МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

КУРСОВОЙ ПРОЕКТ

по дисциплине: Основы программирования

тема: «Разработка графического движка»

Автор работы _____ Пахомов Владислав Андреевич ПВ-223
(подпись)

Руководитель проекта _____ Черников Сергей Викторович
(подпись)

Оценка ____

Белгород 2023 г.

Оглавление

Введение

Большую часть информации человек воспринимает глазами, именно поэтому одним из самых популярных видов контента на сегодняшний день является визуальный контент.

Красочная, яркая и пёстрая или серая, драматичная. За множество лет человечество успело отобразить реальность в графическом формате множество раз в виде картин, фильмов, фотографии.

Компьютеры стали незаменимыми помощниками в создании графического контента. Вычислительные мощности компьютеров, растущие ежегодно, уже позволяют создавать картинку, неотличимую от реальности. Это стало возможно благодаря развитию графических процессоров - отдельному устройству ПК.

Более мощные компьютеры позволяют сегодня использовать более сложные алгоритмы для получения реалистичной картинки, например трассировка лучей и Physically Based Rendering. В последние модели видеокарт добавляются дополнительные ядра, которые способны решать задачи, направленные на рендеринг при помощи данных техник.

Производитель видеокарт поставляет библиотеки, позволяющие работать с этими ядрами. Для видеокарт от компании AMD такой библиотекой является HIP RT, расширяющая библиотеку для работы с видеокартой HIP.

Объект исследования - разработка графического движка.

Предмет исследования - библиотеки для работы с видеокартами от AMD HIP и HIP RT, техники реалистичного рендера трассировка лучей и Physically Based Rendering.

Цель - разработать графический движок, использующий техники трассировка лучей и Physically Based Rendering и аппаратное обеспечение (видеокарту).

Для достижения поставленной цели необходимо решить следующие задачи:

- Изучить техники реалистичного рендера трассировка лучей и Physically Based Rendering.
- Изучить и применить библиотеки для аппаратного ускорения при ис-

пользовании техник реалистичного рендера.

- Подобрать удобный формат хранения информации о 3D-сцене.
- Разработать программу, генерирующую изображение на основе данных о 3D-сцене.

1 Техники реалистичного рендера

1.1 Трассировка лучей

В основе трассировки лучей лежит довольно простая идея. Предположим, нам нужно нарисовать картину, но всё что мы можем сделать - это ставить точки и безошибочно определять цвет, куда мы смотрим. Можно разбить холст на квадраты и методично просматривать каждый из них, определяя цвет и ставя точку соответствующего цвета. Таким образом можно получить картину.

Трассировка лучей работает схожим образом. Из точки наблюдения мы будем испускать луч в соответствующем направлении и определять, в какой цвет окрашивать текущий пиксель.

Точка наблюдения - это координаты камеры. Направление испускаемого луча можно определить по следующей формуле:
x, y - координаты текущего обрабатываемого пикселя,
$AR = \frac{Res_W}{Res_H}$ - соотношение сторон, Res_W - ширина холста, Res_H - высота холста.
$S_H = \frac{2}{1+AR}$, $S_W = 2 - S_H$ - стороны прямоугольника, подобного холсту, причём $S_H + S_W = 2$.
$D = ((x/Res_W) \cdot S_W - \frac{S_W}{2}, (y/Res_H) \cdot S_H - \frac{S_H}{2}, (-S_W/2)/tan(FOV/2))$, где FOV - вертикальный обзор камеры. Дополнительно вектор D можно умножить на матрицу вращения для того, чтобы повернуть обозревателя.

Будем находить, пересёкся ли луч с каким-либо объектом, и если пересёкся, ставить точку его цвета. Иначе - голубую.

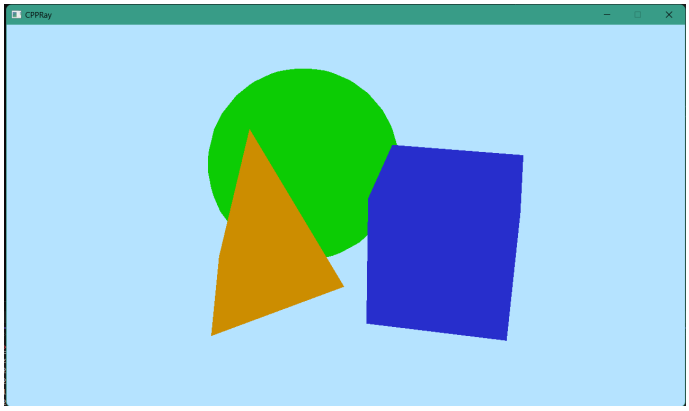


Рисунок 1: Пересечение луча и фигуры

Хотелсь бы немного разнообразить сцену - да будет свет! Введём направленные источники света, иначе говоря - солнце. Солнечный свет имеет направление, и следовательно освещать объекты будет по-разному. Чем больше угол между солнечным лучом и нормалью вершины, тем меньше света он будет получать.

$L_i = L_C \cdot L_I$ - интенсивность света

$Color = BaseColor \cdot L_i \cdot cos(-N, L_D)$, где BaseColor - цвет объекта, L_C - цвет света, L_I - мощность света, N - нормаль объекта, L_D - направление света.

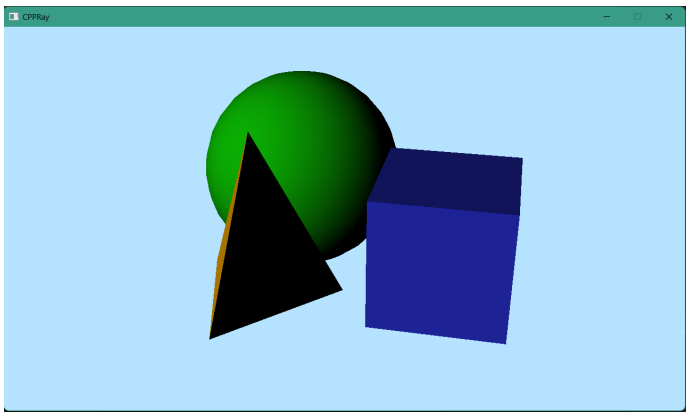


Рисунок 2: Сцена с источником света

Сцена стала немного интересней, однако она всё ещё довольно странная - предметы не отбрасывают тень. Для того, чтобы понять, отбрасывает ли объект тень, можем испустить луч от точки пересечения в противоположном направлении свету. Если мы найдём хоть один объект, котиорый пересекает луч, то значит в данной точке будет тень. Иначе - точка освещена.

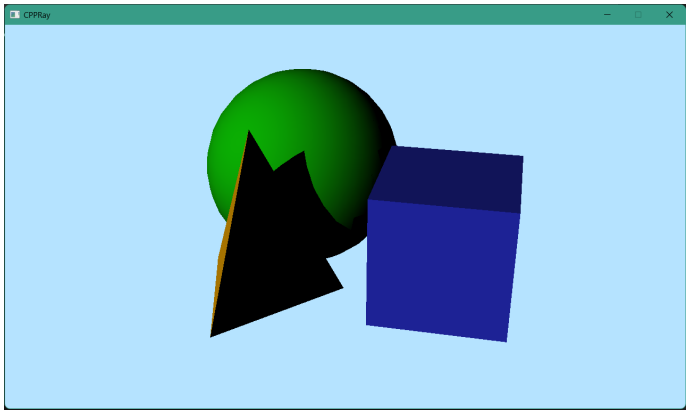


Рисунок 3: Добавление тени

Тетраэдр начал отбрасывать тень на сферу. Можно также добавить другие источники света - точечный свет и лампы. Основным отличием от солнца у этих источников света является затухание. С расстоянием сила света будет становиться меньше. Затухание можно расчитать по следующей формуле:

$Att = max(min(1 - \frac{Distance}{L_R}^4, 1), Distance^2)$, где Distance - расстояние между точкой на объекте и источником света, L_R - радиус света.

Точечный свет находится в одной точке и испускает свет во все стороны, формула для получения света будет следующей:

$L_i = L_C \cdot L_I \cdot Att$

$Color = BaseColor \cdot L_i \cdot cos(-N, L_D)$

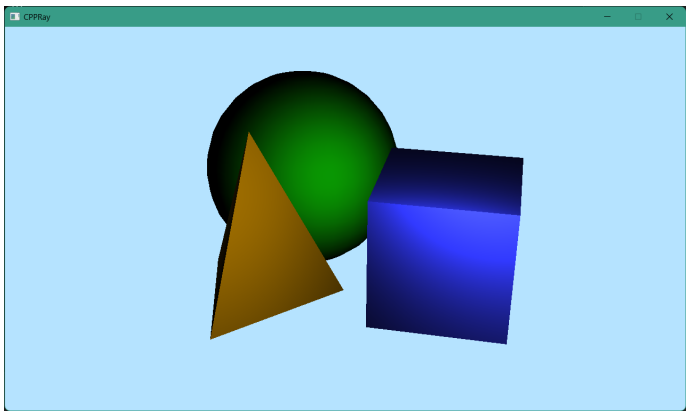


Рисунок 4: Точечный свет

Для определения тени будем испускать луч из источника света в направлении к рассматриваемой в данный момент точке. Если ближайшее пересечение с объектом - пересечение с искомым объектом, то он освещён. Иначе - оставляем тень.

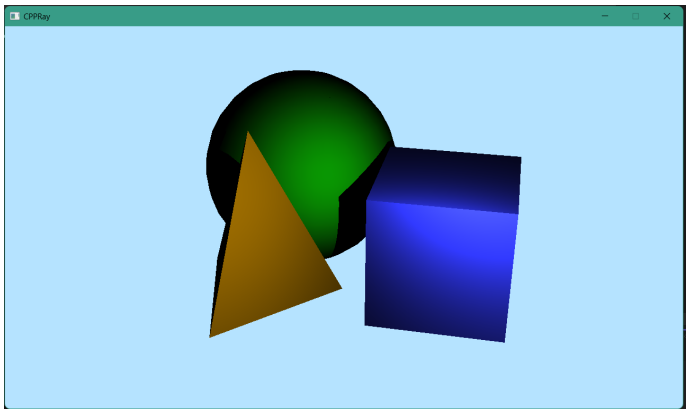


Рисунок 5: Точечный свет с тенью

В источнике света ”лампа”появляются углы внутреннего и внешнего конусов. Свет, находящийся во внутреннем конусе, имеет максимальную интенсивность, между внешним и внутренним конусом затихает, и вне внешнего конуса света нет.

Для вычисления интенсивности света используется следующая функция:

$$k = \begin{cases} 1 & angle < ICA \\ 0 & angle > OCA \\ \frac{angle}{ICA - OCA} + 1 - \frac{ICA}{ICA - OCA} \end{cases}$$

где angle - угол между направлением лампы и вектором от источника света до точки на объекте, ICA - радиус внутреннего конуса, OCA - радиус внешнего конуса.

Формула для вычисления освещённости точки на объекте будет иметь вид:

$$L_i = L_C \cdot L_I \cdot Att \cdot k$$
$$Color = BaseColor \cdot L_i \cdot cos(-N, L_D)$$

Просчёт наличия тени будет выполняться аналогично с точечным светом.

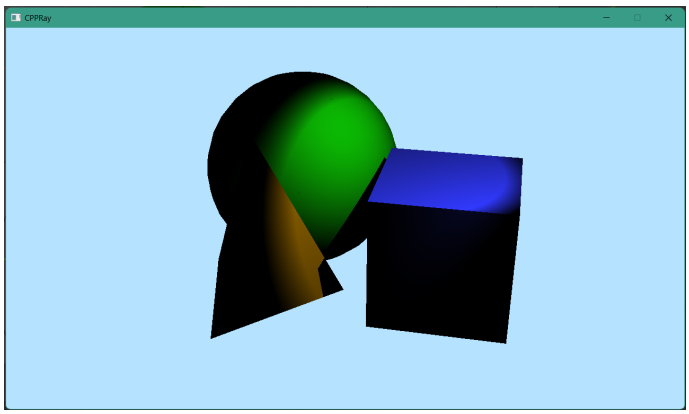


Рисунок 6: Лампа

1.2 Физически корректный рендеринг (PBR)

До сих пор мы рассматривали взаимодействие света и объекта упрощённо, однако в реальности такого не бывает.

Сборник техник Physically Based Rendering (в дальнейшем физически корректный рендеринг) позволяет определить, как свет будет взаимодействовать

с объектом в зависимости от его физических свойств - шероховатости и металличности.

Согласно физически корректному рендерингу, чтобы модель освещения могла быть реалистичной, она должна удовлетворять трём условиям:

- Основываться на модели отражащих микрограней.
- Подчиняться закону сохранения энергии.
- Использовать двулучевую функцию отражательной способности (или BRDF).

В общем виде формула для исходящего света имеет вид:
$L_0 = L_e + L_r$, где L_e - свет, испускаемый объектом, а L_r - свет, отражённый объектом.
Первую часть формулы - испускаемый свет - в данной реализации движка мы опустим и перейдём к отражённому свету. Формула отражённого света в общем случае:
$L_r = \int_\Omega f_r(p, w_i, w_0) L_i (w_i \cdot N) dw_i$, где f_r - BRDF, L_i - интенсивность света, w_i - вектор к источнику света, N - нормаль поверхности, p - рассматриваемая точка объекта, w_0 - вектор к наблюдателю.

Изначально, вычисление L_r предполагает учитывание всех источников света, которые находятся в полусфере, направление которой совпадает с направлением нормали. Однако, данная задача слишком сложна для вычислений на комьютере. Можно учитывать влияние только конкретных источников света - солнца, ламп, точечных источников света. Формулу можно упростить:
$L_r = \sum_n f_r(p, w_i, w_0) L_i (w_i \cdot N)$, где n - количество источников света.

Рассмотрим BRDF подробнее. Цель этой функции заключается в том, чтобы определить, как материал будет отражать свет в определённом направлении. Она состоит их двух частей:
$f_r = k_d \cdot f_{diffuse} + k_s \cdot f_{specular}$
k_d и k_f - коэффициенты, которые влияют на то, насколько сильным будет тот или иной вид освещения. $k_d + k_f \leq 1$, иначе энергия будет браться из ниоткуда - в этом условии заключён закон сохранения энергии. $f_{diffuse}$ - диффузное отражение, $f_{specular}$ - спекулярное отражение, или проще говоря - блеск.

Для вычисления коэффициента k_s можно использовать аппроксимацию Шлика:
$k_s = F_0 + (1 - F_0)(1 - cos\Theta)^5$. F_0 зависит от коэффициента преломления предмета. Для упрощения в программе примем, что $F_0 = 0.5$. Θ - угол между w_i и w_0.
Вычислить k_d легко: $k_d = 1 - k_s$. Металлы отражают только спекулярную часть, поэтому формулу можно немного модифицировать:
$k_d = (1 - k_s)(1 - metallic)$, где metallic - металличность материала.

Рассмотрим диффузную часть отражения $f_{diffuse}$. Для вычисления диффузной части можно использовать модель Ламберта или Орен-Наяра. Второй метод более реалистичный, но требующий больших вычислительных мощностей. Для данного проекта была выбрана модель Ламберта.

$k_s = \frac{BaseColor}{\pi}(w_i \cdot N)$

Перед рассмотрением спекулярной части отражения нужно углубиться в модель отражащих микрограней.

Объекты, которые на макроуровне кажутся гладкими, на микроуровне могут состоять из множества микрограней, которые в разные стороны отражают свет. Из-за таких неровностей свет не будет отражаться идеально или будет поглощаться самим объектом. Моделью, учитывающей такую особенность поверхности, является модель Кука-Торренса:

$$f_{specular} = \frac{DGF}{4(w_0 \cdot N)(L_D \cdot N)}$$

D - функция нормальной дистрибуции. Она описывает, как много микрограней будут повернуты к наблюдателю. Для вычисления D существует несколько моделей: Beckmann, GGX/Trowbridge-Reitz, GGX/Anisotropic. Будем использовать вторую модель.

$\alpha = roughness^2$, где roughness - шероховатость материала.
$D = \frac{\alpha^2}{\pi((N \cdot H)^2(\alpha^2 - 1) + 1)^2}$, где $H = w_o + L_D$.

G - функция геометрической затенённости, описывает, как много поверхности не самозатенено и не скрыто другими микрогранями. Для вычисления G будем использовать модель Schlick-GGX, которая объединяет в себе модели Смита и Шлика-Бекмана.

$k = \frac{\alpha}{2}$
$G1 = \frac{N \cdot w_0}{(N \cdot w_0)(1-k)+k}$
$G = G1(D_L, N)G1(w_0, N)$

F - функция Френеля, описывает, какая часть света была отражена, а какая - преломлена. Мы её уже использовали ранее для вычисления коэффициента диффузного света.
$F = F_0 + (1 - F_0)(1 - cos\Theta)^5$.

Итоговый результат:

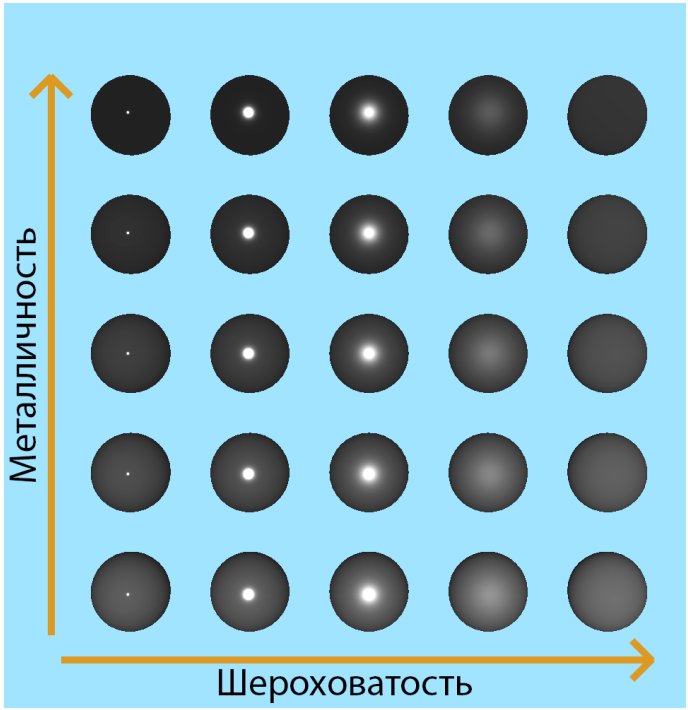


Рисунок 7: Объекты с разным соотношением металличность/шероховатость

2 Реализация алгоритмов

2.1 glTF

Перед тем как начать рендер 3D-сцены, необходимо получить информацию о ней. Можно было ”хардкодить”информацию о ней прямо в коде, однако полученная программа будет крайне негибка: требовать постоянной рекомпиляции, работы с программистом.

Информацию о сцене можно получать из файла. Выбор пал на несколько форматов хранения информации о 3D-сценах:

- STL - простой формат файла, однако не подходящий для реалистичного рендера, он содержит только информацию о форме объектов.
- FBX - популярный формат файла, разрабатываемый в Autodesk. Формат тоже не подходит, так как не содержит информацию для выбранной техники Physically Based Rendering.
- COLLADA - формат, основанный на XML и разработанный для передачи информации между 3D приложениями. Управляется Khronos Group. Формат также не поддерживает Physically Based Rendering.
- glTF - формат, основанный на JSON, расширяемый, легкопонимаемый. Используется в веб-технологиях. Поддерживает все необходимые данные для выбранных техник.

Для хранения сцены был выбран формат glTF. Для обработки glTF-файла была использована библиотека tinygltf.

2.2 HIPRT

HIPRT - библиотека, добавляющая поддержку трассировки лучей в HIP. При помощи HIP можно выполнять вычисления на видеокарте. Особенностью видеокарты является то, что она может выполнять один процесс многократно и параллельно, такой процесс описывается кернелом - функцией, которая будет запускаться на видеокарте. Кернел на основе информации о сцене будет применять описанные выше алгоритмы реалистичного рендера для просчёта

цвета одной точки на холсте, сохранять результаты своей работы в память видеокарты, после чего эти данные можно скопировать из устройства в ОЗУ и вывести получившееся изображение на экран.

Перед тем как запустить кернел, необходимо создать сцену. Сцена - акселирирующая структура данных, которая используется методах HIPRT для пересечения луча и геометрии. Сцена создаётся при помощи структуры hiprtSceneBuild hiprtSceneBuildInput содержит список геометрических объектов instanceCount и их количество instanceCount; кадры с трансформацией объектов и их временем (перемещение, размер, вращение) instanceTransformHeaders и instanceFrames; количество самих кадров frameCount, BVH-ноды nodes, позволяющие ускорить процесс нахождения пересечения геометрии и луча. В данной работе ограничимся геометриями (объектами) сцены и фреймами с их трансформацией. На основе сформированной структуры создаётся сцена, которая в дальнейшем будет использоваться в кернеле:

```
// Загружаем модель из файла, сохраняем результат в geometries, frames, srtHeaders.
loadModel( std::string( path ), ctxt, frames, srtHeaders );

hiprtSceneBuildInput sceneInput;
sceneInput.instanceCount            = geometries.size(); // Количество геометрий
sceneInput.instanceMasks            = nullptr;
sceneInput.instanceTransformHeaders = nullptr;

// Пример копирования данных в память видеокарты - копирование геометрии
CHECK_ORO( oroMalloc(
    reinterpret_cast<oroDeviceptr*>( &sceneInput.instanceGeometries ),
    sizeof( hiprtDevicePtr ) * sceneInput.instanceCount ) );
CHECK_ORO( oroMemcpyHtoD(
    reinterpret_cast<oroDeviceptr>( sceneInput.instanceGeometries ),
    &geometries[0],
    sizeof( hiprtDevicePtr ) * sceneInput.instanceCount ) );

sceneInput.frameType = hiprtFrameTypeMatrix; // Вид трансформации геометрии - матрица трансформации
int frameCount = frames.size();

// Копирование кадров
...

sceneInput.frameCount = frameCount;

// Копирование заголовков трансформации
...
```

```
size_t           sceneTempSize;
hiprtDevicePtr sceneTemp;
CHECK_HIPRT( hiprtGetSceneBuildTemporaryBufferSize( ctxt, sceneInput, options, sceneTempSize ) );
CHECK_ORO( oroMalloc( reinterpret_cast<oroDeviceptr*>( &sceneTemp ), sceneTempSize ) );

// Собираем сцену, после чего её можно использовать в кернеле
CHECK_HIPRT( hiprtCreateScene( ctxt, sceneInput, options, scene ) );
CHECK_HIPRT( hiprtBuildScene( ctxt, hiprtBuildOperationBuild, sceneInput, options, sceneTemp, 0, scene )
↪  );

// Сборка кернела для дальнейшего запуска
buildTraceKernelFromBitcode( ctxt, "../common/Kernels.h", "mainKernel", func );

// Создание изображения в памяти видеокарты
CHECK_ORO( oroMalloc( reinterpret_cast<oroDeviceptr*>( &pixels ), m_res.x * m_res.y * 4 ) );

// Удаляем буффер
CHECK_ORO( oroFree( reinterpret_cast<oroDeviceptr>( sceneTemp ) ) );
```

Рассмотрим получение массивов geometries, frames, srtHeaders.

Геометрия в HIPRT создаётся при помощи hiprtTriangleMeshPrimitive. Эта структура содержит массив с вершинами vertices, их количество vertexCount и шаг между вершинами в байтах в массиве vertexStride. На основе массива вершин формируются треугольники, структура содержит массив индексов из массива вершин triangleIndices. Тройка индексов образует один треугольник. Структура содержит количество треугольников triangleCount и шаг между тройкой индексов. При помощи структуры hiprtGeometryBuildInput создаётся геометрия, которая будет в дальнейшем использоваться при создании сцены:

```
hiprtTriangleMeshPrimitive hipMesh;

// Загружаем индексы из glTF документа
hipMesh.triangleCount  = model.accessors[meshPrimitive.indices].count / 3;
hipMesh.triangleStride = sizeof( hiprtInt3 );

...

// Загружаем вершины из glTF документа

...

// Создаём hiprtGeometryBuildInput, устанавливаем тип примитива - треугольник и копируем в него
↪  полученный примитив
```

```
hiprtGeometryBuildInput geomInput;
geomInput.type                    = hiprtPrimitiveTypeTriangleMesh;
geomInput.triangleMesh.primitive = hipMesh;

size_t              geomTempSize;
hiprtDevicePtr      geomTemp;
hiprtBuildOptions options;
options.buildFlags = hiprtBuildFlagBitPreferFastBuild;
CHECK_HIPRT( hiprtGetGeometryBuildTemporaryBufferSize( ctxt, geomInput, options, geomTempSize ) );
CHECK_ORO( oroMalloc( reinterpret_cast<oroDeviceptr*>( &geomTemp ), geomTempSize ) );

hiprtGeometry geom;
CHECK_HIPRT( hiprtCreateGeometry( ctxt, geomInput, options, geom ) );
CHECK_HIPRT( hiprtBuildGeometry( ctxt, hiprtBuildOperationBuild, geomInput, options, geomTemp, 0, geom )
↪  );

// Геометрия получена, можно добавить её в geometries.

geometries.push_back( geom );
```

Трансформация примитивов из файла glTF будем сохранять в фреймы. В данном движке используется матрица трансформации, поэтому будем использовать структуру hiprtFrameMatrix, которая содержит время и саму матрицу размером $3x4$. В некоторых графических библиотеках используется матрица $4x4$, однако здесь матрица была сокращена для экономии памяти и уменьшения времени работы из-за того, что последняя строка матрицы трансформации, содержащая трансформацию, вращение и размер не содержит полезной информации.

hiprtFrameMatrix также содержит время, реализация анимации в данном движке не предусмотрена, поэтому время будем оставлять равное 0.

```
// Умножаем родительскую матрицу трансформации на текущую
hiprtFrameMatrix localTransformations = parentTransform * getSRTMatrix( translate, rotation, scale );
localTransformations.time             = 0;

// Сохраняем фрейм в массив фреймов
frames.push_back( localTransformations );
```

Чтобы прикрепить к i геометрии трансформацию в j фрейме используется массив hiprtTransformHeader. Структура содержит индекс фрейма из массива и сколько кадров она будет активна. Второй параметр нам не сильно важен, так как в движке не предусмотрена анимация.

```
hiprtTransformHeader header;

// Трансформации будут активны один единственный 0 кадр
header.frameCount = 1;
// Мы сначала добавляем фреймы, после чего обрабатываем примитивы, к которым трансформации
// этих фреймов были применены. Поэтому индекс фрейма - последний.
header.frameIndex = frames.size() - 1;

// Добавляем полученный header.
srtHeaders.push_back( header );
```

Дополнительно в кернеле нам понадобится информация о материалах, свете, камере и нормалям вершин. Структур HIPRT для этих объектов нет, поэтому их нужно будет загрузить из файла в память видеокарты напрямую. Так для примера выглядит загрузка нормалей вершин:

```
// Примитив в файле glTF должен содержать нормали
auto accessorIter = meshPrimitive.attributes.find( "NORMAL" );
if ( accessorIter == meshPrimitive.attributes.end() )
    throw std::string( "Err: no NORMAL attribute specified in primitive" );

int accessorIndex = ( *accessorIter ).second;

int normalCount     = model.accessors[accessorIndex].count;
int normalStride = tinygltf::GetComponentSizeInBytes( model.accessors[accessorIndex].componentType ) * 3;

auto normalBufferView = model.bufferViews[model.accessors[accessorIndex].bufferView];

// Копирование нормалей в ОЗУ
a = (hiprtFloat3*)malloc( normalCount * normalStride );
memcpy(
    a,
    &model.buffers[normalBufferView.buffer].data[0] + normalBufferView.byteOffset,
    normalCount * normalStride );

// Копирование нормалей в память видеокарты
CHECK_ORO(
    oroMalloc( reinterpret_cast<oroDeviceptr*>( &( geomData.back().normals ) ), normalCount *
    ↪  normalStride ) );
CHECK_ORO(
    oroMemcpyHtoD( reinterpret_cast<oroDeviceptr>( geomData.back().normals ), a, normalCount *
    ↪  normalStride ) );

// Удаление буффера из ОЗУ
free( a );
```

После того, как мы загрузили сцену в акселирирующие структуры данных, можно приступать к запуску кернела. Процессы в видеокарте выполняются в 3-мерном пространстве, каждому процессу присваивается координата x, y и z. Так как наша задача - получить двухмерную картинку, вычисления будут двухмерные, а процессу будут присвоены координаты x и y, соответствующие координате отрисовываемого пикселя на холсте.

Основной задачей в трассировке лучей является нахождение пересечения треугольника и луча, для нахождения пересечения HIPRT предоставляет собственные классы, которые на основе сцены и луча, находят пересечения.

```
// Инициализация луча: его направления и точки начала
hiprtRay ray;
ray.origin    = o;
ray.direction = d;

// hiprtSceneTraversalClosest позволяет находить ближайшую точку пересечения луча и геометрии в сцене
hiprtSceneTraversalClosest tr( scene, ray );
// Получаем пересечение
hiprtHit hit = tr.getNextHit();
```

Структура данных hiprtHit содержит полезную информацию о пересечении: instanceID (индекс в массиве геометрий), primID (индекс треугольника в структуре меша), UV, нормаль и t ($hit = ray.origin + t \cdot ray.direction$).

instanceID позволяет определить, с каким именно мешем мы столкнулись, что позволяет, например, ввести систему материалов:

```
auto material = materials[hit.instanceID];
```

Нормаль normal высчитывается на основе обхода треугольника и не всегда корректна. Для просчёта нормали в точке используется UV-координата и массив нормалей вершин. Состовляющие UV-координаты U, V и W = 1 - U - V определяют, насколько далеко точка на треугольнике находится от его вершин.

Нормали вершин можно интерполировать по формуле: $N = w * N_1 + u * N_2 + v * N_3$.

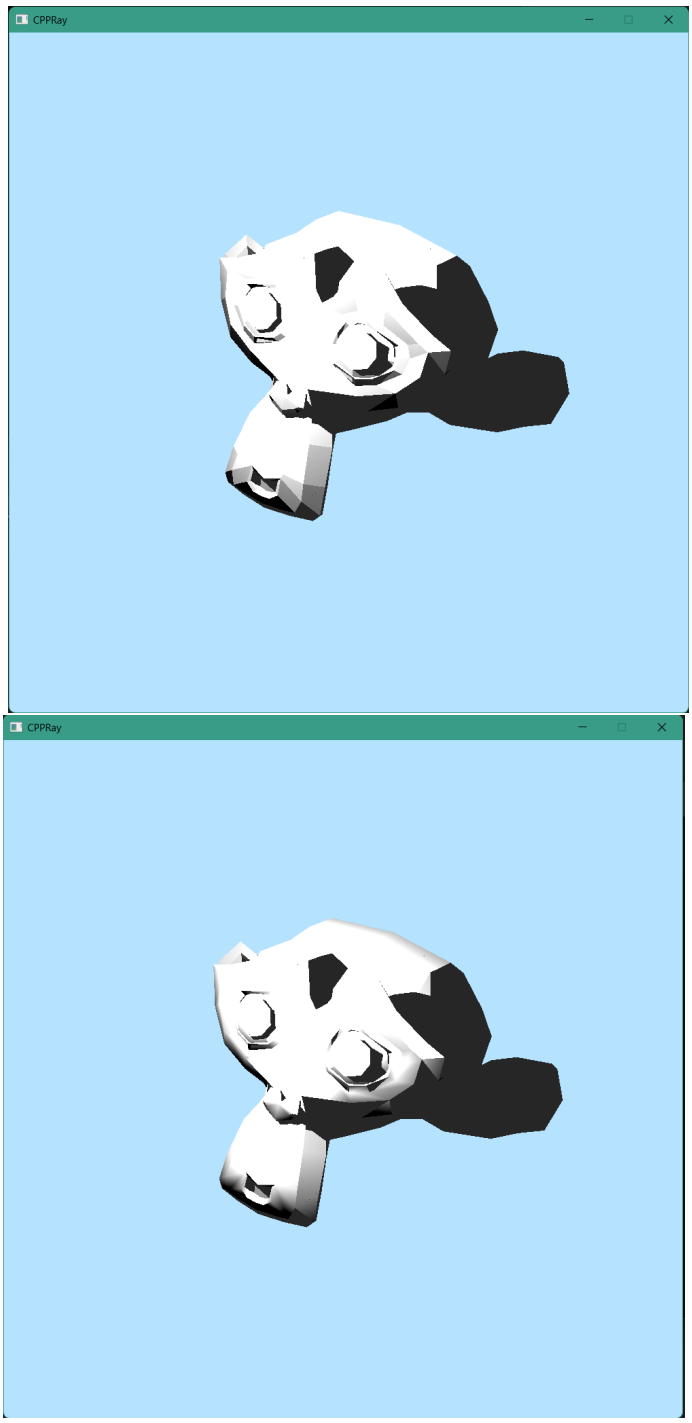


Рисунок 8: Неинтерполированная и интерполированная нормаль

2.3 SFML

После выполнения кернела мы получаем массив пикселей, для упрощения работы с программой полученное изображение можно вывести в отдельном окне. Выполнить вывод в окне можно при помощи OpenGL, однако для упрощения вывода в программе используется обёртка над OpenGL - SFML.

```cpp
int main( int argc, char* argv[] ) {
    // Получаем ширину и высоту изображения из аргументов запуска
    width = std::stoi( argv[2] );
    height = std::stoi( argv[3] );

    // Инициализируем массив пикселей
    data = (u8*)malloc( width * height * 4 );

    // Инициализируем движок, передаём путь из аргументов запуска
    renderEngine.init( 0, width, height, argv[1] );
    // Запускаем движок
    renderEngine.run( data );

    // Создаём окно
    sf::RenderWindow window( sf::VideoMode( width, height ), "CPPRay", sf::Style::Close );

    while ( window.isOpen() ) {
        sf::Event event;
        while ( window.pollEvent( event ) ) {
            if ( event.type == sf::Event::Closed ) {
                window.close();
            }
        }

        // Копируем полученный массив в изображение
        sf::Image image;
        image.create( width, height, data);

        window.clear();

        // Создаём текстуру
        sf::Texture texture;
        texture.setSmooth( true );
        texture.loadFromImage( image );

        // Спрайт
        sf::Sprite sprite;
        sprite.setTexture( texture, true );

        // После чего отрисовываем его
        window.draw( sprite );

        window.display();
    }

    return 0;
}
```

3 Заключение

В ходе курсовой работы были изучены основы техник генерации реалистичной компьютерной графики, инструменты для добавления аппаратной поддержки в видеокарты. Полученный движок способен отрисовывать простые сцены, содержащие примитивы, свет.

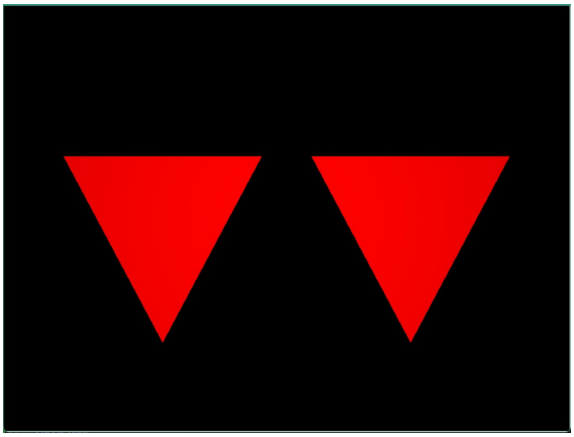
Движку ещё есть куда развиваться: можно расширить возможности материалов (текстурирование, светимость материалов), добавить продвинутые материалы (прозрачные объекты), другие техники реалистичного рендера (непрямое освещение, окружающее затенение), углубление движка в реалистичную сторону, что позволит использовать его например в фильмах, или ускорить движок, что позволит использовать его в играх, добавить анимацию, размытие в движении, более удобный интерфейс.

Полученный движок реализует базовые техники реалистичного рендера и для дальнейшего применения в определённой сфере требует соответствующих доработок.

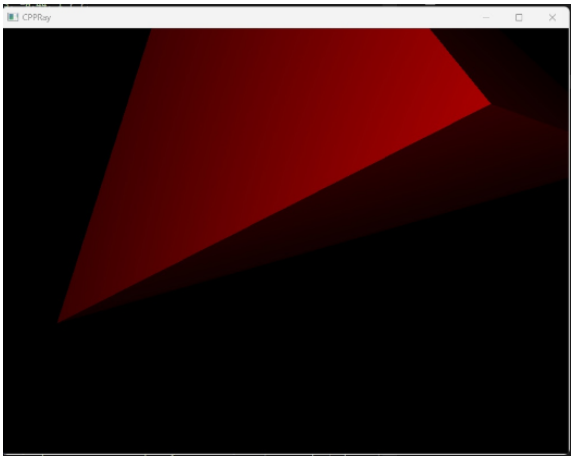
Ссылка на проект: https://github.com/IAmProgrammist/CPPRay

4 Приложение

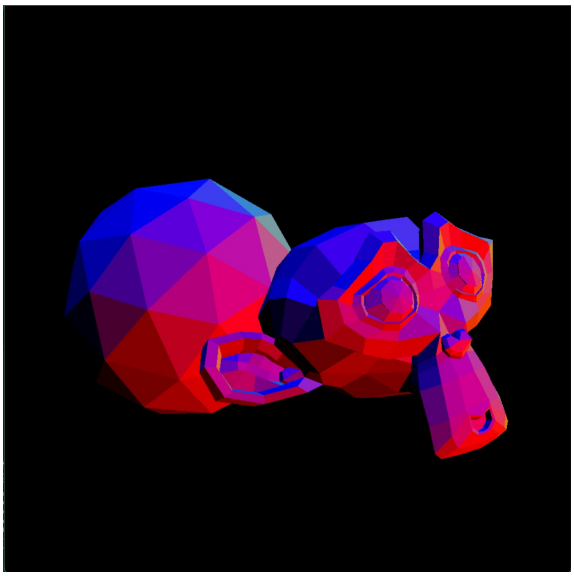
Тестовые рендеры



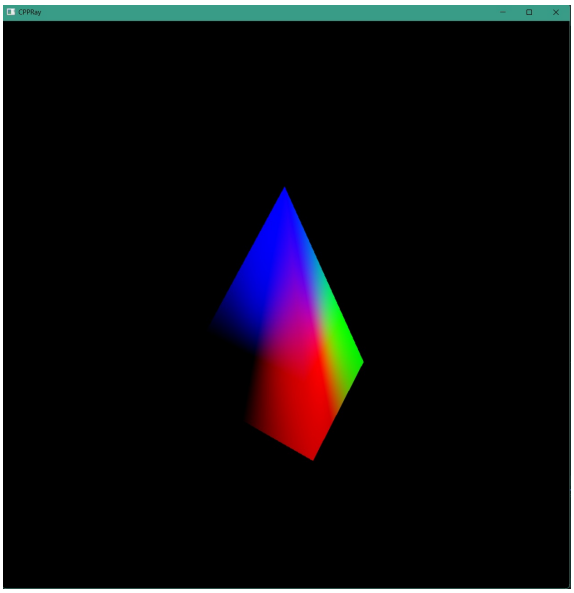
Тестовая отрисовка
простых примитивов
(29.10.2023)



Тестовая отрисовка
простых мешей
(30.11.2023)



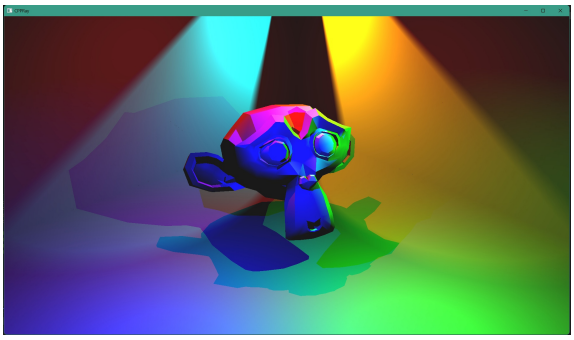
Проверка нормалей
Канал R - X+, G - Y+, B - Z+
(4.11.2023)



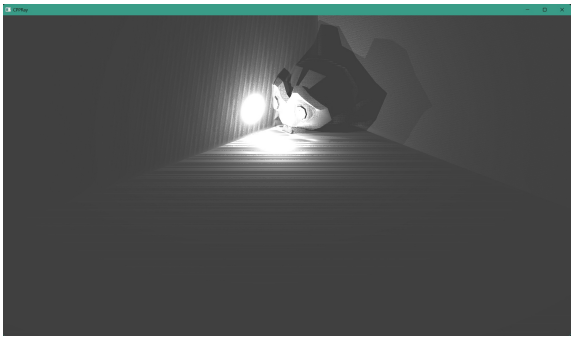
Проверка интерполяции
нормалей
(4.11.2023)

Тестирование примитивной системы освещения (6.11.2023)

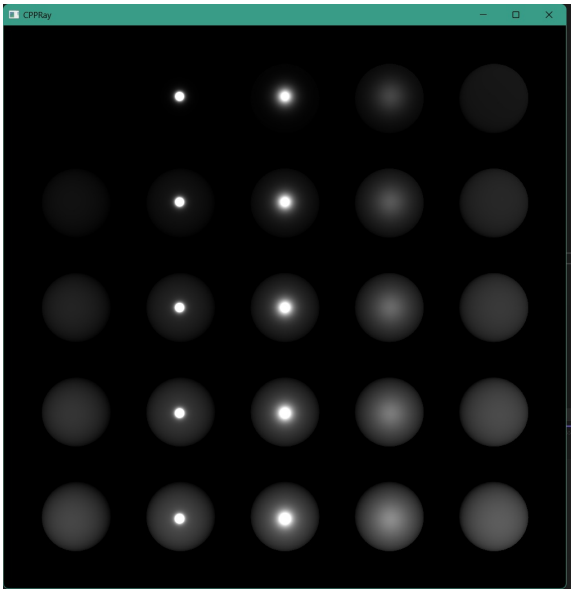
Тестирование лампы и аддитивной модели (13.11.2023)



Тест сцены с несколькими источниками освещения (13.11.2023)

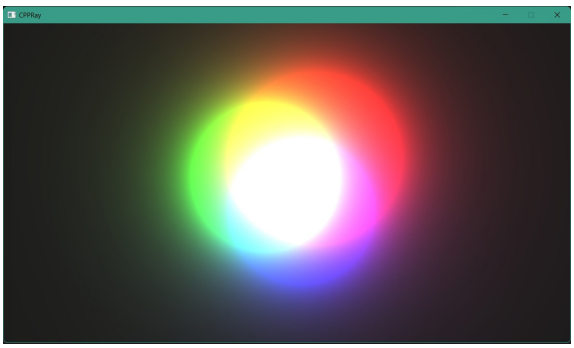


Неудачная попытка ввести непрямое освещение (13.11.2023)



Введение системы PBR (17.12.2023)

Неудачная попытка ввести непрямое освещение (19.12.2023)



Аддитивная модель с PBR
(20.12.2023)

Замок
(21.12.2023)

5 Список источников и литературы

1. HIP Ray Tracing - AMD GPUOpen [Электронный ресурс]
   Режим доступа: https://gpuopen.com/hiprt/

2. glTF 2.0 Specification [Электронный ресурс]
   Режим доступа: https://github.com/KhronosGroup/glTF/blob/main/specification/2

3. tinygltf [Электронный ресурс]
   Режим доступа: https://github.com/syoyo/tinygltf

4. Трёхмерная графика с нуля. Часть 1: трассировка лучей [Электронный ресурс]
   Режим доступа: https://habr.com/ru/articles/342510/

5. LearnOpenGL - PBR - Theory [Электронный ресурс]
   Режим доступа: https://learnopengl.com/PBR/Theory

6. Physically Based Rendering [Электронный ресурс]
   Режим доступа: https://reference.wolfram.com/language/tutorial/PhysicallyBasedl

7. SFML Documentation [Электронный ресурс]
   Режим доступа: https://www.sfml-dev.org/documentation/2.6.1/