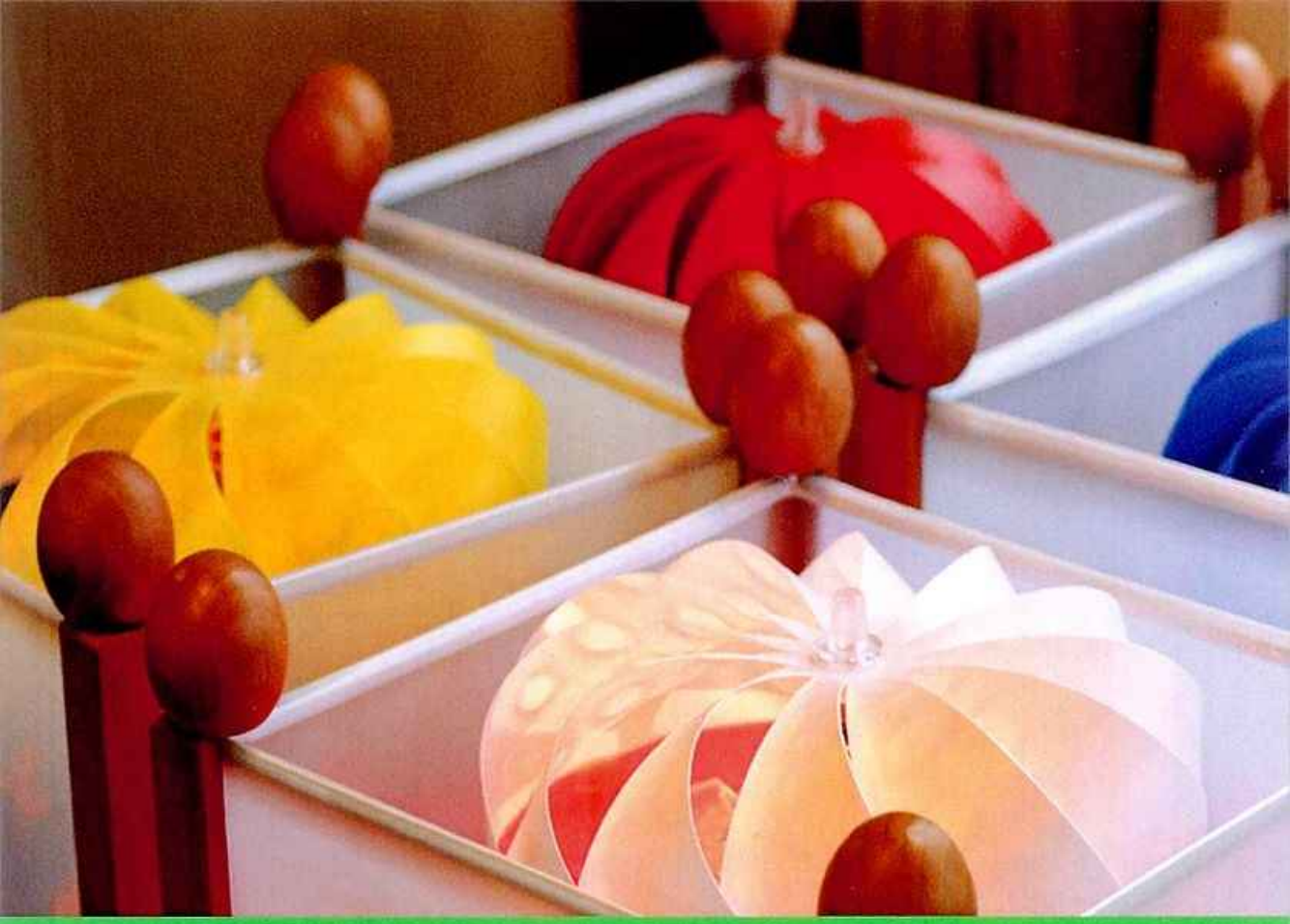

Шейдеры и эффекты в Unity

Книга рецептов

Как с помощью шейдеров и постэффектов добиться потрясающей картинки в проектах на Unity

Кенни Ламмерс

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

@mail.ru group
unity
SCREAM SCHOOL
ШКОЛА КОМПЬЮТЕРНОЙ ГРАФИКИ

Кенни Ламмерс

Шейдеры и эффекты в Unity

Книга рецептов

Как с помощью шейдеров и постэффектов
добиться потрясающей картинки в проектах на Unity

Unity Shaders and Effects Cookbook

Discover how to make your Unity projects look stunning with Shaders and screen effects

Kenny Lammers

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Шейдеры и эффекты в Unity
Книга рецептов

Как с помощью шейдеров и постэффектов
добиться потрясающей картинки
в проектах на Unity

Кенни Ламмерс

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2016

УДК 004.4'2Unity3D
ББК 32.972
Л21

Книга выпущена при поддержке Mail.Ru Group.

Л21 Кенни Ламмерс

Шейдеры и эффекты в Unity. Книга рецептов / пер. с англ. Шапочкин Е. А., под редакцией Симонова В. В. – М.: ДМК Пресс, 2016. – 274 с.: ил.

В книге раскрываются секреты разработки шейдеров в Unity – самом популярном в мире мультиплатформенном инструменте для разработки двух- и трёхмерных игр и приложений. Описываются базовые модели освещения, создание эффектов с помощью текстур, анимация моделей в реальном времени, настройка шейдеров для мобильных устройств, а также использование постэффектов в гейм-плее.

Издание предназначено для Unity-разработчиков, стремящихся использовать максимум возможностей платформы для создания своих собственных шедевров!

УДК 004.4'2Unity3D
ББК 32.972

ОГЛАВЛЕНИЕ

ОБ АВТОРЕ

Кенни Ламмерс работает в индустрии игр уже 13 лет. Он работал на такие компании, как Microsoft, Activision и Surreal Software. В данный момент он работает с двумя компаниями. В первой – Creative TD – он занимается консультированием по Unity3D и созданием материалов для таких компаний, как IGT, Microsoft, Janus Research и Allegorithmic. Второй компанией – Ozone Interactive – он руководит совместно со своим бизнес-партнёром Ноа Каарбо (Noah Kaarbo). Ozone Interactive специализируется на создании интерактивных приложений и высококачественных дизайнов с использованием Unity3D для таких компаний как Amazon, E-line Media, Microsoft и Sucker Punch games. Во время его работы в индустрии игр он создавал персонажей в Zbrush и Maya, писал шейдеры и постэффекты, а так же разрабатывал игры полностью в Unity3D, используя C#. В данный момент он работает над несколькими играми и разрабатывает набор инструментов для облегчения процесса создания игр.

Я бы хотел высказать благодарность всем её заслуживающим, но тогда на это ушла бы целая глава. Перво-наперво я определённо хочу поблагодарить мою маму за то, что она всегда говорила мне, что моя работа должна вести к моей мечте, и за то, что она всегда была мне поддержкой. Я бы хотел поблагодарить моего партнёра по бизнесу Ноа Каарбо за то, что был мне другом и поддержкой на протяжении написания этой книги. Я хочу поблагодарить всех, с кем мне довелось работать, но, что более важно, – я хочу поблагодарить тех, кто побуждал меня всегда совершенствовать мои навыки и тем самым открыл для меня мир этой индустрии. Эти люди – Бэн Каммерано (Ben Cammerano) из MGS, Пол Амер (Paul Amer) из MGS, Филлипо Костанцо (Fillipo Costanzo) из 5D Institute, Алессандро Тенто (Alessandro Tento) из Lakshya, Джеймс Роджерс (James Rogers) из MGS и Тони Гарсия (Tony Garcia) из Unity Technologies. Я бы не добился того, чего добился, без кого-либо из этих людей, и они заслуживают моего самого сильного почтения!

О РЕЦЕНЗЕНТАХ

Винсент Лим (Vincent Lim) закончил The One Academy по специальности цифровая анимация и разработка игр. Сразу же после завершения учёбы он присоединился к Big Ant Studio, где он преобрёл массу бесценного опыта в игровой индустрии. Проведя несколько лет в компании, Винсент многому научился, начиная от низкополигонального моделирования и заканчивая укладкой текстур для создания ландшафта, а также – немного программированию и написанию MEL-скриптов. Работая над разнообразными задачами в Big Ant Studio, Винсент накопил знания об игровых движках, о тонкостях работы разных шейдеров и о пайплайне разработки игр. В компании он занимался оптимизацией пайплайна разработки, создавая с помощью MEL-скриптов инструменты для художников, которые упрощали те или иные задачи на пути 3D-модели, – от заготовки в 3D-редакторе до модели в игре. Получив достаточный базис знаний в Big Ant Studio, Винсент продолжил изучение игровых механик и движков. Это привело его к Unity, навыки работы с которым он продолжает совершенствовать и по сей день.

Кристиан 'XeviaN' Менengхини (Christian 'XeviaN' Meneghini) в молодости был обладателем и поклонником Sinclair ZX Spectrum. Он начал своё знакомство с миром игровой индустрии со спрайтов, жёстко закодированных с помощью Бейсика и Ассемблера. По прошествии времени ему довелось работать с такими замечательными технологиями, как C64, культовым Amiga и всей линейкой процессоров PC, при этом работая с видеокартами от Hercules и CGA от первых 3D-ускорителей до современных. При его специализации на программировании графики и оптимизации производительности, ему очень нравилось заниматься рендерингом и принимать участие в демосцене. Кристиан в своё свободное время сочиняет музыку.

После нескольких лет работы по ночам вместе с друзьями и коллегами, изучения технической документации, написания движков и работы на другие компании Кристиан в 2011 году совместно со своими друзьями Марко Ди Тимотео (Marco Di Timoteo) и Лукой Марчетти (Luca Marchetti) основал небольшую студию, которую они назвали STUDIO EVIL. Их первым продуктом стала игра Syder Arcade – ретро-шутер с 3D-графикой, рассчитанный на PC и MAC, а позднее – портированный под iOS, Android и OUYA.

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Мы очень рады выходу первой переводной книги по Unity, тем более, что это совпало с открытием нашего офиса в России. Мы наблюдаем значительный рост количества российских разработчиков и издателей игр, которые используют Unity и работаем над тем, чтобы содействовать им в постижении новых горизонтов возможностей в разработке кросс-платформенных игр. Всегда хочется добиться максимального качества картинки при разумной производительности. Эта задача актуальна, пожалуй, для проектов всех масштабов и задач. Мы верим, что с помощью данной книги ваша жизнь, работа и опыт использования Unity станут лучше и красочнее.

Искренне ваши,
Unity Technologies в России
russia@unity3d.com
Роман Менякин,
Наталья Свиридова,
Йоана Кодите,
Олег Придюк

ПРЕДИСЛОВИЕ

Мы приветствуем вас на страницах книги «Шейдеры и эффекты в Unity». Эта книга поможет вам освоиться с созданием шейдеров и постэффектов в Unity3D. Вы начнёте ваше путешествие по страницам этой книги с самого начала, с создания наиболее базовых шейдеров и получения представления о структуре шейдерного кода. Эти базовые знания пригодятся вам в последующих главах, в которых вы будете создавать шейдеры, имитирующие людскую кожу, шейдеры, обрабатывающие динамические отражения, а также, создавать постэффекты, такие как эффект ночного видения.

К концу каждой главы, у вас сформируется новый набор навыков, с помощью которых, вы сможете повысить качество ваших шейдеров, а также, сделать более эффективным процесс их написания. Эти главы были сформированные таким образом, что вы можете перейти к любой секции и сразу овладеть любым требуемым навыком, развив его от уровня новичка до эксперта. А, если процесс написания шейдеров для вас в новинку, вы можете читать главы последовательно, одну за другой, постепенно приобретая необходимые знания. В любом случае, вы познакомитесь с приёмами, которые используются в большинстве современных игр.

После того, как вы закончите работу с этой книгой, у вас будет набор шейдеров, который вы сможете использовать в ваших играх, сделанных с помощью Unity3D, а также, у вас появится понимание того, как и что нужно будет добавлять к ним, чтобы получать новые эффекты и удовлетворять запросам производительности. Так что, давайте, перейдём к делу.

Что рассматривается в этой книге

Глава 1 «Диффузный шейдинг» содержит азы написания шейдеров, в этой главе объясняется структура шейдеров в Unity3D. Далее в этой главе данные знания применяются для создания диффузной модели

освещения по умолчанию, а также приводятся тонкости и приёмы, используемые в игровой индустрии для создания собственных моделей освещения.

Глава 2 «Создание эффектов с помощью текстур» описывает использование текстур для создания различных эффектов. В этой главе вы узнаете, как можно анимировать спрайт-листы с помощью шейдера, а также, как использовать различные каналы текстуры для повышения эффективности шейдеров. К концу этой главы, ваши навыки использования текстур будут достаточны для создания ваших собственных эффектов.

Глава 3 «Пусть ваши игры засияют отражённым светом» рассказывает всё, что вам может потребоваться, о создании наиболее распространённых типов бликового отражения – Blinn и Phong. Вы узнаете, как можно применить эти шейдерные эффекты для создания маскированного отражённого освещения, металлического отражения, а так же, узнаете о создании анизотропного отражённого освещения. К концу этой главы, вы будете иметь достаточно знаний, для того, чтобы реализовывать свои собственные specular-эффекты.

Глава 4 «Добавим отражения в ваш мир» рассказывает о применении одного из наиболее распространённых эффектов в современных играх, с помощью которого вы сможете учитывать в ваших шейдерах приёмы создания отражений. Эта глава научит вас всему – начиная с азов организации отражения в шейдерах Unity3D и заканчивая созданием вашей собственной системы динамического отражения с помощью C#.

Глава 5 «Модели освещения» рассказывает о более сложных шейдерах. Вы узнаете, как создавать свои собственные модели освещения, использование которых позволит вам сымитировать произвольный тип поверхности. Каждый рецепт демонстрирует применение различных техник для достижения различных задач, каждая из которых приведёт к развитию ваших навыков написания шейдеров. К концу этой главы, вы создадите ваш собственный шейдер кожи, узнаете об освещении Lit Sphere, и напишете свой шейдер автомобильной краски.

Глава 6 «Прозрачность» продемонстрирует то, что на некотором этапе производства игр вам неизбежно потребуется использовать прозрачность. Практически любая игра, так или иначе, задействует прозрачность для таких объектов, как элементы пользовательского интерфейса, опавшая листва, объекты-трафареты, и т. д. В этой главе вы узнаете, как работать с прозрачностью в Unity3D, и как можно уладить трудные моменты, возникающие при этом.

Глава 7 «Волшебные возможности вершин» рассказывает о том, как можно получить доступ к информации, хранящейся в вершинах 3D сетки объекта. Вы научитесь работать с этими данными использовать их в шейдере, для создания таких эффектов, как блендинг текстур и анимация.

Глава 8 «Настройка шейдеров для мобильных приложений» посвящена способам оптимизации шейдеров в Unity, с помощью встроенных типов и макросов. Так как, эта задача становится особенно важной при работе шейдеров для мобильных платформ.

Глава 9 «Делаем наш шейдерный мир модульным с помощью CgIncludes» продемонстрирует вам, почему нужно повторно использовать уже написанный код, для того, чтобы улучшить ваш навык написания шейдеров. Эта глава покажет, как вы можете создать ваши собственные файлы CgInclude, для хранения и повторного использования повторяющихся блоков кода.

Глава 10 «Создание экранных эффектов в Unity с помощью рендер-текстур» начинается с рассмотрения того, как в современных играх используются экранные эффекты (так же, называемый постэффектами) для изменения итогового отрендеренного изображения игры. Вы узнаете, как вы можете создать ваши собственные экранные эффекты, а также, как можно осуществлять корректировку цвета и наложения текстур для создания различных визуальных эффектов в вашей игре.

Глава 11 «Гейм-плей и экранные эффекты» углубляет ваши знания об экранных эффектах, и показывает, как вы можете создать эффекты, усиливающие атмосферность моментов в вашей игре. Вы научитесь создавать эффекты старого фильма и ночного видения.

Что потребуется при чтении этой книги

Для того чтобы вы смогли выполнить рецепты, приводимые в этой книге, вам понадобится следующее необходимое и опциональное программное обеспечение:

- Unity3D (для глав 10 и 11 вам потребуется Unity3D Pro);
- 3D-приложение, такое как Maya, Max или Blender (опционально);
- приложение для работы с 2D-графикой, такое как Photoshop или Gimp (опционально).

Для кого эта книга

Эта книга предназначена для программистов, работающих с Unity3D, для новичков и продвинутых. Лучше всего, чтобы у вас уже был опыт работы с C# или JavaScript, и вы бы уже владели базовыми навыками работы с Unity. Мы советуем вам взглянуть на руководство для новичков в Unity 3.x по разработке игр от Packt Publishing (http://www.packtpub.com/unity-3-x-game-development-by-example-beginners-guide/book), для того, чтобы получить достаточные базовые навыки по работе с азами Unity3D.

Условные соглашения

В этой книге вы встретите несколько стилей оформления текста, которыми форматируется разная по смыслу информация. Сейчас мы приведём примеры этих стилей и объясним их смысл.

Ключевые слова кода будут приводиться в тексте следующим образом: «введите следующий код в блок `Properties` вашего шейдера».

Блоки кода будут оформляться таким образом:

```
void surf (Input IN, inout SurfaceOutput o)
{
     float4 c;
     c =  pow((_EmissiveColor + _AmbientColor), _MySliderValue);

     o.Albedo = c.rgb;
     o.Alpha = c.a;
}
```

Новые термины и **ключевые слова** будут приводится полужирным шрифтом. Слова, которые вы увидите на экране, в меню или, например, в диалоговых окнах, будут приводиться по тексту следующим образом: «таким образом создаётся палитра кубмапы на закладке **Инспектора компонентов**, пользователь может перетаскивать кубмапу на шейдер».

Так будут оформляться предупреждения и важные примечания.

Так будут оформляться подсказки и приёмы.

Обратная связь с читателями

Мы всегда приветствуем обратную связь с нашими читателями. Сообщите нам, что вы думаете об этой книге – что вам в ней понравилось или же не понравилось. Обратная связь поможет нам преподносить именно тот материал, который вам нужен.

Чтобы связаться с нами, просто пошлите письмо по адресу feedback@packtpub.com, а в теме письма укажите название книги.

Если есть предметная область, в которой вы имеете обширный опыт, и вы заинтересованы в написании книги или в другом содействии, то посетите www.packtpub.com/authors.

Поддержка клиентов

Теперь, когда вы стали покупателем книги Packt, перед вами открываются следующие возможности.

Скачивание программного кода примеров

Вы можете скачать файлы программного кода примеров для всех книг Packt, которые вы заказали с вашего аккаунта на страничке http://www.packtpub.com. Если вы заказали эту книгу откуда-то ещё, то вы можете посетить страницу http://www.packtpub.com/support и зарегистрироваться, чтобы мы выслали вам эти файлы электронным письмом.

Скачивание цветных изображений этой книги

Также мы предоставим вам PDF-файл, содержащий цветные изображения скриншотов/диаграмм, использованных в данной книге. Цветные изображения помогут вам лучше понять содержимое глав. Вы можете скачать этот файл с http://www.packtpub.com/sites/default/files/downloads/5084OT_Images.pdf.

Ошибки и опечатки

Несмотря на то что мы предприняли всё возможное для обеспечения точности содержимого наших книг, ошибки иногда всё-таки встречаются. Если вы найдёте ошибку в какой-либо из наших книг – может быть, ошибку в тексте или в коде, – мы будем рады, если вы сообщите

нам об этом. Таким образом вы сможете облегчить жизнь другим читателям и помочь нам улучшить последующие издания книги. Если вы найдёте какую-либо ошибку, то, пожалуйста, сообщите о ней на странице http://www.packtpub.com/submit-errata – выберите там вашу книгу, кликните на ссылку отправки ошибки и введите её описание. Как только присланная вами ошибка будет подтверждена, она будет загружена на наш веб-сайт или добавлена к списку существующих ошибок в секции ошибок для выбранной книги. Найденные ошибки можно посмотреть, выбрав название книги на странице http://www.packtpub.com/support.

Нарушение авторских прав

В Интернете пиратство защищённого авторскими правами материала является насущной проблемой для всех форм материалов. Мы в Packt относимся к защите наших авторских прав и лицензий очень серьёзно. Если вы найдёте в Интернете в любой форме нелегальные копии наших работ, то, пожалуйста, незамедлительно сообщите нам ссылку или название веб-сайта, чтобы мы смогли принять меры.

Пожалуйста, свяжитесь с нами по адресу copyright@packtpub.com и укажите ссылку на предположительно пиратские материалы. Мы ценим вашу помощь в защите прав наших авторов и нашей возможности предоставлять вам ценную информацию.

Вопросы

Вы можете связаться с нами по адресу questions@packtpub.com, если у вас имеются вопросы касательно любого аспекта этой книги, и мы постараемся на них ответить.

ГЛАВА 1
Диффузный шейдинг

В этой главе будут рассмотрены некоторые наиболее распространённые приёмы, используемые сегодня в игровой индустрии при разработке шейдеров. Вы узнаете о том, как:

- создать простой поверхностный шейдер;
- добавить свойства поверхностному шейдеру;
- использовать свойства в поверхностном шейдере;
- сделать собственную модель диффузного освещения;
- написать модель освещения Half Lambert;
- использовать текстуру для контроля над диффузным шейдингом;
- имитировать эффект BRDF с помощью 2D-текстуры.

Введение

В основе любого хорошего шейдера всегда лежит модель освещения, а точнее, его диффузного (рассеивающего) компонента. Так что имеет смысл начинать написание шейдера именно с него.

Ранее в компьютерной графике диффузный шейдинг делали с помощью так называемой **неперепрограммируемой** (fixed function) **модели освещения**. Она предоставляла графическим программистам единственную модель освещения, которую они могли настраивать с помощью набора параметров и текстур. Сейчас же, с появлением шейдеров и языка Cg, мы получили больше возможностей контролировать освещение. Тем более в Unity с его поверхностными шейдерами.

Диффузный компонент шейдера описывает, как свет отражается от поверхности во всех направлениях. Возможно, вам покажется, что это описание очень похоже на принцип работы зеркала, но в действительности это не так. Зеркальная поверхность отражает изображение объектов окружающей среды, в то время как диффузное освещение рассеивает во все направления суммарный свет, испускаемый источ-

никами света, такими как, например, солнце. Отражения мы рассмотрим в последующих главах, а на данный момент, нам просто нужно знать, чем они отличаются от диффузного освещения.

Чтобы создать базовую модель диффузного освещения, нам нужно будет написать шейдер, в который будут передаваться цвет испускаемого излучения, цвет фонового освещения и суммарный свет от всех источников. Следующие рецепты покажут, как сделать законченную модель диффузного освещения, а также продемонстрируют некоторые известные приёмы работы с текстурами, которые пригодятся при создании более сложных моделей.

К концу этой главы вы научитесь создавать простые шейдеры, которые выполняют основные функции шейдинга. Вооружившись этими знаниями, вы сможете создать практически любой поверхностный шейдер.

Создаём простой поверхностный шейдер

В то время как мы продвигаемся дальше по рецептам этой книги, важно, чтобы вы знали, как настроить рабочую среду в Unity так, чтобы можно было работать эффективно и без каких-либо неудобств. Если вы уже знакомы с созданием шейдеров и настройкой материалов в Unity 4, то можете пропустить этот рецепт. Мы приводим его в этой книге для того, чтобы те, кто только начинает работу с поверхностным шейдингом в Unity 4, могли работать с остальными рецептами книги.

Подготовка

Для того чтобы начать работать с этим рецептом, вам потребуется запустить Unity 4 и создать новый проект. Вы также можете воспользоваться прилагаемым к книге проектом, просто добавляя в него свои собственные шейдеры по мере вашей работы с рецептами из книги. Разобравшись с проектом, вы будете готовы окунуться в прекрасный мир шейдинга!

Как это сделать...

Прежде чем мы займёмся нашим первым шейдером, давайте создадим небольшую сцену, с которой мы будем работать. Для этого, в ре-

дакторе Unity зайдите в меню **Game Object** (Игровой объект) | **Create Other** (Создать другое). Там вы можете создать плоскость (Plane), которая будет играть роль земли, парочку сфер (Sphere), к которым мы будем применять наш шейдер, и направленный источник света (Directional Light), чтобы осветить сцену. После того как мы создали сцену, мы можем перейти к следующим шагам написания шейдера:

1. В панели **Project** (Проект) редактора Unity нажмите правой кнопкой по папке **Assets** (Ресурсы) и выберите **Create** (Создать) | **Folder** (Папку).

Если вы используете проект Unity, предоставляемый с этой книгой рецептов, то вы можете перейти к шагу 4.

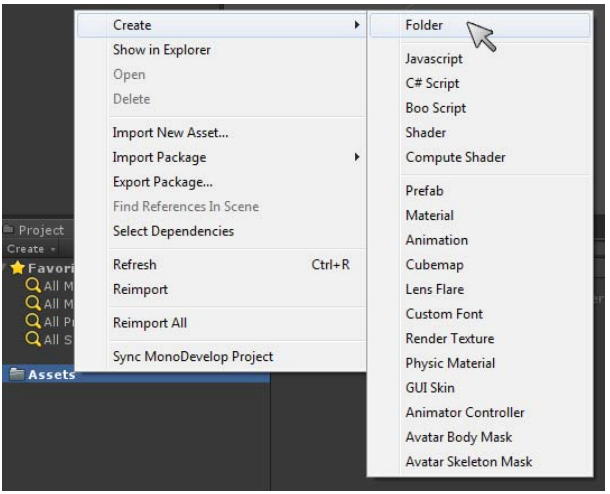

Рис. 1.1. Создание в проекте новой папки

2. Переименуйте папку, которую вы создали, в **Shaders**, нажав на неё правой кнопкой мышки и выбрав **Rename** (Переименовать) из выпадающего списка, либо выбрав папку и нажав **F2** на клавиатуре.
3. Создайте ещё одну папку и назовите её **Materials**.
4. Нажмите правой кнопкой мыши по папке **Shaders** и выберите **Create** (Создать) | **Shader** (Шейдер). После этого нажмите правой кнопкой мыши по папке **Material** и выберите **Create** (Создать) | **Material** (Материал).
5. Переименуйте и Shader, и Material в **BasicDiffuse**.

6. Запустите **BasicDiffuse** шейдер в **MonoDevelop** (редактор скриптов по умолчанию для Unity), сделав двойной щелчок мышкой по нему. Это автоматически запустит редактор и отобразит код шейдера.

Вы увидите, что в только что созданном шейдере уже есть какой-то код. Unity по умолчанию создаёт самый простой diffuse-шейдер с одной текстурой. Изменяя и дополняя этот код, мы научимся разрабатывать наши собственные шейдеры.

7. Теперь давайте переименуем наш шейдер и поместим его в отдельную папку. Первая строчка кода шейдера задаёт его имя и путь, которые будут отображаться в выпадающем списке в Unity при назначении шейдера материалу. Мы переименовали наш шейдер в `"CookbookShaders/BasicDiffuse"`, но вы можете переименовать его во что захотите и когда захотите. Так что сейчас можете за это не волноваться. Сохраните шейдер в MonoDevelop и вернитесь в редактор Unity. Unity автоматически скомпилирует шейдер, когда увидит, что файл был обновлён. На данном этапе ваш шейдер должен выглядеть так:

```
Shader "CookbookShaders/BasicDiffuse"
{
 Properties
 {
   _MainTex ("Base (RGB)", 2D) = "white" {}
 }
 SubShader
 {
    Tags { "RenderType"="Opaque" }
  LOD 200

  CGPROGRAM
  #pragma surface surf Lambert

  sampler2D _MainTex;

  struct Input
  {
   float2 uv_MainTex;
  };

  void surf (Input IN, inout SurfaceOutput o)
  {
   half4 c = tex2D (_MainTex, IN.uv_MainTex);
   o.Albedo = c.rgb;
```

```
    o.Alpha = c.a;
   }
   ENDCG
  }
  FallBack "Diffuse"
}
```

8. Выберите материал **BasicDiffuse**, который мы создали на шаге 4, и посмотрите на панель **Инспектора**. Из выпадающего списка **Shader** (Шейдер) выберите **CookbookShaders | BasicDiffuse** (путь к вашему шейдеру может отличаться, если вы решили использовать другое имя). Таким образом, мы назначим шейдер материалу и сможем теперь применить его к объектам на сцене.

*Чтобы назначить материал объекту, вы можете просто перетащить его из панели **Project** (Проект) на объект в сцене или на панель инспектора при выделенном объекте.*

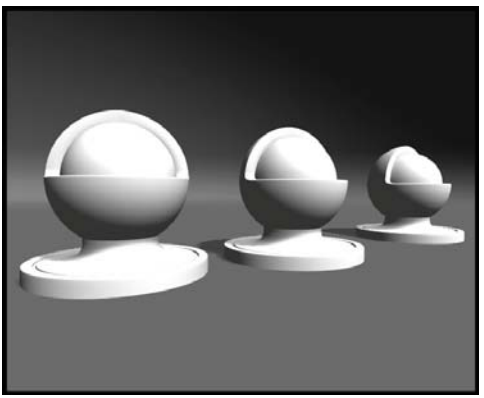

Рис. 1.2. Объекты, используемые в данном рецепте

И хотя пока что особо не на что смотреть, тем не менее мы настроили нашу среду для разработки шейдеров и теперь можем приступить к модификации шейдера под наши нужды.

Как это работает...

Как видите, в Unity настроить среду разработки шейдеров дело буквально нескольких кликов. В поверхностном шейдере незаметно для пользователя работает много компонентов. Unity сделала шейдер-

ный язык Cg более эффективным, генерируя за вас большую часть Cg-кода. Поверхностные шейдеры – это более компонентно ориентированный способ написания шейдеров. Такие задачи, как обработка текстурных координат и матриц преобразований, уже решены за вас, так что вам больше не придётся каждый шейдер начинать с нуля. Раньше же, начав писать новый шейдер, нам бы пришлось переписывать большие куски кода снова и снова. По мере того как вы будете набираться опыта при работе с поверхностными шейдерами, вы, естественно, захотите узнать больше о заложенных в основу функциях языка Cg и о том, как Unity обрабатывает за вас все низкоуровневые задачи, выполняемые **графическим процессором (GPU)**.

Итак, мы создали простой диффузный шейдер, который уже правильно взаимодействует с источниками света и тенями. И всё, что мы сделали, – поменяли одну строчку кода, переименовав наш шейдер.

Дополнительная информация

Если вы хотите узнать подробнее, какие встроенные функции доступны вам при написании поверхностных шейдеров, загляните в папку `Editor\Data\CGIncludes` в установленной версии Unity. В этой папке находятся три файла, на которые вам стоит обратить внимание: `UnityCG.cginc`, `Lighting.cginc` и `UnityShaderVariables.cginc`. На данном этапе наш шейдер использует все эти файлы.

Подробнее на файлах из папки `CGIncludes` мы остановимся в главе 9 «Делаем наш шейдерный мир модульным с помощью CgIncludes».

Добавление свойств поверхностному шейдеру

Свойства шейдера играют очень важную роль в разработке и использовании шейдеров. Для каждого из свойств Unity автоматически создаёт элементы интерфейса в панели **Инспектора**, с помощью которых можно легко настраивать шейдер непосредственно в редакторе. Откройте ваш шейдер в MonoDevelop и обратите внимание на блок кода с третьей по шестую строку. Этот блок называется **Блоком свойств**. На данный момент в нём присутствует лишь одно свойство – **_MainTex**. Если вы посмотрите на ваш материал, который использует этот шейдер, то заметите, что в панели **Инспектора** есть поле для настройки **текстуры**. Это поле было автоматически создано из его описания в блоке свойств.

И опять Unity проделала большую работу за нас, сделав процесс задания и изменения свойств лёгким и эффективным.

Как это сделать...

Давайте посмотрим, как это работает, на примере нашего шейдера **BasicDiffuse**, для этого создадим наши собственные свойства и познакомимся с используемым синтаксисом:

1. В блоке свойств нашего шейдера удалите текущее свойство, стерев у шейдера следующую строчку:

```
_MainTex ("Base (RGB)", 2D) = "white" {}
```

2. Теперь добавьте следующий код, сохраните шейдер и вернитесь в редактор Unity:

```
_EmissiveColor ("Emissive Color", Color) = (1,1,1,1)
```

3. Когда вы вернётесь в Unity и шейдер скомпилируется, вы увидите, что на панели **Инспектора** материала вместо поля для выбора текстуры появилась палитра выбора цвета под названием **Emissive Color** (Цвет испускаемого излучения). Давайте добавим ещё одну строчку кода и посмотрим, что произойдёт. Наберите следующий код:

```
_AmbientColor ("Ambient Color", Color) = (1,1,1,1)
```

4. Мы добавили ещё одну палитру выбора цвета на панель **Инспектора** материала. Давайте добавим еще одно свойство, чтобы получить представление о том, какие ещё типы свойств мы можем создать. В блок свойств добавьте следующий код:

```
_MySliderValue ("This is a Slider", Range(0,10)) = 2.5
```

5. Мы создали ещё один элемент интерфейса, который позволяет нам визуально взаимодействовать с нашим шейдером. В этот раз мы создали слайдер под названием «**This is a Slider**» («Это слайдер»), что проиллюстрировано на следующем скриншоте:

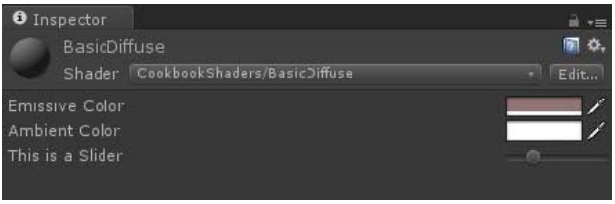

Рис. 1.3. Элементы GUI

Свойства позволяют вам визуально настраивать шейдеры без необходимости менять вручную код.

Как это работает...

Все шейдеры в Unity следуют определенной структуре. Блок свойств – один из тех элементов, которые Unity ожидает увидеть в коде. Это сделано для того, чтобы предоставить вам как программисту шейдеров способ быстро создавать элементы интерфейса, связанные напрямую с кодом шейдера. Свойства, объявляемые вами в блоке свойств, могут быть впоследствии использованы в коде шейдера для изменения числовых значений, цветов и текстур.

Рис. 1.4. Структура свойств

Давайте посмотрим, как это работает. Когда вы добавляете новое свойство, вы должны сопоставить ему **имя переменной**, с которой будет связано это свойство. Эта переменная должна быть определена в коде, и текущее значение свойства будет автоматически ей присваиваться. Таким образом, это экономит нам уйму времени, поскольку нам не нужно самим беспокоиться о передаче внешних параметров в шейдер.

Следующие компоненты описания свойства, – это его название в **Инспекторе** и его тип. Название используется в панели **Инспектора** во время редактирования параметров материала. Тип свойства задаёт тип данных, которые будут передаваться в шейдер через него. В Unity можно использовать свойства следующих типов:

Таблица 1.1. Типы свойств поверхностных шейдеров

`Range (min, max)`	Создаёт свойство типа float (число с плавающей запятой), в виде слайдера с указанным диапазоном от минимального значения до максимального
`Color`	Создаёт цветовую палитру на закладке **Инспектора**, которая вызывает селектор цвета = (float,float,float, float)

`2D`	Создаёт элемент для выбора текстуры, который позволяет пользователю перетаскивать текстуру на шейдер
`Rect`	Создаёт элемент для выбора текстурные кратной степени двойки, функционирует также как 2D элемент GUI
`Cube`	Создаёт элемент для выбора кубической текстуры (cube map) в **Инспекторе** и позволяет пользователю перетаскивать их на шейдер
`Float`	Создаёт свойство типа float в **Инспекторе**, но без слайдера
`Vector`	Создаёт свойство из четырёх float, что позволяет задавать направления или цвета

И последнее, –это значение свойства по умолчанию, которое оно принимает, если это свойство не редактировалось в **Инспекторе**. Так что в примере, изображённом на рисунке, значение по умолчанию свойства `_AmbientColor` типа `Color` задано как `(1,1,1,1)`. Так как это свойство в качестве значения ожидает цвет в формате RGBA, которое в коде хранится в переменной типа `float4`, где `(r,g,b,a) = (x, y, z, w)`, то это свойство в момент его создания принимает значение белого цвета.

Дополнительная информация

Документация по свойствам приводится в руководстве по Unity, которое можно найти по адресу: http://docs.unity3d.com/Documentation/Components/SL-Properties.html.

Использование свойств в поверхностном шейдере

Теперь, после того как мы создали несколько свойств, давайте свяжем их с переменными в коде, чтобы их можно было использовать для настройки шейдера, и сделаем процесс изменения параметров более интерактивным.

К каждому свойству в блоке свойств мы привязали имя переменной, которую это свойство контролирует. Таким образом, мы сможем использовать значения свойств в коде шейдера. Но для этого сперва нужно изменить код шейдера и добавить в него определение этих переменных.

Как это сделать

Следующие шаги демонстрируют вам, как использовать свойства в поверхностных шейдерах:

1. Для начала давайте удалим следующие строки кода, так как мы удалили свойство **_MainTex** в рецепте «Создаём простой шейдер» этой главы:

```
sampler2D _MainTex;
half4 c = tex2D(_MainTex, IN.uv_MainTex);
```

2. Далее добавьте следующие строки кода после строки `CGPROGRAM`.

```
float4 _EmissiveColor;
float4 _AmbientColor;
float _MySliderValue;
```

3. Теперь мы можем использовать значения из свойств нашего шейдера. Давайте сделаем это, присвоив `o.Albedo` сумму значений свойств `_EmissiveColor` и `_AmbientColor`. Для этого добавьте следующий код в функцию `surf`:

```
void surf (Input IN, inout SurfaceOutput o)
{
 float4 c;
 c = pow((_EmissiveColor + _AmbientColor), _MySliderValue);
 o.Albedo = c.rgb;
 o.Alpha = c.a;
}
```

4. Код получившегося шейдера приведён ниже. Если вы сохраните ваш шейдер в MonoDevelop и вернетесь в Unity, то шейдер скомпилируется. Если вы не сделали ошибок, то теперь у вас появится возможность менять излучаемый и фоновый цвета материала, а также вы сможете менять насыщенность итогового цвета с помощью значения слайдера. Здорово, да?

```
Shader "CookbookShaders/BasicDiffuse"
{
  //Объявляем свойства в блоке свойств.
  Properties
  {
    _EmissiveColor ("Emissive Color", Color) = (1,1,1,1)
    _AmbientColor  ("Ambient Color", Color) = (1,1,1,1)
    _MySliderValue ("This is a Slider", Range(0,10)) = 2.5
  }
  SubShader
```

```
  {
    Tags { "RenderType"="Opaque" }
    LOD 200

    CGPROGRAM
    #pragma surface surf Lambert
    //Нам нужно объявить переменные свойств внутри CGPROGRAM,
    //чтобы мы смогли получить доступ к значениям этих
    //переменных из блока свойств.
    float4 _EmissiveColor;
    float4 _AmbientColor;
    float _MySliderValue;

    struct Input
    {
      float2 uv_MainTex;
    };

    void surf (Input IN, inout SurfaceOutput o)
    {
      //После этого мы можем использовать значения свойств
      //в нашем шейдере.
      float4 c;
      c = pow((_EmissiveColor + _AmbientColor), _MySliderValue);
      o.Albedo = c.rgb;
      o.Alpha = c.a;
    }
    ENDCG
  }
  FallBack "Diffuse"
}
```

Функция `pow(arg1, arg2)` *– это встроенная функция, которая выполняет математическую операцию возведения в степень. Аргумент 1 – это число, которое мы хотим возвести в степень, а аргумент 2 – это степень, в которую мы хотим возвести.*

Чтобы узнать больше о функции `pow()`*, загляните в CgTutorial. Это прекрасный бесплатный ресурс, где вы можете больше узнать про шейдинг и который содержит перечень всех функций, доступных в языке шейдинга Cg (http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_e.html).*

Следующий скриншот показывает результат использования свойств для контроля цвета и насыщенности материала из панели **Инспектора**:

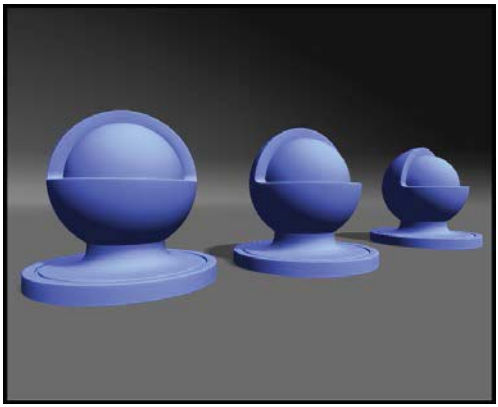

Рис. 1.5. Результат использования свойств

Как это работает...

Когда вы объявляете новое свойство в блоке свойств, вы предоставляете шейдеру способ получить изменённое значение с панели **Инспектора** материала. Это значение хранится в переменной, имя которой мы задали при создании свойства. В данном случае `_AmbientColor`, `_EmissiveColor` и `_MySliderValue` – это переменные, в которых мы храним изменяемые значения. Для того чтобы можно было использовать значения переменных в блоке `SubShader{}`, вам нужно будет создать три новые переменные с именами, такими же как и у переменных свойств. Это автоматически создаст связь между ними, благодаря которой они будут работать с одними и теми же данными. Кроме того, это задаст и тип данных, которые мы хотим хранить в переменных, что нам пригодится в последующей главе – когда мы займёмся оптимизацией шейдеров.

Как только вы объявите переменные в блоке `SubShader`, вы сможете использовать их значения в функции `surf()`. В данном случае мы хотим просуммировать переменные `_EmissiveColor` и `_AmbientColor`, а потом возвести их в степень, которая задана переменной `_MySliderValue` из панели **Инспектора** материала.

Выполнив эти действия, мы задали фундамент, который потребуется для любого шейдера, использующего диффузный компонент.

Делаем собственную модель диффузного освещения

Использовать встроенные в Unity функции освещения, несомненно, просто, но вы быстро перерастёте этот этап и захотите делать гораздо более специализированные световые модели. Исходя из нашего опыта, мы ни разу не работали над проектом, в котором мы использовали только встроенные модели освещения и были этим довольны. Мы создавали собственные модели освещения практически для всего. Благодаря этому мы, например, смогли сделать эффект задней подсветки, реализовать типы освещения, основанные на кубмапах, или даже контролировать, как шейдер реагирует на события гейм-плея, что можно наблюдать на примере шейдера, симулирующего эффект силового поля.

Этот рецепт посвящен созданию собственной модели диффузного освещения, которую мы будем использовать для реализации нескольких различных эффектов.

Как это сделать...

Давайте воспользуемся простым диффузным шейдером, который мы создали по предыдущему рецепту, и модифицируем его, выполнив следующие шаги:

1. Сперва измените директиву `#pragma` в следующем коде:

```
#pragma surface surf BasicDiffuse
```

2. Далее добавьте следующий код:

```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  float difLight = max(0, dot (s.Normal, lightDir));
  float4 col;
  col.rgb = s.Albedo * _LightColor0.rgb * (difLight * atten * 2);
  col.a = s.Alpha;
  return col;
}
```

3. Сохраните шейдер в редакторе MonoDevelop и вернитесь в Unity. Шейдер скомпилируется, и если всё прошло хорошо, вы увидите, что внешне наш материал никак не изменился. Но это не значит, что наш код не работает. Этим кодом мы удалили связь со встроенным в Unity диффузным освещением и созда-

ли нашу собственную модель освещения, которую мы сможем настраивать.

Как это работает...

В этом коде много важных частей, давайте разберём каждую из них по отдельности, чтобы понять, почему код работает именно так:

- директива `#pragma` говорит шейдеру, какую модель освещения использовать для его расчётов. Когда мы создали новый шейдер, он работал, потому что модель освещения Lambert определена в файле `Lighting.cginc`. Поэтому мы могли её использовать в шейдере. Теперь же мы дали указание шейдеру использовать модель освещения под названием **BasicDiffuse**;
- чтобы создать новую модель освещения, нужно написать функцию, которая будет считать эту модель. Название этой функции должно начинаться с Lighting, то есть `Lighting<ваше название>`. Вы можете использовать один из трёх вариантов:
  - `half4 LightingName (SurfaceOutput s, half3 lightDir, half atten){}`
    эта функция используется при Forward-рендеринге, когда направление взгляда не требуется;
  - `half4 LightingName (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten){}`
    эта функция используется при Forward-рендеринге, когда требуется направление взгляда;
  - `half4 LightingName_PrePass (SurfaceOutput s, half4 light){}`
    эта функция используется при Deferred-рендеринге;
- скалярное произведение – это ещё одна встроенная в язык Cg математическая функция. Мы можем использовать её для того, чтобы сравнить направления двух векторов в пространстве. С помощью скалярного произведения можно узнать, параллельны друг другу два вектора или перпендикулярны. Так, применив эту функцию к двум векторам, вы получите значения от –1 до 1, где –1 будет соответствовать случаю, когда сравниваемый вектор параллелен и противонаправлен вашему взгляду, 1 будет соответствовать случаю, когда сравниваемый вектор параллелен и сонаправлен вашему взгляду, 0 же соответствует случаю, когда сравниваемый вектор перпендикулярен направлению вашего взгляда;

> *«Скалярное произведение нормализованных векторов N и L является мерой угла между двумя векторами. Чем меньше угол между векторами, тем больше будет значение скалярного произведения и тем больше падающего света получит поверхность».*
>
> *Источник: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter05.html.*

- чтобы завершить расчёт диффузного компонента, нам нужно перемножить его с данными, предоставленными нам Unity в структуре `SurfaceOutput`. Для этого мы перемножаем значение `s.Albedo` (из нашей функции `surf`) с получаемым значением `_LightColor0.rgb` (оно предоставляется Unity), а потом результат этого произведения умножаем на (`difLight * atten * 2`). После этого мы возвращаем получившийся цвет. Взгляните на следующий код:

```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  float difLight = max(0, dot (s.Normal, lightDir));
  float4 col;
  col.rgb = s.Albedo * _LightColor0.rgb * (difLight * atten * 2);
  col.a = s.Alpha;
  return col;
}
```

На следующем скриншоте показан результат применения нашего простого диффузного шейдера:

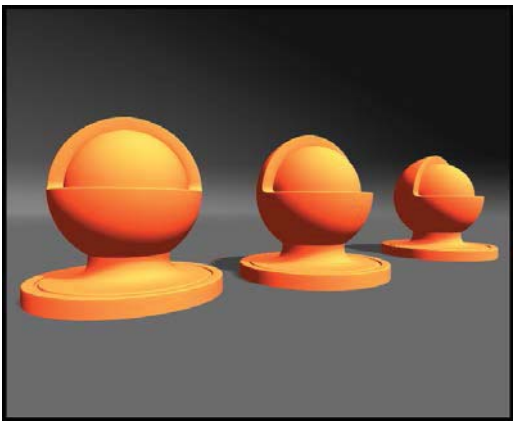

Рис. 1.6. Результат применения диффузного шейдера

Но это ещё не всё...

Воспользовавшись встроенной в язык Cg функцией `max`, мы можем ограничить значения, получаемые из функции скалярного произведения. Функция `max` принимает два аргумента – `max(arg1, arg2)`, – и возвращает наибольший их них. Мы используем её в нашем шейдере, для того чтобы быть уверенными, что значения, которые мы будем применять для вычисления диффузного компонента, всегда больше или равны нулю. Таким образом, мы исключаем из обработки отрицательные значения, особенно `-1`, которое бы создало в шейдере чрезвычайно чёрные области, что было бы неудобно с точки зрения дальнейших вычислений.

В библиотеке языка Cg также есть функция `saturate`. Она принимает один аргумент и возвращает значение в диапазоне от 0 до 1.

Дополнительная информация...

Дополнительную информацию об аргументах функций модели освещения поверхностного шейдера можно найти по адресу: http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaderLighting.html.

Модель освещения Half Lambert

Half Lambert – это техника, созданная в компании Valve, чтобы поверхность объектов, на которые свет «падает» под большим углом, тоже подсвечивалась. В модели освещения Lambert цвет этих объектов уходит в чёрный.

Half Lambert – это техника, которая была впервые использована в оригинальном Half-Life (https://developer.valvesoftware.com/wiki/Half-Life). Техника была разработана для того, чтобы тыльная сторона объекта не теряла формы и не выглядела слишком плоской. Half Lambert – это совершенно не физически реалистичная техника, и она создаёт исключительно визуальные улучшения. Это пример нестрогой модели освещения.

Ссылка: https://developer.valvesoftware.com/wiki/Half_Lambert.

Как это сделать...

Взяв шейдер, на котором мы остановились в предыдущем рецепте, давайте изменим в нём рассчет диффузного цвета:

- измените вычисление диффузного компонента, умножив его на 0,5. Для этого добавьте следующий код в функцию освещения:

```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  float diffLight = dot(s.Normal, lightDir);
  float hLambert = difLight * 0.5 + 0.5;

  float4 col;
  col.rgb = s.Albedo * LightColor0.rgb * (hLambert * atten * 2);
  col.a = s.Alpha;
  return col;
}
```

Результат внедрения техники Half Lambert в нашу модель освещения показан на следующем скриншоте:

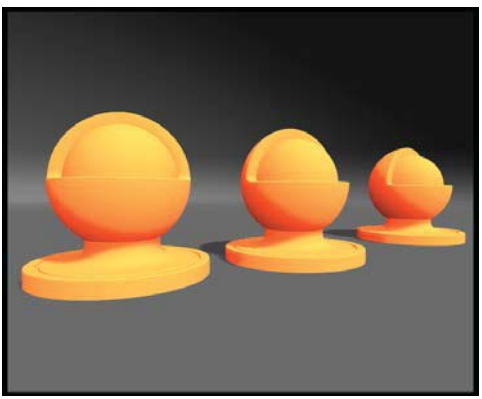

Рис. 1.7. Использование Half Lambert

Как это работает...

Техника Half Lambert преобразует диапазон значений скалярного произведения нормали к поверхности и направления света из [–1, 1] в [0, 1]. Таким образом, значение 1 так и остаётся 1, а 0 преобразуется в 0,5. При этом места с отрицательными значениями тоже становятся подсвечены. Это уменьшает затемнённость модели в местах, на которые свет «падает» под большим углом.

На следующей картинке видно, как сдвигается кривая диффузного компонента при применении техники Half Lambert.

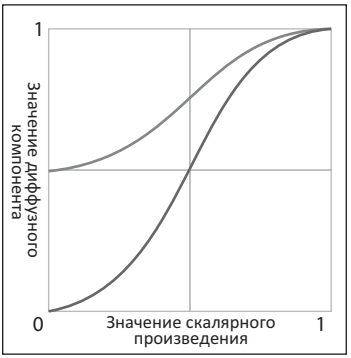

Рис. 1.8. График диффузного компонента при применении техники Half Lambert

Использование текстуры для контроля над диффузным шейдингом

Еще одним отличным инструментом при работе с шейдерами является использование рэмп-текстур для управления цветом диффузного компонента освещения. С помощью рэмп-текстур вы сможете подчеркнуть цвет поверхности, чтобы сымитировать эффекты переотражения света или более сложной системы освещения. Эта техника чаще используется в мультяшных играх, где художественный вид ваших шейдеров будет ценнее, чем физически достоверная световая модель.

Эта техника стала более популярной с игрой TeamFortress 2, в которой Valve нашла уникальный подход к освещению своих персонажей. На эту тему они написали очень известную статью, которую вам определённо стоит прочесть.

Статья Valve по освещению и шейдингу TeamFortress 2 доступна по адресу: http://www.valvesoftware.com/publications/2007/NPAR07_IllustrativeRenderingInTeamFortress2.pdf.

Подготовка

Для начала работы по данному рецепту вам потребуется подготовить рэмп-текстуру в каком-либо графическом редакторе. Для целей кон-

кретно этого примера мы использовали Photoshop, однако сделать градиент можно в любом графическом редакторе.

Рис. 1.9. Градиент

Как это сделать...

Давайте начнём подготовку нашего шейдера с написания следующего кода:

- измените функцию освещения:

```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  float difLight = dot(s.Normal, lightDir);
  float hLambert = difLight * 0.5 + 0.5;
  float3 ramp = tex2D(_RampTex, float2(hLambert)).rgb;

  float4 col;
  col.rgb = s.Albedo * _LightColor0.rgb * (ramp);
  col.a = s.Alpha;
  return col;
}
```

Результат выполнения этого кода представлен на следующей иллюстрации.

Рис. 1.10. Результат выполнения кода

Как это работает...

Встроенная функция `tex2D()` языка Cg возвращает цвет из текстуры в формате RGBA. Она принимает два аргумента: текстуру, из которой мы хотим получить цвет, и UV-координаты пикселя в этой текстуре.

В данном случае мы не собираемся использовать UV-координаты, полученные из вершины 3D-объекта, вместо этого мы хотим отобразить диапазон значений диффузного компонента на рэмп-текстуру. В конечном итоге эта операция «обернёт» текстуру вокруг поверхности объекта, основываясь на направлении источника света, для которого проводятся вычисления.

Мы берем перерасчитанные значения диффузного компонента из Half Lambert, передаем их во `float2()` и получаем текстурные координаты. Если переменная `hLambert` принимает значение 0, то функция `tex2D` ищет значение пикселя с UV-координатами (0, 0). В нашем случае это нежный персиковый цвет из текстуры. Когда переменная `hLambert` принимает значение 1, `tex2D` будет использовать значение пикселя с UV-координатами (1, 1), то есть белый цвет.

Теперь художник уже может лучше контролировать освещение объекта под разными углами. Поэтому эту технику чаще всего можно увидеть в тех проектах, где акцент делается на наглядности.

Имитация эффекта BRDF с помощью 2D-текстуры

Мы можем улучшить предыдущий рецепт, если будем использовать направление взгляда в функции освещения для создания более интересного эффекта. Используя направление взгляда, например, мы сможем сымитировать заднюю подсветку (rim lighting).

Если мы посмотрим на технику диффузного освещения с помощью рэмп-текстуры, то увидим, что мы используем лишь одно значение для UV-координат текстуры. А это значит, что полученный эффект будет одномерным. В этом рецепте мы изменим нашу функцию освещения таким образом, чтобы она использовала дополнительный аргумент – **направление взгляда**.

Направление взгляда, – это направление, в котором пользователь смотрит на объект. Это вектор, который мы можем использовать совместно с нормалью и направлением света. Использование вектора взгляда позволит нам усложнить метод сэмплинга текстуры.

В индустрии компьютерной графики эта техника часто называется **эффект BRDF**. BRDF расшифровывается как Bidirectional Reflectance Distribution Function (двулучевая функция отражательной способности). Эта закрученная формулировка относится всего лишь к тому, как свет отражается от непрозрачной поверхности с учётом направления света и направления взгляда. Чтобы наглядно увидеть эффект работы BRDF шейдера, давайте вернёмся к нашей сцене и продолжим работу над шейдером.

Подготовка

В этот раз для начала нам потребуется более продвинутая рэмп-текстура. Нам нужно включить в неё градиенты по обоим направлениям текстуры.

1. Создайте новую текстуру с размерами 512×512.
2. Создайте градиент, направив его по диагонали из левого нижнего угла квадрата в верхний правый.
3. Создайте ещё один градиент, начинающийся из левого верхнего угла квадрата и заканчивающийся перед его центром.
4. И наконец, создайте градиент, начинающийся из правого нижнего угла квадрата и заканчивающийся перед его центром. У вас должна получиться текстура, показанная на следующей иллюстрации:

Рис. 1.11. Используемая в данном рецепте текстура

Как это сделать...

В этом рецепте мы будем отталкиваться от простого диффузного шейдера:

1. Сначала нам нужно изменить нашу функцию освещения так, чтобы учитывать предоставляемую Unity переменную `viewDir`, что позволит нам получить направление камеры на

сцене, при котором она смотрит на наши объекты. Для этого отредактируйте функцию освещения:

```
inline float4 LightingDiffuse (SurfaceOutput s, fixed3 lightDir,
half3 viewDir, fixed atten)
{
  float difLight = dot(s.Normal, lightDir);
  float hLambert = difLight * 0.5 + 0.5;
  float3 ramp = tex2D(_RampTex, float2(hLambert)).rgb;

  float4 col;
  col.rgb = s.Albedo * _LightColor0.rgb * (ramp);
  col.a = s.Alpha;
  return col;
}
```

2. После этого нам нужно посчитать скалярное произведение направления взгляда и нормали к поверхности (как показано в следующем коде).

```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3
lightDir, half3 viewDir, fixed atten)
{
  float difLight = dot(s.Normal, lightDir);
  float rimLight = dot(s.Normal, viewDir);
  float hLambert = diffLight * 0.5 + 0.5;
  float3 ramp = tex2D(_RampTex, float2(hLambert)).rgb;

  float4 col;
  col.rgb = s.Albedo * _LightColor0.rgb * (ramp);
  col.a = s.Alpha;
  return col;
}
```

3. Для завершения операции нам нужно подставить результат скалярного произведения в функцию `tex2D()`.

```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3
lightDir, half3 viewDir, fixed atten)
{
  float difLight = dot(s.Normal, lightDir);
  float rimLight = dot(s.Normal, viewDir);
  float hLambert = diffLight * 0.5 + 0.5;
  float3 ramp = tex2D(_RampTex, float2(hLambert, rimLight)).rgb;

  float4 col;
  col.rgb = s.Albedo * _LightColor0.rgb * (ramp);
  col.a = s.Alpha;
  return col;
}
```

4. Сохраните ваш шейдер и вернитесь в Unity. Проверьте, что вы используете вашу новую BRDF-текстуру в качестве рэмп-текстуры для Unity. Обратите внимание, что ваше освещение теперь содержит два эффекта подсветки по краям – один на нижней части модели, другой на верхней.

Следующее изображение показывает результат применения BRDF рэмп-текстуры для задания общего диффузного цвета. Эта техника очень удобна для команды разработчиков, поскольку позволяет художнику обновлять текстуру с помощью Photoshop, а не за счёт настройки света в игре:

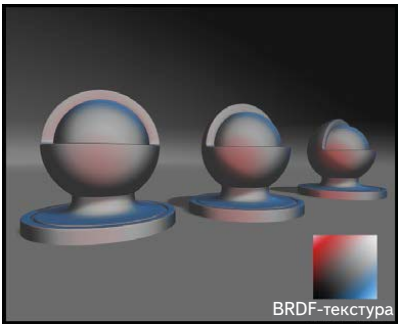

Рис. 1.12. Результат применения BRDF рэмп-текстуры

Как это работает...

Используя направление взгляда, мы можем сделать весьма незамысловатый эффект затухания. Направление взгляда позволяет создавать различные типы эффектов: прозрачность в форме пузырька, заднюю подсветку, эффекты защитного силового поля или даже эффект мультяшного контура.

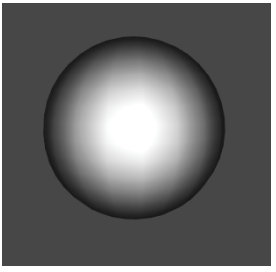

Рис. 1.13. Скалярное произведение направления взгляда и нормали к поверхности

Изображение, расположенное выше, демонстрирует значение скалярного произведения направления взгляда и нормали к поверхности. А поскольку значения `diffLight` и `rimLight` лежат в диапазоне от 0 до 1, мы можем использовать эти значения как координаты при сэмплинге BRDF-текстуры.

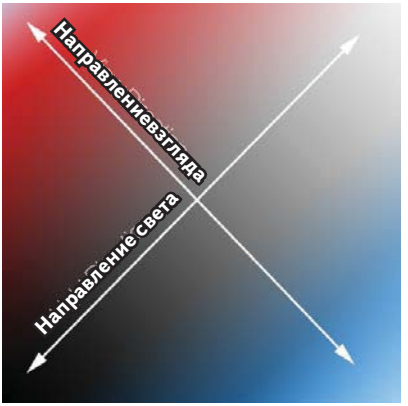

Рис. 1.14. Визуализация выбора шейдером цвета для поверхности

Самое важное в этом рецепте – понять, какие значения мы получаем из скалярных произведений, и то, как мы можем использовать текстуру в функции освещения, чтобы «обернуть» её вокруг поверхности для имитации более сложных эффектов.

Дополнительная информация

Информация о BRDF на Polycount: wiki.polycount.com/BrdfMap.

ГЛАВА 2
Создание эффектов с помощью текстур

В этой главе мы рассмотрим способы использования текстур для создания шейдерных эффектов. Как мы убедились в предыдущей главе, используя текстуры, можно добиться более сложных эффектов освещения. Помимо этого, мы можем использовать текстуры для того, чтобы анимировать, смешивать да и вообще управлять любым другим свойством, каким только пожелаем. В этой главе мы узнаем о следующих методах:

- прокрутка текстур с помощью изменения UV-координат;
- анимирование спрайт-листов;
- упаковка и блендинг текстур;
- использование карт нормалей;
- создание процедурных текстур в редакторе Unity;
- эффект уровней Photoshop.

Введение

Использование текстур – это быстрый способ сделать наши шейдерные эффекты более реалистичными. Однако следует внимательно следить за количеством используемых в шейдере текстур, поскольку накапливаются они очень быстро, а добавление каждой новой текстуры в шейдер будет сказываться на производительности. Это особенно актуально для приложений на мобильных платформах, в которых количество текстур должно быть минимальным, чтобы ваше приложение скачивалось быстро и быстро работало.

Текстуры – это изображения, которые обычно рисуются в графических редакторах, таких как Photoshop, но их можно создавать и непосредственно в Unity. Текстуры накладываются на поверхность объекта с помощью UV-координат, которые задают соотношение между

2D-точкой текстуры и 3D-точкой вершины. Значения пикселей интерполируются между вершинами объекта, что создает иллюзию наложения 2D-картинки на 3D-поверхность.

Как добавить свойство с текстурой в шейдер, мы уже разобрались в прошлой главе, поэтому заново мы на этом останавливаться не будем. Однако, если вы хотите узнать подробнее, как именно работает отображение текстуры на 3D-объект, вы можете прочитать об этом по адресу: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter03.html.

Давайте начнём с того, что посмотрим, что мы можем делать с текстурами и как их использовать, чтобы сделать наши real-time 3D-эффекты ещё более интересными и убедительными. В этой главе мы начнем с самых простых текстурных эффектов, но ближе к концу главы они станут более сложными и более зрелищными.

Прокрутка текстур с помощью изменения UV-координат

Один из приёмов, наиболее распространённых в игровой индустрии на сегодняшний день, – это прокрутка текстур по поверхности объекта. С помощью этого приёма можно создавать эффекты анимации водопадов, рек, потоков лавы и т. п. Кроме того, эта же техника лежит в основе создания эффектов анимированных спрайтов, но про них мы поговорим в следующем рецепте этой главы. Для начала давайте посмотрим, как в поверхностном шейдере сделать простой эффект прокрутки.

Подготовка

Вам потребуется создать новый шейдер и новый материал. Таким образом, у нас будет пустой шейдер, на котором мы потренируемся с эффектом прокрутки.

Как это сделать...

Откройте ваш новый шейдер и следуйте инструкции:

1. Нам понадобятся два новых свойства, с помощью которых мы будем контролировать скорость прокрутки текстуры. Поэтому давайте добавим свойство скорости по оси X и свойство

скорости по оси Y, так, как показано в следующем фрагменте кода.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _ScrollXSpeed ("X Scroll Speed", Range(0, 10)) = 2
  _ScrollYSpeed ("Y Scroll Speed", Range(0, 10)) = 2
}
```

2. Создайте новые переменные в секции `CGPROGRAM`, чтобы мы могли получить доступ к значениям из наших свойств.

```
fixed4 _MainTint;
fixed _ScrollXSpeed;
fixed _ScrollYSpeed;
sampler2D _MainTex;
```

3. Измените код в функции `surf()`, чтобы он изменял UV-координаты, передаваемые в функцию `tex2D()`. Далее используйте встроенную переменную `_Time`, чтобы добавить UV-координатам анимацию по времени, которую мы увидим, когда в редакторе нажмём кнопку **Play**.

```
void surf (Input IN, inout SurfaceOutput o)
{
  //Создадим отдельную переменную для хранения наших UV
  //перед тем, как передать их в функцию tex2D().
  fixed2 scrolledUV = IN.uv_MainTex;

  //Добавим переменные для хранения смещения по x и y
  fixed xScrollValue = _ScrollXSpeed * _Time.x;
  fixed yScrollValue = _ScrollYSpeed * _Time.x;

  //Применяем итоговый сдвиг UV
  scrolledUV += fixed2(xScrollValue, yScrollValue);

  //Применяем текстуру и цвет
  half4 c = tex2D(_MainTex, scrolledUV);
  o.Albedo = c.rgb * _MainTint;
  o.Alpha = c.a;
}
```

Изображение ниже иллюстрирует результат применения системы прокрутки UV-координат для создания простого эффекта течения реки. В книге тяжело показать анимацию, поэтому придётся поверить нам на слово, что картинка движется.

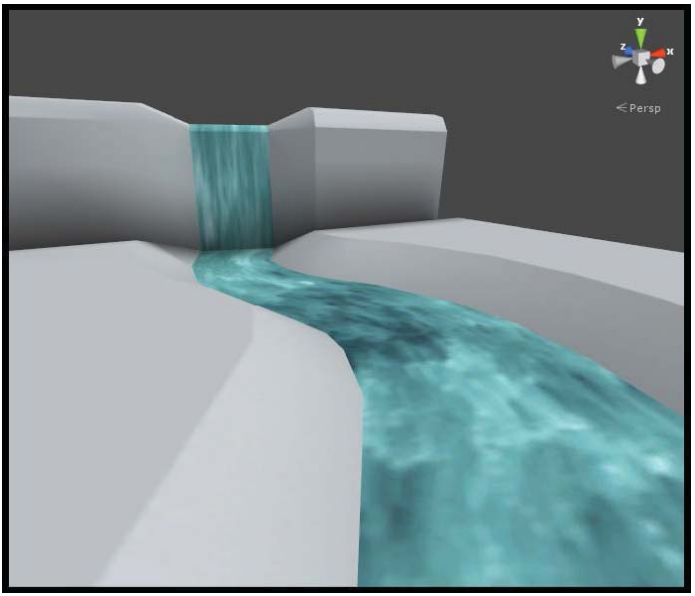

Рис. 2.1. Течение воды

Как это работает...

Мы начали разработку системы прокрутки текстур с объявления пары свойств, которые позволят пользователю шейдера увеличивать или уменьшать скорость эффекта прокрутки. С помощью свойств поверхностная функция получает значения типа `float` из текстовых полей в **Инспекторе** материала. Дополнительную информацию о свойствах шейдеров вы можете найти в главе 1 «Диффузный шейдинг».

Таким образом, установив связь между переменными в коде и текстовыми полями **Инспектора** материала, мы сможем использовать их для изменения значений UV-координат в шейдере.

Сперва мы сохраняем значения UV-координат в отдельной переменной `scrolledUV`. Эта переменная должна иметь тип `float2` или `fixed2`, так как значения UV-координат мы получаем из структуры `Input`.

```
struct Input
{
     float2 uv_MainTex;
};
```

После того как мы получим UV-координаты вершин модели, мы сможем двигать их, используя переменные со скоростями прокрутки по осям и встроенную переменную `_Time`. Эта встроенная переменная имеет тип `float4`, и ее компоненты содержат следующие значения: (t/20, t, t*2, t*3), где t – игровое время. Детальное описание встроенных переменных находится по адресу: http://docs.unity3d.com/Documentation/Components/SL-BuiltinValues.html.

Переменная `_Time` возвращает возрастающие значения типа `float` на основе игрового времени Unity. Поэтому мы можем использовать эти значения для того, чтобы сдвигать UV-координаты, масштабируя время с помощью переменных скоростей прокрутки:

```
//Добавим переменные для хранения смещения по x и y
fixed xScrollValue = _ScrollXSpeed * _Time.x;
fixed yScrollValue = _ScrollYSpeed * _Time.x;
```

Рассчитав правильный сдвиг, мы прибавляем его к первоначальным UV-координатам. Для этого мы используем в следующей строчке оператор +=. Мы хотим к первоначальным значениям UV-координат прибавить значение смещения, а затем подставить его в функцию `tex2D()` в качестве новых UV-координат текстуры. Таким образом, получится эффект движущейся текстуры по поверхности объекта. Но, как вы поняли, на самом деле мы всего лишь изменяем UV-координаты, изображая движение.

```
//Применяем итоговый сдвиг UV
scrolledUV += fixed2(xScrollValue, yScrollValue);

//Применяем текстуру и цвет
half4 c = tex2D(_MainTex, scrolledUV);
```

Анимирование спрайт-листов

Навык анимации с помощью спрайт-листов может пригодиться где угодно. Эту технику можно использовать для создания эффекта частиц или эффекта мультфильма, нарисованного на страницах блокнота, но чаще всего эту технику можно увидеть в 2D-скроллерах (играх, в которых экран постоянно прокручивается в одну сторону).

*Спрайт-лист также называют **атласом спрайтов** (sprite atlas) или просто последовательностью изображений (image sequence). Это большая текстура, в которой упаковано много маленьких изображений в определённом порядке.*

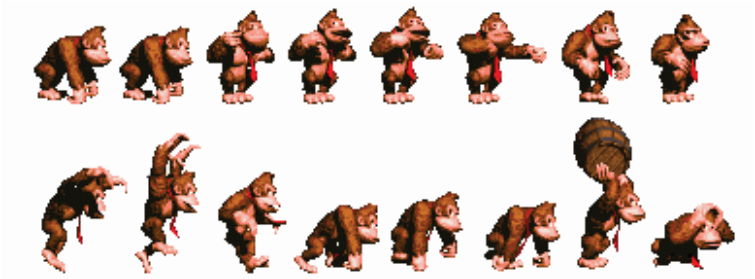

Рис. 2.2. Спрайт-лист с обезьяной

Когда вы переходите от одного маленького изображения из спрайт-листа к другому, вы видите, как обезьяна двигается. Принцип тут такой же, как и в движущихся изображениях, получаемых при перелистывании страниц блокнота, или на киноплёнке. Эффект анимации создаётся за счёт быстрого перемещения по изображениям из спрайт-листа.

В коде этого рецепта будет использовано больше математики, чем в предыдущих, но не волнуйтесь – мы пройдёмся по каждой новой строчке кода и детально их все прокомментируем.

Подготовка

Для того чтобы мы могли протестировать наш шейдер, нам потребуются тестовые изображения. Вы можете сделать спрайт-лист самостоятельно либо найти его в Интернете. Спрайт-лист не должен быть сложным, в нём просто должна быть последовательность кадров, которые мы будем анимировать. Одно лишь важное условие – для нашего шейдера спрайт-лист должен быть горизонтальным и состоять из одной строки изображений. Спрайт-лист, прилагаемый к книге, можно найти по адресу: www.packtpub.com/support.

Рис. 2.3. Спрайт-лист с идущим человеком

Создайте новый материал и новый шейдер. Затем примените этот материал к плоскости на сцене. После этого добавьте текстуру спрайт-листа материалу.

Как это сделать...

Выполните следующие шаги:

1. В блоке свойств шейдера создайте три новых свойства. С помощью них мы будем изменять параметры системы из **Инспектора** материала, не прибегая к жёсткому заданию значений в коде:

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}

  //Добавим следующие свойства
  _CellAmount ("Cell Amount", float) = 0.0
  _Speed ("Speed", Range(0.01, 32)) = 12
}
```

2. Затем сохраните входные UV-координаты в отдельной переменной, для того чтобы в дальнейшем мы смогли работать с их значениями:

```
//Сохраним UV в переменной
float2 spriteUV = IN.uv_MainTex;
```

3. После этого нам нужно будет узнать ширину каждой клетки. В спрайт-листе UV-координаты находятся в диапазоне от 0 до 1, нам нужно вычислить процент от всей текстуры, который занимает одна ячейка:

```
//Вычислим, сколько занимает одна клетка
float cellUVPercentage = 1/_CellAmount;
```

4. Теперь нужно вычислить номер кадра в спрайт-листе, используя который, мы будем смещать UV-координаты:

```
//Вычислим номер кадра для сдвига UV
float frame = fmod(_Time.y * _Speed, _CellAmount);
frame = floor(frame);
```

5. И наконец, мы сдвигаем *x*-координату UV спрайт-листа. После этих действий ваш шейдер, создающий эффект анимации, должен быть готов.

```
//Изменим UV в соответствии с текущим кадром
float xValue = (spriteUV.x + frame) * cellUVPercentage;
spriteUV = float2(xValue, spriteUV.y);
```

На следующем изображении приводится результат смещения UV-координат объекта в поверхностном шейдере. И вновь вам придётся поверить нам на слово, что картинка движется.

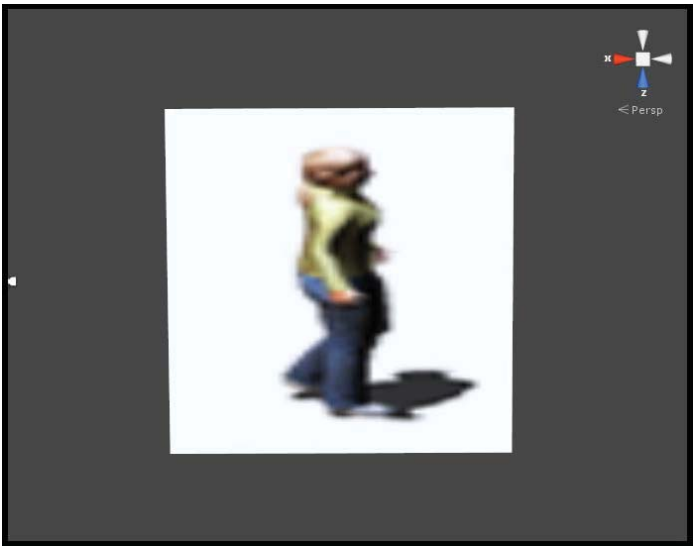

Рис. 2.4. Анимация походки человека

Как это работает...

Код начинается с того, что мы сохраняем значения UV-координат, получаемые из структуры `Input`, в новую переменную. На самом деле это делать не обязательно, поскольку это скорее вопрос удобства, чем жёсткое правило, – просто так код становится более читаемым. В данном случае мы назвали нашу новую переменную `spriteUV`, а тип ей задали как `float2`. Мы сделали так потому, что нам нужно хранить `x` и `y` компоненты UV-координат модели в одной переменной.

Далее нам нужно узнать, какой процент ширины текстуры занимает каждый кадр. Для этого мы просто делим 1 на количество клеток в спрайт-листе. Полученное значение показывает, на сколько нам нужно сдвинуть UV-координаты по оси *x*, чтобы попасть на следующую клетку спрайт-листа.

Кроме этого, нам потребуются значения, которые бы увеличивались со временем, но были бы целыми числами. Например, такое значение может увеличиваться по закономерности 0, 1, 2, 3, 4 и т. д., вплоть до общего количества клеток в нашем спрайт-листе. Для этого мы можем использовать встроенную в CGFX функцию `fmod()`.

Таблица 2.1. Описание функции `fmod`

Функция	Описание
`fmod(x, y)`	Возвращает остаток от деления `x` на `y` с тем же знаком, что и `x`. Если `y` равен 0, то возвращаемый результат зависит от конкретной реализации функции

Если мы подставим значение `x` в функцию `fmod()` и поделим его на значение `y`, то получим остаток от деления. Поэтому если мы будем использовать переменную `_Time` в качестве `x`, а в качестве `y` возьмём значение свойства `_CellAmount`, то в результате мы получим возрастающие с течением времени значения, которые будут повторяться по достижении значения `_CellAmount`. К этому значению мы применяем функцию `floor()`, чтобы отбросить дробную часть. Таким образом, мы получим номер текущего кадра в диапазоне от `0` до `_CellAmount - 1`.

Таблица 2.2. Описание функции `floor`

Функция	Описание
`floor(x)`	Возвращает минимальное целое, не большее, чем `x` (отбрасывает дробную часть)

Теперь нам нужно изменить первоначальные UV-координаты так, чтобы отображался только текущий кадр анимации. Так как UV-координаты находятся в диапазоне от 0 до 1, текущий кадр по `x` будет начинаться в `frame * cellUVPercentage`. К этому значению мы прибавляем первоначальное значение UV-координаты `x`, отмасштабированное из координат всей текстуры (0, 1) до размеров одного кадра (`0`, `cellUVPercentage`). После этого всё, что остаётся сделать, – это передать новые значения UV-координат текстуры в функцию `tex2D`.

Но это ещё не всё...

Вероятно, вы уже догадались, что можно использовать не только одно направление сдвига. Точно так же, как мы использовали два направления в предыдущем рецепте прокрутки, мы можем использовать анимированный 2D спрайт-лист. Для этого вам нужно будет учитывать смещение не только по *x,* но и по *y*.

Принцип работы точно такой же, как и при горизонтальной прокрутке, но так вы сможете обрабатывать большие спрайт-листы в нескольких направлениях. Таким образом, вы видите, что шейдеры могут выполнять весьма непростые задачи. Но чем сложнее шейдер, тем

больше он использует шейдерных инструкций, что может привести к снижению производительности приложения. Для борьбы с этим вы можете вынести код с выбором сдвига в скрипт на C#, взаимодействующий с шейдером, что позволит эти вычисления перенести на CPU и делать их не для каждого пикселя, а всего лишь один раз. Когда речь заходит об оптимизации, то всё упирается в «балансировку» вашего приложения, однако никогда не помешает подумать о том, с чем вам придется столкнуться в будущем, и учесть это при разработке приложения. К этой книге прилагается код на C#, который демонстрирует, как с помощью скрипта, передающего данные шейдеру, можно реализовать простую систему анимации спрайтов. Этот скрипт передает в шейдер номер кадра, вычисление которого он берёт на себя с помощью следующего кода.

```
void FixedUpdate ()
{
 timeValue = Mathf.Ceil(Time.time % 16);
 transform.renderer.material.setFloat("_TimeValue", timeValue);
}
```

Дополнительная информация

Если вы не горите желанием писать с нуля систему анимации спрайтов, то можете заглянуть в Asset Store – там вы найдёте много пакетов, решающих большую часть задач спрайтовой анимации.

Приведём список некоторых из них:

- **SpriteManager** (бесплатный): http://wiki.unity3d.com/index.php?title=SpriteManager
- **2D ToolKit** ($65.00): http://www.unikronsoftware.com/2dtoolkit/
- **Sprite Manager 2** ($150.00): http://anbsoft.com/middleware/sm2/

А если вы ищете хорошее приложение, которое помогло бы вам при создании спрайтов, то следующий список как раз для вас:

- **TimelineFX** ($46.79): http://www.rigzsoft.co.uk/
- **Anime Studio Pro** ($199.99): http://anime.smithmicro.com/index.html
- **Adobe Flash Professional** ($699.00): http://www.adobe.com/products/flash.html

Упаковка и блендинг текстур

Текстуры, помимо всего прочего, хорошо подходят для хранения массивов данных, содержащих не просто цвета пикселей, к которым мы уже привыкли. В RGBA-каналах текстуры могут находиться разные другие данные. Мы даже можем упаковать несколько изображений в одну RGBA-текстуру и использовать каждый компонент R, G, B, A по отдельности, извлекая их независимо друг от друга в коде шейдера.

Результат упаковки отдельных черно-белых изображений в одну RGBA-текстуру представлен на следующем рисунке:

Рис. 2.5. Иллюстрация упаковки черно-белых изображений в RGBA-текстуру

В чём преимущество такого подхода? Большая часть памяти, расходуемая вашим приложением, выделяется под хранение текстур. Поэтому для уменьшения размера приложения мы можем попробовать объединить текстуры, используемые шейдером, в одну общую текстуру.

Любая черно-белая текстура может быть упакована в один из RGBA-каналов другой текстуры. Хотя сначала такой приём может показаться слегка странным, но этот рецепт продемонстрирует применение упаковки текстур и их использование в шейдере.

Например, упакованные текстуры можно использовать, когда вы хотите выполнить блендинг (англ. *to blend* – смешивать) нескольких текстур на одной поверхности. Чаще всего эта техника используется в ландшафтных шейдерах, когда одна текстура должна плавно переходить в другую с помощью некоей контролирующей текстуры или, как в нашем случае, с помощью упакованной текстуры. Этой технике и посвящён текущий рецепт, он покажет, как вы можете сделать основную часть ландшафтного шейдера, в котором используется блендинг четырех текстур.

Подготовка

Давайте создадим новый шейдер и новый материал для него. Политика задания имён для файлов шейдера и материалов остаётся на ваше

усмотрение, поэтому постарайтесь придумывать такие имена, которые будет легко упорядочить и найти в дальнейшем.

После того как вы создадите шейдер и материал, создайте новую сцену для тестирования нашего шейдера.

Вам также понадобится выбрать четыре текстуры, которые вы будете блендить. Текстуры могут быть любыми, но для хорошего ландшафтного шейдера мы советуем использовать траву, грязь, грязь с камнями и просто камни.

В этом рецепте мы будем использовать следующие цветовые текстуры, которые прилагаются к данной книге.

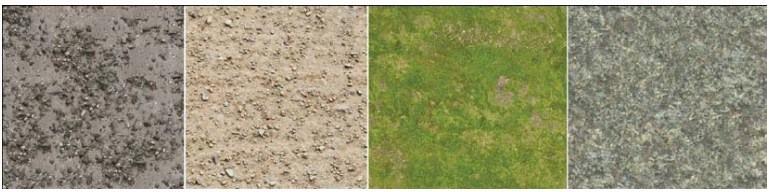

Рис. 2.6. Ландшафтные текстуры, прилагаемые к книге

Кроме того, нам потребуется блендинг-текстура, в которую мы упакуем черно-белые изображения. Итого у нас будет четыре черно-белых канала, которые мы можем использовать для управления размещением текстур на поверхности объекта.

Мы можем использовать сложные блендинг-текстуры для создания очень реалистичного распределения текстур поверхности земли на модели рельефа, что демонстрирует следующее изображение:

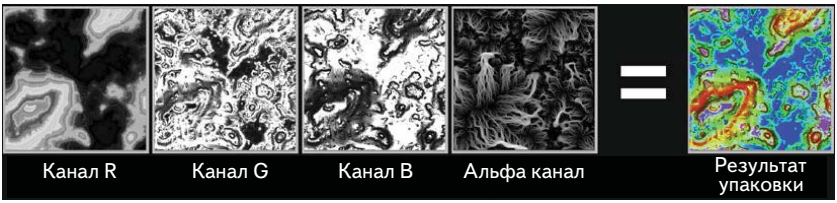

Рис. 2.7. Упаковка сложных текстур реалистичного ландшафта

Как это сделать...

Давайте начнём знакомство с использованием упакованных текстур, для этого выполним следующие шаги:

1. Добавим несколько свойств в блок свойств. Нам потребуются пять объектов `sampler2D`, или же текстур, и два цвета.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)

  //Добавьте эти свойства, чтобы мы могли задавать текстуры
  _ColorA ("Terrain Color A", Color) = (1,1,1,1)
  _ColorB ("Terrain Color B", Color) = (1,1,1,1)
  _RTexture ("Red Channel Texture", 2D) = "" {}
  _GTexture ("Green Channel Texture", 2D) = "" {}
  _BTexture ("Blue Channel Texture", 2D) = "" {}
  _ATexture ("Alpha Channel Texture", 2D) = "" {}
  _BlendTex ("Blend Texture", 2D) = "" {}
}
```

2. После этого нам нужно будет объявить переменные, с помощью которых мы получим доступ к данным в блоке свойств.

```
CGPROGRAM
#pragma surface surf Lambert

float4 _MainTint;
float4 _ColorA;
float4 _ColorB;
sampler2D _RTexture;
sampler2D _GTexture;
sampler2D _BTexture;
sampler2D _ATexture;
sampler2D _BlendTex;
```

3. Итак, теперь наши свойства текстур готовы, и мы их передаём в поверхностную функцию. Нам потребуется изменить структуру `Input`, чтобы позволить пользователю менять параметры тайлинга для отдельных текстур. Так мы сможем использовать параметры `tiling` и `offset` для каждой текстуры:

```
struct Input
{
  float2 uv_RTexture;
  float2 uv_GTexture;
  float2 uv_BTexture;
  float2 uv_ATexture;
  float2 uv_BlendTex;
};
```

4. В функции `surf` мы сохраним информацию о текстурах в соответствующие переменные, чтобы нам было легче работать с данными:

```
//Получаем данные из блендинг-текстуры.
//Здесь мы используем float4, потому что данные хранятся
//в R,G,B и A или X,Y,Z и W каналах.
```

```
float4 blendData = tex2D(_BlendTex, In.uv_BlendTex);

//Получаем цвета из текстур, которые мы хотим блендить
float4 rTexData = tex2D(_RTexture, IN.uv_RTexture);
float4 gTexData = tex2D(_GTexture, IN.uv_GTexture);
float4 bTexData = tex2D(_BTexture, IN.uv_BTexture);
float4 aTexData = tex2D(_ATexture, IN.uv_ATexture);
```

5. Давайте объединим наши текстуры с помощью функции `lerp()`. Она принимает два цвета и производит интерполяцию между ними, используя число в диапазоне от 0 до 1, передаваемое в последнем аргументе:

```
//Теперь нужно объединить все цвета в один
float4 finalColor;
finalColor = lerp(rTexData, gTexData, blendData.g);
finalColor = lerp(finalColor, bTexData, blendData.b);
finalColor = lerp(finalColor, aTexData, blendData.a);
finalColor.a = 1.0;
```

6. И наконец, мы перемножим смешанные цвета текстур с цветами поверхности, для интерполяции между которыми используется красный канал блендинг-текстуры.

```
//Добавим наши цвета ландшафта
float4 terrainLayers = lerp(_ColorA, _ColorB, blendData.r);
finalColor *= terrainLayers;
finalColor = saturate(finalColor);

o.Albedo = finalColor.rgb * _MainTint.rgb;
o.Alpha = finalColor.a;
```

Результат объединения четырёх ландшафтных текстур и применения техники terrain tinting представлен на следующем изображении.

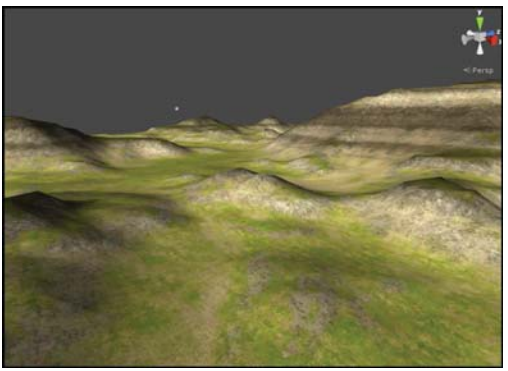

Рис. 2.8. Ландшафт, получаемый в результате применения техники terrain tinting

Как это работает...

Вы могли удивиться количеству кода, но концепция блендинга на самом деле весьма проста. Для этой техники мы используем встроенную функцию `lerp()` из стандартной библиотеки CGFX. Эта функция позволяет нам выбрать значение между первым и вторым аргументами с помощью третьего аргумента, используемого как показатель смешивания.

Таблица 2.3. Описание функции `lerp`

Функция	Описание
`lerp(a, b, f)`	Выполняет линейную интерполяцию (1 – `f`) * `a` + `b` * `f`, где `a` и `b` одного типа (векторного или скалярного). `f` может быть скалярным, или векторным – того же типа что `a` и `b`

Поэтому, к примеру, если нам потребуется найти среднее между 1 и 2, мы можем подставить в функцию `lerp` 0,5 в качестве третьего аргумента – в результате мы получим 1,5. Нам это отлично подходит, поскольку значения отдельных каналов RGBA-текстур являются значениями с плавающей точкой, обычно в диапазоне от 0 до 1.

В шейдере мы просто берем один из каналов из нашей блендинг-текстуры и используем его как параметр функции `lerp` для управления цветом каждого пикселя. К примеру, за основу мы возьмём текстуру с травой и текстуру с грязью, использовать будем красный канал нашей блендинг-текстуры – всё это мы подадим на вход функции `lerp()`. В результате мы получим корректно смешанный цвет для каждого пикселя поверхности. Более наглядная демонстрация результатов работы функции `lerp` приводится на следующем изображении.

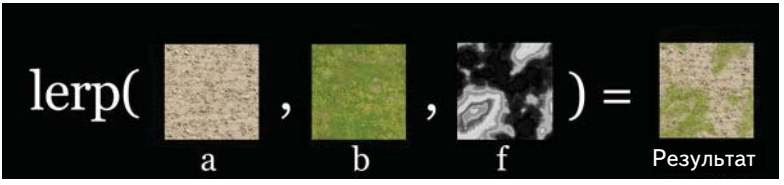

Рис. 2.9. Идея работы функции `lerp`

В коде шейдера для создания финальной текстуры используются все 4 канала блендинг-текстуры плюс 4 отдельные RGBA-текстуры. Эта итоговая текстура и становится нашим цветом, который мы можем перемножить с диффузным освещением.

Дополнительная информация

Рельеф для этого рецепта был сделан с помощью программы World Machine. Используя эту программу, вы можете с лёгкостью создавать очень сложные ландшафтные блендинг-текстуры и модели.

- **World Machine** ($189.00):
  http://www.world-machine.com/

Использование карты нормалей

Одна из наиболее распространённых на сегодняшний день техник, применяемых в игровой индустрии, – это использование карт нормалей. С её помощью мы можем создать видимость детализированной геометрии на малодетализированных моделях. Идея техники заключается в том, что вместо вычисления освещения для каждой вершины мы используем нормаль к поверхности в каждом пикселе, которые закодированы в пикселях карты нормалей, в результате чего у нас получается гораздо более детальная картина распределения света при сохранении небольшого числа полигонов в модели.

> *В 3D-графике normal mapping, также известная как «Dot3 bump mapping», – техника, используемая для имитации освещения впадин и выступов, является вариантом реализации Bump mapping. Техника применяется для добавления деталей без добавления полигонов. Применение этой техники приводит к эффекту существенного улучшения внешнего вида и деталей низкополигональных моделей за счёт генерации карты нормалей из высокополигональной модели, или карты высот. Карты нормалей чаще всего хранятся как обычные RGB-изображения, в которых RGB-компоненты соответствуют координатам X, Y, Z (в указанном порядке) нормалей поверхности.*

Предыдущий фрагмент текста является цитатой из Википедии (http://en.wikipedia.org/wiki/Normal_mapping).

Сегодня существует много способов создания карт нормалей. Некоторые приложения, такие как CrazyBump (http://www.crazybump.com/) и N2DO (http://quixel.se/ndo/), позволяют конвертировать 2D-данные в карты нормалей. Другие приложения, такие как Zbrush (http://www.pixologic.com/) и Mudbox (http://usa.autodesk.com), могут создавать карту нормалей на основе 3D-модели. Сам процесс со-

здания карт нормалей определённо выходит за рамки этой книги, но ссылки, приводимые выше, помогут вам разобраться с этим.

С помощью функции `UnpackNormal()` Unity позволяет легко добавить в шейдер поддержку карт нормалей, не выходя за синтаксис поверхностных шейдеров. Давайте посмотрим, как это делается.

Подготовка

Создайте новый материал и шейдер, примените их к новому объекту на сцене. В этой простой сцене мы сможем сосредоточиться только на технике normal mapping.

Для этого рецепта вам потребуется карта нормалей, однако в проекте Unity, поставляемом вместе с этой книгой, есть уже готовая текстура.

Образец карты нормалей, поставляемый с этой книгой, показан на следующем изображении.

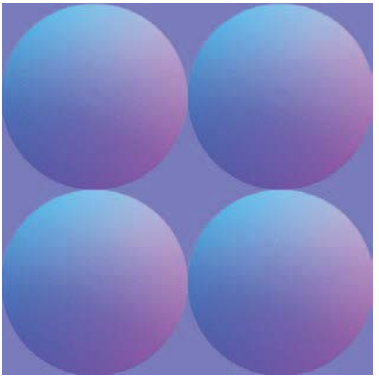

Рис. 2.10. Образец карты нормалей

Как это сделать...

1. Добавим в блок свойств цветовой тон и текстуру:

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _NormalTex ("Normal Map", 2D) = "bump" {}
}
```

2. Свяжите свойства с кодом Cg, добавив определения переменных после инструкции `CGPROGRAM`:

```
CGPROGRAM
#pragma surface surf Lambert

//Свяжем свойства с переменными
sampler2D _NormalTex;
float4 _MainTint;
```

3. Теперь нужно отредактировать структуру `Input`, задав в ней правильное имя переменной нашей текстуры, чтобы мы смогли использовать UV-координаты модели для текстуры карты нормалей.

```
//Убедитесь, что в структуре передаются UV
struct Input
{
 float2 uv_NormalTex;
};
```

4. Заключительным этапом мы извлекаем информацию о нормалях из текстуры карты нормалей с помощью встроенной функции `UnpackNormal()`. После этого потребуется только применить эти нормали к результатам работы поверхностного шейдера:

```
//Получаем направления нормалей из текстуры карты нормалей
//с помощью функции UnpackNormal().
float3 normalMap = UnpackNormal(tex2D(_NormalTex,
IN.uv_NormalTex));

//Применяем новые нормали к модели освещения
o.Normal = normalMap.rgb;
```

Следующее изображение демонстрирует результаты работы нашего шейдера:

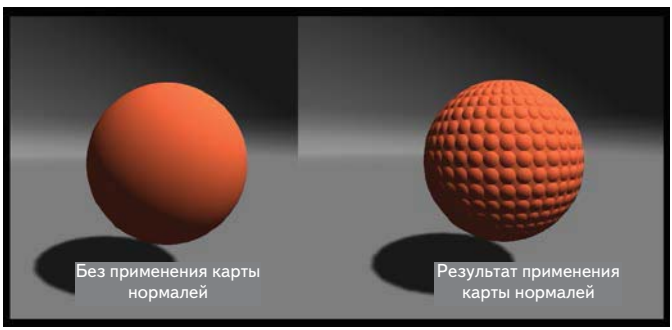

Рис. 2.11. Иллюстрация работы шейдера без и с картой нормалей

Как это работает...

Математика, используемая в normal mapping, выходит за рамки данной главы, тем более что Unity всё равно всё уже сделала за нас. Нам доступны все необходимые функции для использования карт нормалей в шейдерах, так что нам самим не придётся писать один и тот же код снова и снова. Это ещё одна причина, по которой использование поверхностных шейдеров, – это по-настоящему эффективный способ написания шейдеров.

Если вы загляните в файл `UnityCG.cginc`, который можно найти в папке `Data` в директории, куда вы установили Unity, то вы найдёте определения функции `UnpackNormal()`. Чтобы получить корректные данные для функции освещения каждого пикселя, вам нужно лишь использовать эту функцию внутри вашего поверхностного шейдера – Unity возьмет карту нормалей и обработает её за вас. Экономия времени просто колоссальная!

После того как вы распакуете карту нормалей с помощью функции `UnpackNormal()`, результат нужно будет передать в структуру `SurfaceOutput`, чтобы его можно было использовать в функции освещения. Делается это с помощью строки кода `o.Normal = normalMap.rgb;`.

Но это ещё не всё...

Мы можем добавить параметры для настройки карты нормалей шейдера, с помощью которых пользователь сможет регулировать её интенсивность. Для этого нужно всего лишь модифицировать `x` и `y` компоненты вектора нормали.

1. Добавьте ещё одно свойство в блок свойств и назовите его `_NormalIntensity`, так, как показано в следующем фрагменте кода:

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _NormalTex ("Normal Map", 2D) = "bump" {}
  _NormalIntensity ("Normal Map Intensity", Range(0,2)) = 1
}
```

2. Не забудьте объявить соответствующие переменные.

```
//Свяжем свойства с кодом
sampler2D _NormalTex;
float4 _MainTint;
float _NormalIntensity;
```

3. Умножьте x и y компоненты распакованной карты нормалей на этот новый параметр и присвойте получившееся значение переменной normalMap. Теперь пользователь сможет настраивать интенсивность карты нормалей в **Инспекторе** материала:

```
//Получим данные нормалей из текстуры карты нормалей
//с помощью функции UnpackNormal().
float3 normalMap = UnpackNormal(tex2D(_NormalTex,
IN.uv_NormalTex));
normalMap = float3(normalMap.x * _NormalIntensity,
normalMap.y * _NormalIntensity, normalMap.z);
```

Следующее изображение демонстрирует результат модификации карты нормалей нашими скалярными значениями:

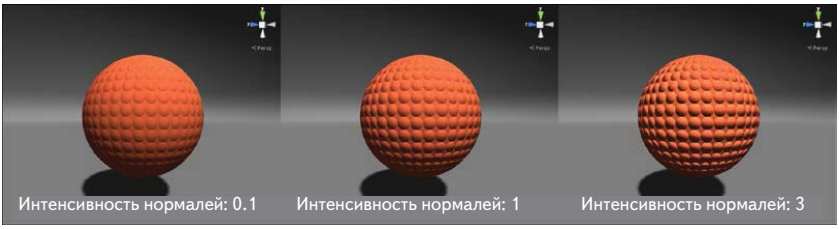

Рис. 2.12. Иллюстрация работы шейдера с различной интенсивностью карты нормалей

Создание процедурных текстур в редакторе Unity

Бывают случаи, когда вам хочется динамически создавать текстуры и модифицировать их пиксели в рантайме для достижения различных эффектов. Обычно такие эффекты называют процедурными текстурными эффектами. Вместо того чтобы вручную создавать новые текстуры в графическом редакторе, вы можете программно сгенерировать двумерный массив пикселей и добавить их в новую текстуру. После этого текстуру нужно передать в шейдер, чтобы он мог использовать её для вычислений.

Эта техника может вам пригодиться, если вы захотите нарисовать динамически созданную текстуру на уже существующей текстуре при взаимодействии игрока с окружающим игровым миром. С её помощью можно реализовать эффект следов от пуль на стене или создавать процедурные фигуры, которые потом использовать в шейдере.

Существует много ситуаций, в которых вам пригодятся создание новой текстуры, заполнение её неким процедурным паттерном и последующая обработка её в вашем шейдере.

Если мы хотим динамически генерировать текстуры, нам не обойтись без специального скрипта, который будет это делать. Давайте посмотрим, как написать такой скрипт, который будет передавать в поверхностный шейдер динамически создаваемую текстуру.

Подготовка

Для того чтобы начать работать с этим рецептом, выполните следующие действия:

1. Создайте новый C# скрипт в вашем проекте и назовите его `ProceduralTexture`.
2. Создайте на сцене пустой `GameObject` и убедитесь, что он находится в (0, 0, 0), затем назначьте ему скрипт `ProceduralTexture.cs`.
3. Далее создайте новый шейдер, новый материал и новый объект, к которому мы применим этот материал. Не забудьте задать имя шейдеру и материалу, чтобы их впоследствии можно было легко найти.
4. Сделав всё это, мы готовы написать код, который будет генерировать круговой градиент, рисовать его на текстуре и передавать эту текстуру в шейдер. К концу этого рецепта вы сделаете текстуру, которая будет выглядеть как это изображение.

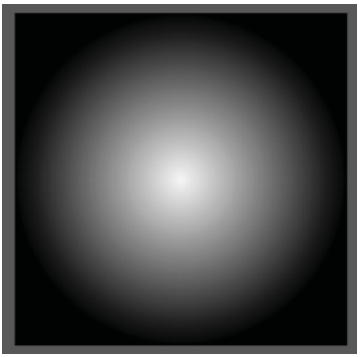

Рис. 2.13. Текстура, которая у вас должна получиться к концу рецепта

Как это сделать...

1. Добавьте переменную для изменения высоты и ширины вашей текстуры, а также переменную типа `Texture2D` для хранения сгенерированной текстуры. Кроме этого, нам потребуется несколько приватных переменных для хранения данных во время выполнения скрипта.

```
#region Public Variables
//С помощью этих переменных мы сможем изменять размер
//текстуры, а также видеть её в редакторе
public int widthHeight = 512;
public Texture2D generatedTexture;
#endregion

#region Private Variables
//Внутренние переменные скрипта
private Material currentMaterial;
private Vector2 centerPosition;
#endregion
```

2. В функции `Start()` нашего скрипта первым делом нам нужно проверить, есть ли материал у объекта, к которому относится скрипт. Если есть, то мы вызовем функцию `GenerateGradient()`, которую мы напишем далее, а возвращаемое ей значение сохраним в переменную `generatedTexture` типа `Texture2D`:

```
void Start ()
{
  //Проверяем, есть ли на этом объекте материал
  if (!currentMaterial)
  {
          currentMaterial = renderer.sharedMaterial;
          if (!currentMaterial)
          {
                  Debug.LogWarning("Cannot find a material on: "
                  + transform.name);
          }
  }

  //Генерируем процедурную текстуру
  if (currentMaterial)
  {
          //Генерируем текстуру градиента
          centerPosition = new Vector2(0.5f, 0.5f);
          generatedTexture = GenerateGradient();
```

```
            //Присваиваем её материалу текущего объекта
            currentMaterial.SetTexture("_MainTex", generatedTexture);
    }
}
```

3. После этого нам надо объявить функцию, которая будет генерировать картинку:

```
private Texture2D GenerateGradient()
{

}
```

4. Теперь добавим в функцию алгоритм, генерирующий круговой градиент для нашей текстуры. Не волнуйтесь, если на данный момент вам что-то не понятно. Мы разберём каждую строку кода в следующей секции этого рецепта.

```
private Texture2D GenerateGradient()
{
  //Создадим новую Texture2D
  Texture2D proceduralTexture = new Texture2D(widthHeight,
widthHeight);

  //Узнаём центр текстуры
  Vector2 centerPixelPosition = centerPosition * widthHeight;

  //Пройдёмся по всем пикселям, определим их расстояние от
  //центра и на основе этого присвоим им значения.
  for (int x = 0; x < widthHeight; x++)
  {
          for (int y = 0; y < widthHeight; y++)
          {
                  //Вычисляем расстояние от центра текстуры
                  //до выбранного пикселя
                  Vector2 currentPosition = new Vector2(x, y);
                  float pixelDistance = Vector2.Distance
                  (currentPosition, centerPixelPosition)/
                  (widthHeight*0.5f);

                  //Инвертируем значения и ограничиваем их
                  //диапазоном [0, 1]
                  pixelDistance = Mathf.Abs(1 - Mathf.Clamp(
                  pixelDistance, 0f, 1f));

                  //Создаём новый цвет пикселя
                  Color pixelColor = new Color(pixelDistance,
                  pixelDistance, pixelDistance, 1.0f);
                  proceduralTexture.SetPixel(x, y, pixelColor);
          }
  }
```

```
        //И наконец, записываем все изменения в текстуру
        proceduralTexture.Apply();

        //Возвращаем текстуру в основную программу.
        return proceduralTexture;
    }
```

Как это работает...

Скрипт начинает свою работу с того, что проверяет, есть ли материал у объекта на сцене, которому мы пытаемся назначить текстуру. Если да, то мы присваиваем переменной `currentMaterial` значение `renderer.sharedMaterial`, то есть наш материал.

После этого мы смотрим на нашу следующую инструкцию `if()` и проверяем, корректен ли материал. Если да, то мы вызываем функцию `GenerateGradient()`, которая вернёт нам экземпляр класса `Texture2D`.

Когда исполнение программы доходит до функции `GenerateGradient()`, то сначала с помощью конструктора `Texture2D()`, в который передаётся наша переменная `widthHeight`, создаётся новая текстура. В результате мы получаем пустую текстуру, в которой в квадрате со стороной `widthHeight` можно устанавливать цвета пикселей.

В новой текстуре мы рассчитываем положение центрального пикселя и сохраняем его в переменной `centerPixelPosition`. После чего мы добавляем два цикла, которые просто проходят по всем пикселям в нашей пустой текстуре. Если вам не знаком цикл `for` в языке C#, то вы можете найти информацию о нём на сайте http://msdn.microsoft.com/en-us/library/ch45axte.aspx.

Далее для каждого пикселя в координате `Vector2(x,y)` мы измеряем расстояние от центрального пикселя с помощью функции `Vector2.Distance()`. Эта функция возвращает число с плавающей точкой, например если на некоторой итерации циклов были выбраны координаты пикселя `Vector2(32,32)`, то мы получим расстояние от центра, равное 316,78, при условии что мы создали текстуру размером 512×512. Это число соответствует расстоянию от центра для пикселя с координатами (32, 32).

После этого нам потребуется нормализовать расстояние до центра так, чтобы оно было в диапазоне от 0,0 до 1,0, чтобы его можно было использовать в качестве цвета (компоненты цвета Unity хранит в диапазоне от 0 до 1). Всё, что нам нужно сделать для этой нормализации, – это поделить расстояние на половину ширины текстуры (или

высоты, так как она квадратная). В данном случае мы делим расстояние на 256, поскольку именно это число является половиной от 512. Поэтому для расстояния 316,78, полученного в предыдущем примере, после нормализации мы получим значение 1,23.

Теперь же нам нужно убедиться, что у нас нет значений, больших единицы или меньших нуля, – поэтому мы используем функцию `Mathf.Clamp()`, которая позволяет нам ограничить их пределами, передаваемыми в функцию в качестве аргументов. Чтобы получить нормализованные значения, мы передаём в эту функцию 0 и 1.

И наконец, мы инвертируем полученные значения, вычитая их из 1, а результат присваиваем всем трем каналам новой цветовой переменной. Взгляните на следующее изображение.

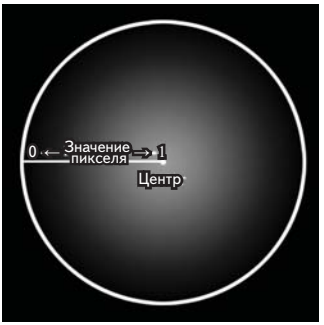

Рис. 2.14. Иллюстрация диапазонов значений пикселей – от центра и до радиуса

Но это ещё не всё...

Теперь, когда вы увидели, как можно генерировать значения пикселей с помощью всего лишь небольшой порции векторной математики, подумайте, какие еще данные можно сгенерировать и хранить в текстурах.

1. Следующий код генерирует картинку с кольцами вокруг центра текстуры:

```
//Получим расстояние от центра текстуры
//до нашего выбранного пикселя.
Vector2 currentPosition = new Vector2(x,y);
float pixelDistance = Vector2.Distance(currentPosition,
centerPixelPosition)/(widthHeight*0.5f);
pixelDistance = Mathf.Abs(1-Mathf.Clamp(pixelDistance, 0f, 1f));
pixelDistance = (Mathf.Sin(pixelDistance * 30.0f) *
pixelDistance);
```

2. Далее приводится код для вычисления скалярного произведения направления от центра изображения к пикселю и осей координат:

```
//Можно делать что-то более интересное с векторами,
//для того чтобы рассчитать другие данные по модели,
//UV или цветам пикселей.
Vector2 pixelDirection = centerPixelPosition - currentPosition;
pixelDirection.Normalize();
float rightDirection = Vector2.Dot(pixelDirection, Vector2.right);
float leftDirection = Vector2.Dot(pixelDirection, -Vector2.right);
float upDirection = Vector2.Dot(pixelDirection, Vector2.up);
```

3. А этот код используются для вычисления угла между направлением от центра изображения к пикселю и осями координат.

```
//Можно делать что-то более интересное с векторами,
//для того чтобы рассчитать другие данные по модели,
//UV или цветам пикселей.
Vector2 pixelDirection = centerPixelPosition - currentPosition;
pixelDirection.Normalize();
float rightDirection = Vector2.Angle(pixelDirection, Vector2.right)/360;
float leftDirection = Vector2.Angle(pixelDirection, -Vector2.right)/360;
float upDirection = Vector2.Angle(pixelDirection, Vector2.up)/360;
```

Результаты, полученные при обработке пикселей с помощью вычислений различных векторов и углов, представлены на следующем изображении.

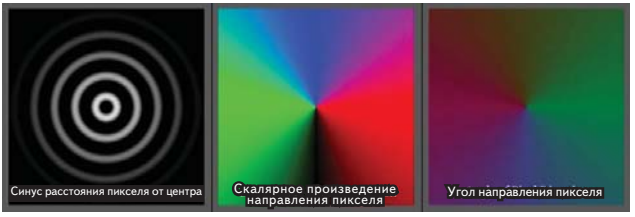

Рис. 2.15. Иллюстрация различных вычислений пикселей

Эффект уровней Photoshop

Если вам когда-либо доводилось заниматься редактированием изображений – ретушировать семейные фотографии, делать игровые текстуры или рисовать картину на компьютере, мы уверены, что вы понимаете, насколько полезен инструмент «уровни» для глобальной настройки всего изображения. Такой же Photoshop-подобный эффект можно реализовать и в шейдере.

Все разнообразные инструменты редактирования изображений и режимы блендинга, которые можно найти в Photoshop, описываются набором математических операций. Потому что, для того чтобы получить финальный цвет изображения, мы умножаем, складываем, вычитаем или сравниваем одни пиксели с другими пикселями.

И хотя мы могли бы написать целую книгу, посвященную рецептам различной математики для Photoshop-эффектов, в этой мы сфокусируемся лишь на уровнях. Более тонкие режимы блендинга мы рассмотрим в главе 10 «Создание экранных эффектов в Unity с помощью рендер-текстур».

Подготовка

Для работы с этим рецептом вам нужно будет сделать новый шейдер и новый материал, а также назначить его объекту на новой сцене Unity. Кроме того, вам понадобится исходная текстура, на которой можно будет протестировать наш код по работе с уровнями. Вы можете использовать материалы, поставляемые с этой книгой.

Как это сделать...

1. В новый шейдер добавьте следующие свойства:

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}

  //Входные значения уровней
  _inBlack ("Input Black", Range(0, 255)) = 0
  _inGamma ("Input Gamma", Range(0, 2)) = 1.61
  _inWhite ("Input White", Range(0, 255)) = 255

  //Выходные значения уровней
  _outWhite ("Output White", Range(0,255)) = 255
  _outBlack ("Output Black", Range(0,255)) = 0
}
```

2. Не забудьте также объявить эти свойства как переменные в блоке `CGPROGRAM`:

```
//Добавим переменные
float _inBlack;
float _inGamma;
float _inWhite;
float _outWhite;
float _outBlack;
```

3. Создайте новую переменную для хранения только красного канала нашей текущей текстуры `_MainTex`:

```
//Переменная для хранения красного канала текстуры
float outRPixel;
```

4. Поскольку данные, предоставляемые нам функцией `tex2D()`, нормализованы в диапазоне от 0 до 1, нам потребуется привести их к диапазону от 0 до 255.

```
//Преобразуем диапазон от 0 до 1 к диапазону от 0 до 255
outRPixel = (c.r * 255.0);
```

5. Далее мы вычитаем из этой переменной значение параметра `Input Black`, что будет делать все пиксели более темными при перемещении слайдера к значению 255.

```
//Вычтем значение чёрного в параметре _inBlack
outRPixel = max(0, outRPixel - _inBlack);
```

6. Затем сделаем все пиксели более белыми при перемещении слайдера `Input White` к 0, а результат возведём в степень входной гаммы.

```
//Увеличим значение белого для каждого пикселя
//с помощью _inWhite
outRPixel = saturate(pow(outRPixel / (_inWhite - _inBlack), _inGamma));
```

7. Перемножим новые значения пикселей с разницей выходного белого и выходного чёрного, а после вновь нормализуем значения пикселей к диапазону от 0 до 1:

```
//Изменим итоговую чёрную точку и белую точку
//и приведём диапазон от 0 до 255 к диапазону от 0 до 1.
outRPixel = (outRPixel * (_outWhite - _outBlack) + _outBlack)/255.0;
```

Следующее изображение демонстрирует итоговый эффект работы над уровнями нашей текстуры в шейдере.

Рис. 2.16. Иллюстрация применения обработки с помощью уровней

Как это работает...

Функция `surf` начинает свою работу с того что, сохраняет цвет текстуры в переменную `c`. На этом этапе мы будем работать с отдельными каналами и менять значения пикселей в них. Для этого мы создаём новую переменную, называем её `outRPixel` и присваиваем ей значение `c.r * 255`. Так мы преобразуем значения из диапазона от 0 до 1 к диапазону от 0 до 255.

После этого из текущего значения пикселя мы вычитаем значение свойства `_inBlack`, для того чтобы сделать цвет пикселя более тёмным. При этом мы контролируем, что после вычитания значение пикселя не станет отрицательным с помощью функции `max()`, которая возвращает нам максимальное из двух значений.

Таблица 2.4. Описание функции `max`

Функция	Описание
`max(a, b)`	Возвращает максимум `a` и `b`

Теперь мы поделим наше изменённое значение пикселей на значение новой точки белого. Узнать значение новой точки белого мы можем, если вычтем значение `_inBlack` из значения `_inWhite`. Это приведёт к тому, что значение пикселя увеличится, то есть он станет ярче. Увеличенное значение пикселя возводится в степень `_inGamma`, что по-существу позволяет перемещать значение серединной точки цвета текущего пикселя.

И наконец, ещё раз модифицируем пиксель с помощью `_outWhite` и `_outBlack`, для того чтобы у вас была возможность итогового глобального контроля за минимальным и максимальным значениями пикселя. После этого результат делится на 255, чтобы нормализовать его к шкале от 0 до 1.

Это значение мы присваиваем `o.Albedo` для вычисления итогового диффузного цвета. По мере того как вы будете экспериментировать со слайдерами на закладке **Инспектора** материала, вы заметите, что теперь у вас появилась возможность детально контролировать контрастность и яркость текстуры.

Но это ещё не всё...

Мы уверены, что вы заметили в шейдере много дублирующегося кода. Мы можем создать собственную функцию внутри шейдера, чтобы привести код в порядок. Так мы будем содержать наш шейдерный

код в чистоте, а с точки зрения разработчика он станет более эффективным. Взгляните на следующую функцию.

```
float GetPixelLevel(float pixelColor)
{
     float pixelResult;
     pixelResult = (pixelColor * 255.0);
     pixelResult = max(0, pixelResult - _inBlack);
     pixelResult = saturate(pow(pixelResult / (_inWhite -
     _inBlack), _inGamma));
     pixelResult = (pixelResult * (_outWhite - _outBlack) +
     _outBlack)/255.0;
     return pixelResult;
}
```

Используя эту новую функцию внутри нашего шейдера для обработки итогового уровня пикселей, мы сократили функцию `surf` до трёх строчек кода по всем каналам вместо прежних 15. Так код становится гораздо чище, и теперь, когда понадобится, мы можем делать изменения всего лишь в одном месте вместо трёх.

Дополнительная информация

Дополнительная информация по эффекту «уровней» находится на сайте GPU Gems (http://http.developer.nvidia.com/GPUGems/gpugems_ch22.html).

ГЛАВА 3
Пусть ваши игры засияют отражённым светом

Вы наверняка играли в такие игры, как Gears of War или Call of Duty, реалистичность графики в которых в некоторые моменты поражает. Как же они этого добиваются? Конечно же тут много составляющих, но одной из них, несомненно, является использование различных вариантов глянцевых материалов. В этой главе раскрываются основы построения визуальных эффектов, основанных на бликах, и демонстрируются некоторые приёмы, используемые в шейдерах гигантов индустрии компьютерных игр. В этой главе вы узнаете о:

- использовании встроенной в Unity Specular модели освещения;
- создании модели освещения Phong;
- создании модели освещения BlinnPhong;
- маскировании глянцевых бликов с помощью текстур;
- разнице между металлическими и мягкими бликами;
- создании анизотропных бликов.

Введение

Эффекты глянцевой поверхности ещё называются эффектами, зависящими от направления взгляда. Такое название связано с тем, что для достижения реалистичного визуального эффекта вам потребуется учитывать направление камеры или пользователя, который смотрит на поверхность объекта. Также для создания правдоподобных бликов важен ещё один параметр – направление источника света. Комбинируя эти два вектора, мы получаем блик – яркий участок на поверхности объекта посредине между ними. Это направление называется полувектор, и к нему мы ещё вернёмся. Кроме этого, мы узнаем, как настраивать блики для имитации металлической поверхности и поверхности ткани.

Использование встроенной в Unity Specular модели

В Unity уже встроена модель Specular освещения, которая называется **BlinnPhong**. Это одна из самых простых, но в то же время эффективных, моделей Specular освещения, которая используется в играх и по сей день. Мы решили взять её за основу и надстраивать на ней наши дополнительные эффекты. В руководстве по Unity вы можете найти примеры использования этой модели освещения, но мы копнём чуть глубже и рассмотрим, как вычисляются её параметры и почему эта модель работает именно так. Этими знаниями мы заложим хороший фундамент, от которого мы сможем отталкиваться в следующих рецептах этой главы.

Подготовка

Для начала выполните следующие действия:

1. Создайте новый шейдер.
2. Создайте новый материал, назовите его и присвойте ему только что созданный шейдер.
3. Создайте плоскость земли и сферу. Поместите её примерно в центр сцены.
4. А теперь давайте добавим направленный источник света, чтобы осветить наш объект.

После того как вы выполнили эти действия, ваша сцена должна выглядеть примерно как на следующем скриншоте.

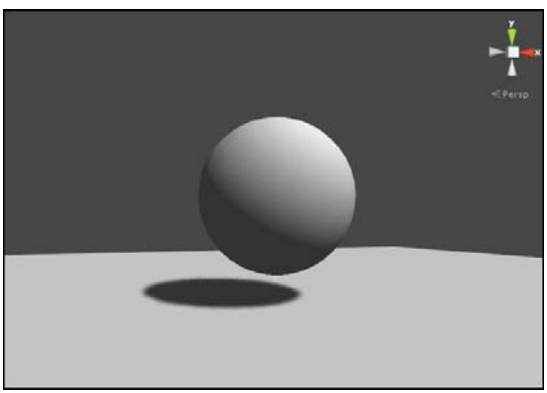

Рис. 3.1. Новая сцена

Как это сделать...

1. Начните с добавления следующих свойств в блок свойств шейдера:

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _SpecColor ("Specular Color", Color) = (1,1,1,1)
  _SpecPower ("Specular Power", Range(0,1)) = 0.5
}
```

2. Убедитесь, что вы добавили переменные в блок `CGPROGRAM`, чтобы мы смогли получить доступ к новым свойствам. Обратите внимание, что нам не нужно объявлять свойство `_SpecColor` как переменную, потому что Unity уже создала для нас эту переменную во встроенной модели Specular освещения. Всё, что нам нужно сделать, – это объявить её в блоке свойств, и тогда мы сможем передавать через неё данные в функцию `surf()`.

```
sampler2D _MainTex;
float _SpecPower;
float4 _MainTint;
```

3. Теперь шейдеру нужно указать, какую функцию освещения ему следует использовать для нашей сферы. Вам уже доводилось видеть модель освещения Lambert, и вы даже тренировались в написании собственной модели, поэтому теперь давайте посмотрим на модель BlinnPhong. Добавьте модель BlinnPhong в директиву `#pragma` следующим образом:

```
CGPROGRAM
#pragma surface surf BlinnPhong
```

4. После этого измените функцию `surf()` так, чтобы она выглядела следующим образом:

```
void surf (Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex) * _MainTint;
  o.Specular = _SpecPower;
  o.Gloss = 1.0;
  o.Albedo = c.rgb;
  o.Alpha = c.a;
}
ENDCG
```

Как это работает...

Этот небольшой Specular шейдер – хорошая стартовая точка для прототипирования других шейдеров, потому что на этом этапе вы можете написать большую часть функционала шейдера, не заботясь о функциях освещения.

Unity предоставляет нам модель освещения, в которой расчёт бликов на поверхности уже реализован. Если вы заглянете в файл `Lighting.cginc`, который находится в каталоге, куда вы установили Unity, в подпапке `Data`, то вы увидите, что в Unity уже есть модели освещения Lambert и BlinnPhong. После того как вы укажете в секции `#pragma surface surf BlinnPhong` и скомпилируете ваш шейдер, он начнёт использовать код световой функции BlinnPhong из файла `Lighting.cginc`, и нам не придётся дублировать этот код в своём шейдере.

Если вы сделаете всё без ошибок и ваш шейдер скомпилируется, то вы увидите результат, похожий на следующий скриншот:

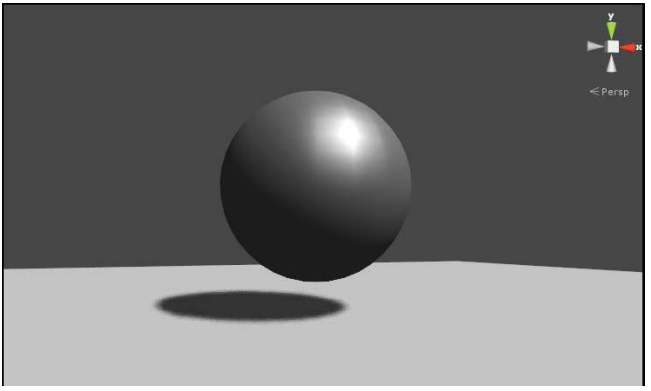

Рис. 3.2. Результат работы шейдера с моделью освещения BlinnPhong

Создаём модель освещения Phong

Модель Specular освещения Phong является наиболее простой и нетребовательной к производительности среди Specular моделей освещения. Её идея заключается в вычислении направления отражённого света от поверхности с учётом направления взгляда пользователя. Эта

модель очень распространена и используется повсеместно, начиная с игр и заканчивая фильмами. И хотя эффект от неё не самый реалистичный в плане точности моделирования отражения, он является замечательным приближением, хорошо подходящим для большинства ситуаций. Более того, если ваш объект находится далеко от камеры, а высокой точности бликов не требуется, то эта модель отлично подходит для реализации Specular освещения в вашем шейдере.

В этом рецепте мы остановимся на том, как написать повертексное освещение в данной модели, а далее с помощью добавления новых параметров в структуру `Input` мы реализуем её попиксельный вариант. Мы отдельно остановимся на их различиях и обсудим, для каких ситуаций какой из них лучше подходит.

Подготовка

Выполните следующие действия:

1. Создайте новый шейдер, материал и объект и задайте им имена так, чтобы их можно было найти в дальнейшем.
2. Назначьте шейдер материалу, а материал – объекту. И в довершение работы со сценой добавьте новый направленный источник света, чтобы мы смогли увидеть наши блики после того, как мы их запрограммируем.

Как это сделать...

1. Вероятно, вы уже видите какие-то шаги решения задачи, однако мы советуем начинать с наиболее базовой части написания шейдера – задания свойств. Поэтому давайте добавим нашему шейдеру следующие свойства.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _SpecularColor ("Specular Color", Color) = (1,1,1,1)
  _SpecPower ("Specular Power", Range(0,30)) = 1
}
```

2. После этого убедимся, что соответствующие переменные добавлены в блок `CGPROGRAM`.

```
float4 _SpecularColor;
sampler2D _MainTex;
float4 _MainTint;
float _SpecPower;
```

3. Теперь, для того чтобы рассчитать Phong Specular, нам нужно добавить собственную модель освещения. Добавьте следующий код в блок `SubShader`. Не волнуйтесь, если на данный момент что-то не понятно, – далее мы рассмотрим каждую строку кода.

```
inline fixed4 LightingPhong (SurfaceOutput s, fixed3 lightDir,
half3 viewDir, fixed atten)
{
  //Вычислим диффузный и отражённый вектор
  float diff = dot(s.Normal, lightDir);
  float3 reflectionVector = normalize(2.0 * s.Normal * diff - lightDir);

  float spec = pow(max(0, dot(reflectionVector, viewDir)), _SpecPower);
  float3 finalSpec = _SpecularColor.rgb * spec;

  fixed4 c;
  c.rgb = (s.Albedo * _LightColor0.rgb * diff) + (_LightColor0.rgb *
finalSpec);
  c.a = 1.0;
  return c;
}
```

4. И наконец, мы должны указать в блоке `CGPROGRAM`, что нужно использовать нашу функцию освещения вместо дефолтной. Чтобы это сделать, мы поменяем секцию `#pragma` следующим образом.

```
CGPROGRAM
#pragma surface surf Phong
```

Следующий скриншот демонстрирует результат использования нашей модели освещения Phong.

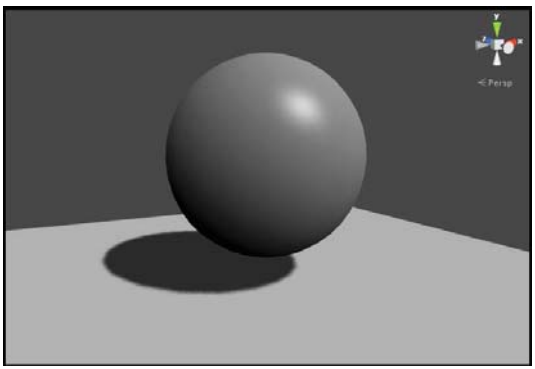

Рис. 3.3. Результат работы шейдера с моделью освещения Phong

Как это работает...

Давайте более подробно рассмотрим функцию освещения, поскольку на данном этапе остальная часть кода шейдера вам уже должна быть вполне знакома.

Начнём мы с того, что будем использовать функцию освещения, в которую передается направление взгляда. Не забывайте, что, чтобы получать нужные параметры в функцию освещения, вам нужно правильно указывать их в сигнатуре этой функции. Взгляните на следующую таблицу либо на её электронную версию по адресу: http://docs.unity3d.com/Documentation/Components/SLSurfaceShaderLighting.html.

Таблица 3.1. Параметры зависящих и не зависящих от направления взгляда функций

Не зависит от направления взгляда	`half4 ваше Название Функции Освещения (SurfaceOutput s, half3 lightDir, half atten);`
Зависит от направления взгляда	`half4 ваше Название Функции Освещения (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten);`

В нашем случае мы будем работать со Specular шейдером, поэтому структура нашей функции освещения должна соответствовать зависящему от направления взгляда формату. Соответственно, мы должны написать следующий код.

```
CGPROGRAM
#pragma surface surf Phong
inline fixed4 LightingPhong (SurfaceOutput s, fixed3 lightDir,
half3 viewDir, fixed atten)
{

}
```

Эти инструкции означают, что мы хотим использовать шейдер, учитывающий направление взгляда. Всегда проверяйте, что название функции освещения совпадает с названием, указанным в директиве `#pragma`, в противном случае Unity не сможет найти вашу модель освещения.

Функция освещения начинается с объявления обычного диффузного компонента и его вычисления с помощью скалярного произведения нормали поверхности и направления света. Таким образом, мы получим значение `1`, когда нормаль к поверхности направлена в сторону света, и `-1`, когда нормаль направлена в противоположную свету сторону.

После этого нам нужно рассчитать вектор отражения, для этого мы умножаем вектор нормали на `2*diff` и вычитаем из него направление света. Таким образом, получается эффект наклона нормали к свету – и даже если нормаль была направлена от источника света, она будет вынуждена к нему повернуться. На следующих скриншотах приводится более наглядное пояснение этой идеи. Скрипт, с помощью которого был получен скриншот во время отладки, можно найти на странице поддержки данной книги по адресу www.packtpub.com/support.

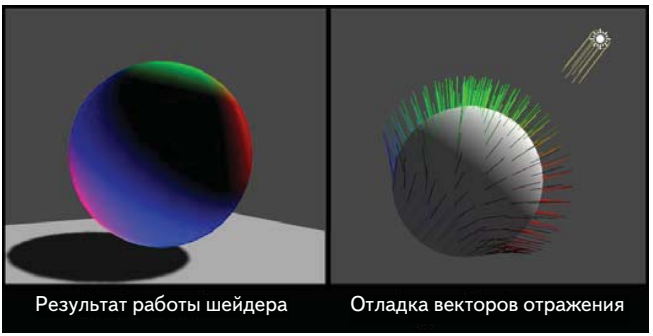

Рис. 3.4. Визуальный эффект в режиме отладки

После этого всё, что нам остаётся, – это вычислить итоговые значения отражённого цвета. Для этого мы считаем скалярное произведение вектора отражения и направления взгляда, а результат возводим в степень `_SpecPower`. После этого мы просто перемножаем значение `spec` и `_SpecularColor.rgb` и получаем блик на поверхности.

На следующем скриншоте приводятся итоговые результаты вычисления модели Phong в нашем шейдере.

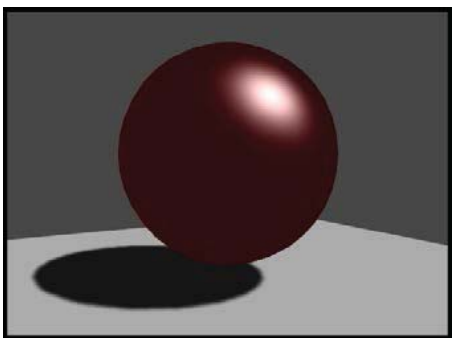

Рис. 3.5. Итоговый результат расчётов Phong отражённого света

Создание модели освещения BlinnPhong

Существует более эффективный способ расчёта и оценки отражённого света – модель Blinn. Её идея заключается в вычислении полу-вектора между направлением взгляда и направлением источника света. В мир Cg этот метод ввёл Джим Блинн (Jim Blinn). Он понял, что, вместо того чтобы считать вектор отражения, гораздо эффективнее просто взять полувектор. Это позволило уменьшить объём кода и время выполнения. Если вы посмотрите на встроенную модель освещения BlinnPhong в файле `Lighting.cginc`, то увидите, что она также использует полувектор, потому-то она и называется **BlinnPhong**. По существу, это просто упрощённый вариант вычисления модели Phong.

Подготовка

Выполните следующие действия:

1. В этот раз, вместо того чтобы создавать новую сцену целиком, давайте будем использовать имеющиеся у нас объект и сцену и сделаем новые шейдер и материал, которые мы назовём `BlinnPhong`.
2. Как только вы создадите новый шейдер, два раза кликните по нему, чтобы запустить MonoDevelop для редактирования нашего шейдера.

Как это сделать...

1. Первым делом нам нужно добавить свойства в блок свойств, для того чтобы мы смогли контролировать параметры блика на поверхности.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _SpecularColor ("Specular Color", Color) = (1,1,1,1)
  _SpecPower ("Specular Power", Range(0.1, 60)) = 3
}
```

2. После этого нам нужно убедиться, что мы создали соответствующие переменные в блоке `CGPROGRAM`, для того чтобы мы могли получить доступ к этим данным из нашего шейдера.

```
sampler2D _MainTex;
float4 _MainTint;
float4 _SpecularColor;
float _SpecPower;
```

3. Теперь нам нужно создать модель освещения, которая будет выполнять вычисления `Diffuse` и `Specular` компонент.

```
inline fixed4 LightingCustomBlinnPhong (SurfaceOutput s,
fixed3 lightDir, half3 viewDir, fixed atten)
{
  float3 halfVector = normalize(lightDir + viewDir);

  float diff = max(0, dot(s.Normal, lightDir));

  float nh = max(0, dot(s.Normal, halfVector));
  float spec = pow(nh, _SpecPower) * _SpecularColor;

  float4 c;
  c.rgb = (s.Albedo * _LightColor0.rgb * diff) + (_LightColor0.rgb *
_SpecularColor.rgb * spec) * (atten * 2);
  c.a =  s.Alpha;
  return  c;
}
```

4. Последнее, что нам нужно сделать, – это в блоке `CGPROGRAM` дать указание шейдеру использовать нашу модель освещения вместо встроенной в Unity, сделав в секции `#pragma` такие изменения.

```
CGPROGRAM
#pragma surface surf CustomBlinnPhong
```

Следующий скриншот иллюстрирует результаты работы модели освещения BlinnPhong.

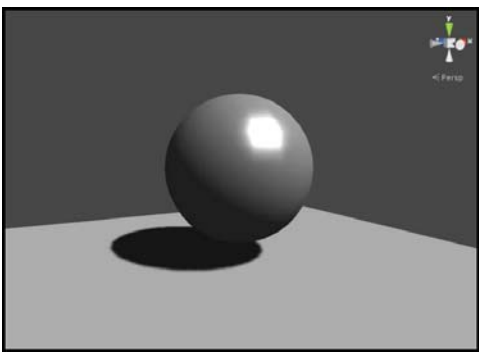

Рис. 3.6. Результаты работы модели освещения BlinnPhong

Как это работает...

Модель Specular освещения BlinnPhong почти в точности повторяет модель Phong, только она более эффективна, поскольку использует меньше кода для достижения такого же эффекта. Этот подход сегодня можно встретить в девяти шейдерах из десяти, поскольку он легче в написании и более производителен.

Вместо того чтобы рассчитывать наш собственный вектор отражения, мы просто вычисляем полувектор между направлением взгляда и направлением освещения. Кроме того, было установлено, что этот подход является физически более точным, чем предыдущий, тем не менее мы приводим и его, поскольку считаем, что вы должны знать обо всех вариантах.

Чтобы получить полувектор, нам нужно просто сложить направление взгляда и направление освещения так, как показано в следующем фрагменте кода.

```
float3 halfVector = normalize(lightDir + viewDir);
```

После этого нам нужно вычислить скалярное произведение нормали вершины и этого нового вектора, чтобы получить основное значение Specular компонента. Далее нам нужно просто возвести его в степень `_SpecPower` и умножить на значение цвета из переменной `_SpecularColor`. По сравнению с предыдущим шейдером, эти вычисления гораздо проще и быстрее, но тем не менее дают нам блик, вполне качественный для большинства реальных ситуаций.

Маскирование глянцевых бликов с помощью текстур

Теперь, когда мы узнали, как добавить блик в шейдер, давайте посмотрим, что можно изменить в коде, чтобы предоставить художнику больше возможностей по контролю за тем, как этот блик выглядит. В этом рецепте мы рассмотрим использование текстур для управления бликом и его интенсивностью.

Техника использования текстур для контроля Specular компонента применяется в современных игровых движках, поскольку позволяет 3D-художнику контролировать итоговый эффект на пиксельном уровне. С помощью этой техники мы можем совместить в одном шейдере матовую и блестящую поверхности. Также с помощью этой

техники можно управлять размером блика или его интенсивностью, используя дополнительную текстуру, что позволяет получать поверхности с очень большим бликом или, наоборот, с чётким маленьким бликом.

Существует множество эффектов, которых можно добиться, комбинируя в шейдере данные из текстур и динамические вычисления. И ключевым моментом эффективной работы является возможность художника контролировать самому, как выглядит шейдер на экране. Давайте посмотрим, как с помощью текстур мы можем управлять нашими Specular моделями освещения. В этом рецепте вы познакомитесь с новыми концепциями, такими как создание собственной структуры `Input`, узнаете, как в функцию освещения передаются данные из структуры output, как после этого они передаются в структуру `Input` и в функцию `surf()`. Понимание того, как передаются данные между этими основными элементами поверхностного шейдера, является ключевым для правильной организации работы с шейдерами.

Подготовка

Выполните следующие действия:

1. Нам потребуется сделать новый шейдер, материал и объект, к которому мы применим наш шейдер и материал.
2. После того как вы добавите шейдер и материал объекту на сцене, два раза кликните по шейдеру, для того чтобы открыть его в MonoDevelop.
3. Нам будет нужна текстура бликов. Для наших целей подойдёт любая текстура, содержащая различные цвета и паттерны. На следующем скриншоте представлена текстура, которую мы будем использовать в этом рецепте.

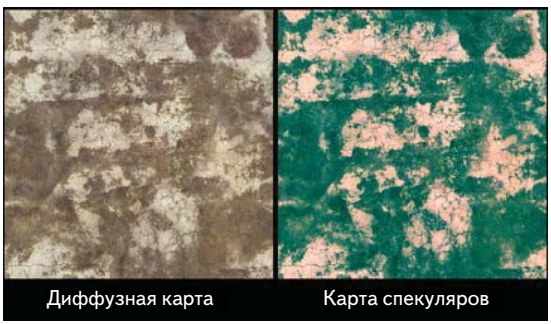

Рис. 3.7. Используемая текстура

Как это сделать...

Выполните следующие действия:

1. Давайте добавим новые переменные в наш блок свойств:

```
Properties
{
  //Объявим свойства, чтобы можно было использовать
  //данные из редактора с панели инспектора.
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _SpecularColor ("Specular Tint", Color) = (1,1,1,1)
  _SpecularMask ("Specular Texture", 2D) = "white" {}
  _SpecPower ("Specular Power", Range(0.1, 120)) = 3
}
```

2. После этого нам потребуется создать соответствующие переменные в блоке `SubShader`, чтобы мы могли получить доступ к данным из блока свойств. Добавьте следующий код после секции `#pragma`.

```
//Получим данные из блока свойств
sampler2D _MainTex;
sampler2D _SpecularMask;
float4 _MainTint;
float4 _SpecularColor;
float _SpecPower;
```

3. Теперь нам нужно будет сформировать собственную структуру `Output`. С её помощью мы сможем хранить и передавать больше данных для функции `surf` и модели освещения. Не волнуйтесь, если что-то не понятно. В следующей секции этого рецепта мы рассмотрим структуру `Output` более подробно. Добавьте следующий код сразу после переменных в блоке `SubShader`.

```
//Создадим нашу выходную структуру.
struct SurfaceCustomOutput
{
  fixed3 Albedo;
  fixed3 Normal;
  fixed3 Emission;
  fixed3 SpecularColor;
  half Specular;
  fixed Gloss;
  fixed Alpha;
};
```

4. Сразу после структуры `Output` мы должны указать используемую модель освещения. В данном случае мы будем пользоваться моделью освещения, которая называется `LightingCustomPhong`. Сразу после структуры `Output` добавьте следующий код.

```
inline fixed4 LightingCustomPhong (SurfaseCustomOutput s,
fixed3 lightDir, half3 viewDir, fixed atten)
{
  //Вычислим диффузный и отражённый векторы
  float diff = dot(s.Normal, lightDir);
  float3 reflectionVector = normalize(2.0 * s.Normal * diff - lightDir);

  //Вычислим Phong блик
  float spec = pow(max(0.0f,dot(reflectionVector, viewDir)),
_SpecPower) * s.Specular;
  float3 finalSpec = s.SpecularColor * spec * _SpecularColor.rgb;

  //Посчитаем итоговый цвет
  fixed4 c;
  c.rgb = (s.Albedo * _LightColor0.rgb * diff) +
(_LightColor0.rgb * finalSpec);
  c.a = s.Alpha;
  return c;
}
```

5. Для того чтобы наша модель освещения заработала, нам нужно указать в блоке `SubShader`, какую модель освещения должен использовать шейдер. Для этого в секции `#pragma` добавьте следующий код.

```
CGPROGRAM
#pragma surface surf CustomPhong
```

6. Поскольку мы собираемся использовать текстуру для изменения значений, использующихся при расчёте блика, нам потребуется ещё одна пара UV-координат, которые будут относиться к этой текстуре. Для этого в структуре `Input` добавьте `uv` перед именем переменной, сопоставленной текстуре. Наберите следующий код сразу после вашей модели освещения.

```
struct Input
{
  //Получим UV-координаты из структуры Input
  float2 uv_MainTex;
  float2 uv_SpecularMask;
};
```

7. И в довершение работы с шейдером нам потребуется заменить функцию `surf()` следующим образом. Так мы сможем пере-

давать информацию о текстуре в нашу функцию модели освещения и использовать данные из пикселей этой текстуры для изменения параметров блика.

```
void surf (Input IN, inout SurfaceCustomOutput o)
{
  //Получим из текстуры информацию о цвете
  float4 c = tex2D(_MainTex, IN.uv_MainTex) * _MainTint;
  float4 specMask = tex2D(_SpecularMask, IN.uv_SpecularMask) *
_SpecularColor;

  //Зададим параметры структуры Output
  o.Albedo = c.rgb;
  o.Specular = specMask.r;
  o.SpecularColor = specMask.rgb;
  o.Alpha = c.a;
}
```

Следующий скриншот демонстрирует результаты маскирования Specular вычислений с помощью информации из текстуры. Теперь у нас есть приятные глазу неоднородности блика по всей поверхности вместо одного общего значения для отражённого света.

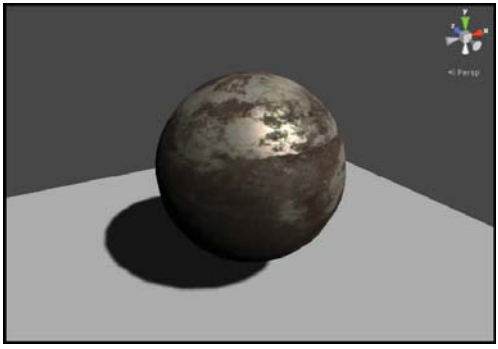

Рис. 3.8. Результат расчёта блика для поверхности

Как это работает...

Вычисления, используемые в этом шейдере, в большой степени похожи на вычисления в шейдере, реализующем эффект Phong, за той лишь разницей, что теперь мы управляем бликом, используя данные из другой текстуры, что добавляет шейдеру эффект глубины и делает его более живым.

Для этого нам нужно передавать информацию из поверхностной функции в функцию освещения. Это необходимо, потому что в функ-

ции освещения мы не можем добраться до UV-координат поверхности. Конечно, мы могли бы процедурно генерировать UV-координаты в функции освещения, однако, если вы захотите распаковать текстуру и получить пиксельную информацию, вам всё равно потребуется использовать структуру `Input`, а единственный способ добраться до структуры `Input` – это использовать функцию `surf()`.

Поэтому, чтобы организовать такую связь данных, нам потребуется определить собственную структуру `SurfaceCustomOutput`. В этой структуре будут содержаться все итоговые данные поверхностного шейдера, и, к счастью для нас, функция освещения и функция `surf()` могут получить доступ к данным в этой структуре. Так как мы создаём нашу собственную структуру, мы можем добавить в неё дополнительные данные. В следующем блоке кода приведена наша новая структура `SurfaceCustomOutput`.

```
//Создадим нашу выходную структуру
struct SurfaceCustomOutput
{
     fixed3 Albedo;
     fixed3 Normal;
     fixed3 Emission;
     fixed3 SpecularColor;
     half Specular;
     fixed Gloss;
     fixed Alpha;
};
```

После того как мы добавим эту структуру в шейдер, нам следует указать функции `surf()` и функции освещения, что они должны использовать эту структуру вместо структуры по умолчанию. Это делается в следующем фрагменте кода.

```
//Создадим нашу выходную структуру
struct SurfaceCustomOutput
{
     fixed3 Albedo;
     fixed3 Normal;
     fixed3 Emission;
     fixed3 SpecularColor;
     half Specular;
     fixed Gloss;
     fixed Alpha;
};

inline fixed4 LightingCustomPhong (SurfaceCustomOutput s, fixed3
lightDir, half3 viewDir, fixed atten)
{
```

```
}

void surf (Input IN, inout SurfaceCustomOutput o)
{
}
ENDCG
```

Обратите внимание, что и функция `surf()`, и функция освещения теперь принимают структуру `SurfaceCustomOutput` в качестве одного из атрибутов. В неё мы добавили еще один элемент, который назвали `SpecularColor`. Таким образом, мы сможем хранить попиксельную информацию из Specular текстуры и использовать её в нашей функции освещения, вместо того чтобы просто перемножать один цвет со всеми значениями блика.

Чтобы передать цвет из нашей текстуры в структуру `SurfaceCustomOutput`, мы используем функцию `tex2D()`, а возвращаемое ею значение записываем в `o.SpecularColor`. После того как эти шаги будут сделаны, мы сможем получить доступ из функции освещения к информации в текстуре.

```
void surf (Input IN, inout SurfaceCustomOutput o}
{
     //Получим из текстур информацию о цвете
     float4 c = tex2D(_MainTex, IN.uv_MainTex) * _MainTint;
     float4 specMask = tex2D(_SpecularMask, IN.uv_SpecularMask) *
_SpecularColor;

     //Зададим параметры в выходной структуре
     o.Albedo = c.rgb;
     o.Specular = specMask.r;
     o.SpecularColor = specMask.rgb;
     o.Alpha = c.a;
}
```

Теперь вы знаете, как получать данные из текстур в функции `surf()` и передавать их в функцию освещения. Этот приём является очень важным для создания сложных шейдерных эффектов.

Металлические и мягкие блики

В этой секции мы рассмотрим способ, с помощью которого мы сможем в одном шейдере получить мягкие и жёсткие блики. Чаще всего вы будете сталкиваться с тем, что для решения большинства задач вам потребуется использовать целый набор шейдеров. А поскольку работать с большим количеством шейдеров может быть чрезвычайно

сложно, обычно программисты пытаются в одном шейдере объединить эффекты металла и ткани. Как он будет выглядеть, зависит от того, как пользователь настроит его свойства. Наша цель в этом рецепте – сделать Specular блик настраиваемым, так чтобы конечный пользователь смог получать мягкий как ткань блеск, а затем с помощью этого же шейдера добиться очень жёсткого металлического блика.

Для достижения такой гибкости мы будем использовать модель освещения, похожую на модель Cook Torrance однако, нашу модель мы слегка доработаем, сделав её чуть более дружественной для художника или того, кто будет использовать этот шейдер.

Подготовка

Выполните следующие шаги:

1. Создайте новую сцену, поставьте на ней сферу, плоскость и источник направленного света. Не забудьте сохранить вашу сцену под новым именем.
2. Создайте новый шейдер и новый материал, придумайте и задайте им имена.
3. И наконец, добавьте шейдер материалу, а материал – сфере на вашей сцене.
4. Кроме того, нам потребуется несколько текстур, которые позволят художнику управлять резкостью блика, определяя, насколько размытым или насколько резким он должен быть. Пример того, как могут выглядеть эти текстуры, приводится на следующем скриншоте.

На этом скриншоте наглядно демонстрируются примеры текстур различной резкости, использованные в этом рецепте.

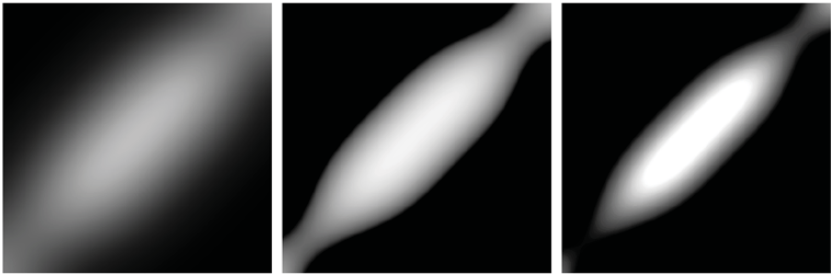

Рис. 3.9. Текстуры различной резкости

Как это сделать...

Выполните следующую последовательность действий:

1. Начнём с самого важного – нам нужно задать свойства, которые мы будем использовать в шейдере. В блок свойств вашего шейдера добавьте следующий код.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _RoughnessTex ("Roughness texture", 2D) = "" {}
  _Roughness ("Roughness", Range(0,1)) = 0.5
  _SpecularColor ("Specular Color", Color) = (1,1,1,1)
  _SpecPower ("Specular Power", Range(0,30)) = 2
  _Fresnel ("Fresnel Value", Range(0,1.0)) = 0.05
}
```

2. После этого нам нужно убедиться, что данные свойств доступны в блоке `SubShader`. Добавьте следующий код в шейдер сразу после секции `#pragma`.

```
sampler2D _MainTex;
sampler2D _RoughnessTex;
float _Roughness;
float _Fresnel;
float _SpecPower;
float4 _MainTint;
float4 _SpecularColor;
```

3. Далее нам нужно объявить нашу новую модель освещения в секции `#pragma`.

```
CGPROGRAM
#pragma surface surf MetallicSoft
#pragma target 3.0

inline fixed4 LightingMetallicSoft (SurfeceOutput s, fixed3
lightDir, half3 viewDir, fixed atten)
{

}
```

4. Теперь мы готовы взяться за функцию модели освещения. Сначала мы сгенерируем все диффузные и зависящие от направления взгляда векторы, поскольку они все нам пригодятся далее.

```
//Вычислим диффузию и направление взгляда
float3 halfVector = normalize(lightDir + viewDir);
float NdotL = saturate(dot(s.Normal, normalize(lightDir)));
```

```
float NdotH_raw = dot(s.Normal, halfVector);
float NdotH = saturate(dot(s.Normal, halfVector));
float NdotV = saturate(dot(s.Normal, normalize(viewDir)));
float VdotH = saturate(dot(halfVector, normalize(viewDir)));
```

5. Следующая секция кода нашего шейдера связана с вычислением значений резкости блика с помощью текстуры для задания формы блика и процедурной генерации небольших неровностей на поверхности объекта. Добавьте следующий код.

```
//Распределение небольших неровностей
float geoEnum =  2.0*NdotH;
float3 G1 = (GeoEnum * NdotV)/ NdotH;
float3 G2 = (geoEnum * NdotL)/ NdotH;
float3 G = min(1.0f, min(G1, G2));

//Достаём цвет из BRDF-текстуры
float roughness = tex2D(_RoughnessTex, float2(NdotH_raw * 0.5
+ 0.5, _Roughness)).r;
```

6. И последний элемент, который понадобится нашему шейдеру, – это вычисление эффекта Френеля, с помощью которого мы сможем маскировать блик, если камера смотрит на объект под слишком острым углом.

```
//Вычислим значение fresnel
float fresnel = pow(1.0 - VdotH, 5.0);
fresnel *= (1.0 - _Fresnel);
fresnel += _Fresnel;
```

7. Теперь, когда у нас есть все компоненты, необходимые для блика, нам нужно их объединить воедино для генерации итогового отражённого значения.

```
//Сформируем итоговый блик
float3 spec = float3(fresnel * G * roughness * roughness) *
_SpecPower;
```

8. Для завершения модели освещения нам просто нужно объединить диффузную и Specular-компоненты.

```
float4 c;
c.rgb = (s.Albedo * _LightColor0.rgb * NdotL)+ (spec *
_SpecularColor.rgb) * (atten * 2.0f);
c.a = s.Alpha;
return  c;
```

После того как вы добавите в ваш шейдер весь этот код, вернитесь в редактор Unity, чтобы шейдер скомпилировался. Если не было до-

пущено ошибок, вы увидите результат, похожий на изображённый на следующем скриншоте.

Рис. 3.10. Результаты работы шейдера

Как это работает...

Возможно, этот шейдер вым покажется гораздо сложнее, чем предыдущие. Но не волнуйтесь, на самом деле всё довольно просто. Вы даже можете попробовать визуально проследить ход выполнения шейдера, устанавливая `c.rgb` в промежуточное `float3` значение на каждом этапе. Таким образом, прямо в редакторе вы будете видеть данные из того шага алгоритма, после которого вы устанавливаете цвет. Этот трюк полезно помнить при отладке шейдеров.

Если мы попробуем сделать это с первым блоком кода, в котором мы выполняем расчёт диффузных и зависящих от направления взгляда векторов, то мы увидим картину, очень похожую на следующий скриншот.

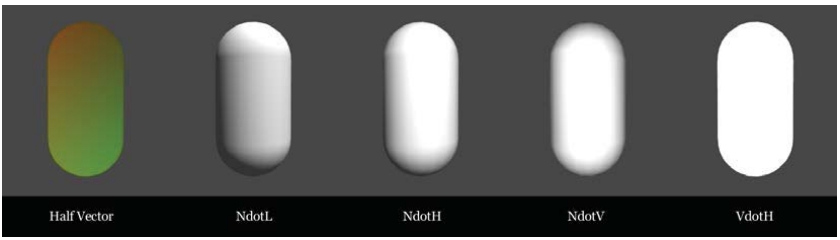

Рис. 3.11. Векторы в режиме отладки

Как только мы получим все нужные данные, мы можем начать работать с ними, почти так же, как со слоями в Photoshop. Начнём мы этот процесс с того, что сгенерируем необходимые значения для симуляции маленьких неровностей и переотражающегося в них света.

```
//Распределение небольших неровностей
float geoEnum = 2.0*NdotH;
float3 G1 = (geoEnum * HdotV)/ NdotH;
float3 G2 = (geoEnum * NdotL)/ NdotH;
float3 G = min(1.0f, min(G1, G2));
```

Одним из ключевых моментов этой модели освещения является возможность контролировать ширину блика или его резкость с помощью текстуры с запеченной Specular-функцией. Таким образом, мы можем процедурно генерировать UV-координаты и выбирать участок текстуры для использования при расчёте нашего блика. Для этих целей мы используем значение `NdotH` – скалярное произведение полувектора и нормали вершины, подставляя его в `float2` переменную в функции `tex2D()`. Эта переменная станет нашими UV-координатами для сэмплинга текстуры. Значение `y` в этой переменной – свойство из панели **Инспектора**. Таким образом пользователь может расширять или сужать ширину блика.

```
//Достаём цвет из BRDF-текстуры
float roughness = tex2D(_RoughnessTex, float2 (NdotH_raw * 0.5 +
0.5, _Roughness)).r;
```

Далее нам нужно добавить эффект Френеля, чтобы при взгляде в направлении, противоположном направлению света, мы получали эффект увеличения интенсивности блика.

```
//Сформируем значение fresnel
float fresnel = pow(1.0 - VdotH, 5.0);
fresnel *= (1.0 - _Fresnel);
fresnel += Fresnel;
```

После того как мы вычислили все эти отдельные части, для того чтобы получить итоговое значение Specular-компонента, нужно просто их перемножить. В данном случае для контроля интенсивности блика мы добавили ещё один множитель – свойство `_SpecPower`.

```
//Сформируем итоговый блик
float3 spec = float3(fresnel * G * roughness * roughness) *
_SpecPower;
```

И последнее, чтобы получить итоговый цвет поверхности, мы комбинируем Specular-компонент с диффузным компонентом. Мы надеемся, что вы видите масштаб изменений, которые можно внести в простую систему, всего лишь за счёт использования других типов векторов и текстур.

Дополнительная информация

Чтобы узнать больше о световой модели Cook Torrance, посетите следующие ссылки:

- http://en.wikipedia.org/wiki/Specular_highlight#Cook.E2.80.93Torrance_model;
- http://content.gpwiki.org/index.php/D3DBook:%28Lighting%29_Cook-Torrance;
- http://forum.unity3d.com/threads/158589-Cook-Torrance.

Создание анизотропных бликов

Анизотропные блики – тип бликов или отражений, которые симулируют направленность царапин на поверхности и соответствующим образом изменяются/растягиваются перпендикулярно направлению царапин. Этот приём окажется очень полезен, когда вам потребуется смоделировать потёртый металл, а не обычный металл с гладкой, полированной и блестящей поверхностью. Вспомните, как выглядит CD- или DVD-диск со стороны данных или как выглядит нижняя часть новенькой кастрюли либо сковородки. При внимательном рассмотрении поверхности вы заметите, что царапины на поверхности направлены в одну сторону – как правило, в сторону стачивания металла. При применении Specular освещения к такой поверхности вы получите отражённый свет, вытянутый перпендикулярно направлению царапин.

В этом рецепте мы рассмотрим, как дополнить расчет блика, чтобы добиться эффекта различных потёртых поверхностей. В следующих рецептах мы увидим, как эту технику можно модифицировать для получения других эффектов, например таких как вытянутые отражения на волосах, но не будем забегать вперёд – сперва изучим основы. Мы будем использовать следующий шейдер в качестве образца анизотропного шейдера (http://wiki.unity3d.com/index.php?title=Anisotropic_Highlight_Shader).

На скриншоте, показанном на рис. 3.12, приводятся различные типы бликов, которые можно получить в Unity с помощью анизотропного шейдера.

Подготовка

Выполните следующие действия:

1. Создайте новую сцену с объектами и источниками освещения, чтобы мы могли видеть эффект нашего шейдера.

2. Создайте новый шейдер и материал, соедините их с нашими объектами.
3. Кроме того, нам понадобится карта нормалей, которую мы будем использовать для задания направленности нашего анизотропного блика.

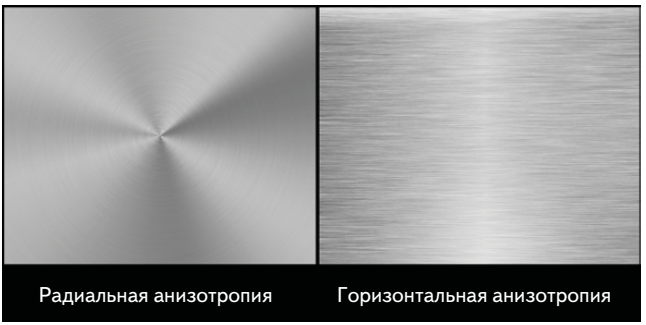

Рис. 3.12. Результаты применения анизотропных шейдеров

На скриншоте, приведенном на рис. 3.13, представлена карта нормалей, которую мы будем использовать для этого рецепта. Найти её можно на странице книги по следующему адресу www.packtpub.com/support.

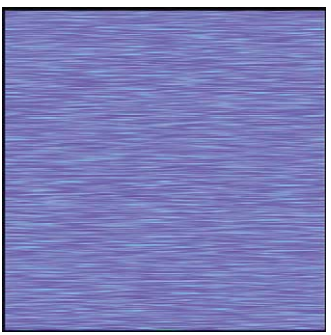

Рис. 3.13. Анизотропная карта нормалей

Как это сделать...

Выполните следующие действия:

1. Сперва нам нужно добавить свойства, которые мы хотим задействовать в шейдере. С их помощью мы получим возможность детально контролировать итоговый вид поверхности.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _SpecularColor ("Specular Color", Color) = (1,1,1,1)
  _Specular ("Specular Amount", Range(0,1)) = 0.5
  _SpecPower ("Specular Power", Range(0,1)) = 0.5
  _AnisoDir ("Anisotropic Direction", 2D) = "" {}
  _AnisoOffset ("Anisotropic Offset", Range(-1,1)) = -0.2
}
```

2. После этого мы должны настроить связь между блоком свойств и кодом, чтобы можно было использовать данные из блока свойств.

```
sampler2D _MainTex;
sampler2D _AnisoDir;
float4 _MainTint;
float4 _SpecularColor;
float _AnisoOffset;
float _Specular;
float _SpecPower;
```

3. На этом шаге мы добавим нашу функцию освещения, которая будет создавать правильный анизотропный эффект для нашей поверхности.

```
inline fixed4 LigntingAnisotropic (SurfaceAnisoOutput s,
fixed3 lightDir, half3 viewDir, fixed atten)
{
  fixed3 halfVector = normalize(normalize(lightDir) +
normalize (viewDir));
  float NdotL = saturate(dot(s.Normal, lightDir));

  fixed HdotA = dot(normalize(s.Normal + s.AnisoDirection),
halfVector);
  float aniso = max(0, sin(radians((HdotA + _AnisoOffset) * 180f)));

  float spec = saturate(pow(aniso, s.Gloss * 128) *
s.Specular);

  fixed4 c;
  c.rgb = ((s.Albedo * _LightColorO.rbg * NdotL) +
(_LightColor0.rgb * _SpecularColor.rgb * spec)) * (atten * 2);
  c.a = 1.0:
  return c;
}
```

4. Чтобы вместо стандартной функции освещения мы могли воспользоваться нашей функцией, мы должны соответственным

образом модифицировать директиву `#pragma`. Кроме того, дадим инструкцию шейдеру использовать в качестве целевой шейдерную модель версии 3.0, что нам даст больше места под текстуры.

```
CGPROGRAM
#pragma surface surf Anisotropic
#pragma target 3.0
```

5. Также мы определим отдельные UV-координаты для анизотропной карты нормалей, добавив следующий код в структуру `Input`. Строго говоря, этот шаг не является обязательным, поскольку мы могли бы использовать UV-координаты из основной текстуры, но при таком подходе мы сможем добиться независимого контроля за размещением эффекта потёртого метала, а значит, мы сможем масштабировать его как захотим.

```
struct Input
{
  float2 uv_MainTex;
  float2 uv_AnisoDir;
}
```

6. И конечно, нам потребуется использовать функцию `surf()`, чтобы передавать корректные данные нашей функции освещения. Мы получаем данные из анизотропной карты нормалей и передаем их дальше в структуре `SurfaceAnisoOutput`.

```
void surf (Input IN, inout SurfaceAnisoOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex) * _MainTint;
  float3 anisoTex = UnpackNormal(tex2D(_AnisoDir, IN.uv_
AnisoDir));

  o.AnisoDirection = anisoTex;
  o.Specular = _Specular;
  o.Gloss = _SpecPower;
  o.Albedo = c.rgb;
  o.Alpha = c.a;
}
```

Следующий скриншот демонстрирует результаты работы нашего анизотропного шейдера. Анизотропная карта нормалей позволяет нам задать направление поверхности и распределить блик по поверхности объекта.

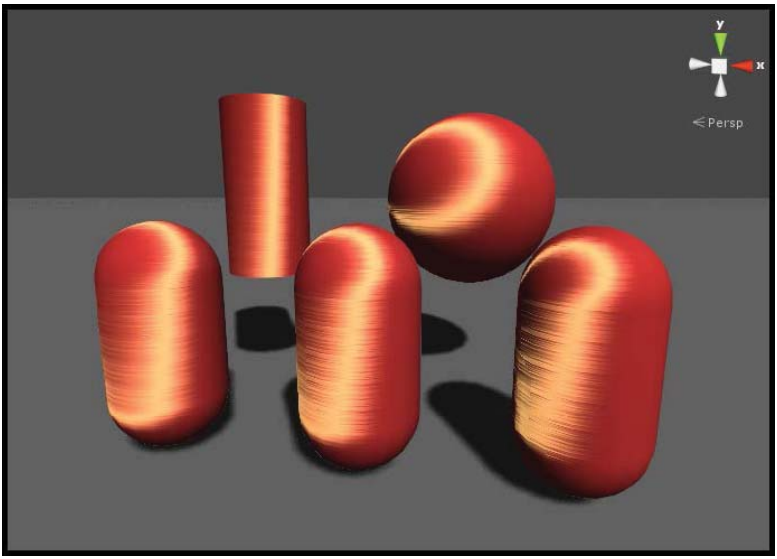

Рис. 3.14. Результат применения анизотропной карты нормалей

Как это работает...

Давайте рассмотрим наш шейдер по частям и объясним, почему мы получаем именно такие эффекты. При этом мы сосредоточимся в основном на функции освещения, поскольку остальные части шейдера на данный момент уже должны быть очевидны.

Начинаем мы с того, что объявляем нашу структуру `SurfaceAnisoOutput`. Делаем мы это потому, что нам нужно получить попиксельную информацию из анизотропной карты нормалей, а единственный способ, которым этого можно добиться в поверхностном шейдере, – это использование функции `tex2D()` в функции `surf()`.

```
struct SurfaceAnisoOutput
{
     fixed3 Albedo;
     fixed3 Normal;
     fixed3 Emission;
     fixed3 AnisoDirection;
     half Specular;
     fixed Gloss;
     fixed Alpha;
};
```

Мы можем использовать структуру `SurfaceAnisoOutput` как посредник между функцией освещения и функцией поверхности. В нашем случае в функции `surf()` мы передаем информацию из текстуры в структуру `SurfaceAnisoOutput`, сохраняя её в переменную `AnisoDirection`. После этого мы можем использовать эту информацию в функции освещения, обращаясь к `s.AnisoDirection`.

А теперь, когда мы можем обмениваться данными между этими функциями, мы можем перейти непосредственно к нашим расчётам освещения. Начинается расчёт с того, что мы, чтобы не делать полного расчёта отражённого света, вычисляем полувектор. Далее находим диффузный компонент света, то есть скалярное произведение нормали к точке и направление света.

```
fixed3 halfVector = normalize(normalize(lightDir) +
normalize(viewDir));
float NdotL = saturate(dot(s.Normal, lightDir));
```

После этого мы начинаем изменять отражение для придания ему правильного вида. Первым делом мы рассчитываем скалярное произведение нормализованной суммы нормали вершины и векторов из нашей анизотропной карты нормалей с рассчитанным на предыдущем этапе значением `halfVector`. В результате чего мы получаем значение, которое будет равно 1, если нормаль вершины (измененная анизотропной картой нормалей) параллельна полувектору, и 0, если перпендикулярна. После этого, для того чтобы получить более тёмный блик посередине и в конечном итоге эффект колец, основанный на полувекторе, мы изменяем это значение с помощью функции `sin()`.

```
fixed HdotA = dot(normalize(s.Normal + s.AnisoDirection),
halfVector);
float aniso = max(0, sin(radians((HdotA + _AnisoOffset) *
180)));
```

Далее мы масштабируем эффект от применения переменной `aniso`, возводя её в степень `s.Gloss`, а затем мы глобально уменьшаем его силу, умножая на `s.Specular`.

```
float spec = saturate(pow(aniso, s.Gloss * 128) * s.Specular);
```

Этот эффект замечательно подходит для создания более сложных типов металлических поверхностей, особенно таких, которые содержат потёртости и визуально заметные направления на поверхности. Также этот эффект хорошо подходит для имитации волос или любой другой мягкой поверхности, содержащей направленную структуру.

На следующем скриншоте приводится итоговый результат расчёта анизотропного освещения.

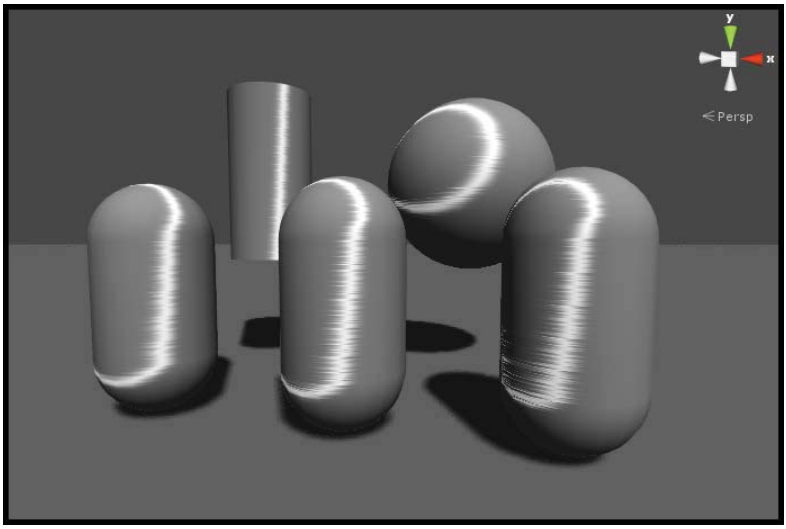

Рис. 3.15. Итоговый результат расчёта анизотропного освещения

ГЛАВА 4
Добавим отражения в ваш мир

Real-time-отражения поднимают качество графики в игре на новый уровень. Для того чтобы получить отражающую поверхность, шейдер материала этой поверхности использует информацию об окружающем мире и симулирует отражение объектов. В основном для создания этого эффекта используется специальный тип текстур – **кубические карты**, или **кубмапы** (cubemap). Эти текстуры состоят из шести плоских текстур, которые выстраиваются вокруг поверхности в форме куба (по одной на каждую грань) и позволяют сэмплировать цвет по направлению в трёхмерном пространстве вместо двумерных UV-координат. Таким образом, мы можем запечь снимок окружающего пространства в один такой кубмап.

В этом рецепте мы рассмотрим, как генерировать кубмапы на основе объектов окружающего мира и как потом использовать полученные кубмапы в шейдерах для создания эффектов отражения. Эта техника замечательно подходит для имитации металла, автомобильной краски и даже пластика. В этой главе вы узнаете о том, как:

- создать кубмап в редакторе Unity;
- использовать кубмап для простых отражений;
- маскировать отражения;
- использовать карты нормалей в отражениях;
- реализовать эффект Френеля;
- написать простую динамическую систему кубических текстур в Unity.

Создание кубических текстур в Unity

Прежде чем мы займёмся изучением того, как создаются шейдеры отражений, нам потребуется научиться создавать кубмапы. Конеч-

но, вы можете найти их в Интернете, но рано или поздно вы захотите научиться создавать свои собственные, хотя бы потому, что готовые статические кубмапы не содержат деталей именно вашего уникального игрового мира. Кубические текстуры, выложенные в Интернете, хорошо подходят лишь для тестирования шейдера, и не более. В программировании графики создание своих кубических текстур, в которые запечатлён именно ваш виртуальный мир, – это ключевой момент для обеспечения реалистичности эффектов отражения. В первом рецепте этой главы мы рассмотрим, как в редакторе Unity сгенерировать кубмап из заданной точки пространства. В конце мы дадим ссылки на сторонние приложения, с помощью которых можно создавать кубмапы. Вооружившись этими знаниями, вы сможете перейти к следующим частям этой главы, поскольку создание кубических текстур и понимание принципов их работы, – это необходимый минимум для выполнения последующих рецептов.

Подготовка

В Unity есть пример на JavaScript, в котором показано, как генерировать кубмап из окружающих объектов. Давайте посмотрим, как это делается. Описание скрипта доступно по следующей ссылке: www.packtpub.com/support. Мы возьмем этот скрипт за основу и перепишем его на C#. А в последнем рецепте этой главы мы рассмотрим, как можно сделать простую систему генерации кубических текстур в нескольких позициях, которую мы будем использовать для того, чтобы менять полученные кубмапы по мере продвижения нашего персонажа по уровню, что даст нам почти real-time-систему отражений.

В этом рецепте мы сфокусируемся на том, как сгенерировать кубмап, что подготовит нас к созданию системы динамических отражений для вашей игры.

Выполните следующие действия:

1. Нам нужно будет добавить объекты на сцену, которые на кубмапе будут выполнять роль источников сета. Поэтому в нашей сцене нам потребуется создать несколько плоскостей. Вы можете это сделать в любом 3D-редакторе, например Maya или Max, или же вы можете использовать стандартную плоскость (Plane) Unity. Подойдёт любой вариант. В конце концов, ваша сцена должна выглядеть примерно следующим образом.

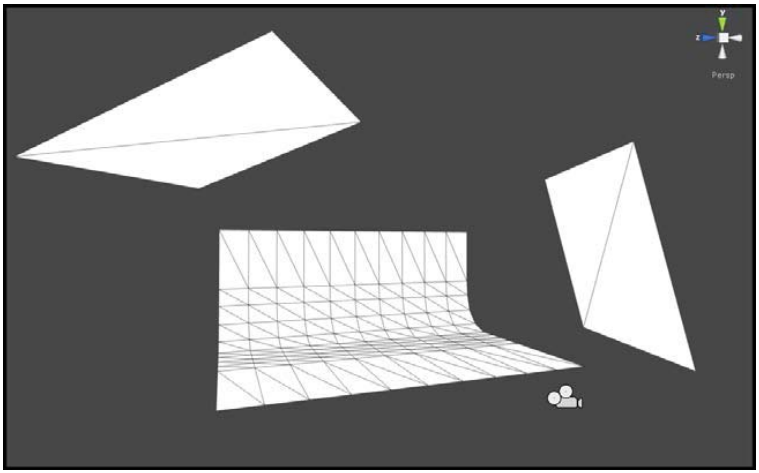

Рис. 4.1. Вид сцены с созданными плоскостями

2. Теперь нам нужно создать текстуры, которые будут имитировать эффекты от разных типов освещения. Поэтому нам понадобятся текстуры, имитирующие затухание и интенсивность источников света нашего окружающего мира. Взгляните на следующий скриншот.

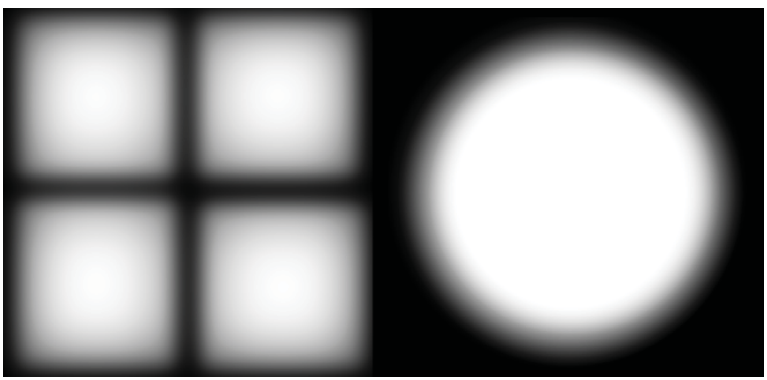

Рис. 4.2. Текстуры для имитации эффектов разных типов освещения

3. Далее, для того чтобы мы смогли использовать наши плоскости и текстуры в качестве источников света в кубмапе, мы применим к ним один из встроенных в Unity шейдеров. В данном случае Unlit/Transparent (Неосвещённый/Прозрачный) шейдер, чтобы текстуры, симулирующие источники света, не теря-

ли яркости. В готовом варианте ваша сцена должна выглядеть следующим образом.

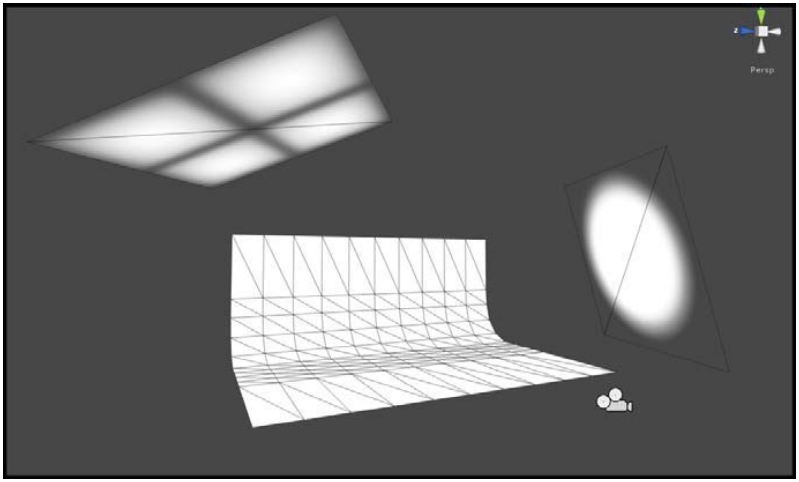

Рис. 4.3. Итоговый вид подготовленной сцены

Как это сделать...

Для написания нашего шейдера выполните следующие шаги:

1. Сперва нам потребуется создать новый скрипт, а так как мы планируем сделать кастомное всплывающее окно в редакторе, нам потребуется сохранить скрипт в папке `Editor`. Создайте эту папку в панели проекта, затем создайте в ней C#-скрипт и назовите его `GenerateStaticCubemap`. После того как вы создадите скрипт, два раза кликните по нему, чтобы запустить его в MonoDevelop.

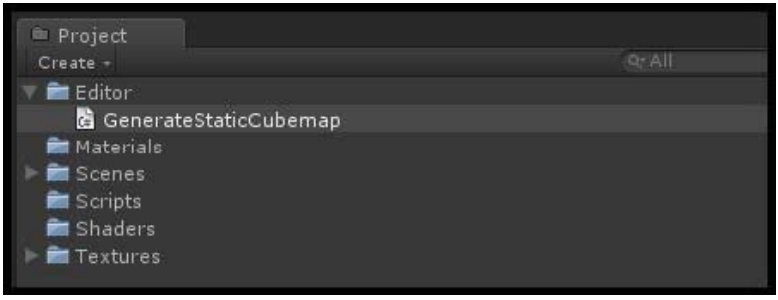

Рис. 4.4. Расположение файла скрипта в дереве проекта

2. Когда наш скрипт откроется в MonoDevelop, мы должны будем его отредактировать, чтобы он выполнял требуемые функции. Для начала нам потребуется добавить директиву `using`, в которой указать, что мы собираемся использовать неймспейс (пространство имён) `UnityEditor`.

```
using UnityEngine;
using UnityEditor;
using System.Collections;
```

3. Для того, чтобы скрипт при выполнении в редакторе открывался во всплывающем окне, нам нужно унаследовать класс `GenerateStaticCubemap` от класса `ScriptableWizard`. Таким образом, у нас появится доступ к некоторым полезным функциям.

```
public class GenerateStaticCubemap : ScriptableWizard
```

4. Также нам нужно будет добавить несколько публичных переменных, в которых мы будем хранить наш кубмап и координаты точки, из которой он был снят. Добавьте следующий код в начале класса.

```
public Transform renderPosition;
public Cubemap cubemap;
```

5. Первая функция в нашем скрипте – это встроенная функция `OnWizardUpdate()`. Она вызывается, когда наше всплывающее окно (wizard) открывается, или когда пользователь взаимодействует с элементами GUI в нем. Поэтому в этой функции мы будем проверять, что пользователь ввёл правильные данные в визард: если пользователь не указал кубмап или ссылку на transform, то мы установим значение булевой переменной `isValid` в `false` и запретим дальнейшие действия.

```
void OnWizardUpdate()
{
  helpString = "Select transform to render " +
          "from and cubemap to render into";
  if (renderPosition != null && cubemap != null)
  {
          isValid = true;
  }
  else
  {
          isValid = false;
  }
}
```

6. Если переменная `isValid` равна true, визард вызовет функцию `OnWizardCreate()`, где мы создадим новую камеру, расположим её там же, где находится предоставленный transform, и вызовем функцию `RenderToCubemap()`, чтобы получить новую кубическую карту.

```
void OnWizardCreate()
{
  //Создадим временную камеру для рендеринга
  GameObject go = new GameObject("CubeCam",
typeof(Camera));

  //Разместим её в нужном месте
  go.transform.position = renderPosition.position;
  go.transform.rotation = Quaternion.identity;

  //Отрендерим кубмап
  go.camera.RenderToCubemap(cubemap);

  //Уничтожим временную камеру
  DestroyImmediate(go);
}
```

7. И наконец, нам потребуется сделать так, чтобы это всплывающее окно открывалось при выборе определенного пункта меню в редакторе Unity. Добавьте следующий код в класс `GenerateStaticCubemap`.

```
[MenuItem("CookBook/Render Cubemap")]
static void RenderCubemap()
{
  ScriptableWizard.DisplayWizard("Render CubeMap",
          typeof(GenerateStaticCubemap), "Render!");
}
```

Как это работает...

В начале нашего скрипта мы объявляем, что он наследуется от класса `ScriptableWizard`. Тем самым мы говорим Unity, что мы создаём новый тип всплывающего окна в редакторе. Поэтому-то мы и поместили наш скрипт в папку `Editor`. Если бы мы поместили его в другое место, то Unity бы не распознала этот скрипт как выполняющийся в редакторе.

С помощью переменных, объявляемых нами на следующем этапе, мы получаем возможность хранить положение, из которого мы хотим отрендерить кубмап, и ссылку на объект кубмапа в панели проекта,

в который мы хотим этот кубмап сохранить. Выставив правильные значения этих переменных, мы сможем сгенерировать наш кубмап.

Далее мы редактируем функцию `OnWizardUpdate()`, которую нам предоставляет класс `ScriptableWizard`. Вызывается эта функция при первом запуске визарда, а также при любом взаимодействии пользователя с элементами GUI визарда. Мы будем использовать эту функцию для проверки того, что пользователь действительно установил правильные значения переменных. Если да, то мы устанавливаем переменную `isValid` в `true`, если нет, то в `false`. Переменная `isValid` – это встроенная переменная класса `ScriptableWizard`. Всё, что она делает, – это позволяет вам включать и выключать кнопку **Create** внизу окна визарда. Так, вы можете быть уверены, что никто не сможет запустить следующую функцию визарда без установленных трансформа и кубмапа.

После того как мы проверили, что пользователь предоставил нам корректные данные, мы можем перейти к функции `OnWizardCreate()`. Именно здесь и создается новый кубмап. Начинается всё с того что мы создаем новый `GameObject` с компонентом `Camera` на нём. Далее, мы выставляем его в координаты предоставленного в параметрах визарда трансформа.

После выполнения этого кода у нас появится новая камера на заданной позиции. Всё, что нам останется сделать после этого, – это вызвать функцию `RenderToCubemap()` и передать ей кубмап, который пользователь выбрал в параметрах визарда. После выполнения этой функции шесть изображений для нашего кубмапа будут созданы и объединены в заданный пользователем объект кубической карты.

Далее для нашего визарда мы добавляем опцию в меню, чтобы пользователь мог вызвать его из главного меню Unity. При нажатии на этот пункт меню будет вызвана статическая функция визарда, которая отобразит его на экране. Таким образом, мы написали небольшой инструмент для создания кубических карт непосредственно в редакторе Unity.

Дополнительная информация

Мы приводим две программы, с помощью которых вы можете генерировать кубмапы и использовать их в своих шейдерах:

- **ATI CubeMapGen:** http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/cubemapgen/;
- **HDR Light StudioPro:** http://www.hdrlightstudio.com/.

Простое отражение с использованием кубической текстуры

Теперь, когда мы умеем создавать наши собственные кубмапы, мы можем перейти к использованию этого нового типа текстур для имитации отражений в нашем шейдере. В основе использования кубических текстур для создания отражений лежит весьма простая идея, но при этом она предоставляет очень мощный инструмент для создания шейдерных эффектов. Мы берём нормаль точки поверхности модели и используем цвет пикселя в кубмапе в этом направлении при генерации отражения на поверхности объекта. Таков, в общих чертах, принцип работы этого метода.

В этом рецепте мы сделаем первые шаги к использованию кубических карт для имитации отражений. Unity может сама вычислять вектор отражения, что избавит нас от необходимости его расчёта вручную. Для этого в структуре `Input` нужно использовать вектор `worldRefl`. Мы будем использовать этот вектор для сэмплинга кубической текстуры. Что даст нам в итоге самый простой способ создания эффекта отражения в поверхностном шейдере.

Подготовка

Прежде чем мы начнём писать код шейдера, нам потребуется подготовить сцену. Для этого выполните следующие действия:

1. Создайте новую сцену, материал и шейдер. Не забудьте присвоить им имена, с помощью которых их легко можно будет найти.
2. Добавьте шейдер материалу, а материал назначьте объекту.
3. Создайте или найдите кубмап, который мы будем использовать в нашем шейдере.

На скриншоте (рис. 4.5) показан кубмап, который мы использовали для этого рецепта. Несмотря на то что у вас будет другой кубмап, мы приводим этот скриншот, чтобы показать, какие текстуры используем мы, и избежать возможных недопониманий.

Как это сделать...

Начните написание вашего шейдера с выполнения следующих действий:

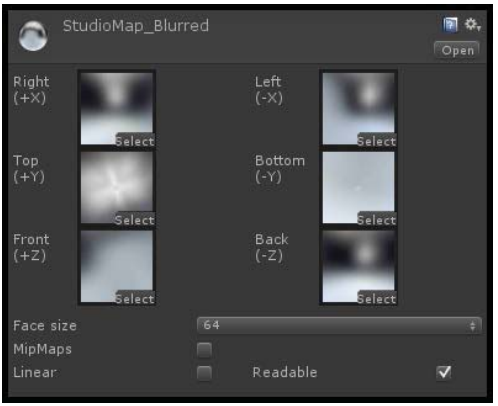

Рис. 4.5. Используемая в рецепте кубическая текстура

1. В блоке свойств создайте новые свойства. Нам понадобится свойство, чтобы хранить нашу кубическую текстуру, и свойство, чтобы управлять величиной отражения.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _Cubemap ("CubeMap", CUBE) = ""{}
  _ReflAmount ("Reflection Amount", Range(0.01, 1)) = 0.5
}
```

2. После этого нам нужно создать связь между блоком свойств и переменными в коде. Это позволит нам получать данные из блока свойств.

```
sampler2D _MainTex;
samplerCUBE _Cubemap;
float4 _MainTint;
float _ReflAmount;
```

3. Чтобы корректно имитировать угол отражения от поверхности, нам потребуется некий вектор с правильным направлением отражения. Для этого мы воспользуемся ещё одним механизмом, встроенным в Unity. Если мы поместим внутрь нашей структуры `Input` следующий код, то получим вектор отражения объектов окружающего мира, пригодный для использования в нашем шейдере.

```
struct Input
{
```

```
  float2 uv_MainTex;
  float3 worldRefl;
};
```

4. После этого мы сможем получить цвет из нашего кубмапа с помощью функции `texCUBE()` и вектора отражения объектов окружающего мира из структуры `Input`. Добавьте в функцию `surf()` следующий код.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex) * _MainTint;
  o.Emission = texCUBE(_Cubemap, IN.worldRefl).rgb *
               _ReflAmount;
  o.Albedo = c.rgb;
  o.Alpha = c.a;
}
```

Результат использования нашего кубмапа вы можете увидеть на следующем скриншоте.

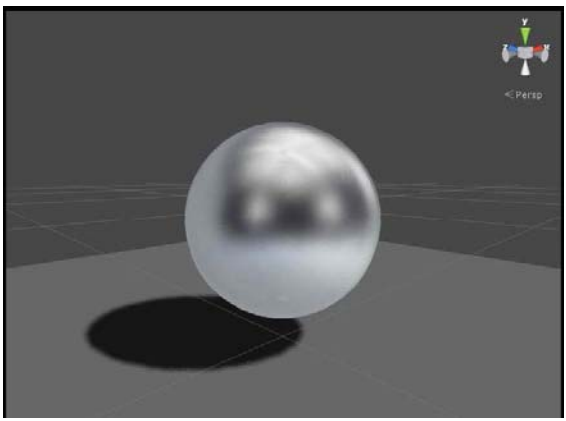

Рис. 4.6. Результат работы шейдера, обрабатывающего нашу кубическую текстуру

Как это работает...

Если вы всё написали без ошибок, то вы увидите, что кубмап отражается от вашего объекта так, как отражались бы настоящие объекты. Это возможно благодаря встроенному в поверхностные шейдеры в Unity свойству структуры `Input`. С помощью свойства `worldRefl` мы можем получить вектор отражения, необходимый для правильной работы с кубической картой. Всего лишь за счёт использования вектора

`worldRefl` внутри функции `texCUBE()` мы можем достать корректное отражение из нашего кубмапа.

На следующем скриншоте показано, как выглядят обрабатываемые шейдером данные отражения во время выполнения отладочного скрипта, который показывает направление нормалей в точках.

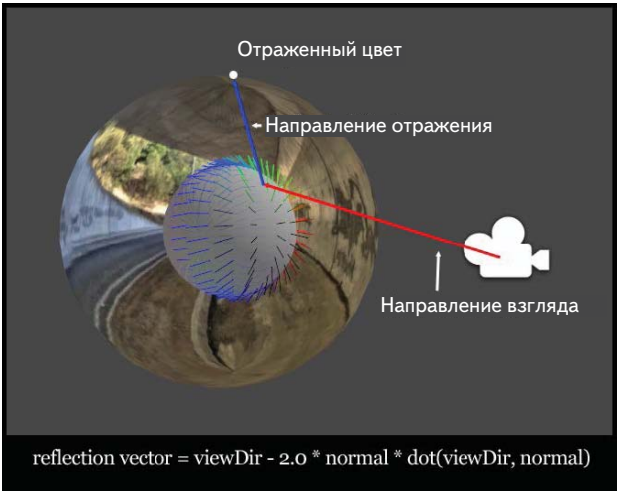

Рис. 4.7. Иллюстрация эффекта отражения в режиме отладки

Маскирование отражений

В предыдущем рецепте мы научились делать зеркальные сферы, теперь мы попробуем сделать более сложную отражающую поверхность. Поскольку практически все объекты в той или иной степени отражают своё окружение, нам будет нужен некий инструмент попиксельного контроля за эффектом отражения.

В этом рецепте мы рассмотрим технику, которая позволит нам управлять отражением с помощью текстур, используемых в качестве маски. Мы будем использовать черно-белые текстуры, для того чтобы задавать коэффициент отражения на поверхности, условившись, что чёрный цвет в текстуре будет соответствовать поверхности без отражения, а белый – полностью зеркальной поверхности. С помощью такого подхода художник, использующий ваш шейдер, сможет лучше контролировать получившийся эффект. Давайте посмотрим, как этот подход реализуется в поверхностных шейдерах Unity.

Подготовка

Давайте подготовим новую сцену для нашего шейдера, реализующего маскированное отражение. Для этого выполните следующие действия.

1. Нам понадобится кубмап, который вы можете сгенерировать, или же воспользуйтесь кубмапом из предыдущего рецепта. Ниже приводится кубмап, который мы использовали в этом рецепте, найти его можно в примерах кода, поставляемых вместе с книгой.

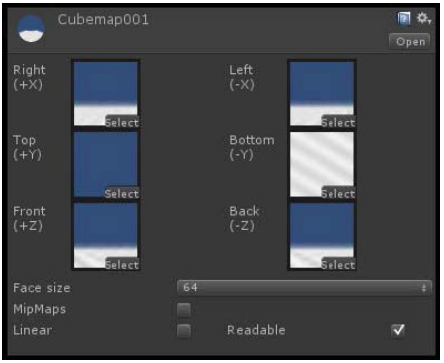

Рис. 4.8. Кубическая текстура, используемая для этого рецепта

2. Кроме того, нам потребуется текстура, содержащая распределение отражающих и неотражающих участков. Не забудьте, что мы условились о том, что чёрный цвет соответствует неотражающей поверхности, белый цвет соответствует полному отражению, а оттенки серого соответствуют неким промежуточным значениям между этими двумя крайностями. Ниже приводится текстура, которую мы использовали в этом рецепте.

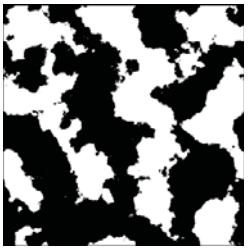

Рис. 4.9. Текстура для маскирования отражения

3. После того как мы создали сцену с объектом, землю и направленный источник света, мы можем взглянуть на наш шейдер во всём его отражающем великолепии.

Как это сделать...

Теперь, когда наша сцена подготовлена, мы можем перейти к написанию кода, который-то и создаст эффект отражения. Выполните следующие действия:

1. Добавьте в ваш шейдер следующие свойства. С их помощью мы сможем назначить в шейдере кубмап и маску отражений.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _ReflAmount ("Reflection Amount", Range(0, 1)) = 1
  _Cubemap ("Cubemap", CUBE) = ""{}
  _ReflMask ("Reflection Mask", 2D) = ""{}
}
```

2. После этого мы добавим в код переменные с такими же названиями.

```
sampler2D _MainTex;
sampler2D _ReflMask;
samplerCUBE _Cubemap;
float4 _MainTint;
float _ReflAmount;
```

3. Чтобы отражение из кубической карты работало корректно, нам потребуется в структуре `Input` объявить свойство `worldRefl`. Направление из него мы будем подставлять в функцию `texCUBE()`, чтобы получить цвет окружения в этом направлении.

```
struct Input
{
  float2 uv_MainTex;
  float3 worldRefl;
};
```

4. Кроме того, нам потребуется добавить в нашу функцию `surf()` следующий код. Мы подробнее остановимся на нем в следующей секции рецепта.

```
void surf(Input IN, inout SurfaceOutput o)
{
```

```
    half4 c = tex2D(_MainTex, IN.uv_MainTex);
    float3 reflection = texCUBE(_Cubemap,IN.worldRefl).rgb;
    float4 reflMask = tex2D(_ReflMask, IN.uv_MainTex);

    o.Albedo = c.rgb * _MainTint;
    o.Emission = (reflection * reflMask.r) * _ReflAmount;
    o.Alpha = c.a;
}
```

На следующем скриншоте приводится результат маскирования отражённого света с помощью текстуры в поверхностном шейдере Unity.

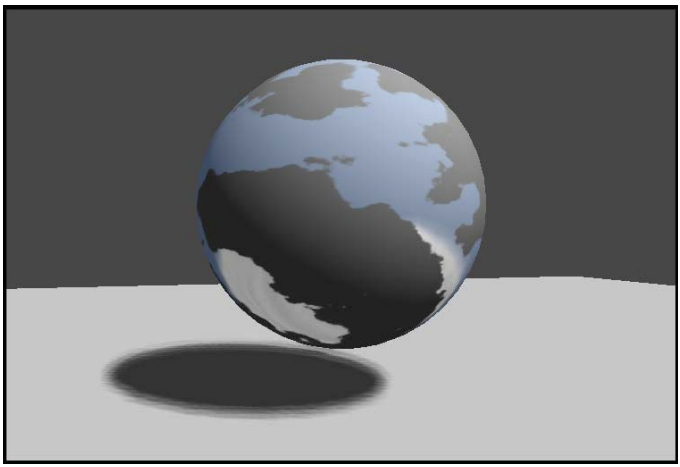

Рис. 4.10. Результат маскирования отражённого света с помощью текстуры

Как это работает...

В начале своей работы шейдер просто достает значение цвета из кубмапа с помощью функции `texCUBE()`. Эта функция встроена в язык CGFX, она возвращает цвет текстуры в заданном направлении, который мы можем использовать в нашем шейдере. Unity помогает нам в этом деле, предоставляя свойство `worldRefl` в структуре `Input`. Как объяснялось в последнем рецепте – это свойство передаёт нам вектор отражения от направления камеры.

После того как мы получим значение отражения в точке, нам нужно будет получить данные из текстуры маски. Это делается с помощью встроенной функции `tex2D()`, применение которой мы уже рассматривали в главе 2 «Создание эффектов с помощью текстур».

Когда данные из обеих текстур получены и сохранены в переменных в нашей функции `surf()`, мы просто перемножаем цвета кубмапа с цветами маски и сохраняем полученный результат в параметр `o.Emission` выходной структуры нашей поверхности. А потом, для того чтобы у нас была возможность глобально контролировать общую интенсивность отражения, мы умножаем результат маскированного отражения на значение свойства `_ReflectionAmount`. Это позволит нам контролировать интенсивность отражения по всей поверхности объекта.

На следующем скриншоте приводятся результаты, которые мы получаем при различных значениях свойства `_ReflectionAmount`.

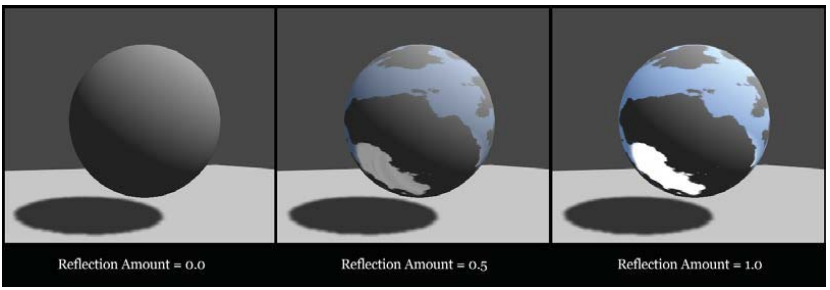

Рис. 4.11. Результаты рассчёта отражения, получаемые при различных значениях общей интенсивности отражения

Карты нормалей и отражения

В определённый момент вам может понадобиться изменять вектор отражения в точке, используя нормаль к поверности. Например, для того чтобы симулировать поверхность замерзшего стекла или ледяного куба. Нельзя ожидать, что игра будет выдавать 60 fps при использовании настолько детализированных моделей, как может понадобиться в этом случае. Для имитации более высокополигональных деталей нам потребуется использовать карту нормалей, поэтому сейчас мы рассмотрим, как использовать информацию из карты нормалей в эффекте отражения.

Для решения поставленной задачи мы воспользуемся ещё одним встроенным в структуру `Input` параметром, с помощью которого мы будем передавать модифицированные нормали поверхности с учетом карты нормалей. Давайте посмотрим, как нам нужно изменить структуру `Input` для создания этого эффекта.

Подготовка

Давайте создадим новую, чистую сцену с помощью следующих действий:

1. Нам вновь потребуется кубмап для создания эффекта отражения. Поэтому вы можете использовать или кубмап из предыдущего рецепта, или сгенерировать новый. Кубическая текстура, которую использовали мы и которая включена в примеры кода, идущие вместе с этой книгой, показана ниже.

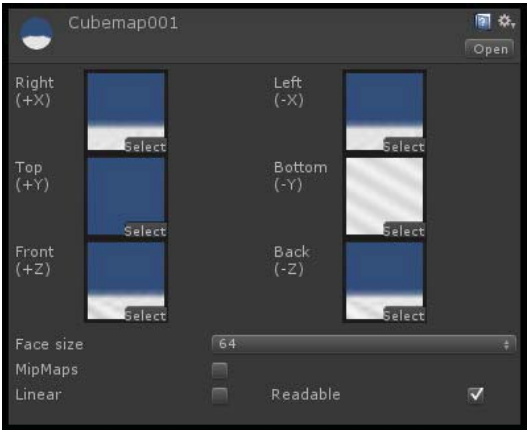

Рис. 4.12. Кубическая текстура, используемая для данного рецепта

2. Нам также понадобится карта нормалей для создания соответственно измененных отражений.

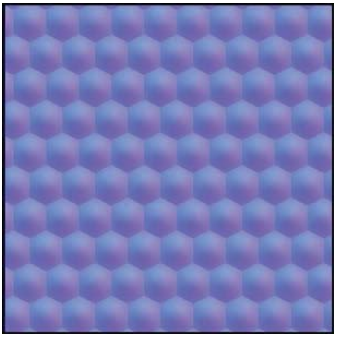

Рис. 4.13. Карта нормалей, используемая для данного рецепта

3. После этого создайте новую сцену с объектом, плоскость земли и направленный источник света, также создайте новый шейдер и материал. Эти действия позволят нам запустить шейдер и проверить, что он работает.

Как это сделать...

Давайте теперь напишем шейдер, чтобы мы могли на практике научиться использовать карты нормалей для отражающего шейдера.

1. Давайте добавим свойства, которые позволят нам назначить свой кубмап и карту нормалей. Этот этап уже должен быть вам знаком. Разработку шейдеров всегда стоит начинать с определения всех свойств, которые вам понадобятся. Добавьте следующий код в блок свойств вашего шейдера.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _NormalMap ("Normal Map", 2D) = "bump" {}
  _Cubemap ("Cubemap", CUBE) = "" {}
  _ReflAmount ("Reflection Amount", Range(0,1)) = 0.5
}
```

2. После этого нам потребуется объявить свойства в блоке SubShader, чтобы мы смогли получить доступ к данным из нашего блока свойств.

```
samplerCUBE _Cubemap;
sampler2D _MainTex;
sampler2D _NormalMap;
float4 _MainTint;
float _ReflAmount;
```

3. Далее вам нужно будет в структуру `Input` добавить следующий код. Здесь и происходит вся магия, с помощью которой получаются отражения, использующие карту нормалей. С помощью макроса `INTERNAL_DATA` мы можем получить доступ к нормалям поверхности после их модификации картой нормалей.

```
struct Input
{
  float2 uv_MainTex;
  float2 uv_NormalMap;
  float3 worldRefl;
  INTERNAL_DATA
};
```

4. И наконец, нам потребуется добавить в функцию `surf()` следующий код, чтобы воспользоваться изменёнными нормалями при вычислении отражений.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex);
  float3 normals = UnpackNormal(tex2D(_NormalMap,
IN.uv_NormalMap)).rgb;

  o.Normal = normals;
  o.Emission = texCUBE(_Cubemap, WorldReflectionVector (IN,
o.Normal)).rgb * _ReflAmount;
  o.Albedo = c.rgb * _MainTint;
  o.Alpha = c.a;
}
```

Следующий скриншот демонстрирует результат использования карты нормалей для воздействия на эффект отражения.

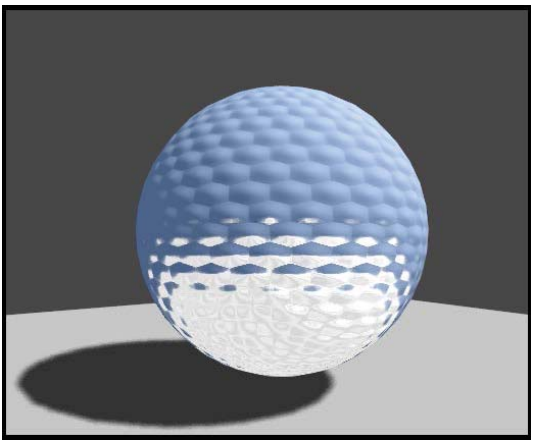

Рис. 4.14. Результат расчёта отражения с учётом карты нормалей

Как это работает...

Обратите внимание, что этот шейдер очень похож на последний написанный нами, за одним очень важным отличием. Мы используем попиксельную карту нормалей для модификации отражения из кубической текстуры. Для этого нам нужно иметь доступ к нормалям поверхности объекта после того, как к ним была применена карта нормалей. Следующим кодом мы применяем карту нормалей:

```
float3 normals = UnpackNormal(tex2D(_NormalMap, IN.uv_NormalMap)).rgb;
o.Normal = normals;
```

После того как эти строки кода выполнятся, нормали к поверхности модели будут изменены, теперь нам нужно использовать их для модификации отражения. Получить доступ к изменённым нормалям мы сможем, если добавим в нашу структуру `Input` строчку `INTERNAL_DATA`, а затем будем использовать `WorldReflectionVector (IN, o.Normal)` вместо координат при сэмплинге нашей кубической текстуры. Это ещё одна функция, встроенная в Unity, наличие которой избавляет нас от того, чтобы каждый раз писать код самим. Таким образом, мы сможем сконцентрироваться на написании существенных для нашего шейдера функций, которые непосредственно участвуют в создании нужных нам эффектов.

Но это ещё не всё...

Есть ещё ряд встроенных свойств, к которым мы можем получить доступ в нашей структуре `Input`, и можете быть уверены – мы остановимся на них в следующих главах. В таблице приводится информация о том, что эти встроенные свойства делают и как их можно использовать. Чтобы получить более детальную информацию, вы можете посетить следующий сайт: http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaders.html.

Таблица 4.1. Таблица описания встроенных свойств

Тип	Свойство	Описание
`float3`	`viewDir`	Содержит направление взгляда для вычисления эффектов параллакса, задней подсветки и т. д.
`float4`	переменная с семантикой `COLOR`	Содержит интерполированный повертекстный цвет
`float4`	`screenPos`	Содержит положение относительно экрана для эффектов отражения
`float3`	`worldPos`	Содержит положение относительно мировых координат
`float3`	`worldRefl`	Содержит вектор отражения в глобальных координатах, если поверхностный шейдер не записывал информацию в `o.Normal`. Для примера посмотрите Reflect-Diffuse шейдер
`float3`	`worldNormal`	Содержит вектор нормали в глобальных координатах, если поверхностный шейдер не записывал информацию в `o.Normal`

Таблица 4.1. (окончание)

Тип	Свойство	Описание
float3	`worldRef;` для использования требуется объявление `INTERNAL_DATA`	Содержит вектор отражения в глобальных координатах, если поверхностный шейдер записывал информацию в `o.Normal`. Чтобы получить вектор отражения, основанный на попиксельной карте нормалей, используйте `WorldReflectionVector (IN, o.Normal)`. Для примера посмотрите Reflect-Bumped шейдер
float3	`worldNormal;` для использования требуется объявление `INTERNAL_DATA`	Содержит вектор нормали в глобальных координатах, если поверхностный шейдер записывал информацию в `o.Normal`. Чтобы получить вектор нормали, основанный на попиксельной карте нормалей, используйте `WorldNormalVector (IN, o.Normal)`

Отражения по Френелю

Один из наиболее часто используемых типов отражений – отражение по Френелю. Эффектом Френеля называется изменение отражающей способности поверхности в зависимости от угла, под которым вы на неё смотрите. Его можно заметить практически на любом типе поверхностей, но чаще всего его используют при создании корпусов машин. Мы сразу видим, что поверхность автомобиля отражает свет, но если посмотреть на нее под более острым углом, вы увидите, что интенсивность отражений и бликов увеличивается, и появляется приятный глазу эффект задней подсветки.

Однако не все поверхности обладают одинаковой степенью отражения по Френелю. Некоторые поверхности, такие как корпуса машин, обладают высокой интенсивностью отражения по Френелю, в то время как другие, например кусок пластика, являются менее отражающими и более блёклыми.

В этом рецепте мы рассмотрим лишь базовую реализацию эффекта отражения по Френелю, поскольку в реальном мире эффект Френеля складывается из отражения и преломления, которые зависят от того, под каким углом человек смотрит на поверхность объекта. Но, поскольку мы еще не останавливались на эффектах преломления, то давайте посмотрим, как этот эффект реализуется в большинстве компаний, выпускающих игры, и как мы можем модифицировать их приёмы для создания интересных визуальных эффектов отражения.

Подготовка

И вновь давайте создадим новую сцену, а также необходимые объекты для неё, чтобы мы могли сконцентрироваться на написании нашего шейдера. Выполните следующую последовательность действий:

1. Для реализации эффекта отражения по Френелю нам потребуется кубическая текстура. Поэтому, вам нужно будет либо сгенерировать новую кубическую текстуру, либо использовать ту, что была в предыдущем рецепте. На следующем скриншоте показан кубмап, который мы использовали в этом рецепте, найти его вы можете по адресу www.packtpub.com/support.

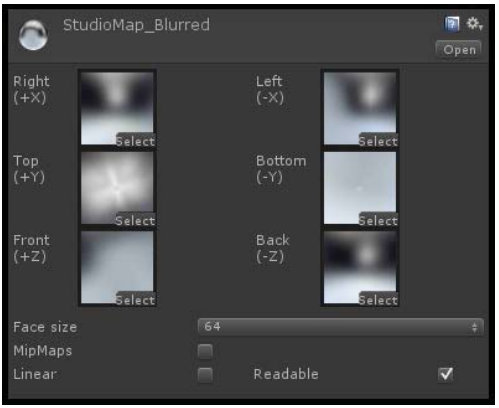

Рис. 4.15. Кубическая текстура, используемая в данном рецепте

2. Создайте новую сцену, объект, плоскость (которая будет играть роль поверхности земли), новый шейдер и новый материал.
3. Также создайте направленный источник света, чтобы мы могли работать с освещением.

Как это сделать...

Давайте перейдём к созданию нашего шейдера и напишем эффект отражения по Френелю. Выполните следующие действия:

1. Сначала нам нужно задать наши свойства в блоке свойств. На этот раз мы будем использовать встроенную модель освещения BlinnPhong, поэтому нам потребуется объявить свойства, необходимые для Specular-компонента модели освещения.

```
Properties
{
  _MainTint("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _Cubemap ("Cubemap", CUBE) = "" {}
  _ReflectionAmount ("Reflection Amount", Range(0,1)) = 1
  _RimPower ("Fresnel Falloff", Range(0.1, 3)) = 2
  _SpecColor ("Specular Color", Color) = (1,1,1,1)
  _SpecPower ("Specular Power", Range(0,1)) = 0.5
}
```

2. Для этого шейдера нам потребуется поддержка шейдерной модели версии 3, благодаря чему мы сможем использовать достаточное количество регистров, чтобы передать все необходимые данные в функцию `surf()`. Для этого нам нужно добавить следующие инструкции `#pragma`.

```
CGROGRAM
#pragma surface surf BlinnPhong
#pragma target 3.0
```

3. После этого нам нужно не забыть создать связь между нашими новыми свойствами и кодом шейдера, для этого мы объявим наши переменные следующим образом:

```
samplerCUBE _Cubemap;
sampler2D _MainTex;
float4 _MainTint;
float _ReflectionAmount;
float _RimPower;
float _SpecPower;
```

4. Чтобы наш эффект отражения заработал, нам потребуется объявить параметры `worldRefl` и `viewDir` в структуре `Input`.

```
struct Input
{
  float2 uv_MainTex;
  float3 worldRefl;
  float3 viewDir;
};
```

5. После этого нам нужно будет рассчитать эффект задней подсветки в функции `surf()`, чтобы создать простой эффект отражения по Френелю.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex);

  float rim = 1.0 - saturate(dot(o.Normal, normalize(IN.viewDir)));
```

```
    rim = pow(rim, _RimPower);

    o.Albedo = c.rgb * _MainTint;
    o.Emission = (texCUBE(_Cubemap, IN.worldRefl).rgb * _ReflectionAmount)
* rim;
    o.Specular = _SpecPower;
    o.Gloss = 1.0;
    o.Alpha = c.a;
}
```

На следующем скриншоте представлены итоговые результаты работы нашего шейдера, реализующего простой эффект Френеля. Помимо прочего, мы можем взять его за основу для написания шейдера поверхности машины.

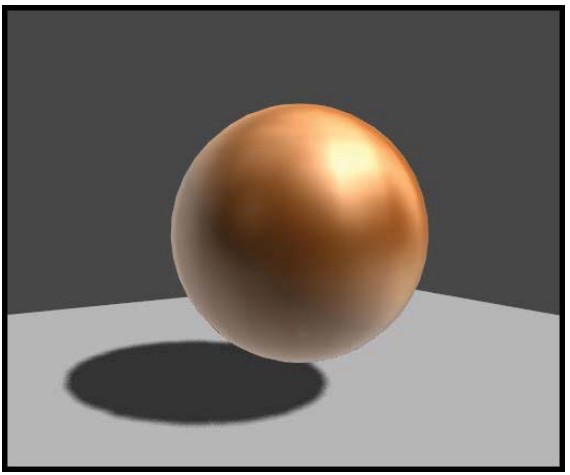

Рис. 4.16. Результат работы шейдера, реализующего эффект Френеля

Как это работает...

В этом примере мы вычисляем величину уменьшения интенсивности отражения, с помощью которой мы можем маскировать участки поверхности с более высокой отражающей способностью и участки с более низкой. Сравнивая направление взгляда с нормалью к поверхности, мы можем вычислить это значение для текущего положения камеры. После чего мы инвертируем это значение и получаем маску, которая будет более белой по краям поверхности и более чёрной в тех местах, где поверхность больше направлена в сторону наблюдателя. Идею иллюстрирует следующий скриншот.

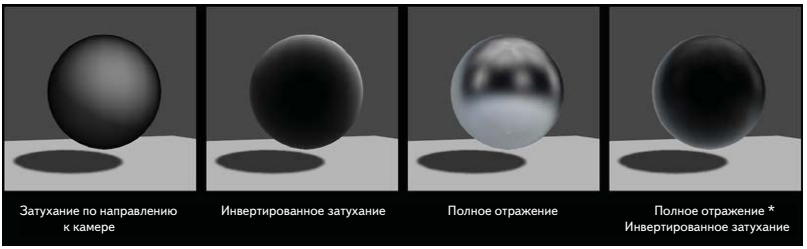

Рис. 4.17. Иллюстрация применения инвертирования при отражении

После этого мы добавляем Specular и диффузные компоненты света и получаем финальный шейдер, реализующий эффект Френеля.

Создание простой динамической системы кубических текстур

Несмотря на то что к этому моменту мы уже узнали много тонкостей работы с кубмапами, мы всё ещё не можем корректно обрабатывать отражения объектов окружающего мира для движущегося объекта. К примеру, если ваш виртуальный мир будет состоять из множества комнат и коридоров, то мы не сможем изготовить одну кубическую текстуру для всего уровня вашей игры. Такая текстура содержала бы неправильную информацию об окружающих объектах в комнатах. И всё, что бы мы получили, – это статичное нереалистичное отражение.

Исправить ситуацию, сделав отражения для разных комнат различными, можно несколькими способами. Первый и наиболее простой – это использовать разные кубмапы на основании того, в какой комнате находится объект. То есть при перемещении объекта из комнаты в комнату соответствующим образом будут заменяться наши кубические текстуры. Второй способ – это изменение кубмапа в реальном времени по мере перемещения игрока в окружающем мире – способ, при котором кубмап генерируется заново каждый фрейм игры. И хотя второй способ более предпочтителен с визуальной точки зрения, потому что вы не заметите «скачка» при смене кубмапа, он весьма затратен с точки зрения необходимых вычислений, поэтому решение о его использовании нужно принимать, учитывая все необходимые для вашей игры ресурсы.

В этом рецепте мы сосредоточимся на первом варианте и покажем, как вы можете сделать очень простую систему для переключения между кубмапами, основываясь на текущем положении игрока в пространстве. В последнем разделе этого рецепта приводится информация о создании системы real-time-отражений, поэтому обратите внимание на него, если вам это интересно и вы хотите увидеть разницу между двумя вышеозвученными подходами.

Подготовка

Выполните следующие действия:

1. Нам потребуется создать новую сцену, задать плоскость земли и поместить туда сферу. Кроме того, создайте направленный источник света, чтобы предоставить нашему шейдеру возможность работать со светом.
2. Далее добавьте на сцену два пустых `GameObject`, и назовите их `pos001` и `pos002`.
3. После этого назначьте сфере новый материал, а ему сопоставьте шейдер отражений по Френелю из нашего предыдущего рецепта. После этих действий ваша сцена должна выглядеть как на следующем скриншоте.
4. И наконец, создайте скрипт и назовите его `SwapCubemaps.cs`.

Чуть ниже приводится скриншот, отображающий итоговый вид нашей сцены, в которой всё готово к созданию нашей динамической системы отражений.

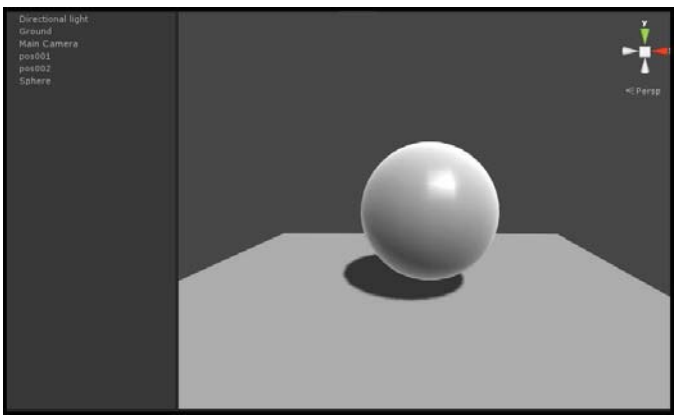

Рис. 4.18. Исходное состояние сцены для работы над системой динамических отражений

Как это сделать...

После того как вы подготовили сцену, можно переходить к программированию системы отражений, для чего вам нужно выполнить следующие шаги:

1. Начнём с того, что перед декларацией класса добавим строчку `[ExecuteInEditMode]`.

```
[ExecuteInEditMode]
public class SwapCubemaps : MonoBehaviour
```

2. После этого нам потребуется объявить несколько переменных для хранения данных нашей системы. Более подробно этот этап мы объясним в следующей секции этого рецепта.

```
public Cubemap cubeA;
public Cubemap cubeB;

public Transform posA;
public Transform posB;

private Material curMat;
private Cubemap curCube;
```

3. Для того чтобы видеть, откуда в 3D-пространстве были сняты наши кубмапы, мы воспользуемся встроенным в Unity методом визуальной отладки – **Gizmos**. Для этого давайте добавим следующий код в конец нашего скрипта.

```
void OnDrawGizmos()
{
  Gizmos.color = Color.green;

  if (posA)
  {
          Gizmos.DrawWireSphere(posA.position, 0.5f);
  }

  if (posB)
  {
          Gizmos.DrawWireSphere(posB.position, 0.5f);
  }
}
```

4. Теперь нам потребуется создать новую функцию, определяющую, какой кубмап мы будем использовать, на основе расстояния объекта от установленных нами двух позиций.

```
private Cubemap CheckProbeDistance()
{
```

```
float distA = Vector3.Distance(transform.position, posA.position);
float distB = Vector3.Distance(transform.position, posB.position);

if (distA <= distB)
{
        return cubeA;
}
else if (distB < distA)
{
        return cubeB;
}
}
```

5. После этого мы будем каждый кадр проверять расстояния от объекта до этих позиций и подставлять соответствующий кубмап в материал.

```
void Update ()
{
  curMat = renderer.sharedMaterial;
  if (curMat)
  {
        curCube = CheckProbeDistance();
        curMat.SetTexture("_Cubemap", curCube);
  }
}
```

После того как вы сохраните шейдер, вернитесь в редактор Unity, чтобы шейдер скомпилировался заново. Далее нажмите на кнопку **Play** и попробуйте подвигать сферу вперёд-назад. Вы должны увидеть эффект, похожий на изображённый на следующем скриншоте.

Рис. 4.19. Изменение отражения при перемещении сферы

Как это работает...

Мы начинаем скрипт с того, что добавляем к нашему классу аттрибут `[ExecuteInEditMode]`. Указание этого аттрибута говорит Unity, что

этот скрипт должен выполняться не только при нажатой кнопке **Play**, но даже когда мы будем просто находиться в редакторе. Таким образом, мы сможем тестировать изменение кубмапов даже без нажатия кнопки **Play** – так получится гораздо быстрее.

Далее мы добавляем в скрипт несколько переменных, чтобы пользователь смог задать две кубические текстуры, и две координаты в трехмерном пространстве, которые будут учитываться при расчёте расстояний. Помимо этого, в скрипте используются две приватные переменные, в которых мы храним текущий материал и кубическую текстуру во время выполнения.

Добавив необходимые переменные, мы можем использовать встроенную функцию `OnDrawGizmos()` для отображения вводимых пользователем положений. На основе этих положений наш скрипт будет определять, в какой момент нужно будет заменить кубмапы.

Вот мы и добрались до сути нашего скрипта. Далее мы объявляем функцию, которая вычисляет расстояние нашей сферы от каждого из заданных положений с помощью метода `Vector3.Distance()`. После этого она сравнивает эти расстояния и возвращает кубмап, который соответствует точке с меньшим расстоянием.

Далее в функции `Update()` мы получаем текущий материал сферы (или объекта, к которому прикреплён скрипт) и назначаем ему кубическую текстуру, которую возвращает предыдущая функция.

Наш скрипт весьма прост и служит лишь для иллюстрации идеи, но при желании вы можете его доработать до состояния законченной системы, содержащей множество кубических текстур для каждой комнаты. Такая система могла бы даже автоматически генерировать кубмапы во время выполнения, что хорошо бы подошло для игр, в которых нецелесообразно, с точки зрения производительности, использовать полностью real-time-отражения.

Но это ещё не всё...

Возможно, вас заинтересует создание системы отражений, работающей в реальном времени, в которой кубмап будет обновляться каждый фрейм. Несомненно, такая система будет выглядеть лучше, но для достижения этой красоты потребуется пожертвовать производительностью. Ниже приводится ссылка на метод, который вам нужно будет использовать в этом случае (http://docs.unity3d.com/Documentation/ScriptReference/Camera.RenderToCubemap.html).

ГЛАВА 5
Модели освещения

В этой главе вы узнаете о том, как создавать следующие модели освещения:

- модель освещения Lit Sphere;
- модель освещения Diffuse Convolution;
- модель освещения автомобильной краски;
- шейдер кожи;
- шейдер ткани.

Введение

На протяжении последних нескольких глав мы рассматривали различные способы использования компонентов языка написания поверхностных шейдеров для создания шейдеров и моделей освещения. В этой главе мы воспользуемся полученными знаниями и напишем целиком шейдеры для разных эффектов.

Мы сконцентрируемся на основных типах шейдеров, наиболее востребованных в игровой индустрии. Эти знания мы применим для совершенствования процесса создания наших собственных шейдеров, что пригодится нам, когда в игре понадобится создать материал с новыми свойствами. Кроме того, мы обсудим некоторые способы повышения эффективности, применимые в случае работы в команде, а также то, каким образом художники из вашей команды смогут пользоваться вашими шейдерами.

Модель освещения Lit Sphere

Модель освещения Lit Sphere, – это любопытный случай использования освещения на основе изображения (Image Based Lighting или IBL). Мы можем использовать 2D-текстуру, для того чтобы полно-

стью запечь в неё окружающее освещение. Как, например, это делается в Zbrush. Если вы знакомы с MatCaps в Zbrush, то освещение Lit Sphere работает по тому же принципу. Мы создаем текстуру, в которую запекаем диффузный свет, блики, отражения, заднюю подсветку, и используем её в нашем шейдере. Единственный минус такого шейдера – так как мы запекли все параметры освещения в текстуру, оно не будет меняться, если, конечно, не менять вручную текстуры в разных точках окружения, наподобие того, что мы видели в главе 4 «Добавим отражения в ваш мир» в рецепте «Создание простой динамической системы кубических карт в Unity3D». Таким образом, шейдер не будет реагировать на источники света, и освещение не будет меняться, когда вы рассматриваете вашу виртуальную модель с разных сторон. Пример текстуры, используемой в модели освещения Lit Sphere, которую часто называют Sphere Map, приводится на следующем скриншоте.

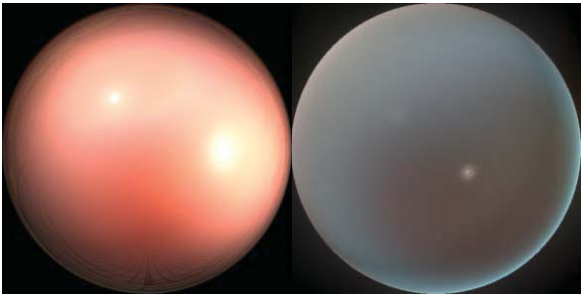

Рис. 5.1. Пример Sphere Map текстуры

Данный тип шейдеров хорошо подходит для создания качественных сцен диорам или для использования в скриптованных игровых сценах, в которых позиция камеры жёстко фиксирована и требуется детальное освещение для персонажей и окружающих их объектов.

Давайте рассмотрим, как создать такую модель освещения и как её использовать в поверхностных шейдерах Unity.

Подготовка

Перед тем как начать работу над шейдером, нам потребуется научиться создавать текстуры освещения, которые мы будем использовать в шейдере. Для этого мы можем использовать Photoshop, но гораздо проще будет использовать небольшую бесплатную утилиту MaCrea, скачать которую можно по адресу: http://www.taron.de/macrea. Эта

программа – замечательный бесплатный инструмент, с помощью которого можно создавать Lit Sphere текстуры. Советуем вам посмотреть обучающие видео об интерфейсе и работе с MaCrea на Vimeo. Видеовведение в MaCrea доступно по адресу: http://vimeo.com/14030320.

После того, как вы освоитесь с созданием сферических карт, мы сможем перейти к следующей части нашего рецепта. На скриншоте показаны интерфейс MaCrea и готовая текстура, созданная в этой программе.

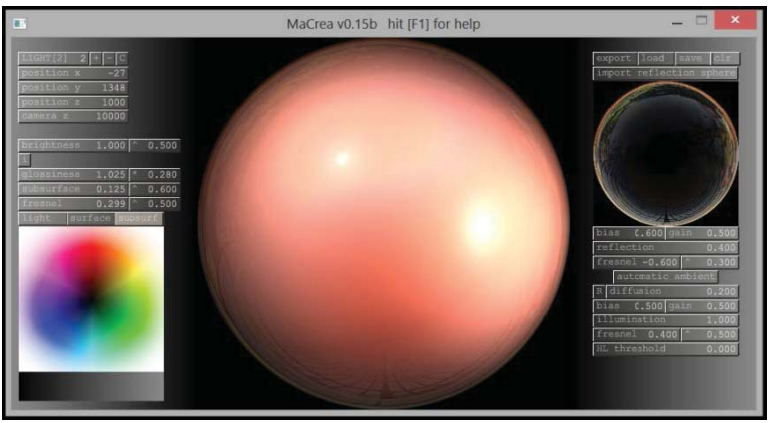

Рис. 5.2. Интерфейс MaCrea и результат работы в ней

Выполните следующие действия:

1. Создайте новую сцену и перетащите в нее несколько объектов, плоскость и источник света.
2. Создайте новый шейдер и материал. После этого назначьте ваш шейдер материалу.

Как это сделать...

После того как наша сцена и её ресурсы будут подготовлены, а наш шейдер готов для редактирования в MonoDevelop, мы можем перейти к написанию модели шейдинга Lit Sphere.

Выполните следующие действия:

1. Как обычно, нам потребуется задать свойства для поверхностного шейдера, чтобы пользователи нашего шейдера смогли выставлять разные текстуры и изменять значения переменных. Поэтому давайте добавим следующий код в блок свойств.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _NormalMap ("Normal Map", 2D) = "bump" {}
}
```

2. Так как для освещения модели шейдер будет использовать только сферическую карту, нам не понадобится функция освещения Lambert, вместо неё мы объявим нашу собственную функцию освещения `Unlit`. Кроме этого, нам понадобится написать ещё и небольшую вершинную функцию, чтобы наш шейдер заработал.

```
CGPROGRAM
#pragma surface surf Unlit vertex:vert
```

3. Далее нам опять нужно убедиться, что мы не забыли разместить наши свойства в блоке `SubShader`, чтобы мы могли использовать данные, изменяемые пользователем в панели **Инспектора** в редакторе Unity.

```
sampler2D _MainTex;
sampler2D _NormalMap;
float4 _MainTint;
```

4. Теперь мы можем добавить функцию освещения, которую мы назвали **Unlit**. Мы это делаем, потому что в данном случае мы не хотим, чтобы светильники в сцене влияли на наш шейдер. Но при этом нам нужно, чтобы наш объект отбрасывал тень. Добавьте в шейдер следующую функцию освещения.

```
inline fixed4 LightingUnlit(SurfaceOutput s, fixed3 lightDir,
fixed atten)
{
  fixed4 c = fixed4(1,1,1,1);
  c.rgb = c * s.Albedo;
  c.a = s.Alpha;
  return c;
}
```

5. Теперь нам потребуется добавить в нашу структуру `Input` дополнительные свойства, чтобы мы могли передавать данные из функции `vert()` в функцию `surf()`.

```
struct Input
{
  float2 uv_MainTex;
  float2 uv_NormalMap;
```

```
  float3 tan1;
  float3 tan2;
};
```

6. Для того чтобы получить правильные значения из сферической карты, нам нужно преобразовать касательные векторы в точке модели. Для этого мы умножаем векторы инвертированной транспонированной матрицы MV (матрица перехода из координат модели в координаты камеры) на матрицу поворота в касательное пространство. Не волнуйтесь, если не все описанные действия вам понятны, – мы объясним их более подробно в следующей секции.

```
void vert(inout appdata_full v, out Input o)
{
  UNITY_INITIALIZE_OUTPUT(Input, o);

  TANGENT_SPACE_ROTATION;
  o.tan1 = mul(rotation, UNITY_MATRIX_IT_MV[0].xyz);
  o.tan2 = mul(rotation, UNITY_MATRIX_IT_MV[1].xyz);
}
```

7. И наконец, мы можем добавить в нашу функцию `surf()` вычисления, в результате которых мы получим правильные значения UV-координат для сферической карты, значение из которой мы передадим в структуру `SurfaceOutput`. И опять – детали этой функции будут объяснены в следующей секции.

```
void surf(Input IN, inout SurfaceOutput o)
{
  float3 normals = UnpackNormal(tex2D(_NormalMap,
IN.uv_NormalMap));
  o.Normal = normals;

  float2 litSphereUV;
  litSphereUV.x = dot(IN.tan1, o.Normal);
  litSphereUV.y = dot(IN.tan2, o.Normal);

  half4 c = tex2D(_MainTex, litSphereUV*0.5+0.5);
  o.Albedo = c.rgb * _MainTint;
  o.Alpha = c.a;
}
```

На следующем скриншоте приводится результат работы нашего шейдера, использующего сферическую карту, или, согласно терминологии Zbrush, – MatCaps.

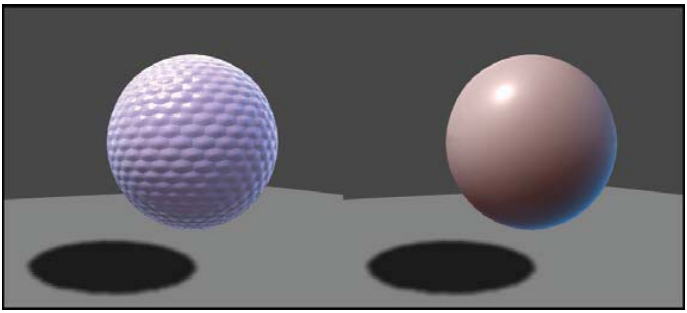

Рис. 5.3. Результат работы шейдера, использующего сферическую карту

Как это работает...

Самое интересное в этой модели освещения происходит внутри функции `vert()`, где мы присваиваем `o.tan1` и `o.tan2` значения касательных векторов в пространстве камеры, для чего умножаем векторы из инвертированной транспонированной матрицы MV (Model*View) на матрицу поворота в касательное пространство. Эти вычисления поворачивают векторы так, чтобы их можно было использовать при сэмплинге сферической карты. Откуда же у нас появляется эта матрица? UNITY_MATRIX_IT_MV – это одно из предоставляемых Unity значений, которое нам не придётся рассчитывать самостоятельно.

На самом деле Unity вычисляет за нас большую часть наиболее распространённых матриц преобразований, используемых в CGFX-шейдерах. Это ещё одно преимущество использования поверхностных шейдеров – нам не нужно делать это самостоятельно. Вместо этого мы можем воспользоваться встроенными значениями.

Вы, наверное, спросите, зачем нам в нашем случае нужно использовать это преобразование векторов. Детальное объяснение матриц преобразований определённо выходит за рамки данной книги, но, говоря простым языком, нам нужно преобразовать касательные векторы в точке модели из локальных координат в координаты камеры, для того чтобы получить правильный цвет из сферической карты и применить его в данной точке. Вы можете считать, что таким образом вы изменяете взаимное расположение модели и себя как зрителя.

Векторы, получившиеся в результате данных преобразований, показаны на скриншоте. Мы будем использовать их для выборки значений из нашей Sphere map текстуры.

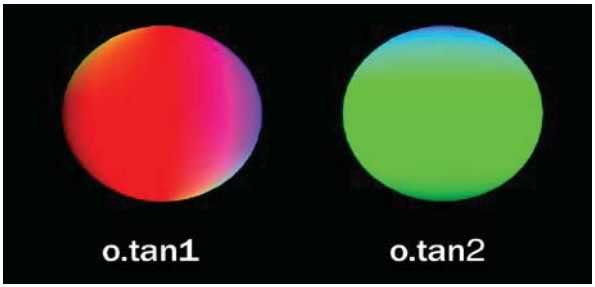

Рис. 5.4. o.tan1 и o.tan2

И наконец, мы используем значения `IN.tan1` и `IN.tan2` в качестве UV-координат в нашей сферической карте. Мы можем использовать эти данные из структуры `Input`, потому что мы их туда сохранили в функции `vert()`.

Это простой и в то же время приятный глазу способ достичь эффекта сложного освещения. Один лишь минус – освещение не будет меняться при изменении источников света на сцене. При таком подходе освещение всегда будет привязано к направлению взгляда камеры, как если бы текстура освещения проецировалась камерой на видимый объект.

Дополнительная информация

Как и в предыдущих главах, много дополнительной информации по нашей теме содержится в Интернете. Мы приводим несколько ссылок, где вы можете найти дополнительную информацию и обучающие материалы по использованию сферических карт и модели освещения Lit Sphere:

- отличное объяснение матриц преобразований вы можете найти в онлайн-книге по Cg: https://developer.nvidia.com/content/cg-tutorial-chapter-4-transformations;
- Тут вы можете найти дополнительную информацию по всем встроенным матрицам преобразований: http://docs.unity3d.com/Documentation/Components/SLBuiltinStateInPrograms.html;
- Дополнительная информация по отражениям в MaCrea: http://vimeo.com/14189456;
- Информация о сэл-шейдинге в MaCrea: http://vimeo.com/14033777.

Модель освещения Diffuse Convolution

Diffuse Convolution (диффузная свёртка) – это процесс размытия кубической карты таким образом, чтобы её общая интенсивность освещения сохранилась, но детали стали размытыми. Этот приём позволяет увеличить глобальное освещение поверхности модели. Для этого нужно отрендерить окружающий мир в кубмап, применить к нему алгоритм diffuse convolution и использовать его для подсветки вашей модели.

В этом рецепте мы рассмотрим применение данной техники в поверхностных шейдерах Unity. Для генерации кубических карт мы будем использовать утилиту CubeMapGen.

Подготовка

Перед тем как приступать к написанию шейдера, нам потребуется создать «свёрнутый» (convoluted) кубмап. Существует несколько способов это сделать, но мы остановимся на утилите CubeMapGen от ATI. Скачать эту утилиту вы можете с сайта производителя по адресу: http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/cubemapgen/.

Следующее изображение показывает интерфейс CubeMapGen, а также кубмап, загруженный в программу.

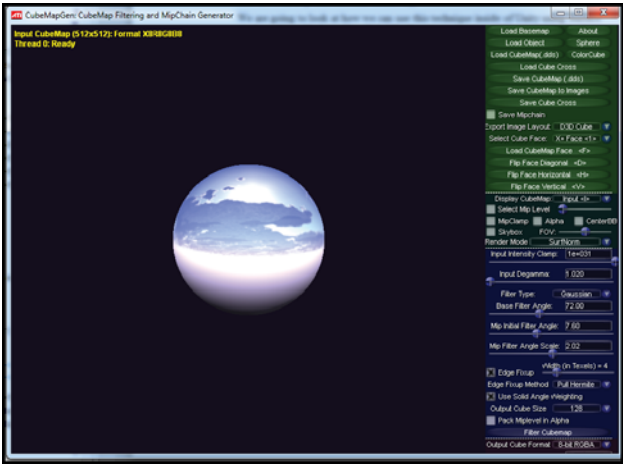

Рис. 5.5. Интерфейс программы CubeMapGen

Давайте пройдёмся по шагам создания кубической карты:

1. Запустите CubeMapGen и загрузите одну из кубических карт, поставляемых с приложением. Они находятся в каталоге, в который вы проинсталлировали CubeMapGen.
2. После того как вы загрузили кубмап в программу, нам нужно будет применить к нему convolution фильтр, то есть специальным образом его размыть. Для этого в синей части интерфейса нам нужно будет выставить следующие параметры: **Filter Type** (Тип фильтра) – **Gaussian** (Фильтр Гаусса), **Base Filter Angle** (Угол фильтра) – 72.00, **Mip Initial Filter Angle** (Начальный мип-угол) – 7.60, **Mip Filter Angle Scale** (Угловой мип-масштаб) – 2.02 и **Edge fix up** (Выравнивание границ) – 4. После этого там же, в синей части интерфейса нажмите кнопку **Filter Cubemap** (Применить фильтр). Это займёт некоторое время, но в результате вы получите что-то вроде изображённого на следующем скриншоте.

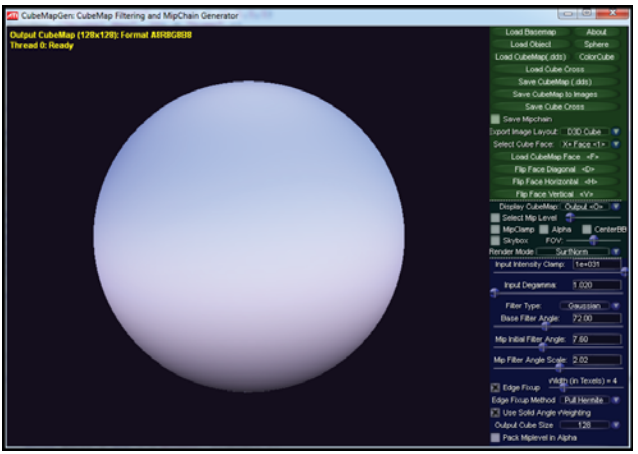

Рис. 5.6. Результат применения фильтра в программе CubeMapGen

3. После того как CubeMapGen закончит применение фильтра, вы сможете сохранить отдельные грани кубмапа, нажав в зелёной секции интерфейса на кнопку **Save Cubemap to Images** (Сохранить кубмап как картинки). В результате вы получите изображения для каждой из сторон кубмапа, которые можно импортировать в Unity и построить на их основе новую кубическую карту.

4. Теперь, когда наша кубическая карта готова, нам нужно будет сделать сцену, в которой мы будем тестировать наш шейдер. Поэтому создайте новую сцену и поместите на неё несколько объектов, а также один направленный источник света. Кроме этого, нам понадобятся новый материал и новый шейдер.

Как это сделать...

После того как мы сгенерировали все ассеты, мы можем перейти к процессу написания шейдера, который будет использовать нашу размытую кубическую карту.

Выполните следующую последовательность действий:

1. Как обычно, мы создадим набор свойств, которые позволят художнику взаимодействовать с нашим шейдером, чтобы он мог настраивать его, как ему покажется нужным.

```
Properties
{
  _MainTint ("Global Tint", Color) = (1,1,1,1)
  _BumpMap ("Normal Map", 2D) = "bump" {}
  _AOMap ("Ambient Occlusion Map", 2D) = "white" {}
  _CubeMap ("Diffuse Convolution Cubemap", Cube) = "" {}
  _SpecIntensity ("Specular Intensity", Range(0, 1)) = 0.4
  _SpecWidth ("Specular Width", Range(0, 1)) = 0.2
}
```

2. После этого нам нужно будет добавить `#pragma` директивы. В данном случае мы создадим новую модель освещения, поскольку мы хотим, чтобы для освещения модели использовался наш кубмап, а не источники освещения на сцене. Кроме этого, нам потребуется объявить использование шейдерной модели версии 3.0, чтобы избежать появления ошибок интерполяции текстур.

```
CGPROGRAM
#pragma surface surf DiffuseConvolution
#pragma target 3.0
```

3. Для того чтобы получить доступ к данным, содержащимся в свойствах шейдера, нам потребуется создать связь между блоком свойств и блоком `SubShader`, объявив соответствующие переменные для каждого из наших свойств. Чтобы создать эту связь, добавьте следующий код:

```
samplerCUBE _CubeMap;
sampler2D _BumpMap;
```

```
sampler2D _AOMap;
float4 _MainTint;
float _SpecIntensity;
float _SpecWidth;
```

4. Наша структура `Input` в этот раз будет весьма простой, поскольку всё, что нам нужно, – это нормали модели в системе мировых координат. Нам потребуется инструкция `INTERNAL_DATA`, поскольку мы будем использовать карту нормалей в нашем шейдере, а эта инструкция предоставит нам модифицированные нормали.

```
struct Input
{
  float2 uv_AOMap;
  float3 worldNormal;
  INTERNAL_DATA
};
```

5. Наша следующая задача – выбрать сигнатуру функции модели освещения. Поскольку мы планируем в нашем шейдере эмулировать блики, нам понадобится направление взгляда.

```
inline fixed4 LightingDiffuseConvolution(SurfaceOutput s,
fixed3 lightDir, fixed3 viewDir, fixed atten)
{

}
```

6. Чтобы от нашей функции освещения был какой-то толк, нам нужно добавить в неё непосредственно вычисления освещения. Давайте начнём с того, что вычислим нужные векторы.

```
//Вычислим все векторы для освещения
viewDir = normalize(viewDir);
lightDir = normalize(lightDir);
s.Normal = normalize(s.Normal);
float NdotL = dot(s.Normal, lightDir);
float3 halfVec = normalize(lightDir + viewDir);
```

7. После этого, нам нужно учесть блики.

```
//Вычислим Specular
float spec = pow(dot(s.Normal, halfVec), s.Specular*128.0) *
s.Gloss;
```

8. И наконец, мы объединяем вычисленные значения модели освещения.

```
fixed4 c;
c.rgb = (s.Albedo * atten) + spec;
```

```
c.a = 1.0f;
return c;
```

9. После того как наша функция освещения будет готова, мы сможем использовать данные из переданных в шейдер текстур. Используя нормаль к поверхности в мировых координатах, мы получаем значение кубмапа и записываем финальный цвет в структуру `SurfaceOutput`.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_AOMap, IN.uv_AOMap);
  float3 normals = UnpackNormal(tex2D(_BumpMap,
IN.uv_AOMap)).rgb;
  o.Normal = normals;

  float3 diffuseVal = texCUBE(_CubeMap,
WorldNormalVector(IN, o.Normal)).rgb;

  o.Albedo = (c.rgb * diffuseVal) * _MainTint;
  o.Specular = _SpecWidth;
  o.Gloss = _SpecIntensity * c.rgb;
  o.Alpha = c.a;
}
```

С помощью нормали к поверхности в мировых координатах мы получаем цвет из размытой (convoluted) кубической текстуры, что придаёт нашей модели весьма реалистичный вид.

Результат применения шейдера Diffuse Convolution представлен на следующем скриншоте.

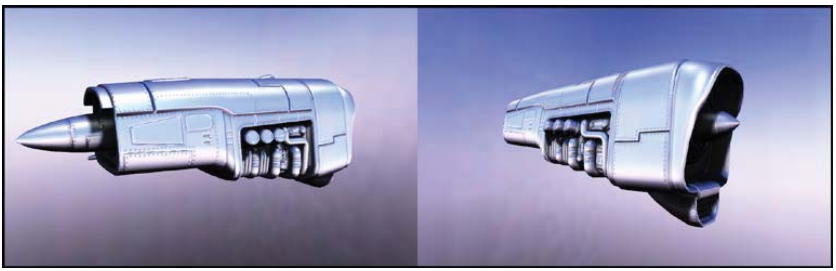

Рис. 5.7. Результат применения шейдера Diffuse Convolution

Как это работает...

Diffuse Convolution – это ещё один простой, но очень эффектный визуальный приём. И хотя он чуть более интерактивен, чем использование сферических карт, освещение всё равно привязано к единст-

венной кубической карте. Конечно, можно обновлять содержимое кубмапа в реальном времени для учёта изменения объектов окружающего мира, но применение фильтра Diffuse Convolution для каждого кадра, может плачевно сказаться на производительности. Однако не волнуйтесь. Специально для этого в Unity есть лайт-пробы (light probes). Объекты в пространстве, которые запоминают параметры освещения вокруг себя. Техника, использующая данный подход, известна как Ambient Cube Shading (окружающее кубическое освещение).

Такой шейдер хорошо подходит для создания небольших сцен, в которых нет ни большого количества движения, ни взаимодействия с освещением. Подобный подход называют освещением, основанным на изображении (Image Based Lighting), так как для освещения объекта мы используем не светильники в сцене, а изображение из кубмапа. Этот приём хорошо подходит для скриптованных сцен, или, например, сцены тюнинга автомобиля в игре.

Основная идея этого шейдера заключается в том, чтобы использовать нормали к поверхности, после того как они были изменены с помощью карты нормалей, для получения цвета из кубической карты. Именно для этого в нашей структуре `Input` мы объявляем параметр `worldNormal` типа `float3` и указываем макрос `INTERNAL_DATA`. После этого мы используем встроенную в Unity функцию `WorldNormalVector()`, чтобы получить итоговый вектор, который мы используем в функции `texCUBE()`. Остальная часть шейдера вам уже должна быть хорошо знакома.

На следующем скриншоте мы видим, как в зависимости от направления нормали к поверхности меняется цвет, получаемый из кубической карты.

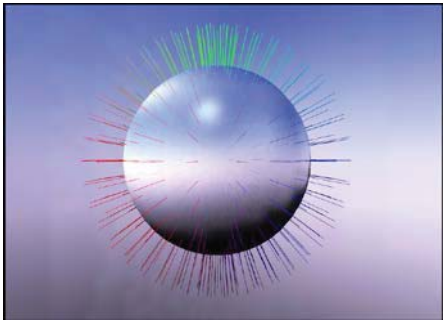

Рис. 5.8. Зависимость цвета, получаемого из кубической карты, от нормали к поверхности

Но это ещё не всё...

Если вы хотите узнать больше о применении лайт-проб (light probes) в Unity, обратите внимание на следующую страницу: http://docs.unity3d.com/Documentation/Manual/LightProbes.html.

Дополнительная информация

Если вы забыли, как делаются кубмапы в Unity, вы всегда можете вернуться к главе 4 «Добавим отражения в ваш мир».

Создание модели освещения автомобильной краски

Один из наиболее распространённых шейдерных эффектов – шейдер поверхности автомобиля. Для его создания нам понадобятся многие изученные нами в предыдущих главах техники, и сейчас-то нам и пригодятся все полученные ранее знания. Мы напишем базу для шейдера, который вы сможете использовать в продакшене в материалах любых моделей автомобилей. Несомненно, это будет один из самых сложных и больших шейдеров среди тех, что мы уже написали, но мы будем делать это по шагам и объясним каждый из них.

Подготовка

Давайте подготовим новую сцену и несколько ассетов, чтобы мы могли перейти к созданию модели освещения автомобильной краски.

Выполните следующие действия:

1. Перво-наперво нам потребуется объект для демонстрации материала, поэтому создайте в пустой сцене новый объект. Удобно, когда есть плоскость, играющая роль земли, и на которую отбрасываются тени от нашего объекта. Поэтому мы рекомендуем вам добавить плоскость в сцену.
2. Чтобы мы смогли перейти к написанию шейдера, нам потребуются новый шейдер и новый материал. Поэтому давайте их создадим и присвоим материал нашему основному объекту, в данном случае – сфере.
3. Для создания этого эффекта нам понадобится BRDF-текстура. Как вы помните из секции одного из пройденных рецептов, посвященной BRDF, вам всего лишь нужно создать текстуру, содержащую цветовые вариации, соответствующие различ-

ным углам взгляда на модель. Проще говоря, нам потребуется цвет для диффузного освещения, зависящий от направления взгляда. На следующем скриншоте приводятся примеры текстур, которые мы использовали для этого автомобильного шейдера.

Рис. 5.9. Использованные BRDF-текстуры

4. Последний этап в подготовке к написанию шейдера – создание кубической карты. Как вы помните, в главе 4 «Добавим отражения в ваш мир» мы использовали скрипт для создания кубической карты прямо из сцены в Unity. Давайте сейчас воспользуемся им же.

Как это сделать...

После того как мы подготовили все необходимые ассеты, мы можем перейти к созданию нашего шейдера. Сначала мы пройдёмся по всему коду шейдера, а затем, рассмотрим его более детально и подробно прокомментируем каждую его часть.

Выполните следующую последовательность действий:

1. Первым делом мы создадим свойства, которые нам понадобятся. В этом шейдере их довольно много, но в следующей секции мы объясним, что каждое из них делает. Несмотря на то что некоторые свойства, вероятно, вам уже будут знакомы, мы всё равно объясним их все.

```
Properties
{
  _MainTint ("Diffuse Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _SpecularColor ("Specular Color", Color) = (1,1,1,1)
  _SpecPower ("Specular Power", Range(0.01, 30)) = 3
  _ReflCube ("Reflection Cube", CUBE) = "" {}
  _BRDFTex ("BRDF Texture", 2D) = "white" {}
  _DiffusePower ("Diffuse Power", Range(0.01, 10)) = 0.5
  _FalloffPower ("Falloff Spread", Range(0.01, 10)) = 3
  _ReflAmount ("Reflection Amount", Range(0.01, 1.0) = 0.5
```

```
  _ReflPower ("Reflection Power", Range(0.01, 3.0)) = 2.0
}
```

2. Поскольку для этого шейдера мы создадим нашу собственную модель освещения, которую назовём `CarPaint`, нам потребуется объявить это в директиве `#pragma`.

```
CGPROGRAM
#pragma surface surf CarPaint
```

3. Чтобы мы смогли получить доступ к свойствам нашего шейдера, мы также объявим их и в нашем `SubShader` блоке. Взгляните на следующий код.

```
sampler2D _MainTex;
sampler2D _BRDFTex;
fixed4 _MainTint;
fixed4 _SpecularColor;
fixed _SpecPower;
fixed _DiffusePower;
fixed _FalloffPower;
fixed _ReflAmount;
fixed _ReflPower;
samplerCUBE _ReflCube;
```

4. Теперь мы можем перейти к написанию нашей модели освещения. Для этого шейдера нам потребуется использовать достаточно большое количество данных, поэтому просмотрите следующий код несколько раз, чтобы получить представление о том, что он делает, прежде чем вставить его в свой шейдер.

```
inline fixed4 LightingCarPaint(SurfaceOutput s, fixed3
lightDir, half3 viewDir, fixed atten)
{
  half3 h = normalize(lightDir + viewDir);
  fixed diff = max(0, dot(s.Normal, lightDir)));

  float ahdn = 1 - dot(h, normalize(s.Normal));
  ahdn = pow(clamp(ahdn, 0.0, 1.0), _DiffusePower);
  half4 brdf = tex2D(_BRDFTex, float2(diff, 1 - ahdn));

  float nh = max(0, dot(s.Normal, h));
  float spec = pow(nh, S.Specular * _SpecPower) * s.Gloss;

  fixed4 c;
  c.rgb = (s.Albedo * _LightColor0.rgb * brdf.rgb +
_LightColor0.rgb * _SpecularColor.rgb * spec)* (atten * 2);
  c.a = s.Alpha + _LightColor0.a * _SpecularColor.a * spec * atten;
  return c;
}
```

5. Давайте обратим наше внимание на структуру `Input` шейдера и добавим в неё следующий код. С его помощью мы сможем создать эффект Френеля, который мы рассматривали в предыдущей главе.

```
struct Input
{
  float2 uv_MainTex;
  float3 worldRefl;
  float3 viewDir;
};
```

6. Теперь мы перейдём к нашей функции `surf()`, в которой осуществляется большинство попиксельных вычислений. В ней мы создадим итоговые визуальные эффекты нашего шейдера автомобильной краски.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex);

  fixed falloff = saturate(1 - dot(normalize(IN.viewDir),
o.Normal));
  falloff = pow(falloff, _FalloffPower);

  o.Albedo = c.rgb * _MainTint;
  o.Emission = pow((texCUBE(_ReflCube, IN.worldRefl).rgb *
falloff), _ReflPower) * _ReflAmount;
  o.Specular = c.r;
  o.Gloss = 1.0;
  o.Alpha = c.a;
}
```

Следующий скриншот демонстрирует результат применения нашего шейдера автомобильной краски к сфере в Unity.

Рис. 5.10. Результат применения шейдера автомобильной краски

Как это работает...

Наш шейдер автомобильной краски на самом деле весьма прост, если рассматривать его по частям, тем более что мы уже рассматривали каждую отдельную часть в главе 1 «Диффузный шейдинг», и в главе 3 «Пусть ваши игры засияют отражённым светом». Так что мы надеемся, что эти знания уже отложились у вас в голове, а сейчас же давайте рассмотрим общую идею эффекта.

С помощью приёма BRDF, который мы разобрали в главе 1 «Диффузный шейдинг», мы создаём эффект градиента между двумя цветами в краске автомобиля. Однако не для всех автомобилей характерен такой визуальный аспект, поэтому от вас как от шейдерного программиста будет зависеть, будете ли вы использовать BRDF-текстуру или какой-то другой источник диффузного цвета.

Далее мы просто рассчитываем эффект Френеля и величину затухания, от которых зависит интенсивность отражения на поверхности автомобиля. Все эти компоненты освещения управляются свойствами из блока свойств, поэтому у художника есть полный контроль за тем, как в итоге будет выглядеть поверхность автомобиля.

Но это ещё не всё...

В Unity Asset Store продаются готовые шейдеры автомобильной краски. Вот ссылка на один из них: http://u3d.as/content/ravel-tammeleht/mo-dy-en-car-paint-shaderpack/2Xe.

Шейдер кожи

Если у вас в игре есть персонажи с натуральной кожей, вам обязательно понадобится шейдер кожи. В этой секции мы рассмотрим один из способов создания шейдера кожи, который вы сможете использовать в своей игре. Разумеется, это будет не самый физически правильный шейдер, но он подойдёт для поставленной задачи, и с помощью него можно будет добиться неплохого визуального эффекта.

Прежде чем начать, нам нужно определиться с тем, что же наша поверхность кожи должна «делать». Вооружившись этой информацией, мы сможем декомпозировать наш шейдер на компоненты, чтобы мы смогли запрограммировать соответствующие эффекты.

Шейдер кожи мы можем разложить на четыре основных компонента. Нельзя сказать, что такое деление является результатом какого-то

правила, но подобная декомпозиция позволит нам в итоге добиться весьма качественного результата. Мы выделим следующие компоненты:

- **Подповерхностное рассеивание** (Subsurface scattering или SSS). Этот эффект наблюдается, если смотреть на тонкую или очень прозрачную кожу на просвет, в результате чего свет, находящийся за ней, создаёт окрашивающий эффект. Для кожи это, как правило, красный оттенок, имитирующий прилегающие кровеносные сосуды. В этой части мы рассмотрим, как можно вычислить кривизну поверхности на основе её карты нормалей.
- **Диффузный.** Как вы наверняка представляете, диффузный компонент цвета кожи не сводится всего лишь к простой шкале серого. И хотя мы всё так же будем использовать технику скалярного произведения вектора света и вектора нормали, нам потребуется применить ещё и технику BRDF, чтобы получить больше возможностей контролировать распределение света по поверхности объекта.
- **Блики.** С бликами на коже не всё так просто, потому что они будут зависеть от того, насколько жирная поверхность кожи. Мы всё ещё можем использовать способы работы с бликами, которые мы выучили, но нам потребуется добавить еще эффект Френеля и заднюю подсветку, чтобы контролировать место расположения блика. Таким образом, мы добьёмся более реалистичного распределения блика на коже. Конечно, мы могли бы для контроля за формой блика использовать текстуры, но в этом рецепте мы будем использовать простые блики, так как про использование текстур мы рассказали в предыдущей главе.
- **Размытые нормали.** Причина, по которой большое количество игровых шейдеров кожи приводит к нереалистичными, или чересчур «пластмассовым» результатам заключается в том, что нормали в карте нормалей указаны очень детально, что хорошо, например, для бликов, поскольку там нас интересуют все детали. Но для диффузного компонента цвета кожи нам важнее мягкий переход цветов.

Подготовка

Давайте подготовим нашу сцену и все ассеты, которые нам понадобятся при реализации различных частей нашего шейдера. Выполните следующие действия:

1. Создайте новую сцену, шейдер и материал. Не забудьте добавить шейдер материалу, а материал назначить объекту. В идеале было бы здорово, если бы вы сейчас работали с моделью головы, но если у вас сейчас этой модели нет под рукой, то мы можем воспользоваться сферой, так же как и в предыдущих главах.
2. Кроме того, нам понадобится BRDF-текстура для вычисления диффузного цвета. BRDF-текстура, поставляемая с книгой доступна по адресу: http://www.packtpub.com/support. Используемая BRDF-текстура должна имитировать цвет кожи для различных тонов кожи. В данном случае мы будем моделировать светлую кожу, поэтому мы используем следующую BRDF-текстуру.

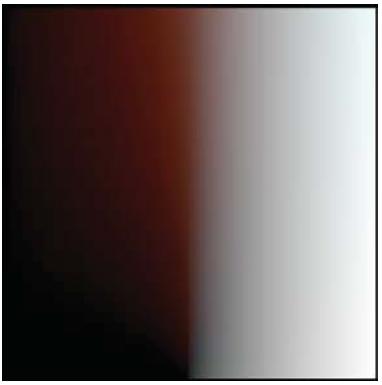

Рис. 5.11. BRDF-текстура для кожи

Как это сделать...

Давайте взглянем на код нашего шейдера. Мы пройдёмся по каждому блоку кода, а потом объясним ключевые моменты в следующей секции. Выполните следующие действия:

1. Для начала нам потребуется добавить в блок свойств свойства, которые позволят применять различные настройки и передавать текстуры в шейдер. Как вы видите, свойств у нас становится довольно много. Возможно, сейчас будет хорошей идеей обсудить с художниками возможность упаковки нескольких текстур (контролирующих разные параметры) в одну, чтобы избавиться от некоторых текстурных свойств и слайдеров. Но в данный момент для наших целей это не важно.

```
Properties
{
  _MainTint ("Global Tint", Color) = (1,1,1,1)
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _BumpMap ("Normal Map", 2D) = "bump" {}
  _CurveScale ("Curvature Scale", Range(0.001, 0.09)) = 0.01
  _CurveAmount ("Curvature Amount", Range(0,1)) = 0.5
  _BumpBiass ("Normal Map Blur", Range(0, 5)) = 2.0
  _BRDF ("BRDF Ramp", 2D) = "white" {}
  _FresnelVal ("Fresnel Amount", Range(0.01, 0.3)) = 0.05
  _RimPower ("Rim Falloff", Range(0.5)) = 2
  _RimColor ("Rim Color", Color) = (1,1,1,1)
  _SpecIntensity ("Specular Intensity", Range(0, 1)) = 0.4
  _SpecWidth ("Specular Width", Range(0, 1)) = 0.2
}
```

2. После этого нам нужно будет объявить несколько инструкций `#pragma`, поскольку этот шейдер будет довольно требователен к ресурсам и ему понадобятся некоторые платформозависимые особенности CGFX. Поэтому, чтобы избежать ошибок компилятора? скопируйте нижеприведённый код. Его объяснение мы дадим в следующей секции.

```
CGPROGRAM
#pragma surface surf SkinShader
#pragma target 3.0
#pragma only_renderers d3d9
```

3. Нашему шейдеру потребуется доступ к данным, которые пользователь выставляет в свойствах шейдера. Для этого нам нужно объявить соответствующие переменные в блоке `SubShader`.

```
sampler2D _MainTex;
sampler2D _BumpMap;
sampler2D _BRDF;
float4 _MainTint;
float4 _RimColor;
float _CurveScale;
float _BumpBiass;
float _CurveAmount;
float _FresnelVal;
float _RimPower;
float _SpecIntensity;
float _SpecWidth;
```

4. Чтобы мы могли использовать возможности поверхностных шейдеров на полную катушку, нам потребуется объявить нашу собственную структуру `SurfaceOutput`. С её помощью мы сможем обмениваться данными между нашей функцией

освещения и нашей поверхностной функцией. Если бы мы использовали встроенную структуру `SurfaceOutput`, то мы не смогли бы передать нашей функции освещения ни размытые нормали, ни значения кривизны поверхности, которые мы рассчитываем попиксельно.

```
struct SurfaceOutputSkin
{
 fixed3 Albedo;
 fixed3 Normal;
 fixed3 Emission;
 fixed3 Specular;
 fixed Gloss;
 fixed Alpha;
 float Curvature;
 fixed3 BlurredNormals;
};
```

5. Чтобы закончить основную часть нашего шейдера, нам потребуется объявить структуру `Input` и определить в ней нужные поля. В данном случае нам будут нужны глобальные координаты каждой вершины модели, нормали вершин в глобальных координатах, а поскольку мы используем в шейдере карту нормалей, то нам также нужно добавить макрос `INTERNAL_DATA`, чтобы мы смогли получить нормали после применения карты нормалей к поверхности.

```
struct Input
{
 float2 uv_MainTex;
 float3 worldPos;
 float3 worldNormal;
 INTERNAL_DATA
};
```

6. Подготовив эти данные, мы можем перейти к написанию нашей функции освещения. Начнём мы с того, что объявим функцию модели освещения, которую назовём `LightingSkinShader()`.

```
inline fixed4 LightingSkinShader(SurfaceOutputSkin s, fixed3
lightDir, fixed3 viewDir, fixed atten)
{

}
```

7. Теперь мы можем добавить в нашу модель освещения необходимые вычисления, чтобы получить правильное освещение кожи. Для начала мы приведём наши векторы в порядок, нормализо-

вав их, чтобы в дальнейшем работать с единичными векторами. Поэтому добавьте в функцию освещения следующий код.

```
//Обработаем векторы освещения
viewDir = normalize(viewDir);
lightDir = normalize(lightDir);
s.Normal = normalize(s.Normal);
float NdotL = dot(s.BlurredNormals, lightDir);
float3 halfVec = normalize(lightDir + viewDir);
```

8. После того как наши векторы будут готовы, мы можем сформировать значения, используемые для получения цвета из BRDF-текстуры.

```
//Создадим BRDF и имитируемый SSS
float3 brdf = tex2D(_BRDF, float2((NdotL * 0.5 + 0.5)* atten,
s.Curvature)).rgb;
```

9. Далее мы займемся эффектом Френеля и задней подсветкой.

```
//Добавим эффект Френеля и заднюю подсветку
float fresnel = saturate(pow(1 - dot(viewDir, halfVec),5.0));
fresnel += _FresnelVal * (1 - fresnel);
float rim = saturate(pow(1 - dot(viewDir, s.BlurredNormals),
_RimPower)) * fresnel;
```

10. После этого мы посчитаем интенсивность блика, точно так же, как мы делали в главе 3 «Пусть ваши игры засияют отражённым светом».

```
//Создадим блик
float specBase = max(0, dot(s.Normal, halfVec));
float spec = pow(specBase, s.Specular*128.0) * s.Gloss;
```

11. Теперь когда мы вычислили все необходимые значения для нашей модели освещения, мы можем объединить их и передать результат.

```
//Итоговый цвет
fixed4 c;
c.rgb = (s.Albedo * brdf * _LightColor0.rgb * atten) +
(spec + (rim * _RimColor));
c.a = 1.0;
return c;
```

12. И наконец, мы перейдём к функции `surf()`, в которой мы получаем данные из текстур, вычисляем размытые нормали и генерируем значение кривизны поверхности для нашей модели на основе карты нормалей.

```
void surf(Input IN, inout SurfaceOutputSkin o)
```

```
{
  //Получим данные из текстур
  half4 c = tex2D(_MainTex, IN.uv_MainTex);
  fixed3 normals = UnpackNormal(tex2D(_BumpMap,
IN.uv_MainTex));
  fixed3 normalBlur = UnpackNormal(tex2Dbias(_BumpMap,
float4 (IN.uv_MainTex, 0.0, _BumpBiass)));

  //Вычислим кривизну поверхности
  float curvature = length(fwidth(WorldNormalVector(IN, normalBlur)))
                 / length(fwidth(IN.worldPos)) * _CurveScale;

  //Добавим вычисленные данные в структуру SurfaceOutput
  o.Normal = normals;
  o.BlurredNormals = normalBlur;
  o.Albedo = c.rgb * _MainTint;
  o.Curvature = curvature;
  o.Specular = _SpecWidth;
  o.Gloss = _SpecIntensity;
  o.Alpha = c.a;
}
```

После того как мы собрали воедино все части нашего шейдера, результат его работы должен выглядеть как на следующем скриншоте.

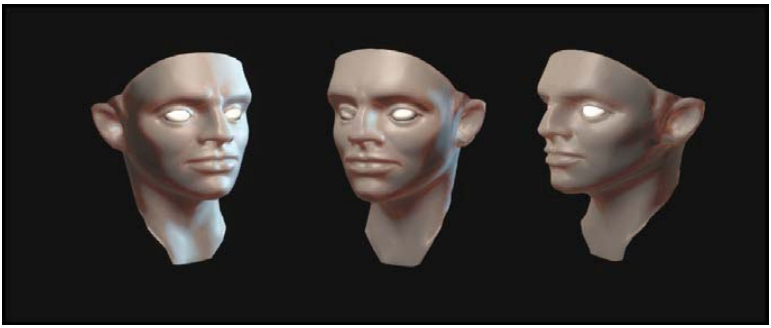

Рис. 5.12. Результат работы шейдера кожи

Как это работает...

Код большей части шейдера нам уже знаком, но есть и несколько новых для нас моментов. К примеру, мы объявили новый тип структуры `SurfaceOutputSkin`, и хотя мы её уже видели, сейчас мы всё равно остановимся на ней подробнее.

Структура `SurfaceOutputSkin` – это наша собственная кастомная структура, с помощью которой мы можем передавать данные из функ-

ции `surf()` в функцию освещения. Можно сказать, что это своего рода транспортный механизм. Когда мы присваиваем ей значения внутри функции `surf()`, эти значения сохраняются в переменных структуры. Эти данные мы можем использовать внутри функции освещения для выполнения попиксельных расчётов освещения.

Ещё один новый для нас элемент – вычисление кривизны поверхности. Проще говоря, мы измеряем величину изменения для нормалей поверхности. Поэтому при изменении кривизны поверхности изменяется и угол между нормалями поверхности. Мы можем использовать эти данные для поиска областей с наибольшей кривизной и вычислить соответствующее черно-белое значение.

В коде вычисления кривизны поверхности вы, вероятно, заметили две новые встроенные в CGFX функции, с помощью которых мы получаем необходимые данные, чтобы найти изменения вектора кривизны. Первая функция – `fwidth()`, она принимает один векторный параметр и возвращает скорость изменения этого вектора в экранных координатах. Поэтому в результате мы получаем вектор, который отражает, как быстро изменяются нормали к поверхности модели, то есть вектор кривизны поверхности. Следующая ссылка ведёт на описание этой функции в документации по Cg: http://http.developer.nvidia.com/Cg/fwidth.html.

С помощью стандартной функции Cg `fwidth()` мы можем получить информацию о кривизне поверхности нашей модели.

Рис. 5.13. Результат вычисления вектора кривизны поверхности модели

Нам не нужен весь вектор целиком, нам потребуется лишь его модуль для каждого пикселя. Поэтому мы можем воспользоваться функцией `length()`, которая вернёт нам длину вектора как значение

с плавающей точкой. Следующая ссылка ведёт на описание функции `length` в документации по Cg: http://http.developer.nvidia.com/Cg/length.html.

Таким образом, для каждого пикселя мы находим модуль вектора кривизны поверхности, который мы будем использовать при сэмплинге BRDF-текстуры.

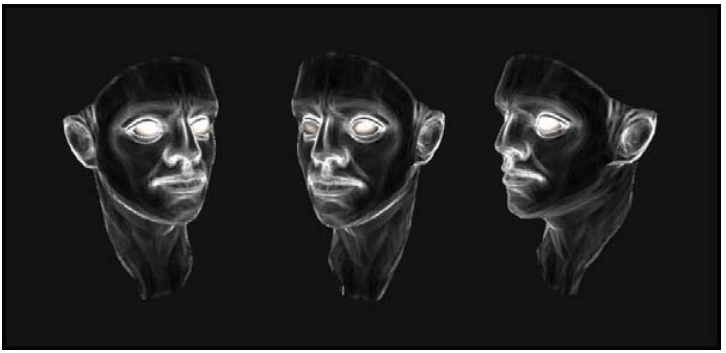

Рис. 5.14. Результат вычисления модуля вектора кривизны поверхности

После этого, мы вычисляем модуль кривизны по поверхности модели, для этого мы делим полученный модуль кривизны в экранных координатах на модуль изменения координат по модели (опять же в экранных координатах) и умножаем на параметр `CurveScale`, чтобы, контролируя его, мы могли изменять интенсивность эффекта кривизны.

Финальный результат вычисления кривизны поверхности в нашем шейдере кожи показан на следующем скриншоте.

Рис. 5.15. Итоговый результат вычислений кривизны поверхности модели для шейдера кожи

И ещё одна новая для нас функция, которую мы используем для получения мягкого диффузного освещения кожи, – это функция `tex2Dbias()`. С помощью неё мы можем сдвигать или изменять текущий mip-уровень в сторону уменьшения или увеличения, используя данные из свойств шейдера, в результате чего мы можем контролировать, насколько размытой будет текстура. На самом деле мы не осуществляем размытие текстуры как таковой, вместо этого мы выбираем более низкий mip-уровень в нашей текстуре. Чтобы узнать больше про mip-текстурирование, мип-мапы и как их генерировать, обратитесь к документации Unity: http://docs.unity3d.com/Documentation/Manual/Textures.html.

Но это ещё не всё...

Эта конкретная реализация шейдера кожи была навеяна парой шейдеров, которые мы увидели в Интернете. Поэтому мы считаем, что будет правильно упомянуть их здесь:

- **Unity forums:** http://forum.unity3d.com/threads/131626-Grittyrealistic-skin-shader;
- **Skin Shader 3:** http://wiki.unity3d.com/index.php?title=Skin_Shader_3.

Шейдер ткани

Шейдинг ткани – это еще одна довольно распространённая задача в гейм-девелопменте и интерактивной real-time-визуализации. Для её решения нужно понимать, как волокна ткани рассеивают свет по поверхности объекта и как от этого получается характерная для ткани текстура. При этом шейдинг ткани очень сильно зависит от направления взгляда, поэтому мы рассмотрим новые приёмы, которые мы сможем использовать для имитации эффекта отражения света по поверхности объекта, а также для создаваемого маленькими волокнами очень отчётливого эффекта задней подсветки.

При работе с этим шейдером мы познакомимся с концепцией детализированных карт нормалей и детализированных текстур. Объединяя две карты нормалей, мы можем добиться более высокой детализированности, чем мы бы получили, работая с одной текстурой 2048×2048. Эта техника поможет нам имитировать микроуровень ямок на поверхности объекта, с помощью которых мы сможем рассеивать бликовую компоненту освещения по более широкой площади.

Ниже приводится итоговый результат работы шейдера ткани, который мы подготовим в этом рецепте.

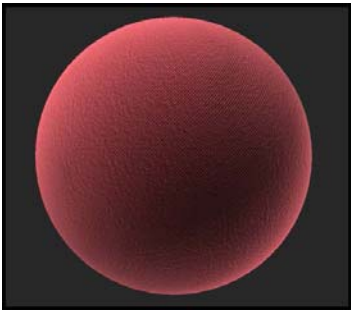

Рис. 5.16. Итоговый результат работы шейдера ткани

Подготовка

Для этого шейдера мы будем использовать три разных типа текстур, чтобы имитировать похожую на ткань поверхность:

1. Детализированную карту нормалей. Она будет дублироваться по поверхности для имитации микроструктуры ткани.
2. Карту вариаций нормалей для создания вариаций в структуре, с помощью которой ткань будет выглядеть менее однообразной и более поношенной.
3. Детализированную диффузную текстуру, которую мы сможем перемножить с базовым цветом для придания цвету ткани большей глубины и реализма, что подчеркнёт её структуру.

На следующем скриншоте представлены три текстуры, которыми мы воспользуемся в этом рецепте. Они также приводятся на странице, посвящённой этой книге, найти вы её можете по адресу: www.packtpub.com/support.

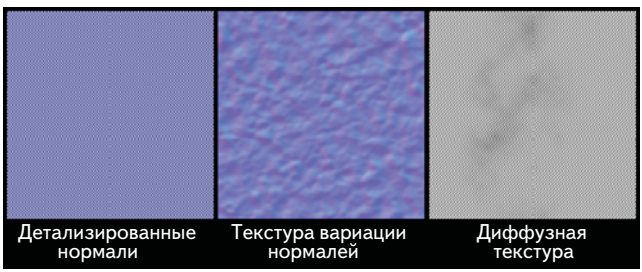

Рис. 5.17. Используемые в рецепте текстуры

Для того чтобы завершить подготовительные работы, нам нужно создать сцену с объектом и направленным источником света. После этого создайте новый шейдер и материал, который мы назначим объекту.

Как это сделать...

Начнём наш шейдер с блока свойств. Выполните следующие действия:

1. Нашему шейдеру потребуется совсем немного свойств – для контроля за тем, какие текстуры мы используем и как будут выглядеть затухание бликов и отражения по Френелю.

```
Properties
{
  _MainTint ("Global Tint", Color) = (1,1,1,1)
  _BumpMap ("Normal Map", 2D) = "bump" {}
  _DetailBump ("Detail Normal Map", 2D) = "bump" {}
  _DetailTex ("Fabric Weave", 2D) = "white" {}
  _FresnelColor ("Fresnel Color", Color) = (1,1,1,1)
  _FresnelPower ("Fresnel Power", Range(0, 12)) = 3
  _RimPower ("Rim Falloff", Range(0, 12)) = 3
  _SpecIntensity ("Specular Intensity", Range(0, 1)) = 0.2
  _SpecWidth ("Specular Width", Range(0, 1)) = 0.2

}
```

2. Поскольку мы хотим иметь возможность полностью контролировать, как свет взаимодействует с нашей тканью, нам потребуется объявить новую модель освещения в секции `#pragma` и указать компилятору использовать шейдерную модель 3.0. Для того чтобы использовать свою модель освещения, нам нужно объявить её имя в директиве `#pragma`.

```
CGPROGRAM
#pragma surface surf Velvet
#pragma target 3.0
```

3. Чтобы мы смогли воспользоваться данными, вводимыми в блок свойств, нам потребуется объявить переменные с такими же именами в блоке `SubShader`.

```
sampler2D _BumpMap;
sampler2D _DetailBump;
sampler2D _DetailTex;
float4 _MainTint;
float4 _FresnelColor;
```

```
float _FresnelPower;
float _RimPower;
float _SpecIntensity;
float _SpecWidth;
```

4. Чтобы можно было контролировать параметры тайлинга каждой из наших детализированных текстур по отдельности, мы должны объявить разные UV-параметры в структуре `Input`. Связь UV-координат с текстурой будет создана автоматически, если вы добавите в имя переменной `"uv"` перед именем текстуры.

```
struct Input
{
  float2 uv_BumpMap;
  float2 uv_DetailBump;
  float2 uv_DetailTex;
};
```

5. Теперь нам нужно написать функцию, которая станет нашей моделью освещения. Начнём написание функции освещения с задания её сигнатуры. Нам нужно выбрать ту, что содержит `viewDir`, потому что шейдинг поверхности ткани зависит от направления взгляда.

```
inline fixed4 LightingVelvet (SurfaceOutput s, fixed3
lightDir, half3 viewDir, fixed atten)
{

}
```

Всегда стоит начинать с определения необходимых для вычисления освещения векторов. Впоследствии нам не придется заботиться об их нормализации при каждом использовании в других частях кода. Давайте добавим вычисление этих векторов в начало функции модели освещения:

```
//Вычислим векторы освещения
viewDir = normalize(viewDir);
lightDir = normalize(lightDir);
half3 halfVec = normalize(lightDir + viewDir);
fixed NdotL = max(0, dot(s.Normal, lightDir));
```

6. Следующая задача – расчёт Specular-компонента. Добавьте следующий код сразу после определения векторов:

```
//Вычислим блик
float NdotH = max(0, dot(s.Normal, halfVec));
float spec = pow(NdotH, s.Specular*128.0) * s.Gloss;
```

Шейдинг ткани очень сильно зависит от направления взгляда на поверхность. Чем более острый угол взгляда к поверхности, тем больше её волокна улавливают свет за ними и увеличивают интенсивность Specular-компонента.

```
//Создадим освещение по Френелю
float NdotV = pow(1 - max(0, dot(halfVec, viewDir)), _FresnelPower);
float NdotE = pow(1 - max(0, dot(s.Normal, viewDir)), _RimPower);
float finalSpecMask = NdotE * HdotV;
```

7. После того как мы закончили наши основные вычисления освещения, нам нужно вернуть из нашей функции итоговый цвет. Дополните модель освещения, добавив следующий код сразу после вычисления эффекта Френеля.

```
//Вернём итоговый цвет
fixed4 c;
c.rgb = (s.Albedo * NdotL * _LightColor0.rgb)
  + (spec * (finalSpecMask * _FresnelColor)) * (atten * 2);
c.a = 1.0;
return c;
```

8. Закончим мы наш шейдер, дописав функцию `surf()`. Нам нужно лишь распаковать наши карты нормалей и перенаправить данные в структуру `SurfaceOutput`.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_DetailTex, IN.uv_DetailTex);
  fixed3 normals = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap)).rgb;
  fixed3 detailNormals = UnpackNormal(tex2D(_DetailBump,
IN.uv_DetailBump)).rgb;
  fixed3 finalNormals = float3(normals.x + detailNormals.x,
                          normals.y + detailNormals.y,
                          normals.z + detialNormals.z);

  o.Normal = normalize(finalNormals);
  o.Specular = _SpecWidth;
  o.Gloss = _SpecIntensity;
  o.Albedo = c.rgb * _MainTint;
  o.Alpha = c.a;
}
```

Следующий скриншот демонстрирует результат работы нашего шейдера ткани на модели, имитирующей ткань.

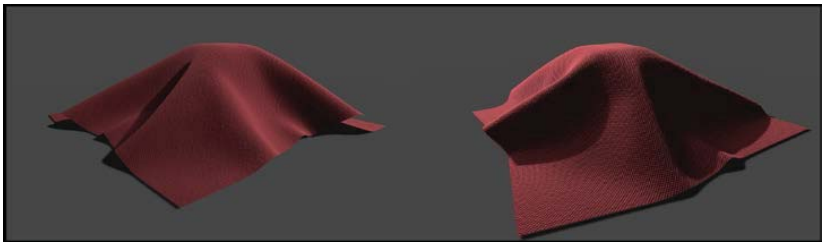

Рис. 5.18. Результат работы шейдера ткани на модели, имитирующей ткань

Как это работает...

На самом деле наш шейдер ткани не такой уж и сложный. В нём мы выполняем очень простые операции со светом, но порой это всё, что необходимо для вашего шейдера. Перед тем, как приступать к созданию шейдера, стоит пристальнее взглянуть на имитируемую поверхность, декомпозировать её на компоненты, а потом запрограммировать их один за другим. Весь фокус будет заключаться в том, как вы будете комбинировать различные вычисления, совсем как при смешении слоёв в Photoshop.

Новая техника, которую мы применили в нашем шейдере ткани, – это комбинирование двух карт нормалей с различной плотностью тайлинга. В соответствии с основами линейной алгебры при сложении двух векторов мы получаем третий. С нашими картами нормалей мы так и поступаем. К карте вариаций нормалей, из которой мы получаем вектор с помощью функции `UnpackNormal()`, мы прибавляем вектор нормали из детализированной карты нормалей. В результате мы получаем новую карту нормалей. После этого мы нормализуем наш итоговый вектор, чтобы он вновь был в диапазоне от 0 до 1. Если мы этого не сделаем, то наша карта нормалей будет выглядеть весьма помятой и визуально неправильной.

И наконец, комбинация вычислений освещения по Френелю и блика позволяет нам создать эффект маленьких волокон ткани, улавливающих свет на острых углах к поверхности объекта.

ГЛАВА 6
Прозрачность

В этой главе вы узнаете о:

- создании прозрачности с помощью параметра `alpha`;
- прозрачном cutoff-шейдере;
- сортировке объектов с помощью очередей рендеринга;
- GUI и прозрачности.

Введение

Реализация прозрачности в шейдерах может оказаться довольно нетривиальной задачей. Но с помощью поверхностных шейдеров Unity мы легко можем сделать шейдер, реализующий эффект полной прозрачности для таких поверхностей, как стекло, или частичной прозрачности для волос и листвы.

Мы начнём с создания простых прозрачных шейдеров, а потом остановимся на том, как прозрачность влияет на порядок отрисовки объектов.

Создание прозрачности с помощью параметра alpha

Начнём мы с того, что разберёмся, как нужно изменить шейдер, чтобы он стал прозрачным. И опять Unity здесь всё делает за нас.

Нам всего лишь потребуется использовать в секции `#pragma` нашего шейдера параметр `alpha`. Эта инструкция говорит Unity, что мы собираемся использовать прозрачность в шейдере. Однако в создании прозрачных шейдеров есть несколько нюансов, так как важным становится порядок отрисовки элементов. В этом рецепте мы рассмотрим основные моменты, с которыми нам придётся столкнуться при создании прозрачного объекта на нашей сцене. Другие особенности прозрачности мы рассмотрим в последующих рецептах.

Подготовка

Для начала работы с этим рецептом нам потребуется подготовить несколько ресурсов и создать новую сцену в редакторе Unity. Давайте выполним следующие действия, чтобы приготовиться к написанию шейдера:

1. Создайте новую сцену и поместите на неё сферу, плоскость и источник направленного света.
2. После этого нам потребуется создать новый шейдер и новый материал. Шейдер будет необходимо назначить материалу, а материал – сфере.
3. Далее нам будет нужна текстура, на основе которой мы будем принимать решение о том, какая часть нашего объекта на сцене должна быть прозрачна, а какая – нет.

На следующем рисунке приводится пример текстуры, которую мы будем использовать для этого рецепта. На этой текстуре есть отдельно красный, зеленый, синий цвета и комбинация всех трех – белый цвет. Таким образом, в качестве маски прозрачности в нашем примере мы можем использовать любой из каналов RGB со значениями от 0 до 1, где 0 – полностью прозрачный, а 1 – полностью непрозрачный.

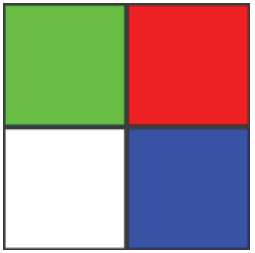

Рис. 6.1. Текстура, использованная нами для этого рецепта

Как это сделать...

Подготовив ассеты, мы можем приступить к написанию нашего поверхностного шейдера для создания эффекта прозрачности.

Выполните следующие действия:

1. Давайте добавим в блок свойств новое свойство, с помощью которого мы будем контролировать уровень прозрачности.

```
Properties
{
```

```
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _TransVal ("Transparency Value", Range(0,1)) = 0.5
}
```

2. После этого нам потребуется добавить в секцию `#pragma` новый параметр, который мы ещё не видели – параметр `alpha`.

```
CGPROGRAM
#pragma surface surf Lambert alpha
```

3. После этого мы добавим в функцию `surf()` нашего шейдера строку, в которой будем присваивать значение параметру `o.Alpha`.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex);
  o.Albedo = c.rgb;
  o.Alpha = c.r * _TransVal;
}
```

Следующий скриншот демонстрирует наш прозрачный шейдер в редакторе Unity.

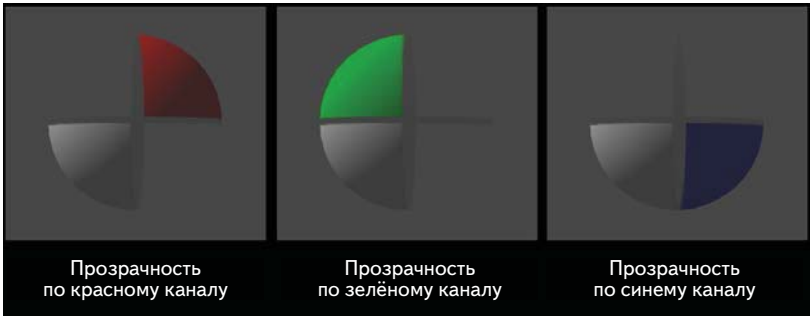

Рис. 6.2. Результат работы шейдера

Как это работает...

Как вы, наверно заметили, с помощью поверхностых шейдеров в Unity сделать эффект прозрачности не составляет никакого труда. Для такого шейдера ключевыми являются два элемента: параметр `alpha` из секции `#pragma` и значение `Alpha` в структуре `SurfaceOutput`, которое и определяет, насколько прозрачным должен быть каждый конкретный пиксель.

Объявление параметра `alpha` в секции `#pragma` говорит Unity, что на экране сейчас нужно будет отрендерить прозрачную поверхность.

Всё, что нужно сделать после этого, – это присвоить значение `o.Alpha` из нашей структуры `SurfaceOutput` (в данном случае – встроенной) – значение от 0 до 1 для каждого пикселя. Говоря на языке цвета, белый (1) будет соответствовать полностью непрозрачной поверхности, а чёрный (0) – полностью прозрачной.

Данная реализация прозрачности является наиболее простой. По мере нашего продвижения по этой главе мы рассмотрим характерные ситуации, которые возникают при использовании полупрозрачных шейдеров в real-time-движках, таких как Unity.

Прозрачный cutoff-шейдер

Unity предоставляет нам ещё один параметр в секции `#pragma`, с помощью которого можно создать более простой эффект прозрачности, известный как отсекаемая прозрачность, или cutoff-прозрачность. Этот тип прозрачности использует некоторое контрольное значение, чтобы определить, нужно ли рисовать данный конкретный пиксель, таким образом, в нашем шейдере будут только либо полностью прозрачные, либо полностью непрозрачные пиксели, в отличие от предыдущего рецепта, в котором в шейдере мы могли использовать весь диапазон оттенков серого для управления прозрачностью объекта. Такой шейдер называется полупрозрачным шейдером.

Но не будем отвлекаться, вернёмся к написанию этого шейдера в Unity.

Подготовка

Давайте начнём работу над нашим шейдером со следующих действий:

1. Создайте новую сцену и поместите на ней обычную сферу и направленный источник света.
2. Далее создайте новый шейдер и новый материал.
3. Назначьте шейдер материалу, а материал – сфере на сцене.
4. После этого нам потребуется новая текстура. Нагляднее будет использовать черно-белую текстуру, таким образом будет более заметен эффект от изменения величины отсечения.

На следующем скриншоте показана текстура, которую мы использовали в этом рецепте. Для её создания мы воспользовались фильтром Render Difference Clouds в Photoshop. Сделанная нами текстура доступна по адресу www.packtpub.com/support.

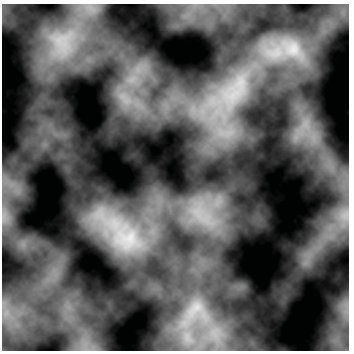

Рис. 6.3. Текстура, использованная нами в этом рецепте

Как это сделать...

Теперь, когда мы подготовили нашу сцену и необходимые ассеты, давайте посмотрим на код нашего шейдера.

Выполните следующие действия:

1. Для начала добавьте в блок свойств свойство, которое позволит нам управлять величиной отсечения в нашем шейдере.

```
Properties
{
  _MainTex ("Base (RGB)", 2D = "white" {}
  _Cutoff ("Cutoff Value", Range(0,1)) = 0.5
}
```

2. После этого нам нужно указать, что мы делаем cutoff-шейдер.

```
CGPROGRAM
#pragma surface surf Lambert alphatest:_Cutoff
```

3. И наконец, мы присваиваем `o.Alpha` значения для каждого пикселя поверхности.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex);
  o.Albedo = c.rgb;
  o.Alpha = c.r;
}
```

Следующий скриншот демонстрирует результат применения нашего cutoff-шейдера при различных значениях cutoff-слайдера в диапазоне между 0 и 1.

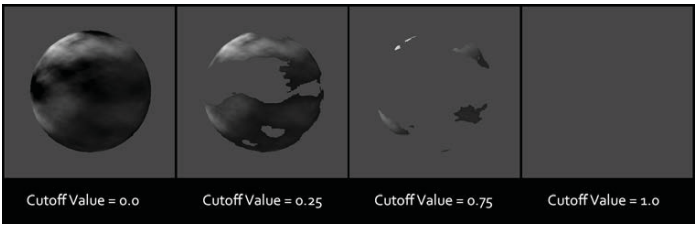

Рис. 6.4. Результат работы cutoff-шейдера

Как это работает...

Unity предоставляет нам довольно много параметров, которые мы можем использовать в секции `#pragma`. С помощью них мы можем изменять и оптимизировать поведение нашего поверхностного шейдера. Это ещё одна причина, по которой поверхностные шейдеры являются таким мощным и эффективным инструментом для итеративного написания шейдеров.

Наш отсекающий шейдер использует новый параметр директивы `#pragma – alphatest:VariableName`. Эта инструкция переводит наш шейдер в режим упрощённой версии прозрачности. В этом режиме вместо значения от 0 до 1, соответствующего прозрачности пикселя, рендерится пиксель или отбрасывается, определяется только значением переменной `_Cutoff`. При таком подходе, если мы на закладке **Инспектора** присвоим переменной `_Cutoff` значение 0,4, все значения меньше 0,4 будут считаться прозрачными, а значения больше 0,4 – непрозрачными.

Эта техника хорошо подходит, если мы беспокоимся о производительности, поскольку блендинг полупрозрачных объектов – гораздо более ресурсоемкая задача, чем простая cutoff-прозрачность. Однако для мобильных устройств наблюдается обратная закономерность, поскольку сэмплинг-текстуры являются достаточно дорогой операцией для этих маленьких GPU. Поэтому, если вы используете Unity для разработки приложения под мобильные устройства, используйте полупрозрачные шейдеры, а технику отсекаемой прозрачности применяйте лишь в случае необходимости.

Сортировка объектов с помощью очередей рендеринга

Для того чтобы действительно понять эффект прозрачности, нам потребуется остановиться на сортировке по глубине, или проще го-

воря – очерёдности отрисовки объектов. В Unity мы можем контролировать порядок отрисовки объектов на экране, что предоставляет нам больше возможностей контроля за тем, какие объекты будут отрисованы поверх каких. Вы можете думать об очерёдности отрисовки как о слоях в Photoshop. Очерёдность отрисовки особенно важна при работе с прозрачностью или с такими элементами, как объекты пользовательского интерфейса.

В этом рецепте вы увидите, как можно использовать данный метод слоёв для рендеринга ваших объектов с помощью встроенных в Unity тегов. Этот момент очень важен, поскольку у вас появится больше возможностей для контроля за рендерингом объектов в игре.

Подготовка

Для начала работы нам потребуется создать необходимые ассеты, чтобы мы могли увидеть, как очерёдность отрисовки объектов в Unity предоставляет нам больше гибкости и контроля за рендерингом.

Выполните следующие действия:

1. Создайте новую сцену и разместите на ней несколько сфер так, чтобы они были выстроены в ряд по любой оси. Наша цель – понять, как мы можем рисовать одни объекты поверх других, вне зависимости от их реального расположения в пространстве.
2. Чтобы увидеть эффект изменения порядка отрисовки объектов, нам потребуются, по крайней мере, два шейдера. Поэтому давайте создадим два новых шейдера и соответственно их назовём. Мы назвали наши шейдеры `Depth001` и `Depth002`.
3. Ваша сцена должна выглядеть, как на следующем скриншоте (рис. 6.5). Такой сетап позволит нам поэкспериментировать с очерёдностью отрисовки объектов.

Как это сделать…

Код, который нам потребуется написать для реализации этого приёма, на самом деле весьма прост. В нём будут присутствовать всего две новые для нас строчки.

Выполните следующую последовательность действий:

1. Сначала нам потребуется определить, в какой очереди рендеринга наш объект будет рисоваться, – для этого нам нужно будет модифицировать блок `Tags{}` внутри блока `SubShader`.

```
Tags { "Queue" = "Geometry-20"}
```

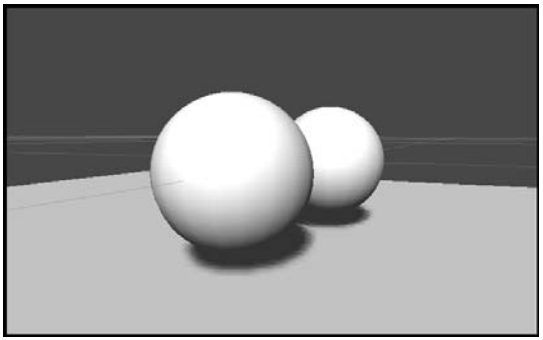

Рис. 6.5. Сцена демонстрирующая порядок отрисовки объектов

2. Далее нам нужно будет дать инструкции Unity, что мы хотим контролировать очерёдность отрисовки данного объекта и что мы не хотим, чтобы он записывался в буфер глубины. Добавьте следующую строку кода сразу после добавленной нами ранее строки `Tags{}`.

```
ZWrite Off
```

3. После того как мы добавили этот код в наш шейдер, нам нужно будет сохранить его, а потом вернуться в Unity, чтобы шейдер скомпилировался заново. После того как шейдер скомпилируется, вы заметите, что одна из сфер рисуется позади всех объектов, даже несмотря на то, что её действительное положение в пространстве находится перед всеми другими объектами. Следующий скриншот демонстрирует результат применения шейдера сортировки глубины.

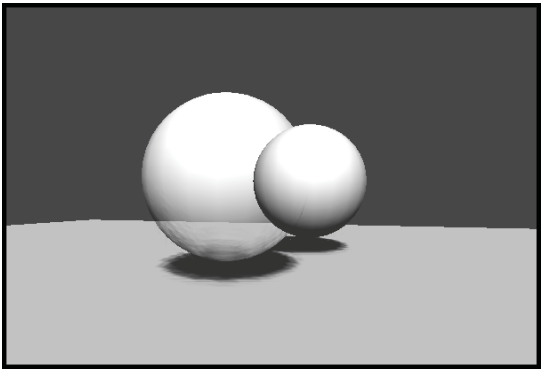

Рис. 6.6. Результат применения шейдера глубины

Как это работает...

По умолчанию, Unity сортирует объекты по их удалённости от камеры. Поэтому если объект будет расположен ближе к камере, то он будет отрисован поверх всех остальных объектов, расположенных дальше от камеры. В большинстве случаев при разработке игр такое поведение всех устраивает, однако всё же бывают ситуации, в которых вам потребуется возможность контролировать сортировку ваших объектов на сцене. Это можно делать с помощью блока `Tags{}`.

В Unity есть дефолтные очереди отрисовки, каждой из которых сопоставлено число, которое определяет, когда объект должен быть отрисован на экране. Эти встроенные очереди называются **Background**, **Geometry**, **AlphaTest**, **Transparent** и **Overlay**. Очереди были добавлены не просто так: их цель – облегчить процесс написания шейдеров, а также упростить взаимодействие с real-time-рендерером. Описание очередей отрисовки приводится в следующей таблице.

Таблица 6.1. Описание очередей отрисовки

Название	Описание	Значение
Background	Эта очередь рендерится первой. Используется для скайбокса и т. п.	1000
Geometry	Эта очередь используется по умолчанию. Она подходит для большинства объектов. Она используется для рендеринга непрозрачных объектов	2000
AlphaTest	Alpha-tested (cutoff) геометрия использует эту очередь. Специальная очередь для объектов с cutoff-прозрачностью позволяет рендерить прозрачные объекты только после того, как отрисуются все непрозрачные объекты	2450
Transparent	Эта очередь отрисовывается после очередей Geometry и AlphaTest от самых дальних объектов к более близким к камере. Она предназначена для объектов с альфа-блендингом (для которых шейдеры не пишут в буфер глубины), например для стекла или эффектов частиц	3000
Overlay	Эта очередь отрисовки предназначена для оверлеев. Всё, что должно рендериться в последнюю очередь, идёт сюда, например блики камеры	4000

Так что, после того как вы определитесь с очередью отрисовки объекта, вам следует присвоить ему тег встроенной очереди. Наш шейдер использовал очередь Geometry, поэтому мы добавили `Tags{"Queue"= "Geometry"}`. Однако при этом мы хотим, чтобы наш объект рисовал-

ся до всех остальных объектов из очереди Geometry, но после очереди Background. Поэтому мы изменили блок `Tags{}` следующим образом: `Tags{"Queue"="Geometry-20"}`. Эта инструкция говорит Unity, что мы хотим, чтобы наш объект считался непрозрачным, но при этом он должен рендериться за всеми остальными непрозрачными объектами. Это становится понятнее, если считать, что Geometry, – это всего лишь число, которое указывает порядок рендеринга объектов. Соответственно, отняв от этого числа 20, мы сказали, что хотим, чтобы наш объект рендерился раньше.

Помимо этого, нам потребуется добавить тег `ZWrite` в блок `SubShader`. Таким образом, мы говорим Unity, что мы хотим сами контролировать сортировку по глубине данного объекта и что мы присвоили новое значение его очереди отрисовки. Так что мы просто устанавливаем `ZWrite` в `Off`.

GUI и прозрачность

Теперь, когда мы уже рассмотрели основы создания прозрачных шейдеров и научились контролировать очерёдность отрисовки объектов, давайте посмотрим на задачу из области практики, в которой нам потребуется использовать прозрачность и контролировать очерёдность отрисовки прозрачных объектов.

Создание GUI для Unity – это, несомненно, очень серьёзная задача. Можно использовать встроенные функции `OnGUI()` для создания элементов GUI с помощью набора прозрачных 2D-изображений, переложив на Unity задачу их отрисовки. Или же можно создать настоящую систему с 3D-элементами GUI, которые вы сможете увидеть на сцене в редакторе Unity. Мы пойдём по последнему пути. Нам потребуется использовать атлас с 2D-изображениями, которые мы поместим на 3D-объекты нашей сцены, чтобы таким образом создать элементы GUI для нашей игры.

Также в этом рецепте мы рассмотрим некоторые сложности, с которыми вы, вероятно, столкнётесь при использовании этого подхода, такие как, например, очерёдность отрисовки, и увидим, как с этими сложностями бороться.

Подготовка

В этом рецепте мы разберём пример очень простого GUI, для которого нам потребуется создать атлас с GUI-элементами для нашей сцены. При создании GUI в 3D обычно помещают текстуры всех элемен-

тов в одну текстуру (атлас), для того чтобы сэкономить на количестве используемых текстур. Это значит, что вся графика для кнопок, для иконок, а иногда даже для текста будет находиться в одной текстуре, альфа-канал которой показывает, где атлас должен быть полностью прозрачен, где непрозрачен, а где полупрозрачен. Взгляните на следующий скриншот, содержащий атлас, который мы будем использовать для этого рецепта.

Рис. 6.7. Атлас, использованный нами для этого рецепта

Давайте начнём создание наброска элементов GUI. В процессе работы с ним нам придётся продумать моменты, с которыми мы можем столкнуться при написании GUI в продакшене.

Выполните следующую последовательность действий:

1. Создайте GUI-атлас, похожий на изображённый на предыдущем скриншоте. Не забудьте добавить в текстуру альфа-канал.
2. Кроме этого, нам потребуется сделать немного простых геометрических объектов для нашего GUI. Мы воспользовались Maya, чтобы сделать меши, на которых будут располагаться GUI-элементы.
3. Создайте новую сцену и поместите на неё плоскость и направленный источник света.
4. Далее создайте новый шейдер и новый материал для нашего GUI, шейдер назначьте материалу.
5. Теперь, чтобы завершить подготовку, нам просто нужно назначить наш материал GUI-объектам на сцене.
6. После того как вы выполните предыдущие шаги, ваша сцена должна стать похожей на изображённую на следующем скриншоте. Вы можете использовать сцену, которая идёт вместе с этой книгой, но всегда более продуктивно упражняться на своих ассетах.

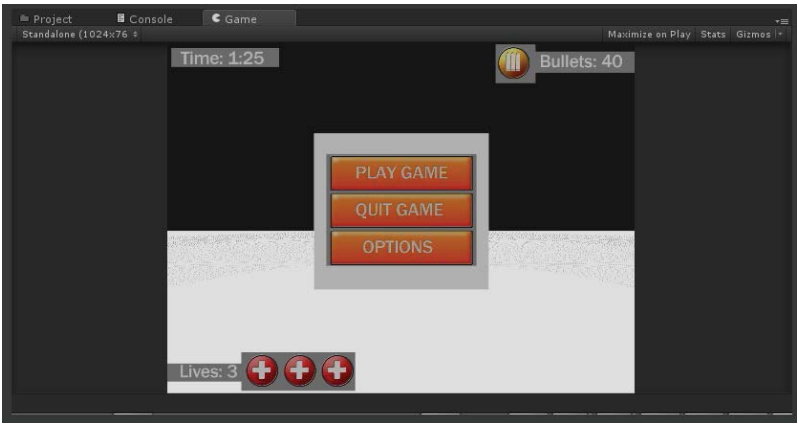

Рис. 6.8. Вид сцены с элементами GUI

Пока что то, что получилось, несильно радует взгляд, да и наша альфа не создаёт эффекта прозрачности, необходимого для настоящего игрового GUI. Нам потребуется создать собственный GUI-шейдер, чтобы наши элементы GUI стали более изящными. Давайте этим и займёмся.

Как это сделать...

Чтобы наши элементы GUI получили прозрачность, нам потребуется создать собственный шейдер, с помощью которого мы сможем указать Unity, что эти объекты должны быть прозрачными.

Выполните следующие действия:

1. Как обычно, нам нужно заполнить блок свойств соответствующими свойствами, чтобы мы смогли взаимодействовать с нашим шейдером из редактора Unity.

```
//После того как мы создадим наши свойства тут, мы сможем
//увидеть их на закладке инспектора.
Properties
{
  _GUITint ("GUI Tint", Color) = (1,1,1,1)
  _GUITex ("Base (RGB) Alpha (A)", 2D) = "white" {}
  _FadeValue ("Fade Value", Range(0,1)) = 1
}
```

2. После этого мы перейдём к `SubShader` блоку, а начнём с того, что зададим тип очереди отрисовки, который нам потребуется, и модель освещения. Мы используем новые теги, на которых

более подробно остановимся в следующей секции. Сейчас же добавьте следующий код в начало блока SubShader:

```
//Тут мы объявляем теги, чтобы сказать Unity, что это будет
//за шейдер.
Tags {"Queue"="Transparent" "IgnoreProjector"="True"
"RenderType"="Transparent"}
ZWrite Off
Cull Back
LOD 200
```

3. После того как мы объявили теги, нам нужно будет перейти к инструкциям #pragma и добавить нашу собственную модель освещения, а также добавить в код несколько, пока ещё новых для нас аргументов. Таким образом, мы сможем создать поверхность, на которую не действуют источники света на сцене (Unlit), и предоставить полный контроль атласу за тем, как выглядят элементы GUI.

```
//Определим модель освещения
CGPROGRAM
#pragma surface surf UnlitGUI alpha novertexlights
```

4. Далее нам потребуется связать переменные в нашем блоке свойств и переменные в блоке CGPROGRAM.

```
//Создадим связь между свойствами и переменными
sampler2D _GUITex;
float4 _GUITint;
float _FadeValue;
```

5. После того как эти шаги подготовки шейдера будут выполнены, мы перейдём к написанию модели освещения Unlit. Она будет достаточно простой, в ней мы будем передавать цветовые значения из атласа в нашу структуру SurfaceOutput.

```
//Unlit – модель освещения
inline fixed4 LightingUnlitGUI(SurfaceOutput s, fixed3
lightDir, fixed3 viewDir, fixed atten)
{
  fixed4 c;
  c.rgb = s.Albedo;
  c.a = s.Alpha;
  return c;
}
```

6. Не забывайте, что раз мы хотим использовать текстуры, то нам нужно передавать UV-координаты для них в структуре Input.

```
struct Input
{
  float uv_GUITex;
};
```

7. Далее мы просто получаем цвет и альфу из текстуры, после чего передаём их в структуру `SurfaceOutput` внутри нашей функции `surf()`.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 texColor = tex2D(_GUITex, IN.uv_GUITex);
  o.Albedo = texColor.rgb * _GUITint.rgb;
  o.Alpha = texColor.a * _FadeValue;
}
```

После того как вы закончите написание шейдера, вы должны увидеть результат, похожий на изображённый на следующем скриншоте, однако если вы используете вашу собственную геометрию и свой атлас, то сцена, конечно, будет выглядеть по-другому. Не считая этих возможных различий, шейдер должен создавать эффект прозрачной поверхности, на которую не действуют источники света.

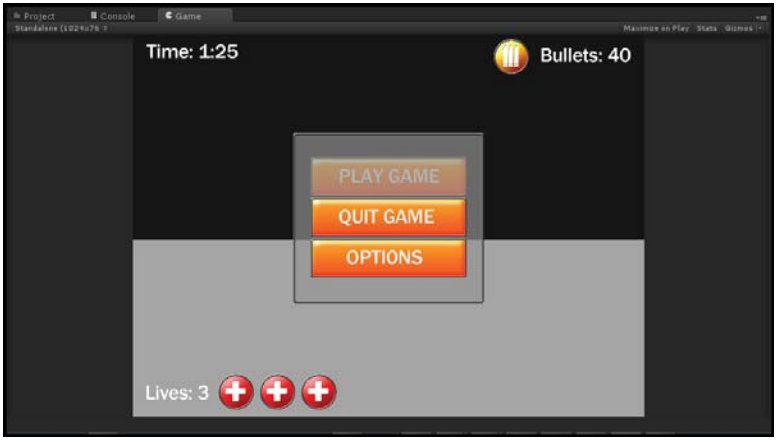

Рис. 6.9. Промежуточный результат работы шейдера

Вероятно, вы заметили, что в наш GUI закралась маленькая ошибка. Фоновая заливка меню рисуется поверх кнопки **Play Game**. Так получилось, потому что меши находятся очень близко, и Unity было тяжело определить, какой объект нужно рисовать первым. А поскольку очерёдность отрисовки опре-

делялась удалённостью от камеры, Unity рендерит фон меню поверх кнопки.

8. Чтобы исправить эту неточность, нам нужно будет изменить очередь отрисовки для каждого материала. Мы не можем просто поменять очередь отрисовки в шейдере, поскольку тогда нам пришлось бы писать шейдер для каждого уровня очереди. Нам нужна возможность индивидуального контроля наших материалов. Поэтому мы напишем небольшой скрипт на C#, который поможет нам в этом деле. Давайте приступим к этому.
9. Для начала создайте новый C#-скрипт.
10. После того как скрипт был создан, кликните по нему два раза, чтобы открыть его в MonoDevelop.
11. Первым делом нам нужно дать указание скрипту выполняться в редакторе, чтобы мы могли в реальном времени наблюдать эффекты от изменения очереди отрисовки. Для этого нам нужно добавить атрибут `[ExecuteInEditMode]` перед объявлением класса.

```
[ExecuteInEditMode]
public class ObjectRenderQueue : MonoBehaviour
```

12. Чтобы мы смогли в реальном времени изменять очерёдность отрисовки, нам потребуется создать переменную, значение которой мы будем менять с закладки **Инспектора**. Поэтому мы объявим новую переменную с именем `queueValue` и сделаем её публичной, для того чтобы она отображалась в **Инспекторе**.

```
//Эта строка кода позволит нам изменять величину очереди
//в редакторе
public int queueValue = 2000;
```

13. После этого мы перейдём к функции `Update()`, где первым делом нам нужно будет проверить, что на объекте, к которому прицеплен этот скрипт, есть материал.

```
//Сначала проверим, есть ли у объекта материал.
Material curMaterial = renderer.sharedMaterial;
```

14. Далее с помощью оператора `if()` мы проверяем, что наша переменная `curMaterial` содержит непустую ссылку на материал. Это мы делаем, чтобы подстраховаться от всяких ненужных сообщений об ошибках, появляющихся в консоли.

```
if (curMaterial != null)
{
```

```
  //Если есть материал, устанавливаем значение очереди
  curMaterial.renderQueue = queueValue;
}
else
{
  //Если материал найти не удалось, выведем отладочное
  //сообщение.
  Debug.LogWarning(transform.name +
                   ": Cannot find a material to set the render
queue!");
}
```

После того как вы написали скрипт, вы сможете его назначить любому элементу GUI, чтобы подкорректировать в редакторе значение очереди на лету и тут же увидеть изменения в отрисовке. Теперь наша GUI-сцена готова, и все наши элементы GUI рисуются в правильном порядке. При этом у нас появились достаточно большие возможности контроля за отрисовкой наших объектов GUI, очень похожие на возможности, предоставляемые слоями в Photoshop. И всё это благодаря созданию шейдера и небольшого скрипта.

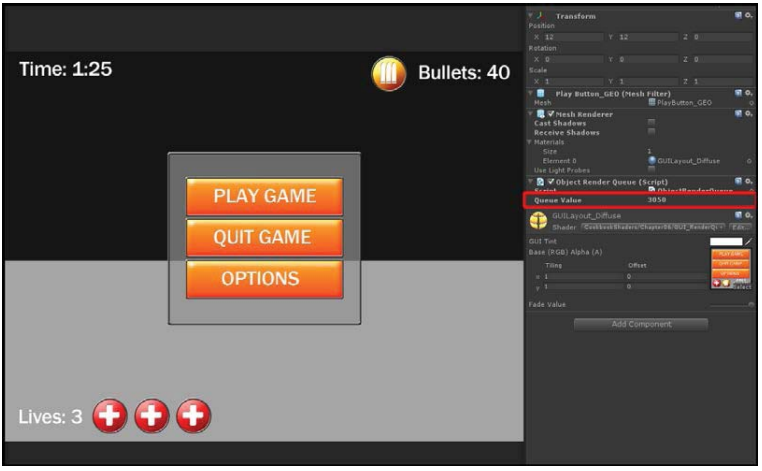

Рис. 6.10. Итоговый результат работы шейдера

Как это работает...

В GUI-шейдере мы добавили несколько новых тегов, которые позволяют нам осуществлять тонкую настройку работы нашего шейдера с рендерером Unity. Объявив инструкцию `"IgnoreProjector"="True"`,

мы говорим Unity, что не хотим, чтобы какие-либо материалы или текстуры с типом `projector` влияли на наши объекты или шейдер. Мы делаем это потому, что хотим, чтобы наши элементы GUI были независимы от сцены. Все эффекты, происходящие на сцене, в том числе создаваемые проекторами, должны затрагивать лишь игровые объекты, но не элементы GUI. Тег `"IgnoreProjector"` – как раз прекрасный способ решения этой задачи.

Второй новый для нас тег – это `"RenderType"="Transparent"`. Аналогично тегу `"Queue"`, этот тег говорит Unity, что это прозрачный шейдер, и используется в работе фулскрин-эффектов, для того чтобы Unity мог правильно отсортировать объекты.

Последнее новшество нашего шейдера – это добавление `novertexlights` в инструкцию `#pragma`. Этот параметр говорит Unity, что мы не хотим использовать повертексное освещение или сферические гармоники для освещения наших объектов. Более того, мы вообще не хотим использовать освещение. Поэтому мы используем этот параметр для того, чтобы облегчить наш шейдер в вычислительном плане, – как раз то, что нам нужно при разработке 3D-системы элементов GUI.

Что касается созданного нами скрипта для изменения очереди рендеринга – скрипт просто обращается к прикреплённому к нашему объекту материалу с помощью конструкции `transform.renderer.sharedMaterial`. Если материал действительно назначен объекту, к которому прикреплён код, то код вернёт материал. Если код не сможет найти материал, то он вернёт `null`.

После этого мы проверяем, смог ли скрипт найти материал, и изменяем значение очереди отрисовки. Если скрипт не смог найти материал, то мы просто выводим в консоль отладочное сообщение, чтобы пользователь знал, что объекту следовало бы назначить материал.

Этот простой пример демонстрирует возможности очередей отрисовки при работе с прозрачностью. Используя эти знания, вы сможете создавать свои более сложные эффекты.

ГЛАВА 7
Волшебные возможности вершин

В этой главе вы узнаете о том, как:

- использовать цвет вершин в поверхностном шейдере;
- анимировать вершины в поверхностном шейдере;
- использовать цвет вершин при создании ландшафта.

Введение

При рендеринге объектов в реальном времени нам никак не обойтись без шейдеров. С их помощью мы можем создавать очень сложные варианты освещения поверхности объектов, но, кроме этого, используя шейдеры, можно непосредственно модифицировать и вершины, из которых состоят ваши объекты. Этот подход имеет много преимуществ, по сравнению с работой с массивом вершин на CPU, так как, используя шейдеры, мы можем это делать намного быстрее, не пересылая каждый раз данные с CPU на GPU.

Вершинная функция выполняется один раз для каждой вершины, переданной на видеокарту (Graphics Processing Unit или GPU). Её задача – преобразовать локальные 3D-координаты вершины таким образом, чтобы при рендеринге она попала в правильное место на 2D-экране. С помощью вершинной функции мы можем изменять такие параметры вершины, как её положение, цвет и UV-координаты. После того как вершинная функция завершает свою работу, управление передается в функцию `surf()`, в которой применяются попиксельные эффекты.

Вершинный шейдер (vertex shader) является мощным инструментом для управления 3D-моделями, с помощью которого мы можем реализовать такие эффекты, как эффект волн на поверхности океана, или эффект развевающегося флага, или, например, раскрасить нашу модель цветами вершин. В этой главе мы рассмотрим применение вершинной функции в поверхностных шейдерах.

Получение цвета вершины в поверхностном шейдере

Давайте начнём эту главу с того, что разберёмся, как в вершинной функции поверхностного шейдера получить доступ к атрибутам каждой вершины модели. Вооружившись этим знанием, мы сможем использовать атрибуты вершин, чтобы создавать по-настоящему интересные визуальные эффекты.

Каждая вершина, передаваемая в вершинную функцию, может содержать различные атрибуты, например координаты вершины (тип `float3`), направление нормали (тип `float3`). Вы даже можете присваивать и получать цвет для каждой вершины (тип `float4`). На этих атрибутах мы остановимся в данном рецепте. Мы рассмотрим, как в поверхностных шейдерах сохранять и использовать цвета вершин.

Подготовка

Для того, чтобы написать этот вершинный шейдер, нам потребуется подготовить несколько ассетов. Выполните следующие шаги:

1. Для того чтобы мы смогли увидеть цвета вершин, нам нужна модель, вершинам которой присвоены некоторые цвета. Вы можете использовать Unity для назначения цветов, но тогда вам придётся написать инструмент, который бы позволил пользователю присваивать вершинам цвета, или же написать некий скрипт, который сделал бы это за вас. В этом рецепте мы просто воспользовались Maya для раскраски вершин нашей модели. Эта модель доступна по адресу: www.packtpub.com/support.
2. Создайте новую сцену и разместите на ней импортированную модель.
3. Создайте новый шейдер и материал. После этого назначьте шейдер материалу, а материал – импортированной модели.

Ваша сцена должна выглядеть примерно так, как показано на следующем скриншоте (рис. 7.1).

Как это сделать...

После того как мы подготовили сцену, шейдер и материал, мы можем перейти к написанию кода нашего шейдера. Откройте код шейдера, сделав по нему двойной щелчок мышкой на закладке проекта в редакторе Unity. После этого выполните следующие действия:

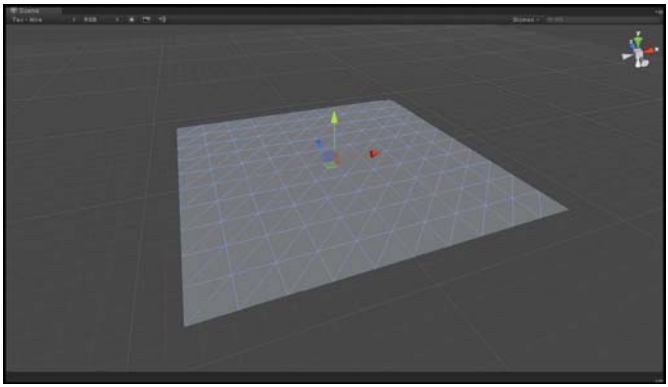

Рис. 7.1. Вид подготовленной сцены

1. Поскольку мы хотим создать очень простой шейдер, нам нет нужды добавлять какие-либо свойства в блок свойств. Однако мы всё равно добавим глобальный цвет, просто для того, чтобы выдержать единообразие формы шейдеров на протяжении книги. Добавьте в блок свойств вашего шейдера следующий код.

```
Properties
{
  _MainTint("Global Color Tint", Color) = (1,1,1,1)
}
```

2. Далее нам нужно сказать Unity, что мы хотим использовать в нашем шейдере вершинную функцию.

```
CGPROGRAM
#pragma surface surf Lambert vertex:vert
```

3. Как обычно, раз мы добавили свойства в наш блок свойств, нам нужно создать соответствующие переменные в блоке `CGPROGRAM`. Добавьте следующий код сразу после инструкции `#pragma`.

```
float4 _MainTint;
```

4. Теперь мы обратимся к структуре `Input`. Чтобы наша функция `surf()` смогла получить доступ к данным, предоставляемым функцией `vert()`, нам нужно будет добавить новую переменную.

```
struct Input
{
```

```
  float2 uv_MainTex;
  float4 vertColor;
};
```

5. Теперь мы можем перейти к написанию функции `vert()`, которая позволит нам получить доступ к цвету каждой вершины нашей модели.

```
void vert(inout appdata_full v, out Input o)
{
  o.vertColor = v.color;
}
```

6. Далее мы можем использовать данные о цвете вершин из нашей структуры `Input`, чтобы присвоить их параметру `o.Albedo` из встроенной структуры `SurfaceOutput`.

```
void surf(Input IN, inout SurfaceOutput o)
{
  o.Albedo = IN.vertColor.rgb * _MainTint.rgb;
}
```

7. После того как мы закончили с написанием кода, мы можем вернуться в Unity, чтобы наш шейдер скомпилировался. Если всё прошло без ошибок, то вы увидите что-то, очень похожее на следующий скриншот.

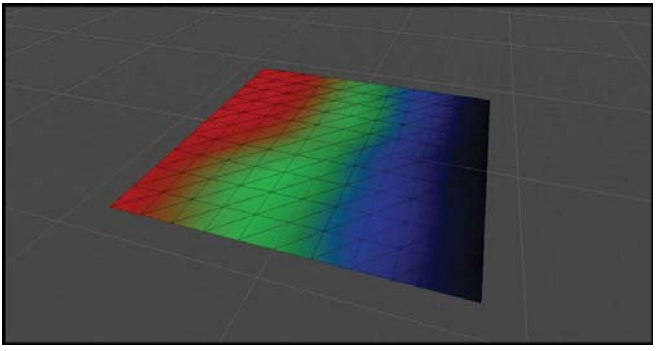

Рис. 7.2. Результат работы шейдера

Как это работает...

Unity предоставляет нам способ получить доступ к информации о вершинах модели, которой назначен шейдер. А это значит, что у нас появляется возможность изменять такие параметры, как положение вершины и её цвет. В этом рецепте мы импортировали меш из Maya

(хотя вы можете использовать любую подходящую программу для работы с 3D), в которой вершинам были назначены цвета. Вы заметите, что после импортирования модели материал, используемый нами по умолчанию, не будет отображать цвета вершин. Нам придётся написать шейдер, чтобы мы смогли извлечь цвета вершин и отобразить их на поверхности модели.

Unity предоставляет нам большой объём встроенного в поверхностные шейдеры функционала, что делает процесс доступа к атрибутам вершин быстрым и эффективным.

Наша первая задача – сказать Unity, что при создании нашего шейдера мы будем использовать вершинную функцию. Сделаем мы это, добавив параметр `vertex:vert` в секцию `#pragma` блока `CGPROGRAM`. Получив эту инструкцию, Unity при компиляции шейдера будет искать функцию с именем `vert`. Если Unity этой функции не найдёт, то будет выдана ошибка компиляции, и вас попросят добавить в шейдер функцию `vert`.

Что приводит нас к шагу 5, где мы добавляем функцию `vert()`. Объявив эту функцию, мы получаем доступ к встроенной структуре данных `appdata_full`, в которой хранится информация о вершине. Из неё мы получаем информацию о цветах вершин, которую передаём в нашу структуру `Input` с помощью следующего фрагмента кода: `o.vertColor = v.color`.

Переменная `o` соответствует нашей структуре `Input`, а переменная `v` – структуре `appdata_full`, содержащей вершинные данные. В этом случае мы просто получаем данные из структуры `appdata_full` и записываем их в нашу структуру `Input`. А после того как цвет вершины оказывается в структуре `Input`, – мы можем использовать его в нашей функции `surf()`. В данном же рецепте мы просто присваиваем цвет параметру `o.Albedo` во встроенной структуре `SurfaceOutput`.

Но это ещё не всё...

Можно получить доступ и к четвёртой компоненте данных цвета вершины. Вероятно, вы заметили, что переменная `vertColor`, которую мы объявили в структуре `Input`, имеет тип `float4`. А это значит, что помимо цвета вершины передаётся ещё и её значение `alpha`. Используя это значение, можно, например, сделать прозрачный материал, или же вы можете использовать его как ещё одну маску для блендинга двух текстур. Как бы то ни было, решать вам, будете вы использовать для ваших шейдеров этот четвёртый компонент цвета или нет. Но, по крайней мере, вы должны знать о нём.

В Unity4 у нас появилась возможность писать шейдеры под DirectX 11. И хоть так у нас и появляются замечательные возможности, вместе с этим процесс компилирования шейдеров становится немного сложнее. Теперь в шейдер нам нужно будет добавить ещё одну строку кода, чтобы правильным образом инициализировать выходную информацию вершин. Следующий фрагмент кода показывает, как будет выглядеть вершинная функция при написании шейдера под DirectX 11.

```
void vert(inout appdata_full v, out Input o)
{
     UNITY_INITIALIZE_OUTPUT(Input, o);
     o.vertColor = v.color;
}
```

После того как мы добавили эту строку кода, наш вершинный шейдер не будет выдавать предупреждений о том, что он не скомпилируется под DirectX 11 правильным образом.

Анимация вершин в поверхностном шейдере

Теперь, когда мы знаем, как получить доступ к цвету каждой вершины, давайте потренируемся обрабатывать и другие данные, например координаты вершины.

В вершинной функции мы можем получить доступ к координатам каждой вершины меша. Что позволяет нам изменять каждую отдельную вершину во время работы шейдера.

В этом рецепте мы создадим шейдер, который позволит нам изменять координаты каждой вершины меша по синусоиде. Этот приём может пригодиться при создании анимации таких объектов, как флаги или волны океана.

Подготовка

Давайте подготовим наши ассеты, чтобы мы смогли перейти к написанию шейдера. Для этого выполните следующие действия:

1. Создайте новую сцену и поместите в её центре плоскость.
2. Создайте новый шейдер и материал.
3. Назначьте шейдер материалу, а материал – плоскости.

После этих действий ваша сцена должна выглядеть, как на следующем скриншоте.

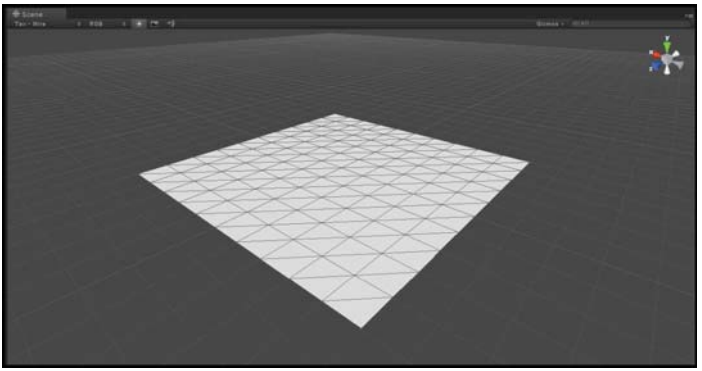

Рис. 7.3. Вид сцены, использованной в данном рецепте

Как это сделать...

Теперь, когда наша сцена готова, вы можете два раза кликнуть по только что созданному шейдеру, чтобы открыть его в MonoDevelop. Далее выполните следующие действия:

1. Давайте начнём написание нашего шейдера с блока свойств. Добавьте в него следующий код.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _tintAmount ("Tint Amount", Range(0,1)) = 0.5
  _ColorA ("Color A", Color) = (1,1,1,1)
  _ColorB ("Color B", Color) = (1,1,1,1)
  _Speed ("Wave Speed", Range(0.1, 80)) = 5
  _Frequency ("Wave Frequency", Range(0, 5)) = 2
  _Amplitude ("Wave Amplitude", Range(-1, 1)) = 1
}
```

2. После этого нам потребуется сообщить Unity о том, что мы хотим использовать вершинную функцию. Для этого добавьте следующий код в секцию `#pragma`.

```
CGPROGRAM
#pragma surface surf Lambert vertex:vert
```

3. Чтобы мы смогли получить доступ к значениям наших свойств, нам потребуется объявить соответствующие переменные в блоке `CGPROGRAM`.

```
sampler2D _MainTex;
float4 _ColorA;
```

```
float4 _ColorB;
float _tintAmount;
float _Speed;
float _Frequency;
float _Amplitude;
float _OffsetVal;
```

4. Мы хотим изменять не только координаты вершин, но и их цвет. Таким образом, мы сможем подкрасить наш объект.

```
struct Input
{
  float2 uv_MainTex;
  float3 vertColor;
};
```

5. Теперь нам нужно написать вершинную функцию, которая будет деформировать меш по синусоиде. Добавьте следующий код после структуры `Input`.

```
void vert(inout appdata_full v, out Input o)
{
  float time = _Time.x * _Speed;
  float waveValueA = sin(time + v.vertex.x * _Frequency) *
_Amplitude;

  v.vertex.xyz = float3(v.vertex.x, v.vertex.y + waveValueA,
v.vertex.z);
  v.normal = normalize(float3(v.normal.x + waveValueA,
v.normal.y, v.normal.z));
  o.vertColor = float3(waveValueA, waveValueA, waveValueA);
}
```

6. И наконец, мы используем функцию `lerp` для интерполяции между двумя заданными цветами, чтобы выделить цветом пики и впадины деформированной поверхности.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex);
  float3 tintColor = lerp(_ColorA, _ColorB, IN.vertColor).rgb;

  o.Albedo = c.rgb * (tintColor * _tintAmount);
  o.Alpha = c.a;
}
```

После того как вы закончите написание вашего шейдера, переключитесь обратно в Unity, чтобы он скомпилировался. После этого вы должны получить результат, похожий на следующий скриншот.

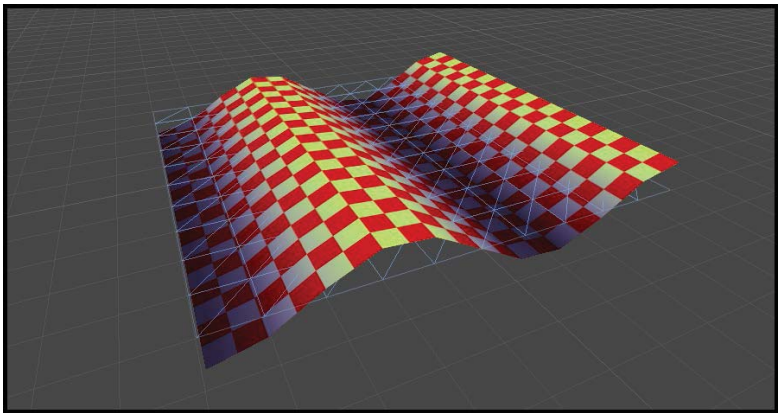

Рис. 7.4. Результат работы шейдера

Как это работает...

Наш шейдер использует тот же принцип работы, что и шейдер из предыдущего рецепта, за исключением того, что теперь мы модифицируем расположение вершин сетки. Этот приём очень удобен при работе с простыми объектами, например флагами, которые вы не хотите разбивать на составляющие и анимировать их с помощью скелетной анимации или иерархического набора трансформаций.

Для получения значения синуса в точке мы используем встроенную в язык Cg функцию `sin()`. Далее мы добавляем его к `y`-координате каждой вершины и получаем волнообразный эффект.

Помимо этого, мы ещё немного модифицировали нормали точек меша, чтобы придать модели чуть более реалистичный шейдинг, учитывающий значения синусоиды в вершинах.

Вы видите, насколько легко создавать даже более сложные вершинные эффекты с помощью встроенных параметров, предоставляемых нам поверхностными шейдерами.

Использование цветов вершин для ландшафта

Одним из наиболее распространённых применений цвета вершин является создание более реалистичного ландшафта или окружающего мира. При этом информация из RGBA-каналов цвета вершины

используется для блендинга разных текстур. Такой подход является весьма эффективным, поскольку вам не нужно импортировать ещё одну специальную текстуру для управления блендингом. Эта техника применяется практически в любой игре, в которой присутствуют сцены с ландшафтами и зданиями.

В этом рецепте вам будет продемонстрирована более продвинутая техника блендинга, в которой используется специальная черно-белая текстура – карта высот, что позволяет добавить больше деталей.

Подготовка

Давайте подготовим для этого рецепта нашу сцену, а также несколько текстур. Выполните следующие действия:

1. Создайте новую сцену и импортируйте в неё меш с раскрашенными вершинами из 3D-редактора. Для данного рецепта мы использовали Maya.
2. Поместите импортированную модель на сцену и создайте источник направленного освещения.
3. Далее создайте новый шейдер и материал. Шейдер назначьте материалу, а материал – модели.

Как это сделать...

После того как вы создадите новую сцену, сделайте двойной щелчок по шейдеру, чтобы открыть его в MonoDevelop. Выполните следующую последовательность действий:

1. Давайте создадим свойства, которые нам понадобятся, чтобы дать пользователю нашего шейдера больший контроль за итоговым визуальным эффектом.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _SecondaryTex ("Secondary Texture", 2D) = "white" {}
  _HeightMap ("HeightMap", 2D) = "white" {}
  _Value ("Value", Range(1, 20)) = 3
}
```

2. После этого нам нужно будет сказать Unity, что мы будем использовать вершинную функцию в нашем поверхностном шейдере. Для этого добавьте следующий код.

```
CGPROGRAM
#pragma surface surf Lambert vertex:vert
```

3. Далее создайте переменные, которые добавят связь нашему блоку CGPROGRAM с блоком свойств.

```
sampler2D _MainTex;
sampler2D _SecondaryTex;
sampler2D _HeightMap;
float _Value;
```

4. Поскольку мы собираемся использовать ещё несколько текстур и цвета вершин, нам потребуется добавить в нашу структуру Input несколько дополнительных параметров.

```
struct Input
{
  float2 uv_MainTex;
  float2 uv_SecondaryTex;
  float3 vertColor;
};
```

5. После этого мы добавим вершинную функцию. В этот раз она будет довольно простой, поскольку всё, что нам нужно сделать, – это получить цвет вершины и передать его в структуру Input.

```
void vert(inout appdata_full v, out Input o)
{
  o.vertColor = v.color.rgb;
}
```

6. Теперь мы можем заняться нашей функцией surf(). В ней нам потребуется первым делом получить данные из текстур для последующего блендинга.

```
//Получим данные из текстур
half4 base = tex2D(_MainTex, IN.uv_MainTex);
half4 secondTex = tex2D(_SecondaryTex, IN.uv_SecondaryTex);
float4 height = tex2D(_HeightMap, IN.uv_MainTex);
```

7. После этого мы вычисляем параметр блендинга, используя красный канал цвета вершины и цвет из карты высот.

```
//Выполним блендинг
float redChannel = 1 - IN.vertColor.r;
float rHeight = height.r * redChannel;
float invertHeight = 1 - height.r;
float finalHeight = (invertHeight * redChannel) * 4;
float finalBlend = saturate(rHeight + finalHeight);
```

8. Следующим шагом мы вычисляем параметр затухания блендинга, данные для которого передаются в зеленом канале и ко-

торый позволяет контролировать четкость границы блендинга текстур.

```
//Давайте добавим деталей для блендинга вершин.
//Пусть будут или очень чёткие, или очень мягкие границы.
float hardness = ((1 - IN.vertColor.g) * _Value) + 1;
finalBlend = pow(finalBlend, hardness);
```

9. И наконец, мы используем функцию `lerp` для интерполяции между двумя текстурами, а полученный в результате цвет передаём в структуру `SurfaceOutput`.

```
//Выдадим итоговый цвет
float3 finalColor = lerp(base, secondTex, finalBlend);
o.Albedo = finalColor;
o.Alpha = base.a;
```

После того как ваш шейдер будет скомпилирован, вы должны получить результат, похожий на изображённый на следующем скриншоте.

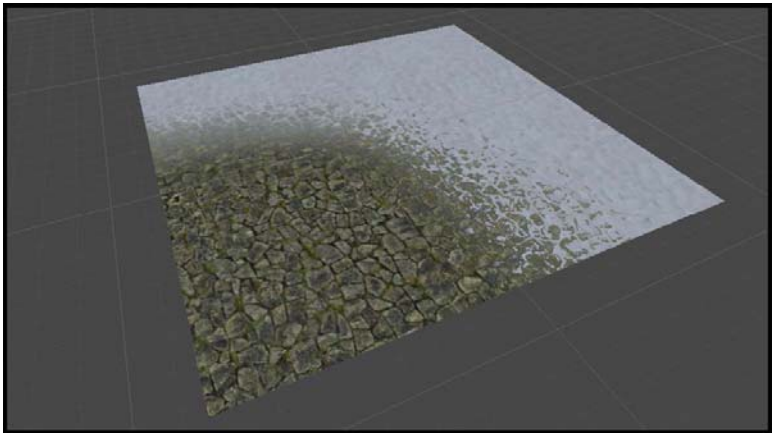

Рис. 7.5. Результат работы шейдера

Как это работает...

Определённо этот шейдер будет посложнее предыдущих, но обратите внимание, что мы практически ничего не делаем в самой вершинной функции. Мы лишь передаём цвета вершин в функцию `surf()`, чтобы использовать их в попиксельных операциях. Мы могли бы реализовать блендинг в вершинной функции, что было бы быстрее и выглядело бы довольно убедительно. Но, к сожалению, таким образом мы

бы получили множество хорошо заметных визуальных артефактов, исправить которые можно, лишь добавив в меш ещё больше вершин, а это не всегда приемлемо.

Поэтому цвета вершин мы умножаем на данные из черно-белой карты высот базовой текстуры, на которую мы хотим наложить вторую текстуру. Выполнив для цветов вершин и карты высот седьмой шаг алгоритма, мы можем получить дополнительный уровень детализации, который позволяет сымитировать эффект объединения одной текстуры с другой. В нашем случае текстура снега смешивается с маленькими трещинками текстуры камня.

Эта техника не так давно стала весьма популярна благодаря таким играм, как Uncharted и Gears of War. А теперь и вы можете использовать её при разработке своих игр.

ГЛАВА 8
Настройка шейдеров для мобильных приложений

В двух следующих главах мы остановимся на оптимизации шейдеров под различные платформы. Мы не будем рассматривать какую-то одну отдельно взятую платформу, вместо этого мы попробуем разбить шейдеры на отдельные элементы, и посмотреть, какие корректировки можно в них внести, для того чтобы повысить их эффективность как на мобильных платформах, так и на любых других. Мы начнём с того, что разберёмся во встроенных в Unity функциях и переменных, используя которые, мы можем снизить расход памяти, а далее перейдём непосредственно к способам оптимизации нашего собственного кода. В этой главе мы остановимся на следующих темах:

- что значит дешевый шейдер?
- профайлинг шейдеров;
- модификация шейдеров для мобильных платформ.

Введение

Знания об оптимизации шейдеров пригодятся вам почти в любом игровом проекте. В разработке всегда наступает момент, когда некоторый шейдер необходимо оптимизировать, или переписать так, чтобы он использовал меньше текстур, но при этом создавал такой же визуальный эффект. Вам как шейдерному программисту, или если хотите – техническому художнику, нужно понимать базовые принципы, которые позволят вам оптимизировать шейдеры, чтобы повысить производительность игры, но при этом не потерять в качестве графики. Эти же знания вам пригодятся, когда вы только приступаете к написанию шейдера. К примеру, если вам будет наперёд известно,

что игра, использующая ваш шейдер, будет запускаться на мобильной платформе, вы можете в функциях освещения использовать половинный вектор для направления взгляда или поменять тип переменных с `float` на `fixed` или `half`. Применение этих и многих других техник скажется на эффективности работы вашего шейдера на целевом аппаратном обеспечении. А теперь давайте начнём знакомство с оптимизацией шейдеров.

Что значит дешевый шейдер?

Сходу ответить на вопрос «что же такое вычислительно дешёвый шейдер?» может быть немного затруднительно, поскольку эффективность шейдеров зависит от многих факторов. Это может быть и количество памяти, выделяемой под переменные. Это может быть и количество используемых текстур. А может быть и так, что наш шейдер работает как надо, но есть способ добиться такого же эффекта, используя меньше кода или входных данных. В этом рецепте мы рассмотрим несколько таких техник и покажем, как их можно комбинировать, чтобы ваш шейдер работал быстрее и эффективнее и при этом создавал высококачественные визуальные эффекты, которых сейчас все ожидают от игр, и на компьютере, и на мобильных.

Подготовка

Для начала работы с этим рецептом нам понадобится несколько ресурсов. Поэтому выполните следующие действия:

1. Создайте новую сцену и поместите на неё простой объект-сферу, а также направленный источник света.
2. Создайте новый шейдер и материал, назначьте шейдер материалу.
3. После этого назначьте материал только что созданной сфере.
4. И наконец, подправьте шейдер таким образом, чтобы он использовал диффузную текстуру, карту нормалей и вашу собственную функцию освещения.

```
Shader "Cookbook/Chapter08/OptimizedShader001"
{
  Properties
  {
          _MainTex ("Base (RGB)", 2D) = "white" {}
          _NormalMap ("Normal Map", 2D) = "bump" {}
```

```
    }

    SubShader
    {
            Tags { "RenderType"="Opaque" }
            LOD 200

            CGPROGRAM
            #pragma surface surf SimpleLambert

            sampler2D _MainTex;
            sampler2D _NormalMap;

            struct Input
            {
                    float2 uv_MainTex;
                    float2 uv_NormalMap;
            };

            inline float4 LightingSimpleLambert(SurfaceOutput s,
                                    float3 lightDir, float atten)
            {
                    float diff = max(0, dot(s.Normal, lightDir));

                    float4 c;
                    c.rgb = s.Albedo * _LightColor0.rgb * (diff *
                                atten * 2);
                    c.a = s.Alpha;
                    return c;
            }

            void surf(Input IN, inout SurfaceOutput o)
            {
                    float4 c = tex2D(_MainTex, IN.uv_MainTex);

                    o.Albedo = c.rgb;
                    o.Alpha = c.a;
                    o.Normal = UnpackNormal(tex2D(_NormalMap,
                                    IN.uv_NormalMap));
            }
            ENDCG
    }
    FallBack "Diffuse"
}
```

Этот код есть не что иное, как модификация дефолтного шейдера, который Unity создала для нас в первом шаге. На данном этапе ваша сцена должна быть похожа на изображённую на следующем скриншоте. Используя такой сетап, мы рассмотрим некоторые базовые концепции оптимизации шейдеров на примере поверхностных шейдеров в Unity.

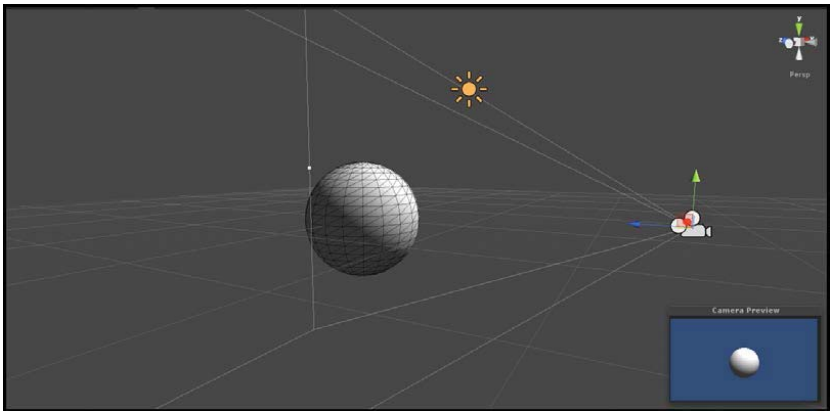

Рис. 8.1. Вид подготовленной сцены

Как это сделать...

Мы начнём с простого диффузного шейдера, с помощью которого мы рассмотрим несколько способов оптимизации шейдеров.

Для начала мы оптимизируем типы наших переменных, чтобы на них выделялось меньше памяти при обработке данных.

Выполните следующие действия:

1. Давайте начнём со структуры `Input` нашего шейдера. Сейчас наши UV-координаты хранятся в переменной с типом `float2`. Изменим её тип на `half2`.

```
struct Input
{
  half2 uv_MainTex;
  half2 uv_NormalMap;
};
```

2. После этого мы можем перейти к нашей функции освещения и уменьшить расход памяти на локальные переменные, изменив их типы следующим образом:

```
inline fixed4 LightingSimpleLambert(SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  fixed diff = max(0, dot (s.Normal, lightDir));

  fixed4 c;
  c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten * 2);
  c.a = s.Alpha;
```

```
  return c;
}
```

3. И наконец, мы заканчиваем эту оптимизацию, изменив типы переменных в функции `surf()`.

```
void surf(Input IN, inout SurfaceOutput o)
{
  fixed4 c = tex2D(_MainTex, IN.uv_MainTex);

  o.Albedo = c.rgb;
  o.Alpha = c.a;
  o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_NormalMap));
}
```

4. Теперь, когда типы наших переменных оптимизированы, мы можем воспользоваться встроенной директивой освещения для контроля за тем, как шейдер работает с источниками света. Таким образом, мы сможем заметно снизить количество светильников, которые вынужден обрабатывать наш шейдер. Измените секцию `#pragma` вашего шейдера следующим образом:

```
CGPROGRAM
#pragma surface surf SimpleLambert noforwardadd
```

5. Мы можем дальше оптимизировать шейдер, если будем использовать один набор UV-координат и для карты нормалей, и для диффузной текстуры. Для этого мы просто в функции `UnpackNormal()` будем использовать UV-координаты `_MainTex` вместо `_NormalMap`.

```
void surf(Input IN, inout SurfaceOutput o)
{
  fixed4 c = tex2D(_MainTex, IN.uv_MainTex);

  o.Albedo = c.rgb;
  o.Alpha = c.a;
  o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
}
```

6. А поскольку нам больше не нужны UV-координаты карты нормалей, нам нужно обязательно убрать их упоминание из структуры `Input`.

```
struct Input
{
  half2 uv_MainTex;
};
```

7. И наконец, мы можем ещё сильнее оптимизировать наш шейдер, указав ему, что он должен работать лишь с определёнными рендерерами.

```
CGPROGRAM
#pragma surface surf SimpleLambert exclude_path:prepass
noforwardadd
```

В результате наших действий мы получили шейдер, который визуально ничем не отличается, но при этом нам удалось сократить время, затрачиваемое шейдером на отрисовку. В следующем разделе мы узнаем, как можно замерить время рендеринга шейдера, сейчас же для нас ключевой идеей является то, что мы можем добиться такого же результата с помощью меньшего количества данных. Помните об этом, когда будете создавать новые шейдеры. На следующем изображении приводится итоговый результат работы нашего оптимизированного шейдера.

Рис. 8.2. Итоговый результат работы шейдера

Как это работает...

Теперь, когда мы увидели, каким образом можно оптимизировать шейдеры, мы можем копнуть ещё чуть глубже и разобраться, почему работают рассмотренные способы, а также взглянуть на несколько других приёмов, которые могут вам пригодиться.

Давайте сначала посмотрим на размер в байтах каждой объявленной нами переменной. Если вы знакомы с программированием, то вы знаете, что с помощью типов данных вы можете объявлять перемен-

ные различного размера. Это значит, что, например, тип данных float занимает наибольший объём памяти. Давайте рассмотрим типы переменных более подробно:

- `float` – для хранения переменной этого типа выделяется 32 бита, кроме того, это наиболее медленный из трёх рассматриваемых нами типов данных. Этот тип данных имеет производные типы `float2`, `float3` и `float4`;
- `half` – это укороченное число с плавающей точкой, для хранения которого используются 16 бит. Хорошо подходит для хранения UV-координат, значений цветов. Обрабатывается гораздо быстрее, чем переменные типа `float`. Этот тип данных имеет производные типы `half2`, `half3` и `half4`;
- `fixed` – это тип данных, который занимает меньше всего памяти и для которого отводится по крайней мере 10 бит. Его можно использовать при вычислении света и преобразовании цветов, он имеет производные типы `fixed2`, `fixed3` и `fixed4`.

Второй шаг нашей оптимизации простого шейдера заключался в объявлении `noforwardadd` в секции `#pragma`. Эта инструкция является флагом, указание которого говорит Unity, что любой объект с этим шейдером будет освещаться попиксельно только одним направленным источником света. Все остальные источники освещения, рассчитываемые этим шейдером, будут принудительно просчитываться повертексно, используя значения сферических гармоник, рассчитываемых внутри Unity. Поскольку наш шейдер выполняет попиксельные операции с помощью карты нормалей, этот эффект будет хорошо видно, если мы разместим на сцене ещё один источник света для подсветки нашей сферы.

Это, конечно, здорово, но что делать, если мы хотим на сцене разместить несколько направленных источников света и при этом иметь возможность контролировать, какой из этих источников будет использоваться для основного попиксельного освещения? Вы могли заметить, что каждый источник света имеет выпадающее меню **Render Mode** (Режим рендеринга). Если вы кликнете на выпадающее меню, то вы увидите несколько флажков: **Auto** (Автоматически), **Important** (Важный) и **Not Important** (Неважный). Выбрав для освещения режим **Important**, вы говорите Unity, что источник света должен, по возможности, считаться попиксельно, а не повертексно, при выборе **Not Important** – наоборот. Если вы оставите выбранным режим **Auto**, Unity будет решать, как поступить с этим светильником самостоятельно.

Поместите ещё один источник света на сцену и уберите текстуру, которая на данный момент выбрана как основная текстура вашего шейдера. Вы заметите, что второй точечный источник света не взаимодействует с картой нормалей, с ней работает лишь направленный источник света, который мы создали первым. Идея этого приёма – в том, что вы можете сэкономить на попиксельных вычислениях, если будете обрабатывать все дополнительные источники света как вершинные, что благотворно скажется на производительности, поскольку попиксельное освещение будет рассчитываться только для главного источника света. Следующий скриншот служит наглядной демонстрацией этой идеи, поскольку точечный источник света не взаимодействует с картой нормалей.

Рис. 8.3. Иллюстрация попиксельного и вершинного освещений источниками света

После этого мы немного навели порядок и сказали текстуре карты нормалей использовать тот же набор UV-координат, что и для основной текстуры, кроме этого, мы избавились от строки кода, с помощью которой в шейдер передавались (ненужные теперь) UV-координаты специально для карты нормалей.

Также мы добавили `exclude_path:prepass` в секцию `#pragma`, для того чтобы на этот шейдер не действовало кастомное освещение при **Deferred**-рендеринге. Это значит, что эффективно мы можем использовать наш шейдер только при **Forward**-рендеринге, который по умолчанию выставлен в настройках основной камеры.

Вы удивитесь тому, насколько можно оптимизировать шейдер, если уделить этому немного времени. Вы уже видели, как можно

упаковывать черно-белые текстуры в одну RGBA-текстуру, а также как использовать текстуры для имитации освещения. Есть много способов оптимизации шейдера, поэтому выбор оптимального способа – вопрос неоднозначный. Но если вы владеете этими техниками оптимизации, вы сможее подогнать ваши шейдеры под вашу игру и под её целевую платформу, обеспечив тем самым стабильный и плавный фреймрейт.

Профайлинг шейдеров

Теперь, когда вы знаете, как уменьшить расход ресурсов шейдерами, давайте посмотрим, как найти эти «проблемные» шейдеры на сцене, на которой может быть много шейдеров, объектов, скриптов – всего этого, выполняющегося одновременно. Поиск одного конкретного объекта или шейдера в пространстве всей игры может быть весьма неприятной задачей, но на этот случай Unity предоставляет нам встроенный профайлер. С его помощью мы можем увидеть кадр за кадром, что же происходит в игре, а также понять, как каждый элемент обрабатывается GPU или CPU.

С помощью профайлера мы можем изолировать такие элементы, как шейдеры, геометрию, и другие объекты, участвующие в рендеринге. Мы можем отфильтровывать элементы до тех пор, пока перед нами не останется работа всего лишь одного объекта. А вот тогда-то мы и сможем увидеть, как его работа сказывается на CPU и GPU.

Давайте внимательно рассмотрим использование профайлера и узнаем, как мы можем отлаживать наши сцены и, что ещё важнее, шейдеры.

Подготовка

Давайте начнём работу с профайлером с того, что подготовим необходимые ассеты и откроем окно профайлера. Выполните следующие действия:

1. Мы воспользуемся сценой, сделанной нами в последнем рецепте. Запустите профайлер Unity из **Window | Profiler** (Окно | Профайлер), или нажатием сочетания клавиш **Ctrl+7**.
2. Теперь нам нужно расклонировать нашу сферу, чтобы посмотреть, как это скажется на рендеринге.

После выполнения этих действий вы должны увидеть нечто похожее на следующий скриншот.

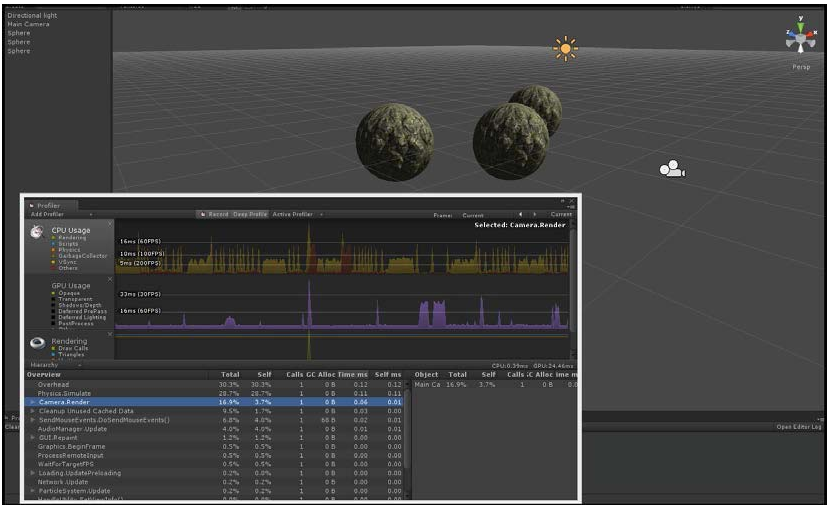

Рис. 8.4. Вид окна профайлера

Как это сделать...

Перед тем как начать работать с профайлером, мы посмотрим на некоторые элементы интерфейса в этом окне. Прежде чем вы нажмёте кнопку **Play**, давайте разберёмся, как мы можем получить в профайлере нужную нам информацию. Для этого выполните следующие действия:

1. Сперва нажмите на большие блоки в окне профайлера, на которых написано: **GPU Usage** (Использование GPU), **CPU Usage** (Использование CPU), **Rendering** (Рендеринг). Эти блоки расположены слева в верхней части окна, что наглядно иллюстрирует скриншот на рис. 8.5.

   Используя эти блоки, мы можем наблюдать за различными данными, характерными для основных функций нашей игры. В блоке **CPU Usage** мы можем увидеть результаты работы большинства наших скриптов, а также игровой физики и общего рендеринга. Блок **GPU Usage** предоставляет нам детальную информацию об элементах, связанных с нашим освещением, тенями и очередями отрисовки. И наконец, в блоке **Rendering** мы можем увидеть информацию о запросах на отрисовку (draw call) и о геометрии, присутствующей на нашей сцене для каждого кадра.

Нажав на один из этих блоков, мы можем отдельно выделить любую информацию, наблюдаемую нами во время профайлинга.

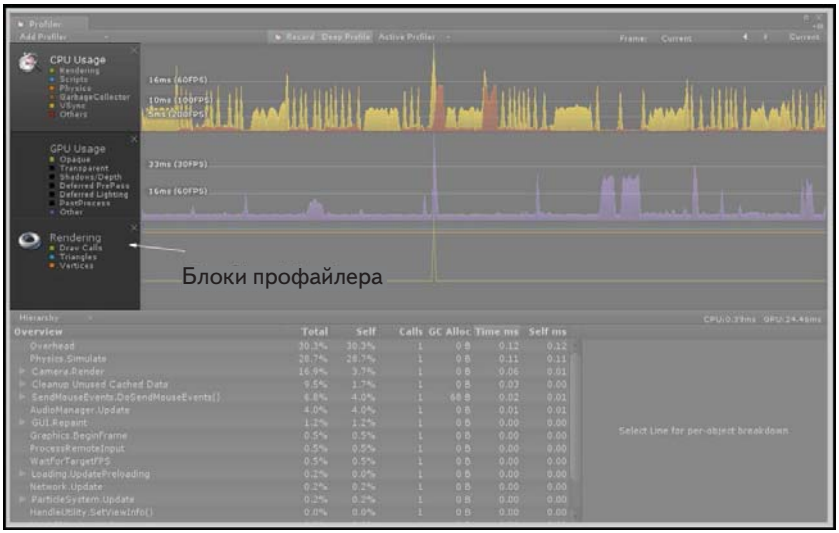

Рис. 8.5. Блоки профайлера

2. Теперь кликните по маленьким цветным квадратикам внутри этих больших блоков и нажмите **Play**, или сочетание клавиш **Ctrl**+**P**, для запуска сцены.

   Таким образом, мы можем отфильтровать данные, которые мы получаем, и углубиться в процесс профайлинга в нужном направлении. Для запущенной сцены снимите выделение со всех полей, кроме **Opaque** в блоке **GPU Usage**. Обратите внимание, что теперь мы видим лишь информацию о том, сколько тратится времени на рендеринг непрозрачных объектов, то есть тех, что находятся в очереди отрисовки **Opaque**.

Рис. 8.6. Время, затрачиваемое на рендеринг непрозрачных объектов

3. Ещё одна замечательная возможность окна профайлера, – это возможность перемещаться по кадрам при нажатии на график.

Это действие автоматически поставит вашу игру на паузу, чтобы вы смогли детально проанализировать внезапный скачок на графике и установить, какие именно элементы плохо влияют на производительность. Попробуйте сами покликать по графику и посмотреть, что показывает профайлер для разных кадров.

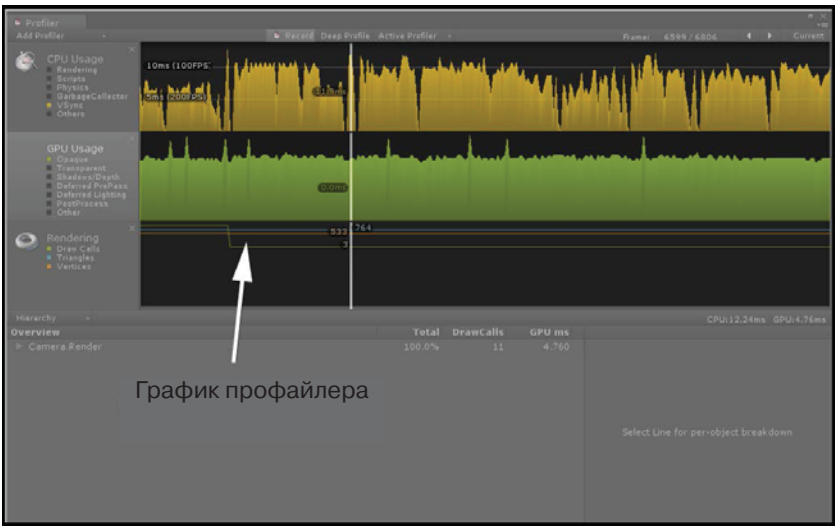

Рис. 8.7. График профайлера

4. Давайте теперь обратим наше внимание на нижнюю часть окна профайлера (рис. 8.8). Вы заметите, что при выделенном блоке **GPU Usage** появляется раскрывающийся список. Мы можем воспользоваться им, чтобы получить ещё более детальную информацию о текущей сессии работы с профайлером и, в данном случае, о том, что сейчас отрисовывает камера и сколько времени это занимает.

   Таким образом, у нас появляется возможность подробно рассмотреть, что же Unity обрабатывает на данном конкретном фрейме. В этом случае мы видим, что три сферы отрисовываются нашим оптимизированным шейдером примерно за 0,14 миллисекунды, при этом происходит семь запросов на отрисовку, обработка которых требует на каждом фрейме 3,1 процента времени работы GPU. Это как раз та информация, которую мы можем использовать для диагностики и

устранения проблем с производительностью шейдеров. Давайте проведём эксперимент, чтобы посмотреть, как скажется на производительности добавление в наш шейдер ещё одной текстуры, а также блендинг двух текстур с помощью функции `lerp`. Результат этого изменения вы сразу заметите в профайлере.

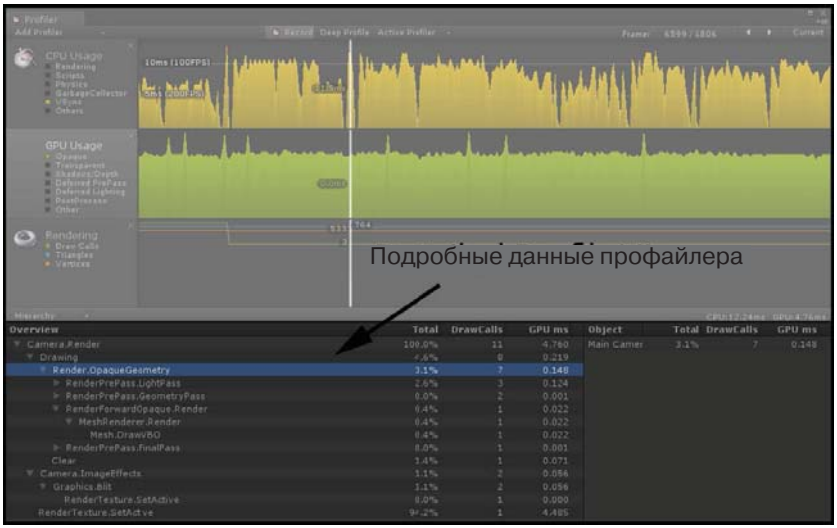

Рис. 8.8. Подробные данные профайлера

5. Измените блок свойств следующим образом, что даст нам возможность работать с ещё одной текстурой.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _BlendTex ("Blend Texture", 2D) = "white" {}
  _NormalMap ("Normal Map", 2D) = "bump" {}
}
```

6. После этого давайте добавим нашу текстуру в блок `CGPROGRAM`:

```
sampler2D _MainTex;
sampler2D _BlendTex;
sampler2D _NormalMap;
```

7. Теперь нам нужно добавить соответствующий код в функцию `surf()`, чтобы мы смогли смешать нашу новую текстуру с диффузной текстурой.

```
void surf(Input IN, inout SurfaceOutput o)
{
  fixed4 c = tex2D(_MainTex, IN.uv_MainTex);
  fixed4 blendTex = tex2D(_BlendTex, IN.uv_MainTex);

  c = lerp(c, blendTex, blendTex.r);

  o.Albedo = c.rgb;
  o.Alpha = c.a;
  o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
}
```

После того как вы сохраните изменения и вернётесь в редактор Unity, мы сможем запустить нашу игру и посмотреть, на сколько миллисекунд дольше будет работать наш шейдер. Нажмите кнопку **Play** и взгляните на результаты измерений в окне профайлера.

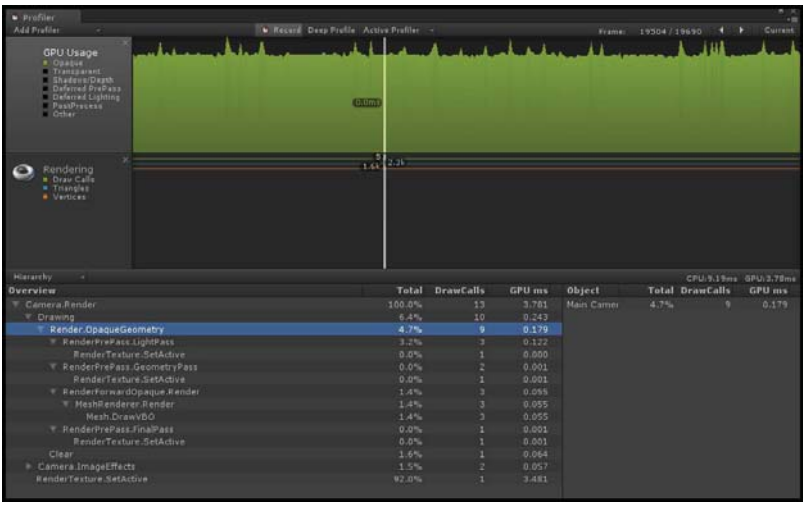

Рис. 8.9. Изменившееся время отрисовки

Как видите, время, затрачиваемое на рендеринг непрозрачной (Opaque) геометрии в сцене, возросло с 0,140 до 0,179 миллисекунды. Это случилось из-за того, что мы добавили ещё одну текстуру и использовали функцию `lerp()`. Возможно, это покажется небольшой разницей, но представьте себе двадцать шейдеров, все работающие по-разному и с разными объектами.

Используя данные профайлера, вы сможете быстрее находить причины снижения производительности и устранять их с помощью приёмов из предыдущего рецепта.

Как это работает...

Описание внутренних механизмов работы этого инструмента однозначно выходит за рамки данной книги, мы остановимся на том, что Unity предоставляет нам способ наблюдения за производительностью компьютера во время работы игры. Профайлер очень тесно связан с CPU и с GPU, благодаря чему мы получаем данные в реальном времени о том, сколько времени тратится на каждый из наших скриптов, объектов и очередей отрисовки. Мы уже видели, что, используя эту информацию, мы можем отслеживать эффективность нашего шейдера и устранять проблемные участки кода.

Но это ещё не всё...

Кроме этого, можно профилировать игру непосредственно на мобильном устройстве. Unity предоставляет нам несколько дополнительных возможностей, появляющихся, если в настройках сборки будет установлен Android или iOS. Мы можем получать информацию в реальном времени от нашего мобильного устройства во время работы игры. Это очень удобно, поскольку вы сможете профилировать непосредственно под конкретное устройство. Больше об этом вы можете узнать в разделе документации Unity, доступном по следующей ссылке: http://docs.unity3d.com/Documentation/Manual/MobileProfiling.html.

Изменение шейдеров для мобильных

Теперь, когда мы рассмотрели несколько способов оптимизации шейдеров, давайте напишем хороший, высококачественный шейдер, специально предназначенный для мобильных устройств. На самом деле нам не составит труда внести несколько изменений в уже написанный нами шейдер таким образом, чтобы на мобильных устройствах он выполнялся быстрее. Эти изменения включают, например, использование директив `approxview` или `halfasview` в настройках освещения. Кроме этого, мы можем сократить количество используемых текстур, а для оставшихся использовать более сильное сжатие. К концу этой секции мы получим хорошо оптимизированный Specular-шейдер, учитывающий карту нормалей, который мы сможем использовать в играх на мобильных платформах.

Подготовка

Перед тем как мы начнём работу, давайте создадим новую сцену и добавим на неё объекты, к которым мы будем применять наш шейдер для мобильных устройств. Для этого выполните следующие действия:

1. Создайте новую сцену и поместите на неё дефолтную сферу и направленный источник света.
2. Создайте новый материал и шейдер. Назначьте шейдер материалу.
3. После этого назначьте материал нашему объекту-сфере.

После этих действий ваша сцена должна стать похожей на изображённую на следующем скриншоте.

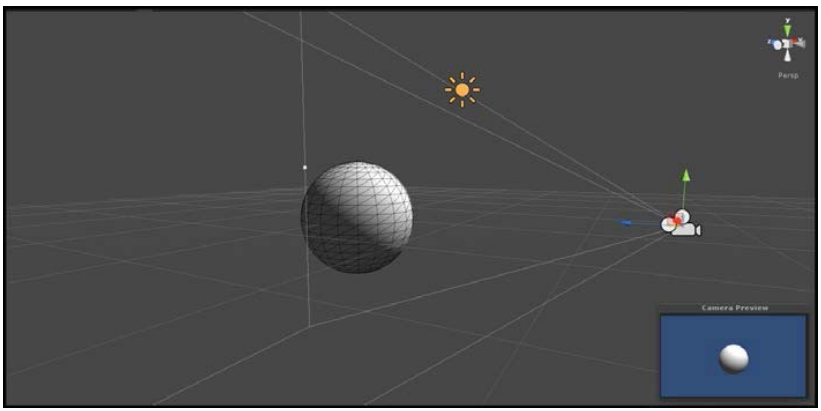

Рис. 8.10. Вид подготовленной сцены

Как это сделать...

В этом рецепте мы напишем с нуля наш шейдер, оптимизированный для мобильных устройств, при этом мы остановимся на конкретных оптимизациях. Выполните следующие действия:

1. Для начала давайте добавим в наш блок свойств текстуры, которые нам понадобятся. В данном случае мы будем использовать диффузную текстуру, содержащую карту блеска в её альфа-канале, а также карту нормалей и слайдер для регулирования интенсивности блика.

```
Properties
{
  _Diffuse ("Base (RGB) Specular (A)", 2D) = "white" {}
```

```
  _SpecIntensity ("Specular Width", Range(0.01, 1)) = 0.5
  _NormalMap ("Normal Map", 2D) = "bump"{}
}
```

2. Далее нам потребуется настроить нашу секцию `#pragma`. Суть настройки состоит во включении или выключении возможностей поверхностного шейдера, от которых будет зависеть, станет он дешевле или дороже в вычислительном плане.

```
CGPROGRAM
#pragma surface surf MobileBlinnPhong exclude_path:prepass
nolightmap noforwardadd halfasview
```

3. После этого нам нужно организовать связь между блоком свойств и секцией `CGPROGRAM`. В противоположность тому, что мы делали обычно, мы будем использовать для слайдера интенсивности блика тип переменной `fixed`, чтобы сократить расход памяти.

```
sampler2D _Diffuse;
sampler2D _NormalMap;
fixed _SpecIntensity;
```

4. Чтобы мы смогли связать наши текстуры с поверхностью объекта, нам потребуются UV-координаты. Теперь мы будем использовать только один набор UV-координат, чтобы в нашем шейдере было задействовано как можно меньше данных.

```
struct Input
{
  half2 uv_Diffuse;
};
```

5. Далее нам потребуется написать нашу функцию освещения и включить в неё несколько новых входных переменных, которые стали нам доступны благодаря инструкциям в секции `#pragma`.

```
inline fixed4 LightingMobileBlinnPhong(SurfaceOutput s,
fixed3 lightDir, fixed3 halfDir, fixed atten)
{
  fixed diff = max(0, dot(s.Normal, lightDir));
  fixed nh = max(0, dot(s.Normal, halfDir));
  fixed spec = pow(nh, s.Specular*128) * s.Gloss;

  fixed4 c;
  c.rgb = (s.Albedo * _LightColor0.rgb * diff + _LightColor0.rgb
* spec) * (atten*2);
  c.a = 0.0;
```

```
  return c;
}
```

6. Завершим мы наш шейдер написанием функции `surf()`, в которой мы будем обрабатывать итоговый цвет нашей поверхности.

```
void surf(Input IN, inout SurfaceOutput o)
{
  fixed4 diffuseTex = tex2D(_Diffuse, IN.uv_Diffuse);
  o.Albedo = diffuseTex.rgb;
  o.Gloss = diffuseTex.a;
  o.Alpha = 0.0;
  o.Specular = _SpecIntensity;
  o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_Diffuse));
}
```

После того как вы закончите написание кода, сохраните ваш шейдер и вернитесь в редактор Unity, чтобы шейдер скомпилировался. Если всё было сделано без ошибок, то вы увидите результат, похожий на изображённый на следующем скриншоте.

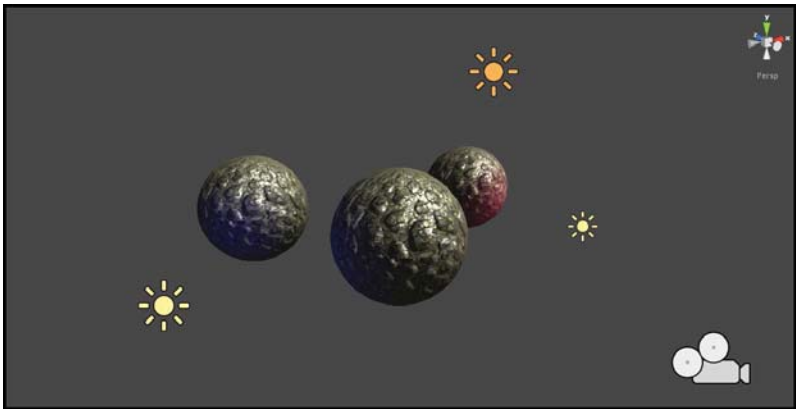

Рис. 8.11. Результат работы шейдера

Как это работает...

Давайте начнём разбор нашего шейдера с того, что разберёмся, что он делает и чего он не делает. Во-первых, он исключает обработку прохода освещения в Deferred-рендеринге. А это значит, что если вы создали функцию освещения, которая была связана с препроходом Deferred-рендерера, то шейдер не будет использовать вашу функцию, вместо этого он задействует функцию освещения по умолчанию, на-

подобие тех, что мы уже использовали в этой книге.

Кроме этого, наш шейдер не поддерживает light mapping, таким образом, наш шейдер не будет пытаться искать карты освещения для объекта, которому назначен шейдер. Это делает наш шейдер легче, так как ему не нужно делать проверку карт освещения.

Мы добавили инструкцию `noforwardadd`, чтобы попиксельное освещение просчитывалось лишь для одного направленного источника света. Все остальные источники света будут обрабатываться повертексно и не будут участвовать в попиксельных операциях вашей функции `surf()`.

И наконец, мы использовали инструкцию `halfasview`, чтобы сказать Unity, что мы не собираемся использовать параметр функции освещения `viewDir`. Вместо этого для расчёта блика мы будем использовать половинный вектор в качестве направления взгляда. При таком подходе шейдер сможет выполнять необходимые вычисления гораздо быстрее, поскольку они будут выполняться повертексно. Нельзя сказать, что этот подход очень уж точен при имитации блика в реальном мире, но на мобильном устройстве это будет незаметно, а наш шейдер будет выполняться быстрее.

Описанные приёмы делают шейдеры более эффективными, а их код – чище. Вам всегда следует использовать только те ресурсы, которые вам будут нужны, а они зависят от вашей целевой платформы и требуемого игрой качества изображения. В конечном итоге сочетания подобных приёмов и позволяют создавать качественные шейдеры для ваших игр.

ГЛАВА 9
Делаем наш шейдерный мир модульным с помощью CgInclude

В этой главе мы рассмотрим следующие темы:

- CgInclude-файлы, встроенные в Unity;
- создание CgInclude-файла для хранения моделей освещения;
- создание шейдеров с использованием директив `#define`.

Введение

За время нашего изучения шейдеров нам уже довелось повидать множество техник и эффектов, но тем не менее, из раза в раз нам приходилось писать одни и те же части кода. В продакшене время стоит дорого, а требования индустрии разработки игр диктуют необходимость быстрого и эффективного итеративного создания шейдеров. Тут-то нам и пригодятся CgInclude-файлы. С их помощью мы можем создать фреймворк, который мы сможем использовать снова и снова, тем самым сделав написание шейдеров модульным.

На самом деле при написании наших поверхностных шейдеров мы уже использовали несколько встроенных CgInclude-файлов. Каждый раз, когда мы использовали встроенные модели освещения Lambert или BlinnPhong, мы использовали код и функции из CgInclude-файлов, уже созданных для нас Unity. Таким образом, снижается количество кода, которое необходимо написать, и выдерживается единообразие освещения и эффектов для всех ваших шейдеров.

Если вы разберётесь с CgInclude-файлами и будете их использовать при написании шейдеров, то сможете значительно ускорить этот процесс, кроме того, если понадобится, такие шейдеры будет легко модифицировать. Так что давайте продолжим изучение шейдеров и

посмотрим, какие модели освещения, функции и встроенные переменные состояний Unity предоставляет нам из коробки.

Встроенные в Unity CgInclude-файлы

Прежде чем начать писать наши собственные CgInclude-файлы, нам нужно разобраться, какие дополнительные возможности для наших шейдеров Unity уже предоставляет по умолчанию. При компиляции поверхностных шейдеров Unity делает очень многое за нас. Именно поэтому процесс написания шейдеров в Unity настолько прост. Функции и макросы, которые используются при генерации шейдерного кода, вы можете найти в папке, куда вы установили Unity, в подпапке `Editor/Data/CGIncludes`. Код из всех этих файлов тем или иным образом участвует в отрисовке объектов на экран. Какие-то из них отвечают за освещение и тени, другие содержат вспомогательные функции, а третьи решают за вас проблемы совместимости платформ. Без этих файлов процесс написания шейдеров был бы гораздо сложнее.

Документацию, предоставляемую Unity, вы можете найти по следующему адресу: http://docs.unity3d.com/Documentation/Components/SL-BuiltinIncludes.html.

Давайте разберёмся с этими встроенными CgInclude-файлами, воспользовавшись некоторыми вспомогательными функциями из файла `UnityCG.cginc`.

Подготовка

До нашего углубления в написание шейдеров нам потребуется подготовить несколько ресурсов для нашей сцены. Прежде чем открыть ваш шейдер в MonoDevelop, выполните следующие действия:

1. Создайте новую сцену и поместите на неё простую модель сферы.
2. Создайте новый шейдер и материал.
3. Назначьте шейдер материалу, а материал – сфере.
4. Создайте направленный источник света и расположите его над сферой.
5. Теперь нам нужно будет открыть файл `UnityCG.cginc` из каталога `CGIncludes`, расположенного в папке, в которую вы установили Unity. Так мы сможем проанализировать код вспомо-

гательных функций и лучше понять, что происходит, когда мы их используем.

6. На этом шаге у вас должна получиться сцена, похожая на изображённую на следующем скриншоте.

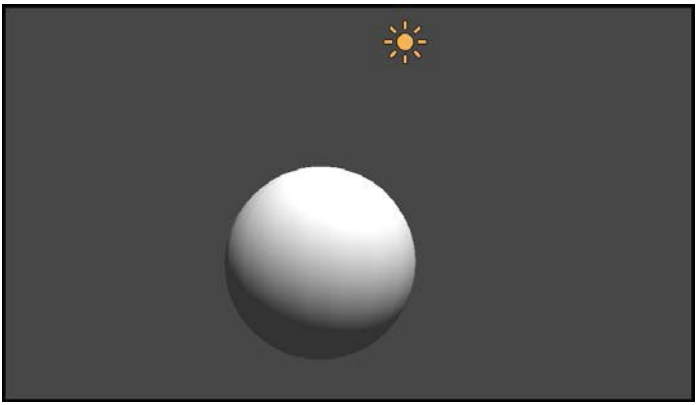

Рис. 9.1. Вид подготовленной сцены

Как это сделать...

После того как мы подготовим сцену, мы можем перейти к экспериментам со встроенными функциями, которые содержатся в файле `UnityCG.cginc`. Сделайте двойной щелчок по созданному для нашей сцены шейдеру, чтобы открыть его в MonoDevelop и вставить в него следующий код:

1. Следующий фрагмент кода добавьте в блок свойств нового шейдера. Мы будем использовать в нашем шейдере одну текстуру и один слайдер.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _DesatValue ("Desaturate", Range(0,1)) = 0.5
}
```

2. После этого давайте убедимся, что мы создали связь между данными из блока свойств и блока `CGPROGRAM`, для этого разместите следующий код после инструкций `CGPROGRAM` и `#pragma`.

```
sampler2D _MainTex;
fixed _DesatValue;
```

3. И наконец, нам нужно добавить в функцию `surf()` следующий код.

```
void surf(Input IN, inout SurfaceOutput o)
{
  half4 c = tex2D(_MainTex, IN.uv_MainTex);

  c.rgb = lerp(c.rgb, Luminance(c.rgb), _DesatValue);

  o.Albedo = c.rgb;
  o.Alpha = c.a;
}
```

После того как вы измените код, вы должны увидеть нечто, похожее на следующий скриншот. Для создания эффекта обесцвечивания основной текстуры нашего шейдера мы воспользовались встроенной функцией Luminance, которая находится в файле `UnityCG.cginc`.

Рис. 9.2. Различные степени преобразования в оттенки серого

Как это работает...

Используя встроенную функцию `Luminance()`, мы можем быстро добавить в наш шейдер эффект обесцвечивания, или если угодно, – эффект перевода в оттенки серого. Мы можем использовать эту функцию потому, что файл `UnityCG.cginc`, в котором она находится, автоматически подключается при работе с поверхностными шейдерами.

Если вы откроете файл `UnityCG.cginc` в MonoDevelop и проскролите до 276 строки, то вы найдёте там реализацию этой функции. Ниже приводится фрагмент кода из этого файла.

```
274|
275|// Преобразуем цвет в оттенки серого.
276|inline fixed Luminance(fixed3 c)
277|{
278| return dot(c, fixed3(0.22, 0.707, 0.071));
279|}
```

Так как эта функция находится в файле `UnityCG.cginc`, который Unity использует при компиляции шейдеров, она автоматически становится доступна нам для использования в наших шейдерах, что уменьшает количество дублирующегося кода, который нам приходится писать. Возможно, вы обратили внимание, что рядом находится файл `Lighting.cginc`. В этом файле объявлены все модели освещения, которые мы можем использовать, когда добавляем в код конструкции вроде `#pragma Surface surf Lambert`. Такая компоновка повышает модульность и степень переиспользуемости кода.

Создание CgInclude-файла для хранения моделей освещения

Знать о существовании встроенных файлов CgInclude – это хорошо, но что делать, если мы хотим написать собственный CgInclude-файл, чтобы хранить свои модели освещения и вспомогательные функции? Для этого сперва нам нужно будет немного разобраться с синтаксисом, прежде чем мы сможем начать эффективно использовать такие файлы при написании шейдеров. Давайте посмотрим, как сделать CgInclude-файл с нуля.

Подготовка

На примере той же сцены, шейдера и материала давайте пройдёмся по процессу создания нового файла для этого рецепта. Выполните следующие действия:

1. Первым делом создайте новый текстовый файл и назовите его, например, `MyCgInclude.txt`.
2. После этого измените его расширение на `.cginc`. Windows выдаст окошко предупреждения, в котором будет написано, что файл таким образом может стать непригодным, но не верьте ему – файл будет работать.
3. Импортируйте этот новый файл `.cginc` в проект Unity и позвольте ему скомпилироваться. Если всё пройдёт успешно, вы увидите, что Unity смогла скомпилировать его в CgInclude-файл.

Теперь мы готовы к тому, чтобы добавить наш собственный код в этот файл. Сделайте двойной щелчок по созданному вами CgInclude-файлу, чтобы открыть его в MonoDevelop.

Как это сделать...

Открыв CgInclude-файл, мы можем приступить к написанию кода, который будет работать в нашем поверхностном шейдере. Добавив следующий код, вы подготовите наш CgInclude-файл к работе с нашими поверхностными шейдерами, при этом вы сможете и в дальнейшем добавлять код в этот файл по мере написания шейдеров. Выполните следующую последовательность действий:

1. Мы начнём наш CgInclude-файл с так называемой директивы препроцессора. Директивы – это инструкции наподобие `#pragma` и `#include`. Добавьте следующий код в начале вашего файла.

```
#ifndef MY_CG_INCLUDE
#define MY_CG_INCLUDE
```

2. Затем каждый раз нам нужно проверять, что мы закрыли наши секции `#ifndef` или `#ifdef` с помощью `#endif`. Нам нужно закрывать их, так же как, например, в операторе `if` в языке C# должны быть две скобки `()`. Добавьте следующий код сразу после директивы `#define`.

```
#endif
```

3. Теперь мы можем заняться содержимым CgInclude-файла. Поэтому мы добавляем в него следующий код.

```
//Настраиваемые встроенные переменные.
fixed4 _MyColor;

//Модели освещения.
inline fixed4 LightingHalfLambert(SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  fixed diff = max(0, dot(s.Normal, lightDir));

  diff = (diff + 0.5) * 0.5;

  fixed4 c;
  c.rgb = s.Albedo * _LightColor0.rgb * ((diff * _MyColor.
rgb) * atten * 2);
  c.a = s.Alpha;
  return c;
}
```

4. После того как вы завершите предыдущие действия, у вас получится первый готовый CgInclude-файл. С помощью этих нескольких строк кода мы можем существенно сократить количество кода, которое нам нужно будет переписывать каждый

раз. Теперь мы можем хранить в этом файле наши модели освещения, которыми мы постоянно пользуемся, чтобы не потерять их. Код получившегося CgInclude-файла приведён ниже.

```
#ifndef MY_CG_INCLUDE
#define MY_CG_INCLUDE

//Настраиваемые встроенные переменные.
fixed4 _MyColor;

//Модели освещения.
inline fixed4 LightingHalfLambert(SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  fixed diff = max(0, dot(s.Normal, lightDir));

  diff = (diff + 0.5) * 0.5;

  fixed4 c;
  c.rgb = s.Albedo * _LightColor0.rgb * ((diff * _MyColor.
rgb) * atten * 2);
  c.a = s.Alpha;
  return c;
}

#endif
```

Но нам потребуется выполнить ещё несколько действий, прежде чем мы сможем использовать код из этого CgInclude-файла. Нам необходимо дать указание шейдеру, с которым мы работаем, что он должен использовать этот файл и его код. Выполните следующие действия.

5. Теперь, когда мы сконцентрировались на нашем шейдере, нам нужно подключить наш новый CgInclude-файл, чтобы можно было добраться до содержащегося в нём кода. Добавьте к директивам компилятора директиву `#include`.

```
CGPROGRAM
#include "MyCGInclude.cginc"
#pragma surface surf Lambert
```

6. Пока что наш шейдер использует встроенную модель освещения Lambert, но мы хотим, чтобы он использовал созданную нами в нашем CgInclude-файле модель освещения Half Lambert. А поскольку мы подключили наш CgInclude-файл, мы можем использовать модель освещения Half Lambert с помощью следующего кода.

```
CGPROGRAM
#include "MyCGInclude.cginc"
#pragma surface surf HalfLambert
```

7. И наконец, мы объявили переменную в нашем CgInclude-файле, чтобы показать, что мы можем подготовить переменные по умолчанию для использования в нашем шейдере. Чтобы увидеть, как это работает, в блок свойств вашего шейдера добавьте следующий код.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _DesatValue ("Desaturate", Range(0,1)) = 0.5
  _MyColor ("My Color", Color) = (1,1,1,1)
}
```

Если не было допущено ошибок, то после возвращения в Unity файл CgInclude и шейдер скомпилируются, и вы увидите, что наш шейдер использует нашу новую модель освещения Half Lambert, а на закладке **Инспектора** материалов появилась палитра цвета. На следующем скриншоте показан результат использования нашего CgInclude-файла.

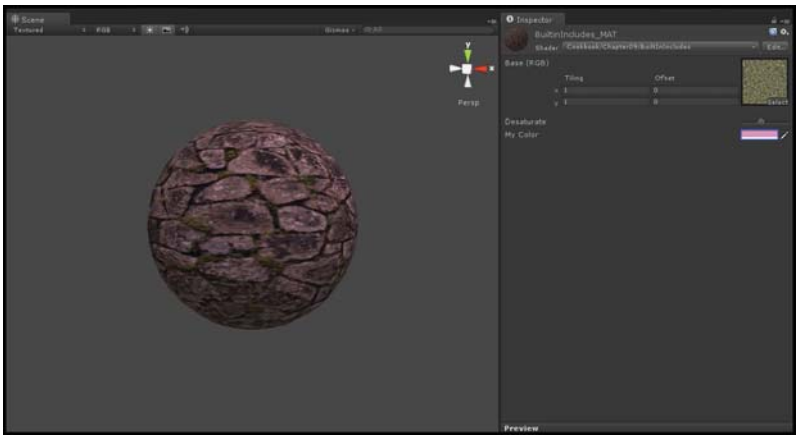

Рис. 9.3. Результат работы нашего шейдера, использующего файлы CgInclude

Как это работает...

При работе с шейдерами мы можем импортировать внешние блоки кода с помощью директивы препроцессора `#include`. Эта директива

говорит Unity, что мы хотим использовать в нашем шейдере код, находящийся в указанном файле, – с этой директивой и связано название файлов CgInclude. Мы «включаем» (include) фрагменты Cg-кода с помощью директивы `#include`.

На самом деле `#include` не создает «ссылок» на файлы, как, например, оператор `import` в Java, он просто копирует при компиляции код из указанного файла в ваш шейдер. Так что если два раза включить один и тот же файл, то в шейдере окажется два одинаковых блока кода с дублирующимися названиями функций и переменных, и такой шейдер не скомпилируется. Конечно, если вы работаете только с одним CgInclude-файлом, легко проследить, чтобы подобного не произошло. Но когда у вас будет несколько CgInclude-файлов, которые ссылаются друг на друга, лучше пользоваться старым проверенным методом работы с помощью `#ifndef`. В данном примере мы используем директиву `#ifndef`, чтобы проверить, не был ли определен идентификатор с именем `MY_CG_INCLUDE`, если нет, мы определяем его с помощью директивы `#define` и вставляем необходимый код. Иначе не делаем ничего. Таким образом, если наш CgInclude-файл был уже когда-то скопирован в код шейдера, идентификатор `MY_CG_INCLUDE` будет уже объявлен, и Unity пропустит дублирующийся код.

Теперь вы видите, какие преимущества даёт нам использование CgInclude-файлов, в которых мы можем хранить все наши модели освещения и наши собственные переменные, что в большой степени снижает количество кода, которое нам нужно писать. Но, ещё большие возможности для придания гибкости вашему шейдеру даёт объявление нескольких состояний функций в файлах CgInclude.

Использование #define в шейдерах

Теперь мы умеем использовать встроенные в Unity CgInclude файлы, а также создавать собственные несложные CgInclude-файлы для объединения в них наших моделей освещения, переменных и вспомогательных функций. Давайте посмотрим, как мы можем использовать CgInclude-файлы более динамично и более эффективно, чтобы повысить степень модульности наших шейдеров с помощью состояний, которые мы можем включать и выключать.

Для демонстрации этой идеи мы изменим нашу модель освещения Half Lambert, которую мы создали в последнем рецепте, так чтобы

иметь возможность включать и выключать эффект Half Lambert. Таким образом, если мы в шейдере объявляем, что используется Half Lambert, наша модель освещения изменится со стандартной модели освещения **NdotL** на модель Half Lambert.

Давайте посмотрим, как это работает, используя уже созданные нами ассеты. Нам потребуется внести в код лишь небольшие изменения.

Как это сделать...

Начнём мы с того, что обратим внимание на наш CgInclude-файл. Мы хотим, чтобы модель освещения имела два состояния. Для этого выполните следующие действия:

1. Первым состоянием нашей модели будет модель диффузного освещения NdotL, а вторым состоянием – модель освещения Half Lambert. Добавьте в ваш файл CgInclude следующий код.

```
#ifndef MY_CG_INCLUDE
#define MY_CG_INCLUDE

//Настраиваемые встроенные переменные.
fixed4 _MyColor;

//Модели освещения.
inline fixed4 LightingCustomLambert(SurfaceOutput s, fixed3
lightDir, fixed atten)
{
  fixed diff = max(0, dot(s.Normal, lightDir));

  #ifdef HalfLambert
  diff = (diff + 0.5)*0.5;
  #endif

  fixed4 c;
  c.rgb = s.Albedo * _LightColor0.rgb *
((diff * _MyColor.rgb) * atten * 2);
  c.a = s.Alpha;
  return c;
}

#endif
```

2. Далее нам нужно будет обновить у нашего шейдера директивы компилятора.

```
CGPROGRAM
#define HalfLambert

#include "MyCGInclude.cginc"
#pragma surface surf CustomLambert
```

3. Сохраните ваш CgInclude-файл и шейдер, затем вернитесь в Unity, чтобы они скомпилировались. Если всё прошло хорошо, то вы не заметите различий. Это потому, что мы дали указание Unity объявить идентификатор `HalfLambert`, который используется в коде нашей функции освещения. Таким образом, код между `#ifdef` и `#endif` будет выполнен.
4. Вернитесь к вашему шейдеру и закомментируйте только что добавленный дефайн. Сохраните его и вернитесь в Unity, чтобы он скомпилировался.

```
CGPROGRAM
//#define HalfLambert

#include "MyCGInclude.cginc"
#pragma surface surf CustomLambert
```

Если всё прошло хорошо, то вы увидите, что наш шейдер теперь использует стандартную модель освещения NdotL. Так происходит потому, что теперь мы больше не объявляем идентификатор `HalfLambert`, поэтому Unity при компиляции пропускает этот участок кода. С помощью этого приёма написание шейдеров становится более гибким и более эффективным, поскольку нам не приходится постоянно переписывать или удалять большие участки кода. На следующем скриншоте приводится результат работы нашего нового модульного шейдера.

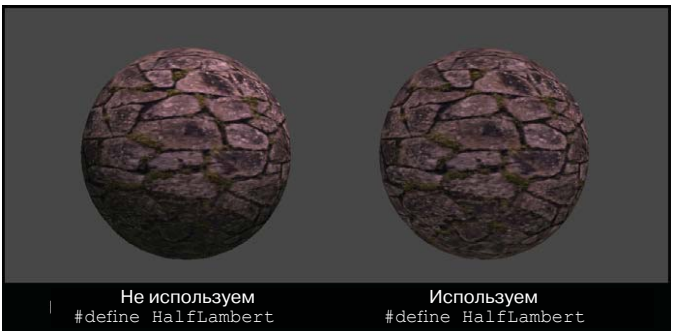

Рис. 9.4. Результат работы шейдера с и без директивы `#define HalfLambert`

Как это работает...

Как видите, объём необходимого нам на данном этапе кода минимален. Используя этот простой метод, мы можем создать большое число вариаций наших моделей освещения. С помощью директивы `#ifdef` мы говорим Unity искать определение, следующее за `#ifdef`, в нашем случае – `HalfLambert`.

Когда мы объявляем в нашем шейдере директиву `#define`, Unity начинает искать указанное определение во всех подключенных файлах. Если Unity встретит нужный `#ifdef`, то она будет использовать код, располагающийся до директивы `#endif`. Нужно понимать, насколько ответственным становится задание имён для этих идентификаторов, ведь вам придётся удостовериться, что вы не собираетесь использовать уже задействованное имя.

Повышение эффективности написания шейдеров с помощью CgInclude-файлов заключается не только в том, что нам не приходится дублировать большие объёмы кода в каждом шейдере, но также и в том, что с помощью этих файлов можно хранить большое количество моделей освещения. Таким образом, становится гораздо проще вспомнить и выбрать нужную модель освещения, а также, например, изменить её так, чтобы она использовала несколько состояний. Представьте, каково было бы попытаться запомнить все модели освещения из этой книги или даже просто выписать их в блокнотик для дальнейшего применения. Использование CgInclude-файлов поможет вам стать более эффективным и организованным шейдерным программистом.

ГЛАВА 10
Создание экранных эффектов в Unity с помощью рендер-текстур

В этой главе вы узнаете о:

- написании скриптов для полноэкранных эффектов;
- реализации эффектов изменения яркости, насыщенности и контраста;
- создании режимов блендинга на подобие Photoshop;
- реализации режима блендинга Overlay.

Введение

Один из наиболее впечатляющих аспектов изучения шейдеров, – это написание ваших собственных полноэкранных эффектов, также известных как постэффекты или фулскрин-эффекты (от англ. *fullscreen*, – полноэкранный). Используя фулскрин-эффекты, мы можем существенно улучшить картинку, например при помощи эффектов Bloom, Motion Blur, HDR и других. Большинство современных игр активно использует постэффекты, например для реализации эффектов глубины (Depth of Field или DOF), свечения (Bloom) или коррекции цвета.

В этой главе мы начнём с изучения скриптов, которые позволят нам создавать эти эффекты. Мы узнаем о **рендер-текстурах** (render texture), о том, что такое **буфер глубины** (depth buffer), и о создании эффектов, с помощью которых мы можем так же детально контролировать цвет финального рендера, как в Photoshop. Начав использовать полноэкранные эффекты в своих играх, вы не только пополните свою копилку приёмов написания шейдеров, но и получите возможность создавать в Unity невероятно красивую картинку в реальном времени.

Создание скриптов для полноэкранных эффектов

Полноэкранные эффекты работают следующим образом: Unity рендерит изображение с камеры, отдаёт текстуру в эффект, который, используя шейдер на GPU, нужным образом изменяет каждый пиксель текстуры и возвращает модифицированное изображение обратно. Таким образом, мы можем выполнять в реальном времени попиксельные операции над отрендеренным изображением игры, что даёт нам больший художественный контроль над финальной картинкой.

Представьте, какого было бы корректировать каждый материал каждого отдельного объекта в вашей игре, чтобы всего лишь изменить контрастность итогового изображения. Конечно, можно было бы поступить и так, но на это ушло бы очень много времени. С помощью фулскрин-эффектов мы можем корректировать итоговое изображение сразу на всём экране, что даёт нам контроль на уровне Photoshop за финальной картинкой.

Чтобы начать разбираться в постэффектах, нам потребуется написать скрипт, который будет выступать в роли посредника между Unity и шейдером, и передавать текущее отрендеренное изображение в виде рендер-текстуры (render texture). Настроив скрипт для передачи текстуры шейдеру, мы сможем использовать его для создания различных полноэкранных эффектов. В качестве нашего первого эффекта мы напишем очень простой эффект перевода в оттенки серого, при применении которого наша игра станет чёрно-белой. Давайте посмотрим, как же это сделать.

Подготовка

Для начала нам нужно будет подготовить несколько ассетов в нашем Unity-проекте. Для этого выполните следующие действия:

1. В текущем проекте создайте новый C#-скрипт и назовите его `TestRenderImage.cs`.
2. Создайте новый шейдер и назовите его `ImageEffect.shader`.
3. Создайте простую сферу и назначьте ей новый материал. Это может быть совершенно произвольный материал, но для нашего примера мы решили использовать простой Specular-материал красного цвета.

4. Далее создайте новый направленный источник света и сохраните сцену.

После того, как вы подготовите все необходимые ресурсы, ваша сцена должна выглядеть примерно как на следующем скриншоте.

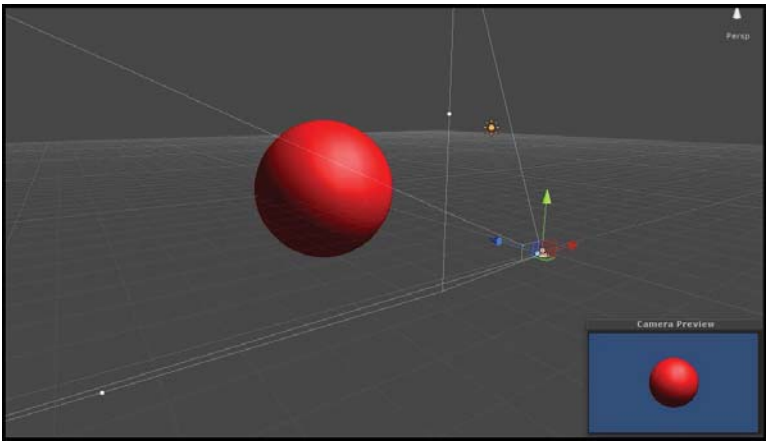

Рис. 10.1. Вид подготовленной сцены

Как это сделать...

Чтобы сделать эффект перевода в оттенки серого, нам понадобятся соответствующий скрипт и шейдер. Поэтому сейчас мы займёмся их созданием. Сначала мы приступим к написанию C#-скрипта. Этот скрипт будет базой для всех наших фулскрин-эффектов. После этого мы напишем шейдер и посмотрим на результаты применения нашего эффекта.

Выполните следующие действия:

1. Откройте C#-скрипт `TestRenderImage.cs` и создайте несколько переменных, которые нам потребуются для хранения важных объектов и данных. Добавьте следующий код в самом начале класса `TestRenderImage`.

```
public class TestRenderImage : MonoBehaviour
{
  #region Variables
  public Shader curShader;
  public float grayScaleAmount = 1.0f;
  private Material curMaterial;
  #endregion
```

2. Чтобы мы смогли редактировать эффекты в реальном времени, когда редактор Unity не находится в режиме play, нам потребуется добавить следующую строчку перед объявлением класса `TestRenderImage`.

```
[ExecuteInEditMode]
public class TestRenderImage : MonoBehaviour
{
```

3. Поскольку наш фулскрин-эффект использует шейдер для выполнения попиксельных операций с отрендеренным изображением для работы шейдера, нам потребуется создать материал. Без него мы не сможем получить доступ к свойствам шейдера. Поэтому мы создаём C#-свойство, которое будет проверять материал и создавать его, если он не был создан. Добавьте следующий код сразу после объявленных на первом шаге переменных.

```
#region Properties
Material material
{
  get
  {
          if(curMaterial == null)
          {
                curMaterial = new Material(curShader);
                curMaterial.hideFlags = HideFlags.HideAndDontSave;
          }
          return curMaterial;
  }
}
#endregion
```

4. Теперь нам нужно проверить, поддерживает ли текущая платформа, под которую мы делаем игру, фулскрин-эффекты. Если не поддерживает, скрипт автоматически выключит себя.

```
void Start()
{
  if(!SystemInfo.supportsImageEffects)
  {
          enabled = false;
          return;
  }

  if(!curShader && !curShader.isSupported)
  {
          enabled = false;
  }
}
```

5. Для того чтобы получать отрендеренное изображение в нашем эффекте, нам нужно добавить в скрипт метод `OnRenderImage()`, в который Unity будет передавать рендер-текстуру с изображением с экрана.

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture
destTexture)
{
  if(curShader!= null)
  {
          material.SetFloat("_LuminosityAmount", grayScaleAmount);
          Graphics.Blit(sourceTexture, destTexture, material);
  }
  else
  {
          Graphics.Blit(sourceTexture, destTexture);
  }
}
```

6. В нашем эффекте мы определили переменную `grayScaleAmount`, которую мы используем для контроля над степенью обесцвечивания итоговой картинки. Её значения должны лежать в диапазоне от 0 до 1, где 0 – эффект выключен, 1 – изображение полностью черно-белое. Мы будем проверять это в методе `Update()`, который выполняется каждый кадр.

```
void Update()
{
  grayScaleAmount = Mathf.Clamp(grayScaleAmount, 0.0f, 1.0f);
}
```

7. И наконец, мы завершим написание скрипта, убедившись, что при его выключении мы удаляем все созданные нами объекты.

```
void OnDisable()
{
  if(curMaterial)
  {
          DestroyImmediate(curMaterial);
  }
}
```

Теперь, если скрипт `TestRenderImage.cs` скомпилировался в Unity без ошибок, мы можем назначить его камере. Давайте назначим его нашей мэйн (от англ. *main*, – главный) камере в сцене. В свойствах скрипта вы должны увидеть значение переменной `grayScaleAmount` и поле для шейдера, но при этом скрипт выдаёт ошибку в окне кон-

соли. В сообщении об ошибке написано, что отсутствует экземпляр объекта и скрипт не может корректно выполниться. Если вы помните, на 4-ом шаге мы осуществляем ряд проверок на наличие шейдера и на его поддержку текущей платформой. А поскольку мы не задали шейдер для нашего скрипта, переменная `curShader` содержит значение `null`, что и приводит к появлению ошибки.

Давайте продолжим работу над нашим фулскрин-эффектом, написав шейдер для него:

1. Начнём написание шейдера с создания переменных, с помощью которых мы сможем передавать данные в шейдер.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _LuminosityAmount ("GrayScale Amount", Range(0.0, 1)) = 1.0
}
```

2. Наш шейдер не будет использовать возможности поверхностных шейдеров в Unity, вместо этого мы сами реализуем нужную нам пиксельную функцию. Таким образом, мы получим более эффективный шейдер для нашего фулскрин-эффекта. Для этого мы создадим новый блок `Pass` в нашем шейдере и добавим в него пока что ещё новые для нас директивы `#pragma`.

```
SubShader
{
  Pass
  {
          CGPROGRAM
          #pragma vertex vert_img
          #pragma fragment frag
          #pragma fragmentoption ARB_precision_hint_fastest
          #include "UnityCG.cginc"
```

3. Чтобы получить доступ к данным, пересылаемым в шейдер из редактора Unity, нам потребуется объявить соответствующие переменные в блоке `CGPRGORAM`.

```
uniform sampler2D _MainTex;
fixed _LuminosityAmount;
```

4. И наконец, всё, что нам остаётся сделать, – это подготовить пиксельную функцию, в нашем случае `frag()`. Тут-то в нашем полноэкранном эффекте и будет происходить всё самое интересное. Эта функция будет обрабатывать каждый пиксель

рендер-текстуры и возвращать новое изображение скрипту `TestRenderImage.cs`.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
  fixed4 renderTex = tex2D(_MainTex, i.uv);

  //Применим значение яркости к нашей рендер-текстуре.
  float luminosity = 0.299 * renderTex.r + 0.587 * renderTex.g +
0.114 * renderTex.b;
  fixed4 finalColor = lerp(renderTex, luminosity, _LuminosityAmount);

  return finalColor;
}
```

После того как наш шейдер будет готов, вернитесь в Unity, чтобы он скомпилировался, и убедитесь, что не было допущено ошибок. Если всё прошло хорошо, назначьте скрипту `TestRenderImage.cs` новый шейдер и измените значение переменной `grayscale`. После этого вы должны увидеть, как ваша игра превращается из цветной в чёрно-белую. Следующее изображение наглядно иллюстрирует результат применения этого эффекта.

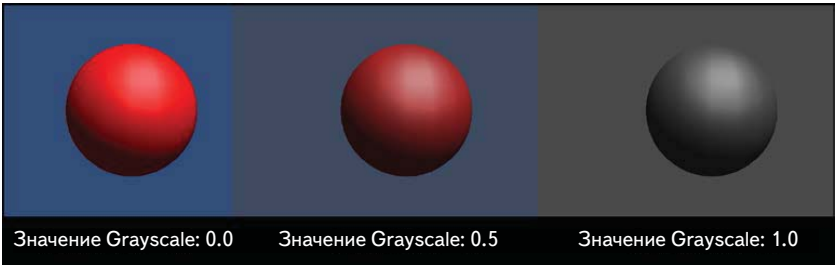

Рис. 10.2. Результат применения фулскрин-эффекта с разными значениями переменной `grayscale`

В итоге у нас появился удобный способ тестировать шейдеры фулскрин-эффектов, избавляющий нас от необходимости каждый раз заново писать C#-скрипты. Давайте копнём чуть глубже и узнаем подробнее о рендер-текстурах и о том, как данные из них обрабатываются на разных этапах выполнения эффектов.

Как это работает...

Для полноценного фулскрин-эффекта в Unity нам необходимо создать скрипт и шейдер. Скрипт нужен для настройки и обновления параметров в реальном времени, а также для передачи рендер-текстуры от камеры, на которой он висит, в шейдер. Шейдер получает текстуру, попиксельно изменяет её и возвращает обратно модифицированное изображение.

В начале скрипта мы проверяем, что текущая целевая платформа поддерживает полноэкранные эффекты и способна выполнить данный шейдер. Потому что вполне может быть, что выбранная платформа не поддерживает полноэкранных эффектов, или используемого нами шейдера. Проверяя это в функции `Start()` при первом запуске скрипта, мы избавляемся от неправильного поведения и неминуемых ошибок во время игры.

Добавив в скрипт метод `OnRenderImage()`, мы говорим Unity, что хотим получать картинку с камеры. Как вы уже, наверное заметили, у этого метода есть два параметра: `sourceTexture` и `destTexture`. В первой текстуре находится изображение с камеры, во вторую текстуру нам нужно поместить измененное изображение. Мы это делаем с помощью функции `Graphics.Blit()`, которая просто-напросто рендерит полноэкранный квад с текстурой и материалом в заданную рендер-текстуру. Вы можете найти дополнительную информацию об этих двух функциях по следующим адресам:

- **OnRenderImage** (http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnRenderImage.html);
- **Graphics.Blit** (http://docs.unity3d.com/Documentation/ScriptReference/Graphics.Blit.html).

Мы рассмотрели очень простой пример, но этот пример показывает основные элементы и возможности фулскрин-эффектов. Скорее всего, вы уже понимаете, насколько мощный инструмент у нас в руках. Используя фулскрин-эффекты, мы получаем контроль над финальной картинкой на уровне таких графических редакторов, как Photoshop. Например, применяя несколько эффектов к камере, можно думать о них как о слоях в Photoshop. Эффекты будут выполнены один за другим, в той же последовательности, в какой они были размещены.

Но это ещё не всё...

Итак, мы подготовили C#-скрипт и шейдер для базового полноэкранного эффекта, а теперь давайте посмотрим, какую ещё полезную информацию можно получить от рендерера Unity.

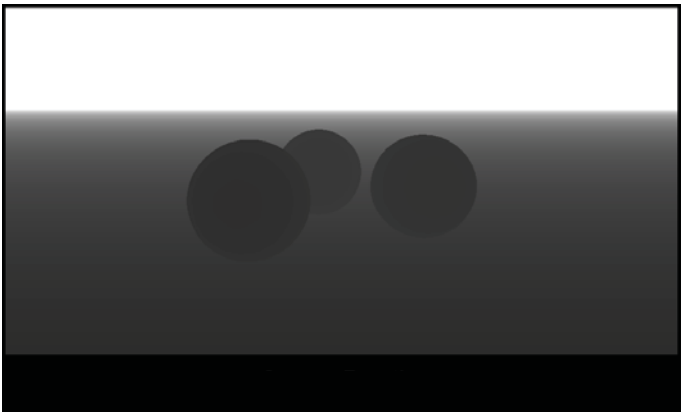

Рис. 10.3. Глубина сцены

Например, включив встроенный в Unity рендеринг глубины у камеры, мы получим доступ к информации о глубине сцены относительно камеры. Если его включить, то мы сможем использовать данные о глубине для большого количества различных эффектов. Давайте посмотрим, как это делается:

1. Создайте новый шейдер и назовите его `SceneDepth_Effect`. После этого кликните по нему два раза, чтобы открыть его в редакторе MonoDevelop.
2. Далее мы создадим свойство для основной текстуры и свойство для контроля за интенсивностью эффекта глубины сцены. Для этого добавьте в ваш шейдер следующий код:

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _DepthPower ("Depth Power", Range(1, 5)) = 1
}
```

3. Теперь нам потребуется добавить соответствующие переменные в наш блок `CGPROGRAM`. Мы добавим ещё одну переменную под названием `_CameraDepthTexture`. Это встроенная в Unity переменная, которая определена в файле `UnityCG.cginc`. В этой переменной содержится информация о глубине, получаемая от камеры.

```
Pass
{
  CGPROGRAM
```

```
#pragma vertex vert_img
#pragma fragment frag
#pragma fragmentoption ARB_precision_hint_fastest
#include "UnityCG.cginc"

uniform sampler2D _MainTex;
fixed _DepthPower;
sampler2D _CameraDepthTexture;
```

4. Мы заканчиваем наш шейдер, используя ещё пару встроенных в Unity функций: `UNITY_SAMPLE_DEPTH()` и `Linear01Depth()`. Первая функция получает информацию о глубине из текстуры `_CameraDepthTexture` и возвращает для каждого пикселя число типа `float`. Но возвращаемые ей значения нелинейны по глубине, так что мы используем функцию `Linear01Depth()`, чтобы получить линейное распределение глубины от 0 до 1. Возводя эти значения в степень, мы можем контролировать, где, относительно положения камеры, будет находиться центральная точка глубины.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
  float d = UNITY_SAMPLE_DEPTH(tex2D(_CameraDepthTexture, i.uv.xy));
  d = pow(Linear01Depth(d), _DepthPower);

  return d;
}
```

5. Теперь, когда мы доделали наш шейдер, давайте обратим внимание на скрипт эффекта. Во-первых, нам нужно добавить переменную `depthPower`, чтобы пользователь мог изменять это значение в шейдере.

```
#region Variables
public Shader curShader;
private material curMaterial;

public float depthPower = 1.0f;
#endregion
```

6. После этого нам нужно обновить метод `OnRenderImage()`, чтобы в нём в шейдер передавалось значение `depthPower`.

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture
destTexture)
{
  if(curShader != null)
```

```
  {
        material.SetFloat("_DepthPower", depthPower);
        Graphics.Blit(sourceTexture, destTexture, material);
  }
  else
  {
        Graphics.Blit(sourceTexture, destTexture);
  }
}
```

7. Чтобы завершить создание эффекта глубины, мы должны сказать Unity включить рендеринг глубины для текущей камеры. Чтобы это сделать, нужно присвоить определённое значение свойству `depthTextureMode` у камеры.

```
void Update()
{
  Camera.main.depthTextureMode = DepthTextureMode.Depth;
  depthPower = Mathf.Clamp(depthPower, 0, 5);
}
```

После того как вы напишете этот код, сохраните ваш скрипт и шейдер, а потом вернитесь в Unity, чтобы они скомпилировались. Если не было допущено ошибок, вы увидите нечто, похожее на следующий скриншот.

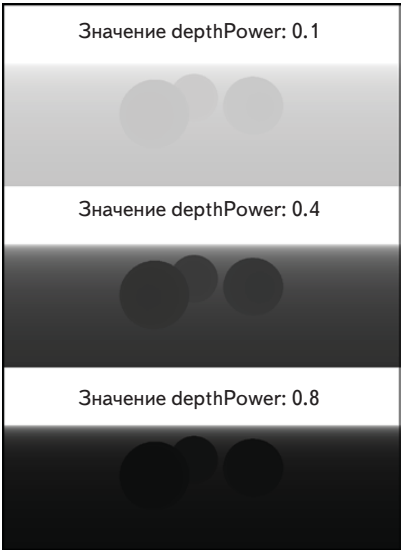

Рис. 10.4. Результаты, получаемые при различных значениях `depthPower`

Корректировка яркости, насыщенности и контраста с помощью полноэкранных эффектов

Теперь, когда у нас есть шаблоны C# скрипта и шейдера для фулскрин-эффектов, мы можем перейти к изучению использования более сложных попиксельных операций для создания визуальных эффектов, наиболее распространённых в современных играх.

Использование постэффектов для корректировки итогового цвета в вашей игре необходимо для предоставления художнику глобального контроля над финальным изображением на экране игрока. Под контролем мы подразумеваем методы корректировки цвета с помощью слайдеров интенсивности красного, синего и зелёного или методы наложения некоторого тона цвета для всего экрана наподобие эффекта сепии.

В этом рецепте мы рассмотрим наиболее базовые операции преобразования цвета: операции коррекции яркости, насыщенности и контраста. Научившись писать эти эффекты, вы получите достаточный базис, чтобы приступить к изучению куда более сложных примеров.

Подготовка

Первым делом нам потребуется подготовить необходимые ассеты. Мы можем использовать нашу тестовую сцену, однако шейдер и скрипт нам потребуются новые. Выполните следующие действия:

1. Создайте новый скрипт и назовите его `BSC_ImageEffect`.
2. Создайте новый шейдер и назовите его `BSC_Effect`.
3. Теперь нам нужно скопировать код C#-скрипта из предыдущего рецепта в наш новый C#-скрипт. Таким образом мы сможем сфокусироваться непосредственно на математике эффектов яркости, насыщенности и контраста.
4. Скопируйте в новый шейдер код шейдера из предыдущего рецепта.
5. Создайте несколько новых объектов на сцене и назначьте им материалы разных цветов. Так мы получим хороший набор цветов для тестирования наших новых постэффектов.

После того как всё будет готово, у вас должна получиться сцена, похожая на изображение на следующем скриншоте.

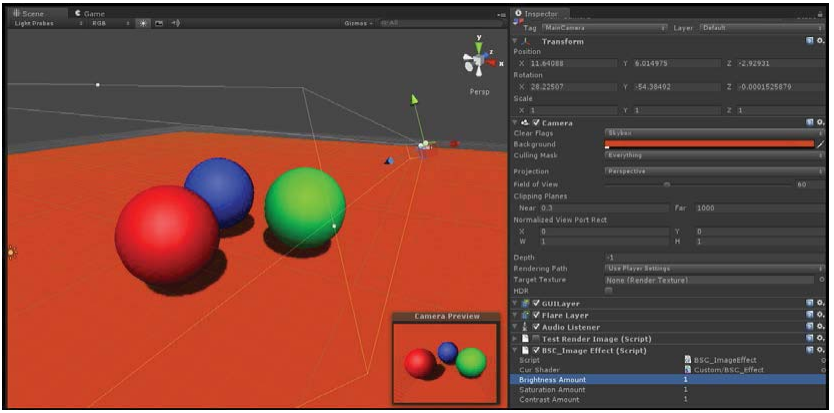

Рис. 10.5. Вид подготовленной сцены

Как это сделать...

Теперь, когда мы подготовили нашу сцену и создали новый скрипт и шейдер, мы можем перейти к написанию кода, необходимого для создания эффектов яркости, насыщенности и контраста. Мы сфокусируемся лишь на попиксельных операциях и настройке переменных для нашего скрипта и шейдера, поскольку подготовка скриптов для создания фулскрин-эффектов уже была описана в предыдущем рецепте. Выполните следующие действия:

1. Откройте ваш новый шейдер и новый скрипт в редакторе MonoDevelop. Для этого просто кликните по ним два раза.
2. Целесообразнее сначала приступить к редактированию шейдера, поскольку так мы сможем узнать, какие переменные потребуются нашему C#-скрипту. Давайте для начала создадим свойства для яркости, насыщенности и контраста. Также не забудьте, что нам нужно оставить свойство `_MainTex`, так как именно через него передаётся рендер-текстура с изображением от камеры.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _BrightnessAmount ("Brightness Amount", Range(0.0, 1)) = 1.0
  _satAmount ("Saturation Amount", Range(0.0, 1)) = 1.0
```

```
  _conAmount ("Contrast Amount", Range(0.0, 1)) = 1.0
}
```

3. Как обычно, чтобы мы смогли получить доступ из блока CGPROGRAM к нашим свойствам, нам потребуется создать соответствующие переменные.

```
Pass
{
  CGPROGRAM
  #pragma vertex vert_img
  #pragma fragment frag
  #pragma fragmentoption ARB_precision_hint_fastest
  #include "UnityCG.cginc"

  uniform sampler2D _MainTex;
  fixed _BrightnessAmount;
  fixed _satAmount;
  fixed _conAmount;
```

4. Теперь нам нужно реализовать операции изменения яркости, насыщенности и контраста. Добавьте следующий код в наш шейдер выше функции `frag()`. Не волнуйтесь, если что-то будет непонятно, – в следующей секции мы объясним все сложные моменты.

```
float3 ContrastSaturationBrightness(float3 color, float brt,
float sat, float con)
{
  //Увеличивайте или уменьшайте значения переменных
  //для независимой корректировки значений каналов r, g, b.
  float AvgLumR = 0.5;
  float AvgLumG = 0.5;
  float AvgLumB = 0.5;

  //Коэффициенты Luminance для извлечения освещённости из текстуры.
  float3 LuminanceCoeff = float3(0.2125, 0.7154, 0.0721);

  //Действия над яркостью.
  float3 AvgLumin = float3(AvgLumR, AvgLumG, AvgLumB);
  float3 brtColor = color * brt;
  float intensityf = dot(brtColor, LuminanceCoeff);
  float3 intensity = float3(intensityf, intensityf, intensityf);

  //Действия над насыщенностью.
  float3 satColor = lerp(intensity, brtColor, sat);

  //Действия над контрастом.
  float3 conColor = lerp(AvgLumin, satColor, con);
  return conColor;
}
```

5. И наконец, нам нужно изменить функцию `frag()`, чтобы она использовала только что созданную функцию `ContrastSaturationBrightness()`. Таким образом, код в этой функции будет выполнен для каждого пикселя рендер-текстуры, а результат передан обратно в наш скрипт.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
  fixed4 renderTex = tex2D(_MainTex, i.uv);

  //Применим преобразования для яркости, насыщенности и контраста.
  renderTex.rgb = ContrastSaturationBrightness(renderTex.rgb,
                                               _BrightnessAmount,
                                               _satAmount,
                                               _conAmount);
  return renderTex;
}
```

Изменив код, вернитесь в редактор Unity, чтобы ваш новый шейдер скомпилировался. Если не было допущено ошибок, мы вернёмся в MonoDevelop, чтобы поработать над нашим скриптом. Давайте начнём с того, что добавим несколько строк кода, которые будут отвечать за передачу необходимых данных в шейдер. Для этого выполните следующие действия:

1. Первым делом нам нужно добавить переменные, в которых будут храниться параметры наших эффектов. В данном случае нам будут нужны слайдер для яркости, слайдер для насыщенности и слайдер для контраста.

```
#region Variables
public Shader curShader;
public float brightnessAmount = 1.0f;
public float saturationAmount = 1.0f;
public float contrastAmount = 1.0f;
private Material curMaterial;
#endregion
```

2. После настройки необходимых переменных нам нужно передать их значения в шейдер. Мы это делаем в методе `OnRenderImage()`:

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture
destTexture)
{
  if(curShader != null)
```

```
    {
            material.SetFloat("_BrightnessAmount", brightnessAmount);
            material.SetFloat("_satAmount", saturationAmount);
            material.SetFloat("_conAmount", contrastAmount);

            Graphics.Blit(sourceTexture, destTexture, material);
    }
    else
    {
            Graphics.Blit(sourceTexture, destTexture);
    }
}
```

3. И наконец, мы добавим код, который будет ограничивать значения параметров в разумном диапазоне. Параметры ограничения – дело вкуса, поэтому вы можете сами подобрать значения, которые вам нравятся.

```
void Update()
{
  brightnessAmount = Mathf.Clamp(brightnessAmount, 0.0f, 2.0f);
  saturationAmount = Mathf.Clamp(saturationAmount, 0.0f, 2.0f);
  contrastAmount = Mathf.Clamp(contrastAmount, 0.0f, 3.0f);
}
```

После того как скрипт и шейдер будут готовы, нам нужно просто повесить скрипт на камеру и назначить ему шейдер. Теперь мы можем контролировать цвет всей картинки с помощью слайдеров. На следующем скриншоте приводится пример использования наших постэффектов.

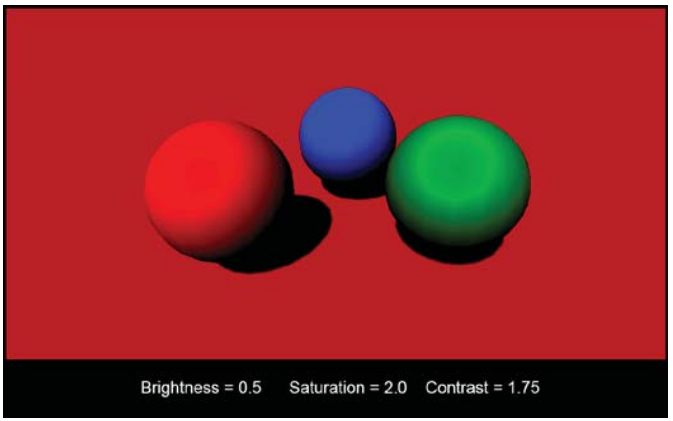

Рис. 10.6. Результат применения постэффекта, корректирующего яркость, насыщенность и контраст

А на этом скриншоте приводится ещё один пример того, чего можно добиться с помощью настройки цветов отрендеренного изображения.

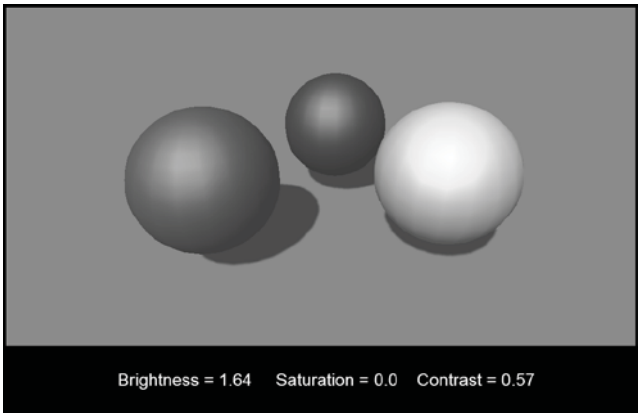

Рис. 10.7. Результат применения постэффекта, корректирующего яркость, насыщенность и контраст с нулевой насыщенностью

Как это работает...

Поскольку мы уже знаем общий принцип работы фулскрин-эффектов, давайте подробно рассмотрим лишь попиксельные операции, которые используются в функции `ContrastSaturationBrightness()`.

Эта функция принимает несколько аргументов. Первый и наиболее важный аргумент, – это цвет пикселя из рендер-текстуры. Остальные аргументы просто управляют эффектами и соответствуют слайдерам в панели **Инспектора**.

В самом начале функции мы объявляем несколько переменных, которые используются в последующем коде при преобразовании цвета. Переменная `LuminanceCoeff` содержит коэффициенты для получения общей яркости изображения. Эти коэффициенты основываются на функциях соответствия цветов CIE, и их значения стандартны в индустрии. Чтобы получить яркость пикселя, нужно вычислить скалярное произведение цвета и этих коэффициентов яркости.

Далее мы умножаем текущий цвет на параметр яркости и используем два раза функцию `lerp`, чтобы, в зависимости от параметров `sat` и `con`, получить нужный по насыщенности и контрастности цвет.

Такие постэффекты, как только что разобранный нами, необходимы для создания качественной графики в играх, поскольку они

позволяют вам корректировать итоговое изображение в игре без необходимости редактировать каждый материал на сцене.

Создание основных режимов блендинга с использованием полноэкранных эффектов

Возможности фулскрин-эффектов не ограничиваются лишь корректировкой цвета изображения с камеры. Мы также можем использовать их для комбинирования других изображений с нашей рендертекстурой. Эта техника похожа на создание нового слоя в Photoshop и выбор ему режима наложения, который управляет блендингом цвета нового изображения со старым, или, как в нашем случае, с рендертекстурой. Это даёт нам очень мощный инструмент, поскольку позволяет художнику имитировать режимы блендинга, применяемые в игре, находясь в редакторе, а не в Photoshop.

В этом рецепте мы рассмотрим наиболее распространённые режимы блендинга, такие как **Multiply** (умножение), **Add** (сложение) и **Overlay** (наложение). Вы увидите, как легко мы можем получить в нашей игре возможности Photoshop по работе с режимами блендинга.

Подготовка

Для начала нам потребуются некоторые ассеты. Поэтому выполните следующие шаги, в которых мы подготовим всё необходимое для реализации эффектов блендинга:

1. Создайте новый скрипт и назовите его `BlendMode_ImageEffect`.
2. Создайте новый шейдер и назовите его `BlendMode_Effect`.
3. Теперь нам нужно скопировать в наш C#-скрипт код из C#-скрипта первого рецепта этой главы. Это позволит нам сфокусироваться непосредственно на шейдере эффекта.
4. Скопируйте в ваш новый шейдер код из того же рецепта.
5. И наконец, нам потребуется ещё одна текстура, которую мы будем накладывать на изображение с камеры с помощью реализованных нами режимов блендинга. В этом рецепте мы будем использовать текстуру грязи. Таким образом, эффекты будут более заметны при тестировании.

На следующем изображении приводится текстура, которую мы использовали. Нам нужна текстура с достаточно большим количеством деталей и большим диапазоном оттенков серого.

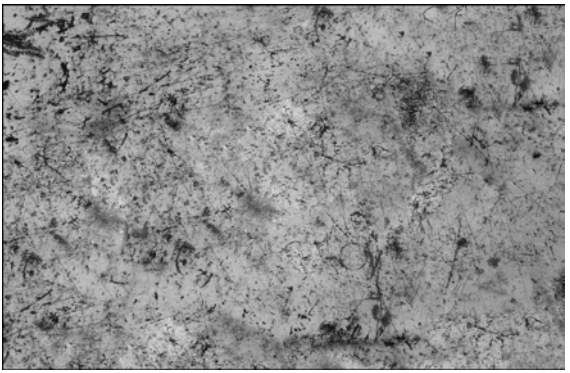

Рис. 10.8. Текстура, использованная нами в этом рецепте

Как это сделать...

Первым мы реализуем режим блендинга **Multiply** в том виде, в каком он используется в Photoshop. Давайте начнём с того, что изменим код нашего шейдера. Откройте шейдер в MonoDevelop, два раза кликнув по нему в панели проекта Unity. Далее выполните следующие действия:

1. Нам понадобятся два новых свойства: текстура, которую мы будем блендить, и слайдер для регулировки прозрачности. Добавьте в ваш шейдер следующий код.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _BlendTex ("Blend Texture", 2D) = "white" {}
  _Opacity ("Blend Opacity", Range(0,1)) = 1
}
```

2. Добавьте соответствующие переменные в блок `CGPROGRAM`, чтобы мы смогли добраться до данных из нашего блока свойств.

```
Pass
{
  CGPROGRAM
  #pragma vertex vert_img
  #pragma fragment frag
  #pragma fragmentoption ARB_precision_hint_fastest
```

```
#include "UnityCG.cginc"

uniform sampler2D _MainTex;
uniform sampler2D _BlendTex;
fixed _Opacity;
```

3. И наконец, нам нужно изменить нашу функцию `frag()` таким образом, чтобы она выполняла умножение наших двух текстур.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
  fixed4 renderTex = tex2D(_MainTex, i.uv);
  fixed4 blendTex = tex2D(_BlendTex, i.uv);

  //Выполним блендинг в режиме Multiply.
  fixed4 blendedMultiply = renderTex * blendTex;

  //Подкорректируем степень блендинга с помощью функции lerp.
  renderTex = lerp(renderTex, blendedMultiply, _Opacity);

  return renderTex;
}
```

4. Сохраните шейдер и вернитесь в редактор Unity, чтобы шейдер скомпилировался. Если не было ошибок, кликните два раза по файлу C#-скрипта, чтобы открыть его в редакторе MonoDevelop.
5. В нашем скрипте нам также потребуется создать соответствующие переменные. Нам будет нужна текстура, которую мы сможем назначить шейдеру, а также слайдер для корректировки итогового значения блендинга.

```
#region Variables
public Shader curShader;
public Texture2D blendTexture;
public float blendOpacity = 1.0f;
private Material curMaterial;
#endregion
```

6. Далее мы передаем значения этих свойств в шейдер в методе `OnRenderImage()`.

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture
destTexture)
{
  if(curShader != null)
```

```
    {
            material.SetTexture("_BlendTex", blendTexture);
            material.SetFloat("_Opacitry", blendOpacity);

            Graphics.Blit(sourceTexture, destTexture, material);
    }
    else
    {
            Graphics.Blit(sourceTexture, destTexture);
    }
}
```

7. И в заключение в методе `Update()` мы нормализуем значение переменной `blendOpacity` в диапазоне от 0,0 до 1,0.

```
void Update()
{
  blendOpacity = Mathf.Clamp(blendOpacity, 0.0f, 1.0f);
}
```

После того как скрипт и шейдер будут готовы, нам нужно просто повесить скрипт на камеру и назначить ему шейдер. Но, чтобы эффект заработал, нам понадобится текстура. В панели **Инспектора** в свойствах скрипта постэффекта вы можете выбрать, какую текстуру использовать. После того как вы назначите текстуру, вы увидите эффект перемножения выбранной текстуры с отрендеренным изображением вашей игры. Следующий скриншот демонстрирует результат применения реализованного нами фулскрин-эффекта.

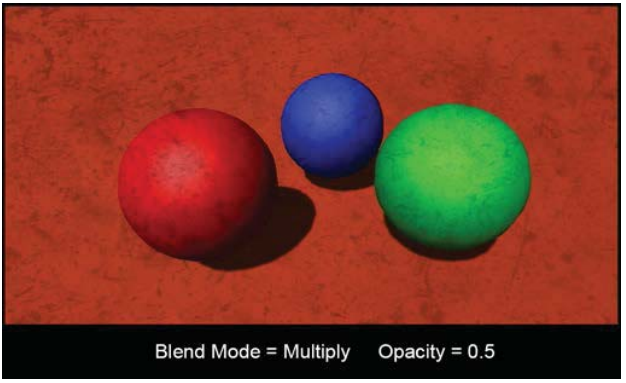

Рис. 10.9. Применение блендинга в режиме Multiply для Opacity 0,5

На следующем скриншоте вы видите пример снижения прозрачности, в результате которого перемножаемое изображение становится гораздо более заметным на фоне отрендеренного изображения.

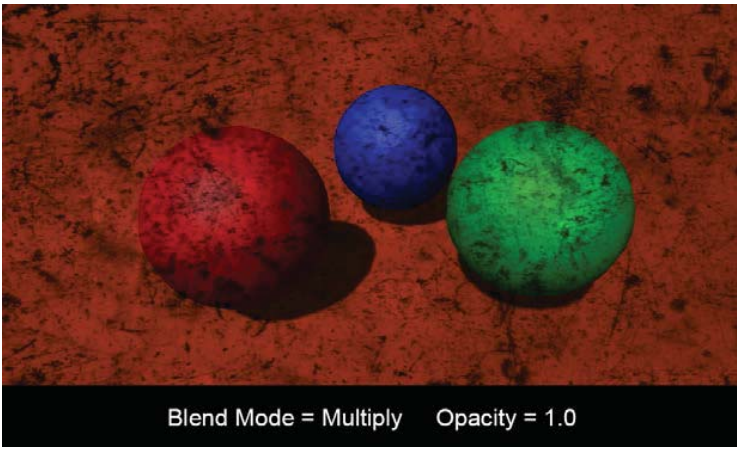

Рис. 10.10. Применение блендинга в режиме Multiply для Opacity 1,0

Теперь, когда мы реализовали наш первый режим блендинга, мы можем добавить к нему еще несколько режимов, чтобы увидеть, как просто это делается и как их использование существенно повышает качество картинки в игре. Но сначала давайте разберёмся с тем, что тут происходит.

Как это работает...

К этому моменту мы уже получили солидный багаж знаний по фулскрин-эффектам, и я уверен, что вы уже видите, какие возможности предоставляет нам Unity. Мы можем буквально повторить для нашей игры функционал Photoshop по работе с блендингом слоёв, чтобы предоставить художнику гибкость контроля, необходимую для достижения высококачественной графики в кратчайшие сроки.

В этом рецепте мы рассмотрели, как перемножать два изображения, используя шейдер. При работе с режимами блендинга необходимо думать на пиксельном уровне. К примеру, если мы используем режим блендинга **Multiply**, мы берём значение каждого пикселя из первоначальной текстуры и перемножаем его со значением пикселя блендинг-текстуры. Аналогичный принцип работает и для режима блендинга **Add**. В этом режиме цвет каждого пикселя исходной текстуры, или рендер-текстуры, складывается с цветом пикселя блендинг-текстуры.

Режим блендинга **Screen** (освещение) немного сложнее, но принцип его работы такой же. Оба изображения, – и рендер-текстура, и

блендинг-текстура, – инвертируются, перемножаются друг с другом, а затем вновь инвертируются. Точно так же, как Photoshop накладывает одни текстуры на другие, используя режимы блендинга, мы можем это делать с помощью фулскрин-эффектов.

Но это ещё не всё...

Давайте продолжим работу с этим рецептом, добавив ещё несколько режимов блендинга в наш фулскрин-эффект. Для этого выполните следующие действия:

1. В шейдере эффекта добавьте следующий код в функцию `frag()`, а также измените значение, возвращаемое нашему скрипту. Кроме этого, нам нужно будет закомментировать режим блендинга Multiply, чтобы он не выполнялся.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
  fixed4 renderTex = tex2D(_MainTex, i.uv);
  fixed4 blendTex = tex2D(_BlendTex, i.uv);

  //Выполним блендинг в режиме Multiply.
  //fixed4 blendedMultiply = renderTex * blendTex;
  fixed4 blendedAdd = renderTex + blendTex;

  //Подкорректируем степень блендинга с помощью функции lerp.
  renderTex = lerp(renderTex, blendedAdd, _Opacity);

  return renderTex;
}
```

   Сохраните шейдер и вернитесь в редактор Unity, чтобы шейдер скомпилировался. Если не было допущено ошибок, то вы увидите результат, похожий на скриншот, показанный на рис. 10.11. Это простой режим блендинга Add.

   Как видите, эффект противоположен режиму Multiply, поскольку теперь мы складываем два изображения.

2. И наконец, давайте добавим ещё один режим блендинга, который называется **Screen** (освещение). Этот режим чуть более сложный, но всё равно прост в реализации. Добавьте следующий код в функцию `frag()` вашего шейдера:

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
```

```
    //используя UV-координаты из структуры v2f_img.
    fixed4 renderTex = tex2D(_MainTex, i.uv);
    fixed4 blendTex = tex2D(_BlendTex, i.uv);

    //Выполним блендинг в режиме Multiply.
    //fixed4 blendedMultiply = renderTex * blendTex;
    //fixed4 blendMultiply = renderTex + blendTex;
    fixed4 blendedScreen = (1.0 - ((1.0 - renderTex) * (1.0 - blendTex)));

    //Подкорректируем степень блендинга с помощью функции lerp.
    renderTex = lerp(renderTex, blendedScreen, _Opacity);

    return renderTex;
}
```

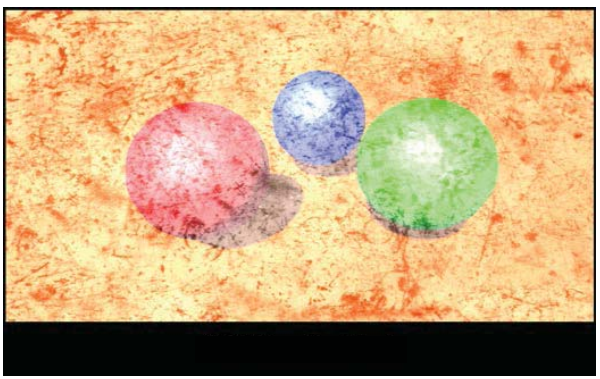

Рис. 10.11. Применение блендинга в режиме Add

Приведённый ниже скриншот демонстрирует результат применения режима блендинга **Screen** при наложении двух изображений.

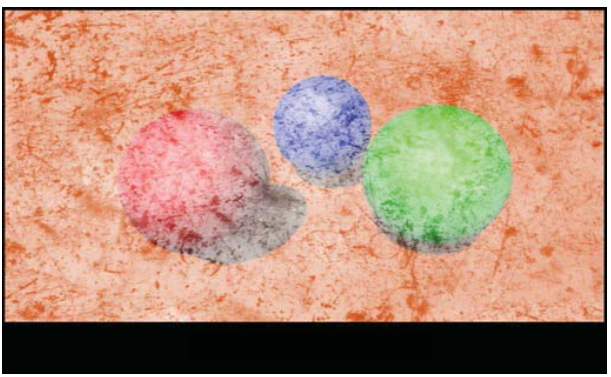

Рис. 10.12. Режим блендинга Screen

Реализация режима блендинга Overlay с использованием полноэкранных эффектов

В нашем последнем рецепте этой главы мы рассмотрим ещё один тип блендинга – **Overlay** (наложение). Этот режим использует некие условия, от которых зависит итоговый цвет каждого пикселя в каждом канале. Так что для его реализации нам понадобится немного больше кода, чем обычно. Давайте посмотрим, как это сделать.

Подготовка

Для этого фулскрин-эффекта нам потребуется подготовить два скрипта наподобие тех, что уже были использованы в предыдущем рецепте этой главы. В этом рецепте мы будем использовать ту же сцену, поэтому новую нам создавать не потребуется. Выполните следующую последовательность действий:

1. Создайте новый скрипт и назовите его `Overlay_ImageEffect`, а также шейдер, который вам следует назвать `Overlay_Effect`.
2. Скопируйте код в файл вашего скрипта из предыдущего C#-скрипта.
3. Скопируйте в файл нового шейдера код из вашего предыдущего шейдера.
4. Назначьте скрипт `Overlay_ImageEffect` основной камере, а шейдер `Overlay_Effect` установите свойству скрипта в **Инспекторе**.
5. После этого два раза кликните по скрипту и по шейдеру, чтобы открыть их в редакторе MonoDevelop.

Как это сделать...

Работу над нашим Overlay – эффектом мы начнём так же, как мы начинали работу с большинством рецептов этой главы, сначала мы разберёмся с шейдером, а потом мы отредактируем C#-скрипт, чтобы он передавал в шейдер корректные данные. Для этого выполните следующую последовательность действий:

1. Для начала нам нужно определить необходимые свойства в нашем блоке свойств. В этом рецепте мы будем использовать такие же свойства, как и в последних рецептах этой главы.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _BlendTex ("Blend Texture", 2D) = "white" {}
  _Opacity ("Blend Opacity", Range(0,1)) = 1
}
```

2. После этого нам нужно создать соответствующие переменные в нашей секции CGPROGRAM.

```
Pass
{
  CGPROGRAM
  #pragma vertex vert_img
  #pragma fragment frag
  #pragma fragmentoption ARB_precision_hint_fastest
  #include "UnityCG.cginc"

  uniform sampler2D _MainTex;
  uniform sampler2D _BlendTex;
  fixed _Opacity;
```

3. Для реализации режима блендинга Overlay нам нужно будет обрабатывать каждый канал цвета пикселя отдельно. Для этого нам потребуется написать в шейдере функцию, которая будет работать со значением отдельного канала, например с каналом красного цвета, и выполнять для него операцию наложения. Добавьте следующий код в ваш шейдер сразу после объявления переменных.

```
fixed OverlayBlendMode(fixed basePixel, fixed blendPixel)
{
  if(basePixel < 0.5)
  {
          return (2.0 * basePixel * blendPixel);
  }
  else
  {
          return (1.0 - 2.0 * (1.0 - basePixel) * (1.0 - blendPixel));
  }
}
```

4. И наконец, нам нужно дописать нашу функцию `frag()` таким образом, чтобы блендинг выполнялся для каждого канала по отдельности.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
```

```
    fixed4 renderTex = tex2D(_MainTex, i.uv);
    fixed4 blendTex = tex2D(_BlendTex, i.uv);

    fixed4 blendedImage = renderTex;

    blendedImage.r = OverlayBlendMode(renderTex.r, blendTex.r);
    blendedImage.g = OverlayBlendMode(renderTex.g, blendTex.g);
    blendedImage.b = OverlayBlendMode(renderTex.b, blendTex.b);

    //Подкорректируем интенсивность блендинга с помощью функции lerp
    renderTex = lerp(renderTex, blendedImage, _Opacity);

    return renderTex;
  }
```

5. После того как вы закончите шейдер, постэффект должен заработать. Сохраните шейдер и вернитесь в Unity, чтобы он скомпилировался. Наш скрипт уже готов, поэтому нам не потребуется вносить в него какие-либо дополнительные изменения. После того как шейдер скомпилируется, вы должны увидеть результат, похожий на следующий скриншот.

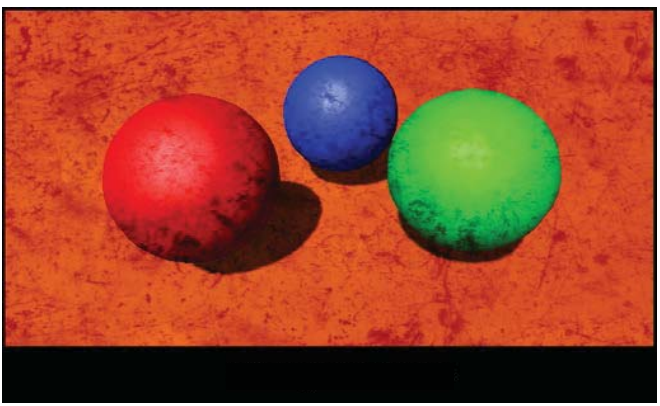

Рис. 10.13. Применение блендинга в режиме наложения

Как это работает...

Реализация режима блендинга **Overlay** определённо сложнее предыдущих, но если вы декомпозируете её на части, то вы увидите, что по существу этот режим является просто комбинацией режима **Multiply** и режима **Screen**. Разница лишь в том, что в данном случае мы выполняем проверку, на основании которой принимаем решение о том, какой режим блендинга использовать для конкретного пикселя.

В функции `OverlayBlendMode` мы проверяем, меньше ли значение канала пикселя, чем 0,5. Если значение оказалось меньше, то мы применяем модифицированный режим блендинга Multiply, если нет, то применяем модифицированный режим блендинга Screen. Мы проделываем это с каждым пикселем, в результате чего получаем итоговое RGB-изображение.

Как видите, с помощью полноэкранных эффектов можно добиться довольно многого. Ограничивающим фактором является лишь целевая платформа и количество памяти, доступной для эффектов. Как правило, эти ограничения определяются в самом начале работы над игровым проектом, поэтому дерзайте и получайте удовольствие от вашей работы с фулскрин-эффектами.

ГЛАВА 11
Гейм-плей и экранные эффекты

В этой главе мы рассмотрим следующие темы:

- создание эффекта старого фильма;
- создание эффекта ночного видения.

Введение

Одной из особенностей real-time-игр является вовлечение игрока в игровой мир, настолько сильное, что игрок воспринимает его как настоящий. Многие современные игры для достижения этой цели активно используют постэффекты.

Например, с помощью постэффектов мы можем превратить игровую атмосферу из спокойной в жуткую, всего лишь откорректировав цвета изображения на экране. Или представьте игровую ситуацию, в которой вы входите в комнату на одном из уровней вашей игры, и игра переходит в видеоролик. Во многих играх этот переход будет реализован с помощью полноэкранного эффекта. Так что следующей нашей задачей будет научиться делать эффекты, запускаемые по событиям гейм-плея.

В этой главе мы узнаем, как можно придать игре вид старого фильма и как во многих шутерах от первого лица реализуется эффект ночного видения. В каждом из этих рецептов мы рассмотрим, как эти эффекты можно связать с гейм-плеем, чтобы они использовались уместно по отношению к игровым событиям.

Создание эффекта старого фильма

Разные игры делаются в разных исторических сеттингах. В одних играх события происходят в фэнтезийных мирах, или в фантастических

мирах далёкого будущего, в других действие происходит на Западе прошлого, как раз тогда, когда были изобретены первые камеры, а кино люди смотрели черно-белое или с так называемым эффектом сепии. Одним словом, – видео того времени имело весьма узнаваемые черты, и их-то мы и воссоздадим с помощью постэффектов в Unity.

Для достижения этого эффекта нам потребуется выполнить несколько действий, и мы рассмотрим, из каких составных частей, помимо очевидного перевода изображения в оттенки серого, складывается этот эффект. Мы сможем это сделать, если проанализируем внешний вид кадров старых фильмов. Давайте посмотрим на следующее изображение и разберёмся, какие у них есть характерные черты.

Рис. 11.1. Кадр из старого фильма

Мы сделали эту картинку, оглядываясь на несколько примеров в Сети. Для создания таких картинок, с помощью которых мы можем лучше понять, как делать тот или иной эффект, удобно использовать Photoshop. Работая с Photoshop, мы понимаем, не только какие элементы нам потребуется реализовать программно, но и получаем представление о том, какие режимы блендинга будут уместны для решения поставленной задачи и как нам нужно будет организовать слои наших эффектов. Исходник Photoshop, созданный нами для этого рецепта под названием `OldFilmEffect_Research_Layout.psd`, доступен на странице, посвящённой данной книге, по адресу: www.packtpub.com/support.

Подготовка

Теперь, когда мы знаем, что у нас должно получиться, давайте обратим наше внимание на то, как сочетаются используемые слои для создания итогового эффекта, а также подготовим необходимые ресурсы.

- **Тонирование в сепию:** этот эффект относительно прост в реализации, поскольку нам потребуется всего лишь привести все пиксели исходной рендер-текстуры к единому цветовому диапазону. Этого легко можно добиться, если использовать освещённость исходного изображения и добавить к ней некий цвет.

  Наш первый слой будет выглядеть следующим образом:

Рис. 11.2. Изображение, тонированное в сепию

- **Эффект виньетки:** на старых фильмах, когда они показываются с помощью старых кинопроекторов, всегда можно заметить нечто вроде «мягких», или нечётких, границ кадра. Это происходит потому, что лампочка, используемая в кинопроекторе, более ярка в центре, чем по краям. Этот эффект называется эффектом виньетки, и именно его мы будем использовать в качестве второго слоя. Добиться этого эффекта можно, наложив полупрозрачную текстуру на весь экран. На следующем изображении приводится этот слой в виде отдельной текстуры;

Рис. 11.3. Слой виньетки

- **Пыль и царапины:** третьим и последним слоем нашего эффекта старого фильма будет слой с пылью и царапинами. В этом слое мы будем использовать две текстуры – одну для пыли и одну для царапин. Две отдельные текстуры нам нужны для того, чтобы анимировать их с разной скоростью. Так мы сможем создать эффект прокрутки киноплёнки, на каждом кадре которой содержатся царапинки и пыль. Следующее изображение демонстрирует текстуру этого эффекта.

Рис. 11.4. Пыль и царапины, использованные в данном рецепте

Теперь давайте подготовим шейдер и скрипт, которые используют эти текстуры. Выполните следующие действия:

1. Для данного рецепта вам потребуются текстура для виньетки, а также текстуры для пыли и царапин – наподобие только что приведённых выше.

2. Создайте новый скрипт под названием `OldFilmEffect.cs` и новый шейдер под названием `OldFilmEffectShader.shader`.
3. После того как вы создадите необходимые файлы, вам нужно будет скопировать в них базовый код эффекта из предыдущей главы. О том, как это сделать, вы можете узнать в главе 10 «Полноэкранные эффекты с использованием рендер-текстур».

Теперь мы можем перейти непосредственно к созданию эффекта старого фильма.

Как это сделать...

По отдельности наши слои, используемые для эффекта старого фильма, довольно просты, но при их сочетании мы получаем интересный визуальный эффект. Давайте пройдёмся по коду скрипта и шейдера, а после этого разберём каждую строчку этого кода и посмотрим, почему код работает именно так, как он работает. К этому моменту у вас уже должен быть код простого фулскрин-эффекта, так что в этом рецепте мы пропустим этот шаг и перейдём сразу к добавлению нового кода. Выполните следующие действия:

1. Начнём мы с C#-скрипта. В первом блоке кода мы определим переменные, которые должны быть видимы в **Инспекторе**, чтобы пользователь мог изменять параметры нашего эффекта. Чтобы понять, какие параметры нам нужны, мы можем руководствоваться нашим Photoshop-файлом, в котором мы сделали набросок эффекта. Добавьте следующий код.

```
#region Variables
public Shader oldFilmShader;

public float OldFilmEffectAmount = 1.0f;

public Color sepiaColor = Color.white;
public Texture2D vignetteTexture;
public float vignetteAmount = 1.0f;

public Texture2D scratchesTexture;
public float scratchesYSpeed = 10.0f;
public float scratchesXSpeed = 10.0f;

public Texture2D dustTexture;
public float dustYSpeed = 10.0f;
public float dustXSpeed = 10.0f;

private Material curMaterial;
```

```
private float randomValue;
#endregion
```

2. Далее нам нужно написать метод `OnRenderImage()`. В нём мы будем передавать значения наших переменных шейдеру, чтобы он смог использовать их при обработке рендер-текстуры.

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture
destTexture)
{
  if(oldFilmShader != null)
  {
          material.SetColor("_SepiaColor", sepiaColor);
          material.SetFloat("_VignetteAmount", vignetteAmount);
          material.SetFloat("_EffectAmount", OldFilmEffectAmount);

          if(vignetteTexture)
          {
               material.SetTexture("_VignetteTex", vignetteTexture);
          }

          if(scratchesTexture)
          {
               material.SetTexture("_ScratchesTex", scratchesTexture);
               material.SetFloat("_ScratchesYSpeed", scratchesYSpeed);
               material.SetFloat("_ScratchesXSpeed", scratchesXSpeed);
          }

          if(dustTexture)
          {
               material.SetTexture("_DustTex", dustTexture);
               material.SetFloat("_dustYSpeed", dustYSpeed);
               material.SetFloat("_dustXSpeed", dustXSpeed);
               material.SetFloat("_RandomValue", RandomValue);
          }

          Graphics.Blit(sourceTexture, destTexture, material);
  }
  else
  {
          Graphics.Blit(sourceTexture, destTexture);
  }
}
```

3. И наконец, мы ограничиваем параметры скрипта, чтобы их значения лежали в правильных диапазонах.

```
void Update()
{
  vignetteAmount = Mathf.Clamp01(vignetteAmount);
  OldFilmEffectAmount = Mathf.Clamp(OldFilmEffectAmount, 0f, 1.5f);
```

```
  randomValue = Random.Range(-1f, 1f);
}
```

4. Завершив написание C#-скрипта, давайте переключимся на шейдер. Мы должны добавить в шейдер переменные, которые соответствуют созданным ранее переменным в скрипте. После этого скрипт и шейдер смогут взаимодействовать. Добавьте следующий код в блок свойств вашего шейдера.

```
Properties
{
  _MaintTex ("Base (RGB)", 2D) = "white" {}
  _VignetteTex ("Vignette Texture", 2D) = "white"{}
  _ScratchesTex ("Scratches Texture", 2D) = "white"{}
  _DustTex ("Dust Texture", 2D) = "white"{}
  _SepiaColor ("Sepia Color", Color) = (1,1,1,1)
  _EffectAmount ("Old Film Effect Amount", Range(0,1)) = 1.0
  _VignetteAmount ("Vignette Opacity", Range(0,1)) = 1.0
  _ScratchesYSpeed ("Scratches Y Speed", Float) = 10.0
  _ScratchesXSpeed ("Scratches X Speed", Float) = 10.0
  _dustXSpeed ("Dust X Speed", Float) = 10.0
  _dustYSpeed ("Dust Y Speed", Float) = 10.0
  _RandomValue ("Random Value", Float) = 1.0
}
```

5. Далее, как обычно, нам нужно добавить эти же переменные в блок CGPROGRAM.

```
SubShader
{
  Pass
  {
          CGPROGRAM
          #pragma vertex vert_img
          #pragma fragment frag
          #pragma fragmentoption ARB_precision_hint_fastest
          #include "UnityCG.cginc"

          uniform sampler2D _MainTex;
          uniform sampler2D _VignetteTex;
          uniform sampler2D _ScratchesTex;
          uniform sampler2D _DustTex;
          fixed4 _SepiaColor;
          fixed _VignetteAmount;
          fixed _ScratchesYSpeed;
          fixed _ScratchesXSpeed;
          fixed _dustXSpeed;
          fixed _dustYSpeed;
          fixed _EffectAmount;
          fixed _RandomValue;
```

6. Теперь мы переходим к нашей функции `frag()`, в которой мы будем попиксельно применять эффект к рендер-текстуре. Начинаем мы с того, что получаем цвет из рендер-текстуры и из текстуры с виньеткой.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
  half2 renderTexUV = half2(i.uv.x, i.uv.y + (_RandomValue *
_SinTime.z * 0.005));
  fixed4 renderTex = tex2D(_MainTex, renderTexUV);

  //Получим цвет из текстуры виньетки.
  fixed4 vignetteTex = tex2D(_VignetteTex, i.uv);
```

7. После этого мы добавляем пыль и царапины.

```
//Возьмём цвет царапин.
half2 scratchesUV = half2(i.uv.x + (_RandomValue * _SinTime.z *
_ScratchesXSpeed), i.uv.y + (_Time.x * _ScratchesYSpeed));
fixed4 scratchesTex = tex2D(_ScratchesTex, scratchesUV);

//Возьмём цвет пыли.
half2 dustUV = half2(i.uv.x + (_RandomValue * (_SinTime.
z * _dustXSpeed)), i.uv.y + (_RandomValue * (_SinTime.z *
_distYSpeed)));
fixed4 dustTex = tex2D(_DustTex, dustUV);
```

8. Далее переводим в сепию.

```
//Получим значения освещённости из рендер-текстуры с помощью
//значений YIQ.
fixed lum = dot(fixed3(0.299, 0.587, 0.114), renderTex.rgb);

//Добавим постоянный цвет значению lum.
fixed4 finalColor = lum + lerp(_SepiaColor, _SepiaColor +
                fixed4(0.1f,0.1f,0.1f,0.1f), _RandomValue);
```

9. И наконец, мы объединяем все наши слои и цвета, чтобы получить текстуру итогового эффекта.

```
  //Создадим постоянный белый цвет, который мы сможем
  //использовать для корректировки прозрачности эффектов.
  fixed3 constantWhite = fixed3(1,1,1);

  //Объединим вместе различные слои, чтобы создать итоговый
  //эффект.
  finalColor = lerp(finalColor, finalColor * vignetteTex,
_VignetteAmount);
  finalColor.rgb *= lerp(scratchesTex, constantWhite,
```

```
_RandomValue);
  finalColor.rgb *= lerp(dustTex.rgb, constantWhite,
(_RandomValue * _SinTime.z));
  finalColor = lerp(renderTex, finalColor, _EffectAmount);

  return finalColor;
}

ENDCG
```

10. После того как вы напишете весь этот код, если не было допущено ошибок, вы увидите нечто, похожее на следующий скриншот. В редакторе Unity нажмите **Play**, чтобы увидеть эффект анимации царапин и пыли, а также медленное смещение изображения.

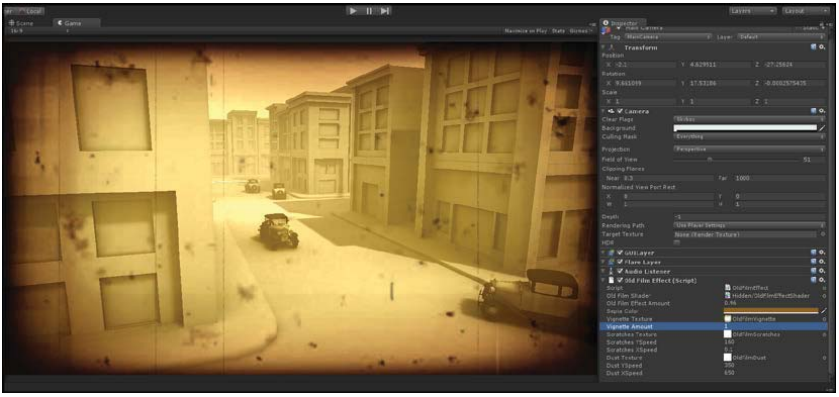

Рис. 11.5. Результат применения постэффекта старого фильма

Как это работает...

Теперь давайте пройдёмся по каждому из слоёв, задействованных в данном эффекте, и рассмотрим каждую строчку кода, чтобы разобраться, как этот код работает. Таким образом, мы поймём, как мы можем улучшить этот эффект.

Теперь, когда наш эффект старого фильма работает, давайте рассмотрим код в функции `frag()`, поскольку весь остальной код к этому моменту уже должен быть понятен.

По аналогии с Photoshop, наш шейдер обрабатывает каждый слой по отдельности, а потом блендит их вместе. Так что на этот процесс удобно смотреть, как на слои в Photoshop. Подобный подход помогает при разработке новых фулскрин-эффектов.

Обратим внимание на первые строки кода функции `frag()`.

```
fixed4 frag(v2f_img i) : COLOR
{
     //Получим цвет из рендер-текстуры,
     //используя UV-координаты из структуры v2f_img.
     half2 renderTexUV = half2(i.uv.x, i.uv.y + (_RandomValue *
_SinTime.z * 0.005));
     fixed4 renderTex = tex2D(_MainTex, renderTexUV);
     fixed4 vignetteTex = tex2D(_VignetteTex, i.uv);
```

В первой строке кода функции мы определяем UV-координаты для главной текстуры. Так как мы пытаемся имитировать эффект старого фильма, нам нужно немного изменять UV-координаты нашей рендер-текстуры на каждом фрейме, будто кадры киноленты слегка дрожат. Дрожание соответствует неидеальной работе механизма перемотки киноленты. Мы используем встроенную переменную `_SinTime`, которая возвращает значение между –1 и 1. Далее мы умножаем её на очень маленькое число, в нашем случае на 0,005, чтобы снизить интенсивность этого эффекта. Получившееся значение вновь умножается на значение переменной `_RandomValue`, которое мы генерируем в скрипте. Это значение «скачет» между –1 и 1, чтобы имитировать «скачки» киноплёнки вперёд-назад. Получив новое значение UV-координат, мы сохраняем его в переменную `renderTexUV` и, используя функцию `tex2D()`, получаем цвет рендер-текстуры в обрабатываемом пикселе. Далее мы просто получаем цвет из текстуры виньетки с помощью той же функции `tex2D()`. Мы не используем на этом шаге анимированные UV-координаты, поскольку текстура виньетки будет привязана к камере, а не к подёргивающейся киноленте.

Следующий фрагмент кода иллюстрирует вторую группу строк в нашей функции `frag()`.

```
//Возьмём цвет царапин.
half2 scratchesUV = half2(i.uv.x + (_RandomValue * _SinTime.z *
_ScratchesXSpeed), i.uv.y + (_Time.x * _ScratchesYSpeed));
fixed4 scratchesTex = tex2D(_ScratchesTex, scratchesUV);

//Возьмём цвет пыли.
half2 dustUV = half2(i.uv.x + (_RandomValue * (_SinTime.z *
_dustXSpeed)), i.uv.y + (_RandomValue * (_SinTime.z *
_distYSpeed)));
fixed4 dustTex = tex2D(_DustTex, dustUV);
```

Этот код очень похож на предыдущий фрагмент, где мы создавали анимированные UV-координаты, чтобы модифицировать положение

слоя эффекта. Мы просто используем значение встроенной в Unity переменной `_SinTime`, чтобы получить значение между –1 и 1, умножаем его на нашу случайную величину, а потом на ещё один множитель, который корректирует общую скорость анимации. Рассчитанные UV-координаты используются для сэмплинга текстур пыли и царапин.

Следующий фрагмент кода отвечает за цвет нашего эффекта старого фильма.

```
//Получим значения освещённости из рендер-текстуры с помощью
//значений YIQ.
fixed lum = dot(fixed3(0.299, 0.587, 0.114), renderTex.rgb);

//Добавим постоянный цвет значению lum.
fixed4 finalColor = lum + lerp(_SepiaColor, _SepiaColor +
                               fixed4(0.1f,0.1f,0.1f,0.1f),
                               _RandomValue);
```

В этом блоке кода мы осуществляем цветовое тонирование всей нашей рендер-текстуры. Для этого мы сначала преобразовываем её в черно-белое изображение. Делаем мы это, используя значения освещённости, переводя цвет в формат YIQ. Значения YIQ, – это цветовое пространство, используемое телевизионной системой NTSC. Чтобы подробнее прочитать про YIQ, вы можете посетить следующие ссылки:

- http://en.wikipedia.org/wiki/YIQ;
- http://www.blackice.com/colorspaceYIQ.htm;
- http://dcssrv1.oit.uci.edu/~wiedeman/cspace/me/infoyiq.html.

Для работы с этим рецептом вы не обязаны знать, что такое YIQ и как в этом пространстве представляются цвета. Важно лишь запомнить, что значение Y соответствует освещенности картинки и из RGB цвета оно получается по специальной формуле с коэффициентами для каждого из каналов. Таким образом, чтобы получить черно-белое изображение, нужно для каждого пикселя вычислить скалярное произведение его цвета с этими коэффициентами, что даст нам значение освещенности в этом пикселе. Это мы и делаем в первой строке приведённого кода.

После того как мы получим значения освещённости, мы сможем просто наложить на него цвет, в который мы хотим тонировать наше изображение. Этот цвет передаётся в шейдер из нашего C#-скрипта, а потом в блок `CGPROGRAM`, где мы можем сложить его с нашей черно-белой текстурой. Таким образом, мы получаем готовое тонированное изображение.

И наконец, мы совмещаем все слои нашего эффекта. Что и делается в следующем коде:

```
//Создадим постоянный белый цвет, который мы сможем
//использовать для корректировки прозрачности эффектов.
     fixed3 constantWhite = fixed3(1,1,1);

//Объединим вместе различные слои, чтобы создать итоговый
//эффект.
     finalColor = lerp(finalColor, finalColor * vignetteTex,
_VignetteAmount);
     finalColor.rgb *= lerp(scratchesTex, constantWhite,
_RandomValue);
     finalColor.rgb *= lerp(dustTex.rgb, constantWhite,
(_RandomValue * _SinTime.z));
     finalColor = lerp(renderTex, finalColor, _EffectAmount);

     return finalColor;
}
```

Последний блок кода достаточно прост и не нуждается в детальном объяснении. Вкратце мы просто умножаем значения всех слоев и получаем итоговую картинку. Так же как в Photoshop, цвета слоев умножаются в режиме наложения Multiply, мы делаем это в коде шейдера. Для того чтобы иметь возможность корректировать степень прозрачности каждого слоя, мы используем функцию `lerp()` с соответствующими параметрами, благодаря чему мы получаем больше художественного контроля за нашим эффектом. Чем больше «ручек для настроек» эффектов можно предоставить художнику, тем лучше.

Создание эффекта ночного видения

Следующий фулскрин-эффект, который мы сделаем, используется определённо чаще. Эффект ночного видения присутствует в таких играх, как Call of Duty Modern Warfare, Halo, а также практически в любом шутере от первого лица, имеющемся сегодня на рынке. Эффект заключается в подсвечивании всего изображения с помощью хорошо узнаваемого лаймово-зелёного цвета.

Прежде чем приступить к созданию эффекта ночного видения, нам нужно воссоздать его в Photoshop, как мы делали в прошлом примере. Для этого нам нужно найти несколько референсных изображений

в Интернете и нарисовать похожую картинку в Photoshop, чтобы посмотреть, какие слои и режимы блендинга нам нужны для получения данного эффекта. На следующем скриншоте изображён результат этого процесса.

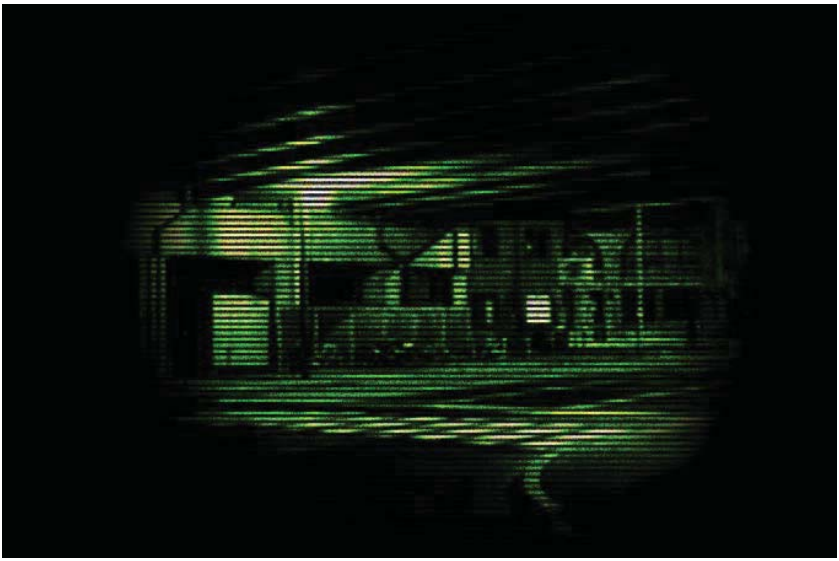

Рис. 11.6. Изображение из прибора ночного видения

А теперь давайте перейдём к разбиению нашего изображения, сделанного в Photoshop, на его составные части, чтобы мы смогли лучше понять, какие нам потребуется подготовить ресурсы. В следующей секции мы рассмотрим как раз этот процесс.

Подготовка

Давайте опять начнём работу над нашим постэффектом с того, что выделим в нём слои. Используя Photoshop, мы можем создать многослойное изображение, чтобы лучше продемонстрировать, как именно мы можем запечатлеть эффект ночного видения.

- **Тонирование в зелёный цвет:** нашим первым слоем будет классический зелёный цвет, который мы ожидаем увидеть на любом изображении, полученном в режиме ночного видения. С помощью этого эффекта мы сможем придать нашим изображениям узнаваемые черты ночного видения.

Рис. 11.7. Тонирование изображения в зелёный цвет

- **Линии сканирования:** чтобы у пользователя создалось и усилилось ощущение, что перед ним теперь находится какой-то другой дисплей, мы добавим поверх нашего тонирования в зелёный цвет линии сканирования (scan lines). Для этих целей мы воспользуемся текстурой, созданной в Photoshop, и позволим пользователю самостоятельно выбирать плотность, чтобы линии сканирования становились больше или меньше.

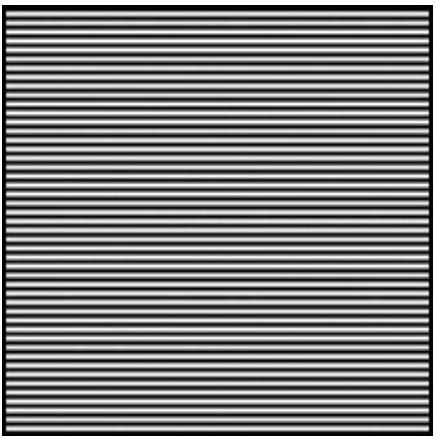

Рис. 11.8. Линии сканирования, используемые в данном рецепте

- **Шумы:** наш следующий слой, – это шумы, или помехи, которые мы наложим поверх тонированного изображения и линий сканирования, чтобы работа оборудования ночного видения не казалась идеальной, а заодно чтобы добавить эффекту деталей. Этот слой усиливает ощущение работы с техническим устройством ночного видения.

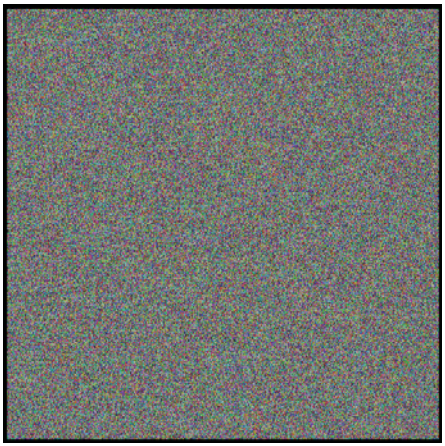

Рис. 11.9. Шумы, используемые в данном рецепте

- **Виньетка:** последним слоем в нашем эффекте ночного видения будет виньетка. Если вы посмотрите, как реализован эффект ночного видения в Call of Duty Modern Warfare, то вы увидите, что в этой игре используется виньетка, имитирующая эффект взгляда через окуляр. Мы тоже воспользуемся этим приёмом.

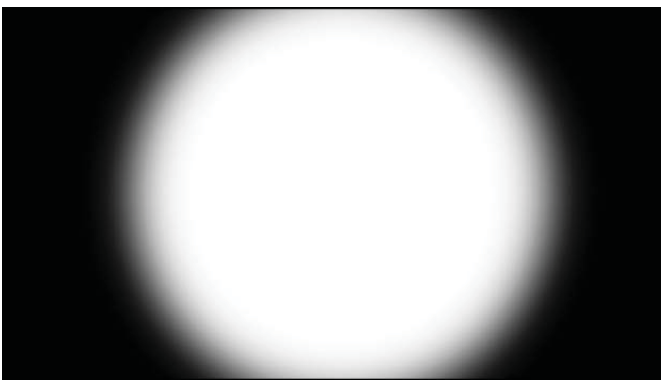

Рис. 11.10. Виньетка, используемая в данном рецепте

Давайте подготовим текстуры, которые нам понадобятся для создания этого постэффекта. Выполните следующие действия:

1. Подготовьте текстуру виньетки, текстуру шума и линий сканирования – наподобие тех, что мы приводили чуть выше.
2. Создайте новый скрипт под названием `NightVisionEffect.cs` и новый шейдер под названием `NightVisionEffectShader.shader`.
3. После того как вы создадите необходимые файлы, вам нужно будет скопировать в них базовый код эффекта из предыдущей главы. О том, как это сделать, вы можете узнать в главе 10 «Полноэкранные эффекты с использованием рендер-текстур».

А теперь, когда мы подготовили необходимые файлы текстур и создали базовые скрипты, мы можем приступить непосредственно к созданию эффекта ночного видения.

Как это сделать...

Мы начнём разработку эффекта со скрипта `NightVisionEffect.cs`. Кликните по нему два раза, чтобы открыть его в MonoDevelop. После этого выполните следующие действия:

1. Нам потребуется создать несколько переменных, чтобы дать пользователю нашего эффекта возможность настраивать его в инспекторе. Добавьте следующий код в скрипт `NightVisionEffect.cs`.

```
#region Variables
public Shader nightVisionShader;

public float contrast = 2.0f;
public float brightness = 1.0f;
public Color nightVisionColor = Color.white;

public Texture2D vignetteTexture;

public Texture2D scanLineTexture;
public float scanLineTileAmount = 4.0f;

public Texture2D nightVisionNoise;
public float noiseXSpeed = 100.0f;
public float noiseYSpeed = 100.0f;

public float distortion = 0.2f;
```

```
public float scale = 0.8f;

private float randomValue = 0.0f;
private Material curMaterial;
#endregion
```

2. Далее нам нужно написать метод `OnRenderImage()`. В нём мы будем передавать значения наших переменных шейдеру, чтобы он смог использовать их при обработке рендер-текстуры.

```
void OnRenderImage(RenderTexture sourceTexture,
RenderTexture destTexture)
{
  if (nightVisionShader != null)
  {
          material.SetFloat("_Contrast", contrast);
          material.SetFloat("_Brightness", brightness);
          material.SetFloat("_NightVisionColor", nightVisionColor);
          material.SetFloat("_RandomValue", randomValue);
          material.SetFloat("_distortion", distortion);
          material.SetFloat("_scale",scale);

          if (vignetteTexture)
          {
                material.SetTexture("_VignetteTex", vignetteTexture);
          }

          if (scanLineTexture)
          {
                material.SetTexture("_ScanLineTex", scanLineTexture);
                material.SetFloat("_ScanLineTileAmount",
                                  scanLineTileAmount);
          }

          if (nightVisionNoise)
          {
                material.SetTexture("_NoiseTex", nightVisionNoise);
                 material.SetFloat("_NoiseXSpeed", noiseXSpeed);
                 material.SetFloat("_NoiseYSpeed", noiseYSpeed);
          }

          Graphics.Blit(sourceTexture, destTexture, material);
  }
  else
  {
          Graphics.Blit(sourceTexture, destTexture);
  }
}
```

3. Чтобы завершить работу со скриптом `NightVisionEffect.cs`, давайте убедимся, что мы ограничиваем значения наших переменных в некотором диапазоне. Приведённые здесь диапазоны произвольны и могут быть в дальнейшем изменены. Мы же используем значения, которые действительно хорошо работают.

```
void Update()
{
  contrast = Mathf.Clamp(contrast, 0f, 4f);
  brightness = Mathf.Clamp(brightness, 0f, 2f);
  randomValue = Random.Range(-1f, 1f);
  distortion = Mathf.Clamp(distortion, -1f, 1f);
  scale = Mathf.Clamp(scale, 0f, 3f);
}
```

4. Теперь мы можем сосредоточить наше внимание на шейдерной части нашего эффекта. Если вы ещё не открыли шейдер, то сейчас самое время это сделать. Начните с добавления свойств.

```
Properties
{
  _MainTex ("Base (RGB)", 2D) = "white" {}
  _VignetteTex ("Vignette Texture", 2D) = "white" {}
  _ScanLineTex ("Scan Line Texture", 2D) = "white" {}
  _NoiseTex ("Noise Texture", 2D) = "white" {}
  _NoiseXSpeed ("Noise X Speed", Float) = 100.0
  _NoiseYSpeed ("Noise Y Speed", Float) = 100.0
  _ScanLineTileAmount ("Scan Line Tile Amount", Float) = 4.0
  _NightVisionColor ("Night Vision Color", Color) = (1,1,1,1)
  _Contrast ("Contrast", Range(0,4)) = 2
  _Brightness ("Brightness", Range(0,2)) = 1
  _RandomValue ("Random Value", Float) = 0
  _distortion ("Distortion", Float) = 0.2
  _scale ("Scale (Zoom)", Float) = 0.8
}
```

5. Для того чтобы мы смогли передавать данные из нашего блока свойств в блок `CGPROGRAM`, нам нужно объявить в нём переменные с такими же именами.

```
SubShader
{
  Pass
  {
          CGPROGRAM
          #pragma vertex vert_img
          #pragma fragment frag
          #pragma fragmentoption ARB_precision_hint_fastest
```

```
#include "UnityCG.cginc"

uniform sampler2D _MainTex;
uniform sampler2D _VignetteTex;
uniform sampler2D _ScanLineTex;
uniform sampler2D _NoiseTex;
fixed4 _NightVisionColor;
fixed _Contrast;
fixed _ScanLineTileAmount;
fixed _Brightness;
fixed _RandomValue;
fixed _NoiseXSpeed;
fixed _NoiseYSpeed;
fixed _distortion;
fixed _scale;
```

6. Мы также добавим в наш эффект искажение линз, чтобы мы могли достовернее воссоздать эффект обзора через оккуляры, где изображение по углам претерпевает искажения. Добавьте в блок `CGPROGRAM` следующий код сразу после объявления переменных.

```
float2 barrelDistortion(float2 coord)
{
  //Алгоритм искажения линз
  //Смотри на http://www.ssontech.com/content/lensalg.htm.

  float2 h = coord.xy - float2(0.5, 0.5);
  float r2 = h.x * h.x + h.y * h.y;
  float r = 1.0 + r2 * (_distortion * sqrt(r2));

  return f * _scale * h + 0.5;
}
```

7. Теперь мы можем сконцентрироваться на основной части нашего шейдера `NightVisionEffect`. Давайте начнём с сэмплинга текстур. В функцию `frag()` нашего шейдера добавьте следующий код.

```
fixed4 frag(v2f_img i) : COLOR
{
  //Получим цвет из рендер-текстуры,
  //используя UV-координаты из структуры v2f_img.
  half2 distortedUV = barrelDistortion(i.uv);
  fixed4 renderTex = tex2D(_MainTex, distortedUV);
  fixed4 vignetteTex = tex2D(_VignetteTex, i.uv);
```

8. Далее мы получаем значения текстур линий сканирования и шумов, используя анимированные UV-координаты.

```
//Обработаем линии сканирования и шумы.
half2 scanLinesUV = half2(i.uv.x * _ScanLineTileAmount,
                          i.uv.y * _ScanLineTileAmount);
fixed4 scanLineTex = tex2D(_ScanLineTex, scanLinesUV);

half2 noiseUV = half2(i.uv.x + (_RandomValue * _SinTime.z *
_NoiseXSpeed), i.uv.y + (_Time.x * _NoiseYSpeed));
fixed4 noiseTex = tex2D(_NoiseTex, noiseUV);
```

9. После этого нам нужно вычислить значение освещённости основной текстуры и применить к ней цвет ночного видения, чтобы привести её к характерному, узнаваемому виду ночного видения.

```
//Получим значения освещённости из рендер-текстуры
//с помощью значений YIQ.
fixed lum = dot(fixed3(0.299, 0.587, 0.114), renderTex.rgb);
lum += _Brightness;
fixed4 finalColor = (lum * 2) + _NightVisionColor;
```

10. И наконец, нам нужно объединить все слои воедино и вернуть итоговый цвет нашего эффекта ночного видения.

```
 //Итоговые значения
 finalColor = pow(finalColor, _Contrast);
 finalColor *= vignetteTex;
 finalColor *= scanLineTex * noiseTex;

 return finalColor;
}
```

Теперь сохраните шейдер и вернитесь в редактор Unity, чтобы ваш шейдер и скрипт скомпилировались. Если не возникло никаких ошибок, для того чтобы увидеть результаты применения эффекта, нажмите в редакторе кнопку **Play**. Вы должны увидеть нечто похожее на изображение, показанное на рис. 11.11.

Как это работает...

Как вы, наверное заметили, эффект ночного видения очень похож на эффект старого фильма, что лишний раз демонстрирует нам перспективы создания модульных компонент в Unity. Мы можем получить абсолютно разные эффекты, используя один и тот же код, но применяя другие текстуры и другие параметры тайлинга.

Единственное отличие данного эффекта состоит в том, что мы добавляем в наш постэффект искажение от линз. Давайте разберёмся, как он работает.

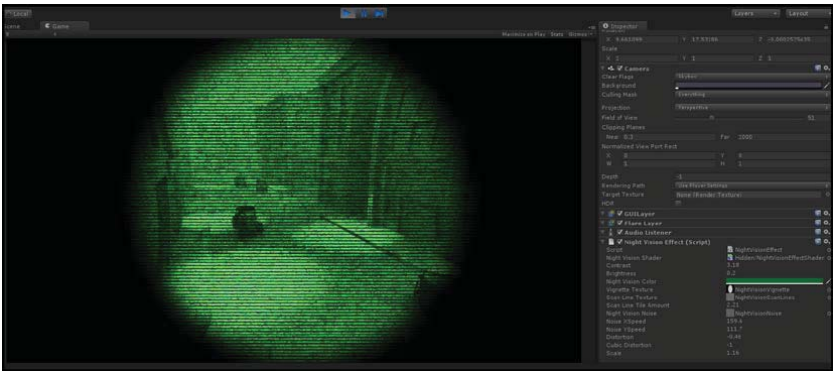

Рис. 11.11. Результат применения эффекта ночного видения

Следующий фрагмент кода используется для реализации искажения линз. Этот фрагмент кода мы получили от создателей SynthEyes, и этот код можно свободно использовать в ваших собственных эффектах.

```
float2 barrelDistortion(float2 coord)
{
     //Алгоритм искажения линз
     //Смотри на http://www.ssontech.com/content/lensalg.htm.

     float2 h = coord.xy - float2(0.5, 0.5);
     float r2 = h.x * h.x + h.y * h.y;
     float r = 1.0 + r2 * (_distortion * sqrt(r2));

     return f * _scale * h + 0.5;
}
```

В функции `barrelDistortion()` в первой строке кода мы ищем центр изображения рендер-текстуры. Когда у нас есть центр изображения, мы можем растягивать пиксели по мере их удаления от него. Таким образом, мы имитируем эффект искажения основной рендер-текстуры выпуклой поверхностью линзы. Этот эффект смотрится очень хорошо, когда он используется в таких постэффектах, как ночное видение.

После того как UV-координаты будут обработаны для имитации эффекта растяжения, мы продолжаем работу с шейдером, так же как мы с ним работали раньше, – применяем анимацию UV-координат и попиксельные операции, в результате чего мы и получаем итоговый эффект ночного видения.

Дополнительная информация

Узнать больше об эффекте искажения линз вы можете по следующему адресу: http://www.ssontech.com/content/lensalg.htm.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ