МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
имени первого Президента России Б. Н. Ельцина

ИНСТИТУТ ЕСТЕСТВЕННЫХ НАУК И МАТЕМАТИКИ

Департамент математики, механики и компьютерных наук

Real-time rendering. Модели освещения. Реализация и тестирование.

Направление подготовки 02.03.02 “Фундаментальная информатика и информационные технологии”

Директор департамента:
к. ф.-м. н.
Е.С. Пьянзина

Выпускная квалификационная
работа бакалавра
**Коробейникова Николая
Николаевича**

Нормоконтролер:
О.А. Суслова

Научный руководитель:
П.А. Васёв

Соруководитель:
к. ф.-м. н., доцент
И.С. Стародубцев

Екатеринбург

2022

РЕФЕРАТ

Коробейников Н. Н., Real-time rendering. Модели освещения. Реализация и тестирование. Выпускная квалификационная работа бакалавра: 50 стр. 22 рис., 3 табл., 32 источника, 1 приложение.

Ключевые слова: компьютерная графика, освещение, рендер, физически корректное освещение, PBR, Vulkan.

Предмет исследования - современные модели освещения, используемые в программах для визуализации трехмерных сцен.

Цель работы - анализ используемых на данный момент моделей освещения и реализацию наиболее интересных моделей на практике в качестве проекта рендера.

Результат работы - рендер, который может быть использован при разработке игровых движков или просто для разработки программ, которым нужно уметь взаимодействовать с 3D объектами.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Мотивацией к написанию данной работы послужил интерес автора к современным технологиям рендеринга, в том числе интерес к моделям освещения.

Освещение - это основной компонент, без которого невозможно получить фотореалистичную картинку. Большее внимание здесь будет уделено рассмотрению подходов для создания именно фотореалистичной картинки, потому что принципы, применяемые в этой области, более интересны с теоретической точки зрения, по мнению автора.

Итогом данной работы мы хотим видеть анализ используемых на данный момент моделей освещения и реализацию наиболее интересных моделей на практике в качестве проекта рендера, который в будущем может быть использован при разработке игровых движков или просто для разработки программ, которым нужно уметь взаимодействовать с 3D объектами.

БАЗОВЫЕ ПОНЯТИЯ

- Рендеринг в реальном времени (Real-time rendering) - вид 3D рендеринга, где расчеты и отображение происходят в режиме реального времени. Чаще будет использоваться английское написание в силу своей краткости.
- Фотореалистичность - похожесть сгенерированного изображения на фотографию.
- Фотореалистичное изображение - изображение, похожее на фотографию.
- Фотореализм - направление компьютерной графики, стремящееся к тому, чтобы конечное изображение было неотличимо от фотографии.
- Цветовое пространство RGB - подмножество видимого цветового пространства, порождаемое RGB базисом (красный, зеленый, синий).
- Игровой движок (Game engine) - базовое программное обеспечение компьютерной игры.
- Рендер - компьютерная программа (программный компонент) предназначенная для получения изображения по компьютерной модели.

ОСНОВНАЯ ЧАСТЬ

1 Выбор инструментов

Историю и обзор современных графических API можно найти в приложении Б, здесь же будут приведены результаты проведенных исследований.

1.1 Выбор графического API

В настоящей работе выбор был сделан в пользу Vulkan, за счет его кроссплатформенности и возможности поддержки нескольких высокоуровневых шейдерных языков, в том числе HLSL, GLSL. Осветим основные особенности, доступные нам при разработке на этом API.

Последняя спецификация: Vulkan 1.3 (25 янв 2022).

Основные поддерживаемые платформы: Windows, Nintendo Switch, Stadia, MoltenVK, Linux, Android.

Vulkan построен на 3-х идеях:

- высокая производительность, достигаемая за счет приближения API к железу с возможностью напрямую управлять ресурсами процессора и графических ускорителей;
- мультиплатформенность;
- высокая степень предсказуемости за счет малого контроля со стороны драйвера.

По сравнению с аналогичными по функциональности версиями OpenGL, много контроля перешло с уровня драйвера на уровень приложения, в том числе контроль за памятью, управление потоками и командными буферами. Также Vulkan поддерживает несколько Front-End компиляторов, которые компилируют код шейдеров на языке высокого уровня (HLSL,

GLSL и другие) в шейдерный код на языке SPIR-V, и полученные шейдеры на языке SPIR-V (файлы с этими шейдерами часто имеют расширение .spv) уже используются вулканом во время работы.

1.2 Выбор языка программирования

Было решено выбрать мультипарадигменный язык программирования C++ как максимально подходящий для нашей конечной цели, к важным преимуществам этого языка можно отнести:

- скорость работы;
- возможности для явного контроля памяти;
- кроссплатформенность;
- распространенность (в т.ч. в мире игровых движков).

2 Предметная область

Определившись с инструментами, которые мы будем использовать для реализации наших целей, можно перейти к рассмотрению предметной области. Далее будем обсуждать вещи, связанные непосредственно с алгоритмами отрисовки, фокусируясь на Real-time рендеринге с получением фотореалистичной картинки.

2.1 Модели освещения

Фактически задача расчета освещения для тела сводится к задаче расчета освещения в каждой видимой точке поверхности этого тела. Видимыми в данном случае будем называть те точки поверхности, для

которых найдутся соответствующие пиксели\фрагменты на экране при формировании итогового изображения. Существует множество моделей освещения для проведения таких расчетов, рассмотрим далее основные из них. Моделью освещения будем называть набор алгоритмов, предназначенных для расчета освещения.

Введем еще одно понятие. Освещенностью будем называть значение RGB (точка в цветовом пространстве RGB с красным, зеленым и синим базисными цветами), получаемое в результате расчетов конкретной модели освещения для конкретной точки поверхности тела.

Отметим тот факт, что почти все модели освещения для расчета освещенности используют информацию о нормали к поверхности тела в точке расчета, часто учитывая закон косинусов Ламберта[1] (Lambert's cosine law).

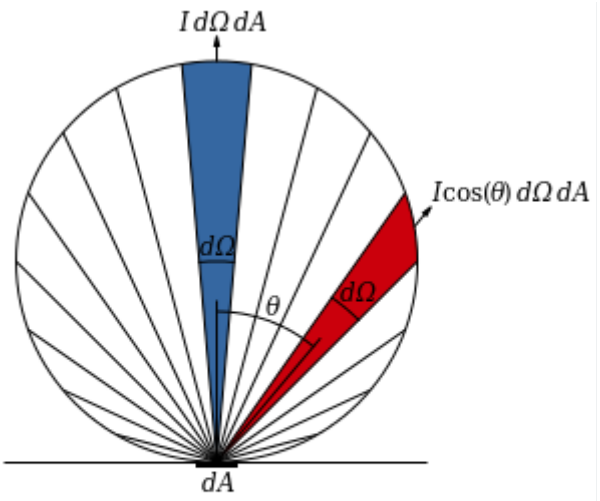


Рисунок 2.1.1: связь количества излучения с направлением излучения. Иллюстрация закона косинусов Ламберта [34]

Рассмотрим несколько способов расчета освещенности:

[1] Ламберта закон или закон косинусов – закон в оптике, согласно которому радиальная интенсивность излучения от ламбетовской поверхности (излучателя) прямо пропорциональна косинусу угла между направлением на наблюдателя и нормалью к поверхности.

- flat - освещенность полигона определяется освещенностью одной из его вершин;
- Gouraud - вычисление освещенности проводится в вершинах, после чего линейно интерполируется на полигонах;
- Phong - освещенность вычисляется для каждого фрагмента[2].

Все модели освещения можно условно разделить на 2 класса: локальные\объектно-ориентированные (Object-oriented lighting) и глобальные (Global illumination). В локальных моделях учитывается только прямое освещение (свет исходящий непосредственно из источника света), а в глобальных моделях в дополнение к прямому освещению рассчеты производятся и для непрямого освещения (отраженный свет).

Локальные модели:

- banded - дискретное множество допустимых значений освещенности;
- Lambert - освещенность считается по закону косинуса Ламберта;
- half-Lambert (Diffuse Wrap) - модификация предыдущей модели, за счет которой у задней части тела появляется освещенность;
- Phong Lighting - самая что ни есть классическая модель, освещенность представляется как сумма diffuse и specular компонент, что позволяет получать настраиваемые блики;
- Blinn-Phong Lighting - модификация предыдущей модели, основной смысл в уменьшении расчетов при схожей картинке;

[2] Важно различать понятия пикселя (физический объект, часть экрана) и фрагмента (все, для чего вызывается фрагментный шейдер). В одном пикселе может быть несколько фрагментов (например, такая ситуация возникает при использовании MSAA, с таким подходом может быть, что один пиксель в итоге будет относиться к двум разным объектам, которые имеют разные параметры поверхности) и наоборот, в одном фрагменте может быть несколько пикселей (например, картинка рендерится в разрешении ниже, чем разрешение экрана).

- Minnaert Lighting - лунный шейдер, подходит также для пупырчатых материалов и некоторых тканей;
- Oren-Nayer Lighting - визуализация грубых (шероховатых) материалов;
- модели на основе PBR (Physically Based Rendering) - класс моделей, где в основе расчетов освещенности лежат конкретные[3] физические явления и законы.

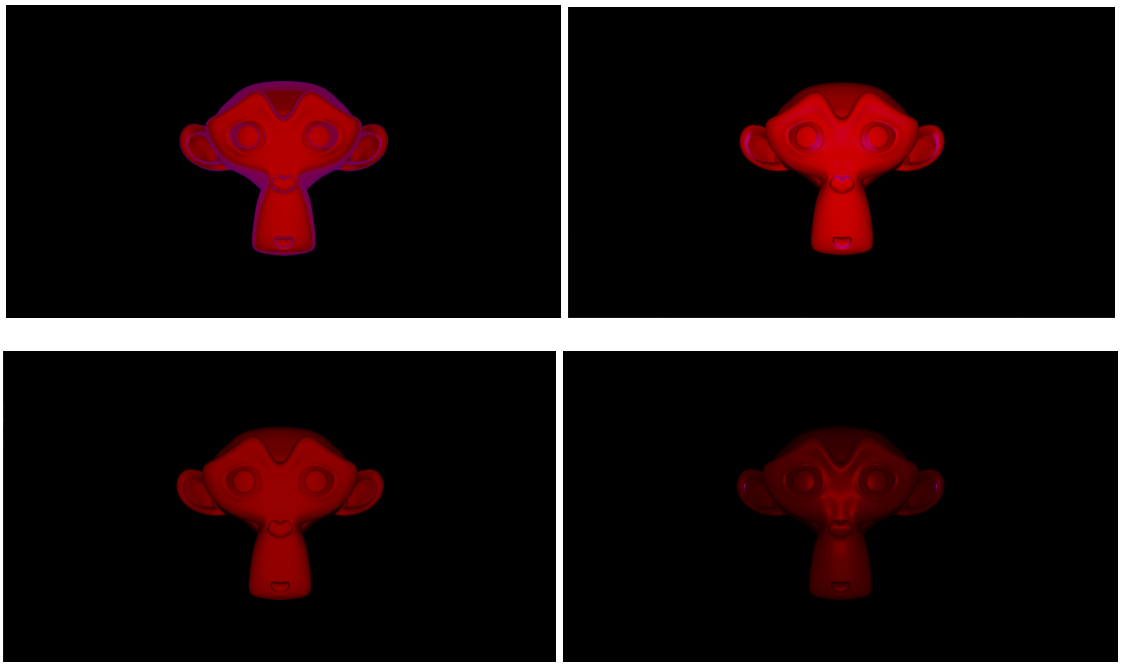


Рисунок 2.1.2: модели освещения в нашей реализации:
слева сверху - Phong, справа сверху - Blinn-Phong, слева снизу - Minnaert, справа снизу - PBR (GGX)

[3] Другие модели могут строиться на эмпирических закономерностях, а некоторые могут быть просто вольно выведенными (придуманными), чтобы лучше соответствовать нужному стилю

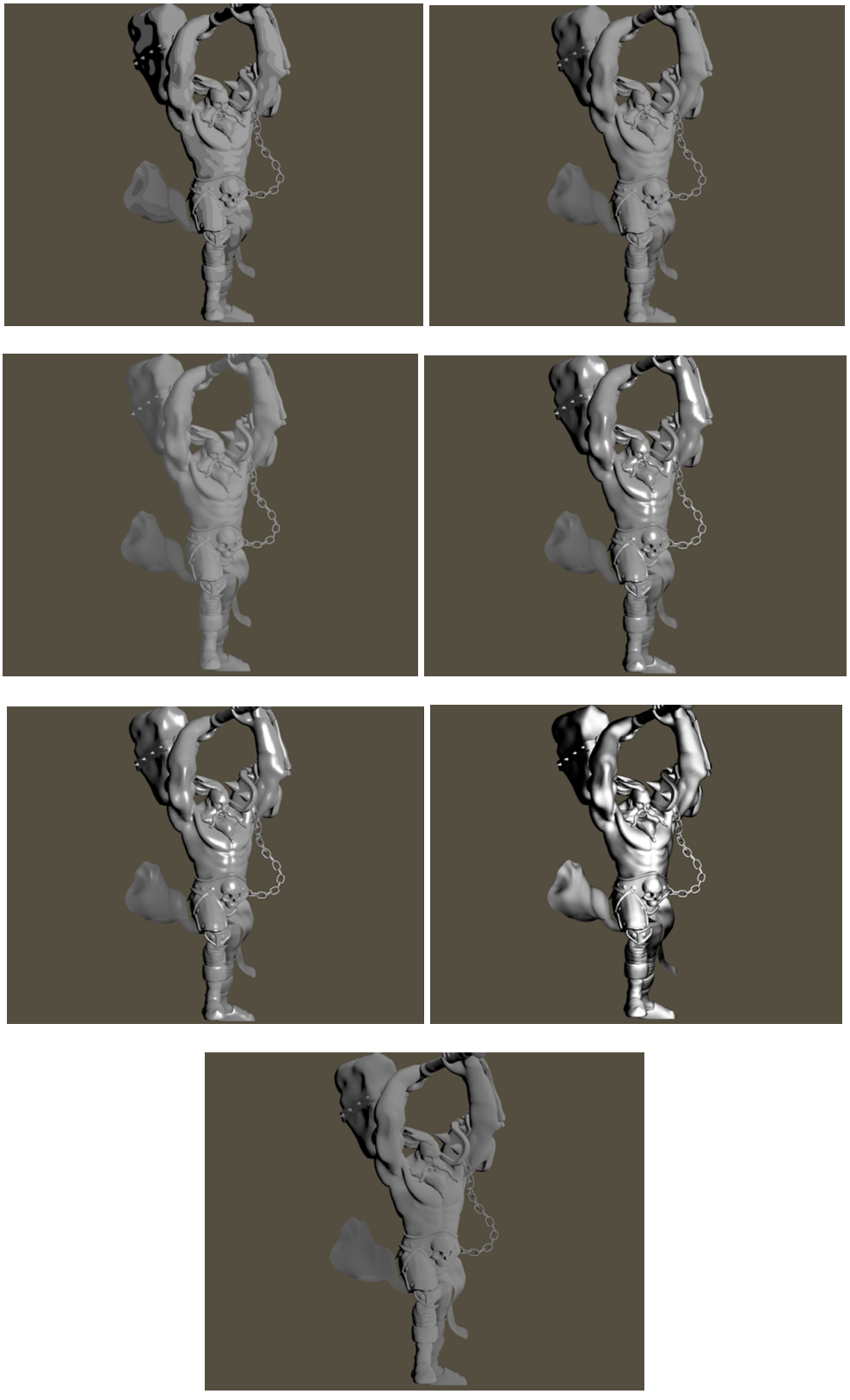


Рисунок 2.1.3: иллюстрация моделей освещения в Unity. По порядку слева направо, сверху вниз - Banded, Lambert, Half-Lambert, Phong, Blinn-Phong, Minnaert, Oren-Nayer [20]

Глобальные модели:

- ray Tracing - класс алгоритмов, моделирующих распространение света путем некоторого количества отражений. Использование таких алгоритмов в real-time рендеринге на потребительских устройствах стало возможным после появления соответствующих архитектур, где трассировка реализована на аппаратном уровне (RT ядра);
- path Tracing - основанный на идее Ray Tracing метод Монте Карло для определения освещенности пикселей на итоговом изображении за счет приближения интеграла из уравнения рендеринга;
- photon Mapping - основанный на идее Ray Tracing метод, заключающийся в независимом испускании лучей камерой и источниками света и определении точек их пересечения для дальнейшего определения итоговой освещенности;
- radiosity - применение метода конечных элементов для нахождения интеграла в уравнении рендеринга для диффузных сцен;
- metropolis Light Transport - метод Монте Карло для аппроксимации уравнения рендеринга, использующий Path Tracing и имеющий возможность строить новые пути лучей из уже построенных.

Рисунок 2.1.4: демонстрация возможностей real-time ray tracing в Unreal Engine 4 [35]

Изображения, получаемые с использованием глобальных моделей освещения, часто являются более фотореалистичными, чем изображения использующие только локальные модели. При всем этом локальные модели освещения могут использоваться в процессе расчета глобальными моделями.

Но изображения, получаемые в результате работы глобальных моделей освещения, дороже в вычислительном плане (иногда на порядки), а значит такие изображения генерируются дольше, что критично для их применения real-time рендеринге. Поэтому частым решением для использования этих моделей в real-time является одноразовый просчет глобального освещения и сохранение полученной информации в в специально созданной для этого текстуре (или текстурах), после чего эта информация используется уже для генерации итогового изображения в real-time.

2.2 Physically Based Rendering (PBR)

Для более детального изучения выберем модели на основе PBR, потому что это направление сейчас бурно развивается и представляет интерес в плане теоретических подходов. В основе моделей лежит аппроксимация уравнения рендеринга, которая учитывает физические законы, связанные со светом, в том числе в достаточной степени аппроксимирует взаимодействие света и поверхностей, распространение света в среде и т.д.

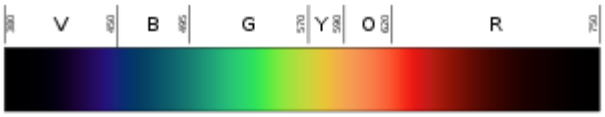
Эти модели стали очень популярными[4] в последние 5 лет, потому что позволяют добиться фотореалистичной картинки, и все расчеты достаточно хорошо стандартизированы, что позволяет ускорить разработку графики (прежде всего за счет того, что художник, создавая PBR материал в сторонней программе может перенести этот материал в какой-нибудь движок и этот материал будет выглядеть в движке точно так, как он выглядел в сторонней программе до импорта).

2.2.1 Физика света

Наше восприятие цвета это психофизическое явление, физиологическое восприятие физических раздражителей.

Сначала нужно определиться, в каких единицах мы будем измерять количество света. Для этого посмотрим в сторону физических определений, приводимых в радиометрии. В этом случае мы имеем дело со светом как с электромагнитной волной (поперечная волна с определенной длиной λ и частотой ν) и измеряем ее излучение.

[4] Без труда их реализацию можно найти в последних версиях Blender, Unity, Unreal Engine и т.п.



Color	Wavelength (nm)	Frequency (THz)	Photon energy (eV)
violet	380–450	670–790	2.75–3.26
blue	450–485	620–670	2.56–2.75
cyan	485–500	600–620	2.48–2.56
green	500–565	530–600	2.19–2.48
yellow	565–590	510–530	2.10–2.19
orange	590–625	480–510	1.98–2.10
red	625–750	400–480	1.65–1.98

Рисунок 2.2.1.1: связь частоты, длины волны и энергии фотона для видимого света [32]

В радиометрии базовый термин - поток излучения (radiant flux) Ф - поток излучаемой энергии за время t - энергия - измеряется в Вт. Дальше определяется несколько производных терминов, самый важный для нас - светимость (radiance) - измеряется в Вт/(м^{2}ср) - это поток энергии через единичную площадь и единичный телесный угол[5] за время t.

Таблица 2.2.1.1: единицы измерения в радиометрии

Название	Обозначение	Единица измерения
поток излучения (radiant flux)	Ф	Вт
irradiance	E	Вт/м^2
radiant intensity	I	Вт/ср
светимость (radiance)	L	Вт/(м^{2}ср)

[5] Стерадиан, ср

Связанная область - фотометрия, отличающаяся от радиометрии только тем, что все измеряется в зависимости от чувствительности человеческого глаза. Результаты, получаемые в радиометрии переводятся в фотометрические умножением на фотометрическую кривую (CIE photometric curve), которая характеризует способность глаза воспринимать свет с разной длиной волны. Исторически сложилось, что базовой единицей в этой области является кандела[6], которая соответствует единице интенсивности излучения в радиометрии.

Таблица 2.2.1.2: соответствие понятий в радиометрии и фотометрии

Радиометрическое понятие, единица измерения	Фотометрическое понятие, единица измерения
поток излучения (radiant flux), Вт	поток светимости (luminous flux), Люмен
irradiance, Вт/м^2	illuminance, Люкс
radiant intensity, Вт/ср	luminous intensity, Кандела
светимость (radiance), Вт/(м^{2}ср)	светимость (luminance), Кандела/м^2

Важный момент, что любой свет - это пучок электромагнитных волн разных длин, значит нам нужно уметь раскладывать такой пучок по какому-нибудь базису для более эффективного вычисления нужных нам величин. Самым популярным базисом является базис RGB (красный, зеленый, синий). Но базис RGB не является базисом пространства всех цветов, которые воспринимает человеческий глаз. Поэтому при вычислениях светимости в виде 3-х RGB чисел могут и будут возникать ошибки, а итоговые результаты будут отличаться от тех, что мы могли бы получить в жизни. Наглядно разница видимого пространства и пространства RGB показана на рис. 2.2.1.2.

[6] от слова candle - свеча

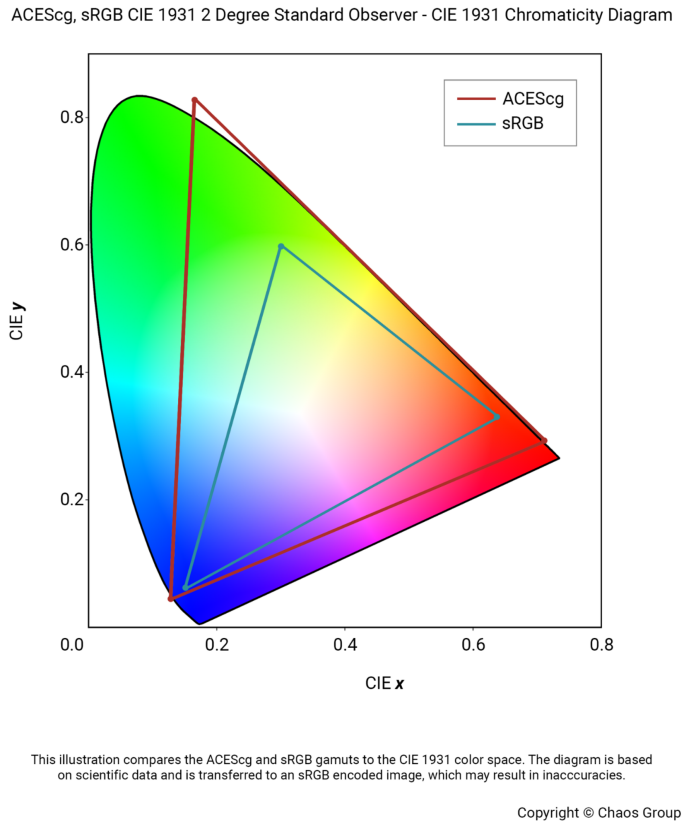


Рисунок 2.2.1.2: разница видимого цветового пространства и пространства RGB (sRGB) [36]

2.2.2 Рассеивание и преломление

Перед тем, как рассматривать основную математику, которая лежит в основе PBR, нужно поговорить еще про несколько вещей. Считаем, что между отображаемым объектом и камерой всегда есть какой-то однородный посредник (homogeneous medium), например, воздух или вода. Свойства этого посредника, конечно, тоже нужно учитывать при расчетах светимости объекта. Если взять в рассмотрение обычный воздух (считаем, что тумана нет), то эффект посредника не будет заметен для объектов, которые находятся вблизи наблюдателя, а значит в этом случае расчеты для

эффекта посредника можно не проводить. Более детально эта тема раскрыта в [1].

Физический смысл рефракции (преломления) состоит в том, что свет в разных средах распространяется с разной скоростью, хотя частота волны неизменна. Такой эффект приводит к тому, что световая волна меняет свое направление распространения при переходе из одной среды в другую. Наглядно эффект показан на рис. 2.2.2.1.

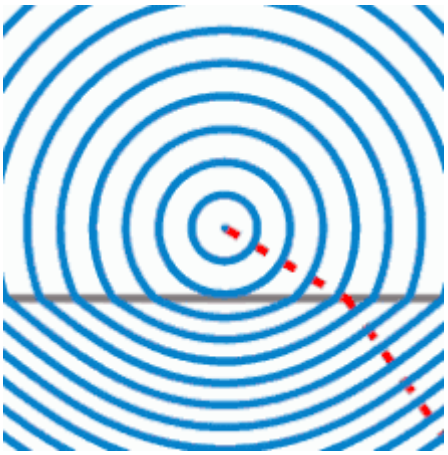


Рисунок 2.2.2.1: преломление в волновой оптике [37]

Отношение скорости света в вакууме к скорости света в среде называется индексом рефракции этой среды (index of refraction, IOR) и обозначается буквой n. Значение для воздуха $n = c/v \approx 1.3$.

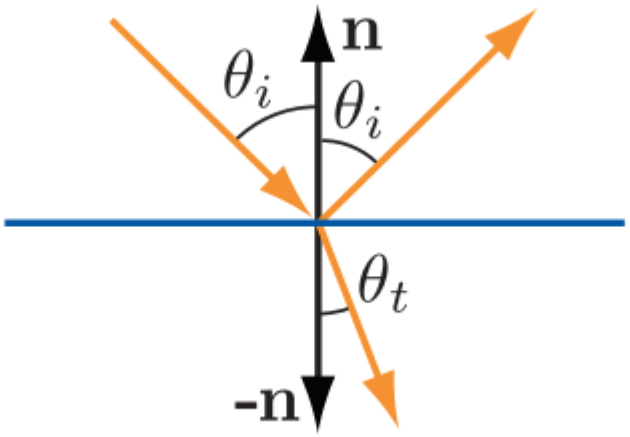


Рисунок 2.2.2.2: связь углов падения, отражения и преломления [1]

Говоря о геометрической оптике, где свет представляется лучом, можно рассмотреть типичную ситуацию отражения-преломления. Угол отраженного луча равен углу падающего луча θ_i. А преломление в общем случае описывается «Законом Снеллиуса» (Snell's law).

$$\frac{sin(\theta_t)}{sin(\theta_i)} = \frac{n_1}{n_2} \quad (1)$$

Чаще всего, конечно, используют другую форму этого уравнения, которая получаются из данного домножением обеих частей на $sin(\theta_i)$, такая манипуляция позволяет избежать отдельного рассмотрения случая с делением на 0.

2.2.3 Поверхность

Еще один нюанс состоит в том, что абсолютно плоских поверхностей в природе не существует. Поверхность имеет свою

геометрию, которую можно представлять как набор неровностей (irregularities). Неровности, которые меньше длины световой волны, никак не будут влиять на поведение этой волны при ее взаимодействии с поверхностью, поэтому такой случай сразу убираем из рассмотрения. Рассматриваемую нами геометрию поверхности можно разделить на 3 класса:

- наногеометрия - размер неровностей $a * \lambda$, где $a \in [1, 100]$ - проявляются волновые эффекты света (дифракция, интерференция);
- микрогеометрия - размер неровностей $\in [100 * \lambda, fragment\ size]$ - имеет место локальная плоскость;
- макрогеометрия (или просто геометрия) - размер неровностей $\in [fragment\ size, \infty]$ - определяет форму тела.

Далее будем вести разговор в большинстве про микрогеометрию, потому что основные визуальные эффекты связаны именно с ней. К наногеометрии вернемся позже. В таком случае мы можем не рассматривать волновые свойства света, а сосредоточиться только на геометрической оптике, в которой свет представляется лучами и которая гораздо проще в вычислениях.

Еще одним важным упрощением будет то, что мы рассматриваем не саму геометрию поверхности, а ее некое статистическое представление, как показано на рис. 2.2.3.1.

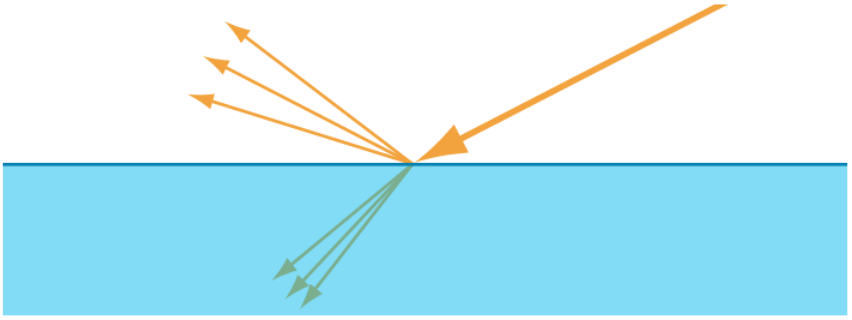


Рисунок 2.2.3.1: статистическое представление поверхности [1]

Т.е. вместо того, чтобы смотреть на геометрию в конкретной точке и точно определять нормаль поверхности в этой точке, будем рассматривать вероятность того, что нормаль в этой точке направлена в нужном направлении, другими словами, смотрим на вероятностное распределение микроструктурных нормалей.

Также нужно учитывать подповерхностное рассеивание (subsurface scattering), которое возникает потому, что преломленный луч рассеивается внутри тела и какая-то часть таким образом рассеянного света может снова выйти из-под поверхности тела.

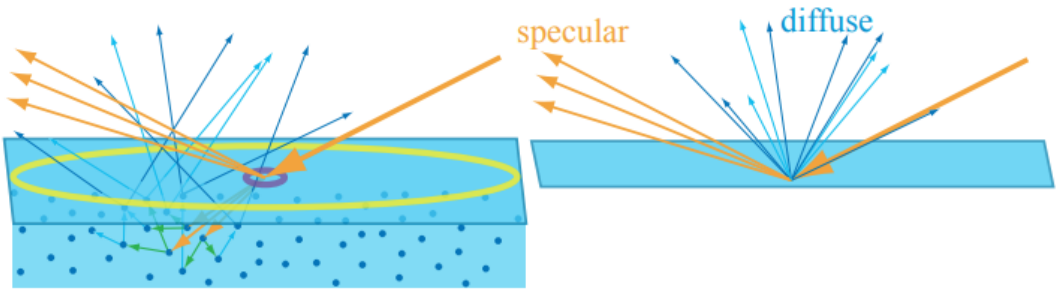


Рисунок 2.2.3.2: слева показано подповерхностное рассеивание, справа привычное разделение на diffuse и specular компоненты светимости [1]

2.2.4 Уравнение рендеринга

Смысл расчета освещения для тела - это вычисление светимости (Radiance) поверхности тела в конкретной точке. Один из самых распространенных способов записи такого расчета в общем случае - уравнение рендеринга. Уравнение было представлено Кайя (Kajiya) в 1986-м году [3]. Широко используются разные формы для записи этого уравнения, мы же будем пользоваться следующей:

$$L_o(p, v) = L_e(p, v) + \int_{l \in \Omega} f(l, v) L_i(p, l)(n \cdot l)^+ dl, \qquad (2)$$

где p - конкретная точка поверхности тела,

v - вектор взгляда (**v**iew vector),

l - вектор света (**l**ight vector),

$(x)^+ = x$, если $x \geq 0$; 0, если $x < 0$;

$(n \cdot l)^+$ - неотрицательное скалярное произведение,

Ω - единичная полусфера над точкой p,

f(l, v) - двулучевая функция отражательной способности (Bidirectional reflectance distribution function) - BRDF,

L(p, s) - функция светимости в точке p, в направлении s,

L_o - исходящая светимость (**o**utgoing),

L_e - произвольная светимость (**e**mitted) - явление флуоресценции,

L_i - входящая светимость (**i**ngoing).

В локальных моделях освещения L_i зависит только от прямых источников света, поэтому интеграл вполне может быть превращен в конечную сумму,

при условии, что источники света не имеют объема (такими источниками являются, например, ambient, directional, point) и их конечное количество. Такие источники света, конечно, являются нефизическими абстракциями, поэтому нужно понимать, что из-за этого может возникать отклонение, влияющее на фотореалистичность изображения.

2.2.5 BRDF f(l, v)

Будем считать, что BRDF одинакова для всей поверхности отрисовываемого тела, т.е. не будем рассматривать случай spatial BRDF (SBRDF).

Количество отраженного света может зависеть от длины волны, этот эффект может быть смоделирован двумя способами:

- длину волны можно передавать в функцию BRDF как дополнительный параметр;
- сделать так, чтобы BRDF возвращал несколько значений для разных длин волн.

Первый подход иногда используется в оффлайн рендеринге, но для real-time рендеринга всегда применяют второй вариант. По сути, для нас это означает, что мы будем воспринимать возвращаемое BRDF значение как RGB вектор.

В качестве BRDF можно брать произвольные функции, проблема в таком подходе лишь в том, что итоговое изображение не всегда будет близко к фотореалистичному.

Для того, чтобы BRDF была физичной, она должна удовлетворять двум условиям:

1. теорема взаимности (Helmholtz reciprocity):

$$f(l,\ v)\ =\ f(v,\ l) \tag{3}$$

2. закон сохранения энергии:

$$\int_{l\in\Omega} f(l,v)(n\cdot l)\ dl\ \in\ [0,\ 1] \tag{4}$$

(может не выполняться для произвольно излучающих поверхностей)

На практике для real-time алгоритмов эти условия могут нарушаться без возникновения ощутимых артефактов, но все равно лучше стараться подбирать аппроксимации, хотя бы частично удовлетворяющие этим условиям.

BRDF вполне может быть константой, такой случай даже имеет свое название - BRDF Ламберта:

$$f\,(l,\ v) = \text{const} = \rho_{ss}\pi^{-1} \tag{5}$$

2.2.6 Построение BRDF и теория микрограней

Теория микрограней основана на моделировании микрогеометрии в качестве множества микрограней (microfacets), каждая микрогрань представляет собой плоскость с единственной нормалью m и такая грань отражает свет в соответствии с функцией micro-BRDF $f_{\mu}(l,\ v,\ m)$. А итоговая BRDF получается как комбинация таких micro-BRDF по всей поверхности тела. Каждую такую грань для простоты расчетов чаще всего

считают идеальным зеркалом Френеля, т.е. грань отражает весь входящий с направления l свет в направлении вектора отражения $r(l,\ m)$ (**r**eflectance vector). Такое приближение используется для вычисления Specular term BRDF. Кроме зеркала Френеля, существуют и другие варианты для построения micro-BRDF. Например, для воспроизведение эффекта подповерхностного рассеивания изспользуют Diffuse micro-BRDF, а для учета волновых свойств света Diffraction micro-BRDF.

BRDF может быть вычислено по micro-BRDF с помощью следующего соотношения, детали вывода в [2, 4]:

$$f(l,v) = \int_{m\in\Omega} \frac{f_{\mu}(l,v,m)G_2(l,v,m)D(m)(m\cdot l)^{+}(m\cdot v)^{+}}{|n\cdot l||n\cdot v|} dm, \qquad (6)$$

где $f_{\mu}(l,\ v,\ m)$ - micro-BRDF,

$G_2(l,v,m)$ - функция маскирования-затенения (masking-shadowing function),

$D(m)$ - функция распределения нормалей (normal distribution function - NDF).

Т.е. для построения функции BRDF, нам нужно определить 3 функции, которые являются множителями в произведении под интегралом. Но прежде чем приступать к рассмотрению каждой отдельной функции, необходимо обозначить еще несколько геометрических эффектов, которые влияют на видимое представление поверхности тела.

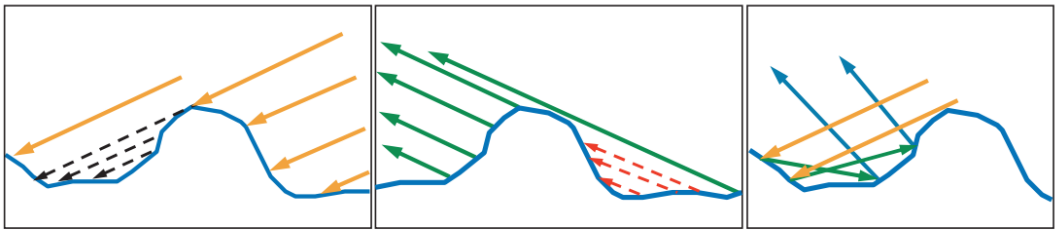


Рисунок 2.2.6.1: эффекты затенения, маскировки и множественного отражения [1]

На рис. 2.2.6.1 слева показан эффект затенения (shadowing), он заключается в том, что свет вообще не попадает на часть поверхности тела. В середине проиллюстрирован эффект маскировки (masking), заключающийся в скрытии от наблюдателя части поверхности тела. Справа показан эффект множественного отражения (interreflection) от поверхности тела.

Рассмотрение подынтегральных функций начнем с NDF. Функция распределения нормалей (Normal Distribution Function) $D(m)$ представляет собой статистическое распределение нормалей микрограней по поверхности микрогеометрии.

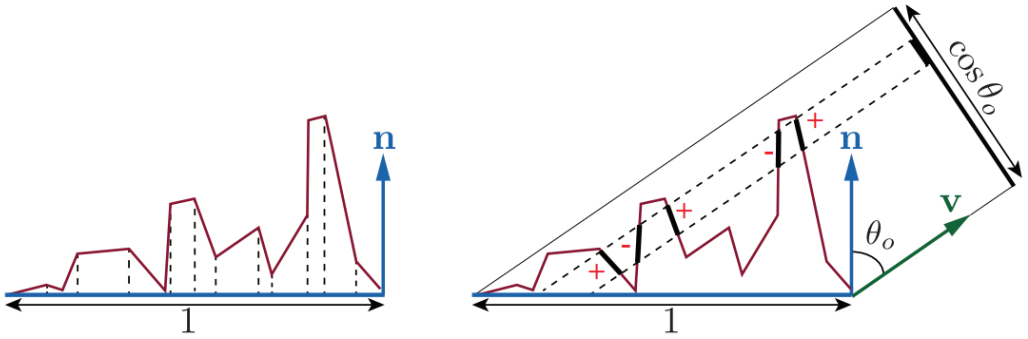


Рисунок 2.2.6.2: свойства NDF. Рисунки сделаны двумерными для простоты восприятия [1]

Строго NDF определяется через следующие равенства:

$$\int_{m\in\Theta} D(m)dm = S_{\mu} \tag{7}$$

Где θ - единичная сфера с центром в точке поверхности;

смысл уравнения (7) в том, что интегрируя NDF по сфере, получаем площадь микроповерхности (сумма площадей микрограней).

$$\int_{m\in\Theta} D(m)(n \cdot m)dm = 1 \tag{8}$$

Интегрируя NDF с учетом скалярного произведения по сфере, получаем площадь макроповерхности (плоского участка), предполагаем что площадь участка макроповерхности равна 1. Случай показан на рис. 2.2.6.2 слева.

$$\int_{m\in\Theta} D(m)(v \cdot m)dm = v \cdot m \tag{9}$$

Обобщая формулу (8), получаем соотношение (9) для вектора v. Этот случай показан рис. 2.2.6.2 справа.

$$\int_{m\in\Theta} G_1(v, m)D(m)(v \cdot m)^{+}dm = v \cdot m \tag{10}$$

Можно заметить, что в формуле (9) скалярное произведение вполне может быть меньше 0, потребность в таком случае может быть обоснована геометрически с помощью рис. 2.2.6.2 справа. Чтобы убрать из

рассмотрения случай с $(v \cdot m) < 0$, нам понадобится новая функция $G_1(v, m)$.

$G_1(v, m)$ - функция маскировки (masking function), она дает нам информацию о том, какая часть микрограней с нормалью m видима в направлении вектора v, что позволяет применить функцию $(x)^+$ к скалярному произведению и получить форму (10). Функцию $G_1(v, m)D(m)$ называют распределением видимых нормалей.

В публикации Гейца (Heitz) [2] были исследованы различные варианты для реализации masking function, и сделаны следующие выводы. В рассмотренной им литературе соотношению (10) удовлетворяют только две функции маскировки:

- Smith masking function, выведенная из нормального распределения Гаусса [5];
- Torrance-Sparrow “V-cavity” function.

И значит только они являются математически верными. Также Гейц показал, что из этих двух функция Смита лучше приближает поведение случайных микроповерхностей, а еще функция Смита удовлетворяет свойству normal-masking independence, при котором $G_1(v, m)$ не зависит от направления m, если $m \cdot v \geq 0$.

Функция Смита имеет следующую форму:

$$G_1(v, m) = \frac{\chi^+(m \cdot v)}{1 + \Lambda(v)} \quad (11)$$

Где $\chi^+(x)$ - положительная характеристическая функция.

$$\chi^{+}(x) = \begin{cases} 1, \text{если } x > 0 \\ 0, \text{если } x \leq 0 \end{cases} \quad (12)$$

А функция $\Lambda(.)$ выводится отдельно для каждой NDF, алгоритм вывода описан в статьях Walter et al. [4] and Heitz [2].

$G_2(l, v, m)$ - функция маскирования-затенения (masking-shadowing function) учитывает не только эффект маскировки, но и эффект затенения, эта функция выводится из $G_1(m,\ v)$. Есть несколько используемых форм:

- разделяемая форма (Separable form):

$$G_2(l, v, m) = G_1(v,\ m)G_1(l,\ m) \quad (13)$$

  Используя эту форму, мы предполагаем, что затенение и маскировка - это несвязанные (не коррелирующие) эффекты, хотя в реальности это не так. Поэтому эта форма может вызвать “перезатенение” (overshadowing) итоговой BRDF;

- Smith height-correlated masking-shadowing function:

$$G_2(l, v, m) = \frac{\chi^{+}(m \cdot v)\chi^{+}(m \cdot l)}{1 + \Lambda(v) + \Lambda(l)} \quad (14)$$

  Эта форма является лучшим из известных и доступных в real-time рендеринге вариантов за счет своей относительной вычислительной простоты и хорошей степени аппроксимации реальных явлений.

Функция $f_{\mu}(l, v, m)$ определяется для микрограней по аналогии с BRDF для поверхности, и для физичности должна удовлетворять тем же ограничениям, что накладываются на обычную BRDF.

Далее рассмотрим некоторые результаты, связанные с построением BRDF, но перед этим отвлечемся на введение вспомогательных функций.

2.2.7 Функция отражения

Вернемся к рассмотрению оптической стороны вопроса. Взаимодействие света с плоским интерфейсом между двумя средами удовлетворяет уравнениям Френеля. Для конкретной поверхности уравнения Френеля могут быть записаны в виде функции отражения $F(\theta_i)$, которая позволяет вычислить, какая часть света была отражена (1 - отражен весь свет, 0 - нет отраженного света), и которая зависит только от угла падения света. Значение $F(\theta_i)$ варьируется на видимом спектре, поэтому будем считать, что $F(\theta_i)$ возвращает RGB вектор.

Запишем некоторые свойства функции $F(\theta_i)$:

1. когда $\theta_i = 0$, т.е. свет падает перпендикулярно плоскости поверхности, значение $F(\theta_i) = F(0) = F_0$ является параметром вещества, который определяет specular color этого вещества;
2. для всех длин волн выполняется $\lim_{\theta_i \to \pi/2} F(\theta_i) = 1$.

Эффект возрастания отражения при углах близких к $\pi/2$ в публикациях, связанных с визуализацией, часто называют эффектом Френеля.

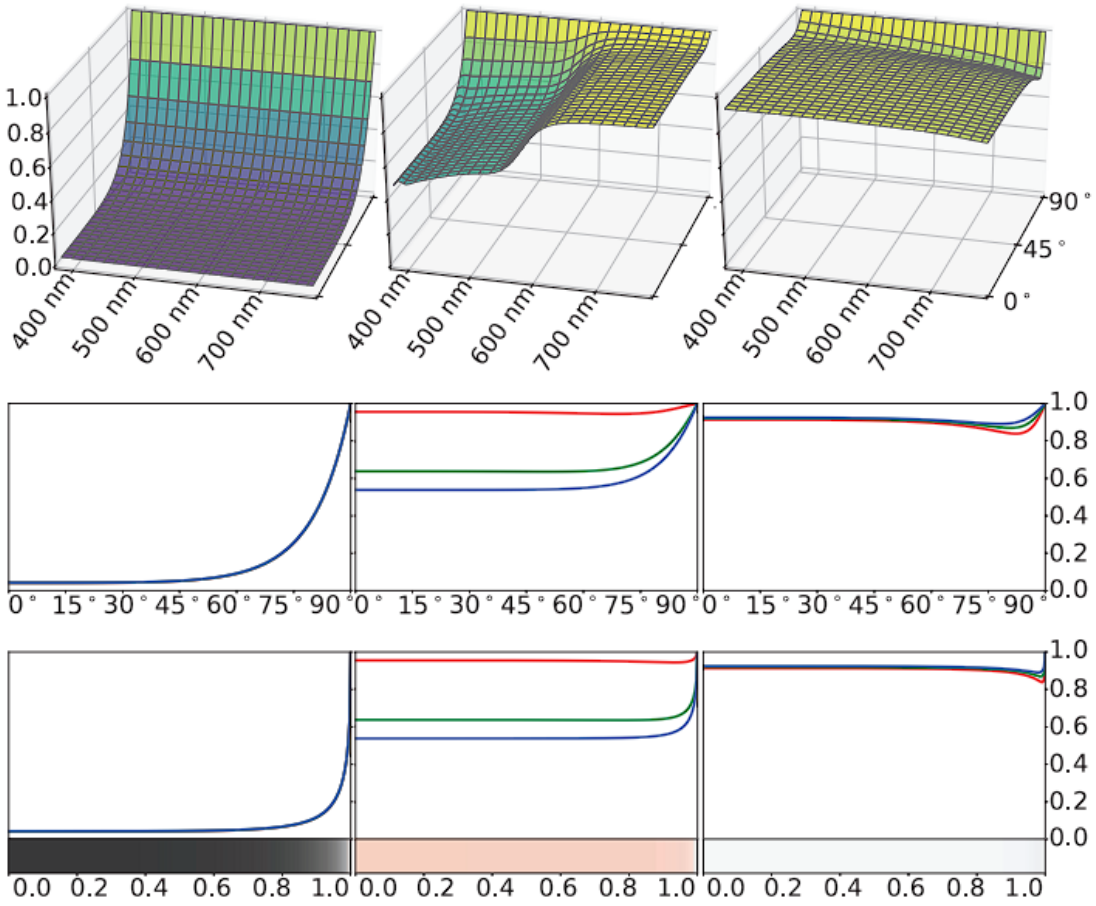


Рисунок 2.2.7.1: в первой строке функция отражения для поверхностей стекла, меди и алюминия (слева направо). Во второй и третьих строках показаны двумерные представления этих функций при фиксированных длинах волн [1]

Но прямое использование уравнений Френеля в рендеринге сопряжено с рядом трудностей. Например, для их вычисления нужно знать индексы рефракции (IOR), которые могут быть комплексными числами и зависеть от длины волны.

Поэтому используются аппроксимации функции отражения, самой популярной из которых является аппроксимация Шлика:

$$F(n,\ l)\ \approx F_{0}\ +\ (1\ -\ F_{0})(1\ -\ \ (n\ \cdot\ l)^{+})^{5} \qquad (15)$$

Или в обобщенной форме:

$$F(n,\ l) \approx F_0 + (F_{90} - F_0)(1\ -\ (n \cdot l)^{+})^{\frac{1}{p}} \qquad (16)$$

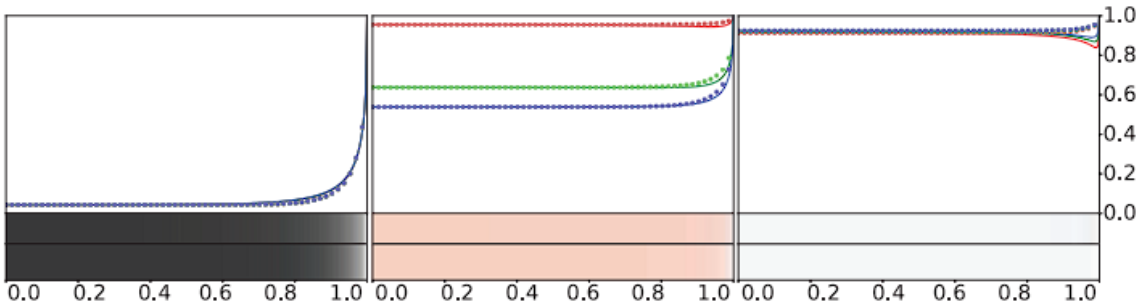


Рисунок 2.2.7.2: аппроксимация Шлика для материалов с рис. 2.2.7.1 показана точками [1]

2.2.8 Примеры BRDF и реализация

Теперь рассмотрим сами функции, получающиеся исходя из изложенных выше рассуждений.

В реализации из основного уравнения рендеринга имеем: [ссылка]

```
void main()
{
    vec3 v = normalize(fragView);
    vec3 n = normalize(fragNormal);
    vec3 rho = texture(texSampler, fragTexCoord).xyz;

    vec3 lighting = vec3(0.0);
    for (int i = 0; i < lights.directionalCount; i++)
    {
        vec3 l = lights.directional[i].direction.xyz;
        vec3 brdf;
```

```
        getBRDF(n, v, l, rho, brdf);

        lighting += lights.directional[i].color.xyz * max(0,
dot(l, n)) * brdf;
    }

    for (int i = 0; i < lights.pointCount; i++)
    {
        vec3 pointOffset = lights.point[i].position.xyz -
fragPosition;
        float sqrLen = max(dot(pointOffset, pointOffset),
1.0);
#if defined(LINEAR_FALOFF)
        float pointIntensity = 1 / sqrt(sqrLen);
#elif defined(SQR_FALOFF)
        float pointIntensity = 1 / sqrLen;
#endif

        vec3 l = normalize(pointOffset);
        vec3 brdf;
        getBRDF(n, v, l, rho, brdf);

        lighting += lights.point[i].color.xyz * pointIntensity
* max(0, dot(l, n)) * brdf;
    }

    outColor = vec4(lighting, 1.0) + lights.ambient.color;
}
```

BRDF для поверхности за счет линейности интеграла часто раскладывают на несколько слагаемых (terms), каждое из которых вычисляется, руководствуясь своей логикой для воспроизведения определенных визуальных эффектов. Рассмотрим несколько видов таких слагаемых подробнее.

На практике будем пользоваться только Diffuse и Specular слагаемыми, потому что они позволяют добиться хорошего баланса между итоговой сложностью вычислений и фотореалистичностью получаемого результата.

Функция для вычисления BRDF у нас имеет следующую структуру: [ссылка]

```
void getBRDF(in vec3 n, in vec3 v, in vec3 l, vec3 rho, out
vec3 value)
{
    ...
    value = specular + diffuse;
}
```

2.2.9 Specular term

Тот факт, что $f_{\mu}(l, v, m) = 0$ для всех $h \neq m$[7] существенен при выводе specular term, потому что интеграл превращается в подынтегральную функцию от аргумента $h = m$. Детали вывода можно найти в публикациях Walter et al. [4], Heitz [2], and Hammon [6]. Мы же воспользуемся результатом.

$$f_{spec}(l, v) = 0.25 * F(h, l)G_2(l, v, h)D(h)|n \cdot l|^{-1}|n \cdot v|^{-1} \qquad (17)$$

В реализации это выглядит так: [ссылка]

```
void getBRDF(in vec3 n, in vec3 v, in vec3 l, vec3 rho, out
vec3 value)
{
    vec3 h = normalize(l + v);
```

[7] h - half vector, вычисляется как h = l + v / |l + v|

```
    vec3 F;
    getFresnelReflectance(n, l, F);
    float G;
    getMaskingShadowing(l, v, h, G);
    float D;
    getNormalDistribution(n, h, D);

    vec3 specular = F * G * D / (4 * abs(dot(n, l)) *
abs(dot(n, v)));
    vec3 diffuse = ...

    value = specular + diffuse;
}
```

Мы уже рассмотрели способы аппроксимации функции отражения $F(h,\ l)$, используем аппроксимацию Шлика (15). [ссылка]

```
void getFresnelReflectance(in vec3 n, in vec3 l, out vec3
reflectance)
{
     float NdotL = dot(n, l);
     reflectance = material.F0.xyz + (1 - material.F0.xyz) *
pow(1 - max(0, NdotL), 5);
}
```

Теперь посмотрим на варианты для вычисления 2-х других функций из формулы (17).

Мы будем обсуждать в основном изотропные NDF, которые и используются в большей части современных приложений. Изотропными (isotropic) NDF называются такие $D(\theta_m)$, значение которых зависит только угла θ_m между нормалью поверхности тела в точке n и микронормалью m. В общем случае D(n, m) называют анизотропными (anisotropic).

Исторически первой функцией распределения нормалей (NDF), используемой в теории микрограней является Beckmann NDF:

$$D(m) = \frac{\chi^{+}(n \cdot m)}{\pi \alpha_b^2 (n \cdot m)^4} exp\left(\frac{(n \cdot m)^2 - 1}{\alpha_b^2 (n \cdot m)^2}\right) \quad (18)$$

Параметр $\alpha_b \in (0, 1]$ в данном случае описывает грубость поверхности.

И в реализации: [ссылка]

```
void getNormalDistribution(in vec3 n, in vec3 m, out float
normalDistribution)
{
    float NdotM = dot(n, m);
    float sqrAlpha = material.alpha * material.alpha;

    float sqrNdotM = NdotM*NdotM;
    normalDistribution = float(NdotM > 0) * exp((sqrNdotM -
1)/(sqrAlpha * sqrNdotM)) / (PI * sqrAlpha * sqrNdotM *
sqrNdotM);
}
```

Руководствуясь работой Гейца [2], формо-независимыми (shape-invariant) NDF будем называть такие изотропные NDF, для которых эффект варьирования параметра грубости эквивалентен растяжению или сжатию микроповерхности (микрограней). Для формо-независимых NDF вывод функции $\Lambda(...)$ для вычисления G_2 может быть упрощен. Для них $\Lambda(...)$ может быть представлена в виде функции от единственной переменной a [2]:

$$a = \frac{n \cdot s}{\alpha \sqrt{1 - (n \cdot s)^2}}, \quad (19)$$

где, вместо s можно использовать либо v, либо l.

Для Beckmann NDF $\Lambda(a)$ имеет следующий вид:

$$\Lambda(a) = \frac{erf(a)-1}{2} + \frac{1}{2a\sqrt{\pi}} exp(-a^2), \quad \text{где } erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{20}$$

Но в связи с тем, что тут должна быть вычислена функция $erf(a)$, обычно используют аппроксимацию, которая достаточно хорошо приближает исходную функцию:

$$\Lambda(a) \approx \begin{cases} \frac{1-1.259a+0.396a^2}{3.535a+2.181a^2}, & \text{если } a < 1.6 \\ 0, & \text{если } a \geq 1.6 \end{cases} \tag{21}$$

Сравнение графиков этих функций приведено на рис. 18.

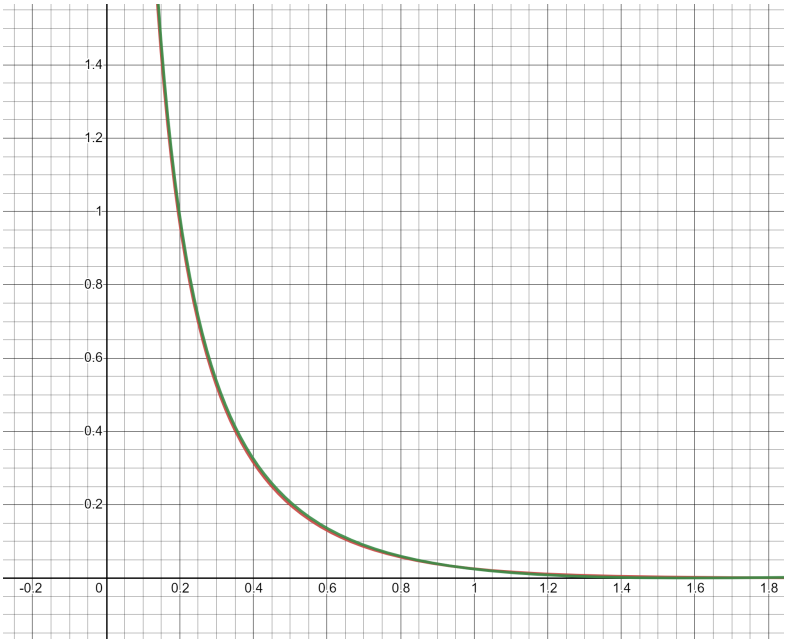


Рисунок 2.2.9.1: красным показан оригинал, зеленым - аппроксимация [38]

В реализации, учитывая возможность использовать форму (13), имеем: [ссылка]

```
void getMaskingShadowing(in vec3 l, in vec3 v, in vec3 m, out
float maskingShadowing)
{
    float MdotV = dot(m, v);
    float MdotL = dot(m, l);
    float sqrMdotV = MdotV*MdotV;
    float numerator = float(MdotV > 0 && MdotL > 0);

    float a = MdotV / (material.alpha * sqrt(1 - sqrMdotV));
    float sqrA = a*a;
    float denominator = 1 + 2 * float(a < 1.6) * (1 - 1.259*a
+ 0.396*sqrA)/(3.535*a + 2.181*sqrA);

#if defined(LambdaSqr)
    float lambda = (denominator - 1) / 2;
    denominator = denominator + lambda * lambda;

#endif

    maskingShadowing = numerator / denominator;
}
```

Самая популярная функция NDF, используемая на данный момент в приложениях - GGX. Она была впервые представлена Блинном (Blinn) [7] в 1977-м году. Спустя 30 лет была переоткрыта Волтером и другими (Walter et al.) [4] и представлена под названием “распределение GGX”. Само распределение выглядит следующим образом:

$$D(m) = \frac{\chi^{+}(n\cdot m)\alpha_g^2}{\pi(1+(n\cdot m)^2(\alpha_g^2-1))^2} \qquad (22)$$

Параметр $\alpha_g \in [0, 1)$ тут тоже описывает грубость поверхности по аналогии с Beckmann NDF.

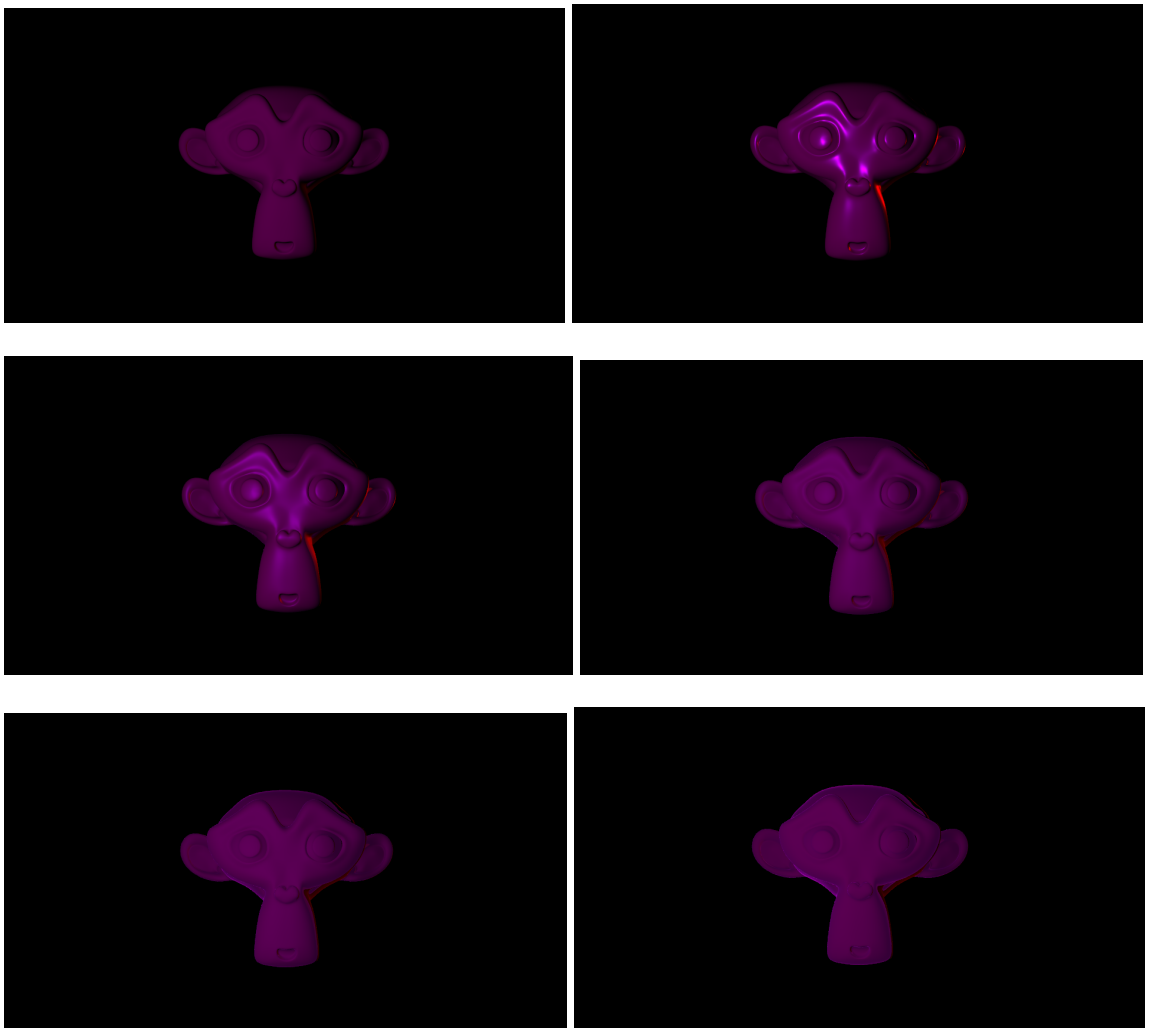


Рисунок 2.2.9.1: варьирование параметра грубости GGX. Слева направо, сверху вниз: $\alpha_g = 0,\ 0.1,\ 0.25,\ 0.5,\ 0.75,\ 1$

В реализации: [ссылка]

```
void getNormalDistribution(in vec3 n, in vec3 m, out float
normalDistribution)
{
    float NdotM = dot(n, m);
    float sqrAlpha = material.alpha * material.alpha;
```

```
    float temp = 1 + NdotM * NdotM * (sqrAlpha - 1);
    normalDistribution = float(NdotM > 0) * sqrAlpha / (PI *
temp * temp);
}
```

GGX также является формо-независимой функцией, так что $\Lambda(a)$ может быть представлена в следующем виде:

$$\Lambda(a) = \frac{-1+\sqrt{1+\frac{1}{a^2}}}{2} \tag{23}$$

В реализации имеем: [ссылка]

```
void getMaskingShadowing(in vec3 l, in vec3 v, in vec3 m, out
float maskingShadowing)
{
    float MdotV = dot(m, v);
    float MdotL = dot(m, l);
    float sqrMdotV = MdotV*MdotV;
    float numerator = float(MdotV > 0 && MdotL > 0);

    float denominator = sqrt(1 + material.alpha *
material.alpha * (1 - sqrMdotV) / sqrMdotV);

#if defined(LambdaSqr)
    float lambda = (denominator - 1) / 2;
    denominator = denominator + lambda * lambda;

#endif

    maskingShadowing = numerator / denominator;
}
```

До этого мы рассматривали изотропные NDF’ы, которые симметричны относительно нормали n и могут быть записаны в виде функции от одной переменной θ (угол между n и m), такие функции являются частным случаем анизотропных NDF.

Но на практике анизотропные NDF зачастую получаются обобщением существующих изотропных вариантов. Обобщение для Beckmann NDF выглядит так:

$$D(m) = \frac{\chi^+(n\cdot m)}{\pi\alpha_x\alpha_y(n\cdot m)^4} exp\left(-\frac{\frac{(t\cdot m)^2}{\alpha_x^2}+\frac{(b\cdot m)^2}{\alpha_y^2}}{(n\cdot m)^2}\right) \quad (24)$$

$$a = \frac{n\cdot s}{\sqrt{\alpha_x^2(t\cdot s)^2+\alpha_y^2(b\cdot s)^2}}, \quad (25)$$

где $\alpha_x, \alpha_y \in (0, 1]$ - параметры грубости в направлениях tangent (t) и bitangent (b) векторов в точке поверхности тела соответственно.

Для GGX ситуация аналогичная:

$$D(m) = \frac{\chi^+(n\cdot m)}{\pi\alpha_x\alpha_y\left(\frac{(t\cdot m)^2}{\alpha_x^2}+\frac{(b\cdot m)^2}{\alpha_y^2}+(n\cdot m)^2\right)^2} \quad (26)$$

2.2.10 Diffuse term

Для расчета этого слагаемого BRDF обычно применяется модель Ламберта, самый простой случай будет выглядеть так:

$$f_{diff}(l, v) = (1 - F(n, l))\frac{\rho_{ss}}{\pi} \quad (27)$$

И в реализации: [ссылка]

```
void getBRDF(in vec3 n, in vec3 v, in vec3 l, vec3 rho, out
vec3 value)
{
    ...
    vec3 specular = ...
    vec3 diffuse = (1 - F) * rho / PI;

    value = specular + diffuse;
}
```

Ширли (Shirley) и его коллеги предложили вариант парного диффузного слагаемого для плоских поверхностей, в нем учитывается закон сохранения энергии и теорема взаимности [8]. В выводе используется аппроксимация Шлика для функции отражения [9].

$$f_{diff}(l, v) = (1 - F_0)\frac{21\rho_{ss}}{20\pi}(1 - (1 - (n \cdot l)^+)^5)(1 - (1 - (n \cdot v)^+)^5) \quad (28)$$

2.2.11 Другие слагаемые

Specular term не учитывает эффект множественного отражения, поэтому может наблюдаться эффект перезатенения (overshadowing), который более заметен на грубых поверхностях.

Multibounce term используется для того, чтобы учесть эффект множественного отражения. Пример такого слагаемого и его вывод можно найти в работе Imageworks [10]. Хотя на практике это слагаемое

используется редко из-за малого вклада в итоговое изображение, полученные Imageworks результаты можно пронаблюдать на рис. 2.2.11.1.

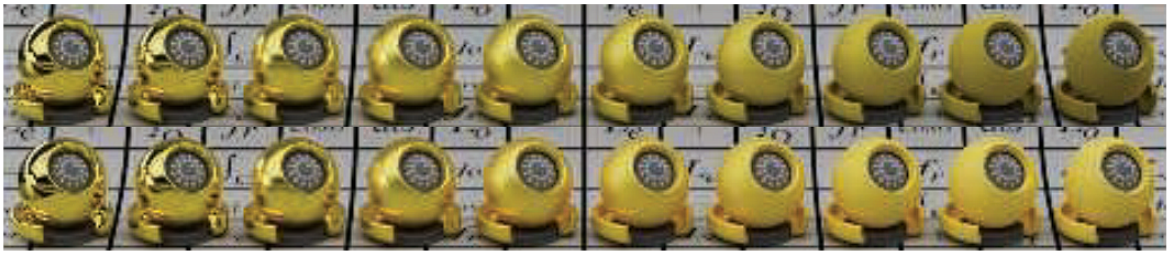


Рисунок 2.2.11.1: сверху multibounce term не учитывается, а снизу учитывается, в сравнении можно наблюдать эффект перезатенения верхней строки, который более заметен при параметрах грубости близких к 1 [1]

Specular term также не учитывает волновые свойства света и полностью полагается на геометрическую оптику.

Wave optics term позволяет исправить это. В этом случае мы сталкиваемся с наногеометрией, которая может способствовать проявлению волновых свойств света. Интерес для нас представляют явления дифракции и интерференции. Хотя в сообществе до недавнего времени было принято соглашение, что эти явления почти никак не влияют на итоговую картинку, в последнее время ситуация изменилась в связи с исследованиями [11].

Дифракция лучше всего наблюдается на поверхностях с периодической наногеометрией, например, на DVD дисках или у некоторых насекомых, см. рис. 20. В [12] авторы предложили модель, которая комбинирует теорию дифракции и теорию микрограней.

Рисунок 2.2.11.2: пример дифракции

Интерференция была изучена в работе [13], и существуют методы для воспроизведения этого эффекта в real-time, об используемом в Call Of Duty: Infinite Warfare методе можно прочитать в [14].

Рисунок 2.2.11.3: кожа без учета и с учетом интерференции [1]

2.3 Архитектура приложения

На момент последнего коммита (22 мая 2022 года) в проекте было 2539 строк кода не считая шейдеров, ~75% этих строк обеспечивали логику работы с API Vulkan. Схему VulkanApplication можно найти в Приложении В. Здесь же мы подробнее остановимся на общей архитектуре приложения.

Задача класса VulkanApplication - отрисовка настраиваемой трехмерной сцены, обеспечение взаимодействия сцены с пользователем (обработка нажатий на клавиши управления, движений мыши). В этом классе находится вся логика, отвечающая за настройку и взаимодействие с API Vulkan.

Класс VulkanApplication ссылается на экземпляр класса Scene, который хранит в себе всю необходимую информацию о настроенной сцене. В том числе Scene хранит структуру LightSources, класс Camera, список экземпляров класса SceneObject и список экземпляров класса Vertex. Диаграмма связанных со Scene классов показана на рис. 2.3.2. Здесь стоит отметить пару моментов, связанных с оптимизацией:

- для оптимизации памяти при хранении трехмерной модели используется буфер индексов indices;
- массивы vertices и indices содержат информацию обо всех объектах вместе и последовательно. Они имеют аналогичную структуру, устройство массива vertices показано на рис. 2.3.1. Информация о том, какому объекту принадлежит информация о вершине\индексе содержится в экземпляре класса SceneObject конкретного объекта. Все это сделано для того, чтобы можно было нарисовать все объекты сцены с помощью одного вызова отрисовки. Уменьшение количества вызовов отрисовки позволяет сократить количество обращений к

драйверу видеокарты, что положительно сказывается на производительности графики.

Рисунок 2.3.1: схема массива vertices, который хранит информацию о вершинах нескольких объектов. Разными цветами показаны разные объекты сцены

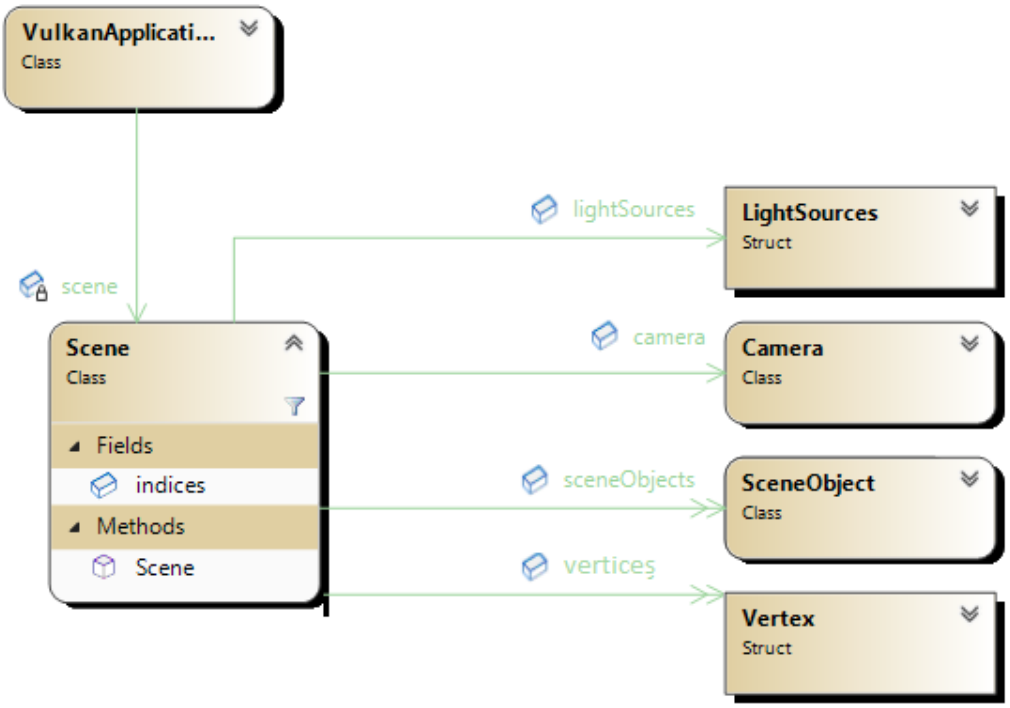


Рисунок 2.3.2: диаграмма классов, связанных со Scene

Рассмотрим каждый класс и структуру, используемые классом Scene, поподробнее. Начнем со структуры LightSources. Эта структура используется для хранения и передачи на графический ускоритель информации обо всех источниках света, размещенных на сцене.

Сейчас поддерживается несколько типов источников света (PointLight, AmbientLight, DirectionalLight), их иерархия и поля показаны на рис. 2.3.3.

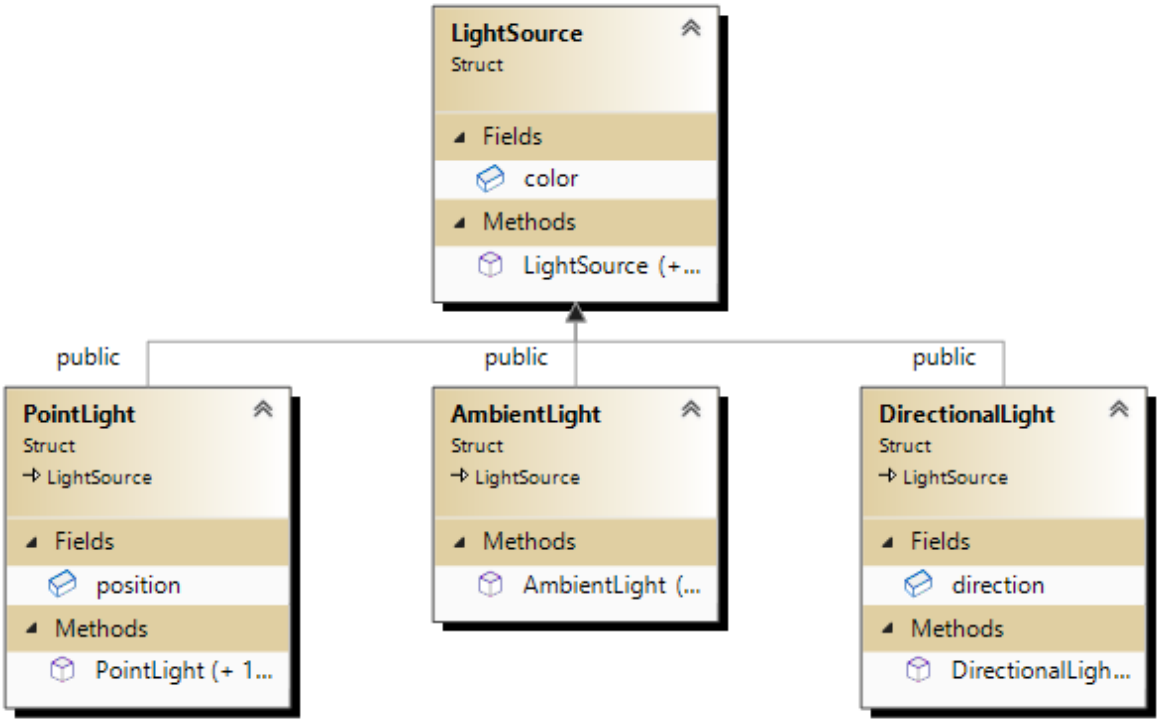


Рисунок 2.3.3: иерархия источников света

В сцене может быть настроен один AmbientLight, какое-то количество PointLight и DirectionalLight. Информация об этих источниках и хранится в структуре LightSources, при этом эта структура удобно расположена в памяти для прямой передачи ее на графический ускоритель, даже учитывая выравнивание. Также в этом случае нам удалось обойтись без динамического выделения памяти на графическом ускорителе путем единовременного выделения такого количества памяти, которое потребуется, если в сцене будет настроено максимальное число источников света всех типов. Константы для определения максимального числа источников света задаются через код, хотя их инициализация вполне

может быть вынесена на стадию создания сцены (момент выделения памяти под LightSources).

Класс Camera хранит в себе все параметры (позицию, систему координат в виде 3-х векторов, углы pitch и yaw) и настройки камеры (например, FOV), а также имеет методы, позволяющие изменять эти параметры, что помогает обеспечить взаимодействие пользователя с камерой. Конечно, можно обойтись хранением меньшего количества информации для сохранения позиции и ориентации камеры в пространстве, но тогда нам придется жертвовать удобством в управлении (может возникать Эйлеров замок (Gimbal lock) или появляться крен камеры). Выбор в данном случае был сделан в пользу удобства управления.

Класс SceneObject хранит информацию об объекте сцены, зависимости можно видеть на рисунке 2.3.4.

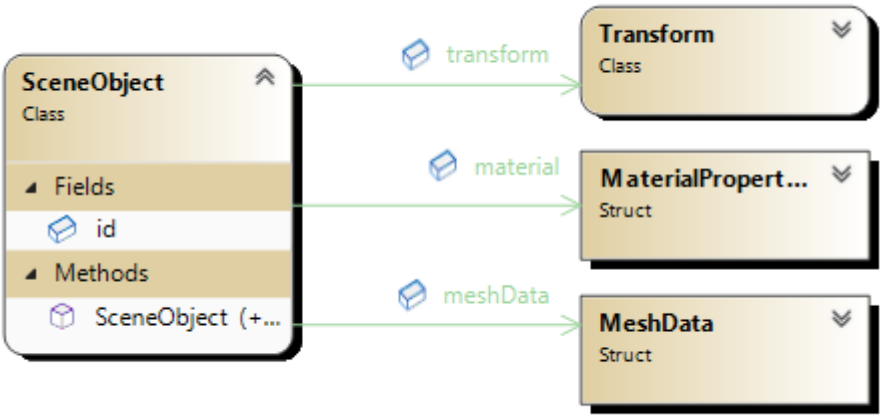


Рисунок 2.3.4: зависимости SceneObject

Каждый объект сцены имеет уникальный идентификатор (id), который назначается в момент его создания. Также у каждого объекта сцены есть

компонент класса Transform, в котором содержится вся логика позиционирования и ориентирования объекта в пространстве.

Сам объект класса Transform хранит в себе позицию как вектор, ориентацию как кватернион и растяжение по осям как вектор. Этой информации достаточно, чтобы можно было представить любое аффинное преобразование пространства, а следовательно любое аффинное преобразование тела. Помимо Transform в SceneObject еще содержится информация о материале объекта в виде экземпляра структуры MaterialProperties. В этой структуре хранится вся информация, необходимая для работы с моделями освещения (Например, параметры для PBR). Последним компонентом SceneObject является MeshData, в ней хранится информация о геометрии объекта сцены, а именно первый индекс и количество индексов в массиве индексов indices.

Для того, чтобы для каждого объекта на видеокарту передавалась своя model matrix, реализована система динамических буферов общего назначения (Dynamic uniform buffer), подробнее о ее реализации написано в отчете по преддипломной практике. Такая система часто используется при разработке рендеров для игровых движков.

Структура Vertex содержит информацию о вершинах моделей объектов сцены, по большому счету, структура нужна только для того, чтобы удобно передавать эту информацию на графический ускоритель для последующего ее использования в шейдерах.

На рис. 2.3.5 показана визуализация сцены с тремя объектами сцены (Сюзанна, низкополигональная сфера, высокополигональная сфера) и двумя точечными источниками света (фиолетовый и красный).

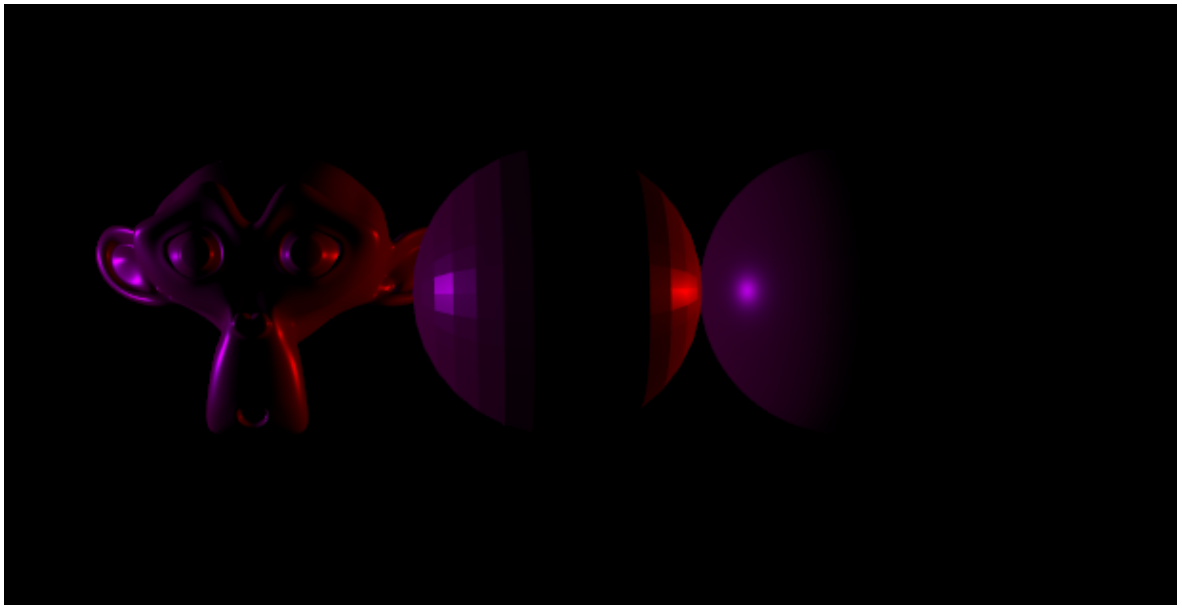


Рисунок 2.3.5: Сюзанна, низкополигональная сфера, высокополигональная сфера

Замеры производительности рендера показывают в среднем в районе 1500 fps, что является хорошим результатом, скриншот с замеров приведен на рис. 2.3.6. В сцене находилась только высокополигональная сфера, один направленный источник света, два точечных и один ambient.

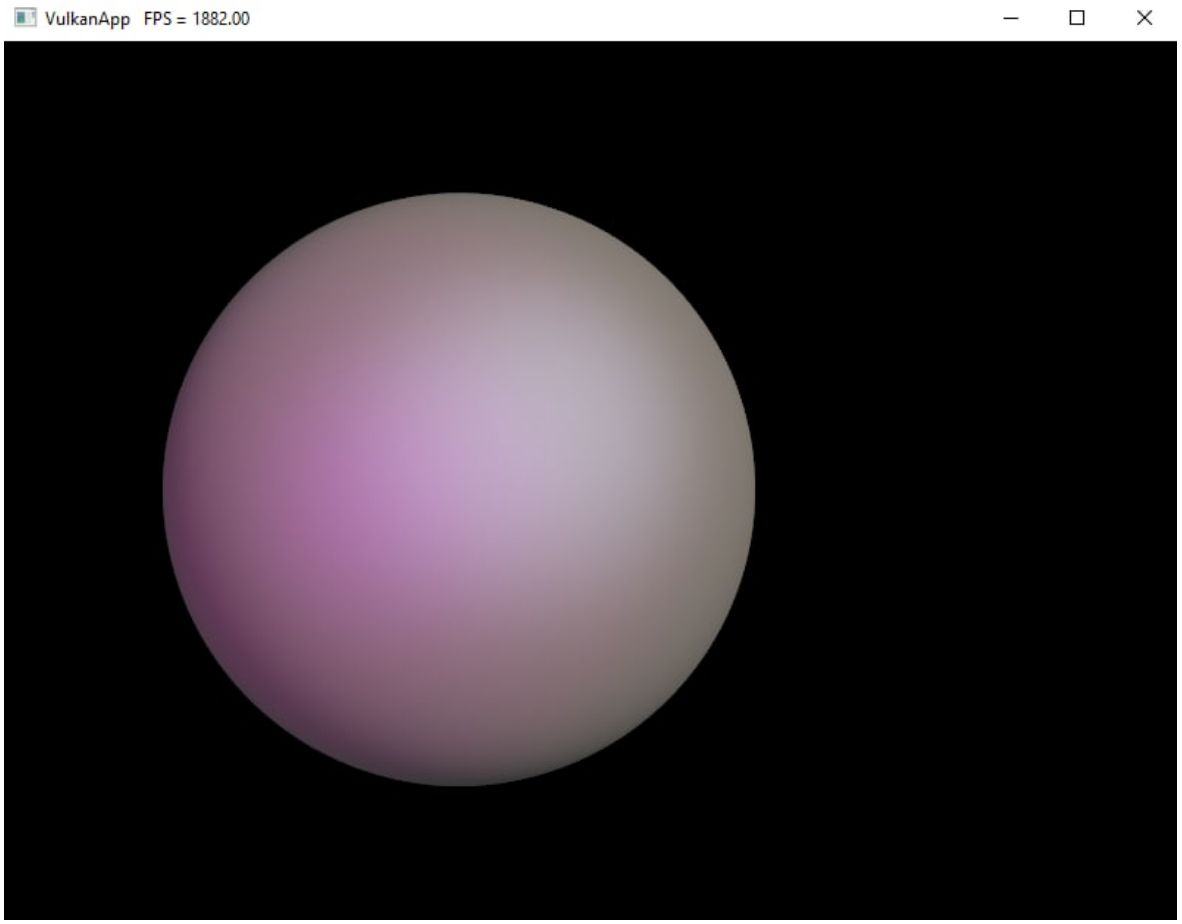


Рисунок 2.3.6: Замеры производительности рендера в обычном состоянии

Всего в рендере было реализовано четыре полноценные модели освещения:

- PBR (возможна настройка через ключевые слова), всего вариантов $2^3 = 8$;
- Phong;
- Blinn-Phong;
- Minnaert.

Т.е. всего 11 вариантов шейдеров. Для тестирования визуальной стороны моделей освещения собрана сцена, показанная на рис. 2.3.7.

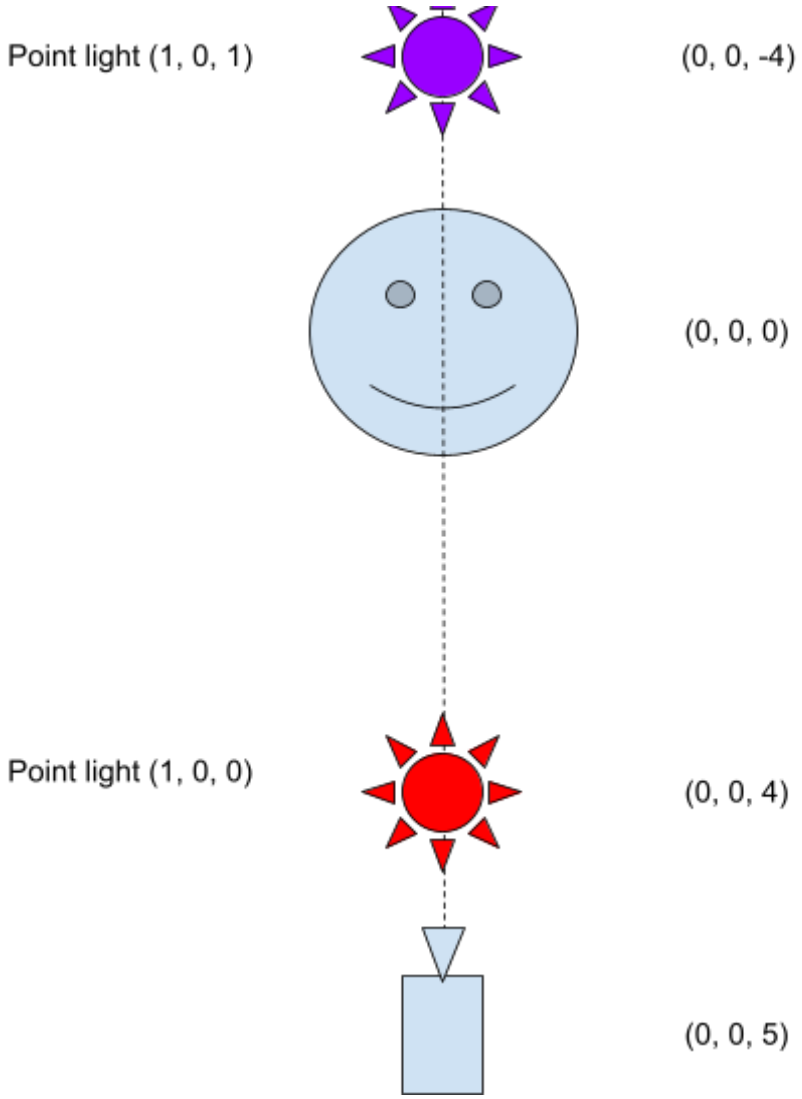


Рисунок 2.3.7: сцена для тестирования моделей освещения

Полученные результаты представлены на рисунке 2.3.8. Видно, что наиболее близким к фотореализму являются варианты PBR. При этом модели PBR могут очень гибко настраиваться за счет варьирования параметров в MaterialProperties, что тоже способствует повышению фотореалистичности. Все это в совокупности и дает огромный рост популярности PBR по сравнению с классическими моделями в последнее время. Классические модели становятся все более нишевыми и используются в основном в мобильном сегменте или инди разработке, где важно подобрать особенный визуальный стиль.

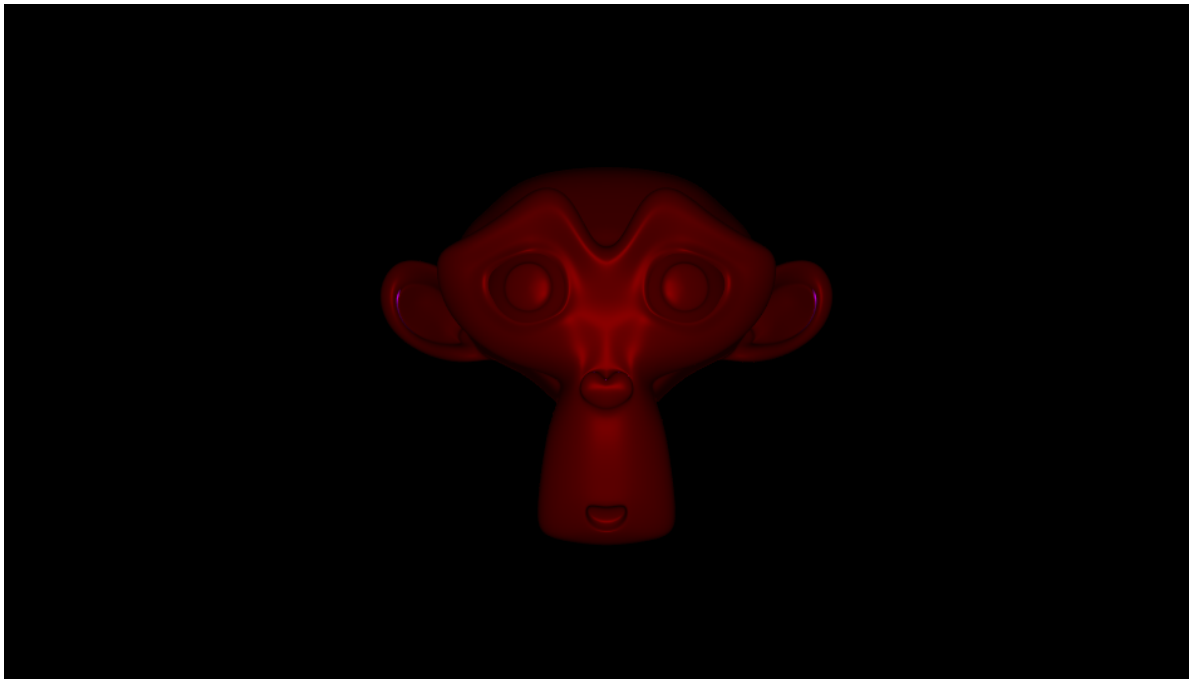


Рисунок 2.3.8а: PBR, GGX

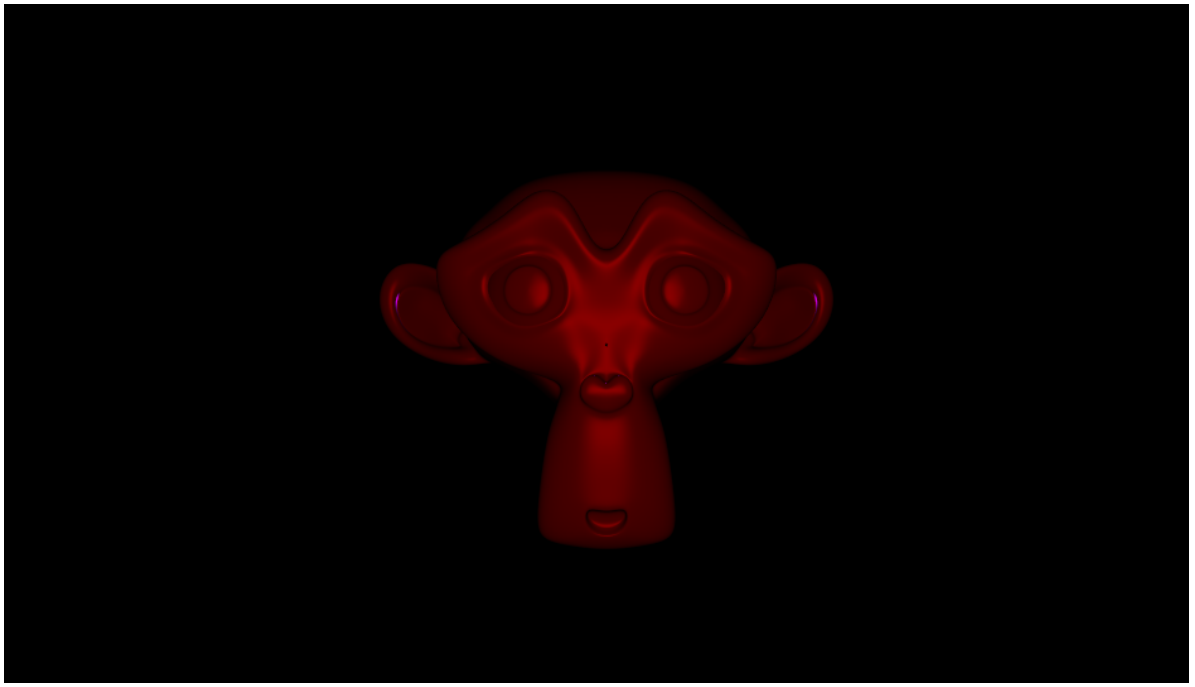


Рисунок 2.3.8б: PBR, Beckmann

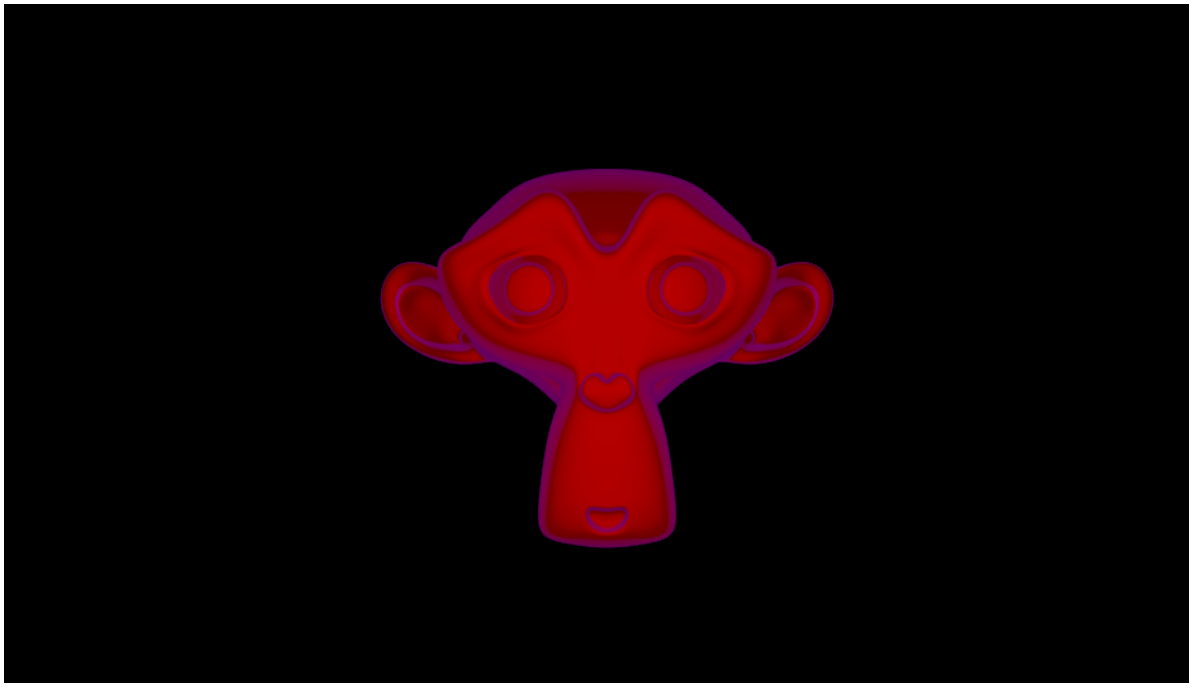


Рисунок 2.3.8в: Phong

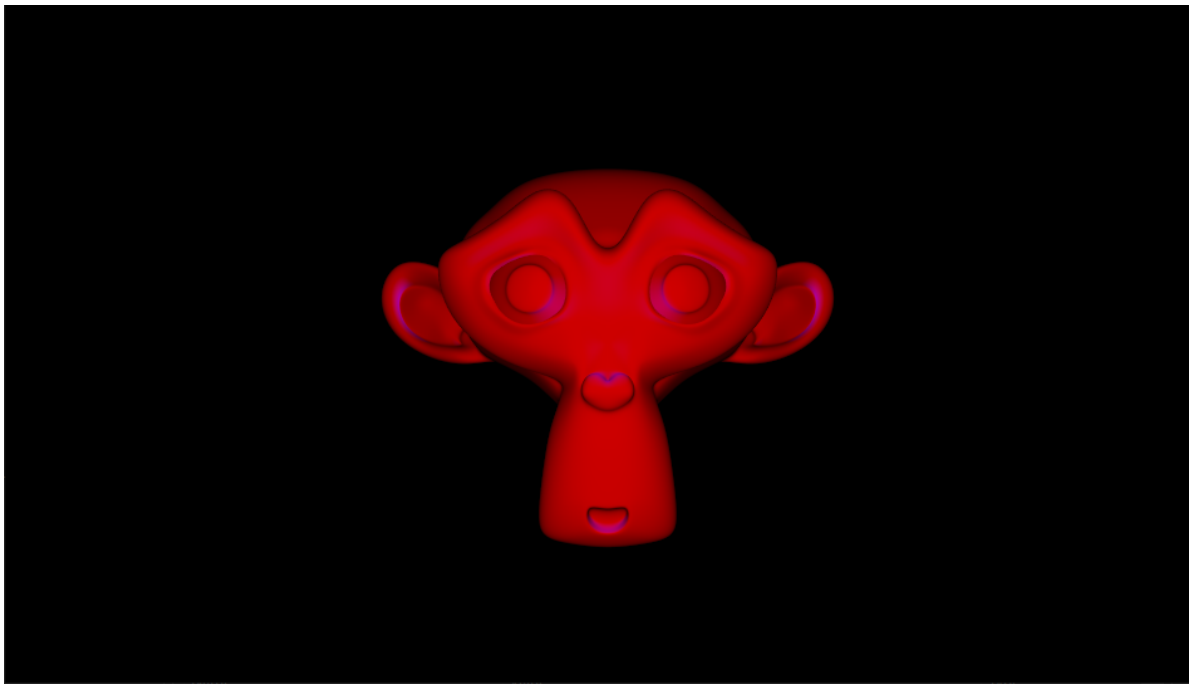


Рисунок 2.3.8г: Blinn-Phong

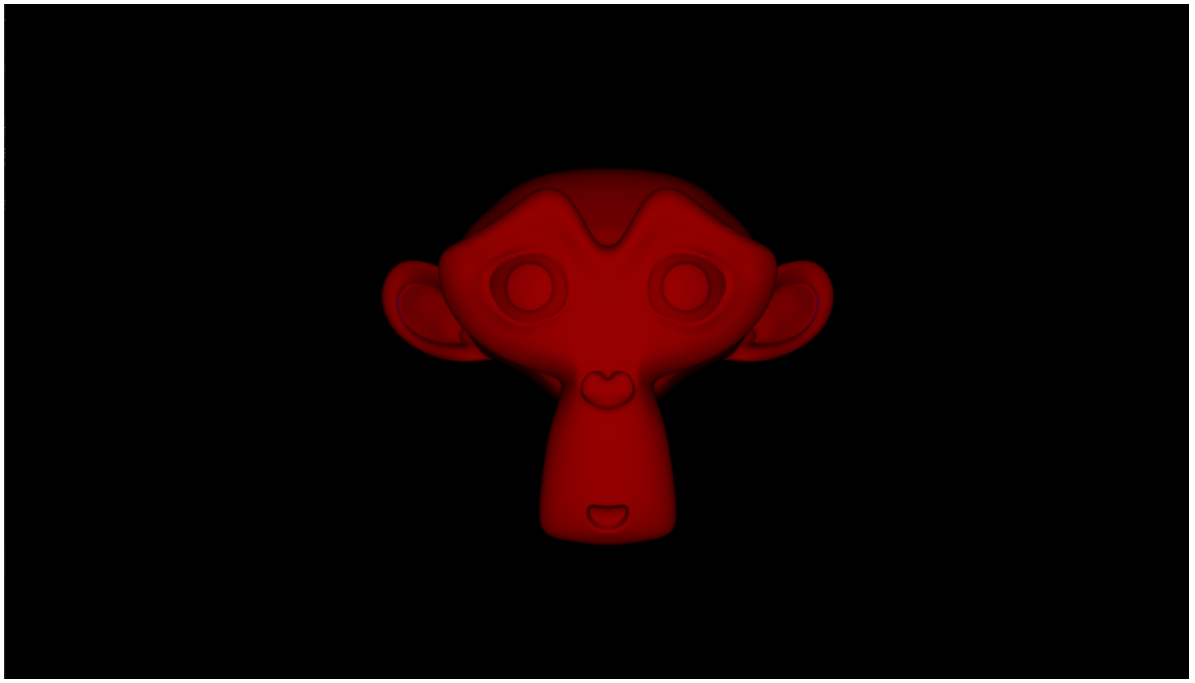


Рисунок 2.3.8д: Minnaert

ЗАКЛЮЧЕНИЕ

Мы провели анализ современных моделей освещения, детально изучили разработки, связанные с PBR, рассмотрев необходимую для этого физическую теорию. Создали рендер, который может быть использован при разработке игровых движков или просто для разработки программ, которым нужно уметь взаимодействовать с 3D объектами. Реализовали в нем несколько моделей освещения, в том числе разные вариации PBR.

ИСТОЧНИКИ

1. Akenine-Moller T., Haines E., Hoffman N. Real-time rendering. – AK Peters/crc Press, 2019.
2. Heitz E. Understanding the masking-shadowing function in microfacet-based BRDFs //Journal of Computer Graphics Techniques. – 2014. – Т. 3. – №. 2. – С. 32-91.
3. Kajiya J. T. The rendering equation //Proceedings of the 13th annual conference on Computer graphics and interactive techniques. – 1986. – С. 143-150.
4. Walter B. et al. Microfacet Models for Refraction through Rough Surfaces //Rendering techniques. – 2007. – Т. 2007. – С. 18th.
5. Smith B. Geometrical shadowing of a random rough surface //IEEE transactions on antennas and propagation. – 1967. – Т. 15. – №. 5. – С. 668-671.
6. Hammon E., Engineer L. S., Entertainment R. Pbr diffuse lighting for ggx+ smith microsurfaces //Game Dev. Conf. – 2017.
7. Blinn J. F. Models of light reflection for computer synthesized pictures //Proceedings of the 4th annual conference on Computer graphics and interactive techniques. – 1977. – С. 192-198.
8. Shirley P. et al. A practitioners' assessment of light reflection models //Proceedings The Fifth Pacific Conference on Computer Graphics and Applications. – IEEE, 1997. – С. 40-49.
9. Schlick C. An inexpensive BRDF model for physically‐based rendering //Computer graphics forum. – Edinburgh, UK : Blackwell Science Ltd, 1994. – Т. 13. – №. 3. – С. 233-246.
10. Kulla C., Conty A. Revisiting physically based shading at imageworks //SIGGRAPH Course, Physically Based Shading. – 2017. – Т. 2. – №. 3.
11. Holzschuch N., Pacanowski R. Identifying diffraction effects in measured reflectances //Eurographics Workshop on Material Appearance Modeling. – Eurographics Association, 2015. – С. 31-34.
12. Holzschuch N., Pacanowski R. A two-scale microfacet reflectance model combining reflection and diffraction //ACM Transactions on Graphics (TOG). – 2017. – Т. 36. – №. 4. – С. 1-12.
13. Belcour L., Barla P. A practical extension to microfacet theory for the modeling of varying iridescence //ACM Transactions on Graphics (TOG). – 2017. – Т. 36. – №. 4. – С. 1-14.

14. Drobot M., Micciulla A. Practical Multilayered Materials in Call of Duty: Infinite Warfare //Physically Based Shading eory Practice-SIGGRAPH Courses. – 2017.
15. Khronos group [Электронный ресурс]: режим доступа: https://www.khronos.org/about/ свободный.
16. Vulkan [Электронный ресурс]: режим доступа: https://www.vulkan.org/ свободный.
17. Графический конвейер [Электронный ресурс]: режим доступа: https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/ свободный.
18. Модели освещения, классификация [Электронный ресурс]: режим доступа: https://en.wikipedia.org/wiki/Computer_graphics_lighting#Polygonal_shading свободный.
19. Курс по разработке графики [Электронный ресурс]: режим доступа: https://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/ свободный.
20. Классические модели освещения [Электронный ресурс]: режим доступа: https://www.jordanstevenstechart.com/lighting-models свободный.
21. OpenGL, обучающий ресурс [Электронный ресурс]: режим доступа: https://learnopengl.com/ свободный.
22. Освещение в Unreal Engine [Электронный ресурс]: режим доступа: https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/ свободный.
23. Краткий обзор PBR [Электронный ресурс]: режим доступа: https://www.jordanstevenstechart.com/physically-based-rendering свободный.
24. PBR book [Электронный ресурс]: режим доступа: https://www.pbr-book.org/ свободный.
25. BRDF browser [Электронный ресурс]: режим доступа: https://graphics.stanford.edu/~smr/brdf/bv/ свободный.
26. Vulkan, обучающий ресурс [Электронный ресурс]: режим доступа: https://vulkan-tutorial.com/ свободный.
27. Vulkan, обучающий ресурс [Электронный ресурс]: режим доступа: https://vkguide.dev/ свободный.

28. Решение вопросов реализации [Электронный ресурс]: режим доступа: https://ru.stackoverflow.com/ свободный.
29. Решение вопросов реализации [Электронный ресурс]: режим доступа: https://www.reddit.com/r/vulkan/ свободный.
30. Спецификация Vulkan [Электронный ресурс]: режим доступа: https://www.khronos.org/registry/vulkan/specs/ свободный.
31. Примеры на Vulkan [Электронный ресурс]: режим доступа: https://github.com/SaschaWillems/Vulkan свободный.
32. Видимый спектр [Электронный ресурс]: режим доступа: http://en.wikipedia.org/wiki/Visible_spectrum свободный.
33. Демонстрация возможностей PBR [Электронный ресурс]: режим доступа: https://xage.ru/eight-hi-res-screens-red-dead-redemption-2-for-pc/ свободный.
34. Закон косинусов Ламберта [Электронный ресурс]: режим доступа: https://en.wikipedia.org/wiki/Lambert%27s_cosine_law свободный.
35. Real-time ray tracing, Unreal Engine [Электронный ресурс]: режим доступа: https://www.youtube.com/watch?v=J3ue35ago3Y свободный.
36. Воспринимаемое световое пространство и sRGB [Электронный ресурс]: режим доступа: https://docs.chaos.com/display/VMAX/ACEScg+Workflow+Setup свободный.
37. Преломление [Электронный ресурс]: режим доступа: https://ru.wikipedia.org/wiki/Преломление свободный.
38. Аппроксимация Beckmann Lambda function [Электронный ресурс]: режим доступа: https://www.desmos.com/calculator/slpji1qiwg свободный.

ПРИЛОЖЕНИЕ А

Проект рендера: https://github.com/hypelive/VulkanApp

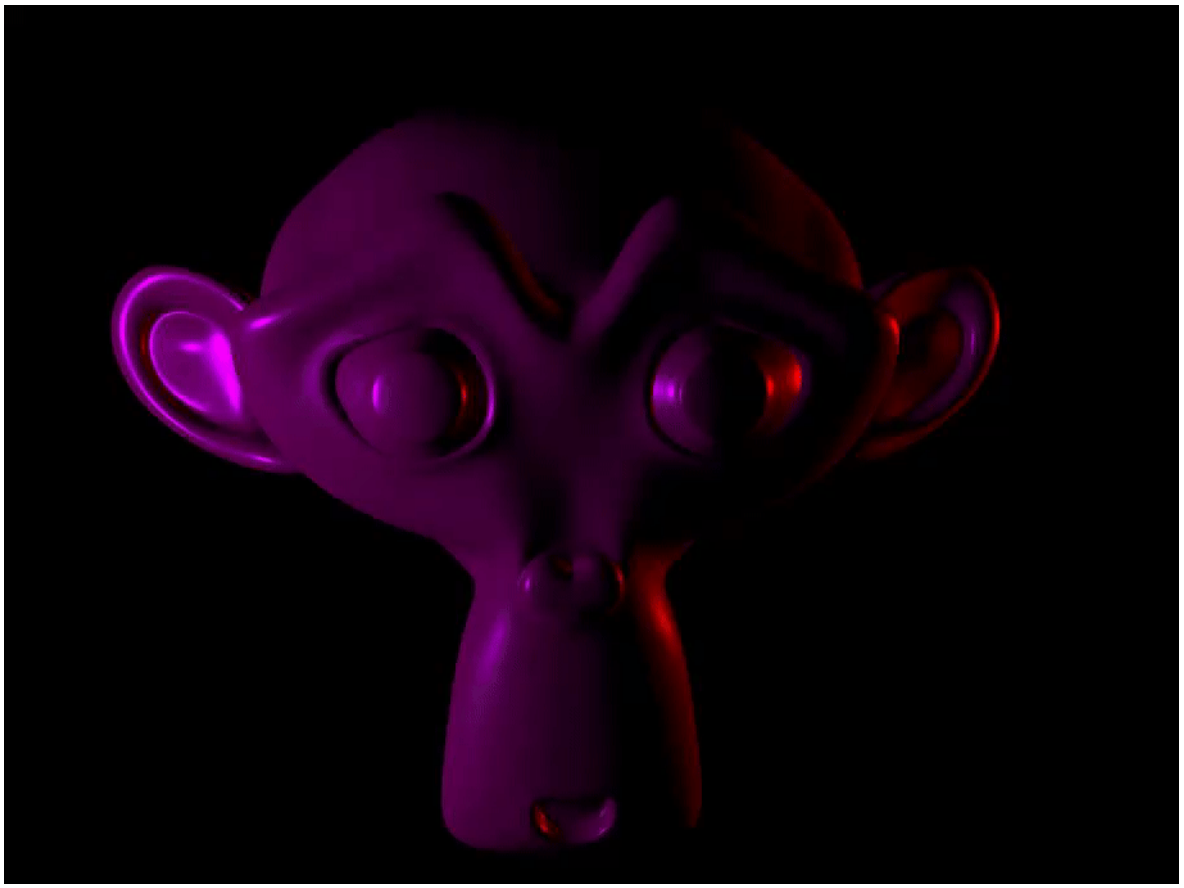


Рисунок А.1: демонстрация проекта рендера

ПРИЛОЖЕНИЕ Б

1 Выбор инструментов

В этой главе будет описаны препосылки выбора определенных инструментов для реализации заданной цели.

Инструментами в нашей работе будут являться конкретные язык программирования и графическое API. Начнем с выбора API.

1.1 История программируемого шейдинга и графических API

Революция на рынке потребительских устройств, произошедшая с выходом 3d ускорителя 3dfx Voodoo Graphics (1996) подтолкнула сообщество на активное развитие направления 3d графики, потому что оно стало более доступным как для разработчиков (стало возможно отрисовывать полноценные 3d объекты, до этого чаще имитировали 3d через 2d, например, в Wolfenstein), так и для пользователей (стоимость 3d ускорителей снизилась). Если смотреть именно на рынок игр, то можно заметить, что в это время произошел бум 3d игр (большинство из которых были шутерами, например, в это время вышли Quake, Half-Life, Unreal). Однако разработчикам графики хотелось больше гибкости в инструментах, потому что разработка под ранние спецификации графических API (DirectX, OpenGL) “больше напоминала процесс настройки в CSS, чем программирование”. Собственно, работа заключалась в том, чтобы выставить правильную конфигурацию для отрисовки нужных элементов, такой подход называют “fixed function style”, к примеру, Unity до сих пор частично поддерживает эту функциональность в своём ShaderLab[8]. В этом

[8] ShaderLab - язык для разработки шейдеров в Unity.

контексте стоит различать понятия Конфигурируемый (Configurable) и Программируемый (Programmable). В случае же с программируемым API мы уже пишем полноценный код, а разработка похожа на программирование на каком-нибудь C. Сразу отметим, что в первом подходе мы можем оптимизировать процессы за счет аппаратных реализаций конкретных алгоритмов, но теряем в универсальности. Для второго подхода возможностей для аппаратных оптимизаций намного меньше, но это компенсируется возможностью реализации программистом широкого спектра алгоритмов отрисовки и возможностью их существенной программной оптимизации.

Важным моментом в истории стал выход видеоускорителя GeForce 3 и вместе с ним спецификации API DirectX 8.0 в 2001-м году. Это была первая видеокарта, поддерживающая программируемые стадии графического конвейера (render pipeline) - добавлены вершинный (vertex) и фрагментный (fragment), с ограничениями, шейдеры. Тогда же DirectX определила понятие Шейдерная модель (Shader Model), чтобы различать видеокарты с разными разными шейдерными возможностями (доступными функциями).

В 2002 году Microsoft выпустила DirectX 9.0 с улучшенной и расширенной поддержкой шейдеров (Shader Model 2.0 + HLSL). OpenGL от Khronos Group развивался вместе с DirectX, и чаще всего после релиза новой версии DirectX тот же функционал появлялся и в новой версии OpenGL.

Также с некоторого момента развивалась спецификация OpenGL ES как сокращенная версия OpenGL, предназначенная для “встроенных систем” (по большей части для мобильных устройств).

DirectX 10.0 выпущен в 2006-м году, в нем была добавлена еще одна программируемая стадия - стадия геометрии (geometry stage). С выходом

DirectX 12.0 определилили еще один вид шейдеров tessellation stage shader и добавили возможность исполнять код на видеокарте вне графического конвейера с помощью compute shader.

Можно наблюдать эволюцию ускорителей и API от идей полностью конфигурируемого конвейера (fully configurable\fixed function render pipeline) к идеям полностью программируемого конвейера (fully programmable render pipeline).

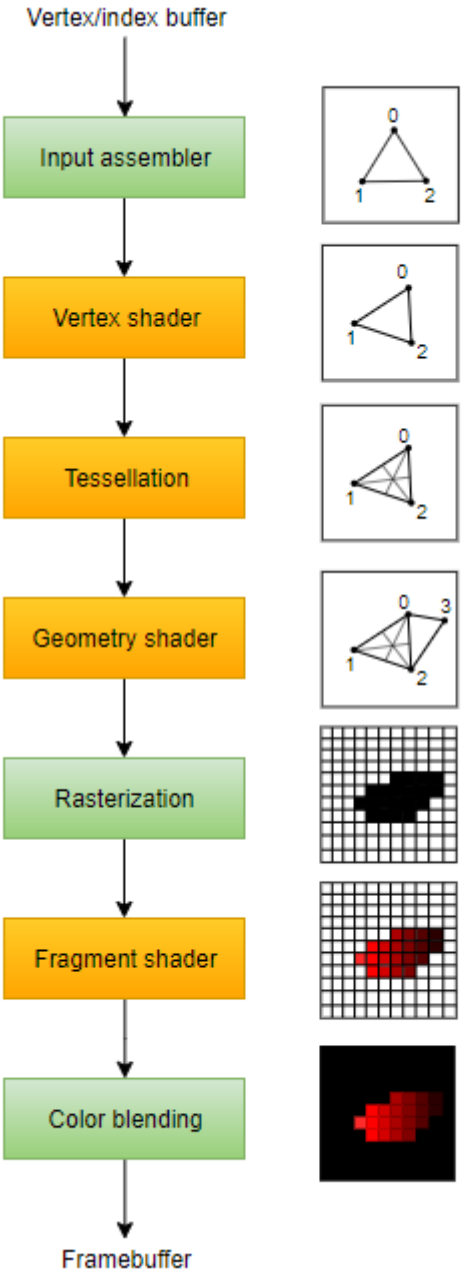


Рисунок Б.1: схема графического конвейера [26]

На рис. Б.1 представлена общая схема графического конвейера в современных версиях графических API (в частности в Vulkan). Зеленым показаны конфигурируемые стадии, желтым - программируемые. Более подробную информацию об этих стадиях можно получить по ссылке Graphics pipeline - Wikipedia.

Приведем некоторые актуальные API и укажем некоторые их особенности в таб. Б.1:

Прим.

В столбце про платформозависимость Vendor specific означает, что API на конкретных платформах от узкого списка производителей, Cross platform же означает, что API можно использовать на любых платформах, под которые есть соответствующий драйвер.

В столбце про уровень абстракций аппаратного обеспечения условно указано, с насколько близкими к железу абстракции позволяет работать API. Low level - абстракции близкие к аппаратным средствам, high level - высокие абстракции.

Таблица Б.1: современные графические API

Название	Год первого выпуска спецификации	Платформозависимость	Уровень абстракций аппаратного обеспечения	Используемый шейдерный язык
DirectX 12	2015	Vendor specific	low level	HLSL
DirectX 11.3	2015	Vendor specific	high level	HLSL
Vulkan	2016	Cross platform	low level	SPIR-V

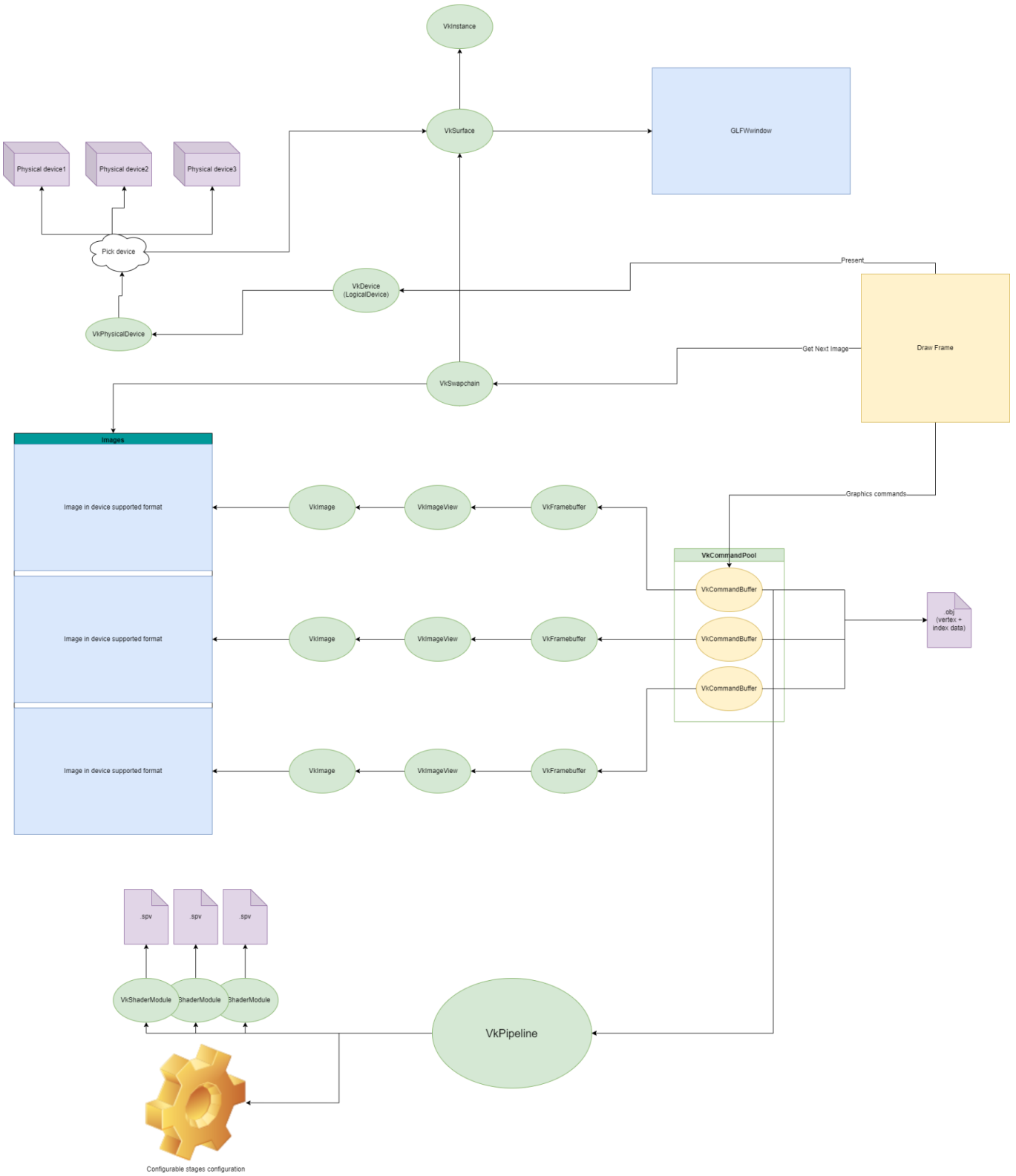
Metal	2014	Vendor specific	low level	MSL
OpenGL 4.6	2017	Cross platform	high level	GLSL
OpenGL ES 3.2	2015	Cross platform	high level	GLSL
WebGL 2	2017	Cross platform	high level	GLSL

Для отрисовки в реальном времени (real-time rendering) в играх на PC сейчас чаще всего используют либо DirectX 11.3\12.0, либо Vulkan, либо вместе. Т.к. они являются наиболее гибкими в плане реализуемых алгоритмов, и имеют больше возможностей для оптимизации графики, за счет приближения к идее полностью программируемого графического конвейера, а значит позволяют выдавать более красивую картинку при тех же аппаратных ресурсах за счет оптимизаций со стороны программистов графики.

Рисунок Б.2: демонстрация возможностей Vulkan на примере компьютерной игры Red Dead Redemption 2 [33]

ПРИЛОЖЕНИЕ В

Архитектура VulkanApplication



Инициализацией всего необходимого, управлением ресурсами и объектами сцены (камера, источники света, центральный объект)

занимается объект класса *VulkanApplication*. Для запуска приложения используется метод *VulkanApplication.run()*. [ссылка]

```
void run()
{
    initWindow();
    initVulkan();
    mainLoop();
    cleanup();
}
```

В этом методе производится инициализация окна GLFW и ресурсов Vulkan, в том числе создание графического конвейера, свапчейна и логического девайса, далее запускается основной цикл, внутри которого вызывается функция *drawFrame()*. После выхода из главного цикла происходит очистка выделенных ресурсов. [ссылка]

```
void mainLoop()
{
    while (!glfwWindowShouldClose(window))
    {
        glfwPollEvents();
        drawFrame();
    }
    ...
}
```

drawFrame() [ссылка] в свою очередь делает 3 действия, причем делает все это асинхронно:

- Получить следующее изображение из свапчейна
- Нарисовать на полученном фреймбуффере картинку
- Показать полученное изображение

Вся синхронизация при этом выполняется с помошью ресурсов приложения, а именно семафоров и фенсов.