МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра дифференциальных уравнений и системного анализа

Моделирование графических эффектов в среде «Unity»

Курсовая работа

Коростелёва Романа
Александровича

студента 2 курса,
специальность 1-31 03 09
Компьютерная математика
и системный анализ

Научный руководитель:
Старший преподаватель, А.В.
Кушнеров

Минск, 2022

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Моделирование и рендеринг реального мира со всеми его неровностями и несовершенства остаются одной из самых больших проблем в компьютерной графике.

Unity — современный игровой движок, использующийся при написании огромного количества новых игр. Как следствие, реалистичное моделирование графических эффектов на нем является актуальной задачей, встающей каждый день над большим количеством разработчиков. С каждой версией Unity предоставляет всё больше инструментов для моделирования графики, а также оптимизирует работу с нею.

В данной работе показывается функционал Unity для моделирования графических эффектов, основы работы с графическими эффектами и моделируется такой оптический эффект как иридисценция.

ГЛАВА 1
ОБЗОР ФУНКЦИОНАЛА СРЕДЫ МОДЕЛИРОВАНИЯ

1.1 Что такое Unity?

Unity – кроссплатформенный игровой движок, разработанный Unity Technologies, впервые представленный в 2005 на WWDC эксклюзивно для MAC OS X. Движок с того времени существенно расширил поддержку различных платформ: Desktop(Windows, Mac, Linux), Mobile(iOS, Android, tvOS), Web(WebGL), Console(PlayStation, Xbox, Nintendo Switch, Stadia), VR(Oculus, PlayStation VR, Google's ARCore, Apple's ARKit, HoloLens, Magic Leap, Steam VR, Google Cardboard). Особенно популярен движок в мобильной разработке и использовался для таких игр как Pokémon Go, Monument Valley, Call of Duty: Mobile, Beat Saber and Cuphead. Движок прост в освоении и пользуется спросом среди инди-разработчиков.

Unity используется для создания двумерных и трехмерных игр, интерактивных симуляторов и других экспериментов. Приложения, созданные с помощью Unity, поддерживают DirectX и OpenGL. Движок активно используется как крупными разработчиками (Blizzard, EA, QuartSoft, Ubisoft), так и разработчиками инди-игр (Pathologic, Kerbal Space Program, Slender: The Eight Pages, Slender: The Arrival, Surgeon Simulator 2013, Baeklyse Apps: Guess the actor и т. д.) благодаря бесплатной версии, удобному интерфейсу и простоте работы.

Кроме разработки игр Unity используется другими отраслями: киноиндустрия, автомобилестроение, архитектура, машиностроение, строительство и вооруженными силами США.

1.2 Возможности игрового движка

Кроме разработки игр Unity используется другими отраслями: киноиндустрия, автомобилестроение, архитектура, машиностроение, строительство и вооруженными силами США. Прежде всего, стоит отметить, что в среду разработки Unity интегрирован игровой движок - другими словами, вы можете протестировать свою игру, не выходя из редактора. Также Unity поддерживает импорт форматов файлов, что позволяет разработчику самостоятельно разрабатывать модели в более удобном для пользователя приложении и использовать Unity по назначению - для разработки продукта.

Редактор Unity имеет простой интерфейс перетаскивания, который прост в настройке, состоящий из разных окон, чтобы вы могли отлаживать игру непосредственно в редакторе. Движок поддерживает два языка сценариев: C#, JavaScript (модификация). Ранее была поддержка Boo (диалекта Python), но он был удален в 5-й версии. Физические расчеты создаются физическим движком NVIDIA PhysX.

Проект в Unity разделен на сцены — отдельные файлы, содержащие ваши игровые миры со своими объектами, сценариями и настройками. Сцены могут содержать реальные объекты (модели) и пустые игровые объекты – объекты, у которых нет модели ("пустышки").[5] Объекты, в свою очередь, содержат наборы компонентов, с которые используются в скриптах. Кроме того, объекты имеют имя (в Unity разрешено использовать два или более объектов с одинаковым именем), может быть Tag(метка) и слой, на котором он должен отображаться. Следовательно, каждый объект в сцене обязательно имеет компонент Transform – он хранит координаты положения, поворота и размера объекта по всем трем осям. Объекты с видимой геометрией по умолчанию также имеют Mesh Renderer, который делает объектную модель видимой.

Объектам можно добавить коллизию, которая в Unity называется Collaider. Существуют следующие типы Collider:

- **Box collider** (физическая модель образует куб)
- **Sphere collider** (физическая модель образует сферу)
- **Capsule collider** (физическая модель образует капсулу)
- **Mesh collider** (физическая модель полностью повторяет реальную геометрию объекта)
- **Wheel collider** (физическая модель колеса)
- **Terrain collider (**тип физической модели, созданный специально для использования на объекте типа Terrain)

Кроме того, Unity поддерживает физику тела и тканей, а также физику тряпичной куклы (Ragdoll). Редактор имеет систему наследования объектов; дочерние объекты повторяют все изменения положения, поворота и масштабирования родительского объекта. Скрипты редактора прикрепляются к объектам как отдельные компоненты.

Unity 3D поддерживает систему Level Of Detail (аббревиатура LOD), суть которой в том, что на большом расстоянии от игрока очень детализированные модели заменяются менее детализированными и наоборот, а также систему Oclussion culling, суть которой в том, что объекты, геометрия и коллизии, которые не находятся в поле зрения камеры, не визуализируются, что снижает нагрузку на процессор и оптимизирует проект.

Редактор Unity поддерживает написание и редактирование шейдеров. В редакторе Unity есть компонент для создания анимации, но анимацию также можно

предварительно создать в 3D-редакторе и импортировать вместе с моделью, а затем разделить на файлы.

Помимо пустого игрового объекта и моделей, на сцену можно добавлять ещё такие объекты типа GameObject:

- **Система частиц**
- **Камера**
- **GUI текст**
- **GUI текстура**
- **3D текст**
- **Точечный свет**
- **Направленный свет**
- **Освещение территории**
- **Источник света, имитирующий солнце**
- **Стандартные примитивы**
- **Деревья**
- **Terrain (земля)**

Движок поддерживает множество популярных форматов, таких как:

- .3ds, .max, .obj, .fbx, .dae, .ma, .mb, .blend для трёхмерных моделей;
- .mp3, .ogg, .aiff, .wav, .mod, .it, .sm3 для звуковых файлов;
- .psd, .jpg, .png, .gif, .bmp, .tga, .tiff, .iff, .pict, .dds для изображений;
- .mov, .avi, .asf, .mpg, .mpeg, .mp4 для видеофайлов.
- .txt, .htm, .html, .xml, .bytes для текста

Модели, звуки, текстуры, материалы, скрипты можно запаковать в формат .unityassets и передать другим разработчикам или выложить в свободный доступ. Тот же формат используется во внутреннем магазине Unity Asset Store, где разработчики могут бесплатно и за деньги делиться различными элементами, необходимыми для создания игр. Чтобы использовать Unity Asset Store, вам необходимо иметь учетную запись разработчика Unity. В Unity есть все компоненты, необходимые для создания мультиплеера. Вы также можете использовать интуитивный метод контроля версий. Например, Tortoise SVN или Source Gear.

ГЛАВА 2
ОСНОВЫ РАБОТЫ С ГРАФИЧЕСКИМИ ЭФФЕКТАМИ В СРЕДЕ «UNITY»

2.1 Что такое шейдер?

Для моделирования графических эффектов мы будем пользоваться специальными программами – шейдерами.[1, с.3] Шейдер можно описать следующим образом:

- симуляция того, что происходит на поверхности на микроскопическом уровне, что делает итоговое изображение реалистичным для наших глаз
- Код, запускающийся на графическом процессоре

Внутри компьютеров у нас нет вычислительной мощности, необходимой для имитации реальности с таким уровнем детализации. Если бы нам пришлось симулировать все это целиком, атомы и все остальное, на визуализацию чего-либо ушли бы годы. В большинстве средств визуализации поверхности представлены в виде 3D-моделей, которые в основном представляют собой точки в 3D-пространстве (вершины) в определенной позиции, которые затем группируются в треугольники, которые затем снова группируются для формирования 3D-формы. Даже простая модель может иметь тысячи вершин. Наша 3D-сцена, состоящая из моделей, текстур и шейдеров, визуализируется в 2D-изображение, состоящее из пикселей. Это делается путем проецирования положений каждой из вершин на правильные положения в плоскости с применением любых текстур к соответствующим поверхностям и выполнением шейдеров для каждой вершины 3D-моделей и каждого потенциального пикселя конечного изображения.

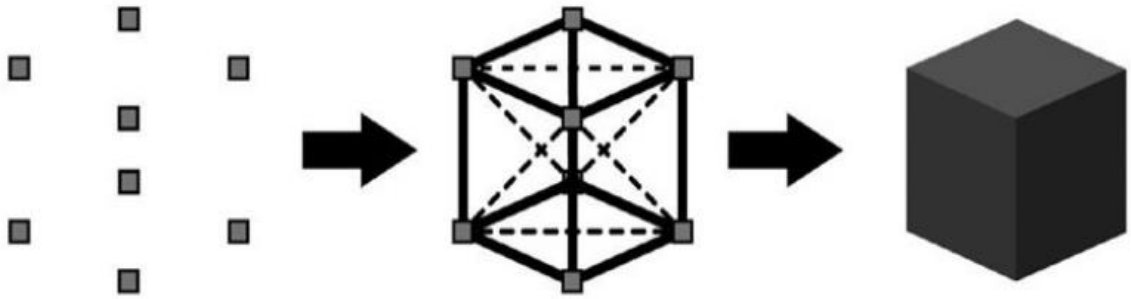

Рисунок 2.1 Процесс обработки изображения

Независимо от того, насколько далеко вы готовы зайти в деталях при моделировании, невозможно соответствовать уровню деталей в реальном мире. Мы можем использовать наши математические знания о том, как работает освещение в шейдере, чтобы наши 3D-модели выглядели максимально реалистично, компенсируя невозможность имитации поверхностей на более высоком уровне детализации. Мы также можем использовать его для достаточно быстрой визуализации наших сцен, чтобы наша игра могла отрисовывать приличное количество кадров в секунду. Оптимальным количеством кадров в секунду в играх считаются значения между 30 и 60.

Вычисления, выполняемые в шейдере освещения записывается следующим образом и называется *уравнение рендеринга*[2]*:*

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t)\, L_i(x, \omega_i, \lambda, t)(\, \omega_i \cdot n) d\, \omega_i$$

где:

- λ – длина волны света
- t – время
- ω_o – направление выходящего света
- ω_i– отрицательное направление входящего света

• $L_o(x, \omega_o, \lambda, t)$ – итоговое излучение заданной длины волны λ исходящего вдоль направления ω_o во время t, из заданной точки x

• $L_e(x, \omega_o, \lambda, t)$ – испускаемое излучение

• $L_i(x, \omega_i, \lambda, t)$ – входящее излучение

• $\int_\Omega .. d\omega_i$– интеграл по полусфере входящих направлений

• $f_r(x, \omega_i, \omega_o, \lambda, t)$ — двунаправленная функция распределения отражения, количество излучения отраженного от i к o в точке x, во время t, при длине волны

• n – поверхностная нормаль к x

• $\omega_i \cdot n$ – поглощение входящего излучения по заданному углу, часто записывается как cos(Θ)

2.2 Создание шейдера

Пускай у нас имеется сцена со сферой в Unity. Мы напишем свой шейдер и применить его к этой сфере. Для примера сделаем шейдер, красящий нашу сферу зеленый цвет.

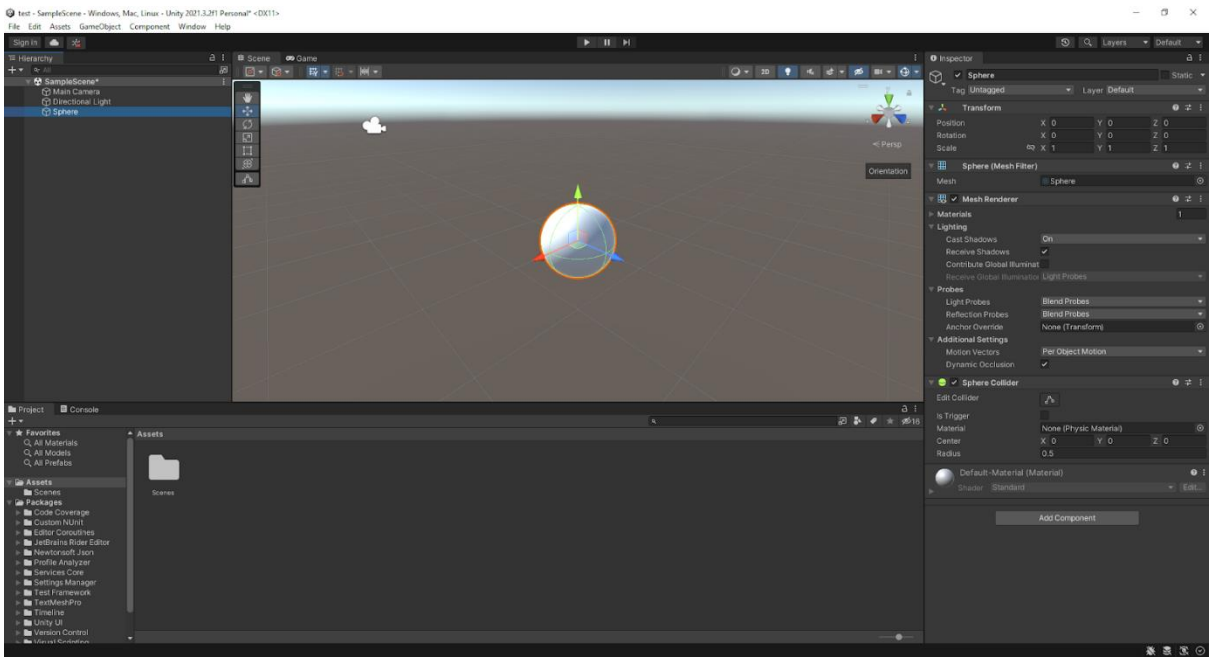

Рисунок 2.2 Сцена со сферой

Теперь нам нужно создать материал, к которому мы привяжем шейдер. Материал – это файл, который содержит все настройки относительно поверхности, то есть всё то, что шейдер использует для дальнейшего рендеринга: все цвета, текстуры и остальная информация. Давайте создадим новый материал.

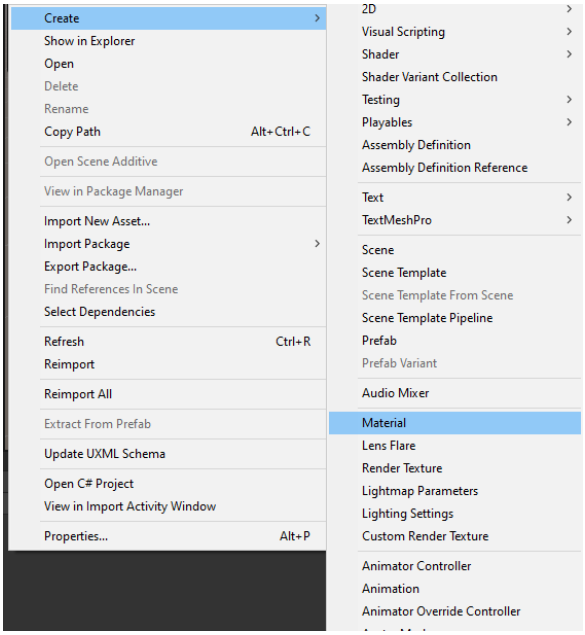

Рисунок 2.3 Создаем материал

После этого необходимо создать сам шейдер.

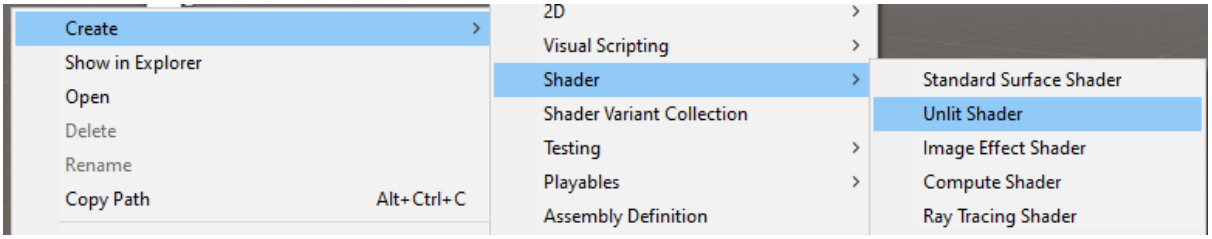

Рисунок 2.4 Создаем шейдер

Теперь нужно привязать шейдер к материалу и применить материал на нашу сферу.

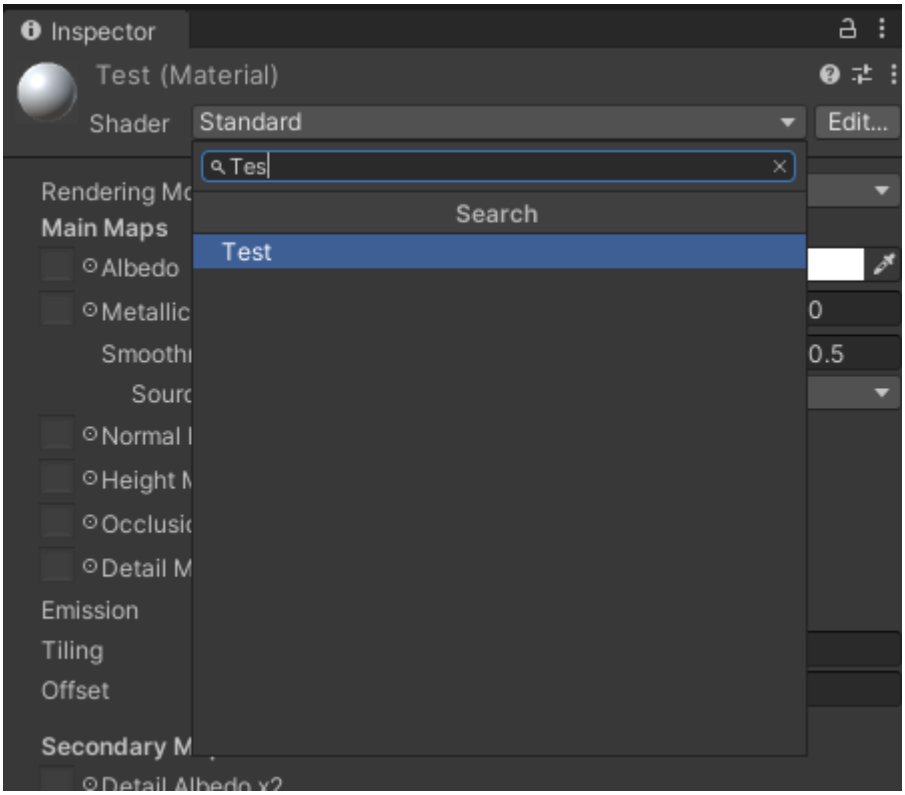

Рисунок 2.3 Привязка шейдера

После этого перетаскиваем наш материал на сферу и применяем. Готово! Сфера стала полностью белой без теней. Это произошло потому, что мы выбрали шейдер, не учитывающий освещение. Давайте теперь посмотрим на сам код шейдера и попробуем его изменить.

```
Shader "Unlit/Test"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                UNITY_FOG_COORDS(1)
                float4 vertex : SV_POSITION;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
```

```
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                UNITY_TRANSFER_FOG(o,o.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // sample the texture
                fixed4 col = tex2D(_MainTex, i.uv);
                // apply fog
                UNITY_APPLY_FOG(i.fogCoord, col);
                return col;
            }
            ENDCG
        }
    }
}
```

Код шейдера пишется на языке Cg/HLSL, написанном на C. Нас интересует функция ***frag***: именно она отвечает за отрисовку фрагментов на физической модели. Давайте уберем все расчеты в функции и будем фиксировано возвращать зеленый цвет.

```
fixed4 frag (v2f i): SV_Target
            {
                return fixed4(0,1,0,1);
            }
```

Как можем видеть, сфера действительно стала зеленой, со своей задачей мы справились.

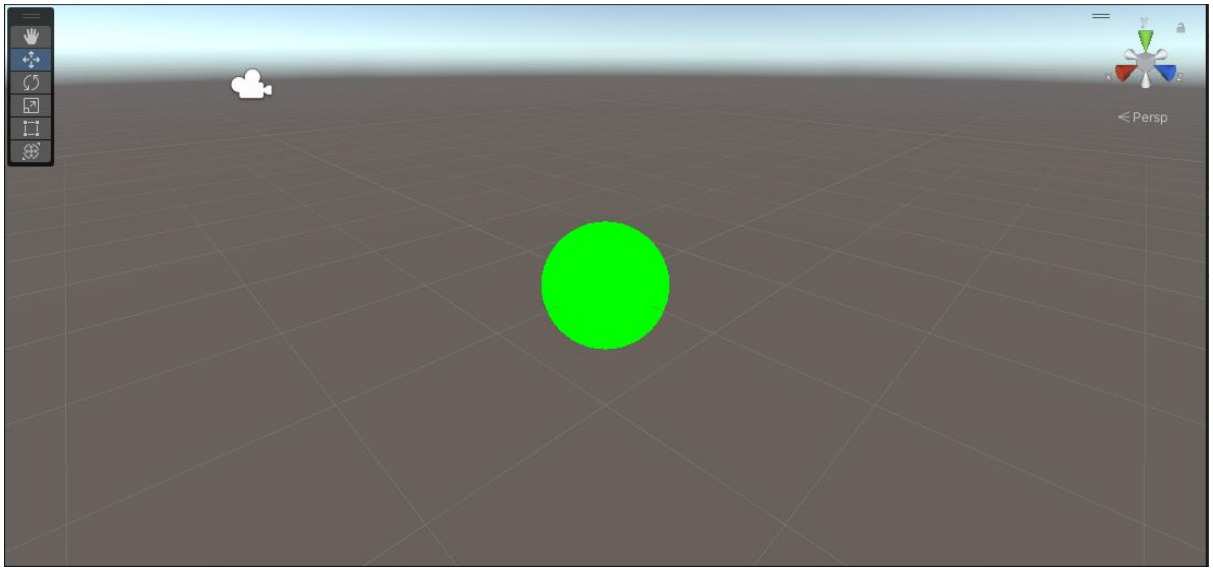

Рисунок 2.4 Сфера с шейдером

ГЛАВА 3
МОДЕЛИРОВАНИЕ ИРИДИСЦЕНЦИИ

3.1 Что такое иридисценция?

Иридисценция – оптическое явление, которое характеризуется изменением цвета в зависимости от угла освещения и обзора. Она проявляется на ряде поверхностей: лужа бензина, чешуя жука, мыльные пузыри, CD диски. Так происходит потому, что иридисценция возникает вследствие взаимодействия света и микроскопических структур, которые находятся на поверхностях всех этих объектов. И дорожки имеют тот же порядок величины длины волн света, с которым они взаимодействуют.

Рисунок 3.1 Пример иридисценции

3.2 Вывод формулы

Давайте выведем уравнение. Пускай у нас есть материал, имеющий неоднородности с периодом *d*. Угол между падающим лучом и нормалью поверхности обозначим за Θ_L. Возьмем также, что наблюдатель получает лучи под этим же углом. Каждая неоднородность рассеивает свет во всех направлениях, поэтому всегда будут присутствовать лучи света, падающие на наблюдателя, вне зависимости от Θ_L.

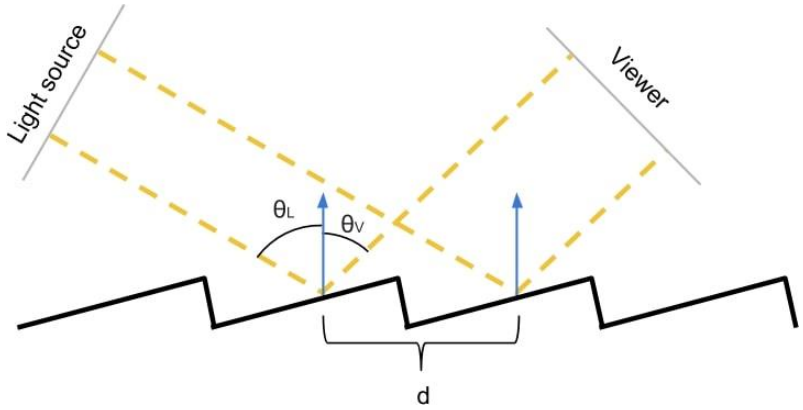

Рисунок 3.2 Падение лучей на поверхность

Два луча света, изображенные выше, проходят разное расстояние до наблюдателя. Для того, чтобы понять, усиливают или гасят лучи друг друга, необходимо вычислить, насколько они не совпадают по фазе.

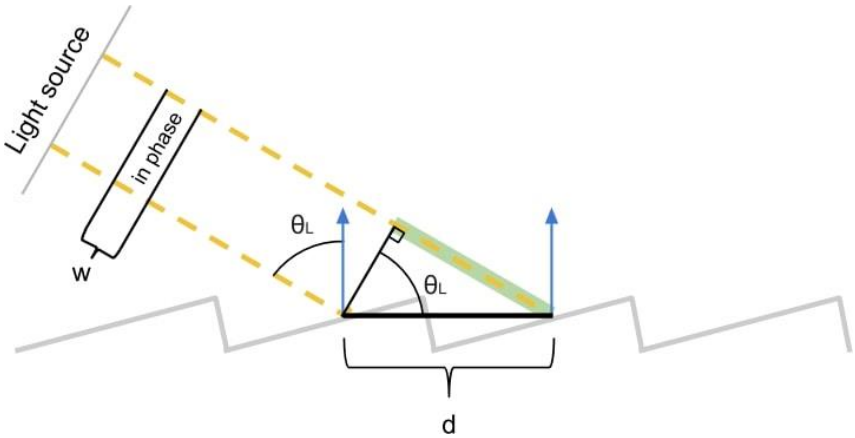

Рисунок 3.3 Лучи до падения на поверхность

Эти два луча будут находиться точно в фазе, пока первый из них не упадет на поверхность. Второй луч проходит дополнительное расстояние x (выделено зеленым), после чего также падает на поверхность. Несложно подсчитать, что $x = d \cdot \sin \Theta_L$.

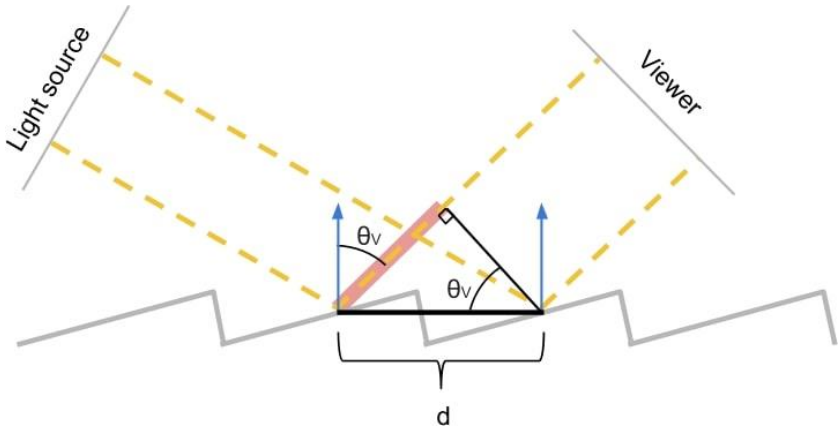

Рисунок 3.4 Лучи после падения на поверхность

Аналогично можно рассчитать дополнительное расстояние y, которое проходит первый луч, пока второй не столкнется с поверхностью. Теперь $y = d \cdot \sin \Theta_V$.

Если разность этих длин равна нулю или кратна длине волны ω, то они совпадают по фазе. Следовательно условие совпадения лучей по фазе можно записать следующим образом:

$$d \cdot \sin \Theta_L + d \cdot \sin \Theta_V = n \cdot \omega$$

$$\sin \Theta_L + \sin \Theta_V = \frac{n \cdot \omega}{d}$$

3.3 Преобразование волны в цвет

Для полноценного моделирования иридесценция нам нужно воспроизвести цвет из длины. Если мы построим график для каждой длины волны соответствующие им компоненты R, G и B, мы получим что-то вроде этого:

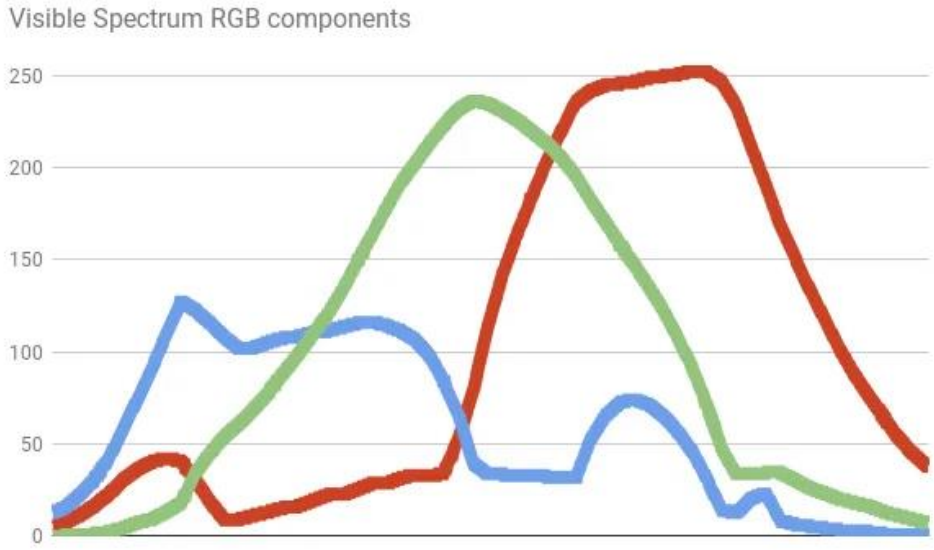

Рисунок 3.5 Видимый RGB спектр

Для реализации своей цветовой схемы я буду использовать параболы вида $y = 1 - x^2$. [3, с. 127]А точнее, $bump(x) = \begin{cases} |x| > 1 & 0 \\ else & 1 - x^2 \end{cases}$. Выглядеть она будет следующим образом:

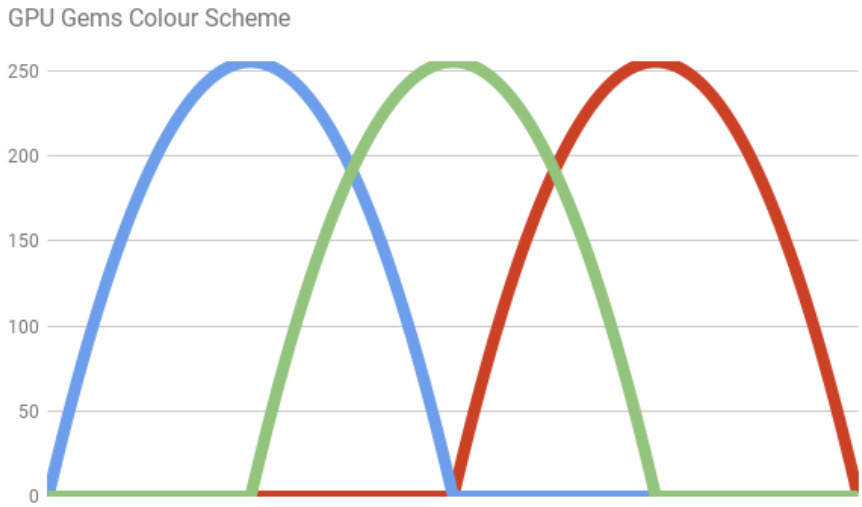

Рисунок 3.6 Цветовая схема GPU Gems

Основными преимуществами данной цветовой схемы является простота и скорость.

Реализуется она следующим образом:

```
fixed3 bump3 (fixed3 x)
{
 float3 y = 1 - x * x;
 y = max(y, 0);
 return y;
}

fixed3 spectral(float w)
{
 // w: [400, 700]
 // x: [0,   1]
 fixed x = saturate((w - 400.0)/300.0);

 return bump3( fixed3(
 4 * (x - 0.75), // Красный
 4 * (x - 0.5), // Зеленый
 4 * (x - 0.25) // Синий
 )
 );}
```

3.4 Реализация шейдера

Остается только написать шейдер. Для наших целей подойдет *Standart Surface Shader*. Нашему шейдеру понадобится новое свойство – расстояние d, используемое в уравнении дифракционной решетки. Добавим его в блок *Properties*.

```
Properties
{
 _Color ("Color", Color) = (1,1,1,1)
 _MainTex ("Albedo (RGB)", 2D) = "white" {}
 _Glossiness ("Smoothness", Range(0,1)) = 0.5
 _Metallic ("Metallic", Range(0,1)) = 0.0

 _Distance ("Grating distance", Range(0,10000)) = 1600 // nm
}
```

Нужно заменить функцию освещения шейдера на свою

```
#pragma surface surf Diffraction fullforwardshadows
```

Теперь добавим функцию обработки глобального освещения.

```
void LightingDiffraction_GI(SurfaceOutputStandard s, UnityGIInput data, inout
UnityGI gi)
{
 LightingStandard_GI(s, data, gi);
}
```

Для каждого пикселя значения Θ_L (определяемого *направлением света*), Θ_v (определяемого *направлением обзора*) и d(*расстояние между зазорами*) известны. Неизвестными переменными являются w и n. Проще всего будет перебрать в цикле значения, чтобы увидеть, какие длины волн удовлетворяют уравнению решётки. Когда мы будем знать, какие длины волн вносят вклад в конечное иридисцентное отражение, то мы вычислим соответствующие им цвета и сложим их.

```
inline fixed4 LightingDiffraction(SurfaceOutputStandard s,
fixed3 viewDir, UnityGI gi)
{
 // Исходный цвет
 fixed4 pbr = LightingStandard(s, viewDir, gi);


 // Вычисляет цвет отражения
 fixed3 color = 0;
 for (int n = 1; n <= 10; n++)
 {
 float d = _Distance;
 float sin_thetaL = gi.light.dir;
 float sin_thetaV = viewDir;


 float wavelength = abs(sin_thetaL - sin_thetaV) * d / n;
 color += spectral(wavelength);
 }
 color = saturate(color);


 // Прибавляет цвет отражения к цвету материала
 pbr.rgb += color;
```

```
 return pbr;
}
```

В данном случае мы берем n = 10, но чем больше n, тем точнее получаемое изображение. В результате мы получаем готовый графический эффект.

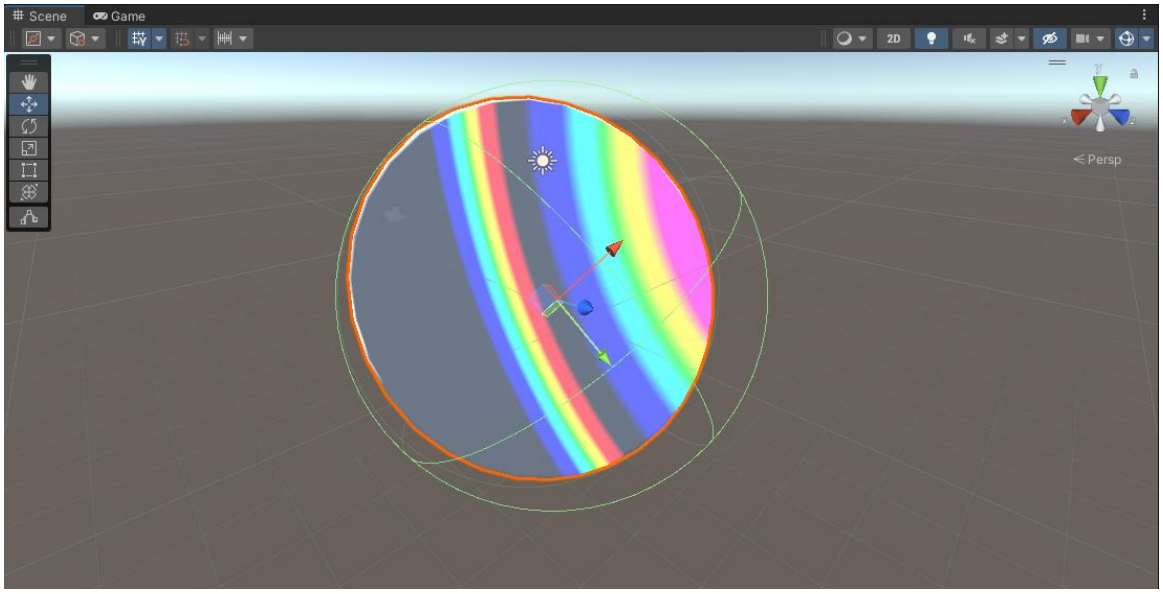

Рисунок 3.7 Иридисценция на диске

ЗАКЛЮЧЕНИЕ

Unity является мощным инструментом моделирования, предоставляющий широкий спектр возможностей и готовых решений.

В ходе данной работы были рассмотрены различные возможности движка для моделирования графических эффектов, разобраны основы создания шейдеров, а также реализован шейдер для иридисценции.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Claudia Doppioslash, Physically Based Shader Development for Unity 2017, 2018 — 242 с..
2. Kajiya, James T., The rendering equation, *1986* — 143с.
3. Randima Fernando, GPU gems, 2004 — 784 с.
4. John P. Doran, Alan Zucconi, Unity 2018 Shaders and Effects Cookbook 3thd edition, 2018 — 534 с.
5. Unity documentation[Электронный ресурс] – Режим доступа: https://docs.unity3d.com/Manual/index.html – Дата доступа: 16.05.2022.

ПРИЛОЖЕНИЕ А.

Ниже приведен код шейдера для иридисценции.

```
Shader "Custom/Iridence"
{
       Properties
  {
 _Color ("Color", Color) = (1,1,1,1) // Цвет
 _MainTex ("Albedo (RGB)", 2D) = "white" {} // Текстура
 _Glossiness ("Smoothness", Range(0,1)) = 0.5
 _Metallic ("Metallic", Range(0,1)) = 0.0
 _Distance ("Grating distance", Range(0,10000)) = 1600 //
Дистанция между волнам
}
        Subshader
{
    Tags { "RenderType"="Opaque" }
    CGPROGRAM
    #pragma surface surf Diffraction fullforwardshadows
    #include "UnityPBSLighting.cginc"
    float _Distance;
    sampler2D _MainTex, _Normal;
    fixed4 _Color;
    float3 worldTangent;

    void LightingDiffraction_GI(SurfaceOutputStandard s,
UnityGIInput data, inout UnityGI gi)
    {
        LightingStandard_GI(s, data, gi);
    }

    struct Input
     {
         float2 uv_Maintex;
         float2 uv_Normal;
     };
```

```
        void surf(Input IN, inout SurfaceOutputStandard o)
        {
            fixed4 c = tex2D(_MainTex, IN.uv_Maintex) * _Color;
            o.Albedo = c.rgb;
            o.Normal = UnpackNormal(tex2D(_Normal, IN.uv_Normal));
        }

      inline fixed3 bump3 (fixed3 x)
{
 float3 y = 1 - x * x;
 y = max(y, 0);
 return y;
}

fixed3 spectral(float w)
{
 // w: [400, 700]
 // x: [0,   1]
 fixed x = saturate((w - 400.0)/300.0);

 return bump3( fixed3(
 4 * (x - 0.75), // Red
 4 * (x - 0.5), // Green
 4 * (x - 0.25) // Blue
 )
 );}

    inline fixed4 LightingDiffraction(SurfaceOutputStandard s,
fixed3 viewDir, UnityGI gi)
{
 // Исходный цвет
 fixed4 pbr = LightingStandard(s, viewDir, gi);


 // Вычисляет цвет отражения
 fixed3 color = 0;
 for (int n = 1; n <= 50; n++)
 {
 float d = _Distance;
```

```
 float sin_thetaL = gi.light.dir;
 float sin_thetaV = viewDir;

 float wavelength = abs(sin_thetaL - sin_thetaV) * d / n;
 color += spectral(wavelength);
 }
 color = saturate(color);

 // Прибавляет цвет отражения к цвету материала
 pbr.rgb += color;
 return pbr;
}


     ENDCG
}
}
```