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

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.

ISBN 978-5-97060-213-3

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

ISBN 978-1-84969-508-4 (англ.)
ISBN 978-5-97060-213-3 (рус.)

ОГЛАВЛЕНИЕ

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.

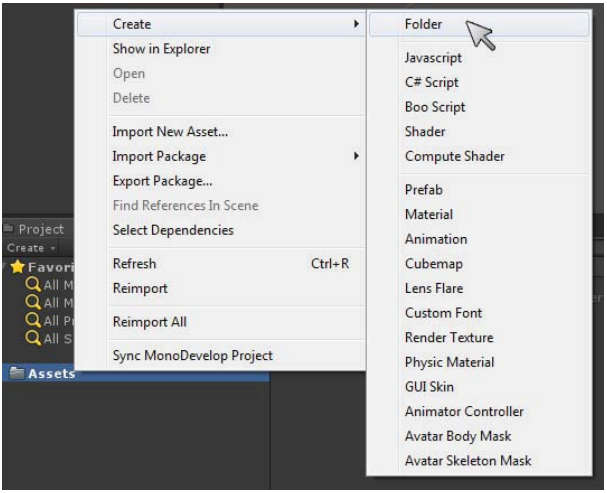

Рис. 1.1. Создание в проекте новой папки

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.*

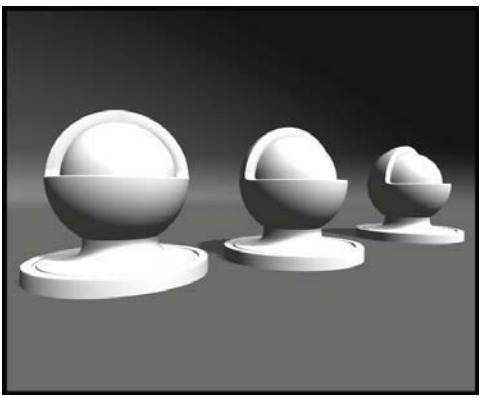

Рис. 1.2. Объекты, используемые в данном рецепте

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:

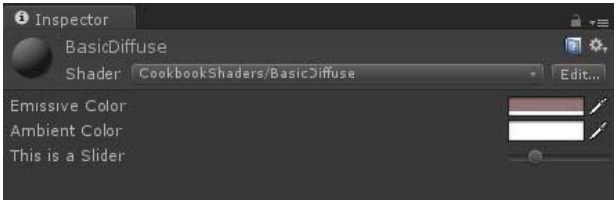

Рис. 1.3. Элементы GUI

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**:

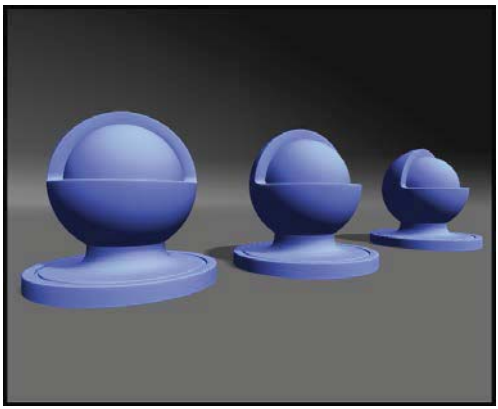

Рис. 1.5. Результат использования свойств

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:

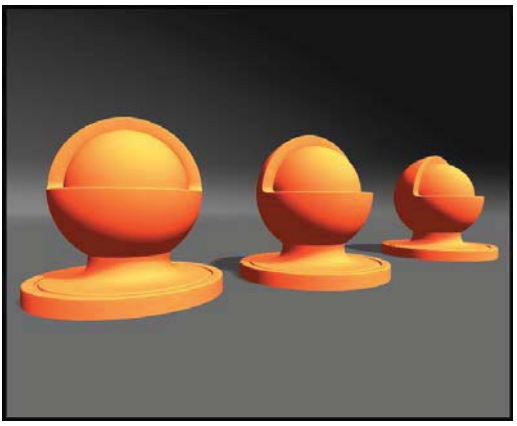

Рис. 1.6. Результат применения диффузного шейдера