

Институт Прикладной Математики имени М.В. Келдыша Российской  
Академии Наук

На правах рукописи

Фролов Владимир Александрович

# Методы решения проблемы глобальной освещенности на графических процессорах

05.13.11 – Математическое и программное обеспечение вычислительных  
машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ

на соискание ученой степени

кандидата физико-математических наук

Научный руководитель

д. ф.-м. н., проф.

Галактионов Владимир Александрович

Москва – 2013

# Содержание

<b>Введение</b> . . . . .	4
<b>Обзор литературы</b> . . . . .	15
<b>Глава 1. Введение в предметную область</b> . . . . .	16
1.1. Базовые понятия, модели, определения и сокращения . . . . .	16
1.2. Трассировка лучей . . . . .	19
1.3. Проблема глобальной освещенности . . . . .	53
1.4. Стохастическая трассировка лучей . . . . .	59
1.5. Прямая трассировка (Light Tracing) . . . . .	83
1.6. Двухнаправленная трассировка путей . . . . .	85
1.7. Перенос света Метрополиса (MLT) . . . . .	87
1.8. Фотонные карты . . . . .	90
1.9. Кэш Освещенности (Irradiance Cache) . . . . .	108
1.10. Выводы к первой главе . . . . .	115
<b>Глава 2. Анализ литературных источников</b> . . . . .	116
2.1. Алгоритмы на центральном процессоре . . . . .	116
2.2. Резюме по методам вычисления глобальной освещенности . . . . .	134
2.3. Краткое введение в программирование GPU . . . . .	135
2.4. Алгоритмы на массивно-параллельных процессорах . . . . .	152
2.5. Программные решения на центральных процессорах . . . . .	173
2.6. Программные решения на графических процессорах . . . . .	178
<b>Заключение</b> . . . . .	181
<b>Литература</b> . . . . .	182

Приложение А.	Название приложения . . . . .	199
---------------	-------------------------------	-----

# Введение

**Актуальность работы** Проблема глобальной освещенности - одна из основных проблем расчета освещения. Освещенность принято разделять на локальную и глобальную. Локальной освещенностью точки поверхности называют освещенность, вызванную прямым попаданием света от его источника. Глобальной освещенностью точки поверхности называют освещенность, вызванную как прямым так и непрямым попаданием света в данную точку (т.е. в результате одного или более переотражений). В настоящее время расчет глобальной освещенности на электронно-вычислительных машинах (ЭВМ) находит широкое применение. Выделим три основных группы сфер применения алгоритмов расчета глобальной освещенности.

Первую группу сфер применения алгоритмов вычисляющих глобальное освещение составляет синтез реалистичных изображений:

1. Оценка внешнего вида любых промышленных изделий, находящихся в проекте. Прежде всего это моделирование внешнего вида переносных электронных устройств (телефоны, ноутбуки, планшеты), персональных транспортных средств (автомобилей, мотоциклов и других транспортных средств, для которых важен внешний вид).
2. Архитектурный дизайн - моделирование экстерьеров, интерьеров помещений, салонов транспортных средств в целях оценки уровня освещенности и эстетичности соответствующего интерьера или экстерьера. Для архитектурных проектов особенно важно уметь демонстрировать как будет выглядеть спроектированное здание или интерьер дома в различных условиях освещения.
3. Создание рекламных и демонстрационных презентаций. В данной сфере применение можно условно разделить на демонстрацию решения,



находящегося в проекте и дополненную реальность - встраивание виртуального 3D объекта в видео-последовательность или изображения, снятые с реального мира.

4. Киноиндустрия, мультипликация и индустрия компьютерных игр. В последнем случае реалистичный расчет освещенности применяется особенно часто для создания карт освещенности.

Обозначенные выше сферы применения требуют высокого уровня реализма, позволяющего получить изображения виртуальной сцены, визуально неотличимое от фотографии такой же сцены в реальном мире. В связи с этим можно говорить о фотореалистичном синтезе изображений - синтезе изображений, не отличимых от фотографий. Синтез таких изображений был бы невозможен без вычисления глобальной освещенности.

Вторая группа сфер применения расчета глобальной освещенности связана с моделированием оптических эффектов для определенного ряда задач:

1. Расчет уровня освещенности приборов в кабине самолета, поезда или другого транспортного средства, а также офисных помещений в целях соблюдения государственных стандартов.
2. Моделирование осветительных приборов, панелей ЖК-дисплеев.
3. Расчет нежелательных слепящих или ухудшающих видимость эффектов, вызванных отражениями света от зеркальных поверхностей. Среди примеров таких эффектов можно отметить отражения светящихся элементов приборной панели в лобовом стекле автомобиля, отражения ярких источников света на экране монитора, отражения солнечного света от ярких поверхностей автомобилей и архитектурных сооружений.

Сферы применения из данной группы требуют высокоточного расчета освещения в виртуальной сцене и, как следствие, вычисления глобальной освещенности.

Третья группа сфер применения алгоритмов вычисления глобальной освещенности связана с расчетом переноса не-светового излучения. Особенно важна задача расчета переноса нейтронов в ядерных реакторах. Алгоритмы глобальной освещенности могут быть использованы для расчета переноса не-светового излучения, если поведение такого излучения для заданной модели в достаточной мере описывается законами геометрической оптики.



Таблица 1. Примеры применения разработанной системы.

В настоящий момент все известные промышленные системы, использующие графические процессоры для расчета глобальной освещенности и реалистичного синтеза изображений основаны на различных вариациях метода Монте-Карло, дающих несмещенную оценку решения на всем множестве

точек изображения. В соответствии с известным определением, несмещенной оценкой будем называть такую оценку, которая при стремлении числа Монте-Карло выборок к бесконечности сходится к точному решению. Если оценка не обладает этим свойством, будем называть ее смещенной. Для методов с несмещенной оценкой характерна точность в пределе, однако, они обладают медленной сходимостью по своей природе, поскольку такие методы вычисляют искомое значение цвета методом Монте-Карло независимо для каждого пиксела изображения. В связи с этим, существующие системы реалистичной визуализации и расчета освещенности, использующие центральные процессоры, но более интеллектуальные методы расчета во многих случаях имеют сравнимое или даже меньшее время построения изображения. Как следствие - системы расчета освещения на графических процессорах получили ограниченное распространение.

Такие алгоритмы как кэш освещенности и фотонные карты дают смещенную оценку (то есть не обладают точностью в пределе), но позволяют получить приемлемое изображение за гораздо меньшее время, поскольку они в значительной мере переиспользуют результаты вычислений. Однако их реализация на графических процессорах затруднена по ряду причин:

1. Переиспользование результатов вычислений уменьшает общий объем вычислений, однако, приводит к необходимости явной синхронизации и необходимости построения ускоряющих структур на графических процессорах.
2. Методы реалистичной визуализации со смещенной оценкой решения требуют особые способы контроля точности. Существующие подходы на основе смещенных методов или не Монте-Карло (не основанных на Монте-Карло трассировке лучей) алгоритмов на графических процессорах не удовлетворяют необходимым критериям точности и направлены

на интерактивный синтез изображений для компьютерных игр.

В связи с этим, в настоящий момент не существует систем расчета освещенности, способных эффективно использовать методы со смещенной оценкой для расчета освещенности с высокой точностью и синтеза реалистичных изображений. Однако выигрыш от таких алгоритмов на центральном процессоре может быть весьма значительным (до 10 раз). По этой причине в настоящей работе было принято решение разработать систему расчета освещенности на графических процессорах на основе алгоритмов со смещенной оценкой, но обладающих высокой точностью. Достаточно высокой точностью будем считать такую точность, при которой человеческий глаз не заметит разницы между изображением, построенным эталонным методом с предельно-большим временем расчета (методом Монте-Карло трассировки лучей) и изображением, построенным исследуемым методом. Предельно-большим временем расчета будем называть такое время, при котором эталонный метод Монте-Карло не оставляет на изображении видимого для человеческого глаза шума (или оставляет незначительный шум, сливающийся с деталями изображения).

**Цель диссертационной работы** состоит в разработке эффективных методов решения проблемы глобальной освещенности на графических процессорах на основе алгоритмов, дающих смещенную оценку решения и разработке системы расчета освещенности, реализующей эти методы. Под эффективностью метода понимается следующие два требования:

1. Метод, реализуемый на графическом процессоре должен требовать сравнимый объем вычислений и памяти по отношению к его однопоточному аналогу.
2. Производительность метода должна линейно масштабироваться с уве-

личением числа ядер и вычислительной мощности графического процессора.

Необходимым требованием к решению в целом является обеспечение контроля точности и возможность достижения такой точности, при которой человеческий глаз не заметит разницы между изображением, построенным эталонным методом с предельно-большим временем расчета (методом Монте-Карло трассировки лучей) и изображением, построенным на основе разработанных методов.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. Исследовать существующие алгоритмы и методы вычисления глобальной освещенности и синтеза реалистичных изображений на центральных и графических процессорах.
2. Выявить основные проблемы, возникающие при адаптации наиболее эффективных алгоритмов вычисления глобальной освещенности со смещенной оценкой решения к архитектуре графических процессоров и предложить методы их решения.
  - а. Разработать алгоритм распределения работы для метода Монте-Карло на графических процессорах в применении к задаче вычисления глобальной освещенности на множестве пикселей изображения.
  - б. Разработать параллельную реализацию алгоритма кэширования освещенности на графических процессорах, позволяющую значительно снизить время построения изображения при сравнимом с эталонным методом Монте-Карло качестве.

- в. Разработать метод построения ускоряющей структуры на графических процессорах, позволяющий эффективно реализовать алгоритм фотонных карт.
- 3. Создать систему расчета освещенности на графических процессорах с использованием технологии CUDA.
- 4. Интегрировать разработанную систему в среду трехмерного моделирования 3d Studio Max.

### **Научная новизна**

Получены следующие новые результаты в области расчета глобальной освещенности на графических процессорах.

1. Предложен алгоритм распределения работы для эффективной реализации метода Монте-Карло на графических процессорах в применении к задаче вычисления глобальной освещенности и фото-реалистичного синтеза изображений. Алгоритм позволяет эффективно распределить вычислительные ресурсы графического процессора на множестве пикселей изображения с неравномерной сложностью расчета. В отличие от появившихся позднее аналогов, алгоритм позволяет сохранять пространственную близость групп лучей, что повышает эффективность использования ресурсов графического процессора за счет снижения числа расходящихся по разным веткам групп потоков.
2. Предложена параллельная реализация алгоритма кэширования освещенности на графических процессорах, позволяющая снизить время построения изображения в 10 раз по сравнению с традиционным методом Монте-Карло при сравнимом качестве получаемого изображения.

В отличие от существующих параллельных реализаций кэша освещенности или реализаций, частично использующих графические процессоры, в предложенном подходе впервые решена проблема одновременной вставки в кэш большого числа не-дублирующих друг друга записей. За счет этого достигнута масштабируемость алгоритма и возможность легкой интеграции в существующие системы расчета освещения на графических процессорах.

3. Разработан метод построения ускоряющей структуры на графических процессорах, позволяющий от 2 до 5 раз ускорить сбор освещенности в алгоритме фотонных карт по сравнению с существующими методами. Разработанный метод впервые позволил строить окто-деревья со множественными ссылками над множеством объектов ненулевого размера полностью параллельно на графических процессорах. Метод прост в реализации и использует только 2 примитива параллельного программирования - параллельное добавление элементов в буфер и параллельную сортировку. В отличие от других способов построения фотонной карты, предложенный метод строит ускоряющую структуру, позволяющую выполнить процесс сбора освещенности в заданной точке со всех фотонов в единственном цикле сбора, не содержащем ветвлений.

### **Практическая значимость**

Разработанный программный комплекс может быть использован для расчета освещения и реалистичной визуализации при работе в среде моделирования 3D Studio Max. Основные сферы применения:

1. Оценка внешнего вида и реалистичная визуализация любых проектов промышленных изделий.
2. Архитектурный дизайн - моделирование экстерьеров, интерьеров поме-

щений, салонов транспортных средств для оценки уровня освещенности и эстетичности соответствующего интерьера или экстерьера.

3. Создание рекламных и демонстрационных презентаций.
4. Расчет нежелательных оптических эффектов при проектировании архитектурных и инженерных сооружений.

**На защиту выносятся следующие основные результаты и положения:**

1. Алгоритм распределения работы для эффективной реализации метода Монте-Карло на графических процессорах в применении к задаче вычисления глобальной освещенности и фото-реалистичного синтеза изображений.
2. Параллельная реализация алгоритма кэширования освещенности на графических процессорах, позволяющая снизить время построения изображения в 10 раз по сравнению с традиционным методом Монте-Карло при сравнимом качестве получаемого изображения.
3. Метод построения окто-дерева со множественными ссылками на графических процессорах, способный работать с объектами ненулевого размера и от 2 до 5 раз ускоряющий сбор освещенности в методе фотонных карт по сравнению с аналогами.
4. Система расчета освещенности на графических процессорах, значительно превосходящая по скорости существующие аналоги.

**Апробация работы** Основные результаты диссертации докладывались на следующих конференциях:



1. Международная Конференции по Компьютерной Графике и Зрению GraphiCon'2009 (Москва, факультет ВМК МГУ).
2. Международная Конференции по Компьютерной Графике и Зрению GraphiCon'2010 (Санкт-Петербургский государственный университет информационных технологий, механики и оптики).
3. Международная Конференции по Компьютерной Графике и Зрению GraphiCon'2012 (Москва, факультет ВМК МГУ).
4. Международная Конференции по Компьютерной Графике и Зрению GraphiCon'2013 (Институт автоматизации и процессов управления ДВО РАН, Дальневосточный Федеральный Университет).
5. Семинар по компьютерной графике и машинному зрению Ю.М. Баяковского (Москва, факультет ВМК МГУ).
6. Семинар направления «Программирование» им. М. Р. Шура-Бура в ИПМ им. М. В. Келдыша.
7. Тринадцатый научно-практический семинар новые информационные технологии в автоматизированных системах. Московский государственный институт электроники и математики.

**Публикации.** Материалы диссертации опубликованы в 8 печатных работах, из них 2 статей в рецензируемых журналах [1, 2] 4 статьи в сборниках трудов конференций, 1 в учебном пособии и 1 тезис доклада.

**Личный вклад автора** Содержание диссертации и основные положения, выносимые на защиту, отражают персональный вклад автора в опубликованные работы. Подготовка к публикации полученных результатов проводилась совместно с соавторами, причем вклад диссертанта был определяю-

щим. Все представленные в диссертации результаты получены лично автором.

**Структура и объем диссертации** Диссертация состоит из введения, обзора литературы, 4 глав, заключения и библиографии. Общий объем диссертации  $P$  страниц, из них  $p_1$  страницы текста, включая  $f$  рисунков. Библиография включает  $B$  наименований на  $p_2$  страницах.

## Обзор литературы

# Глава 1

## Введение в предметную область

Первая глава представляет из себя введение в предметную область и содержит основные определения и термины. В главе проводится обзор базовых алгоритмов решения проблемы глобальной освещенности. Первая глава также может рассматриваться как руководство к созданию программного обеспечения для решения проблемы глобальной освещенности на центральном процессоре.

### 1.1. Базовые понятия, модели, определения и сокращения

В целях улучшения восприятия материала, ниже собраны воедино базовые понятия, определения и сокращения, наиболее часто используемые в работе.

1. Локальным освещением называют освещение вызванное прямым попаданием света от источника на поверхность.
2. Глобальным освещением называют освещение вызванное прямым и непрямым попаданием света на поверхность.
3. Локальная модель освещения - модель освещения точки поверхности, предусматривающая только локальное освещения.
4. Мировым пространством называют трехмерное пространство, в котором задана геометрия виртуальной сцены.

5. Пространством камеры называют трехмерное пространство, в котором виртуальная камера находится в точке  $(0,0,0)$  и направлена в положительную сторону по оси  $Z$   $(0,0,1)$ .
6. Усеченным экранным пространством (или clip space) называют пространство внутри трехмерного единичного куба (координаты всех точек пространства лежат в интервале  $[-1,1]$ ). При проекции на экранную плоскость точки с координатами  $(-1,-1,z)$  переходят в левый нижний угол экрана, а точки с координатами  $(1,1,z)$  - в правый верхний.
7. Перспективным преобразованием называют процесс перевода точки из пространства камеры в усеченное экранное пространство.
8. Модель Ламберта - модель, предполагающая рассеивание света, падающего на поверхность равномерно во все стороны.
9. Диффузные поверхности (или Ламбертовские поверхности) - поверхности, отражающие свет преимущественно по модели Ламберта.
10. Каустик - эффект, образуемый отраженным или преломленным светом на диффузных поверхностях. Типичный пример каустика – солнечный зайчик от стакана воды, когда через него просвечивает солнце.
11. ДФО (или BRDF) - Двухнаправленная Функция Отражения (Bidirectional reflectance distribution function).
12. Рендер-система - Программная или программно-аппаратная система расчета изображений.
13. Несмещенная рендер-система - Рендер-система, позволяющая получать несмещенную оценку интеграла освещенности, используя метод Монте-Карло.

14. Монте-Карло трассировка путей (или обратная трассировка путей) - базовый алгоритм оценки интеграла освещенности в точке, реализующий метод Монте-Карло.
15. Сэмпл - выборка в методе Монте-Карло.
16. Сэмплирование - процесс генерации выборок для оценки интеграла при помощи метода Монте-Карло.
17. AABB - Axis Aligned Binding Box,- выровненный по осям координат прямоугольный параллелипипед.
18. SAH - Surface Area Heuristic, - функция оценки стоимости трассировки некоторого объекта, ограниченного при помощи AABB или другой фигуры лучей.
19. CPU - Cenral Processing Unit, центральный процессор.
20. GPU - Graphics Processing Unit, графический процессор.
21. warp - группа потоков, имеющих один общий счетчик команд.
22. Дивергентные потоки - потоки из группы warp, разошедшиеся по различным участкам кода вследствие выполнения операции ветвления.
23. API - Application Programming Interface, - программный интерфейс.

В соответствии с введенными выше определениями, освещение принято разделять на локальное и глобальное. Непрямое попадание света (которое необходимо вычислять для решения проблемы глобальной освещенности), вызвано переотражениями световой энергии от различных объектов трехмерной сцены и является трудоемким в плане вычислений. Далее будут рассмотрены базовые алгоритмы, позволяющие учесть переотражения света от находящихся в виртуальной сцене объектов.

## 1.2. Трассировка лучей

Алгоритм трассировки лучей был впервые продемонстрирован Turner Whitted-ом в 1980-ом году [3]. Этот алгоритм часто называют обратной трассировкой лучей или "Whitted-style" трассировкой. Он позволяет получить такие эффекты как тени, отражения и преломления.

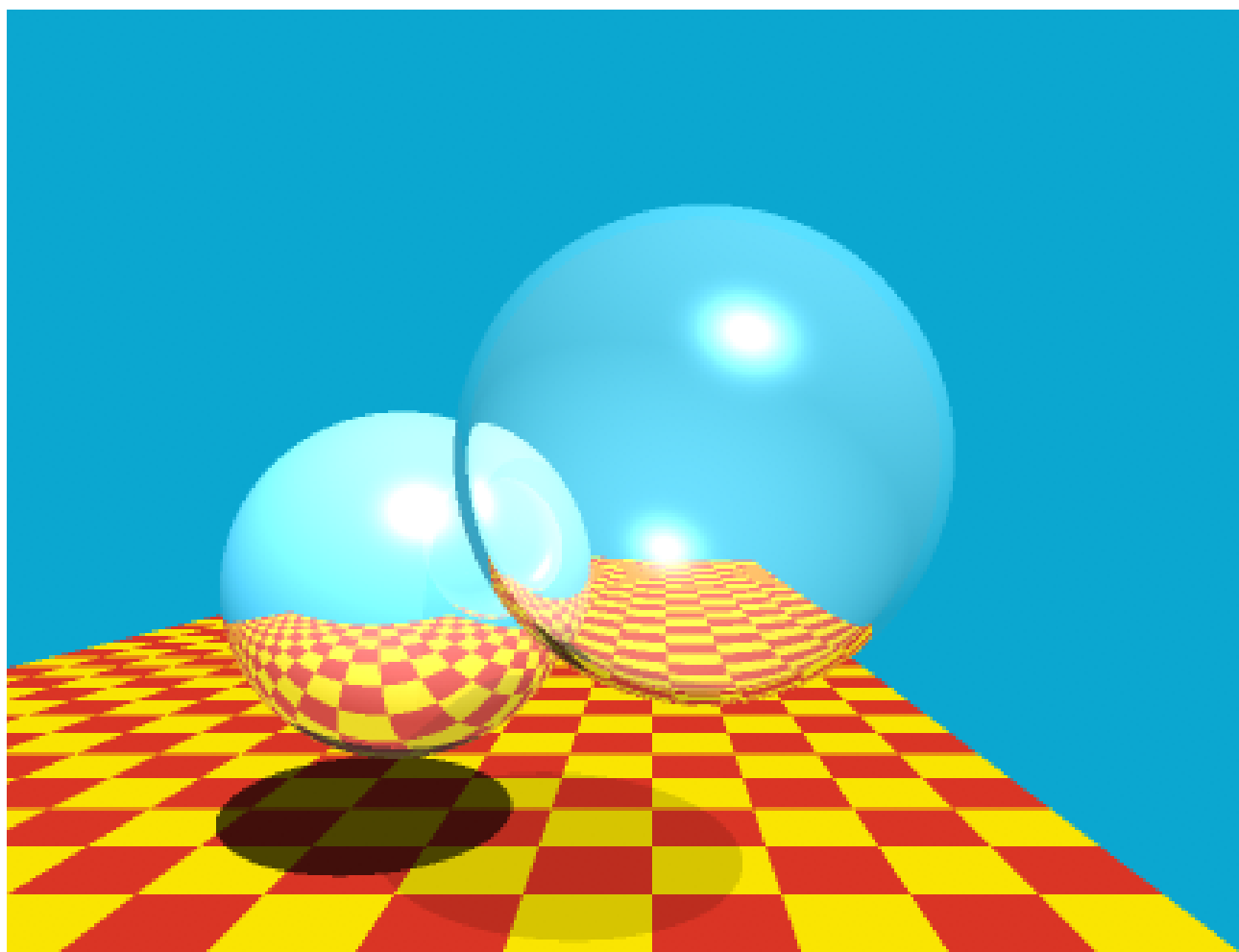
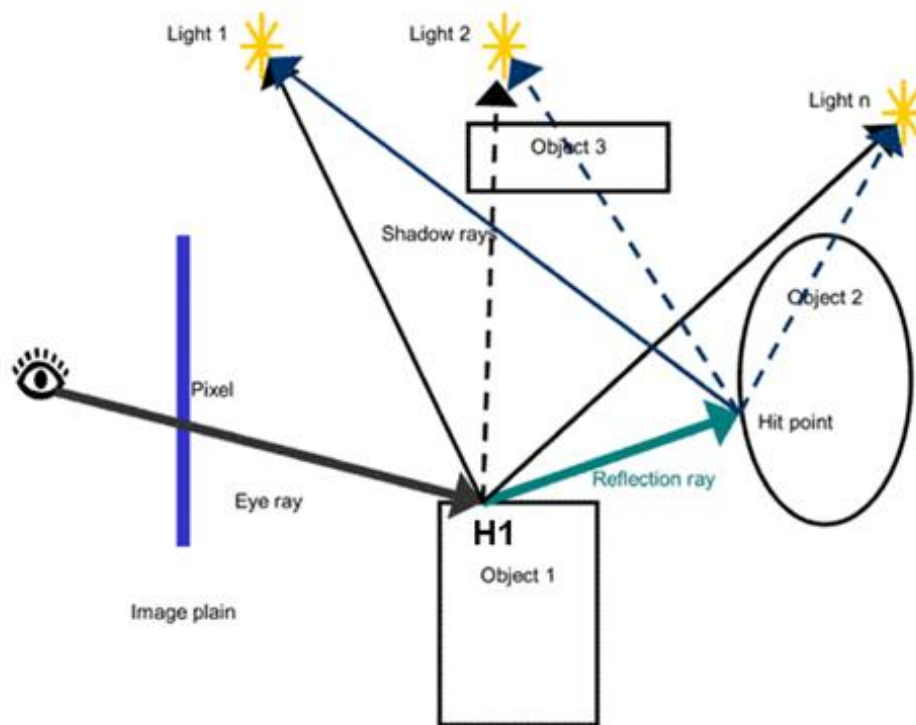


Рис. 1.1. Whitted-style трассировка лучей.

Обратная трассировка лучей выглядит следующим образом: из виртуального глаза через каждый пиксел изображения испускается луч и находится точка его пересечения с поверхностью сцены (для упрощения изложения объемные эффекты вроде тумана не рассматриваются). Лучи, выпущенные из глаза называют первичными. Допустим, первичный луч пересекает некий объект в

точке H1 (рис. 1.2).



2

Рис. 1.2. Схема обратной трассировки лучей.

Для расчета тени необходимо определить для каждого источника освещения, видна ли из него эта точка. Предположим пока, что все источники света точечные (т.е. имеют пренебрежимо малый размер). Тогда для каждого точечного источника света до него испускается теневой луч из точки H1 и проверяется, пересекает ли луч какие-то объекты на своем пути. Если теневой луч находит пересечение с другими объектами, расположенными ближе чем источник света, значит, точка H1 находится в тени от этого источника и освещать ее не надо. Иначе, считаем освещение по некоторой локальной модели (Модель Фонга, Блина-Фонга, Кука-Торранса и т.д.).

При учете вклада освещения от точечного источника важно делить интенсивность источника света на квадрат расстояния до него. Этот прием позволяет корректно учитывать вклад источников и его объяснение будет дано позже, при рассмотрении алгоритма трассировки путей.



Освещение со всех видимых (из точки Н1) источников света просто складывается. Далее, если материал объекта 1 имеет отражающие свойства, из точки Н1 испускается отраженный луч и для него вся процедура трассировки рекурсивно повторяется. Аналогичные действия должны быть выполнены, если материал имеет преломляющие свойства.

В трассировке лучей для представления самого луча, как правило, используется параметрическое уравнения прямой 1.1.

$$\overline{X(t)} = \overline{O} + \overline{D} * t \quad (1.1)$$

В выражении 1.1  $\overline{X(t)}$  обозначает некоторую точку на прямой.  $\overline{O}$  - точка начала (сокращение от Origin) луча, или точка вылета.  $\overline{D}$  - нормализованный (имеющий единичную длину) вектор направления луча (сокращение от Direction).  $t$  - скалярная величина, большая или равная нулю.

Таким образом для задания луча достаточно хранить 2 вектора -  $\overline{O}$  и  $\overline{D}$ . Задания значения параметра  $t$  позволит получить любую точку лежащую на этом луче.

Важным моментом при вычислении отраженного и преломленного лучей является необходимость добавлять небольшое смещение к позиции луча. В случае отраженного луча нужно прибавлять смещение  $\varepsilon * \overline{n}$ , в случае преломленного -  $(-1) * \text{sign}(\text{dot}(\overline{n}, \overline{D})) * \varepsilon * \overline{n}$ . Это делается для того, чтобы отраженный (или преломленный) луч не встретил пересечение на следующем шаге трассировки с той же самой поверхностью. В выражениях использованы следующие обозначения:

1.  $\overline{n}$  - вектор нормали к поверхности.
2.  $\overline{D}$  - вектор направления луча.
3.  $\varepsilon$  - некоторая малая величина, близкая к пределу машинной точности

(но больше него) при вычислении точки пересечения луча и поверхности.

4. *sign* - функция, возвращающая 1 если ее аргумент больше или равен 0 и -1 в противном случае.
5. *dot* - скалярное произведение двух векторов.

### 1.2.1. Генерация луча

Как уже было сказано, на начальном этапе алгоритма трассировки лучей через каждый пиксел экрана необходимо выпустить луч. Это можно сделать руководствуясь несложными геометрическими соображениями, рассчитав направления лучей в пространстве камеры (выражения 1.2). Пространство камеры - это такое пространство, где камера находится в точке (0,0,0) и смотрит в положительном направлении оси z.

$$\begin{aligned}
 fov &:= \frac{\pi}{2} \\
 D.x &:= x + 0.5 - w/2 \\
 D.y &:= y + 0.5 - h/2 \\
 D.z &:= -w/\tan(\frac{fov}{2}) \\
 D &:= \text{normalize}(D)
 \end{aligned} \tag{1.2}$$

Где:

1.  $\bar{x}$  - горизонтальная координата пиксела в пространстве изображения.
2.  $\bar{y}$  - вертикальная координата пиксела в пространстве изображения.
3.  $\bar{w}$  - размер изображения по горизонтали.

4.  $\bar{h}$  - размер изображения по вертикали.
5.  $fov$  - Угол обзора камеры. Обычно равен  $\frac{\pi}{2}$ .
6. *normalize* - функция, осуществляющая нормализацию вектора (приведение к вектору с единичной длиной).

Что касается начала луча  $\bar{O}$ , то в пространстве камеры это точка  $(0,0,0)$ . Данный подход моделирует так называемую камеру-обскура. Однако он обладает одним недостатком. При создании системы визуализации, в которой сцена может отрисовываться как трассировкой лучей, так и растеризацией (например при помощи библиотеки OpenGL), могут возникнуть трудности с получением геометрически-совпадающей картинки для растеризатора и трассировщика лучей. Причина этого заключается в том, что во всех современных системах, основанных на растеризации (и не только), используется понятия видовой матрицы (*mView*) и матрицы проекции (*mProj*). Видовая матрица переводит мир из мирового пространства в пространство камеры (т.е. преобразовывает пространство так что, камера перемещается в  $(0,0,0)$  и смотрит в положительном направлении оси  $z$ ). Эта матрица не вызывает осложнений, т.к. всегда можно преобразовать луч при помощи обратной матрицы, чтобы эмитировать перемещение камеры. Однако матрицу проекции необходимо пересчитать. Следующая формула используется для вычисления матрицы перспективной проекции в OpenGL [4]:

$$mProj = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Где:

- $e = 1/\tan(\text{FOV}/2)$
- FOV - Field Of View; угол обзора. Обычно равен  $\frac{\pi}{2}$ .
- $a$  - aspect ratio;  $a = h/w$ ;
- $f$  - far clip plane; дальняя плоскость отсечения.
- $n$  - near clip plane; ближняя плоскость отсечения.

В случае камеры-обскуры,  $f$  стремится к бесконечности, а  $n$  - к нулю. Таким образом, получаем:

$$mProj = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

В целях обеспечения совместимости с существующими приложениями, где матрица проекции уже задана (или задается извне и менять нельзя), необходимо использовать другой алгоритм генерации луча (выражение 1.3):

$$P1.x := 2 * (x + 0.5)/w - 1$$

$$P1.y := -2 * (y + 0.5)/h + 1$$

$$P1.z := 0$$

$$P1.w := 1$$

$$P2 := (mView * mProj)^{-1} * P1$$

$$P2 := P2 * (1/P2.w)$$

$$P2.y := -P2.y$$

$$D := \text{normalize}(P2)$$

(1.3)

В выражение 1.3 используется распространенная в трехмерной компьютерной графике однородная система координат [4], где вводится четвертая координата  $w$  в целях расширения множества преобразований, которые можно записать при помощи матриц. В соответствии с этим, матрицы  $mView$  и  $mProj$  имеют размер  $4 \times 4$ .

### 1.2.2. Устранение ступенчатости

Рассмотрим причину появления ступенчатости. Изображение хранится в памяти как двумерная матрица пикселей. Однако считается, что изображение предметов реального мира это не просто набор пикселей. Авторы [5] склонны рассматривать изображение как непрерывную двумерную функцию. То, что мы видим на экране – приближение этой функции, ее дискретизованное в заданном разрешении представление. Ступенчатость - результат дискретизации.

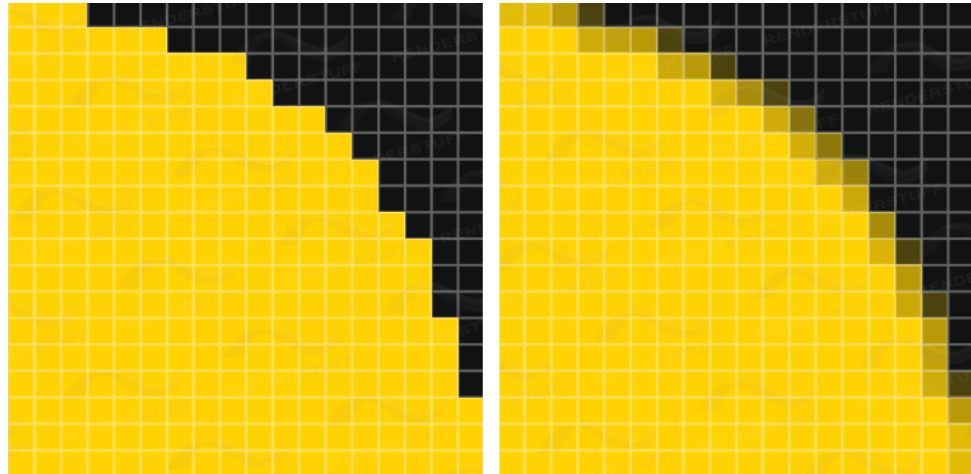


Рис. 1.3. Устранение ступенчатости.

Трассировка лучей в самом общем понимании - это метод, позволяющий восстановить значения функции изображения с помощью точечных выборок, то есть значений этой функции в точках. Самый простой способ устранения ступенчатости - трассировать больше чем 1 лучей на пиксел с некоторыми

смещениями внутри самого пиксела, а полученный результат усреднять. Однако для того чтобы уменьшить ступенчатость, достаточно часто не только трассируют больше чем 1 луч на пиксел, но и применяют фильтрацию. Рассмотренное выше усреднение соответствует так называемому box-фильтру. Достаточно простым, но более эффективным методом является билинейная фильтрация, при которой вклад от луча распределяется по 4 ближайшим пикселям.

Как правило процесс вычисления значения функции стараются отделить от способа хранения изображения. По этой причине во многих системах визуализации и расчета освещения присутствуют такие классы как *Sampler* (отвечает за фильтрацию) и *Integrator* (отвечает собственно за расчет отсвещения).

### 1.2.3. Прозрачные объекты и тени

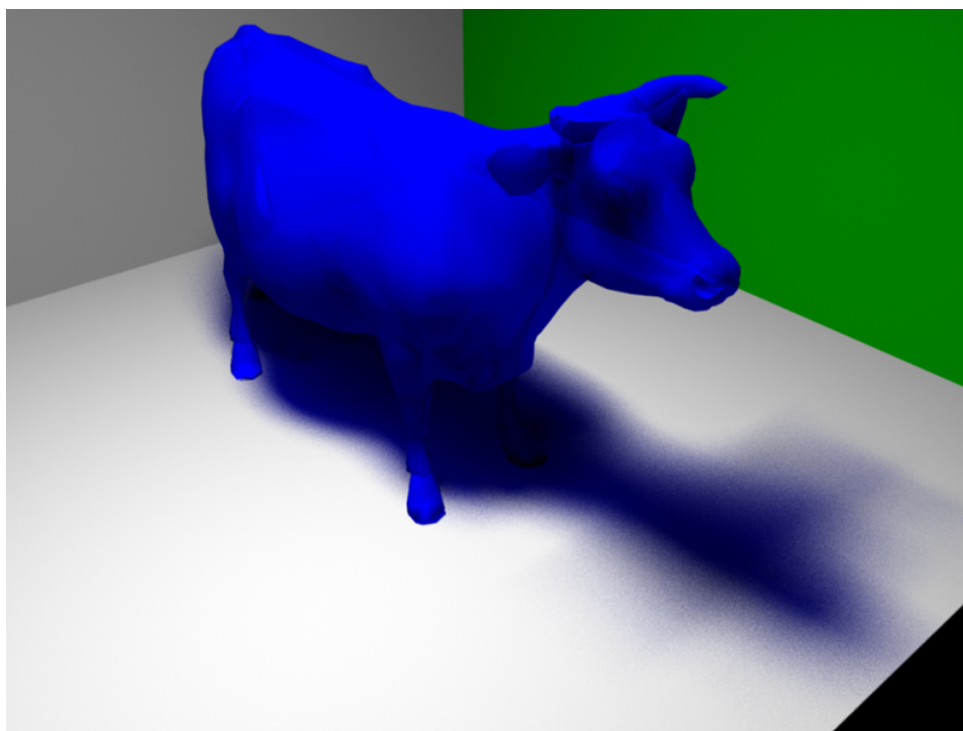


Рис. 1.4. Цветная тень.

При расчете тени, можно использовать более сложную модель. Если есть вероятность того, что один объект перекрывается другим прозрачным объек-

том, можно трассировать теневой луч сквозь прозрачный объект, рассчитывая по закону Бугера-Ламберта-Бэра затухание в прозрачной среде в зависимости от пройденного расстояния ( этот же закон нужно использовать и при учете затухания внутри прозрачных объектов для обычных лучей). При таком алгоритме тень становится цветной. Разумеется, тени, рассчитанные таким образом корректны, только если прозрачный объект, отбрасывающий тень, имеет близкий к единице коэффициент преломления (считаем что коэффициент преломления воздуха равен 1). Если это не так, то под прозрачным объектом образуется сложная картина, называемая каустиком. Типичный пример каустика – солнечный зайчик от стакана воды, когда через него просвечивает солнце. Корректные способы расчета каустиков будут рассмотрены далее в разделе про вычисление интеграла освещенности.

#### **1.2.4. Корректный расчет преломлений**

В рассмотренном ранее алгоритме не обсуждалась возможность получения реалистичных изображения преломляющих объектов. Здесь необходимо отметить 2 основных момента. Во-первых существует эффект так называемого полного внутреннего отражения при котором выходя из более плотной среды в менее плотную, при большом угле падения, луч полностью отражается внутрь более плотной среды и преломления не происходит. Во-вторых, учета только закона Снэллиуса для описания преломляющих объектов недостаточно. Поведение света для преломляющих объектов с достаточно-высокой степенью реалистичности описывается при помощи формул Френеля, которые устанавливают зависимость между долей отраженного и преломленного света в зависимости от угла падения луча и показателя преломления материала [6]. Хотя следует отметить, что и этого может быть недостаточно для реалистичной визуализации некоторых кристаллов, где необходимо учитывать

такие эффекты как двулучепреломление, поляризацию и интерференцию (на тонкой пленке оксида или другого покрытия кристалла) [7].

### 1.2.5. Поиск пересечений

Поиск пересечений луча с геометрическими объектами сводится к нахождению всех общих точек луча и объекта. Объекты в компьютерной графике, как правило представляются набором примитивов. Далее будут рассмотрены 2 наиболее важных вида примитивов - треугольник и прямоугольный параллелипипед. Треугольник важен, поскольку в современной компьютерной графике поверхности в подавляющем большинстве случаев задаются именно при помощи треугольников, а прямоугольный параллелипипед необходим для реализации эффективного поиска пересечений на основе BVH деревьев (которые будут рассмотрены далее).

#### Треугольник, барицентрический тест

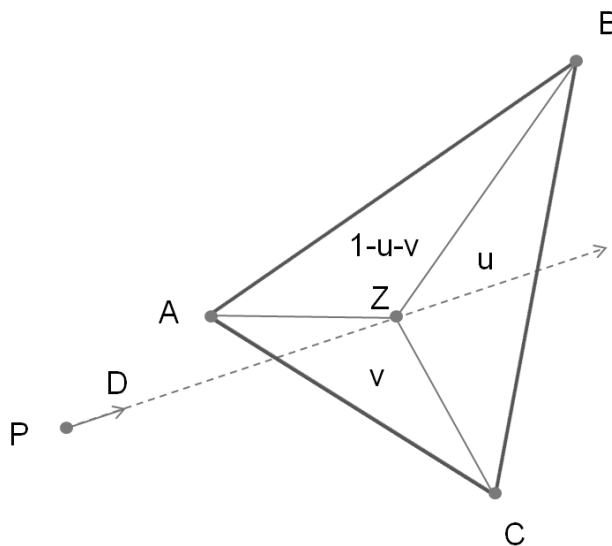


Рис. 1.5. Барицентрический тест треугольника и луча.

Введем следующие обозначения:

- $P$  - точка из которой вылетает луч;



- $D$  - направление луча;
- $A, B, C$  - вершины треугольника;
- Тройка  $(u, v, 1-u-v)$  - барицентрические координаты. Они представляют собой отношения площадей маленьких треугольников к большому треугольнику. То есть  $u = S(ZCB)/S(ABC)$ ,  $v = S(ZAC)/S(ABC)$ ,  $1-u-v = S(ZAB)/S(ABC)$ ;

$$\begin{aligned}
Z(u, v) &= u * A + v * B + (1 - u - v) * C \\
Z(u, v) &= P + t * D \\
P + t * D &= u * A + v * B + (1 - u - v) * C
\end{aligned} \tag{1.4}$$

Имея 3 точки на плоскости, можно выразить любую другую точку через ее барицентрические координаты. Первое уравнение получается из определения барицентрических координат, выражая точку пересечения  $Z$ . С другой стороны, эта же точка  $Z$  лежит на прямой. Второе уравнение таким образом, это параметрическое уравнение прямой. Приравняв правые части уравнений 1 и 2 получаем третье уравнение, которое, по сути, является системой 3-х уравнений ( $P, D, A, B, C$  - векторы) с 3-мя неизвестными  $(u, v, t)$ . Проведя алгебраические преобразования получим ответ в следующем виде:

$$\begin{aligned}
E1 &:= B - A \\
E2 &:= C - A \\
T &:= P - A \\
P &:= \text{cross}(D, E2) \\
Q &:= \text{cross}(T, E1)
\end{aligned} \tag{1.5}$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\text{dot}(P, E1)} * \begin{bmatrix} \text{dot}(Q, E2) \\ \text{dot}(P, T) \\ \text{dot}(Q, D) \end{bmatrix}$$

### Треугольник, юнит-тест (Woops unit test)

Основная идея данного алгоритма заключается в том, чтобы посчитать матрицу преобразования треугольника в некий единичный треугольник (отсюда название) с вершинами  $(1,0,0)$ ;  $(0,1,0)$ ;  $(0,0,0)$ ; и нормалью  $(0,0,1)$ . Во время подсчета пересечения с треугольником луч преобразуется этой матрицей в пространство, где треугольник имеет единичное представление. Назовем такое преобразование  $T_{\Delta}$ . После преобразования вычислить пересечение намного проще, так как нужно считать пересечение с заранее известным треугольником -  $(1,0,0)$ ;  $(0,1,0)$ ;  $(0,0,0)$ .

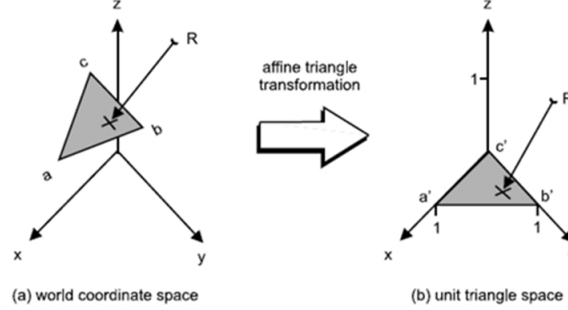


Рис. 1.6. Преобразование  $T_{\Delta}$ .

Пусть задан треугольник с вершинами A,B,C. Рассмотрим преобразование  $T_{\Delta}^{-1}$ :

$$T_{\Delta}^{-1} = \begin{bmatrix} A_x - C_x & B_x - C_x & N_x - C_x \\ A_y - C_y & B_y - C_y & N_y - C_y \\ A_z - C_z & B_z - C_z & N_z - C_z \end{bmatrix} * X + \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} \quad (1.6)$$

Заметим, что применив данное преобразование к треугольнику  $(1,0,0); (0,1,0); (0,0,0)$  - получим исходный треугольник  $(A,B,C)$ . То есть  $T_{\Delta}^{-1}$  в действительности является обратным преобразованием к  $T_{\Delta}$

$$T_{\Delta}^{-1} * \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = A \quad T_{\Delta}^{-1} * \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = B \quad (1.7)$$

$$T_{\Delta}^{-1} * \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = C \quad T_{\Delta}^{-1} * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = N \quad (1.8)$$

Для того чтобы найти нужное нам преобразование необходимо дополнить матрицу  $T_{\Delta}^{-1}$  до 4x4 и (добавить еще одну строчку  $(0,0,0,1)$ ) и найти обратную матрицу. Это будет соответствовать преобразованию  $T_{\Delta}$ .

Для каждого треугольника матрицу  $T_{\Delta}$  нужно рассчитать один раз и сохранить. Пересечение луча и треугольника в юнит-тесте вычисляется следующим образом (здесь  $M$  - матрица преобразования  $T_{\Delta}$ ).

$$\begin{aligned} \overline{O} &:= mul3x4(M, \overline{O}) \\ \overline{D} &:= mul3x3(M, \overline{D}) \\ t &:= -o.z/d.z; \\ u &:= o.x + t * d.x \\ v &:= o.y + t * d.y \end{aligned} \quad (1.9)$$

Функция `mul3x4` выполняет умножение подматрицы 3x3 на трехмерный вектор и добавляет к результату последний столбец (3 его компоненты). Функция `mul3x3` просто умножает подматрицу 3x3 на трехмерный вектор.

## Прямоугольный параллелипипед (AABB)

Прямоугольный параллелипипед (по англ. Axis Aligned Bounding Box - или AABB) обычно представляется в виде координат 2 точек. Нижнего левого угла (третье измерение опущено) -  $\text{boxMin}$  и правого верхнего угла -  $\text{boxMax}$ . Все точки, координаты которых находятся в интервале  $[\text{boxMin}, \text{boxMax}]$  по всем 3 измерениям находятся внутри параллелипипеда. Для того чтобы вычислить пересечение луча и прямоугольного параллелипипеда сначала вычисляют точки пересечения луча со всеми 6 плоскостями параллелипипеда (получая значения координат  $t$  в уравнении луча), после чего из каждой пары плоскостей (для  $x, y$  и  $z$ ) выделяют минимум и максимум (рис. 1.7). Результирующие координаты ищут как максимальный из минимумов ( $t_{\min}$ ) и минимальный из максимумов ( $t_{\max}$ ) - выражение 1.10.

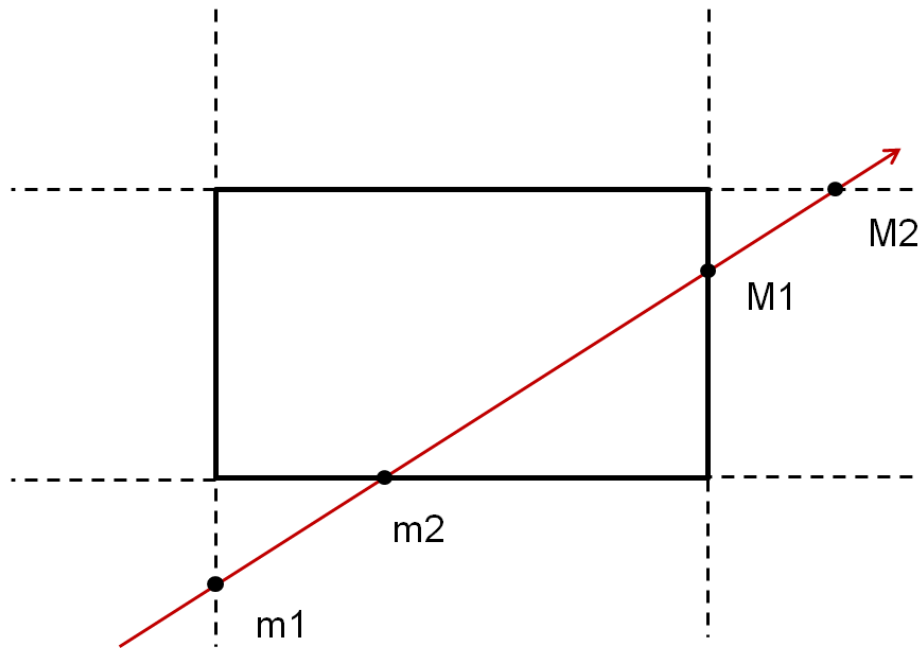


Рис. 1.7. Пересечение луча и AABB. Точки  $m1$  и  $m2$  - минимумы пересечения луча с парой плоскостей.  $M1$  и  $M2$  - максимумы.

$$\begin{aligned}
l_x &:= (1/D.x) * (boxMin.x - O.x); \\
h_x &:= (1/D.x) * (boxMax.x - O.x); \\
l_y &:= (1/D.y) * (boxMin.y - O.y); \\
h_y &:= (1/D.y) * (boxMax.y - O.y); \\
l_z &:= (1/D.z) * (boxMin.z - O.z); \\
h_z &:= (1/D.z) * (boxMax.z - O.z); \\
m1 &:= \min(l_x, h_x); \\
M1 &:= \max(l_x, h_x); \\
m2 &:= \min(l_y, h_y); \\
M2 &:= \max(l_y, h_y); \\
m3 &:= \min(l_z, h_z); \\
M3 &:= \max(l_z, h_z); \\
t_{min} &:= \max(m1, m2, m3) \\
t_{max} &:= \min(M1, M2, M3)
\end{aligned} \tag{1.10}$$

### 1.2.6. Ускорение поиска пересечений

#### Регулярные сетки

Ускорение поиска пересечений в трассировке лучей - довольно обширная тема.

Если сцена представлена набором примитивных объектов (сфера, ААВВ, треугольник), то ускорять процесс поиска пересечений при помощи структур пространственного разбиения имеет смысл, как правило, если число объектов больше одного-двух десятков. В противном случае, стоимость поиска может быть сравнима со стоимостью перебора всех объектов вслепую. Как правило,

это происходит из-за большого количества ошибок предсказателя ветвлений при выполнении сложного кода (особенно верно для иерархического поиска в дереве).

Данная ускоряющая структура регулярно разбивает пространство на  $N^3$  вокселей, где  $N$  - разрешение сетки. При поиске в регулярной сетке проверяются только те объекты, которые оказались внутри вокселей на пути луча.

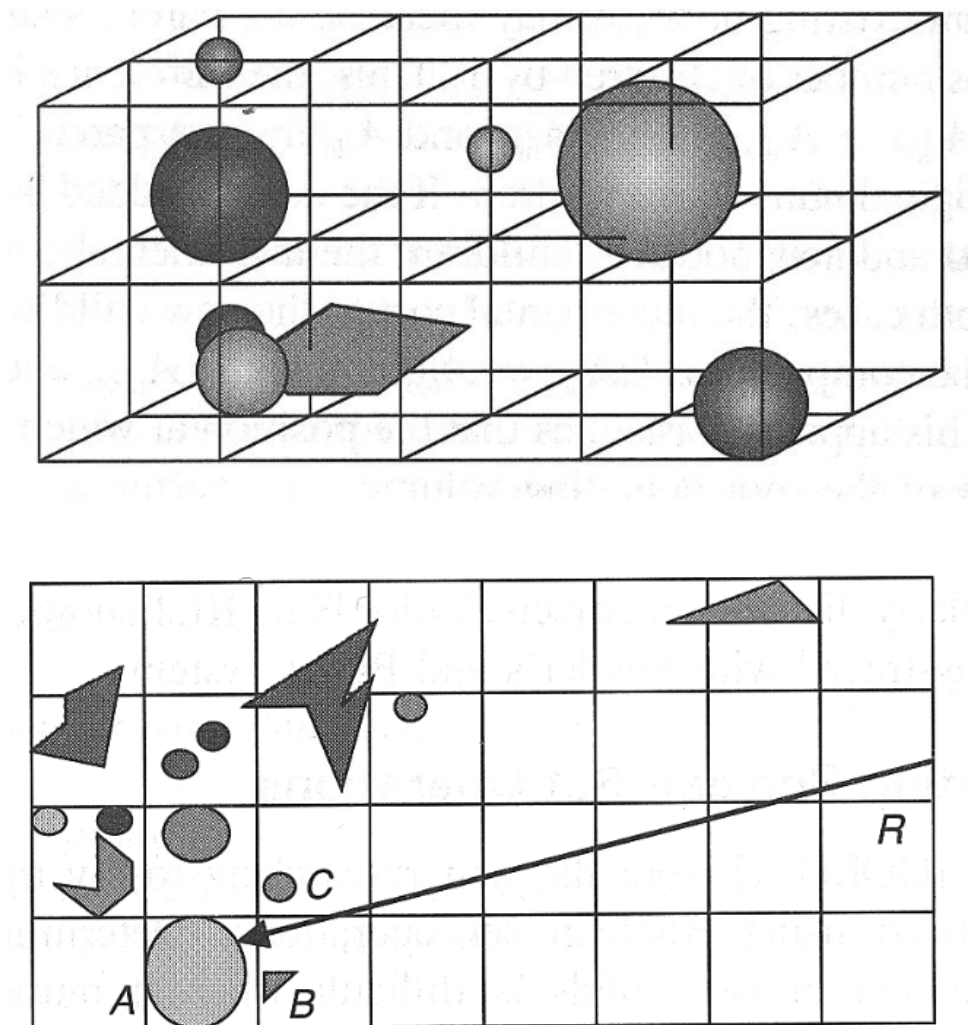


Рис. 1.8. Регулярное разбиение пространства.

Алгоритм построения сетки сводится к нахождению геометрических пересечений между вокселями и объектами сцены. Каждый воксел должен содержать список всех указателей или индексов объектов с которыми он имеет пересечение.

Алгоритм поиска носит фамилию своего автора Фуджимото [8]. Функция поиска состоит из 2 частей. Инициализация и перебор вокселей. Инициализирующая часть весьма объемная, однако, она выполняется для каждого луча 1 раз. Алгоритм 1 и рисунок 1.9 демонстрируют поиск Фуджимото.

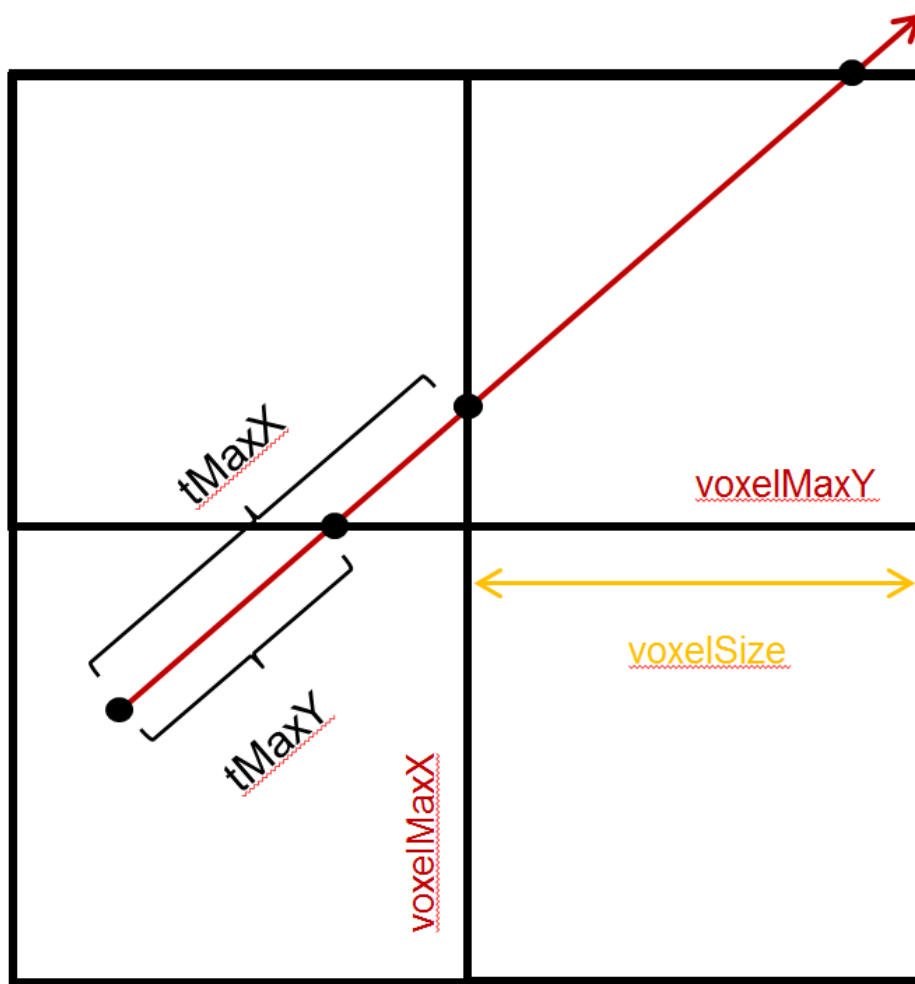


Рис. 1.9. Поиск в регулярной сетке.

**Исходные параметры:** Луч - ray, ограничивающий куб - box

**Результат:** Перебор вокселей, лежащих на пути луча

Инициализирующая часть алгоритма:

$(tMaxX, tDeltaX, stepX) \leftarrow \text{ВычислитьЗначения}(\text{ray}, \text{box});$

$(tMaxY, tDeltaY, stepY) \leftarrow \text{ВычислитьЗначения}(\text{ray}, \text{box});$

Основная часть алгоритма:

**while**  $(x < N)$  and  $(x \geq 0)$  and  $(y < N)$  and  $(y \geq 0)$  **do**

    ОбработатьВоксел(x,y);

**if**  $tMaxX < tMaxY$  **then**

$tMaxX \leftarrow tMaxX + tDeltaX;$

$x \leftarrow x + stepX;$

**else**

$tMaxY \leftarrow tMaxY + tDeltaY;$

$y \leftarrow y + stepY;$

**end**

**end**

### **Алгоритм 1:** Обход регулярной сетки Fujimoto

Несмотря на простой алгоритм перебора вокселей, регулярная сетка - не очень хорошая ускоряющая структура. Причин этому несколько:

1. Высокие затраты памяти, пропорционально  $N^3$ .
2. Отсутствие адаптивности или проблема чайника на стадионе. Регулярная сетка одинаково разбивает все пространство независимо от того, какая геометрия и как расположена на сцене. Размер вокселя выбирается исходя из некоторого среднего размера объектов. При этом если выбрать размер сетки большим, поиск пересечений значительно замедлится для небольших высокополигональных объектов. А если размер вокселя выбрать маленьким, не только значительно возрастут затраты



памяти но и замедлится трассировка на всей остальной сцене, т.к. в процессе трассировки луча придется перебирать большее число вокселей.

3. Проблема повторных пересечений. Эта проблема также усугубляется с ростом разрешения сетки. Объекты, пересекающие несколько вокселей при поиске пересечений будут встречаться несколько раз. Следовательно, при трассировке в регулярной сетке луч будет вычислять пересечение с такими примитивами более чем один раз. Проблему можно частично амортизировать сохраняя идентификатор луча в примитивах, с которым пересечение уже было посчитано. Если в данный момент времени идентификатор луча равен идентификатору сохраненному в примитиве, вычислять пересечение не нужно. Однако этот подход трудно применять при параллельной реализации трассировки лучей.

## **Окто-деревья и иерархические сетки**

Перейдя к иерархическому представлению, можно в некоторой степени решить первые 2 проблемы регулярной сетки (затраты памяти и неадаптивность). Иерархическая сетка представляет из себя дерево (обычно небольшой глубины, но зато очень широкое), в узлах которого лежат относительно небольшие регулярные сетки. Если размер сетки положить равным  $2 \times 2 \times 2$  получаем окто-дерево. Таким образом, окто-дерево представляет из себя иерархическое разбиение пространства на 8 частей (рис. 1.10).

Окто-деревья широко используются в компьютерной графике для различных задач пространственного поиска, но они довольно плохо подходят для ускорения трассировки лучей. Алгоритм обхода окто-деревьев достаточно сложен. При этом, адаптивность у окто-дерева низкая, так как чтобы ограничить небольшой объект (чайник на стадионе) нужно много уровней подразбиения. Окто-дерево, таким образом, обычно содержит множество пустых узлов и

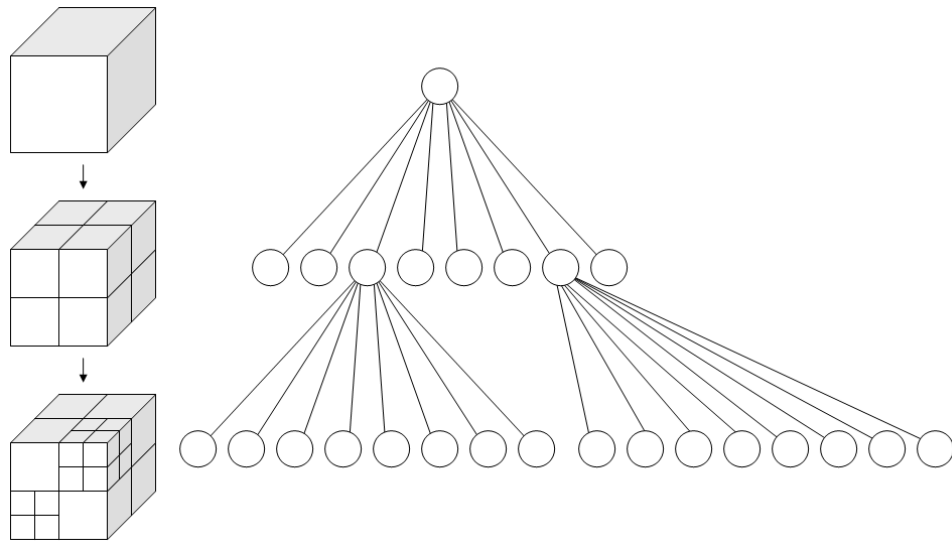


Рис. 1.10. Окто-дерево.

сложные регионы обходятся медленно (рис. 1.11). При этом, проблема повторных пересечений по-прежнему остается актуальной для окто-деревьев, поскольку при разбиении один объект очень часто попадает сразу в несколько дочерних узлов.

## kd-деревья

Рассмотрим структуру бинарного пространственного разбиения, называемую kd-дерево (аббревиатура kd расшифровывается как k-dimensional). Эта структура представляет собой бинарное дерево ограничивающих параллелепипедов, вложенных друг в друга. Каждый параллелепипед в kd-дереве разбивается плоскостью, перпендикулярной одной из осей координат на два дочерних параллелепипеда.

Вся сцена целиком содержится внутри корневого параллелепипеда, но, продолжая рекурсивное разбиение параллелепипедов, можно прийти к тому, что в каждом листовом параллелепипеде будет содержаться лишь небольшое число примитивов. Таким образом, kd-дерево позволяет использовать бинарный поиск для нахождения примитива, пересекаемого лучом.

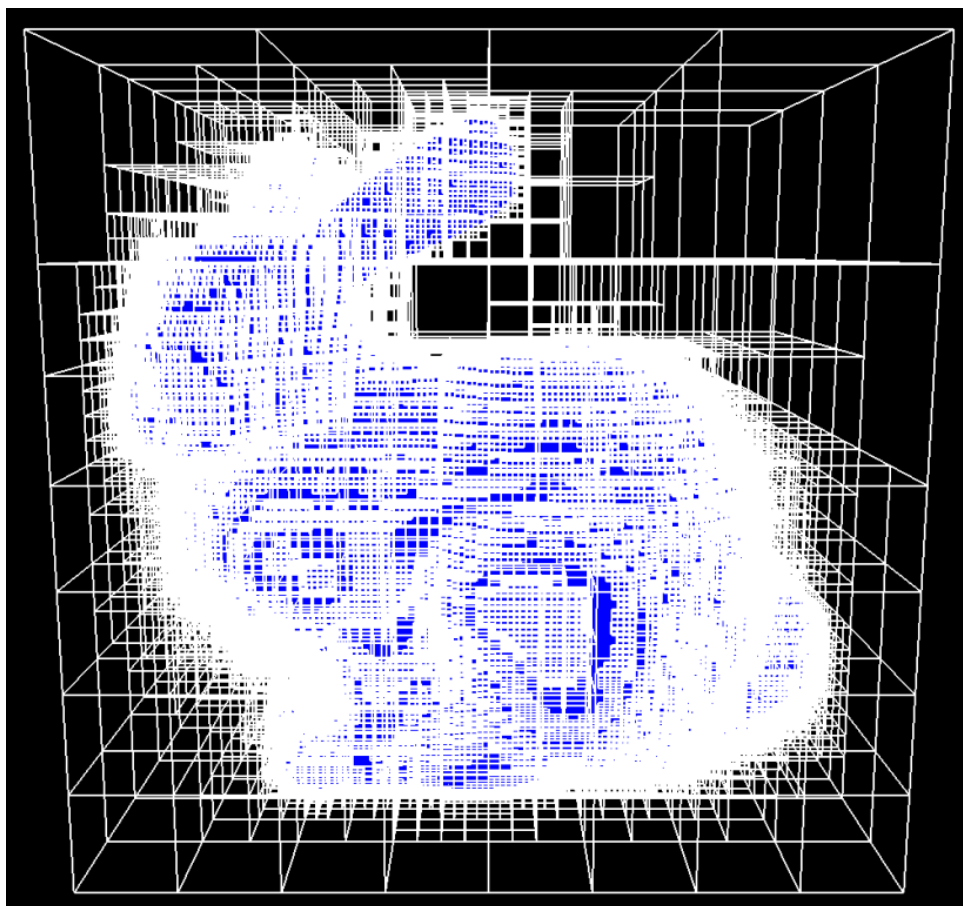


Рис. 1.11. Модель стэнфордского кролика в окто-дереве. Количество белого цвета на изображении визуально позволяет оценить сложность трассировки.

Если плоскость, разбивающую пространство выбирать каждый раз по середине параллелипипеда (при этом текущий узел можно разбивать всегда по оси, в которой он имеет максимальный размер), kd-дерево будет эквивалентно окто-дереву и наследует все его недостатки (за исключением сложного алгоритма обхода). Однако, основное преимущество kd-дерева над окто-деревом заключается как раз в том, что плоскость разбиения можно ставить не только по середине разбиваемого узла, а в любом месте.

Алгоритм построения kd-дерева можно представить следующим образом (будем называть прямоугольный параллелепипед сокращенно AABV).

1. 'Добавить' все примитивы в ограничивающий AABV. Т.е построить ограничивающий все примитивы AABV, который будет соответствовать

корневому узлу дерева.

2. Если примитивов в узле мало или достигнут предел глубины дерева, завершить построение.
3. Выбрать плоскость разбиения, которая делит данный узел на два дочерних. Будем называть их правым и левым узлами дерева.
4. Добавить примитивы, пересекающиеся с AABV левого узла в левый узел, примитивы, пересекающиеся с AABV правого узла в правый.
5. Для каждого из узлов рекурсивно выполнить данный алгоритм начиная с шага 2.

На рисунке 1.12 изображен процесс построения kd-дерева с учетом некоего оптимального алгоритма выбора разбивающей плоскости. Самым сложным в построении kd-дерева является 3-ий шаг. От него напрямую зависит эффективность ускоряющей структуры. Существует несколько способов выбора плоскости разбиения, рассмотрим их по порядку.

### **Разбиение по середине.**

Самый простой способ - выбирать плоскость разбиения по центру. Сначала выбираем ось (x, y или z), в которой AABV имеет максимальный размер, затем разбиваем его по центру. Ранее обсуждалось, что kd-дерево в этом случае имеет плохую адаптивность и эквивалентно окто-дереву.

### **Разбиение по медиане.**

Если разбивать узел на два дочерних таким образом, чтобы в правом и левом поддереве количество примитивов было одинаково, будет построено сбалансированное дерево. Это не очень удачная идея. Все дело в том, что сбалансированные деревья могут помочь только если искомый элемент каждый раз нахо-

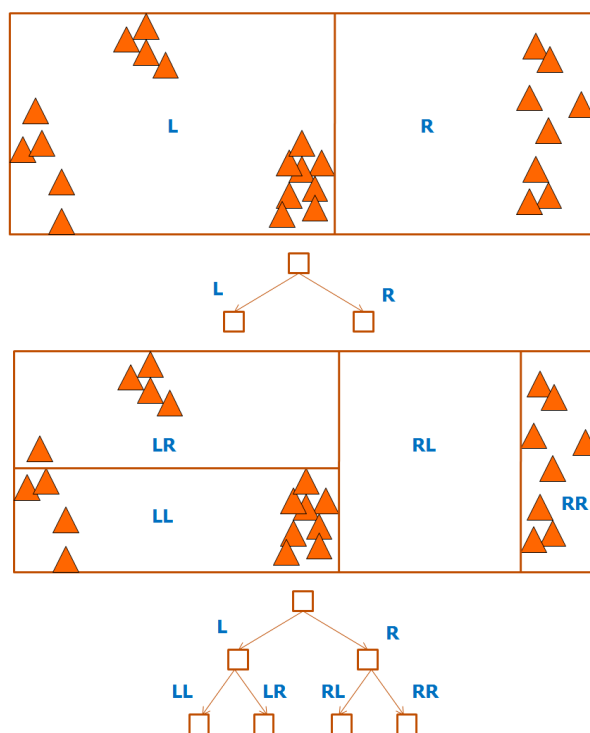


Рис. 1.12. Построение kd-дерева. L - обозначает левое поддерево. R - правое.

дится в случайном узле, то есть если распределение лучей по узлам во время поиска будет равномерно. В действительности, это не так. Лучей в среднем пойдет больше в тот узел, который больше по своей площади поверхности, а при медианном разбиении эти площади у узлов могут быть разные.

### Разбиение на основе оптимизации функции стоимости.

Каковы же критерии хорошо-построенного kd-дерева? На интуитивном уровне такой критерий можно описать фразой 'как можно больше пустого пространство должно быть отброшено как можно быстрее'. Для решения поставленной задачи используем формальный подход. Введем функцию стоимости поиска, которая будет отражать, насколько дорого по вычислительным ресурсам производить поиск в данном узле случайным набором лучей. Будем вычислять ее по следующей формуле [9]:

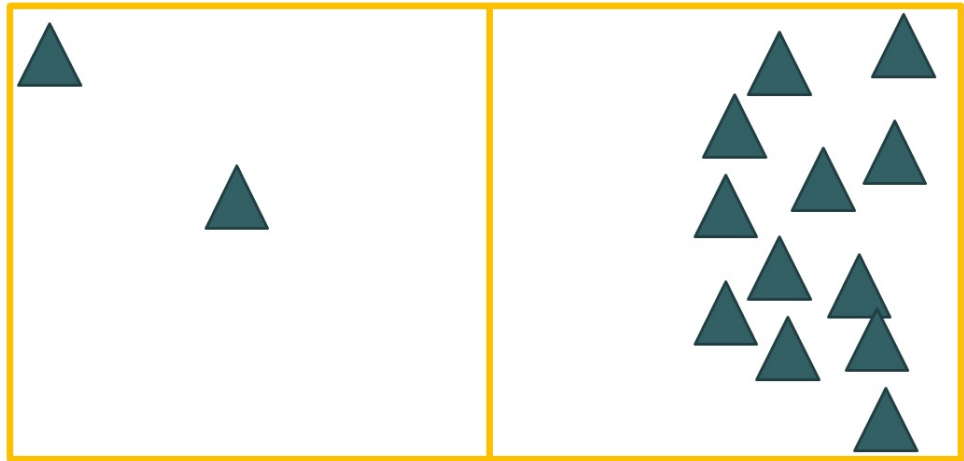


Рис. 1.13. Разбиение по середине.

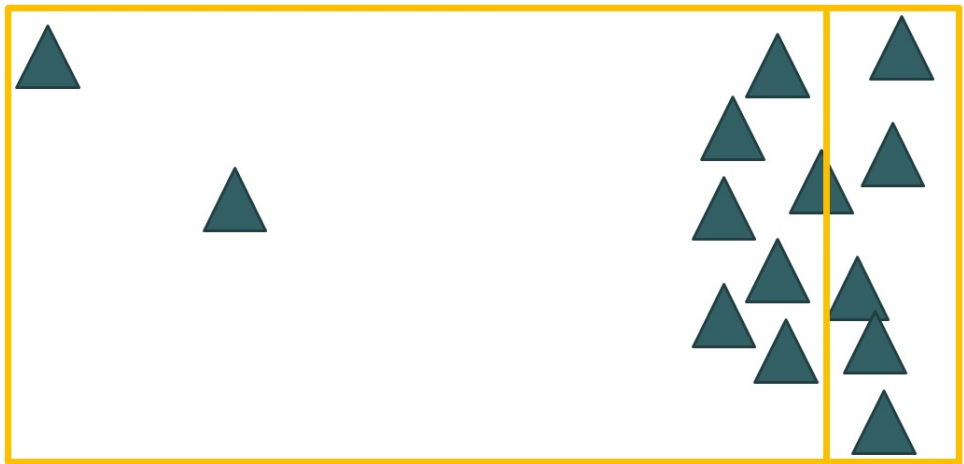


Рис. 1.14. Разбиение по медиане.

$$SAH(x) = CostEmpty + SA(Left) * N(Left) + SA(Right) * N(Right)$$

В приведенной выше формуле  $SA(Left)$  и  $SA(Right)$  - площадь поверхности (SurfaceArea) соответственно левого и правого дочерних узлов, получающихся при разбиении.  $N(Left)$  и  $N(Right)$  - количество примитивов, попавших при разбиении соответственно в левое и правое поддерево. Аргумент функции  $x$  является одномерной координатой плоскости разбиения. На рисунке 1.15 можно увидеть, что SAH сразу отбрасывает большие пустые пространства, плотно ограничивая геометрию.

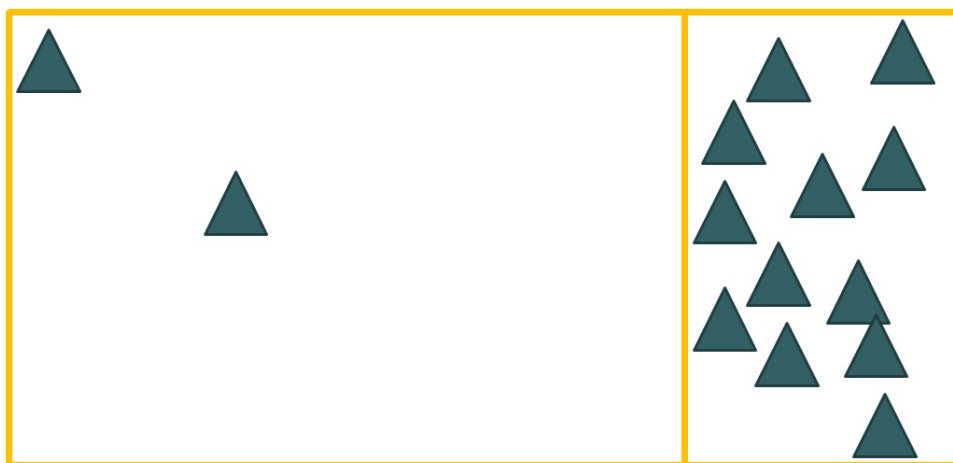


Рис. 1.15. Разбиение с учетом оптимизации стоимости.

Хорошими кандидатами на минимум SAN могут служить границы примитивов. Простой алгоритм построения выглядит следующим образом: каждый раз при выборе плоскости нужно перебрать всевозможные границы примитивов по трем измерениям, вычислить в них значение функции стоимости и найти минимум среди всех этих значений. Когда мы вычисляем SAN для каждой плоскости, то нам необходимо знать  $N(\text{left})$  и  $N(\text{right})$  - количества примитивов справа и слева от плоскости. Если вычислять  $N$  простым перебором, в итоге получится квадратичный по сложности алгоритм построения. Действенным способом ускорения поиска минимума функции SAN является использование какого-либо метода минимизации одномерной функции с несколькими начальными приближениями. Например, метод золотого сечения. Это избавляет от необходимости полного квадратичного перебора. Также популярен метод называемый binning [10, 11], который ускоряет поиск за счет снижения точности поиска положения найденного минимума.

### **Критерий остановки.**

Одним из критериев остановки может служить сама функция SAN. Если оцениваемая функцией суммарная стоимость поиска в дочерних узлах больше стоимости поиска в родительском узле, разбиение стоит остановить. Одна-

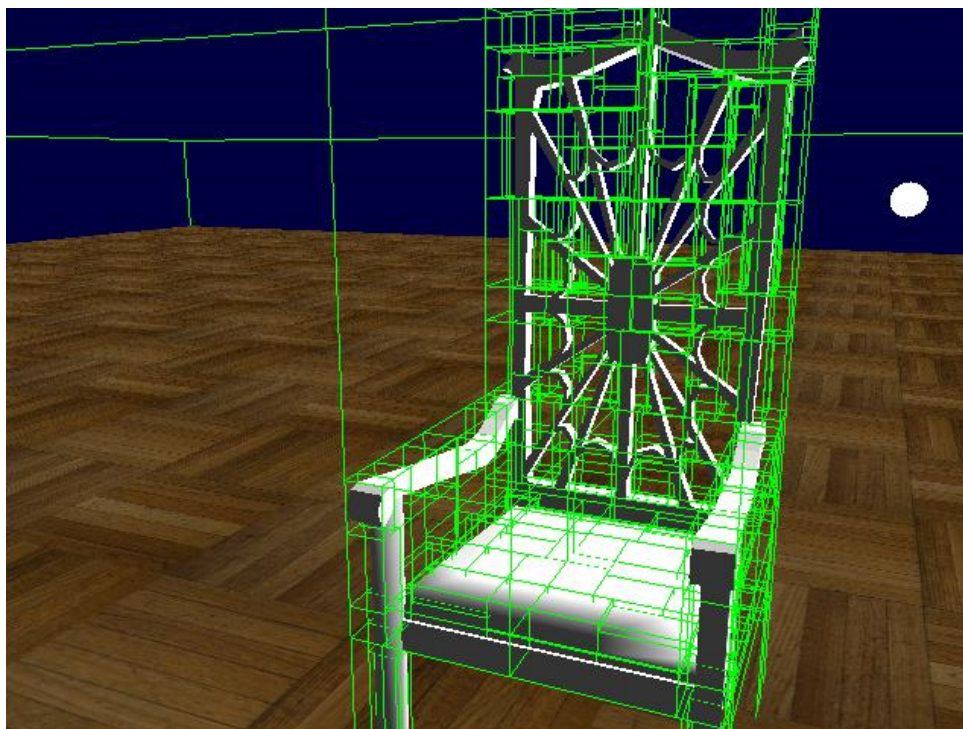


Рис. 1.16. kd-дерево, построенное с учетом SAN.

ко, весьма неочевидным фактом является то, что такой критерий остановки далеко не оптимален. Дело в том, что увеличение SAN при разбиении текущего уровня дерева еще не означает замедление трассировки для всего дерева. Рассмотрим цилиндр, составленный из полигонов на рисунке 1.17. Узел kd-дерева, обозначенный буквой А является проблемным. Какая-бы плоскость разбиения не была выбрана, SAN полученных дочерних узлов будет больше чем SAN родительского узла. Таким образом, SAN в данном случае говорит о том, что подразбиение нужно остановить. Но это не всегда так.

Если представить, что цилиндр - высокополигональный объект, дальнейшее подразбиение (даже с учетом выбора плоскости по центру) приведет к тому, что в листьях окажется не более чем 1-2 примитива. Однако остается открытым вопрос - будет ли при этом быстрее трассировка лучей и если да, то на сколько. И очередной возникающий вопрос формулируется так: Если существуют такие плоскости разбиения, которые увеличивают суммарный SAN текущего узла (который может быть оценен как  $SAN(\text{левого}) + SAN(\text{правого})$ ),



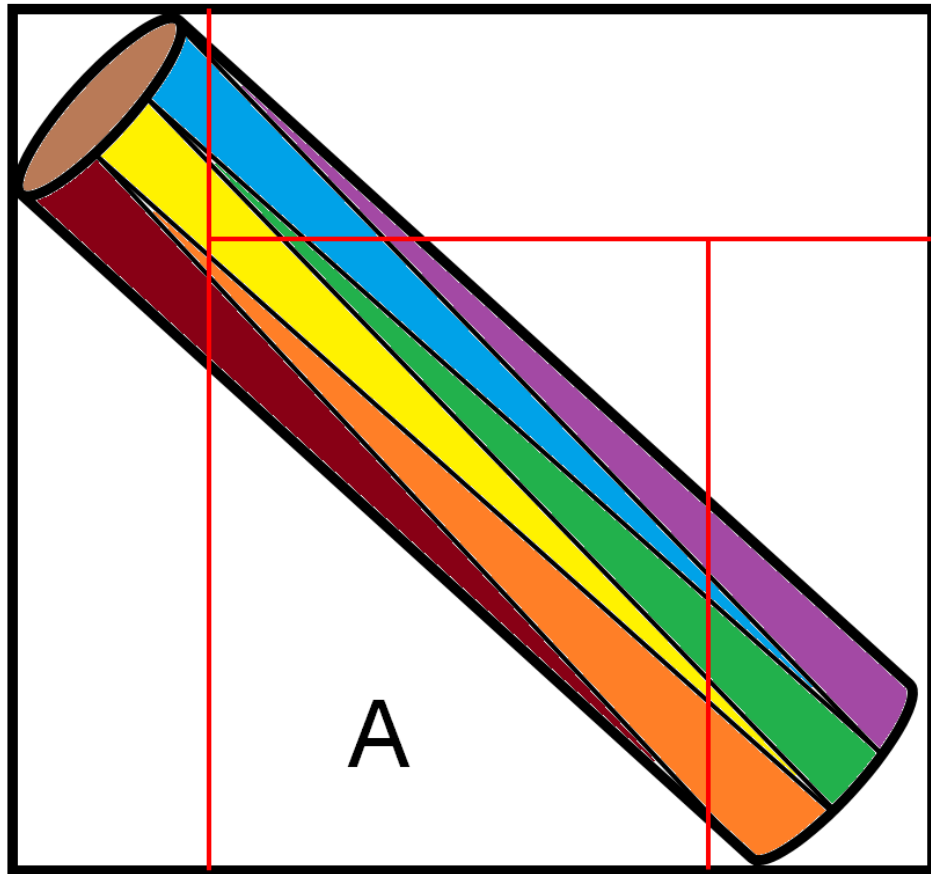


Рис. 1.17. Цилиндр, составленный из треугольников. Какую бы плоскость разбиения не была выбрана для узла A, это не позволит уменьшить SAH.

но при этом все-же позволяют ускорить трассировку, когда именно следует выбирать такие плоскости (т.е. на каком уровне дерева)?

Техника, позволяющая решить описанную выше проблему и ответить на поставленные вопросы называется в англоязычной литературе 'Early Split Clipping' [12]. Будем называть ее ранним подразбиением примитивов. Она говорит, что примитивы, которые плохо аппроксимируются при помощи AABV нужно подразбивать на несколько более мелких примитивов еще до построения дерева с тем, чтобы рассматривать такие 'сложные' плоскости наравне со всеми остальными в процессе оптимизации SAH. Подробнее данная техника будет рассмотрена при обсуждении BVH деревьев.

**Поиск в kd-дереве.**

Классический алгоритм бинарного поиска в kd деревьях (kd-tree traversal в англоязычной литературе), состоит в следующем: На первом шаге алгоритма необходимо посчитать пересечение луча с ограничивающим сцену корневым параллелепипедом (AABB) и запомнить информацию о пересечении в виде двух координат (в пространстве луча) –  $t\_near$  и  $t\_far$ , обозначающих пересечение с ближней и дальней плоскостями соответственно. На каждом следующем шаге необходима информация только о текущем узле (его адрес) и этих двух координатах.

При этом нет необходимости вычислять пересечение луча и дочернего параллелепипеда, достаточно лишь узнать пересечение с разбивающей параллелепипед плоскостью (обозначим соответствующую координату как  $t\_split$ ).

В случае если  $t\_split \geq t\_far$  (рис. 1.18) или если  $t\_split < t\_near$  (рис. 1.19) луч пересекает только один дочерний узел, поэтому можно просто отбросить правый (соответственно левый) узел и продолжить поиск пересечения в оставшемся узле.

В случае если луч пересекает оба дочерних узла (рис. 1.20), необходимо сначала поискать пересечение в ближнем узле и если оно не найдено, искать его в дальнем. Так как в общем случае неизвестно, сколько раз произойдет последнее событие (отсутствие пересечения в ближнем узле), необходим стек. Каждый раз, когда луч пересекает оба дочерних узла, адрес дальнего узла,  $t\_near$  и  $t\_far$  помещаются в стек и поиск продолжается в ближнем. Если в ближнем узле пересечение не найдено, из стека достаются адрес дальнего узла,  $t\_near$ ,  $t\_far$  и поиск продолжается в дальнем узле.

Почти все рассмотренные ранее проблемы регулярных сеток kd-дерево успешно решает. Оно обладает простым алгоритмом поиска, хорошей адаптивностью, быстро отбрасывая пустые пространства и значительно амортизирует проблему повторных пересечений. Однако серьезный недостаток kd-дерева – сложный алгоритм построения. Дело здесь не только в необходимости вычис-

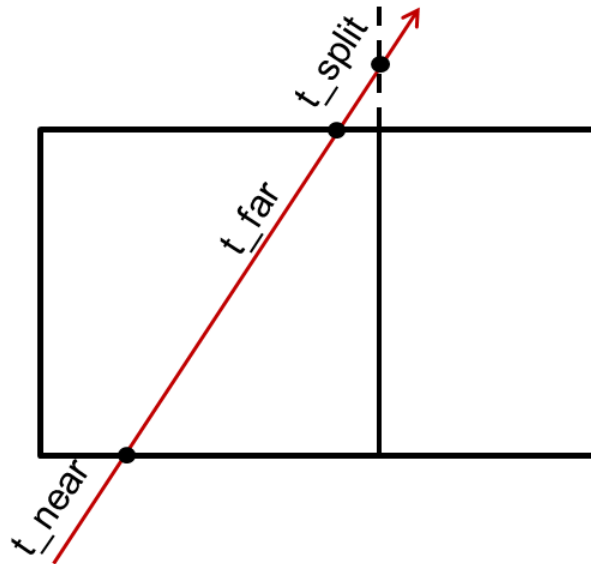


Рис. 1.18. Поиск в kd-дереве. ( $t\_split \geq t\_far$ ).

ления функции стоимости SAH, но, что гораздо более важно, перед началом построения kd-деревя невозможно сказать заранее, сколько памяти потребуется для его построения. Поскольку при разбиении узла все примитивы теоретически могут попасть как в левый, так и в правый дочерний узел, при разбиении узла с числом примитивов  $N$  необходимо иметь  $2N$  свободной памяти.

## BSP-деревья

В отличие от kd-деревьев, в которых плоскости разбиения имеют всего 3 возможных ориентации (XoY, XoZ, YoZ), в BSP (Binary Space Partion) деревьях плоскости разбиения могут быть ориентированны произвольным образом 1.21. Это значительно усложняет выбор плоскости с учетом оптимизации SAH (и как следствие сам алгоритм построения), но частично решает проблему обозначенную на рисунке 1.17. В работе [13] было показано преимущество BSP-деревьев в скорости поиска над kd-деревьями.

Тем не менее, даже BSP дерево не всегда гарантирует уменьшение числа примитивов в дочерних узлах (рис. 1.21, узел с тремя треугольниками справа

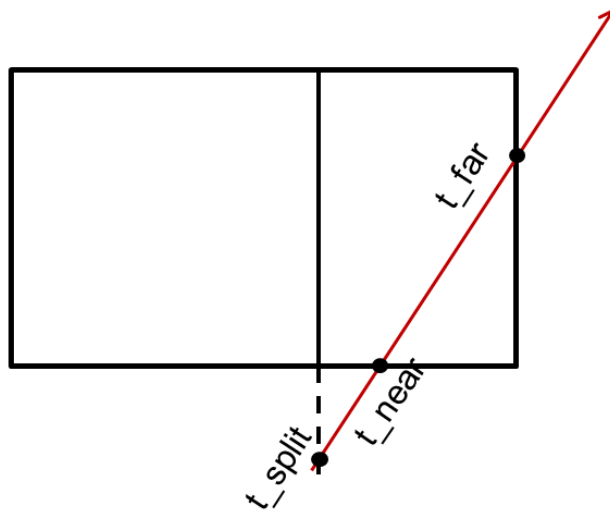


Рис. 1.19. Поиск в kd-дереве. ( $t_{split} < t_{near}$  ).

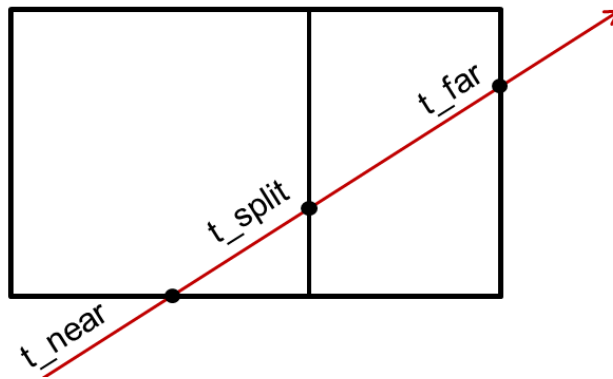


Рис. 1.20. Поиск в kd-дереве. ( $t_{near} \leq t_{split} \leq t_{far}$  ).

вверху). На практике из этого следует, что проблема с неизвестным количеством памяти, требующимся для построения дерева, все еще актуальна для BSP деревьев.

## BVH-деревья

Этот вид ускоряющей структуры является наиболее практичным и часто используемым в современных трассировщиках лучей. BVH расшифровывается как Bounding Volume Hierarchy - Иерархия Ограничивающих Объемов. BVH-дерево состоит из вложенных друг в друга объемов, ограниченных некоторыми примитивами (рис. 1.22). Такие деревья можно классифицировать по

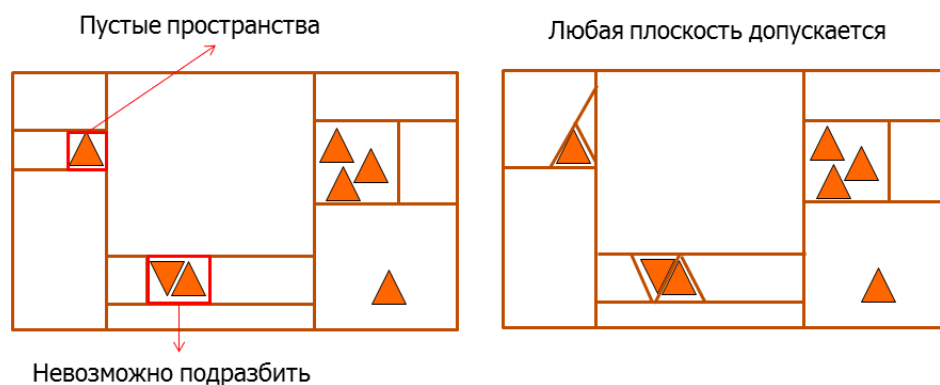


Рис. 1.21. kd-дерево (слева); BSP-дерево (справа).

типу ограничивающего примитива.

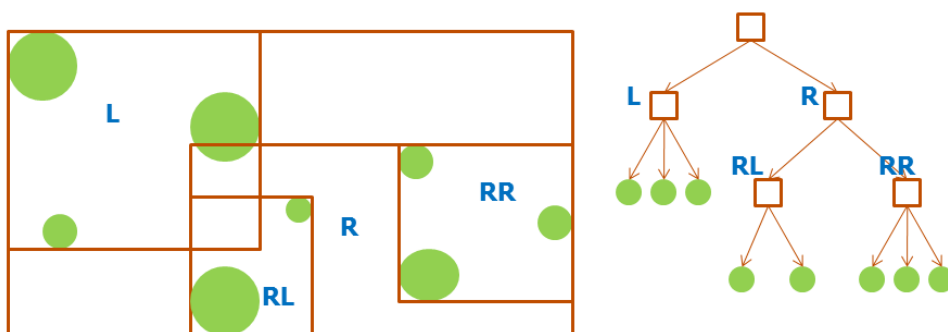


Рис. 1.22. BVH-дерево. В отличие от других видов пространственных деревьев, узлы дерева могут пересекаться друг с другом.

В дальнейшем будем рассматривать только деревья, использующие прямоугольные параллелипеды (Axis Aligned Bounding Box, AABB) в качестве ограничивающих примитивов и под BVH деревом понимать дерево состоящее из вложенных друг в друга прямоугольных параллелипедов, стороны которых выровнены по осям координат.

## Две стратегии разбиения в BVH дереве

При построении дерева существует небольшая, но довольно существенная путаница в том, какие примитивы включать в дочерние узлы и как именно производить разбиение. Вопрос, который порождает путаницу звучит так: "Что делать, если примитив пересекает оба дочерних узла?". Существует 2

стратегии разбиения.

Первая стратегия называется разбиением по объектам (рис. 1.23). При использовании этой стратегии необходимо найти некоторое разбиение нашего исходного множества примитивов на 2 подмножества. Затем достаточно лишь посчитать ограничивающий объем для обоих подмножеств. В этом случае ответ на поставленные ранее вопросы звучит так: Такой примитив всегда идет в тот узел, в котором он содержится полностью. Такой узел всегда существует по построению указанного разбиения.

Вторая стратегия называется пространственным разбиением (рис. 1.23). При использовании этой стратегии сначала выбирается некоторая плоскость, которая делит объем текущего узла на 2 части. Для полученных дочерних объемов ограничивающие AABV пересчитываются в соответствии с геометрией. В этом случае ответ на поставленный вопрос звучит так: "Такой примитив нужно добавлять в каждый из дочерних узлов, который он пересекает".

Рассмотренные стратегии порождают 2 принципиально разных вида BVH дерева. Стратегия разбиения по объектам позволяет строить такое дерево, в котором для каждого объекта хранится не более 1 ссылки. Будем называть такие деревья деревьями с одиночными ссылками (single reference trees).

В противоположность этому, стратегия пространственного разбиения может породить несколько ссылок на 1 объект. Будем называть такие деревья деревьями со множественными ссылками (multiple reference trees).

BVH деревья с одиночными ссылками позволяют значительно упростить алгоритм построения при относительно небольшой потере в эффективности поиска. В дальнейшем под BVH деревом будем понимать BVH дерево с одиночными ссылками, построенными при помощи стратегии разбиения по объектам.

## **Построение BVH дерева.**

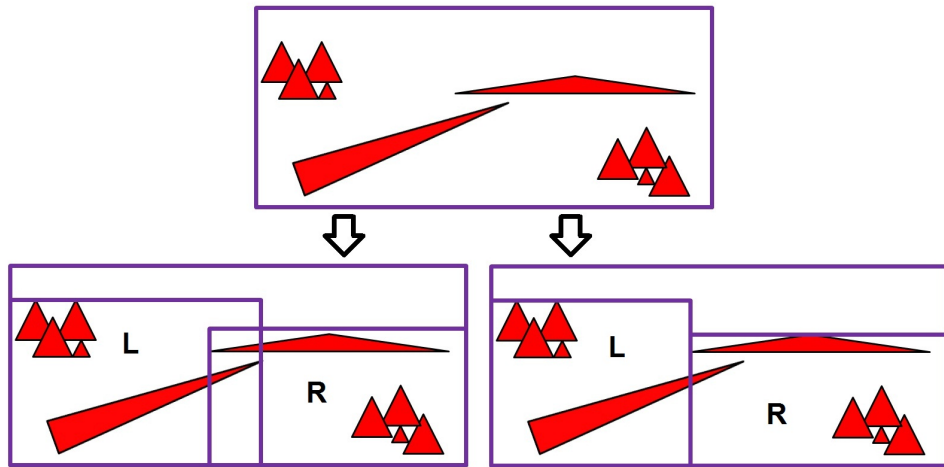


Рис. 1.23. Разбиение по объектам (слева) и пространственное разбиение (справа). Тонкий длинный треугольник справа при стратегии разбиения по объектам попадет только в левый узел L. А при стратегии пространственного разбиения он попадет в оба дочерних узла L и R.

Алгоритм построения оптимального BVH дерева столь-же сложен, сколь и алгоритм построения оптимального kd-дерева. В теории даже сложнее. Когда мы строим kd-дерево, то на каждом шаге подразбиения минимизируем функцию стоимости SАН. В случае kd-дерева мы выбираем только плоскость разбиения и можем перебрать все возможные границы примитивов по трем измерениям (обычно SАН вычисляют на границах), чтобы найти такую плоскость, в которой функция SАН будет минимальна. То есть в худшем случае, при отсутствии оптимизаций нам нужно вычислить SАН лишь  $6 \cdot N$  раз (где  $N$  - число примитивов) и найти минимум.

В случае BVH всё сложнее, поскольку вариантов возможных разбиений не  $6 \cdot N$  а  $S(N, 2) = 2^{N-1} - 1$  (где  $S$  - число Стирлинга 2 рода). Перебрать их все не представляется возможным. В отличие от kd-дерева, BVH на каждом уровне вообще говоря может произвести разбиение на 2 совершенно произвольных подмножества примитивов. Всего таких подмножеств  $2^{N-1} - 1$ .

Однако можно использовать упрощение, не сильно ухудшающее качество дерева, но позволяющее значительно упростить процедуру его построения. Бу-

дем строить BVH дерево сверху вниз. При построении, в каждом узле дерева сортируем примитивы по минимумам (или центрам) их AABV и 3 осям координат. Таким образом, получаем 3 массива. Первый отсортирован по X координатам минимумов (или центров) AABV объектов, второй по их Y координатам и третий по их Z координатам. Затем для каждого из полученных массивов (длинной  $n$ ) необходимо перебрать все разбиения на подмножества вида:

$$\begin{aligned} &[0; 1..n] \\ &[0..1; 2..n] \\ &[0..3; 4..n] \\ &\dots \\ &[0..n-1; n] \end{aligned}$$

Для каждого из этих разбиений вычисляем значение SAH. Запоминая ограничивающие AABV в специальном массиве для получаемых подмножеств мы можем не пересчитывать эти AABV полностью, что сводит поиск минимума SAH всего к 2 проходам по массиву. С конца к началу и с начала к концу. Описанный выше подход используется в реализации трассировки лучей из работы [14].

Среди эффективных подходов, позволяющих значительно сократить время построения BVH дерева для анимированных сцен следует отметить подход из работы [15]. За счет кластеризации входных данных, в работе [15] достигнуто уменьшение объема обрабатываемых построителем примитивов и, как следствие, значительное ускорение построения BVH дерева.



## Резюме по ускоряющим структурам

В сочетании с ранним подразбиением, BVH дерево на практике показало себя с наилучшей стороны [14]. Алгоритм построения BVH достаточно прост и занимает определенный объем памяти, пропорционально количеству входных примитивов.

### 1.3. Проблема глобальной освещенности

Как обсуждалось в начале данной главы, освещение принято разделять на локальное и глобальное. Локальным освещением называют освещение вызванное прямым попаданием света от источника на поверхность. Глобальным освещением называют освещение вызванное прямым и непрямым попаданием света на поверхность. Непрямое попадание света, в свою очередь, вызвано переотражениями световой энергии от различных объектов трехмерной сцены и является трудоемким в плане вычислений.

Один из способов получения фотореалистичных изображений - моделирование оптических процессов с целью точного вычисления освещенности каждой точки сцены и. Как следствие, появляется необходимость в вычислении глобальной освещенности. Трассировка лучей, описанная выше, позволила бы рассчитать изображение корректно, если бы поверхности объектов реального мира были гладкими и отражали свет строго по закону 'угол падения равен углу отражения'. Однако, из-за наличия микрорельефа поверхностей это не так, и свет приходящий строго с одного направления, распределяется по полусфере вокруг нормали к поверхности в соответствии со свойствами материала.

Для моделирования свойств материала вводят Двухнаправленную Функцию Отражения (ДФО) [16]. В английской терминологии обозначается как BRDF

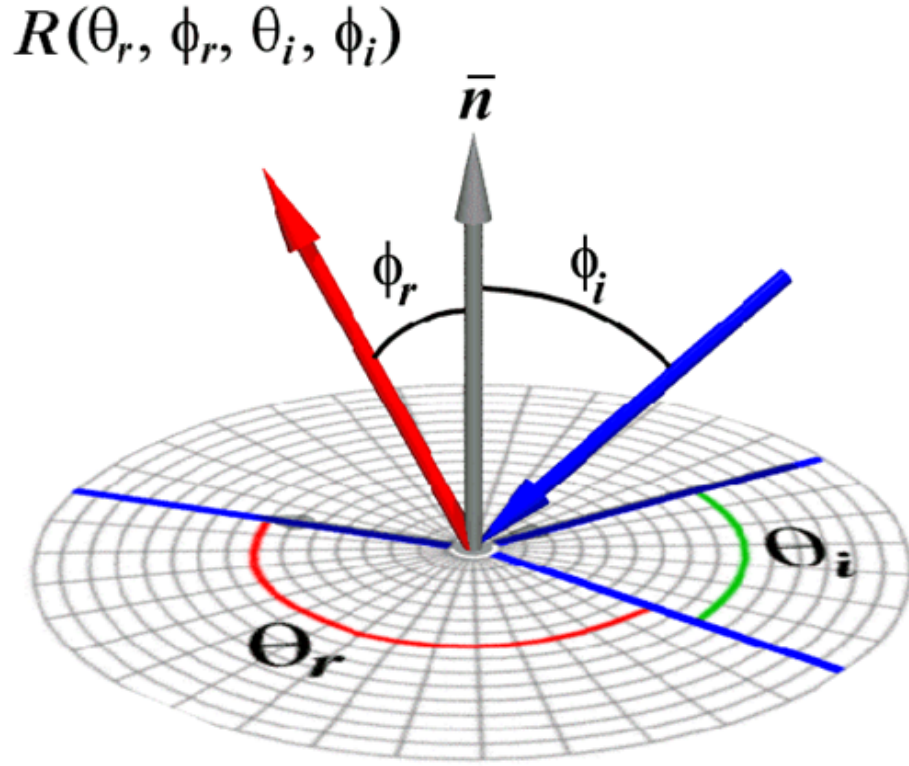


Рис. 1.24. Двухнаправленная Функция Отражения.

(Bidirectional Reflectance Distribution Function). ДФО связывает по некоторому закону интенсивность и угол падающего света с интенсивностью и углом отраженного поверхностью света. Зная ДФО материала, можно вычислить освещенность точки поверхности при помощи интеграла освещенности [17]:

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i \quad (1.11)$$

В уравнении 1.11 функция  $L(\phi_i, \theta_i)$  задает яркость света, падающего с направления задаваемого вектором  $l_{\phi_i, \theta_i}$ .  $R(\phi_i, \theta_i, \phi_r, \theta_r)$  - ДФО.  $n$  - вектор нормали к поверхности. Вычисляемая интегрированием интенсивность освещения в точке  $I(\phi_r, \theta_r)$  является не числом, а функцией. Другими словами, она дает значения интенсивности света, отражаемой поверхностью под разными углами. Таким образом, выполняя интегрирование по полусфере приходящего (падающего) в точку освещения  $L$  с учетом свойств поверхности  $R$ , получаем

новую функцию распределения освещения в пространстве  $I$ , обусловленную свойствами отражения поверхности (материала) в некоторой точке  $x$ . Если зафиксировать конкретный луч (например выпущенный для конкретного пиксела и виртуальной камеры), направление  $\phi_r, \theta_r$  было бы фиксированно и задавалось бы этим лучом.

Формула 1.11 берется из тех соображений, что освещенность в точке поверхности складывается из света, падающего на поверхность со всех направлений и отражающегося в заданном направлении по определенному закону (который задает ДФО). Эта формула не более чем математическая модель и она верна не всегда. Например, в случае стекла нужно учитывать свет, приходящий из под поверхности и проводить интегрирование по полной сфере. В случае кожи ситуация еще сложнее, так как необходимо учитывать такой эффект как подповерхностное рассеивание [18, 19].

Размерность интеграла освещенности (формула 1.11) зависит от количества учитываемых переотражений света, поскольку функция  $L$  в некоторой точке  $x$  будет зависеть от функции  $I$  в некоторой другой точке  $y$ . Из-за большой размерности интеграла для его вычисления обычно используют метод Монте-Карло, поскольку его скорость сходимости не зависит от размерности интеграла.

### 1.3.1. Метод Монте-Карло

Пусть  $u$  - случайная величина, равномерно распределенная на отрезке  $[a, b]$ . Тогда  $f(u)$  - тоже случайная величина. Исходя из определения, ее математическое ожидание равно:

$$Ef(u) = \int_a^b f(x)p(x)dx$$

$$p(x) = \frac{1}{b-a} \quad (1.12)$$

Где  $p(x)$  - плотность вероятности равномерного распределения. Выражая  $\int_a^b f(x)dx$  из формулы 1.12 получаем:

$$\int_a^b f(x)dx = (b-a)Ef(u) \approx \frac{b-a}{N} \sum_{i=1}^N f(u_i) \quad (1.13)$$

В применении к интегралу освещенности (имеем право взять полусферу единичной площади):

$$I(\phi_r, \theta_r) \approx \frac{\sum_{i=1}^N L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i})}{N} \quad (1.14)$$

Итак, мы можем оценивать значения интеграла при помощи вычисления суммы достаточно большого числа значений под-интегральной функции [20]. Далее будут рассмотрены некоторые определения и понятия математической статистики.

### **Выборка по значимости (Importance Sampling).**

Эффективным способом улучшения сходимости метода Монте-Карло является выборка по значимости (метод существенной выборки, importance sampling) [20]. Данный метод предполагает использование случайной величины не с равномерным распределением, а с таким распределением, которое пропорционально модулю вычисляемой функции. Идея выборки по значимости базируется на том, что некоторые значения случайной величины в процессе

моделирования имеют большую значимость для оцениваемой функции, чем другие. Если эти 'более вероятные' значения будут появляться в процессе выбора случайной величины чаще, дисперсия оцениваемой функции уменьшится и скорость сходимости, таким образом, увеличится [20]. При этом, для того чтобы метод давал корректный результат, необходимо учитывать получаемые значения с весами, обратно-пропорциональными вероятностям появления выборки.

При этом, если распределение случайных величин не равномерное, оценка для метода Монте-Карло запишется в более общем виде [20]:

$$\int_a^b f(x)dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(u_i)}{p(u_i)} \quad (1.15)$$

где  $p(u_i)$  - функция плотности вероятности случайной величины  $u$ .

### **Смещенные и несмещенные оценки и методы.**

В данном разделе мы кратко рассмотрим понятие смещенных и несмещенных оценок в применении к вычислению интеграла освещенности чтобы пояснить такие часто встречающиеся англо-язычные термины как biased renderer и unbiased renderer.

Пусть  $X_1..X_n$  случайная выборка из распределения, зависящего от параметра  $\theta \in \Theta$

Статистикой называется любая измеримая функция от выборки  $\theta_i$ , не зависящая от  $\theta$  (важно, что значение статистики можно вычислить при каждой реализации выборки, не зная значения  $\theta$ ). В случае метода Монте-карло под интегральное выражение может называться статистикой.

Точечной оценкой называется любая статистика, принимающая некоторые значения на области определения  $\Theta$ .

Оценка называется несмещенной (unbiased) если ее математическое ожидание равно оцениваемому параметру. В противном случае оценка называется смещенной (biased).

Оценка называется состоятельной (consistent) если она сходится по вероятности (сходимость почти наверное) к оцениваемому параметру.

Будем называть метод или алгоритм вычисления интеграла освещенности несмещенным, если он позволяет получить несмещенную оценку интеграла освещенности для индивидуальной точки или набора точек трехмерного пространства. В противном случае будем называть метод смещенным.

Будем называть метод вычисления интеграла освещенности состоятельным, если он позволяет получить состоятельную оценку интеграла освещенности для индивидуальной точки или набора точек трехмерного пространства.

Таким образом, если программное обеспечение для вычисления освещенности и фотореалистичной визуализации (англ. renderer) использует в том или ином виде метод Монте-Карло и при этом позволяет при бесконечно-большом времени расчета получить абсолютно точное решение, говорят что это несмещенный рендерер (unbiased renderer). К таким программам, например, можно отнести все виды трассировщиков путей. Если программа использует метод Монте-Карло, но при этом в пределе (при бесконечно-большом времени расчета) не позволяет получить абсолютно точное решение, говорят, что такой рендерер является смещенным (например программы, использующие фотонные карты и/или кэш освещенности). К программам, не использующим метод Монте-Карло понятия смещенный/несмещенный неприменимы. Например, некорректно использовать указанные термины по отношению к программе, использующей метод излучательности [21].

Смещенность алгоритма еще не означает, что он вычислит интеграл с низкой точностью. На практике смещение (bias) проявляется в виде цветных пятен. Если смещение небольшое, пятна незаметны для глаза. Как правило, сме-

щенные методы на начальном этапе работы (при небольшом числе выборок или небольшом времени работы) дают более приемлимую для человеческого глаза оценку, чем несмещенные. Однако в пределе, при длительном расчете несмещенные методы обычно дают более качественное изображение.

## 1.4. Стохастическая трассировка лучей

Стохастическая трассировка лучей (также называемая Монте-Карло трассировкой) вычисляет значение интеграла освещенности (для фиксированного направления  $\phi_r, \theta_r$ ) напрямую по формуле 1.14. Метод был предложен James-ом Кajiya в 1986 году [17].

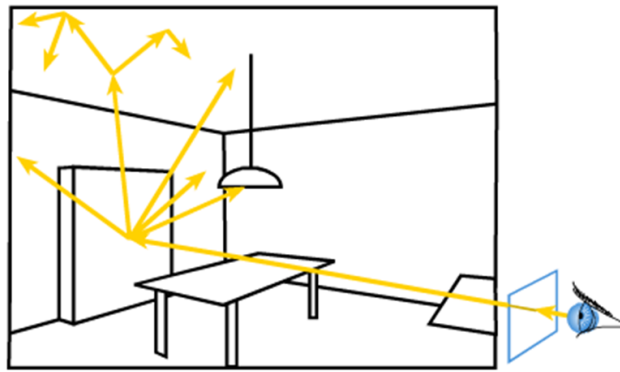


Рис. 1.25. Иллюстрация монте-карло трассировки лучей.

Из виртуальной камеры испускается луч, который трассируется в сцену. В точке пересечения с поверхностью выпускается некоторое число лучей по полусфере и для каждого луча процедура выполняется рекурсивно (рис. 1.25). Полученные значения яркости на каждом уровне рекурсии складываются с учетом формулы 1.14.

В дальнейшем будем рассматривать модификацию стохастической трассировки лучей, при которой отраженный луч всегда один (рис 1.26). Такое упрощение немного замедляет сходимость метода, но позволяет упростить реализацию и при желании избавиться от рекурсии в реализации алгоритма.

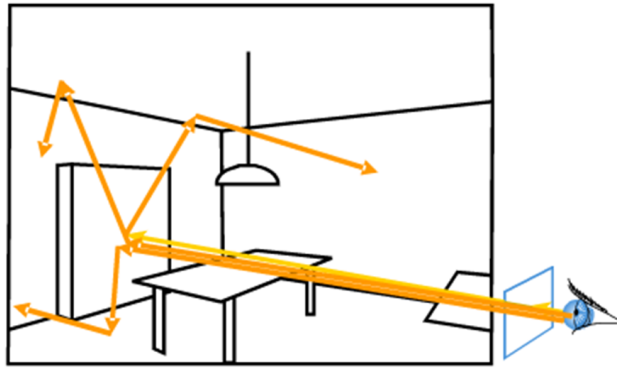


Рис. 1.26. Иллюстрация монте-карло трассировки путей.

Для начала будем считать, что в сцене все источники света имеют ненулевой размер и заданы в виде геометрических объектов, поверхность которых излучает свет. В целях упрощения изложения также будем полагать пока, что все материалы в сцене Ламбертовы (т.е. равномерно рассеивают свет во все сторо-



ны). Тогда следующий алгоритм позволит вычислить интеграл освещенности:

**Исходные параметры:** ray - Луч, depth - глубина рекурсии

**Результат:** Монте-Карло выборка значения освещенности

**Function** *TracePath*(ray: Ray; depth : Integer) : Integer **is**

```
    if depth == MAX_DEPTH then
        | return Black;
    end

    hit ← RaySceneIntersection(ray)

    if not hit.exists then
        | return Black;
    end

    m ← hit.material

    if m.IsLight then
        | return m.emittance;
    end

    newRay.O ← hit.pos
    newRay.D ← RandomUnitVectorInHemisphereOf(hit.norm)
    cosTheta ← dot(newRay.D, hit.norm)
    BRDF ← m.reflectance * cosTheta
    return BRDF * TracePath(newRay, depth + 1) ;

end
```

**Алгоритм 2:** Простейший вид Монте-Карло трассировки путей. В данном псевдокоде emittance обозначает светимость и измеряется в ваттах на квадратный метр.

Данный алгоритм также называют обратной трассировкой путей. Для генерации случайного вектора по полусфере необходимо реализовать функцию *RandomUnitVectorInHemisphereOf*, отображающую случайные числа  $r_1$  и  $r_2$  с равномерным распределением из интервала  $[0,1]$  на сферу. Такое отобра-

жение можно реализовать используя формулы 1.16 (результатирующие координаты -  $(x, y, z)$ ) [6].

$$\begin{aligned}
 \phi &= 2 * \pi * r_1 \\
 h &= 2 * r_2 - 1 \\
 x &= \sin(\phi) * \sqrt{1 - h^2} \\
 y &= \cos(\phi) * \sqrt{1 - h^2} \\
 z &= h
 \end{aligned} \tag{1.16}$$

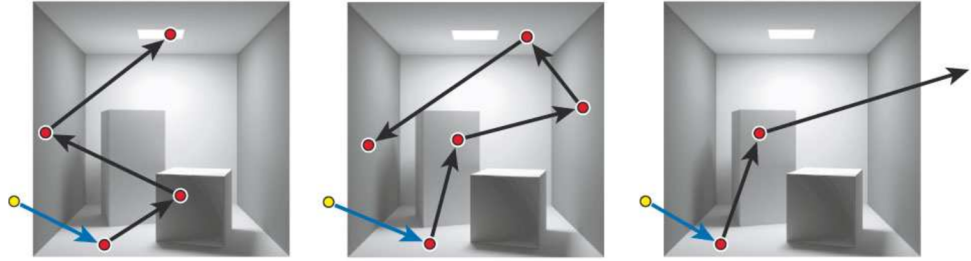


Рис. 1.27. Иллюстрация алгоритма трассировки путей. Корректный перенос света моделируется путем просчета большого числа путей.

#### 1.4.1. Прогрессивное вычисление интеграла

Для того чтобы иметь возможность визуализировать промежуточный результат в процессе расчета изображения, удобно использовать прогрессивную схему вычисления интеграла. Будем полагать, что мы используем простейший бокс-фильтр, усредняющий все пути для каждого пиксела. То есть цвет пиксела вычисляется по следующей формуле:

$$C = \sum_{i=1}^N \frac{P_i}{N} \tag{1.17}$$

Где:

1.  $P_i$  - значение Монте-Карло выборки на  $i$ -ом шаге.
2.  $N$  - число Монте-Карло выборок.

Будем вычислять цвет пиксела в виде сходящейся к  $C$  последовательности  $\{C_i\}$ . Пусть далее  $C_i$  - цвет пиксела изображения на  $i$ -ом проходе. Тогда он может быть вычислен прогрессивно по следующей схеме:

$$\begin{aligned}
 C_0 &= P_0 \\
 C_1 &= \frac{1}{2}C_0 + \frac{1}{2}P_1 \\
 C_2 &= \frac{2}{3}C_1 + \frac{1}{3}P_2 \\
 C_3 &= \frac{3}{4}C_2 + \frac{1}{4}P_3 \\
 &\dots \\
 C_n &= \frac{n}{n+1}C_{n-1} + \frac{1}{n+1}P_n
 \end{aligned} \tag{1.18}$$

Данная схема позволяет пользователю наблюдать постепенно улучшающееся изображение (с постепенно снижающимся уровнем шума). Если пользователя устраивает полученное изображение на шаге  $i$ , он может остановить расчет.

### Критерий остановки

Однако ручное управление не всегда возможно применить. Например, пользователь не может контролировать каждый отдельный пиксель изображения. Одним из наиболее стабильных критериев, позволяющих оценить точность полученного решения (и осуществить автоматическую остановку расчета) является дисперсия  $D$  и получаемое на ее основе стандартное отклонение  $\sigma$ . Дисперсию и стандартное отклонение удобно вычислять по формуле 1.19.

$$D = \frac{\sum_{i=1}^n X_i^2 - \frac{(\sum_{i=1}^n X_i)^2}{n}}{n-1}$$

$$\sigma = \sqrt{\frac{D}{n}} \quad (1.19)$$

Для реализации формулы 1.19 достаточно накапливать прогрессивно (по схеме 1.18) сумму цветов и сумму квадратов цветов.

#### 1.4.2. Применение выборки по значимости

Известно, что для Ламбретовых поверхностей наибольший вклад дают лучи близкие к направлению нормали, а наименьший - лучи почти перпендикулярные нормали. Это следует того, что значение падающей освещенности умножается на косинус между нормалью и направлением падающей освещенности (*cosTheta* в алгоритме 2).

Чтобы применить выборку по значимости, необходимо генерировать такой вектор отражения, который соответствует косинусоидальному распределению вокруг направления нормали (рис. 1.28). Это можно сделать эффективно используя соотношения 1.20, которые позволяют отобразить 2 равномерно-распределенных числа на сферу с косинусоидальным распределением [6]. В соотношениях 1.20 переменная  $e$  задает степень косинуса в косинусоидальном распределении.

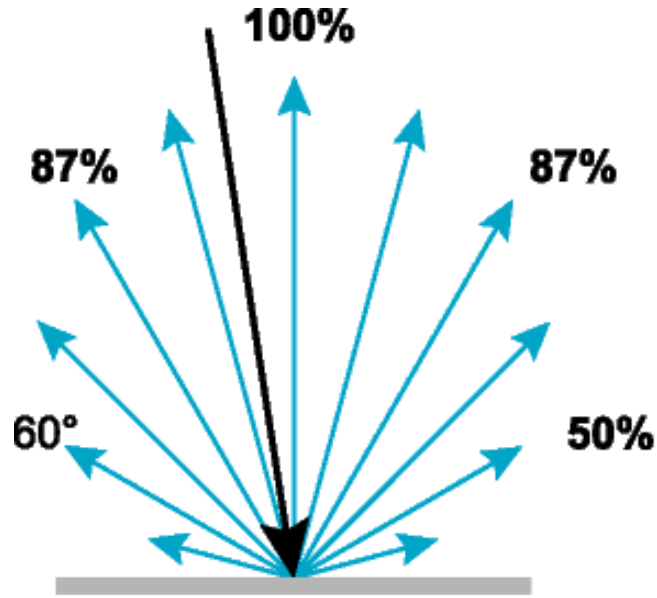


Рис. 1.28. Выборка по значимости для Ламбертовского отражения.

$$\begin{aligned}
 e &= 1 \\
 \phi &= 2 * \pi * r_1 \\
 \cos\theta &= (1 - r_2)^{1/(e+1)} \\
 \sin\theta &= \sqrt{1 - \cos^2\theta} \\
 x &= \sin\theta * \cos(\phi) \\
 y &= \sin\theta * \sin(\phi) \\
 z &= \cos\theta
 \end{aligned} \tag{1.20}$$

Тогда для того, чтобы результат не изменился, необходимо делить значение BRDF (или приходящей яркости умноженной на BRDF) на плотность вероятности выбранного распределения, то есть  $\cos\theta$ . В результате, при применении выборки по значимости для Ламбертовских материалов  $\cos\theta$  уходит, а функция *RandomUnitVectorInHemisphereOf* заменяется на функ-

цию *RandomCosineVectorOf* и алгоритм 2 переходит в алгоритм 3.

**Исходные параметры:** ray - Луч, depth - глубина рекурсии

**Результат:** Монте-Карло выборка значения освещенности

**Function** *TracePath*(ray: Ray; depth : Integer) : Integer **is**

```
    if depth == MAX_DEPTH then
        | return Black;
    end

    hit ← RaySceneIntersection(ray)

    if not hit.exists then
        | return Black;
    end

    m ← hit.material

    if m.IsLight then
        | return m.emittance;
    end

    newRay.O ← hit.pos
    newRay.D ← RandomCosineVectorOf(hit.norm)
    BRDF ← m.reflectance

    return BRDF * TracePath(newRay, depth + 1) ;
```

**end**

**Алгоритм 3:** Трассировка путей с учетом косинусоидального распределения отраженных лучей.

### 1.4.3. Учет сложных материалов

Ранее был рассмотрен алгоритм трассировки путей (алгоритмы 2 и 3), и делалось предположение, что все материалы в сцене Ламбертовы. Такие материалы отражают свет равномерно во все стороны. Однако, большинство материалов реального мира обладают намного более сложными ДФО, а так-

же могут не только отражать но и пропускать свет (рис. 1.29).

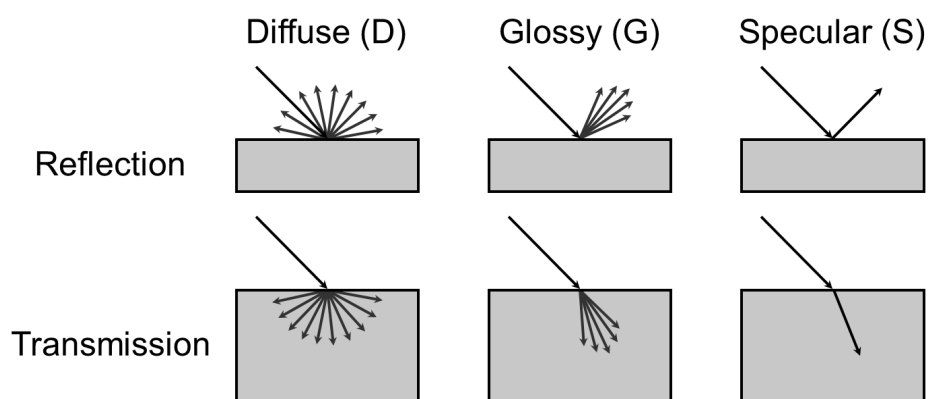


Рис. 1.29. Различные виды взаимодействия света с материалом. Diffuse(D) - Ламбертовское отражения. Specular(S) - зеркальное. Glossy(G) - матовое.

Идеально-зеркальное(S) и матовое(G) отражения моделируются при помощи единого механизма (который будет рассмотрен ниже), поэтому в дальнейшем будем рассматривать взаимодействия G и S совместно и называть такой тип отражения зеркальным (S).

Сложные ДФО часто представляются в виде композиции более простых компонент. Классической композицией является композиция ДФО Ламберта и Фонга (или любой другой ДФО, моделирующей зеркальное и/или матовое (glossy) отражение). В этом случае ДФО получается как линейная комбинация (с коэффициентами  $k_d$  и  $k_s$ ) отдельных компонент. Помимо этого, для прозрачных объектов необходимо учитывать пропускание (по аналогии с ДФО вводят функцию ДПФ или BTDF) с коэффициентом  $k_t$ . Для того чтобы учитывать такие композитные свойства материала, используют следующий механизм для выбора отраженного/преломленного луча:

$$\begin{aligned}
\Sigma &= kd + ks + kt \\
\xi &= rnd(0, 1) \\
\xi \in [0, \frac{kd}{\Sigma}] &\Rightarrow diffuse \\
\xi \in [\frac{kd}{\Sigma}, \frac{kd + ks}{\Sigma}] &\Rightarrow specular \\
\xi \in [\frac{kd + ks}{\Sigma}, 1] &\Rightarrow transmission
\end{aligned}$$

kd - коэффициент диффузного отражения. ks - коэффициент зеркального отражения. kt - коэффициент пропускания. Важно отметить, что при использовании модели выбора компоненты ДФО указанной выше, полученное значение яркости не нужно умножать на значения коэффициентов (kd, ks или kt) т.к. эти коэффициенты получаются стохастически из самой схемы.

Если необходимо учитывать цвет, схема усложняется:

$$\begin{aligned}
Pd &= max(kd.r, kd.g, kd.b) \\
Ps &= max(ks.r, ks.g, ks.b) \\
Pt &= max(kt.r, kt.g, kt.b) \\
\Sigma &= Pd + Ps + Pt \\
\xi &= rnd(0, 1) \\
\xi \in [0, \frac{Pd}{\Sigma}] &\Rightarrow diffuse \\
\xi \in [\frac{Pd}{\Sigma}, \frac{Pd + Ps}{\Sigma}] &\Rightarrow specular \\
\xi \in [\frac{Pd + Ps}{\Sigma}, 1] &\Rightarrow transmission
\end{aligned}$$

Однако это еще не все. Теперь полученную яркость необходимо делить на



значение плотности вероятности распределения выбранной компоненты. Например, если было выбрано зеркальное отражение, яркость луча необходимо пересчитать по следующей формуле:

$$\begin{aligned} color.r &= color.r / \left(\frac{Ps}{\Sigma}\right) \\ color.g &= color.g / \left(\frac{Ps}{\Sigma}\right) \\ color.b &= color.b / \left(\frac{Ps}{\Sigma}\right) \end{aligned}$$

Для того чтобы применять выборку по значимости с учетом матовых (glossy) ДФО, необходимо генерировать лучи со степенным косинусоидальным распределением в направлении отраженного вектора. Такое распределение представляет собой лепесток (cosine lobe) вокруг вектора отражения [6]. Чтобы его реализовать, необходимо изменить значение переменной  $\epsilon$  в соотношениях 1.20 присвоив ему соответствующую степень.

Результат работы алгоритма представлен на рисунке 1.30. Алгоритм позволяет корректно вычислить интеграл освещенности и получить весь спектр эффектов геометрической оптики (рис. 1.30).

#### 1.4.4. Теневые лучи

Представленный ранее алгоритм работает чрезвычайно долго, причем его скорость сильно зависит от размера источника освещения; алгоритм сходится тем дольше, чем меньше размер источника. Причина такой зависимости заключается в том, что вероятность случайного луча попасть в источник света зависит от его размера и расстояния до источника. Из этого следует, что трассировка путей в том виде, в котором она была рассмотрена ранее, не применима для сцен с точечными источниками. Чтобы устранить зависимость

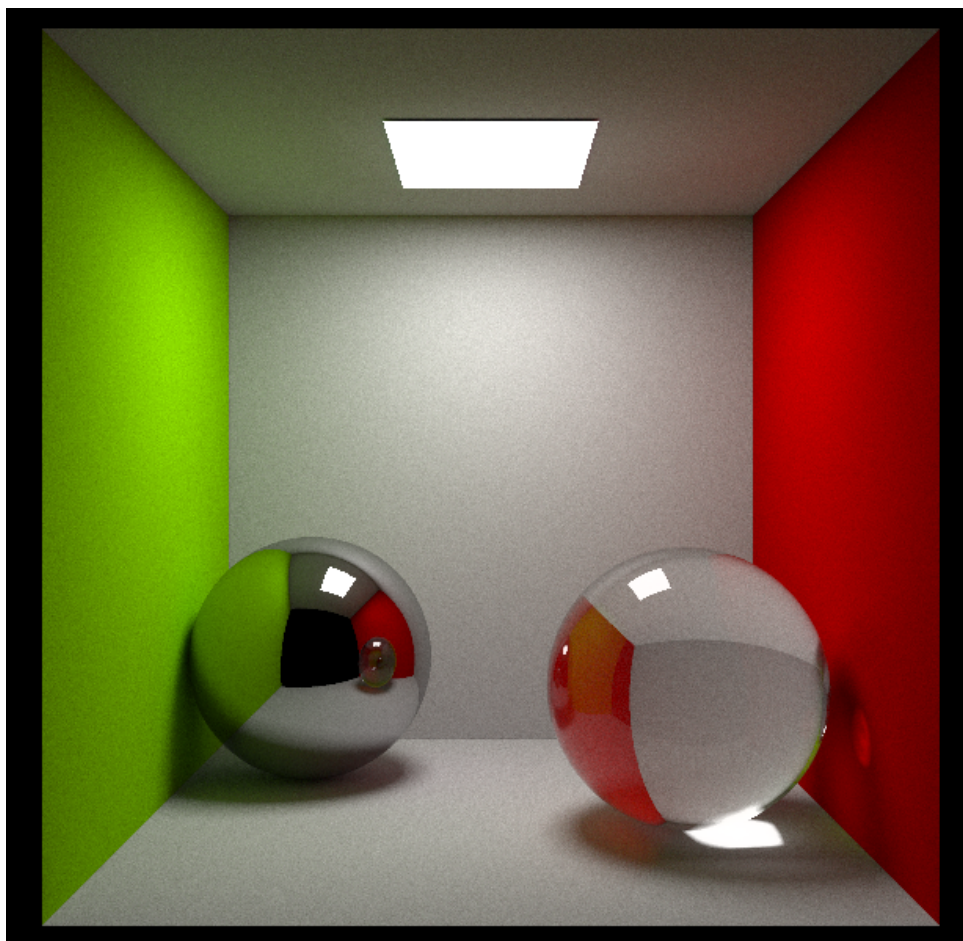


Рис. 1.30. Пример работы алгоритма трассировки путей на классической сцене Cornell Box.

времени расчета от размера источника и добавить поддержку точечных источников света, вводят так называемые теневые лучи. Идея заключается в том, чтобы генерировать выборки в соответствии с формой ДФО, создавая случайные отраженные лучи по полусфере, но также пускать лучи напрямую к источникам света, (рис. 1.31).

При реализации теневых лучей следует иметь ввиду 2 важных момента:

1. Чтобы не учитывать источник света 2 раза, случайные лучи, полученные в результате сэмплирования ДФО при попадании в источник света должны терминироваться и возвращать черный цвет.
2. Если для случайного луча вероятность попасть в источник света зави-

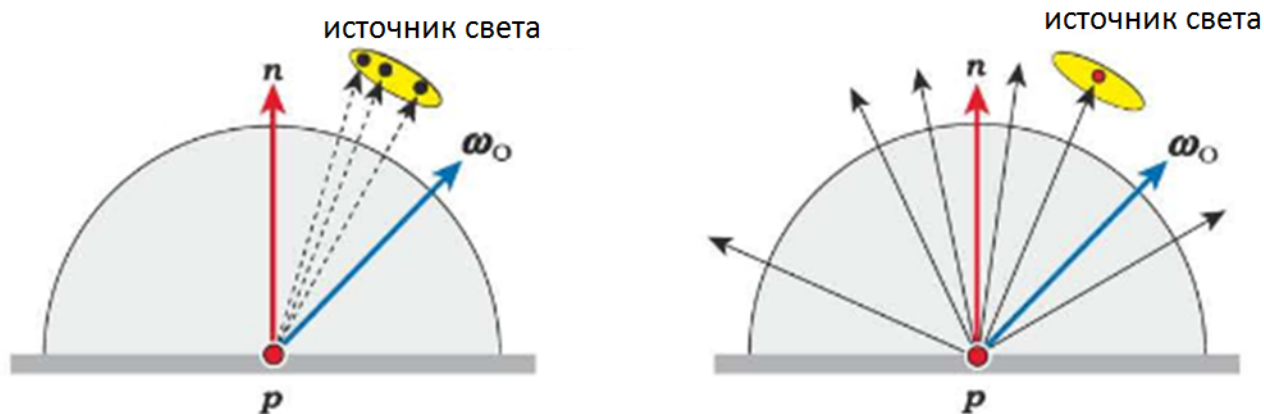


Рис. 1.31. Сэмплирование (генерация выборок) по ДФО (справа) и по источнику света (слева). Первую стратегию генерации выборок принято называть неявной. Вторую - явной.

сит от его площади поверхности и расстояния до него, то для теневых лучей вероятность попадания луча в источник света равна единице по определению (поскольку лучи всегда испускаются точно в источник). Для того чтобы корректно применить теневые лучи, необходимо вычислять такое же значение яркости теневого луча, как если бы луч попал в источник случайно, сэмплируя ДФО. Для этого вводится переменная *implicitP* (алгоритм 4). Она отражает вероятность попадания в источник света при использовании неявной стратегии.  $R$  - расстояние от точки, из которой испускается теневой луч до источника света; *light.area* - площадь поверхности источника; *emittance* - ранее использованная светимость поверхности источника света (измеряется в ваттах на квадратный метр).

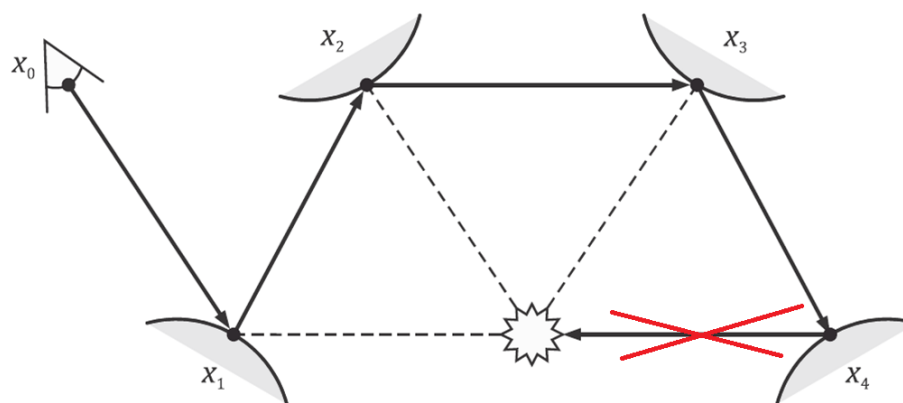


Рис. 1.32. Теневые лучи обозначены пунктиром. Отраженный луч, попавший в источник зачеркнут перекрестной красной линией. При возникновении такие лучи терминируются с тем чтобы не учитывать источник 2 раза.

**Исходные параметры:** ray - Луч, depth - глубина рекурсии

**Результат:** Монте-Карло выборка значения освещенности

**Function** *TracePath*(ray: Ray; depth : Integer) : Integer **is**

```
    if depth == MAX_DEPTH then
        | return Black;
    end

    hit ← RaySceneIntersection(ray)

    if not hit.exists then
        | return Black;
    end

    m ← hit.material

    if m.IsLight then
        | return Black;
    end

    lpos ← LightSample(light)
    shadow ← Visibility(hit.pos, lpos)
    R ← dist(hit.pos, lpos)
    sdir ← normalize(lpos − hit.pos)
    cosTheta1 ← −dot(sdir, light.norm)
    cosTheta2 ← dot(sdir, hit.norm)
    implicitP ← light.area * cosTheta1 * cosTheta2 / (π * R2)
    newRay.O ← hit.pos
    newRay.D ← RandomCosineVectorOf(hit.norm)
    BRDF ← m.reflectance
    explicitColor ← shadow * BRDF * light.emittance * implicitP
    return explicitColor + BRDF * TracePath(newRay, depth + 1) ;
end
```

**Алгоритм 4:** Трассировка путей с теневыми лучами.

Коэффициент *implicitP* получается при переходе от рассмотренной ранее сферической формы уравнения освещенности (уравнение 1.11) к площадной форме этого уравнения (рис 1.33, соотношение 1.21).

$$I(x, \psi) = \int_{\psi'} R(x, \psi') L(x, \psi') \frac{dA' \cos \theta \cos \theta'}{\|x - x'\|} \quad (1.21)$$

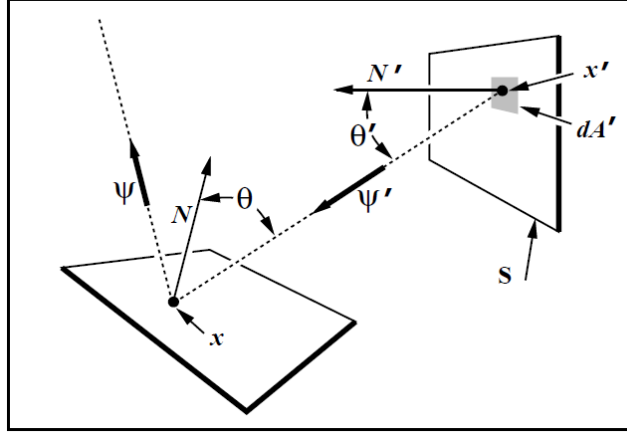


Рис. 1.33. Иллюстрация к площадной форме интеграла освещенности [22].

На рисунке 1.32 изображена ситуация, при которой случайный отраженный луч попадает в источник света. Такой луч необходимо терминировать и вернуть 0 в качестве яркости от источника света.

Реализация трассировки путей с теневыми лучами значительно быстрее сходится (рис. 1.35) и позволяет обрабатывать источники любых размеров. Однако, как нетрудно заметить (рис. 1.34), источник света стал черным и на изображении исчезают каустики. Первую проблему (черный источник света) легко исправить если возвращать цвет источника вместо черного при отсутствии диффузных и матовых (glossy) переотражений. Это всегда можно делать поскольку теневые лучи применяются только для учета матовой и диффузной компонент ДФО.

Вторая проблема - отсутствие каустик. Чтобы вернуть каустики, определим для начала в каких условиях они возникают. Каустик - яркое пятно на диф-

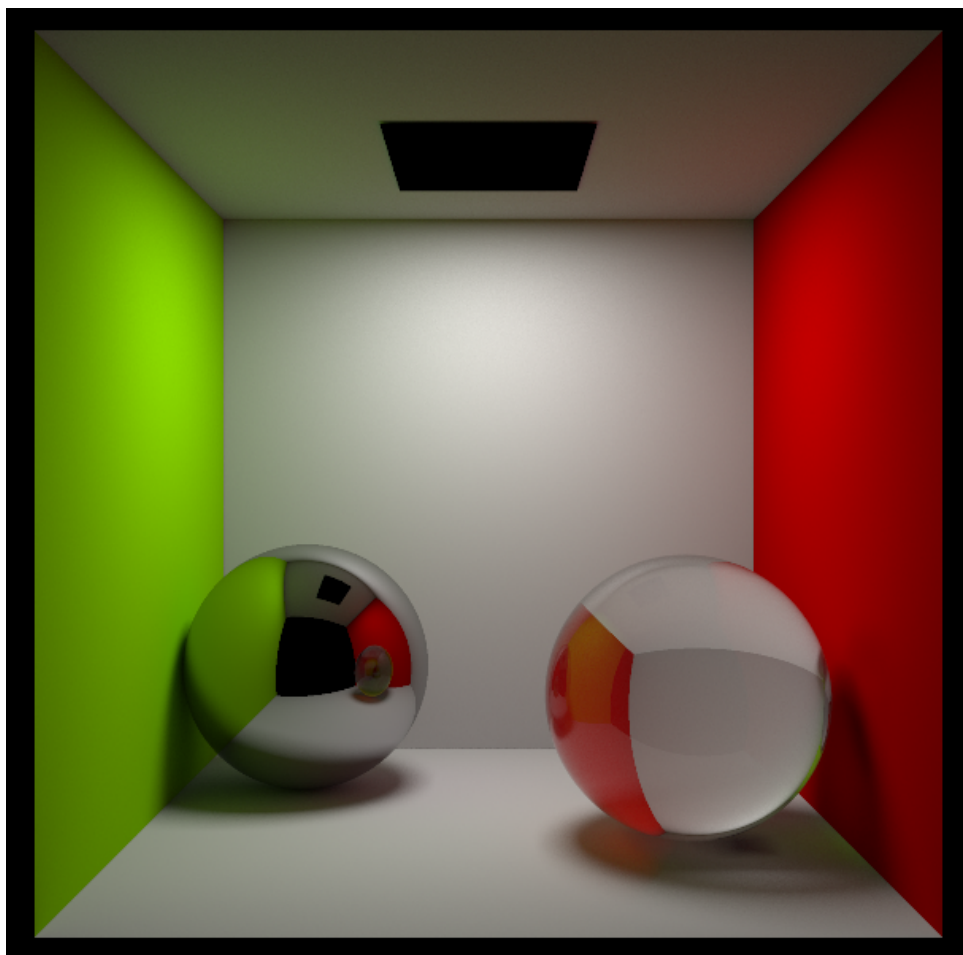


Рис. 1.34. Пример работы алгоритма трассировки путей с теневыми лучами (алгоритм 4) на классической сцене Cornell Box.

фузной поверхности, вызванное непрямым освещением посредством зеркальных отражений (или преломлений) света, излученного из источника. Таким образом, если в процессе трассировки путей после диффузного переотражения встретилось зеркальное, на месте упомянутого диффузного отражения может возникнуть каустик. Добавить каустики в трассировку путей с теневыми лучами, таким образом, достаточно просто:

1. Трассируем путь из глаза с применением теневых лучей до тех пор пока не встретилось зеркальное переотражение после диффузного.
2. Как только указанное условие выполнилось, превращаем оставшуюся часть пути в 'каустический путь'. Такой путь не использует теневые



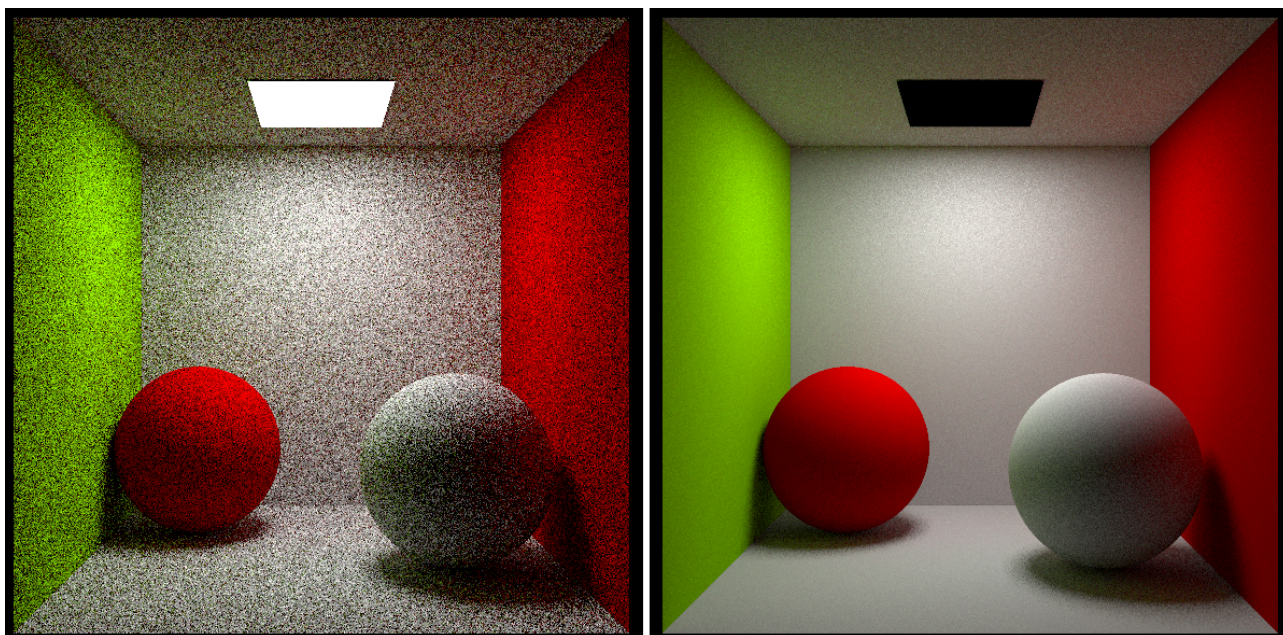


Рис. 1.35. Сравнение простого алгоритма трассировки путей (64 пути на пиксел, слева на изображении) и алгоритма трассировки путей использующего теневые лучи (32 пути на пиксел, справа на изображении). Оба изображения получены за примерно одинаковое время.

лучи и строит оставшуюся его часть руководствуется только формой ДФО.

В результате получем изображение на рисунке 1.30.

Рисунок 1.35 показывает преимущество трассировки путей с теневыми лучами над простой трассировкой путей, рассмотренной ранее. Поскольку трассировка путей без теневых лучей обрабатывает пути быстрее (в описанной выше реализации на основе алгоритмов 3 и 4 примерно в 2 раза), корректно сравнивать 64 пути на пиксел для трассировки путей без теневых лучей и 32 пути на пиксел с теневыми лучами.

#### 1.4.5. Две стратегии сэмплирования

Ранее были рассмотрены стратегиями сэмплирования (стратегиями генерации выборок) - сэмплирование по ДФО и сэмплирование по источнику света (рис.



1.31). Будем называть сэмплирование по ДФО неявной стратегией, а сэмплирование по источнику - явной.

Для того чтобы иметь корректный результат, мы могли использовать явную и неявную стратегии (превращая оставшуюся часть пути в 'каустический' путь), но не могли смешивать их. То есть если сгенерированный по неявной стратегии путь попадал в источник света из той-же точки, в которой уже был учтен теневой луч по явной стратегии, неявный луч приходилось исключать из расчетов. Однако, в этом случае эффективность расчета падает, особенно если неявных лучей попадающих в источник много. Далее рассмотрим, как можно смешивать явную и неявную стратегии чтобы учитывать их одновременно и избежать избыточных вычислений.

#### **1.4.6. Многократная выборка по значимости**

Для того чтобы корректно комбинировать явную и неявную стратегии, Эриком Вичем был предложен метод многократной выборки по значимости (англ. Multiple Importance Sampling, MIS) [23], который позволяет скомбинировать результаты нескольких способов выборки (нескольких стратегий сэмплирования) в одну несмещенную оценку. Для обратной трассировки путей, ламбертовского отражения и площадного источника света с равномерным распределением, схема многократной выборки по значимости будет иметь следующий вид:

$$p_i = \cos(\theta)/\pi$$

$$p_e = 1/A$$

$$w_e = \frac{p_e^\beta}{p_i^\beta + p_e^\beta}$$

$$w_i = \frac{p_i^\beta}{p_i^\beta + p_e^\beta}$$

$$\beta = 2$$

$\theta$  - угол между нормалью и отраженным лучом.  $p_i$  - плотность вероятности для неявной стратегии. Следует отметить - в данной формуле считается, что сэмплирование ДФО было произведено при помощи косинусоидального распределения для ламбертовского отражения (в формуле для  $p_i$  в числителе стоит косинус в первой степени). Если это не так, то  $p_i$  нужно вычислять исходя из того, как выбирался случайный отраженный луч.  $p_e$  - плотность вероятности для явной стратегии.  $A$  - площадь поверхности источника света.  $w_i$  - вес для учета яркости полученной по неявной стратегии.  $w_e$  - вес для учета яркости полученной по явной стратегии. Отметим, что  $p_e$  зависит от площади источника света, которая, в свою очередь, зависит от глобального масштаба сцены. Результат будет корректен если вклад теневого луча  $L_e$  учитывается как было описано ранее с использованием вероятности *implicitP*. В этом случае, изменение масштаба не повлияет на результат, поскольку площадь сократится в формуле [1.22](#).

Для того чтобы реализовать многократную выборку по значимости нужно отдельно хранить яркость пришедшую от источника света в направлении теневого луча ( $L_i$ ) и в случае если отраженный луч попал в источник света (и дал яркость  $L_e$ ) результирующая яркость для данного  $k$ -ой выборки должна быть вычислена по формуле [1.22](#):

$$L_k = w_e \frac{L_e}{p_e} + w_i \frac{L_i}{p_i} \quad (1.22)$$

Оптимальность такого взвешивания показана в [23]. На рис 1.36 показано сравнение рассмотренной ранее простой трассировки путей и трассировки путей с применением MIS. В действительности, на сцене из рисунка 1.36 комбинирование стратегий (то есть учет либо, одной либо другой стратегии, но не обоих сразу), дало бы практически неотличимый от правой картинке (на рис. 1.36) изображение. Ситуация, в которой многократная выборка по значимости действительно дает существенный выигрыш изображена на рис 1.37.

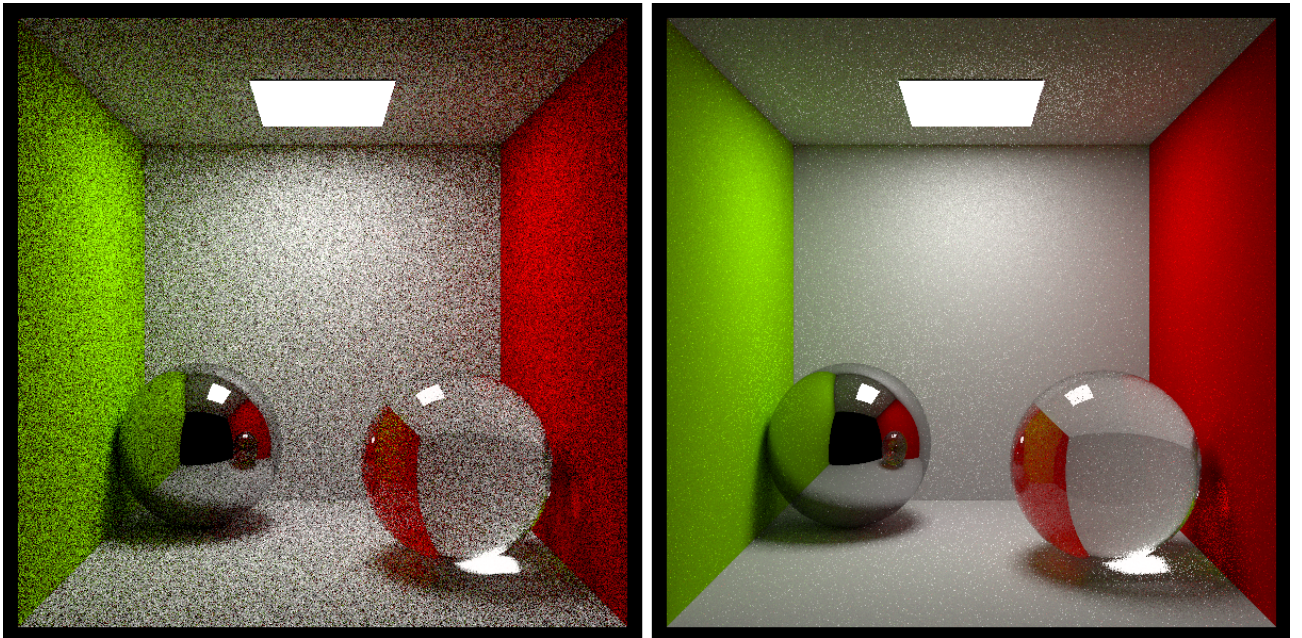


Рис. 1.36. Сравнение простого алгоритма трассировки путей (64 пути на пиксел, слева на изображении) и алгоритма трассировки путей использующего многократную выборку по значимости (32 пути на пиксел, справа на изображении). Обе картинке получены за примерно одинаковое время.

#### 1.4.7. Русская рулетка и глубина трассировки

Одной из причин низкой эффективности трассировки путей путей может являться завышенное значение максимальной глубины трассировки. Стеклян-

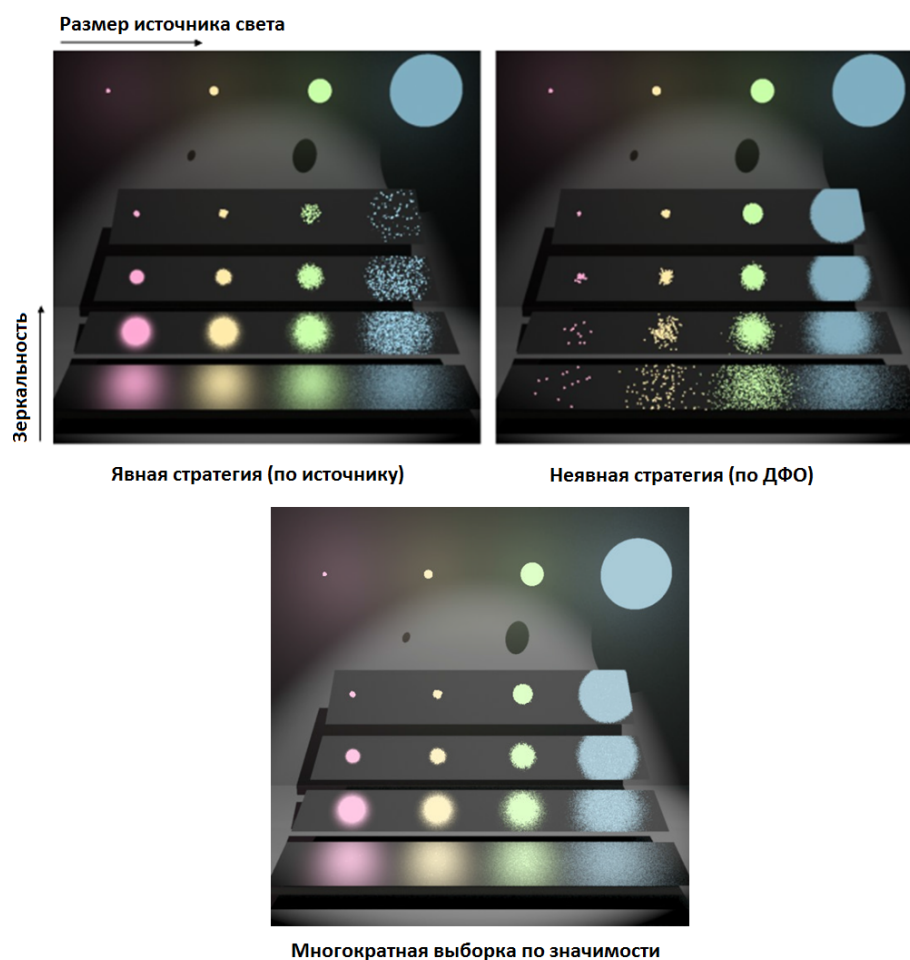


Рис. 1.37. Сравнение различных стратегий генерации выборки для сцены с матовым (glossy) отражением. Явная стратегия дает лучший результат, если источник находится далеко от поверхности или имеет малый размер. Неявная стратегия дает лучший результат, если источник имеет большой размер или находится близко к поверхности. Многократная выборка по значимости использует преимущества обоих подходов.

ные объекты могут требовать значительного числа переотражений для корректной визуализации. В то же время, лучи попавшие на диффузные объекты, как правило, уже после нескольких переотражений теряют значимость. Одним из простых решений является терминирование пути при достижении определенного числа диффузных отскоков и/или низкого 'коэффициента пропускания' (произведение коэффициентов отражения во всех точках пути). Параметр терминирования при этом требует ручной настройки. Однако, такой метод вносит смещение в получаемое решение и может дать неверный

результат на сцене с очень яркими источниками и сложным перераспределением световой энергии [24].

Одно из решений, позволяющих автоматически вбирать глубину терминирования пути - стохастическое терминирования пути на основе механизма русской рулетки (алгоритм 5). Фактически, русская рулетка позволяет трассировать меньше путей на большой глубине. Однако, она присваивает им большую значимость, за счет чего в пределе получается верный результат.

**Исходные параметры:** ray - Луч, depth - глубина рекурсии

**Результат:** Монте-Карло выборка значения освещенности

**Function** *TracePath*(ray: Ray; depth : Integer) : Integer **is**

*pabsorb*  $\leftarrow$  *terminationProbability*(depth, ...)

**if** *random*(0.0, 1.0) < *pabsorb* **then**

        return Black;

**end**

    ...;

    return result\*(1/(1-pabsorb)) ;

**end**

**Алгоритм 5:** Трассировка путей с русской рулеткой.

Механизм русской рулетки, вообще говоря, увеличивает дисперсию в оцениваемом решении [25], поэтому должен применяться осторожно - только тогда когда ожидаемый вклад от текущего пути достаточно мал. В этом случае:

1. Дисперсия повысится незначительно, так как в среднем вклад от путей, для которых применяется русская рулетка относительно мал.
2. По сравнению с простым терминированием время трассировки не увеличится, поскольку число лучей трассируемых на большую глубину невелико.
3. Решение станет более точным за счет учета большего числа переотражений.

#### 1.4.8. Резюме по обратной трассировке путей

Рассмотренный алгоритм обратной трассировки путей с применением теневых лучей и многократной выборки по значимости является несмещенным и достаточно эффективным методом для вычисления интеграла освещенности; за исключением каустик, которые вычисляются этим алгоритмом довольно медленно. Для эффективного вычисления каустик далее будет рассмотрен алгоритмы прямой трассировки путей (Light Tracing) и алгоритм фотонных карт.

#### 1.4.9. Грамматика путей

Для классификации различных ситуаций, возникающих в процессе трассировки путей часто используют понятие грамматики путей [26]. Этот способ классифицирует пути при помощи строк и регулярных выражений. Каждый символ в строке обозначает определенное событие, произошедшее с лучом (путем) в процессе трассировки.

1. S - (Specular); зеркальное отражение/преломление
2. D - (Diffuse); диффузное отражение
3. G - (Glossy); матовое отражение/преломление
4. V - (Volume); объемное рассеивание
5. L - (Light); источник света
6. E - (Eye); глаз

При этом под символами S и G понимается не только отражение но и преломление света. Грамматика удобна не только для описания различных ситуаций, возникающих в трассировке путей, но она позволяет также классифицировать видимые эффекты, однозначно ставя им в соответствие классы

путей которые эти эффекты вызывают (для упрощения изложения материала объемное рассеивание  $V$  мы далее не рассматриваем). Например,  $EDL$  - прямое освещение диффузной поверхности.  $E(S|G)^+L$  - яркий блик.  $EDSL$  - каустик, видимый из камеры напрямую, обусловленный одним зеркальным переотражением света (например солнечный зайчик от зеркала).  $EDS^+L$  - каустик, видимый из камеры напрямую, вызванный одним или более зеркальным переотражением.  $ES^+DS^+L$  - каустик, видимый через стекло или зеркало, вызванный одним или более зеркальным переотражением.

При помощи грамматики путей удобно классифицировать алгоритмы по типу освещения, которые они способны рассчитывать. Например, Трассировка лучей Уиттеда, рассмотренная нами в самом начале, строит пути  $ES^*L$ . Обратная трассировка путей строит пути вида  $E(S|D|G)^*L$ .

## 1.5. Прямая трассировка (Light Tracing)

В противоположность обратной трассировки путей, рассмотренной выше, алгоритм прямой трассировки путей начинает свою работу из источника света и строит пути вида  $L(S|D|G)^*E$ . При каждом соударении с поверхностью создается аналог теневого луча, направленный в камеру. Если луч успешно доходит до камеры, значение яркости записывается в пиксель, соответствующий проекции начала луча на экранную плоскость. Для реализации операции проекции вы можете использовать матрицу  $mProj$ , которая была рассмотрена нами в разделе "Генерация луча".

Пути создаваемые на источнике должны быть сгенерированы в соответствии с распределением световой энергии для источника света (рис. 1.38). Обратите внимание, что если каждая точка поверхности излучает свет равномерно во все стороны (например площадный источник из рис. 1.30), то исходя из геометрических соображений и применяя выборку по значимости, необходимо

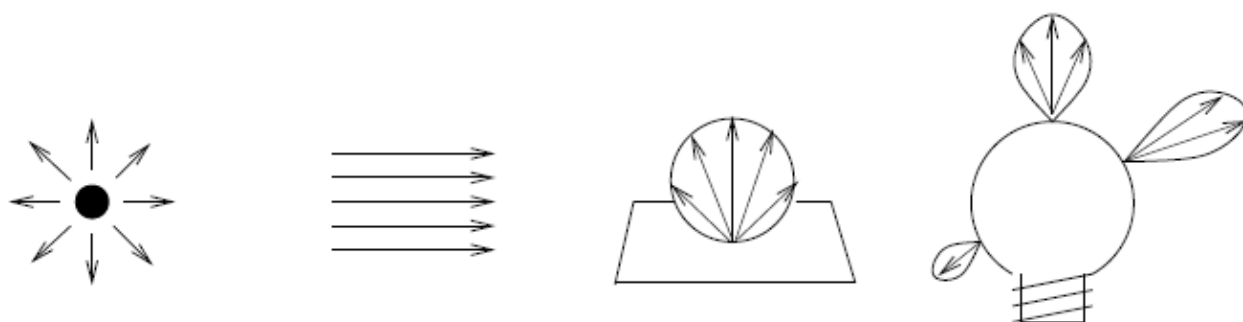


Рис. 1.38. Иллюстрация различных видов распределения световой энергии для различных источников света.

использовать косинусоидальное распределение для расчета направления пути точно так же, как мы вычисляли ранее диффузное отражение. В обратной трассировке путей данный косинус получался стохастически, за счет того что число лучей попавших в источник зависит от направления, с которого они были выпущены.

Существующий миф о крайне низкой скорости прямой трассировки путей (или лучей) далеко не всегда соответствует действительности. Такие эффекты как каустики чрезвычайно быстро могут быть рассчитаны именно при помощи прямой трассировки. В связи с этим, возникает вопрос о том, можно ли комбинировать прямую и обратную трассировки путей для того, чтобы исключить недостатки обоих алгоритмов.

Метод, реализующий эту идею существует и называется 'усеченная двунаправленная трассировка путей' [24]. Идея этого метода в упрощенном представлении изложена ниже:

1. Будем производить обратную трассировку путей с теньевыми лучами. При этом отключим расчет каустиков. Результат сохраняем в изображение с номером 1. На изображении 1 мы получим картинку без каустиков.
2. Будем производить прямую трассировку из источника света, но на этот



раз будем сохранять вклад только от тех путей, для которых первое переотражение было зеркальным. Результат сохраняем в изображение с номером 2. На изображении 2 мы получим только каустики.

3. Финальное изображение можно получить простым сложением изображений 1 и 2.

Усеченная двунаправленная трассировка путей дает почти корректный результат, поскольку изображения 1 и 2 будут нести в себе освещение, обусловленное различными типами переотражений. Слово 'почти' означает, что при помощи такого метода не удастся эффективно рассчитать каустики, не видимые камерой напрямую. То есть не удастся эффективно вычислять освещение, вызванное путями вида  $ES^+DS^+L$ . Время расчета таких эффектов будет сопоставимо со временем их расчета обыкновенной трассировкой путей.

## 1.6. Двунаправленная трассировка путей

Идея объединить преимущества прямой и обратной трассировки привела к созданию алгоритма двунправленной трассировки путей (Bidirectional Path Tracing, BDPT) [23]. Основная идея двунаправленной трассировки путей заключается в том, чтобы соединять тeneвыми лучами не только источник с точками пути (как в обратной трассировке) или камеру с точками пути (как в прямой трассировке), но и различные точки путей друг с другом (рис. 1.39). Двунаправленная трассировка путей обладает лучшей сходимостью (по сравнению с прямой или обратной трассировкой) на сценах со сложными условиями освещения (преобладающее вторичное освещение) только при реализации многократной выборки по значимости. В противном случае, изображение получается сильно зашумленным. Однако даже при реализации многократной выборки по значимости, двунаправленная трассировка путей по прежнему

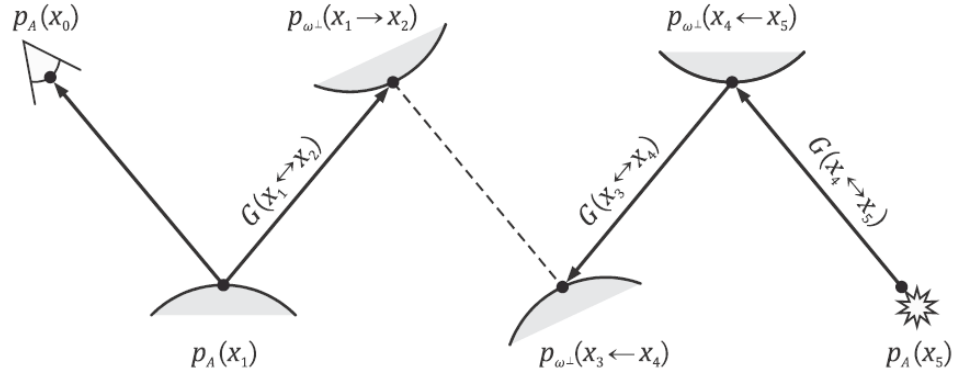


Рис. 1.39. Идея двунаправленной трассировки путей.

не позволяет эффективно рассчитывать пути вида  $ES^+DS^+L$  (каустики, видимые через зеркало или стекло). Такие пути могут быть эффективно вычислены при помощи фотонных карт, переноса света Метрополиса и метода соединения вершин (vertex merging)[27].

Поскольку теневые лучи в двунаправленной трассировке могут соединять произвольные точки различных путей, реализация многократной выборки по значимости значительно усложняется. Если ранее для вычисления весов достаточно было рассмотреть только 2 стратегии, то теперь число рассматриваемых вариантов значительно увеличивается. Рассмотрим путь на рисунке 1.39.

### 1.6.1. Многократная выборка по значимости в BDPT

Рассмотрим некоторый путь  $x_0..x_5$ . Для такой последовательности точек существует несколько способов соединения их теньвым лучом (обозначено вертикальной чертой):

$$\begin{array}{c}
x_0 \mid x_1 x_2 x_3 x_4 x_5 \\
x_0 x_1 \mid x_2 x_3 x_4 x_5 \\
x_0 x_1 x_2 \mid x_3 x_4 x_5 \\
x_0 x_1 x_2 x_3 \mid x_4 x_5 \\
x_0 x_1 x_2 x_3 x_4 \mid x_5
\end{array}$$

При вычислении весов для какого-либо из случаев, многократная выборка по значимости должна учитывать все остальные случаи. Формула 1.23 применяется для вычисления веса конкретного пути.

$$w_i = \frac{p_i^\beta}{\sum_{j=1}^N p_j^\beta} \quad (1.23)$$

Например, для случая  $x_0 x_1 x_2 | x_3 x_4 x_5$   $i$  равно 2;  $N$  равняется 5.  $p_i$  - плотность вероятности распределения отраженного луча для точки с номером 2. Таким образом, для указанного случая из 6 точек  $x_0..x_5$  необходимо просчитать всевозможные пути (всего 5) и яркость  $k$ -ого сэмпла может быть вычислена по формуле 1.24 ( $N = 5$ ):

$$L_k = \sum_{j=1}^N w_j \frac{L_j}{p_j} \quad (1.24)$$

## 1.7. Перенос света Метрополиса (MLT)

Трассировка путей обладает достаточно низкой скоростью на сценах со сложными условиями освещения. Сценами со сложными условиями освещения будем называть 2 вида сцен:

1. Сцены, на которых преобладает вторичное освещение вызванное узкими и яркими световыми пятнами (рис. 1.40).
2. Сцены, на которых  $ES^+DS^+L$  пути вносят значительный вклад (рис. 1.41).

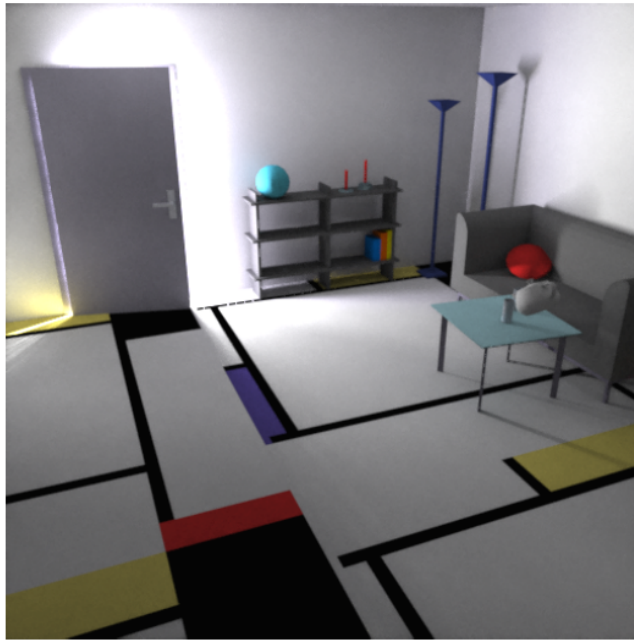


Рис. 1.40. Пример сцены со сложными условиями освещения.

проблема на таких сценах заключается в том, что в большинстве мест на сцене функция падающего освещения имеет один или более резких максимумов. Для того чтобы интеграл от такой функции вычислить с высокой точностью методом Монте-Карло с равномерным распределением случайной величины, нужно большое число выборок.

Для улучшения скорости сходимости на таких сценах Э. Вичем и Л. Гибсом был разработан алгоритм переноса света Метрополиса (Metropolis Light Transport, MLT) [28]. Идея этого метода аналогична применению выборки по значимости к под-интегральному выражению 1.25, однако имеет совершенно иную природу.

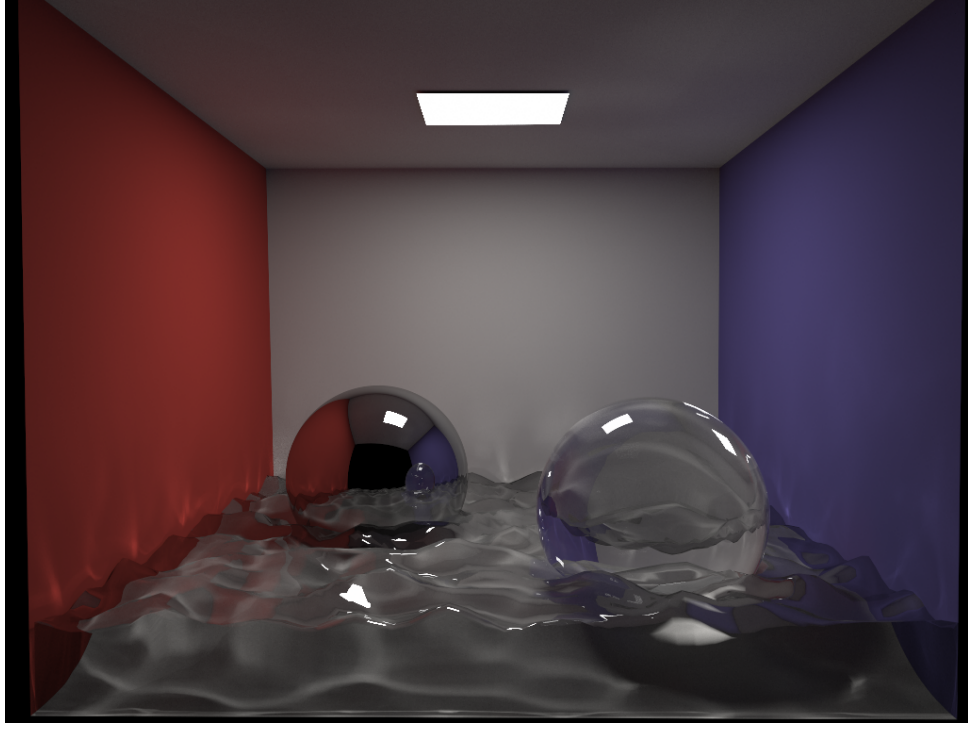


Рис. 1.41. Пример сцены с  $ES^+DS^+L$  путями. Каустики, под водой (рис. 1.40).

$$L(\phi_i, \theta_i)R(\phi_i, \theta_i, \phi_r, \theta_r)\cos(n, l_{\phi_i, \theta_i}) \quad (1.25)$$

Рассмотрим Метод Монте-Карло в общем виде (формула 1.26).

$$\int_a^b f(x)dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(u_i)}{p(u_i)} \quad (1.26)$$

Если бы мы имели функцию распределения  $p(x)$  пропорциональную  $f(x)$ , тогда для некоторой константы  $c$  можно записать  $p(x) = cf(x)$ . Подставив константу в выражение для интеграла 1.26, получим оценку для константы:

$$c = \frac{1}{\int_a^b f(x)dx} \quad (1.27)$$

Таким образом, мы имеем возможность оценивать константу  $c$  взяв, например, выборки с равномерным распределением и применив метод монте-карло.

Алгоритм Метрополиса отвечает за оставшуюся часть задачи - создание  $p(x)$  пропорционально  $f(x)$ . Это делается следующим образом:

1. Пусть  $X$  - начальный вектор случайных параметров, которые были использованы при вычислении  $P_i$ .
2. Сгенерируем путь  $P_{i+1}$  путем мутаций случайных параметров  $X$ . Т.е.  $Y = mutate(X)$ .
3. Вычислим некоторым специальным образом вероятность мутации  $T(X|Y) = p(X \rightarrow Y)$  и вероятность обратной мутации  $T(Y|X) = p(Y \rightarrow X)$ .
4. Вычислим вероятность принятия мутации  $X \rightarrow Y$  по формуле 1.28.

$$a(Y|X) = \min\{1, \frac{f(y)T(X|Y)}{f(x)T(Y|X)}\} \quad (1.28)$$

Обоснование того почему выше-описанный алгоритм работает можно найти в [28]. Важным свойством алгоритма Метрополиса является то, что он строит  $p(x)$  даже если значения  $f(x)$  могут быть вычислены только стохастически, при помощи выборок. Это именно та ситуация, которая присутствует в трассировке путей. Наиболее сложная часть в алгоритме переноса света Метрополиса - выбор хорошей стратегии мутации и вычисление вероятности мутаций. За подробностями реализации алгоритма MLT рекомендуется обратиться к [29] и [28].

## 1.8. Фотонные карты

Рассмотренные ранее несмещенные методы на основе трассировки путей работают в терминах яркости. Каждый путь в них отвечал за перенос яркости

от источника света к камере. Метод фотонных карт работает в терминах потока. Идея этого метода заключается в том, чтобы вычислить распределение световой энергии по сцене при помощи трассировки множества частиц, переносящих порцию световой энергии. После чего можно оценивать интеграл освещенности, выполняя на поверхности поиск ближайших фотонов. Рассмотрим интеграл освещенности (формула 1.29). Выразим функцию падающей освещенности в терминах потока. Подставим  $L(\phi_i, \theta_i)$  в интеграл и получим:

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i \quad (1.29)$$

$$L(\phi_i, \theta_i) = \frac{d^2\Phi}{\cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i dA} \quad (1.30)$$

$$(1.31)$$

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} \frac{d^2\Phi}{\cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i dA} R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i \quad (1.32)$$

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} \frac{d^2\Phi}{dA} R(\phi_i, \theta_i, \phi_r, \theta_r) \quad (1.33)$$

Дифференциальный поток можно аппроксимировать суммой энергий фотонов в окрестности. Таким образом, в результате преобразований получаем сумму К ближайших фотонов.

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} \frac{d^2\Phi}{dA} R(\phi_i, \theta_i, \phi_r, \theta_r) \quad (1.34)$$

$$I(\phi_r, \theta_r) \approx \sum_{i=1}^K R(\phi_i, \theta_i, \phi_r, \theta_r) \frac{\Delta\Phi(\phi_i, \theta_i)}{\Delta A} = \frac{1}{\pi r^2} \sum_{i=1}^K R(\phi_i, \theta_i, \phi_r, \theta_r) \Delta\Phi(\phi_i, \theta_i) \quad (1.35)$$

В выражении 1.34 переменная  $r$  отвечает за радиус диска на поверхности, в

пределах которого выполняется поиск ближайших фотонов. Метод фотонных карт, таким образом, состоит из 4 шагов:

1. Испускание фотонов из источников света
2. Трассировка фотонов
3. Построение фотонной карты
4. Сбор освещенности

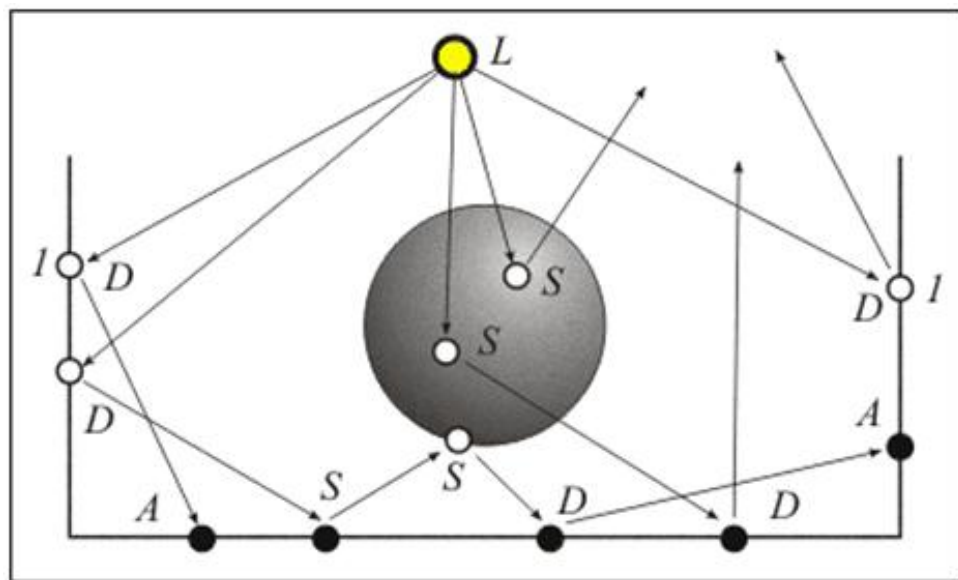


Рис. 1.42. Процесс трассировки и сохранения фотонов на поверхностях.

**Испускание фотонов.** Фотоны в данном методе – это частицы, переносящие некоторую небольшую порцию световой энергии. На начальном этапе фотоны испускаются из источника света в соответствии с распределением световой энергии для данного источника. Например известно, что точечный или сферический источник света (такой как солнце) испускают свет равномерно во всех направлениях. Площадные источники света имеют косинусоидальное



распределение, имеющее максимум по направлению, совпадающему с нормалью к плоскости источника. Стадия испускания фотонов не отличается от стадии испускания путей вида  $L(S|D|G)^*E$  в прямой трассировке путей.

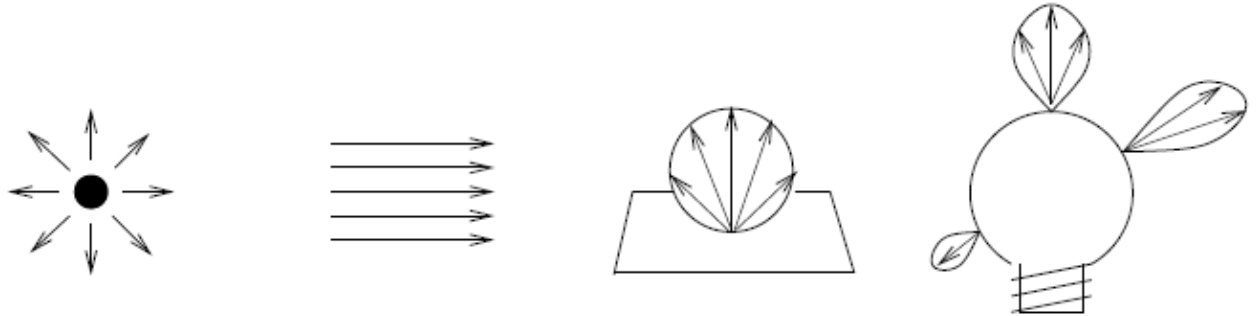


Рис. 1.43. Иллюстрация различных видов распределения световой энергии для различных источников света.

**Трассировка фотонов. Русская рулетка.** Трассировка фотонов происходит похожим на прямую трассировку путей образом за исключением одного важного отличия - механизма русской рулетки (этот механизм можно также применять и в трассировке путей [24], но исторически он появился именно в методе фотонной карты) [30]. Легче всего продемонстрировать механизм русской рулетки на примере. Допустим мы имеем ламбертовский материал с коэффициентом отражения 0.25 и фотон падающий на эту поверхность с энергией равной 1. Очевидным решением является модифицировать энергию фотона, умножив ее на 0.25 и продолжить трассировку фотона. Механизм русской рулетки поступает по другому. Вместо того чтобы модифицировать энергию фотона, его энергия сохраняется, но с вероятностью 0.25 фотон отражается и продолжает трассировку, а с вероятностью 0.75 - погибает (рис. 1.44). Таким образом, если на поверхность падает 1000 фотонов, 250 отразится, сохранив энергию, а остальные 750 - погибнут. За счет такого механизма повышается точность решения, поскольку в местах с высокой освещенностью

будет сохраняться больше фотонов, чем в местах с низкой освещенностью; в неосвещенных областях фотоны будут отсутствовать. Нетрудно заметить, что



Рис. 1.44. Русская рулетка.

при использовании русской рулетки в чистом виде все фотоны будут иметь единичную энергию. То есть, энергию фотона хранить не обязательно, а для того чтобы оценить значение освещенности, достаточно либо посчитать число фотонов в заданном радиусе сбора, либо найти радиус, в котором лежит  $K$  ближайших фотонов. Мы вернемся к этому вопросу позже, когда будем рассматривать процесс сбора освещенности. При этом для того чтобы выполнять правильно сбор освещенности, нам потребуется хранить позицию фотона и нормаль к поверхности в точке удара о поверхность.

**Сохранение фотонов в фотонной карте.** Вопрос сохранения фотонов на поверхностях (в фотонной карте) заслуживает отдельного рассмотрения. Фотоны сохраняются на всех диффузных (ламбертовых) поверхностях, которые они ударяют. Как правило фотоны используются только при вычислении части интеграла, обусловленной диффузной компонентой. Зеркальная компо-

нента не может быть вычислена при помощи фотонов по той же причине, по которой прямая трассировка путей (Ligh Tracing) отображает зеркала и стекла черными: вероятность встретить фотон (или путь от источника), отразившийся в строго заданном направлении стремится к нулю. Из этого следует, что компонента, обусловленная матовым отражением (glossy reflection) все же может быть вычислена при помощи фотонных карт, однако эффективность такого расчета будет тем ниже, чем более зеркальный характер приобретает ДФО. Следует подчеркнуть, что фотон в течении трассировки может быть сохранен несколько раз - на всех диффузных поверхностях, которые он ударяет.

Итак, Фотоны сохраняются на всех диффузных поверхностях, которые они ударяют. Однако, одним из часто-применяемых трюков является в буквальном смысле 'пропуск' первого отскока для процедуры сохранения фотона. Фотоны в этом случае сохраняются только после первого отскока а освещение, получаемое в результате интегрирования фотонной карты - целиком вторичное. Этот метод значительно снижает число фотонов, необходимое для расчета вторичного освещения. Первичное освещение гораздо быстрее вычисляется при помощи трассировки лучей/путей, поэтому данный прием имеет важное значение для ускорения процесса рендеринга.

**Построение фотонной карты.** В самом простом случае можно хранить фотонну карту в виде обыкновенного массива. Однако это не позволит в последствие выполнять эффективный поиск ближайших фотонов. Поэтому под построением фотонной карты мы будем понимать построение ускоряющей структуры, позволяющей выполнять эффективный поиск ближайших фотонов. Существует довольно большое число способов, как делать это эффективно. Способы могут различаться для поиска k-ближайших или всех фотонов в заданном радиусе. Среди наиболее эффективны структур следует отметить

пространственные хэш-таблицы и деревья, построенные при помощи эвристики VVH [31] аналогично эвристике SAH, рассмотренной нами ранее.

**Сбор освещенности.** После того как фотонная карта построена, наступает стадия сбора освещенности. Из виртуальной камеры испускаются лучи и выполняется обратная или стохастическая трассировка лучей (или трассировка путей). В местах соударений луча и поверхности производится сбор освещенности. Причем, для стохастической трассировки лучей (или путей), диффузные переотражения не учитываются. Компонента, обусловленная диффузными переотражениями получается при интегрировании фотонной карты (формула 1.36). Диффузные переотражения в процессе обратной трассировки лучей/путей вычисляются при использовании метода Финального Сбора, который мы рассмотрим позже.

$$I(\phi_r, \theta_r) \approx \frac{1}{\pi r^2} \sum_{i=1}^K R(\phi_i, \theta_i, \phi_r, \theta_r) \Delta\Phi(\phi_i, \theta_i) \quad (1.36)$$

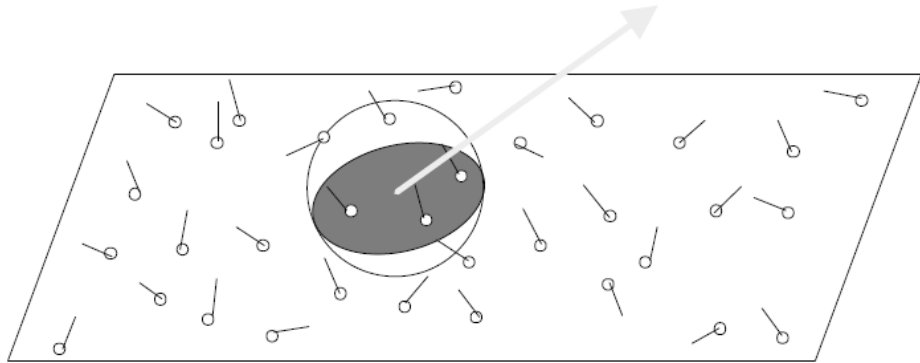


Рис. 1.45. Иллюстрация к формуле 1.36. Сбор освещенности или интегрирование фотонной карты. В данной формуле  $r$  - радиус диска, полученного при проекции сферы на поверхность.  $\pi r^2$  - площадь диска.

Существует 2 основные стратегии сбора освещенности:

1. Сбор  $k$ -ближайших фотонов (динамический радиус сбора), в соответствии с формулой 1.36. Рисунок 1.46.
2. Сбор всех фотонов в заданном радиусе (фиксированный радиус сбора). Данная стратегия упрощает процесс сбора и дает корректный результат в пределе (при стремлении общего числа фотонов к бесконечности), однако не обеспечивает постоянное значение  $K$  в формуле 1.36.

Поиск ближайших фотонов, как правило, использует специальную структуру 'max-heap' для упорядочивания списка фотонов [30]. Однако, поиск  $k$ -ближайших включает в своей основе более простой поиск всех фотонов в заданном радиусе, поскольку даже при поиске  $k$ -ближайших необходимо начинать с некоторого радиуса и уметь находить все фотоны в нем.

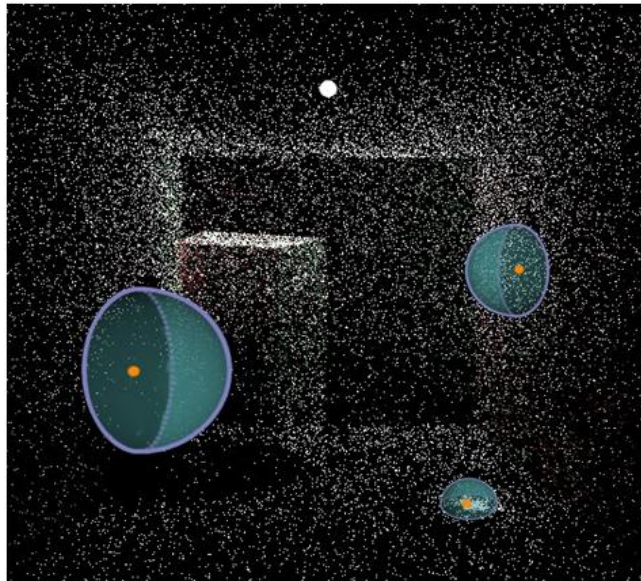


Рис. 1.46. Сбор  $k$ -ближайших фотонов. Радиус сбора выбирается динамически (чтобы ограничить ровно  $k$  фотонов) в зависимости от плотности фотонной карты.

### 1.8.1. Финальный сбор

Один из артефактов, возникающих при реализации сбора освещенности - темные края поверхностей (рис 1.48, 1.49). Данный артефакт возникает по

причине того, что на границах поверхностей освещение собирается только с половины диска. При этом площадь, вычисляемая как  $\pi r^2$  для целого диска остается прежней. Один из способов, позволяющий убрать темные края заключается в том, чтобы найти реальную площадь элемента поверхности на котором происходил сбор при помощи вычисления площади пересечения сферы сбора и всех треугольников с нормалью, близкой к нормали в точке сбора (то есть вычислить честную проекцию сферы на поверхность). Вторым методом называется Финальный Сбор (Final Gathering, FG). Вместо непосредственного сбора освещенности с фотонов, в методе финального сбора из заданной точки испускается некоторое число лучей по полусфере и освещенность собирается уже в тех местах, куда попали лучи (рисунок 1.47).

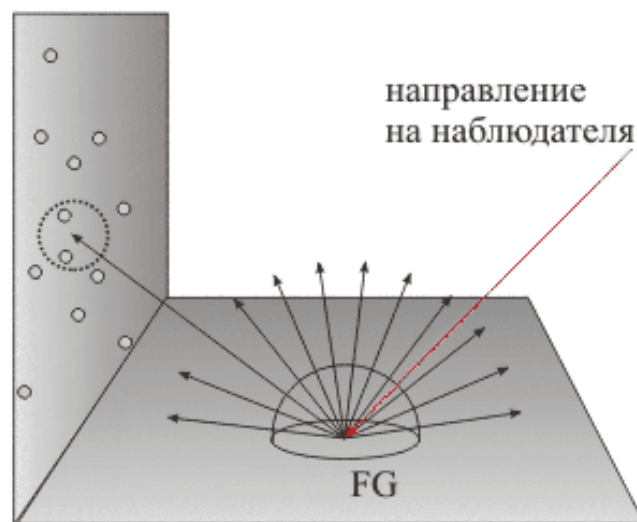


Рис. 1.47. Иллюстрация метода финального сбора.

При использовании финального сбора сами фотонные карты, таким образом, задействуются только для вычисления аппроксимации вторичной освещенности. Помимо избавления от темных краев, финальный сбор значительно повышает точность решения, позволяя обрабатывать значительно число фотонов. Хотя скорость финального сбора в среднем сравнима со скоростью обычной трассировки путей, финальный сбор, как правило, дает меньше шума (особенно для сцен со сложными условиями освещения) чем обычная трассировка

путей. Метод финального сбора наиболее часто применяется совместно с кэшем освещенности (будет рассмотрен рассморим далее), поскольку для кэша освещенности важна даже не столько точность решения, сколько отсутствие шума в нем.

**Карты светимости (radiosity maps).** Процесс поиска ближайших фотонов является, как правило, узким местом финального сбора. Поскольку в данном случае при помощи самих фотонных карт достаточно получить лишь грубую оценку освещенности, существуют различные методы ускорения финального сбора, использующие этот факт. К таким методам относятся карты светимости [32] или октанные текстуры [33]. Их смысл заключается в том, чтобы предрасчитать светимость в точках поверхности [32] (или в объеме [33]) и запомнить ее в виде карты светимости - двумерной или трехмерной (возможно разряженной) текстуры. При попадании луча финального сбора в определенную точку поверхности поиск ближайших фотонов не выполняется. Вместо этого производится быстрая выборка значения из карты светимости.

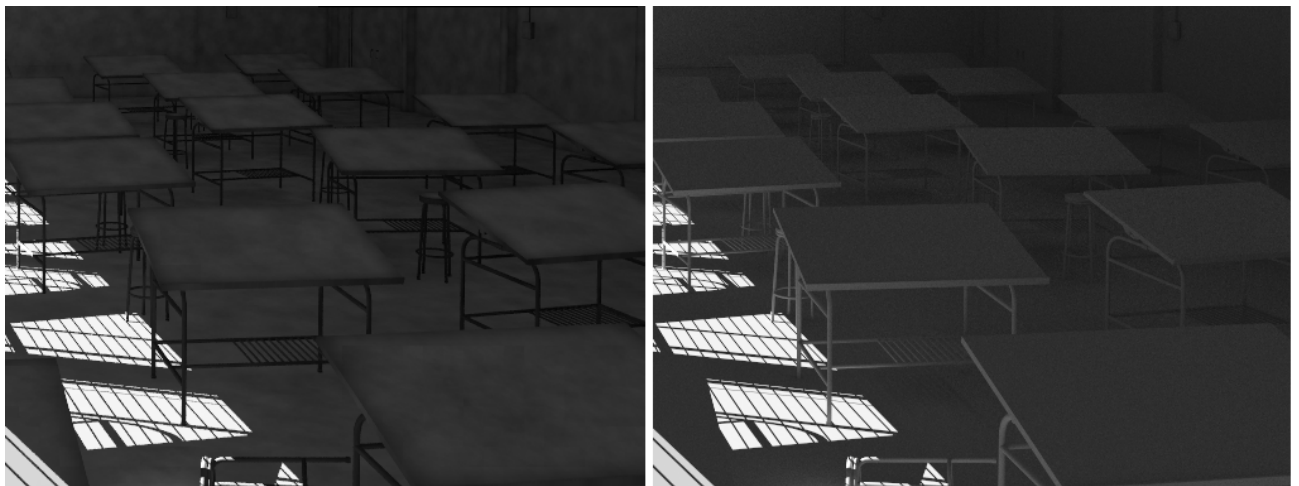


Рис. 1.48. Сбор освещенности с 10М фотонов (слева). Финальный сбор с 1М фотонов (справа).

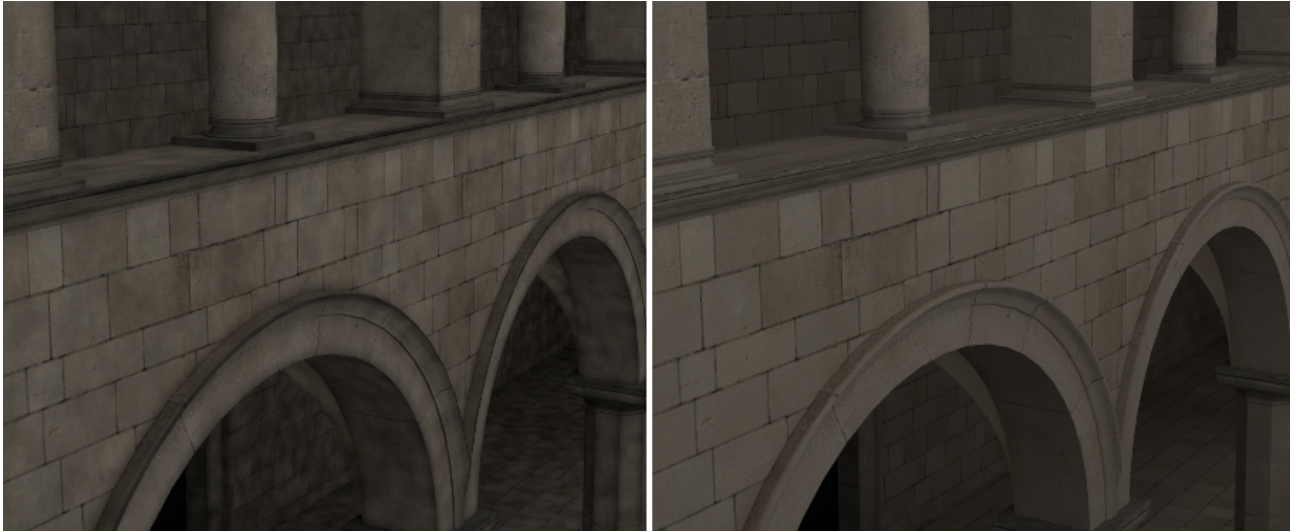


Рис. 1.49. Сбор освещенности с 10М фотонов (слева). Финальный сбор с 1М фотонов (справа).

### 1.8.2. Прогрессивные фотонные карты

Для фотонных карт в классической реализации серьезной проблемой является объем потребляемой памяти. Для того, чтобы получить изображение высокого качества может потребоваться до нескольких миллиардов фотонов. Хранить все фотоны в памяти в этом случае не представляется разумным, поскольку даже если хранить фотоны в соответствии с компактным представлением в 16 байт, один миллиард фотонов займет  $16 * 10^9 \approx 15Gb$ .

Для того чтобы иметь возможность обрабатывать произвольное число фотонов имея фиксированный объем памяти был разработан метод прогрессивных фотонных карт [34]. Основная идея метода прогрессивных фотонных карт заключается в том, чтобы повторять все 4 шага (испускание фотонов из источника света, трассировка, построение фотонной карты и сбор освещенности) несколько раз для порций фотонов фиксированного размера (например, по 1 миллиону фотонов). При этом, при обработке каждой следующей порции фотонов радиус сбора и аккумулируемый поток уменьшаются в соответствии с формулой 1.37. Такой метод называется Прогрессивными фотонными картами (Progressive Photon Mapping, PPM [34]). Если выбрана стратегия с фикси-



рованным радиусом сбора, то для каждой следующей порции фотнов радиус сбора уменьшается в соответствие с соотношением 1.38. Такой алгоритм называется Стохастическими Прогрессивными Фотонными Картами (Stochastic Progressive Photon Mapping, SPPM [35], [36]). Параметр  $\alpha$  варьируется в пределах от 0 до 1. Рекомендуемое значение - больше или равно  $2/3$ .

$$\frac{r_{i+1}^2}{r_i^2} = \frac{N_i + \alpha M_i}{N_i + M_i} \quad (1.37)$$

$$\frac{r_{i+1}^2}{r_i^2} = \frac{i + \alpha}{i + 1} \quad (1.38)$$

В соотношении 1.37  $N_i$  - число уже собранных фотонов.  $M_i$  - число вновь добавленных фотонов (на очередном проходе).

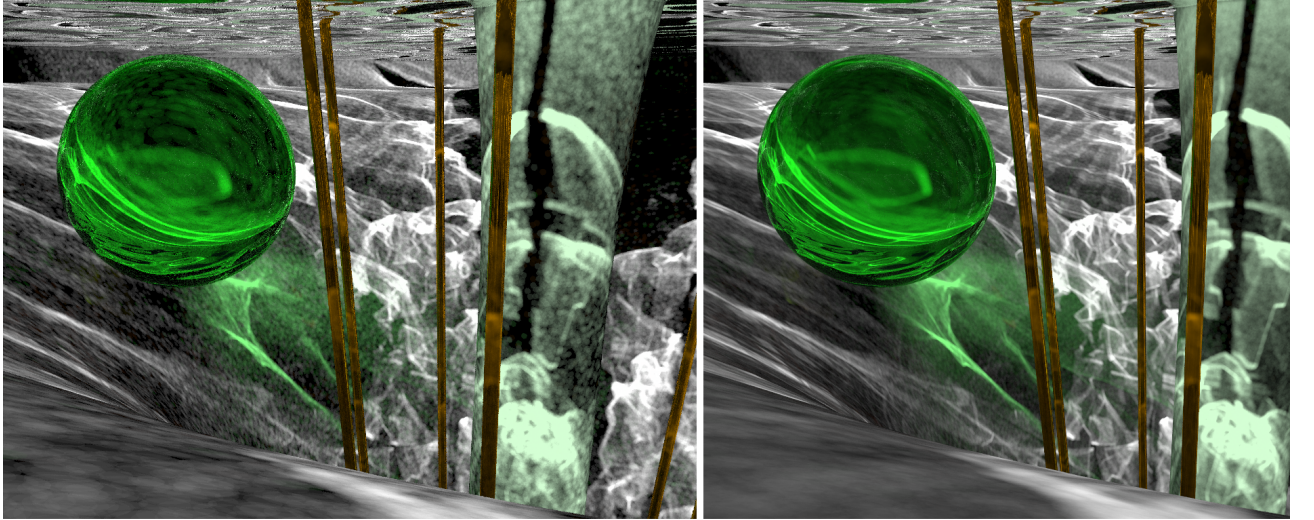


Рис. 1.50. Стохастические прогрессивные фотонные карты. 1М фотонов слева. 10М (10 проходов) - справа.

Метод стохастических прогрессивных карт значительно проще в реализации, чем обыкновенные прогрессивные фотонные карты, поскольку он не фиксирует точки сбора (что необходимо делать в методе прогрессивных фотонных

карт)[35], [36]. Причем, в [35] было показано, что стохастические прогрессивные фотонные карты обладают лучшей сходимостью. Результаты для различных порций фотонов могут быть скомбинированы простым усреднением, как и в обыкновенной трассировке путей (с.м. 'прогрессивное вычисление интеграла'). Алгоритм стохастических прогрессивных фотонных карт выглядит следующим образом:

1. Заранее выделяем некоторый объем памяти для хранения достаточно большой порции фотонов (например 1 миллион фотонов).
2. Трассируем фотоны из источника света до тех пор пока в фотонной карте не накопится указанное число фотонов. Важно отметить при этом, что при расчете доли световой энергии, которую переносит один фотон, необходимо в качестве суммарного числа фотонов использовать количество выпущенных из источника света (а не количество акумулированных) фотонов.
3. Строим ускоряющие структуры для быстрого поиска ближайших фотонов.
4. Трассируем пути из виртуальной камеры. В точках пересечения с диффузными поверхностями собираем освещенность по формуле 1.36. При этом используем фиксированный радиус сбора  $r_i$ . Цвет, как и при обыкновенной трассировке путей накапливается в пикселах.
5. Уменьшаем радиус в соответствии с формулой 1.38. Переходим к шагу 2.

### 1.8.3. Некоторые важные оптимизации

**Три фотонные карты** В целях эффективности (повышения точности) в [30] предлагается хранить 3 отдельные фотонные карты:

1. Глобальная фотонная карта. Данная карта содержит фотоны, прошедшие путями вида  $L(S|D|G|V)^+D$  и используется для вычисления диффузного освещения.
2. Каустическая фотонная карта. Данная карта содержит фотоны, прошедшие путями вида  $LS^+D$  и используется для вычисления каустиков.
3. Объемная фотонная карта. Данная карта содержит фотоны, прошедшие путями вида  $L(S|D|G|V)^+V$  и используется для вычисления объемных эффектов.

Разделение фотонных карт позволяет обрабатывать каустики и диффузное освещение (а также объемное рассеивание) при помощи различных порций фотонов. Это особенно важно при использовании финального сбора, поскольку в этом случае для диффузного освещения с финальным сбором нужно гораздо меньше фотонов чем для каустиков.

**Определение видимости.** Рассмотрим изображение сцены классной комнаты на рис. 1.51. Для расчета освещения на этой сцене фотоны испускались снаружи. При этом, лишь небольшая часть фотонов проникла внутрь комнаты через окно. Остальные фотоны сохраняются с обратной стороны стен и потолка, приводя к неэффективному расходованию вычислительных ресурсов. Для того чтобы не сохранять фотоны на снаружи комнаты, необходимо перед тем, как производить трассировку фотонов, пометить некоторым образом поверхности, для которых фотоны сохраняются в фотонной карте.

Простейший способ заключается в трассировке некоторого небольшого числа путей для каждого пиксела из виртуальной камеры и прямой пометки пересекаемых треугольников. При этом, каждую сторону треугольника необходимо помечать отдельно. Чтобы идентифицировать сторону треугольника, можно использовать векторное произведение для вычисления нормали. Поскольку

вершины треугольника хранятся в памяти в заданном порядке, такой способ однозначно позволяет идентифицировать сторону по нормали. Недостаток данного метода заключается в том, что некоторые тонкие примитивы (треугольники) могут быть пропущены алгоритмом. Хотя, если используется финальный сбор, это не имеет значения поскольку такие примитивы вносят почти нулевой вклад в результирующее значение интеграла. Т.е. чем тоньше треугольник, тем больше вероятность не попасть в него в процессе трассировки путей из виртуальной камеры. Но также уменьшается вероятность попасть в данный треугольник в процессе сбора освещенности.

Еще один способ заключается в трассировке так называемых частиц видимости [37]. Частицы видимости представляют из себя фотоны, испускаемые из камеры. Таким образом, сначала строится фотонная карта видимости, в которой все фотоны трассируются из виртуальной камеры, а затем, используя карту видимости, строится фотонная карта освещенности.



Рис. 1.51. Комната, освещенная солнцем через окно. Лишь  $\frac{1}{7}$  часть выпущенных из источника света фотонов попадает внутрь комнаты.

**Карты проекций.** При оптимизированной реализации построения ускоряющей структуры и сбора освещенности, одним из наиболее ресурсоемких этапов в фотонных картах становится трассировка фотонов. Причина низкой скорости трассировки заключается не том, что медленно происходит сам процесс трассировки одиночного фотона, а в том, на сложных сценах значительная часть испускаемых из источника света фотонов погибает в процессе трассировки поскольку фотоны не достигают видимых областей. Это особенно верно при реализации определения видимости, рассмотренном выше и при трассировки каустических фотонов. На сцене из рисунка 1.51 только  $\frac{1}{7}$  часть фотонов выпущенных из источника света попала внутрь комнаты. На классической сцене 'cornell box' (рис. 1.25) только  $\frac{1}{10}$  часть каустических фотонов, выпущенных из источника света, попала в зеркальный или стеклянный шары, образуя каустики. Русская рулетка дополнительно усугубляет ситуацию, стохастически убивая фотоны.

Карты проекций [30] - это метод, позволяющий не выпускать фотоны в те области, где они гарантированно погибают, не сохраняясь ни на одной поверхности. Идея этого метода заключается в том, чтобы спроецировать сцену на источник света (например при помощи растеризации в кубическую текстурную карту) и пометить на этой проекции 'мертвые области'. После чего в эти мертвые области фотоны не трассируются совсем, погибая еще при рождении на источнике света.

**Приминение фильтрации.** Довольно простым и эффективным способом, позволяющим улучшить четкость каустиков при помощи метода фотонных карт является применение фильтра на основе ядра Епанечникова [38]. При таком подходе освещенность при сборе домножается на определенный вес в зависимости от расстояния до фотона (формула 1.39, нормирующий множитель '2' вынесен за сумму).

$$w_i = 1 - d^2/r^2 \quad (1.39)$$

$$I(\phi_r, \theta_r) \approx \frac{2}{\pi r^2} \sum_{i=1}^K R(\phi_i, \theta_i, \phi_r, \theta_r) \Delta\Phi(\phi_i, \theta_i) w_i \quad (1.40)$$

Возможно использование и других типов фильтров. Например в [30] предлагается использовать 'cone filter' и фильтра Гаусса. Однако, фильтрация на основе ядра Епанечникова эффективнее в вычислительном плане, поскольку не требует вычисления квадратного корня и оперирует отношениями квадратов расстояний.

#### 1.8.4. Объемные фотонные карты

При помощи алгоритма фотонных карт можно достаточно эффективно вычислять эффекты основанные на объемном рассеянии света. В этом случае фотоны сохраняются прямо в объеме, а сбор освещенности происходит при помощи 'марширования по лучу' (ray marching).

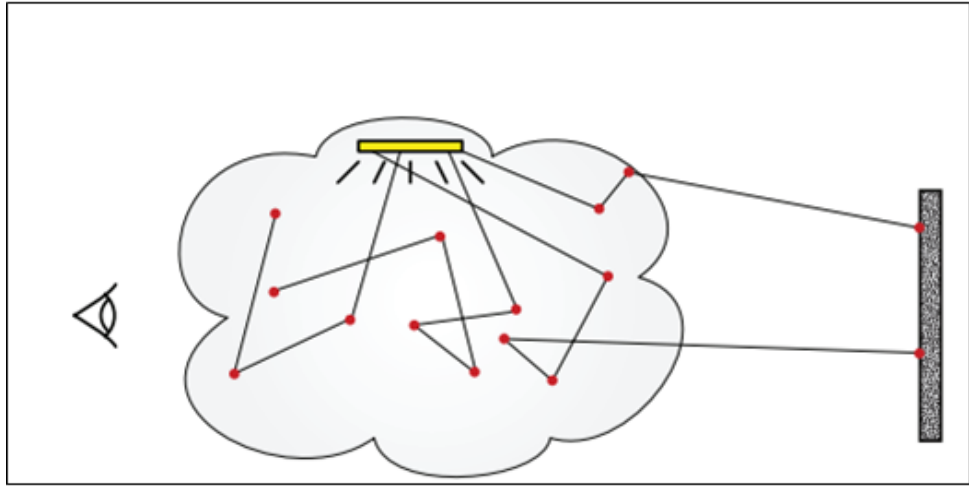


Рис. 1.52. Трассировка фотонов внутри объема [36].

$$\frac{r_{i+1}^3}{r_i^3} = \frac{i + \alpha}{i + 1} \quad (1.41)$$

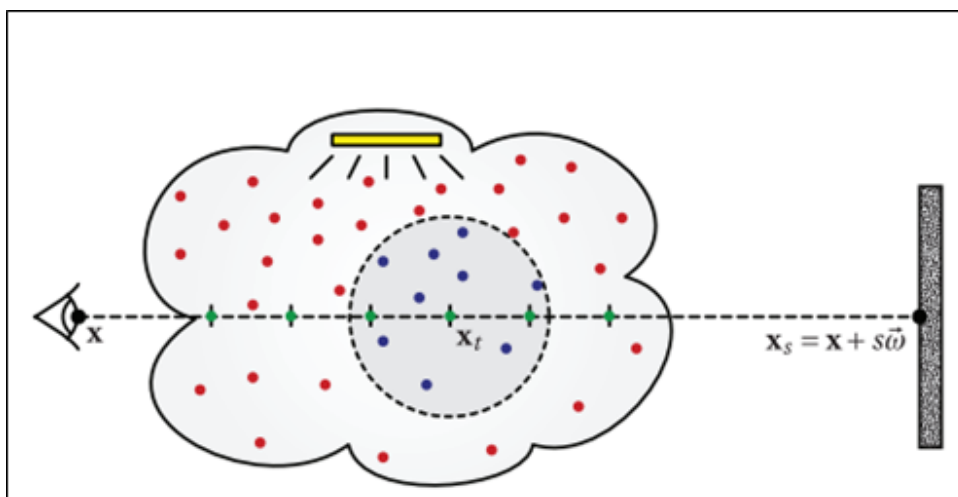


Рис. 1.53. Сбор освещенности при помощи марширования вдоль луча [36].

Для стохастических прогрессивных объемных фотонных карт соотношение 1.38 модифицируется в 1.41 с учетом того что процесс происходит в объеме.

### 1.8.5. Резюме по фотонным картам

Фотонные карты - метод дающий смещенную оценку (смещенный метод). Прогрессивные фотонные карты и стохастические прогрессивные фотонные карты - также методы смещенные, но при этом состоятельные (сходимость по вероятности) [34], [34]). На практике смещение проявляется на изображении в виде цветных пятен (как и в кэше освещенности). В фотонных картах цветные пятна достаточно долго не удаляются с изображения в процессе расчета и изображение лишенное пятен может потребовать обработки нескольких миллиардов фотонов. При таком большом числе фотонов гораздо более простая в реализации прямая и/или обратная трассировка путей даст более точное и приемлимое для человеческого глаза решение.

Финальный сбор позволяет снизить число необходимых фотонов примерно на порядок. Однако, он может быть использован только для вычисления диффузного вторичного освещения и его корость сравнима со скоростью обыкновенной трассировки путей. Для эффективной реализации финального сбора

рекомендуется использовать карты светимости [32] или октанные текстуры [33]. Последний метод предпочтительнее, поскольку для него не нужно реализовывать отображение поверхностей трехмерных объектов на двумерную текстуру (что в общем случае является нетривиальной задачей).

Тем не менее, фотонные карты наряду с методом соединения вершин [27] являются одним из самых эффективных алгоритмов для вычисления путей вида  $S^+DS^+$  (каустики, видимые через стекло или зеркало). Такие пути, как правило, значительно хуже вычисляются трассировкой путей.

## 1.9. Кэш Освещенности (Irradiance Cache)

Кэш Освещенности (Irradiance Cache, IC) – это не способ вычисления интеграла освещенности. Это лишь способ ускорения вычисления этого интеграла на множестве точек. Алгоритм был впервые представлен в работе [39]. Основная идея заключается в том, что вторичное освещение разделяется на две компоненты – низкочастотную (диффузную) и высокочастотную (отражающую). Низкочастотная компонента на изображении и в трехмерном пространстве меняется плавно, поэтому ее можно вычислить каким-либо из методов лишь в очень небольшом числе точек, а в остальных - интерполировать [40]. Рисунки 1 и 2 демонстрируют пример использования кэша освещенности.

Высокочастотная компонента обычно обусловлена резкими максимумами BRDF, поэтому она может быть вычислена сэмплированием с помощью относительно небольшого числа (что конечно не всегда так) лучей только в максимумах BRDF (importance sampling).

Различают кэш освещенности в трехмерном пространстве сцены (world space) и в экранной плоскости (screen space). Реализация кэша в экранной плоскости проще и эффективнее. Интерполяция также производится в пространстве экрана. Однако, кэш освещенности в пространстве экрана может быть исполь-



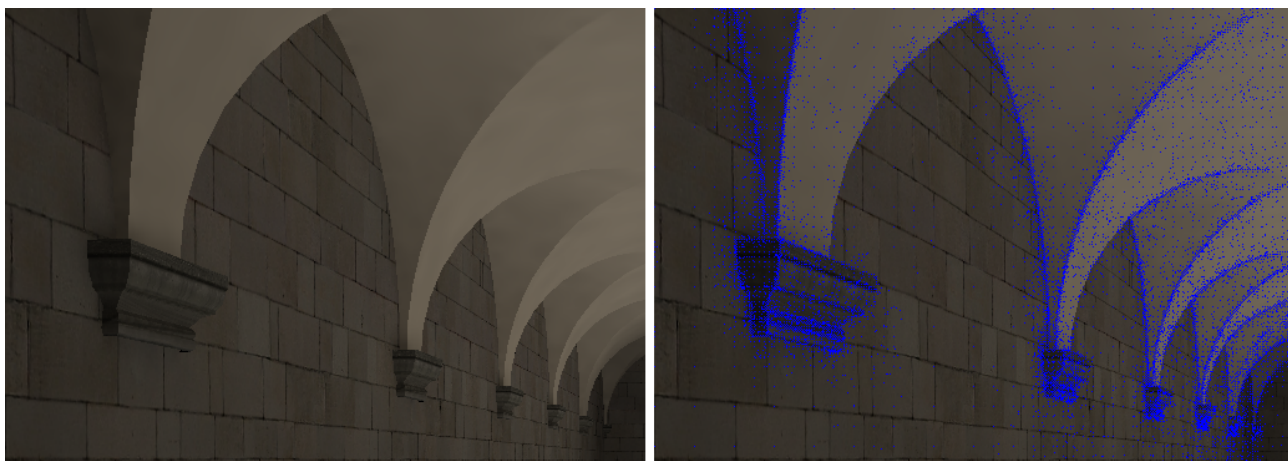


Рис. 1.54. Освещение вычисленное при помощи кэша освещенности (слева). Визуализированные точки кэша (справа).

зован по очевидным причинам только для первично видимых поверхностей. Как правило, кэш освещенности вычисляется на лету. Каждый раз при вычислении вторичного диффузного освещения делается попытка выборки из кэша. Если значение может быть выбрано так, что ошибка меньше допустимой величины, оно получается как результат интерполяции ближайших точек кэша. Если ошибка больше допустимой величины, значение честно вычисляется с помощью трассировки лучей по всем направлениям (или финального

сбора из фотонной карты) и полученная точка сохраняется в КЭШе [40].

**Исходные параметры:**  $p$  - точка на поверхности,  $n$  - нормаль к  
поверхности в точке  $p$

**Результат:** Значение освещенности

```
if InterpolationIsPossible( $p, n$ ) then
    return IrradianceCacheLookUp( $p, n$ ) ;
else
     $I \leftarrow \textit{EvalIntegral}(p, n)$  ;
     $R \leftarrow \textit{CalcValidityRadius}(p, n)$ ;
    InsertRecordInCache( $I, R, p, n$ ) ;
    return  $I$ ;
end
```

**Алгоритм 6:** Алгоритм кэширования освещенности

Алгоритм кэширования вторичного освещения содержит 2 узких места - расчет собственно вторичного освещения в точках кэша и финальный алгоритм интерполяции освещения во всех остальных точках.

### 1.9.1. Вычисление вторичного освещения

Стандартным способом вычисления вторичного освещения в точках кэша является монте-карло трассировка лучей или финальный сбор в сочетании с фотонными картами. Однако в [40] предлагается вычислять вторичную освещенность в точках с помощью растеризации в кубическую текстурную карту. Данная операция может производиться очень быстро т.к. имеет аппаратную поддержку даже в довольно старых видеокартах (обычно используется для симуляции отражений на объектах).

Несмотря на преимущество по скорости, недостаток этого алгоритма в том, что он не может учитывать зеркальных отскоков света и может быть в ряде случаев медленнее трассировки лучей на больших сценах. Причина это-

го заключается в том, что при растеризации вся геометрия обрабатывается графическим процессором. Она проходит весь графический конвейер: преобразование вершин, возможно тесселяцию и геометрические преобразования над примитивами, интерполяция атрибутов вершин и вычисление значений освещенности в пикселах. Допустим, геометрия – это некоторое здание, разделенное на комнаты. Всё здание содержит 2 миллиона вершин. Пусть виртуальный наблюдатель находится в одной из комнат. Несмотря на то, что комната может составлять 10-20% всей геометрии, при вычислении освещения в точке каждый раз потребуется пропускать через графический конвейер все 2 миллиона вершин. Тот факт, что 90% сцены может быть закрыто стеной или каким-то близко-расположенным объектом никак не используется. В то же время, скорость трассировки лучей при правильной реализации зависит от размера входных данных сублинейно (обычно логарифмически), а не линейно, как в случае растеризации.

Отдельного внимания заслуживает вопрос точности вычисления освещенности в точках кэша. Допустим для вычисления освещенности при помощи трассировки путей в пикселах на некоторой сцене будет достаточно около 1000 путей на пиксел. Тогда для вычисления освещенности в точках кэша потребуется может потребоваться большая точность (порядка 4000 путей). Недостаточная точность вычисления для попиксельной трассировки путей приводит к шуму на изображении, который в некоторой степени фильтруется глазом и мало заметен. В случае кэша освещенности, недостаточная точность приводит к появлению цветных пятен, которые гораздо более заметны для человеческого глаза, чем шум.

### 1.9.2. Вычисление радиуса валидности

Одним весьма неочевидным моментом при реализации кэша освещенности является вычисление радиуса валидности точки кэша (переменная  $R$ , алгоритм 6). Подразумевается, что каждая точка иррадианс кэша может быть использована для интерполяции только в некоторой определенной области. Есть по крайней мере 2 причины, по которым радиус валидности точки нужно ограничивать. Первая причина - стремление избежать артефактов. Вторая - скорость выборки из кэша. Если слишком много точек будут затрагиваться при каждой выборке, интерполяция станет медленной.

Обычно область валидности ограничивают сферой с центром в точке иррадианс кэша и некоторым радиусом, называемым радиусом валидности (хотя есть работы где для этой цели используются эллипсоиды). В [3] для вычисления радиуса валидности было использовано 5 различных эвристик. Каждая из них имеет недостатки, но все вместе они дают удовлетворительный результат.

Из самых простых - используется эвристика расстояния до ближайших поверхностей. Эта эвристика вычисляется исключительно из геометрических соображений. При оценке освещенности, для всех трассируемых лучей запоминается расстояние, на котором они ударились о поверхность и из этих расстояний берется наименьшее. Однако при использовании этого подхода, в процессе нахождения минимума важно не принимать в расчет лучи, имеющие маленький угол с тангент-плоскостью (рис. 1.55).

В противном случае на вогнутых криволинейных поверхностях радиус валидности точки всегда будет равным нулю. Альтернативный метод реализации эвристики расстояния до поверхностей заключается в том, чтобы не искать минимальное расстояние а вычислить так называемое "среднее гармоническое расстояние" (формула 1.42).

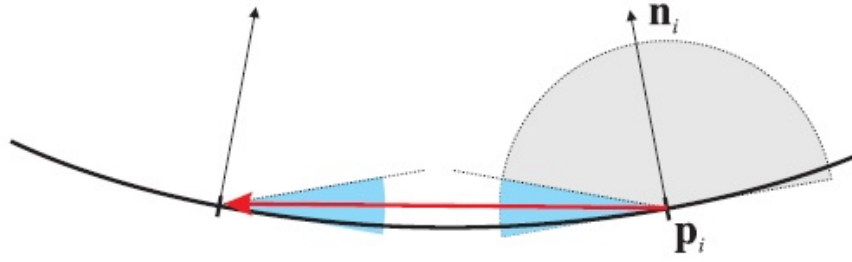


Рис. 1.55. Точки расположенные на вогнутой поверхности. Если принимать в расчет лучи, идущие перпендикулярно нормали, радиус валидности получится слишком маленьким.

$$R_i^{HMD} = \frac{N}{\sum_{i=1}^N \frac{1}{r_i}} \quad (1.42)$$

Число  $N$  в формуле 1.42 - это количество лучей используемых при сэмплинге полусферы.  $r_i$  - расстояние от точки кэша до поверхности в которую ударился соответствующий луч. Формула 1.42 для среднего гармонического расстояния характерна тем, что чем меньше расстояние  $r_i$  тем больше оно влияет на финальный результат. Дополнительно размер радиуса валидности рекомендуется ограничивать снизу размером 2-4 пикселей (спроецированных на сцену в world space) и сверху размером 20-64 пикселей. Конкретные числа могут варьироваться.

### 1.9.3. Структуры данных и интерполяция

Для хранения точек кэша освещенности обычно используется специальное октодерево со множественными ссылками (multiple reference octree) [40]. Точки кэша излучения в таком октодереве хранятся только в листьях, но каждый лист хранит список всех сфер, которые он пересекает. Основной плюс такого октодерева в том, что возможно осуществить простой нерекурсивный поиск от корня к листу с последующим перебором всего лишь нескольких точек кэша освещенности.

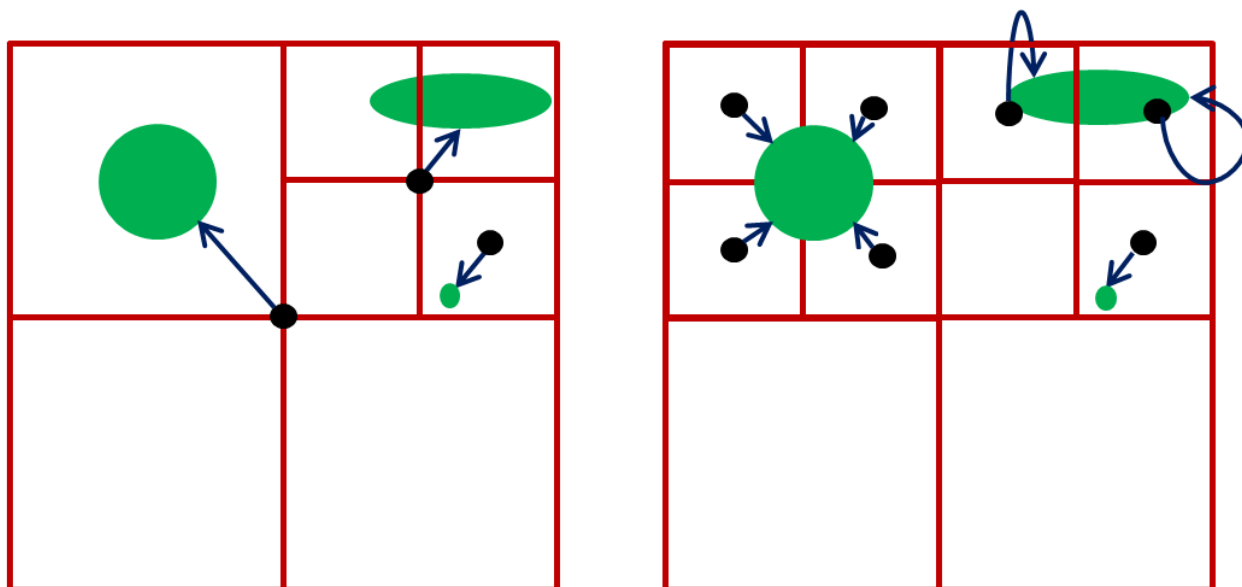


Рис. 1.56. Октодерево с одиночными ссылками (слева) и окто-дерево с множественными ссылками (справа). Стрелки обозначают ссылки.

**Интерполяция.** В заданной точке трехмерного пространства, при помощи поиска в октодереве необходимо найти все сферы (с центрами в точках иррадианс кэша и радиусами равными радиусам валидности записей кэша), которые пересекают данную точку. Затем, пройдя в цикле по всем точкам, нужно собрать с них освещение. Одна из практических формул была предложена Tabellion-ом и Lamorlette в [32].

#### 1.9.4. Резюме по кэшу освещенности

Кэширование вторичного освещения - весьма эффективная техника ускорения вычисления глобально освещения, позволяющая снизить количество необходимых вычислений практически на 3 порядка (для больших разрешений). Допустим имеется изображение размером 1024x1024 пиксела. Если вычислять вторичное освещение в каждой точке, то потребуется суммарно протрассировать порядка  $10^9$  лучей только для вычисления вторичной освещенности. При использовании кэша освещенности количество точек, в которых необходимо вычислить вторичную освещенность сокращается с миллиона до

нескольких тысяч. Соответственно, число лучей падает с  $10^9$  до  $10^6 - 10^7$ .

Однако, кэширование освещенности имеет определенные недостатки:

1. Кэш освещенности хорошо работает только на сценах с преимущественно гладкими поверхностями. На сценах с большим количеством геометрических деталей (например карты нормалей для имитации микрорельефа) кэш освещенности не даст преимущества и может даже замедлить процесс редеринга.
2. Кэш освещенности достаточно сложно сделать параллельным.
3. Кэш освещенности дает мерцание при анимации. Если стохастическая трассировка путей дает шум, который эффективно удаляется при помощи размытия в движении, то пятна вызванные кэшированием освещенности удалить гораздо труднее.
4. Кэш освещенности все же снижает точность решения и является смещенным методом. Чем ниже точность, тем больше выигрыш в скорости, но тем более заметны артефакты.

## 1.10. Выводы к первой главе

## Глава 2

### Анализ литературных источников

В данной главе проводится обзор существующих алгоритмов (исключая алгоритмы рассмотренные в предыдущей главе) и программных решений как на центральных, так и на массивно-параллельных процессорах. Обсуждаются тенденции в современных исследованиях проблемы глобальной освещенности и рассматриваются наиболее передовые методы.

#### 2.1. Алгоритмы на центральном процессоре

##### 2.1.1. Излучательность

Метод излучательности (Radiosity) в компьютерной графике был впервые представлен в работе [21]. Идея алгоритма Излучательности заключается в том, чтобы разбить поверхность сцены на множество маленьких кусочков - патчей; после чего производится расчет переноса энергии между отдельными патчами.

Алгоритм вводит следующие ограничения на входную сцену:

1. Все поверхности имеют ДФО Ламберта (отражают свет равномерно во все стороны).
2. Сцена замкнута, т.е. суммарная энергия в системе должна сохраняться.

На первом этапе алгоритма все поверхности сцены делятся на патчи. Патч – это элементарная единица поверхности. Дискретизация поверхности на патчи позволяет заменить интеграл в уравнении освещенности на конечную сумму интегралов специального вида. Каждый такой интеграл, называемый форм-фактором, задает взаимное влияние двух отдельных патчей (т.е. сколько энер-



гии переходит от одного патча к другому). Если мы знаем значение каждого форм-фактора (для каждой пары патчей), то процесс синтеза изображений сводится к решению системы линейных алгебраических уравнений. Основная трудность в алгоритме – расчет форм-факторов [41].

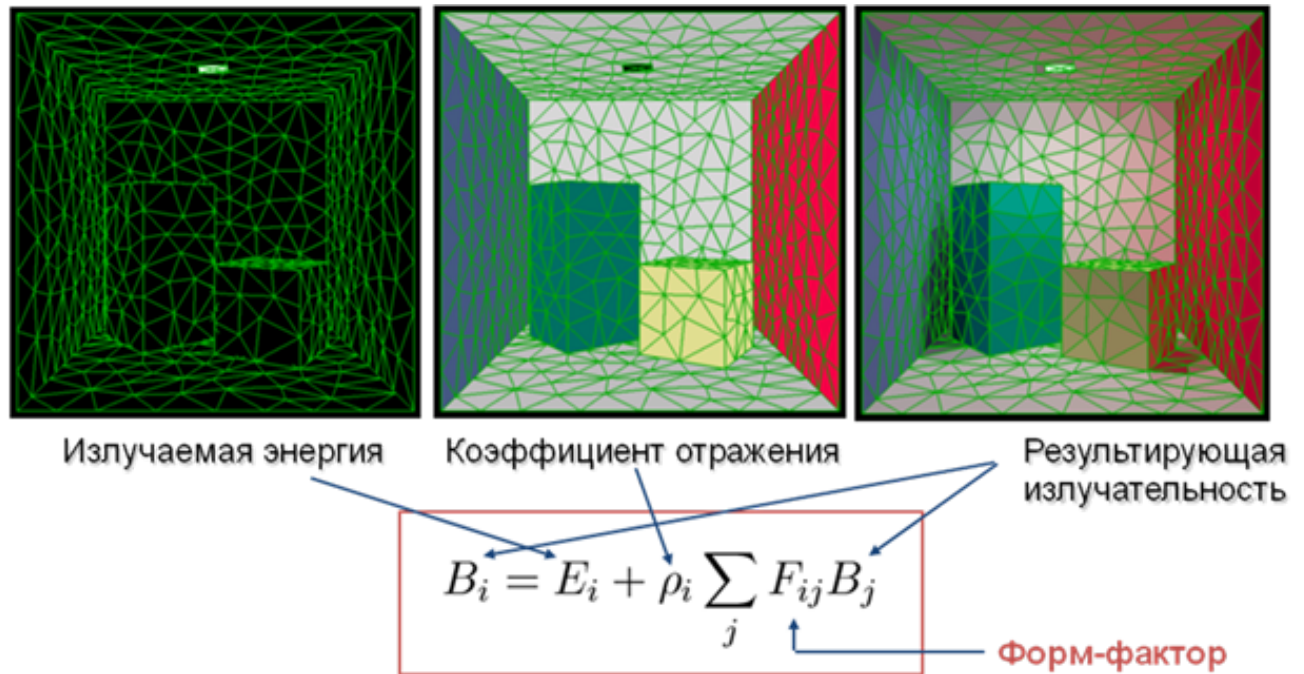


Рис. 2.1. Иллюстрация алгоритма излучательности и получаемой системы уравнений [41].

Форм фактор, в свою очередь, можно рассчитать для каждой пары патчей при помощи Аналогии Нуссельта [42] зная только геометрию сцены.

Основными недостатками метода излучательности являются:

1. Поддержка только Ламбреттовых материалов.
2. Артефакты на изображении (т.е. низкая точность) в местах перехода между патчами.
3. Большой объем потребляемой памяти и вычислений при увеличении точности и необходимости расчета резких переходов свет-тень даже при адаптивном разбиении (разный размер патчей в зависимости от того, насколько быстро на поверхности происходит смена света и тени).

Тем не менее излучательность достаточно часто используется для вычисления вторичного диффузного освещения, когда высокая точность не требуется. Например, в известной компьютерной игре Battlefield 3 для предрасчета вторичного освещения [43].

## Мгновенная Излучательность и VPL

Основная идея метода мгновенной излучательности (Instant Radiosity; рис. 2.2) заключается в том, чтобы аппроксимировать вторичное освещение в сцене при помощи некоторого числа виртуальных точечных источников света (VPL, Virtual Point Light) [44]. При этом сами виртуальные источники могут расставляться по сцене различными способами. Освещение от точечных источников света может быть вычислено при помощи трассировки лучей или метода карт теней. Однако, заслуживает внимания вопрос эффективной реализации при большом числе источников. Одно из возможных решений основывается на построении специального дерева для источников света [45].

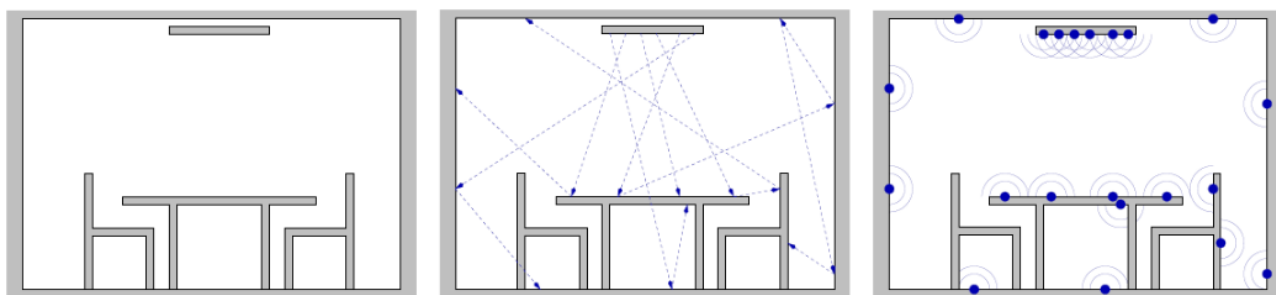


Рис. 2.2. Иллюстрация метода мгновенной излучательности.

Мгновенная излучательность породила целое направление исследования решения проблемы глобальной освещенности, которое называется 'Методом Многих Источников' (Many Light Method) [46]. Идея вычислять освещение при помощи множества источников вторичного освещения применяется в основном если предпочтение отдается скорости, но не точности решения [46]. Здесь следует обратить внимание на работы [45], [47], [48], которые значи-

тельно улучшают точность решения проблемы глобальной освещенности при помощи виртуальных источников.

### 2.1.2. Подход на основе облаков точек

Данный подход (называется Point Based Global Illumination или PBGI) позволяет используется в программном решении Render Man компании Pixar [49]. Он хорошо зарекомендовал себя в производстве анимационным компьютерных фильмов (т.е. мультфильмов), поскольку позволяет достаточно быстро получить свободное от шума (что особенно важно для анимации) приближение диффузного глобального освещения.

На первом этапе алгоритм преобразует освещенные поверхности в облака точек (правильнее будет сказать элементов поверхности), сохраняющих позицию, нормаль, радиус (задающий площадь элемента поверхности) и цвет (значение освещенности). Точки называются англоязычным термином 'сюрфел'(surfel), происходящем от слово-сочетания SURFace-ELEment.

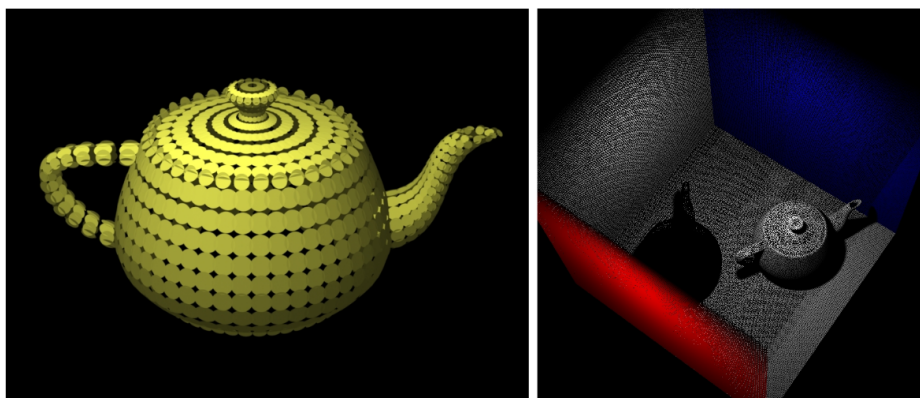


Рис. 2.3. Представление освещенной поверхности в виде множества сюрфелов [49].

После того как множество сюрфелов получено, над ними строится окто-дерево. Для каждого листа окто-дерева вычисляется излучаемая освещенность следующим образом:

1. Если сюрфелы в узле мало отличаются друг от друга по нормали, вы-

числяется один сюрфел с усредненной позицией, нормалью, площадью и цветом.

2. Если сюрфелы сильно различаются по нормали, значение излучаемой освещенности сохраняется в представлении, основанном на сферических гармониках [50]. При этом оказывается, что коэффициенты гармоник для узла могут быть вычислены как сумма коэффициентов отдельных сюрфелов.

Описанный выше метод применяется иерархически, строя более высокие уровни окто-дерева. При этом, интересным фактом является то что в родительских узлах коэффициенты гармоник могут быть вычислены точно так же простым суммирование коэффициентов дочерних узлов [49].

Глобальная освещенность затем вычисляется путем рекурсивного обхода окто-дерева и растеризации отдельных сюрфелов или кластеров сюрфелов. Для того, чтобы определить можно ли использовать текущий уровень дерева (представляющий целый кластер точек) предлагается контроль точности на основе телесного размера кластера (площадь проекции узла на провехность деленная на квадрат расточния).

Для того чтобы обеспечить учет видимости сюрфелов (т.е. обеспечить расчет вторичных теней убедившись что сюрфелы, закрытые друими сюрфелами не вносят вклад в получаемое значение освещенности), используется растеризация геометрического приближения сцены на куб вокруг точки, в которой вычисляется освещенность (будем называть ее целевой точкой).

Далее, в зависимости от расстояния до сюрфела, их растеризация производится с использованием 3 стратегий:

1. Если от целевой точки расстояние до сюрфела большое, используется аппроксимация в виде узла окто-дерева, сохраняющая единственный сюрфел или гармоники.

2. Для сюрфелов, для которых аппроксимация узлом недостаточна использовалась растеризация отдельно каждого сюрфела.
3. Для сюрфелов которые находились ближе некоторого порога друг к другу использовался алгоритм локальной трассировки лучей. Это работает быстрее чем обыкновенная трассировка лучей т.к. нужно вычислять пересечение луча лишь с небольшим числом сюрфелов и их цвет уже известен.

Для поддержки множественных диффузных отскоков света в [49] предлагается повторять алгоритм в течении нескольких проходов. Учитывая этот, метод PBGI можно считать в некотором роде аналогом иерархической излучательности [51]. В сочетании с собственно алгоритмом PBGI в [49] использовался модифицированный кэш освещенности для ускорения вычисления глобального освещения и отмечается его высокая эффективность.

### 2.1.3. Методы на основе Монте-Карло интегрирования

В предыдущей главе были детально рассмотрены методы на основе Монте-Карло интегрирования и метод фотонных карт. Чтобы избежать повторения, в данной главе они не описываются. Однако, необходимо рассмотреть работы, предлагающие значительные улучшения по сравнению с оригинальными методами а также работы основанные на комбинированных методах.

### Методы на основе MLT

**Kelemen style MLT** Оригинальный алгоритм MLT, разработанный Э. Вичем и Л. Гибасом [28] и обсуждаемый в первой главе использует мутации в мировом пространстве или, иначе говоря, мутации в пространстве путей. Причем, одной общей стратегии мутации для всех видов путей не существует.

Такая стратегия строится путем комбинирования большого числа различных стратегий для отдельных случаев (мутации в пространстве линз, мутации каустик, мутации на источнике света и.т.д.). Это значительно усложняет реализацию, поскольку:

1. Усложняется процесс вычисления вероятностей перехода  $T(X|Y)$  и  $T(Y|X)$ . Если вероятности будут вычислены неправильно, нарушится условие, гарантирующее сходимость к правильному решению (detailed balance).
2. Параметры алгоритма мутации необходимо настраивать в зависимости от входных данных.

В работе [52] предложен простой и стабильный (независящий от входных данных) метод MLT, основанный на мутациях путей в 'первичном пространстве выборок' - единичном кубе, в котором генерируются случайные числа для метода Монте-Карло. При этом, в работе проводятся некоторые выкладки, позволяющие значительно упростить вычисление вклада текущей выборки (текущего пути) в изображение, которое фактически удалось свести к формулам 2.1, 2.2:

$$w_{new} = \frac{a + large\_step}{(I/b + p_{large})M} \quad (2.1)$$

$$w_{old} = \frac{1 - a}{(I_{old}/b + p_{large})M} \quad (2.2)$$

В формулах 2.1, 2.2 приняты следующие обозначения:

- $w_{new}$  - вес нового пути.
- $w_{old}$  - вес старого пути.
- $a = \min(1, \frac{I_{old}}{I})$ .

- $I = lum(color)$  - яркость текущего пути.
- $I_{old} = lum(colorPrev)$  - яркость предыдущего принятого пути.
- $large\_step$  - переменная, которая равна 1 если текущий путь сделал так называемый 'большой шаг' и 0 в противном случае. Большой шаг соответствует генерированию полностью случайного вектора параметров пути (то есть все числа перегенирируются без использования мутаций).
- $b$  - константа нормализации, такая что плотность вероятности под-интегральной функции  $p = \frac{I^*}{b}$ . Под  $I^*$  в данном случае понимается проинтегрированная яркость пиксела (в отличие от обозначений  $I$  и  $I_{old}$  для яркости отдельных выборок).
- $p_{large}$  - вероятность так называемого 'большого' шага.
- $M$  - номер текущей выборки.

Далее алгоритм генерирует случайное число с равномерным распределением от 0 до 1 и:

- Если это число меньше  $a$ , мутация принимается. Новый путь (его вклад и вес) запоминается в качестве старого, а старый возвращается в качестве результата.
- Если это число больше или равно  $a$ , мутация отвергается и возвращаются вклад и вес нового пути.

Таким образом, если мутация принимается, вычисление вклада принятого пути откладывается до следующей итерации, а в результирующее изображение вносит вклад предыдущей принятый путь. Если же мутация отвергается, то в результирующее изображение вносится вклад отвергнутого пути. Следует отметить также, что за счет настраивания вероятности  $p_{large}$  можно улучшить

свойства алгоритма на темных участках (где оригинальный MLT работает, как правило, хуже чем обыкновенная Монте-Карло трассировка путей поскольку отводит на темные области меньшее число выборов) [52].

**Energy Redistribution Path Tracing (ERPT)** Алгоритм ERPT [53] является гибридом MLT и идеи фильтрации, о которой пойдет речь далее. В отличие от MLT, ERPT использует короткие последовательности мутаций и перераспределяет энергию путей на плоскости изображения. В отличие от фильтров и кэша освещенности, ERPT позволяет получить несмещенное решение. Идея, которая лежит в основе алгоритма ERPT называется 'поток энергии' (energy flow) и основывается на том факте, что интегралы освещенности для соседних пикселей, как правило, коррелируют.

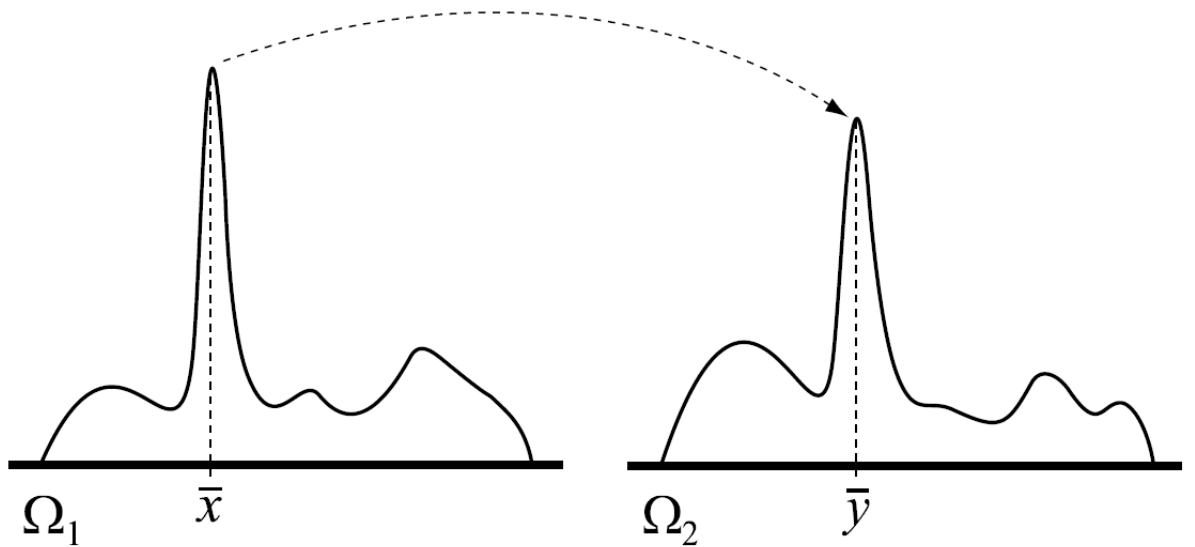


Рис. 2.4. Иллюстрация идеи 'потока энергии' в алгоритме ERPT. Поток энергии связывает коррелирующие интегралы и устанавливает перенос энергии между ними.

Предположим, что мы имеем два коррелирующих интеграла  $I_1$  и  $I_2$  на соответствующих областях определения их под-интегральных функций  $\Omega_1$  и  $\Omega_2$  (рис. 2.4). Далее предположим, что в процессе интегрирования  $I_1$  в  $\Omega_1$  была найдена точка  $\bar{x}$  с высокой значимостью. Поскольку  $I_1$  и  $I_2$  коррелиру-



ют, высока вероятность найти на области определения  $\Omega_2$  недалеко от точки  $\bar{x}$  точку  $\bar{y}$ , которая также будет иметь высокую значимость. В действительности эта же самая идея лежит в основе MLT, однако, в алгоритме ERPT ослабляются ограничения на реализацию потока энергии (рис. 2.5). Вместо детального баланса потока энергий используется так называемый общий баланс. Общий баланс потока энергий требует чтобы суммарная входящая и выходящая энергия для точки были равны. Тогда как детальный баланс является более строгим ограничением и требует чтобы для каждой пары точек энергия перетекаемая между ними сохранялась [53].

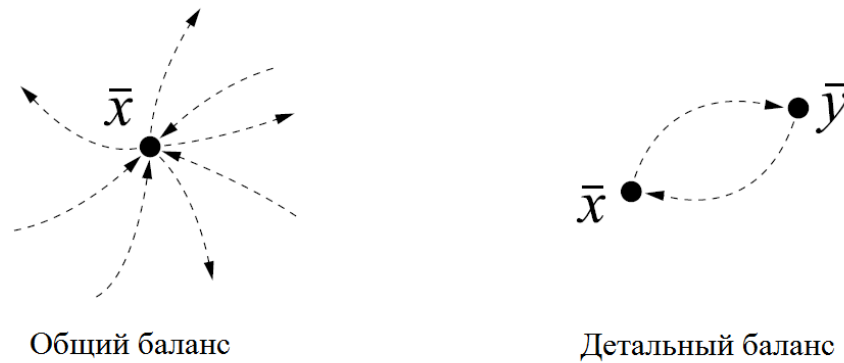


Рис. 2.5. Общий (general balance) и детальный (detailed balance) баланс потока энергий.

ERPT применяют так называемый фильтр сбалансированного потока энергий (balanced energy flow filters) - набор правил, при помощи которых поток энергии может быть реализован таким образом, чтобы условие общего баланса сохранялось.

Сравнения, предоставленные в работе [53] позволяют сделать вывод о том, что методы MLT и ERPT - приблизительно одинаковы по эффективности.

**Gradient Domain MLT** В работе [54] было предложено расширение оригинального алгоритма MLT, позволяющее улучшить сходимость метода. Алгоритм сначала вычисляет вертикальную и горизонтальную разницу (градиен-

ты) изображения а также грубое приближение самого изображения при помощи обыкновенного алгоритма MLT, после чего производится реконструкция финального изображения решая уравнение пуассона.

### Метод соединения вершин (Vertex Merging)

В работе [27] с целью улучшения алгоритма двунаправленной трассировки путей на путях вида *SDS* был предложен метод соединения вершин (Vertex Merging, VM). Идея основывается на том, чтобы соединять пути на диффузной поверхности даже если они попали не строго в 1 точку, а в находятся друг от друга на расстоянии некотороно небольшого радиуса  $r$  (рис 2.6). Данный метод может рассматриваться как расширение финального сбора, при котором вместо фотонов храняться пути от источника света, а вместо сбора непосредственно освещенности применяется полностью аналогичное BDPT соединения вершин пути сбора и путей из источника, закончившихся на той же поверхности в некотром радиусе от точки сбора.

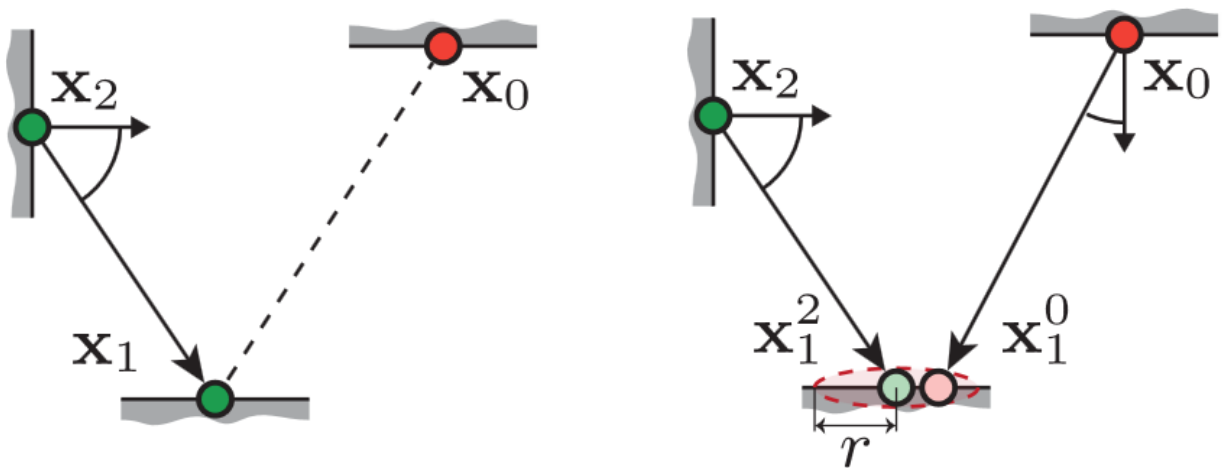


Рис. 2.6. Иллюстрация алгоритма VM. Соединение пути в классическом BDPT изображено слева. Соединение в VM - справа.

## Регуляризация в пространстве путей

В работе [55] предложена похожая на метод соединения вершин идея о том, что можно соединять точки различных путей для переотражений вида  $SDS$  даже если при зеркальном отражении ( $S$ ) попадание в точку где произошло диффузное отражение ( $D$ ) неточно (рис 2.7). Однако, данный метод отличается тем, что используют преобразование  $S$  отражения в  $D$  отражение и может быть применен ко всем видам Монте-Карло интеграторов, а не только к BDPT, как метод соединения вершин.

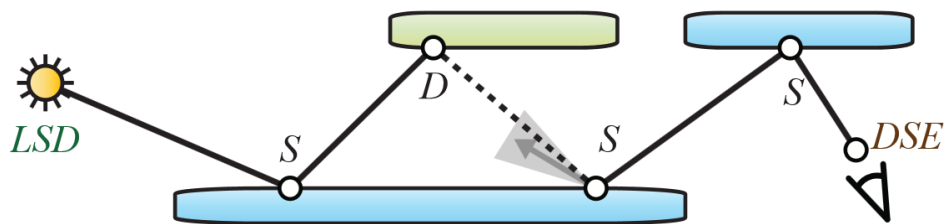


Рис. 2.7. Иллюстрация процесса регуляризации в пространстве путей.

В работе [55] отмечается что угол регуляризации (помеченный на рисунке 2.7 серым треугольником) может быть вычислен из параметра радиуса сбора в методе VM (то есть сам способ соединения похож на способ в методе VM). Однако, в отличие от VM, в данном методе будут соединяться не все пути, а только так называемые 'несэмплируемые' пути. То есть те пути, которые не могут быть получены без внесения смещения методами Монте-Карло (в любом виде - PT, BDPT, MLT, ERPT и.т.д. ). Это позволяет уменьшить смещение(bias) в получаемом решении. Помимо вышеперечисленных отличий, важным результатом метода [55] по сравнению с предыдущими работами является возможность получения изображения точечных (или бесконечно малых) источников света при многократных зеркальных переотражениях.

#### 2.1.4. Методы на основе фотонных карт

Метод фотонных карт, рассмотренный в первой главе, имеет весьма обширный простор для улучшений. Одним из наиболее практичных методов, позволяющих решить большинство проблем оригинального алгоритма фотонных карт является алгоритм Стохастических Прогрессивных Фотонных Карт (SPPM), рассмотренные в 1 главе. Однако, заслуживает рассмотрения методы, дополнительно улучшающие фотонные карты сами по себе а также альтернативные SPPM решения и комбинированные методы.

#### Оценка освещенности

Алгоритм фотонных карт для решения проблемы глобальной освещенности породил подзадачу 'оценки плотности фотонов' (Photon Density Estimation) [36]. Для решения этой задачи применяются различные подходы.

В работе [38] был предложен метод, позволяющий значительно снизить смещение (bias) в получаемом решении и сделать изображение каустик более четкими. Идея метода состоит в том, чтобы хранить дополнительно на каждый фотон так называемый 'дифференциал фотона' (photon differential) по аналогии с дифференциалами лучей [56] (рис. 2.8). Дифференциалы фотонов позволяют аппроксимировать одним лучом целый пучок фотонов. Именно этот аппроксимируемый пучок фотонов и называется дифференциалом фотона в работе [38].

Трассировка таких пучков технически не отличается от обыкновенной трассировки фотонов. При взаимодействии с поверхностью дифференциалы фотонов пересчитываются аналогично дифференциалам лучей в работе [56]. Дифференциалы используются далее во время сбора для повышения точности. Обыкновенные фотоны вносят вклад в интеграл освещенности на поверхности в виде круга. Дифференциалы фотонов вносят вклад в виде эллипса,

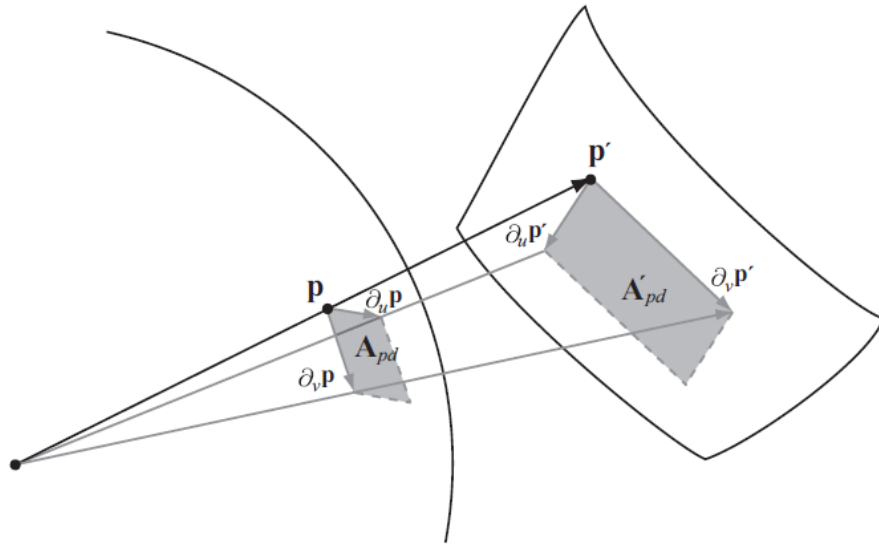


Рис. 2.8. Трассировка луча и его дифференциальных векторов от точки  $p$  к точке  $p'$ .

делая, таким образом, фильтрацию анизотропной. Метод дифференциалов фотонов требуют хранения дополнительно 24 байт на фотон и увеличивает объем вычислений, выполняемых при сборе за счет умножения матрицы  $3 \times 3$  на вектор разницы между позицией фотона и точкой сбора. Тем не менее, повышение точности решения значительное, что позволяет считать метод практичным.

В работе [57] предложен подход, называемый 'прогрессивной релаксацией фотонов' (progressive photon relaxation). Ключевым моментом метода является использование фиксированного объема памяти и фиксированного числа сохраняемых фотонов, называемых в работе 'устойчивыми фотонами' (persistent photons). Метод основан на построении диаграмм Вороного с последующей регуляризацией ячеек диаграммы на основе информации о вновь поступающих, временных (transient), фотонах. В среднем по эффективности метод не на много превосходит стохастические прогрессивные фотонные карты и является достаточно сложным в реализации. Преимущество метода релаксации заключается том, что он работает в мировом пространстве и позволяет

получить решение высокой точности, не производя оценку освещенности на каждом проходе (что требуется, например, в методах RPPM и SPPM). Можно сказать, что метод позволяет построить карту освещенности высокой степени детализации и качества. В числе плюсов метода также следует отметить построение структуры с высокой степенью регулярности, что может положительно сказаться на производительности сбора освещенности при реализации на массивно-параллельных системах.

В работе [58] для ускорения финального сбора был предложен подход, называемый обратными фотонными картами (reverse photon mapping). Основная идея метода заключается в том, чтобы вместо фотонов запоминать лучи сбора kd-дерева специального вида. После чего производить трассировку фотонов накапливая результат для лучей сбора. Серьезным недостатком метода является большой объем потребляемой памяти поскольку число лучей сбора, как правило, чрезвычайно велико.

## **Комбинированные методы на основе фотонных карт**

Идея разделения освещенности на независимые компоненты, по-видимому, впервые была высказана в виде алгоритма кэша освещенности [39]. В дальнейшем многие исследователи приходили к тому, что вычислять отдельные компоненты освещенности можно более эффективно за счет априорных знаний о природе компоненты и применении соответствующего алгоритма [59], [35], [60]. В работе [61] апробирована идея комбинирования метод Монте-Карло и прогрессивных фотонных карт с целью использования преимуществ обоих методов. Освещение разделялось на 2 части: освещение вызванное каустиками (пути вида  $E(S|G)^+DL$  или  $E(S|G)^+D(S|G)^+L$ ) и все остальное освещение. Первую часть освещения предлагается вычислять при помощи метода стохастических прогрессивных фотонных карт, а вторую при помо-

щи стандартной Монте-Карло трассировки путей. Поскольку множества путей из 1 и второй части не имеют пересечений, а их объединение составляет все множество путей, метод приводит к корректному результату. В работе [62] предложен улучшенный способ комбинирования результата при помощи взвешивания на основе многократной выборки по значимости.

В работе [63] был предложен иной способ комбинирования фотонных карт и Монте-Карло трассировки. Вместо того чтобы использовать фотонные карты для вычисления непосредственно освещенности, оценка освещенности на основе фотонной карты использовалась для вычисления плотности вероятности падающего освещения. После чего к методу Монте-Карло применялась выборка по значимости на основе ДФО и посчитанной плотности вероятности падающего освещения. Этот метод является альтернативой MLT, когда падающее освещение неравномерно в высокой степени.

#### **2.1.5. Методы на основе кэша освещенности**

Помимо исследований методов вычисления интеграла освещенности, значительное число работ направлено в сторону ускорения вычисления интеграла освещенности на множестве точек. Идея кэширования освещенности получила множество направлений развития.

В работах J.Krivanek-а [64], [65], [40] предложено улучшение контроля ошибки экстраполяции, многопроходный адаптивный алгоритм, который, при добавлении новой точки в кэш, сравнивает относительное отклонение освещенности в соседних точках, что уменьшает количество артефактов (но приводит к большому числу дополнительных вычислений) и расширение метода кэширования освещенности на умеренно-зеркальные ДФО. Падающая освещенность сохранялась при помощи сферических гармоник [50]. Значительным преимуществом кэша падающей освещенности над простым кэшем освещенности

является способность ускорять вычисления на поверхностях с высокой геометрической детализацией на основе микро-релефа.

В [66] описывается использование процедуры репроекции вместе с кэшем освещенности. Идея репроекции состоит в том, что точки пересечения лучей и поверхностей сцены, которые были вычислены ранее и записаны в кэш, проецируются на полусферу в новой точке поверхности, в которой необходимо вычислить освещение. Недостатком подхода описываемого в [66] является наличие 'пустот' на полусфере вокруг точки, на которую выполняется репроекция. Репроекция получила дальнейшее развитие в работе [67], где устранены основные недостатки кэша падающей освещенности на основе репроекции за счет специального алгоритма расслоения репроецируемых точек (это позволяет корректно реализовать экстраполяцию освещения от нескольких записей кэша с учетом закрытия одних объектов другими) и перетрассировки в местах, где были обнаружены пустоты, и представлена высокоэффективная параллельная реализация на основе SSE инструкций центрального процессора.

В работе [67] также высказывается идея о том что можно использовать кэш падающей освещенности для оценки плотности вероятности падающего освещения, чтобы ускорить сходимость метода Монте-Карло, если необходимо получать несмещенное решение.

В работе [68] предложено разделять падающее освещение на дальнее и ближнее с тем, чтобы сохранять только дальнее освещение в кэше падающей освещенности. Поскольку освещение от дальних объектов меняется, как правило, значительно более плавно чем освещение от ближних, сохранение в кэше только 'дальней' освещенности позволяет значительно снизить число записей кэша. Ближнее освещение в работе [68] вычислялось при помощи аппроксимации, основанной на методе излучательности без учета функции видимости. Это значительно ускоряет вычисление ближнего освещения, однако, приводит



к некорректному результату. Тем не менее, развитие идеи разделения падающего освещения на ближнее и дальнее с корректным вычислением ближнего освещения, освещенное в [67], можно считать перспективным направлением.

### 2.1.6. Методы на основе фильтрации

Идея кэширования освещенности, обсуждаемая ранее говорит о том, некоторую функцию освещенности можно вычислить на небольшом подмножестве точек трехмерного пространства, а в остальных точках - интерполировать. Скорость возрастает, таким образом, за счет пере-использования информации и уменьшения объема вычислений.

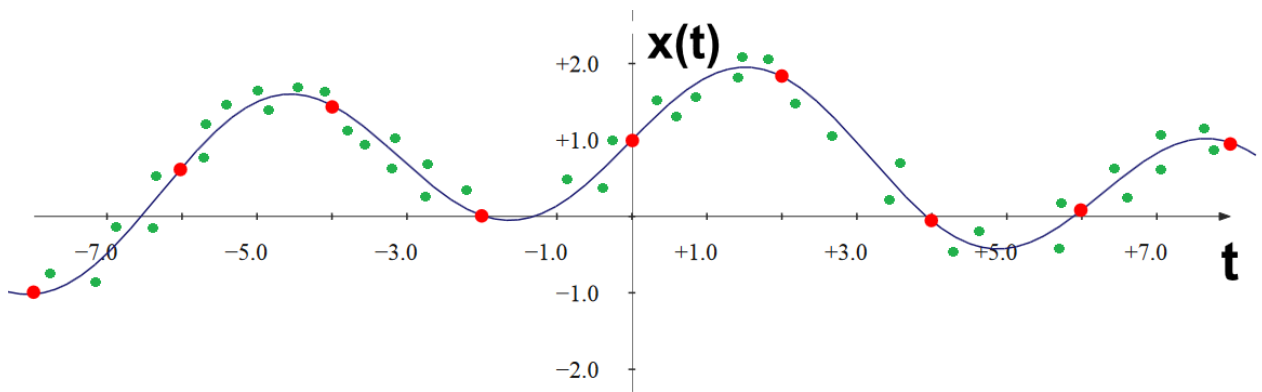


Рис. 2.9. Интерполяция и фильтрация. Красные точки обозначают места, в которых значения функции  $x(t)$  были вычислены точно. Зеленые точки обозначают значения точечных выборок в методе Монте-Карло при оценки функции  $x(t)$  на множестве точек оси  $t$ . Применяв к ним операцию фильтрации можно восстановить с некоторой точностью искомую функцию  $x(t)$ .

Фильтрация (рис. 2.9) ускоряет расчет отсвещения за счет того же эффекта - переиспользования информации. Однако, вместо того чтобы вычислять значение функции в нескольких точках с высокой точностью, этот метод принимает на вход Монте-Карло выборки. А именно: результирующую яркость путей с некоторой дополнительной информацией - позицией первого пересечения, нормалью к поверхности в этой точке, текстурными координатам,

координатами на линзе (если моделируется эффект глубины резкости) и случайными параметрами самого пути. Таким образом, строится многомерное пространство где каждое измерение является некоторым параметром пути. В этом пространстве затем производится фильтрация. Одно из наиболее качественных решений было получено в работе [69], где фильтрация производится с учетом зависимости между случайными величинами в методе трассировки лучей Монте-Карло и значениями в полученной выборке.

Основной проблемой подходов на основе многомерной фильтрации является высокая алгоритмическая сложность многомерных фильтров. Однако, следует отметить работу [70], где предложен подход с линейной сложностью в зависимости от числа измерений и количества пикселей изображения и, с учетом применения графических процессоров, достигнута скорость обработки поступающих сэмплов (выборок) порядка десятков кадров в секунду в разрешении 10 мега-пикселей.

## **2.2. Резюме по методам вычисления глобальной освещенности**

Несмотря на значительное число существующих методов компьютерной графики, сфокусированных на решении проблемы глобальной освещенности, область продолжает активно исследоваться. Об этом свидетельствует анализ объема публикаций предлагающих новые методы за 2012 и 2013 годы. Среди причин способствующих неугасающему интересу исследователей к проблеме глобально освещенности следует отметить:

1. Высокую актуальность проблемы и широкий спектр применения ее решений, от индустрии развлечений до точной визуализации промышленных изделий, оптического моделирования и расчета переноса нейтронов

в реакторах.

2. Высокую вычислительную сложность алгоритмов и большое время расчета. Ни одно современное исследование не обходится без анализа эффективности предложенного метода.
3. Специфическими требованиями области (где-то требуется точность, где-то скорость, где-то простота реализации) и, как следствие, высокой специализацией многих методов.
4. Достаточно высокую сложность алгоритмов в плане реализации и поддержки коллективом программистов и стремление исследователей сделать алгоритмы проще, а решения - практичнее.

Среди практичных подходов, обладающих высокой точностью, следует отметить Монте-Карло трассировку, усеченную двунаправленную трассировку, Kelemen-Style MLT, подходы на основе регуляризации, соединения вершин, фотонных карт, кэша освещенности, а также комбинированные подходы.

Поскольку одним из важнейших факторов является производительность, следует обратить особое внимание на работы по решению проблемы глобальной освещенности на параллельных и массивно-параллельных системах. Это особенно важно в связи с ростом популярности массивно-параллельных процессоров. Далее будут рассмотрены проблемы и пути их решения при реализации алгоритмов вычисления интеграла освещенности на массивно-параллельных системах.

### **2.3. Краткое введение в программирование GPU**

Когда первые видеокарты только появились в продаже, они представляли собой достаточно простые (по сравнению с центральным процессором)

узкоспециализированные устройства, предназначенные для того чтобы снять с процессора нагрузку по визуализации двухмерных данных. С развитием игровой индустрии и распространением трехмерных игр появился спрос на ускорение 3D графики. Со времени создания компанией 3Dfx первых видеокарт Voodoo, (1996 г.) и вплоть до 2001 года в GPU был реализован только фиксированный набор операций над входными данными. У программистов не было никакого выбора в алгоритме визуализации, и для повышения гибкости появились шейдеры — небольшие программы, выполняющиеся видеокартой для каждой вершины либо для каждого пиксела. В их задачи входили преобразования над вершинами и затенение — расчет локального освещения в точке, например по модели Фонга. Начиная с этого момента видеокарты перестали называть просто ускорителями и появился термин графические процессоры (Graphics Processing Unit, GPU) и на сегодняшний день графические процессоры представляют собой чрезвычайно сложные и высокопроизводительные системы с широкими возможностями. Опуская исторические моменты эволюции GPU и их программно-аппаратной модели, рассмотрим далее современные GPU и их возможности.

### **2.3.1. Аппаратная модель современных GPU**

Чтобы понять причину возникновения программно-аппаратной модели GPU, необходимо обратиться к современным методам построения центральных процессоров в частности и вычислительных систем вообще. Основы построения современных вычислительных устройств достаточно подробно изложены в [71]. Рассмотрим далее одну из наиболее значительных трудностей при создании конвейерных вычислительных устройств. Допустим, имеется простой конвейерный процессор со следующими стадиями в конвейера:

1. F - Fetch (Выборка команды из памяти).

2. D - Decode (Декодирование команды и выборка данных из регистрового файла).
3. X1,X2,X3 - eXecute; (3 стадии выполнения команды - допустим мы имеем дело с простой реализацией плавающей точки).
4. W - Write Back (запись данных обратно в регистровый файл).

При том что латентность (время выполнения в конвейере) каждой инструкции в такой системе 6 тактов, пропускная способность - 1 инструкция за такт. Это хорошо известное свойство конвейерных систем. Однако, конвейер позволяет достичь производительности в 1 операцию за такт только при условии независимости команд друг от друга. Рассмотрим ситуацию (листинг 2.1), в которой условие независимости нарушается и построим для такой ситуации диаграмму прохождения по конвейеру (листинг 2.2). Будем считать что в процессоре реализована передача операндов непосредственно на X1 с выходного регистра стадии X3 (bypassing или forwarding [71]). Поскольку умножение (*fmul*) выполняется в конвейере 3 такта (по условию), следующая команда (*fadd*) вынуждена ожидать значения операнда R1 на стадии декодирования, останавливая весь конвейер.

```
fmul R1, R2, R3
fadd R0, R1, R4
fsub R6, R0, R5
```

Listing 2.1. Код вызывающий задержки в конвейере

такт	0	1	2	3	4	5	6	7	8	9	10	11
fmul	F	D	X1	X2	X3	W						
fadd		F	D	D	D	X1	X2	X3	W			
fsub			F	F	F	D	D	D	X1	X2	X3	W

Listing 2.2. Диаграмма прохождения команд CPU по конвейеру с задержками

Нетрудно посчитать, что в случае 3-ёх стадийного конвейера и непрерывного потока зависимых инструкций производительность упадет в 3 раза. Современные центральные процессоры могут иметь от 10 до 20 стадий в конвейере и, как следствие, падение производительности будет значительно более существенным. Проблема задержек в конвейере актуальна не только для вычислительных инструкций но также и для операций с памятью (причем даже в большей степени). Поиск в древовидных структурах на современных центральных процессорах в действительности мало-эффективен именно по этой причине. Каждый кэш промах может стоить сотни тактов простоя. А при большом размере дерева кэш промахи практически неизбежны. Важно отметить, что алгоритмы вычисления глобальной освещенности рассмотренные в этой и предыдущих главах используют такой поиск чрезвычайно часто и именно он, как правило, является узким местом всей системы.

Рассмотрение способов решения данной проблемы задержки конвейера и ликвидации простоев в современных центральных процессорах выходит за рамки литературного обзора данной работы. Однако их необходимо перечислить. Это такие механизмы как внеочередное выполнение команд (out of order execution), кэши (в том числе и неблокирующие - out of order memory system), явный параллелизм инструкций (EPIC) [2.10](#). Эти механизмы позволяют в некоторой степени амортизировать или решить проблемы зависимости по данным. Однако, они как правило чрезвычайно требовательны к ресурсам кристалла (кроме EPIC) и крайне сложны в реализации.

Известно, что GPU выполняют потоки в SIMD группах, называемых в термине warp [\[73\]](#). Однако, эта особенность не столь важна для понимания механизма, позволяющего избавиться от задержек в GPU. Оставим пока эту особенность в стороне и рассмотрим как проблема задержек и простоев решается в GPU. В графических процессорах мультипроцессор (вычислительное ядро) всегда выполняет множество потоков одновременно (вернее групп по-

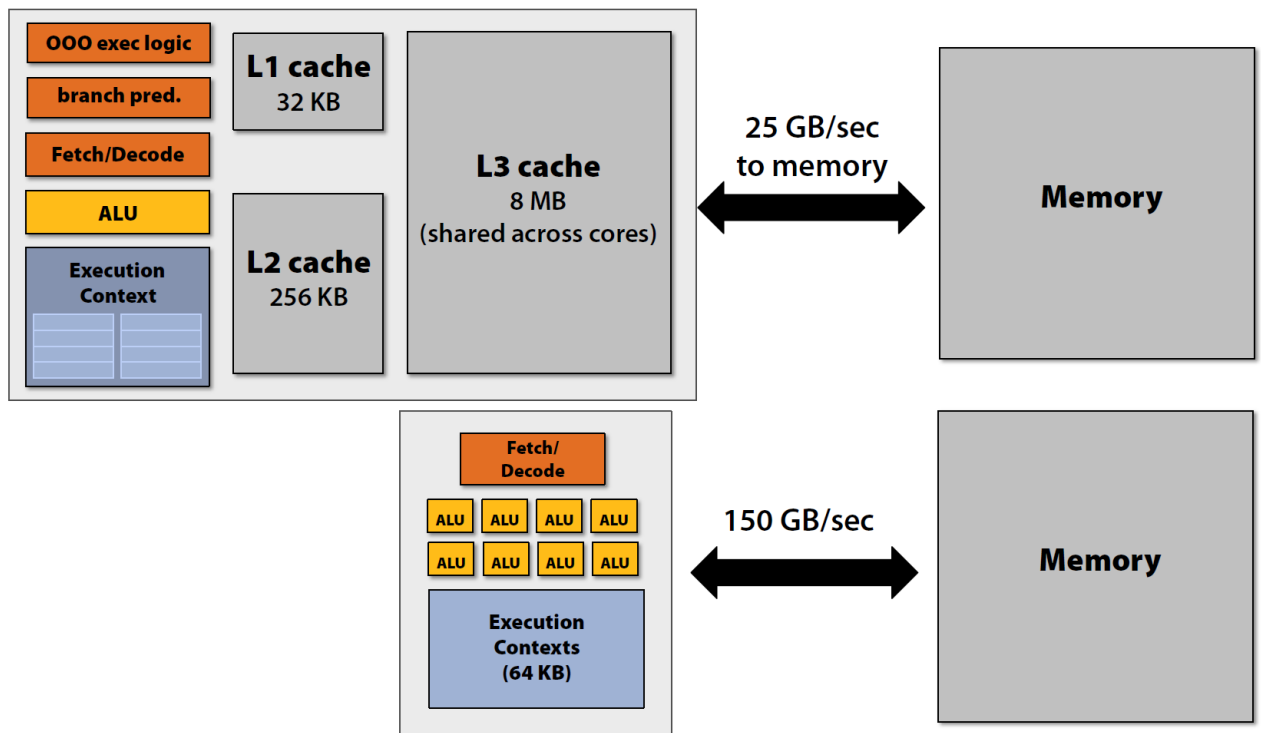


Рис. 2.10. Соккрытие латентности памяти на CPU и GPU достигается при помощи разных механизмов [72].

токов, но для рассматриваемого модельного случая будем считать пока, что размер группы - 1 поток). То есть, можно условно считать что процессор имеет достаточно активных потоков, готовых к выполнению. Будем считать, что в рассматриваемом модельном случае GPU выполняет всего 3 потока одновременно (3 группы по 1 потоку). При этом, будем помещать в конвейер команды по очереди из каждого потока - сначала из первого, затем из второго, потом из третьего и, наконец, снова из первого. Тогда диаграмма из листинга 2.2 превратиться в диаграмму 2.3 и задержки будут устранены.

такт	0	1	2	3	4	5	6	7	8	9	10	11	12	13
fmul(1)	F	D	X1	X2	X3	W								
fmul(2)		F	D	X1	X2	X3	W							
fmul(3)			F	D	X1	X2	X3	W						
fadd(1)				F	D	X1	X2	X3	W					
fadd(2)					F	D	X1	X2	X3	W				
fadd(3)						F	D	X1	X2	X3	W			
fsub(1)							F	D	X1	X2	X3	W		

<code>fsub(2)</code>	F	D	X1	X2	X3	W
<code>fsub(3)</code>	F	D	X1	X2	X3	W

Listing 2.3. Диаграмма прохождения команд GPU по конвейеру без задержек

Один мультипроцессор современных коммерческих GPU способен выполнять до 2048 потоков одновременно (64 группы warp по 32 потока) [73]. За счет наличия такого большого числа активных потоков, GPU способны скрывать при помощи рассмотренного механизма не только латентность вычислительных операций, но и латентность доступа к памяти (рисунок 2.10). При этом, хотя кэши в GPU существуют, они, во-первых не являются столь необходимыми как в случае CPU, а во-вторых, работают по иному принципу, стараясь в большей степени разделять данные между потоками, используя пространственную когерентность данных (2D и 3D текстур) и алгоритмов.

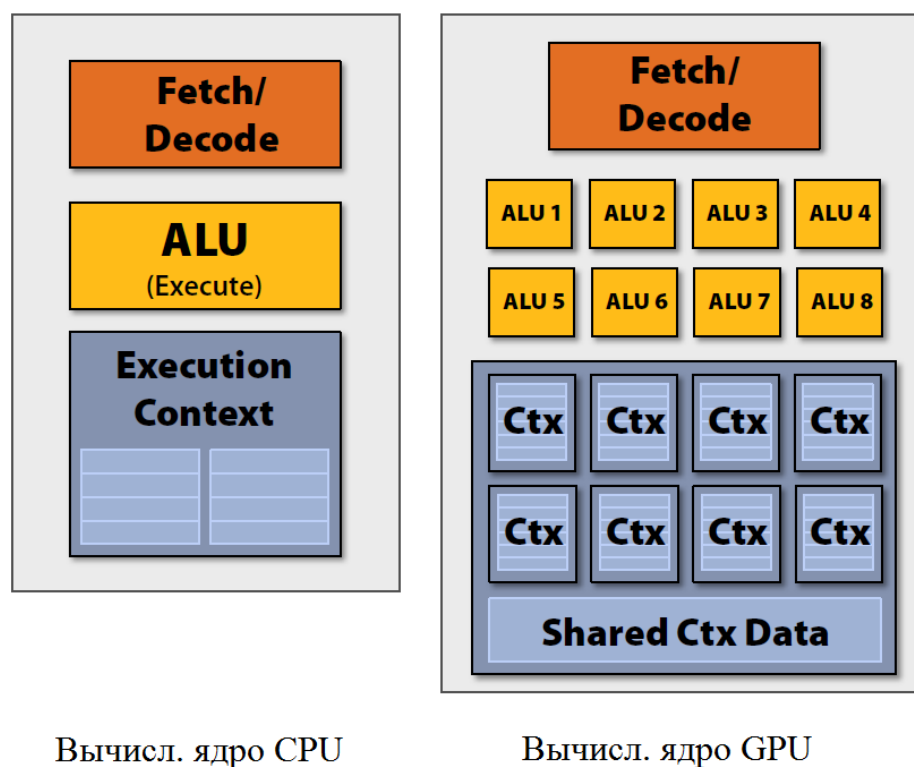


Рис. 2.11. Различия в аппаратной модели вычислительных ядер центральных (CPU, слева) и графических (GPU, справа) процессоров [72].



Теперь рассмотрим организацию типичного мультипроцессора GPU (рис. 2.11). В каждый определенный момент времени мультипроцессор способен выполнять на всех ядрах только одну инструкцию. При этом, Арифметико Логические Устройства (АЛУ) мультипроцессоров могут быть как скалярными так и векторными. Необходимость в явной поддержке векторных операций и параллелизма на уровне инструкций отпадает, поскольку мультипроцессор имеет достаточно независимых команд благодаря большому числу независимых потоков. Показательным примером является отказ от VLIW архитектуры в новейших графических процессорах компании AMD с архитектурой GNC.

Вернемся к вопросу об организации вычислений группами. Объединение потоков в группы (warp), обозначенное ранее, позволяет дополнительно экономить ресурсы кристалла, отводя больше места на функциональные устройства. Однако это далеко не главная причина объединения потоков в группы. Гораздо более важно, что такое объединение позволяет объединять много запросов в память в один большой синхронный запрос (coalescing [73]). Такая модель позволяет работать подсистеме памяти значительно более эффективно [71].

В каждый определенный момент времени мультипроцессор способен выполнять на всех ядрах только одну инструкцию, возникает вопрос - как быть с ветвлениями? Ведь если в коде программы встречается ветвление, то инструкции будут уже разные. Здесь применяется следующее решение (рис. 2.12). При расхождении на ветвлениях сначала маскируются потоки, неактивные внутри ветки if (рис. 2.12) и происходит выполнение этой ветки. После чего, замаскированные потоки становятся активными и маскируются потоки, для которых была выполнена ветка if. Таким образом, мультипроцессор выполнит обе ветки кода. Эта особенность является достаточно важной, поскольку потери на ветвления могут быть значительными (в среднем от 2 до

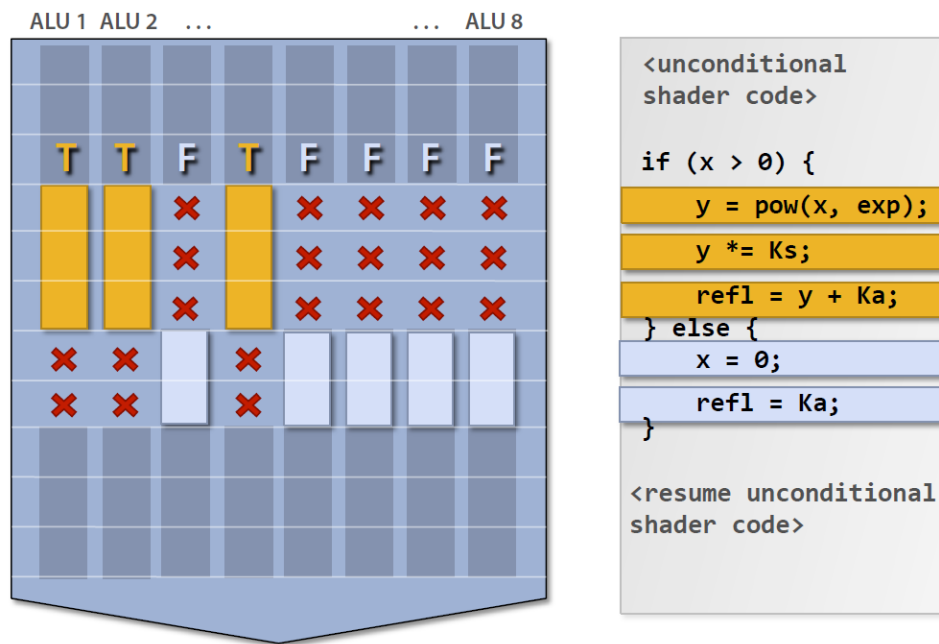


Рис. 2.12. Выполнение кода с ветвлением на GPU [72].

4 раз). При создании программ для GPU, эту особенность необходимо учитывать и по-возможности не допускать расхождения потоков из 1 группы warp по различным участкам кода.

Еще одной важной особенностью GPU является то, что большая часть локальных переменных программы хранится в регистрах, а сами регистровые файлы гораздо больше чем регистровые файлы центральных процессоров. Тем не менее они имеют ограниченный объем и чем меньше локальных переменных использует программа, тем больше потоков может одновременно выполнять мультипроцессор GPU [73]. А это число, как было описано выше, напрямую влияет на способность мультипроцессора скрывать латентность вычислительных операций и, что гораздо более важно, латентность подсистемы памяти. В связи с этой особенностью, важно избегать объемного кода с большой степенью вложенности процедур и большим числом локальных переменных, поскольку компилятор для GPU старается встраивать (inline) код, заменяя вызов процедуры на тело самой процедуры. Эффективность встраивания падает с ростом глубины вызовов и сложности программного

кода.

Отдельно следует сказать и о некоторых непрограммируемых блоках GPU, имеющих важное значение в программировании общего назначения. Это такие блоки как:

1. ROP - блоки графического процессора, выполняющие Z-буферизацию, полноэкранное сглаживание и запись итогового изображения (с алфасмешиванием или без) в кадровый буфер в видеопамяти. Используется в программировании общего назначения для реализации атомарных операций [74]. Справедливости ради следует сказать, что современные центральные процессоры, как правило, также имеют специальную логику для реализации атомарных операций.
2. Разделяемая память и кэш второго уровня L2. Изначально предназначались для передачи данных между стадиями графического конвейера и кэширования текстур, но могут быть использованы для сообщения данными между потоками одного блока (обычно группа из 64-512 потоков) [73].
3. Тектурные блоки, позволяющие осуществлять не просто чтение данных, но и фильтрацию, преобразование форматов и декомпрессию текстур на аппаратном уровне на лету. К сожалению, в API программирования GPU общего назначения CUDA и OpenCL возможно использование только ограниченного подмножества их функциональности.
4. GMU - блок, появившийся в архитектуре GPU компании Nvidia - Kepler. Этот блок ответственен за механизм создания работы на GPU (GPU Work Creation, GWC) и позволяют осуществлять вызов ядер из самих ядер [75].

5. RDMA (Remote Direct Memory Access) блоки - составляющая GPU, осуществляющая возможность прямого доступа (без участия CPU) к памяти GPU из других устройств [75]. Как правило, используются для передачи данных между несколькими GPU.

Завершая обзор архитектуры GPU, можно сказать, что GPU - это система, ориентированная на максимальную пропускную способность операций. При этом, не важно как долго выполняется каждая отдельная операция, важно только конечное время, поделенное на число выполненных за это время операций.

### **2.3.2. Программная модель современных GPU**

Программная модель современных GPU основана на концепции параллелизма по данным, называемым иначе - ОПМД (Одна Программа Множественные Данные) или SIMT (Single Program Multiple Data). Центральный процессор (а в случае процессоров Kepler и сам GPU) асинхронно создает очереди запросов на выполнение небольших программ, называемых ядрами (kernel [73]). Запросы извлекаются из каждой очереди по одному (но возможно параллельное извлечение из нескольких очередей - конкурентное выполнение ядер [73]) и выполняются на GPU.

Причем, GPU имеют 2 различных режима работы, в которых они могут выполнять:

1. Команды отрисовки графических примитивов на основе графического конвейера (графический режим).
2. Команды на запуск вычислений общего назначения (вычислительный режим).

Команды копирования данных доступны в обоих режимах, хотя иногда выделяются в отдельный тип команд.

## OpenGL 4.3

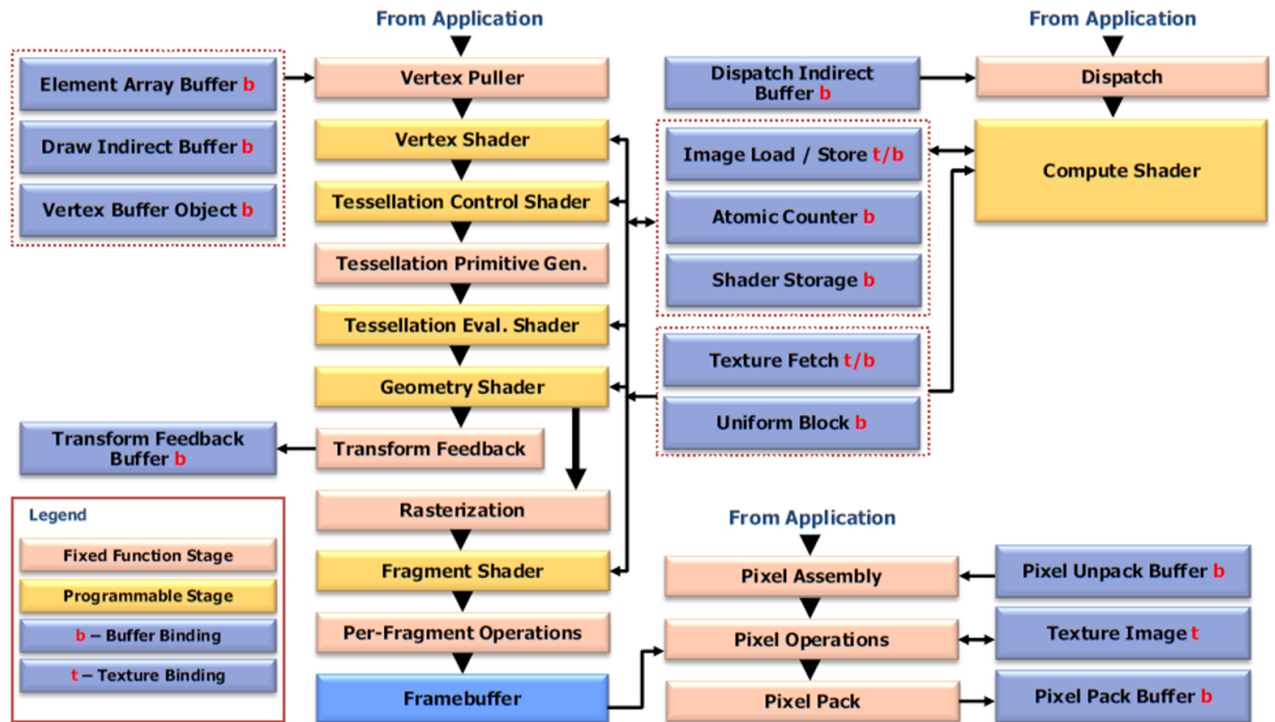


Рис. 2.13. Современный стандарт OpenGL 4.3; В левой части изображения структура API на основе графического конвейера, в правой - структура API для вычислений общего назначения.

Соответственно этим 2 режимам, существуют 2 программные модели GPU и 4 распространенных API (Application Programming Interface) - (CUDA и OpenCL) для вычислений общего назначения и (OpenGL и DirectX) для программирования графического конвейера. В настоящий момент графические API, за исключением специфических для производителя GPU вещей (например создание работы непосредственно на GPU в CUDA), являются значительно более мощными и в смысле функциональности использования возможностей GPU могут считаться надмножеством API вычислений общего назначения. На рисунке 2.13 обозначена структура графического API OpenGL 4.3,

в котором с правой стороны находятся вычислительные шейдеры (Compute Shaders), предназначенные для реализации вычислений общего назначения средствами API OpenGL 4.3. Аналогично, API DirectX имеет свои вычислительные шейдеры.

Рассмотрим далее принципы использования таких API как CUDA и OpenCL. Каждый запрос на выполнения вычислительного ядра запускает множество копий программы-ядра на GPU. При этом, потоки имеют трехмерный (в общем случае) индекс и объединены в группы, называемые блоками потоков (в OpenCL такой блок называется рабочей группой). Совокупность всех блоков называется сеткой (grid).

На рисунке 2.14 изображена такая организация с использованием двумерного индекса; третье измерение не используется. Каждый блок потоков гарантированно выполняется на 1 мультипроцессоре (хотя 1 мультипроцессор может выполнять несколько блоков), за счет чего возможна передача данных между потоками в пределах одного блока через быструю разделяемую память, физически расположенную близко к мультипроцессору. Двумерная или трехмерная организация, в свою очередь, позволяет сделать обработку двумерных и трехмерных данных более естественной. Процесс вычислений можно представить в виде следующей последовательности действий:

1. Выделить память для входных и выходных данных на GPU.
2. Скопировать входные данные из памяти CPU в память GPU.
3. Запустить 1 или более вычислительных ядер.
4. Дождаться окончания выполнения всех ядер путем явной или неявной синхронизации (происходит при вызове функций копирования данных на сторону CPU).

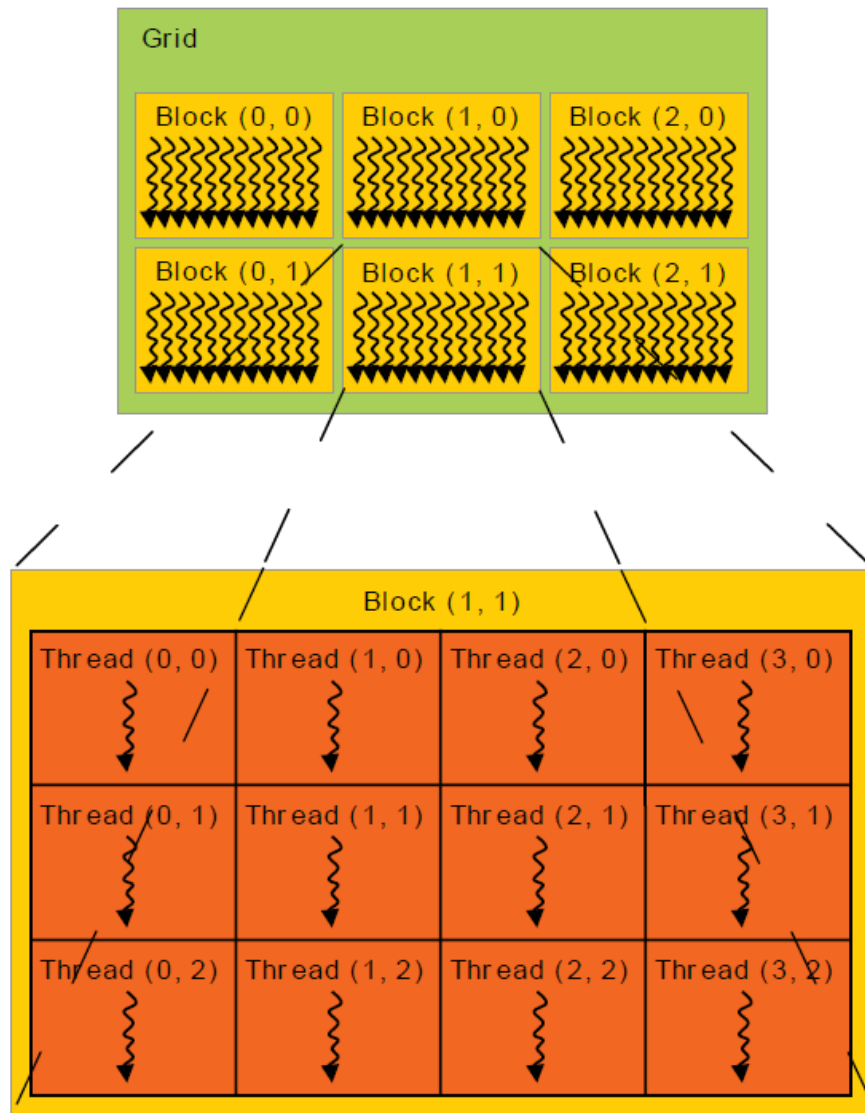


Рис. 2.14. Организация потоков в группы в двух измерениях.

5. Скопировать выходные данные обратно на сторону CPU либо предать их в область видимости графических API для дальнейшей визуализации. Повторять шаги 3-5 до тех пор пока необходимо что-то вычислять.
6. Если работа окончена, освободить ранее выделенную память.

Как правило, каждый отдельный поток на GPU независимо обрабатывает небольшое (по этой причине такой вид параллелизма иногда называют 'мелкозернистым') число элементов данных. Например, при простейшей реализации разностной схемы каждый поток будет вычислять значение одной ячейки

для одной итерации схемы. При реализации трассировки лучей, один поток будет обрабатывать единичный луч. При расчете столкновений один поток будет отвечать за 1 физический объект.

Более сложные алгоритмы предполагают синхронизацию/взаимодействие между группой потоков и/или всеми потоками сетки и должны быть реализованы в нескольких ядрах. Такие алгоритмы могут быть реализованы при помощи набора более простых операций - редукции, уплотнения, сортировки. Эти операции допускают эффективную программную реализацию на GPU с использованием разделяемой памяти и последовательным запуском ядер, каждое из которых реализует часть алгоритма. Достаточно простым примером здесь является реализация поиска минимума некоторой функции  $f$  на некотором множестве значений  $data$ . Сначала каждый поток вычисляет свое значение функции  $f$  для заранее определенных входных данных  $data[i]$ . Где  $i$  - идентификатор потока.  $data$  - массив данных. После чего минимум среди всех значений может быть найден при помощи операции редукции. Следует отметить, что частичная редукция (внутри одного блока потоков) при этом может совершаться в том же ядре, которое вычисляло значение функции  $f(data[i])$ . Еще один пример - построение дерева. Если алгоритм построения дерева удастся свести к сортировке (причем, очень желательно - к сортировке 32 или 16 битных чисел, для которых хорошо работает поразрядная сортировка), тогда построение такого дерева может быть эффективно реализовано на GPU. В соответствие с рассмотренной выше моделью, для того чтобы алгоритм мог быть эффективно реализован на GPU он должен удовлетворять следующим условиям:

1. На каждом шаге алгоритм должен иметь достаточно большое число относительно-простых независимых операций. Каждая такая операция будет назначена на отдельный поток. Под термином 'достаточно' здесь



понимается столько операций, сколько необходимо для полной загрузки GPU работой. Под 'относительной простотой' понимается отсутствие рекурсии, ограниченное число локальных переменных и небольшой уровень вложенности функций, при котором компилятор способен эффективно распределить локальные переменные по регистрам GPU.

2. Зависимости по данным между различными операциями необходимо сводить к минимуму. Хорошим случаем является такой, при котором различным потокам нужно обмениваться данными только локально - внутри своей группы. В случае когда это сделать невозможно, алгоритм должен сводиться к набору примитивных операций - редукции, уплотнения, сортировки.
3. Если значимая часть вычислений выполняется на CPU, желательно реализовывать алгоритм в виде конвейера, чтобы операции копирования данных и вычисления на CPU перекрывались по времени с вычислениями на GPU.

## **API программирования GPU**

Хотя графические API являются более мощными, трудоемкость разработки программ с их использованием значительно выше по целому ряду причин:

1. Графические API, как правило, работают с GPU на более низком уровне, чем API для вычислений общего назначения. Например, при создании буфера или текстуры в DirectX программист обязан при помощи флагов самостоятельно указывать, как данный ресурс будет использован - для записи, чтения, копирования с CPU на GPU или с GPU на CPU. Если нужный флаг (например на запись) не был установлен, при попытке записи в такой ресурс будет сгенерирована ошибка. Во многих случаях

такой детальный контроль над ресурсами не нужен, а лишь вынуждает программиста тратить время на работу с несущественными деталями.

2. Поскольку графические API призваны в первую очередь обеспечивать доступ ко все функциональности GPU, они значительно более объемные. При этом, DirectX и OpenGL имеют различные недостатки:

а. Помимо того, что DirectX поддерживается только в операционной системе Windows, его главный недостаток - ограниченное время жизни. DirectX всегда создавался под определенное поколение графических процессоров. При выходе нового поколения GPU появлялась новая версия DirectX. При этом, каждая версия DirectX жестко фиксирует набор используемых функциональных возможностей GPU. Чтобы использовать новую функциональность GPU, необходимо использовать другую версию API. Это выливается в необходимость переписывания исходного кода под новые версии API, которые могут различаться достаточно сильно (особенно DirectX 9 и 10 версии). При этом, если в GPU появилась какая-то новая функциональность, а новая версия DirectX еще не вышла - использовать эту возможность разработчик просто не сможет.

б. Основной недостаток OpenGL - слабая поддержка на уровне драйвера, особенно касательно компиляторов шейдерных программ. Поскольку шейдерные программы компилируются на машине пользователя, при разработке необходимо не только строго следовать стандарту, но еще и использовать минимальное подмножество языка GLSL (язык шейдеров в OpenGL) и функциональности самого OpenGL. Основная причина такой необходимости заключается в том, что разные производители (и даже разные версии драйверов

одного и того же производителя!) немного по-разному трактуют стандарт и не совсем строго ему следуют. Эта проблема знакома всем кто разрабатывал программы на C++ под различные компиляторы. Однако, если в случае компиляторов C++ проблемы несовместимости могут быть устранены на этапе сборки программы, то в случае шейдеров OpenGL они возникнут на машине пользователя. Справедливости ради следует сказать, что такая проблема характерна и для OpenCL. Однако, вследствие меньшей функциональности самого API, для OpenCL это не так значимо.

Корнем проблем всех API программирования GPU является противоречивость требований, предъявляемых к ним - переносимость с одной стороны и доступ к широкой функциональности GPU + достижение высокой эффективности - с другой. API для вычислений общего назначения CUDA является проприетарным API от компании NVIDIA. Среди его достоинств необходимо отметить:

1. Стабильную поддержку на уровне драйвера и заранее известный набор аппаратуры, на которой может быть запущена программа. Это упрощает разработку и оптимизацию программного кода.
2. Статическую компиляцию ядер (шейдеров) в бинарное представление. Компиляция осуществляется на машине разработчика.
3. Гомогенность кода разрабатываемой системы. CUDA компилятор использует расширение языка C++ и поддерживает значительную часть стандарта C++.
4. Наличие значительного числа библиотек.

- а. thrust - библиотека, реализующая базовые примитивы параллельного программирования (Параллельную сортировку, уплотнение, вычисление префиксной суммы).
- б. CUBLAS, MAGMA, cuSPARSE, CUSP, HiPLAR - реализация базовых операций линейной алгебры.
- в. CURAND - реализация генераторов псевдо-случайных и квази-случайных чисел.
- г. cuFFT - реализация быстрого преобразования фурье
- д. NPP - библиотека для обработки изображений.
- е. Другие библиотеки, предназначенные для специализированных задач [76].

При этом, существование таких проектов как PGI CUDA-x86 [77] и GPUOcelot [78] позволяет говорить о переносимости в перспективе.

## 2.4. Алгоритмы на массивно-параллельных процессорах

Особенности массивно-параллельных процессоров и массивного параллелизма самого по себе делают во многих случаях невозможным применение алгоритма, разработанного для центрального процессора один в один. Более того, даже если это возможно, в целях лучшей производительности алгоритм приходится сильно модифицировать. Метод Монте-Карло трассировки путей как раз является примером такого алгоритма. Несмотря на то, что каждый отдельный путь можно обрабатывать независимо, для достижения максимальной (или хотя бы приемлимой) эффективности необходимо произвести ряд модификаций алгоритма (модификации рассмотрены ниже). В связи с этим, исследования в области решения проблемы глобальной освещенности на массивно-параллельных системах можно условно поделить на 2 части:

1. Исследования о том, как эффективно реализовать некоторый известный алгоритм на GPU, если принципиальные трудности по его реализации отсутствуют, однако, по каким-либо причинам, оцениваемая производительность не достигается. Будем условно называть данную группу исследований 'исследованиями производительности алгоритмов на GPU'.
2. Исследования о том, как реализовать на массивно-параллельной системе алгоритм, изначально спроектированный для однопроцессорной системы. Такой алгоритм в силу своей природы не может быть напрямую реализован на массивно-параллельной системе и требует переработки в самих основах. Будем называть эту группу 'исследованиями по распараллеливанию алгоритмов на GPU' или 'исследованиями по созданию высоко-параллельных алгоритмов'.

Данная работа в большей степени относится ко второй группе исследований. Однако, вопрос эффективной реализации важен и исследования производительности с учетом архитектурных особенностей GPU также затрагиваются.

#### **2.4.1. Трассировка лучей на GPU**

Ранние работы по реализации трассировки лучей на GPU исследовали проблему эффективной реализации без-стекового поиска в ускоряющих структурах в условиях ограниченных возможностей графических API DirectX 9 и OpenGL2 и их аппаратных реализаций [79], [80], [81].

Можно легко ошибиться в предположении, что с появлением новых технологий и GPU с более широкими возможностями и ресурсами задача реализации трассировки лучей на современных GPU тривиальна, поскольку достаточно лишь назначить каждому потоку свой луч и портировать существующий CPU код. В действительности это не так. Портированный 'как есть' код бу-

дет обладать чрезвычайно низкой эффективностью вследствие особенностей выполнения кода на GPU, а именно [82]:

1. Локальные переменные компилятор помещаются по возможности в регистры GPU. Мультипроцессор GPU имеет ограниченное число регистров, причем вычисления организованы таким образом, что число активных потоков (занятость мультипроцессора) и эффективность выполнения кода на GPU напрямую зависят от количества регистров, занимаемых ядром. Чем больше регистров занято после определенного порога, тем ниже эффективность. CPU код, изначально рассчитанный на стековую архитектуру процессора в этом случае будет чрезвычайно неэффективным.
2. Эффективность подсистемы памяти также зависит от числа активных потоков на мультипроцессоре, поскольку механизм сокрытия латентности доступа к памяти основан на выполнении большого числа потоков параллельно.

Указанные проблемы с портированием возникают не для любого кода, а только для достаточно сложного кода с высоким уровнем вложенности функций, большим числом локальных переменных, ветвлений и объемной функциональностью. Для методов на основе трассировке лучей эта проблема характерна вследствие большого разнообразия материалов, источников света, структур пространственного разбиения и, в некоторых случаях, геометрических объектов.

В работе [82] выделены 2 способа решения, называемые "убер-ядро" (uber-kernel или mega-kernel, и "разделенное ядро" (separate kernel). При реализации способа "убер-ядро" исходный код логически разбивается на состояния конечного автомата, в котором каждое состояние - некоторая отдельная, достаточно тяжелая часть кода. Во время выполнения, происходит периодическая смена

состояний и переход между соответствующими ветками кода, сохраняя некоторые важные данные в разделяемой или локальной памяти. Суть такого подхода в том, что он позволяет использовать одни и те же регистры для разных переменных.

Однако в [82] отмечаются недостатки данного подхода, среди которых неэффективное использование ресурсов, потери производительности на ветвлениях (когда одна часть группы `war` находится в одном состоянии, а другая - в другом), сложность поддержки и отладки.

### **Конвейер трассировки лучей**

Проблемы подхода убер-ядра обсуждаются позднее в [24] и [83]. В работах [82] и [24] предлагается использовать подход 'разделенного ядра', формируя и настраивая конвейер трассировки лучей за счет замещения и/или вставки ядер на различных этапах конвейера. Этот способ предполагает реализацию различных частей в отдельных ядрах и организацию взаимодействия между ними путем сохранения данных в глобальной памяти. Работа [83] развивает данный подход на основе очередей на выполнение работы (каждая очередь запускает на выполнение только одно конкретное ядро), сортировки потоков по номеру ядра и механизма создания работы на GPU.

#### **2.4.2. Монте-Карло Трассировка путей на GPU**

Несмотря на тот факт, что Монте-Карло трассировка достаточно легко может быть реализована на GPU, ее эффективная реализация вызывает ряд определенных проблем, связанных с природой графических процессоров.

**Контроль глубины трассировки.** Различные потоки заканчивают трассировку на различной глубине. Это приводит к тому, что через некоторое количество переотражений внутри группы потоков `war` лишь небольшое число

потоков остаются активными. Для наглядности можно переформулировать эту проблему следующим образом: Если в группе потоков warp есть хотя бы один поток, которому необходимо трассировать путь на некоторой глубине  $k$ , то группа продолжает находиться на мультипроцессоре и все ее неактивные потоки выполняются вхолостую.

В работе [84] представлено простое и эффективное решение данной проблемы, называемое русской рулеткой на warp (per warp russian roulette). Основная идея данного метода заключается в том, чтобы применять метод русской рулетки не к каждому потоку отдельно (как было рассмотрено в 1 главе), а сразу ко всей группе потоков warp. Решение о терминировании, таким образом, принимается сразу для всей группы, что исключает возникновение групп warp с малым числом активных потоков при терминировании лучей посредством русской рулетки.

Для решения проблемы неоднородной загрузки GPU из-за различной глубины терминирования потоков, в работах [84] и [24] исследовался метод повышения эффективности использования GPU, называемый регенерацией путей. Сообщается об ускорении от 20 до 100% для случаев, требующих учет большого числа переотражений. Метод был впервые предложен в [85] и исследован на стандартной трассировке путей, двунаправленной трассировке путей и алгоритме MLT. В работе [85] сообщается об ускорении от 20 до 50%. Метод работает следующим образом:

1. Для трассируемых лучей выделяется буффер, вмещающий в себя данные ровно для  $N$  лучей.
2. При каждом переотражении производится операция уплотнения с тем чтобы отсечь мертвые потоки.
3. В освободившееся место добавляются новые лучи с тем чтобы загрузить



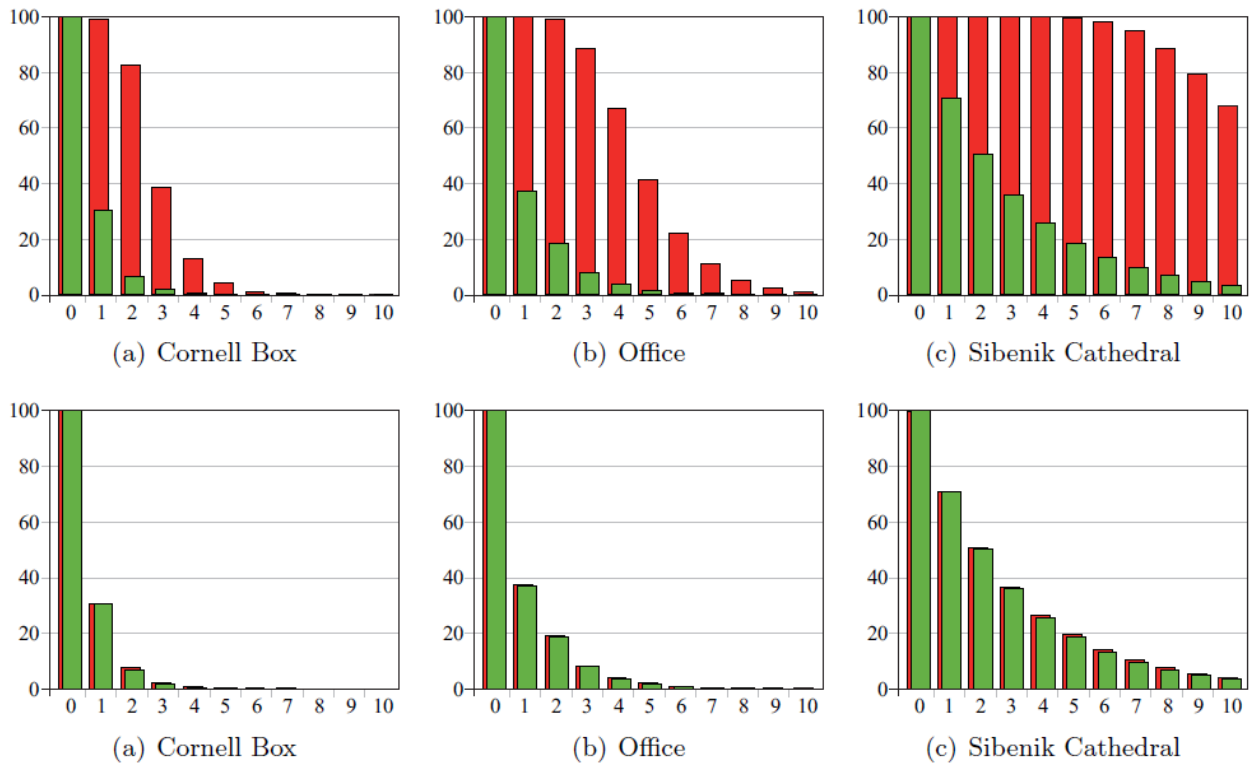


Рис. 2.15. Применение на различных сценах русской рулетки на warp из работы [84]. Красные столбцы отражают число потоков, находящихся на GPU. Зеленые столбцы отражают число активных, выполняющих полезную работу, потоков. Верхние 3 изображения сделаны для обыкновенной русской рулетки на поток. Нижние 3 - для русской рулетки на группу потоков warp.

GPU работой.

Несмотря на указанные преимущества в скорости, метод регенерации путей не лишен недостатков. Во-первых, он вынуждает применять уплотнение, которое занимает определенное время. Во-вторых, он нарушает когентность лучей в буфере что может привести к падению производительности.

**Дивергентные потоки.** По сравнению с трассировкой лучей, трассировка путей обладает значительно большей дивергентностью потоков, поскольку при диффузном отражении лучи расходятся по полусфере в различные стороны. Это приводит к неэффективному использованию мультипроцессоров

и, как следствие, низкой производительности.

В работе [86] описывается подход, называемый когерентной трассировкой путей. Идея данного подхода заключается в том, чтобы использовать одно состояние случайного генератора для всех лучей (в группе варп, блоке или вообще всех на экране).

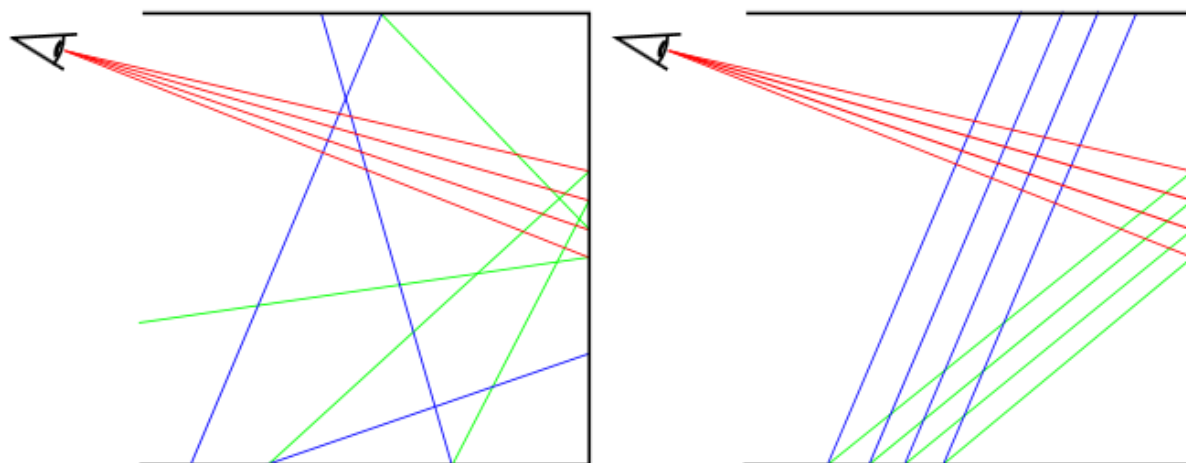


Рис. 2.16. Стандартная трассировка путей (слева). Когерентная трассировка путей (справа). [86].

Данный подход является эффективным но меняет характер артефактов, превращая шум в более заметные для глаза артефакты с резкими границами. Для того чтобы убрать границы в [86] применяется случайное перемешивание к схеме нумерования пикселей.

### 2.4.3. Параллельный Кэш освещенности

Реализация параллельного кэша освещенности даже на центральном процессоре сопряжена с рядом трудностей.

1. Первая проблема (проблема синхронизации) связана с обеспечением консистентности данных кэша - каждый поток должен видеть один и тот же экземпляр данных кэша.

2. Вторая проблема (последовательная природа) связана с тем, что алгоритм вычисляет расположение записей кэша и их радиусы валидности на основании эвристических методов, опираясь на то, что точки вставляются в кэш последовательно, одна за одной [40].

Один из простых способов решения проблемы синхронизации, используемый в [mitsuba, проверить] и [67] заключается в том, чтобы не синхронизировать данные кэша между потоками вообще. Первый недостаток этого подхода состоит в том, что на изображении могут появиться видимые границы между областями, которые были использованы разными потоками т.к. каждый поток будет иметь доступ к своему уникальному подмножеству записей кэша - и, как следствие, в одной той же точке трехмерного пространства для разных потоков результат интерполяции будет различен. Вторым недостатком заключается в том, что при увеличении числа потоков будет возрастать объём избыточных вычислений, поскольку каждый поток будет вычислять полностью независимый набор записей, а их перекрытие никак не контролируется.

Следующий способ решения проблемы синхронизации - использование разделяемой структуры данных в том или ином виде. Здесь возможны различные варианты - синхронизация при вставке единичной записи [87], синхронизация на основе слияния групп [88] и более сложные подходы - [89], [90].

В работе [89] предложен подход, использующий пространственную когерентность кэша освещенности для реализации алгоритма на кластерных системах. Все узлы кластера разбивались на 2 подмножества - узлы, занимающиеся вычислением записей кэша и узлы, отвечающие за построение изображения в целом. За счет использования такого разделения и высокой пространственной когерентности операций чтения из кэша, авторам [89] удалось сохранять частоту обновления кэша на высоком уровне без существенных потерь в производительности.

Работы [87], [88] и [89] используют для синхронизации критические секции. Корректная реализация с использованием критических секций блокирует как запись в кэш, так и чтение из него. Это негативно сказывается на масштабируемости реализации при увеличении числа параллельно исполняющихся потоков [90]. В работе [90] предлагается подход к построению разделяемой структуры кэша освещенности на основе атомарных операций современных CPU. За счет использования атомарных операций в работе [90] удастся избежать блокирования при чтении из кэша освещенности.

Наконец, еще один способ решения проблемы синхронизации заключается в том, чтобы отделить процесс построения кэша освещенности от процесса его использования и разделить, таким образом, процесс построения изображения на 2 стадии:

1. Стадия построения кэша освещенности. На данной стадии необходимо покрыть всё пространство сцены, куда попадают лучи из камеры записями из кэша. При этом, на данной стадии консистентность кэша не важна, поскольку результат интерполяции не используется для вычисления финального изображения.
2. Финальная стадия построения изображения. На ней новые точки в кэш освещенности не добавляются.

Идея такого подхода обсуждается в [65], [40] и [5]. Причем, в указанных работах двухстадийный алгоритм используется не для параллельной реализации, а для повышения точности интерполяции (рисунок 2.17). Ранее была рассмотрена ситуация, при которой разные потоки, имея различные подмножества записей кэша в одной и той же точке пространства будут давать различные результаты интерполяции.

В действительности эта проблема возникает даже при однопоточной реализации кэша освещенности, поскольку в некоторой точке пространства  $X$  в раз-

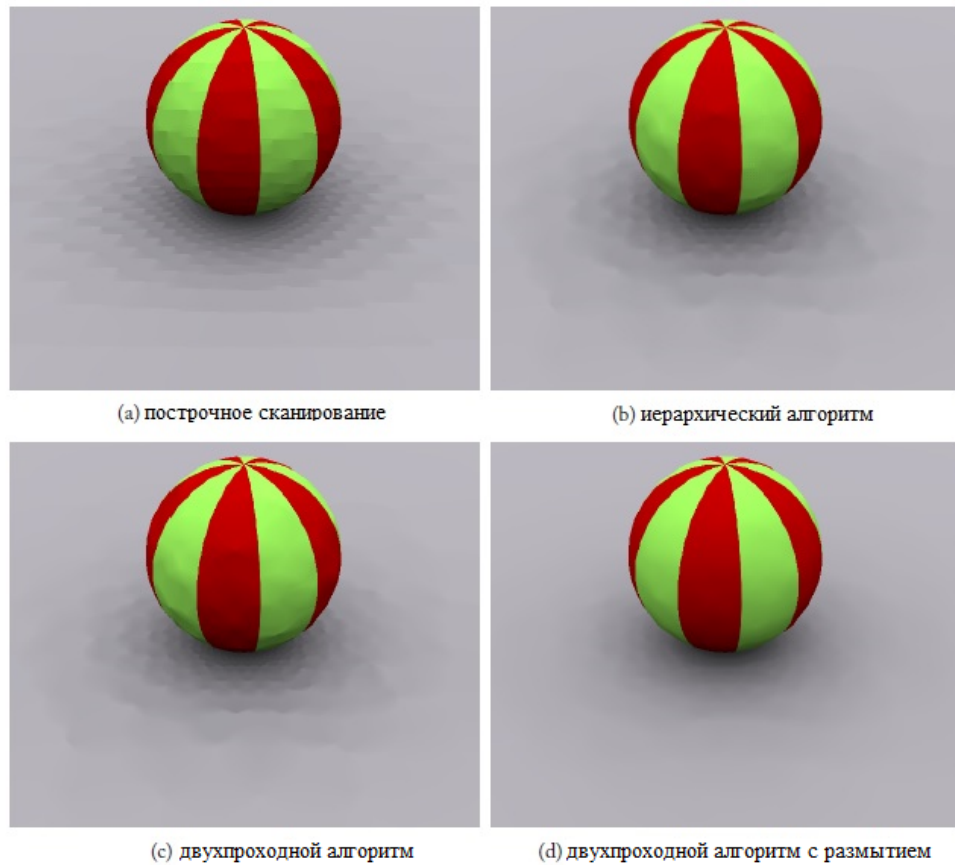


Рис. 2.17. Сравнение различных подходов построения изображения с применением кэша освещенности из работы [40]. Под 'размытием' (случай d) в данном случае понимается небольшое увеличение радиусов валидности для всех записей (в 1.6 раза).

ные моменты времени при интерполяции может быть затронуто различное число записей [40]. Это возможно так как стандартный алгоритм не может гарантировать, что в процессе построения изображения вблизи точки  $X$  не появятся новые записи, которые станут участвовать в интерполяции при следующих выборках из кэша в точке  $X$ .

Поскольку разделение процесса построения изображения на 2 стадии позволяет повысить точность, будем рассматривать данный подход как наиболее практичный способ решения проблемы синхронизации, поскольку повышение точности важно при реализации кэша освещенности. При этом, что касается проблемы последовательной природы кэша освещенности, здесь следует выделить 2 основных момента.

1. Алгоритм создания самого множества записей кэша не предполагает одновременную вставку многих точек в кэш. Причем, здесь существуют 2 проблемы:
  - а. Проблема остановки алгоритма построения кэша. Сам процесс расстановки записей реализует ленивую схему вычислений. Оригинальный алгоритм (6) вычисляет новые точки (или записи) кэша по запросу. Поэтому в нем проблема остановки не существовала. При необходимости новая точка (запись) всегда может быть вычислена. В двух-проходной схеме это не так. Процесс создания кэша освещенности должен уметь останавливаться. Реализация из [65] использует последовательный алгоритм (называемый 'Full convergence for a set of shading points'). На 1 проходе алгоритм запоминает в kd-дереве все точки, в которых потребуется осуществлять выборку из кэша (shading points). При вставке новой записи в кэш освещенности, из дерева удаляются точки (shading points), покрытые областью, на которую распространяется влияние вставляемой записи. В [40] предлагается использовать упрощение рассмотренного выше алгоритма, помечая пикселы изображения, для которых не встретилось высоких значений оценки ошибки интерполяции. Когда все пикселы помечены, можно остановить алгоритм. Хотя параллельная реализация обоих решений теоретически возможна, эта проблема требует исследования, поскольку здесь возникает все та же необходимость синхронизации структур данных, хранящих shading points или маркеры пикселей.
  - б. Проблема выбора оптимального расположения записей кэша. В рассмотренных ранее подходах, при увеличении числа одновременно вставляемых в кэш записей неизбежно падение эффективности ре-

шения в целом. Это связано с тем, что при большом числе одновременно вставляемых точек (записей), некоторое число запросов на вычисление освещенности могут возникнуть в близко-расположенных областях пространства. Такие запросы являются избыточными, поскольку в этих областях пространства последовательный алгоритм обошелся бы всего одной точкой.

2. Алгоритм вычисления радиусов валидности. Эвристика Neighbour Clamping [40], значительно повышающая точность решения на основе Кэша освещенности, предполагает последовательную вставку записей в кэш. То же верно и об алгоритме контроля плотности, описываемом в разделе 5.1 работы [65]. Отсутствие учета соседства точек при параллельной вставке большого числа независимых записей приведет к тому, что радиусы валидности записей будут слишком большими. Как будет показано в главе 3 - это, в свою очередь, ведет к высокой степени перекрытия записей и медленной интерполяции.

## **Кэш освещенности на GPU**

Впервые GPU был использован при реализации кэша освещенности в работе [91] и в дальнейшем обсуждается в [40]. Рассматриваемый в этих работах подход позволяет заменить стандартный алгоритм интерполяции на растеризацию сплатов. Это позволяет избежать поиска в дереве на GPU при финальной визуализации сцены, что было особенно для ранних GPU, не имеющих поддержку циклов. В работах [91] и [40] также предлагается использовать растеризацию на GPU при расчете освещенности в точках кэша. Метод, обсуждаемый в этих работах имеет следующие ограничения:

1. Интерполяция при помощи растеризации сплатов позволяет визуализировать вторичное освещение только на первично-видимых поверхно-

стях. Хотя существуют методы визуализации отражений на основе растеризации (и в этом случае, растеризация сплатов также применима), такие методы работают для специальных случаев и/или визуализируют отражения не совсем корректно [92]. Поэтому, при точной визуализации и/или визуализации на основе трассировки лучей, данный метод неприменим.

2. При использовании растеризации для вычисления непрямого освещения, значения освещенности сохраняемые в кэше, несут информацию только о первом переотражении. Такое решение неприемлемо, если необходимо учитывать большее число переотражений света. Хотя этот недостаток можно побороть при использовании фотонных карт в сочетании с картами светимости (irradiance maps), остается открытым вопрос об эффективности реализации такого подхода на современных GPU. Оценим эффективность этого подхода с учетом поледних данных о производительности трассировки лучей и растеризации на GPU.

а. Современные GPU могут обеспечить производительность обработки порядка 100-300 миллионов (М) лучей в секунду [14]. С учетом необходимости трассировки лучей на большую чем 1 глубину и/или необходимости выполнения финального сбора, эти числа нужно разделить как минимум на 2. Однако, с учетом того что лучи из точек кэша можно выпускать когерентными группами и что большинство из них достаточно быстро находит пересечение с ближайшими объектами, слишком сильно занижать производительность не следует. Поэтому примем за среднюю производительность 50М монте-карло выборок в секунду.

б. Современные компьютерные игры способны растеризовать сцену



со сложным первичным освещением в несколько миллионов треугольников со скоростью 100 кадров в секунду. Чтобы усилить сравнение, будем считать, что мы можем растеризовать сцену с производительностью в 500 кадров в секунду.

- в. Тогда за 1 секунду при помощи растеризации можно вычислить порядка 500 точек. При этом, если среднее число лучей на точку порядка 1000, за 1 секунду при помощи Монте-Карло трассировки лучей можно вычислить освещенность в  $(50\,000\,000/1000) = 50$  тысяч точках.

Полученный результат не должен вызывать удивления. Алгоритм растеризации для каждой точки вынужден полностью пропускать всю геометрию по графическому конвейеру и плохо подходит для вычисления освещенности на множестве точек. И хотя существуют различные способы, позволяющие в теории значительно сократить время растеризации при использовании такого подхода (например многомерная растеризация [93]), в системе расчета освещенности на основе трассировки лучей видится более выгодным использовать трассировку лучей для вычисления освещенности в точках кэша.

#### **2.4.4. Фотонные карты на GPU**

Фотонные карты на GPU – довольно популярная тема для исследований. Одна из основных трудностей при реализации фотонных карт – эффективное построение ускоряющих структур для последующего поиска ближайших фотонов. Здесь необходим метод, который бы позволял быстро находить ближайшие фотоны в заданном радиусе вокруг заданной точки трехмерного пространства. Реализация такого метода на GPU встречает следующие трудности:

1. Построение ускоряющей структуры на GPU, а особенно иерархической структуры, - нетривиальная задача сама по себе. С одной стороны необходимо уметь сводить алгоритм построения такой структуры к набору примитивных параллельных операций (сортировка, уплотнение, добавление в буфер), которые могут быть реализованы на GPU действительно эффективно. С другой стороны, необходимо избегать нерегулярного доступа в память и контролировать занимаемый алгоритмом построения дерева объем. С учетом ограниченности объема памяти GPU это особенно важно.
2. Необходимо обращать особое внимание на скорость последующего поиска в такой структуре и, как следствие, скорость операции сбора освещенности, поскольку конечная цель применения ускоряющих структур - ускорить именно операцию сбора освещенности и уменьшить время расчета изображения.
3. При реализации в промышленной системе не последнее место занимает вопрос трудоемкости реализации и вопрос поддержки исходного кода. Сложные алгоритмы в индустрии, как правило, приживаются значительно реже чем простые, поскольку в промышленных решениях, как правило, необходимо не только реализовать алгоритм ежиножды но организовать поддержку с исправлением возможных ошибок. С учетом поддержки, стоимость реализации сложных алгоритмов многократно возрастает.

В работе [94] алгоритм фотонных карт был впервые реализован на графических процессорах. Аппаратные ограничения GPU того времени не позволяли производить запись в произвольную ячейку памяти, что создавало целый ряд проблем при реализации ускоряющих структур на GPU. В [94] предложено

2 метода построения ускоряющих структур на основе регулярной сетки. Первый метод основан на сортировке фотонов по номеру вокселя, в которой он попадает. После чего при помощи бинарного поиска находился первый фотон в отсортированном массиве фотонов и формировался список индексов. Второй метод использует аппаратные возможности на основе буфера трафарета и является более быстрым, но позволяет сохранять ограниченное число фотонов на каждый воксел. Метод реализовывает запись индекса фотона в определенную ячейку используя аппаратный механизм на основе буфера трафарета. Недостатком обоих методов из работы [94] является высокий расход памяти.

В работе [95] был предложен эффективный алгоритм построения kd-деревьев с учетом эвристики VVN [31] и алгоритм поиска k-ближайших фотонов. Алгоритм из работы [95] изначально ориентирован на построения kd-деревья над множеством полигонов. Он использует 2 различных подхода в зависимости от количества примитивов в узле и хранит данные в динамических списках на GPU. Алгоритм [95] достаточно объемный и сложный в реализации и, кроме того, расходует больше памяти, чем его CPU аналог. Что касается сбора k-ближайших фотонов - производительность алгоритма сбора низкая вследствие итеративного алгоритма поиска оптимального радиуса сбора.

В [96], в целях упрощения алгоритма построения и повышения его эффективности был предложен следующий подход: Использованное для ускорения трассировки лучей kd-дерево подразбивалось далее так, чтобы листья его имели сопоставимый с радиусом сбора размер. После трассировки фотонов, для каждого фотона вычислялся индекс узла kd-деревья, в который этот фотон попадает и индексы копировались сторону CPU, где при помощи механизма быстрых списков за 2 прохода, строилась результирующая ускоряющая структура:

1. На первом проходе итерировались все индексы листьев для фотонов и для каждого листа kd-дерева составлялся список фотонов попадающих в него.
2. На втором проходе все списки объединялись в линейный массив, который копировался на сторону GPU.

Алгоритм построения, таким образом, имеет линейную сложность -  $O(N)$ . Причем, само дерево строилось 1 раз, а при обновлении порции фотонов достаточно было изменить только результирующий массив индексов фотонов. Среди недостатков описанного выше подхода следует отметить необходимость априорного знания о геометрии сцены и копирование данных между CPU и GPU.

В [97] для ускорения сбора освещенности было использовано BVH-дерево и фиксированный радиус сбора. Среди алгоритмов построения BVH дерева на GPU достаточно эффективны подходы из работ [98, 99]. Использование BVH дерева в целом является неплохим решением, и основным недостатком этого и других подходов, использующих классические деревья, является рекурсивный алгоритм обхода дерева при сборе освещенности. Причины падения производительности при использовании такого обхода - высокая степени дивергентности потоков на GPU и использования локальной памяти.

В работе [100] предложен эффективный подход построения ускоряющей структуры на основе лексикографической сортировки координат фотонов. После окончания трассировки все фотоны сортировались в лексикографическом порядке по координатам соответствующих вокселей. При таком выборе оператора сравнения все фотоны, принадлежащие одному вокселю, являются эквивалентными и занимают непрерывный сегмент в отсортированной фотонной карте. Над отсортированным массивом строилась воксельная ускоряющая структура на основе 3D текстур. Данный метод похож на первый алгоритм

из работы [94] и его основным недостатком является высокий расход памяти. В работе [101] предложен метод пространственных хэш-таблиц на основе “хэширования кукушки”. Авторы не исследовали метод в применении к алгоритму фотонных карт и в работе отмечается, что для различных применений могут потребоваться различные хэш-функции.

Авторы работы [102] использовали хэш-таблицы для реализации фотонных карт, причем, для того чтобы исключить коллизии во время поиска, дополнительно производили сортировку пар (ключ, значение) по ключу. При этом поиск ближайших фотонов в точке  $(x, y, z)$  производился в ячейке  $\text{hash}(x, y, z)$  и также в 27 соседних ячейках. [102] отмечает, что для глобальной фотонной карты по сравнению с kd-деревом из работы [95] этот метод не дает ускорения сбора с ростом искомого числа ближайших фотонов (т.е. с ростом радиуса сбора). Это может быть объяснено потерей производительности на ветвлениях из-за наличия вложенного цикла, что необходимо для поиска в 27 соседних ячейках. При большом радиусе сбора и, как следствие, значительном времени итерирования фотонов в листьях, время обхода узлов дерева становится пренебрежимо малым. Однако, если в лист попадает только часть потоков группы *warpr* (назовем эту часть активной), остальные потоки будут вынуждены ждать до тех пор, пока лист не будет пройден активной частью группы *warpr*. Но эта проблема одинаково наблюдается как для классических деревьев, так и для хэш-таблиц из работы [102].

#### [НУЖНАКАРТИНКА]

В работе [103] предложен иной подход на основе пространственных хэш-таблиц. При возникновении коллизии авторы предлагают стохастически сохранять только 1 фотон на элемент таблицы. Данный подход замедляет трассировку фотонов (в 3-4 раза в соответствии со средним числом коллизий), что может быть нецелесообразно, т.к. при сложных условиях освещения трассировка фотонов будет занимать много больше времени, чем построение ускоря-

ющих структур при помощи любого из известных методов. Позже, в работе [104] на основе метода из [101] был реализован алгоритм фотонных карт и было продемонстрировано, что решение на основе метода из работы [103] работает медленнее чем решение на основе метода из работы [101].

Среди практичных подходов следует упомянуть работу [105], авторы которой использовали растеризацию сфер с включенным альфа-смешиванием. Это позволяет реализовать сбор освещенности на GPU вообще без построения каких-либо укоряющих структур. Данный подход аналогичен растеризации сплатов из работы [91], рассмотренной ранее и имеет тот же набор ограничений - а именно, - может быть использован только совместно с растеризацией но неприменим в трассировке лучей. В работе [106] в дальнейшем исследовалась производительность различных подходов к решению задачи сбора освещенности в применении к интерактивной визуализации на основе растеризации сплатов.

Также стоит отметить методы, вычисляющие карты освещенности (а также карты светимости) в текстурном пространстве (texture space) [107]. Данные методы, как правило, вместо сбора освещенности с фотонов используют обратную операцию - распределение освещенности с отдельных фотонов по карте освещенности. Значительное преимущество этого подхода - возможность расчета освещенности независимо от положения камеры. Недостаток подхода состоит в необходимости наложения текстурных координат для создания карты освещенности, что не во всех случаях может быть сделано полностью автоматически с высоким качеством.

Хотя подходы на основе окто-деревьев используются при реализации фотонных карт достаточно часто, публикаций по данной тематике относительно немного. Причина этого заключается в том, что значительное число работ по построению окто-деревьев на GPU было опубликовано в смежных областях. Одна из работ, использующих окто-деревья на GPU для фотонных карт -

[108]. Причем, в работе [108] обсуждается эффективная реализация поиска  $k$ -ближайших фотонов над окто-деревьями. Сравнения, сделанные в данной работе показывают примерно одинаковое качество при использовании сбора освещенности с фиксированным радиусом и использовании сбора с  $k$ -ближайших фотонов. По этой причине далее будем считать, что можно использовать фиксированный радиус сбора и рассмотрим некоторые работы, обсуждающие возможность построения окто-деревьев на GPU.

**Построение окто-деревьев на GPU.** Алгоритмы построения окто-деревьев на GPU используют факт взаимосвязанности окто-дерева и трехмерной Z-кривой, которая получается в результате вычисления Z-индекса (или кода Мортон) [109]. Зафиксируем некоторую глубину дерева ' $k$ '. Пусть у нас имеется такой набор точек в трехмерном пространстве, что в каждый узел построенного в будущем окто-дерева попадет не более чем 1 точка. Если вычислить для точек коды Мортон и затем отсортировать точки по этим кодам, мы получим массив, эквивалентный окто-дереву, где в каждом листе хранится ровно по одному элементу (одной точке). Выполняя далее поиск в трехмерном пространстве в произвольной точке  $(x,y,z)$ , мы вычисляем ее код Мортон, и применив бинарный поиск для полученного кода Мортон ( $ZInxed(x,y,z)$ ), находим листовой узел окто-дерева, в который попала точка  $(x,y,z)$ .

В работе [110] было впервые замечено соответствие между Z-кривой и окто-деревом и использовано свойство отбрасывания младших битов Z-индекса для получения идентификаторов старших узлов. Окто-дерево в [110] представлялось набором массивов  $L1..Lk$ , размером  $8^i$  элементов для каждого уровня глубиной  $i$  (т.е. несуществующие узлы хранились в памяти на регулярной сетке). Вопрос хранения каких-либо элементов данных в окто-дереве в работе [110] не рассматривается. Таким образом, построенное окто-дерево

не может быть использовано непосредственно для пространственного поиска элементов, а может только отвечать на вопрос имеются ли искомые элементы в некотором объеме или нет.

В работе [111] окто-дерево было использовано для построения поверхности из набора точек при помощи алгоритма марширующих кубов. Алгоритм построения использовал серию из параллельных операций уплотнения и сортировки. В отличие от работы [110], Zhou и др. сохраняют только существенные узлы в памяти, а также сохраняют списки точек на всех уровнях дерева. При этом, в [111] было замечено, что имея отсортированный массив точек по их Z-индексу в узле достаточно сохранять лишь 2 смещения, обозначающих начало и конец последовательности точек для данного узла в отсортированном массиве. Причем, при переходе от уровня  $k$  на уровень  $k-1$  это свойство сохраняется и точки пересортировывать не нужно. Достаточно пересчитать указатели соответствующие началу и концу последовательности.

Если в рассмотренных ранее работах только отдельные уровни дерева строились параллельно, то в работе [99] был предложен метод, использующий массивный параллелизм GPU при построении всего дерева. Основная идея алгоритма заключается в том, чтобы пронумеровать все узлы дерева при помощи индексов таким образом, чтобы индекс родительского узла всегда являлся префиксом по отношению к индексам его дочерних узлов, если рассматривать индексы как битовые строки. Тогда, отсортированный массив индексов будет эквивалентен дереву. Следует отметить, что свойство префиксов кодов Мортонa было описано ранее в работе [110].

#### **2.4.5. Приближенные методы на GPU**

Существует достаточно большое число работ, вычисляющих глобальное освещение приближенно. Сюда можно отнести такие методы как Light Propagation



Volumes (LPV) [112], трассировка конусов над воксельными моделями [113], метод 'Radiance Hits' [114], методы на основе мгновенной излучательности [115] и старейший из методов - аппроксимация вторичного освещения при помощи предрассчитанных сферических гармоник [116].

Данные методы обладают хорошими временными характеристиками. Многие из них широко используются в современных компьютерных играх. Однако, в данной работе они не рассматриваются детально, поскольку их точность при этих временных характеристиках чрезвычайно низкая. Если же говорить о тех из них, которые могли бы обеспечить необходимую точность в пределе - [112] и [113], то при такой точности они работают не быстрее, чем обыкновенная Монте-Карло трассировка путей (а в случае метода [112] потребуется еще и невероятно большой объем памяти).

## 2.5. Программные решения на центральных процессорах

В настоящий момент число программных решений, позволяющих в той или иной мере решать проблему глобальной освещенности, достаточно большое. Будем называть такие программные решения рендер-системами. Среди наиболее известных рендер-систем следует отметить:

1. Mental Ray, Pixar RenderMan, Houdini, Arnold, Indigo, Maxwell, VRay, HyperShot, Thea Render, Lightworks'Artisan, ZEANY, Lumion, Clarisse, Twinmotion, DirectViz, keyshot, modo, PhotoView 360, fryrender, SWAP, Turtle, Brazil, final render, Cheetah3D, kerkethea, EIAS9, Team Render (CINEMA 4D).
2. Specter, Inspirer, Inspirer 2, lumicept, Tonatiuh, SolTRACE, rtt , dialux, lighttools, speos, ASAP, tracepro, zemax, DiamCalc.

### 3. PBRT, Mitsuba, LuxRender, POV-Ray, Sunflow, Yafaray, Pixie, Aqsis, Radiance.

Разделение на 3 группы, представленное выше сделано следующим образом: к первой группе относятся системы, ориентированные на реалистичный синтез изображений; ко второй группе относятся системы, предназначенные для физически-точного расчета освещения; к третьей группе относятся системы, ориентированные в большей степени на реалистичный синтез изображений, но используемые в основном в обучающих и исследовательских целях. Для систем из 3 группы важна доступность и расширяемость исходных кодов.

Такое большое разнообразие существующих программных решений обусловлено в первую очередь чрезвычайно широким спектром применения рендер-систем и, в некоторых случаях, высокой специализацией (например система DiamCalc ориентирована на реалистичную визуализацию драгоценных камней). В действительности, даже рендер-системы общего назначения имеют свою специализацию. Например, системы RenderMan, Houdini, Arnold ориентированы в первую очередь на создание кино-эффектов. В таких системах важна поддержка огромного спектра эффектов и, что гораздо более важно, возможность тонкой настройки всех этапов визуализации. Причем, настройка подразумевает не просто установку predetermined параметров, а, как правило, создание специализированного производственного процесса, направленного на визуализацию определенного материала в кино. В качестве примера можно привести генерацию и визуализацию лесного массива в мультфильме САВВА, моделирование катастрофы поезда в фильме Метро, визуализация боевых действий в фильме Сталинград (2013 года) и.т.д. По этой причине существует специальная высоко-оплачиваемая профессия - технический художник (technical artist). Эти люди призваны организовывать производственный процесс целиком, превращая работу художников, программистов и актеров в конечный результат. При этом временные затраты на создание самих моде-

лей и анимации настолько велики, что время расчета самих кадров фильма, хотя и важно, не всегда имеет решающее значения.

Другие системы (Например MentalRay, VRay) ориентированны на работу индивидуальных дизайнеров, работающих над архитектурными или рекламными проектами в небольших дизайнерских студиях. В таких системах чрезмерно широкие возможности и настройки не только не нужны, но и создают дополнительные трудности пользователям, которым необходимо уметь разбираться иногда не только функциональных возможностях системы, но и в параметрах используемых алгоритмов. Для этих систем важен относительно-простой и быстрый процесс проектирования и визуализации. Причем, скорость является критичным параметром, поскольку время визуализации может занимать значительную часть рабочего времени дизайнера. Более того, необходимо уметь демонстрировать потенциальному заказчику интерьер и/или экстерьер в различных условиях освещения, а также различные варианты самого интерьера. По этой причине достаточно востребованной функциональностью рендер-систем является "переосвещение"(relighting) [117] - возможность менять условия освещения в реальном времени на основе предрасчитанной информации. Пользователи таких систем, как правило, профессиональные дизайнеры и художники, имеющие навыки в работе со сложными пакетами трехмерного моделирования.

Такие системы как HyperShot еще в большей степени ориентированы на быстрое создание рекламных роликов и презентаций. Основная цель - создание рекламной презентации за несколько минут, имея на руках сырую модель, спроектированную в САПР пакете. В таких системах важна не только скорость визуализации, но и скорость создания самой модели. Не имея специальных навыков (как в случае предыдущей группы систем), пользователь должен иметь возможность за небольшое число действий достигать конечного результата.

Наконец, наличие самых разнообразных специализированных программных решений (например программа для создания виртуальных моделей мебели - PRO100) и существование разнообразных сапр-систем, создает спрос на интегрированные в производственный процесс систем визуализации.

Вторая причина большого разнообразия рендер-систем - низкая производительность существующих систем и, как следствие, конкуренция систем, интегрированных в одни и те же пакеты трехмерного моделирования. Здесь следует отметить чрезвычайно популярный пакет 3D Studio Max компании Autodesk, который используется во многих областях промышленности и бизнеса и служит полем для соперничества самых различных программных решений.

Что касается систем, ориентированных на высокоточное моделирование, такие системы, как правило, обладают в высокой степени спецификой решаемой задачи. Среди таких систем необходимо отметить программные и программно-аппаратные комплексы, разработанные в Институте Прикладной Математики имени Келдыша РАН (ИПМ РАН) - Specter, Inspirer, Inspirer 2, lumicapt. Данные комплексы прочно закрепились в современной промышленности и обеспечивают интеграцию с популярной САПР-системой CATIA. Обозначенные выше системы позволяют проектировать сложные оптические системы и решают такие задачи как ([118], [119]):

1. Физически-аккуратный расчет освещенности и построение фотореалистичных изображений сцен содержащих оптически сложные материалы при различных условиях искусственного и естественного [120] освещения [121].
2. Моделирование осветительных приборов со сложными свойствами распределения света, сложных оптических систем и устройств, включая системы, содержащие новейшие микроструктурные рассеивающие ма-

териалы, применяемые в современном проектировании оптических светопроводящих систем [122], [123].

3. Измерение оптических свойств плоских образцов материалов [124].
4. Моделирование сложных автомобильных красок, красящих покрытий с высокой концентрацией пигмента [125], тканей [126].
5. Расчет карт освещенности и последующая интерактивная визуализация в режиме навигации сцен [127].
6. Моделирования освещенности и синтез реалистичных изображений через Интернет [128].
7. Автоматизацию задания параметров сцены с целью упрощения процесса работы пользователей [119].

Одним из краеугольных камней реалистичной визуализации и расчета освещенности является производительность. Опыт разработки высоко-производительных вычислительных систем в последние годы приводит к ряду важных результатов, среди которых необходимо обратить особое внимание на специализированные программно-аппаратные модели (такие как массивно-параллельные системы на основе GPU). На сегодняшний день графические процессоры обладают значительно большей производительностью (и, более того, значительно большим запасом возможной производительности) именно благодаря специальной программной-аппаратной модели. Эта модель отличается от классических программ для центральных процессоров тем, что в ней, так называемый параллелизм данных описывается явно на уровне алгоритмов.

Большинство рендер-систем, закрепившихся в промышленности на сегодняшний день используют центральные процессоры. Эти системы были

спроектированы в 80-ых и 90-ых годах прошлого века и в лучшем случае предполагали параллелизм на основе кластерных систем. Причем, параллелизм не закладывался в алгоритмы изначально, а реализовывался постфактум, в буквальном смысле 'как получится'. Показательным примером является рендер-система Radiance, использующая алгоритм кэширования освещенности и работы [87–89], пытающиеся сделать кэш освещенности параллельным, не меняя оригинального алгоритма и архитектуру системы.

Однако, такой путь развития имеет свой предел и по этой причине в настоящее время активно развиваются новые системы (использующие как CPU так и GPU), в которых параллелизм заложен изначально на уровне алгоритмов.

## 2.6. Программные решения на графических процессорах

Среди существующих рендер-систем реалистичной визуализации, использующих GPU следует отметить такие проекты как Arion, Octane, V-Ray RT, LuxRender GPU, I-Ray, OptiX (API), OpenRL (API), Gelato (проект закрыт), RenderAnts, Render Bro (проект по-видимому закрыт), Centi Leo, Cycles Render, RTRays, Hydra.

Последняя в списке система (Hydra) была создана в результате данной диссертационной работы. Она будет детально описана в 4 главе данной работы. Как обсуждалось ранее, в настоящее время все известные промышленные системы, использующие графические процессоры для расчета глобальной освещенности и реалистичного синтеза изображений основаны на различных вариациях метода Монте-Карло. Плюсом данного подхода является возможность получения точного решения в пределе, при достаточно-большом числе Монте-Карло выборок. Однако, использование метода Монте-Карло в чистом виде негативно сказывается на времени расчета изображения.

В результате, несмотря на то, что современные GPU при одинаковой сто-

имости способны до 10 раз опережать CPU на задаче трассировки лучей [14], время построения изображения в современных рендер-системах на CPU (при примерно одинаковом качестве) сравнимо или даже меньше времени построения изображения при помощи систем на основе GPU. Причина этого заключается в том, что некоторые методы, рассмотренные в 1 и 2 главах позволяют при сравнимом качестве получаемого изображения значительно сократить объем производимых вычислений. За счет этого, CPU решения в ряде случаев продолжают выигрывать у GPU решений. Известная рендер система Octane, является ярким примером такого случая [129]. Характерным здесь является тот факт, что Octane позиционируется именно как одна из самых быстрых несмещенных рендер-систем (компания-разработчик сравнивает его исключительно с другими 'CPU unbiased renderers'), поскольку время расчета изображения в нем может быть в итоге достаточно большим.

Среди существующих систем многие используют графические процессоры лишь частично, для реализации отдельных алгоритмов на GPU (например для ускорения пересечение луча и сцены). К таким системам относятся Arion и LuxRender. Такие решения имеют преимущество в легкости поддержки исходного кода и расширяемости. Однако их производительность ограничена во-первых из-за необходимости передачи данных между CPU и GPU и, во-вторых, все же, производительностью CPU кода.

Среди рендер-систем использующих графические процессоры, следует отметить разработку Нижегородского Государственного Университета имени Лобачевского - систему RTRays. Помимо обычной Монте-Карло трассировки путей, данная система реализует собственный оригинальный алгоритм расчета освещенности - усеченную двунаправленной трассировку путей [24]. Этот алгоритм позволяет рассчитывать значительно более широкий спектр эффектов нежели при помощи обыкновенной Монте-Карло трассировки. Из других положительных сторон следует отметить расширяемость системы, возможность

ее использования в обучающем процессе и быстрый алгоритм построения BVH дерева, реализованный полностью на GPU.

Стоит отметить и другую отечественную разработку - систему Centi Leo. Как и в системе RTRays, в Centi Leo построение BVH дерева реализовано полностью на GPU. Однако, Centi Leo - единственная из существующих на сегодня систем интегрирования освещенности, способная эффективно работать с большими объемами геометрии и текстур (объемы, не помещающиеся в памяти GPU).

Следует сказать и о таких системах как OptiX и OpenRL, которые не являются законченными рендер-системами, а представляют из себя специализированные API, предназначенные для создания рендер-систем на их основе. Причем, OptiX предназначен для разработке систем на основе трассировки лучей для GPU производимых компанией Nvidia, а OpenRL имеет в настоящий момент специализированную аппаратную реализацию от компании Caustic Graphics [130].

Хотя специализированный чип Caustic Graphics имеет определенные преимущества, по соотношению цена/производительность в настоящий момент он проигрывает решению от Nvidia (OptiX). Из недостатков OptiX следует отметить серьезное падение производительности при реализации сложных алгоритмов. Это падение связано с тем, что в системе OptiX весь пользовательский код собирается компилятором в 1 большое мега-ядро, которое работает по принципу конечного автомата. Такой подход позволяет достичь значительной гибкости, однако, он негативно влияет на производительность решения (в 2-3 раза медленнее чем подход на основе множества небольших ядер) [83].



# Заключение

Выводы

## Литература

1. Фролов . . , Харламов. . . , Игнатенко . . Смещённое решение интегрального уравнения светопереноса на графических процессорах при помощи трассировки путей и кэша освещённости // ПРОГРАММИРОВАНИЕ. 2011. Т. 37, № 5. С. 47–60.
2. Frolov V., Vostryakov K., Kharlamov A., Galaktionov V. Implementing Irradiance Cache in a GPU Realistic Renderer // Trans. on Comput. Sci. XIX, LNCS 7870. 2013. Vol. 7870, no. 1. P. 17–32.
3. Whitted T. An improved illumination model for shaded display // Commun. ACM. 1980. — jun. Vol. 23, no. 6. P. 343–349. URL: <http://doi.acm.org/10.1145/358876.358882>.
4. Wright R., Lipchak B., Haemel N. OpenGL®; Superbible: Comprehensive Tutorial and Reference, Fourth Edition. Fourth edition. Addison-Wesley Professional, 2007. ISBN: [9780321498823](#).
5. Pharr M., Humphreys G. Physically Based Rendering, Second Edition: From Theory To Implementation. 2nd edition. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: [0123750792](#), [9780123750792](#).
6. Suffern K. Ray Tracing from the Ground Up. Natick, MA, USA: A. K. Peters, Ltd., 2007. ISBN: [1568812728](#).
7. Debelov V. A., Kozlov D. S. A Local Model of Light Interaction with Transparent Crystalline Media // IEEE Transactions on Visualization and Computer Graphics. 2013. Vol. 19, no. 8. P. 1274–1287.
8. Fujimoto A., Tanaka T., Iwata K. Tutorial: computer graphics; image synthesis / Ed. by K. I. Joy, C. W. Grant, N. L. Max, L. Hatfield. New

- York, NY, USA: Computer Science Press, Inc., 1988. P. 148–159. URL: <http://dl.acm.org/citation.cfm?id=95075.95111>.
9. MacDonald D. J., Booth K. S. Heuristics for ray tracing using space subdivision // *Vis. Comput.* 1990. — Vol. 6, no. 3. P. 153–166. URL: <http://dx.doi.org/10.1007/BF01911006>.
  10. Popov S., Günther J., Seidel H.-P., Slusallek P. Experiences with Streaming Construction of SAH KD-Trees // *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. 2006. — sep. P. 89–94.
  11. Wald I. *On Fast Construction of SAH-based Bounding Volume Hierarchies* // *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. RT '07. Washington, DC, USA: IEEE Computer Society, 2007. P. 33–40. URL: <http://dx.doi.org/10.1109/RT.2007.4342588>.
  12. Ernst M., Greiner G. *Early Split Clipping for Bounding Volume Hierarchies* // *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. RT '07. Washington, DC, USA: IEEE Computer Society, 2007. P. 73–78. URL: <http://dx.doi.org/10.1109/RT.2007.4342593>.
  13. Ize T., Wald I., Parker S. G. Ray tracing with the BSP tree // *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*. 2008.
  14. Aila T., Laine S. *Understanding the efficiency of ray traversal on GPUs* // *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New York, NY, USA: ACM, 2009. P. 145–149. URL: <http://doi.acm.org/10.1145/1572769.1572792>.
  15. Garanzha K. The Use of Precomputed Triangle Clusters for Accelerated Ray Tracing in Dynamic Scenes // *Computer Graphics Forum*. 2009. Vol. 28, no. 4. P. 1199–1206.

16. Nicodemus F. E. Directional Reflectance and Emissivity of an Opaque Surface // [Applied Optics](#). 2009. Vol. 4, no. 7. P. 767–773.
17. Kajiya J. T. The rendering equation // [SIGGRAPH Comput. Graph.](#) 1986. — Vol. 20, no. 4. P. 143–150. URL: <http://doi.acm.org/10.1145/15886.15902>.
18. Nicodemus F. E., Richmond J. C., Hsia J. J. et al. Radiometry / Ed. by L. B. Wolff, S. A. Shafer, G. Healey. USA: Jones and Bartlett Publishers, Inc., 1992. P. 94–145. URL: <http://dl.acm.org/citation.cfm?id=136913.136929>.
19. Jensen H. W., Buhler J. A Rapid Hierarchical Rendering Technique for Translucent Materials // [ACM Trans. Graph.](#) 2002. — Vol. 21, no. 3. P. 576–581. URL: <http://doi.acm.org/10.1145/566654.566619>.
20. Соболев . Численные методы Монте-Карло. 1 edition. М.: Наука., 1973.
21. Goral C. M., Torrance K. E., Greenberg D. P., Battaile B. [Modeling the interaction of light between diffuse surfaces](#) // Proceedings of the 11th annual conference on Computer graphics and interactive techniques. SIGGRAPH '84. New York, NY, USA: ACM, 1984. P. 213–222. URL: <http://doi.acm.org/10.1145/800031.808601>.
22. Shirley P., Wang C. Distribution Ray Tracing: Theory and Practice // In Proceedings of the Third Eurographics Workshop on Rendering. 1992. P. 33–43.
23. Veach E. Robust monte carlo methods for light transport simulation: Ph. D. thesis. Stanford, CA, USA: Stanford University, 1998. AAI9837162.
24. Боголепов . Методы глобального освещения для интерактивного синтеза изображений сложных сцен на графических процессорах: Ph.D. thesis.

- НГГУ им. Лобачевского, Нижний Новгород, Россия: Нижний Новгород, 2013.
25. Szecsi L., Szirmay-Kalos L., Kelemen C. [Variance Reduction for Russian-roulette](#) // WSCG. 2003.
26. Heckbert P. S. Adaptive radiosity textures for bidirectional ray tracing // [SIGGRAPH Comput. Graph.](#) 1990. — Vol. 24, no. 4. P. 145–154. URL: <http://doi.acm.org/10.1145/97880.97895>.
27. Georgiev I., Křivánek J., Slusallek P. [Bidirectional light transport with vertex merging](#) // SIGGRAPH Asia 2011 Sketches. SA '11. New York, NY, USA: ACM, 2011. P. 27:1–27:2. URL: <http://doi.acm.org/10.1145/2077378.2077412>.
28. Veach E., Guibas L. J. [Metropolis light transport](#) // Proceedings of the 24th annual conference on Computer graphics and interactive techniques. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997. P. 65–76. URL: <http://dx.doi.org/10.1145/258734.258775>.
29. Cline D., Egbert P. A Practical Introduction to Metropolis Light Transport: Tech. rep.: Brigham Young University, 2005.
30. Jensen H. W., Christensen P. [High quality rendering using ray tracing and photon mapping](#) // ACM SIGGRAPH 2007 courses. SIGGRAPH '07. New York, NY, USA: ACM, 2007. URL: <http://doi.acm.org/10.1145/1281500.1281593>.
31. Wald I., Gunther J., Slusallek P. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic // Computer Graphics Forum. 2004. Vol. 22, no. 3. P. 595–603. Proceedings of Eurographics.

32. Tabellion E., Lamorlette A. [An approximate global illumination system for computer generated films](#) // ACM SIGGRAPH 2004 Papers. SIGGRAPH '04. New York, NY, USA: ACM, 2004. P. 469–476. URL: <http://doi.acm.org/10.1145/1186562.1015748>.
33. Востряков . Глобальное освещение с помощью октанных текстур // Материалы 16-ой международной конференции Графикон'2006. Graph-iCon'06. Новосибирск, Россия: Графикон, 2006.
34. Hachisuka T., Ogaki S., Jensen H. W. Progressive photon mapping // [ACM Trans. Graph.](#) 2008. — Vol. 27, no. 5. P. 130:1–130:8. URL: <http://doi.acm.org/10.1145/1409060.1409083>.
35. Hachisuka T., Jensen H. W. Stochastic progressive photon mapping // [ACM Trans. Graph.](#) 2009. — Vol. 28, no. 5. P. 141:1–141:8. URL: <http://doi.acm.org/10.1145/1618452.1618487>.
36. Hachisuka T., Jarosz W., Bouchard G. et al. [State of the art in photon density estimation](#) // ACM SIGGRAPH 2012 Courses. SIGGRAPH '12. New York, NY, USA: ACM, 2012. P. 6:1–6:469. URL: <http://doi.acm.org/10.1145/2343483.2343489>.
37. Suykens F., Willems Y. D. Density Control for Photon Maps // Proceedings of the Eurographics Workshop on Rendering Techniques 2000. London, UK, UK: Springer-Verlag, 2000. P. 23–34. URL: <http://dl.acm.org/citation.cfm?id=647652.732120>.
38. Schjøth L., Frisvad J. R., Erleben K., Sporring J. [Photon differentials](#) // Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia. GRAPHITE '07.

- New York, NY, USA: ACM, 2007. P. 179–186. URL: <http://doi.acm.org/10.1145/1321261.1321293>.
39. Ward G. J., Rubinstein F. M., Clear R. D. A ray tracing solution for diffuse interreflection // *SIGGRAPH Comput. Graph.* 1988. — jun. Vol. 22, no. 4. P. 85–92. URL: <http://doi.acm.org/10.1145/378456.378490>.
40. Krivanek J., Gautron P. Practical Global Illumination with Irradiance Caching (Synthesis Lectures in Computer Graphics and Animation). Morgan and Claypool Publishers, 2009. ISBN: [1598296442](#), [978-1598296440](#).
41. Лебедев . История развития алгоритмов глобального освещения // Компьютерная графика и мультимедиа; сетевой журнал. 2011. Vol. 1, no. 9. 23 p. URL: <http://cgm.computergraphics.ru/issues/issue19/globalillum>.
42. Cohen M. F., Wallace J., Hanrahan P. Radiosity and realistic image synthesis. San Diego, CA, USA: Academic Press Professional, Inc., 1993. ISBN: [0-12-178270-0](#).
43. Kenny M. Lighting you up in Battlefield 3. 2013.
44. Keller A. [Instant radiosity](#) // Proceedings of the 24th annual conference on Computer graphics and interactive techniques. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997. P. 49–56. URL: <http://dx.doi.org/10.1145/258734.258769>.
45. Walter B., Fernandez S., Arbree A. et al. [Lightcuts: a scalable approach to illumination](#) // ACM SIGGRAPH 2005 Papers. SIGGRAPH '05. New York, NY, USA: ACM, 2005. P. 1098–1107. URL: <http://doi.acm.org/10.1145/1186822.1073318>.

46. Křivánek J., Hašan M., Arbree A. et al. [Optimizing realistic rendering with many-light methods](#) // ACM SIGGRAPH 2012 Courses. SIGGRAPH '12. New York, NY, USA: ACM, 2012. P. 7:1–7:217. URL: <http://doi.acm.org/10.1145/2343483.2343490>.
47. Walter B., Khungurn P., Bala K. Bidirectional lightcuts // [ACM Trans. Graph.](#) 2012. — Vol. 31, no. 4. P. 59:1–59:11. URL: <http://doi.acm.org/10.1145/2185520.2185555>.
48. Ou J., Pellacini F. [LightSlice: matrix slice sampling for the many-lights problem](#) // Proceedings of the 2011 SIGGRAPH Asia Conference. SA '11. New York, NY, USA: ACM, 2011. P. 179:1–179:8. URL: <http://doi.acm.org/10.1145/2024156.2024213>.
49. Christensen P. H. Point Based Global Illumination // SIGGRAPH 2010 Course: Global Illumination Across Industries. SIGGRAPH '10. New York, NY, USA: ACM, 2010.
50. Green R. Spherical Harmonic Lighting: The Gritty Details. 2003.
51. Hanrahan P., Salzman D., Aupperle L. A rapid hierarchical radiosity algorithm // [SIGGRAPH Comput. Graph.](#) 1991. — jul. Vol. 25, no. 4. P. 197–206. URL: <http://doi.acm.org/10.1145/127719.122740>.
52. Kelemen C., Szirmay-Kalos L., Antal G., Csonka F. A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm // [Computer Graphics Forum](#). Vol. 23. 2002. P. 531–540.
53. Cline D., Talbot J., Egbert P. Energy redistribution path tracing // [ACM Trans. Graph.](#) 2005. — Vol. 24, no. 3. P. 1186–1195. URL: <http://doi.acm.org/10.1145/1073204.1073330>.



54. Lehtinen J., Karras T., Laine S. et al. Gradient-domain metropolis light transport // *ACM Trans. Graph.* 2013. — Vol. 32, no. 4. P. 95:1–95:12. URL: <http://doi.acm.org/10.1145/2461912.2461943>.
55. Kaplanyan A. S., Dachsbacher C. Path Space Regularization for Holistic and Robust Light Transport // *Computer Graphics Forum (Proc. of Eurographics 2013)*. 2013. Vol. 32, no. 2.
56. Igehy H. *Tracing ray differentials* // *Proceedings of the 26th annual conference on Computer graphics and interactive techniques. SIGGRAPH '99*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999. P. 179–186. URL: <http://dx.doi.org/10.1145/311535.311555>.
57. Spencer B., Jones M. W. Progressive photon relaxation // *ACM Trans. Graph.* 2013. Vol. 32, no. 1. P. 7:1–7:11.
58. Havran V., Herzog R., Seidel H.-P. Fast Final Gathering via Reverse Photon Mapping // *Computer Graphics Forum (Proceedings of Eurographics 2005)*. 2005. — August. Vol. 24, no. 3. P. 323–333.
59. Bekaert P., Slusallek P., Cools R. et al. A custom designed density estimation method for light transport: Research Report MPI-I-2003-4-004. Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany: Max-Planck-Institut für Informatik, 2003. — September.
60. Vorba J., Krivánek J. Bidirectional Photon Mapping // *CESCG 2011*. 2011.
61. Doidge I. C., Jones M. W., Mora B. Mixing Monte Carlo and progressive rendering for improved global illumination // *Vis. Comput.* 2012. — Vol. 28, no. 6-8. P. 603–612. URL: <http://dx.doi.org/10.1007/s00371-012-0703-2>.

62. Hachisuka T., Pantaleoni J., Jensen H. W. A path space extension for robust light transport simulation // *ACM Trans. Graph.* 2012. — Vol. 31, no. 6. P. 191:1–191:10. URL: <http://doi.acm.org/10.1145/2366145.2366210>.
63. Jensen H. W. Importance Driven Path Tracing using the Photon Map // in *Eurographics Rendering Workshop*. Springer-Verlag, 1995. P. 326–335.
64. Křivánek J. Radiance Caching for Global Illumination Computation on Glossy Surfaces: Ph.d. thesis / Université de Rennes 1 and Czech Technical University in Prague. 2005. — December. URL: <http://www.cgg.cvut.cz/~havran/dissertation/index.htm>.
65. Křivánek J., Bouatouch K., Pattanaik S. N., Žára J. Making Radiance and Irradiance Caching Practical: Adaptive Caching and Neighbor Clamping // *Rendering Techniques 2006*, Eurographics Symposium on Rendering. Nicosia, Cyprus: Eurographics Association, 2006. — June. P. 127–138.
66. Jensen H. W., Christensen P. High quality rendering using ray tracing and photon mapping // *SIGGRAPH 2002 Course 43*. New York, NY, USA: Association for Computing Machinery, Aug. 2002, ACM SIGGRAPH, 2002. URL: <http://www.cs.princeton.edu/courses/archive/fall02/cs526/papers/course43sig02.pdf>.
67. Востряков . Высокочастотный кэш излучения // Материалы 16-ой международной конференции Графикон'2009. GraphiCon'09. Москва, Россия: Графикон, 2009.
68. Arikan O., Forsyth D. A., O'Brien J. F. Fast and Detailed Approximate Global Illumination by Irradiance Decomposition // *Proceedings of ACM SIGGRAPH 2005*. ACM Press, 2005. URL: <http://graphics.cs.berkeley.edu/papers/Arikan-FAD-2005-07/>.

69. Sen P., Darabi S. On filtering the noise from the random parameters in Monte Carlo rendering // *ACM Trans. Graph.* 2012. — Vol. 31, no. 3. P. 18:1–18:15. URL: <http://doi.acm.org/10.1145/2167076.2167083>.
70. Gastal E. S. L., Oliveira M. M. Adaptive Manifolds for Real-Time High-Dimensional Filtering // *ACM TOG.* 2012. Vol. 31, no. 4. P. 33:1–33:13. Proceedings of SIGGRAPH 2012.
71. Hennessy J. L., Patterson D. A. Computer Architecture, Fifth Edition: A Quantitative Approach. 5th edition. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: [012383872X](#), [9780123838728](#).
72. Lefohn A., Houston M. Beyond Programmable Shading // SIGGRAPH 2009 Course. New York, NY, USA: Association for Computing Machinery, Aug. 2009, ACM SIGGRAPH, 2009. URL: <http://s09.idav.ucdavis.edu/>.
73. NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2013. — July.
74. Kulshrestha C. U. D. A. P. P. O. U. W. C. D. C. N. U. B. R. M. P. A. B., Narayan (Fremont. Processing global atomic operations using the bending unit datapath. 2013. — April. URL: <http://www.freepatentsonline.com/8411103.html>.
75. NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture Kepler GK110, 2013. — July.
76. Nvidia. CUDA Accelerated Libraries. 2013.
77. Group P. PGI CUDA-x86. 2013.
78. Farooqui N., Kerr A., Damos G. et al. [A Framework for Dynamically Instrumenting GPU Compute Applications Within GPU Ocelot](#) // Proceedings of

- the Fourth Workshop on General Purpose Processing on Graphics Processing Units. GPGPU-4. New York, NY, USA: ACM, 2011. P. 9:1–9:9. URL: <http://doi.acm.org/10.1145/1964179.1964192>.
79. Foley T., Sugerman J. [KD-tree acceleration structures for a GPU raytracer](#) // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. HWWS '05. New York, NY, USA: ACM, 2005. P. 15–22. URL: <http://doi.acm.org/10.1145/1071866.1071869>.
  80. Horn D. R., Sugerman J., Houston M., Hanrahan P. [Interactive k-d tree GPU raytracing](#) // Proceedings of the 2007 symposium on Interactive 3D graphics and games. I3D '07. New York, NY, USA: ACM, 2007. P. 167–174. URL: <http://doi.acm.org/10.1145/1230100.1230129>.
  81. Günther J., Popov S., Seidel H.-P., Slusallek P. Realtime Ray Tracing on GPU with BVH-based Packet Traversal // Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007. 2007. — P. 113–118.
  82. Фролов ., Игнатенко . Интерактивная трассировка лучей и фотонные карты на GPU // Материалы международной конференции Графикон 2009. 2009.
  83. Laine S., Karras T., Aila T. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs // Proceedings of High-Performance Graphics 2013. 2013.
  84. Novak J. Global Illumination Methods on GPU with CUDA. 2009.
  85. van Antwerpen D. [Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU](#) // Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. HPG '11. New York, NY, USA: ACM, 2011. P. 41–50. URL: <http://doi.acm.org/10.1145/2018323.2018330>.

86. Sadeghi I., Chen B., Jensen H. W. Coherent Path Tracing // Journal of Graphics, GPU, and Game Tools. 2009. Vol. 14, no. 2. P. 33–43.
87. Ward G. J. [The RADIANCE lighting simulation and rendering system](#) // Proceedings of the 21st annual conference on Computer graphics and interactive techniques. SIGGRAPH '94. New York, NY, USA: ACM, 1994. P. 459–472. URL: <http://doi.acm.org/10.1145/192161.192286>.
88. Koholka R., Mayer H., Goller A. MPI-parallelized Radiance on SGI CoW and SMP. // ACPC / Ed. by P. Zinterhof, M. Vajtersic, A. Uhl. Vol. 1557 of Lecture Notes in Computer Science. Springer, 1999. P. 549–558.
89. Debattista K., Santos L. P., Chalmers A. Accelerating the Irradiance Cache through Parallel Component-Based Rendering . // EGPGV / Ed. by A. Heirich, B. Raffin, L. P. P. dos Santos. Eurographics Association, 2006. P. 27–34.
90. Dubla P., Debattista K., Santos L. P., Chalmers A. Wait-Free Shared-Memory Irradiance Cache. // EGPGV / Ed. by K. Debattista, D. Weiskopf, J. Comba. Eurographics Association, 2009. P. 57–64.
91. Gautron P., Krivanek J., Bouatouch K., Pattanaik S. N. Radiance Cache Splatting: A GPU-Friendly Global Illumination Algorithm. // Rendering Techniques / Ed. by O. Deussen, A. Keller, K. Bala et al. Eurographics Association, 2005. P. 55–64.
92. Nvidia. Cube Map OpenGL Tutorial. 1999.
93. Nilsson J., Clarberg P., Johnsson B. et al. [Design and novel uses of higher-dimensional rasterization](#) // Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics. EGGH-HPG'12.

- Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2012. P. 1–11. URL: <http://dx.doi.org/10.2312/EGGH/HPG12/001-011>.
94. Purcell T. J., Donner C., Cammarano M. et al. Photon Mapping on Programmable Graphics Hardware // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. Eurographics Association, 2003. P. 41–50.
  95. Zhou K., Hou Q., Wang R., Guo B. Real-time KD-tree construction on graphics hardware // *ACM Trans. Graph.* 2008. — Vol. 27, no. 5. P. 126:1–126:11. URL: <http://doi.acm.org/10.1145/1409060.1409079>.
  96. Фролов . ., Игнатенко . . Интерактивная трассировка лучей и фотонные карты на GPU. // Труды 22-й Международной Конференции по Компьютерной Графике и Зрению Графикон'2009. Москва: Московский Государственный Университет им. М.В.Ломоносова, 2009. P. 255–262.
  97. Fabianowski B., Dingliana J. Interactive Global Photon Mapping // *Computer Graphics Forum*. 2009. Vol. 28, no. 4. P. 1151–1159.
  98. Garanzha K., Pantaleoni J., McAllister D. [Simpler and Faster HLBVH with Work Queues](#) // Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. HPG '11. New York, NY, USA: ACM, 2011. P. 59–64. URL: <http://doi.acm.org/10.1145/2018323.2018333>.
  99. Karras T. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. 2012. P. 33–37.
  100. Боголепов ., Турлапов . . МОДЕЛИРОВАНИЕ КАУСТИК В РЕАЛЬНОМ ВРЕМЕНИ НА ОСНОВЕ КОМБИНИРОВАННЫХ ВОЗМОЖНОСТЕЙ OpenCL И ШЕЙДЕРОВ // Вестник

- Нижегородского университета им. Н.И. Лобачевского. 2011. Vol. 2, no. 3. P. 180–186.
101. Alcantara D. A., Sharf A., Abbasinejad F. et al. Real-time parallel hashing on the GPU // *ACM Trans. Graph.* 2009. — Vol. 28, no. 5. P. 154:1–154:9. URL: <http://doi.acm.org/10.1145/1618452.1618500>.
  102. Fleisz M. Photon Mapping on the GPU // Master of Science. 2009. Vol. School of Informatics, University of Edinburgh, no. 1. P. 1–60.
  103. Hachisuka T., Jensen H. W. *Parallel progressive photon mapping on GPUs* // ACM SIGGRAPH ASIA 2010 Sketches. SA '10. New York, NY, USA: ACM, 2010. P. 54:1–54:1. URL: <http://doi.acm.org/10.1145/1899950.1900004>.
  104. Carlberg K. Stochastic Progressive Photon Mapping Using Parallel Hashing // Master Thesis. 2011. Vol. Lund University, no. 1. P. 1–51.
  105. McGuire M., Luebke D. Hardware-Accelerated Global Illumination by Image Space Photon Mapping // Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics. New York, NY, USA: ACM, 2009. — August. URL: <http://graphics.cs.williams.edu/papers/PhotonHPG09/>.
  106. Mara M., McGuire M., Luebke D. Toward Practical Real-Time Photon Mapping: Efficient GPU Density Estimation // Interactive 3D Graphics and Games 2013. 2013. — March. URL: <http://graphics.cs.williams.edu/papers/PhotonI3D13/>.
  107. Czuczor S., Szirmay-kalos L., Szecsi L., Neumann L. EUROGRAPHICS 2005 / J. Dingliana and F. Ganovelli Short Presentations Photon Map Gath-

- ering on the GPU †. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.115.8745>.
108. Li S., Simons L. C., Pakaravoor J. B. et al. kANN on the GPU with Shifted Sorting // Proceedings of High Performance Graphics 2012 / Ed. by C. Dachsbacher, J. Munkberg, J. Pantaleoni; High Performance Graphics 2012. The Eurographics Association 2012, 2012.
  109. Morton G. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company, 1966. URL: <http://books.google.ru/books?id=9FFdHAAACAAJ>.
  110. Ajmera P., Goradia R., Chandran S., Aluru S. Fast, parallel, GPU-based construction of space filling curves and octrees // Proceedings of the 2008 symposium on Interactive 3D graphics and games. I3D '08. New York, NY, USA: ACM, 2008. P. 10:1–10:1. URL: <http://doi.acm.org/10.1145/1342250.1357022>.
  111. Zhou K., Gong M., Huang X., Guo B. Data-Parallel Octrees for Surface Reconstruction // IEEE Transactions on Visualization and Computer Graphics. 2011. — Vol. 17, no. 5. P. 669–681. URL: <http://dx.doi.org/10.1109/TVCG.2010.75>.
  112. Kaplanyan A., Dachsbacher C. Cascaded light propagation volumes for real-time indirect illumination // Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games. I3D '10. New York, NY, USA: ACM, 2010. P. 99–107. URL: <http://doi.acm.org/10.1145/1730804.1730821>.
  113. Crassin C., Neyret F., Sainz M. et al. Interactive Indirect Illumination Us-



- ing Voxel Cone Tracing. 2011. — sep. URL: <http://maverick.inria.fr/Publications/2011/CNSGE11b>.
114. Papaioannou G. Real-Time Diffuse Global Illumination Using Radiance Hints. // High Performance Graphics / Ed. by C. Dachsbacher, W. Mark, J. Pantaleoni. Eurographics Association, 2011. P. 15–24.
  115. Hašan M., Pellacini F., Bala K. Matrix Row-Column Sampling for the Many-Light Problem // ACM SIGGRAPH 2007 papers. New York, NY, USA: ACM, 2007.
  116. Green R. Spherical Harmonic Lighting: The Gritty Details // Archives of the Game Developers Conference. 2003. — mar. URL: <http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf>.
  - 117.
  118. Галактионов . Программные технологии синтеза реалистичных изображений: Ph.D. thesis. Институт прикладной математики им. Келдыша, Москва, Россия: Москва, 2006.
  119. Волобой . Программные технологии автоматизации построения реалистичных изображений: Ph.D. thesis. Институт прикладной математики им. Келдыша, Москва, Россия: Москва, 2012.
  120. Valiev I., Voloboy A., Galaktionov V. [Improved Model of IBL Sunlight Simulation](#) // Proceedings of the 24th Spring Conference on Computer Graphics. SCCG '08. New York, NY, USA: ACM, 2010. P. 27–32.
  121. Adinets A., Barladian B., Galaktionov V. et al. Physically Accurate Rendering with Coherent Ray Tracing. 2006.
  - 122.

- 123.
124. Волобой ., Галактионов ., Ершов . et al. Аппаратно-программный комплекс для измерения светорассеивающих свойств поверхностей // Информационные технологии и вычислительные системы. 2006. по. 4.
125. Волобой ., Ершов ., Клышинский ., Поздняов . Моделирование распространения света в тонком красящем слое с высокой концентрацией частиц // Труды 20-ой международной конференции по компьютерной графике и зрению ГРАФИКОН-2010. Санкт-Петербург, Россия: 2010. Р. 155–162.
- 126.
127. Барлядян ., Волобой ., Шапиро . Оптимизация представления карт освещенности и яркости для их интерактивной визуализации // Труды 19-ой международной конференции по компьютерной графике и зрению ГРАФИКОН-2009. Москва, Россия: 2009. Р. 267–270.
128. Барлядян ., Волобой ., Вьюкова . et al. Моделирование освещенности и синтез фотореалистичных изображений с использованием Интернет технологий // Программирование. 2005. Vol. 5. Р. 3–18.
129. ОТОУ. Octane Renderer. 2013.
130. CausticGraphics. OpenRL. 2011.

## Приложение А

### Название приложения