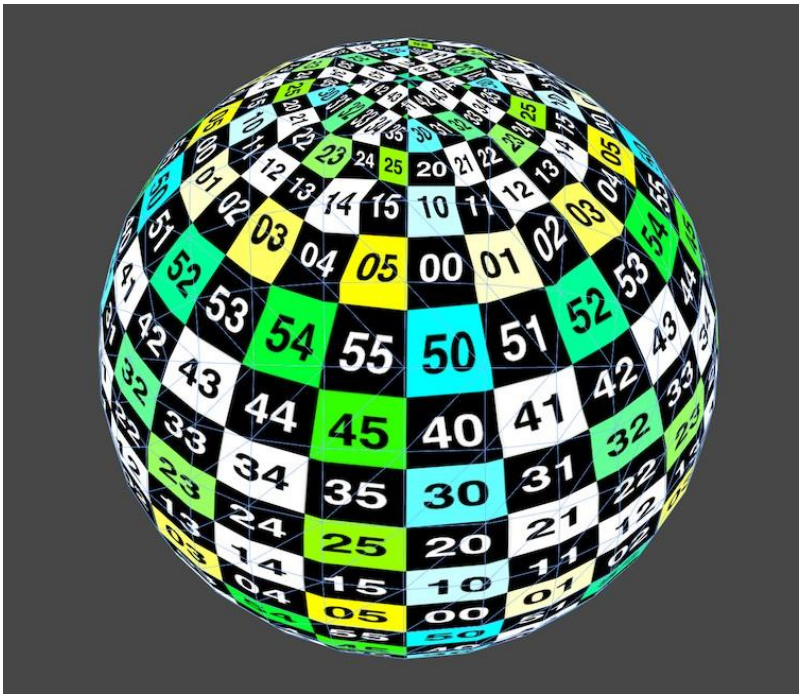
Рендеринг. Основы шейдеров.

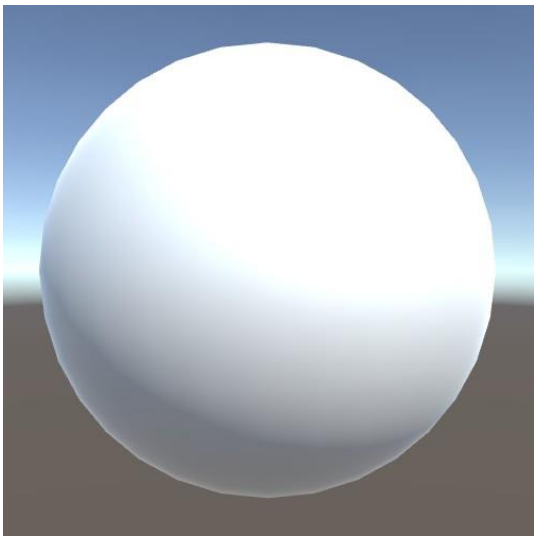
Преобразование вершин. Цветовые пиксели.
Использование свойств шейдеров
Передача данных от вершин к фрагментам
Изучение скомпилированного кода шейдера
Пример текстур, тайлинг и смещение



Сфера с текстурой

1 Исходная сцена

Когда вы создаете новую сцену в Unity, у вас есть камера по умолчанию и направленный источник света. Создайте простую сферу через *GameObject / 3D Object / Sphere*, разместите ее на сцене и направьте камеру на нее.

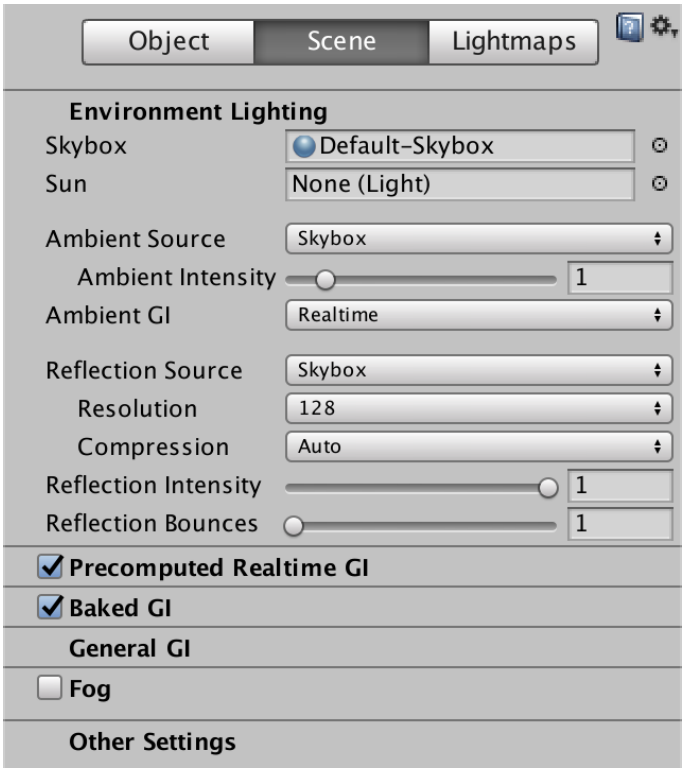


Окно Game с правильно размещенной сферой и камерой

Это очень простая сцена, но на ней уже происходит много сложного рендеринга. Давайте разбираться в происходящем более подробно.

1.1 Упрощаем рендеринг до минимума

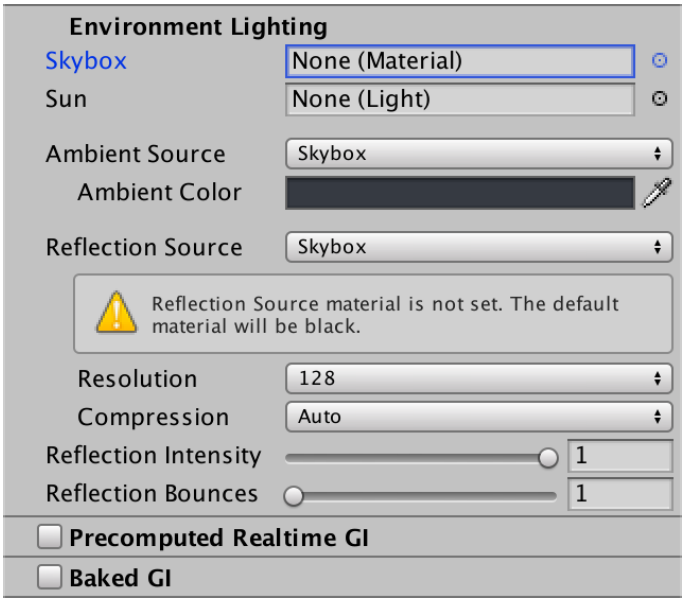
Взгляните на настройки освещения сцены через *Window / Lighting*. Это вызовет окно освещения с тремя вкладками. Нас интересует только вкладка Сцена, которая активна по умолчанию.



Настройки освещения

Существует раздел об окружающем освещении, в котором можно выбрать skybox. Этот skybox используется для создания фона сцены, окружающего освещения и отражения. Установите его в положение "Нет", чтобы он был выключен.

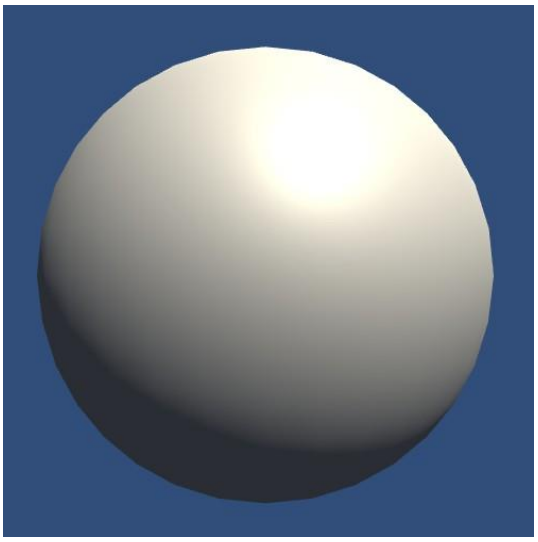
Пока вы этим занимаетесь, вы также можете отключить предварительно рассчитанные и работающие в режиме реального времени панели глобального освещения. Мы не будем использовать их в ближайшее время.



Skybox убран

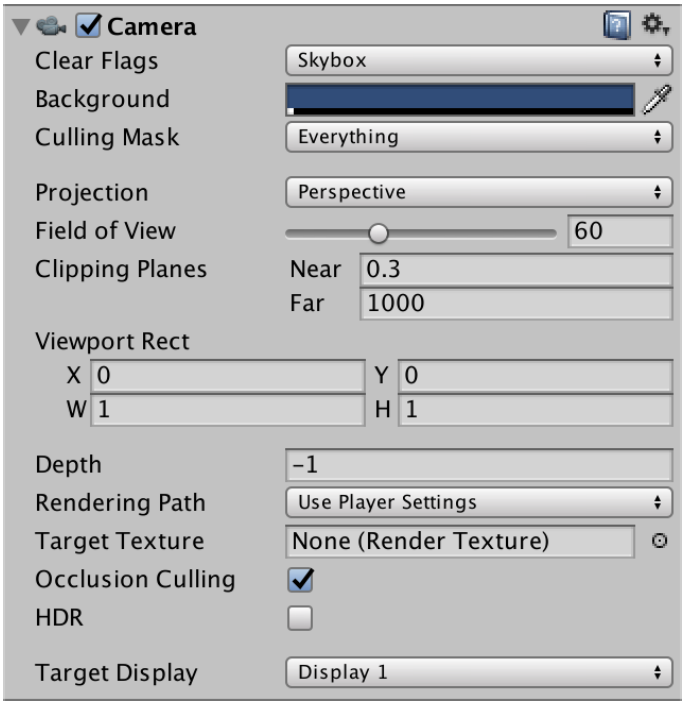
Без skybox источник окружающей среды автоматически переключается на сплошной цвет. Цвет по умолчанию - темно-серый с очень слабым синим оттенком. Отражения становятся сплошными черными, как видно из предупреждающего окошка.

Как и следовало ожидать, сфера стала темнее, а фон теперь сплошной цвет. Однако, фон стал темно-синим. Откуда взялся этот цвет?



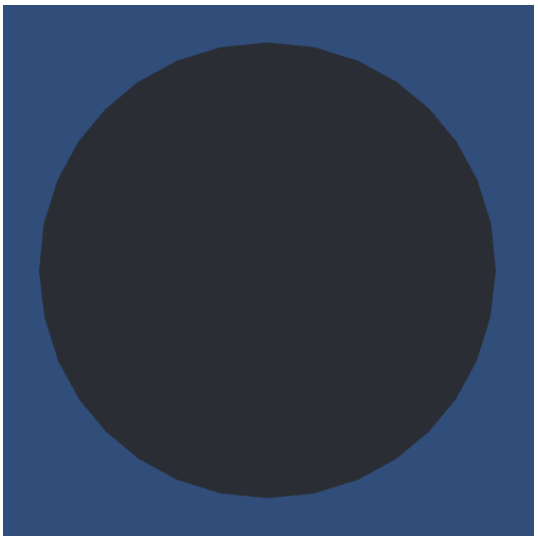
Упрощенное освещение

Цвет фона определяется для каждой камеры. По умолчанию он отображает skybox, но в случае его отсутствия происходит откат (fallback) к сплошному цвету.



Настройки камеры по умолчанию

Для дальнейшего упрощения рендеринга отключите объект направленного света или удалите его. Это позволит избавиться от прямого света в сцене, а также от теней, которые будут отбрасываться из-за его наличия. Остается сплошной фон с силуэтом сферы в цвете окружающей среды.



In the dark.

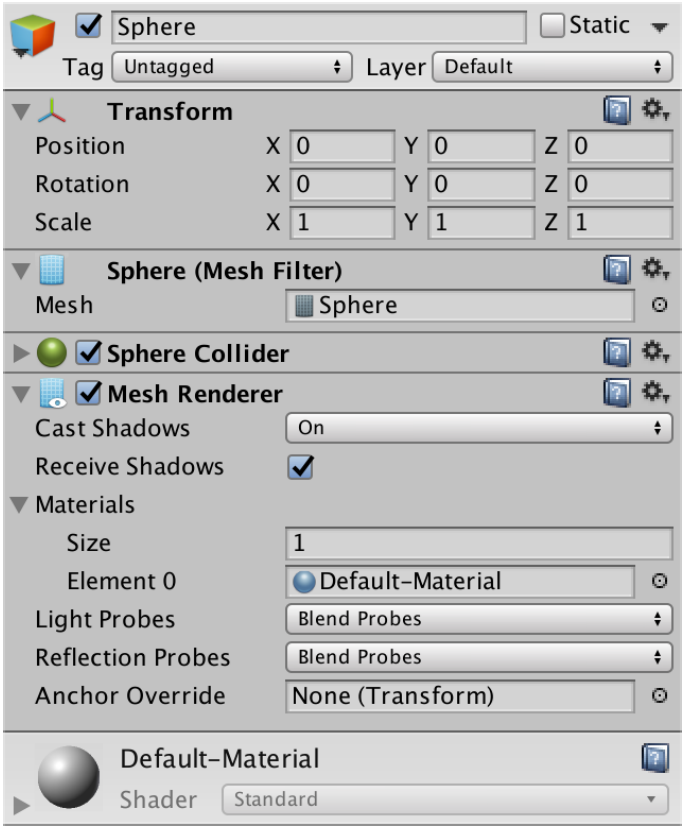
2 От объекта к изображению

Наша очень простая сцена нарисована в два этапа. Во-первых, изображение заполняется цветом фона камеры. Затем поверх этого рисуется силуэт нашей сферы.

Откуда Unity знает, что она должна нарисовать сферу? У нас есть объект сферы, и этот объект имеет компонент рендерера сетки (mesh renderer). Если этот объект попадет в объектив камеры, он должен быть отрисован. Unity проверяет это, определяя пересекает ли bounding box объекта рамки объектива камеры.

Что такое bounding box?

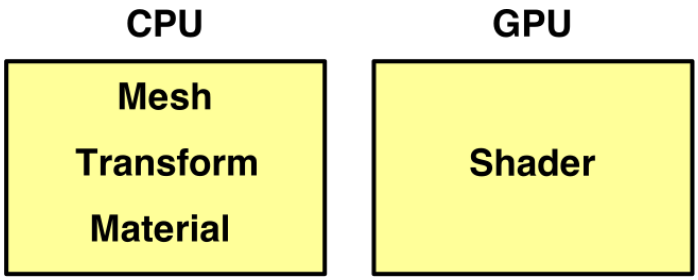
Возьми любую геометрическую сетку (mesh). Теперь найдите самую маленькую коробку, в которую поместится эта фигура. Это и есть bounding box. Она автоматически получается из сетки объекта.
Другими словами, bounding box получается путем аппроксимаций объема, занимаемого сеткой. Если вы не видите коробку, вы определенно не видите меш.



Объект сферы по умолчанию

Компонент трансформации (Transform) используется для изменения положения, ориентации и размера меша и bounding box.

Перед GPU ставится задача рендеринга меша объекта. Конкретные инструкции по рендерингу определяются материалом объекта. Материал ссылается на шейдер, который является программой для GPU.

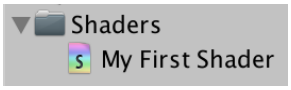


Кто что контролирует

Наш объект в настоящее время имеет материал по умолчанию, который использует стандартный шейдер Unity. Мы собираемся заменить его нашим собственным шейдером, который мы построим с нуля.

2.1 Ваш первый шейдер

Создайте новый шейдер через *Assets / Create / Shader / Unlit Shader* и дайте ему имя, напрмеир, *My First Shader*.



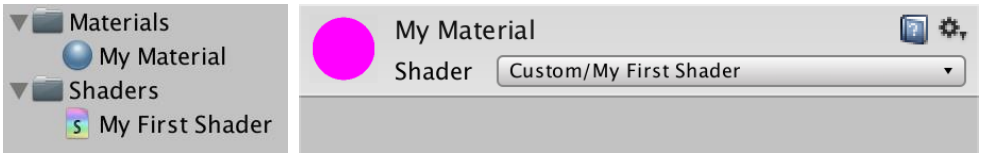
Ваш первый шейдер

Откройте шейдерный файл и удалите его содержимое, чтобы мы могли начать с нуля. Шейдер определяется ключевым словом шейдер. За ним следует строка, описывающая пункт меню для шейдера, который вы можете использовать для выбора этого шейдера. Он не должен совпадать с именем файла. После этого идет блок с содержимым шейдера.

```
Shader "Custom/My First Shader" {

}
```

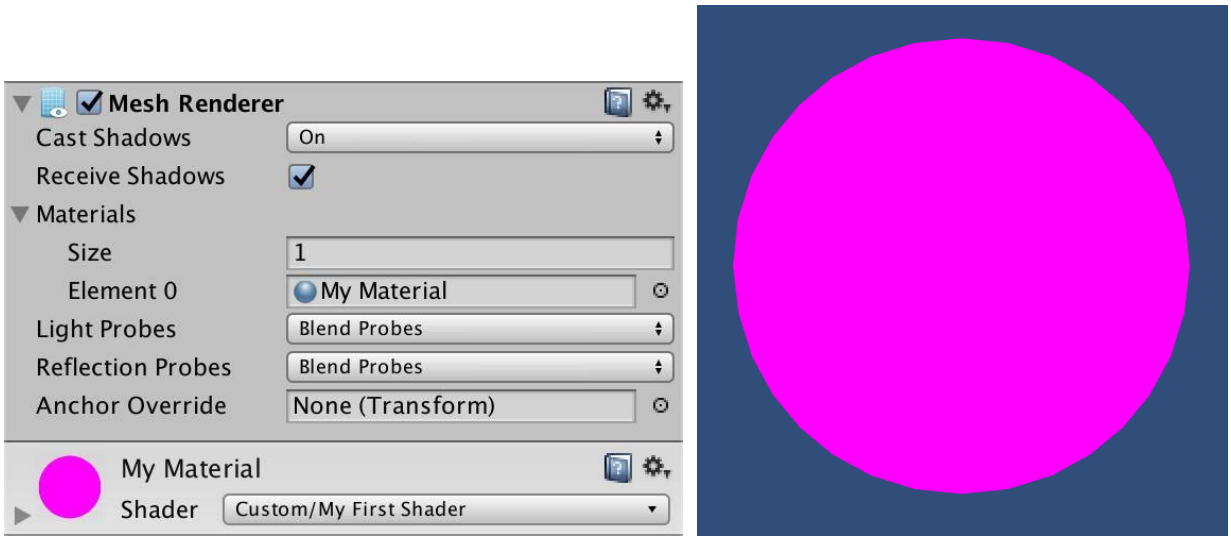
Сохраните файл. Вы получите предупреждение, что шейдер не поддерживается, потому что у него нет ни подшейдеров (Sub-Shaders), ни запасных (Fallback) шейдеров. Это потому, что он пуст.

Несмотря на то, что шейдер не работает, мы уже можем присвоить его материалу. Поэтому создайте новый материал через меню Assets / Create / Material и выберите наш шейдер из шейдерного меню.



Материал с вашим шейдером

Измените наш объект сферы так, чтобы он использовал наш собственный материал, а не материал по умолчанию. Сфера станет пурпурной. Это происходит потому, что Unity переключится на шейдер ошибок, который использует этот цвет, чтобы привлечь ваше внимание к проблеме.



Материал с вашим шейдером, примененным к мешу

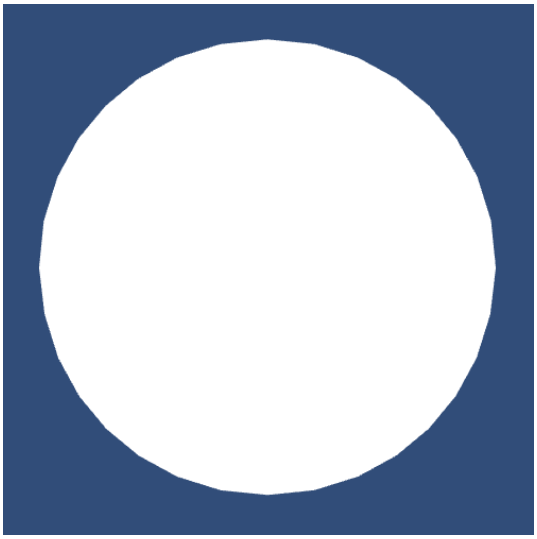
Ошибка шейдера упоминала подшейдеры. Вы можете использовать их, чтобы сгруппировать несколько вариантов шейдеров вместе. Это позволяет предоставить различные подшейдеры для различных платформ сборки или уровней детализации. Например, вы можете иметь один подшейдер для настольных компьютеров, а другой - для мобильных. Нам нужен только один блок сабшейдеров.

```
Shader "Custom/My First Shader" {

    SubShader {

    }
}
```

Когда CPU хочет нарисовать изображение модели на экране, он делает запрос от GPU на отрисовку модели (drawcall). Когда изображение визуализируется на экране, это называется проходом (pass). Подшейдер должен содержать хотя бы один проход. Мы будем использовать один проход, но можно иметь больше. Наличие более одного прохода означает, что объект будет отрисовываться несколько раз, что требуется для множества эффектов.

```
Shader "Custom/My First Shader" {

    SubShader {

        Pass {

        }
    }
}
```

Наша сфера теперь может стать белой, так как мы используем поведение по умолчанию пустого прохода. Если это произойдет, это означает, что у нас больше нет шейдерных ошибок. Однако, вы все равно можете увидеть старые ошибки в консоли. Не пугайтесь, просто старые логи не стираются самостоятельно, но вы всегда можете убрать их вручную, чтобы они вас не смущали.



Белая сфера

2.2 Шейдерные программы

Пришло время написать нашу собственную шейдерную программу. Мы делаем это с помощью языка шейдеров Unity, который является вариантом языков шейдеров HLSL и CG. Мы должны указать начало нашего кода ключевым словом CGPROGRAM. А заканчивать мы должны ключевым словом ENDCG.

```
        Pass {
            CGPROGRAM

            ENDCG
        }
```

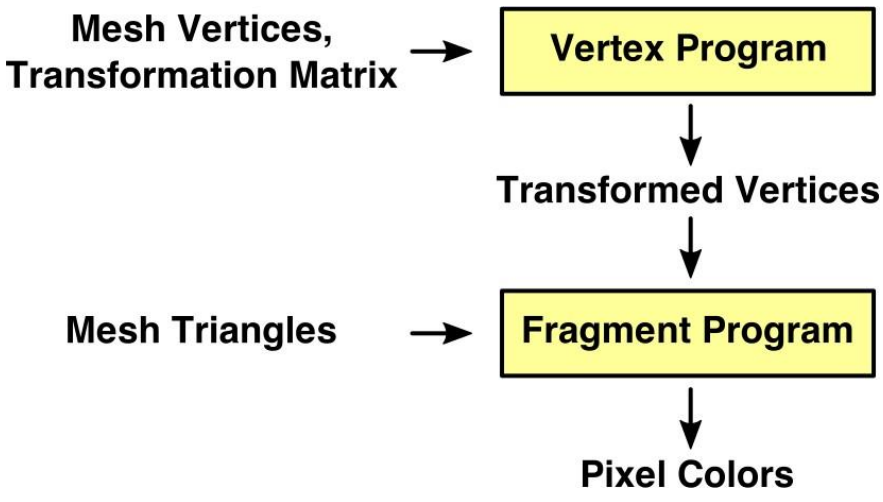
Для чего нужны эти ключевые слова?

Шейдерные проходы могут содержать и другие утверждения помимо шейдерной программы. Поэтому программа должна быть как-то разделена. Почему бы не использовать для этого другой блок? Точного ответа на данный вопрос нет. Скорее всего это старые конструкторские решения, которые когда-то имели смысл, но сейчас уже не имеют. Из-за обратной совместимости мы все еще застряли с ними.

Шейдерный компилятор теперь жалуется, что в нашем шейдере нет вершинных (vertex) и фрагментных (fragment) программ. Шейдеры состоят из двух программ каждая. Простыми словами, вершинная программа рисует геометрию объекта, преобразуя его из пространства объекта (object-space) в пространство дисплея (display place), а фрагментная ее раскрашивает.

Что, в сущности, такое object-space и display place?

Хоть мы и имеем дело с 3D миром, мы всё равно смотрим на него через монитор компьютера, мобильного устройства и т.п., то есть видим плоское изображение (потому что экран – это 2D холст). Таким образом речь идет о проекции трехмерного объекта на экран пользовательского устройства.



Vertex и fragment программы.

Мы должны сообщить компилятору, какие программы использовать, с помощью прагматических директив.

```
CGPROGRAM

#pragma vertex MyVertexProgram
#pragma fragment MyFragmentProgram

ENDCG
```

Что такое pragma?

Слово "прагма" происходит от греческого языка и означает действие или то, что должно быть сделано. Оно используется во многих языках программирования для выдачи специальных директив компилятора.

Компилятор опять жалуется, на этот раз из-за того, что не может найти программы, которые мы указали. Это потому, что мы их еще не определили.

Вершинные и фрагментные программы пишутся как методы, совсем как в C#, хотя обычно их называют функциями. Просто создадим два пустых метода void с соответствующими именами.

```
CGPROGRAM

#pragma vertex MyVertexProgram
#pragma fragment MyFragmentProgram

void MyVertexProgram () {

}

void MyFragmentProgram () {

}

ENDCG
```

В этот момент шейдер скомпилируется, и сфера исчезнет. Или вы все равно получите ошибки. Все зависит от того, какую платформу для рендеринга использует ваш редактор. Если вы используете Direct3D 9, то скорее всего, вы получите ошибки.

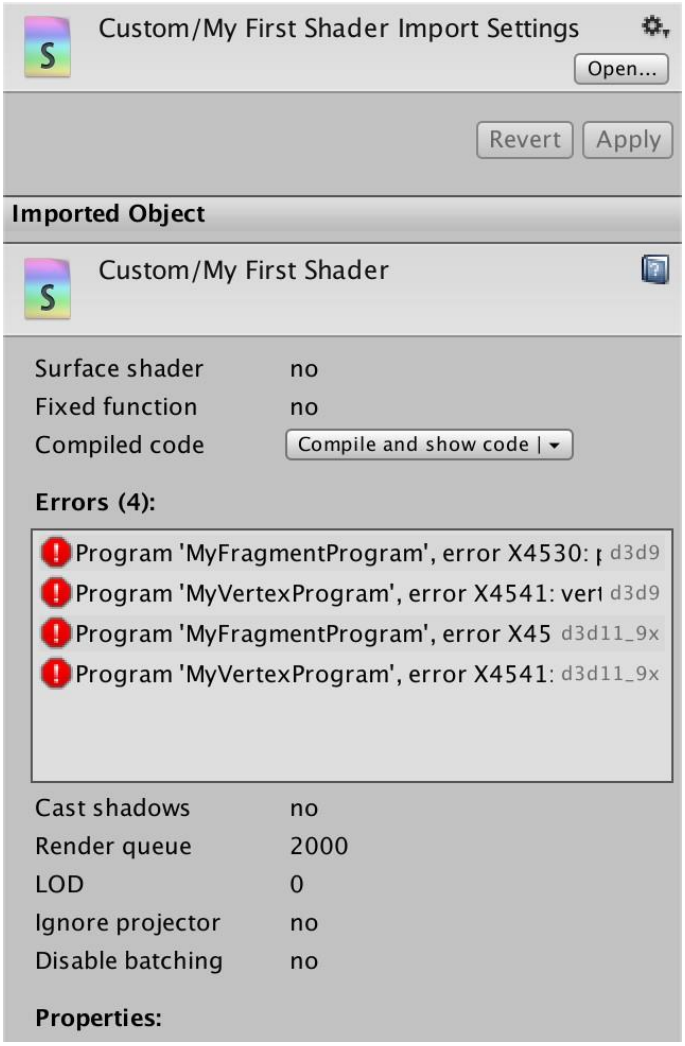
2.3 Компиляция шейдеров

Шейдерный компилятор Unity берет наш код и преобразует его в другую программу, в зависимости от целевой платформы. Различные платформы требуют разных решений. Например, Direct3D для Windows, Metal для Macs, OpenGL ES для Android и так далее. Здесь мы имеем дело не с одним компилятором, а с несколькими.

Какой компилятор вы в итоге используете, зависит от того, на что вы нацелены. И

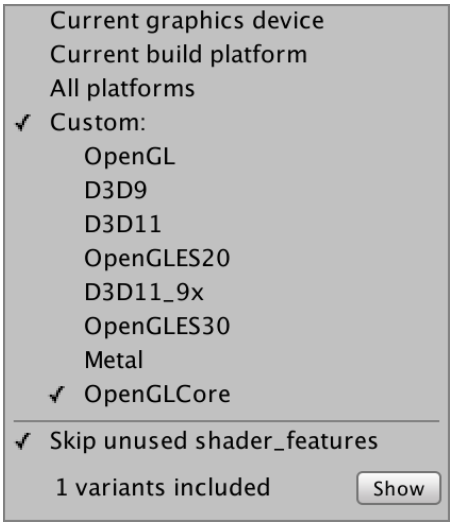
поскольку эти компиляторы не идентичны, вы можете получить разные результаты для разных платформ. Например, наши пустые программы прекрасно работают с OpenGL и Direct3D 11, но не работают при таргетировании Direct3D 9.

Выберите шейдер в редакторе и посмотрите в окно инспектора. В нем отображается некоторая информация о шейдере, в том числе и об ошибках текущего компилятора. Также имеется запись "Скомпилированный код" с кнопкой "Скомпилировать и показать код" и выпадающим меню. Если вы нажмете на кнопку, Unity скомпилирует шейдер и откроет его вывод в редакторе, чтобы вы могли осмотреть сгенерированный код.



Инспектор шейдера со списком ошибок

Вы можете выбрать, для каких платформ вы вручную компилируете шейдер, через выпадающее меню. По умолчанию компиляция производится для графического устройства, используемого вашим редактором. Вы можете вручную скомпилировать и для других платформ, либо для вашей текущей платформы сборки, всех платформ, на которые у вас есть лицензии, либо с помощью пользовательского выбора. Это позволяет вам быстро убедиться, что ваш шейдер компилируется на нескольких платформах, без необходимости делать полные сборки.



Выбор OpenGLCore.

Чтобы скомпилировать выбранные программы, закройте всплывающее окно и нажмите кнопку *Compile и show code*. Нажав маленькую кнопку Show внутри всплывающего окна, вы увидите использованные варианты шейдеров, что сейчас не очень удобно. Например, вот результирующий код, когда наш шейдер компилируется для OpenGlCore.

```
Shader "Custom/My First Shader" {
 SubShader {
  Pass {
   GpuProgramID 16807
 Program "vp" {
 SubProgram "glcore " {
 "#ifdef VERTEX
 #version 150
 #extension GL_ARB_explicit_attrib_location :require
 #extension GL_ARB_shader_bit_encoding : enable
 void main()
 {
     return;
 }
 #endif
 #ifdef FRAGMENT
 #version 150
 #extension GL_ARB_explicit_attrib_location :require
 #extension GL_ARB_shader_bit_encoding : enable
 void main()
 {
     return;
 }
 #endif
 "
 }
 }
 Program "fp" {
 SubProgram "glcore " {
 "// shader disassembly not supported on glcore"
 }
 }
  }
 }
 }
```

Сгенерированный код разбит на два блока, vp и fp, для вершинных и фрагментированных программ. Однако, в случае OpenGL обе программы попадают в блок vp. Две основные функции соответствуют двум нашим пустым методам. Поэтому остановимся на них и проигнорируем другой код

```
#ifdef VERTEX
void main()
{
    return;
}
#endif
#ifdef FRAGMENT
void main()
{
    return;
}
#endif
```

А вот сгенерированный код для Direct3D 11, разобранный на интересные части. Он выглядит совсем по-другому, но очевидно, что код делает не так уж много.

```
Program "vp" {
SubProgram "d3d11 " {
      vs_4_0
   0: ret
}
}
Program "fp" {
SubProgram "d3d11 " {
      ps_4_0
   0: ret
}
}
```

Пока мы работаем над нашими программами, в лабораторной будет часто показываться скомпилированный код для OpenGLCore и D3D11, так что вы можете получить представление о том, что происходит под капотом.

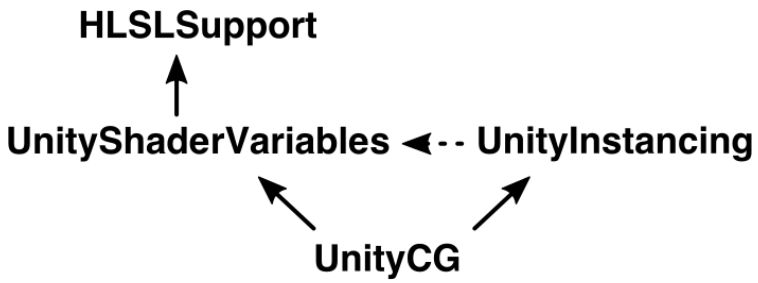
2.4 Использование дополнительных файлов

Для создания функционального шейдера необходимо много шаблонного кода, который определяет общие переменные, функции и другие вещи. Если бы это была программа на C#, мы бы поместили этот код в другие классы. Но в шейдерах нет классов. Это всего лишь один большой файл со всем кодом, без группировки по классам или пространствам имен.

К счастью, мы можем разделить код на несколько файлов. Можно использовать директиву #include для загрузки содержимого другого файла в текущий файл. Типичным включаемым файлом является UnityCG.cginc, так что давайте сделаем это.

```
CGPROGRAM

#pragma vertex MyVertexProgram
#pragma fragment MyFragmentProgram

#include "UnityCG.cginc"

void MyVertexProgram () {

}

void MyFragmentProgram () {

}

ENDCG
```

UnityCG.cginc один из прилагаемых файлов шейдеров, которые связаны с Unity. Он включает в себя несколько других важных файлов и содержит некоторые общие функциональные возможности.



Файловая иерархия UnityCG.

UnityShaderVariables.cginc определяет целую кучу шейдерных переменных, необходимых для рендеринга, таких как трансформация, камера и световые данные. Все они задаются Unity, когда это необходимо.

HLSLSupport.cginc настраивает все так, чтобы вы могли использовать один и тот же код независимо от того, на какую платформу вы нацелены. Таким образом, вам не нужно беспокоиться об использовании специфичных для платформы типов данных и тому подобного.

UnityInstancing.cginc реализует функцию инстанцирования, которая является специфической техникой рендеринга для уменьшения количества обращений к отрисовке. Хотя он не включает файл напрямую, он зависит от UnityShaderVariables.

Обратите внимание, что содержимое этих файлов эффективно копируется в ваш собственный файл, заменяя директиву include. Это происходит на этапе препроцессирования, на котором выполняются все директивы препроцессирования. Эти директивы - все утверждения, которые начинаются с хэша, например #include и #pragma. После завершения этого шага код снова обрабатывается и фактически компилируется.

2.5 Реализация выходных данных

Чтобы что-то отрисовать, наши шейдерные программы должны давать результаты. Например, ершинная программа должна вернуть конечные координаты вершины.

Измените тип функции с void на float4. Функция float4 - это просто коллекция из четырех чисел с плавающей точкой. Просто верните пока 0.

```
float4 MyVertexProgram () {
	return 0;
}
```

Теперь мы получаем ошибку по поводу пропущенной семантики. Компилятор видит, что мы возвращаем коллекцию из четырех флотов, но не знает, что представляют собой эти данные. Так что он не знает, что с ними должен делать GPU. Нужно быть предельно конкретным о выходных данных нашей программы.

В данном случае мы пытаемся вывести положение вершины. Мы должны указать это, прикрепив к нашему методу семантику SV_POSITION. SV означает системное значение, а POSITION - конечную позицию вершины.

```
float4 MyVertexProgram () : SV_POSITION {
	return 0;
}
```

Программа фрагмента должна выводить значение цвета RGBA для одного пикселя. Для этого также можно использовать float4. Возвращая 0, мы получим сплошной черный цвет.

```
float4 MyFragmentProgram () {
	return 0;
}
```

Программа-фрагмент также требует семантики. В этом случае мы должны указать, куда должен быть записан конечный цвет. Мы используем SV_TARGET, который является целью шейдера по умолчанию. Это буфер кадра, который содержит изображение, которое мы генерируем.

```
float4 MyFragmentProgram () : SV_TARGET {
	return 0;
}
```

Вывод вершинной программы используется как ввод для фрагментной программы. Это говорит о том, что фрагментная программа должна получить параметр, совпадающий с выводом вершинной программы.

```
float4 MyFragmentProgram (float4 position) : SV_TARGET {
	return 0;
}
```

Неважно, какое имя мы дадим параметру, но мы должны убедиться, что используем правильную семантику

```
float4 MyFragmentProgram (
	float4 position : SV_POSITION
) : SV_TARGET {
	return 0;
}
```

Наш шейдер снова компилируется без ошибок, но при этом сфера исчезла. Это не должно удивлять, потому что мы разрушаем все ее вершины до единой точки.

Если вы посмотрите на скомпилированные программы OpenGLCore, то увидите, что теперь они пишут в выводимые значения. И наши единичные значения действительно были заменены четырехкомпонентными векторами.

```
#ifdef VERTEX
void main()
{
	gl_Position = vec4(0.0, 0.0, 0.0, 0.0);
	return;
}
#endif
#ifdef FRAGMENT
layout(location = 0) out vec4 SV_TARGET0;
void main()
{
	SV_TARGET0 = vec4(0.0, 0.0, 0.0, 0.0);
	return;
}
#endif
```

То же самое верно и для программ D3D11, хотя синтаксис отличается.

```
Program "vp" {
SubProgram "d3d11 " {
      vs_4_0
      dcl_output_siv o0.xyzw, position
   0: mov o0.xyzw, l(0,0,0,0)
   1: ret
}
}
Program "fp" {
SubProgram "d3d11 " {
      ps_4_0
      dcl_output o0.xyzw
   0: mov o0.xyzw, l(0,0,0,0)
   1: ret
}
}
```

2.6 Преобразование вершин

Чтобы вернуть нашу сферу обратно, наша вершинная программа должна выдать правильное положение вершины. Для этого нам необходимо знать положение объектного пространства вершины (object-space). Мы можем получить к ней доступ, добавив в нашу функцию переменную с семантикой POSITION. Позиция будет представлена в виде однородных координат вида $\begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$, поэтому ее тип - **float4**.

```
float4 MyVertexProgram (float4 position : POSITION) : SV_POSITION {
    return 0;
}
```

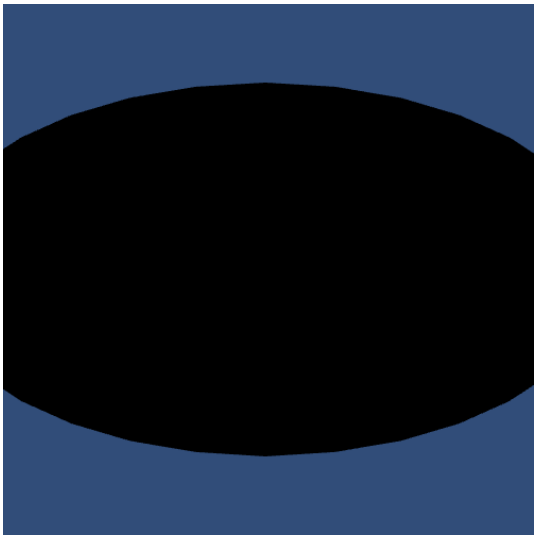
Давайте начнем с прямого возвращения этой позиции.

```
float4 MyVertexProgram (float4 position : POSITION) : SV_POSITION {
    return position;
}
```

Скомпилированные программы для работы с вершинами теперь будут вводить вершины и копировать их в свои выходные данные.

```
in  vec4 in_POSITION0;
void main()
{
    gl_Position = in_POSITION0;
    return;
}
```

```
Bind "vertex" Vertex
      vs_4_0
      dcl_input v0.xyzw
      dcl_output_siv o0.xyzw, position
   0: mov o0.xyzw, v0.xyzw
   1: ret
```



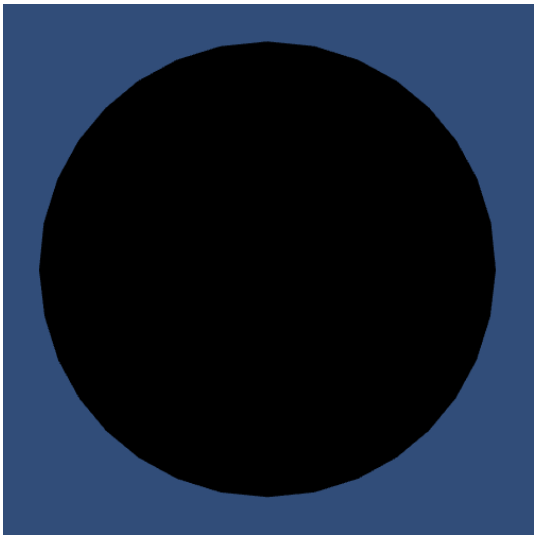
Позиции вершин в сыром виде

Черная сфера станет видимой, но она будет искажена. Это потому, что мы используем object-space позиции как display позиции. Таким образом, перемещение сферы не будет визуально иметь никакого значения.

Мы должны умножить положение необработанной вершины на матрицу отображения модели. Эта матрица объединяет иерархию трансформирования объекта с преобразованием и проекцией камеры.

Матрица 4 на 4 MVP определена в UnityShaderVariables как UNITY_MATRIX_MVP. Мы можем использовать функцию mul, чтобы умножить ее на позицию вершины. Это корректно спроецирует нашу сферу на дисплей. Вы также можете перемещать, поворачивать и масштабировать ее, и изображение будет изменяться согласно ожиданиям.

```
float4 MyVertexProgram (float4 position : POSITION) : SV_POSITION {
    return mul(UNITY_MATRIX_MVP, position);
}
```



Корректное отображение.

Если вы проверите вершинную программу OpenGLCore, то заметите, что внезапно появилось множество однотипных переменных (uniform variables), несмотря на то что они не используются и будут игнорироваться.

Вы также увидите матричное умножение, закодированное как связка умножения и сложения.

```
uniform     vec4 _Time;
uniform     vec4 _SinTime;
uniform     vec4 _CosTime;
uniform     vec4 unity_DeltaTime;
uniform     vec3 _WorldSpaceCameraPos;
…
in  vec4 in_POSITION0;
vec4 t0;
void main()
{
    t0 = in_POSITION0.yyyy * glstate_matrix_mvp[1];
    t0 = glstate_matrix_mvp[0] * in_POSITION0.xxxx + t0;
    t0 = glstate_matrix_mvp[2] * in_POSITION0.zzzz + t0;
    gl_Position = glstate_matrix_mvp[3] * in_POSITION0.wwww + t0;
    return;
}
```

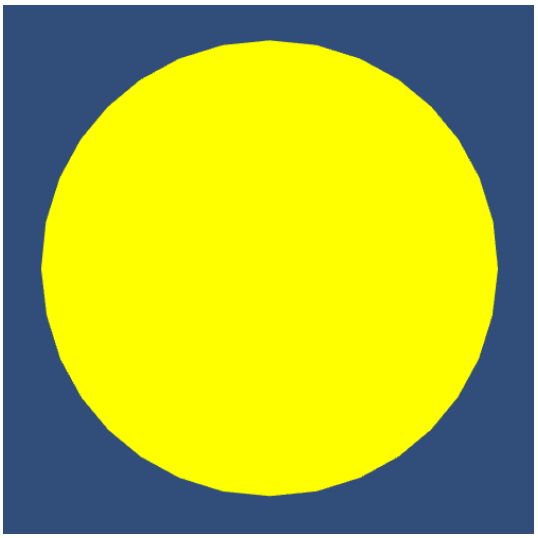
Компилятор D3D11 не утруждает себя включением неиспользуемых переменных. Он кодирует умножение матриц с помощью mul и трех mad инструкций. Mad инструкция представляет собой умножение с последующим сложением.

```
Bind "vertex" Vertex
ConstBuffer "UnityPerDraw" 352
Matrix 0 [glstate_matrix_mvp]
BindCB  "UnityPerDraw" 0
      vs_4_0
      dcl_constantbuffer cb0[4], immediateIndexed
      dcl_input v0.xyzw
      dcl_output_siv o0.xyzw, position
      dcl_temps 1
   0: mul r0.xyzw, v0.yyyy, cb0[1].xyzw
   1: mad r0.xyzw, cb0[0].xyzw, v0.xxxx, r0.xyzw
   2: mad r0.xyzw, cb0[2].xyzw, v0.zzzz, r0.xyzw
   3: mad o0.xyzw, cb0[3].xyzw, v0.wwww, r0.xyzw
   4: ret
```

3 Раскрашивающие пиксели

Теперь, когда мы получили правильную форму, давайте добавим немного цвета. Самое простое - использовать постоянный цвет, например, желтый.

```
            float4 MyFragmentProgram (
                float4 position : SV_POSITION
            ) : SV_TARGET {
                return float4(1, 1, 0, 1);
            }
```



Желтая сфера

Конечно, нам не всегда нужны желтые предметы. В идеале, наш шейдер поддерживает любой цвет. Тогда вы могли бы использовать материал, чтобы настроить, какой цвет применить. Это делается через свойства шейдера.

3.1 Свойства шейдеров

Свойства шейдеров объявляются в отдельном блоке. Добавьте его в верхнюю часть шейдера.

```
Shader "Custom/My First Shader" {

	Properties {
	}

	SubShader {
		…
	}
}
```

Поместите свойство с именем _Tint в новый блок. Вы можете назвать его любым именем, но конвенция начинается с подчеркивания, за которым следует заглавная буква, а затем строчная. Идея заключается в том, что ничто больше не использует эту конвенцию, что предотвращает случайное дублирование имен.

```
	Properties {
		_Tint
	}
```

Имя свойства должно сопровождаться строкой и типом в скобках, как будто вы вызываете метод. Строка используется для обозначения свойства в инспекторе материалов. В данном случае типом является Color.

```
	Properties {
		_Tint ("Tint", Color)
	}
```

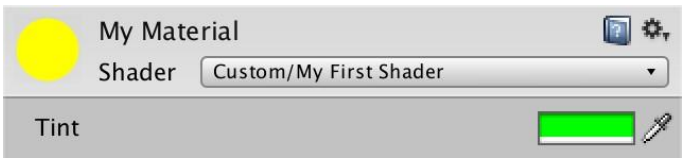
Последняя часть декларации свойств - это присвоение значения по умолчанию. Установим его в белый цвет.

```
	Properties {
		_Tint ("Tint", Color) = (1, 1, 1, 1)
	}
```

Теперь наше свойство оттенка должно появиться в разделе свойств нашего шейдерного инспектора.

Свойство шейдера

При выборе материала вы увидите новое свойство Tint, установленное на белый цвет. Вы можете изменить его на любой цвет, например, зеленый.



Свойство материала

3.2 Доступ к свойствам

Чтобы действительно использовать это свойство, нужно добавить переменную в шейдерный код. Ее имя должно точно совпадать с именем свойства, так что это будет _Tint. Затем мы можем просто вернуть эту переменную в нашей фрагментной программе.

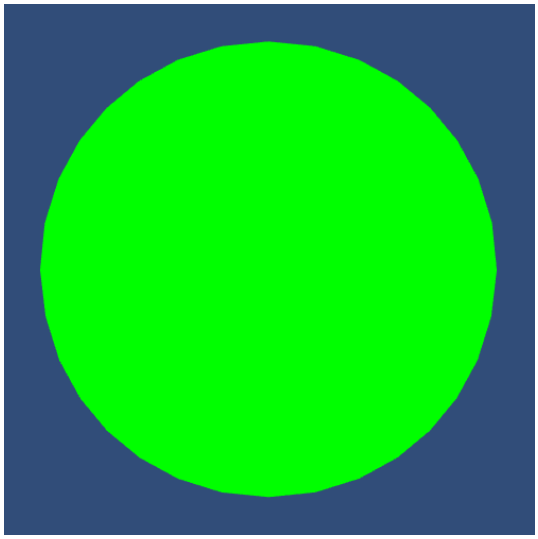
```
			#include "UnityCG.cginc"

			float4 _Tint;

			float4 MyVertexProgram (float4 position : POSITION) : SV_POSITION {
				return mul(UNITY_MATRIX_MVP, position);
			}

			float4 MyFragmentProgram (
				float4 position : SV_POSITION
			) : SV_TARGET {
				return _Tint;
			}
```

Обратите внимание, что переменная должна быть определена, прежде чем ее можно будет использовать. Хотя в классе C# можно без проблем менять порядок полей и методов, для шейдеров это не так. Компилятор работает сверху вниз. Он не будет смотреть вперед.

В скомпилированные фрагментные программы теперь включена переменная tint.

```
uniform		vec4 _Time;
uniform		vec4 _SinTime;
uniform		vec4 _CosTime;
uniform		vec4 unity_DeltaTime;
uniform		vec3 _WorldSpaceCameraPos;
…
uniform		vec4  Tint;
layout(location = 0) out vec4 SV_TARGET0;
void main()
{
    SV_TARGET0 = _Tint;
    return;
}
```

```
ConstBuffer "$Globals" 112
Vector 96 [_Tint]
BindCB  "$Globals" 0
      ps_4_0
      dcl_constantbuffer cb0[7], immediateIndexed
      dcl_output o0.xyzw
   0: mov o0.xyzw, cb0[6].xyzw
   1: ret
```



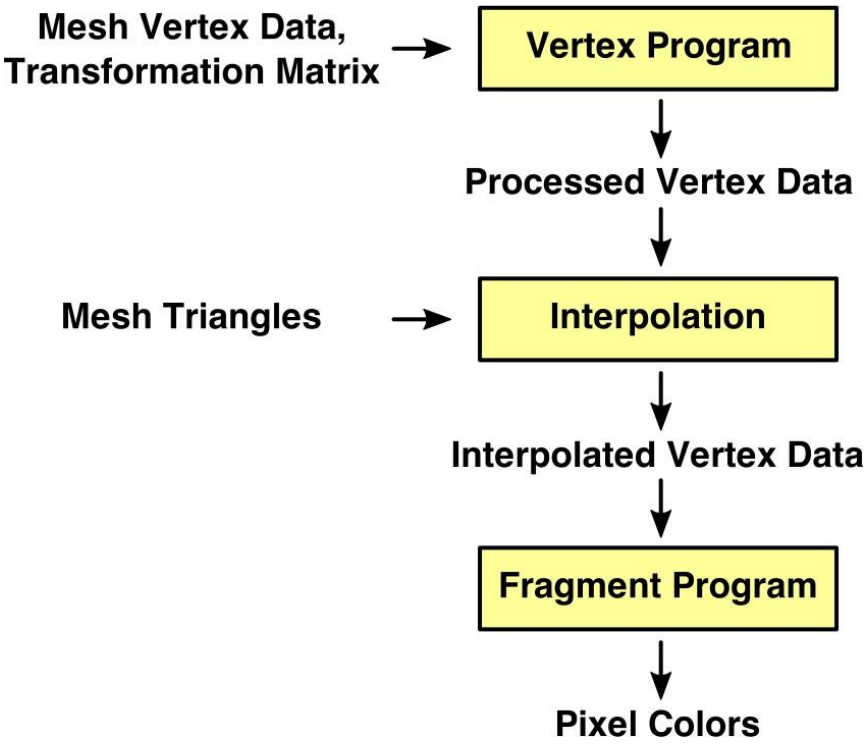
Зеленая сфера

3.3 От вершин к фрагментам

До сих пор мы давали всем пикселям один и тот же цвет, но это сильно нас ограничивает. Обычно, данные о вершинах играют большую роль. Например, мы могли бы интерпретировать их положение как цвет.

Однако, преобразованная позиция не очень полезна. Поэтому давайте вместо этого используем локальную позицию в меше как цвет. Как передать эти дополнительные данные из программы по вершинам во фрагмент программы?

GPU создает изображения, разбивая его на треугольники. Для этого требуется три обработанные вершины, соединенными между собой линиями. Для каждого пикселя, покрытого таким треугольником, вызывается фрагмент программа, обрабатывающая интерполированные данные.



Интерполция вершин

Таким образом, вывод вершинной программы в качестве входной информации для фрагментной программы вообще не используется. Процесс интерполяции находится между ними. Здесь интерполируются данные SV_POSITION, но можно интерполировать и другие вещи.

Для доступа к интерполированной локальной позиции добавьте параметр во фрагмент программы. Так как нам нужны только X, Y и Z компоненты, то float3 нам хватит. Затем мы можем вывести позицию как цвет. Нам достаточно предоставить четвертую компоненту цвета, которая может просто остаться 1.

```
float4 MyFragmentProgram (
	float4 position : SV_POSITION,
	float3 localPosition
) : SV_TARGET {
	return float4(localPosition, 1);
}
```

Еще раз приходится использовать семантику, чтобы подсказать компилятору, как интерпретировать эти данные. Мы используем TEXCOORD0.

```
float4 MyFragmentProgram (
	float4 position : SV_POSITION,
	float3 localPosition : TEXCOORD0
) : SV_TARGET {
	return float4(localPosition, 1);
}
```

Мы не работаем с координатами текстуры, так почему же TEXCOORD0?

Потому что нет общей семантики для интерполированных данных. Все просто используют семантику текстурных координат для всего, что интерполировано и не является положением вершины. TEXCOORD0, TEXCOORD1, TEXCOORD2 и так далее.

Это сделано из соображений совместимости.
Также существует специальная семантика цвета, но она используется редко и доступна не на всех платформах.

Скомпилированные шейдеры фрагментов теперь будут использовать интерполированные данные вместо однородного оттенка.

```
in  vec3 vs_TEXCOORD0;
layout(location = 0) out vec4 SV_TARGET0;
void main()
{
    SV_TARGET0.xyz = vs_TEXCOORD0.xyz;
    SV_TARGET0.w = 1.0;
    return;
}
```

```
    ps_4_0
     dcl_input_ps linear v0.xyz
     dcl_output o0.xyzw
  0: mov o0.xyz, v0.xyzx
  1: mov o0.w, l(1.000000)
  2: ret
```

Конечно, вершинная программа должна вывести локальную позицию, чтобы это работало. Мы можем сделать это, добавив к ней выходной параметр, с тем же семантическим TEXCOORD0. Имена параметров функций вершины и фрагмента не обязательно должны совпадать. Все дело в семантике.

```
float4 MyVertexProgram (
    float4 position : POSITION,
    out float3 localPosition : TEXCOORD0
) : SV_POSITION {
    return mul(UNITY_MATRIX_MVP, position);
}
```

Для передачи данных через вершинную программу скопируйте компоненты X, Y и Z из позиции в позицию localPosition.

```
float4 MyVertexProgram (
    float4 position : POSITION,
    out float3 localPosition : TEXCOORD0
) : SV_POSITION {
    localPosition = position.xyz;
    return mul(UNITY_MATRIX_MVP, position);
}
```

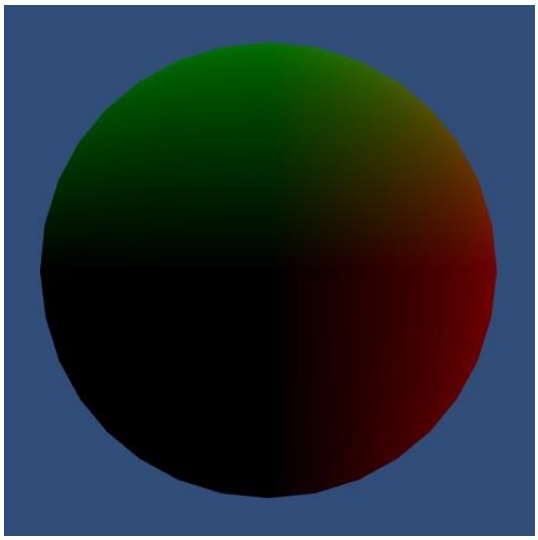
Что делает xyz?

Это гибкий доступ к одному компоненту вектора. Вы можете использовать его для фильтрации, переупорядочивания и повторения компонентов float. Например, .x, .xy, .yx, .xx. В данном случае мы используем его для захвата первых трех компонент позиции, игнорируя четвертую. Все четыре компонента будут .xyzw. Можно использовать и соглашения по наименованию цветов, например .rgba.

Дополнительный вывод вершинной программы включается в шейдеры компилятора, и мы увидим, как наша сфера окрасится.

```
in  vec4  in_POSITION0;
out vec3 vs_TEXCOORD0;
vec4 t0;
void main()
{
    t0 = in_POSITION0.yyyy * glstate_matrix_mvp[1];
    t0 = glstate_matrix_mvp[0] * in_POSITION0.xxxx + t0;
    t0 = glstate_matrix_mvp[2] * in_POSITION0.zzzz + t0;
    gl_Position = glstate_matrix_mvp[3] * in_POSITION0.wwww + t0;
    vs_TEXCOORD0.xyz = in_POSITION0.xyz;
    return;
}
```

```
Bind "vertex" Vertex
ConstBuffer "UnityPerDraw" 352
Matrix 0 [glstate_matrix_mvp]
BindCB  "UnityPerDraw" 0
      vs_4_0
      dcl_constantbuffer cb0[4], immediateIndexed
      dcl_input v0.xyzw
      dcl_output_siv o0.xyzw, position
      dcl_output o1.xyz
      dcl_temps 1
   0: mul r0.xyzw, v0.yyyy, cb0[1].xyzw
   1: mad r0.xyzw, cb0[0].xyzw, v0.xxxx, r0.xyzw
   2: mad r0.xyzw, cb0[2].xyzw, v0.zzzz, r0.xyzw
   3: mad o0.xyzw, cb0[3].xyzw, v0.wwww, r0.xyzw
   4: mov o1.xyz, v0.xyzx
   5: ret
```



Интерпретация локальных позиций как цветов

3.4 Использование структур

Как вы думаете, списки параметров наших программ выглядят беспорядочно? Будет только хуже по мере того, как мы будем передавать между ними все больше и больше данных. Так как вывод вершин должен совпадать с вводом фрагмента, было бы удобно, если бы мы могли определить список параметров в одном месте. К счастью, мы можем это сделать.

Мы можем определить структуры данных, которые представляют собой просто набор переменных. Они сродни структурам в C#, за исключением того, что синтаксис немного отличается. Вот структура, которая определяет данные, которые мы интерполируем. Обратите внимание на использование точки с запятой после ее определения.

```
struct Interpolators {
	float4 position : SV_POSITION;
	float3 localPosition : TEXCOORD0;
};
```

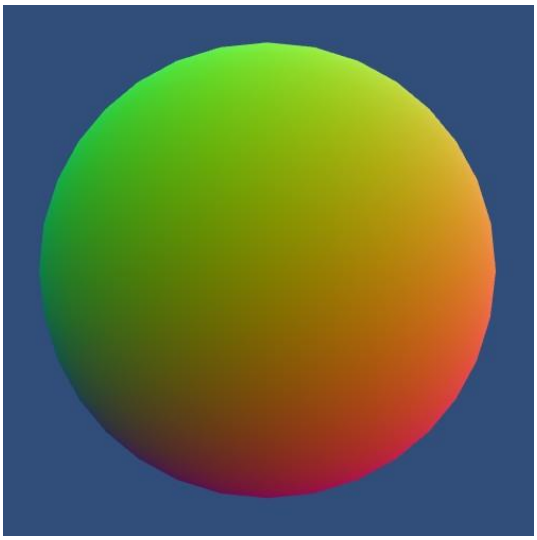
Использование этой структуры делает наш код намного аккуратнее.

```
float4 _Tint;

struct Interpolators {
	float4 position : SV_POSITION;
	float3 localPosition : TEXCOORD0;
};

Interpolators MyVertexProgram (float4 position : POSITION) {
	Interpolators i;
	i.localPosition = position.xyz;
	i.position = mul(UNITY_MATRIX_MVP, position);
	return i;
}

float4 MyFragmentProgram (Interpolators i) : SV_TARGET {
	return float4(i.localPosition, 1);
}
```

3.5 Преобразование цветов

Из-за того, что отрицательные цвета сводятся к нулю, наша сфера становится довольно темной. Так как по умолчанию сфера имеет object-space радиус равный ½, цветовые каналы оказываются где-то между значениями -½ и ½. Мы хотим переместить их в диапазон 0-1, что мы можем сделать, добавив ½ ко всем каналам.

```
	return float4(i.localPosition + 0.5, 1);
```



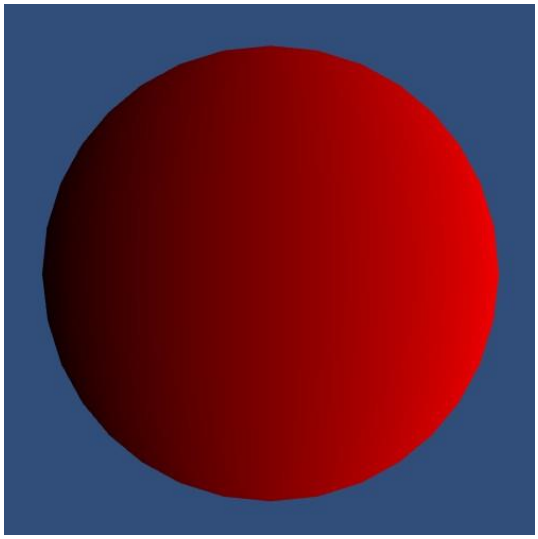
Раскрашенная локальная позиция.

Мы также можем применить наш оттенок путем его учета в результате.

```
return float4(i.localPosition + 0.5, 1) * _Tint;
```

```
uniform     vec4  Tint;
in  vec3 vs_TEXCOORD0;
layout(location = 0) out vec4 SV_TARGET0;
vec4 t0;
void main()
{
    t0.xyz = vs_TEXCOORD0.xyz + vec3(0.5, 0.5, 0.5);
    t0.w = 1.0;
    SV_TARGET0 = t0 * _Tint;
    return;
}
```

```
ConstBuffer "$Globals" 128
Vector 96 [_Tint]
BindCB  "$Globals" 0
      ps_4_0
      dcl_constantbuffer cb0[7], immediateIndexed
      dcl_input_ps linear v0.xyz
      dcl_output o0.xyzw
      dcl_temps 1
   0: add r0.xyz, v0.xyzx, l(0.500000, 0.500000, 0.500000, 0.000000)
   1: mov r0.w, l(1.000000)
   2: mul o0.xyzw, r0.xyzw, cb0[6].xyzw
   3: ret
```

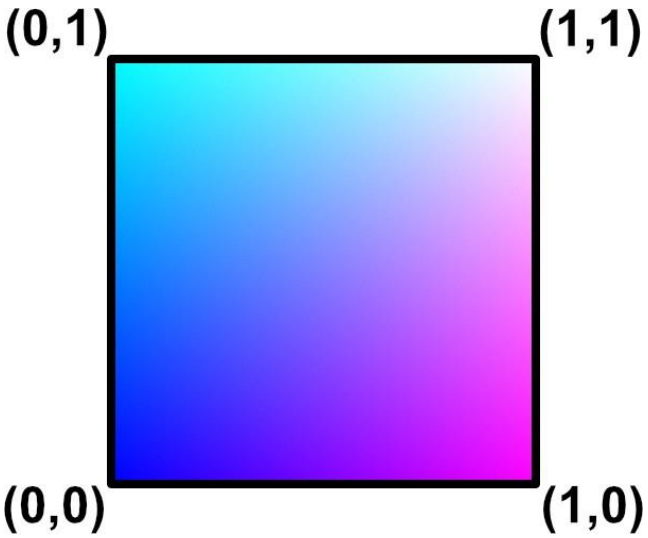


Локальная позиция с красной заливкой, учитывается только X

4 Текстурирование

Если вы хотите добавить в mesh более явные детали и разнообразие, не добавляя больше полигонов, вы можете использовать текстуру. Затем вы проецируете изображение на треугольники сетки.

Координаты текстуры используются для управления проекцией. Это пары 2D координат, которые покрывают все изображение в однокомпонентной квадратной области, независимо от фактического соотношения сторон текстуры. Горизонтальные координаты известны как U, а вертикальные - как V. Поэтому их обычно называют координатами UV.



UV coordinates covering an image.

Координата U увеличивается слева направо. Таким образом, она равна 0 с левой стороны изображения, ½ наполовину, и 1 с правой стороны. Координата V работает таким же образом, по вертикали. Она увеличивается снизу вверх, за исключением Direct3D, где она идет сверху вниз. Обычно об этой разнице не нужно беспокоиться.

4.1 Использование UV координат

Unity меши по умолчанию имеют UV-координаты, подходящие для текстурного отображения. Вершинная программа может получить к ним доступ через параметр с семантикой TEXCOORD0.

```
Interpolators MyVertexProgram (
	float4 position : POSITION,
	float2 uv : TEXCOORD0
) {
	Interpolators i;
	i.localPosition = position.xyz;
	i.position = mul(UNITY_MATRIX_MVP, position);
	return i;
}
```

Наша вершинная программа теперь использует более одного входного параметра. И снова мы можем использовать структуру для их группировки.

```
struct VertexData {
	float4 position : POSITION;
	float2 uv : TEXCOORD0;
};

Interpolators MyVertexProgram (VertexData v) {
	Interpolators i;
	i.localPosition = v.position.xyz;
	i.position = mul(UNITY_MATRIX_MVP, v.position);
	return i;
}
```

Просто передадим UV-координаты прямо во фрагмент программы, заменяя локальную позицию.

```
struct Interpolators {
	float4 position : SV_POSITION;
	float2 uv : TEXCOORD0;
//	float3 localPosition : TEXCOORD0;
};

Interpolators MyVertexProgram (VertedData v) {
	Interpolators i;
//	i.localPosition = v.position.xyz;
	i.position = mul(UNITY_MATRIX_MVP, v.position);
	i.uv = v.uv;
	return i;
}
```

Мы можем сделать UV-координаты видимыми, так же как и локальное положение, интерпретируя их как цветовые каналы. Например, U становится красным, V - зеленым, а синий - всегда 1.

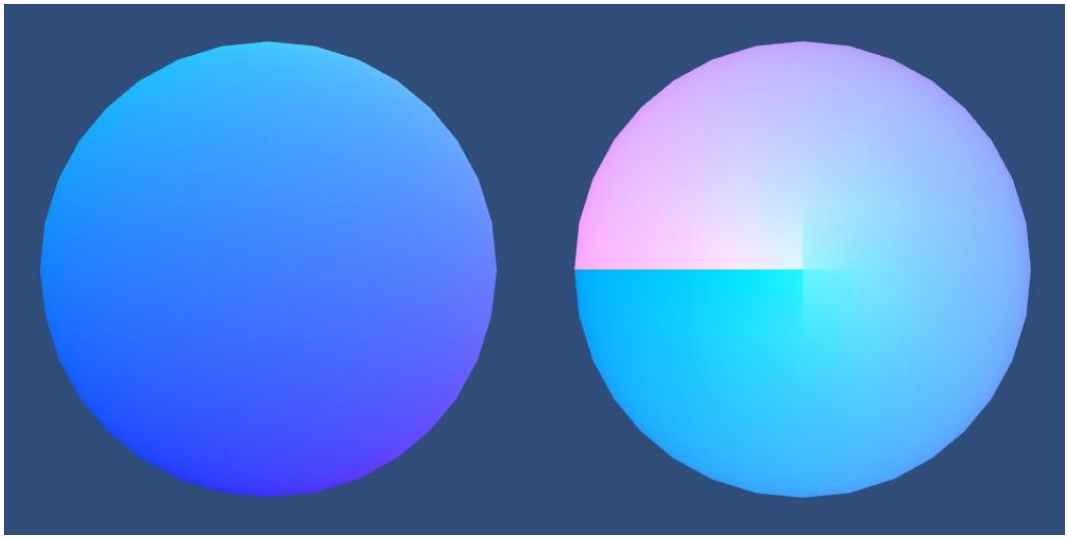
```
float4 MyFragmentProgram (Interpolators i) : SV_TARGET {
	return float4(i.uv, 1, 1);
}
```

Вы увидите, что скомпилированные вершинные программы теперь копируют UV-координаты из данных вершин в выходные данные интерполятора.

```
in  vec4  in_POSITION0;
in  vec2  in_TEXCOORD0;
out vec2 vs_TEXCOORD0;
vec4 t0;
void main()
{
    t0 = in_POSITION0.yyyy * glstate_matrix_mvp[1];
    t0 = glstate_matrix_mvp[0] * in_POSITION0.xxxx + t0;
    t0 = glstate_matrix_mvp[2] * in_POSITION0.zzzz + t0;
    gl_Position = glstate_matrix_mvp[3] * in_POSITION0.wwww + t0;
    vs_TEXCOORD0.xy = in_TEXCOORD0.xy;
    return;
}
```

Unity обертывает координаты UV вокруг своей сферы, сворачивая верхнюю и нижнюю части изображения на полюсах. Вы увидите, как шов идет с севера на южный полюс, где соединяются левая и правая стороны изображения. Таким образом, вдоль этого шва вы получите значения координат U как 0, так и 1. Для этого вдоль шва будут дублироваться вершины, которые будут идентичны, за исключением их координат U.

```
Bind "vertex" Vertex
Bind "texcoord" TexCoord0
ConstBuffer "UnityPerDraw" 352
Matrix 0 [glstate_matrix_mvp]
BindCB  "UnityPerDraw" 0
      vs_4_0
      dcl_constantbuffer cb0[4], immediateIndexed
      dcl_input v0.xyzw
      dcl_input v1.xy
      dcl_output_siv o0.xyzw, position
      dcl_output o1.xy
      dcl_temps 1
   0: mul r0.xyzw, v0.yyyy, cb0[1].xyzw
   1: mad r0.xyzw, cb0[0].xyzw, v0.xxxx, r0.xyzw
   2: mad r0.xyzw, cb0[2].xyzw, v0.zzzz, r0.xyzw
   3: mad o0.xyzw, cb0[3].xyzw, v0.wwww, r0.xyzw
   4: mov o1.xy, v1.xyxx
   5: ret
```



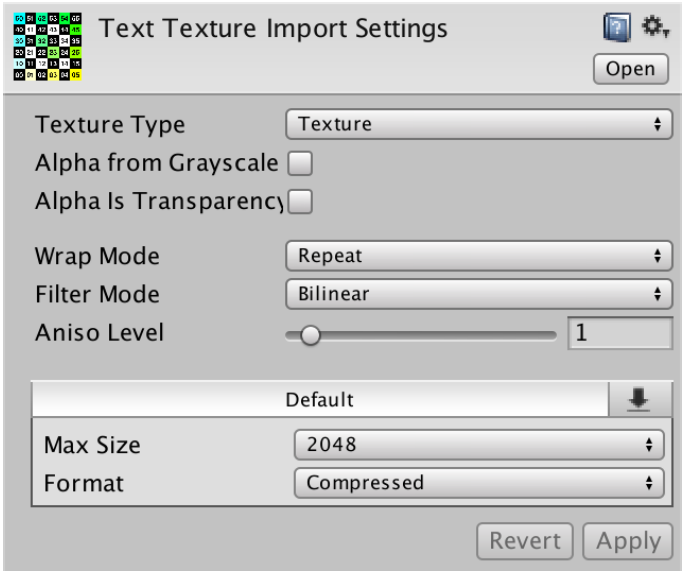
UV представление в виде цвета

4.2 Добавление текстуры

Чтобы добавить текстуру, необходимо импортировать файл изображения. Например, вот такую.

Текстура для тестов

Вы можете добавить изображение в проект, перетащив его в представление проекта. Вы также можете сделать это через пункт меню *Asset / Import New Asset...* Изображение будет импортировано как 2D текстура с настройками по умолчанию.



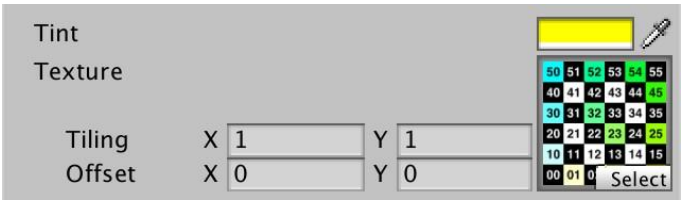
Импортированная текстура с настройками по умолчанию.

Чтобы использовать текстуру, нужно добавить еще одно свойство шейдера. Тип обычного свойства текстуры - 2D, так как существуют и другие типы текстур. Значением по умолчанию является строка, ссылающаяся на одну из текстур Unity по умолчанию - белую, черную или серую.

Конвенция предусматривает название основной текстуры как _MainTex, так что мы будем использовать его. Это также позволяет использовать удобное свойство Material.mainTexture для доступа к нему через скрипт, в случае необходимости.

```
    Properties {
        _Tint ("Tint", Color) = (1, 1, 1, 1)
        _MainTex ("Texture", 2D) = "white" {}
    }
```

Now we can assign the texture to our material, either by dragging or via the *Select* button.



Текстура, назначенная на материал.

Мы можем получить доступ к текстуре в нашем шейдере, используя переменную с типом sampler2D.

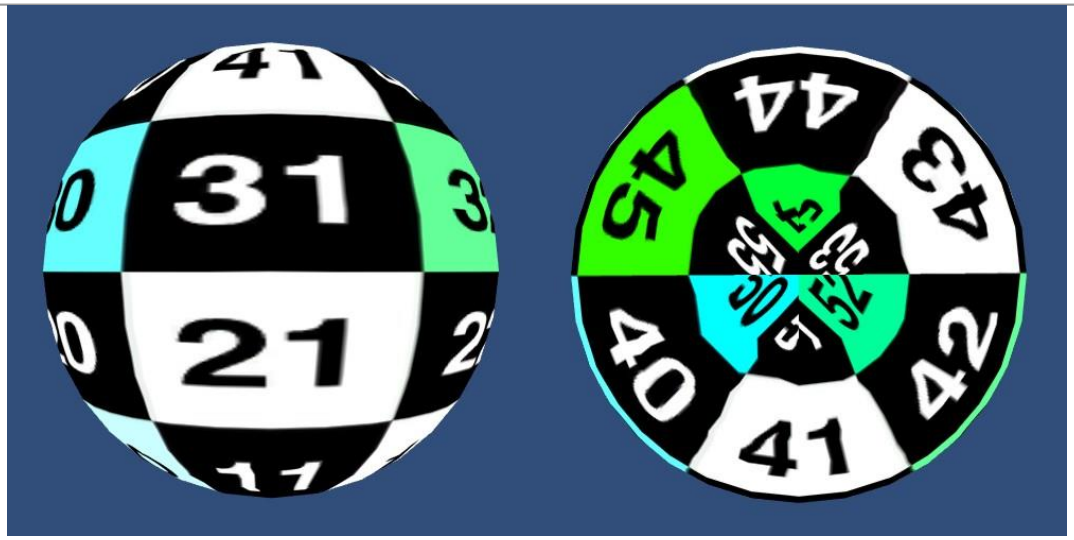
```
            float4 _Tint;
            sampler2D _MainTex;
```

Примерка текстуры с UV координатами производится во фрагментной программе с помощью функции tex2D.

```
            float4 MyFragmentProgram (Interpolators i) : SV_TARGET {
                return tex2D(_MainTex, i.uv);
            }
```

```
uniform  sampler2D _MainTex;
in  vec2 vs_TEXCOORD0;
layout(location = 0) out vec4 SV_TARGET0;
void main()
{
    SV_TARGET0 = texture(_MainTex, vs_TEXCOORD0.xy);
    return;
}
```

```
SetTexture 0 [_MainTex] 2D 0
      ps_4_0
      dcl_sampler s0, mode_default
      dcl_resource_texture2d (float,float,float,float) t0
      dcl_input_ps linear v0.xy
      dcl_output o0.xyzw
   0: sample o0.xyzw, v0.xyxx, t0.xyzw, s0
   1: ret
```



Текстурированная сфера

Теперь, когда текстура отображается для каждого фрагмента, она будет проецироваться на сферу. Как и ожидалось, она обернута вокруг него, но вблизи полюсов она будет выглядеть довольно странной. Почему это так?

Искажение текстуры происходит потому, что интерполяция линейна относительно треугольников. Сфера Unity имеет только несколько треугольников вблизи полюсов, где больше всего искажаются координаты UV. Таким образом, UV-координаты изменяются нелинейно от вершины к вершине, но между вершинами их изменение является линейным. В результате прямые линии в текстуре внезапно меняют направление на границах треугольников.

Линейная интерполяция относительно треугольников

Различные сетки имеют разные координаты UV, что создает разные изображения. По умолчанию сфера Unity использует текстурное отображение продольных и поперечных широт, в то время как сетка является кубической сферой низкого разрешения. Этого достаточно для тестирования, но для лучшего результата лучше использовать пользовательскую сферу mesh.

Наконец, мы можем учитывать цветовой оттенок для регулировки текстурированного внешнего вида сферы.

```
                return tex2D(_MainTex, i.uv) * _Tint;
```

Текстурированная сфера с желтым оттенком

4.3 Tiling и Offset

После того, как мы добавили свойство текстуры в наш шейдер, инспектор материалов не просто добавил поле текстуры. Он также добавил элементы управления покрытием (Tiling) и смещением (Offset). Однако, изменение этих 2D векторов в настоящее время не имеет никакого эффекта.

Эти дополнительные данные о текстуре хранятся в материале и также могут быть получены шейдером. Вы можете сделать это с помощью переменной, которая имеет то же имя, что и связанный с ней материал, плюс суффикс _ST. Тип этой переменной должен быть float4.

Что означает _ST?

Суффикс _ST обозначает Scale and Translation. Почему _TO не используется, имея в виду Tiling and Offset? Потому что Unity всегда использовал _ST, и обратная совместимость мандатов остается такой, даже если терминология могла измениться.

```
sampler2D _MainTex;
float4 _MainTex_ST;
```

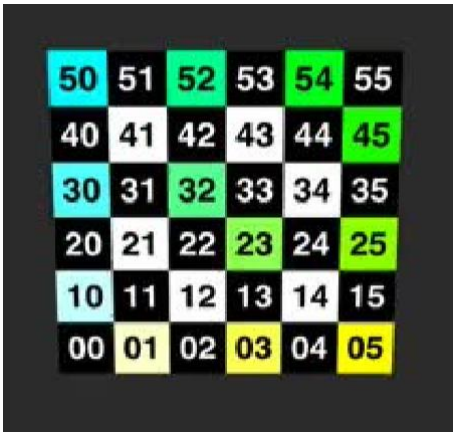
Вектор tiling используется для масштабирования текстуры, поэтому по умолчанию он (1, 1). Он хранится в XY части переменной. Чтобы его использовать, просто умножьте его на координаты UV. Это можно сделать как в вершинном шейдере, так и в шейдере фрагментов. Это имеет смысл делать в вершинном шейдере, поэтому мы выполняем умножения только для каждой вершины, а не для каждого фрагмента.

```
Interpolators MyVertexProgram (VertexData v) {
	Interpolators i;
	i.position = mul(UNITY_MATRIX_MVP, v.position);
	i.uv = v.uv * _MainTex_ST.xy;
	return i;
}
```

Tiling.

The offset portion moves the texture around and is stored in the ZW portion of the variable. It is added to the UV after scaling.

```
i.uv = v.uv * _MainTex_ST.xy + _MainTex_ST.zw;
```



Offset.

UnityCGG.cginc содержит удобный макрос, который упрощает этот шаблон для нас. Мы можем использовать его в качестве удобного сокращения.

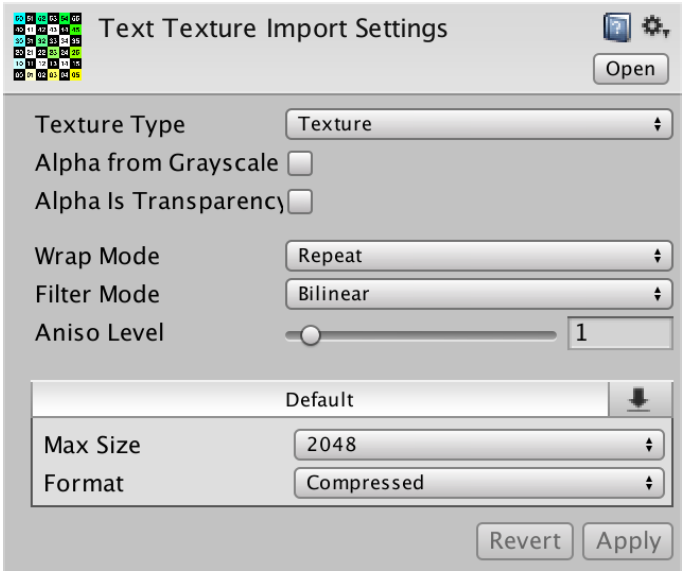
```
i.uv = TRANSFORM_TEX(v.uv, _MainTex);
```

Что такое макрос?

Макрос похож на функцию, за исключением того, что он вычисляется на этапе препроцессинга, до того, как код скомпилирован по-настоящему. Это позволяет оперировать кодом в текстовом виде, например, добавлять _ST к имени переменной.

5 Настройки текстур

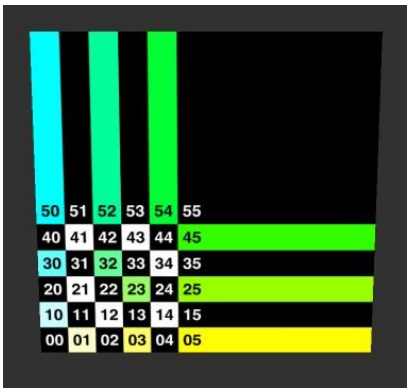
До сих пор мы использовали настройки импорта текстур по умолчанию. Давайте посмотрим, что они делают.



Настройки текстуры по умолчанию

Режим Wrap Mode определяет, что происходит при обработке данных с UV-координатами, которые лежат за пределами диапазона 0-1. Когда режим обертки установлен на clamped, UV ограничивается тем, что остается внутри диапазона 0-1. Это означает, что пиксели, находящиеся за пределами края, такие же, как и пиксели, лежащие на краю. Когда режим обертывания настроен на repeat, UV оборачивается вокруг объекта. Это означает, что пиксели за краем те же самые, что и на противоположной стороне текстуры. Режим по умолчанию - повторить текстуру, в результате чего она становится «плиточной» (tile).

Если у вас нет tiling текстуры, то вместо нее нужно применить clamp UV-координаты. Это предотвратит повторение текстуры, вместо этого граница текстуры будет реплицирована, в результате чего она будет выглядеть растянутой.



Tiling (2, 2) при clamped режиме

5.1 Mipmaps и Filtering

Что происходит, когда пиксели текстуры - тексели - не совсем совпадают с пикселями, на которые они проецируются? Существует несовпадение, которое должно быть как-то устранено. Как это сделать контролируется режимом фильтрации.

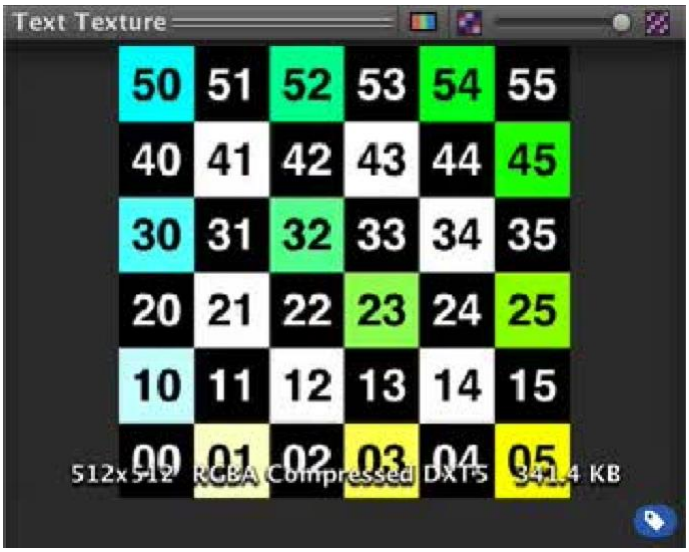
Самый простой режим фильтрации - Точка (без фильтра) (Point (no filter)). Это означает, что когда текстура отображается в каких-то UV-координатах, используется ближайший тексель. Это придаст текстуре блочный вид, если только текстуры не будут точно отображать пиксели. Поэтому он обычно используется для точного рендеринга пикселей, или когда требуется блочный стиль.

По умолчанию используется билинейная фильтрация. Когда текстура отображается где-то между двумя текселями, эти два текселя интерполируются. Так как текстуры 2D, это происходит как по оси U, так и по оси V. Следовательно, используется билинейная фильтрация, а не просто линейная фильтрация.

Этот подход работает, когда плотность текселей меньше плотности пикселей дисплея, а значит, когда вы приближаетесь к текстуре. Результат будет выглядеть размытым. В противоположном случае, когда вы приближаетесь к текстуре, это не срабатывает. Соседние пиксели дисплея получат сэмплы, расположенные на расстоянии более одного текселя друг от друга. Это означает, что части текстуры будут пропущены, что вызовет резкие переходы, как если бы изображение было резким.

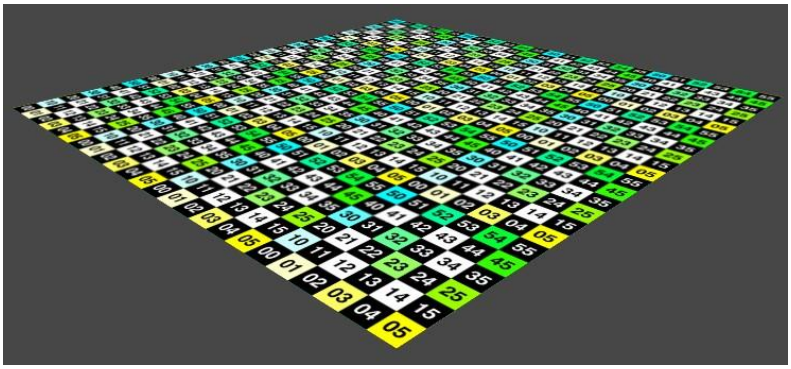
Решением этой проблемы является использование текстуры меньшего размера всякий раз, когда плотность текстуры становится слишком высокой. Чем меньше текстура

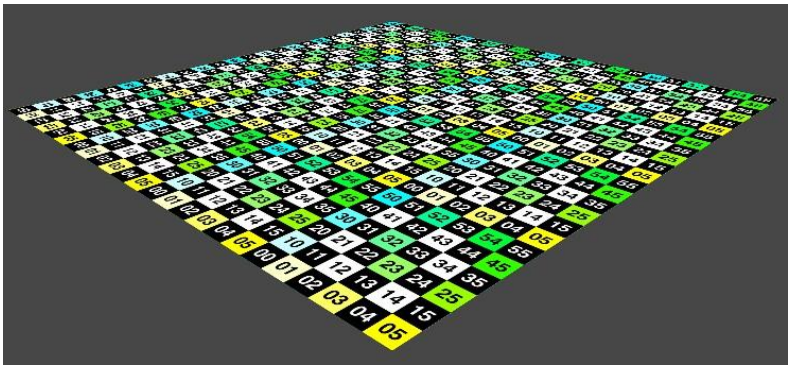
отображается на дисплее, тем меньшая версия текстуры должна быть использована. Эти более маленькие версии известны как mipmaps и автоматически генерируются для вас. Каждая последующая мип-карта имеет в два раза меньшую ширину и высоту, чем предыдущий уровень. Таким образом, когда исходная текстура имеет размер 512x512, mip-карта имеет размер 256x256, 128x128, 64x64, 32x32, 16x16, 8x8, 4x4 и 2x2.



Mipmap уровни

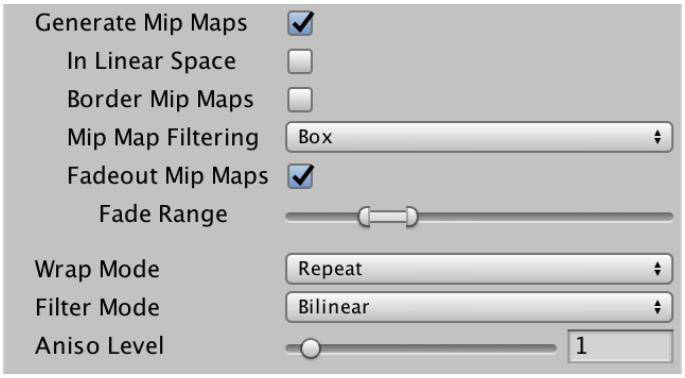
Вы можете отключить mipmaps, если хотите. Для начала установите для типа "Тип текстуры" значение "Дополнительно". Вы можете отключить mipmaps и применить изменение. Хороший способ увидеть разницу - использовать плоский объект, как квадрат, и смотреть на него под углом.





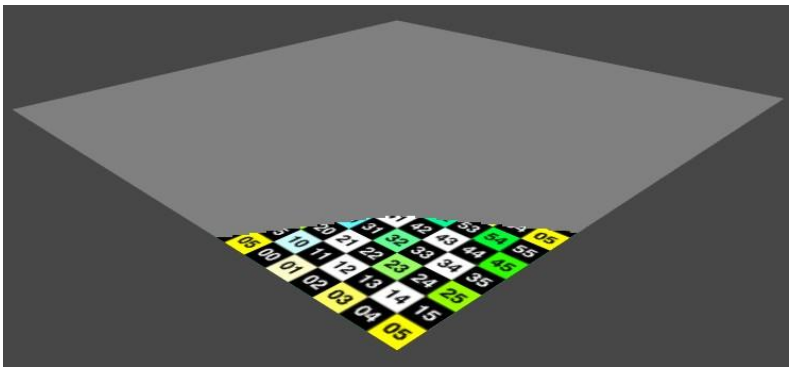
Разница с включённым и выключенным Mipmap.

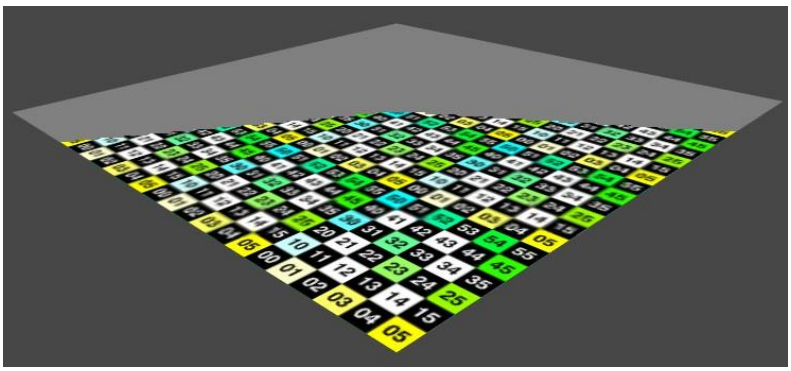
Так какой уровень мип-карты используется где, и насколько они отличаются друг от друга? Мы можем сделать переходы видимыми, включив функцию Fadeout Mip Maps в расширенных настройках текстуры. При включении этого параметра в инспекторе появится ползунок Диапазон затухания (Fade Range). Он определяет диапазон мип-карт, через который мип-карта будет переходить в сплошной серый цвет. Сделав этот переход одним шагом, вы получите резкий переход в серый цвет. Чем дальше вы будете двигать на один шаг вправо, тем позже произойдёт переход.

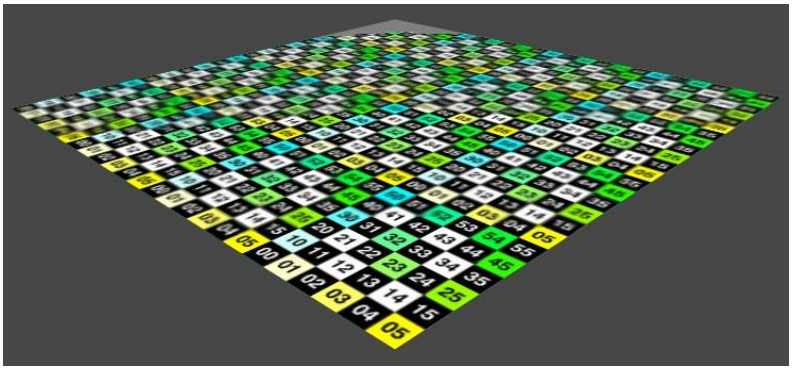


Расширенные настройки mipmaps.

Чтобы получить хорошее изображение этого эффекта, установите пока уровень текстуры Aniso равным 0.







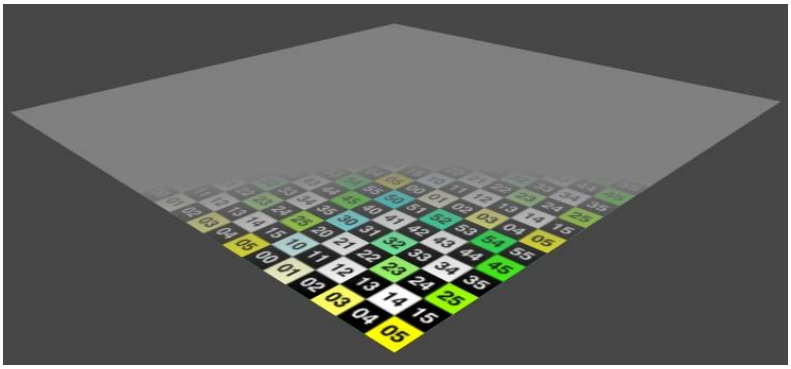
Последовательные уровни mipmap.

Как только вы узнаете, где находятся различные уровни mipmaps, вы сможете увидеть внезапное изменение качества текстуры между ними. По мере уменьшения проекции текстуры увеличивается плотность текстуры, что делает ее более резкой. Пока внезапно не начинает действовать следующий уровень мип-карт, и он снова становится размытым.

Так что без mipmap вы переходите от размытости, к чрезмерной резкости. С mipmaps вы переходите от размытости к резкости, к неожиданной размытости снова, к резкости, к неожиданной размытости снова, и так далее.

Эти размыто-острые полосы характерны для билинейной фильтрации. Вы можете избавиться от них, переключив режим фильтрации на трилинейный. Это работает так же, как и билинейная фильтрация, но также интерполирует между соседними уровнями мип-карты. Отсюда и трилинейный.

Это делает дискретизацию более затратной, но при этом сглаживает переходы между уровнями мип-карты.



Трилинейная фильтрация

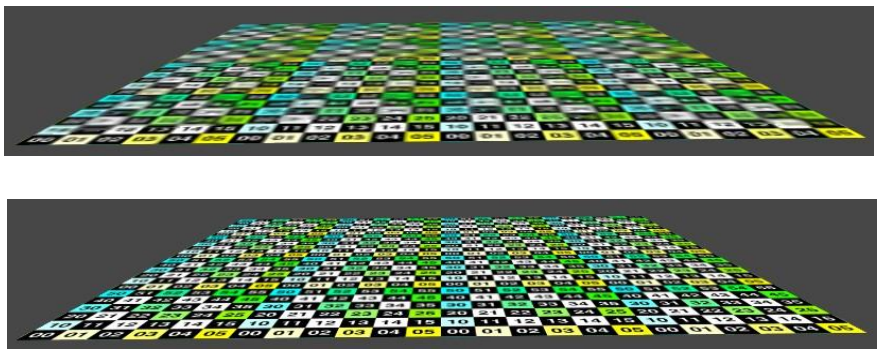
Другой полезный метод - анизотропная фильтрация. Вы могли заметить, что при установке значения 0 текстура стала более расплывчатой. Это связано с выбором уровня mipmap.

Что означает «анизотропная»?

Грубо говоря, когда что-то кажется похожим с разных сторон, то это изотропно. Например, куб без каких либо приметных свойств. В противном случае объект анизотропен. Например, кусок дерева, потому что его текстура направлена в одну конкретную сторону.

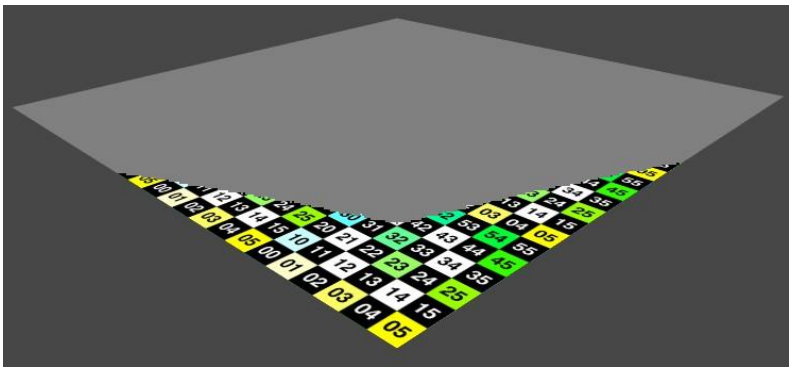
Когда текстура проецируется под углом, из-за перспективы, вы часто получаете одно из ее измерений, искаженное намного больше, чем другое. Хорошим примером является текстурированная плоскость земли. На расстоянии размерность текстуры, ориентированная вперед-назад, будет выглядеть намного меньше, чем размерность, ориентированная влево-право.

Какой уровень mipmap будет выбран, основан на наихудшем измерении. Если разница большая, то в одном измерении получится очень размытый результат. Анизотропная фильтрация смягчает это за счет разделения измерений. Кроме равномерного масштабирования текстуры, она также предоставляет версии, которые масштабируются в разных размерах. Так что у вас есть mipmap не только для 256x256, но и для 256x128, 256x64 и так далее.



Без и с использованием анизотропной фильтрации

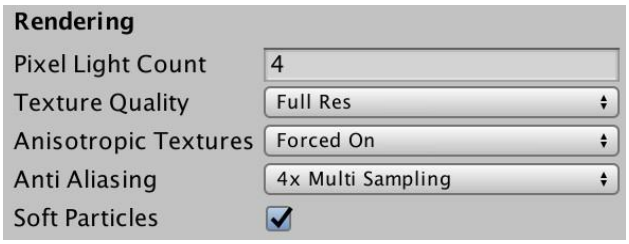
Обратите внимание, что эти дополнительные mipmaps не генерируются заранее, как обычные mipmaps. Вместо этого они моделируются путем обработки дополнительных текстурных сэмплов. Таким образом, они не требуют больше места, но более затратны для сэмплинга.



Анизотропная билинейная фильтрация, переходящая в серый цвет.

Насколько глубоко проходит анизотропная фильтрация контролируется уровнем Aniso. При 0 он отключен. При 1 он становится включенным и дает минимальный эффект. В 16 - максимум. Однако на эти настройки влияют Quality настройки проекта.

Вы можете их обнаружить по пути *Edit / Project Settings / Quality*. Вы там найдете настройку *Anisotropic Textures* в секции *Rendering*.



Настройки качества рендеринга

Когда анизотропные текстуры отключены, анизотропная фильтрация не будет происходить независимо от настроек текстуры. Когда установлено значение Per Texture, оно полностью контролируется каждой отдельной текстурой. Его также можно установить в значение Forced On, которое будет действовать так, как если бы для каждого уровня анизотропной текстуры было установлено значение не менее 9. Однако текстура с уровнем Aniso, установленным в 0, все равно не будет использовать анизотропную фильтрацию.