Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Высшая школа программной инженерии

Работа допущена к защите

Директором ВШ ПИ

_____ П.Д. Дробинцевым

«___» _____20__ г.

# ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

# РАЗРАБОТКА БИБЛИОТЕКИ ДЛЯ ВИЗУАЛИЗАЦИИ ТРЁХМЕРНЫХ СЦЕН В РЕАЛЬНОМ ВРЕМЕНИ С ИСПОЛЬЗОВАНИЕМ АЛГОРИТМА ТРАССИРОВКИ ПУТИ

по направлению 09.03.04 Программная инженерия
по образовательной программе
09.03.04_01 Технология разработки и сопровождения качественного программного продукта

Выполнил
студент гр. 43504/2 А.В. Веселов

Руководитель
доцент, к.т.н. И.А. Андреев

Санкт-Петербург

2018

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

УТВЕРЖДАЮ

Директор ВШ ПИ

_____ П.Д. Дробинцев

«___» _____20__ г.

**З А Д А Н И Е**
**по выполнению выпускной квалификационной работы**

Студенту *Веселову Александру Владимировичу, гр. 43504/2*
(фамилия, имя, отчество, номер группы)

1. Тема работы: *Разработка библиотеки для визуализации трёхмерных сцен в реальном времени с использованием алгоритма трассировки пути*

2. Срок сдачи студентом законченной работы:

3. Исходные данные по работе:
*Описание алгоритма трассировки пути, спецификация библиотеки OpenCL, спецификация форматов Wavefront OBJ, Radiance HDR, S3TC, описание языка MAXScript*

4. Содержание работы (перечень подлежащих разработке вопросов):
*Введение и постановка задачи;*
*Обзор методов визуализации;*
*Теоретические основы алгоритма трассировки пути;*
*Обзор способов улучшения алгоритма;*
*Программная реализация;*
*Обзор полученных результатов и сравнение с другими продуктами.*

5. Перечень графического материала (с указанием обязательных чертежей):

6. Консультанты по работе:

7. Дата выдачи задания *«5» февраля 2018 г.*

Руководитель ВКР _____ *И. А. Андреев*
(подпись) (инициалы, фамилия)

Задание принял к исполнению _____
*(дата)*

Студент _____ *А. В. Веселов*
(подпись) (инициалы, фамилия)

# РЕФЕРАТ

На 88 с., 43 рисунка, 1 таблицу.

ТРАССИРОВКА ПУТИ, ТРАССИРОВКА ЛУЧЕЙ, ФИЗИЧЕСКИ КОРРЕКТНЫЙ РЕНДЕРИНГ, BRDF, МЕТОД МОНТЕ-КАРЛО, ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ, GPGPU

В данной работе рассматривается создание программного продукта для визуализации фотореалистичных изображений в реальном времени. Его работа основана на алгоритме трассировки пути, который обеспечивает максимальную реалистичность получаемого изображения. Система физически корректных материалов дает правильное освещение и отражения. Вычисления распараллеливаются на графическом процессоре. Это позволяет визуализировать масштабные, высоко детализированные трехмерные сцены со скоростью до 30 кадров в секунду, с чем не сравнятся "классические" алгоритмы трассировки лучей, на выполнение которых может уходить несколько часов.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Рендеринг (англ. rendering – «визуализация») – термин в компьютерной графике, обозначающий автоматический процесс генерирования фотореалистичных, либо нефотореалистичных изображений по трехмерной модели (сцене) с помощью компьютерной программы.

Сцена состоит из объектов, описанных с помощью некоторого языка или структуры данных, и может содержать информацию о геометрии, положении точки наблюдателя, текстурах, освещении, свойствах материалов, анимации и т.д. Программа, осуществляющая визуализацию, преобразует данную информацию в двухмерное растровое изображение или последовательность изображений.

**Актуальность исследования**

Различают пре-рендеринг и рендеринг в реальном времени.

- Пре-рендеринг (англ. pre-rendering) – сложный вычислительный процесс, при котором визуализация изображения не является интерактивной. В этом случае рендеринг осуществляется на достаточно мощном оборудовании, после чего результат сохраняется в качестве последовательности кадров для дальнейшего воспроизведения на целевой платформе, от которой не требуется большого количества ресурсов. В зависимости от сложности сцены и точности моделирования, время визуализации одного кадра может занимать от нескольких минут до нескольких суток.

Пре-рендеринг используется в фильмах, телевидении, рекламе, визуализации архитектуры и дизайна, визуализации неизменяющихся или медленно изменяющихся данных. Его плюсами является реалистичность и высокое качество эффектов, минусами – требовательность к вычислительным ресурсам, длительное время

получения результата, а также невозможность в общем случае изменить уже готовый результат вычислений.

• Рендеринг в реальном времени (англ. real-time rendering) – процесс визуализации, при котором пользователь может интерактивно взаимодействовать с виртуальным окружением, при этом результат вычислений моментально отображается на том же самом оборудовании с приемлемой для человеческого глаза частотой. Программы для визуализации изображений в реальном времени используют приближенные алгоритмы, сложность вычислений которых гораздо меньше, чем при пре-рендеринге.

Наиболее часто рендеринг в реальном времени применяется в компьютерных играх и симуляторах, визуализации быстро изменяющихся данных. Его преимуществами являются интерактивность и отсутствие высоких требований к ресурсам, недостатком – более низкое качество визуализации по сравнению с алгоритмами пре-рендеринга.

**Объект и предмет исследования**

Объектом исследования в данной работе является компьютерная графика, а предметом – алгоритм трассировки пути и его реализация на языке OpenCL.

**Цель исследования**

Необходимо разработать библиотеку для визуализации трехмерных сцен в реальном времени, основанную на алгоритме трассировки пути с использованием фреймворка OpenCL, и добиться максимального быстродействия алгоритма.

**Задачи работы**

1. Исследовать основы и методы визуализации фотореалистичных изображений в компьютерной графике;

2. Изучить и применить язык OpenCL в реализации программного продукта;
3. Реализовать библиотеку для рендеринга трёхмерных сцен в реальном времени;
4. Оценить быстродействие и сравнить реализованный программный продукт с известными аналогами.

**Теоретическая и методологическая база исследования**

Данная работа охватывает такие направления наук, как математический анализ, линейная алгебра и аналитическая геометрия, геометрическая и физическая оптика, теория вероятностей.

**Информационная база**

Информационной базой для данной работы служат знания, полученные во время обучения в университете, а также различные статьи, найденные по данной теме в Интернете. Также большое влияние на разработку данного проекта оказала книга Мэтта Фарра, Грега Хамфриса и Венцеля Якоба «Physically Based Rendering: From Theory to Implementation, 3rd Edition».

**Степень научной разработанности проблемы и научная новизна**

Сама по себе основа теории физически корректного рендеринга разработана давно. Однако лишь в последнее время, с ростом вычислительных мощностей, стала возможной реализация алгоритмов, позволяющих быстро получать фотореалистичные изображения. В связи с этим появилось множество исследований и практических приёмов для ускорения и улучшения качества рендеринга. В итоге, мы имеем быстро развивающуюся и перспективную область науки.

**Практическая значимость**

На данный момент на рынке существует множество инструментов визуализации. Но многие из них выполняют свою работу в режиме «офлайн», то есть, не позволяют получать результат в интерактивном режиме, либо имеют экспериментальную поддержку рендеринга в реальном времени. В данном исследовании применяются общие подходы к методам визуализации, но делается акцент на интерактивность. Помимо этого, в отличие от большинства других, данный проект имеет открытый исходный код.

# ГЛАВА 1. ОБЗОР МЕТОДОВ ВИЗУАЛИЗАЦИИ

В настоящее время разработано множество технологий рендеринга, которые имеют общую *минимальную* задачу: спроецировать модель сцены, описанную (чаще всего) в векторном виде в памяти компьютера на экран наблюдателя, представляющий собой матрицу пикселей.

Обычно простого построения силуэтов объектов сцены недостаточно: нужно создать иллюзию материалов, из которых они состоят, рассчитать освещение и затенение, отражения и преломления, туман, облака, цвет неба и т.д.

С точки зрения физики, все эти эффекты основаны на распространении света. Свет может рассматриваться либо как электромагнитная волна, скорость распространения в вакууме которой постоянна, либо как поток фотонов – частиц, обладающих определённой энергией, импульсом, собственным моментом импульса и нулевой массой.

Однако, точное моделирование распространения света, как в реальной жизни, в настоящее время является невыполнимой задачей, поскольку отслеживание огромного количества частиц света требует практически безграничное количество аппаратных ресурсов и времени.

На деле рендеринг – это попытка решения некоего уравнения, описывающего распространение света в трехмерной сцене, причем уравнение учитывает только корпускулярные свойства света.

Уравнение рендеринга было опубликовано Дэвидом Иммелем и Джеймсом Каждия в 1986 году на конференции SIGGRAPH. Мы его рассмотрим в следующих главах.

Все методы визуализации так или иначе решают уравнение рендеринга, то есть более или менее приближенно вычисляют значение интеграла в нём.

По этому признаку их можно разделить на два типа:

1. Рендеринг без допущений (англ. unbiased rendering).

Алгоритмы этого класса обычно используют метод Монте-Карло для нахождения значения интеграла освещенности, не привнося систематических ошибок или искажений в его оценке. Рендеринг без допущений старается максимально правдиво описать поведение света, поэтому он часто используется для генерации эталонного изображения, с которым сравниваются другие методы визуализации. Ошибки рендеринга возникают за счёт дисперсии и проявляются в виде высокочастотного шума. При увеличении количества итераций метода Монте-Карло в $n$ раз, шум уменьшается в $\sqrt{n}$ раз, что ограничивает применимость рендеринга без допущений в интерактивных приложениях.

2. Рендеринг с допущениями (англ. biased rendering).

Biased-алгоритмы вносят аппроксимацию в решение интеграла освещенности. При рендеринге с допущениями, оценка интеграла может содержать систематическую ошибку, которая может проявляться либо в виде размытости освещения, либо в его низком разрешении, либо в исключении из рассмотрения некоторых путей прохождения освещения и т.д. Но это не означает, что результат для всех алгоритмов этого типа будет неверным: при увеличении количества итераций решение интеграла может сходиться к правильному, если оценка интеграла является состоятельной.

(а) (б)

Рис. 1.1. Расчёт отражений в рендеринге без допущений (а) и с допущениями (б)

С точки зрения проецирования сцены на экран, можно выделить два типа методов рендеринга:

1. Растеризация (англ. rasterization) – процесс преобразования векторного описания объектов в набор пикселей. Сцена состоит из множества треугольников, положение и форма которых в виртуальном пространстве задается трехмерными координатами вершин. При рендеринге учитывается тот факт, что треугольник – плоская фигура, поэтому достаточно спроецировать вершины на экран наблюдателя, а все промежуточные точки поверхности интерполировать с учётом глубины.

2. Рейкастинг (англ. ray casting) – метод рендеринга, при котором из камеры наблюдателя для каждого пикселя изображения испускается луч, и находится самый близкий объект, блокирующий путь распространения этого луча. При использовании рейкастинга имеется возможность рендеринга гладких объектов без аппроксимации их полигональными поверхностями (например, треугольниками).

После данных преобразований, для обоих методов, цвет каждого пикселя рассчитывается на основании радиус-вектора спроецированной точки, её вектора нормали, положения и направления

наблюдателя, а также положения источников света и прочих параметров.

Растеризация имеет полную аппаратную поддержку, поэтому производительность её значительно выше, чем у рейкастинга, который в основном реализуется программно.

Для более реалистичного моделирования освещения используются алгоритмы глобального освещения (англ. global illumination), учитывающие не только прямой свет, но и отражённый свет от различных поверхностей (англ. indirect illumination), расширяющие описанные выше методы.

С точки зрения моделирования глобального освещения, методы визуализации можно разделить на рассмотренные выше алгоритмы рендеринга без допущений и с допущениями.

Алгоритмы рендеринга без допущений продолжают идею рейкастинга, испуская отраженные от поверхности пересечения вторичные лучи, повторяя данный процесс много раз. Этот процесс называется трассировка лучей (англ. ray tracing) и имеет некоторые разновидности и оптимизации:

1. Path tracing;
2. Bidirectional Path Tracing;
3. Metropolis Light Transport и т.д.

Алгоритмы рендеринга с допущениями:

1. Метод излучательности (англ. radiosity) – рассматривает частный случай решения уравнения освещенности, когда материалы являются диффузными. Материал называется диффузным, если вся энергия, отраженная от его поверхности, рассеивается равномерно по всем направлениям. BRDF диффузных материалов равняется константе, поэтому интеграл в уравнении освещенности

принимает более простой вид. Все поверхности сцены разбиваются на небольшие фрагменты – патчи (англ. patches), каждый из которых наделён свойствами излучать, поглощать и отражать свет, после чего для каждого патча на каждой итерации подсчитывается полученная им от других патчей энергия, а также доля энергии, которая будет излучена патчем на следующей итерации;

2. Метод фотонных карт (англ. photon mapping) – состоит из трех частей: трассировка фотонов, построение фотонной карты и сбор освещённости;

3. Spherical Harmonic Mapping – алгоритм глобального освещения, заменяющий интеграл освещенности функциями, использующими сферические гармоники с предварительно рассчитанными коэффициентами;

4. Voxel Cone Tracing – осуществляется разбиение пространства на воксельную сетку, после чего для каждого пикселя экрана трассируется несколько конусов, собирающих освещенность с разных направлений;

5. Distance Field Global Illumination;

6. Light Propagation Volumes и т.д.

## 1.1. Трассировка пути

Трассировка пути (англ. path tracing) – алгоритм рендеринга без допущений, дающий наиболее реалистичный и физически корректный результат.

Для моделирования освещения мы можем из каждого источника света испускать очень много лучей (в идеале столько же, сколько в реальности фотонов) и отслеживать их поглощение и отражение от различных поверхностей. Данный метод называется

методом прямой трассировки, и его «грубое» применение является нецелесообразным, так как большинство лучей, испущенных источником, не попадает в приемник, а значит, и не влияет на формируемое в нем изображение. Лишь очень малая часть лучей после всех отражений и преломлений в конце концов попадает в камеру наблюдателя.

В оптике время изотропно, это означает, что оба направления времени равноправны. Таким образом, можно следить за лучами не от источника к камере наблюдателя, а в противоположном направлении, от камеры к источнику. Это избавляет от необходимости тратить ресурсы на трассировку лучей, не попавших в камеру.

Трассировка пути является обобщением традиционной трассировки лучей, алгоритм которой трассирует лучи в направлении от виртуальной камеры сквозь пространство; луч «отскакивает» от предметов до тех пор, пока полностью не поглотится или рассеется.

Трассировка пути является наиболее простым, наиболее точным с физической стороны и наиболее медленным по производительности методом рендеринга. Этот метод естественным способом воспроизводит множество оптических эффектов, которые тяжело достижимы или вообще недостижимы другими методиками рендеринга: глобальное освещение, глубина резко изображаемого пространства (англ. depth of field), размытие в движении (англ. motion blur) и каустика.

Огромное преимущество данного алгоритма по сравнению со другими заключается в том, что можно визуализировать промежуточный результат. Картинка рассчитывается постепенно, и, если она устраивает пользователя, алгоритм можно остановить.

# ГЛАВА 2. ОПИСАНИЕ ТРЕХМЕРНОЙ СЦЕНЫ

## 2.1. Формат Wavefront OBJ

Мы будем использовать стандартное описание сцены в виде полигональной сетки из треугольников. Для этого был выбран один из самых распространенных форматов OBJ.

Формат файлов OBJ – это простой формат данных, который содержит только 3D геометрию, а именно, позицию каждой вершины, связь координат текстуры с вершиной, нормаль для каждой вершины, а также индексы вершин, которые образуют полигоны.

Формат файла является открытым и может быть экспортирован и импортирован в большинстве приложений для моделирования трехмерной графики, таких как Maya, XSI, Blender, 3Ds Max, Milkshape 3d, Modo, Cinema 4D и т.д.

Файл начинается объявления библиотеки материалов, которую будет использовать модель:

```
# это комментарий
mtllib box.mtl
```

Далее идет список вершин с координатами в пространстве, список нормалей вершин и список текстурных координат:

```
# вершины (x, y, z)
v  0.0000 0.0000 30.0000
v  0.0000 40.0000 30.0000
v  60.0000 40.0000 30.0000
...
# нормали (x, y, z)
vn 0.0000 0.0000 -1.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 1.0000 0.0000
...
# текстурные координаты (u, v, w)
vt 0.0000 0.0000 0.0000
vt 0.0000 1.0000 0.0000
vt 1.0000 1.0000 0.0000
```

Затем для каждого объекта сцены указывается материал, после чего идёт список строк (начинающихся с f), описывающий грани (полигоны) объекта в виде индексов элементов из предыдущих списков (вершина/текстурная координата/нормаль):

```
g Plane01
usemtl Wall
f 1/1/1 2/2/1 3/3/1
f 3/3/1 4/4/1 1/1/1
...
```

Формат MTL предназначен для описания материалов, использующихся в сцене, применяется совместно с форматом OBJ. Его описание приведено ниже:

```
newmtl название_материала1      # Объявление очередного материала
# Цвета (r g b)
Ka 0.0000 1.0000 0.1647         # Цвет окружающего освещения
Kd 0.0000 1.0000 0.1647         # Цвет поглощения
Ke 0.0000 0.0000 0.0000         # Цвет излучения
# Параметры отражения
Ks 0.0900 0.0900 0.0900         # Цвет зеркального отражения
Ns 5.0000                       # Коэффициент зеркального отражения
Ni 1.5000                       # Шероховатость материала
# Параметры прозрачности
d                               # Прозрачность
Tr                              # 1 – d
Tf 1.0000 1.0000 1.0000         # Цвет прозрачности
# Модель освещения
illum 2
#Следующий материал
newmtl название_материала2
...
```

## 2.2. Поиск пересечений

Проверка пересечения луча с треугольником – наиболее сложная и медленная по времени выполнения часть процесса трассировки.

Алгоритм поиска пересечения луча с треугольником [7]:

**Шаг 1**. Нахождение точки пересечения луча и плоскости треугольника.

Мы знаем, что точка пересечения $P$ находится где-то на луче, определяемом его началом $O$ и его направлением $R$. Параметрическое уравнение луча выглядит следующим образом:

$$P = O + tR$$

где $t$ – расстояние от $O$ до $P$. Для нахождения $P$ необходимо найти $t$.

Уравнение плоскости треугольника:

$$Ax + By + Cz + D = 0$$

где $A$, $B$, $C$ – компоненты нормали треугольника, $D$ – расстояние от центра координат до плоскости.

Нормаль треугольника вычисляется как нормированное векторное произведение двух сторон треугольника:

$$N = \frac{(V_2 - V_1) \times (V_3 - V_1)}{\|(V_2 - V_1) \times (V_3 - V_1)\|}$$

Если $|N \cdot R| < \varepsilon$, то считается, что луч параллелен плоскости треугольника и пересечения нет.

$D$ вычисляется как скалярное произведение радиус-вектора любой вершины треугольника (выберем первую), и нормали:

$$D = (V_1 \cdot N)$$

Объединив уравнения, получаем выражение для $t$:

$$N_x P_x + N_y P_y + N_z P_z + D = 0$$

$$N_x(O_x + tR_x) + N_y\big(O_y + tR_y\big) + N_z(O_z + tR_z) + (V_1 \cdot N) = 0$$

$$(N \cdot O) + t(N \cdot R) + (V_1 \cdot N) = 0$$

$$t = -\frac{(N \cdot O) + (V_1 \cdot N)}{(N \cdot R)}$$

Если $t \leq 0$, то точка пересечения находится «позади» направления луча и пересечения нет.

**Шаг 2**. Проверка, находится ли точка $P$ внутри треугольника.

Точка $P$ находится внутри треугольника, если её барицентрические координаты неотрицательны (рис. 2.2).

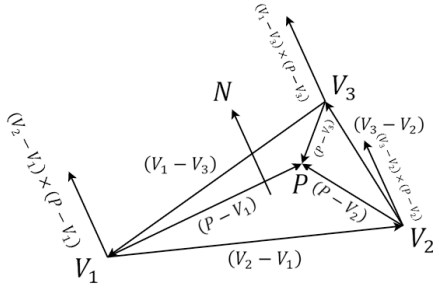



Рис.2.2. Иллюстрация проверки

$$\begin{cases} Bary_{V_1} = \big((V_2 - V_1) \times (P - V_1) \cdot N\big) > 0 \\ Bary_{V_2} = \big((V_3 - V_2) \times (P - V_2) \cdot N\big) > 0 \\ Bary_{V_3} = \big((V_1 - V_3) \times (P - V_3) \cdot N\big) > 0 \end{cases} \Rightarrow P \text{ внутри треугольника.}$$

**Шаг 3**. Интерполяция нормалей.

Интерполируем нормали вершин треугольника $N_1$, $N_2$, $N_3$ для получения сглаженной нормали в точке $P$ на основании барицентрических координат.

$$N_{shading} = \frac{N_1 Bary_{V_1} + N_2 Bary_{V_2} + N_3 Bary_{V_3}}{\left\| N_1 Bary_{V_1} + N_2 Bary_{V_2} + N_3 Bary_{V_3} \right\|}$$

# ГЛАВА 3. УСКОРЕНИЕ ТРАССИРОВКИ

Ускоряющие структуры являются одним из главных компонентов любого рейтрейсера. Наивная реализация процесса поиска пересечения луча и треугольника представляет собой последовательный перебор абсолютно всех полигонов сцены. Для детализированных сцен это составляет большую проблему, так как время поиска растет линейно с увеличением количества полигонов. Тем самым, мы теряем время в случае, когда луч проходит, например,

вдалеке от большинства примитивов, при этом пересечения с ними всё равно проверяются. Задача акселерационных структур – быстро отбросить целые группы примитивов, позволяя рассматривать только те полигоны, которые находятся вблизи траектории прохождения луча.

Поскольку проверка на пересечение луча с объектами занимает основное время выполнения при трассировке лучей, было разработано большое количество алгоритмов для ускорения данного процесса. Грубо говоря, всех их можно разделить на два основных подхода: разбиение пространства и разбиение объектов. Алгоритмы разбиения пространства осуществляют декомпозицию трёхмерного пространства на области (например, на блоки, ориентированные вдоль осей координат) и рассматривают, в какой области лежит каждый примитив. В момент поиска пересечения луча с примитивами, составляется последовательность областей, через которые проходит луч, после чего проверяются на пересечение только те примитивы, которые находятся в этих областях.

С другой стороны, разбиение объектов основано на последовательном дроблении всей сцены на более мелкие множества составляющих её объектов. Например, модель комнаты может быть разбита на четыре стены, пол, потолок и стул, который в ней находится. Если луч не пересечёт объем, ограничивающий комнату, все эти объекты могут быть отброшены. Иначе для луча будет осуществлена проверка на пересечение с каждым из них. Если же луч пересечётся с объёмом, ограничивающий стул, будет можно будет произвести проверку на пересечение с его ножками, сиденьем и спинкой. В случае отсутствия пересечения с этими объектами, стул может быть отброшен.

Оба этих подхода достаточно успешно применяются при разрешении основной проблемы поиска пересечения примитивов с

лучом, и нет фундаментальных причин предпочитать один алгоритм другому.

## 3.1. Равномерная сетка

Одним из самых простых вариантов разбиения пространства является его разбиение на регулярную воксельную сетку [10]. Идея данного метода состоит в том, что мы можем рассматривать при поиске пересечений только те примитивы, которые содержатся в ячейках сетки, через которые проходит луч. Затраты на обход равномерной сетки минимальны, и данный метод хорош в случае, если примитивы в сцене расположены равномерно. Если же примитивы расположены неравномерно, и в некоторых ячейках сетки находится большое количество примитивов, то такая ускоряющая структура будет неэффективна, так как будет уходить много времени на последовательный перебор примитивов в одной ячейке.

### 3.1.1. Построение сетки

Данная структура данных представляет собой два массива: массив индексов примитивов в сцене и массив ячеек, указывающий на элементы массива в каждой ячейке параллелепипеда.

Создание равномерной сетки происходит при загрузке модели сцены и представляет собой рекурсивный процесс (рис. 3.1):

1. Сначала углубляемся в рекурсию, последовательно уменьшая вдвое количество разбиений ограничивающего объема сцены до того момента, пока оно не будет равным 2;
2. Затем для каждой ячейки сетки инициализируем ограничивающий объём;

3. Если мы на нижнем уровне рекурсии, обходим все примитивы в сцене и проверяем их на пересечение ограничивающим объёмом, созданным на шаге 2, пересекающиеся добавляем в массив индексов;
4. Если мы находимся не на нижнем уровне рекурсии, то обходим и проверяем пересечение только с теми примитивами, которые пересекаются с ограничивающим объёмом, рассмотренном на более глубоком уровне рекурсии;
5. Добавляем индекс текущей ячейки и количество элементов в ней в массив ячеек.

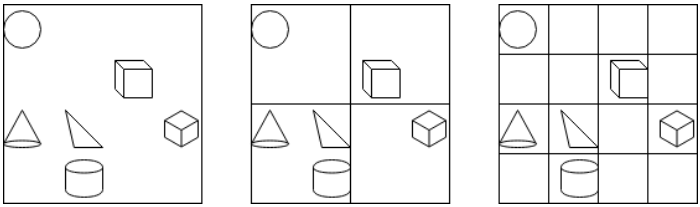


Рис. 2.1. Рекурсивное построение регулярной сетки

### 3.1.2. Обход сетки

При поиске пересечения луча и треугольника, пробегаем только по тем ячейкам разбиения, через которые прошел луч. Это делается с помощью алгоритма 3DDA. Он состоит из двух функций:

- Инициализация параметров $t_{max}$, $step$, $t_{delta}$:

```
void DDA_prepare(Ray ray, float3* tmax, float3* step, float3* tdelta)
{
  float3 cell_min = floor(ray.origin / CELL_SIZE) * CELL_SIZE;
  float3 cell_max = cell_min + CELL_SIZE;
  float3 tmax_neg = (cell_min - ray.origin) / ray.dir;
  float3 tmax_pos = (cell_max - ray.origin) / ray.dir;
  *tmax = (ray.dir < 0) ? tmax_neg : tmax_pos;
  *step = (ray.dir < 0) ? (float3)(-CELL_SIZE, -CELL_SIZE, -
CELL_SIZE) :
    (float3)(CELL_SIZE, CELL_SIZE, CELL_SIZE);
  *tdelta = fabs((float3)(CELL_SIZE, CELL_SIZE, CELL_SIZE) /
ray.dir);
```

```
}
```

Здесь `tmax` в своих компонентах хранит расстояние от точки начала луча до точки пересечения с плоскостями разбиения, параллельными XY, XZ, YZ; `step` – длина шага для перехода в соседнюю ячейку, `tdelta` – расстояние от точки пересечения луча с плоскостью разбиения до точки пересечения луча со следующей такой же параллельной плоскостью.

Данные значения наглядно проиллюстрированы на рис. 3.2 для двумерного случая.

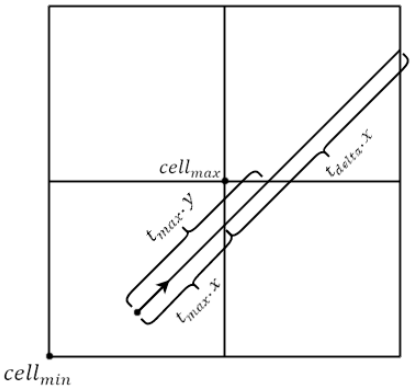



Рис. 3.2. Иллюстрация DDA_prepare

- Обход сетки:

Смотрим, расстояние до какой плоскости от текущей точки наименьшее, шагаем по соответствующей координате:

```
void DDA_step(float3* current_pos, float3* tmax, float3 step, float3
tdelta)
{
    if ((*tmax).x < (*tmax).y) {
        if ((*tmax).x < (*tmax).z) {
             (*tmax).x += tdelta.x;
             (*current_pos).x += step.x;
        } else {
             (*tmax).z += tdelta.z;
```

```
            (*current_pos).z += step.z;
        }
    } else {
        if ((*tmax).y < (*tmax).z)
                {
            (*tmax).y += tdelta.y;
            (*current_pos).y += step.y;
        } else {
            (*tmax).z += tdelta.z;
            (*current_pos).z += step.z;
        }
    }
}
```

Обход сетки показан для двумерного случая на рис. 3.3:

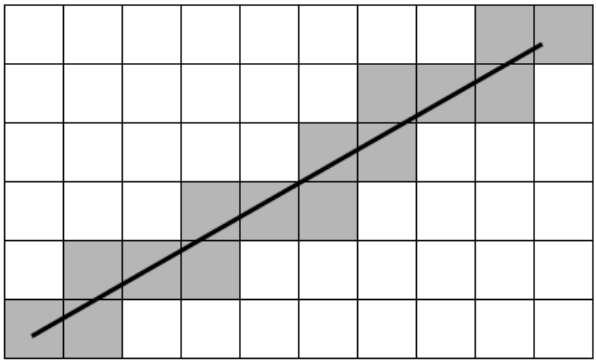


Рис. 3.3. Иллюстрация DDA_step

## 3.2. Иерархия ограничивающих объемов

Иерархия ограничивающих объемов (англ. Bounding Volume Hierarchy, BVH) – подход ускорения процесса трассировки, основанный на разбиении объектов, при котором элементы сцены составляют иерархию стоящих отдельно друг от друга множеств объектов [3, с. 255]. На рис. 3.4 показана иерархия ограничивающих объемов простой сцены.

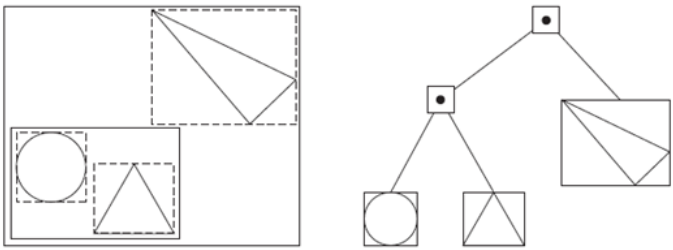


Рис. 3.4. Иерархия ограничивающих объёмов

Примитивы хранятся в листах дерева, а каждый внутренний узел дерева хранит параллелепипед, ограничивающий примитивы, которые лежат в нём. Таким образом, при обходе сцены мы можем отбросить поддеревья, соответствующие ограничивающим объемам, с которыми луч не пересекается.

Свойство разбиения объектов заключается в том, что каждый примитив встречается в иерархии только один раз, в отличие от разбиения пространства, где каждый примитив может лежать в нескольких областях разбиения, и проверка на пересечение с ним может быть проведена несколько раз, когда луч проходит сквозь соседние области. Другое свойство – фиксированный объём памяти, требуемый для хранения BVH-структуры. Для BVH, использующей двоичное дерево, которое хранит только один примитив в каждом листе, общее количество узлов составляет $2n - 1$, где $n$ – количество примитивов. Из них $n$ листов и $n - 1$ внутренних узлов.

Иерархия ограничивающих объёмов отличается более высокой скоростью построения от деревьев разбиения пространства, которые, в свою очередь, дают более высокую скорость обхода дерева. Кроме того, BVH, в общем случае, численно более устойчивы и менее склонны к пропуску пересечений из-за ошибок округления.

### 3.2.1. Построение дерева

Построение BVH-дерева состоит из трёх этапов. На первом вычисляется информация об ограничивающем объёме каждого примитива и сохраняется в массиве, который будет использован в процессе построения дерева. На втором шаге строится дерево, согласно выбранному алгоритму. В результате получается бинарное дерево, в котором каждый внутренний узел хранит указатели на дочерние узлы, и каждый лист указывает на примитивы, хранящиеся в нём. На третьем шаге дерево преобразуется в более компактное и эффективное представление без указателей, которое и будет использоваться в процессе рендеринга.

После того, как ограничивающие объемы каждого из примитивов вычислены, нам необходимо разбить их на два дочерних поддерева. Для $n$ примитивов у нас есть $2^n - 2$ вариантов разбиения для получения двух не пустых объёмов. На практике, при построении BVH, рассматривают разбиение только вдоль осей координат, что сокращает количество вариантов разбиения до $6n$.

Сначала выбирается ось, вдоль которой будет осуществляться разбиение примитивов. Мы выбираем ту, которая соответствует наибольшей проекции ограничивающего объёма центральных точек рассматриваемого набора примитивов. (Альтернативный подход – попробовать разбить объекты по трём осям и выбрать лучший результат, но на практике основной подход достаточно хорошо себя демонстрирует). рис. 3.5 иллюстрирует данную стратегию.

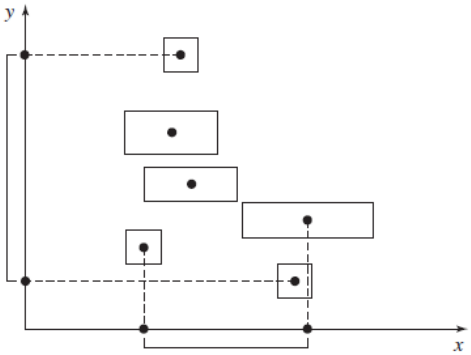



Рис. 3.5. Выбор оси, вдоль которой будет осуществляться разбиение примитивов. Мы выбираем ось с наибольшей величиной проекции ограничивающего объёма центральных точек примитивов

Если область, ограничивающая центральные точки объектов, имеет нулевой объём, рекурсия останавливается и создаётся лист, хранящий в себе примитив, иначе разбиение двух дочерних областей продолжается.

Основная задача разбиения здесь – выбрать такое разбиение примитивов, при котором два результирующих множества не будут сильно пересекаться, иначе нам придётся более часто обходить оба дочерних поддерева, что будет занимать дольше времени.

Простым методом разбиения является разбиение на две области относительно точки, лежащей в середине проекции на выбранную ось. Два множества примитивов образуется путём определения, с какой стороны относительно точки разбиения они находятся. Может произойти так, что примитивы имеют большие ограничивающие объёмы так, что их не разбить таким образом на две группы. В этом случае мы формируем два множества одинакового размера: в первое множество отправляются половина объектов, имеющих наименьшее значение центральной координаты вдоль оси разбиения, в другое

множество – оставшаяся половина с наибольшим значением координаты.

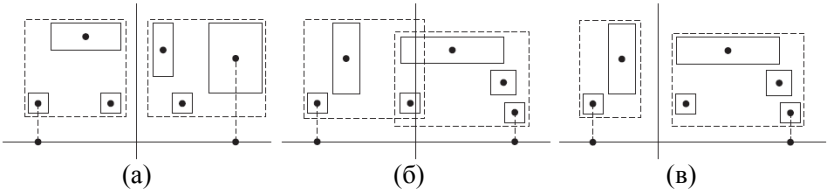



Рис. 3.6. Разбиение примитивов относительно центральной точки на выбранной оси. Ограничивающие объёмы двух дочерних множеств показаны штриховой линией

Для такого расположения примитивов, как показано на рис. 3.6а, метод разбиения относительно центральной точки работает успешно. Для расположения примитивов, как на рис. 3.6б, данный метод является субоптимальным решением: два ограничивающих объёма пересекаются. Если же эти объекты разбить, как показано на рис. 3.6в, ограничивающие объёмы будут меньше, и не будут пересекаться, что даст лучшую производительность при рендеринге.

Однако, большинство пакетов трассировки используют более «умный» методом разбиения, основанный на критерии Surface Area Heuristic (SAH). Модель SAH оценивает расходы на нахождение пересечений, включая время, затраченное на обход узлов дерева и время, затраченное на нахождение пересечение луча и примитивов, после чего происходит минимизация данного критерия по разбиениям.

Идея, использующаяся в модели SAH, заключается в следующем. В любой момент при построении ускоряющей структуры (разбиение объектов или разбиение пространства), мы можем создать узел с листом для заданной области и геометрии. В этом случае, любой луч, который пересечёт эту область, будет проверен на пересечение с содержащейся в ней геометрии, затратив время (цену):

$$\sum_{i=1}^{N} t_{\text{isect}}(i)$$

где $N$ – количество примитивов в области и $t_{isect}(i)$ – время нахождения пересечения луча с $i$-ым примитивом.

Другим вариантом является разбить область на две дочерних. В этом случае, будет затрачена цена

$$c(A,B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i)$$

где $t_{\text{trav}}$ – время, которое тратится на обход внутреннего узла и на определение, какой дочерний узел пересечёт луч, $p_A$ и $p_B$ – вероятности того, что луч пересечёт области $A$ и $B$ соответственно $a_i$ и $b_i$ – индексы объектов в дочерних областях, $N_A$ и $N_B$ – количество объектов в этих двух областях соответственно. Выбор разбиения влияет как на вероятности, так и на количество примитивов по каждую сторону разбиения.

Мы можем допустить, что время $t_{\text{isect}}(i)$ одинаково для всех примитивов, на самом деле, это предположение не так далеко от реальности, и практически не влияет на производительность.

Вероятности $p_A$ и $p_B$ могут быть вычислены, исходя из определения геометрической вероятности (рис. 3.7). Можно показать, что для выпуклой области $B$, содержащейся в другой выпуклой области $A$, условная вероятность, что равномерно распределенный случайный луч, проходящий через $A$, пересечет $B$, равен отношению их площадей $s_B$ и $s_A$:

$$p(B|A) = \frac{s_B}{s_A}$$

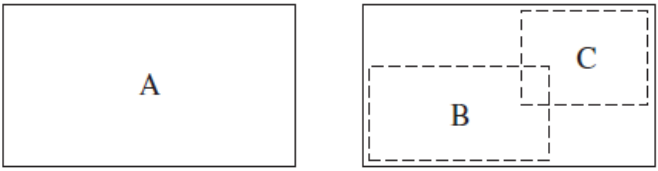



Рис. 3.7. Если узел иерархии с площадью $s_A$ разбивается на два дочерних с площадями $s_B$ и $s_C$, вероятности, что если луч пересёк $A$, также пересечёт $B$ и $C$, равны $\frac{s_B}{s_A}$ и $\frac{s_C}{s_A}$, соответственно

Вместо того, чтобы рассматривать все $2n$ вариантов разбиения вдоль выбранной оси и определять по приведенному критерию, какое разбиение лучшее, мы можем разбить область на некоторое фиксированное количество одинаковых интервалов. Этот подход значительно быстрее, при этом полученное разбиение даёт практически такой же по эффективности результат. Данная идея иллюстрирована на рис. 3.8.

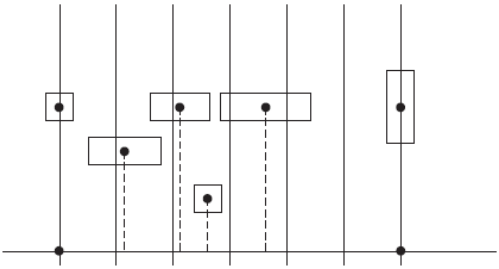


Рис. 3.8. Выбор плоскости разбиения с использованием Surface Area Heuristic при построении BVH

Для каждого из вариантов мы можем определить количество примитивов, лежащих по обе стороны плоскости разбиения. Далее мы считаем значение SAH для каждого из варианта, произвольным образом установив стоимость пересечения луча с примитивом и стоимость обхода узла, например, $t_{\text{isect}} = 1$ и $t_{\text{trav}} = 1/8$. Одно из этих двух

значений должно быть равно 1, так как они являются относительными. После вычисления стоимостей для каждого варианта разбиения, мы находим минимальное среди этих значений.

Если выбранное разбиение имеет меньшую стоимость, чем стоимость построения листа с примитивами или количество примитивов в области, которую мы разбиваем больше, чем максимально дозволенное количество объектов в листе, мы распределяем объекты в области по двум дочерним внутренним узлам в соответствии с выбранным разбиением. Иначе мы создаём лист, содержащий примитивы.

После того, как мы построили BVH-дерево, последним шагом является его компактное представление – линейный массив в памяти. Узлы исходного дерева располагаются в порядке увеличения глубины дерева, что означает, что первый дочерний узел располагается в памяти сразу после родительского узла. Смещение в памяти второго дочернего узла мы храним явным образом. На рис. 3.9 показан порядок хранения узлов в памяти.

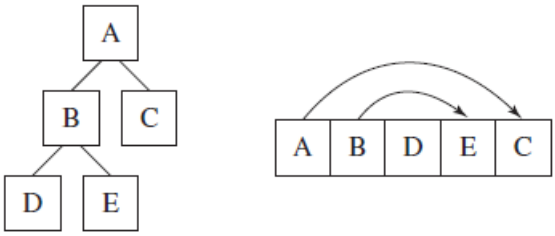



Рис. 3.9. Линейное расположение BVH в памяти

### 3.2.2. Обход дерева

Обход дерева иерархии ограничивающих объёмов достаточно прост: в нём нет рекурсивных вызовов процедур и не требуется больших объёмов памяти для хранения текущего состояния обхода.

При вызове процедуры обхода дерева, мы храним индекс узла, который нам нужно посетить. Изначально он равен 0, то есть, мы начинаем с корня. Кроме того, мы храним в массиве индексы узлов, которые нам нужно будет посетить в будущем, этот массив используется в качестве стека.

Обходя узел, мы проверяем, пересекает ли луч ограничивающий объём узла (или начала луча находится внутри него). Если узел является листом, то мы проверяем пересечение луча и каждого примитива, лежащего в листе, либо переходим к обработке дочерних узлов, добавляя их индексы в стек. Если пересечение не найдено, то мы переходим к обработке следующего узла в стеке (либо обход закончен, если стек пуст).

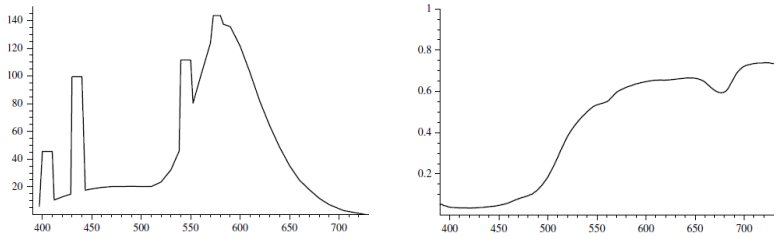
# ГЛАВА 4. УПРАВЛЕНИЕ ЦВЕТОМ И РАДИОМЕТРИЯ

С целью точно описать, каким образом представляется свет и как использовать это представление для вычисления изображений, мы должны разобрать некоторые основы *радиометрии* – науке о распространении и измерении электромагнитного излучения в окружающей среде. Наибольший интерес для нас в рендеринге представляют электромагнитное излучение с длинами волн ($\lambda$) в диапазоне приблизительно от 380 нм до 780 нм, соответствующем видимому для человека излучению. Относительно данного диапазона,

малым длинам волн ($\lambda \approx 400$ нм) соответствуют синие цвета, средним ($\lambda \approx 550$ нм) – зелёные, и большим длинам волн ($\lambda \approx 650$ нм) – красные. Далее мы рассмотрим четыре основных величины, которые описывают электромагнитное излучение: поток, силу излучения, облучённость и энергетическую яркость. Эти величины описаны своей спектральной плотностью излучения (англ. Spectral Power Distribution, SPD) – функция распределения, которая описывает интенсивность излучения в зависимости от длины волны [3, с. 313].

## 4.1. Спектральная плотность излучения

SPD реальных объектов может быть довольно сложной, на рис. 4.1 показан график спектрального распределения излучения флуоресцентного источника света (а) и спектральное распределение отражательной способности лимонной корки (б). Рендеру для выполнения вычислений с SPD требуется компактный, эффективный и точный способ представления подобных функций.




(а) (б)

Рис. 4.1. Примеры функций спектральной плотности излучения

Данный вопрос решается путём нахождения хороших базисных функций для представления SPD. Суть использования базисных функций заключается в отображении бесконечномерного пространства всех возможных функций в пространство малой размерности коэффициентов $c_i \in \mathbb{R}$. Например, тривиальной базисной функцией

является константная функция $B(\lambda) = 1$. Произвольная SPD может быть представлена в этом базисе единственным коэффициентом $c$, равным её среднему значению. Это, очевидно, очень грубая аппроксимация, так как подавляющее большинство SPD имеют более сложную форму. Поэтому для представления спектральной плотности излучения используются различные другие базисные функции, по-своему представляющие компромисс между точностью и сложностью операций с ними.

## 4.2. Пространство цветов XYZ

Замечательной особенностью зрительной системы человека является то, что она позволяет представлять цвета для восприятия с помощью всего трёх чисел с плавающей точкой [9]. Трехкомпонентная теория цветового зрения говорит, что все видимые SPD могут быть точно представлены для человека, как наблюдателя, с помощью трёх значений: $x_\lambda$, $y_\lambda$ и $z_\lambda$. Для заданной SPD $S(\lambda)$, данные значения получаются путём интегрирования её произведения с кривыми цветового соответствия $X(\lambda)$, $Y(\lambda)$ и $Z(\lambda)$ (рис. 4.2).

$$x_\lambda = \frac{1}{\int Y(\lambda)} \int_\lambda S(\lambda)\, X(\lambda) \mathrm{d}\lambda$$

$$y_\lambda = \frac{1}{\int Y(\lambda)} \int_\lambda S(\lambda)\, Y(\lambda) \mathrm{d}\lambda$$

$$z_\lambda = \frac{1}{\int Y(\lambda)} \int_\lambda S(\lambda)\, Z(\lambda) \mathrm{d}\lambda$$

Эти кривые были утверждены Международной комиссией по освещению (фр. Commission Internationale de l'Éclairage, CIE) в 1931 году в качестве стандарта после того, как в 20-х годах независимо друг от друга учёные Джон Гилд и Дэвид Райт провели серию опытов над

человеческим зрением. Для создания удобной и универсальной системы спецификации цвета комитет CIE провел усреднения данных измерений Гилда и Райта. Для модели брались условия, чтобы компонента $Y$ соответствовала визуальной яркости сигнала, координата $Z$ соответствовала отклику s-колбочек, а координата $X$ была всегда неотрицательной.

Можно заметить, что различные SPD могут иметь схожие значения $x_\lambda$, $y_\lambda$ и $z_\lambda$. Для человека свет с такими SPD будет выглядеть одинаково. Пары таких спектров называются метамерами.

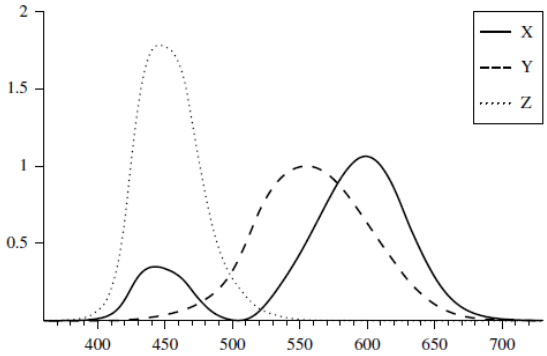


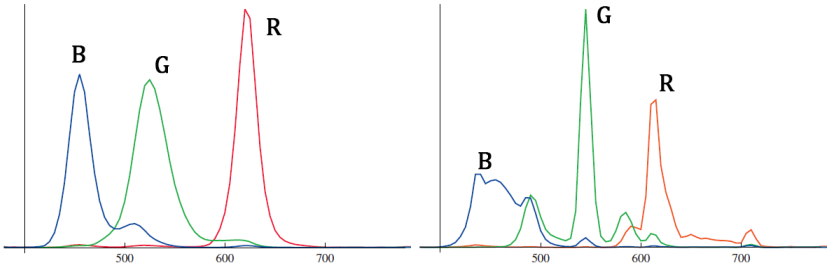

Рис. 4.2. Кривые цветового соответствия, определенные в модели CIE XYZ

Это приводит нас к тонкому вопросу о представлении функций спектральной плотности излучения. Большинство цветовых пространств пытаются описать видимые цвета, используя только три коэффициента, используя теорию трёх цветов. Хотя XYZ хорошо показывает себя для представления отдельно заданной SPD, эта модель не является хорошим набором базисных функций для работы со спектрами. Например, значения XYZ будут хорошо представлять визуальное восприятие кожицы лимона или флуоресцентного

источника света в отдельности, произведение соответствующим им XYZ значений дадут отличный цвет по сравнению с произведением более точных репрезентаций спектров, а затем вычисления результирующего XYZ значения.

## 4.3. Пространство цветов RGB

Когда мы хотим отобразить RGB цвет на экране, спектр, который на самом деле излучается люминофором ЭЛТ монитора, элементами LCD или LED дисплея, или ячейками плазменной панели, определяется взвешенной суммой трёх кривых спектрального отклика, соответствующих красному, синему и зеленому цвету [3, с. 325]. На рис. 4.3 показаны кривые отклика LED и LCD дисплеев, и можно увидеть, что они существенно отличаются. На рис. 4.4 изображены кривые спектральной плотности излучения RGB цвета (0.6,0.3,0.2) на LED и LCD мониторах.




(а) (б)

Рис. 4.3. Кривые отклика для LCD (а) и LED (б) дисплея

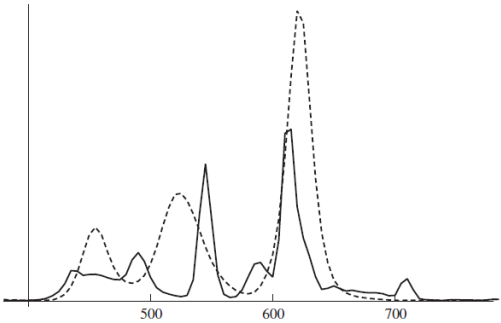



Рис. 4.4. SPD отображения RGB цвета (0.6, 0.3, 0.2) на LED и LCD дисплеях

Пример, проиллюстрированный на рис. 4.4, показывает, что использование заданного RGB цвета на самом деле будет иметь значимость только в том случае, когда мы знаем характеристики дисплея, на котором этот цвет будет отображаться.

Имея заданные значения $(x_\lambda, y_\lambda, z_\lambda)$ в пространстве XYZ, мы можем получить соответствующие значения в пространстве RGB, используя кривые отклика $R(\lambda)$, $G(\lambda)$, $B(\lambda)$ конкретного дисплея, путём «свертки» их с спектром $S(\lambda)$:

$$r = \int R(\lambda)\, S(\lambda)\mathrm{d}\lambda = \int R(\lambda)\big(x_\lambda X(\lambda) + y_\lambda Y(\lambda) + z_\lambda Z(\lambda)\big)\mathrm{d}\lambda$$

$$= x_\lambda \int R(\lambda)X(\lambda)\mathrm{d}\lambda + y_\lambda \int R(\lambda)Y(\lambda)\mathrm{d}\lambda$$

$$+ z_\lambda \int R(\lambda)Z(\lambda)\mathrm{d}\lambda$$

Как мы видим, выражение представляет собой взвешенную сумму интегралов, которые мы можем предварительно рассчитать для заданных кривых отклика, и преобразовывать XYZ цвета в пространство RGB путём умножения на матрицу преобразования.

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{pmatrix} \int R(\lambda)X(\lambda)\mathrm{d}\lambda & \int R(\lambda)Y(\lambda)\mathrm{d}\lambda & \int R(\lambda)Z(\lambda)\mathrm{d}\lambda \\ \int G(\lambda)X(\lambda)\mathrm{d}\lambda & \int G(\lambda)Y(\lambda)\mathrm{d}\lambda & \int G(\lambda)Z(\lambda)\mathrm{d}\lambda \\ \int B(\lambda)X(\lambda)\mathrm{d}\lambda & \int B(\lambda)Y(\lambda)\mathrm{d}\lambda & \int B(\lambda)Z(\lambda)\mathrm{d}\lambda \end{pmatrix} \begin{bmatrix} x_\lambda \\ y_\lambda \\ z_\lambda \end{bmatrix}$$

В различных стандартах заданы разные матрицы преобразования. Одним из наиболее известных стандартов является sRGB, созданный совместно компаниями HP и Microsoft в 1996 году для унификации использования модели RGB в мониторах, принтерах и Интернет-сайтах. sRGB использует основные цвета, описанные стандартом BT.709, аналогично студийным мониторам и HD-телевидению, а также гамма-коррекцию, аналогично мониторам с электронно-лучевой трубкой. Такая спецификация позволила sRGB в точности отображаться на обычных ЭЛТ-мониторах и телевизорах, что стало в своё время основным фактором, повлиявшим на принятие sRGB в качестве стандарта. Преобразование XYZ в RGB, согласно стандарту sRGB выглядит следующим образом:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{pmatrix} 3.2404542 & -1.5371385 & -0.4985314 \\ -0.9692660 & 1.8760108 & 0.0415560 \\ 0.0556434 & -0.2040259 & 1.0572252 \end{pmatrix} \begin{bmatrix} x_\lambda \\ y_\lambda \\ z_\lambda \end{bmatrix}$$

## 4.4. Радиометрия

Радиометрия предоставляет набор идей и математических инструментов для описания распространения и отражения света в пространстве. Она формирует основу для алгоритмов рендеринга. Стоит отметить, что изначально радиометрия не основывалась на физической оптике, а представляла собой абстракцию, предполагая, что освещение представляет собой поток набора частиц. Таким образом, в

данную концепцию не вписываются такие понятия, как поляризация, дифракция и интерференция света, хотя позже в радиометрию были добавлена теория из уравнений Максвелла, в результате чего радиометрия получила прочную физическую основу.

Теория о *переносе излучения* основана на радиометрических принципах и оперирует на уровне геометрической оптики, где макроскопичеких свойств света достаточно для описания того, как он взаимодействует с объектами, имеющими намного больший размер, чем длина световой волны.

При рендеринге мы предполагаем, что геометрическая оптика является адекватной моделью для описания распространения света. Это приводит нас к нескольким основным допущениям, которые мы будем явным образом использовать:

- **Линейность**. Совокупный результат двух входов оптической системы равен сумме результатов каждого из двух входов по отдельности;
- **Сохранение энергии**. При отражении света от поверхности, либо при его распространении сквозь несущую среду не должно появляться большего количества энергии по сравнению с тем, сколько её было изначально;
- **Отсутствие поляризации и фотолюминесценции**. Учёт данных эффектов будет занимать большее количество времени при рендеринге, хотя при этом они бы не добавили большой практической ценности для нас;
- **Устойчивое состояние**. Считается, что освещение в окружающей среде достигло равновесия, и его распределение не меняется во времени.

Основными физическими величинами в рендеринге являются поток, облучённость/излучательность, сила излучения и энергетическая яркость. Каждая из них может быть получена из энергии (измеряемой в джоулях), применяя к ней пределы по времени, площади поверхности и направлениям. В общем случае, каждая из этих радиометрических величин зависит от частоты волны, но мы не будем записывать это в явном виде.

### 4.4.1. Энергия излучения

Источники света излучают фотоны, каждый из которых имеет соответствующую длину волны и переносит определенное количество энергии. Фотон с длиной волны $\lambda$ имеет энергию

$$Q = \frac{hc}{\lambda}$$

где $c = 299\,472\,458$ м/с – скорость света, $h \approx 6.626 \cdot 10^{-34}$ м$^2$кг/с – постоянная Планка.

### 4.4.2. Поток излучения

Энергия измеряет количество работы, выполненной в течение некоторого периода времени, однако, из сделанного нами предположении об устойчивом состоянии освещения, нас больше интересует измерение мгновенного значения энергии. Поток излучения, также называемый мощностью, является количеством энергии, проходящей сквозь поверхность или область пространства в единицу времени. Поток излучения может быть найден путём взятия предела:

$$\Phi = \lim_{\Delta t \to 0} \frac{\Delta Q}{\Delta t} = \frac{\mathrm{d}Q}{\mathrm{d}t}$$

Поток измеряется в джоулях в секунду (Дж/с), или ваттах (Вт).

Например, для источника освещения, излучавшего $Q = 200\,000$ Дж в течение часа, если в течение времени этого времени

излучалось одинаковое количество энергии, мы можем найти поток излучения:

$$\Phi = 200\,000 \text{ Дж} / 3\,600 \text{ с} \approx 55.6 \text{ Вт}$$

С другой стороны, имея заданный поток в качестве функции от времени, мы можем найти путём интегрирования энергию, излучённую в течение заданного периода времени:

$$Q = \int_{t_0}^{t_1} \Phi(t)\mathrm{d}t$$

Полное излучение источников света описывается в терминах потока. На рис. 4.5 демонстрируется, что поток точечного источника света, равен количеству энергии, проходящей сквозь воображаемые сферы вокруг него. Значение потока, проходящего через обе сферы, является одинаковым.

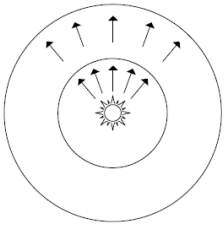


Рис. 4.5. Поток излучения Ф равен количеству энергии, проходящей сквозь сферическую поверхность

### 4.4.3. Облучённость и излучательность

Зная конечную площадь $A$, через которую проходит поток излучения, мы можем определить среднюю плотность потока, как $E = \Phi/A$. Эта величина может называться *облучённостью* ($E$), если излучение падает на поверхность, либо *излучательностью*, если излучение покидает поверхность, и измеряется в Вт/м$^2$. Для точечного источника света, показанного на рис. 4.5, облучённость точки на внешней сфере меньше облучённости точки, лежащей на внутренней

сфере, так как площадь внешней сферы больше. В частности, если точечный источник освещения излучает одинаковое количество энергии во всех направлениях, то точка на сфере с радиусом $r$ имеет облучённость

$$E = \frac{\Phi}{4\pi r^2}$$

Эта формула объясняет, почему значение энергии в точке обратно пропорционально квадрату расстояния до источника света.

В более общем виде мы можем определить облучённость и излучательность в точке $p$ путём взятия предела:

$$E(p) = \lim_{\Delta A \to 0} \frac{\Delta\Phi(p)}{\Delta A} = \frac{\mathrm{d}\Phi(p)}{\mathrm{d}A}$$

Данное уравнение является основой закона Ламберта, который говорит, что количество излучения, падающего на поверхность, пропорционально косинусу угла между направлением света и нормалью поверхности (рис. 4.6).

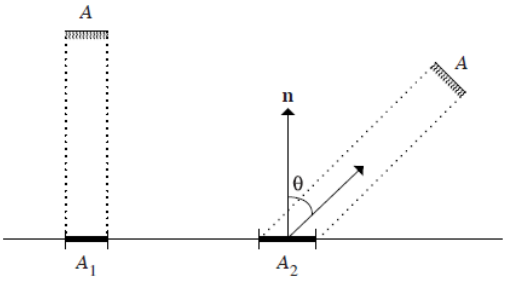



Рис. 4.63. Иллюстрация закона Ламберта

Рассмотрим источник света с площадью $A$ и потоком Ф. Если свет падает перпендикулярно на поверхность (слева на рис. 4.6), площадь облучённой поверхности $A_1$ равна $A$. Облучённость любой точки внутри $A_1$ равна

$$E_1 = \frac{\Phi}{A}$$

Однако, если свет падает на поверхность не под прямым углом, площадь облучённой поверхности становится больше. Если $A$ мала, то площадь $A_2$ приблизительно равна $A/\cos\theta$. Тогда облучённость любой точки внутри $A_2$ равна

$$E_2 = \frac{\Phi\cos\theta}{A}$$

Мы можем также интегрировать облучённость по площади для того, чтобы найти поток излучения:

$$\Phi = \int_A E(p)\mathrm{d}A$$

### 4.4.4. Сила излучения

Продолжим рассматривать точечные источники света. Если мы расположим такой источник света внутри единичной сферы, мы можем вычислить угловую плотность излучённого потока, которая называется силой излучения. Она равна пределу отношения светового потока к телесному углу при вершине конуса, соответствующего рассматриваемому направлению излучения, при стремлении телесного угла к нулю, и измеряется в ваттах на стерадиан (Вт/ср):

$$I = \lim_{\Delta\omega\to 0}\frac{\Delta\Phi}{\Delta\omega} = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega}$$

Как и ранее, мы можем получить поток, проинтегрировав силу излучения по всем направлениям Ω:

$$\Phi = \int_\Omega I(\omega)\mathrm{d}\omega$$

Сила излучения описывает распределение мощности источника света по направлениям, но она имеет значимость только для точечных источников света.

### 4.4.5. Энергетическая яркость

Наконец, самой важной радиометрической величиной является *энергетическая яркость* $L$. Облучённость и излучательность дают нам дифференциальную мощность на дифференциальной площадке, но они не рассматривают распределение мощности по направлениям. Энергетическая яркость учитывает это, и определяется, как

$$L(p,\omega) = \lim_{\Delta\omega\to 0} \frac{\Delta E_\omega(p)}{\Delta\omega} = \frac{\mathrm{d}E_\omega(p)}{\mathrm{d}\omega}$$

где $E_\omega$ обозначает облучённость на поверхности, перпендикулярной направлению $\omega$.

Энергетическая яркость – это плотность потока на площадь, и на телесный угол. В терминах потока, она определяется, как

$$L = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega\ \mathrm{d}A^\perp}$$

где $\mathrm{d}A^\perp$ – проекция площади $\mathrm{d}A$ на воображаемую плоскость, перпендикулярную направлению $\omega$, как показано на рис. 4.7.

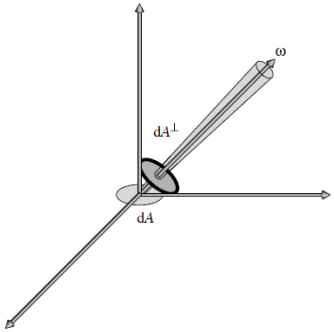



Рис. 4.7. Энергетическая яркость определяется, как плотность потока на телесный угол . $\mathrm{d}\omega$ и спроецированную площадь $\mathrm{d}A^\perp$

Когда свет взаимодействует с поверхностями в сцене, функция энергетической яркости $L$, в основном, не является непрерывной на границе раздела двух сред. Поэтому имеет смысл брать односторонние

пределы в местах разрыва для отличия функций энергетической яркости по разные стороны поверхности

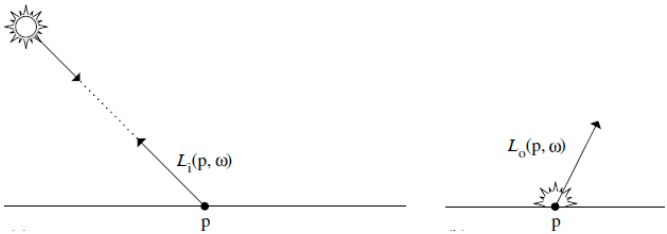
$$L^+(p,\omega) = \lim_{t\to 0^+} L\big(p + tn_p, \omega\big)$$

$$L^-(p,\omega) = \lim_{t\to 0^-} L\big(p + tn_p, \omega\big)$$

где $n_p$ – нормаль поверхности в точке $p$. Однако, отслеживание односторонних пределов делает повествование слишком громоздким. Мы можем это исправить путём введения различия между энергетической яркостью, которая приходит в точку (например, освещение от источника), и энергетической яркостью, которая покидает точку (например, отражение от поверхности). Распределение энергетической яркости, которая приходит в точку $p$, может быть математически описана в виде функции позиции и направления. Эта функция обозначается, как $L_i(p,\omega)$. Функция, соответствующая энергетической яркости, покидающей точку, обозначается, как $L_o(p,\omega)$. В обоих случаях вектор $\omega$ направлен из точки $p$ (рис 4.8).

$$L_i(p,\omega) = \begin{cases} L^+(p,-\omega), & \omega \cdot n_p > 0 \\ L^-(p,-\omega), & \omega \cdot n_p < 0 \end{cases}$$

$$L_o(p,\omega) = \begin{cases} L^+(p,\omega), & \omega \cdot n_p > 0 \\ L^-(p,\omega), & \omega \cdot n_p < 0 \end{cases}$$




(а) (б)

Рис. 4.8. Функции входящей (а) и исходящей (б) энергетической яркости описывают её распределение в точках $p$ по направлениям $\omega$

Кроме того, свойство рассматриваемых функций заключается в том, что в тех точках, где $L(p, \omega)$ непрерывна, так, что $L^+ = L^-$, что означает

$$L_o(p, \omega) = L_i(p, -\omega) = L(p, \omega)$$

## 4.5. Поверхностное отражение

При попадании света на поверхность, она рассеивает свет, отражая его некоторую часть обратно в окружающую среду. Для моделирования данного эффекта нам необходимо знать спектральное распределение и распределение по направлениям отражённого света. Например, кожица лимона поглощает свет в «синем» диапазоне, при этом отражает практически весь свет в «красном» и «зелёном» диапазонах. Таким образом, освещённый белым светом лимон будет выглядеть жёлтым. При этом кожица имеет одинаковый цвет вне зависимости, с какой стороны на неё смотреть, за исключением некоторых направлений, вдоль которых мы можем увидеть блики. Совсем другим примером является зеркало: свет, который мы увидим отражённым в точке, практически полностью зависит от направления нашего взгляда.

Для описания этого механизма существует абстракция: BRDF (англ. Bidirectional Reflectance Distribution Function – «двунаправленная функция распределения отражательной способности»). BRDF задаёт формальное описание отражения от поверхности. Рассмотрим рис. 4.9: нам нужно знать, какое количество энергетической яркости $L_o(p, \omega_o)$ покидает поверхность в направлении $\omega_o$, к наблюдателю в зависимости от энергетической яркости $L_i(p, \omega_i)$, приходящей в точку вдоль направления $\omega_i$.

Если направление $\omega_i$ рассматривать, как дифференциальный конус направлений, то дифференциальная облучённость в точке $p$ равна

$$\mathrm{d}E(p,\omega_i) = L_i(p,\omega_i)\cos\theta_i\,\mathrm{d}\omega_i$$

Дифференциальная энергетическая яркость будет отражена в направлении $\omega_o$ в зависимости от этой облучённости. Так как мы сделали предположение о линейности оптической системы, отражённая энергетическая яркость будет линейно пропорциональна облучённости:

$$\mathrm{d}L_o(p,\omega_o) \propto \mathrm{d}E(p,\omega_i)$$

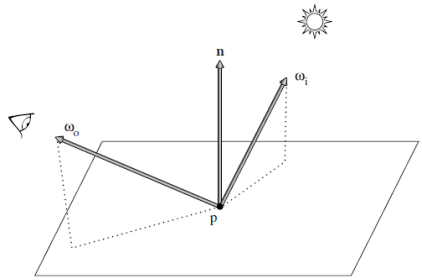



Рис. 4.9. Направления входящего и исходящего излучения, используемые в BRDF

BRDF задаёт пропорциональность между облучённостью и энергетической яркостью для пары направлений $\omega_i$ и $\omega_o$ в точке $p$:

$$f_r(p,\omega_o,\omega_i) = \frac{\mathrm{d}L_o(p,\omega_o)}{\mathrm{d}E(p,\omega_o)} = \frac{\mathrm{d}L_o(p,\omega_o)}{L_i(p,\omega_i)\cos\theta_i\,\mathrm{d}\omega_i}$$

Физически корректные BRDF имеют два важных свойства:

1. **Симметричность**. Для всех пар направлений $\omega_i$ и $\omega_o$, $f_r(p,\omega_o,\omega_i) = f_r(p,\omega_i,\omega_o)$;
2. **Сохранение энергии**. Полная энергия отражённого света меньше либо равна энергии входящего света для всех направлений:

$$\int_{\mathcal{H}^2(n)} f_r(p, \omega_o, \omega') \cos\theta' \, \mathrm{d}\omega' \leq 1$$

Используя определение BRDF, мы имеем

$$\mathrm{d}L_o(p, \omega_o) = f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i$$

Мы можем проинтегрировать это уравнение по полусфере, ориентированной вдоль нормали поверхности $\mathcal{H}^2(n)$ для нахождения энергетической яркости, рассеиваемой в направлении $L_o$ путём суммирования падающего на точку $p$ излучения со всех направлений $\omega_i$:

$$L_o(p, \omega_o) = \int_{\mathcal{H}^2} f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i$$

Таким образом, мы получаем фундаментальное в компьютерной графике уравнение рендеринга, описывающее, как распределение входящего излучения в точке преобразуется в распределение исходящего излучения.

# ГЛАВА 5. МОДЕЛИ ОТРАЖЕНИЯ

В предыдущей главе мы ввели понятие двунаправленной функции распределения отражательной способности (BRDF) – абстракцию для описания отражения света от поверхности.

Модели отражения получают различными способами [3, с. 507]:

**Измерение**. BRDF материалов измеряют в лабораторных условиях, после чего используются табличные данные, либо коэффициенты базисных функций;

**Феноменологическая модель.** Попытка описать внешнее поведение явления так, как оно бы вело себя в реальности, однако

данное описание непосредственно из фундаментальной теории не следует.

**Симуляция.** При наличии низкоуровневой информации о составе поверхности, например, о цвете и составе частиц в краске, либо расположению и толщине нитей в ткани, мы можем смоделировать распространение света от микрогеометрии, и получить набор коэффициентов базисных функций для использования в рендеринге.

**Физическая оптика.** Некоторые модели отражения были получены на основании подробного описания поведения света путем рассматривания его в качестве волны и вычисления решения уравнений Максвелла.

**Геометрическая оптика.** При наличии низкоуровневого описания поверхности и её геометрических свойств, в некоторых случаях можно получить модель отражения непосредственно из этих описаний. При этом сложные волновые эффекты вроде поляризации игнорируются.

Отражения от поверхности можно разделить на 4 типа: диффузное (англ. diffuse), глянцевое (англ. glossy specular), зеркальное (англ. perfect specular), и световозвращающее (англ. retro-reflective). Большинство реальных поверхностей комбинируют в себе эти 4 типа.

## 5.1. Касательное пространство

Для начала введём систему координат касательного пространства (англ. tangent space) к поверхности примитива, в которой будут происходить вычисления отражений (рис 5.1). Она задаётся тремя ортонормированными базисными векторами $(s, t, n)$, где $s$, $t$ – касательные векторы к поверхности в точке пересечения луча и примитива, $n$ – нормаль к поверхности.

При этом нормализованный вектор направления $\omega$ удобно выражать в сферических координатах $(\theta, \varphi)$.

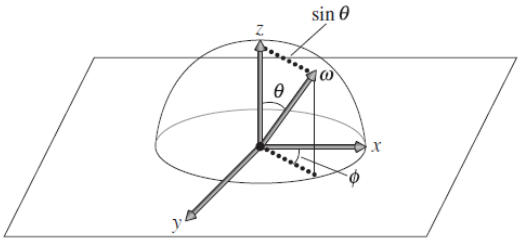



Рис. 5.1. Система координат касательного пространства

## 5.2. Зеркальное отражение

Поведение света при отражении от идеально гладкой поверхности легко описывается как физической, так и геометрической оптикой: для входящего излучения, падающего под направлением $\omega_i$, всё исходящее излучение распространяется в единственном направлении $\omega_o$, при этом зенитный угол падения к нормали равен углу отражения $\theta_i = \theta_o$, а азимутальный угол отражения равен $\varphi_o = \varphi_i + \pi$.

### 5.2.1. Эффект Френеля

Помимо направления необходимо также вычислить долю входного излучения, которая будет отражена. Формулы Френеля описывают количество отражённого от поверхности излучения в зависимости от показателя преломления вещества и угла падения для двух видов поляризации световой волны: параллельной и перпендикулярной. Поскольку обычно свет не является поляризованным, коэффициент отражения Френеля представляет собой среднее значения квадратов параллельной и перпендикулярной составляющей.

Для диэлектриков формулы Френеля выглядят следующим образом:

$$r_{\parallel} = \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t} \quad r_{\perp} = \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}$$

где $r_{\parallel}$ – коэффициент отражения Френеля для параллельно поляризованного света, $r_{\perp}$ – для перпендикулярно поляризованного, $\eta_i$ и $\eta_t$ – индексы преломления веществ снаружи и внутри границы раздела сред, $\theta_i$ – угол падения света к нормали, $\theta_t$ – угол преломления, вычисляющийся по закону Снеллиуса:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

Для неполяризованного света коэффициент отражения Френеля равен:

$$F_r = \frac{1}{2}({r_{\parallel}}^2 + {r_{\perp}}^2)$$

В отличие от диэлектриков, проводники имеют комплексный индекс преломления $\bar{\eta} = \eta + \mathrm{i}k$. Здесь мнимая часть $k$ – поглощательная способность (коэффициент экстинкции) вещества. Для проводников составляющие $r_{\perp}$ и $r_{\parallel}$ равны:

$$r_{\perp} = \frac{a^2 + b^2 - 2a\cos\theta_i + \cos^2\theta_i}{a^2 + b^2 + 2a\cos\theta_i + \cos^2\theta_i}$$

$$r_{\parallel} = r_{\perp}\frac{\cos^2\theta_i\,(a^2 + b^2) - 2a\cos\theta_i \sin^2\theta_i + \sin^4\theta_i}{\cos^2\theta_i\,(a^2 + b^2) + 2a\cos\theta_i \sin^2\theta_i + \sin^4\theta_i}$$

где $a^2 + b^2 = \sqrt{(\eta^2 - k^2 - \sin^2\theta_i)^2 + 4\eta^2 k^2}$ и $\eta + \mathrm{i}k = \bar{\eta}_t / \bar{\eta}_i$.

На рис. 5.2 представлены графики зависимости отражательной способности некоторых веществ в зависимости от угла отражения. Значения $\eta$ и $k$ взяты из [13].

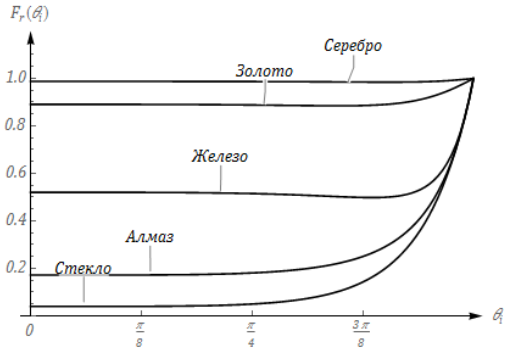



Рис. 5.2. Коэффициенты Френеля различных веществ в зависимости от угла падения

Как мы можем видеть, формулы для точного вычисления коэффициентов Френеля достаточно сложны для использования их в рендеринге в реальном времени, поэтому их можно заменить аппроксимацией Шлика:

$$F_r(\theta_i) = F_0 + (1 - F_0)(1 - \cos\theta_i)^5$$

где $F_0 = \begin{cases} \left(\frac{\eta_i - \eta_t}{\eta_i + \eta_t}\right)^2 & \text{для диэлектриков} \\ \frac{(\eta_i - \eta_t)^2 + k^2}{(\eta_i + \eta_t)^2 + k^2} & \text{для проводников} \end{cases}$

Делая интуитивное предположение о том, что BRDF зеркального отражения равна нулю везде, кроме единственного направления отражения, мы можем записать её, как

$$f_r(\omega_o, \omega_i) = \delta(\omega_i - \omega_r)F_r(\omega_i)$$

где $\omega_r$ – направление отражения света, $\delta(\omega)$ – дельта-функция. Однако, при подставлении этого выражения в уравнение рендеринга и интегрировании его, мы получаем дополнительный множитель $|\cos\theta_r|$

$$L_o(\omega_o) = \int \delta(\omega_i - \omega_r)F_r(\omega_i)L_i(\omega_i)|\cos\theta_i|\mathrm{d}\omega_i = F_r(\omega_i)L_i(\omega_r)|\cos\theta_r|$$

Таким образом, корректная запись BRDF для зеркального отражения:

$$f_r(\omega_o, \omega_i) = F_r(\omega_i) \frac{\delta(\omega_i - \omega_r)}{|\cos \theta_r|}$$

## 5.3. Модель Ламберта

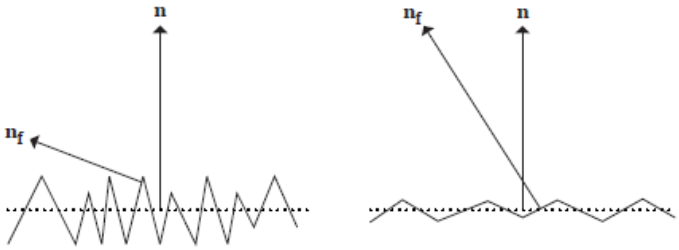
Одной из простейших BRDF является модель Ламберта. Она представляет собой описание диффузной поверхности, одинаково рассеивающей свет во всех направлениях. Хотя эта модель не обладает полной физической корректностью, она приемлема для описания множества реальных материалов, таких, как матовая краска, бетон и т.д. BRDF модели Ламберта выглядит следующим образом:

$$f_r(\omega_o, \omega_i) = \frac{R}{\pi}$$

где $R$ – коэффициент отражения поверхности.

## 5.4. Модель микрограней

Данный подход моделирования отражения основан на идее того, что шероховатая поверхность состоит из множества микрограней (англ. microfacets), которые описываются статистически (рис 5.3).




(а) (б)

Рис. 5.3. Более (а) и менее (б) шероховатые поверхности, состоящие из микрограней

Модель микрограней состоит из различных компонент, которые описывают как распределение микрограней в целом, так и отражательную способность микрограней в отдельности. Для вычисления отражения, необходимо учесть локальные эффекты: одни микрограни могут загораживать или затенять другие, свет может отражаться от одних микрограней к другим. Различные BRDF, основанные на модели микрограней рассматривают данные эффекты с разной степенью точности.

### 5.4.1. Функция распределения нормалей

Функция распределения нормалей (англ. normal distribution function, NDF) $D(\omega_h)$ характеризует долю микрограней на дифференциальной площадке $\mathrm{d}A$, ориентированных вдоль направления $\omega_h$. Наиболее простой является функция распределения Блинна-Фонга:

$$D_{Blinn}(\omega_h) = \frac{1}{\pi\alpha^2}(n \cdot \omega_h)^{\left(\frac{2}{\alpha^2}-2\right)}$$

где $n$ – геометрическая нормаль поверхности, $\alpha$ – коэффициент шероховатости поверхности, $\alpha = \sqrt{\sigma}$, $\sigma$ – среднеквадратическое отклонение микрограней от геометрической нормали.

Другой, более точной и физически корректной NDF является функция распределения GGX:

$$D_{GGX}(\omega_h) = \frac{\alpha^2}{\pi((n \cdot \omega_h)^2(\alpha^2 - 1) + 1)^2}$$

Сравнение NDF Блинна-Фонга и GGX при одинаковом параметре $\alpha$ приведено на рис. 5.4 и рис. 5.5.

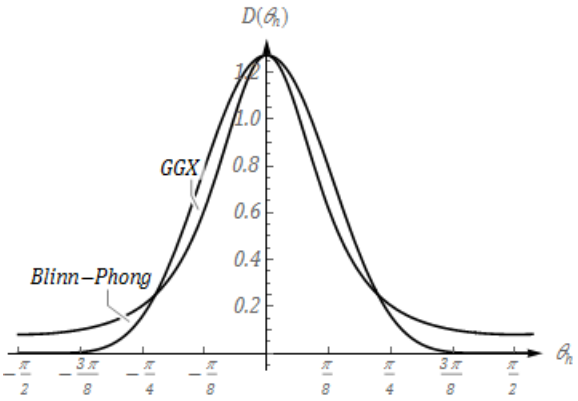



Рис. 5.4. Графики распределений Блинна-Фонга и GGX при $\alpha = 0.5$

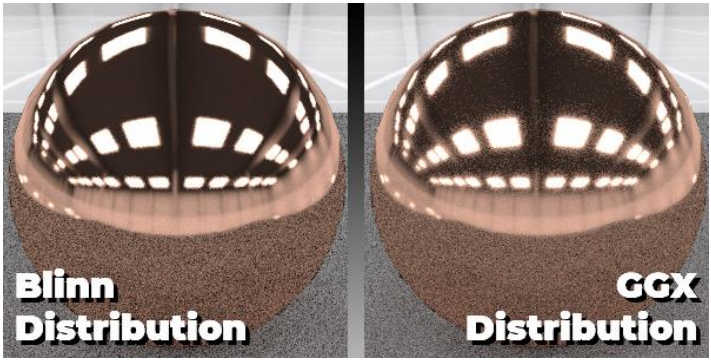



Рис. 5.5. Рендеринг металла с использованием распределения Блинна-Фонга и GGX

### 5.4.2. Геометрическая составляющая

Геометрическая составляющая $G(\omega_o)$ производит учёт взаимного перекрытия и затенения микрограней и характеризует долю перекрытых микрограней вдоль направления отражения $\omega_o$ (рис 5.6).

$$G_{GGX}(\omega_o) = \frac{2(n \cdot \omega_o)}{(n \cdot \omega_o) + \sqrt{\alpha^2 + (1 - \alpha^2)(n \cdot \omega_o)^2}}$$

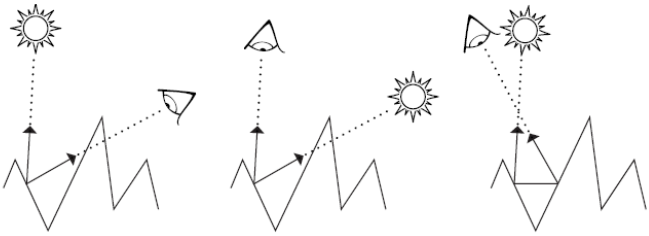


Рис. 5.6. Взаимное перекрытие, затенение и отражение света от микрограней

### 5.4.3. Модель Кука-Торренса

Данная модель объединяет в себе функцию распределения нормалей $D(\omega_h)$, коэффициент Френеля $F_r(\omega_i)$ и геометрическую составляющую $G(\omega_o)$:

$$f_r(\omega_o, \omega_i) = \frac{D(\omega_h)F_r(\omega_i)G(\omega_o)}{4\cos\theta_o \cos\theta_i}$$

здесь $\omega_i$ – вектор падения, $\omega_o$ – вектор отражения, $\omega_h = \frac{\omega_o+\omega_i}{\|\omega_o+\omega_i\|}$ – вектор полпути, соответствующий направлению нормали микрограни.

# ГЛАВА 6. МОДЕЛИРОВАНИЕ КАМЕРЫ

## 6.1. Простая камера

Наиболее простым устройством для получения фотографий является камера-обскура с игольчатым объективом (англ. pinhole camera). Она представляет собой закрытую коробку с крошечным отверстием. Свет проникает через отверстие и попадает на фотобумагу, находящуюся с противоположной стороны коробки [3, с. 355].

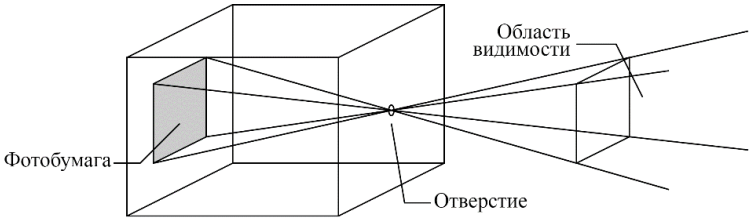



Рис. 6.1. Камера-обскура с игольчатым объективом

На рис. 6.1 мы видим, что, соединив углы фотобумаги (плёнки) с отверстием и продолжив лучи наружу, мы получаем двойную пирамиду, которая называется областью видимости. Объекты, которые находятся вне этой пирамиды, не могут попасть на плёнку.

Так как мы не собираемся располагать объекты внутри камеры-обскуры, мы можем абстрагироваться от приведённого выше описания и рассматривать только наружную область видимости (рис. 6.2). Она представляет собой усечённую пирамиду (англ. frustum), в которой расстояние от точки наблюдения (глаза наблюдателя) до ближней плоскости отсечения равно расстоянию от игольчатого отверстия до плёнки, а дальняя плоскость находится на очень большом расстоянии от точки наблюдения. Плёнка соответствует грани, отсечённой ближней плоскостью отсечения, при этом будет осуществляться перспективная проекция объектов, находящиеся в области видимости.

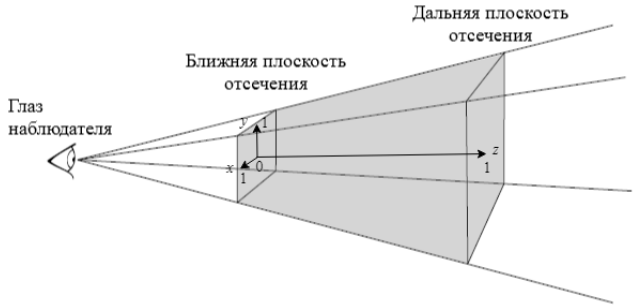



Рис. 6.2. Усечённая пирамида видимости

В данной усечённой пирамиде задаётся пространство экрана, ось z которого направлена вдоль направления взгляда и изменяется от 0 до 1 от ближней плоскости отсечения до дальней, а x и y оси лежат на ближней плоскости отсечения. Так как рендеринг ведётся в растровое изображение разрешением $w \times h$, также задаётся двумерное растровое пространство, координата $(0, 0)$ которого соответствует координате $(-1, -1, 0)$ экранного пространства, а координата $(w, h)$ соответствует координате $(1, 1, 0)$.

Горизонтальным углом обзора (англ. field of view, FOV) называется угол между верхней и нижней гранью пирамиды видимости.

Перевод координат точки $(x_{raster}, y_{raster})$ растрового пространства в экранное:

$$AspectRatio = \frac{w}{h}$$
$$x_{screen} = \left(2\frac{x_{raster}}{w} - 1\right) \cdot \tan\frac{fov}{2} \cdot AspectRatio$$
$$y_{screen} = \left(2\frac{y_{raster}}{h} - 1\right) \cdot \tan\frac{fov}{2}$$
$$z_{screen} = 0$$

### 6.1.1. Инициализация лучей

Направление луча:

$$Dir = \frac{x_{screen} \cdot CamRight + y_{screen} \cdot CamUp + CamFront}{\|x_{screen} \cdot CamRight + y_{screen} \cdot CamUp + CamFront\|}$$

где $x_{screen}$, $y_{screen}$ – координаты пикселя изображения в экранном пространстве, $CamUp$, $CamFront$ и $CamRight$ – векторы, описывающие перевод координат из мирового пространства в пространство камеры (задаются пользователем).

Начальная точка $O$ луча равна позиции камеры: $Origin = CamPos$.

## 6.2. Реалистичная камера

В отличие от идеальной камеры с игольчатым объективом, реальные камеры имеют объектив, содержащий в себе систему линз, которые фокусируют свет через апертуру на плоскость пленки. Линзы могут фокусироваться только на одной плоскости (фокальной плоскости), чем дальше объекты от неё находятся, тем они сильнее размыты на фотографии. Чем больше диафрагма, тем более выражен данный эффект, называющийся глубиной резко изображаемого пространства (англ. depth of field).

Для моделирования такой камеры мы можем использовать модель тонкой линзы. При использовании этого приближения, параллельные лучи, проходящие через линзу, фокусируются в точке, называемой фокальной точкой. Расстояние до неё называется фокусным расстоянием.

Изображение точки $-z$, находящейся с одной стороны от тонкой линзы с фокусным расстоянием $f$ (сама линза находится в центре координат), будет сфокусировано на плоскости в точке $z'$:

$$\frac{1}{z'} - \frac{1}{z} = \frac{1}{f}$$

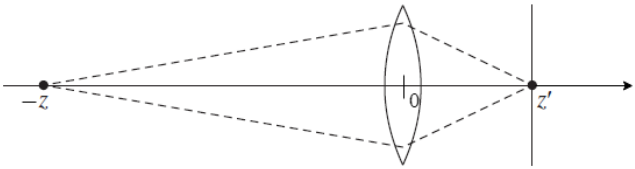



Рис. 6.3. Фокусирование изображения точки тонкой линзой

Точки, не лежащие на фокальной плоскости, будут рассеиваться на плёнке в виде пятна (англ. circle of confusion). Размер

пятна рассеяния зависит от диаметра апертуры, через которую проходит луч, фокусного расстояния, а также расстояния от линзы до объекта.

Рис. 6.4. Рендеринг с использованием реалистичной модели камеры

### 6.2.1. Инициализация лучей

Для инициализации луча нам потребуются начальная точка $O_{old}$ и направление $Dir_{old}$, которые были получены при моделировании камеры с игольчатым отверстием.

Новая начальная точка $O_{new}$ выбирается случайным образом на линзе, и трансформируется в мировые координаты:

$$P_{lens} = Aperture \cdot RandomSampleDisk()$$

$$O_{new} = CamPos + P_{lens\_x} \cdot CamRight + P_{lens\_y} \cdot CamUp$$

где $Aperture$ – размер апертуры, $RandomSampleDisk()$ – функция получения случайной точки на единичном диске.

Для нахождения нового направления $Dir_{new}$, задаём точку, в которой будут фокусироваться лучи, и задаём, чтобы он проходил через неё и точку на линзе:

$$P_{focus} = O_{new} + Dir_{old} \cdot FocalDistance$$

$$Dir_{new} = \frac{P_{focus} - O_{new}}{\left\| P_{focus} - O_{new} \right\|}$$

# ГЛАВА 7. ИНТЕГРИРОВАНИЕ МЕТОДОМ МОНТЕ-КАРЛО

Рассмотрим уравнение рендеринга:

$$L_o(p, \omega_o) = \int_{\mathcal{H}^2} f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i$$

Данное уравнение обычно не имеет аналитического решения, поэтому для его нахождения интеграла нам необходимо обратиться к численным методам. Хотя стандартные методы численного интегрирования с использованием квадратурных формул весьма эффективны при решении гладких интегралов с малой размерностью, их скорость сходимости значительно уменьшается при нахождении интегралов функций большой размерности, а также имеющих в себе разрывы, которые используются в рендеринге.

Интегрирование методом Монте-Карло позволяет решить данную проблему, так как его скорость сходимости не зависит от размерности подынтегрального выражения. Данный метод использует последовательность случайных чисел для оценки значения интеграла. Одно из важных свойств этого метода заключается в том, что требование к подынтегральному выражению $f(x)$ только одно: быть вычислимым в той области, на которой необходимо найти значение интеграла $\int f(x)\mathrm{d}x$. Побочным эффектом метода Монте-Карло является шум, который будет сходиться к истинному значению интеграла со скоростью $O\left(n^{-1/2}\right)$, где $n$ – количество итераций вычисления интеграла.

Пусть мы хотим вычислить одномерный интеграл $\int_a^b f(x)\mathrm{d}x$. Для случайной последовательности $X_i \in [a, b]$ с плотностью

распределения $p(x)$ математическое ожидание оценки по методу Монте-Карло

$$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$$

$E[F_N]$ в точности равно значению интеграла:

$$E[F_N] = E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] = \frac{1}{N}\sum_{i=1}^{N}\int_b^a \frac{f(x)}{p(x)}p(x)\mathrm{d}x = \frac{1}{N}\sum_{i=1}^{N}\int_b^a f(x)\mathrm{d}x$$
$$= \int_b^a f(x)\mathrm{d}x$$

Данный одномерный случай легко расширяется и на большую размерность. Пусть имеется тройной интеграл:

$$\int_{x_0}^{x_1}\int_{y_0}^{y_1}\int_{z_0}^{z_1} f(x,y,z)\mathrm{d}x\mathrm{d}y\mathrm{d}z$$

Тогда для его решения мы берём последовательность точек $X_i = (x_i, y_i, z_i)$ из параллелепипеда с размерами от $(x_0, y_0, z_0)$ до $(x_1, y_1, z_1)$, и, скажем, распределение их равномерно, то есть

$$p(x,y,z) = \frac{1}{(x_1 - x_0)}\frac{1}{(y_1 - y_0)}\frac{1}{(z_1 - z_0)}$$

тогда оценка интеграла методом Монте-Карло будет равна:

$$F_N = \frac{(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)}{N}\sum_{i=1}^{N} f(X_i)$$

## 7.1. Выборка значений функций отражения

Значение интеграла в уравнении рендеринга можно оценить с помощью метода Монте-Карло следующим образом:

$$L_o(p,\omega_o) = \int_{\mathcal{H}^2} f_r(p,\omega_o,\omega_i) L_i(p,\omega_i) \cos\theta_i \,\mathrm{d}\omega_i$$

$$\approx \frac{1}{N}\sum_{j=1}^{N} \frac{f_r(p,\omega_o,\omega_j) L_i(p,\omega_j) \cos\theta_j}{p(\omega_j)}$$

где $\omega_j$ – последовательность случайных направлений входного излучения, распределенных в соответствии с плотностью распределения $p(\omega)$

В главе 5 мы рассмотрели некоторые двунаправленные функции распределения отражательной способности. Из них модель зеркального отражения сейчас мы не будем рассматривать, так как BRDF этой модели представляет собой дельта-функцию, и значение интеграла мы можем найти точно без использования метода Монте-Карло. Для остальных моделей нам необходимо правильно выбрать случайную последовательность для быстрого схождения интеграла.

Идея использования выборки по значимости (англ. importance sampling) основывается на том, что некоторые значения случайной величины в процессе моделирования должны иметь большую значимость (вероятность) для оцениваемой функции, чем другие. Это способствует уменьшению дисперсии оценки интеграла. Если же распределение случайных значений выбрать неправильно, то это повлечёт за собой уменьшение скорости схождения оценки. Более значимыми направлениями $\omega_j$ для нас являются те, в которых BRDF $f_r(\omega_o,\omega_i)$ имеет большее значение.

BRDF диффузной модели освещения представляет собой константу, соответственно направления $\omega_j$ (в касательном пространстве) необходимо выбирать равномерно по всей полусфере $\mathcal{H}^2$:

$$\varphi = 2\pi\xi_1$$
$$\theta = \arccos\xi_2$$
$$\sin\theta = \sqrt{1-\cos\theta}$$
$$\omega_j = (\cos\varphi\sin\theta\,, \sin\varphi\sin\theta\,, \cos\theta)$$

где $\xi_1$ и $\xi_2$ – равномерно распределённые случайные вещественные числа от 0 до 1.

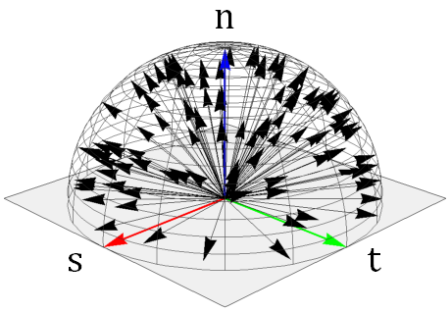



Рис. 7.1. Выборка случайных направлений при интегрировании уравнения рендеринга с использованием модели Ламберта

Плотность распределения равна: $p(\omega_j) = \frac{1}{2\pi}$

При использовании модели микрограней, мы сначала выбираем направление вектора полпути $\omega_h$, который соответствует нормалям самих микрограней, а затем применяем функцию отражения для получения вектора $\omega_j$:

$$\omega_j = \omega_o - 2(\omega_o \cdot \omega_h)\omega_h$$

Вектор $\omega_h$ необходимо выбрать в соответствии с функцией распределения нормалей микрограней $D(\omega_h)$. Для этого мы будем использовать метод обратного преобразования. Сначала необходимо убедиться, что функция распределения нормализована:

$$\int_{\mathcal{H}^2} D(\omega_h)(n \cdot \omega_h)\mathrm{d}\omega_h = 1$$

Нормализованная плотность распределения функции GGX в соответствии с телесным углом равна:

$$p_h(\omega_h) = \frac{\alpha^2(n \cdot \omega_h)}{\pi((n \cdot \omega_h)^2(\alpha^2 - 1) + 1)^2}$$

При переводе в сферические координаты получаем:

$$p_h(\theta, \varphi) = \frac{\alpha^2 \cos\theta \sin\theta}{\pi(\cos^2\theta\,(\alpha^2 - 1) + 1)^2}$$

После этого нам нужно получить функцию распределения $P_h(\theta)$. Для этого сначала вычисляем $p_h(\theta)$, затем интегрируем её:

$$p_h(\theta) = \int_0^{2\pi} p_h(\theta, \varphi)\mathrm{d}\varphi = \frac{2\alpha^2 \cos\theta \sin\theta}{(\cos^2\theta\,(\alpha^2 - 1) + 1)^2}$$

$$P_h(\theta) = \int_0^{\theta} \frac{2\alpha^2 \cos\theta' \sin\theta'}{(\cos^2\theta'\,(\alpha^2 - 1) + 1)^2}\mathrm{d}\theta'$$

$$= \frac{\alpha^2}{\cos^2\theta\,(\alpha^2 - 1)^2 + (\alpha^2 - 1)} - \frac{1}{\alpha^2 - 1}$$

Подставляя вместо $P_h(\theta)$ равномерно распределенное случайное вещественное число $\epsilon \in [0,1]$, выражаем зенитный угол $\theta$, распределённый в соответствии с заданной плотностью.

$$\theta = \arccos\sqrt{\frac{1 - \epsilon}{\epsilon(\alpha^2 - 1) + 1}}$$

Азимутальный угол, как и в предыдущем случае, распределён равномерно: $\varphi = 2\pi\xi$, так как $D(\omega_h)$ от него не зависит.

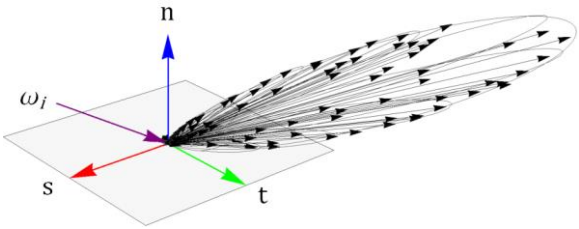



Рис. 7.2. Выборка случайных направлений при интегрировании уравнения рендеринга с использованием модели микрограней

Осталась одна деталь: мы знаем плотность распределения вектора полпути $p_h(\omega_h)$, однако интеграл в уравнении рендеринга берётся по отношении к вектору $\omega_j$, распределение $p(\omega_j)$ которого не совпадает с распределением $p_h(\omega_h)$, и нам необходимо найти его. Это нам поможет сделать иллюстрация на рис. 7.3.

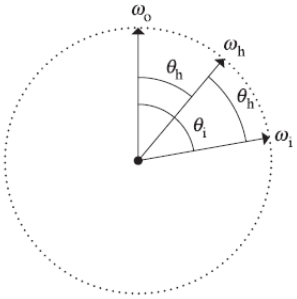



Рис. 7.3. Взаимное расположение направлений $\omega_o$, $\omega_j$ и $\omega_h$

Рассмотрим отношение дифференциальных телесных углов $\mathrm{d}\omega_h$ и $\mathrm{d}\omega_j$:

$$\frac{\mathrm{d}\omega_h}{\mathrm{d}\omega_j} = \frac{\sin\theta_h\, d\theta_h d\varphi_h}{\sin\theta_j\, d\theta_j d\varphi_j} = \frac{\sin\theta_h\, d\theta_h d\varphi_h}{2\sin 2\theta_h\, d\theta_h d\varphi_h} = \frac{\sin\theta_h}{4\cos\theta_h \sin\theta_h} = \frac{1}{4\cos\theta_h}$$

Отсюда плотность распределения $p(\omega_j)$:

$$p(\omega_j) = \frac{p_h(\omega_h)}{4(\omega_j \cdot \omega_h)}$$

(а) (б)

Рис. 7.4. Рендеринг без использования (а) и с использованием (б) выборки по значимости при одинаковом количестве итераций метода Монте-Карло

# ГЛАВА 8. ПЕРЕНОС ИЗЛУЧЕНИЯ

Обобщим всё вышесказанное в предыдущих главах. Снова вернёмся к уравнению рендеринга:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\mathcal{H}^2} f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i$$

Помимо интеграла отражения энергетической яркости, здесь добавилось слагаемое $L_e(p, \omega_o)$, которое обозначает излучённую энергетическую яркость в точке $p$ в направлении $\omega_o$. Если в точке $p$ находится источник света, то данное слагаемое будет ненулевым.

Поскольку мы предполагаем, что в воздухе у нас нет никакой участвующей среды, дополнительно рассеивающей свет (туман, дым и т.д.), энергетическая яркость не меняется вдоль лучей. Тогда, если мы зададим функцию бросания лучей $t(p, \omega)$, которая находит ближайшую точку $p'$ пересечения луча с начальной точкой $p$ и направлением $\omega$, мы можем записать, что выходное излучение в точке $p'$ равно входному излучению в точке $p$:

$$L_i(p,\omega) = L_o(t(p,\omega),-\omega)$$

В случае, если сцена не замкнута, мы можем сделать так, чтобы функция бросания лучей возвращала специальное значение $\Lambda$, такое, что $L_o(\Lambda,\omega)$ либо возвращала 0, либо делала выборку из текстуры окружения (англ. environment map). Тогда уравнение *переноса излучения* будет выглядеть следующим образом:

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{\mathcal{H}^2} f_r(p,\omega_o,\omega_i) L_o(t(p,\omega),-\omega)\cos\theta_i \,\mathrm{d}\omega_i$$

## 8.1. Поверхностная форма уравнения переноса излучения

Использование функции бросания лучей в уравнении переноса излучения вызывает некоторые трудности, так как она описывает неявное соотношение между геометрическими объектами в сцене. Для того, чтобы прояснить это соотношение, запишем уравнение переноса излучения в так называемой поверхностной форме [3, с. 865].

Сначала определим выходное излучение из точки $p'$ в точку $p$, если они взаимно не перекрыты, как:

$$L(p' \to p) = L(p',\omega)$$

где $\omega = \frac{p-p'}{\|p-p'\|}$

Также мы можем записать BRDF в точке $p'$, как

$$f(p'' \to p' \to p) = f(p',\omega_o,\omega_i)$$

где $\omega_i = \frac{p''-p'}{\|p''-p'\|}$ и $\omega_o = \frac{p-p'}{\|p-p'\|}$.

Кроме того, нам необходим множитель: якобиан перехода от телесного угла к площади для того, чтобы преобразовать уравнение переноса излучения от интеграла по направлению к интегралу по площади поверхности. Он будет равен $\cos\theta' / r^2$.

Объединим якобиан, $\cos\theta$ из уравнения рендеринга и бинарную функцию видимости $V$ ($V = 1$, если две точки не перекрыты, и $V = 0$, если перекрыты) в выражение

$$G(p \leftrightarrow p') = V(p \leftrightarrow p')\frac{\cos\theta\cos\theta'}{\|p - p'\|^2}$$

Получаем поверхностную форму уравнения переноса излучения:

$$L(p' \to p) = L_e(p' \to p) + \int_A f(p'' \to p' \to p)\, L(p'' \to p')G(p'' \leftrightarrow p')\mathrm{d}A(p'')$$

где $A$ – все поверхности в сцене.

## 8.2. Интегрирование по путям

Получив поверхностную форму уравнения переноса излучения, содержащего в себе рекурсию, мы можем определить его в качестве явного интеграла по путям.

Далее записаны несколько слагаемых, которые описывают излучение, попадающее в точку $p_0$ из точки $p_1$:

$$\begin{aligned} L(p_1 \to p_0) = {} & L_e(p_1 \to p_0) \\ & + \int_A L_e(p_2 \to p_1) f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1)\mathrm{d}A(p_2) \\ & + \int_A \int_A L_e(p_3 \to p_2) f(p_3 \to p_2 \to p_1) G(p_3 \leftrightarrow p_2) \\ & \times f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1)\mathrm{d}A(p_3)\mathrm{d}A(p_2) + \cdots \end{aligned}$$

На рис. 8.1 показан путь, состоящий из четырёх вершин, соответствующий третьему слагаемому в формуле. Он содержит в себе вершину $p_0$ – объектив камеры, две вершины $p_1$ и $p_2$ на поверхностях, отражающих свет, и $p_3$ на поверхности, излучающую свет.

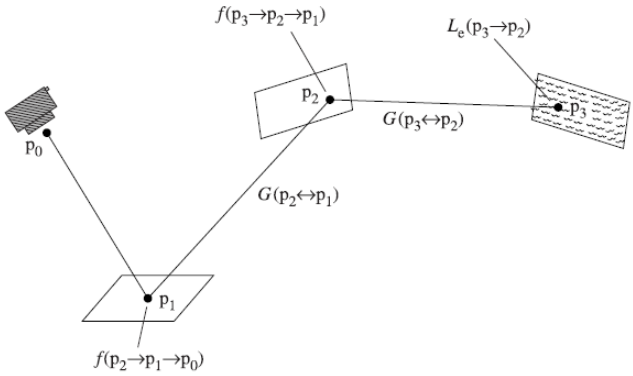



Рис. 8.1. Путь, состоящий из четырёх вершин

Бесконечная сумма слагаемых может быть записана компактно:

$$L(p_1 \to p_0) = \sum_{n=1}^{\infty} P(\bar{p}_n)$$

где $P(\bar{p}_n)$ описывает количество излучения, распространённого вдоль пути $\bar{p}_n$, состоящего из $n+1$ вершин $\bar{p}_n = p_0, p_1, \dots, p_n, p_0$ – вершина на плёнке камеры, либо линзе объектива, а $p_n$ – на источнике света, и

$$P(\bar{p}_n) = \int_A \int_A \dots \int_A L_e(p_n \to p_{n-1}) \times \left( \prod_{i=1}^{n-1} f(p_{i+1} \to p_i \to p_{i-1})\, G(p_{i+1} \leftrightarrow p_i) \right) \mathrm{d}A(p_2) \dots \mathrm{d}A(p_n)$$

В данном выражении присутствует множитель, который следует выделить отдельно:

$$T(\bar{p}_n) = \prod_{i=1}^{n-1} f(p_{i+1} \to p_i \to p_{i-1})\, G(p_{i+1} \leftrightarrow p_i)$$

Он называется *пропускной способностью* пути и описывает долю излучения, которая пришла от источника света до камеры через все вершины пути.

## 8.3. Алгоритм трассировки пути

Для уравнения переноса излучения в форме интегрирования по путям нам нужно оценить излучение $L(p_1 \to p_0)$ из точки пересечения луча $p_1$, брошенного из камеры, в точку $p_0$ самой камеры. С этим связано две проблемы [3, с. 872]:

1. Как оценить значение бесконечной суммы слагаемых $P(\bar{p}_i)$, используя конечное количество вычислений?
2. Каким образом генерировать путь $\bar{p}_i$ для слагаемого $P(\bar{p}_i)$?

Для решения первой проблемы можно использовать метод «русской рулетки». Мы можем сделать предположение, что для физически корректных сцен пути, содержащие большее количество вершин, имеют меньшую пропускную способность, чем пути с меньшим количеством вершин. Мы можем прекращать добавление очередного слагаемого к сумме с вероятностью $q_i$, и на некотором шаге суммирование завершится. Уравнение переноса излучения будет выглядеть следующим образом:

$$L(p_1 \to p_0) = \frac{1}{1-q_1}\left(P(\bar{p}_1) + \frac{1}{1-q_2}\left(P(\bar{p}_2) + \frac{1}{1-q_3}(P(\bar{p}_3) + \ldots\right.\right.$$

Решением второй проблемы является постепенное конструирование пути, начиная от вершины камеры $p_0$. В цикле по каждой вершине $p_i$ мы генерируем новое направление $\omega_i$ луча в соответствии с BRDF в точке $p_i$, как это было описано в главе 7, а затем используем функцию бросания лучей для нахождения новой вершины $p_{i+1} = t(p_i, \omega_i)$.

Алгоритм трассировки пути, делающий одну итерацию оценки интеграла в уравнении рендеринга методом Монте-Карло, выглядит следующим образом:

1. Сгенерировать луч из камеры в соответствии с тем, как это было описано в главе 6;
2. Энергетическая яркость $L = 0$, пропускная способность пути $T = 1$;
3. В цикле по вершинам пути $p_i, i > 0$:
   3.1. Найти точку пересечения $p_i$ луча со сценой, используя алгоритм пересечения луча с примитивом, как это было описано в главе 2 и ускоряющую структуру, о которых шла речь в главе 3;
   3.2. Добавить к $L$ энергию излучения источника света $L_e$:
      3.2.1. Если в точке $p_i$ находится источник света: $L = L + T \cdot L_e(p, \omega_o)$;
      3.2.2. Если пересечения не было найдено, делаем выборку из карты окружения: $L = L + T \cdot L_e(\omega_o)$;
   3.3. Остановить цикл, если пересечения не было найдено или было достигнуто значение максимальной глубины трассировки;
   3.4. Сгенерировать направление $\omega_i$ и сделать выборку значения из BRDF для обновления пропускной способности пути: $T = T \cdot \frac{f_r(p_i, \omega_o, \omega_i)}{p(\omega_i)}(\omega_i \cdot n)$;
   3.5. Прекратить вычисление суммы в соответствии с методом «русской рулетки» и вероятностью $q_i = \max(0.05, 1 - \|T\|)$;
4. Вернуть $L$.

Каждый раз после выполнения данного алгоритма мы получаем значение $L_j$, уточняя значение оценки интеграла уравнения рендеринга. При количестве итераций метода Монте-Карло, стремящемуся к бесконечности, мы получаем истинное значение интеграла: $L = \lim_{N \to \infty} \frac{1}{N} \sum_{j=1}^{N} L_j$

# ГЛАВА 9. ИСПОЛЬЗОВАНИЕ ПАРАЛЛЕЛИЗМА

Как было отмечено ранее, основными недостатками алгоритма трассировки пути является высокая алгоритмическая сложность и низкая скорость визуализации. Однако, так как метод трассировки каждый раз начинает процесс определения цвета каждого пикселя заново, данный алгоритм можно легко распараллелить, тем самым получив большой прирост производительности, особенно при использовании GPU. Проблема использования GPU заключается в том, что метод трассировки, в отличие от метода растеризации, не имеет аппаратной реализации в графическом процессоре, то есть нам необходимо реализовывать весь графический конвейер вручную.

Это возможно сделать, используя GPGPU (англ. General-purpose computing for graphics processing units, неспециализированные вычисления на графических процессорах) – технологию, позволяющую использовать графический процессор для выполнения расчётов в приложениях для общих вычислений, которые обычно проводятся на центральном процессоре.

Писать программы с использованием GPGPU можно с помощью OpenCL, фреймворка NVIDIA CUDA, вычислительных шейдеров, библиотеки C++ AMP и др.

В данной работе используется стандарт OpenCL, он состоит из API для координации параллельных вычислений в гетерогенной среде и собственного языка программирования. Его основное преимущество – кроссплатформенность, это означает, что один и тот же код, написанный на OpenCL, может работать на различных процессорах Intel, AMD и NVIDIA (более того, программа может запускаться даже без GPU). Другим преимуществом OpenCL является то, что он может использовать все доступные CPU и GPU в системе одновременно.

## 9.1. Краткое описание стандарта OpenCL

Сам OpenCL является довольно большим стандартом, который можно рассматривать с разных сторон. В стандарте [8] приводится так называемая иерархия моделей, описывающие основные идеи OpenCL:

- Модель платформы (англ. Platform Model);

Платформа OpenCL состоит из хоста, соединенного с устройствами (CPU, GPU, DSP или FPGA), поддерживающими OpenCL. Каждое OpenCL-устройство состоит из вычислительных блоков (англ. compute unit), которые далее разделяются на один или более элементы-обработчики (англ. processing elements, далее PE):

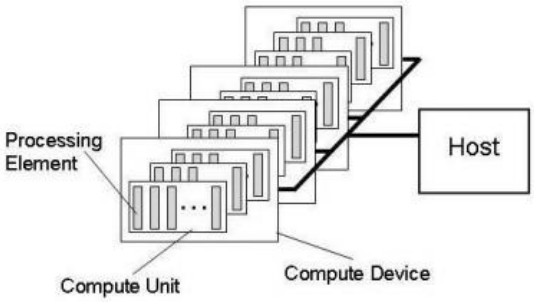



Рис. 9.1. Модель платформы OpenCL

OpenCL-приложение отправляет команды устройствам на выполнение вычислений на PE. PE в рамках вычислительного блока выполняют один поток команд как SIMD блоки (одна инструкция выполняется всеми одновременно, обработка следующей инструкции не начнется, пока все PE не завершат исполнение текущей инструкции), либо как SPMD-блоки (у каждого PE собственный счетчик инструкций).

- Модель исполнения (англ. Execution Model);

Выполнение OpenCL-программы состоит из двух частей: ядра (англ. kernel) – базовой единицы исполняемого кода, выполняющейся

на устройстве OpenCL, и хоста (англ. host) – программы на любом языке программирования, использующей OpenCL API.

Программист пишет код ядра на OpenCL C, варианте языка C99. Данный код может быть скомпилирован заранее, либо компилироваться каждый раз при запуске приложения.

Когда ядро ставится в очередь на исполнение, определяется пространство индексов (NDRange). Для каждой точки этого пространства выполняется копия (англ. instance) ядра с собственным id, которая называется рабочей единицей (англ. work-item). Каждый work-item выполняет один и тот же код, но конкретный путь исполнения (например, ветвления) и данные, с которыми он работает, могут быть различными.

Рабочие единицы организуются в рабочие группы (англ. work-groups). Группы предоставляют более крупное разбиение в пространстве индексов. Каждой группе приписывается свой id с такой же размерностью, которая использовалась для адресации отдельных рабочих единиц. Каждому элементу в группе сопоставляется локальный id. Таким образом, рабочие единицы могут быть адресованы как по глобальному id, так и по комбинации группового и локального id. Схематически это показано на рис. 9.2.

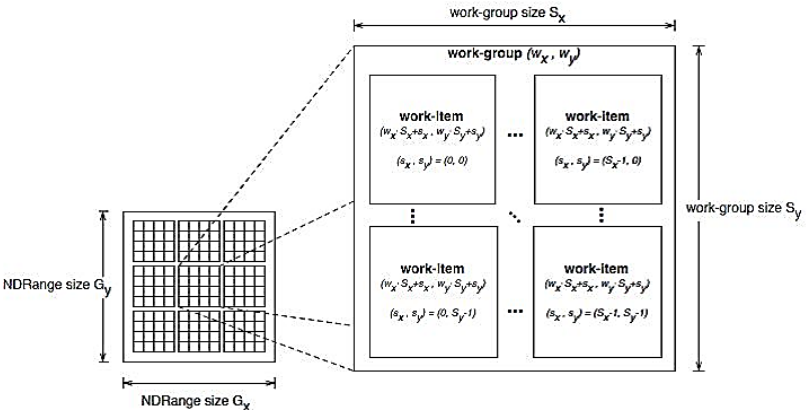



Рис. 9.2. Индексационное рабочее пространство OpenCL

Модель исполнения OpenCL приведена на рис. 9.3. Здесь используются следующие понятия:

- Контекст (англ. context) – интерфейс между хостом и устройством, который управляет всеми ресурсами OpenCL (программами, ядрами, командной очередью, буферами). Он получает и распределяет ядра и передает данные;
- Программа (англ. program) – вся программа, написанная на языке OpenCL (одно или несколько ядер и функции);
- Очередь команд (англ. command queue) – позволяет отправлять команды ядра на устройство (выполнение может быть в порядке или не по порядку);
- Объекты памяти (англ. memory objects) – данные (буферы и изображения), которые передаются от хоста к устройству и наоборот.

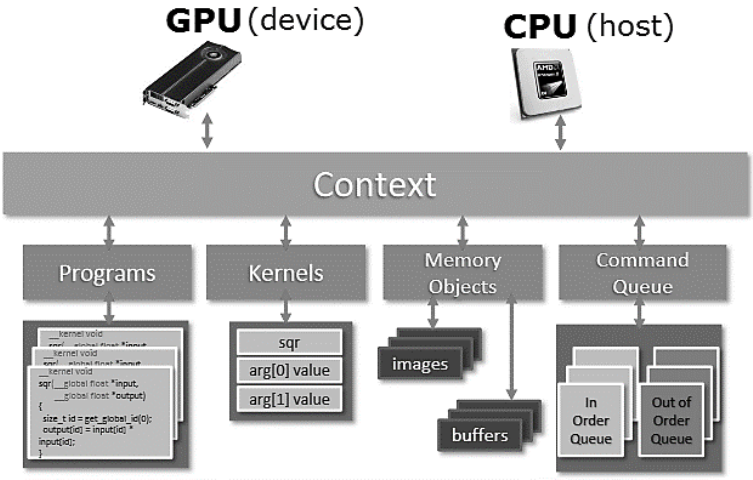



Рис. 9.3. Модель исполнения OpenCL

- Программная модель (англ. Programming Model);

OpenCL поддерживает две программные модели: параллелизм данных и параллелизм заданий, также поддерживаются гибридные модели. Основная модель, определяющая дизайн OpenCL – параллелизм данных.

- Модель памяти (англ. Memory Model);

В устройстве OpenCL имеется четыре уровня памяти, которые формируют иерархию памяти (от большой и медленной до малой и быстрой):

- Global Memory (аналог ОЗУ) – самая большая, но и самая медленная форма памяти, которая может быть прочитана и записана всеми рабочими элементами (потоками) и всеми рабочими группами на устройстве и также может быть прочитана/записана хостом;
- Constant Memory – небольшая часть глобальной памяти на устройстве, может быть прочитана всеми рабочими элементами на

устройстве (но не записана) и может быть прочитана/записана хостом. Постоянная память немного быстрее глобальной памяти;

- Local Memory (аналог кэш-памяти процессора) – память, совместно используемая рабочими элементами в одной рабочей группе. Локальная память позволяет обмениваться результатами с рабочими элементами, принадлежащими одной рабочей группе. Локальная память намного быстрее глобальной памяти (до 100x);
- Private Memory (аналог регистров процессора) – самый быстрый тип памяти. Каждый рабочий элемент (поток) имеет небольшое количество частной памяти для хранения промежуточных результатов, которые могут использоваться только этим рабочим элементом.

## 9.2. Инициализация и запуск OpenCL

Код ядра пишется в отдельном файле, назовем его kernel.cl. Точкой входа может быть любая функция, объявление которой предваряется ключевым словом `__kernel` [2].

```
__kernel void main(<список входных и выходных аргументов>)
{
    <instructions(get_global_id(0))>
}
```

`get_global_id(0)` – индекс рабочей единицы в пространстве NDRange, от которого зависит ход инструкций ядра.

Код хоста пишется с использованием OpenCL Runtime API, который представляет из себя API для C, но есть также и официальная C++ обертка, которая представляет из себя набор классов, соответствующих объектам OpenCL и поддерживающим подсчеты ссылок и бросание исключений. Код будем писать на C++.

Сначала получаем список доступных платформ:

```
std::vector<cl::Platform> all_platforms;
cl::Platform::get(&all_platforms);
```

Здесь платформа – конкретная реализация GPGPU от производителя (NVIDIA CUDA, AMD APP, Intel OpenCL).

Далее для каждой платформы получаем список доступных устройств:

```
        std::vector<cl::Device> platform_devices;
        platform.getDevices(CL_DEVICE_TYPE_ALL, &platform_devices);
```

Получаем контекст, через который будем посылать команды устройствам, загружаем программу из файла kernel.cl и собираем её. Возникшие ошибки при сборке программы будут отображены в консоли.

```
cl::Context context(platform_devices, 0, 0, 0, 0);
std::ifstream input_file("src/kernel.cl");
std::string curr_line;
std::string source;
source += buf.str() + "\n";
while (std::getline(input_file, curr_line)) {
    source += curr_line + "\n";
}
cl::Program program(context, source);
errCode = program.build(platform_devices);
if (errCode != CL_SUCCESS) {
    std::cerr << "Error building: " <<
program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(platform_devices[0])
    << " (" << errCode << ")" << std::endl;
}
```

Для собранной программы создаем объект kernel указывая, что точкой входа будет функция main() в файле kernel.cl. Также создаем очередь команд.

```
cl::Kernel kernel(program, "main", &errCode);
cl::CommandQueue queue(context, platform_devices[0], 0, 0);
```

Далее инициализируются буферы памяти, через которые будет осуществляться передача информации от ядра к хосту и наоборот. Например, инициализация буфера, соответствующего матрице

пикселей для вывода визуализированного изображения на экран, выглядит следующим образом:

```
cl::Buffer pixelBuffer(context, CL_MEM_READ_WRITE, global_work_size *
    sizeof(cl_float4), 0, &errCode);
```

Здесь `CL_MEM_READ_WRITE` – флаг, отвечающий за то, как может использоваться выделенная область памяти:

- `CL_MEM_READ_WRITE`, `CL_MEM_WRITE_ONLY`, `CL_MEM_READ_ONLY` – объект памяти может быть прочитан и записан ядром/только записан/только прочитан;
- `CL_MEM_USE_HOST_PTR` – объект будет использовать уже выделенную (и используемую программой) память хоста по указанному адресу;
- `CL_MEM_ALLOC_HOST_PTR` – память для объекта будет выделена из памяти хоста – т. е. из оперативной памяти;
- `CL_MEM_COPY_HOST_PTR` – при создании объекта будет произведён аналог memcpy с указанного адреса.

Значение `global_work_size` – соответствует количеству пикселей на экране (`width * height`) которое необходимо визуализировать.

В ядро можно передавать и отдельные параметры, с помощью метода `kernel.setArg(index, size, argPtr)`.

Остается в бесконечном цикле работы программы, каждая итерация которого соответствует рендерингу отдельного кадра, запустить ядро, добавив команду запуска ядра в очередь команд:

```
queue.enqueueNDRangeKernel(kernel, cl::NullRange,
    cl::NDRange(global_work_size));
queue.finish();
```

## 9.3. Трассировка пути

Трассировка пути осуществляется в функции `Render()`, находящейся в ядре. Она возвращает энергетическую яркость $L$ пикселя, соответствующего выпущенному лучу.

```
float3 Render(Ray* ray, const Scene* scene, unsigned int* seed,
__read_only image2d_t tex)
{
    float3 radiance = 0.0f;
    float3 beta = 1.0f;
```

Цикл конструирования пути по вершинам:

```
    for (int i = 0; i < kMaxBounce; ++i)
    {
```

Осуществляется поиск пересечения луча со сценой:

```
        IntersectData isect = Intersect(ray, scene);
```

Если пересечения не было найдено, то делаем выборку из текстуры окружения и прекращаем цикл:

```
        if (!isect.hit) {
            radiance += beta * max(SampleSky(tex, ray->dir), 0.0f);
            break;
        }
```

Добавляем к $L$ энергию излучения источника света:

```
        const __global Material* material = &scene-
>materials[isect.object->mtlIndex];
        radiance += beta * material->emission * 50.0f;
```

Генерируем направление $\omega_i$ и делаем выборку из BRDF, обновляем значение пропускной способности пути:

```
        float3 wi;
        float3 wo = -ray->dir;
        float pdf = 0.0f;
        float3 f = SampleBrdf(wo, &wi, &pdf, isect.texcoord,
isect.normal, material, seed);
        if (pdf <= 0.0f) break;
        beta *= f * dot(wi, isect.normal) / pdf;
        *ray = InitRay(isect.pos + wi * 0.01f, wi);
    }
```

По окончании цикла возвращаем энергетическую яркость:

```
    return max(radiance, 0.0f);
}
```

## 9.4. Трассировка в гетерогенной среде

OpenCL позволяет выполнять вычисления одновременно на различных устройствах. Соответственно, трассировку пути можно делать не только на GPU, но и подключить к данному процессу CPU, либо другие устройства, доступные в системе. Для этого мы создаем свой контекст OpenCL для каждой платформы, после чего в главном цикле приложения отправляем каждому устройству задачу: отрендерить горизонтальную полосу экрана небольшой ширины. Различные устройства справляются с данной задачей за разное время, в итоге получается, что более мощное устройство в процессе рендеринга кадра обработает большее количество элементов экрана, а менее мощное – меньшее количество. По окончании рендеринга всех элементов кадра, изображение выводится на экран. Данный процесс показан на рис. 9.4.

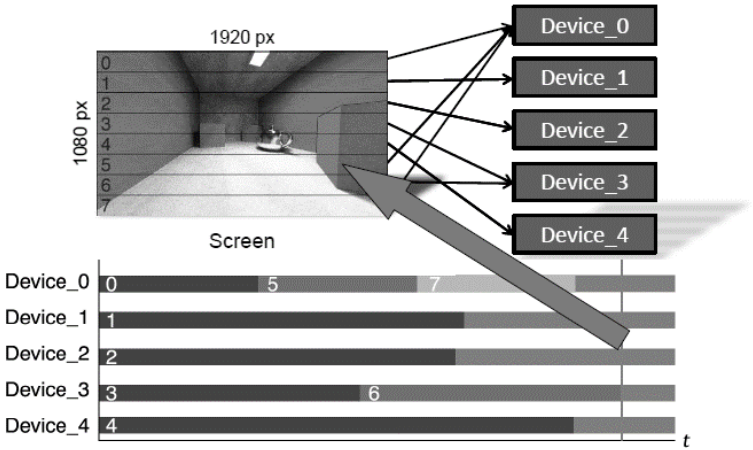



Рис. 9.4. Трассировка в гетерогенной среде

# ГЛАВА 10. РЕЗУЛЬТАТЫ

Результаты работы реализованного алгоритма трассировки пути изображены на рис. 10.1 и рис. 10.2.

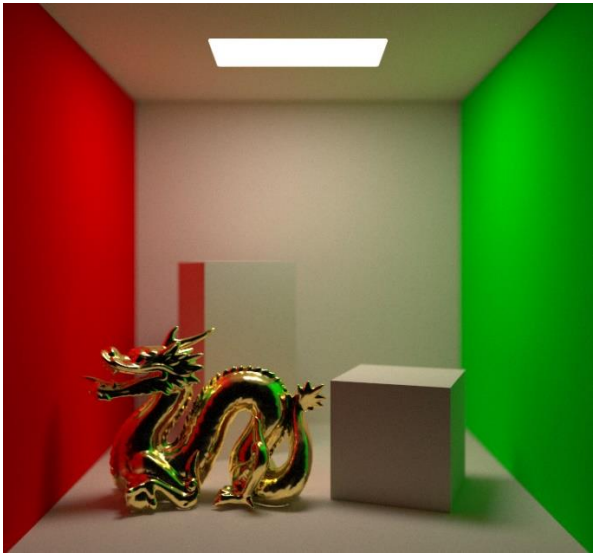


Рис. 10.1. Визуализация сцены Cornell Box с расположенной в ней моделью Стэнфордского дракона

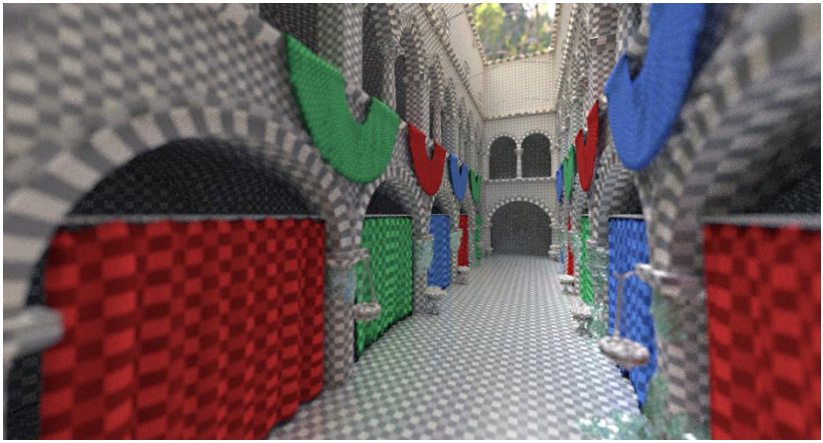


Рис. 10.2. Визуализация сцены Sponza Atrium

Сравнение алгоритмов поиска пересечения луча и треугольника без использования ускоряющей структуры и с использованием показывает, что при линейном поиске сложность алгоритма можно оценить, как $O(N)$, а при использовании ускоряющих структур сложность оценивается, как $O(\log N)$. Графии зависимости времени поиска пересечений лучей с примитивами при разрешении 1280x720 пикселей при использовании и без использования ускоряющих структур показаны на рис 10.3.

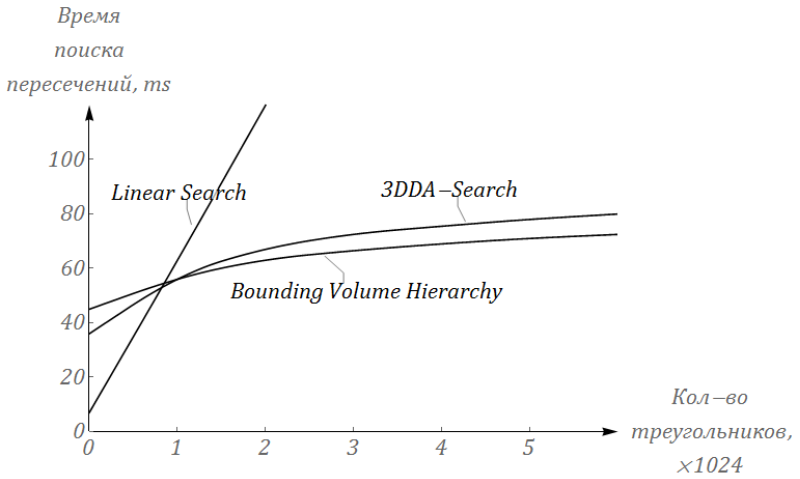



Рис. 10.3. Сравнение времени поиска пересечений без использования и с использованием ускоряющих структур

Сравнение соотношения нагрузки GPU и CPU при трассировке в гетерогенной среде показывает, что при небольшом количестве фрагментов кадра практически всю работу выполняет GPU, но, при увеличении количества фрагментов, данное соотношение выравнивается (рис. 10.4).

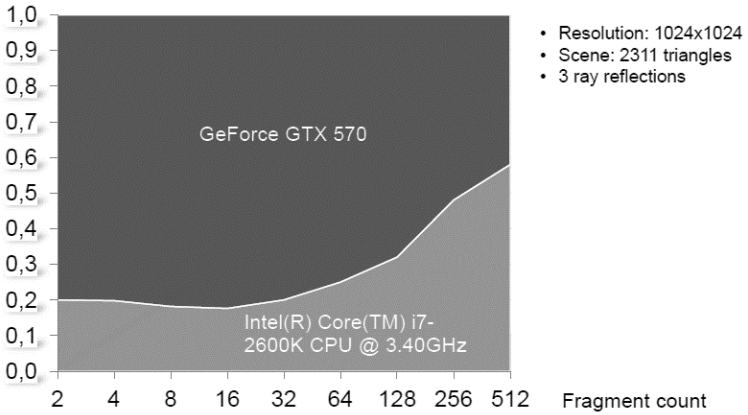



Рис. 10.4. Соотношение нагрузки GPU и CPU в зависимости от количества фрагментов кадра при использовании их одновременно в процессе рендеринга

Стоит аккуратно выбирать количество фрагментов кадра, так как большое их количество сильно замедляет процесс рендеринга: большая доля времени уходит на постановку задачи в очередь и на передачу данных от хоста к ядру и наоборот.

## 10.1. Сравнение с другими продуктами

Проведем сравнение реализованного продукта с известными аналогами. Поскольку все они основаны на алгоритме трассировки пути без допущений, основным показателем, по которому мы будем сравнивать, является время рендеринга нескольких кадров, возьмём их количество равным 64, а также оценка дисперсии шума, возникающего на изображении вследствие использования метода Монте-Карло. Сравнение будем проводить на сценах Cornell Box и, Sponza Atrium при разрешении 1280x720 пикселей, вычисления будут выполняться на графическом процессоре NVIDIA GeForce GTX 570.

Сравнение инструментов визуализации

| | Оцениваемый параметр | Sponza Atrium, 262144 треугольников | Cornell Box + Stanford Dragon, 48796 треугольников |
|---|---|---|---|
| NVIDIA Mental Ray | $t_{\text{ренд}}$, с | 46 | 21 |
| | $\sigma^2$ | 1.44e-04 | 2.87e-05 |
| V-Ray Adv | $t_{\text{ренд}}$, с | 61 | 33 |
| | $\sigma^2$ | 1.26e-04 | 2.62e-05 |
| Autodesk RayTracer | $t_{\text{ренд}}$, с | 39 | 15 |
| | $\sigma^2$ | 1.59e-04 | 5.32e-05 |
| Разработанный в работе продукт | $t_{\text{ренд}}$, с | 52 | 11 |
| | $\sigma^2$ | 2.21e-04 | 6.75e-05 |

Как можем видеть, реализованная нами библиотека рендеринга уступает некоторым аналогам в скорости визуализации сцен, содержащих большое количество объектов, однако, она работает быстрее при рендеринге небольших сцен. К сожалению, в разработанном нами продукте при одинаковом количестве кадров уровень дисперсии шума выше аналогов, это связано с тем, что другие продукты используют более сложные методы, такие, как двунаправленная трассировка и множественная выборка по значимости, дающие более быстрое схождение интеграла уравнения рендеринга при аналогичном количестве итераций метода Монте-Карло.

# ЗАКЛЮЧЕНИЕ

В ходе данной работы был реализована библиотека визуализации трёхмерных сцен без допущений, основанная на алгоритме трассировки пути. Он дает качественное и реалистичное изображение, однако в «чистом» виде он работает достаточно медленно. Поэтому были изучены и применены различные методы для оптимизации и параллелизации алгоритма.

В итоге мы имеем следующую функциональность:

- Использование метода Монте-Карло для оценки интеграла в уравнении рендеринга;
- Ускорение трассировки пути с использованием иерархии ограничивающих объемов или равномерного разбиения пространства;
- Реалистичная камера с эффектом глубины резко изображаемого пространства;
- Интегрирование с 3D-пакетами (3Ds Max, Blender и т.д.);
- Система материалов;
- Поддержка карт окружения;
- Поддержка различных аппаратных и программных платформ;
- Использование OpenCL для гетерогенных вычислений.

Существует множество путей развития данного программного продукта:

Поддержка текстур на данный момент ограничена, используются только карты окружения для моделирования небесного освещения. Повсеместное использование текстур и карт нормалей на объектах повысит реалистичность визуализируемой сцены.

На данный момент реализованы только модели отражения материалов. Помимо них, большой интерес представляют материалы, пропускающие свет, а также участвующие среды, дополнительно рассеивающие свет (туман, дым и т.д.).

Обратная трассировка пути работает достаточно быстро при рендеринге открытых пространств, куда поступает достаточно света от глобального источника освещения. С другой стороны, рендеринг закрытых пространств идет значительно медленнее, так как имеется небольшая вероятность, что луч попадет в локальный источник света. Ускорить сходимость интеграла в уравнении рендеринга позволяют алгоритмы двунаправленной трассировки пути.

Существенно уменьшить дисперсию в оценке интеграла освещения при использовании метода Монте-Карло позволяет алгоритм Metropolis Light Transport. Данный алгоритм динамически адаптируется и строит распределение случайной величины пропорционально любой, заранее неизвестной оцениваемой функции.

Использование иерархии ограничивающих объёмов значительно ускоряет трассировку пути, однако, построение самой иерархии представляет собой достаточно медленный процесс, который выполняется перед началом визуализации. Из-за этого мы не можем использовать анимированные объекты при рендеринге. Ускорить построение с помощью параллелизма позволяет структура HLBVH (англ. Hierarchical Linear Bounding Volume Hierarchy) и другие модификации.

Исходный код проекта находится в репозитории на Github:
https://github.com/AlexanderVeselov/RayTracing

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. AMD, The AMD OpenCL™ Zone, http://developer.amd.com/tools-and-sdks/opencl-zone/, 20.03.2016
2. Gamedev.net, Realtime raytracing with OpenCL I, https://www.gamedev.net/blog/1241/entry-2254170-realtime-raytracing-with-opencl-i/, 10.04.2017
3. Matt Pharr, Wenzel Jakob, Greg Humphreys. Physically Based Rendering: From Theory to Implementation, Third Edition: — Cambridge: Morgan Kauffman Publishers, 2016. — 1266 с.
4. NVIDIA Developer, NVIDIA OpenCL SDK Code Samples, https://developer.nvidia.com/opencl/, 20.03.2016
5. RayTracey's blog, OpenCL path tracing tutorial 1: Firing up OpenCL, http://raytracey.blogspot.ru/2016/11/opencl-path-tracing-tutorial-1-firing.html, 15.06.2017
6. Render.ru, Rendering: Теория – Компьютерная графика и анимация, http://render.ru/books/show_book.php?book_id=521, 14.05.2017
7. Scratchapixel 2.0, Ray Tracing: Rendering a Triangle (Ray-Triangle intersection: Geometric Solution), https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution, 14.05.2017
8. The Chronos Group Inc, The OpenCL Specification, https://www.khronos.org/opencl/, 2.10.2015
9. Tomas Akenine-Möller, Eric Haines, Naty Hoffman. Real-Time Rendering, Third Edition: — CRC Press, 2008. — 1045 с.
10. Быстрая трассировка лучей, http://ray-tracing.ru/, 15.11.2015

11. Игнатенко А. Трассировка лучей. Алгоритмы поиска пересечений, лекция 11, http://courses.graphicon.ru/files/courses/imagesynt/2010/lectures/mis_lect11_10.pdf, 23.04.2017
12. Хабрахабр, OpenCL. Подробности технологии, https://habrahabr.ru/post/72650/, 2.10.2015
13. RefractiveIndex.info, Refractive index database, https://refractiveindex.info/, 17.05.2018