position = pos;
    }
    _open = !_open;
  }
}
```

Смещение открытой двери относительно закрытой.

Булева переменная для слежения за открытым состоянием двери.

Открываем или закрываем дверь в зависимости от ее состояния.

Первая переменная определяет смещение, возникающее при открывании двери. Дверь при этом сдвигается на указанное расстояние, а затем, чтобы ее закрыть, это значение вычитают. Далее следует закрытая переменная типа `Boolean`, отслеживающая состояние двери. В методе `Operate()` преобразованию `transform` объекта присваивается новое положение, путем добавления или вычитания величины смещения в зависимости от того, закрыта или открыта дверь. После этого состояние переменной `_open` меняется на противоположное.

Как это всегда бывает с сериализованными переменными, переменная `dPos` появляется на панели Inspector. Но она представляет собой структуру `Vector3`, поэтому под именем переменной вместо одного поля появятся три. Укажите относительное положение двери в открытом состоянии; я решил, что она должна уходить вниз, поэтому смещение составило `0`, `-2.9`, `0` (высота объекта `door` равна 3, и смещение вниз на расстояние 2.9 оставит только небольшой порог, выступающий над полом).

ПРИМЕЧАНИЕ Преобразование совершается мгновенно, а вы, возможно, предпочитаете видеть, как открывается дверь. В главе 3 мы говорили о том, что плавное движение объектов можно получить анимацией по начальной и конечной точке. Значение английского термина *tweening* зависит от контекста. В программировании игр он относится к командам кода, приводящим объекты в движение. Рекомендую обратить внимание на инструмент iTween, который можно скачать с официального сайта http://itween.pixelplacement.com.

Теперь нужно вызвать открывающую и закрывающую дверь функцию `Operate()` (обе ситуации обрабатываются одной и той же функцией). Соответствующий сценарий для персонажа пока отсутствует; давайте его напишем.

9.1.2. Проверка расстояния и направления

Создайте сценарий с именем `DeviceOperator`. Следующий код реализует управляющую клавишу для управления расположенными поблизости устройствами.

Листинг 9.2. Клавиша управления устройством

```
using UnityEngine;
using System.Collections;

public class DeviceOperator : MonoBehaviour {
  public float radius = 1.5f;

  void Update() {
    if (Input.GetButtonDown("Fire3")) {
      Collider[] hitColliders = Physics.OverlapSphere(transform.
                                position, radius);
      foreach (Collider hitCollider in hitColliders) {
        hitCollider.SendMessage("Operate",
            SendMessageOptions.DontRequireReceiver);
      }
    }
  }
}
```

Расстояние, на котором становится возможной активация устройств.

Реакция на кнопку ввода, заданную в настройках ввода в Unity.

Метод OverlapSphere() возвращает список ближайших объектов.

Метод SendMessage() пытается вызывать именованную функцию независимо от типа целевого объекта.

Основная часть этого листинга вам уже знакома, но в середине кода появился важный новый метод. В код первым делом задается расстояние, с которого становится возможным управление устройством. Затем в методе `Update()` рассматривается клавиатурный ввод. Клавиша прыжка уже задействована в сценарии `RelativeMovement`, поэтому воспользуемся клавишей `Fire3` (в настройках ввода проекта она определена как левая клавиша Command).

Следующим идет важный новый метод `OverlapSphere()`. Он возвращает массив всех объектов, расположенных не более чем на определенном расстоянии от текущего местоположения. В метод передаются положение персонажа и переменная `radius`, а он распознает все объекты рядом с персонажем. С полученным списком можно поступать как угодно (например, вы можете установить бомбу и назначить объектам силу взрыва). Мы же вызовем для всех этих объектов метод `Operate()`.

Для его вызова вместо обычной точечной нотации применяется метод `SendMessage()`. Этот подход вы уже встречали раньше, при работе с кнопками пользовательского интерфейса. В данном случае мы прибегаем к нему из-за отсутствия точных данных о типе целевого объекта. Как вы помните, метод `SendMessage()` работает со всеми объектами `GameObjects`. Правда, на этот раз мы снабдим его параметром `DontRequireReceiver`. Дело в том, что у большинства объектов, возвращаемых методом `OverlapSphere()`,отсутствует метод `Operate()`. Обычно, если объект не может получить сообщение, метод `SendMessage()` выводит сообщение об ошибке. Но сейчас этого не требуется, ведь мы и так знаем, что большая часть объектов проигнорирует рассылаемые сообщения.

Готовый сценарий нужно присоединить к объекту `player`. После этого открывать и закрывать дверь можно будет, встав рядом с ней и нажав клавишу.

Осталась одна маленькая деталь. На данном этапе положение персонажа не имеет значения, главное, чтобы он располагался близко к двери. Но можно сделать так, чтобы дверь открывалась, только если персонаж стоит к ней лицом. В главе 8 мы определяли направление движения персонажа через скалярное произведение. Эта математическая операция с парой векторов возвращает значения в диапазоне от –1 до 1, где 1 означает одно и то же направление векторов, а –1 указывает на диаметрально противоположные направления. Вот код, который нужно добавить в сценарий `DeviceOperator`.

Листинг 9.3. Код, позволяющий открывать дверь, только стоя к ней лицом

```
...
foreach (Collider hitCollider in hitColliders) {
  Vector3 direction = hitCollider.transform.position - transform.position;
  if (Vector3.Dot(transform.forward, direction) > .5f) {
    hitCollider.SendMessage("Operate",
          SendMessageOptions.DontRequireReceiver);
  }
}
...
```

Сообщение отправляется только при корректной ориентации персонажа.

Чтобы воспользоваться скалярным произведением, первым делом нужно определить, какое направление мы будем проверять. Это направление от персонажа к объекту. Нужный вектор создается вычитанием координат персонажа из координат объекта. Затем вызывается метод `Vector3.Dot()`, в который передаются вычисленный вектор направления и вектор движения персонажа вперед. Если возвращенный методом результат близок к 1 (особенно при наличии в коде условия «больше, чем 0,5»), значит, направление векторов практически совпадает.

Теперь дверь не открывается, если персонаж стоит к ней спиной, как бы близко к ней он ни находился. Такой подход работает с устройствами любого вида. Чтобы убедиться в его гибкости, рассмотрим еще один пример.

9.1.3. Монитор, меняющий цвет

Итак, мы создали закрывающуюся и открывающуюся дверь. Схема управления, с которой вы познакомились в рамках этого упражнения, применима к устройствам любого вида. Для демонстрации создадим еще одно устройство; на этот раз — меняющий цвет монитор, прикрепленный к стене.

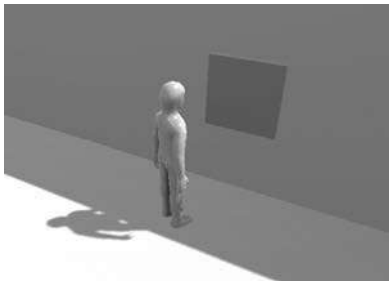


Рис. 9.2. Встроенный в стену монитор, умеющий менять цвет

Создайте новый куб и подвиньте его к стене практически до соприкосновения. Например, я ввел в поля Position значения `10.9`, `1.5`, `-5`. Теперь создайте сценарий `ColorChangeDevice`, добавьте в него код из следующего листинга и присоедините к монитору. Запустите игру, подойдите к монитору и нажмите клавишу, которой вы открывали дверь; монитор начнет менять цвет, как показано на рис. 9.2.

Листинг 9.4. Сценарий устройства, которое может менять цвет

```
using UnityEngine;
using System.Collections;

public class ColorChangeDevice : MonoBehaviour {
  public void Operate() {
    Color random = new Color(Random.Range(0f,1f),
          Random.Range(0f,1f), Random.Range(0f,1f));
    GetComponent<Renderer>().material.color = random;
  }
}
```

Объявление метода с таким же именем, как в сценарии для двери.

Эти числа представляют собой RGB-значения в диапазоне от 0 до 1.

Цвет задается в назначенном объекту материале.

Здесь мы объявили функцию с тем же самым именем, что и в сценарии управления дверью. Управляющий устройствами код всегда содержит функцию с именем `Operate` — она нужна для активации. В данном случае ее код присваивает материалу объекта случайный цвет (напоминаю, что цвет не является атрибутом самого объекта, объекту назначается материал, а у него уже есть цвет).

ПРИМЕЧАНИЕ В большинстве приложений для компьютерной графики цвет определяется компонентами Red, Blue и Green, параметры объекта Color в Unity лежат в диапазоне от `0` до `1`, а не от `0` до `255`, как это обычно бывает (в том числе и в интерфейсе выбора цвета в Unity).

Итак, мы рассмотрели первый подход к управлению устройствами в играх и даже создали пару демонстрационных устройств. Но взаимодействовать с элементами сцены можно и другим способом. Например, врезаясь в них. Давайте посмотрим, как это делается.

9.2. Взаимодействие с объектами через столкновение

В предыдущем разделе мы управляли устройствами при помощи клавиатуры, но это не единственный способ взаимодействия персонажа с элементами уровня. Существует и такой крайне эффективный подход, как реакция на столкновение с персонажем. При этом большую часть операций выполняет Unity благодаря встроенным в игровой движок механизмам распознавания столкновений и модели физической среды. Но Unity помогает только распознать столкновение, а запрограммировать реакцию объекта не сможет никто, кроме вас.

Мы рассмотрим три варианта реакции на столкновение, встречающиеся в играх:

- Толчок и падение.
- Срабатывание устройства.
- Исчезновение в момент контакта (при сборе элементов).

9.2.1. Столкновение с препятствиями, обладающими физическими свойствами

Давайте создадим набор поставленных друг на друга коробок, которые при столкновении с персонажем будут разлетаться по сцене. Моделирование подобных вещей требует сложных физических расчетов, но эту задачу берут на себя встроенные функции Unity, которые и обеспечат нам реалистичный вид разлетающихся коробок.

По умолчанию объекты в Unity лишены физических свойств. Эта функция включается при добавлении компонента `Rigidbody`. Впервые речь о нем зашла в главе 3, где этот компонент потребовался огненным шарам, которые бросали в игрока противники. Здесь я еще раз повторю, что в Unity модель физической среды работает только для объектов с компонентом `Rigidbody`. Щелкните на кнопке Add Component и найдите строку Rigidbody в разделе Physics.

Создайте куб и добавьте к нему компонент `Rigidbody`. Нам требуется несколько таких кубов, положенных друг на друга. Например, в примере проекта, который можно скачать с сайта книги, пять кубов положены в два ряда, как показано на рис. 9.3.

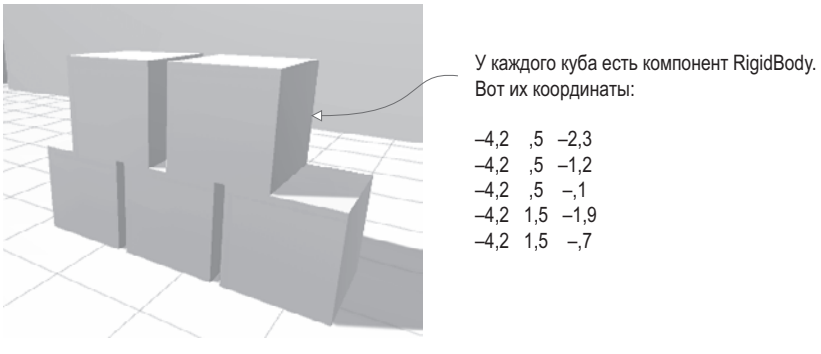


Рис. 9.3. Куча коробок, с которой будет сталкиваться персонаж

Кубы готовы реагировать на приложенные к ним силы. Чтобы источником силы стал персонаж, внесите дополнение из следующего листинга в присоединенный к персонажу сценарий `RelativeMovement` (это один из сценариев, написанных в предыдущей главе).

Листинг 9.5. Добавление физической силы в сценарий RelativeMovement

```
...
public float pushForce = 3.0f;  <-- Величина прилагаемой силы.
...
void OnControllerColliderHit(ControllerColliderHit hit) {
  _contact = hit;

  Rigidbody body = hit.collider.attachedRigidbody;
  if (body != null && !body.isKinematic) {
    body.velocity = hit.moveDirection * pushForce;
  }
}
...
```

Проверяем, есть ли у участвующего в столкновении объекта компонент Rigidbody, обеспечивающий реакцию на приложенную силу.

Назначаем физическому телу скорость.

Особых объяснений этот код не требует: при любом столкновении персонажа с объектом проверяется, есть ли у этого объекта компонент `Rigidbody`. В случае положительного результата проверки этому компоненту присваивается скорость.

Запустите игру и заставьте персонаж побежать прямо на кучу коробок. В момент столкновения куча вполне реалистично рассыплется. Как видите, имитация физического явления не потребовала от вас особых усилий! Благодаря встроенному в Unity механизму имитации физики, не приходится писать объемный код. Мы легко заставили объекты двигаться в ответ на столкновение, но возможен и другой вариант реакции — генерация события срабатывания. Давайте воспользуемся таким событием для управления дверью.

9.2.2. Управление дверью через объект trigger

Если раньше дверь управлялась нажатием клавиши, то теперь она будет открываться и закрываться в ответ на столкновение персонажа с другим объектом сцены. Создайте еще одну дверь и поместите ее в другой проем (я сделал копию уже существующей двери и поместил ее в точку с координатами `-2.5`, `1.5`, `-17`). Теперь создайте еще один куб, который будет играть роль триггера, и установите в разделе Box Collider флажок Is Trigger (вы уже делали подобное, работая над огненными шарами в главе 3). В меню Layer, расположенном в верхнем правом углу панели Inspector, выберите команду Ignore Raycast. Напоследок отключим этому объекту возможность формирования теней (напоминаю, что это делается в разделе Mesh Renderer).

ВНИМАНИЕ Эти небольшие операции легко пропустить, а они крайне важны. Невозможно использовать объект в качестве триггера, если не установлен флажок Is Trigger. Кроме того, обязательно укажите для слоя настройку Ignore Raycast, чтобы триггер не учитывался в операциях, связанных с бросанием лучей.

ПРИМЕЧАНИЕ Когда в главе 3 вы в первый раз столкнулись с объектами-триггерами, к ним требовалось добавить компонент `Rigidbody`. Теперь же он не требуется, так как триггеру не нужно реагировать на действия игрока (раньше учитывалась реакция на столкновения со стенами). Сейчас для работы триггера нужно, чтобы у него или у входящего с ним в контакт объекта была включена модель физической среды; под это требование подпадает не только компонент `Rigidbody`, но и назначенный персонажу компонент `CharacterController`.

Выберите для объекта-триггера такие положение и размер, чтобы в него попала дверь и некоторая область вокруг двери. Я ввел в поля Position значения `-2.5`, `1.5`, `-17` (как у самой двери), а в поля Scale — значения `7.5`, `3`, `6`. Кроме того, имеет смысл назначить триггеру полупрозрачный материал, чтобы его пространство отличалось от остальных объектов. Создайте новый материал командой меню Assets и выделите его на вкладке Project. В верхней части панели Inspector находится раскрывающийся список Rendering Mode (там сейчас выбрано значение Opaque, предлагаемое по умолчанию), в котором нужно выбрать вариант Transparent.

Теперь щелкните на поле образца цвета для вызова окна диалога. В основной части окна выберите зеленый цвет и с помощью нижнего ползунка уменьшите значение канала прозрачности. Перетащите материал со вкладки Project на объект. Рисунок 9.4 демонстрирует вид триггера после назначения материала.

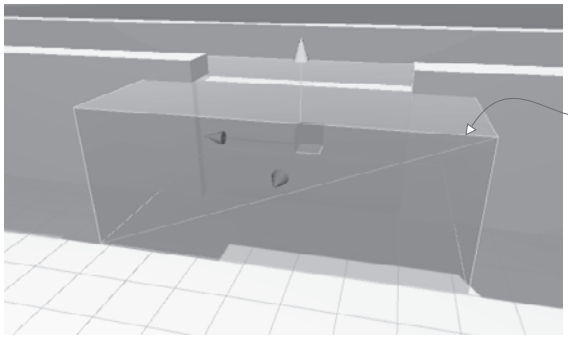


Рис. 9.4. Зона триггера, окружающая дверь, которой он управляет

ОПРЕДЕЛЕНИЕ *Триггеры* часто называются зонами, а не объектами, чтобы провести грань между твердыми телами и объектами, через которые вы можете пройти.

Запустите игру и убедитесь, что персонаж свободно проходит через зону триггера; столкновения с объектом при этом регистрируются, но больше не влияют на движение персонажа. Реакция на столкновения такого рода задается программно. В частности, мы хотим, чтобы триггер управлял дверью. Поэтому создайте новый сценарий `DeviceTrigger` и скопируйте в него код из следующего листинга.

Листинг 9.6. Код для триггера, контролирующего устройство

```
using UnityEngine;
using System.Collections;

public class DeviceTrigger : MonoBehaviour {
  [SerializeField] private GameObject[] targets;

  void OnTriggerEnter(Collider other) {
    foreach (GameObject target in targets) {
      target.SendMessage("Activate");
    }
  }

  void OnTriggerExit(Collider other) {
    foreach (GameObject target in targets) {
      target.SendMessage("Deactivate");
    }
  }
}
```

Список целевых объектов, которые будет активировать триггер.

Метод OnTriggerEnter() вызывается при попадании объекта в зону триггера...

... а метод OnTriggerExit() вызывается при выходе объекта из зоны триггера.

В этом листинге определяется массив целевых объектов; в большинстве случаев он будет состоять из одного элемента, но потенциально один триггер может управлять целым набором устройств. Цикл применяется для рассылки сообщений всем целевым объектам внутри как метода `OnTriggerEnter()`, так и метода `OnTriggerExit()`. Эти методы вызываются однократно при входе другого объекта в зону триггера и выходе из нее (вместо того, чтобы раз за разом вызываться, пока объект находится в зоне триггера).

Обратите внимание, что теперь рассылаются другие сообщения; на этот раз для двери нужно определить функции `Activate()` и `Deactivate()`. Добавьте в сценарий управления дверью код следующего листинга.

Листинг 9.7. Добавление функций активации и деактивации к скрипту DoorOpenDevice

```
...
public void Activate() {
  if (!_open) {  <-- Открывает дверь при условии, что она закрыта.
    Vector3 pos = transform.position + dPos;
    transform.position = pos;
    _open = true;
  }
}
public void Deactivate() {
  if (_open) {  <-- Аналогично, закрывает дверь при условии, что она открыта.
    Vector3 pos = transform.position - dPos;
    transform.position = pos;
    _open = false;
  }
}
...
```

Код новых методов `Activate()` и `Deactivate()` практически совпадает с кодом уже знакомого вам метода `Operate()`, просто мы разделили функции открытия и закрытия двери, в то время как раньше одна функция отвечала за обе операции.

Итак, весь необходимый код написан, и зону триггера можно заставить открывать и закрывать дверь. Свяжите сценарий `DeviceTrigger` с зоной триггера, а дверь свяжите со свойством `Targets`; на панели Inspector сначала укажите размер массива, а затем перетащите объекты со вкладки Hierarchy на появившиеся поля. У нас всего одна дверь, которая будет управляться этим триггером, поэтому введите в поле Size значение 1 и перетащите объект door на поле Element 0.

Запустите игру и посмотрите, что происходит, когда персонаж подходит к двери и отходит от нее. При входе персонажа в зону триггера дверь автоматически открывается, при выходе оттуда автоматически закрывается.

Это еще один замечательный способ добавления интерактивности в игровые уровни! Но триггеры применимы не только к таким устройствам, как двери; с их помощью можно добавлять в игру инвентарь.

УПРАЖНЕНИЕ: ТРИГГЕР ДЛЯ УСТРОЙСТВ В ДВУМЕРНОМ ПЛАТФОРМЕРЕ

Вы научились реализовывать триггеры в трехмерных играх, но в двумерной игре описанная схема действий тоже работает. Просто объекты будут реагировать на двумерные коллайдеры, используя метод `OnTrigger2D`. В качестве упражнения попробуйте добавить зоны триггера и устройства в двумерный платформер из главы 6.

9.2.3. Сбор предметов

В играх часто встречаются предметы, которые может подбирать персонаж. Это оборудование, пакеты для восстановления здоровья и бонусы. Базовый механизм сбора

путем столкновения очень прост; сложности начинаются после завершения этого процесса, но об этом мы поговорим чуть позже.

Создайте сферу и поместите ее в произвольном месте сцены на уровне талии персонажа. Уменьшите ее размеры, введя в поля Scale значения `0.5`, `0.5`, `0.5`. Остальные параметры оставьте такими же, как для большой зоны триггера. В настройках коллайдера установите флажок Is Trigger, в настройках слоя выберите вариант Ignore Raycast и создайте новый материал, чтобы присвоить объекту яркий цвет. Из-за малых размеров объекта мы не будем делать его полупрозрачным, поэтому ползунок, отвечающий за альфа-канал, трогать не нужно. Кроме того, в главе 8 упоминались настройки, управляющие отбрасыванием теней; создавать тени в данном случае или нет, выбираете вы сами. Для небольших предметов, предназначенных для сбора, лично я предпочитаю их отключать.

Создайте новый сценарий с именем `CollectibleItem`.

Листинг 9.8. Сценарий, удаляющий элемент при контакте с персонажем

```
using UnityEngine;
using System.Collections;

public class CollectibleItem : MonoBehaviour {
  [SerializeField] private string itemName;

  void OnTriggerEnter(Collider other) {
    Debug.Log("Item collected: " + itemName);
    Destroy(this.gameObject);
  }
}
```

Укажите имя этого элемента на панели Inspector.

Это очень короткий и простой сценарий. Присвойте элементу значение `name`, чтобы в сцену можно было поместить несколько элементов. Метод `OnTriggerEnter()` вызывает исчезновение сферы. При этом на консоль выводится отладочное сообщение, которое мы впоследствии заменим кодом.

ВНИМАНИЕ Метод `Destroy()` должен вызываться для параметра `this.gameObject`, а не `this`! Не путайте эти вещи; ключевое слово `this` ссылается только на компонент сценария, в то время как выражение `this.gameObject` ссылается на объект, к которому присоединен сценарий.

Добавленная в код переменная должна появиться на панели Inspector. Введите в соответствующее поле имя, чтобы идентифицировать элемент; я выбрал для первого элемента имя energy (энергия). Затем создайте несколько копий элемента и поменяйте их имена; я использовал имена ore (металл), health (здоровье) и key (ключ) (точность написания имен крайне важна, так как впоследствии они появятся в коде). Заодно назначьте каждому элементу собственный материал, чтобы они имели разные цвета. Я выбрал голубой цвет для элемента energy, темно-серый — для элемента ore, розовый — для элемента health и желтый — для элемента key.

СОВЕТ Вместо имен, как в нашем случае, в более сложных играх элементам зачастую присваиваются идентификаторы, используемые для поиска дальнейшей информации. Например, элементу может быть назначен идентификатор 301, который совпадает с определенным отображаемым именем, изображением, описанием и т. п.

Превратим все элементы в шаблоны экземпляров, чтобы упростить их добавление на игровой уровень. В главе 3 вы узнали, что эта операция осуществляется перетаскиванием объектов со вкладки Hierarchy на вкладку Project.

ПРИМЕЧАНИЕ После этой операции имя объекта на панели Hierarchy выделяется синим цветом — это признак объектов, которые являются экземплярами шаблона. Щелкните правой кнопкой мыши на имени такого объекта и выберите команду Select Prefab для выделения шаблона, экземпляром которого является объект.

Перетаскивая шаблоны, расположите элементы на открытых участках уровня; для тестирования создавайте по несколько экземпляров одного элемента. Запустите игру и «соберите» элементы. Все выглядит здорово, но пока ничего не происходит. Давайте проследим за собранными элементами; для этого нужно создать структуру кода, которая будет отвечать за управление инвентарем.

9.3. Управление данными инвентаризации и состоянием игры

Мы запрограммировали процедуру сбора элементов, и теперь для игрового инвентаря требуется фоновый диспетчер данных (напоминающий веб-форму с ячейками). Мы напишем код, напоминающий лежащую в основе многих веб-приложений MVC-архитектуру. Ее преимущество состоит в разделении отображаемых на экране объектов и хранении данных, что упрощает эксперименты и итеративную разработку. Даже в случае сложноорганизованных данных и (или) отображения объектов редактирование части приложения не влияет на его остальные фрагменты.

Существуют различные варианты таких структур, потому что в каждой игре свои требования к управлению данными.

Например, в ролевой игре требования к управлению данными высоки, поэтому целесообразно подумать о MVC-архитектуре. Головоломки не имеют подобных требований, соответственно, построение для них сложной несвязанной структуры диспетчеров данных — напрасная трата времени и сил. Состояние такой игры легко отслеживается объектами-контроллерами (более того, вы делали это в предыдущих главах).

Но сейчас перед нами стоит задача по управлению инвентарем игрока. Давайте запрограммируем соответствующую структуру кода.

9.3.1. Диспетчер игрока и диспетчер инвентаря

Управление данными следует разбить на отдельные хорошо продуманные модули, у каждого из которых своя зона ответственности. Для управления состоянием игрока (например, его здоровьем) напишем модуль `PlayerManager`, а для управления инвентарем — модуль `InventoryManager`. Диспетчеры данных будут вести себя как компонент Модель в MVC-архитектуре; роль Контроллера в большинстве сцен играет невидимый объект (в данном случае он не нужен, но вспомните объект `SceneController` из предыдущих глав), остальная часть сцены выступает аналогом Представления.

У нас будет расположенный на более высоком уровне «*диспетчер диспетчеров*» для слежения за отдельными модулями. Кроме хранения списка диспетчеров этот

высокоуровневый диспетчер будет контролировать их жизненный цикл, в частности, занимаясь их инициализацией в начальный момент. Доступ остальных игровых сценариев к этим централизованным модулям осуществляется через главный диспетчер. В частности, для связи с нужным модулем остальной код может использовать ряд статических свойств главного диспетчера.

ДОСТУП К ЦЕНТРАЛИЗОВАННЫМ МОДУЛЯМ СОВМЕСТНОГО ИСПОЛЬЗОВАНИЯ

Для решения задачи по соединению частей программы с централизованными модулями общего доступа создано множество паттернов проектирования. К примеру, паттерн одиночки (singleton) упоминался еще в первой книге «Банды четырех».

Однако многие разработчики программного обеспечения предпочитают пользоваться такими альтернативами, как локатор служб (service locator) и инъекция зависимостей (dependency injection). В своем коде я попытался найти компромисс между простотой статических переменных и гибкостью локатора служб.

Представленный вашему вниманию проект объединяет простой в применении код с возможностью замены модулей. Например, запрашивая `InventoryManager` с использованием паттерна одиночки, мы всегда будем ссылаться на один и тот же класс, плотно связывая код с этим классом; в то же время запрос инвентаря через локатор служб позволяет возвращать как `InventoryManager`, так и `DifferentInventoryManager`. Иногда удобно иметь возможность переключения между слегка различающимися версиями одного модуля (например, при развертывании игры на разных платформах).

Чтобы главный диспетчер одним и тем же способом ссылался на другие модули, эти модули должны унаследовать свойства от общей основы. Мы обеспечим соблюдение этого условия с помощью интерфейса; многие языки программирования (в том числе C#) позволяют задавать своего рода сценарий, которому должны следовать остальные классы. Как `PlayerManager`, так и `InventoryManager` будет реализовывать общий интерфейс (он будет называться `IGameManager`), и основной объект `Managers` сможет воспринимать их как тип `IGameManager`. Иллюстрация этой схемы показана на рис. 9.5.

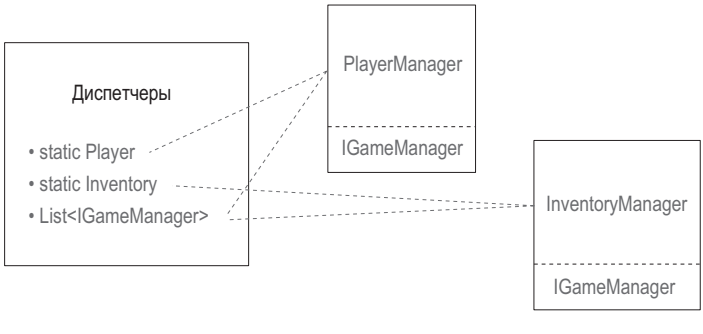


Рис. 9.5. Схема связи модулей

Описанный вариант архитектуры состоит из невидимых модулей, существующих в фоновом режиме, но для запуска этого кода Unity все равно требуются связанные с объектами сцены сценарии. И, как в случае с привязанными к сцене контроллерами

в предыдущих проектах, мы создадим пустой объект `GameObject`, который будет связывать эти данные с диспетчерами.

9.3.2. Программирование диспетчеров

Я объяснил все концепции, которыми мы будем пользоваться, и пришло время написать код. Первым делом создайте сценарий `IGameManager` и скопируйте в него код следующего листинга.

Листинг 9.9. Базовый интерфейс, который будут реализовывать диспетчеры данных

```
public interface IGameManager {
   ManagerStatus status {get;}  <-- Перечисление, которое нужно определить.

   void Startup();
}
```

Этот файл практически не содержит кода. Обратите внимание, что там отсутствует даже наследование от класса `MonoBehaviour`. Сам по себе интерфейс ничего не делает и существует только затем, чтобы задавать структуру для других классов. Этот интерфейс объявляет одно свойство (переменную, у которой есть функция чтения) и один метод. Как свойство, так и метод должны присутствовать у реализующего данный интерфейс класса. Свойство `status` сообщает остальной части кода, завершил ли модуль инициализацию. Метод `Startup()` предназначен для обработки процесса инициализации диспетчера, поэтому именно в нем выполняются все связанные с этим процессом задания, а функция задает состояние диспетчера.

Обратите внимание, что свойство относится к типу `ManagerStatus`; это пока еще не написанное перечисление. Поэтому создайте сценарий `ManagerStatus.cs`.

Листинг 9.10. ManagerStatus: возможные состояния для IGameManager

```
public enum ManagerStatus {
  Shutdown,
  Initializing,
  Started
}
```

Это еще один файл, в котором почти нет кода. На этот раз мы перечисляем возможные состояния диспетчеров, принудительно устанавливая для свойства `status` одно из указанных значений.

Теперь, когда у нас есть интерфейс `IGameManager`, его можно реализовывать в других сценариях. Листинги 9.11 и 9.12 содержат код сценариев `PlayerManager` и `InventoryManager`.

Листинг 9.11. InventoryManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;   <-- Импортируем новые структуры данных (использовавшиеся в листинге 9.14).

public class InventoryManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}  <-- Свойство читается откуда угодно, но задается только в этом сценарии.
```

```
  public void Startup() {
    Debug.Log("Inventory manager starting...");
    status = ManagerStatus.Started;
  }
}
```

Сюда идут все задачи запуска с долгим временем выполнения.

Для задач с долгим временем выполнения используем состояние 'Initializing'.

Листинг 9.12. PlayerManager

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

public class PlayerManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  public int health {get; private set;}
  public int maxHealth {get; private set;}

  public void Startup() {
    Debug.Log("Player manager starting...");

    health = 50;
    maxHealth = 100;

    status = ManagerStatus.Started;
  }

  public void ChangeHealth(int value) {
    health += value;
    if (health > maxHealth) {
      health = maxHealth;
    } else if (health < 0) {
      health = 0;
    }

    Debug.Log("Health: " + health + "/" + maxHealth);
  }
}
```

Наследуем класс и реализуем интерфейс.

Эти значения допускают инициализацию сохраненными данными.

Другие сценарии не могут напрямую задавать переменную health, но могут вызывать эту функцию.

Пока что `InventoryManager` — это оболочка, заполнением которой мы займемся позднее, в то время как `PlayerManager` уже обладает всей необходимой функциональностью. Оба диспетчера наследуют от класса `MonoBehaviour` и реализуют интерфейс `IGameManager`. Это означает, что оба диспетчера получили всю функциональность `MonoBehaviour`, но при этом обязаны реализовывать структуру, заданную интерфейсом `IGameManager`. Эта структура состоит из одного свойства и одного метода, которые и определяют наши диспетчеры.

Свойство `status` доступно для чтения из любого места (функция чтения общедоступна), но задается только внутри сценария (функция записи закрыта). Кроме того, оба диспетчера определяют метод `Startup()`. Их инициализация сразу же завершается (`InventoryManager` пока ничего не делает, в то время как `PlayerManager` задает пару

значений), поэтому состоянию присваивается значение `Started`. Но у модулей данных могут быть длительные задания как часть процесса инициализации (например, загрузка сохраненных данных). Загрузкой этих заданий займется метод `Startup()`, а состоянию диспетчера будет присвоено значение `Initializing`. После завершения заданий измените состояние на `Started`.

Замечательно: мы наконец готовы связать все вместе внутри одного главного диспетчера! Создайте сценарий `Managers` и введите в него код следующего листинга.

Листинг 9.13. Диспетчер для диспетчеров

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

[RequireComponent(typeof(PlayerManager))]
[RequireComponent(typeof(InventoryManager))]

public class Managers : MonoBehaviour {
  public static PlayerManager Player {get; private set;}
  public static InventoryManager Inventory {get; private set;}

  private List<IGameManager> _startSequence;

  void Awake() {
    Player = GetComponent<PlayerManager>();
    Inventory = GetComponent<InventoryManager>();

    _startSequence = new List<IGameManager>();
    _startSequence.Add(Player);
    _startSequence.Add(Inventory);

    StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    foreach (IGameManager manager in _startSequence) {
      manager.Startup();
    }

    yield return null;

    int numModules = _startSequence.Count;
    int numReady = 0;

    while (numReady < numModules) {
      int lastReady = numReady;
      numReady = 0;

      foreach (IGameManager manager in _startSequence) {
        if (manager.status == ManagerStatus.Started) {
          numReady++;
        }
      }
```

Убеждаемся, что различные диспетчеры существуют.

Статические свойства, которыми остальной код пользуется для доступа к диспетчерам.

Список диспетчеров, который просматривается в цикле во время стартовой последовательности.

Асинхронно загружаем стартовую последовательность.

Продолжаем цикл, пока не начнут работать все диспетчеры.

```
      if (numReady > lastReady)
        Debug.Log("Progress: " + numReady + "/" + numModules);
      yield return null;  ◄——  Остановка на один кадр перед следующей проверкой.
    }

    Debug.Log("All managers started up");
  }
}
```

Важнее всего в этом паттерне статические свойства на самом верху. Именно они разрешают другим сценариям доступ к различным модулям через такой синтаксис, как `Managers.Player` или `Managers.Inventory`. Изначально эти свойства пусты, но они заполняются сразу после запуска кода в методе `Awake()`.

СОВЕТ Подобно методам `Start()` и `Update()`, метод `Awake()` автоматически предоставляется классом `MonoBehaviour`. Как и метод `Start()`, он однократно запускается в начале выполнения кода. Но в принятой в Unity последовательности выполнения кода метод `Awake()` располагается перед методом `Start()`, отвечая за связанные с инициализацией задания, которые должны запускаться перед любыми другими модулями.

Кроме того, метод `Awake()` выводит последовательность запуска, а затем загружает сопрограмму для запуска всех диспетчеров. Для их добавления он создает объект `List` и прибегает к методу `List.Add()`.

ОПРЕДЕЛЕНИЕ Объект `List` представляет собой такую структуру данных из языка C#, как коллекция. Списки похожи на массивы: они хранят последовательные наборы элементов определенного типа. Но размер списка можно поменять в процессе работы, в то время как у массивов этот параметр не редактируется.

Все диспетчеры реализуют интерфейс `IGameManager`, поэтому код может перечислить их как принадлежащие к данному типу и вызвать определенный в каждом из них метод `Startup()`. Стартовая последовательность запускается как сопрограмма, то есть асинхронно с другими частями игры (например, анимированным индикатором выполнения на заставке).

Функция запуска сначала в цикле просматривает список диспетчеров и для каждого из них вызывает метод `Startup()`. Затем она входит в цикл, проверяющий, загрузился ли каждый из диспетчеров, и ожидает завершения этого процесса. В конце она уведомляет нас о загрузке и завершает свою работу.

СОВЕТ Инициализация написанных нами диспетчеров настолько проста, что происходит без задержки. В общем случае такая стартовая последовательность на базе сопрограмм позволяет элегантно обрабатывать асинхронные задания запуска с долгим временем выполнения, такие как загрузка сохраненных данных.

Итак, структура кода готова. Вернитесь в редактор Unity и создайте пустой объект `GameObject`; как обычно, расположите его в точке с координатами `0,0,0` и присвойте ему значимое имя, например `Game Managers`. Свяжите с этим объектом сценарии `Managers`, `PlayerManager` и `InventoryManager`.

В процессе воспроизведения игры ничего не изменится, но на консоли появится ряд сообщений, информирующих о выполнении стартовой последовательности. Диспетчеры запускаются корректно, значит, пришло время заняться программированием диспетчера инвентаря.

9.3.3. Хранение инвентаря в объекте collection: списки и словари

Собранные персонажем элементы можно сохранить в виде объекта `List`. Следующий листинг добавляет такой список в диспетчер инвентаря.

Листинг 9.14. Добавление элементов в InventoryManager

```
...
private List<string> _items;

public void Startup() {
  Debug.Log("Inventory manager starting...");

  _items = new List<string>();   <-- Инициализируем пустой список элементов.

  status = ManagerStatus.Started;
}

private void DisplayItems() {   <-- Выводим на консоль сообщение о текущем инвентаре.
  string itemDisplay = "Items: ";
  foreach (string item in _items) {
    itemDisplay += item + " ";
  }
  Debug.Log(itemDisplay);
}

public void AddItem(string name) {   <-- Другие сценарии не могут напрямую управлять списком элементов, но могут вызывать этот метод.
  _items.Add(name);

  DisplayItems();
}
...
```

В сценарии `InventoryManager` появилось два важных дополнения. Во-первых, добавлен объект `List` для хранения элементов. Во-вторых, появился открытый метод `AddItem()`, который может вызываться другим кодом. Он добавляет элемент в список и выводит список на консоль. Давайте слегка скорректируем сценарий `CollectibleItem`, добавив в него вызов нового метода `AddItem()`.

Листинг 9.15. Применение нового диспетчера InventoryManager в сценарии CollectibleItem

```
...
void OnTriggerEnter(Collider other) {
  Managers.Inventory.AddItem(itemName);
  Destroy(this.gameObject);
}
...
```

Теперь в процессе сбора элементов персонажем список инвентаря в выводимом на консоль сообщении будет увеличиваться. Но всплывет недостаток такой структуры данных, как список: в нем появляются все собранные элементы одного типа (например, второй элемент Health), как показано на рис. 9.6, хотя разумнее было бы подсчитывать количество таких элементов. Разумеется, можно представить вариант игры, в котором каждый элемент нужно отслеживать отдельно, но, как правило, подсчитывается количество копий одного элемента. Устранить этот недостаток вполне реально с сохранением списка, но более естественно и эффективно прибегнуть к другой структуре данных, которая называется словарем.

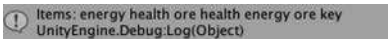


Рис. 9.6. Консольное сообщение с многократным перечислением однотипных элементов

ОПРЕДЕЛЕНИЕ Структура данных `Dictionary` также взята из языка C#. Доступ к записям словаря осуществляется по идентификатору (или ключу), а не по их местоположению. Словарь напоминает хеш-таблицу, но более гибок, так как ключами могут выступать данные практически любого типа (например, можно попросить «верните записи для этого объекта `GameObject`»).

Замените в сценарии `InventoryManager` структуру данных `List` структурой `Dictionary`. Для этого вставьте вместо кода из листинга 9.14 следующий код.

Листинг 9.16. Словарь элементов в InventoryManager

```
...
private Dictionary<string, int> _items;

public void Startup() {
  Debug.Log("Inventory manager starting...");

  _items = new Dictionary<string, int>();

  status = ManagerStatus.Started;
}

private void DisplayItems() {
  string itemDisplay = "Items: ";
  foreach (KeyValuePair<string, int> item in _items) {
    itemDisplay += item.Key + "(" + item.Value + ") ";
  }
  Debug.Log(itemDisplay);
}

public void AddItem(string name) {
  if (_items.ContainsKey(name)) {
    _items[name] += 1;
  } else {
    _items[name] = 1;
  }

  DisplayItems();
}
...
```

При объявлении словаря указываются два типа: тип ключа и тип значения.

Перед вводом новых данных проверяем, не существует ли уже такой записи.

Этот код не сильно отличается от предыдущего, но различия имеют большое значение. Если раньше вы не сталкивались с такими структурами данных, как `Dictionary`, обратите внимание, что при ее объявлении указывается два типа. Если структура `List` объявляется только с одним типом (типом сохраняемых в нее значений), то `Dictionary` объявляет как тип ключей (то есть идентификаторов, по которым будет вестись поиск), так и тип значений.

В методе `AddItem()` появился дополнительный алгоритм. Раньше каждый элемент просто добавлялся в список. Теперь же первым делом требуется проверить, нет ли в словаре такого элемента; именно этим занимается метод `ContainsKey()`. Если перед нами новый элемент, счет начинается с единицы, если такой элемент уже существует, на единицу увеличивается сохраненное значение. Запустите воспроизведение сцены, и вы убедитесь, что теперь в сообщениях о сборе инвентаря элементы одного типа объединяются, как показано на рис. 9.7.

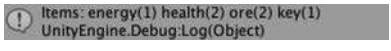


Рис. 9.7. Консольное сообщение с объединенными элементами одного типа

Наконец-то мы систематизировали в инвентаре игрока собранные элементы! Скорее всего, вам кажется, что для такой простой задачи кода слишком много. Если бы это была единственная стоявшая перед нами цель, решение действительно можно было бы назвать излишне сложным. Но смысл всей тщательно разрабатываемой архитектуры кода в сохранении всех данных в отдельных гибких модулях. Полезность данного паттерна становится очевидной по мере усложнения игры. Давайте напишем пользовательский интерфейс, и вы увидите, что управлять отдельными частями кода станет намного проще.

9.4. Интерфейс для использования и подготовки элементов

В игре коллекцией собранных элементов можно пользоваться по-разному, но в любом случае потребуется какой-то пользовательский интерфейс, чтобы игрок мог видеть свой инвентарь. После этого мы добавим к этому интерфейсу интерактивность, дав игрокам возможность пользоваться элементами. Мы рассмотрим конкретные примеры (подготовку ключа и применение пакетов здоровья), после чего вы самостоятельно сможете адаптировать этот код для элементов других типов.

ПРИМЕЧАНИЕ В главе 7 мы говорили о том, что в Unity есть как более старый GUI непосредственного режима, так и более новая система UI на основе спрайтов. В этой главе мы поработаем с GUI непосредственного режима, поскольку этот интерфейс быстрее в реализации и требует меньшего количества настроек, что хорошо подходит для упражнений. При этом UI на основе спрайтов имеет более завершенный вид, поэтому этот интерфейс выбирают для итогового варианта игры.

9.4.1. Отображение элементов инвентаря в UI

Чтобы пользовательский интерфейс начал отображать информацию об элементах инвентаря, в сценарий `InventoryManager` нужно добавить пару методов. В настоящее время список элементов закрыт и доступен только в соответствующем диспетчере.

Чтобы сделать его видимым, потребуются открытые методы доступа к данным. Добавьте их из следующего листинга.

Листинг 9.17. Добавление в сценарий InventoryManager методов доступа к данным

```
...
public List<string> GetItemList() {  <-- Возвращаем список всех ключей словаря.
  List<string> list = new List<string>(_items.Keys);
  return list;
}

public int GetItemCount(string name) {  <-- Возвращаем количество указанных элементов в инвентаре.
  if (_items.ContainsKey(name)) {
    return _items[name];
  }
  return 0;
}
...
```

Метод `GetItemList()` возвращает список элементов инвентаря. У вас может появиться вопрос, зачем чуть раньше мы потратили столько сил на преобразование списка элементов инвентаря в другой формат. Дело в том, что теперь элемент каждого типа появляется в списке всего один раз. Например, при наличии в инвентаре двух пакетов здоровья имя «health» в списке будет всего одно, ведь этот список был создан из ключей элементов словаря, а не из отдельных элементов.

Метод `GetItemCount()` возвращает количество элементов указанного типа в инвентаре. Например, вызов `GetItemCount("health")` представляет собой вопрос «Сколько пакетов здоровья содержит инвентарь?». Это дает возможность отображать в UI не только все элементы, но и их количество.

Благодаря наличию этих методов в сценарии `InventoryManager`, мы можем создать пользовательский интерфейс. Давайте отобразим все элементы в виде горизонтальной панели в верхней части экрана. Для каждого элемента будет свой значок, поэтому нужно импортировать в проект соответствующие изображения. Ресурсы из папки Resources Unity обрабатывает особым способом.

СОВЕТ Ресурсы из папки Resources доступны для загрузки методом `Resources.Load()`. В остальных случаях для вставки их в сцену пользуйтесь редактором Unity.

Рисунок 9.8 демонстрирует четыре значка вместе с адресом папки, в которой они хранятся. Создайте папку Resources, а затем внутри нее — папку Icons.

Рис. 9.8. Графические ресурсы для значков оборудования в папке Resources

Создайте пустой объект `GameObject` с именем Controller и свяжите с ним новый сценарий `BasicUI`.

Листинг 9.18. Сценарий BasicUI, отображающий инвентарь

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

public class BasicUI : MonoBehaviour {
  void OnGUI() {
    int posX = 10;
    int posY = 10;
    int width = 100;
    int height = 30;
    int buffer = 10;

    List<string> itemList = Managers.Inventory.GetItemList();
    if (itemList.Count == 0) {
      GUI.Box(new Rect(posX, posY, width, height), "No Items");
    }
    foreach (string item in itemList) {
      int count = Managers.Inventory.GetItemCount(item);
      Texture2D image = Resources.Load<Texture2D>("Icons/"+item);
      GUI.Box(new Rect(posX, posY, width, height), new GUIContent("
                      (" + count + ")", image));
      posX += width+buffer;
    }
  }
}
```

Отображаем сообщение, информирующее об отсутствии инвентаря.

Метод, загружающий ресурсы из папки Resources.

При каждом прохождении цикла сдвигаемся в сторону.

Этот листинг отображает собранные элементы в виде горизонтальной полосы, показывая заодно их количество (рис. 9.9). Как упоминалось в главе 3, все представители класса `MonoBehaviour` автоматически отвечают на метод `OnGUI()`, после визуализации сцены запускаемый в каждом кадре.

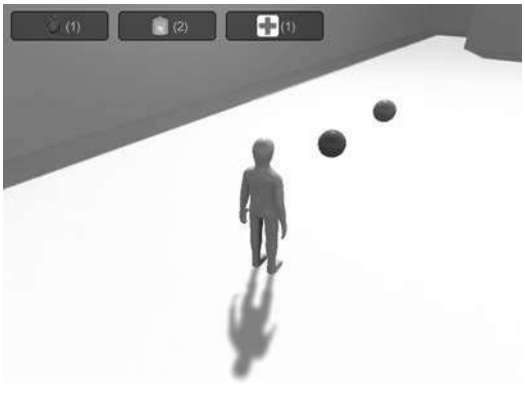


Рис. 9.9. Отображение инвентаря средствами UI

Внутри метода `OnGUI()` первым делом определяется набор значений для положений элементов пользовательского интерфейса. В процессе циклического просмотра всех элементов эти значения увеличиваются, что позволяет расположить их в ряд. Рисуемые

элементы интерфейса — в данном случае GUI.Box — представляют собой неинтерактивные поля, в которых отображаются текст и значки.

Метод Resources.Load() загружает ресурсы из папки Resources. Это удобный способ загрузки ресурсов по именам; обратите внимание, что имя каждого элемента передается как параметр. Нужно указать тип загружаемых ресурсов, в противном случае метод вернет обобщенные объекты.

Теперь пользовательский интерфейс отображает собранный инвентарь. Пришло время им воспользоваться.

9.4.2. Ключ для запертых дверей

Рассмотрим пример применения инвентаря, чтобы потом аналогичным способом вы смогли использовать элементы произвольного типа. Начнем с подготовки ключа, который будет открывать двери.

В настоящее время сценарий DeviceTrigger инвентаря не видит (ведь он написан до появления связанного с инвентарем кода). Коррективы, которые следует внести в сценарий, показаны в следующем листинге.

Листинг 9.19. Запрос ключа в сценарии DeviceTrigger

```
...
public bool requireKey;

void OnTriggerEnter(Collider other) {
  if (requireKey && Managers.Inventory.equippedItem != "key") {
    return;
  }
...
```

Как видите, для поиска ключа оказалось достаточно новой открытой переменной и условного оператора. Логическая переменная requireKey появляется на панели Inspector в виде флажка, что позволяет требовать ключ только от определенных триггеров. Условие в начале метода OnTriggerEnter() — есть ли в диспетчере InventoryManager готовый к работе ключ; соответственно, нужно добавить в сценарий InventoryManager код из следующего листинга.

Листинг 9.20. Код подготовки инвентаря для сценария InventoryManager

```
...
public string equippedItem {get; private set;}
...
public bool EquipItem(string name) {
  if (_items.ContainsKey(name) && equippedItem != name) {
    equippedItem = name;
    Debug.Log("Equipped " + name);
    return true;
  }

  equippedItem = null;
  Debug.Log("Unequipped");
  return false;
}
...
```

Проверяем, что в инвентаре есть указанный элемент, но он еще не подготовлен.

В верхнюю часть сценария добавлено свойство `equippedItem`, которое проверяется другим фрагментом кода. Затем мы добавили открытый метод `EquipItem()`, позволяющий другому коду менять подготовленные элементы. Этот код готовит элемент, если он еще не готов к применению, или кладет его обратно в инвентарь, если пользоваться им уже можно.

Напоследок, чтобы дать игроку возможность подготовить элементы к использованию, указанная функциональность добавляется в UI. Это делается в следующем фрагменте кода.

Листинг 9.21. Добавление к сценарию BasicUI функции подготовки элементов

```
...
    foreach (string item in itemList) {

       int count = Managers.Inventory.GetItemCount(item);
       GUI.Box(new Rect(posX, posY, width, height), item + "(" + count + ")");
       posX += width+buffer;
    }

    string equipped = Managers.Inventory.equippedItem;
    if (equipped != null) {
      posX = Screen.width - (width+buffer);
      Texture2D image = Resources.Load("Icons/"+equipped) as Texture2D;
      GUI.Box(new Rect(posX, posY, width, height),
              new GUIContent("Equipped", image));
    }


    posX = 10;
    posY += height+buffer;

    foreach (string item in itemList) {
      if (GUI.Button(new Rect(posX, posY, width, height), "Equip "+item)) {
        Managers.Inventory.EquipItem(item);
      }
      posX += width+buffer;
    }
  }
}
```

Курсивом выделен ранее присутствовавший в сценарии код. Тут он фигурирует в качестве точки отсчета.

Отображаем подготовленный элемент.

Просматриваем все элементы в цикле для создания кнопок.

Запускаем вложенный код при щелчке на кнопке.

Для отображения подготовленного элемента снова применяется метод `GUI.Box()`. Но он не интерактивен, поэтому ряд кнопок Equip мы нарисуем методом `GUI.Button()`. Он создает кнопку, которая при щелчке выполняет код внутри оператора `if`.

Весь нужный код написан. Выделите триггер в сцене и установите в разделе Device Trigger флажок Require Key, после чего запустите игру. Попробуйте войти в зону триггера без ключа — ничего не произойдет. Поднимите ключ и щелкните на кнопке, чтобы подготовить его к использованию; после этого при входе в зону триггера дверь начнет открываться.

Из спортивного интереса поместите ключ в точку с координатами `-11`, `5`, `-14` и подумайте, как его можно достать. А пока рассмотрим процесс использования пакетов здоровья.

9.4.3. Восстановление здоровья персонажа

Еще инвентарь часто применяется для восстановления здоровья персонажа. Внесем в код два изменения: добавим новый метод в сценарий `InventoryManager` и новую кнопку в пользовательский интерфейс (это листинги 9.22 и 9.23 соответственно).

Листинг 9.22. Новый метод в сценарии InventoryManager

```
...
public bool ConsumeItem(string name) {
  if (_items.ContainsKey(name)) {  <-- Проверяем, есть ли в инвентаре нужный элемент.
    _items[name]--;
    if (_items[name] == 0) {  <-- Удаляем запись, если количество становится равным 0.
      _items.Remove(name);
    }
  } else {  <-- Отвечаем, что в инвентаре нет нужного элемента.
     Debug.Log("cannot consume " + name);
     return false;
  }

  DisplayItems();
  return true;
}
...
```

Листинг 9.23. Добавление кнопки здоровья в базовый UI

```
...
    foreach (string item in itemList) {
      if (GUI.Button(new Rect(posX, posY, width, height), "Equip "+item)) {
        Managers.Inventory.EquipItem(item);
      }

      if (item == "health") {  <-- Начало нового кода.
        if (GUI.Button(new Rect(posX, posY + height+buffer, width, height),
           "Use Health")) {  <-- Запускаем вложенный код при щелчке на кнопке.
          Managers.Inventory.ConsumeItem("health");
          Managers.Player.ChangeHealth(25);
        }
      }

      posX += width+buffer;
    }
  }
}
```

Курсивом выделен ранее присутствовавший в сценарии код. Тут он фигурирует в качестве точки отсчета.

Новый метод `ConsumeItem()`, по сути, диаметрально противоположен методу `AddItem()`; он проверяет наличие указанного элемента в инвентаре и уменьшает значение, если таковой не обнаружен. Умеет он обрабатывать и ситуацию, когда количество элементов уменьшается до нуля. Код UI вызывает этот новый метод работы с инвентарем, который, в свою очередь, вызывает метод `ChangeHealth()`, с самого начала присутствовавший в сценарии `PlayerManager`.

Когда вы соберете несколько пакетов здоровья, а затем ими воспользуетесь, на консоли появятся соответствующие сообщения. Вот вы и узнали еще один вариант использования элементов инвентаря!

Заключение

- Управлять устройствами можно как с клавиатуры, так и с помощью триггеров столкновения.
- Объекты, для которых включена модель физической среды, могут реагировать на силу столкновения и зоны триггеров.
- Сложными игровыми состояниями управляют через специальные объекты общего доступа.
- Коллекции объектов можно систематизировать с помощью таких структур данных, как `List` или `Dictionary`.
- Отслеживание подготовленных состояний элементов инвентаря позволяет влиять на другие части игры.

Часть III
УВЕРЕННЫЙ ФИНИШ

Вы уже много знаете о Unity. Умеете программировать элементы управления персонажем, создавать блуждающих по сцене противников и добавлять в игру интерактивные устройства. Вы способны пользоваться при создании игр как двумерной, так и трехмерной графикой! Этого *почти* достаточно для разработки полноценной игры. Почти, но не совсем. Нужно изучить еще ряд вещей, таких как добавление звука; кроме того, вы должны уметь собирать воедино разрозненные фрагменты, с которыми мы до этого работали.

Вы уже на финишной прямой, осталось всего четыре главы!

10 Подключение к интернету

- ✓ Графика для неба, сгенерированная с помощью скайбокса.
- ✓ Скачивание данных с помощью веб-запросов в сопрограммах.
- ✓ Анализ распространенных форматов данных, таких как XML и JSON.
- ✓ Отображение графики, скачанной из интернета.
- ✓ Отправка данных на веб-сервер.

В этой главе мы поговорим о способах отправки и получения данных по Сети. Все созданные нами проекты в разных игровых жанрах находятся исключительно на вашей машине. Но все большую важность приобретают подключение к интернету и обмен данными. Многие игры существуют исключительно в Сети с постоянным подключением к сообществу игроков. Для их обозначения используется аббревиатура MMO (Massively Multiplayer Online — массовая многопользовательская онлайновая игра). Шире всего известны так называемые массовые многопользовательские ролевые онлайновые игры (Massively Multiplayer Online Role-Playing Games — MMORPG). Даже если для игрового процесса постоянное подключение не требуется, современные видеоигры оснащены такими функциями, как, например, отправка сведений о набранных очках в глобальный список лучших результатов. В Unity эта функциональность тоже есть, и мы подробно ее рассмотрим.

В Unity поддерживаются разные варианты передачи данных по Сети, подходящие под разные задачи. Но в этой главе в основном будет рассматриваться наиболее общий вид сетевых взаимодействий — отправка HTTP-запросов.

ЧТО ТАКОЕ HTTP-ЗАПРОСЫ?

Думаю, большинство читателей уже знают, что представляют собой HTTP-запросы, но на всякий случай дам небольшое пояснение: HTTP — это протокол для отправки запросов к веб-серверам и получения оттуда ответов. Например, когда при щелчке на ссылке браузер (клиент) отправля-

ет запрос на определенный адрес, а расположенный по этому адресу сервер возвращает новую страницу. Такие запросы выполняются различными методами: например, существуют методы GET и POST, используемые для получения и отправки данных соответственно.

Популярность HTTP-запросов объясняется их надежностью. При проектировании самих запросов и предназначенной для их обработки инфраструктуры закладывается устойчивость к сбоям и способы исправления различных ошибок.

Вспомните, как работают современные одностраничные веб-приложения (в отличие от старых добрых веб-страниц, генерируемых на стороне сервера). Разрабатываемая в Unity онлайновая игра на базе HTTP-запросов будет, по сути, толстым клиентом, взаимодействующим с сервером по технологии Ajax. Хорошее знание принципов старой школы порой заставляет опытных разработчиков делать неверные шаги. У видеоигр требования к производительности зачастую куда строже, чем у веб-приложений, и именно разница в требованиях может влиять на проектные решения.

ВНИМАНИЕ В веб-приложениях и видеоиграх время может ощущаться по-разному. При обновлении скачиваемой страницы половина секунды, как правило, вообще ничего не значит, в то время как в разгар активных действий в игре подобная задержка способна испортить все удовольствие. Понятие *быстро* целиком и полностью зависит от ситуации.

Онлайновые игры, как правило, подключаются к специальному серверу; но в процессе обучения мы будем подключаться к открытым источникам данных, включая источники прогнозов погоды и доступных для скачивания изображений. Последний раздел этой главы потребует настройки собственного веб-сервера; его можно просто пропустить, хотя я предлагаю вам максимально простой способ настройки на примере программного обеспечения с открытым исходным кодом.

В процессе работы над упражнениями этой главы вам много раз придется прибегать к HTTP-запросам, чтобы понять принцип их работы в Unity. Вот план наших действий:

1. Настройка натурной сцены (создание неба, реагирующего на данные метеонаблюдений).
2. Написание кода для запроса прогнозов погоды из интернета.
3. Анализ ответа и корректировка сцены в соответствии с полученной информацией.
4. Скачивание и отображение графики.
5. Отправка данных на ваш собственный сервер (в нашем случае это будут записи журнала о погоде за прошедшее время).

В качестве основы для проекта этой главы можно взять любую игру. К проекту будут добавляться новые сценарии, никак не затрагивающие существующий код. Я воспользовался демонстрационным роликом из главы 2, в котором можно от первого лица наблюдать за изменениями неба. Непосредственной связи с игровым процессом текущий проект не имеет, но умение передавать данные по Сети пригодится для большинства создаваемых вами игр (например, генерировать противников можно на базе ответов сервера).

Поэтому приступим к работе!

10.1. Натурная сцена

Мы будем получать из интернета информацию о погоде, и первым делом следует настроить сцену для наблюдения за погодными условиями. Сложнее всего создать небо, поэтому начнем с более простой задачи. Назначим текстуру камня геометрии игрового уровня. Как и в главе 4, я скачал изображения для стен и пола с сайта www.textures.com. Напоминаю, что скачанные изображения нужно отредактировать таким образом, чтобы их размер выражался степенями двойки, например 256 × 256. Затем импортируйте их в проект Unity, создайте материалы и назначьте им изображения (то есть перетащите каждое из них на соответствующую ячейку в редакторе параметров материала). Перетащите материалы на стены и пол в сцене и увеличьте параметр Tiling (попробуйте значение 8 или 9 в одном или в обоих направлениях), чтобы изображение перестало ужасным образом растягиваться на всю поверхность.

Теперь можно заняться небом.

10.1.1. Генерация неба с помощью скайбокса

Первым делом импортируйте графику для скайбокса, как вы делали в главе 4. Скачать подходящие изображения можно с сайта www.93i.de. На этот раз добавим к набору TropicalSunnyDay набор DarkStormy (в этом проекте предполагается более сложный вариант неба). Эти текстуры нужно перетащить на вкладку Project и (как объяснялось в главе 4) выбирать в раскрывающемся списке Wrap Mode вариант Clamp.

Пришло время создать материал для скайбокса. В верхней части настроек откройте меню Shader со списком доступных вариантов шейдеров. Наведите указатель мыши на раздел Skybox и выберите в дополнительном меню вариант 6-Sided. У материала появятся шесть ячеек для текстур (вместо одной ячейки Albedo, характерной для стандартного шейдера).

Перетащите изображения типа SunnyDay на ячейки для текстур нового материала. Окончания имен текстур соответствуют именам ячеек, на которые их нужно поместить (top, front и т. д.). После этого новый материал можно использовать в качестве скайбокса для сцены.

Перетащите новый материал на поле Skybox в окне Lighting (окно открывается командой Lighting из меню Window). Нажмите кнопку Play, и вы увидите примерно такую же сцену, как на рис. 10.1.

Рис. 10.1. Сцена с фоновыми изображениями неба

Натурная сцена готова! Скайбокс позволяет легко превратить окружающее персонаж пространство в естественную сцену. Однако встроенный шейдер для скайбокса имеет один серьезный недостаток — изображения нельзя менять, и небо получается статичным. Поэтому создадим свой вариант шейдера.

10.1.2. Программно управляемое небо

Изображения из набора TropicalSunnyDay великолепно имитируют солнечный день, но что делать, когда требуется переход от солнечной погоды к облачности? Очевидно, что нужен второй набор снимков (демонстрирующих облачное небо), а значит, и новый шейдер для скайбокса.

В главе 4 вы узнали, что шейдер реализуется короткой программой, указывающей способ визуализации изображения. Это означает возможность программировать новые варианты раскрасок, что и требуется в данном случае. Мы создадим новый шейдер, в котором используется два варианта изображений для скайбокса и переход между ними. Впрочем, нужный вариант уже существует в коллекции сценариев Unity-сообщества по адресу http://wiki.unity3d.com/index.php?title=SkyboxBlended.

Создайте новый сценарий шейдера. Для этого вызовите меню с командой Create, как при создании новых сценариев на языке C#, но на этот раз выберите в нем вариант Shader. Присвойте новому ресурсу имя SkyboxBlended и дважды щелкните на нем, чтобы получить доступ к редактированию сценария. Скопируйте код с вики-страницы и вставьте его в наш сценарий. Команда `Shader "Skybox/Blended"` в верхней строке заставляет Unity добавить новый шейдер в категорию Skybox (в которой находится и обычный скайбокс).

ПРИМЕЧАНИЕ Углубляться в детали программирования раскрасок мы не будем. Это сложная тема, выходящая за рамки темы данной книги. Если вы захотите самостоятельно ее изучить, начать можно отсюда: https://docs.unity3d.com/ru/current/Manual/ShadersOverview.html.

Теперь для материала можно выбрать шейдер Skybox Blended. Появятся 12 полей текстуры, два набора по шесть изображений. Первым шести полям, как и раньше, следует назначить изображения из набора TropicalSunnyDay, а для остальных текстур используйте набор DarkStormy.

В верхней части настроек нового шейдера есть ползунок Blend. Он контролирует отображаемую долю изображений; когда вы перетаскиваете его из крайнего левого положения в крайнее правое, скайбокс переходит от солнечной к облачной погоде. Для тестирования несколько раз запустите воспроизведение сцены при разных положениях ползунка. Разумеется, во время игры никто так делать не будет, поэтому давайте напишем код, меняющий вид неба.

Создайте пустой объект и присвойте ему имя `Controller`. Создайте сценарий с именем `WeatherController`. Перетащите сценарий на пустой объект и введите в него код листинга 10.1.

Ключевой для этого кода метод `SetFloat()` находится почти в конце листинга. До этого момента все более-менее знакомо, а с этим методом вы пока не сталкивались. Он задает для материала числовое значение. *Что это за значение*, определяет первый

параметр метода. В рассматриваемом случае у материала есть свойство Blend (обратите внимание: в коде имена свойств материала начинаются с нижнего подчеркивания).

Листинг 10.1. Сценарий WeatherController, осуществляющий переход от солнечной погоды к облачности

```
using UnityEngine;
using System.Collections;

public class WeatherController : MonoBehaviour {
  [SerializeField] private Material sky;
  [SerializeField] private Light sun;

  private float _fullIntensity;

  private float _cloudValue = 0f;

  void Start() {
    _fullIntensity = sun.intensity;
  }

  void Update() {
    SetOvercast(_cloudValue);
    _cloudValue += .005f;
  }

  private void SetOvercast(float value) {
    sky.SetFloat("_Blend", value);
    sun.intensity = _fullIntensity - (_fullIntensity * value);
  }
}
```

Ссылка может быть не только на объекты сцены, но и на материал на вкладке Project.

Начальная интенсивность рассматривается как «полная».

Увеличиваем значение в каждом кадре для обеспечения непрерывности перехода.

Корректируем значение Blend материала и интенсивность света.

В остальной части кода мы определили несколько переменных, в том числе для материала и света. В случае материала потребовалась ссылка на только что созданный составной материал для скайбокса, а как обстоят дела с освещением? Ведь при переходе от солнечной к облачной погоде сцена становится темнее, поэтому при увеличении параметра Blend мы уменьшаем интенсивность освещения. В качестве основного осветителя сцены выступает направленный источник света; выделите его для доступа к параметрам на панели Inspector.

ПРИМЕЧАНИЕ Усовершенствованная система освещения в Unity (которая называется Enlighten) учитывает скайбокс для достижения реалистичных результатов. Но этот подход корректно работает только в случае статичного скайбокса, поэтому эту систему лучше «заморозить». В нижней части окна диалога Lighting снимите флажок Auto generate; после этого обновление будет происходить только после щелчка на кнопке. Установите ползунок Blend для скайбокса в центральное положение и щелкните на кнопке рядом с флажком Auto, чтобы вручную создать карту освещенности (информацию об освещенности, которая сохраняется в новой папке, где имя совпадает с именем сцены).

В начале работы сценарий инициализирует интенсивность света. Это начальное значение он сохраняет, считая его за «полную» интенсивность. Позднее эта полная интенсивность появится в сценарии, когда потребуется приглушить свет.

Затем код увеличивает значение каждого кадра на постоянную величину и использует его для корректировки неба, а именно: в каждом кадре вызывает метод `SetOvercast()`, который инкапсулирует все внесенные в сцену изменения. Назначение метода `SetFloat()` объяснялось выше, последняя же строка меняет интенсивность света.

Запустите воспроизведение сцены. Должен получиться результат с рис. 10.2: через пару секунд солнечный день в сцене превратится в пасмурный и облачный.

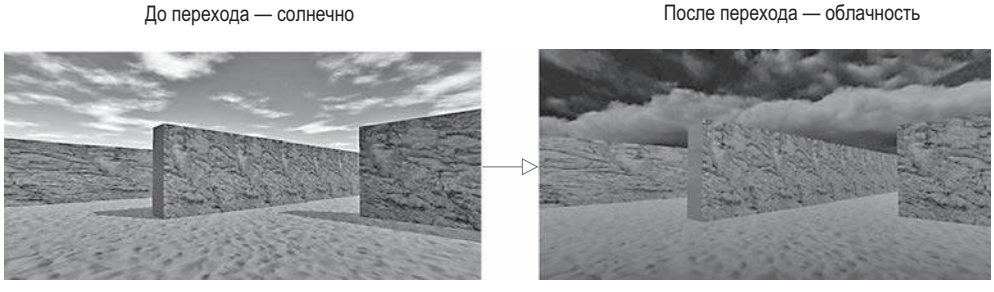


Рис. 10.2. До и после: переход сцены от солнца к облачности

ВНИМАНИЕ Здесь мы сталкиваемся с неожиданной особенностью Unity. В материале фиксируются результаты изменения параметра Blend. После остановки игры объекты сцены возвращаются в исходное состояние, но ресурсы, добавленные непосредственно со вкладки Project (как материал для нашего скайбокса), меняются насовсем. Такое происходит только в редакторе Unity (изменения не переносятся из игры в игру после развертывания за пределами редактора) и может привести к крайне неприятным ошибкам, если забыть о данной особенности.

Очень интересно наблюдать за переходом от солнца к облачности. Но это только подготовка к решению основной задачи: синхронизации погоды в игре с реальными погодными условиями. Для этого нужно научиться скачивать сводку погоды из интернета.

10.2. Скачивание метеорологических данных

Напишем для нашей сцены код, скачивающий из интернета информацию о погоде и корректирующий вид неба в соответствии с полученными данными. Это хороший пример задачи на выборку данных с помощью HTTP-запросов. Я обычно получаю метеоданные с сайта OpenWeatherMap. Мы будем пользоваться их прикладным программным интерфейсом по адресу https://openweathermap.org/api. Этот API обеспечивает доступ к службе через команды кода, а не через графический интерфейс.

ОПРЕДЕЛЕНИЕ *Веб-службой*, или *веб-API*, называется подключенный к интернету сервер, возвращающий данные в ответ на запросы. С технической точки зрения разницы между API и веб-узлом не существует; веб-узел — это служба, возвращающая данные веб-странице, а браузеры интерпретируют HTML-данные, превращая их в видимые документы.

ПРИМЕЧАНИЕ Веб-службы часто требуют регистрации, даже для бесплатного обслуживания. Например, если вы перейдете на страницу API для OpenWeatherMap, там есть инструкции по получению ключа API, значение, которое вы вставляете в запросы.

Код, который вам предстоит написать, структурирован вокруг уже знакомой по главе 9 архитектуры диспетчеров. На этот раз центральный диспетчер диспетчеров будет инициализировать класс `WeatherManager`, который отвечает за получение и сохранение метеорологических данных, но для этого ему требуется связь с интернетом.

Поэтому в коде появится вспомогательный класс `NetworkService`, который позаботится о подключении к интернету и HTTP-запросах. Класс `WeatherManager` будет заставлять класс `NetworkService` отправлять запросы и возвращать полученные ответы. Принцип функционирования такой структуры кода показан на рис. 10.3.

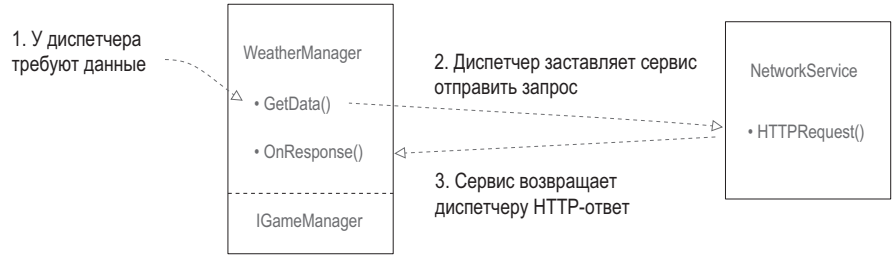


Рис. 10.3. Структура кода, передающего данные по сети

Очевидно, что этот механизм будет работать, только если у класса `WeatherManager` есть доступ к объекту `NetworkService`. Для решения этой задачи создадим объект в сценарии `Managers` и в момент инициализации различных диспетчеров будем вставлять в них объект `NetworkService`. В результате ссылку на этот объект получит не только диспетчер `WeatherManager`, но и все созданные после этого диспетчеры.

Воспроизводить архитектуру из главы 9 начнем с копирования сценариев `ManagerStatus` и `IGameManager`. Я напомню, что `IGameManager` — это интерфейс, который должны реализовывать все диспетчеры, а `ManagerStatus` — это перечисление, которым пользуется `IGameManager`. Внесем в сценарий `IGameManager` изменения, связанные с появлением класса `NetworkService`. Создайте сценарий `NetworkService` (удалите строку `:MonoBehaviour` и пока оставьте его пустым) и внесите в сценарий `IGameManager` следующие изменения.

Листинг 10.2. Добавление в сценарий IGameManager класса NetworkService

```
public interface IGameManager {
  ManagerStatus status {get;}

  void Startup(NetworkService service);
}
```

Метод Startup теперь принимает один параметр — вставленный объект.

Для реализации нашего слегка отредактированного интерфейса создадим сценарий `WeatherManager`. Добавьте в него следующий код:

Листинг 10.3. Начальный вариант сценария WeatherManager

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

```
public class WeatherManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  // Сюда добавляется величина облачности (сценарий 10.8)
  private NetworkService _network;

  public void Startup(NetworkService service) {
    Debug.Log("Weather manager starting...");

    _network = service;   <-- Сохраняем вставленный объект NetworkService.

    status = ManagerStatus.Started;
  }
}
```

Начальный вариант сценария `WeatherManager` ничего не делает. Это минимальное количество кода, необходимое сценарию `IGameManager` для реализации класса: здесь объявляется свойство интерфейса `status` и выполняется функция `Startup()`. В следующих разделах этот пустой каркас заполнится. А пока скопируйте сценарий `Managers` из главы 9 и заставьте его запускать сценарий `WeatherManager`.

Листинг 10.4. Сценарий Managers.cs, инициализирующий сценарий WeatherManager

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

[RequireComponent(typeof(WeatherManager))]   <-- Вместо диспетчеров персонажа и инвентаря требуем новый диспетчер погоды.

public class Managers : MonoBehaviour {
  public static WeatherManager Weather {get; private set;}

  private List<IGameManager> _startSequence;

  void Awake() {
    Weather = GetComponent<WeatherManager>();

    _startSequence = new List<IGameManager>();
    _startSequence.Add(Weather);

    StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    NetworkService network = new NetworkService();   <-- Создаем экземпляры класса NetworkService для вставки во все диспетчеры.

    foreach (IGameManager manager in _startSequence) {
      manager.Startup(network);   <-- Во время загрузки диспетчеров передаем им сетевую службу.
    }

    yield return null;

    int numModules = _startSequence.Count;
    int numReady = 0;
```

```
    while (numReady < numModules) {
      int lastReady = numReady;
      numReady = 0;

      foreach (IGameManager manager in _startSequence) {
        if (manager.status == ManagerStatus.Started) {
          numReady++;
        }
      }

      if (numReady > lastReady)
        Debug.Log("Progress: " + numReady + "/" + numModules);

      yield return null;
    }

    Debug.Log("All managers started up");
  }
}
```

Это все, что нужно для архитектуры, заданной сценарием `Managers`. Как и в предыдущей главе, создайте в сцене пустой объект, который будет играть роль диспетчера, и присоедините к нему сценарии `Managers` и `WeatherManager`. После корректной настройки он начнет выводить на консоль сообщения о загрузке, чем его функция пока и ограничится.

Подготовительный этап, состоящий в основном из копирования, завершен! Можно писать код.

10.2.1. Запрос веб-данных через сопрограмму

Сценарий `NetworkService` пока пуст. Напишем для него код реализации HTTP-запросов. Для этого нужен класс `UnityWebRequest`, обеспечивающий взаимодействие с интернетом в Unity. Создание экземпляра объекта-запроса с использованием URL-адреса приводит к отправке запроса по этому адресу.

Ждать завершения запроса классу `UnityWebRequest` позволяют сопрограммы. В первый раз вы столкнулись с ними в главе 3, где они применялись для временной остановки кода. Напомню определение: сопрограммами называются специальные функции, которые запускаются в фоновом режиме основной программы, в цикле выполняют код и возвращают результат в программу. Вместе с методом `StartCoroutine()` мы использовали ключевое слово `yield`, которое заставляло сопрограмму на время остановиться, вернуть управление программе, а в следующем кадре снова начать свою работу.

В главе 3 сопрограмма возвращала метод `WaitForSeconds()`, что останавливало метод на указанное количество секунд. При отправке запроса работа метода будет прерываться до завершения этого запроса. Работа программы в данном случае напоминает асинхронные Ajax-вызовы в веб-приложениях: сначала посылается запрос, потом продолжается выполнение остальной части программы, а через некоторое время приходит ответ.

Это была теория, а теперь давайте писать код

Реализуем все вышеописанное. Откройте сценарий `NetworkService` и замените шаблон по умолчанию содержимым следующего листинга.

Листинг 10.5. Выполнение HTTP-запросов в сценарии NetworkService

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;
using System;

public class NetworkService {
  private const string xmlApi =
"http://api.openweathermap.org/data/2.5/weather?q=Chicago,
     us&mode=xml&APPID=<your api key>";

  private IEnumerator CallAPI(string url, Action<string> callback) {
    using (UnityWebRequest request = UnityWebRequest.Get(url)) {

      yield return request.Send();

      if (request.isNetworkError) {
         Debug.LogError("network problem: " + request.error);
      } else if (request.responseCode !=
            (long)System.Net.HttpStatusCode.OK) {
         Debug.LogError("response error: " + request.responseCode);
      } else {
         callback(request.downloadHandler.text);
      }
    }
  }

  public IEnumerator GetWeatherXML(Action<string> callback) {
    return CallAPI(xmlApi, callback);
  }
}
```

URL-адрес для отправки запроса.

Создаем объект UnityWebRequest в режиме GET.

Пауза в процессе скачивания.

Проверяем ответ на наличие ошибок.

Делегат можно вызвать так же, как и исходную функцию.

Каскад ключевых слов yield в вызывающих друг друга методах сопрограммы.

ВНИМАНИЕ Тип `Action` находится в пространстве имен `System`; обратите внимание на дополнительный оператор `statement` в верхней части сценария. Не забывайте об этом при написании собственных сценариев!

Надеюсь, вы помните особенности данного проекта: диспетчер `WeatherManager` заставляет сценарий `NetworkService` извлекать нужные данные. Код, который мы написали, пока не запускается; позднее он будет вызываться сценарием `WeatherManager`. Объяснять смысл этого листинга я начну снизу.

Вложенные методы сопрограммы

Добавленный в сопрограмму метод `GetWeatherXML()` заставляет сценарий `NetworkService` создавать HTTP-запрос. Обратите внимание, что в качестве типа возвращаемого

значения указан тип `IEnumerator`. Именно он должен объявляться как тип возвращаемого значения методами, добавленными в сопрограмму.

Отсутствие ключевого слова `yield` в методе `GetWeatherXML()` на первый взгляд может показаться странным. Ведь именно оно останавливает выполнение сопрограммы, а значит, его наличие предполагается по умолчанию. Но у нас есть набор вставленных друг в друга методов. Если первый метод сопрограммы вызывает какой-то другой метод, который останавливается, выполнив только часть своего кода, остановка и возобновление работы сопрограммы будут осуществляться внутри второго метода. То есть в нашем случае ключевое слово `yield` в методе `CallAPI()` прерывает сопрограмму, начавшуюся в методе `GetWeatherXML()`. Этот принцип работы показан на рис. 10.4.

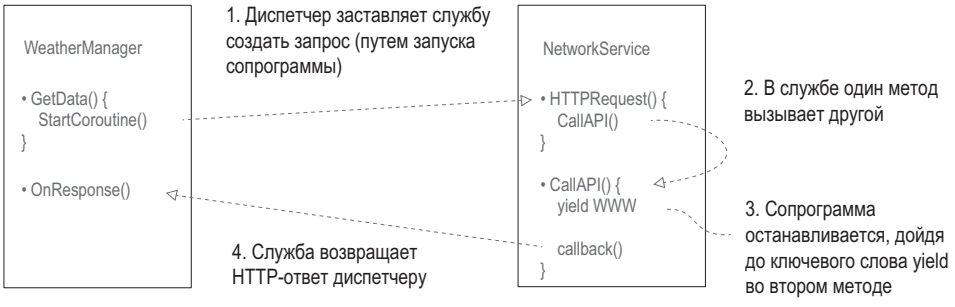


Рис. 10.4. Схема работы сопрограммы, управляющей передачей данных по сети

Затем мы видим непонятный параметр `callback` типа `Action`.

Принцип работы обратного вызова

В начале сопрограммы вызывается метод с параметром `callback`, принадлежащим типу `Action`. Но что это за тип?

ОПРЕДЕЛЕНИЕ Тип `Action` является делегатом (в C# существуют разные подходы к работе с делегатами, но я изложу самый простой). Делегат — это ссылка на какой-то другой метод/функцию. Он позволяет сохранять функцию (или, точнее, указатель на нее) в переменной и передавать эту переменную в качестве параметра другой функции.

Если вы раньше не сталкивались с концепцией делегатов, представьте, что они позволяют передавать функции точно так же, как числа или строки, и вызывать их позже. Без делегатов был бы возможен только прямой вызов функций. Делегаты позволяют информировать код о других методах, которые можно вызвать позже. Такое поведение требуется во многих случаях, особенно при реализации функций обратного вызова.

ОПРЕДЕЛЕНИЕ *Обратным вызовом* (callback) называется вызов функции, используемой для обмена данными с вызывающим объектом. Объект A может сообщить объекту B об одном из своих методов. Позднее объект B может вызвать этот метод для обмена данными с объектом A.

Например, в рассматриваемом случае обратный вызов позволил, дождавшись завершения HTTP-запроса, переслать обратно полученный ответ. Метод `CallAPI()` сначала

отправляет HTTP-запрос, затем останавливается, ожидая завершения этого запроса, и, наконец, с помощью метода `callback()` возвращает полученный ответ.

Обратите внимание на синтаксис `<>`, используемый с ключевым словом `Action`; указанный в угловых скобках тип требуется параметрам для соответствия делегату. Другими словами, функция, на которую указывает делегат `Action`, должна иметь параметры указанного типа. В данном случае параметром является единственная строка, поэтому у метода обратного вызова должна быть примерно такая сигнатура:

```
ИмяМетода (строковое значение)
```

Возможно, вы лучше поймете концепцию обратного вызова на примере из листинга 10.6. Надеюсь, увидев этот дополнительный код, вы благодаря моим объяснениям без труда поймете, что там происходит.

Остальная часть листинга 10.5 не должна вызвать затруднений. Объект `request` создается внутри оператора `using`, поэтому он удаляется из памяти после завершения работы с ним. Условный оператор проверяет HTTP-ответ на наличие ошибок. Возможны ошибки двух типов: сбой запроса из-за проблем с интернет-подключением и некорректность возвращенных данных. Константа `const` объявляется с URL-адресом, по которому код будет отправлять запросы. (Если хотите, можете поменять этот URL-адрес и получать сведения о погоде из другого места.)

Использование кода передачи данных по сети

Этот сценарий включает в себя код `NetworkService`. Воспользуемся им в сценарии `WeatherManager`.

Листинг 10.6. Добавление в сценарий WeatherManager кода для работы с NetworkService

```
...
public void Startup(NetworkService service) {
  Debug.Log("Weather manager starting...");

  _network = service;
  StartCoroutine(_network.GetWeatherXML(OnXMLDataLoaded));
  status = ManagerStatus.Initializing;
}

public void OnXMLDataLoaded(string data) {
  Debug.Log(data);

  status = ManagerStatus.Started;
}
...
```

Начинаем загрузку данных из интернета.

Меняем состояние со Started на Initializing.

Метод обратного вызова сразу после загрузки данных.

В код этого диспетчера внесены три основных изменения: запуск сопрограммы для скачивания данных из интернета, задание другого состояния загрузки и определение метода обратного вызова для получения ответа.

С началом сопрограммы все просто. Все сложные аспекты работы с сопрограммами реализованы в сценарии `NetworkService`, поэтому сейчас остается только вызвать метод `StartCoroutine()`. Потом вы меняете состояние загрузки, так как на самом

деле инициализация диспетчера не завершена; сначала он должен получить данные из интернета.

ВНИМАНИЕ Методы передачи данных по Сети всегда нужно начинать с функции `StartCoroutine()`; обычный вызов в их случае неприменим. Об этом легко забыть, потому что создание объектов-запросов за пределами сопрограммы не приводит к ошибкам компиляции.

Вызов метода `StartCoroutine()` должен сопровождаться активацией. То есть нужно добавить скобки, а не просто указать имя функции. В нашем случае в качестве одного из параметров методу сопрограммы требуется функция обратного вызова, которую следует задать. Для обратного вызова воспользуемся функцией `OnXMLDataLoaded()`; обратите внимание, что ее параметр относится к типу `string`, что совпадает с объявлением `Action<string>` в сценарии `NetworkService`. Пока что от функции обратного вызова многого не требуется — она просто проверяет корректность полученных данных и выводит их на консоль. После этого последняя строка функции меняет состояние загрузки диспетчера, уведомляя о том, что теперь он полностью загружен.

Нажмите кнопку Play. При хорошей связи с интернетом данные на консоли появятся быстро. Это всего лишь длинная строка, но отформатированная особым образом, что дает возможность воспользоваться содержащейся в ней информацией.

10.2.2. Разбор XML-кода

Существующие в виде длинных строк данные обычно состоят из отдельных битов информации. Эти биты информации извлекаются методом синтаксического разбора.

ОПРЕДЕЛЕНИЕ *Синтаксическим разбором* (parsing) называется анализ фрагментов кода с последующим его разделением на отдельные фрагменты данных.

Строка для синтаксического разбора должна быть отформатирована таким образом, чтобы у вас (точнее, у кода-анализатора) была возможность идентифицировать отдельные фрагменты. Существует пара стандартных форматов передачи данных через интернет; из них чаще всего применяется *XML*.

ОПРЕДЕЛЕНИЕ *Язык XML* (Extensible Markup Language — расширяемый язык разметки) задает правила структурированного кодирования документов аналогично HTML-страницам.

К счастью, Unity (точнее, встроенная в это приложение среда разработки Mono) предлагает функциональность для анализа XML-кода. Запрошенный нами прогноз погоды имеет формат XML, поэтому добавим в сценарий `WeatherManager` код, который будет анализировать ответ и извлекать из него информацию об облачности. Код вы найдете в интернете, он довольно длинный, но нас интересует только фрагмент, содержащий, к примеру, такие данные, как `<clouds value="40" name="scattered clouds"/>`.

Мы не только добавим код, анализирующий XML, но и воспользуемся системой сообщений, знакомой по главе 7. Ведь следует уведомить сцену о скачанных и проанализированных данных. Создайте сценарий с именем `Messenger` и вставьте в него код со страницы http://wiki.unity3d.com/index.php/CSharpMessenger_Extended.

После этого нужно создать сценарий `GameEvent` (его код приведен в следующем листинге). Как объяснялось в главе 7, эта система сообщений дает замечательный несвязанный способ передавать сообщения о событиях остальной программе.

Листинг 10.7. Сценарий GameEvent

```
public static class GameEvent {
  public const string WEATHER_UPDATED = "WEATHER_UPDATED";
}
```

Теперь, когда у нас есть система сообщений, скорректируйте сценарий `WeatherManager` в соответствии со следующим листингом.

Листинг 10.8. Синтаксический разбор XML-кода в сценарии WeatherManager

```
...
using System;
using System.Xml;
...
public float cloudValue {get; private set;}
...
public void OnXMLDataLoaded(string data) {
  XmlDocument doc = new XmlDocument();
  doc.LoadXml(data);
  XmlNode root = doc.DocumentElement;

  XmlNode node = root.SelectSingleNode("clouds");
  string value = node.Attributes["value"].Value;
  cloudValue = Convert.ToInt32(value) / 100f;
  Debug.Log("Value: " + cloudValue);

  Messenger.Broadcast(GameEvent.WEATHER_UPDATED);

  status = ManagerStatus.Started;
}
...
```

Обязательно добавьте нужные операторы using.

Облачность редактируется внутренне, в остальных местах это свойство предназначено только для чтения.

Разбиваем XML-код на структуру с возможностью поиска.

Извлекаем из данных один узел.

Преобразуем значение в число типа float в диапазоне от 0 до 1.

Рассылаем сообщение, информирующее остальные сценарии.

Как видите, самые важные изменения появились внутри метода `OnXMLDataLoaded()`. Раньше он всего лишь выводил данные на консоль, позволяя убедиться, что они переданы корректно. Теперь же в методе появились команды, анализирующие XML-код.

Мы начинаем с создания нового пустого XML-документа, который послужит контейнером для разбираемой XML-структуры. Следующая строка разбивает строку данных, превращая ее в структуру из XML-документа. Мы начинаем с корня XML-дерева, чтобы дальше можно было выполнять поиск по этому дереву.

На данном этапе в XML-структуре уже можно искать узлы, чтобы извлечь отдельные биты информации. Нас интересует только узел `<clouds>`. Первым делом ищем его в XML-документе, затем извлекаем из него атрибут `value`. Этот атрибут задает облачность в виде целого числа в диапазоне от 0 до 100, но для последующей корректировки состояния сцены требуется число типа `float` в диапазоне от 0 до 1. Преобразование в данном случае осуществляется простой математической операцией.

Наконец, после извлечения из полных данных сведений об облачности мы рассылаем сообщение об обновлении погодных данных. Пока этого сообщения никто

не слышит, но издатель не обязан ничего знать о подписчиках (собственно, в этом и состоит смысл несвязанной системы рассылки сообщений). Позднее мы добавим в сцену подписчика.

Замечательно! Код, анализирующий XML-данные, готов! Но до того, как мы начнем менять вид сцены на базе полученного значения, я хотел бы рассмотреть еще один.

10.2.3. Анализ текста в формате JSON

Перед следующим этапом проекта познакомимся с альтернативным форматом передачи данных. Этот формат распространен так же, как XML, и называется *JSON*.

ОПРЕДЕЛЕНИЕ *JSON* (JavaScript Object Notation) — это текстовый формат обмена данными на основе JavaScript, назначение которого аналогично XML. Собственно, формат JSON проектировался как облегченная альтернатива XML. Хотя JSON-синтаксис является производным от JavaScript, на конкретный язык этот формат не ориентирован и легко используется с самыми разными языками программирования.

В среде разработки Mono анализатор формата JSON отсутствует. Впрочем, есть ряд доступных для скачивания анализаторов, например MiniJSON (https://gist.github.com/darktable/1411710) и SimpleJSON (http://wiki.unity3d.com/index.php/SimpleJSON).

Мы будем пользоваться анализатором MiniJSON. Создайте сценарий с именем `MiniJSON` и вставьте туда код анализатора. Теперь этой библиотекой можно пользоваться для анализа данных в формате JSON. Данные в формате XML мы получали от сервиса OpenWeatherMap, но иногда часть данных оттуда поступает в формате JSON. Для получения таких данных отредактируйте сценарий `NetworkService` в соответствии со следующим листингом.

Листинг 10.9. Заставляем сценарий NetworkService запрашивать JSON-, а не XML-данные

```
...
private const string jsonApi =  ◄——  URL-адрес на этот раз выглядит чуть иначе.
"http://api.openweathermap.org/data/2.5/weather?q=Chicago,us&APPID=<your api key>";
...
public IEnumerator GetWeatherJSON(Action<string> callback) {
  return CallAPI(jsonApi, callback);
}
...
```

Фактически перед нами уже знакомый код для скачивания XML-данных, просто в нем фигурирует другой URL-адрес. Этот запрос возвращает те же самые данные, что и раньше, но в другом формате. И теперь мы уже будем искать фрагмент кода `"clouds":{"all":40}`.

На этот раз не потребуется писать много кода. Ведь код запросов выделен в аккуратные отдельные функции, поэтому ничего сложного в добавлении последующих HTTP-запросов нет. Видите, как здорово! Давайте отредактируем сценарий `WeatherManager`, заставив его запрашивать данные в формате JSON, а не XML.

Листинг 10.10. Заставляем сценарий WeatherManager запрашивать JSON-данные

```
...
using MiniJSON;
...
public void Startup(NetworkService service) {
  Debug.Log("Weather manager starting...");

  _network = service;
  StartCoroutine(_network.GetWeatherJSON(OnJSONDataLoaded));

  status = ManagerStatus.Initializing;
}
...
public void OnJSONDataLoaded(string data) {
  Dictionary<string, object> dict;
  dict = Json.Deserialize(data) as Dictionary<string,object>;

  Dictionary<string, object> clouds = (Dictionary<string,object>)
              dict["clouds"];
  cloudValue = (long)clouds["all"] / 100f;
  Debug.Log("Value: " + cloudValue);

  Messenger.Broadcast(GameEvent.WEATHER_UPDATED);

  status = ManagerStatus.Started;
}
...
```

Обязательно добавим оператор using.

Измененный сетевой запрос.

Вместо пользовательского XML-контейнера анализируется содержимое словаря.

Синтаксис изменился, но функциональность кода осталась прежней.

Как видите, код для работы с данными в формате JSON напоминает код для работы с данными в формате XML. Основное отличие состоит в том, что анализатор JSON-данных работает со стандартным словарем, а не с нестандартным XML-контейнером. В коде встречается процедура *десериализации* — возможно, вы не знаете, что это такое.

ОПРЕДЕЛЕНИЕ *Десериализация* (deserialization) представляет собой процесс, обратный сериализации. То есть данные преобразуются в форму, доступную для передачи и сохранения, например в JSON-строку.

В сценарии изменился только синтаксис, а все операции остались неизменными. Из фрагмента данных извлекается значение (по какой-то причине на этот раз оно содержится в переменной с именем `all`, но это странности API) и делается то же самое несложное математическое преобразование к значению типа `float` в диапазоне от 0 до 1.

Настало время применить полученное значение к нашей сцене.

10.2.4. Изменение вида сцены на базе метеорологических данных

Независимо от способа форматирования данных, как только из полученного ответа извлечено значение облачности, его можно использовать в методе `SetOvercast()` сценария `WeatherController`. Строка данных в формате XML или JSON в итоге превращается в набор слов и чисел. Одно из этих чисел метод `SetOvercast()` принимает в качестве

параметра. В разделе 10.1.2 использовалось число, увеличивавшееся в каждом кадре, но также легко мы можем вставить в код значение, возвращенное сервисом с сайта прогнозов погоды.

Вот как выглядит сценарий `WeatherController` после внесения в него изменений.

Листинг 10.11. Сценарий WeatherController, реагирующий на скачанные метеорологические данные

```
using UnityEngine;
using System.Collections;

public class WeatherController : MonoBehaviour {
  [SerializeField] private Material sky;
  [SerializeField] private Light sun;

  private float _fullIntensity;

  void Awake() {   <-- Добавляем/удаляем подписчиков на событие.
    Messenger.AddListener(GameEvent.WEATHER_UPDATED, OnWeatherUpdated);
  }
  void OnDestroy() {
    Messenger.RemoveListener(GameEvent.WEATHER_UPDATED, OnWeatherUpdated);
  }

  void Start() {
    _fullIntensity = sun.intensity;
  }

  private void OnWeatherUpdated() {
    SetOvercast(Managers.Weather.cloudValue);   <-- Используем величину облачности из сценария WeatherManager.
  }

  private void SetOvercast(float value) {
    sky.SetFloat("_Blend", value);
    sun.intensity = _fullIntensity - (_fullIntensity * value);
  }
}
```

Обратите внимание, что изменения сводятся не только к дополнениям; удалены несколько фрагментов тестового кода, а именно локальное значение облачности, увеличивавшееся в каждом кадре. Оно больше не требуется, так как мы будем пользоваться значением из сценария `WeatherManager`.

Подписчики добавляются и удаляются в методах `Awake()` и `OnDestroy()` — это методы класса `MonoBehaviour`, вызываемые при активации или удалении объекта. Каждый такой подписчик принадлежит к системе рассылки сообщений. При получении сообщения он вызывает метод `OnWeatherUpdated()`, получающий значение облачности из сценария `WeatherManager` и, в свою очередь, вызывающий метод `SetOvercast()`, использующий это значение. В результате вид сцены контролируется скачанными из интернета метеорологическими данными.

Запустите воспроизведение сцены и убедитесь, что небо обновляется в соответствии с информацией об облачности из прогноза погоды. Скорее всего, запрос погоды займет некоторое время; в настоящей игре до завершения загрузки неба можно спрятать сцену за экраном загрузки.

ДРУГИЕ СПОСОБЫ ПЕРЕДАЧИ ДАННЫХ ПО СЕТИ В ИГРАХ

При всей устойчивости к сбоям и надежности HTTP-запросов задержка ответа — слишком существенный недостаток для большинства игр. Поэтому они хороши для относительно медленной передачи сообщений на сервер (например, значений перемещения в играх со сменой хода или данных о набранных очках), но для таких игр, как сетевой шутер от первого лица, требуется другой подход к передаче данных по Сети.

Существуют различные технологии взаимодействия, а также приемы, позволяющие скомпенсировать запаздывание. К примеру, в Unity для многопользовательских игр применяется высокоуровневый API, построенный поверх более низкого транспортного уровня.

Передовые технологии создания сетевых игр — сложная тема, рассмотрение которой выходит за рамки данной книги. Вы можете самостоятельно заняться ее изучением, начав со страницы https://docs.unity3d.com/ru/current/Manual/UNetOverview.html.

Теперь, когда вы научились получать из интернета числовые и строковые данные, рассмотрим способы работы с изображениями.

10.3. Рекламный щит

Ответы от веб-сервиса почти всегда приходят в виде текстовых строк формата XML или JSON, но по Сети передаются другие данные. Кроме текстовой информации чаще всего запрашиваются изображения. Объект `UnityWebRequest` применяется в числе прочего и для их скачивания.

Рассмотрим вариант решения такой задачи на примере моделирования рекламного щита, украшенного скачанным из интернета изображением. Нужен код, решающий две задачи: скачивание изображения и назначение этого изображения рекламному щиту. Третьей задачей станет сохранение изображения для его показа на разных рекламных щитах.

10.3.1. Загрузка изображений из интернета

Первым делом напишем код загрузки изображения. Для тестирования скачаем свободно распространяемые фотографии пейзажей. Пример такой фотографии вы видите на рис. 10.5. Загруженное изображение не появится на рекламном щите; сценарий отображения картинки я покажу вам в следующем разделе, но до этого давайте напишем код для получения графического файла.

Структура кода для скачивания данных и графики практически одинакова. За загрузку изображений отвечает помещенный в отдельный модуль очередной диспетчер (он называется `ImagesManager`). Напомню, что за подключение к интернету и отправку HTTP-запросов отвечает сценарий `NetworkService`, к которому и будет обращаться новый диспетчер. Первым делом отредактируем код сценария `NetworkService`. Следующий листинг добавляет туда возможность скачивания изображений.

Рис. 10.5. Озеро Морейн в национальном парке Банф, Канада

Листинг 10.12. Скачивание изображений в сценарии NetworkService

```
...
private const string webImage =
"http://upload.wikimedia.org/wikipedia/commons/c/c5/Moraine_Lake_17092005.jpg";
...
public IEnumerator DownloadImage(Action<Texture2D> callback) {
  UnityWebRequest request = UnityWebRequestTexture.GetTexture(webImage);
  yield return request.Send();
  callback(DownloadHandlerTexture.GetContent(request));
}
...
```

Эта константа с другими URL-адресами помещается в верхнюю часть сценария.

Вместо строки этот обратный вызов принимает объекты Texture2D.

Получаем загруженное изображение с помощью служебной программы DownloadHandler.

Код скачивания изображений выглядит практически идентично коду скачивания данных. Эти варианты кода различаются типом метода обратного вызова; обратите внимание, что на этот раз обратный вызов работает не со строками, а с объектами типа `Texture2D`. Все дело в типе возвращаемого ответа: раньше скачивалась строка данных, теперь — изображение. Теперь давайте создадим новый диспетчер. Создайте сценарий `ImagesManager` и скопируйте туда код следующего листинга.

Листинг 10.13. Диспетчер ImagesManager, запрашивающий и сохраняющий изображения

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

public class ImagesManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  private NetworkService _network;

  private Texture2D _webImage;
```

Переменная для хранения скачанного изображения.

```
  public void Startup(NetworkService service) {
    Debug.Log("Images manager starting...");

    _network = service;

    status = ManagerStatus.Started;
  }

  public void GetWebImage(Action<Texture2D> callback) {
    if (_webImage == null) {  <-- Проверяем, нет ли уже сохраненного изображения.
      StartCoroutine(_network.DownloadImage(callback));
    }
    else {
      callback(_webImage);  <-- При наличии сохраненного изображения сразу
    }                           активируется обратный вызов (без скачивания).
  }
}
```

Самая интересная часть этого кода — метод `GetWebImage()`; все остальное в сценарии представляет собой стандартные свойства и методы, реализующие интерфейс диспетчера. Именно метод `GetWebImage()` возвращает (через функцию обратного вызова) скачанное из Сети изображение. Первым делом мы проверяем, не содержит ли переменная `_webImage` уже сохраненного изображения: в случае отрицательного результата проверки активируется сетевой вызов для скачивания. Если же в переменной `_webImage` уже есть сохраненное изображение, метод `GetWebImage()` возвращает его (чтобы не скачивать заново).

ПРИМЕЧАНИЕ Мы еще не загрузили и не сохранили ни одного изображения, то есть переменная `_webImage` пуста. Но есть код, указывающий, что делать при наличии сохраненного изображения, поэтому в следующих разделах останется только слегка его отредактировать. Я вынес процесс редактирования в отдельный раздел, так как он включает в себя несколько хитрых приемов.

Разумеется, диспетчер `ImagesManager`, как и все остальные модули диспетчеров, нужно добавить в сценарий менеджеров; этот процесс демонстрируется в следующем листинге.

Листинг 10.14. Добавление нового диспетчера в сценарий Managers.cs

```
...
[RequireComponent(typeof(ImagesManager))]
...
public static ImagesManager Images {get; private set;}
...
void Awake() {
  Weather = GetComponent<WeatherManager>();
  Images = GetComponent<ImagesManager>();

  _startSequence = new List<IGameManager>();
  _startSequence.Add(Weather);
  _startSequence.Add(Images);

  StartCoroutine(StartupManagers());
}
...
```

В отличие от настроек диспетчера `WeatherManager`, при начальной загрузке диспетчера `ImagesManager` метод `GetWebImage()` автоматически не вызывается. Вместо этого код ожидает процесса активации, который мы запрограммируем в следующем разделе.

10.3.2. Вывод изображения на щит

Только что написанный диспетчер `ImagesManager` ничего не сделает, пока мы его не активируем. Смоделируем рекламный щит, который будет вызывать методы в сценарии `ImagesManager`. Создайте куб и поместите его в центр сцены, введя в поля Position значения `0`, `1.5`, `–5`, а в поля Scale — значения `5`, `3`, `0.5`. Результат показан на рис. 10.6.

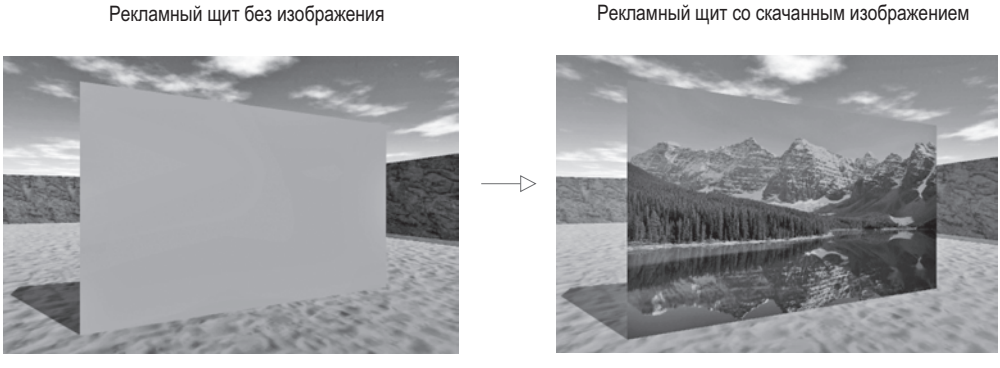


Рис. 10.6. Рекламный щит до и после вывода скачанного из интернета изображения

Создадим устройство, напоминающее меняющий цвета монитор из главы 9. Скопируйте сценарий `DeviceOperator` и свяжите его с объектом `player`. Возможно, вы помните, что этот сценарий срабатывает рядом с устройствами при нажатии клавиши Fire3 (она задается в настройках ввода проекта и в данном случае ей соответствует левая клавиша Command). Кроме того, создайте для устройства Billboard сценарий с именем `WebLoadingBillboard`, свяжите его с рекламным щитом и скопируйте в него код следующего листинга.

Листинг 10.15. Сценарий WebLoadingBillboard для устройства

```
using UnityEngine;
using System.Collections;

public class WebLoadingBillboard : MonoBehaviour {
   public void Operate() {
      Managers.Images.GetWebImage(OnWebImage);
   }

   private void OnWebImage(Texture2D image) {
      GetComponent<Renderer>().material.mainTexture = image;
   }
}
```

Вызываем метод в сценарии ImagesManager.

При обратном вызове скачанное изображение назначается материалу.

Этот код решает две основные задачи: в процессе эксплуатации устройства вызывает метод `ImagesManager.GetWebImage()` и применяет изображение из функции обратного вызова. Текстуры назначены материалам, поэтому в материале рекламного щита можно менять текстуру. Вид щита в процессе игры показан на рис. 10.6.

СКАЧИВАНИЕ ДРУГИХ РЕСУРСОВ

Скачивание изображений с помощью экземпляров класса `UnityWebRequest` — достаточно простая операция. А как обстоят дела с остальными ресурсами, такими как сеточные объекты и шаблоны экземпляров? Класс `UnityWebRequest` обладает свойствами, связанными с текстом и графикой, поэтому в случае других ресурсов ситуация несколько усложняется.

Впрочем, в Unity есть механизм AssetBundles, позволяющий скачивать что угодно. Коротко говоря, вы сначала упаковываете ресурсы в пакет (bundle), который затем скачивается, и Unity извлекает их из пакета. Подробное рассмотрение процесса создания и скачивания пакетов выходит за рамки данной книги; но при желании изучить эту тему вы можете начать с руководства, которое найдете на страницах https://docs.unity3d.com/ru/current/Manual/AssetBundlesIntro.html и https://docs.unity3d.com/Manual/UnityWebRequest-DownloadingAssetBundle.html.

Потрясающе! Изображение, взятое из интернета, появилось на рекламном щите. Но код можно оптимизировать, добавив в него возможность работы с несколькими щитами. Именно этим мы сейчас и займемся.

10.3.3. Кэширование скачанного изображения

В разделе 10.3.1 я упоминал, что сценарий `ImagesManager` не умеет сохранять скачанное изображение. Поэтому чтобы отобразить его на разных рекламных щитах, его придется скачивать несколько раз. Это неэффективно, ведь мы каждый раз заново делаем уже выполненную работу. Давайте заставим сценарий `ImagesManager` кэшировать скачанные изображения.

ОПРЕДЕЛЕНИЕ *Кэшировать* (cache) означает сохранить на локальной машине. Наиболее распространенный (но не единственный!) контекст этого термина касается изображений, скачанных из интернета.

Решить задачу в данном случае поможет функция обратного вызова в сценарии `ImagesManager`, сначала сохраняющая изображение, а затем выполняющая обратный вызов из сценария `WebLoadingBillboard`. Решение в данном случае является нетривиальным (в отличие от текущего кода, в котором фигурирует обратный вызов из сценария `WebLoadingBillboard`), потому что код заранее не знает, каким будет обратный вызов из сценария `WebLoadingBillboard`. Другими словами, невозможно добавить в сценарий `ImagesManager` метод, вызывающий определенный метод в сценарии `WebLoadingBillboard`, так как коду неизвестно, что это за метод. Мы можем выйти из положения, прибегнув к *лямбда-функциям*.

ОПРЕДЕЛЕНИЕ *Лямбда-функцией* (lambda function), или *анонимной функцией*, называется функция, не имеющая имени. Обычно лямбда-функции создаются «на лету» внутри других функций.

Такой нетривиальный инструмент кодирования, как лямбда-функция, поддерживается рядом языков программирования, в том числе C#. Добавив ее в сценарий `ImagesManager`, мы сможем создать функцию обратного вызова «на лету», взяв метод, переданный из сценария `WebLoadingBillboard`. При таком подходе нет нужды заранее знать, какой из методов будет вызываться, ведь лямбда-функции изначально не существует! Посмотрим на практике, как это все выглядит.

Листинг 10.16. Обратный вызов с помощью лямбда-функции в сценарии ImagesManager

```
...
using System;
...
public void GetWebImage(Action<Texture2D> callback) {
  if (_webImage == null) {
    StartCoroutine(_network.DownloadImage((Texture2D image) => {
      _webImage = image;
      callback(_webImage);
    }));
  }
  else {
    callback(_webImage);
  }
}
...
```

Сохраняем скачанное изображение.

Обратный вызов используется в лямбда-функции, а не отправляется непосредственно в сценарий NetworkService.

Основные изменения претерпела функция, передаваемая в метод `NetworkService.DownloadImage()`. Раньше код передавался через один и тот же метод обратного вызова в сценарии `WebLoadingBillboard`. Но после редактирования отправляемый в сценарий `NetworkService` обратный вызов превратился в лямбда-функцию, объявленную в момент вызова метода из сценария `WebLoadingBillboard`. Обратите внимание на синтаксис объявления лямбда-функции: `() => {}`.

Превращение обратного вызова в отдельную функцию дало не только возможность вызвать метод в сценарии `WebLoadingBillboard`; именно лямбда-функция сохраняет локальную копию скачанного изображения. В итоге методу `GetWebImage()` достаточно выполнить скачивание один раз; во всех последующих вызовах будет использоваться уже локальная копия.

Сделанная оптимизация применяется к последующим вызовам, поэтому ее эффект появится только при наличии нескольких рекламных щитов. Сделаем копию щита, чтобы в сцене появилась хотя бы пара объектов. Выделите щит, выберите команду Duplicate (в меню Edit или в меню, вызываемом щелчком правой кнопки мыши) и переместите копию в сторону (например, измените координату *X* на 18).

Запустите игру и посмотрите, что получилось. В управлении первым щитом отмечается значительная пауза, связанная со скачиванием изображения из интернета. Зато при подходе ко второму щиту изображение появляется сразу же, ведь оно уже загружено.

Это важная оптимизация процесса скачивания изображений (именно поэтому у браузеров режим кэширования графики включен по умолчанию). Теперь осталась всего одна задача, с решением которой я хочу вас познакомить: отправка данных на сервер.

10.4. Отправка данных на веб-сервер

Мы рассмотрели несколько примеров скачивания данных, а вот обратный процесс пока не выполнялся ни разу. Задания этого раздела требуют наличия сервера, на который будут отправляться данные. Если у вас нет доступа к серверу, раздел можно просто пропустить. Желающие могут скачать программное обеспечение с открытым исходным кодом и настроить тестовый сервер.

Я рекомендую для этой цели сервер XAMPP. Его можно скачать на странице https://www.apachefriends.org/ru/index.html. После установки появился доступ к папке htdocs по адресу http://localhost/, как будто вы работаете с сервером в интернете. Создайте в этой папке папку uia — именно сюда мы будем сохранять работающие на стороне сервера сценарии.

Какой бы вариант вы ни выбрали — XAMPP или собственный веб-сервер, задача будет одна: в момент достижения персонажем контрольной точки отправить на сервер информацию о погоде. Контрольной точкой послужит зона триггера, напоминающая дверь из главы 9. Создайте куб, укажите, что его коллайдер используется как триггер, и назначьте ему полупрозрачный материал, как и в предыдущем случае (напоминаю, что нужно настроить режим визуализации материала). Итоговый вид этого объекта показан на рис. 10.7.

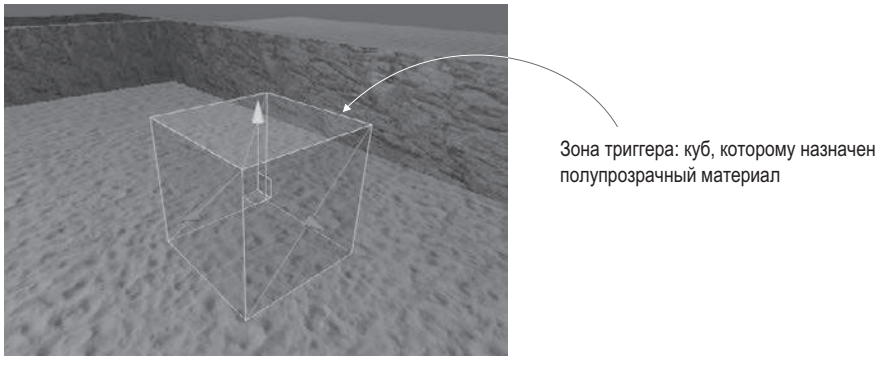


Рис. 10.7. Контрольная точка, вызывающая отправку данных на сервер

Теперь, когда в сцене появился триггер, напишем код, который он будет активировать.

10.4.1. Слежение за погодой: отправка запросов post

Код, активируемый в контрольной точке, будет последовательно включаться в нескольких сценариях. Как и в случае со скачиванием данных, при отправке данных сценарий `WeatherManager` заставляет сценарий `NetworkService` сделать запрос, причем последний полностью отвечает за передачу данных через сеть по протоколу HTTP. Вот коррективы, которые нужно внести в сценарий `NetworkService`.

Листинг 10.17. Сценарий NetworkService с возможностью отправки данных

```
...
private const string localApi = "http://localhost/uia/api.php";
...
private IEnumerator CallAPI(string url, WWWForm form, Action<string>
    callback) {
  using (UnityWebRequest request = (form == null) ?
    UnityWebRequest.Get(url) : UnityWebRequest.Post(url, form)) {

    yield return request.Send();

    if (request.isError) {
      Debug.LogError("network problem: " + request.error);
    } else if (request.responseCode != (long)System.Net.HttpStatusCode.OK)
    {
      Debug.LogError("response error: " + request.responseCode);
    } else {
      callback(request.downloadHandler.text);
    }
  }
}

public IEnumerator GetWeatherXML(Action<string> callback) {
  return CallAPI(xmlApi, null, callback);
}
public IEnumerator GetWeatherJSON(Action<string> callback) {
  return CallAPI(jsonApi, null, callback);
}

public IEnumerator LogWeather(string name, float cloudValue, Action<string>
       callback) {
  WWWForm form = new WWWForm();
  form.AddField("message", name);
  form.AddField("cloud_value", cloudValue.ToString());
  form.AddField("timestamp", DateTime.UtcNow.Ticks.ToString());

  return CallAPI(localApi, form, callback);
}
...
```

Обращаемся к сценарию на стороне сервера; при необходимости вы можете его поменять.

Аргументы, добавленные к параметрам метода CallAPI().

Запрос POST с использованием объекта WWWForm или запрос GET без этого объекта.

Вызовы изменены из-за измененных параметров.

Определяем форму с аргументами для отправки.

Отправляем метку времени вместе с информацией об облачности.

Прежде всего, обратите внимание на новый параметр метода `CallAPI()`. Это объект `WWWForm`, представляющий собой набор значений, отправляемый вместе с HTTP-запросом. Код содержит условный оператор, использующий наличие объекта `WWWForm` для изменения создаваемого запроса. Обычно мы отправляем запрос `GET`, а объект `WWWForm` меняет его на запросы `POST`. Все остальные изменения в коде вызваны этим основным нововведением (например, модификация кода метода `GetWhatever()` из-за параметров метода `CallAPI()`).

Следующий листинг демонстрирует код, который нужно добавить в сценарий `WeatherManager`.

Листинг 10.18. Добавление к сценарию WeatherManager кода отправки данных

```
...
public void LogWeather(string name) {
  StartCoroutine(_network.LogWeather(name, cloudValue, OnLogged));
}
private void OnLogged(string response) {
  Debug.Log(response);
}
...
```

И наконец, задействуем этот код, добавив к зоне триггера в сцене сценарий, управляющий контрольной точкой. Создайте сценарий `CheckpointTrigger`, свяжите его с зоной триггера и скопируйте в него код из следующего листинга.

Листинг 10.19. Сценарий CheckpointTrigger для зоны триггера

```
using UnityEngine;
using System.Collections;

public class CheckpointTrigger : MonoBehaviour {
  public string identifier;

  private bool _triggered;  <-- Проверяем, срабатывала ли уже контрольная точка.

  void OnTriggerEnter(Collider other) {
    if (_triggered) {return;}

    Managers.Weather.LogWeather(identifier);  <-- Вызываем для отправки данных.
    _triggered = true;
  }
}
```

На панели Inspector появится параметр Identifier; назовите его, например, checkpoint1. Запустите код. При входе в зону триггера на сервер начнут отправляться данные. Но в ответ вы получите сообщение об ошибке, так как на сервере пока отсутствует сценарий, получающий запрос. Именно его созданием и завершится данный раздел.

10.4.2. Серверный код в PHP-сценарии

Серверу требуется сценарий, который будет получать отправляемые игрой данные. Вопрос написания таких сценариев выходит за рамки данной книги, поэтому детально мы его рассматривать не будем. Мы просто напишем на скорую руку PHP-сценарий, так как это самый простой подход к решению задачи. Создайте в папке htdocs (или там, где располагается ваш веб-сервер) текстовый файл api.php и скопируйте в него код из следующего листинга.

Листинг 10.20. Серверный сценарий на языке PHP, получающий наши данные

```
<?php

$message = $_POST[‘message’];  <-- Извлекаем присланные данные в переменные.
$cloudiness = $_POST['cloud_value'];
$timestamp = $_POST['timestamp'];
```

```
$combined = $message." cloudiness=".$cloudiness." time=".$timestamp."\n";

$filename = "data.txt";  ◄── Определяем имя файла, в который будет выполняться запись.
file_put_contents($filename, $combined, FILE_APPEND | LOCK_EX);  ◄── Записываем файл.

echo "Logged";

?>
```

Сценарий записывает полученные данные в файл data.txt, соответственно, на сервере нужно создать файл с таким именем. Как только сценарий `api.php` будет готов, вы увидите, что в файле data.txt при достижении контрольной точки в игре появляются данные о погоде. Великолепно!

Заключение

- Скайбокс предназначен для отображения неба, которое визуализируется позади всех остальных объектов сцены.
- В Unity есть объект `UnityWebRequest`, предназначенный для скачивания данных.
- Распространенные форматы данных, такие как XML и JSON, легко доступны для синтаксического анализа.
- Материалы могут отображать фотографии, скачанные из интернета.
- Объект `UnityWebRequest` позволяет также отправлять данные на веб-сервер.

11 Звуковые эффекты и музыка

- ✓ Импорт и воспроизведение аудиоклипов для получения звуковых эффектов.
- ✓ Двумерные звуки для пользовательского интерфейса и трехмерные звуки в сцене.
- ✓ Уровни громкости всех звуков в процессе воспроизведения.
- ✓ Воспроизведение фоновой музыки в процессе игры.
- ✓ Плавный переход от одного музыкального трека к другому.

Когда речь заходит о видеоиграх, основное внимание уделяется графике, хотя существует и такой важный аспект, как звуковое сопровождение. В большинстве игр играет фоновая музыка и присутствуют звуковые эффекты. Соответственно, в Unity присутствует функциональность для их создания. Можно импортировать в Unity и воспроизводить аудиофайлы различных форматов, регулировать громкость звука и даже обрабатывать звуки, исходящие из определенной точки сцены.

ПРИМЕЧАНИЕ Как в двумерных, так и в трехмерных играх звук обрабатывается одинаково. Материал этой главы будет рассматриваться на примере трехмерной игры, но все процедуры применимы и к двумерным играм.

Начнем мы с рассмотрения звуковых эффектов. Они представляют собой короткие аудиоклипы, воспроизводимые во время определенных действий (например, в эту категорию попадает звук выстрела, сопровождающий стрельбу игрока по врагам). Звуковые клипы с музыкой имеют большую продолжительность (часто речь идет о минутах), а их воспроизведение не привязано к конкретным событиям в игре. В конечном счете все сводится к одному виду аудиофайлов и одинаковому коду их воспроизведения, но тот факт, что файлы с музыкой обычно намного продолжительнее клипов со звуковыми эффектами (более того, часто оказывается, что это самые большие файлы в игре!), заслуживает, чтобы их выделили в отдельный раздел.

В этой главе мы возьмем игру без звукового сопровождения и сделаем следующее:

1. Импортируем аудиофайлы для звуковых эффектов.
2. Добавим звуковые эффекты к противникам и процессу стрельбы по ним.
3. Запрограммируем диспетчер управления звуком, чтобы контролировать громкость воспроизведения.
4. Оптимизируем загрузку музыки.
5. Сделаем отдельную регулировку громкости для музыки и звуковых эффектов, в том числе для переходящих один в другой музыкальных треков.

ПРИМЕЧАНИЕ Эта глава не привязана к конкретному проекту; мы просто добавляем звук к существующему демонстрационному ролику. Все упражнения демонстрируются на примере шутера от первого лица, созданного в главе 3. Вы можете скачать соответствующий фрагмент проекта или воспользоваться любым другим демонстрационным роликом по своему вкусу.

Надеюсь, у вас уже открыт демонстрационный ролик и можно сделать первый шаг: импортировать звуковые эффекты.

11.1. Импорт звуковых эффектов

Чтобы появилась возможность воспроизведения звуков, нужно импортировать в Unity-проект аудиофайлы. Процедура начинается с подбора файлов нужного формата, которые затем переносятся в Unity и настраиваются под конкретные цели.

11.1.1. Поддерживаемые форматы файлов

Раздел, посвященный графическим ресурсам в главе 4, содержал сведения и о различных форматах аудио. Они перечислены в табл. 11.1.

Таблица 11.1. Форматы аудиофайлов, поддерживаемые в Unity

Тип файла	Достоинства и недостатки
WAV	Формат, предлагаемый в Windows по умолчанию. Несжатый звуковой файл
AIF	Формат, предлагаемый в Mac по умолчанию. Несжатый звуковой файл
MP3	Сжатый звуковой файл; ради уменьшения размера жертвуется качеством
OGG	Сжатый звуковой файл; ради уменьшения размера жертвуется качеством
MOD	Формат для музыкальных трекеров. Предназначен для эффективной записи цифровой музыки
XM	Формат для музыкальных трекеров. Предназначен для эффективной записи цифровой музыки

Форматы отличаются друг от друга в основном степенью сжатия. Сжатие уменьшает размер файла за счет удаления из него некоторой части информации. Оно реализовано таким образом, что удалению подвергается наименее важная информация, поэтому качество звука остается вполне приемлемым. Хотя потеря качества незначительна, для

коротких звуковых клипов следует выбирать форматы без сжатия. Для более длинных звуковых клипов (особенно с музыкой) желателен формат со сжатием, в противном случае размер файла становится недопустимо большим.

Впрочем, при работе в Unity на выбор влияет еще один фактор...

СОВЕТ В Unity есть возможность сжимать аудиофайлы после импорта. Поэтому в процессе разработки игры имеет смысл пользоваться форматами без сжатия даже для длинных музыкальных клипов, а не импортировать сжатые аудиофайлы.

ЦИФРОВОЙ ЗВУК

В общем случае аудиофайлы сохраняют форму сигнала, которая создается в наушниках при прослушивании музыки. Звук представляет собой набор распространяющихся в воздухе волн, соответственно, различные звуки состоят из звуковых волн различных размеров и частот. В аудиофайлах эти волны записываются с созданием множества их замеров через короткие интервалы времени и сохранением состояния волны в каждом замере.

Чем чаще записываются замеры волн, тем точнее получается запись изменений волны во времени и тем меньше интервалы между изменениями. Но это означает большее количество сохраняемых данных и, как следствие, больший размер файла. При сжатии размер файла уменьшается за счет различных приемов, например удаления данных на звуковых частотах, неразличимых человеческим ухом.

Музыкальные трекеры представляют собой особый тип программных музыкальных секвенсоров для создания музыки. Если в аудиофайлах звук хранится в виде обычных волн, секвенсоры сохраняют нечто, больше напоминающее ноты: файл-трекер содержит последовательность нот с такой информацией, как интенсивность и частота каждой из них. Эти «ноты» состоят из небольших волн, но общее количество сохраненных данных невелико, так как каждая нота используется в последовательности много раз. Написанная таким способом музыка при всей своей эффективности представляет собой крайне специализированные аудиофайлы.

Так как сжать звуковой файл можно после импорта, всегда выбирайте файлы в формате WAV или AIF. У импорта коротких звуковых эффектов и музыки, скорее всего, будут разные настройки (в частности, будет отличаться момент сжатия файла средствами Unity), но исходный файл всегда должен быть без сжатия.

Музыкальные файлы создаются разными способами (например, в приложении Б упоминается инструмент Audacity, позволяющий записывать звук при помощи микрофона), но для нашего проекта скачаем образцы звуков с одного из многочисленных бесплатных сайтов. В качестве источника ресурсов я предлагаю сайт https://freesound.org/. Нам нужны клипы в формате WAV.

ВНИМАНИЕ «Бесплатные» аудиофайлы предлагаются под различными вариантами лицензий, поэтому всегда проверяйте, разрешено ли использовать конкретный клип нужным вам способом. К примеру, многие бесплатные образцы музыки предназначены только для некоммерческого применения.

В моем примере проекта фигурируют следующие звуковые эффекты (разумеется, вам ничто не мешает скачать собственные варианты клипов; не забудьте проверить отсутствие лицензионных ограничений):

- thump от пользователя hy96;
- ding от пользователя Daphne_in_Wonderland;

- swish bamboo pole от пользователя ra_gun;
- fireplace от пользователя leosalom.

Теперь скачанные файлы нужно импортировать в Unity.

11.1.2. Импорт аудиофайлов

Собранную вами коллекцию аудиофайлов нужно импортировать в Unity. В главе 4 вы узнали, что любые ресурсы нужно вставить в проект, и только после этого их можно будет использовать в игре.

Простой механизм импорта работает со всеми видами ресурсов: вы перетаскиваете файлы из папки на компьютере на вкладку Project в Unity (создайте для этой цели папку с именем Sound FX). Как видите, все очень просто! Но, как и у остальных ресурсов, у аудиофайлов есть настройки импорта, которые задаются на панели Inspector (рис. 11.1).

Флажок Force To Mono устанавливать не нужно. Он переключает между моно- и стереозвуком; зачастую звук записывается в стерео, то есть в файле присутствуют одновременно два сигнала: один для левого, другой для правого наушника или колонки. Чтобы уменьшить размер файла, можно убрать половину звуковой информации. После этого на обе колонки будет посылаться один сигнал вместо двух. При импорте в режиме моно появляется доступ к флажку Normalize.

Следом идут флажки Load In Background (Загрузка в фоновом режиме) и Preload Audio Data (Предварительная загрузка аудиоданных). Настройки предварительной загрузки — это попытка найти баланс между производительностью воспроизведения и использованием памяти; заранее загруженный аудиофайл занимает память, ожидая своей очереди, зато в процессе его воспроизведения не приходится тратить время на загрузку. Загрузка звука в фоновом режиме позволяет программе продолжить свою работу; этот вариант в общем случае подходит для длинных музыкальных клипов, но возможность немедленно запустить воспроизведение звука отсутствует. Для коротких звуковых клипов флажок Load In Background обычно не устанавливается, что гарантирует их полную загрузку перед воспроизведением. Сейчас мы импортируем короткие звуковые эффекты, и устанавливать этот флажок не нужно.

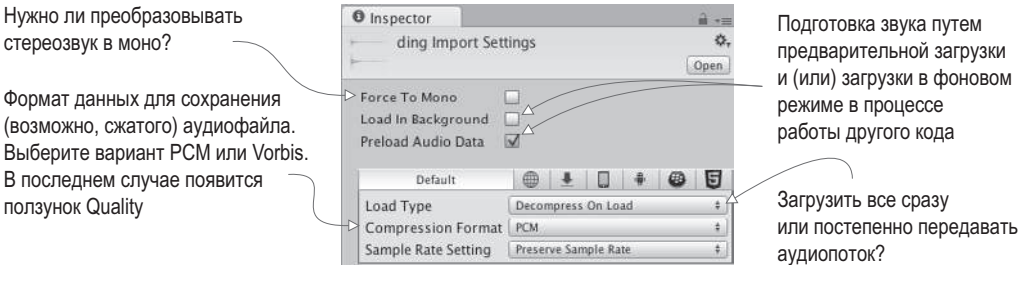


Рис. 11.1. Настройки импорта для аудиофайлов

Наконец, наиболее важными параметрами являются Load Type (Загружаемый тип) и Compression Format (Формат сжатия). Последний отвечает за формат сохраненных аудиоданных. Как упоминалось в предыдущем разделе, музыку следует сжать, поэтому

для нее выбирайте вариант Vorbis (это название аудиоформата со сжатием). Короткие звуковые клипы в сжатии не нуждаются, поэтому выбирайте для них вариант PCM (Pulse Code Modulation — импульсно-кодовая модуляция). Третий вариант — ADPCM — вариация PCM, дающая чуть более качественный звук.

Раскрывающийся список Load Type позволяет указать способ загрузки данных из файла. Объем памяти у компьютера ограничен, а аудиофайлы могут быть очень большими, поэтому иногда имеет смысл запускать воспроизведение звука в процессе его передачи в память, избавляя компьютер от необходимости единовременно загружать весь файл. Однако такая передача звука требует много вычислительных ресурсов, поэтому быстрее всего воспроизводятся файлы, предварительно загруженные в память. Но даже в этом случае вы можете выбрать, будут ли данные представлены в сжатой форме или их следует распаковать для более быстрого воспроизведения. Наши аудиоклипы имеют небольшой размер, и воспроизведение в процессе загрузки не требуется, поэтому можно выбрать вариант Decompress On Load.

Последнему параметру Sample Rate оставим значение по умолчанию Preserve Sample Rate. В этом случае Unity не будет менять частоту семплирования импортированного файла. Итак, в нашем проекте появились готовые к работе аудиофайлы.

11.2. Звуковые эффекты

Теперь нужно сделать так, чтобы добавленные к проекту звуковые файлы зазвучали. Код активации звуковых эффектов понять несложно, но аудиосистема в Unity состоит из фрагментов, которые должны работать согласованно.

11.2.1. Система воспроизведения: клипы, источник, подписчик

Возможно, вы ожидаете, что для проигрывания звука в Unity достаточно указать, какие клипы нужно воспроизводить, но на самом деле для этого нужно задать три компонента: `AudioClip`, `AudioSource` и `AudioListener`. Подобное разделение связано с поддержкой трехмерного звука: различные компоненты сообщают Unity информацию о местоположении, которая используется для управления трехмерным звуком.

2D- И 3D-ЗВУК

Звук в играх бывает двумерный и трехмерный. С первым вы уже знакомы, это стандартный звук, воспроизводимый обычным образом. Выражение «2D-звук» в большинстве случаев означает «не 3D-звук».

3D-звук характерен для трехмерного моделирования, и, возможно, с ним вы еще не сталкивались — это звук, исходящий из конкретных мест сцены. Его громкость и тон зависят от положения слушателя. Например, звуковой эффект, включенный с большого расстояния, будет слышен крайне слабо.

В Unity поддерживаются оба вида звуков, и вы сами выбираете вид звука для каждого источника. Для фоновой музыки следует выбирать двумерный звук, в то время как трехмерный звук в большинстве звуковых эффектов создает ощущение присутствия в сцене.

Представьте магнитолу, проигрывающую компакт-диск в комнате. Зашедший в комнату человек четко услышит звук. После выхода из комнаты он начнет слышать его

тише, и в конце концов, на каком-то расстоянии от источника, звук вообще исчезнет. При перемещении магнитолы по комнате громкость звука меняется. Эту аналогию иллюстрирует рис. 11.2. Компакт-диск — аналог компонента `AudioClip`, магнитола — компонента `AudioSource`, а человек — компонента `AudioListener`.

Рис. 11.2. Компоненты, управляющие аудиосистемой в Unity

Первым компонентом является *аудиоклип*. Он связан с реальным звуковым файлом, который мы импортировали в предыдущем разделе. Эти необработанные данные о форме сигнала — основа всего, что делает аудиосистема, но сам по себе аудиоклип не выполняет никаких действий.

Следующий объект — это *источник звука*. Именно он воспроизводит аудиоклипы. Это абстракция того, что на самом деле делает аудиосистема, но именно благодаря ей концепция трехмерного звука становится более понятной. 3D-звук, воспроизводимый конкретным источником, расположен в той же точке, что и этот источник; и хотя 2D-звуки также воспроизводятся источником, их местоположение не имеет значения.

Третьим объектом в составе аудиосистемы в Unity является *слушатель звука*. Как понятно из названия, это объект, который слышит звуки, проецируемые из источников. Это еще одна абстракция реальных операций аудиосистемы (ведь очевидно, что реальный слушатель в данном случае тот, кто играет в игру!), но аналогично тому, как положение источника звука определяет координаты места, из которого исходит звук, положение слушателя звука определяет координаты места, в котором слышен звук.

УСОВЕРШЕНСТВОВАННАЯ СИСТЕМА УПРАВЛЕНИЯ ЗВУКОМ

Функция Audio Mixers (аудиомикшеры) впервые появилась в Unity 5. Аудиомикшеры позволяют обрабатывать аудиосигналы и накладывать на клипы различные эффекты. Более подробно с этой функцией можно познакомиться в документации к Unity, например, посмотрев обучающее видео http://mng.bz/Mlp3.

Компоненты `AudioClip` и `AudioSource` нужно назначать, а вот компонентом `AudioListener` оснащена камера, которая по умолчанию появляется в каждой новой сцене. Как правило, нужно, чтобы 3D-звуки реагировали на положение наблюдателя.

11.2.2. Зацикленный звук

Итак, давайте настроим наш первый звук в Unity! Аудиоклипы уже импортированы, у используемой по умолчанию камеры есть компонент `AudioListener`, так что остается назначить только компонент `AudioSource`. Добавим треск огня к шаблону `Enemy` — это персонаж, который хаотично перемещается по сцене.

ПРИМЕЧАНИЕ Противники издают такие звуки, как будто они охвачены пламенем, поэтому можно связать с ними систему частиц, чтобы придать соответствующий звуку вид. Можно скопировать систему частиц, созданную в главе 4, превратив объект `Particle` в шаблон экземпляра, а затем выбрав в меню Asset команду Export Package. В качестве альтернативы можно повторить процедуру из главы 4, создав новый объект с нуля (перетащите шаблон Enemy в сцену для редактирования, а затем выберите в меню GameObject команду Apply Changes To Prefab).

Обычно для редактирования шаблона экземпляра его нужно перетащить в сцену, но в данном случае эту процедуру можно выполнить напрямую сразу после добавления компонента к объекту. Выделите шаблон Enemy, чтобы его свойства появились на панели Inspector. Затем добавьте новый компонент, выбрав вариант Audio > Audio Source. На панели Inspector появится компонент AudioSource.

Укажите источнику аудиоклип, который нужно воспроизвести. Перетащите файл со звуком со вкладки Project на ячейку Audio Clip панели Inspector; для этого примера мы используем звуковой эффект fireplace (рис. 11.3).

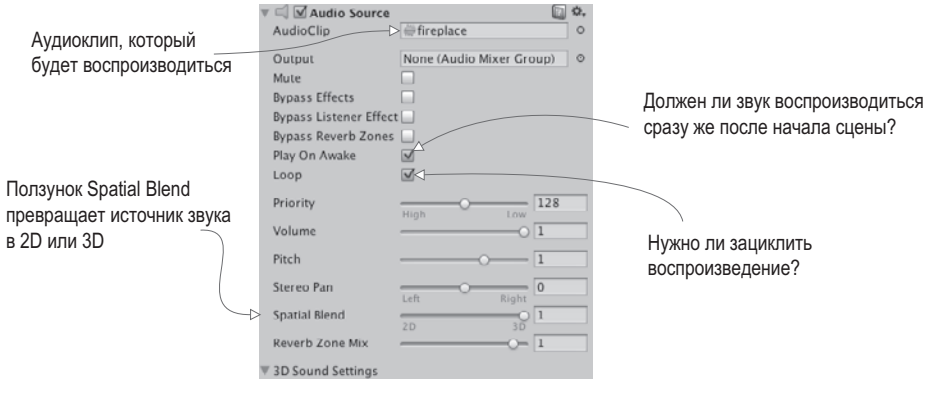


Рис. 11.3. Настройки компонента AudioSource

Установите флажки Play On Awake и Loop (обязательно убедитесь, что флажок Mute сброшен). Флажок Play On Awake заставляет источник звука начать воспроизведение сразу после загрузки сцены (в следующем разделе вы узнаете, как активировать звуки вручную во время игры). Флажок Loop заставляет источник звука играть непрерывно, повторяя клип снова и снова.

Мы хотим, чтобы этот источник аудио испускал 3D-звуки. Как уже объяснялось, 3D-звук имеет в сцене определенное положение. Это свойство источника звука регулируется ползунком Spatial Blend. Именно он отвечает за превращение звука из двумерного в трехмерный; установите его в крайнее правое положение.

Теперь проверьте, включен ли звук у вас в колонках, и запустите игру. Вы услышите исходящее от врага потрескивание, которое ослабевает по мере его удаления от персонажа, так как в сцене используется источник 3D-звука.

11.2.3. Активация звуковых эффектов из кода

Настройка компонента `AudioSource` на автоматическое воспроизведение хорошо подходит для циклических звуков, но в большинстве случаев нужно, чтобы звуковые эффекты возникали в ответ на команды кода. Для этого все равно требуется компонент AudioSource, но воспроизведение звука запускается программно.

Добавьте компонент `AudioSource` к объекту `player` (а не к камере). Связывать с компонентом конкретный аудиоклип на этот раз нет нужды, так как он определяется в коде. Сбросьте флажок Play On Awake, ведь звук будет возникать в ответ на команду. Ползунок Spatial Blend установите в положение 3D, так как эффект привязан к положению в сцене.

Добавьте в отвечающий за стрельбу сценарий `RayShooter` код следующего листинга.

Листинг 11.1. Добавление в сценарий RayShooter звуковых эффектов

```
...
[SerializeField] private AudioSource soundSource;
[SerializeField] private AudioClip hitWallSound;
[SerializeField] private AudioClip hitEnemySound;
...
if (target != null) {
    target.ReactToHit();
    soundSource.PlayOneShot(hitEnemySound);
} else {
    StartCoroutine(SphereIndicator(hit.point));
    soundSource.PlayOneShot(hitWallSound);
}
...
```

Ссылаемся на два звуковых файла, которые нужно воспроизвести.

Если переменная target не равна null, значит, игрок выстрелил во врага, поэтому...

...вызываем метод PlayOneShot() для воспроизведения звука Hit An Enemy или...

...метод PlayOneShot() для воспроизведения звука Hit A Wall, если игрок промазал.

В новом варианте кода в верхней части сценария появилось несколько сериализованных переменных. Выделите камеру и перетащите объект `player` (обладающий компонентом `AudioSource`) на ячейку soundSource панели Inspector. Затем перетащите на ячейки Hit Wall Sound и Hit Enemy Sound аудиоклипы, которые будут воспроизводиться в каждом случае (`swish` при попадании в стену и `ding` при поражении врага).

Еще две добавленные строки — это методы `PlayOneShot()`. Именно они заставляют источник звука воспроизвести указанный клип. Методы добавляются в условную инструкцию для переменной `target` и при попадании в разные цели воспроизводят разные звуки.

ПРИМЕЧАНИЕ Задать клип можно было в компоненте `AudioSource`, вызвав для его воспроизведения метод `Play()`. Но несколько звуков будут обрывать друг друга, поэтому мы воспользовались методом `PlayOneShot()`. Чтобы понять суть проблемы, замените метод `PlayOneShot()` вот таким кодом и сделайте несколько быстрых выстрелов:

```
soundSource.clip=hitEnemySound; soundSource.Play();
```

Запустите игру и сделайте несколько выстрелов. В игре появилось несколько звуковых эффектов. Все остальные виды звуковых эффектов добавляются аналогичным

образом. Но для полноценной звуковой системы в игре требуется несколько больше, чем набор несвязанных звуков; как минимум, у игрока должна быть возможность контролировать громкость.

Этот элемент управления мы реализуем в следующем разделе с помощью специального модуля.

11.3. Интерфейс управления звуком

Продолжая доработку созданной в предыдущих главах архитектуры, добавим туда диспетчер `AudioManager`. Напомню, что объект `Managers` обладает списком модулей кода, используемых в игре, таких как, к примеру, диспетчер игрового инвентаря. На этот раз будет создан диспетчер управления звуком, который мы тоже добавим в список. Этот центральный модуль позволит регулировать громкость звука в игре и даже выключать звук совсем. Изначально он будет работать только со звуковыми эффектами, но в следующих разделах мы добавим объекту `AudioManager` возможность регулировать громкость музыки.

11.3.1. Центральный диспетчер управления звуком

Для настройки диспетчера `AudioManager` первым делом нужно добавить в проект фреймворк `Managers`. Из проекта главы 10 скопируйте сценарии `IGameManager`, `ManagerStatus` и `NetworkService`. Их мы редактировать не будем. Напоминаю, что `IGameManager` — это интерфейс, который должны реализовывать все диспетчеры, а `ManagerStatus` — перечисление, которым пользуется `IGameManager`. Сценарий `NetworkService`, отвечающий за подключение к интернету, в этой главе не потребуется.

ПРИМЕЧАНИЕ Скорее всего, Unity выведет на экран предупреждение, ведь сценарий `NetworkService` назначен, но не задействован. Просто проигнорируйте его; мы хотим предоставить фреймворку возможность подключения к интернету, просто в этой главе данная функциональность не требуется.

Также скопируйте файл `Managers`, который мы отредактируем с учетом нового диспетчера управления звуком. Пока оставьте его без изменений (или, если вас раздражают появляющиеся сообщения об ошибках компиляции, просто добавьте комментарии к строкам, приводящим к появлению ошибки!). Создайте новый сценарий `AudioManager`, на который может ссылаться код сценария `Managers`.

Листинг 11.2. Заготовка кода для сценария AudioManager

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

public class AudioManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  private NetworkService _network;

  // Место для элементов управления громкостью (листинг 11.4)
```

```
  public void Startup(NetworkService service) {
    Debug.Log("Audio manager starting...");

    _network = service;

    // Инициализация источников музыки (11.10)

    status = ManagerStatus.Started;
  }
}
```

Сюда поместить все длительные задачи, запускаемые на старте.

Если есть длительные задания, запускаемые при старте, присваиваем состоянию значение Initializing.

Этот сценарий напоминает диспетчеры из предыдущих глав — это минимум кода, необходимый интерфейсу `IGameManager` для реализации класса. Теперь можно добавить новый диспетчер в сценарий `Managers`.

Листинг 11.3. Сценарий Managers после добавления сценария AudioManager

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

[RequireComponent(typeof(AudioManager))]

public class Managers : MonoBehaviour {
  public static AudioManager Audio {get; private set;}

  private List<IGameManager> _startSequence;

  void Awake() {
    Audio = GetComponent<AudioManager>();

    _startSequence = new List<IGameManager>();
    _startSequence.Add(Audio);

    StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    NetworkService network = new NetworkService();

    foreach (IGameManager manager in _startSequence) {
      manager.Startup(network);
    }

    yield return null;

    int numModules = _startSequence.Count;
    int numReady = 0;

    while (numReady < numModules) {
      int lastReady = numReady;
      numReady = 0;

      foreach (IGameManager manager in _startSequence) {
        if (manager.status == ManagerStatus.Started) {
          numReady++;
        }
      }
```

В этом проекте в списке AudioManager, а не PlayerManager, и т. п.

```
      if (numReady > lastReady)
        Debug.Log("Progress: " + numReady + "/" + numModules);

      yield return null;
    }

    Debug.Log("All managers started up");
  }
}
```

Как и в предыдущих главах, создайте пустой объект, который будет играть роль диспетчера, и свяжите с ним сценарии `Managers` и `AudioManager`. При воспроизведении игры на консоли появятся сообщения о запуске диспетчеров, но диспетчер управления звуком пока не несет никакой функциональной нагрузки.

11.3.2. Интерфейс для управления громкостью

Основа сценария `AudioManager` уже есть, пришло время добавить туда средства контроля громкости. Отвечающие за эту функциональность методы затем будут использоваться визуальными элементами пользовательского интерфейса, позволяя выключать звуковые эффекты и регулировать громкость.

Применим новые инструменты создания пользовательского интерфейса, с которыми вы встречались в главе 7. Будет создано всплывающее окно с рис. 11.4 с кнопкой и ползунком, отвечающим за громкость звука. Вот этапы создания такого окна. Я не буду рассматривать их подробно. Детали реализации вы можете вспомнить самостоятельно, обратившись к главе 7.

1. Импортируем изображение popup.png как спрайт (параметру Texture Type присваиваем значение Sprite).
2. В окне диалога Sprite Editor формируем со всех сторон границу размером 12 пикселов (не забудьте применить сделанные изменения).
3. Создаем в сцене холст (GameObject > UI > Canvas).
4. Устанавливаем для холста флажок Pixel Perfect.
5. (По желанию.) Присваиваем объекту имя HUD Canvas и переключаемся в режим работы с двумерной графикой.
6. Создаем связанное с холстом изображение (GameObject > UI > Image).
7. Присваиваем новому объекту имя Settings Popup.
8. Перетаскиваем на ячейку Source Image этого объекта спрайт popup.
9. В раскрывающемся списке Image Type выбираем вариант Sliced и устанавливаем флажок Fill Center.
10. Помещаем изображение всплывающего окна в точку с координатами 0, 0.
11. Меняем размеры всплывающего окна до 250 по ширине и 150 по высоте.
12. Создаем кнопку (GameObject > UI > Button).
13. Делаем кнопку дочерним объектом по отношению к всплывающему окну (путем перетаскивания на вкладке Hierarchy).
14. Помещаем кнопку в точку с координатами 0, 40.

15. Раскрываем иерархический список кнопки, чтобы выделить связанную с ней текстовую метку.
16. Меняем текст на Toggle Sound.
17. Создаем ползунок (GameObject > UI > Slider).
18. Делаем ползунок дочерним объектом всплывающего окна и помещаем его в точку с координатами 0, 15.
19. Присваиваем параметру Value ползунка (в нижней части панели Inspector) значение 1.

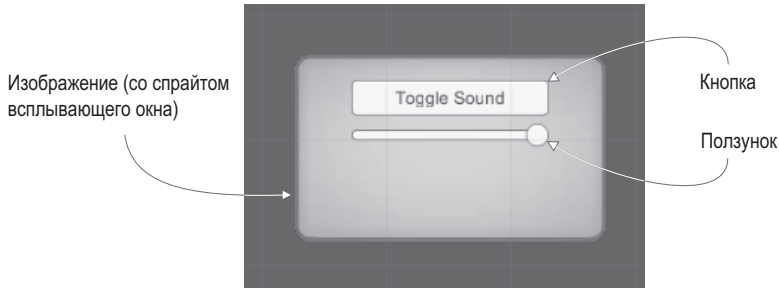


Рис. 11.4. Элемент UI для выключения звука и регулировки громкости

Теперь напишем для всплывающего окна код. Сценарий понадобится как для самого окна, так и для функции управления громкостью, которую он будет вызывать. Для начала отредактируйте код сценария `AudioManager` в соответствии со следующим листингом.

Листинг 11.4. Добавление в сценарий AudioManager средств регулировки громкости звука

```
...
public float soundVolume {   <-- Свойство для громкости с функцией чтения и функцией доступа.
  get {return AudioListener.volume;}      | Реализуем функцию чтения/функцию доступа
  set {AudioListener.volume = value;}     | с помощью компонента AudioListener.
}

public bool soundMute {   <-- Добавляем аналогичное свойство для выключения звука.
  get {return AudioListener.pause;}
  set {AudioListener.pause = value;}
}

public void Startup(NetworkService service) {   <-- Выделенный курсивом код уже был
  Debug.Log("Audio manager starting...");       | в сценарии, тут он показан для справки.

  _network = service;

  soundVolume = 1f;   <-- Инициализируем значение (в диапазоне от 0 до 1; 1 соответствует полной громкости).

  status = ManagerStatus.Started;
}
...
```

В сценарий AudioManager добавлены свойства soundVolume и soundMute. Функция чтения и задающая функция для этих свойств реализованы с помощью глобальных переменных класса AudioListener. Класс AudioListener может регулировать громкость всех звуков, получаемых всеми экземплярами компонента AudioListener. Задание свойства soundVolume в сценарии AudioManager оказывает такой же эффект, как задание громкости в компоненте AudioListener. Преимущество такого подхода состоит в инкапсуляции: все, что имеет отношение к звуку, обрабатывается одним диспетчером, и внешнему коду не нужно знать детали реализации.

После добавления этих методов в сценарий AudioManager можно написать сценарий для всплывающего окна. Создайте сценарий с именем SettingsPopup и добавьте туда содержимое следующего листинга.

Листинг 11.5. Сценарий SettingsPopup с элементами управления громкостью

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour {

  public void OnSoundToggle() {
    Managers.Audio.soundMute = !Managers.Audio.soundMute;
  }

  public void OnSoundValue(float volume) {
    Managers.Audio.soundVolume = volume;
  }
}
```

Кнопка переключает свойство mute диспетчера управления звуком.

Ползунок регулирует свойство volume диспетчера управления звуком.

В этом сценарии мы видим два метода, влияющие на свойства объекта AudioManager: метод OnSoundToggle() задает свойство soundMute, а метод OnSoundValue() — свойство soundVolume. Как обычно, перетащите сценарий SettingsPopup на объект Settings Popup в пользовательском интерфейсе.

Чтобы эти методы можно было вызывать с помощью кнопки и ползунка, свяжите всплывающее окно с событиями взаимодействия указанных элементов управления. На панели Inspector для кнопки найдите поле OnClick. Щелкните на кнопке со знаком + (плюс), чтобы добавить к этому событию новый элемент. Перетащите объект Settings Popup на ячейку для объекта в новом элементе и найдите в меню вариант SettingsPopup; чтобы кнопка начала вызывать данную функцию, выберите вариант OnSoundToggle().

Аналогичным способом свяжем с методом ползунок. Первым делом найдите событие взаимодействия в настройках ползунка — в данном случае оно будет называться OnValueChanged. Щелкните на кнопке со знаком + (плюс) для добавления нового элемента и перетащите на ячейку для объекта объект Settings Popup. В меню функций найдите сценарий SettingsPopup и выберите в разделе Dynamic Float вариант OnSoundVolume().

ВНИМАНИЕ Напоминаю, что здесь требуется функция из раздела Dynamic Float, а не Static Parameter! Указанный метод присутствует в обоих разделах, но в последнем случае он сможет получить только одно заранее введенное значение.

Элементы управления теперь работают, но в проект нужно внести небольшие коррективы. Сейчас экран все время закрывает всплывающее окно. Сделаем так, чтобы оно открывалось только при нажатии клавиши M. Создайте новый сценарий `UIController`, свяжите его с контроллером в сцене и введите код следующего листинга.

Листинг 11.6. Сценарий UIController, вызывающий и скрывающий всплывающее окно

```
using UnityEngine;
using System.Collections;

public class UIController : MonoBehaviour {
  [SerializeField] private SettingsPopup popup;

  void Start() {
    popup.gameObject.SetActive(false);
  }

  void Update() {
    if (Input.GetKeyDown(KeyCode.M)) {
      bool isShowing = popup.gameObject.activeSelf;
      popup.gameObject.SetActive(!isShowing);

      if (isShowing) {
         Cursor.lockState = CursorLockMode.Locked;
         Cursor.visible = false;
      } else {
         Cursor.lockState = CursorLockMode.None;
         Cursor.visible = true;
      }
    }
  }
}
```

Ссылаемся на всплывающее окно в сцене.

Инициализируем всплывающее окно в скрытом состоянии.

Вызываем и скрываем всплывающее окно при помощи клавиши M.

Вместе со всплывающим окном вызываем курсор.

Для подключения этой ссылки на объект перетащите всплывающее окно настроек на ячейку сценария. Теперь запустите игру и попытайтесь подвигать ползунок (напоминаю, что UI активируется нажатием клавиши M), стреляя по сторонам, чтобы слышать звуковые эффекты; вы убедитесь, что их громкость меняется в соответствии с положением ползунка.

11.3.3. Звуки пользовательского интерфейса

Внесем в сценарий `AudioManager` еще одно дополнение. Сделаем так, чтобы щелчки на кнопках UI имели звуковое сопровождение. Эта задача сложнее, чем кажется на первый взгляд. Ведь требуется компонент `AudioSource`. Когда звуки испускаются объектами сцены, место его присоединения очевидно. Но звуковые эффекты UI не являются частью сцены, поэтому нужно настроить специальный компонент `AudioSource` для диспетчера `AudioManager`. Он будет использоваться, если в сцене отсутствуют другие источники звука.

Создайте новый пустой объект `GameObject` и сделайте его дочерним по отношению к основному игровому диспетчеру. У этого нового объекта будет компонент `AudioSource`, используемый диспетчером управления звуком, присвойте ему имя `Audio`. Добавьте

к нему компонент AudioSource (на этот раз оставьте ползунок Spatial Blend в положении 2D, так как пользовательский интерфейс не имеет определенного положения в сцене) и добавьте в сценарий AudioManager код следующего листинга.

Листинг 11.7. Воспроизведение звуковых эффектов в диспетчере AudioManager

```
...
[SerializeField] private AudioSource soundSource;
...
public void PlaySound(AudioClip clip) {
    soundSource.PlayOneShot(clip);
}
...
```

Ячейка переменной на панели Inspector для ссылки на новый источник звука.

Воспроизводим звуки, не имеющие другого источника.

На панели Inspector появилась новая ячейка переменной; перетащите на нее объект Audio. Теперь добавим звуковой эффект UI к сценарию всплывающего окна.

Листинг 11.8. Добавление звуковых эффектов в сценарий SettingsPopup

```
...
[SerializeField] private AudioClip sound;
...
public void OnSoundToggle() {
   Managers.Audio.soundMute = !Managers.Audio.soundMute;
   Managers.Audio.PlaySound(sound);
}
...
```

Ячейка на панели Inspector для ссылки на звуковой клип.

Воспроизводим звуковой эффект при нажатии кнопки.

Перетащите звуковой эффект UI на ячейку переменной; я использовал 2D-звук thump. В результате теперь щелчок на кнопке UI сопровождается этим звуком (если, конечно, вы не уменьшили его громкость до нуля!). И хотя сам UI не имеет источника звука, диспетчер AudioManager воспроизводит нужный звуковой эффект через свой источник.

Замечательно, все звуковые эффекты настроены! Пришло время музыки.

11.4. Фоновая музыка

Вставим в игру фоновую музыку. Для этого ее нужно добавить в диспетчер AudioManager. В начале данной главы говорилось о том, что музыкальные клипы, по сути, не отличаются от звуковых эффектов. Цифровой звук представляет собой те же самые звуковые волны, команды его воспроизведения тоже по большей части одни и те же. Основное различие состоит в размере клипа, но именно этот фактор вызывает ряд последствий.

Во-первых, для музыкальных треков, как правило, требуется много компьютерной памяти, и этот параметр необходимо оптимизировать. Нужно отслеживать два аспекта: загрузка музыки в память до того, как она начнет использоваться, и потребление слишком большого объема памяти при загрузке.

Оптимизация *в процессе* загрузки выполняется методом Resources.Load(), с которым вы познакомились в главе 9. Если помните, он позволяет загружать ресурсы по имени. Это удобный инструмент, но есть и другие причины загружать ресурсы из папки Resources. К примеру, это реализация отложенной загрузки. Как правило, все ресурсы загружаются в Unity вместе со сценой, но ресурсов из папки Resources, ожидающих

извлечения с помощью кода, это не касается. Сейчас нас интересует *загрузка музыкальных клипов по требованию*. Без этого музыка займет слишком много места в памяти, причем еще до того, как ею начнут пользоваться.

ОПРЕДЕЛЕНИЕ *Загрузка по требованию* (lazy-loading) означает, что загрузка файла откладывается до момента, когда этот файл понадобится. Заранее загруженные данные реагируют быстрее (например, немедленно начинается воспроизведение звука), но загрузка по требованию экономит память в ситуациях, когда быстрая реакция не имеет особого значения.

Второй способ решения проблемы с потреблением памяти связан с потоковой передачей музыки с диска. Как объяснялось в разделе 11.1.2, потоковая передача звука позволяет компьютеру обойтись без загрузки файла целиком. Тип загрузки выбирается на панели Inspector импортированного аудиоклипа.

В конечном счете подготовка к воспроизведению музыки делится на несколько этапов, в том числе включающих в себя оптимизацию потребления памяти.

11.4.1. Музыкальные циклы

Процесс воспроизведения музыки представляет собой уже знакомую последовательность шагов, которой мы придерживались при программировании звуковых эффектов пользовательского интерфейса (фоновая музыка также является 2D-звуком, то есть не имеет источника в сцене). Поэтому мы просто повторим ее:

1. Импортируем аудиоклипы.
2. Настроим компонент `AudioSource` для использования диспетчером `AudioManager`.
3. Напишем код для воспроизведения аудиоклипа в диспетчере `AudioManager`.
4. Добавим элементы управления музыкой в пользовательский интерфейс.

Каждый шаг будет слегка модифицирован, так как теперь мы работаем с музыкой, а не со звуковыми эффектами.

Шаг 1. Импорт аудиоклипов

Скачайте или запишите музыкальные треки. В прилагаемом к книге примере проекта используются следующие композиции с сайта www.freesound.org:

- loop от пользователя Xythe/Ville Nousiainen;
- Intro Synth от пользователя noirenex.

Перетащите файлы в Unity и отредактируйте настройки импорта на панели Inspector. Я уже упоминал, что настройки аудиоклипов со звуковыми эффектами и с музыкой в общем случае различаются. Прежде всего, для звука со сжатием следует выбирать формат Vorbis. Напомню, что в результате сжатия размер файлов значительно уменьшается. Одновременно слегка ухудшается качество звука, но для длинных музыкальных клипов это вполне допустимый компромисс; установите ползунок Quality на отметку 50 %.

Затем нужно выбрать параметр Load Type. Еще раз напомню, что сейчас требуется чтение музыки с диска, а не ее полная загрузка. Выберите в меню Load Type вариант Streaming. Установите флажок Load In Background, чтобы в процессе загрузки музыки игра не останавливалась и не замедлялась.

И даже после всех этих настроек для корректной загрузки требуется еще поместить аудиофайл в нужное место. Надеюсь, вы помните, что метод `Resources.Load()` загружает только содержимое папки Resources. Создайте новую папку с этим именем, а внутри нее — папку Music и перетащите туда аудиофайлы, как показано на рис. 11.5.

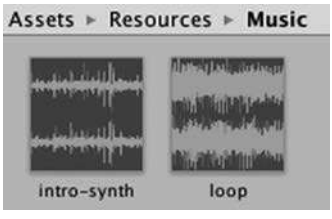


Рис. 11.5. Музыкальные аудиоклипы, помещенные в папку Resources

Шаг 2. Настройка компонента audiosource для использования диспетчером управления звуком

Теперь нужно создать новый компонент `AudioSource` для воспроизведения музыки. Создайте пустой объект `GameObject`, присвойте ему имя Music 1 (единица появилась потому, что позднее мы добавим объект Music 2) и сделайте его дочерним по отношению к объекту `Audio`.

Добавьте к объекту Music 1 компонент `AudioSource` и отредактируйте его параметры. Сбросьте флажок Play On Awake, но установите флажок Loop; если звуковой эффект обычно проигрывается один раз, то музыка, как правило, воспроизводится в цикле. Ползунок Spatial Blend оставьте в положении 2D, так как музыка не привязана к конкретной точке сцены.

Возможно, вы захотите уменьшить параметр Priority. Для звуковых эффектов мы оставили заданное по умолчанию значение `128`, но для музыки имеет смысл снизить его примерно до `60`. При наложении друг на друга различных звуков этот параметр указывает Unity, какие из звуков имеют больший приоритет. Причем, как ни странно, более низкие значения указывают на более высокий приоритет. При одновременном воспроизведении слишком большого количества звуков аудиосистема начинает пропускать некоторые из них; увеличив приоритет музыки, мы гарантируем, что она будет звучать, даже если одновременно запустится множество звуковых эффектов.

Шаг 3. Программирование механизма воспроизведения аудиоклипов в диспетчере управления звуком

Мы настроили источник аудио Music, поэтому нужно добавить код следующего листинга в сценарий `AudioManager`.

Листинг 11.9. Воспроизведение музыки в диспетчере AudioManager

```
...
[SerializeField] private AudioSource music1Source;

[SerializeField] private string introBGMusic;
[SerializeField] private string level1BGMusic;
```

Эти строки указывают имена музыкальных клипов.

```
...
public void PlayIntroMusic() {
  PlayMusic(Resources.Load("Music/"+introBGMusic) as AudioClip);
}
public void PlayLevelMusic() {
  PlayMusic(Resources.Load("Music/"+levelBGMusic) as AudioClip);
}

private void PlayMusic(AudioClip clip) {
  music1Source.clip = clip;
  music1Source.Play();
}

public void StopMusic() {
  music1Source.Stop();
}
...
```

Загрузка музыки intro из папки Resources.

Загрузка основной музыки из папки Resources.

Воспроизведение музыки при помощи параметра AudioSource.clip.

Как обычно, при выборе игрового диспетчера на панели Inspector можно увидеть новые сериализованные переменные. Перетащите на ячейку для источника аудио клип Music 1. Затем введите в две строковые переменные имена музыкальных файлов: intro-synth и loop.

Остальная часть нового кода вызывает команды загрузки и воспроизведения музыки (или в последнем добавленном методе — остановки музыки). Команда `Resources.Load()` загружает именованный ресурс из папки Resources (учитывая, что нужные файлы расположены во вложенной папке Music). Эта команда возвращает обобщенный объект, но его можно преобразовать к конкретному типу (в данном случае — к типу AudioClip), воспользовавшись ключевым словом as.

Затем загруженный аудиоклип передается в метод `PlayMusic()`, который настраивает клип в компоненте `AudioSource` и вызывает метод `Play()`. Как я уже объяснял, звуковые эффекты лучше воспроизводятся методом `PlayOneShot()`, но установка клипа в компоненте `AudioSource` является более надежным подходом, позволяющим, кроме всего прочего, на время или совсем останавливать воспроизведение музыки.

Шаг 4. Добавление в UI элементов управления музыкой

Чтобы новые методы воспроизведения музыки выполняли какие-либо действия, их нужно откуда-то вызывать. Добавим к нашему пользовательскому интерфейсу дополнительные кнопки, щелчки на которых позволят включать разную музыку. Сейчас я опять перечислю этапы процедуры с краткими пояснениями (подробности вы сможете найти в главе 7):

1. Измените ширину всплывающего окна на 350 (чтобы на нем поместились дополнительные кнопки).
2. Создайте новую кнопку и сделайте ее дочерним объектом по отношению к всплывающему окну.
3. Ширину кнопки сделайте равной 100 и поместите ее в точку с координатами 0, –20.
4. Раскройте иерархический список кнопки, выделите текстовую метку и поменяйте текст на Level Music.

5. Повторите эту последовательность еще два раза, чтобы создать две дополнительные кнопки.
6. Одну из них поместите в точку с координатами –105, –20, вторую — в точку с координатами 105, –20 (чтобы она появилась с другой стороны).
7. Текстовую метку первой кнопки поменяйте на Intro Music, второй — на No Music.

Теперь у нас есть три кнопки для воспроизведения различной музыки. Скопируйте метод из следующего листинга в сценарий `SettingsPopup`, который мы свяжем с каждой из кнопок.

Листинг 11.10. Добавление элементов управления музыкой в сценарий SettingsPopup

```
...
public void OnPlayMusic(int selector) {
  Managers.Audio.PlaySound(sound);

  switch (selector) {
  case 1:
    Managers.Audio.PlayIntroMusic();
    break;
  case 2:
    Managers.Audio.PlayLevelMusic();
    break;
  default:
    Managers.Audio.StopMusic();
    break;
  }
}
...
```

Этот метод получает от кнопки численный параметр.

Вызываем для каждой кнопки свою музыкальную функцию в диспетчере AudioManager.

Обратите внимание, что на этот раз функция принимает параметр типа `int`; как правило, связанные с кнопками методы лишены параметров и просто срабатывают по щелчку. Но сейчас кнопки должны различаться, именно поэтому каждой присваивается свой номер.При помощи стандартной процедуры свяжем кнопку с этим кодом: добавьте элемент в список OnClick на панели Inspector, перетащите всплывающее окно на ячейку объекта и выберите в меню подходящую функцию. На этот раз нам предоставят текстовое поле для ввода числа, так как метод `OnPlayMusic()` принимает в качестве параметра число. Введите 1 для Intro Music, 2 для Level Music и произвольное число для No Music (у меня это был ноль). Оператор `switch` в методе `OnMusic()` воспроизводит клип `intro` или `level` в зависимости от числа или останавливает воспроизведение, если число не равно ни 1, ни 2.

Щелкая на кнопках в процессе игры, вы сможете включать музыку. Великолепно! Код загружает аудиоклипы из папки `Resources`. Производительность воспроизведения оптимизирована. Осталось доработать две вещи: сделать отдельные элементы управления громкостью и обеспечить плавный переход при переключении музыки.

11.4.2. Отдельная регулировка громкости

В игре уже есть элемент управления громкостью, который в числе прочего влияет и на фоновую музыку. Но в большинстве игр громкость звуковых эффектов и музыки контролируется по отдельности. Давайте посмотрим, как этого достичь.

Первым делом нужно сделать так, чтобы компоненты `AudioSources` музыки игнорировали настройки компонента `AudioListener`. Мы хотим, чтобы регулятор громкости и кнопка отключения звука, связанные с глобальным компонентом `AudioListener`, продолжали влиять на все звуковые эффекты, но не затрагивали бы фоновую музыку. Листинг 11.11 содержит код, заставляющий источник музыки игнорировать громкость, задаваемую компонентом `AudioListener`. Заодно этот код создает регулятор громкости и возможность выключения музыки. Добавьте его к диспетчеру аудио.

Листинг 11.11. Раздельное управление громкостью музыки в диспетчере управления звуком

```
...
private float _musicVolume;
public float musicVolume {
  get {
    return _musicVolume;
  }
  set {
    _musicVolume = value;

    if (music1Source != null) {
      music1Source.volume = _musicVolume;
    }
  }
}
...
public bool musicMute {
  get {
    if (music1Source != null) {
      return music1Source.mute;
    }
    return false;
  }
  set {
    if (music1Source != null) {
      music1Source.mute = value;
    }
  }
}

public void Startup(NetworkService service) {
  Debug.Log("Audio manager starting...");

  _network = service;

  music1Source.ignoreListenerVolume = true;
  music1Source.ignoreListenerPause = true;

  soundVolume = 1f;
  musicVolume = 1f;

  status = ManagerStatus.Started;
}
...
```

Непосредственный доступ к закрытой переменной невозможен, только через функцию задания свойства.

Напрямую регулируем громкость источника звука.

Значение предлагается по умолчанию, если AudioSource отсутствует.

Выделенный курсивом код уже был в сценарии, тут он приведен для справки.

Эти свойства заставляют компонент AudioSource игнорировать громкость компонента AudioListener.

Выделенный курсивом код уже был в сценарии, тут он приведен для справки.

Этот код дает возможность непосредственно регулировать громкость в компоненте AudioSource, игнорируя источники глобальных параметров громкости, заданные в компоненте AudioListener. Существуют свойства для управления громкостью и включения/выключения звука отдельного источника музыки.

Метод `Startup()` инициализирует источник музыки со свойствами `ignoreListenerVolume` и `ignoreListenerPause`, имеющими значение `true`. Как следует из их названий, они заставляют источник звука игнорировать глобальные настройки громкости в компоненте `AudioListener`.

Вы можете нажать кнопку Play и удостовериться, что существующий элемент регулировки громкости на громкость музыки больше не влияет. Значит, в пользовательском интерфейсе требуется еще один элемент управления, который будет отвечать за громкость. Отредактируйте сценарий `SettingsPopup` в соответствии со следующим листингом.

Листинг 11.12. Элементы управления громкостью в сценарии SettingPopup

```
...
public void OnMusicToggle() {
  Managers.Audio.musicMute = !Managers.Audio.musicMute;
  Managers.Audio.PlaySound(sound);
}

public void OnMusicValue(float volume) {
  Managers.Audio.musicVolume = volume;
}
...
```

Повторяем элемент управления включением звука, но на этот раз используя musicMute.

Повторяем элемент управления громкостью звука, но на этот раз используя musicVolume.

Особых пояснений этот код не требует — по большей части он повторяет элементы управления громкостью звука. Бросается в глаза замена свойств объекта `AudioManager` с `soundMute`/`soundVolume` на `musicMute`/`musicVolume`.

В редакторе создайте кнопку и ползунок. Вот последовательность действий:

1. Измените высоту всплывающего окна на 225 (чтобы появилось место для дополнительных элементов управления).
2. Создайте кнопку для пользовательского интерфейса.
3. Сделайте ее потомком по отношению к всплывающему окну.
4. Поместите кнопку в точку с координатами 0, –60.
5. Раскройте иерархию кнопки и выделите текстовую метку.
6. Измените ее текст на Toggle Music.
7. Создайте ползунок (командой того же самого меню UI).
8. Сделайте ползунок потомком всплывающего окна и поместите его в точку с координатами 0, –85.
9. Присвойте параметру Value ползунка (в нижней части панели Inspector) значение 1.

Свяжите эти элементы пользовательского интерфейса с кодом в сценарии `SettingsPopup`. Найдите список OnClick/OnValueChanged в настройках UI-элементов, щелкните на кнопке со знаком + (плюс) для добавления новой записи и перетащите всплывающее окно на

ячейку для объекта, после чего выберите в меню функцию. В первом случае это будет вариант OnMusicToggle(), во втором — OnMusicValue(). Обе функции находятся в разделе Dynamic Float.

Теперь запустите код и убедитесь, что на звуковые эффекты и фоновую музыку влияют разные элементы управления. Но на самом деле это еще не все. Остался последний штрих — плавное микширование музыкальных треков.

11.4.3. Переход между песнями

В качестве финального штриха заставим диспетчер управления звуком постепенно уменьшать и увеличивать громкость при переходе от одной мелодии к другой. Сейчас в процессе переключения мелодия просто обрывается и начинается новый трек, что несколько режет ухо. Этот переход можно сгладить, сделав так, чтобы громкость предыдущего трека быстро затухала, в то время как громкость нового увеличивалась от нуля. Это простой, но хорошо продуманный код, объединяющий изученные вами методы регулировки громкости с сопрограммой, постепенно меняющей громкость.

Листинг 11.13 содержит код, который следует добавить в сценарий `AudioManager`. Большая его часть сводится к простой концепции: у нас есть два отдельных источника звука, мы пользуемся ими для воспроизведения отдельных музыкальных треков и пошагово увеличиваем громкость одного источника, одновременно пошагово уменьшая громкость другого (как обычно, выделенный курсивом код уже присутствует в сценарии и показан для справки).

Листинг 11.13. Плавный переход между треками в сценарии AudioManager

```
...
[SerializeField] private AudioSource music2Source;

private AudioSource _activeMusic;
private AudioSource _inactiveMusic;

public float crossFadeRate = 1.5f;
private bool _crossFading;
...
public float musicVolume {
  ...
  set {
    _musicVolume = value;

    if (music1Source != null && !_crossFading) {
      music1Source.volume = _musicVolume;
      music2Source.volume = _musicVolume;
    }
  }
}
...
public bool musicMute {
  ...
  set {
   if (music1Source != null) {
     music1Source.mute = value;
```

Второй компонент AudioSource (первый тоже сохраняем).

Следим за тем, какой из источников активен, а какой нет.

Переключатель, позволяющий избежать ошибок в процессе перехода.

Регулировка громкости обоих источников музыки.

```
      music2Source.mute = value;
    }
  }
}

public void Startup(NetworkService service) {
  Debug.Log("Audio manager starting...");

  _network = service;

  music1Source.ignoreListenerVolume = true;
  music2Source.ignoreListenerVolume = true;
  music1Source.ignoreListenerPause = true;
  music2Source.ignoreListenerPause = true;

  soundVolume = 1f;
  musicVolume = 1f;

  _activeMusic = music1Source;
  _inactiveMusic = music2Source;

  status = ManagerStatus.Started;
}
...
private void PlayMusic(AudioClip clip) {
  if (_crossFading) {return;}
    StartCoroutine(CrossFadeMusic(clip));
  }
private IEnumerator CrossFadeMusic(AudioClip clip) {
  _crossFading = true;

  _inactiveMusic.clip = clip;
  _inactiveMusic.volume = 0;
  _inactiveMusic.Play();

  float scaledRate = crossFadeRate * _musicVolume;
  while (_activeMusic.volume > 0) {
    _activeMusic.volume -= scaledRate * Time.deltaTime;
    _inactiveMusic.volume += scaledRate * Time.deltaTime;

    yield return null;
  }

  AudioSource temp = _activeMusic;

  _activeMusic = _inactiveMusic;
  _activeMusic.volume = _musicVolume;

  _inactiveMusic = temp;
  _inactiveMusic.Stop();

  _crossFading = false;
}

public void StopMusic() {
  _activeMusic.Stop();
  _inactiveMusic.Stop();
}
...
```

Инициализируем один из источников как активный.

При изменении музыкальной композиции вызываем сопрограмму.

Этот оператор yield останавливает операции на один кадр.

Временная переменная, используемая, когда мы меняем местами переменные _active и _inactive.

Первым дополнением стала переменная для второго источника музыки. Нужно сохранить первый объект `AudioSource`, но создать его копию (убедитесь, что настройки остались теми же, — установите флажок Loop), а затем перетащить его на ячейку панели Inspector. При этом код задает переменные для активного и неактивного состояний источника звука, но это закрытые переменные, которые не появляются на панели Inspector. Именно они определяют, музыка из какого источника воспроизводится в конкретный момент времени.

Теперь в процессе воспроизведения музыки код вызывает сопрограмму. Она заставляет второй объект, `AudioSource`, начать воспроизведение нового музыкального трека, в то время как старый воспроизводится на втором источнике звука. После этого сопрограмма пошагово увеличивает громкость новой музыки, одновременно уменьшая громкость старой. После завершения перехода (то есть в момент, когда уровни громкости поменялись местами) сопрограмма меняет местами «активный» и «неактивный» источники.

Потрясающе! Мы добавили фоновую музыку в аудиосистему игры.

FMOD: ЗВУКОВОЙ ИНСТРУМЕНТ ДЛЯ ИГР

Аудиосистема в Unity приводится в действие популярной программной аудиобиблиотекой FMOD. Ее можно скачать с сайта www.fmod.org, но в Unity она уже встроена, хотя и лишена наиболее нетривиальных функциональных возможностей (о них можно узнать на сайте библиотеки).

Эти нетривиальные функциональные возможности предлагает подключаемый к Unity модуль FMOD Studio, но для выполнения упражнений этой главы более чем достаточно предоставленного по умолчанию набора, в который входят самые необходимые для создания аудиосистемы в играх компоненты. Они удовлетворяют нуждам большинства разработчиков, дополнительный же модуль требуется только для добавления в игры сложных звуковых эффектов.

Заключение

- Для звуковых эффектов следует использовать несжатые аудиофайлы, а для фоновой музыки — сжатые, но в качестве исходного материала в обоих случаях можно брать файлы формата WAV, так как Unity умеет сжимать импортированный звук.
- Для аудиоклипов возможен как 2D-звук, всегда воспроизводимый одинаково, так и 3D-звук, громкость которого зависит от положения слушателя.
- Громкость звуковых эффектов легко регулируется на глобальном уровне с помощью компонента `AudioListener`.
- Можно по отдельности регулировать громкость отдельных источников звука.
- Вы можете выполнять переходы между треками фоновой музыки, задавая громкость отдельных источников звука.

12 Объединение фрагментов в готовую игру

- ✓ Сборка объектов и кода из других проектов.
- ✓ Программирование элементов управления с помощью мыши.
- ✓ Обновление пользовательского интерфейса при переходе от старой системы к новой.
- ✓ Загрузка новых уровней в ответ на достижение поставленных целей.
- ✓ Настройка условий выигрыша/проигрыша.
- ✓ Сохранение и загрузка текущего состояния игры.

В этой главе мы соберем воедино все ранее созданное. Большинство предыдущих глав содержало разрозненный материал, и не было необходимости рассматривать игру в целом. Но сейчас мы рассмотрим процесс объединения разработанных фрагментов, чтобы вы знали, каким образом строится игра. Обсудим мы и обобщенную структуру игры, в том числе переход с одного уровня на другой и процесс завершения (например, появление надписи Game Over, когда персонаж умирает, или надписи Success, когда он доходит до выхода). Я покажу, каким образом можно сохраниться, потому что по мере увеличения объема игры растет и важность сохранения полученных игроком результатов.

ВНИМАНИЕ В этой главе по большей части рассматриваются задачи, подробности решения которых объяснялись в предыдущих главах, поэтому я ограничусь общими рекомендациями. Если вы не понимаете каких-то вещей, перечитайте соответствующую главу (например, главу 7, если у вас возникли сложности с элементами пользовательского интерфейса).

Рассмотрим в качестве проекта ролевой боевик. В таких играх камера располагается сверху и смотрит четко вниз, как показано на рис. 12.1, а направления перемещений персонажа указываются щелчками мыши. Возможно, вы знакомы с игрой *Diablo*,

которая представляет собой как раз ролевой боевик. Я специально перешел к новому жанру, чтобы познакомить вас с как можно большим числом различных типов игр!

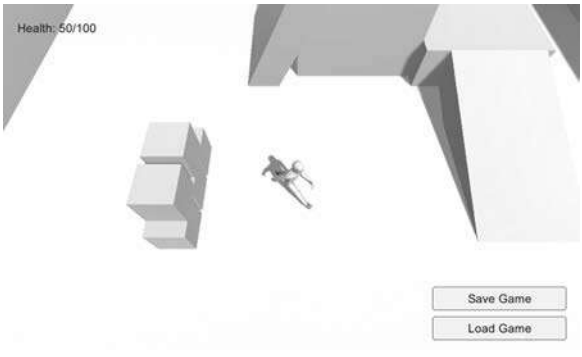


Рис. 12.1. Снимок при наблюдении сверху вниз

С такими масштабными играми, как в этой главе, вы еще не работали. Мы рассмотрим следующие темы:

- Вид на сцену сверху и перемещения методом наведения и щелчка.
- Возможность управлять устройствами путем щелчка на них.
- Разбросанные элементы, которые можно собирать.
- Инвентарь, отображаемый в окне UI.
- Бродящие по уровню противники.
- Возможность сохранять игру и возобновлять ее с прерванной точки.
- Три уровня, которые нужно завершать по очереди.

Как видите, вам предстоит насыщенная программа; к счастью, это практически последняя глава!

12.1. Изменение назначения проектов для получения ролевого боевика

Основой нашего ролевого боевика послужит проект из главы 9. Скопируйте его папку и откройте в Unity. Те, кто пропустил данную главу, могут просто скачать соответствующий пример проекта.

Этот проект выбран в качестве основы, потому что он наиболее полно отвечает нашим целям, а значит, требует наименьшего количества модификаций (в сравнении с остальными проектами). В конечном счете вместе будут сведены ресурсы из разных глав, так что с технической точки зрения нет никакой разницы, откуда начинать.

Вот краткий список фрагментов проекта из главы 9:

- Персонаж с настроенным контроллером анимации.
- Камера, следующая за персонажем.

- Уровень с полом, стенами и наклонными поверхностями.
- Источники света и тени.
- Работающие устройства, в том числе монитор, меняющий цвет.
- Инвентарь, который можно собирать.
- Фреймворк из интерфейсных диспетчеров.

Как видите, большая часть работы по созданию демонстрационной версии ролевой игры уже выполнена, но остались детали, которые требуется отредактировать или добавить.

12.1.1. Сборка ресурсов и кода из разных проектов

Первым делом следует обновить фреймворк диспетчеров и добавить в проект противников, управляемых компьютером. Первую задачу мы решали в главе 10, добавляя во фреймворк из главы 9 новые детали. Программированию противников была посвящена глава 3.

Обновление фреймворка диспетчеров

Проще всего обновить диспетчеры, поэтому эту задачу мы решим первой. Интерфейс `IGameManager` редактировался в главе 10.

Листинг 12.1. Отредактированный интерфейс IGameManager

```
public interface IGameManager {
  ManagerStatus status {get;}

  void Startup(NetworkService service);
}
```

В коде этого листинга появилась ссылка на сценарий `NetworkService`, значит, его нужно скопировать в проект. Перетащите в папку нового проекта файл со сценарием из проекта главы 10. Мы поменяли интерфейс, с которым работает сценарий `Managers.cs`, поэтому давайте его отредактируем.

Листинг 12.2. Изменения в сценарии Managers

```
...
private IEnumerator StartupManagers() {  <-- Исправления в начале метода.
  NetworkService network = new NetworkService();

  foreach (IGameManager manager in _startSequence) {
    manager.Startup(network);
  }
...
```

Напоследок отредактируем сценарии `InventoryManager` и `PlayerManager` с учетом внесенных в интерфейс изменений. Следующий листинг демонстрирует исправления в сценарии `InventoryManager`; аналогичные правки, но с другими именами, вносятся и в сценарий `PlayerManager`.

Листинг 12.3. Изменения в сценарии InventoryManager с учетом модификаций в IGameManager

```
...
private NetworkService _network;

public void Startup(NetworkService service) {
  Debug.Log("Inventory manager starting...");

  _network = service;

  _items = new Dictionary<string, int>();
...
```

Одни и те же правки в обоих сценариях, просто с разными именами.

После этих небольших изменений игра должна функционировать так же, как и раньше. Мы скорректировали внутренние особенности, а игровой процесс остался без изменений. Это была самая простая часть, дальше будет сложнее.

Копирование противников, оснащенных искусственным интеллектом

Нам нужно не только отредактировать сценарий `NetworkServices` из главы 10, но и добавить в проект обладающего искусственным интеллектом противника из главы 3. Реализация такого персонажа требует целого набора сценариев и графических ресурсов, которые сейчас нужно импортировать.

Первым делом скопируйте все сценарии (напоминаю, что в сценариях `WanderingAI` и `ReactiveTarget` программировалось поведение противника, сценарий `Fireball` определял снаряд, которым противник атаковал компонент `PlayerCharacter`, а сценарий `SceneController` отвечал за порождение новых противников):

- `PlayerCharacter.cs`
- `SceneController.cs`
- `WanderingAI.cs`
- `ReactiveTarget.cs`
- `Fireball.cs`

Заодно импортируем в проект материал Flame и шаблоны Fireball и Enemy. Тем, кто предпочитает использовать противника из главы 11, а не из главы 3, понадобится также материал fire particle.

При импорте обычно нарушаются связи между ресурсами, и их нужно восстановить, чтобы все снова начало работать. В частности, сценарии, скорее всего, некорректно связаны с шаблонами экземпляров. Например, на панели Inspector можно увидеть, что у шаблона Enemy отсутствуют два сценария. Чтобы исправить ошибку, щелкните на кнопке в виде окружности, как показано на рис. 12.2, и выберите в списке сценариев варианты WanderingAI и ReactiveTarget. Аналогичным образом проверьте шаблон Fireball и, если нужно, повторно соедините его со сценарием. После этого проверьте ссылки на материалы и текстуры.

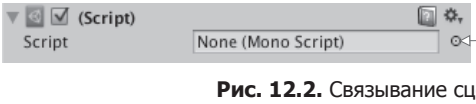


Рис. 12.2. Связывание сценария с компонентом

К объекту-контроллеру добавьте сценарий `SceneController.cs` и перетащите шаблон Enemy на одноименную ячейку компонента панели Inspector. Возможно, потребуется перетащить шаблон Fireball на компонент сценария объекта Enemy (выделите шаблон Enemy и на панели Inspector посмотрите поле WanderingAI). Кроме того, свяжите сценарий `PlayerCharacter.cs` с объектом `player`, чтобы противники начали атаковать игрока.

Запустите игру и посмотрите, как двигается противник. Огненные шары летят в персонажа, пока не причиняя особого вреда; выделите шаблон Fireball и присвойте его параметру Damage значение 10.

ПРИМЕЧАНИЕ Пока что точность слежения за персонажем и попыток его поразить у противника невелика. Я начал бы исправление ситуации с расширения сектора обзора врага (воспользовавшись для этого скалярным произведением, как было показано в главе 9). Вам предстоит потратить много времени на доработку игры. К ней относится и редактирование поведения противников. Этот процесс крайне важен для окончательной версии, но в книге мы им заниматься не будем.

В главе 3 здоровье персонажа фигурировало только в качестве тестового атрибута. Но теперь в игре есть диспетчер персонажа, а значит, можно отредактировать сценарий `PlayerCharacter`, добавив в него средства для управления здоровьем.

Листинг 12.4. Добавляем в сценарий PlayerCharacter возможность использовать здоровье в диспетчере игрока

```
using UnityEngine;
using System.Collections;

public class PlayerCharacter : MonoBehaviour {
  public void Hurt(int damage) {
    Managers.Player.ChangeHealth(-damage);
  }
}
```

Используйте в диспетчере PlayerManager это значение вместо переменной в объекте PlayerCharacter.

Итак, мы собрали из фрагментов ранее выполненных проектов демонстрационный ролик. В сцене появился противник, что сделало игру более захватывающей. Но элементы управления и угол обзора до сих пор такие же, как и в демонстрационном ролике от третьего лица. Нужно создать для нашей ролевой игры элементы управления с помощью мыши (point-and-click controls).

12.1.2. Элементы управления с помощью мыши

Демонстрационному ролику требуется камера, нацеленная сверху вниз, и управление перемещениями персонажа с помощью мыши (см. рис. 12.1). В настоящее время мышь управляет камерой, в то время как перемещения персонажа контролируются с клавиатуры (это мы запрограммировали в главе 8), то есть это диаметрально противоположно тому, что нам нужно. Кроме того, мы заставим реагировать на щелчки мыши меняющий цвета монитор. Огромного количества правок в обоих случаях код не требует, поэтому давайте просто возьмем и внесем необходимые коррективы в сценарии движения и устройства.

Обзор сцены сверху вниз

Первым делом присвойте координате *Y* камеры значение 8, чтобы поднять ее над сценой. Кроме того, давайте уберем из сценария `OrbitCamera` управление с помощью мыши и оставим в качестве элементов управления только клавиши со стрелками.

Листинг 12.5. Удаление средств управления с помощью мыши из сценария OrbitCamera

```
...
void LateUpdate() {
  _rotY -= Input.GetAxis("Horizontal") * rotSpeed;  <-- Меняем направление на обратное.
  Quaternion rotation = Quaternion.Euler(0, _rotY, 0);
  transform.position = target.position - (rotation * _offset);
  transform.LookAt(target);
}
...
```

БЛИЖНЯЯ/ДАЛЬНЯЯ ПЛОСКОСТИ ОТСЕЧКИ КАМЕРЫ

Так как дело дошло до настроек камеры, я хотел бы упомянуть о такой вещи, как *ближняя / дальняя плоскости отсечки*. Раньше эти параметры не рассматривались, так как нам прекрасно подходили их значения по умолчанию, но в других проектах они могут потребоваться.

Выделите камеру и обратите внимание на настройку Clipping Planes на панели Inspector; именно здесь указываются оба значения: Near и Far. Они задают переднюю и заднюю границы, внутри которых происходит визуализация сеток: полигоны, оказавшиеся ближе, чем задано значением Near, и дальше, чем задано значением Far, отсекаются.

Значения параметров Near / Far должны быть, с одной стороны, как можно ближе друг к другу, с другой — отстоять друг от друга достаточно далеко для визуализации сцены в целом. При слишком большом расстоянии между этими плоскостями (ближняя располагается слишком близко, а дальняя — слишком далеко) алгоритм визуализации перестает различать, какие полигоны ближе. В результате возникает ошибка визуализации, называемая *z-конфликтом* (z-fighting), когда полигоны мерцают один поверх другого.

Мы подняли камеру повыше, и теперь при воспроизведении игры сцена будет демонстрироваться сверху. Но движение все еще управляется с клавиатуры, поэтому давайте напишем сценарий для перемещений путем наведения и щелчка.

Код движения

Основная идея этого кода (проиллюстрированная на рис. 12.3) сводится к автоматическому перемещению персонажа в указанную точку. Эта точка задается щелчком мыши. При этом код, перемещающий персонажа, напрямую на мышь не реагирует, движение персонажа косвенно управляется при помощи щелчков.

ПРИМЕЧАНИЕ Описанный здесь алгоритм перемещения можно использовать для персонажей с искусственным интеллектом. Однако в этом случае целевая точка не задается щелчками мыши, а просто находится на заданной траектории.

Создайте сценарий `PointClickMovement` и скопируйте в него код сценария `RelativeMovement` (ведь он должен реализовывать падения и анимацию). Замените компонент `RelativeMovement` объекта `player`. Затем отредактируйте код нового сценария в соответствии с листингом 12.6.

Рис. 12.3. Схема работы элементов управления с помощью мыши

Листинг 12.6. Новый код движения в сценарии PointClickMovement

```
...
public class PointClickMovement : MonoBehaviour {   ← Исправим имя после вставки кода.
...
public float deceleration = 25.0f;
public float targetBuffer = 1.5f;
private float _curSpeed = 0f;
private Vector3 _targetPos = Vector3.one;
...
void Update() {                                          Бросаем луч
  Vector3 movement = Vector3.zero;                       в точку щелчка.

  if (Input.GetMouseButton(0)) {   ← Задаем целевую точку по щелчку мыши.
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit mouseHit;
    if (Physics.Raycast(ray, out mouseHit)) {
      _targetPos = mouseHit.point;   ← Устанавливаем цель в точке попадания луча.
      _curSpeed = moveSpeed;
    }                                                    Поворачиваемся по
  }                                                      направлению к цели только
                                                         во время быстрого движения.
  if (_targetPos != Vector3.one) {   ← Если целевая точка задана, перемещаем.
    if (_curSpeed > moveSpeed * .5f) {
      Vector3 adjustedPos = new Vector3(_targetPos.x, transform.position.y,
                            _targetPos.z);
      Quaternion targetRot = Quaternion.LookRotation(adjustedPos -
                             transform.position);
      transform.rotation = Quaternion.Slerp(transform.rotation, targetRot,
                           rotSpeed * Time.deltaTime);
    }
    movement = _curSpeed * Vector3.forward;
    movement = transform.TransformDirection(movement);

    if (Vector3.Distance(_targetPos, transform.position) < targetBuffer) {
      _curSpeed -= deceleration * Time.deltaTime;   ← При приближении к цели
      if (_curSpeed <= 0) {                           снижаем скорость до 0.
        _targetPos = Vector3.one;
      }
    }                                                    Дальше код остается
  }                                                      без изменений.
  _animator.SetFloat("Speed", movement.sqrMagnitude);
...
```

Мы практически полностью убрали начало метода `Update()`, так как этот код отвечал за управление перемещением с клавиатуры. Обратите внимание, что новый код содержит два основных оператора `if`: один выполняется при щелчке мышью, второй — после задания целевой точки.

Целевая точка задается в месте щелчка мышью. Именно тут пригодится метод испускания луча: он позволит определить, какая точка сцены попала под указатель. Место попадания луча фиксируется как целевая точка.

Второй условный оператор первым делом обеспечивает поворот персонажа лицом к целевой точке. Метод `Quaternion.Slerp()` выполняет этот поворот плавно, а не скачком. Кроме того, при замедлении персонажа поворот блокируется (иначе персонаж, почти достигший цели, будет совершать странные движения вокруг своей оси). Затем координаты, задающие направление движения персонажа, преобразуются от локальных к глобальным (для того, чтобы пойти вперед). В конце проверяется расстояние между персонажем и целью: если персонаж почти достиг нужной точки, его скорость постепенно уменьшается, а в конце происходит удаление целевой точки.

УПРАЖНЕНИЕ: ОТКЛЮЧАЕМ УПРАВЛЕНИЕ ПРЫЖКАМИ

Сейчас в сценарии присутствует элемент управления прыжками, скопированный из сценария `RelativeMovement`. При нажатии клавиши Space персонаж подпрыгивает, что недопустимо, когда перемещения управляются мышью. Поэтому самостоятельно отредактируйте код в условной инструкции `if (hitGround)`.

Итак, теперь наш персонаж управляется мышью. Запустите игру, чтобы посмотреть, как это выглядит на практике. Теперь давайте заставим реагировать на мышь наши устройства.

Управление устройствами с помощью мыши

В главе 9 управление устройствами осуществлялось с клавиатуры. Нам же нужно, чтобы они управлялись мышью. Для этого создадим сценарий, от которого будут наследовать все устройства; именно туда мы поместим процедуру управления посредством мыши. Присвойте новому сценарию имя `BaseDevice` и скопируйте в него код следующего листинга.

Листинг 12.7. Сценарий BaseDevice, срабатывающий по щелчку мыши

```
using UnityEngine;
using System.Collections;

public class BaseDevice : MonoBehaviour {
  public float radius = 3.5f;

  void OnMouseDown() {  ◄—— Функция, запускаемая щелчком.
    Transform player = GameObject.FindWithTag("Player").transform;
    if (Vector3.Distance(player.position, transform.position) < radius) {
      Vector3 direction = transform.position - player.position;
      if (Vector3.Dot(player.forward, direction) > .5f) {
```

```
        Operate();
      }
    }
  }

  public virtual void Operate() {
    // здесь код поведения конкретного устройства
  }
}
```

Если персонаж рядом с устройством и стоит к нему лицом, вызываем метод Operate().

Ключевое слово virtual указывает на метод, который можно переопределить после наследования.

Большая часть операций выполняется внутри метода `OnMouseDown()`, так как именно его вызывает класс `MonoBehaviour` после щелчка на объекте. Первым делом проверяется расстояние до персонажа, а затем с помощью скалярного произведения определяется, повернут ли он в сторону устройства. Метод `Operate()` пока представляет собой пустую оболочку, которая будет заполняться кодом устройств, наследующих данный сценарий.

ПРИМЕЧАНИЕ Этот код ищет в сцене объект с тегом Player, поэтому назначьте данный тег объекту `player`. Раскрывающийся список Tag находится в верхней части панели Inspector; можно задать свой тег, но среди тегов, предлагаемых по умолчанию, уже есть нужный нам вариант. Выделите объект `player` и затем выберите для него в меню тег Player.

Теперь, когда у нас есть сценарий `BaseDevice`, можно отредактировать сценарий `ColorChangeDevice` в соответствии со следующим листингом.

Листинг 12.8. Добавляем в сценарий ColorChangeDevice код наследования от сценария BaseDevice

```
using UnityEngine;
using System.Collections;

public class ColorChangeDevice : BaseDevice {
  public override void Operate() {
    Color random = new Color(Random.Range(0f,1f),
      Random.Range(0f,1f), Random.Range(0f,1f));
    GetComponent<Renderer>().material.color = random;
  }
}
```

Наследование от BaseDevice, а не от MonoBehaviour.

Переопределим этот метод из базового класса.

Благодаря наследованию от класса `BaseDevice`, а не от класса `MonoBehaviour`, сценарий получает функциональность в виде возможности управления при помощи мыши. Затем он переопределяет пустой метод `Operate()`, добавляя туда поведение, меняющее цвет монитора.

Теперь устройство управляется щелчками мыши. Кроме того, у персонажа был удален компонент сценария `DeviceOperator`, так как этот сценарий задает управление устройством с клавиатуры.

К сожалению, новый вариант управления устройством конфликтует с элементами управления перемещениями. Ведь целевая точка тоже задается щелчком мыши, но мы не хотим, чтобы она появлялась в момент щелчка на устройствах. Эту проблему помогают решить слои; аналогично тому, как мы присвоили персонажу тег, объекты

можно распределить по разным слоям. Добавим в сценарий `PointClickMovement` код, проверяющий, к какому слою принадлежит объект.

Листинг 12.9. Корректировка кода, обрабатывающего щелчки мышью, в сценарии PointClickMovement

```
...
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit mouseHit;
if (Physics.Raycast(ray, out mouseHit)) {
  GameObject hitObject = mouseHit.transform.gameObject;
  if (hitObject.layer == LayerMask.NameToLayer("Ground")) {
    _targetPos = mouseHit.point;
    _curSpeed = moveSpeed;
  }
}
...
```

Добавленный код; остальное приведено для справки.

Листинг добавляет в код обработки щелчков мыши условный оператор, проверяющий, принадлежит ли слою Ground объект, на котором был сделан щелчок. Раскрывающийся список Layers (как и список Tags) находится в верхней части панели Inspector; раскройте его, чтобы посмотреть доступные варианты. Как и в случае с тегами, несколько слоев заданы по умолчанию. В данном случае требуется новый слой, поэтому выберите в меню вариант Add Layer. Введите в пустое поле (например, в поле 8) название Ground. Присутствующий в коде метод `NameToLayer()` преобразует имена в номера слоев, что дает возможность использовать имя вместо номера.

Теперь, когда в меню появился слой Ground, поместите в него все объекты, по которым может ходить персонаж, то есть пол здания вместе с наклонными поверхностями и платформами. Выделите все эти объекты и выберите в меню Layers вариант Ground.

Запустите игру и убедитесь, что щелчки на меняющем цвет мониторе не приводят персонаж в движение. Великолепно! Работа над элементами наведения и щелчка закончена. В наш проект осталось добавить только импортированный пользовательский интерфейс.

12.1.3. Замена старого GUI новым

В главе 9 использовался графический пользовательский интерфейс непосредственного режима, так как его было проще запрограммировать. Но он выглядит не так красиво, как интерфейс из главы 7, поэтому давайте воспользуемся новой системой. Новый интерфейс лучше доработан визуально. Рисунок 12.4 демонстрирует, что именно нам предстоит создать.

Начнем с настройки графики для UI. Как только все изображения элементов UI окажутся в сцене, можно соединить с ними соответствующие сценарии. Я перечислю этапы создания, не вдаваясь в детали; их вы можете вспомнить самостоятельно, обратившись к главе 7:

1. Импортируйте изображение `popup.png` как спрайт (выберите нужный вариант в списке Texture Type).
2. В окне диалога Sprite Editor задайте со всех сторон границы размером 12 пикселов (не забудьте зафиксировать изменения кнопкой Apply).

Рис. 12.4. Вид UI для проекта этой главы

3. Создайте холст (GameObject > UI > Canvas).
4. Установите для холста флажок Pixel Perfect.
5. По желанию: присвойте объекту имя HUD Canvas и переключитесь в режим отображения 2D.
6. Создайте связанный с холстом текст (GameObject > UI > Text).
7. Задайте для объекта Text привязку к верхнему левому углу и положение `100`, `-40`.
8. В качестве текста метки введите Health.
9. Создайте связанное с холстом изображение (GameObject > UI > Image).
10. Присвойте новому объекту имя Inventory Popup.
11. Назначьте спрайт всплывающего окна ячейке Source Image изображения.
12. Выберите в меню Image Type вариант Sliced и установите флажок Fill Center.
13. Расположите изображение всплывающего окна в точке с координатами `0`, `0` и сделайте его ширину равной `250`, а высоту — `150`.

ПРИМЕЧАНИЕ Напоминаю, что для перехода от просмотра трехмерной сцены к просмотру двумерного интерфейса нужно нажать кнопку режима 2D view и дважды щелкнуть на объекте Canvas или Building, чтобы объект поместился в границы окна.

В углу появилась метка Health, а в центре — большое всплывающее окно голубого цвета. Давайте запрограммируем эти элементы. Код интерфейса будет использовать уже знакомую по главе 7 систему диспетчеров, поэтому скопируйте сценарий `Messenger`. Затем создайте сценарий `GameEvent` и введите в него код следующего листинга.

Листинг 12.10. Сценарий GameEvent, который будет использоваться с системой диспетчеров

```
public static class GameEvent {
  public const string HEALTH_UPDATED = "HEALTH_UPDATED";
}
```

Пока определено только одно событие; постепенно мы добавим еще несколько. Разошлите сообщение об этом событии из сценария `PlayerManager`, как показано в следующем листинге.

Листинг 12.11. Рассылка сообщения health из сценария PlayerManager.cs

```
...
public void ChangeHealth(int value) {
  health += value;
  if (health > maxHealth) {
    health = maxHealth;
  } else if (health < 0) {
    health = 0;
  }

  Messenger.Broadcast(GameEvent.HEALTH_UPDATED);  <-- Добавляем строку в конец этой функции.
}
...
```

Это сообщение рассылается каждый раз, когда метод `ChangeHealth()` завершает свою работу, сообщая остальной программе об изменении параметра health. В качестве реакции на это событие должна меняться метка health, поэтому создайте сценарий `UIController` и введите в него код следующего листинга.

Листинг 12.12. Обслуживающий интерфейс сценарий UIController

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class UIController : MonoBehaviour {
  [SerializeField] private Text healthLabel;  <-- Ссылаемся на UI-объект в сцене.
  [SerializeField] private InventoryPopup popup;

  void Awake() {  <-- Задаем подписчика для события обновления здоровья.
    Messenger.AddListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  }
  void OnDestroy() {
    Messenger.RemoveListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  }

  void Start() {
    OnHealthUpdated();  <-- При загрузке вызываем функцию вручную.

    popup.gameObject.SetActive(false);  <-- Инициализирует всплывающее
  }                                         окно как скрытое.

  void Update() {
    if (Input.GetKeyDown(KeyCode.M)) {  <-- Отображаем всплывающее окно клавишей M.
       bool isShowing = popup.gameObject.activeSelf;
       popup.gameObject.SetActive(!isShowing);
       popup.Refresh();
    }
```

```
  }
                                          Подписчик события вызывает функцию
                                          для обновления метки health.
  private void OnHealthUpdated() {
     string message = "Health: " + Managers.Player.health + "/" + Managers.
                      Player.maxHealth;
     healthLabel.text = message;
  }
}
```

Присоедините этот сценарий к объекту `Controller` и удалите сценарий `BasicUI`. Кроме того, создайте сценарий `InventoryPopup` (добавьте в него пустой открытый метод `Refresh()`; остальной код мы напишем позже) и свяжите его со всплывающим окном (это объект `Image`). Теперь можно перетащить всплывающее окно на ячейку для ссылки в компоненте `Controller`; свяжите с этим компонентом еще и метку health.

Эта метка меняется при ранении персонажа и при использовании им пакетов здоровья, а при нажатии клавиши M всплывающее окно становится видимым. Остается скорректировать одну деталь. Щелчок на всплывающем окне вызывает движение персонажа, как и в случае с устройствами. Но щелчок на UI-элементе не должен приводить к заданию целевой точки. Поэтому внесите показанные в следующем листинге изменения в сценарий `PointClickMovement`.

Листинг 12.13. Проверка UI в сценарии PointClickMovement

```
using UnityEngine.EventSystems;
...
void Update() {
  Vector3 movement = Vector3.zero;
  if (Input.GetMouseButton(0) && !EventSystem.current.IsPointerOverGameObject()) {
...
```

Обратите внимание на условный оператор, проверяющий местоположение указателя мыши в момент щелчка. На этом работу над общей структурой интерфейса можно считать завершенной, поэтому давайте перейдем к всплывающему окну с инвентарем.

Всплывающий список инвентаря

Пока у нас есть только пустое всплывающее окно, в то время как там должен отображаться список игрового инвентаря, как показано на рис. 12.5. Вот последовательность создания этих объектов UI:

1. Создайте четыре изображения и сделайте их потомками всплывающего окна (путем перетаскивания на вкладке Hierarchy).
2. Создайте четыре текстовые метки и сделайте их потомками всплывающего окна.
3. Расположите изображения в точках с координатой *Y*, равной `0`, и координатами *X*, равными `-75`, `-25`, `25` и `75`.
4. Расположите текстовые метки в точках с координатой *Y*, равной `50`, и координатами *X*, равными `-75`, `-25`, `25` и `75`.
5. Выберите для текста (не для привязки!) выравнивание по горизонтали Center, выравнивание по вертикали, а высоту сделайте равной 60.

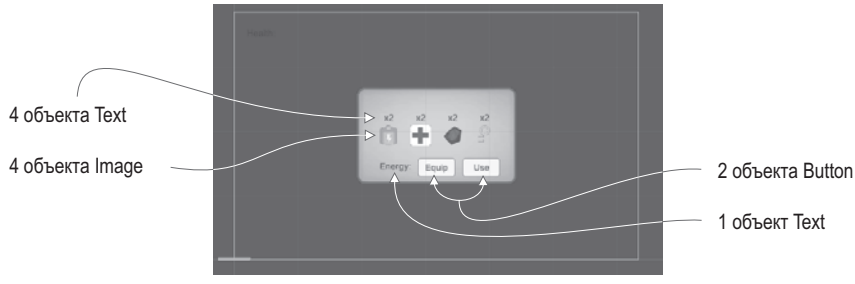


Рис. 12.5. Схема UI для отображения инвентаря

6. В папке `Resources` превратите все значки инвентаря в спрайты (изначально они являются текстурами).
7. Перетащите эти спрайты на ячейку Source Image объектов Image (заодно щелкните на кнопке Set Native Size).
8. Введите `x2` для всех текстовых меток.
9. Добавьте еще одну текстовую метку и две кнопки, сделав их потомками всплывающего окна.
10. Расположите текстовую метку в точке с координатами `-120`, `-55` и выберите для выравнивания по горизонтали вариант Right.
11. В качестве текста метки введите Energy.
12. Присвойте параметру Width обеих кнопок значение `60`, а затем расположите в точках с координатой *Y*, равной `-50`, и координатами *X*, равными `0` и `70`.

На одной кнопке напишите Equip, на другой — Use.

Это визуальные элементы для всплывающего окна со списком инвентаря, а сейчас мы напишем для него код. Введите содержимое следующего листинга в сценарий `InventoryPopup`.

Листинг 12.14. Полный сценарий InventoryPopup

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

public class InventoryPopup : MonoBehaviour {
  [SerializeField] private Image[] itemIcons;
  [SerializeField] private Text[] itemLabels;

  [SerializeField] private Text curItemLabel;
  [SerializeField] private Button equipButton;
  [SerializeField] private Button useButton;

  private string _curItem;

  public void Refresh() {
```

Массивы для ссылки на четыре изображения и текстовые метки.

```
List<string> itemList = Managers.Inventory.GetItemList();

int len = itemIcons.Length;
for (int i = 0; i < len; i++) {
  if (i < itemList.Count) {
    itemIcons[i].gameObject.SetActive(true);
    itemLabels[i].gameObject.SetActive(true);

    string item = itemList[i];

    Sprite sprite = Resources.Load<Sprite>("Icons/"+item);
    itemIcons[i].sprite = sprite;
    itemIcons[i].SetNativeSize();

    int count = Managers.Inventory.GetItemCount(item);
    string message = "x" + count;
    if (item == Managers.Inventory.equippedItem) {
      message = "Equipped\n" + message;
    }
    itemLabels[i].text = message;

    EventTrigger.Entry entry = new EventTrigger.Entry();
    entry.eventID = EventTriggerType.PointerClick;
    entry.callback.AddListener((BaseEventData data) => {
       OnItem(item);
    });

    EventTrigger trigger = itemIcons[i].GetComponent<EventTrigger>();
    trigger.delegates.Clear();
    trigger.delegates.Add(entry);
  }
  else {
    itemIcons[i].gameObject.SetActive(false);
    itemLabels[i].gameObject.SetActive(false);
  }
}

if (!itemList.Contains(_curItem)) {
  _curItem = null;
}
if (_curItem == null) {
  curItemLabel.gameObject.SetActive(false);
  equipButton.gameObject.SetActive(false);
  useButton.gameObject.SetActive(false);
}
else {
  curItemLabel.gameObject.SetActive(true);
  equipButton.gameObject.SetActive(true);
  if (_curItem == "health") {
    useButton.gameObject.SetActive(true);
  } else {
    useButton.gameObject.SetActive(false);
  }
```

Проверка списка инвентаря в процессе циклического просмотра всех изображений элементов UI.

Загружаем спрайт из папки Resources.

Изменение размеров изображения под исходный размер спрайта.

На метке может появиться не только количество элементов, но и слово "Equipped".

Превращаем значки в интерактивные объекты.

Лямбда-функция, позволяющая по-разному активировать каждый элемент.

Сброс подписчика, чтобы начать с чистого листа.

Добавление функции-подписчика к классу EventTrigger.

Скрываем изображение/текст при отсутствии элементов для отображения.

Скрываем кнопки при отсутствии выделенных элементов.

Отображаем выделенный в данный момент элемент.

Используем кнопку только для элемента health.

```
      curItemLabel.text = _curItem+":";
    }
  }

  public void OnItem(string item) {
    _curItem = item;
    Refresh();
  }

  public void OnEquip() {
    Managers.Inventory.EquipItem(_curItem);
    Refresh();
  }

  public void OnUse() {
    Managers.Inventory.ConsumeItem(_curItem);
    if (_curItem == "health") {
      Managers.Player.ChangeHealth(25);
    }
    Refresh();
  }
}
```

Функция, вызываемая подписчиком события щелчка мыши.

Актуализируем отображение инвентаря после внесения изменений.

С огромным сценарием закончено! Теперь нужно связать все элементы интерфейса. Компонент сценария обладает ссылками на различные объекты, включая два массива; раскройте оба массива и сделайте их длину равной 4, как показано на рис. 12.6. Четыре изображения перетащите на ячейки массива значков, а четыре текстовые метки — на ячейки массива меток.

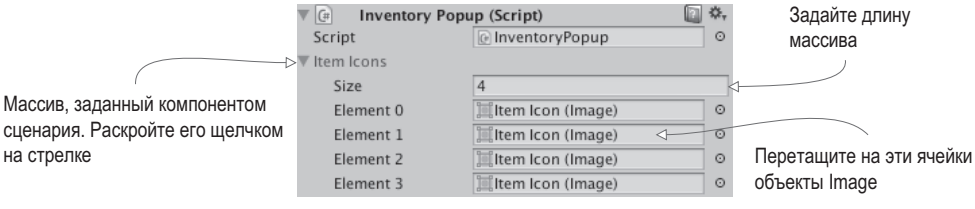


Рис. 12.6. Отображение массива на панели Inspector

ПРИМЕЧАНИЕ Если вы не уверены, какой объект нужно связывать с определенной ячейкой (они все выглядят одинаково), щелкните на ячейке и посмотрите, какой объект будет выделен на вкладке Hierarchy.

Аналогичным образом добавьте в ячейки компонента ссылки на текстовую метку и на кнопки в нижней части всплывающего окна. После связывания этих объектов добавим к обеим кнопкам подписчика на событие `OnClick`. Свяжите эти события со всплывающим окном и выберите подходящий вариант — OnEquip() или OnUse().

Наконец, добавим ко всем четырем изображениям элементов компонент `EventTrigger`. Сценарий `InventoryPopup` редактирует этот компонент для каждого значка, поэтому

лучше, чтобы во всех случаях он присутствовал! Компонент `EventTrigger` находится в разделе Event (напомню, что для добавления нужно щелкнуть на кнопке Add Component). Возможно, вам будет удобнее добавить его методом копирования. Для этого нужно щелкнуть на маленькой кнопке с изображением шестерни справа от имени компонента, выбрать в появившемся меню команду Copy Component, выделить объект, в который его нужно скопировать, и воспользоваться командой Paste As New. Выполните эту операцию, но не назначайте подписчиков события, так как это уже сделано в коде сценария `InventoryPopup`.

На этом работа над пользовательским интерфейсом для отображения инвентаря завершается! Запустите игру, чтобы посмотреть, что происходит со всплывающим окном, когда вы собираете инвентарь и щелкаете на кнопках. Сборка фрагментов предыдущих проектов закончена; теперь я объясню, как создать более масштабную игру с нуля.

12.2. Разработка общей игровой структуры

Теперь, когда у нас есть функционирующая демонстрационная версия ролевой игры, нужно создать для нее общую игровую структуру. Под этими словами я подразумеваю различные уровни игры. Проект из главы 9 состоял из одного уровня, в то время как согласно нашему плану их должно быть три.

Для этого потребуется еще сильнее уменьшить связь сцены с отвечающими за ее обработку диспетчерами, в результате придется рассылать сообщения об их состояниях (так же, как диспетчер `PlayerManager` рассылает информацию об обновлениях состояния здоровья персонажа). Создайте сценарий `StartupEvent` (листинг 12.15); происходящее при загрузке выделено в отдельный сценарий, так как эти события идут в связке с постоянно использующейся системой диспетчеров, в то время как класс `GameEvent` связан с конкретной игрой.

Листинг 12.15. Сценарий StartupEvent

```
public static class StartupEvent {
  public const string MANAGERS_STARTED = "MANAGERS_STARTED";
  public const string MANAGERS_PROGRESS = "MANAGERS_PROGRESS";
}
```

Теперь можно приступать к редактированию диспетчеров, добавляя к ним средства рассылки сообщений о новых событиях!

12.2.1. Управление ходом миссии и набором уровней

Пока что проект состоит из одной сцены, в которой в числе прочего находится главный диспетчер. Проблема в том, что при таком подходе у каждой сцены будет собственный набор игровых диспетчеров, в то время как нам требуется набор, доступный всем сценам. Для этого создадим отдельную сцену `Startup`, которая будет инициализировать диспетчеры, а затем давать к ним доступ остальным игровым сценам.

Также нужен новый диспетчер, обрабатывающий процесс прохождения игры. Создайте сценарий `MissionManager` и скопируйте в него содержимое следующего листинга.

Листинг 12.16. Сценарий MissionManager

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

public class MissionManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  public int curLevel {get; private set;}
  public int maxLevel {get; private set;}

  private NetworkService _network;

  public void Startup(NetworkService service) {
    Debug.Log("Mission manager starting...");

    _network = service;

    curLevel = 0;
    maxLevel = 1;

    status = ManagerStatus.Started;
  }

  public void GoToNext() {
    if (curLevel < maxLevel) {  <-- Проверяем, достигнут ли последний уровень.
      curLevel++;
      string name = "Level" + curLevel;
      Debug.Log("Loading " + name);
      SceneManager.LoadScene(name);  <-- Команда загрузки сцены
    } else {
      Debug.Log("Last level");
    }
  }
}
```

По большей части ничего необычного в этом листинге не происходит. Обратить внимание имеет смысл на расположенный ближе к концу метод `LoadLevel()`; я уже упоминал о нем в главе 5, но до текущего момента он нас не особо интересовал. Этот метод служит для загрузки файла сцены; в главе 5 он использовался для перезагрузки одной и той же сцены, теперь же мы с его помощью начнем загружать произвольные сцены, передавая имена соответствующих файлов.

Свяжите этот сценарий с объектом `Game Managers`. Заодно добавьте к сценарию `Managers` новый компонент (см. следующий листинг).

Листинг 12.17. Добавление нового компонента в сценарий Managers

```
...
[RequireComponent(typeof(MissionManager))]

public class Managers : MonoBehaviour {
  public static PlayerManager Player {get; private set;}
```

```
  public static InventoryManager Inventory {get; private set;}
  public static MissionManager Mission {get; private set;}
  ...
  void Awake() {
    DontDestroyOnLoad(gameObject);

    Player = GetComponent<PlayerManager>();
    Inventory = GetComponent<InventoryManager>();
    Mission = GetComponent<MissionManager>();

    _startSequence = new List<IGameManager>();
    _startSequence.Add(Player);
    _startSequence.Add(Inventory);
    _startSequence.Add(Mission);

    StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    ...
      if (numReady > lastReady) {
        Debug.Log("Progress: " + numReady + "/" + numModules);
        Messenger<int, int>.Broadcast(
          StartupEvent.MANAGERS_PROGRESS, numReady, numModules);
      }

      yield return null;
    }

    Debug.Log("All managers started up");
    Messenger.Broadcast(StartupEvent.MANAGERS_STARTED);
  }
...
```

Команда Unity для сохранения объекта между сценами.

Событие загрузки рассылается вместе с относящимися к нему данными.

Событие загрузки рассылается без параметров.

Большая часть этого кода уже знакома (процесс добавления диспетчера `MissionManager` ничем не отличается от процесса добавления всех прочих диспетчеров), но появились и два новых фрагмента. Во-первых, событие, рассылаемое с двумя целочисленными значениями. Раньше вы видели обобщенные события без значений и сообщения с одним числом, но синтаксис позволяет рассылать произвольное количество численных значений.

Вторым новым фрагментом кода является метод `DontDestroyOnLoad()`. В Unity он обеспечивает сохранение объекта между сценами. Обычно при загрузке новой сцены все объекты удаляются, но метод `DontDestroyOnLoad()` позволяет переносить их в следующую сцену.

Разделение сцен для загрузки и игрового уровня

Объект Game Managers будет присутствовать во всех сценах, поэтому следует разделить диспетчеры и игровые уровни. На вкладке Project создайте копию файла сцены (Edit > Duplicate) и присвойте двум файлам соответствующие имена: Startup и Level1. Откройте файл Level1 и удалите объект Game Managers (он будет предоставляться сценой Startup). Откройте сцену Startup и удалите все, кроме объектов Game Managers, Controller, HUD Canvas

и EventSystem. Отрегулируйте камеру, удалив компонент OrbitCamera и выбрав в меню Clear Flags вариант SolidColor вместо варианта Skybox. Удалите сценарные компоненты у объекта Controller, а также объекты UI (метку health и объект InventoryPopup), которые являются потомками объекта Canvas.

Наш пользовательский интерфейс теперь пуст, поэтому создайте новый ползунок, как показано на рис. 12.7, и сбросьте флажок Interactable. У объекта Controller теперь тоже отсутствуют компоненты сценария, поэтому создайте сценарий `StartupController` и присоедините его к контроллеру.

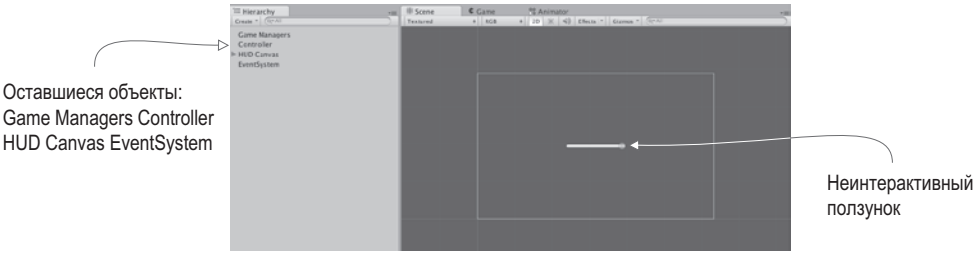


Рис. 12.7. Сцена Startup после удаления всего ненужного

Листинг 12.18. Новый сценарий StartupController

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class StartupController : MonoBehaviour {
  [SerializeField] private Slider progressBar;

  void Awake() {
    Messenger<int, int>.AddListener(StartupEvent.MANAGERS_PROGRESS,
OnManagersProgress);
    Messenger.AddListener(StartupEvent.MANAGERS_STARTED, OnManagersStarted);
  }
  void OnDestroy() {
    Messenger<int, int>.RemoveListener(StartupEvent.MANAGERS_PROGRESS,
OnManagersProgress);
    Messenger.RemoveListener(StartupEvent.MANAGERS_STARTED, OnManagersStarted);
  }

  private void OnManagersProgress(int numReady, int numModules) {
    float progress = (float)numReady / numModules;
    progressBar.value = progress;  <-- Обновляем ползунок данными о процессе загрузки.
  }

  private void OnManagersStarted() {
    Managers.Mission.GoToNext();  <-- После загрузки диспетчеров загружаем следующую сцену.
  }
}
```

Теперь свяжите объект `Slider` с ячейкой на панели Inspector. Остался всего один подготовительный шаг. Нужно добавить две сцены в список Build Settings. Процедура создания приложения рассматривается в следующей главе, а пока просто выберите в меню File

команду Build Settings для просмотра и корректировки списка сцен. Щелкните на кнопке Add Current, чтобы добавить в список текущую сцену (загрузите обе сцены и проделайте эту операцию для каждой из них).

ПРИМЕЧАНИЕ Добавление в список Build Settings обеспечивает загрузку сцен. Без этого Unity не знает, какие из сцен доступны. В главе 5 эта операция не требовалась, так как мы не переходили от одной сцены к другой, а только перезагружали текущую сцену.

Теперь можно загрузить игру, нажав кнопку Play в сцене `Startup`. Объект `Game Managers` будет присутствовать в обеих сценах.

ВНИМАНИЕ Диспетчеры загружаются в сцене `Startup`, поэтому игра должна запускаться именно отсюда. Можно запомнить, что перед нажатием кнопки Play всегда требуется переходить к этой сцене, а можно воспользоваться сценарием http://wiki.unity3d.com/index.php/SceneAutoLoader, который после запуска воспроизведения будет автоматически перебрасывать вас в нужное состояние.

СОВЕТ По умолчанию при загрузке уровня система освещения восстанавливает карты освещения. Но это работает только на стадии редактирования уровня; при загрузке уровней во время игры карты освещения не генерируются. Как это делалось в главе 10, можно снять флажок Auto Generate в нижней части окна с параметрами освещенности (Window > Lighting > Settings) и щелкнуть на кнопке, чтобы получить карты освещения вручную (напоминаю, что вы не должны трогать появляющуюся при этом папку lighting).

Это структурное изменение отвечает за совместное использование диспетчеров различными сценами, но на уровнях пока отсутствуют индикаторы успешного прохождения уровней и гибели персонажа.

12.2.2. Завершение уровня

Для завершения уровня в сцену нужно поместить объект, который в ответ на касание персонажа будет информировать диспетчер `MissionManager` об успешном достижении цели. Для этого потребуется пользовательский интерфейс, реагирующий на сообщение о завершении уровня, поэтому добавьте в сценарий `GameEvent` содержимое следующего листинга.

Листинг 12.19. Код завершения уровня, добавляемый в сценарий GameEvent.cs

```
public static class GameEvent {
  public const string HEALTH_UPDATED = "HEALTH_UPDATED";
  public const string LEVEL_COMPLETE = "LEVEL_COMPLETE";
}
```

Теперь в сценарий `MissionManager` нужно добавить новый метод для слежения за целями миссий и рассылки новых сообщений о событиях.

Листинг 12.20. Метод слежения за целями в сценарии MissionManager

```
...
public void ReachObjective() {
  // здесь может быть код обработки нескольких целей
  Messenger.Broadcast(GameEvent.LEVEL_COMPLETE);
}
...
```

Отредактируем сценарий `UIController`, как показано в следующем листинге, чтобы он начал реагировать на завершение уровня.

Листинг 12.21. Новый подписчик на событие в сценарии UIController

```
...
[SerializeField] private Text levelEnding;
...
void Awake() {
  Messenger.AddListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  Messenger.AddListener(GameEvent.LEVEL_COMPLETE, OnLevelComplete);
}
void OnDestroy() {
  Messenger.RemoveListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  Messenger.RemoveListener(GameEvent.LEVEL_COMPLETE, OnLevelComplete);
}
...
void Start() {
  OnHealthUpdated();

  levelEnding.gameObject.SetActive(false);
  popup.gameObject.SetActive(false);
}
...
private void OnLevelComplete() {
  StartCoroutine(CompleteLevel());
}
private IEnumerator CompleteLevel() {
  levelEnding.gameObject.SetActive(true);
  levelEnding.text = "Level Complete!";

  yield return new WaitForSeconds(2);

  Managers.Mission.GoToNext();
}
...
```

В течение двух секунд отображаем сообщение, а потом переходим на следующий уровень.

Обратите внимание, что в листинге есть ссылка на текстовую метку. Откройте сцену `Level1` и создайте новый текстовый объект UI. Эта метка будет содержать сообщение о завершении уровня, появляющееся в центре экрана. Присвойте параметру Width этого текста значение 240, параметру Height — значение 60, для выравнивания по горизонтали и вертикали выберите вариант Center, а параметр Font Size сделайте равным 22. В текстовое поле введите слова Level Complete! и свяжите текстовый объект со ссылкой levelEnding в сценарии `UIController`.

Осталось создать объект, прикосновение к которому позволит завершить уровень. Пример такого объекта показан на рис. 12.8. Он напоминает доступные для сбора элементы инвентаря. Ему нужно назначить материал и связать его со сценарием, а затем мы превратим этот объект в шаблон экземпляра.

Создайте куб и введите в поля Position значения `18`, `1`, `0`. Установите флажок Is Trigger в разделе Box Collider, отключите возможность отбрасывать и получать тени в разделе Mesh Renderer и поместите объект в слой Ignore Raycast. Создайте материал с именем

objective; сделайте его ярко-зеленым и выберите вариант шейдера Unlit > Color для получения равномерного яркого вида.

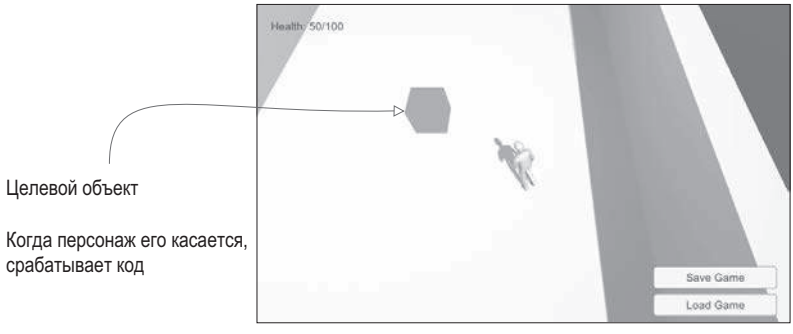


Рис. 12.8. Объект, к которому нужно прикоснуться для завершения уровня

Теперь создайте сценарий `ObjectiveTrigger` и свяжите его с целевым объектом.

Листинг 12.22. Код сценария ObjectiveTrigger для применения к целевым объектам

```
using UnityEngine;
using System.Collections;

public class ObjectiveTrigger : MonoBehaviour {
  void OnTriggerEnter(Collider other) {
    Managers.Mission.ReachObjective();
  }
}
```

Вызываем новый целевой метод в сценарии MissionManager.

Перетащите полученный объект со вкладки Hierarchy на вкладку Project, чтобы превратить его в шаблон экземпляра; на следующих уровнях его можно будет вставлять в сцены. Запустите игру и попробуйте добраться до целевого объекта. Появится сообщение о завершении уровня.

Теперь нужно создать сообщение, которое будет появляться в момент смерти персонажа.

12.2.3. Смерть персонажа

Смерть персонажа наступает при полной утрате здоровья (из-за успешных атак противника). Первым делом добавьте еще одну строку в сценарий `GameEvent`:

```
public const string LEVEL_FAILED = "LEVEL_FAILED";
```

Далее отредактируйте сценарий `PlayerManager`, чтобы падение уровня здоровья до нуля сопровождалось рассылкой сообщения.

Листинг 12.23. Рассылка сообщения о смерти персонажа из сценария PlayerManager

```
...
public void Startup(NetworkService service) {
  Debug.Log("Player manager starting...");
```

```
  _network = service;

  UpdateData(50, 100);                    ← Вызываем метод обновления, вместо того
                                            чтобы задавать переменные напрямую.
  status = ManagerStatus.Started;
}

public void UpdateData(int health, int maxHealth) {
  this.health = health;
  this.maxHealth = maxHealth;
}

public void ChangeHealth(int value) {
  health += value;
  if (health > maxHealth) {
    health = maxHealth;
  } else if (health < 0) {
    health = 0;
  }

  if (health == 0) {
    Messenger.Broadcast(GameEvent.LEVEL_FAILED);
  }
  Messenger.Broadcast(GameEvent.HEALTH_UPDATED);
}

public void Respawn() {  ← Возвращаем игрока в исходное состояние.
  UpdateData(50, 100);
}
...
```

Для перезагрузки уровня добавим небольшой метод в сценарий `MissionManager`.

Листинг 12.24. Сценарий MissionManager, позволяющий начать уровень сначала

```
...
public void RestartCurrent() {
  string name = "Level" + curLevel;
  Debug.Log("Loading " + name);
  SceneManager.LoadScene(name);
}
...
```

Теперь добавим в сценарий `UIController` еще одного подписчика на событие.

Листинг 12.25. Реакция на гибель персонажа в сценарии UIController

```
...
Messenger.AddListener(GameEvent.LEVEL_FAILED, OnLevelFailed);
...
Messenger.RemoveListener(GameEvent.LEVEL_FAILED, OnLevelFailed);
...
private void OnLevelFailed() {
  StartCoroutine(FailLevel());
}
private IEnumerator FailLevel() {
```

```
  levelEnding.gameObject.SetActive(true);
  levelEnding.text = "Level Failed";

  yield return new WaitForSeconds(2);

  Managers.Player.Respawn();
  Managers.Mission.RestartCurrent();
}
...
```

Используем ту же самую текстовую метку, но с другим сообщением.

После двухсекундной паузы начинаем текущий уровень сначала.

Запустите игру и позвольте противникам сделать несколько выстрелов; появится сообщение о проигрыше. Прекрасная работа — теперь в случае неудачного прохождения можно завершать уровни и начинать их заново! Пришло время добавить в игру возможность следить за успехами игрока.

12.3. Продвижение по уровням

Пока что каждый уровень управляется независимо от других и без связи с игрой в целом. Требуется пара деталей, которые сделают процесс прохождения более завершенным: сохранение достижений игрока и определение факта окончания игры (а не только уровня).

12.3.1. Сохранение и загрузка уровня

Процессы сохранения и загрузки важны для большинства игр. В Unity и Mono для этой цели добавлена функциональность ввода-вывода. Но перед тем как мы начнем с ней работать, в сценарии `MissionManager` и `InventoryManager` следует добавить метод `UpdateData()`. Принцип его работы такой же, как у сценария `PlayerManager`. Он дает внешнему по отношению к диспетчеру коду возможность обновлять данные внутри диспетчера. Отредактированные версии диспетчеров даны в листингах 12.26 и 12.27.

Листинг 12.26. Метод UpdateData() в сценарии MissionManager

```
...
public void Startup(NetworkService service) {
  Debug.Log("Mission manager starting...");

  _network = service;

  UpdateData(0, 1);

  status = ManagerStatus.Started;
}

public void UpdateData(int curLevel, int maxLevel) {
  this.curLevel = curLevel;
  this.maxLevel = maxLevel;
}
...
```

Отредактируем эту строку, использовав новый метод.

Листинг 12.27. Метод UpdateData() в сценарии InventoryManager

```
...
public void Startup(NetworkService service) {
  Debug.Log("Inventory manager starting...");

  _network = service;

  UpdateData(new Dictionary<string, int>());  <-- Инициализируем пустой список.

  status = ManagerStatus.Started;
}

public void UpdateData(Dictionary<string, int> items) {
  _items = items;
}

public Dictionary<string, int> GetData() {      <-- Для сохранения данных необходима
  return _items;                                    функция чтения.
}
...
```

Теперь, когда метод `UpdateData()` появился в разных диспетчерах, новый модуль кода позволит сохранять данные. Для сохранения потребуется процедура, называемая *сериализацией* данных.

ОПРЕДЕЛЕНИЕ *Сериализовать* (serialize) означает перекодировать пакет данных в форму, допускающую сохранение.

Нашу игру мы сохраним в виде двоичных данных, но имейте в виду, что C# допускает также сохранение в виде текстовых файлов. Например, строки в формате JSON, с которыми вы работали в главе 10, представляли собой данные, сериализованные в виде текста. В предыдущих главах мы пользовались методом `PlayerPrefs`, но сейчас требуется сохранить локальный файл (метод `PlayerPrefs` предназначен для сохранения набора нескольких значений, таких как настройки). Создайте сценарий `DataManager` (см. следующий листинг).

ВНИМАНИЕ В интернет-играх отсутствует доступ к файловой системе, то есть они не могут сохранять свое состояние в виде локальных файлов. Это сделано в целях безопасности. Сохранение данных в этом случае осуществляется путем их отправки на сервер.

Листинг 12.28. Новый сценарий для диспетчера данных

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

public class DataManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  private string _filename;
```

```
  private NetworkService _network;

  public void Startup(NetworkService service) {
    Debug.Log("Data manager starting...");

    _network = service;

    _filename = Path.Combine(Application.persistentDataPath, "game.dat");
    status = ManagerStatus.Started;
  }

  public void SaveGameState() {
    Dictionary<string, object> gamestate = new Dictionary<string, object>();
    gamestate.Add("inventory", Managers.Inventory.GetData());
    gamestate.Add("health", Managers.Player.health);
    gamestate.Add("maxHealth", Managers.Player.maxHealth);
    gamestate.Add("curLevel", Managers.Mission.curLevel);
    gamestate.Add("maxLevel", Managers.Mission.maxLevel);

    FileStream stream = File.Create(_filename);
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(stream, gamestate);
    stream.Close();
  }

  public void LoadGameState() {
    if (!File.Exists(_filename)) {
      Debug.Log("No saved game");
      return;
    }

    Dictionary<string, object> gamestate;

    BinaryFormatter formatter = new BinaryFormatter();
    FileStream stream = File.Open(_filename, FileMode.Open);
    gamestate = formatter.Deserialize(stream) as Dictionary<string, object>;
    stream.Close();

    Managers.Inventory.UpdateData((Dictionary<string, int>)
   gamestate["inventory"]);
    Managers.Player.UpdateData((int)gamestate["health"], (int)
   gamestate["maxHealth"]);
    Managers.Mission.UpdateData((int)gamestate["curLevel"], (int)
   gamestate["maxLevel"]);
    Managers.Mission.RestartCurrent();
  }
}
```

Генерируем полный путь к файлу game.dat.

Словарь, который будет подвергнут сериализации.

Создаем файл по указанному адресу.

Сериализуем объект Dictionary как содержимое созданного файла.

Продолжаем загрузку только при наличии файла.

Словарь для размещения загруженных данных.

Обновляем диспетчеры, снабжая их десериализованными данными.

В методе `Startup()` с помощью свойства `Application.persistentDataPath`, которое Unity предоставляет как место для сохранения файлов, генерируется путь к файлу. Конкретный путь к файлу на каждой платформе будет своим, но Unity рассматривает его отвлеченно с помощью статической переменной (кстати, в путь включаются данные

из полей Company Name и Product Name, входящих в состав настроек Player Settings, поэтому, если нужно, отредактируйте эти переменные). Метод `File.Create()` создает двоичный файл; если вам нужен текстовый файл, воспользуйтесь методом `File.CreateText()`.

ВНИМАНИЕ При конструировании пути для разных платформ используются разные разделители. В C# это обстоятельство учитывается с помощью поля Path.DirectorySeparatorChar.

Откройте сцену `Startup` и найдите объект `Game Managers`. Добавьте к этому объекту в качестве компонента сценарий `DataManager`, а затем вставьте новый диспетчер в сценарий `Managers`, как показано в листинге 12.29.

Листинг 12.29. Добавления диспетчера DataManager в сценарий Managers.cs

```
...
[RequireComponent(typeof(DataManager))]
...
public static DataManager Data {get; private set;}
...
void Awake() {
  DontDestroyOnLoad(gameObject);

  Data = GetComponent<DataManager>();
  Player = GetComponent<PlayerManager>();
  Inventory = GetComponent<InventoryManager>();
  Mission = GetComponent<MissionManager>();

  _startSequence = new List<IGameManager>();  ◄—— Диспетчеры запускаются по порядку.
  _startSequence.Add(Player);
  _startSequence.Add(Inventory);
  _startSequence.Add(Mission);
  _startSequence.Add(Data);

  StartCoroutine(StartupManagers());
}
...
```

ВНИМАНИЕ Так как диспетчер `DataManager` пользуется остальными диспетчерами (с целью их обновления), нужно сделать так, чтобы в загрузочной последовательности остальные диспетчеры фигурировали раньше.

Рис. 12.9. Кнопки сохранения и загрузки в нижней правой части экрана

Наконец, чтобы получить возможность доступа к функциям диспетчера DataManager, добавим кнопки, как показано на рис. 12.9. Создайте две кнопки, сделав их потомками объекта HUD Canvas (а не всплывающего окна Inventory). Назовите их (указав имена в настройках присоединенных текстовых объектов) Save Game и Load Game, в окне Anchor Presets выберите вариант bottom-right и поместите их в точки с координатами –100, 65 и –100, 30 соответственно.

Эти кнопки нужно связать с функциями в сценарии UIController, поэтому воспользуйтесь методами из следующего листинга.

Листинг 12.30. Методы загрузки и сохранения в сценарии UIController

```
...
public void SaveGame() {
  Managers.Data.SaveGameState();
}

public void LoadGame() {
  Managers.Data.LoadGameState();
}
...
```

Свяжите эти функции с подписчиками на событие OnClick кнопок (добавьте слушание в настройки события OnClick, перетащите объект UIController и выберите функции в меню). Теперь запустите игру, соберите несколько элементов инвентаря, воспользуйтесь пакетом здоровья, чтобы увеличить эту характеристику, и сохраните игру. Заново запустите игру и проверьте свой инвентарь, чтобы удостовериться, что он отсутствует. Щелкните на кнопке Load; у вас появятся уровень здоровья и инвентарь, которые были на момент сохранения игры!

12.3.2. Победа при полном прохождении всех уровней

Сохранять достижения игрока имеет смысл в играх, состоящих из множества уровней, у нас же пока есть всего один, которым вы пользовались для тестирования. Для правильной обработки набора уровней требуется умение распознавать завершение не только одного уровня, но и игры в целом. Первым делом добавьте в сценарий GameEvent еще одну строку:

```
public const string GAME_COMPLETE = "GAME_COMPLETE";
```

Теперь заставим сценарий MissionManager рассылать сообщение о прохождении последнего уровня.

Листинг 12.31. Рассылка сообщения о завершении игры сценарием MissionManager

```
...
public void GoToNext() {
...
  } else {
    Debug.Log("Last level");
    Messenger.Broadcast(GameEvent.GAME_COMPLETE);
  }
}
```

На это сообщение должен реагировать сценарий `UIController`.

Листинг 12.32. Добавление подписчика события в сценарий UIController

```
...
Messenger.AddListener(GameEvent.GAME_COMPLETE, OnGameComplete);
...
Messenger.RemoveListener(GameEvent.GAME_COMPLETE, OnGameComplete);
...
private void OnGameComplete() {
  levelEnding.gameObject.SetActive(true);
  levelEnding.text = "You Finished the Game!";
}
...
```

Пройдите уровень до конца (для этого персонаж должен коснуться целевого объекта) и посмотрите, что получится. Первым делом должно появиться сообщение Level Complete, которое через пару секунд сменяется сообщением о завершении игры.

Добавление уровней

Теперь можно добавить произвольное количество дополнительных уровней, а диспетчер `MissionManager` будет следить за последним уровнем. Это последнее, что осталось сделать в этой главе, ведь нужно продемонстрировать, как выглядит прохождение многоуровневой игры.

Дважды продублируйте сцену `Level1` (цифры в именах файлов при этом автоматически увеличатся до Level2 и Level3). Новые уровни следует добавить в список Build Settings, чтобы они могли загружаться в процессе игры. Отредактируйте каждую сцену, чтобы уровни отличались друг от друга; вы можете произвольно менять элементы сцен, сохраняя, однако, следующие объекты: объект `player` с тегом Player, объект `floor`, находящийся в слое Ground, целевой объект для выхода с уровня, объекты `Controller`, HUD `Canvas` и `EventSystem`.

Также нужно внести правки в сценарий `MissionManager`, чтобы он начал загружать новые уровни. Присвойте параметру maxLevel новое значение 3, заменив вызов метода `UpdateData(0, 1)` вызовом `UpdateData(0, 3)`. Запустите игру. Вы начнете с уровня Level1. Коснувшись целевого объекта, вы окажетесь на следующем уровне! Кстати, рекомендую выполнить сохранение на более высоком уровне, чтобы еще раз убедиться, что игра сохраняет данные о ваших достижениях.

УПРАЖНЕНИЕ: ВСТАВКА ЗВУКА В ПОЛНУЮ ВЕРСИЮ ИГРЫ

Глава 11 целиком посвящена созданию звуковых эффектов в Unity. Процесс интеграции звука в проект отдельно не рассматривался, но к настоящему моменту вы уже должны понимать, как он реализован. Я призываю вас попробовать свои силы и самостоятельно добавить в наш проект функциональность из предыдущей главы. Подсказка: поменяйте клавишу, вызывающую окно с настройками звука, чтобы она не мешала пользоваться всплывающим окном с перечнем инвентаря.

Вы научились создавать игры с множеством уровней. Осталась еще одна задача, которую мы и рассмотрим в последней главе нашей книги: как предоставить пользователям доступ к игре.

Заключение

- Программа Unity позволяет легко менять предназначение ресурсов и кода из проектов в различных игровых жанрах.
- Метод бросания луча позволяет определять точку, на которой щелкнул игрок.
- В Unity существуют простые методы как для загрузки уровней, так и для сохранения определенных объектов при переходе с уровня на уровень.
- Переход с уровня на уровень возникает в ответ на различные события в игре.
- Для сохранения данных в поле Application.persistentDataPath можно пользоваться методами ввода-вывода из языка C#.

13 Развертывание игр на устройствах игроков

- ✓ Создание пакетов прикладных программ для различных платформ.
- ✓ Задание параметров сборки, таких как значок или имя приложения.
- ✓ Взаимодействие веб-игр с веб-страницей.
- ✓ Разработка подключаемых модулей для приложений на мобильных платформах.

До этого момента мы рассматривали исключительно процесс программирования различных игр в Unity, но без внимания остался важный заключительный шаг: доставка этих игр пользователям. Игра, существующая лишь в редакторе Unity, представляет интерес разве что для своего разработчика. На последнем этапе Unity выступает во всем своем блеске, позволяя строить приложения для огромного количества игровых платформ. Именно этой теме и посвящена наша последняя глава.

Под словосочетанием «построение» для платформы я подразумеваю генерацию запускаемого на этой платформе прикладного пакета. На каждой платформе (Windows, iOS и т. п.) своя форма пакета, но как только вы сгенерировали исполняемый файл, появляется возможность распространять игру и играть в нее без привязки к Unity. Один проект Unity можно развернуть на разных платформах — его не нужно каждый раз генерировать заново.

Принцип «построй один раз и развертывай где угодно» применим к подавляющему большинству игровых функций, но, к сожалению, не ко всем. По моим оценкам, 95% написанного в Unity кода (в частности, практически все, что мы делали в этой книге) не имеет привязки к платформе и прекрасно работает везде. Но кое-какие вещи зависят от выбранной платформы, и мы рассмотрим их достаточно подробно.

Приложение Unity позволяет создавать приложения для следующих платформ:

- ❍ Windows PC;
- ❍ Mac OS X;

- Linux;
- WebGL;
- iOS;
- Android;
- Windows Phone;
- Tizen;
- Oculus Rift;
- Steam VR/Vive;
- Daydream;
- HoloLens.

Кроме того, можно попросить доступ у владельцев следующих платформ:

- XBox One;
- PlayStation 4;
- PS Vita;
- Switch;
- 3DS.

Видите, какой длинный список! Положа руку на сердце — он потрясающе длинный, намного длиннее списка поддерживаемых платформ практически любого другого инструмента разработки игр. Подробно мы рассмотрим первые шесть вариантов из этого списка, и это будут платформы, представляющие интерес для подавляющего большинства пользователей Unity, но не забывайте, что список доступных вариантов намного больше.

Посмотреть все варианты платформ можно в окне Build Settings. В предыдущей главе оно применялось для добавления загружаемых сцен. Для доступа к нему выберите в меню File команду Build Settings. В главе 12 нас интересовал только список в верхней части, теперь же обратим внимание и на расположенные внизу кнопки (рис. 13.1). Много места занимает список платформ; активные в настоящий момент платформы отмечены значком Unity. Достаточно выделить платформы в этом списке и щелкнуть на кнопке Switch Platform.

ПРИМЕЧАНИЕ При установке Unity установщик запрашивает, какие модули экспорта вы хотите использовать, и затем вы можете создавать только выбранные модули. Если позже вы захотите установить модуль, который вы не выбрали изначально, в окне Build Settings есть кнопка для его добавления.

ВНИМАНИЕ В крупных проектах переход на другую платформу часто занимает много времени; приготовьтесь к ожиданию. Это связано с тем, что Unity выполняет повторное сжатие всех ресурсов (таких, как текстуры) оптимальным для каждой платформы способом.

В нижней части этого окна находятся кнопки Player Settings и Build. Щелчок на кнопке Player Settings открывает настройки приложения на панели Inspector, к числу которых относятся имя и значок приложения.

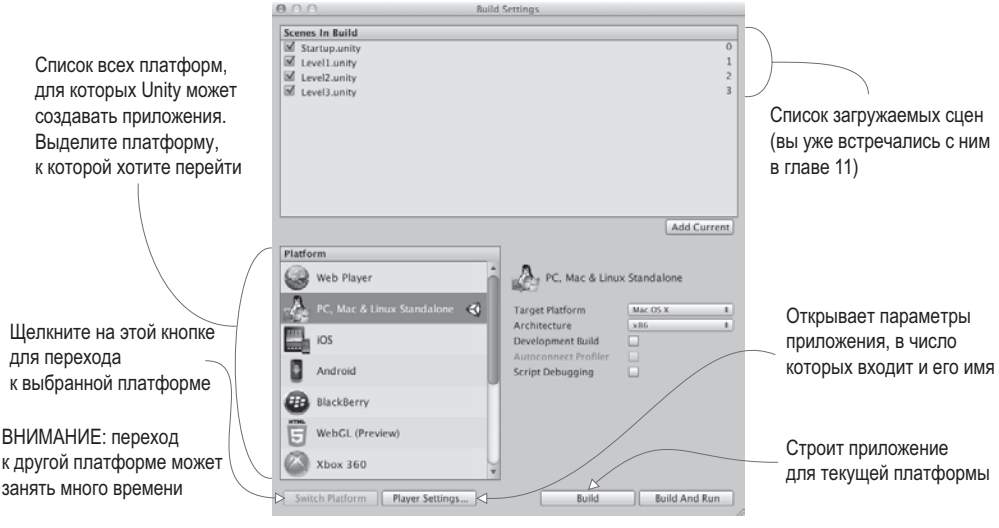


Рис. 13.1. Окно Build Settings

СОВЕТ Кнопка Build And Run отличается от кнопки Build тем, что автоматически запускает сгенерированное приложение. Я обычно предпочитаю делать это вручную, соответственно, кнопкой Build And Run практически не пользуюсь.

Щелчок на кнопке Build открывает окно выбора файла, в котором нужно указать адрес для генерации пакета приложения. Сразу после указания местоположения файла начинается процесс построения, после чего Unity создает исполняемый файл для активной в данный момент платформы; давайте рассмотрим этот процесс для наиболее популярных платформ: настольных компьютеров, интернета и мобильных устройств.

13.1. Приложения для настольных компьютеров: Windows, Mac и Linux

Тем, кто только приступает к созданию игр в Unity, проще всего начать с развертывания игры на настольных компьютерах под управлением Windows PC, Mac OS X или Linux. Unity работает на настольных компьютерах, соответственно, приложение будет генерироваться для той машины, которой вы уже пользуетесь.

ПРИМЕЧАНИЕ Для упражнений этой главы можно открыть любой проект по вашему вкусу. Я уверяю, что все будет работать; более того, рекомендую для каждого следующего раздела открывать новый проект, чтобы убедиться в способности Unity развертывать любой проект на любой платформе!

13.1.1. Создание приложения

Первым делом выберите в меню File команду Build Settings, чтобы открыть одноименное окно. По умолчанию выбран вариант PC, Mac & Linux Standalone, но если вас интересует другой вариант, укажите его в списке и щелкните на кнопке Switch Platform.

С правой стороны находится раскрывающийся список Target Platform. Он позволяет выбрать между платформами Windows PC, Mac OS X и Linux. В списке слева эти варианты рассматривались как один, но на самом деле это разные платформы, поэтому выберите нужную.

После этого остается щелкнуть на кнопке Build. Откроется окно диалога, в котором нужно будет выбрать местоположение будущего приложения. После этого начнется процесс построения, для крупных проектов занимающий изрядное время, но для наших крохотных демонстрационных проектов он должен завершиться достаточно быстро.

СОБСТВЕННЫЙ СЦЕНАРИЙ ПОСТ-СБОРКИ

Базовый процесс построения превосходно подходит для большинства ситуаций, но некоторые разработчики предпочитают при построении каждой игры совершать дополнительные действия (например, перемещать справочные файлы в папку, в которой находится приложение). Такие задачи можно легко автоматизировать, поместив их в сценарий, запускаемый после завершения процесса сборки.

Первым делом создайте новую папку на вкладке Project и присвойте ей имя Editor; именно здесь должны находиться все сценарии, влияющие на редактор Unity (это касается и процесса сборки). Создайте в этой папке сценарий с именем `TestPostBuild` и введите в него следующий код:

```
using UnityEngine;
using UnityEditor;
using UnityEditor.Callbacks;

public static class TestPostBuild {
 [PostProcessBuild]
 public static void OnPostprocessBuild(BuildTarget target, string
pathToBuiltProject) {
   Debug.Log("build location: " + pathToBuiltProject);
 }
}
```

Директива `[PostProcessBuild]` заставляет сценарий запускать расположенную непосредственно за ней функцию. Эта функция получит местоположение построенного приложения; после этого вы сможете использовать данный параметр в различных командах для работы с файловой системой из языка C#.

Приложение появится в указанной вами папке; запустите его двойным щелчком, как любую другую программу. Мои поздравления! Как видите, это было несложно. Процесс построения приложений совершенно тривиален, но допускает различные варианты настройки; посмотрим, как мы можем его доработать.

СОВЕТ Полноэкранные игры в операционной системе Windows закрываются комбинацией клавиш Alt+F4, а на компьютерах Mac — комбинацией клавиш Cmd+Q. В готовой игре должна быть кнопка, вызывающая метод `Application.Quit()`.

13.1.2. Настройки проигрывателя: имя и значок приложения

Вернемся в окно Build Settings, но на этот раз щелкнем не на кнопке Build, а на кнопке Player Settings. На панели Inspector появится список настроек, показанный на рис. 13.2; они контролируют различные аспекты готового приложения.

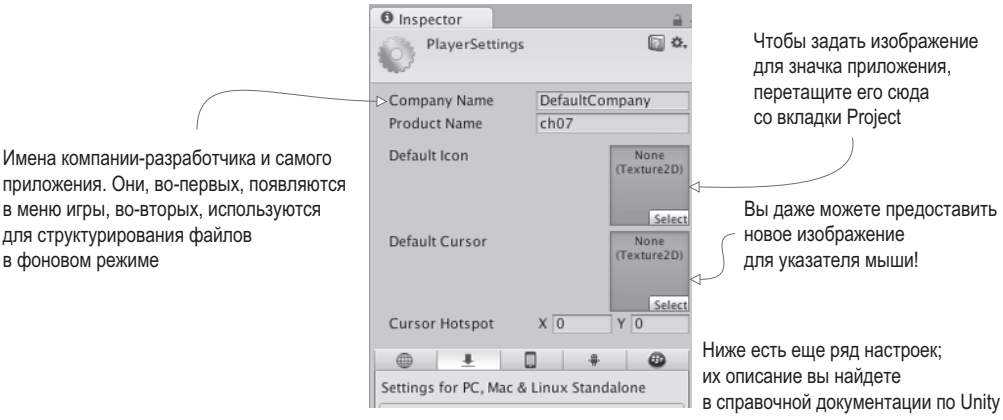
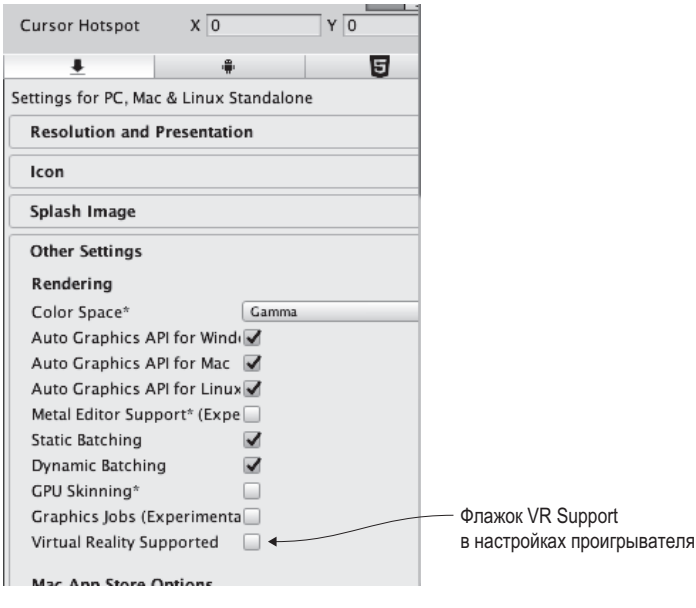


Рис. 13.2. Настройки проигрывателя на панели Inspector

ПОДДЕРЖКА ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ

У Unity есть встроенная поддержка VR-оборудования, такого как очки Oculus и шлем Vive, но с технической точки зрения при построении приложений виртуальная реальность не считается отдельной платформой. Ее поддержку можно просто включить для соответствующих платформ сборки, например для стационарных компьютеров или Android. Перейдите на нужную вкладку в настройках плеера (кнопки выбора платформы находятся сразу под верхним разделом) и раскройте раздел Other Settings. Именно здесь можно установить флажок Virtual Reality Supported, если вы разрабатываете приложение для виртуальной реальности!



Включение поддержки виртуальной реальности

Так как настроек в данном случае много, лучше почитать о них в руководстве пользователя. Вот адрес страницы: https://docs.unity3d.com/ru/current/Manual/class-PlayerSettings.html.

Смысл трех верхних параметров очевиден: Company Name (название компании), Product Name (название продукта) и Default Icon (значок, предлагаемый по умолчанию). Укажите значение первых двух. В поле Company Name — название студии-разработчика, а в поле Product Name — название игры. Затем задайте значок игры, перетащив нужное изображение со вкладки Project (при необходимости сначала импортируйте его в проект); когда приложение будет построено, это изображение появится как его значок.

Значок и имя приложения придают результатам вашего труда ощущение законченности. Другим вариантом настройки поведения приложений является зависящий от платформы код.

13.1.3. Компиляция в зависимости от платформы

По умолчанию написанный код запускается на всех платформах одним и тем же способом. Но Unity предоставляет ряд директив компилятора (известных как *определения платформ*), которые заставляют код работать исключительно на указанной платформе. Полный список определений платформ на странице https://docs.unity3d.com/ru/current/Manual/PlatformDependentCompilation.html.

Директивы существуют для всех поддерживаемых Unity платформ, соответственно, на каждой из них вы можете запускать свою версию кода. Обычно большую часть кода помещать внутрь директив не требуется, но отдельные фрагменты иногда имеет смысл запускать с привязкой к конкретной платформе. Некоторые варианты сборки существуют исключительно на одной платформе, поэтому требуются директивы компилятора, позволяющие обойти это ограничение. Следующий листинг демонстрирует пример такого кода.

Листинг 13.1. Сценарий PlatformTest как пример кода с привязкой к платформе

```
using UnityEngine;
using System.Collections;

public class PlatformTest : MonoBehaviour {
  void OnGUI() {
#if UNITY_EDITOR  <-- Этот раздел запускается только в редакторе.
    GUI.Label(new Rect(10, 10, 200, 20), "Running in Editor");
#elif UNITY_STANDALONE  <-- Только в приложениях для рабочего стола/автономных приложениях.
    GUI.Label(new Rect(10, 10, 200, 20), "Running on Desktop");
#else
    GUI.Label(new Rect(10, 10, 200, 20), "Running on other platform");
#endif
    }
}
```

Создайте сценарий `PlatformTest` и скопируйте в него код этого листинга. Свяжите его с произвольным объектом сцены (для тестирования подойдет любой объект), и в верхней левой части экрана появится маленькое сообщение. При воспроизведении игры в редакторе Unity это будет сообщение Running in the Editor (выполняется в редакторе), но

если вы сгенерируете приложение и запустите его, появится уже другой текст: Running on Desktop (выполняется на настольном компьютере). В каждом случае запускается свой вариант кода!

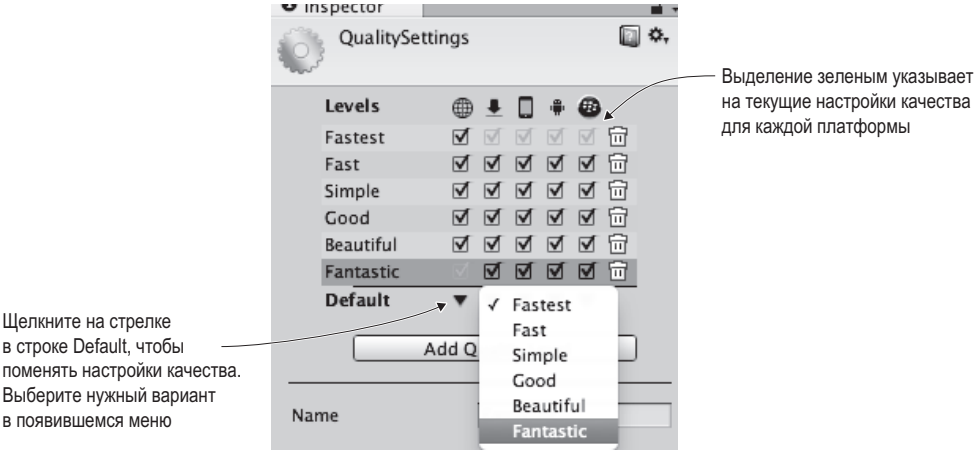
Для тестирования мы воспользовались определением платформы, воспринимающим все платформы на базе настольных компьютеров как одну, но, как можно прочитать в документации, существуют отдельные определения платформ для Windows, Mac и Linux. Более того, отдельные определения существуют для всех поддерживаемых в Unity платформ, и для каждой из них вы можете создать свой вариант кода. На этом мы перейдем к рассмотрению следующей важной платформы: интернета.

НАСТРОЙКИ КАЧЕСТВА

На генерируемое приложение также влияют настройки проекта, доступ к которым осуществляется через меню Edit. В частности, именно тут настраивается визуальное качество готового приложения. Выберите в меню Edit команду Project Settings, а затем в дополнительном меню — команду Quality.

На панели Inspector появятся элементы управления качеством, из которых важнее всего расположенная сверху группа флажков. В верхнем ряду находятся значки возможных платформ, а сбоку указаны варианты настроек качества. Установленные флажки показывают доступные для данной платформы настройки, а выделенные зеленым — текущие настройки. В большинстве случаев по умолчанию применяется вариант Fastest (соответствующий минимальному качеству), но если все выглядит плохо, можно выбрать вариант Fantastic; щелчок на стрелке в нижнем ряду под нужным значком платформы открывает меню.

Одновременное наличие в интерфейсе флажков и меню Default может показаться избыточным, но на самом деле это не так. Разные платформы зачастую обладают разными функциональными возможностями графических средств, поэтому Unity позволяет индивидуально задавать уровни качества для целевых платформ (например, самое высокое качество для настольных компьютеров и минимальное качество для мобильных устройств).



Сетка настроек качества на панели Inspector

13.2. Создание игр для интернета

Основная целевая платформа для создаваемых с помощью Unity игр — настольные компьютеры, но существуют и другие важные варианты развертывания, например развертывание в интернете. Оно требуется играм, которые запускаются в веб-браузерах и, соответственно, доступны для игроков в интернете.

13.2.1. Проигрыватель Unity и HTML5/WebGL

Раньше создаваемые в Unity варианты программ для интернета воспроизводились с помощью специальных подключаемых модулей браузера. В браузерах в то время попросту отсутствовали встроенные средства отображения трехмерной графики. Но в последние годы начал развиваться стандарт *WebGL*. С технической точки зрения он отличается от HTML5, хотя эти термины связаны друг с другом и часто используются как синонимы.

Начиная с Unity 5, в список платформ в окне Build Settings был добавлен вариант WebGL, который в следующих версиях превратился в основной вариант создания приложений для интернета. Именно это направление развития было выбрано в компании. Оно обусловлено в числе прочего давлением со стороны производителей браузеров, которые в реализации интерактивных веб-приложений, в том числе игр, предпочитают отходить от дополнительных подключаемых модулей в пользу `HTML5/WebGL`.

13.2.2. Игра, встроенная в веб-страницу

Откройте какой-нибудь другой проект (чтобы убедиться, что работать можно с любым проектом) и вызовите окно Build Settings. Перейдите к платформе WebGL и щелкните на кнопке Build. Появится окно выбора файла; введите для данного приложения имя WebTest и при необходимости выберите для него более безопасное место (вне проекта Unity).

На этот раз в процессе сборки будет создана папка, содержащая файл index.html, а также вложенные в нее папки с кодом и другими ресурсами игры. Откройте страницу index.html, и в центре вы увидите встроенную игру.

Никакими особыми характеристиками страница не обладает; это всего лишь пример для тестирования игры. На ней можно настроить код и даже представить собственную версию веб-страницы (эта тема будет рассмотрена ниже). Одним из наиболее важных вариантов адаптации является добавление возможности взаимодействия Unity с браузером. Давайте посмотрим, каким образом это делается.

13.2.3. Обмен данными с JavaScript в браузере

Созданная в Unity онлайновая игра может обмениваться данными с браузером (точнее, с запущенным в браузере сценарием JavaScript), причем обмен может идти в обоих направлениях — как от Unity к браузеру, так и от браузера к Unity. Для отправки сообщений браузеру следует написать код JavaScript в библиотеку кода, а Unity будет пользоваться функциями этой библиотеки с помощью специальных команд.

В обратной ситуации все несколько сложнее: код JavaScript в браузере идентифицирует объект по имени, после чего Unity передает этому именованному объекту сцены сообщение. То есть в сцене должен присутствовать объект, который будет получать данные от браузера.

Чтобы посмотреть, как все это выглядит на практике, создайте сценарий с именем `WebTestObject`. Кроме того, создайте в активной сцене пустой объект с именем `JSListener` (ему следует присвоить именно это имя, потому что оно фигурирует в коде листинга 13.4). Свяжите с этим объектом новый сценарий и скопируйте в него следующий код.

Листинг 13.2. Сценарий WebTestObject для тестирования механизма обмена данными с браузером

```
using UnityEngine;
using System.Runtime.InteropServices;

public class WebTestObject : MonoBehaviour {
  private string _message;

  [DllImport("__Internal")]   <-- Импортируем функцию из библиотеки JS.
  private static extern void ShowAlert(string msg);

  void Start() {
    _message = "No message yet";
  }

  void Update() {
    if (Input.GetMouseButtonDown(0)) {   <-- Щелчок мыши вызывает функцию в браузере.
      ShowAlert("Hello out there!");
    }
  }

  void OnGUI() {
    GUI.Label(new Rect(10, 10, 200, 20), _message);   <-- Отображаем сообщение в верхнем
  }                                                        левом углу экрана.

  public void RespondToBrowser(string message) {   <-- Функция, вызываемая браузером.
    _message = message;
  }
}
```

Основное новшество здесь — команда `DLLImport`. Она импортирует функцию из библиотеки JavaScript, которая будет использоваться в коде C#. Предполагается, что у нас есть библиотека JavaScript, поэтому сейчас мы ее напишем. Создайте папку с именем Plugins, внутри которой создайте папку WebGL. В ней должен находиться файл WebTest с расширением jslib. Для этого проще всего создать текстовый файл вне Unity, присвоить ему указанное имя и импортировать путем перетаскивания. Приложение Unity распознает его как библиотеку JavaScript. Введите в этот файл следующий код.

Листинг 13.3. Библиотека JavaScript для сценария WebTest

```
var TestLib = {

  ShowAlert: function(msg) {
    window.alert(Pointer_stringify(msg));
  },

}
mergeInto(LibraryManager.library, TestLib);
```

Импортированная функция, которая вызывается из кода на языке C#.

Этот файл jslib содержит как объект JavaScript с нужными функциями, так и команду, добавляющую этот объект в диспетчер библиотек Unity. Обратите внимание, что написанная функция кроме стандартных команд JavaScript включает в себя метод `Pointer_stringify()`. Переданная из Unity строка превращается в цифровой идентификатор, соответственно, Unity предоставляет функцию для поиска строки, на которую он указывает.

Теперь еще раз сгенерируйте приложение для интернета, чтобы посмотреть, как работает новый код. При щелчке на встроенной в веб-страницу игре сценарий `WebTestObject` вызывает функцию в коде JavaScript; сделайте несколько щелчков — и в браузере появится окно с оповещением!

ПРИМЕЧАНИЕ Для запуска кода в браузере в Unity есть также метод `Application.ExternalEval()`; метод `ExternalEval` вместо вызова определенных функций запускает произвольные фрагменты JavaScript. Это устаревший метод, и пользоваться им не стоит, но в некоторых случаях он привлекает своей простотой. Например, для перезагрузки страницы достаточно написать:

```
Application.ExternalEval("location.reload();")
```

Итак, мы протестировали сообщения, которые игра Unity посылает коду JavaScript на веб-странице, но страница, в свою очередь, может посылать сообщения Unity. Давайте посмотрим, как это происходит. Для этого на страницу потребуется добавить новый код и кнопки. К счастью, Unity предоставляет простой способ настройки веб-страниц. В частности, при создании сборки для платформы WebGL заполняется *шаблон* плеера. И можно выбрать пользовательский шаблон вместо заданного по умолчанию.

В операционных системах семейства Windows шаблоны находятся в папке установки Unity Editor\Data\PlaybackEngines\WebGLSupport\BuildTools\WebGLTemplates. На машинах Mac это папка /PlaybackEngines/WebGLSupport/BuildTools/WebGLTemplates. Откройте шаблон в текстовом редакторе. По большей части он состоит из стандартного кода HTML и JavaScript и набора специальных тегов, которые Unity замещает сгенерированной информацией. Сами встроенные шаблоны плееров Unity лучше не трогать, но они (особенно *минимальная версия*) могут послужить прекрасной основой для ваших собственных вариантов. Просто скопируйте минимальную версию в текстовый редактор.

В Unity создайте папку WebGLTemplates (без пробелов); сюда сохраняются пользовательские шаблоны. Внутри этой папки создайте папку с именем WebTest; именно в нее мы положим свой новый шаблон index.html. Откройте его в текстовом редакторе и введите следующий код.

Листинг 13.4. Шаблон WebGL, позволяющий браузеру взаимодействовать с Unity

```
<!doctype html>
<html lang="en-us">
<head>
<title>Unity WebGL Player | %UNITY_WEB_NAME%</title>
<style>
body { background-color: #333; }
</style>

<script src="%UNITY_WEBGL_LOADER_URL%"></script>
<script>
var gameInstance = UnityLoader.instantiate("gameContainer", "%UNITY_WEBGL_BUILD_
URL%");

function SendToUnity() {
  gameInstance.SendMessage("JSListener",
    "RespondToBrowser", "Hello from the browser!");
}
</script>
</head>

<body>
<div id="gameContainer" style="width: %UNITY_WIDTH%px; height: %UNITY_
    HEIGHT%px; margin: auto"></div>
<br><input type="button" value="Send" onclick="SendToUnity();" />
</body>
</html>
```

Превращаем страницу из белой в темную.

Метод SendMessage() указывает на именованный объект в Unity.

Кнопка, вызывающая функцию JavaScript.

В минимальную версию шаблона листинг 13.4 добавляет всего несколько строк. Два важных дополнения в данном случае — это функция в заголовке сценария и кнопка ввода внизу; добавленный стиль меняет цвет страницы, делая встроенную игру более заметной. HTML-тег кнопки связан с функцией JavaScript, которая вызывает в экземпляре Unity метод `SendMessage()`. Этот метод вызывает функцию именованного объекта в Unity; его первый параметр — это имя объекта, второй — название метода, а третий — строка, передаваемая при вызове метода.

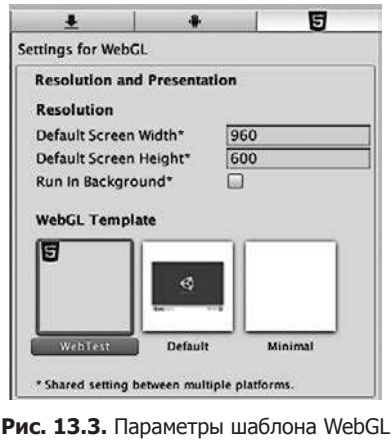


Рис. 13.3. Параметры шаблона WebGL

Итак, наш вариант шаблона готов, но еще нужно заставить Unity использовать его вместо заданного по умолчанию. Снова откройте настройки плеера (для этого нужно щелкнуть на кнопке Player Settings в окне Build Settings) и найдите раздел WebGL Template, показанный на рис. 13.3. В настоящее время выбран вариант Default, но в перечне присутствует и вариант WebTest (созданный нами). Остается только выбрать его.

Снова выполните сборку для платформы WebGL. Откройте сгенерированную вебстраницу. На этот раз в ее нижней части есть кнопка. Щелкните на ней, и вы увидите, как в Unity отобразилось измененное сообщение!

Итак, мы разобрались, как осуществляется взаимодействие с браузером в случае сборок для интернета. Осталось познакомиться с последней платформой (точнее, с набором платформ) — мобильными приложениями.

13.3. Сборки для мобильных устройств: iOS и Android

Мобильные устройства являются еще одной важной целевой платформой. По моим впечатлениям (впрочем, научно не подтвержденным), больше всего коммерческих игр в Unity создается именно в виде приложений для мобильных устройств.

ОПРЕДЕЛЕНИЕ *Мобильным* называется портативное устройство, которое пользователи носят с собой. Изначально этим термином называли смартфоны, но потом в эту группу попали еще и планшеты. Двумя наиболее распространенными компьютерными платформами для таких устройств являются iOS (от Apple) и Android (от Google).

Настройка процесса сборки для мобильных устройств сложнее, чем для настольных компьютеров и интернета, поэтому данный раздел относится к необязательным — можете прочитать его с ознакомительными целями, не выполняя упражнений; упражнения могут делать только те читатели, кто приобрел лицензию разработчика для iOS и установил инструменты разработчика для Android.

ВНИМАНИЕ Мобильные устройства настолько часто обновляются, что на момент чтения данной книги процесс сборки может отличаться от описываемого. Высокоуровневые концепции, скорее всего, останутся теми же, но точное описание последовательности выполняемых команд и нажимаемых кнопок вам придется искать в выложенной в интернете документации. Вот ссылки на страницы с документацией от Apple http://mng.bz/z1VP и Google https://developer.android.com/studio/build/.

СЕНСОРНЫЙ ВВОД

Ввод информации в случае мобильных устройств совсем не такой, как при работе с настольным компьютером или в интернете. Он осуществляется посредством прикосновений к экрану, а не с помощью мыши и клавиатуры. Соответственно, в Unity поддерживается функциональность обработки касаний, в том числе такой код, как `Input.touchCount` и `Input.GetTouch()`.

Эти команды используются при написании привязанного к платформе кода на мобильных устройствах. Но такой способ обработки ввода неудобен, поэтому существует ряд упрощающих ввод фреймворков. Например, поищите в магазине Asset Store фреймворки Fingers или Lean Touch.

Теперь, после всех этих оговорок, можно перейти к объяснению общего процесса сборки для iOS и Android. Еще раз напоминаю, что указанные платформы время от времени меняют детали этого процесса.

13.3.1. Настройка инструментов сборки

Мобильные устройства существуют отдельно от компьютера, на котором разрабатывается игра, и именно этот факт несколько осложняет процесс генерации и развертывания приложений. Потребуется настроить множество специализированных устройств, прежде чем можно будет щелкнуть на кнопке Build.

Инструменты сборки для iOS

На высоком уровне процесс развертывания созданной в Unity игры на платформе iOS требует построения Xcode-проекта, а затем его превращения в IPA (iOS App Package — пакет приложения для iOS). Создать IPA сразу Unity не может, так как все iOS-приложения должны проходить через инструменты сборки от Apple. Это означает, что нужно установить Xcode (программную интегрированную среду разработки от Apple), включая iOS SDK.

ВНИМАНИЕ Все это означает, что вы должны работать на компьютере Mac, так как Xcode запускается только в OS X. Разработка игры может выполняться на компьютере с Windows или Mac, а вот сборка iOS-приложения возможна только на машинах Mac.

Скачайте Xcode с сайта Apple со страницы https://developer.apple.com/xcode/.

ПРИМЕЧАНИЕ Чтобы продавать игры для iOS в App Store, требуется членство в программе Apple-разработчиков. Его стоимость составляет 99 долларов в год; подписка осуществляется на странице https://developer.apple.com/programs/.

После установки Xcode вернитесь в Unity и переключитесь на платформу iOS. Нужно скорректировать настройки проигрывателя для iOS-приложения (напоминаю, что для доступа к ним нужно открыть окно Build Settings и щелкнуть на кнопке Player Settings). В настройках проигрывателя вы должны сразу попасть на вкладку iOS, но если это не так, перейдите на нее, щелкнув на значке iPhone. Найдите внизу раздел Other Settings, а в нем — параметр Identification. Нужно скорректировать строку Bundle Identifier, чтобы Apple-устройство смогло правильно идентифицировать приложение.

ПРИМЕЧАНИЕ Параметр Bundle Identifier для iOS называется Package Name в версии для Android. Оба параметра применяются одним и тем же способом для обеих платформ. Идентификатор в данном случае должен составляться по правилам, применимым для любого другого пакета кода: буквами нижнего регистра в форме com.названиекомпании.названиепродукта.

Другим важным параметром, существующим как для iOS, так и для Android, является Version (то есть номер версии приложения). Но его форма в большинстве случаев зависит от платформы; к примеру, недавно в iOS был добавлен короткий номер версии, отдельный от основной версии пакета. Есть еще параметр Scripting Backend, которому раньше по умолчанию присваивался вариант Mono, но новая технология IL2CPP поддерживает обновления платформ, например 64-битные двоичные файлы.

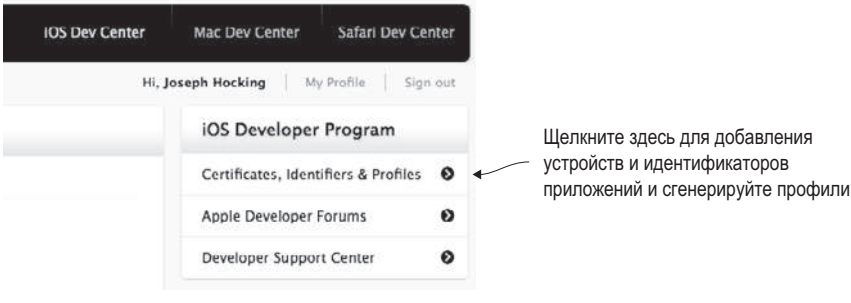
Теперь щелкните на кнопке Build. Выберите место для нового файла сборки, там появится сгенерированный Xcode-проект. Он допускает прямое редактирование (некоторые простые модификации могут быть частью сценария пост-сборки). В любом случае,

откройте этот проект; в папке со сборкой находится множество файлов, вам нужно дважды щелкнуть на файле с расширением .xcodeproj (он помечен значком с изображением чертежа). Интегрированная среда разработки Xcode загрузит это приложение; большая часть настройки данного проекта уже выполнена в Unity, вам же остается настроить профиль, который будет использоваться в дальнейшем.

ПРОФИЛИ IOS

Из всех аспектов разработки iOS самым часто меняющимся являются профили (provisioning profiles). Это файлы, используемые для идентификации и авторизации. Apple строго контролирует, какие приложения могут запускаться на каждом устройстве; отправленные в Apple на утверждение приложения пользуются специальными профилями, позволяющими добавлять их в App Store, в то время как находящиеся на стадии разработки приложения имеют профили, связанные с зарегистрированными устройствами.

Идентификатор UDID своего устройства iPhone и ID приложения (это параметр Bundle Identifier в Unity) нужно добавить на панель управления, расположенную на сайте Apple для разработчиков iOS. Полностью этот процесс объясняется на странице https://developer.apple.com/support/code-signing/.



Место управления профилями в центре разработчиков iOS

Убедитесь, что параметр Scheme Destination на верхней панели окна не указывает на симулятор, а имеет значение iOS Device (в противном случае часть настроек окажется недоступной). Среда разработки Xcode попытается автоматически настроить профили, но при необходимости их можно отредактировать вручную. Выберите свое приложение в списке проектов с левой стороны среды разработки Xcode. Появятся несколько относящихся к этому проекту вкладок; на вкладке Build Settings найдите раздел Code Signing, чтобы настроить профили, как показано на рис. 13.4.

После настройки профилей можно приступать к созданию приложения. Выберите в меню Product команду Run или Archive. Это меню содержит множество команд, в том числе соблазнительную команду Build, но для наших целей требуется только один из вышеупомянутых вариантов. Команда Build генерирует исполняемые файлы, но не собирает их в пакет для iOS, в то время как

- команда Run тестирует приложение на устройстве iPhone, связанном с компьютером при помощи USB-кабеля;
- команда Archive создает прикладной пакет, который можно пересылать на другие зарегистрированные устройства (у Apple это называется *специальной сборкой*).

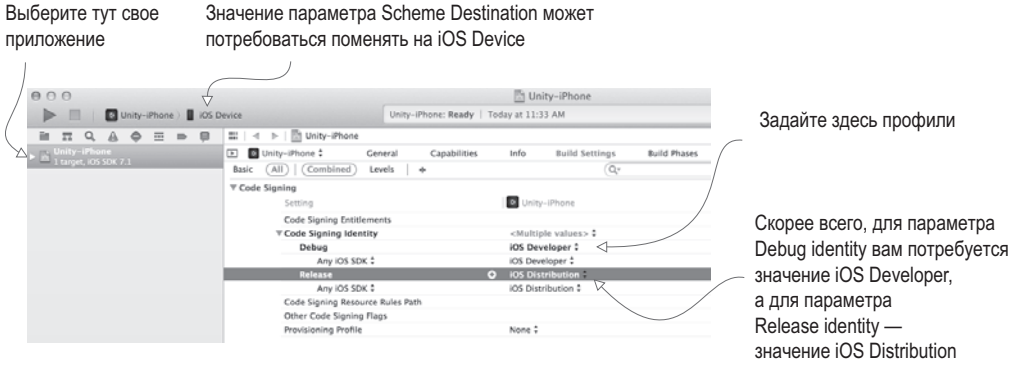


Рис. 13.4. Настройка профилей в интегрированной среде разработки Xcode

Команда Archive создает не готовый прикладной пакет, а, скорее, комплект в промежуточном состоянии между необработанными файлами кода и IPA-файлом. Готовый архив появляется в окне Organizer интегрированной среды разработки Xcode; выберите сгенерированный архив в этом окне и щелкните на расположенной справа кнопке Export. Появится вопрос, каким образом следует распространять приложение — через магазин или в виде специальной сборки.

Выбрав распространение в виде специальной сборки, вы получите IPA-файл, который можно будет отправить тестерам. Можно сделать это напрямую, передав им файл для установки через iTunes, но удобнее воспользоваться сервисом TestFlight (https://developer.apple.com/testflight/).

Инструменты сборки для Android

В отличие от приложений iOS, файлы формата APK (Android Application Packge) Unity может генерировать напрямую. Для этого нужно добавить в Unity путь к Android SDK, включающий в себя нужный компилятор. Скачайте Android SDK с сайта Android и укажите путь к этому файлу в окне Unity Preferences, как показано на рис. 13.5. Скачать SDK можно здесь: http://developer.android.com/sdk/index.html.

СОВЕТ Для базовой сборки вам понадобится только SDK в режиме командной строки. Но, скорее всего, вы захотите загрузить Android Studio. Мы будем использовать этот инструмент позже в этой главе.

После этого нужно задать идентификатор приложения, как вы уже делали для iOS. Параметр Package Name находится в настройках проигрывателя на панели Inspector: укажите идентификатор в виде com.названиекомпании.названиепродукта (как объяснялось в разделе 13.3.1). Затем щелкните на кнопке Build, чтобы запустить процесс сборки. Как и в случае с другими сборками, вас первым делом попросят указать, куда вы хотите сохранить файл. В указанном месте будет создан APK-файл.

Сгенерированный пакет приложения следует установить на устройство. Получить APK-файл на телефон Android можно, скачав его из интернета или посредством USB-подключения устройства к компьютеру (так называемый *режим sideload*). Конкретная процедура передачи данных зависит от устройства, а после ее завершения приложение

можно установить с помощью диспетчера файлов. По какой-то причине в устройства Android диспетчеры файлов не встраиваются, но их можно бесплатно скачать из магазина приложений Play Store.

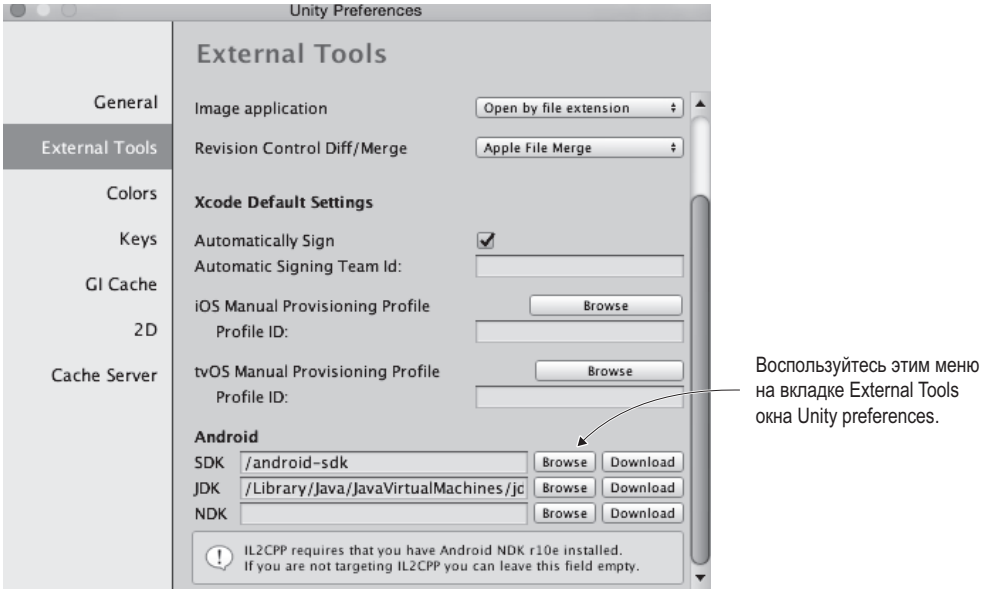


Рис. 13.5. Задание пути к Android SDK в окне Unity preference

Найдите в диспетчере файлов свой APK-файл и установите приложение. Как видите, в своей основе процесс сборки для платформы Android намного проще, чем для платформы iOS. К сожалению, с процессами настройки сборки и добавления внешних модулей ситуация ровно обратная. В данном случае все намного сложнее, чем для iOS. Подробности вы узнаете в разделе 13.3.3, а пока давайте поговорим о сжатии текстур.

13.3.2. Сжатие текстур

Ресурсы, в том числе и текстуры, могут сильно увеличивать файл приложения. Решается эта проблема теми или иными вариантами сжатия. Мобильные приложения не должны занимать много места, поэтому для них применяются различные способы сжатия изображений, каждый со своими достоинствами и недостатками. Соответственно, может возникнуть необходимость внесения коррективов в процесс сжатия текстур в Unity.

Управление сжатием текстур является неотъемлемой частью создания приложений для мобильных устройств, хотя эта процедура может применяться и для остальных платформ. Впрочем, в силу большей технической зрелости этих платформ на этот аспект можно не обращать особого внимания, в то время как мобильные устройства к нему крайне чувствительны.

Сжатие текстур в данном случае выполнит Unity; в большинстве инструментов разработки эту процедуру приходится выполнять вручную, в то время как Unity, как

правило, импортирует несжатые изображения, постфактум указывая вариант сжатия в настройках импорта (рис. 13.6).

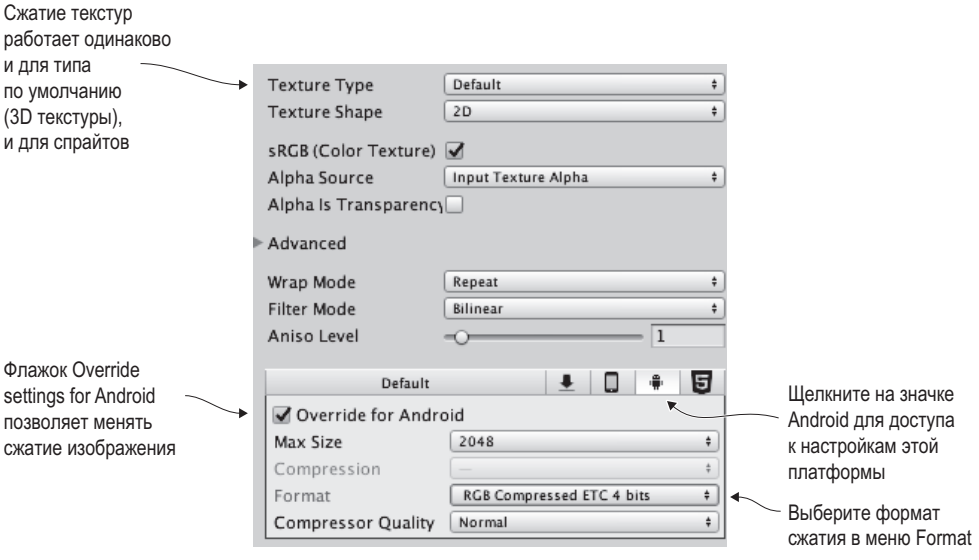


Рис. 13.6. Настройки сжатия текстур на панели Inspector

Эти настройки сжатия предоставляются по умолчанию, и в отдельных случаях может потребоваться их корректировка. В частности, для платформы Android сжатие изображений имеет свои особенности. По большей части они обусловлены многообразием устройств Android. Например, так как все устройства iOS пользуются практически одинаковыми видеопроцессорами, в приложениях iOS может использоваться процедура сжатия, оптимизированная для их GPU — ускорителей графики. Для приложений Android подобное единообразие аппаратного обеспечения недоступно, поэтому для них процедуру сжатия текстур приходится сводить к набору минимально необходимых действий, а именно: все устройства iOS используют графические ускорители PowerVR, соответственно, в приложениях для iOS может применяться оптимизированный формат сжатия PVR. Эти графические ускорители порой встречаются и у устройств Android, но с такой же частотой встречаются ускорители Adreno от Qualcomm, Mali от ARM и другие варианты. В результате приложения для Android в общем случае пользуются алгоритмом Ericsson (Ericsson Texture Compression, ETC), который поддерживается всеми устройствами Android. В Untity по умолчанию применяется ETC2 (вторая, усовершенствованная версия).

В большинстве случаев прекрасно подходят настройки по умолчанию, но если требуется отредактировать сжатие текстуры, пользуйтесь настройками, показанными на рис. 13.6. Чтобы переопределить параметры по умолчанию для платформы Android, перейдите на вкладку Android и выберите нужный формат сжатия в меню Format (как ни странно, оно называется *именно так*, а не Compression). Иногда оказывается, что некоторые важные изображения лучше оставить без сжатия. Они будут иметь большой

размер, но куда лучшее качество. Когда большая часть текстур подвергается сжатию, единичные текстуры исходного размера не должны слишком сильно повлиять на размер итогового файла.

На этом мы закончим обсуждение текстур и перейдем к последней теме этой главы — разработке подключаемых модулей.

13.3.3. Подключаемые модули

Инструмент Unity обладает богатейшей встроенной функциональностью, но по большей части это общая для всех платформ функциональность. Использование же привязанных к конкретной платформе наборов инструментов (таких, как Play Game Services для Android) часто требует для Unity дополнительных модулей.

ПРИМЕЧАНИЕ Для функциональных особенностей, связанных с платформами iOS и Android, доступно множество готовых модулей. Принцип управления ими аналогичен описываемому в этом разделе, просто вы пользуетесь уже готовым кодом, написанным специально для вас.

Процесс обмена данными с внутренними модулями аналогичен процессу обмена данными с браузером. Со стороны Unity есть специальные команды, вызывающие функции внутри модулей. Модули же, со своей стороны, для отправки сообщений объектам в Unity-сценах пользуются методом `SendMessage()`. Конкретная реализация кода зависит от платформы, но принцип функционирования во всех случаях сохраняется.

ВНИМАНИЕ Как и исходный процесс сборки, процесс разработки подключаемых модулей для мобильных устройств часто меняется — не со стороны Unity, а со стороны кода аппаратной платформы. Я описываю в этой главе общий принцип, а актуальную информацию, касающуюся деталей реализации, вы можете найти в интернете.

Кроме того, модули для обеих платформ Unity хранит в одном и том же месте. Создайте на вкладке Project папку Plugins. Внутри этой папки создайте еще две — для Android и для iOS; их содержимое Unity будет копировать в процессе сборки.

У файлов подключаемых модулей также есть настройки, указывающие, для какой платформы они предназначены. Как правило, Unity решает это автоматически (модули из папки iOS настраиваются под iOS, модули из папки Android — под Android и т. п.), но если нужно, ищите эти настройки на панели Inspector.

Подключаемые модули iOS

Подключаемый модуль — это всего лишь некий код для аппаратной платформы, к которому обращается Unity. Поэтому начните с создания сценария `TestPlugin` (скопируйте в него код следующего листинга).

Листинг 13.5. Сценарий TestPlugin, вызывающий из Unity код для iOS

```
using UnityEngine;
using System;
using System.Collections;
using System.Runtime.InteropServices;
```

```
public class TestPlugin : MonoBehaviour {
  private static TestPlugin _instance;

  public static void Initialize() {
    if (_instance != null) {
      Debug.Log("TestPlugin instance was found. Already initialized");
      return;
    }
    Debug.Log("TestPlugin instance not found. Initializing...");

    GameObject owner = new GameObject("TestPlugin_instance");
    _instance = owner.AddComponent<TestPlugin>();
    DontDestroyOnLoad(_instance);
  }

  #region iOS
  [DllImport("__Internal")]
  private static extern float _TestNumber();

  [DllImport("__Internal")]
  private static extern string _TestString(string test);
  #endregion iOS

  public static float TestNumber() {
    float val = 0f;
    if (Application.platform == RuntimePlatform.IPhonePlayer)
      val = _TestNumber();
    return val;
  }

  public static string TestString(string test) {
    string val = "";
    if (Application.platform == RuntimePlatform.IPhonePlayer)
      val = _TestString(test);
    return val;
  }
}
```

Объект создается в этой статической функции, поэтому в редакторе его создавать не нужно.

Тег, определяющий раздел кода; сам по себе он ничего не делает.

Ссылка на функцию в коде iOS.

Вызывается в случае платформы IPhonePlayer.

Первым делом обратите внимание, что статическая функция `Initialize()` создает в сцене постоянный объект, избавляя от необходимости делать это вручную в редакторе Unity. Код, создающий объекты с нуля, вам раньше не встречался, так как в большинстве случаев намного проще воспользоваться для этой цели шаблоном экземпляра, но сейчас лучше будет получить нужный объект программно (это даст возможность пользоваться сценарием модуля без редактирования сцены).

Именно здесь происходит основное действие, в том числе использование атрибута `DLLImport` и статических внешних команд. Именно они связывают Unity с функциями написанного нами кода для устройств. Затем эти функции вызываются в методах сценария (с условным оператором, проверяющим, что код запущен на платформе iPhone/iOS).

Теперь нужно протестировать функции модуля. Создайте сценарий `MobileTestObject`, а также пустой объект сцены, с которым нужно будет связать этот сценарий. Добавьте в сценарий следующий код:

Листинг 13.6. Использование подключаемого модуля в сценарии MobileTestObject

```
using UnityEngine;
using System.Collections;

public class MobileTestObject : MonoBehaviour {
  private string _message;

  void Awake() {
    TestPlugin.Initialize();  <-- Инициализация модуля в начале кода.
  }

  // Используем это для инициализации
  void Start() {
    _message = "START: " + TestPlugin.TestString("ThIs Is A tEsT");
  }

  // Функция Update вызывается в каждом кадре
  void Update() {

    // Убеждаемся, что пользователь коснулся экрана
    if (Input.touchCount==0){return;}

    Touch touch = Input.GetTouch(0);  <-- Отвечаем на ввод данных методом касания.
    if (touch.phase == TouchPhase.Began) {
      _message = "TOUCH: " + TestPlugin.TestNumber();
    }
  }

  void OnGUI() {
    GUI.Label(new Rect(10, 10, 200, 20), _message);  <-- Отображаем сообщение в углу экрана.
  }
}
```

Этот сценарий инициализирует представляющий модуль объект и в ответ на ввод данных прикосновением к экрану вызывает методы этого модуля. После запуска сценария на устройстве вы увидите, как в ответ на прикосновение к экрану меняется тестовое сообщение в углу.

Теперь осталось написать код, на который ссылается сценарий `TestPlugin`. Для устройств iOS он пишется на языке Objective C и (или) C, поэтому потребуется как файл заголовка с расширением .h, так и файл реализации с расширением .mm. Как уже упоминалось, такие файлы должны находиться в папке Plugins/iOS/ на вкладке Project. Создайте в этой папке сценарии `TestPlugin.h` и `TestPlugin.mm`; в файл с расширением .h скопируйте следующий код.

Листинг 13.7. Заголовок TestPlugin.h для iOS-кода

```
#import <Foundation/Foundation.h>

@interface TestObject : NSObject {
NSString* status;
}

@end
```

Объяснение таких сложных вопросов, как программирование для iOS, выходит за рамки данной книги, поэтому, чтобы понять смысл данного кода, обратитесь к справочной документации. Код из следующего листинга нужно ввести в файл с расширением .mm.

Листинг 13.8. Реализация TestPlugin.mm

```
#import "TestPlugin.h"

@implementation TestObject
@end

NSString* CreateNSString (const char* string)
{
if (string)
return [NSString stringWithUTF8String: string];
else
return [NSString stringWithUTF8String: ""];
}

char* MakeStringCopy (const char* string)
{
if (string == NULL)
return NULL;

char* res = (char*)malloc(strlen(string) + 1);
strcpy(res, string);
return res;
}

extern "C" {
  const char* _TestString(const char* string) {
    NSString* oldString = CreateNSString(string);
    NSString* newString = [oldString uppercaseString];
    return MakeStringCopy([newString UTF8String]);
  }

  float _TestNumber() {
    return (arc4random() % 100)/100.0f;
  }
}
```

И снова подробное объяснение кода выходит за рамки нашей книги. Обратите внимание, сколько функций используется для преобразования строк Unity в форму, которую понимает данный код.

СОВЕТ В рассмотренном примере взаимодействие осуществлялось только в одном направлении: от Unity к модулю. Но код модуля также может отправлять данные в Unity методом `UnitySendMessage()`. Сообщения отправляются именованному объекту в сцене; в процессе инициализации модуля создается предназначенный именно для этого объект `TestPlugin_instance`.

Теперь, когда у нас есть код для аппаратной платформы, можно сгенерировать приложение для iOS и протестировать его работу на устройстве. Потрясающе! Но мы пока научились создавать модули только для iOS, давайте посмотрим, как это делается на Android.

Подключаемые модули для Android

Создание модуля для Android со стороны Unity представляет собой практически аналогичный процесс. Не потребуется даже вносить правки в сценарий `MobileTestObject`. Дополнения, показанные в следующем листинге, нужно вставить только в сценарий `TestPlugin`.

Листинг 13.9. Редактирование сценария TestPlugin под платформу Android

```
...
  #region iOS
  [DllImport("__Internal")]
  private static extern float _TestNumber();

  [DllImport("__Internal")]
  private static extern string _TestString(string test);
  #endregion iOS

#if UNITY_ANDROID
  private static Exception _pluginError;
  private static AndroidJavaClass _pluginClass;
  private static AndroidJavaClass GetPluginClass() {
    if (_pluginClass == null && _pluginError == null) {
      AndroidJNI.AttachCurrentThread();
      try {
        _pluginClass = new AndroidJavaClass("com.testcompany.testplugin.
                       TestPlugin");
      } catch (Exception e) {
        _pluginError = e;
      }
    }
    return _pluginClass;
  }

  private static AndroidJavaObject _unityActivity;
  private static AndroidJavaObject GetUnityActivity() {
    if (_unityActivity == null) {
      AndroidJavaClass unityPlayer = new AndroidJavaClass("com.unity3d.player.
UnityPlayer");
      _unityActivity = unityPlayer.GetStatic<AndroidJavaObject>
                       ("currentActivity");
    }
    return _unityActivity;
  }
#endif

  public static float TestNumber() {
    float val = 0f;
    if (Application.platform == RuntimePlatform.IPhonePlayer)
      val = _TestNumber();
#if UNITY_ANDROID
    if (!Application.isEditor && _pluginError == null)
      val = GetPluginClass().CallStatic<int>("getNumber");
#endif
    return val;
  }
```

Обеспечиваемая Unity функциональность AndroidJNI.

Имя запрограммированного нами класса; измените его, если нужно.

Unity создает экран для приложения Android.

Обращение к функциям в модуле .jar.

```
  public static string TestString(string test) {
    string val = "";
    if (Application.platform == RuntimePlatform.IPhonePlayer)
      val = _TestString(test);
#if UNITY_ANDROID
      if (!Application.isEditor && _pluginError == null)
      val = GetPluginClass().CallStatic<string>("getString", test);
#endif
    return val;
  }
}
```

Думаю, вы заметили, что большинство дополнений появилось внутри определений платформы `UNITY_ANDROID`; как объяснялось в начале этой главы, эти директивы компилятора приводят к тому, что код применяется только для определенной платформы, во всех остальных случаях он просто игнорируется. Если код для iOS на остальных платформах не производил никаких действий (он ничего не делал, но и не вызывал ошибок), код модулей для Android будет компилироваться, только если инструмент Unity настроен на работу с платформой Android.

В частности, обратите внимание на обращения к объекту `AndroidJNI`. Эта система в Unity обеспечивает связь с кодом устройства для Android. Другим, возможно, не совсем понятным элементом будет класс `Activity`; в приложениях для Android он соответствует экрану приложения. Экраном в нашем случае будет служить Unity, значит, коду модуля нужен доступ к этому экрану, чтобы в случае необходимости обойти его.

Наконец, вам требуется код для Android. Если код для iOS пишется на таких языках, как Objective C и C, приложения для Android программируются на языке Java. Но для модуля мы не можем взять и написать код на Java; модуль должен находиться в JAR-архиве. К сожалению, в данном случае подробное рассмотрение происходящего выходит за рамки книги, поэтому я сделаю только краткий обзор. Первым делом следует установить интегрированную среду разработки Android Studio, если это не было сделано во время установки Android SDK.

Этапы настройки проекта подключаемого модуля в Android Studio показаны на рис. 13.7. Первым делом выберите команду Start a New Project. В появившемся конфигурационном окне присвойте новому проекту имя TestPluginProj; содержимое поля Company domain в данном случае не имеет значения, так как мы делаем тестовый проект, но обязательно обратите внимание, в какую папку он будет сохранен, так как позднее вам потребуется его найти. Щелкните на кнопке Next. Параметр Minimum SDK сейчас тоже не имеет значения, но выберите вариант Add No Activity (так как мы конструируем подключаемый модуль, а не приложение для Android).

После щелчка на кнопке Finish придется немного подождать, так как настройка нового проекта занимает некоторое время. Как только откроется редактор, выберите в меню File команду New > New Module, чтобы добавить библиотеку. В окне настроек выберите вариант Java Library, присвойте библиотеке имя test-plugin, щелкните на кнопке Edit, чтобы ввести в поле Java package name значение com.testcompany.testplugin, а в поле Java class name — значение TestPlugin. Откройте вкладку Project (это делается кнопкой, расположенной на левом краю экрана), раскройте дерево test-plugin и дважды щелкните на строке TestPlugin, чтобы открыть этот класс.

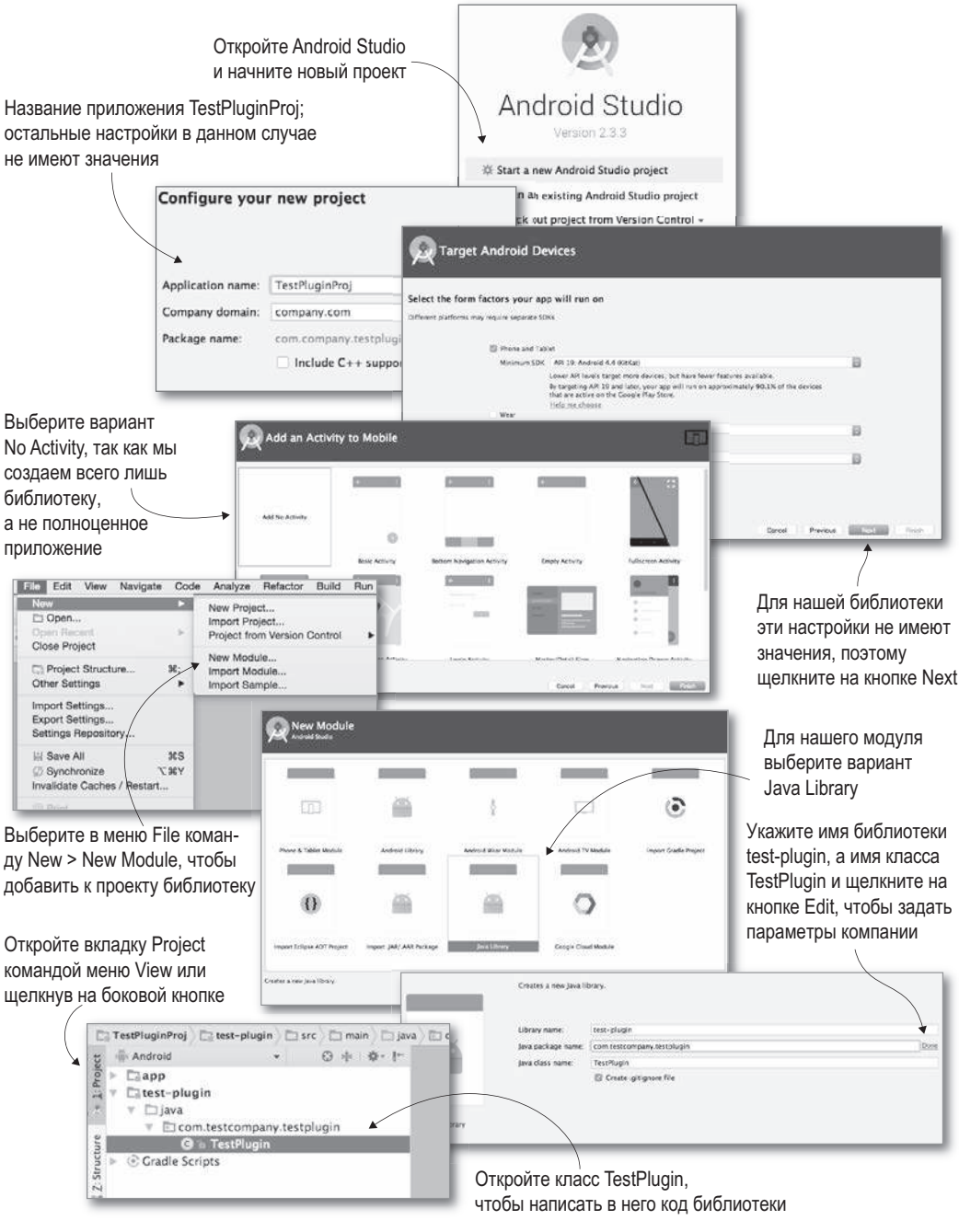


Рис. 13.7. Подготовка интегрированной среды разработки Android Studio к созданию подключаемого модуля

Класс `TestPlugin` пока пуст, и в него надо добавить функции подключаемого модуля. Скопируйте в него код на языке Java из листинга 13.10.

Листинг 13.10. Файл TestPlugin.java, компилируемый в JAR-архив

```
package com.testcompany.testplugin;

public class TestPlugin {
  private static int number = 0;

  public static int getNumber() {
    number++;
    return number;
  }

  public static String getString(String message) {
    return message.toLowerCase();
  }
}
```

Теперь этот код можно упаковать в архив JAR. В меню Build выберите команду Build APK; вопреки своему названию, эта команда выполняет и сборку библиотек. После завершения сборки найдите в папке /test-plugin/build/libs/ файл test-plugin.jar. Перетащите его в Unity в папку с подключаемыми модулями Android, чтобы выполнить импорт.

МАНИФЕСТ ANDROID И ПАПКА RESOURCES

Для нашего простого тестового модуля это было не нужно, но модулям для Android часто требуется редактировать файл манифеста. Все приложения для Android контролируются основным конфигурационным файлом AndroidManifest.xml; если вы не предоставляете этот файл, Unity создает его базовую версию, но лучше сделать это самостоятельно, поместив файл манифеста в папку Plugins/Android/ вместе с содержащим наш модуль JAR-архивом.

При сборке приложения для Android система Unity помещает сгенерированный файл манифеста в папку Temp по адресу StagingArea/AndroidManifest.xml. Скопируйте его, чтобы отредактировать вручную (пример такого файла вы найдете и в сопроводительных фрагментах кода).

Аналогично, существует папка res, в которой сохраняются такие ресурсы, как, к примеру, собственные нестандартные значки; она создается внутри папки plugins.

Сгенерированный сценарием сборки JAR-файл сохраняется в папке Plugins/Android (для ясности люди часто копируют сюда весь Java-проект, но с технической точки зрения значение имеет только JAR-архив). Выполните сборку игры. Теперь при любом касании экрана сообщение будет меняться. Аналогично iOS-модулям, модули для Android могут посылать данные объекту сцены методом `UnityPlayer.UnitySendMessage()` (этому Java-коду потребуется импортировать из Unity библиотеку/JAR-архив для проигрывателя Android).

Я обошел вниманием большую часть процесса разработки JAR-архивов для Android, но это связано как с крайней трудоемкостью, так и с частой изменчивостью этого процесса. Если вы хотите самостоятельно разрабатывать модули игр для Android, читайте документацию на сайте Android-разработчиков.

Поздравляю, вы добрались до конца!

Мои поздравления, вы освоили все этапы развертывания Unity-игр на мобильных устройствах. Базовый процесс сборки для всех платформ очень прост (и осуществляется с помощью единственной кнопки), трудности представляет настройка приложений под различные платформы. Но теперь вы готовы к самостоятельному плаванию и к созданию собственных игр!

Заключение

- Для расширения функциональности приложений Unity поддерживает нестандартные подключаемые модули.
- Существует множество параметров сборок, к их числу относятся значок приложения и его название.
- Веб-игры могут обмениваться данными с веб-страницей, в которую они встроены, позволяя создавать интересные варианты приложений.
- Для расширения функциональности Unity поддерживает нестандартные подключаемые модули.

Послесловие

Теперь вы знаете все, что нужно, чтобы с помощью Unity создать полноценную игру. Точнее, вы знаете все по поводу программирования, ведь для первоклассной игры требуются еще и такие вещи, как первоклассная графика и звуковое сопровождение. Но чтобы стать талантливым разработчиком игр, одних технических навыков мало. Посмотрим правде в глаза: ваша конечная цель не сводится к освоению Unity. Вы хотите создавать хорошие игры, а Unity — всего лишь инструмент (пусть и очень хороший), который может в этом помочь.

Кроме технических навыков, позволяющих реализовать все особенности игры, требуется и такое качество, как твердость характера. Я имею в виду упорство и веру в себя, которые дают силы продолжать работу над сложными, но интересными проектами. Это называют «способностью доводить дело до конца». Выработать этот навык можно единственным способом: закончить множество проектов. Это выглядит как замкнутый круг (чтобы уметь завершать проекты, сначала нужно завершить множество проектов), но важно понимать, что достигнуть этой цели с маленьким и простым проектом куда проще, чем с большим и сложным.

Соответственно, для продвижения вперед нужно начинать с множества мелких проектов, постепенно переходя от простого к сложному. Многие новички совершают одну и ту же ошибку, сразу хватаясь за сложную задачу. Это происходит по двум причинам. Во-первых, желание самостоятельно воссоздать свою любимую игру. Во-вторых, недооценка сложности задачи. Кажется, что все начинается отлично, но быстро накапливается множество сложных задач, и в конечном счете новичок падает духом и прекращает свои попытки.

Начинать нужно с малого. С проектов, реализация которых кажется почти очевидной. Именно такие проекты, «простые почти до банальности», приведены в данной книге. Их выполнение даст вам хороший старт. Вы сможете попробовать свои силы в реализации более крупного проекта. Главное, не торопиться и не приниматься сразу за сложную задачу. Вы нарабатываете навыки и уверенность в своих силах, так что усложнение должно быть постепенным.

Этот совет в ответ на вопрос, как научиться разработке игр, вы будете слышать практически все время. По просьбе создателей Unity авторы сериала Extra Credits (это сериал, посвященный разработке игр) создали несколько серий, посвященных первым шагам в изучении этого искусства. Ссылки на них вы найдете на странице http://mng.bz/X9ec. Видео снабжено русскими субтитрами.

Проектирование игр

Сериал от Extra Credits не сводится к набору видеороликов, заказанных Unity. В нем рассматриваются самые разные темы, но основной упор делается на такую дисциплину, как проектирование игр.

ОПРЕДЕЛЕНИЕ *Проектированием игр* (game design) называется процесс описания будущей игры, включающий формулировку целей, правил и задач. Его не следует путать с *визуальным проектированием* (visual design), в процессе которого разрабатывается внешний вид, а не функциональность; это распространенная ошибка, потому что в среднем, по статистике, люди привыкли употреблять английский термин *design* в контексте графического дизайна.

ОПРЕДЕЛЕНИЕ Центральной частью проектирования игр является разработка *игровой механики*, то есть принципов взаимодействия игры с игроком; это отдельные действия (или системы действий) в процессе игры. Механика часто задается правилами игры, в то время как игровые задачи возникают путем применения механики к конкретным ситуациям. Например, перемещения персонажа — это механика, а вот его блуждания по лабиринту — игровая задача, базирующаяся на этой механике.

Теория проектирования игр зачастую слишком сложна для новичков. Самые успешные (и приносящие чувство удовлетворения своим создателям!) игры имеют интересную и оригинальную игровую механику. Но уделять этому аспекту слишком пристальное внимание на начальном этапе не следует, потому что это будет отвлекать вас от других аспектов разработки, например изучения программирования. Начинать лучше с попыток проектирования уже существующих игр. Их клонирование — прекрасная практика на начальных этапах. Постепенно вы приобретете все нужные навыки, чтобы приступить к проектированию собственных игр.

Принимая во внимание вышесказанное, любой успешный разработчик игр должен интересоваться их проектированием. Существует множество способов знакомства с этой темой. Вы уже знаете о сериале *Extra Credits*, а это ссылки на полезные ресурсы (на английском):

- www.gamasutra.com — предложения работы, обновления игр, хорошие/плохие новости об играх; все, что вы хотите знать об искусстве создания игр и связанной с этим коммерческой деятельности.
- www.lostgarden.com — хорошо написанные и продуманные статьи о теории проектирования игр, искусстве их создания и коммерческом применении результатов вашего труда (цитата с главной страницы сайта).
- www.sloperama.com — перейдите по ссылке School-a-rama (верхний правый прямоугольник) на страницу с советами по проектированию и созданию игр.

Существуют и прекрасные книги, посвященные данной теме. Например:

- *Game Design Workshop*, Tracy Fullerton.
- *A Theory of Fun for Game Design*, Raph Koster.
- *The Art of Game Design*, Jesse Schell.
- Мэннинг Д., Батфилд-Эддисон П. Unity для разработчика: Мобильные мультиплатформенные игры. — СПб.: Питер, 2018. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

Продвижение вашей игры

Четвертое видео от *Extra Credits* посвящено продвижению игр. Иногда разработчики избегают мыслей о том, как поступить с готовой игрой. Они сосредоточены на процессе создания, но такой подход, скорее всего, закончится провалом. Ведь самую лучшую игру вряд ли можно назвать успешной, если никто о ней не узнает!

Слова *продвижение*, *маркетинг* часто ассоциируются с рекламой. При наличии денежных средств реклама не самый плохой способ продажи результатов своего труда. Но существуют и более дешевые и даже бесплатные способы оповестить мир о появлении новой игры. Их детали имеют свойство меняться, но в видео упоминаются и общие стратегии, такие как публикация твитов (или публикация в социальных сетях вообще) и создание рекламного видео для публикации на сайте YouTube с обзорами, комментариями блогеров и т. п. Будьте настойчивы и проявляйте изобретательность!

А теперь вперед, к созданию прекрасных игр. Для этого у вас есть прекрасный инструмент Unity, которым вы научились пользоваться. Удачи в ваших начинаниях!

Приложения

Приложение А. Перемещение по сцене и клавиатурные комбинации

Управление приложением Unity осуществляется с помощью мыши и клавиатуры, но *точный порядок действий* новичкам неочевиден. Мышь и клавиатура позволяют перемещаться по сцене и осматривать трехмерные объекты. Есть в Unity и клавиатурные комбинации, ускоряющие выполнение наиболее распространенных операций.

В этом приложении я расскажу об элементах управления вводом, кроме того, вы можете воспользоваться информацией со следующих страниц: https://docs.unity3d.com/ru/current/Manual/SceneViewNavigation.html и https://docs.unity3d.com/ru/current/Manual/UnityHotkeys.html

А.1. Навигация с помощью мыши

Навигация в сцене осуществляется с помощью трех основных маневров: перемещение, облет и масштабирование. Все они сводятся к нажатию кнопки мыши и перетаскиванию с одновременным удерживанием комбинации клавиш Ctrl и Alt (или Option на компьютерах Mac). Конкретный порядок действий зависит от того, с какой мышью — однокнопочной, двухкнопочной или трехкнопочной — вы работаете; все эти варианты перечислены в табл. А.1.

Таблица А.1. Элементы управления навигацией для мышей различных типов

Действие	Трехкнопочная мышь	Двухкнопочная мышь	Однокнопочная мышь
Перемещение (Move)	Нажатие средней кнопки мыши / перетаскивание	Удерживание клавиш Alt и Command плюс нажатие левой кнопки мыши / перетаскивание	При нажатых клавишах Alt+Command+нажатие кнопки мыши / перетаскивание
Облет (Orbit)	Удерживание клавиши Alt плюс нажатие левой кнопки мыши / перетаскивание	Удерживание клавиши Alt плюс нажатие левой кнопки мыши / перетаскивание	Удерживание клавиши Alt плюс нажатие кнопки мыши / перетаскивание
Масштабирование (Zoom)	Удерживание клавиши Alt плюс нажатие правой кнопки мыши / перетаскивание	Удерживание клавиши Alt плюс нажатие правой кнопки мыши / перетаскивание	Удерживание клавиш Alt и Ctrl плюс нажатие кнопки мыши / перетаскивание

ПРИМЕЧАНИЕ Хотя Unity можно управлять с помощью одно- и двухкнопочной мыши, я крайне рекомендую приобрести трехкнопочную мышь (которая к тому же прекрасно работает с Mac OS X).

Навигационные маневры выполняются не только с помощью мыши, есть и варианты управления с клавиатуры. При нажатой правой кнопке мыши клавиши WASD начинают играть роль клавиш со стрелками, перемещая игрока по сцене в манере, общей для большинства игр от первого лица. Выполнение операций при нажатой клавише Shift ускоряет эти операции, заставляя персонаж перемещаться быстрее. Но самой важной является клавиша F, нажатие которой при выделенном объекте в сцене вызывает автоматическое панорамирование и масштабирование, позволяющее сфокусироваться на этом объекте. Если, перемещаясь по сцене, вы утратили представление о том, где находитесь, достаточно выделить какой-либо объект на вкладке Hierarchy и нажать клавишу F.

A.2. Клавиатурные комбинации

В Unity определены комбинации клавиш для быстрого доступа к важным функциям. Самыми важными являются клавиши W, E, R и T, активирующие инструменты преобразования Translate, Rotate и Scale (если вы забыли, какую функцию они выполняют, перечитайте главу 1), а также инструмент 2D Rect. Эти клавиши расположены в одном ряду, поэтому при работе на них обычно держат левую руку, в то время как правая манипулирует мышью.

Существуют и другие клавиатурные комбинации, которые могут оказаться полезными при работе с Unity. Они перечислены в табл. A.2.

Таблица A.2. Полезные клавиатурные комбинации

Клавиатурная комбинация	Назначение
W	Translate (перемещение выделенного объекта)
E	Rotate (поворот выделенного объекта)
R	Scale (изменение размера выделенного объекта)
T	Инструмент Rect tool (манипулирование двумерными объектами)
F	Фокусировка на выделенном объекте
V	Привязка к вершинам
Ctrl/Command+Shift+N	Создание объекта `GameObject`
Ctrl/Command+P	Воспроизведение игры
Ctrl/Command+R	Обновление проекта
Ctrl/ Command+1	Отображение в активном окне вкладки Scene
Ctrl/Command+2	Активация окна Game
Ctrl/Command+3	Переход на панель Inspector
Ctrl/Command+4	Переход на вкладку Hierarchy
Ctrl/Command+5	Переход на вкладку Project
Ctrl/Command+6	Переход на вкладку Animation

В Unity определены и другие клавиатурные комбинации, но они соответствуют достаточно сложным операциям.

Приложение Б. Внешние инструменты, используемые вместе с Unity

Для решения некоторых задач в процессе разработки игр в Unity приходится прибегать к внешним инструментам. Один из них обсуждался в главе 1; с технической точки зрения инструмент MonoDevelop представляет собой отдельное приложение, хотя и входит в один пакет с Unity. Аналогичным образом разработчики используют другие дополнительные инструменты для задач, которые невозможно решить средствами Unity.

Это не означает недостаточную функциональность Unity. Просто процесс разработки игр столь сложен и многогранен, что любой хорошо спроектированный инструмент с четким предназначением по определению имеет узкую специализацию. Приложение Unity является связующим элементом и движком для всех компонентов игры, обеспечивая их совместную работу. Соответственно, создание отдельных элементов может выполняться внешними инструментами. Рассмотрим несколько категорий программного обеспечения, которое может вам пригодиться.

Б.1. Инструменты программирования

Вы уже знакомы с наиболее интересным инструментом программирования для Unity — с приложением MonoDevelop. Но есть и другие программы, которые следует иметь в виду.

Б.1.1. Visual Studio

Как упоминалось в главе 1, Unity поставляется с приложением MonoDevelop, причем эта интегрированная среда разработки действует как на платформе Windows, так и на Mac. Однако при работе в операционных системах семейства Windows вы можете воспользоваться также средой разработки Visual Studio. Недавно корпорация Microsoft приобрела компанию SyntaxTree, занимавшуюся совершенствованием механизма интеграции Visual Studio: http://unityvs.com.

Б.1.2. Xcode

Инструмент Xcode представляет собой среду программирования от Apple (в частности, это IDE, но туда входят и SDK для платформ Apple). Хотя большая часть работы выполняется в Unity, при развертывании игр на платформе iOS для отладки и профилирования приложений требуется Xcode: https://developer.apple.com/xcode/.

Б.1.3. Android SDK

Если для развертывания в iOS необходимо приложение Xcode, для развертывания игр на платформе Android нужно скачать Android SDK. Но в отличие от сборки игр для iOS, внешние по отношению к Unity инструменты разработки не требуются — достаточно указать в Unity настройки, относящиеся к Android SDK: http://developer.android.com/sdk/index.html.

Б.1.4. SVN, Git или Mercurial

Любой крупный проект по разработке программного обеспечения включает в себя множество версий файлов кода, поэтому программисты разработали особый класс

программного обеспечения, который называется VCS (Version Control System — система контроля версий). Три наиболее популярные системы — это Subversion (также известна как SVN) (http://subversion.apache.org/), Git (http://git-scm.com/) и Mercurial (https://www.mercurial-scm.org). Если вы еще не пользуетесь системой контроля версий, рекомендую установить какую-нибудь из них. Папку проекта Unity заполняет временными файлами и рабочими настройками, но для системы управления версиями интерес представляют только папки Assets (убедитесь, что ваша система контроля версий воспринимает генерируемые Unity файлы метаданных) и Project Settings.

Б.2. Приложения для работы с трехмерной графикой

Приложение Unity умеет обрабатывать двумерную графику (этой теме посвящены главы 5 и 6), но изначально это был движок для трехмерных игр, до сих пор обладающий мощным функционалом для работы с 3D-графикой. Специализирующиеся на трехмерной графике художники пользуются по крайней мере одним из перечисленных в этом разделе инструментов.

Б.2.1. Maya

Программа Maya (https://www.autodesk.ru/products/maya/overview) представляет собой редактор трехмерной графики и анимации, изначально применявшийся в киноиндустрии. Ее функционал позволяет решить практически любую возникающую перед художником задачу, от создания прекрасной анимации до эффективных, готовых к использованию в играх моделей. Смоделированная в Maya трехмерная анимация (например, двигающиеся персонажи) допускает экспорт в Unity.

Б.2.2. 3ds Max

Другой широко распространенный редактор для работы с трехмерной графикой и анимацией — 3ds Max (https://www.autodesk.ru/products/3ds-max/overview) — предоставляет практически идентичную с Maya функциональность. Совпадает и основная схема работы. Приложение 3ds Max работает только в операционных системах семейства Windows (в то время как остальные инструменты, в том числе Maya, являются кроссплатформенными), но часто применяется в игровой индустрии.

Б.2.3. Blender

Не столь распространенное, как 3ds Max или Maya, приложение Blender (https://www.blender.org/) сравнимо с этими двумя редакторами. Ведь оно тоже позволяет решать практически все возникающие задачи 3D-моделирования и при этом относится к приложениям с открытым исходным кодом. Вы можете бесплатно скачать версию Blender для любой платформы. Так как это бесплатное приложение, предполагается, что в процессе чтения вы будете использовать именно его.

Б.2.4. SketchUp

Это простой в применении инструмент моделирования, особенно подходящий для создания архитектурных элементов. В отличие от остальных упоминавшихся в этом разделе приложений, SketchUp (https://www.sketchup.com/) подходит для решения далеко

не всех задач. Основной акцент в нем сделан на упрощении моделирования зданий и простых форм. При разработке игр им имеет смысл пользоваться на этапе создания геометрической модели сцены и для редактирования уровней.

Б.3. Редакторы двумерной графики

Двумерная графика является важным компонентом всех игр вне зависимости от того, отображается ли она напрямую в двумерных играх или в виде текстур на поверхности трехмерных моделей. В разработке игр применяется ряд инструментов для работы с двумерной графикой, список которых вы найдете в этом разделе.

Б.3.1. Photoshop

Графический редактор Photoshop (https://www.photoshop.com/) является самым распространенным из упоминаемых здесь приложений. Встроенные в него инструменты позволяют редактировать существующие изображения, накладывать фильтры и даже рисовать изображения с нуля. В Photoshop поддерживается множество форматов файлов, в том числе все форматы, используемые в Unity.

Б.3.2. GIMP

Эта аббревиатура происходит от GNU Image Manipulation Program и является названием наиболее известного редактора двумерной графики с открытым исходным кодом. Программа GIMP (https://www.gimp.org/) в плане функциональности и удобства применения уступает Photoshop, но это все равно полезный и к тому же бесплатный графический редактор.

Б.3.3. TexturePacker

Все перечисленные выше инструменты применяются не только в разработке игр, приложение TexturePacker используется только в этой области. Но оно отменно справляется с задачами, для которых было создано: со сборкой листов спрайтов двумерных игр. Если вы разрабатываете двумерную игру, рекомендую установить себе TexturePacker (https://www.codeandweb.com/texturepacker).

Б.3.4. Aseprite, Pyxel Edit

Пиксельная графика — один из самых узнаваемых стилей графики для двумерных игр. С технической точки зрения для создания пиксельной графики можно применять даже Photoshop, но существуют и специализированные инструменты, например Aseprite (https://www.aseprite.org/) и Pyxel Edit (https://www.pyxeledit.com/).

Б.4. Программы для работы со звуком

Существует невообразимое количество программ для работы со звуком, включая как звуковые редакторы (которые работают с исходными звуковыми файлами), так и секвенсоры (создающие музыку из последовательности нот). Чтобы дать вам представление о доступных возможностях, рассмотрим два основных инструмента редактирования звука (из не вошедших в список я хотел бы отметить Logic, Ableton и Reason).

Б.4.1. Pro Tools

Это приложение для работы со звуком (http://www.avid.com/pro-tools) может похвастаться множеством полезных функций и рассматривается многочисленными производителями музыки и звукооператорами как индустриальный стандарт. Оно часто используется для решения различных профессиональных задач, в том числе при разработке игр.

Б.4.2. Audacity

Не относящаяся к числу профессиональных инструментов для работы со звуком, программа Audacity (https://sourceforge.net/projects/audacity/) представляет собой отличный редактор для решения не слишком сложных задач, например подготовки коротких аудиофайлов со звуковыми эффектами для игры. Эту программу часто выбирают те, кто предпочитает программы с открытым исходным кодом.

Приложение В. Моделирование скамейки в программе Blender

Разрабатывая уровни в главах 2 и 4, мы добавляли на них стены и пол. А как быть с более детализированными объектами? Представьте, что мы хотим обставить комнаты мебелью. Для решения этой задачи потребуется внешний редактор трехмерной графики. В начале главы 4 мы говорили, что трехмерные модели в играх представляют собой сеточные объекты (то есть трехмерные фигуры). Давайте на примере простой скамейки, показанной на рис. В.1, рассмотрим принцип создания таких объектов.

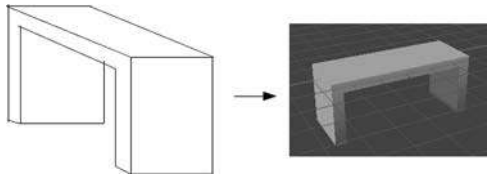


Рис. В.1. Схема простой скамьи, которую вы собираетесь моделировать

В приложении Б перечислен ряд инструментов для работы с трехмерной графикой. Для этого упражнения воспользуемся программой с открытым исходным кодом Blender. Смоделированный в ней сеточный объект будет экспортирован в Unity в виде графического ресурса.

СОВЕТ Моделирование — это большая сложная тема, поэтому мы затронем только необходимые для создания скамейки функции. Если вы захотите более подробно познакомиться с процессом моделирования объектов, воспользуйтесь соответствующей литературой и онлайновыми справочными материалами (для начала рекомендую сайт www.blender.org).

ВНИМАНИЕ У меня установлена версия Blender 2.67, соответственно, именно к ней относятся все объяснения и снимки экрана. Версии программы Blender выходят часто, поэтому у вас расположение кнопок или имена команд могут слегка отличаться от описанного.

В.1. Создание сеточной геометрии

Запустите Blender; начальный экран с кубом посреди сцены показан на рис. В.2. Управление камерой осуществляется с помощью средней кнопки мыши: перетаскивание при нажатой средней кнопке ведет к повороту камеры, при нажатой клавише Shift — к панорамированию сцены, а при нажатой клавише Ctrl — к масштабированию.

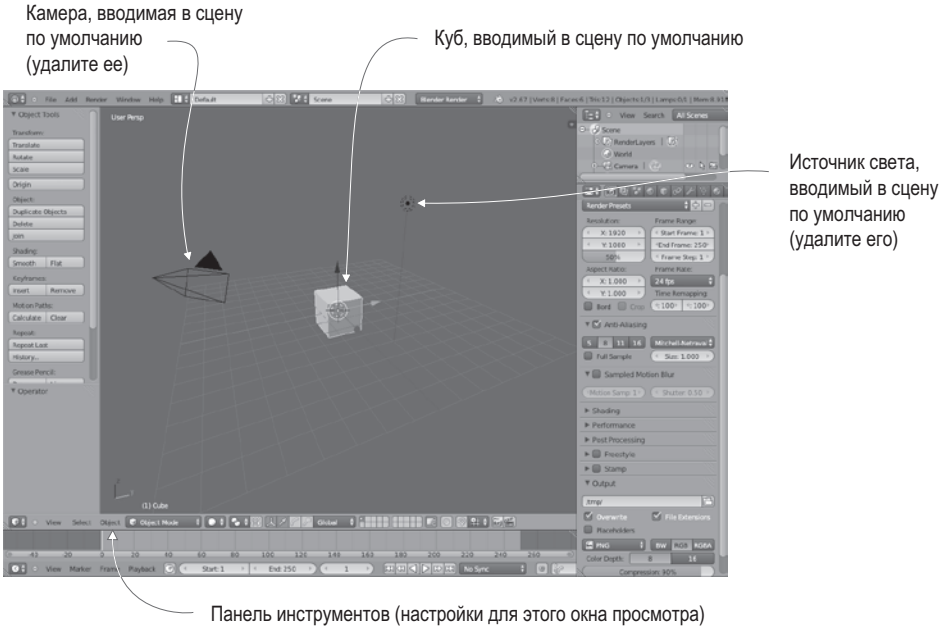


Рис. В.2. Начальный экран в программе Blender

Работа с программой Blender начинается в режиме Object mode, что, как несложно догадаться, означает манипуляцию объектами в целом путем перемещения их по сцене. Для доступа к редактированию формы объекта нужно перейти в режим Edit mode — для этого используется меню, показанное на рис. В.3 (нужный пункт меню появляется

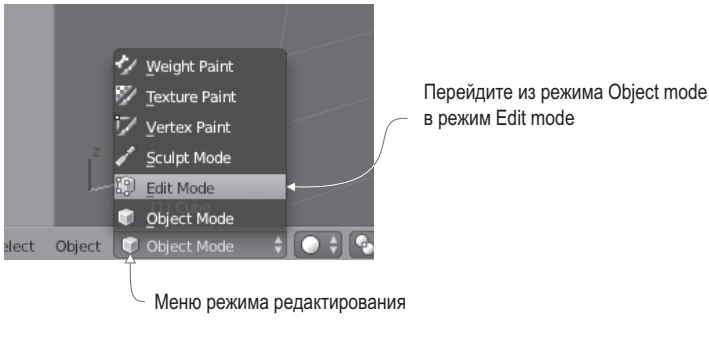


Рис. В.3. Меню переключения между режимами

только при наличии в сцене выделенного объекта, впрочем, сразу после загрузки приложения Blender объект выделятся автоматически). Кроме того, при первом переключении в режим редактирования вы попадаете в режим выделения вершин. Показанные на рис. В.4 кнопки позволяют переключаться между режимами выделения вершин (Vertex), ребер (Edge) и граней (Face). Именно таким способом осуществляется выделение различных элементов сетки.

ОПРЕДЕЛЕНИЕ *Элементами сетки* (mesh elements) называются определяющие ее геометрию вершины, ребра и грани, то есть отдельные угловые точки, соединяющие эти точки линии и образованные соединенными линиями фигуры.

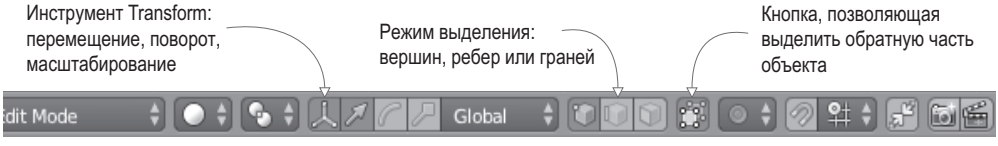


Рис. В.4. Элементы управления в нижней части окна просмотра

КЛАВИАТУРНЫЕ КОМБИНАЦИИ И РАБОТА С МЫШЬЮ В BLENDER

На рис. В.4 мы видим в числе прочего различные инструменты преобразования. Как и в Unity, преобразования сводятся к перемещениям, поворотам и масштабированию. Первая кнопка включает и отключает габаритный контейнер преобразования (Transform Gizmo), обозначаемый в сцене стрелками; я рекомендую оставить его включенным, потому что без него работать с инструментами преобразования можно только с помощью клавиатурных комбинаций, которые в Blender зачастую крайне неочевидны, как и приемы работы с мышью.

Если управление камерой при помощи средней кнопки мыши интуитивно понятно, то за выделение элементов сцены отвечает правая кнопка мыши (в то время, как в большинстве приложений эта операция выполняется левой кнопкой). Что еще более странно, для выделения куба требуется нажать клавишу B, а затем щелкнуть на объекте левой кнопкой мыши и перетащить указатель. Чтобы добавить элемент в выделенный набор (вместо замены одного выделенного элемента другим), удерживайте во время щелчка на элементе клавишу Shift, а для снятия выделения достаточно нажать клавишу A.

Теперь, когда вы знакомы с основными элементами управления в приложении Blender, рассмотрим функции редактирования модели. Для начала превратим куб в длинную тонкую доску. Выделите все вершины модели (в том числе и вершины задней грани) и активируйте инструмент Scale. Для уменьшения вертикальных размеров перетащите мышью синюю стрелку, которая соответствует оси *Z*, а затем таким же способом растяните объект вдоль оси *Y*, чтобы придать ему форму, показанную на рис. В.5.

Переключитесь в режим выделения граней (кнопкой, показанной на рис. В.4) и выделите две самые маленькие грани доски. Далее в меню Mesh, расположенном в нижней части окна, выберите команду Extrude Individual, как показано на рис. В.6. В результате при перемещении указателя мыши на концах доски появятся дополнительные секции; слегка вытяните их и щелкните левой кнопкой мыши для завершения процесса. Эти дополнительные секции увеличивают скамейку на ширину ее ножек, предоставляя дополнительную геометрию.

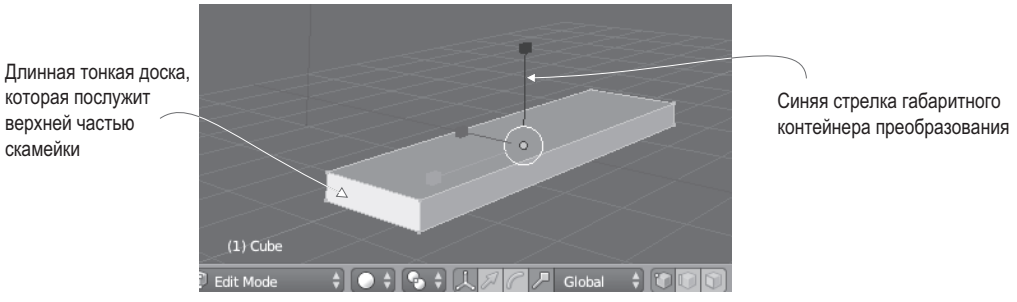


Рис. В.5. Сетка, после масштабирования превратившаяся в длинную и тонкую доску

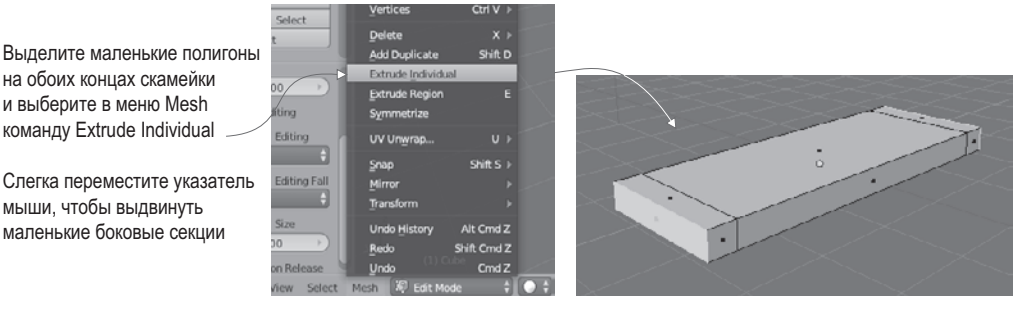


Рис. В.6. Создание дополнительных фрагментов объекта

ОПРЕДЕЛЕНИЕ Процедура *выдавливания* (extrude) создает новую геометрию с сечением в форме выделенных граней. Существует две команды, определяющие действия в случае набора выделенных элементов: команда Extrude Individual выдавливает каждый элемент как отдельный фрагмент, в то время как команда Extrude Region выдавливает все элементы как одно целое.

Посмотрите на нижнюю часть нашей доски и выделите два узких элемента на ее концах. Еще раз воспользуйтесь командой Extrude Individual, чтобы сформировать ножки скамейки, как показано на рис. В.7.

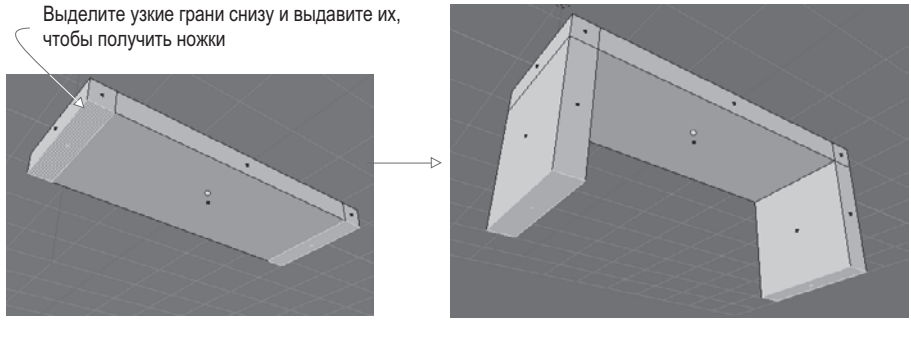


Рис. В.7. Создание ножек

Фигура готова! Но перед экспортом модели в Unity нужно добавить к ней материал.

B.2. Назначение материала

На поверхности трехмерных моделей можно отображать двумерные изображения, называемые текстурами. В случае больших плоских поверхностей процесс назначения материала очевиден; достаточно растянуть его по поверхности. А что делать с моделями более сложной формы? Здесь на помощь приходит такое понятие, как *текстурные координаты*.

Эти координаты определяют положение различных точек текстуры относительно сетки. Фактически с их помощью сеточные элементы проецируются на области текстуры. Представьте себе оберточную бумагу, как показано на рис. B.8; трехмерная модель — это заворачиваемый в бумагу ящик, текстура — бумага, а координаты показывают, какая сторона бумаги придется на конкретную грань ящика. Они задают на двумерной картинке точки и фигуры; эти фигуры соответствуют полигонам сетки, благодаря чему части изображения появляются на фрагментах сетки.

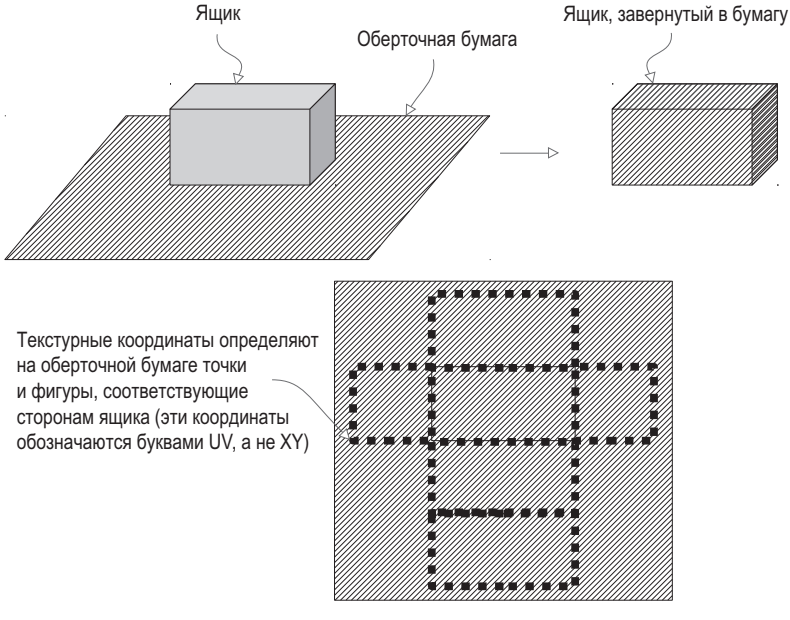


Рис. B.8. Оберточная бумага хорошо иллюстрирует принцип работы текстурных координат

СОВЕТ Еще текстурные координаты называют *UV-координатами*. Это название появилось из-за того, что текстурные координаты обозначаются буквами *U* и *V*, в то время как для обозначения координат трехмерной модели используются буквы *X*, *Y* и *Z*.

Процесс совмещения фрагментов одной сущности с фрагментами другой называется *проецированием* (mapping) — соответственно, термин *текстурное проецирование* (texture mapping) обозначает процесс создания текстурных координат. Из аналогии с оберточной бумагой родилось еще одно название этого процесса — *распаковывание* (unwrapping). Есть также термины, образованные смешением обоих понятий, например

UV-распаковывание (UV unwrapping); постарайтесь не запутаться в многочисленных, по существу синонимичных, понятиях, связанных с текстурным проецированием.

Сам по себе процесс текстурного проецирования достаточно сложен, но, к счастью, приложение Blender оснащено инструментами, превращающими его в рутинную процедуру. Первым делом нужно определить швы; если снова представить процесс упаковки ящика в бумагу (точнее, лучше представить обратный процесс — распаковку), становится понятно, что далеко не каждая часть трехмерной фигуры при превращении ее в плоскость остается бесшовной. И там, где фрагменты этой фигуры расходятся в стороны, появляются швы. Приложение Blender позволяет выделять ребра и объявлять их швами.

Переключитесь в режим выделения ребер (см. кнопки на рис. В.4) и выделите ребра с внешней стороны нижней части скамейки. Далее выберите в меню Mesh команду Edges и затем — Mark Seam, как показано на рис. В.9. После этого Blender отделит нижнюю часть скамейки с целью проецирования текстур. Проделайте ту же операцию для сторон скамейки, но полностью их не разделяйте. Вместо этого превратите в швы только ребра, идущие вверх по ножкам; в этом случае стороны сохранят соединение со скамейкой, но раскроются, как крылья.

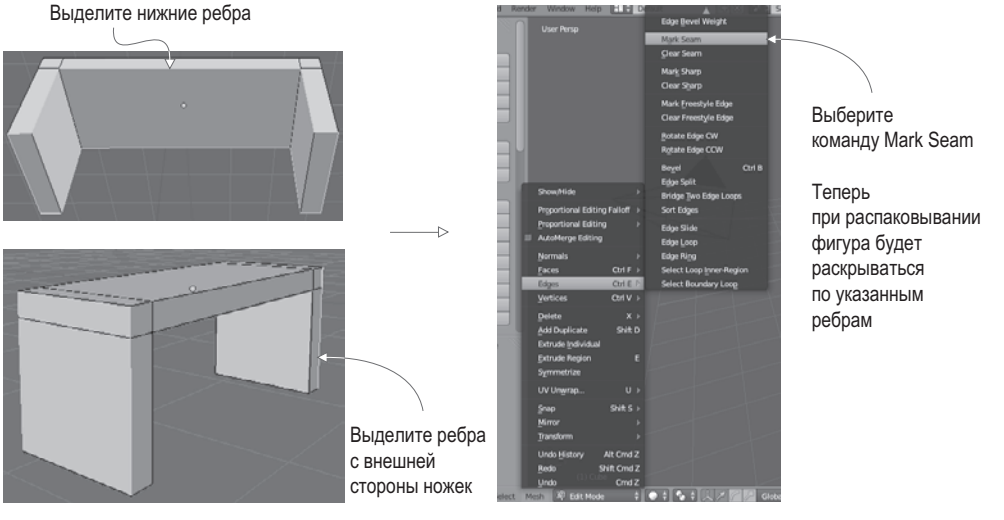


Рис. В.9. Ребра вдоль нижней части скамейки и вдоль ножек превращаются в швы

Пометив все швы, воспользуйтесь командой Texture Unwrap. Для начала выделите сетку целиком (не забудьте невидимую для нас обратную сторону объекта). Затем выберите в меню Mesh команду UV Unwrap > Unwrap, чтобы сгенерировать текстурные координаты. Но пока вы их не увидите, так как по умолчанию Blender демонстрирует трехмерное отображение сцены. Для просмотра текстурных координат нужно переключиться в UV-редактор, воспользовавшись крайним слева меню панели инструментов, которое называется Viewports (нас интересует не слово View, а маленький значок, показанный на рис. В.10).

После этого вы увидите полигоны скамейки, разложенные на плоскости, отделенные друг от друга и раскрытые по указанным швам. Для шейдера текстуры эти

UV-координаты должны быть видимы в программе редактирования изображений. Снова обратимся к рис. В.10: выберите в меню UVs команду Export UV Layout и сохраните изображение под именем bench.png (именно это имя будет использоваться для импорта в Unity).

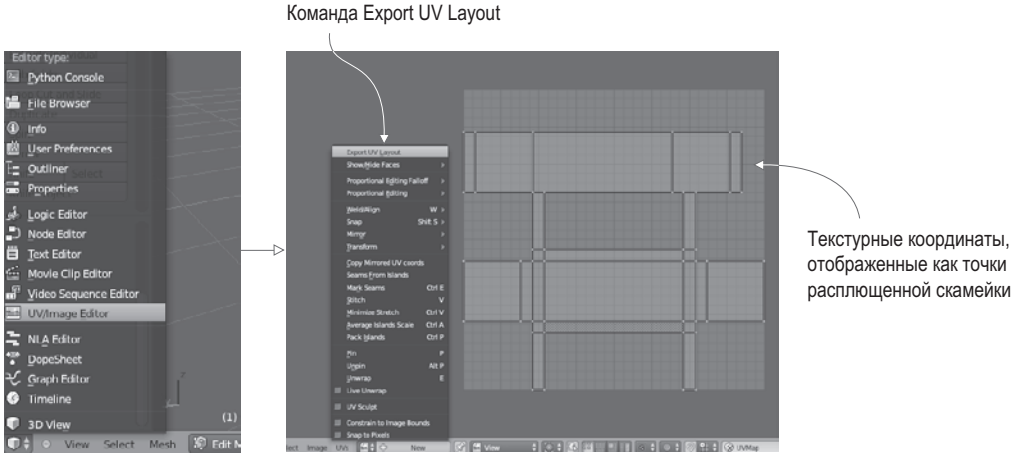


Рис. В.10. Чтобы увидеть текстурные координаты, смените 3D View на UV Editor

Откройте это изображение в графическом редакторе и раскрасьте части текстуры в разные цвета. Благодаря UV-координатам эти цвета окажутся на соответствующих гранях. Например, на рис. В.11 темно-синим цветом окрашена нижняя часть скамейки, а красным — ее боковые стороны. Теперь изображение можно вернуть в программу Blender и использовать в качестве текстуры для модели. Для этого выберите в меню Image команду Open Image.

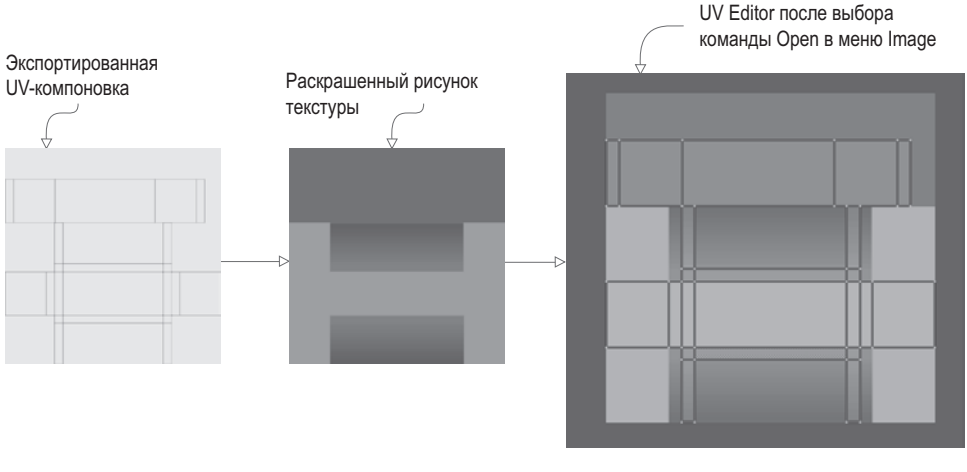


Рис. В.11. Раскрасьте экспортированное UV-изображение и верните текстуру в Blender

Теперь можно вернуться к трехмерному представлению (с помощью того же самого меню, которым мы пользовались для переключения в редактор UV-координат). Текстура на модели все еще не видна, но это легко исправить. Достаточно удалить присутствующий в сцене по умолчанию источник света и включить отображение текстур в окне проекции, как показано на рис. В.12.

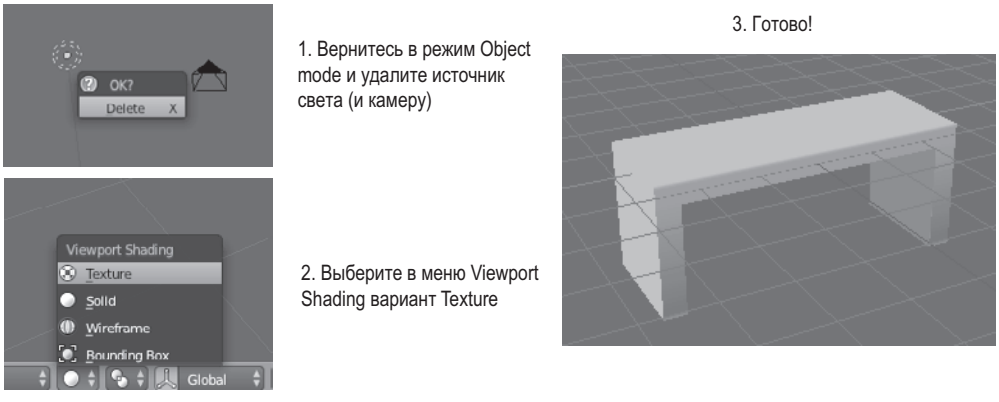


Рис. В.12. Отображение текстуры на поверхности модели

Чтобы удалить источник света, первым делом вернитесь в режим работы с объектами и выделите его (используйте меню, при помощи которого вы переходили в режим редактирования). Нажмите клавишу X для удаления выделенного объекта. Наконец, выберите в меню Viewport Shading команду Texture. В окне проекции появится готовая скамейка с назначенной ей текстурой!

Сохраните модель. По умолчанию Blender сохраняет файлы с расширением .blend, то есть в собственном формате. Воспользуйтесь именно этим форматом, чтобы корректно сохранить все функциональные особенности модели Blender. Позднее потребуется экспортировать модель в файл другого формата, предназначенный для импорта в Unity. Обратите внимание, что изображение текстуры вместе с моделью не сохраняется; сохраняется ссылка на него, но вам по-прежнему требуется сам графический файл.

Джозеф Хокинг

Unity в действии. Мультиплатформенная разработка на C#

2-е международное издание

Перевела с английского *И. Рузмайкина*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Бульченко*
Художественный редактор	*С. Маликова*
Корректоры	*Н. Викторова, И. Тимофеева*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 26.09.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 1500. 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