Федеральное государственное бюджетное учреждение науки

Институт прикладной математики им М.В. Келдыша Российской академии наук

На правах рукописи

Гаранжа Кирилл Владимирович

Интерактивный синтез реалистичных изображений больших 3D сцен с применением графических процессоров

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация

на соискание учёной степени

кандидата физико-математических наук

Научный руководитель:

д. ф.-м. н., проф. Галактионов Владимир Александрович

Москва 2014

Содержание

Введение

Актуальность работы. С момента появления компьютерной графики синтез компьютерных изображений нашёл множество полезных приложений. Наиболее известными являются приложения из сферы киноиндустрии и развлечений, где синтез изображений используется для создания визуально красивых компьютерных игр и спецэффектов кино. Кроме этого приложения компьютерного синтеза изображений можно найти в сфере рекламы, медицине, архитектурном и инженерном дизайне. Во многих этих областях синтезируемое изображение содержит реалистичное глобальное освещение, рассчитанное в виртуальной 3D сцене. Например, в сфере архитектурного дизайна реалистичный синтез изображений используется для ответа на вопрос, как будет выглядеть спроектированное здание или интерьер дома в различных условиях освещения. В сфере рекламы и кинопроизводства компьютерная графика используется в процессе смешивания виртуальных 3D объектов и видеозаписей для визуализации сложных сцен и сценариев, что было бы невозможно или очень дорого с использованием одних лишь съёмок на видеокамеру. Подобные сферы приложения требуют высокого уровня реализма, дающего на выходе изображения сцены с различных ракурсов, визуально неотличимые от реальных фотографий таких же сцен в реальном мире.

Реалистичные алгоритмы визуализации используют математические модели физического переноса света на основе лучевой оптики в моделированной виртуальной сцене для синтеза изображений, которые снимаются с различных ракурсов с учётом свойств камеры наблюдателя. Для большого количества приложений реалистичные алгоритмы визуализации требуют большой вычислительной нагрузки для создания одного изображения. Например, расчёт одного кадра изображения со сложным освещением может занимать несколько часов при использовании одного ядра CPU Поэтому большинство подобных алгоритмов используются для оффлайн визуализации (т.е. без отклика в режиме реального времени) даже при использовании больших многоузловых вычислительных систем. Отсутствие интерактивности во многих алгоритмах реалистичной визуализации (быстрого отклика в генерации реалистичного изображения) осложняет работу разработчика виртуальной среды.

Также большого количества дополнительных вычислительных ресурсов требует использование динамических сцен, где в любой момент времени любой объект или его часть могут изменить свою форму или положение в мировой системе координат, в которой задаётся виртуальная сцена. Как правило, в алгоритмах визуализации, применяющих реалистичную модель распространения света, используется трассировка лучей, где в свою очередь требуются, чтобы геометрические данные сцены были организованы внутри специальной геометрической базы данных ГБД (часто называемой ускоряющей структурой УС), которая позволяет ускорять расчёт реалистичного глобального освещения.

Во многих сферах компьютерной графики, включая кинопроизводство, архитектурный и инженерный дизайн, всё чаще используются графические процессоры (GPU) [39] благодаря более быстрым вычислениям (в 10-20 раз) и более высокой пропускной способности внутренней памяти по сравнению с традиционными архитектурами центральных процессоров (CPU). Недостатком GPU является фиксированный размер памяти. В существующих реализациях GPU размер памяти является небольшим (1-12 GB) по сравнению с объёмом оперативной памяти центрального процессора (60-100 GB и более).

Во всех сферах, в которых используется синтез реалистичных изображений, происходит рост требований к объёму детализации моделируемых и визуализируемых виртуальных 3D сцен. Например, современные виртуальные сцены из сфер киноиндустрии и архитектурного проектирования могут занимать от нескольких гигабайт до нескольких сотен гигабайт или нескольких терабайт данных. Для эффективного исполнения все современные реалистичные алгоритмы визуализации требуют хранения всех данных сцены в памяти, наиболее близко располагающейся к процессору. Часто подобное требование физически невозможно осуществить. Физическая память процессора может быть мала для хранения всей сцены целиком, это актуально для графических процессоров и также для центральных процессоров. Подобное ограничение требует разработки сложных структур данных и кэширования для амортизации расходов, связанных с доставкой данных сцены из внешнего хранилища (дискового или сетевого пространства) в оперативную память.

Цель диссертационной работы. Исходя из роста требований задач реалистичной визуализации к сложности сцен, массового распространения вычислительно мощных и ограниченных в объёме памяти графических процессоров возникают следующие цели:

- разработка и программная реализация эффективных алгоритмов построения геометрической базы данных (ускоряющей структуры), обеспечивающей эффективную трассировку лучей в анимированных сценах высокой сложности, в том числе с применением графических процессоров;

- разработка и программная реализация эффективных алгоритмов поиска пересечений лучей с применением графического процессора для массивных сцен, для которых объём данных может многократно (в десятки раз) превышать размеры физической памяти графического процессора. Алгоритм поиска пересечений должен обеспечивать возможность приложения в любых алгоритмах расчёта глобального освещения, в том числе в интерактивных алгоритмах, и исполняться на массовых графических процессорах.

Научная новизна. Для ускорения процесса построения геометрической базы данных (ГБД) в рамках настоящей работы разработаны следующие ортогональные алгоритмы (главы 2-5):

1) Алгоритм обновления ГБД в каждом кадре анимации сцены, использующий структуру ранее построенной ГБД (глава 2). Разработанный алгоритм применим для широкого класса анимированных сцен, включая анимации взрывов и структурированное движение.
2) Алгоритм генерации множества кластеров связанных полигонов и использования множества кластеров в качестве строительных блоков в контексте процесса построения ГБД, т.к. множество кластеров на порядок меньше множества полигонов (глава 3). Используется предположение, что в процессе анимации связность треугольников сохраняется постоянной, и анимация осуществляется за счёт изменения координат вершин полигонов. Подобное предположение применимо для любых анимированных сцен, в том числе со-

держащих анимации взрывов, если обломки взрывов и места разрывов обломков рассчитаны заранее.

3) Алгоритмы построения ГБД на графическом процессоре (главы 4 и 5), потребляющие в несколько раз меньше памяти, работающие на порядок быстрее, чем аналоги на CPU и на графическом процессоре. Также решена проблема разрезания полигонов неоднородных размеров в рамках фиксированного ограничения размера памяти. Подобное разрезание необходимо для построения ГБД, обеспечивающей более эффективный поиск пересечения лучей.

Разработаны масштабируемые алгоритмы поиска пересечений лучей и фильтрации текстур (глава 6), позволяющие с использованием серийных графических процессоров реалистично визуализировать массивные сцены, размер которых может на порядки превышать размер физической памяти графического процессора. Разработанные алгоритмы на порядок быстрее существующих аналогов, исполняющихся на CPU и на графическом процессоре. Разработанные алгоритмы применимы совместно с любыми алгоритмами расчёта глобального освещения, основанными на трассировке лучей, в том числе интерактивных.

Разработанный алгоритм поиска пересечений лучей позволил достичь логарифмической зависимости скорости поиска пересечений от размера кэша графического процессора (глава 6). Это позволяет использовать кэш память меньшего размера с сохранением приемлемой скорости поиска пересечений.

Практическая значимость. На основе разработанных алгоритмов с использованием технологи CUDA параллельного программирования на GPU [79] реализован программный комплекс интерактивной реалистичной визуализации, применение которого позволяет привести к существенному повышению производительности труда в архитектурном, инженерном проектировании и киноиндустрии.

Разработанный программный комплекс, в частности, позволяет реалистично визуализировать изображения в интерактивном режиме с разрешением Full HD самолёта Боинг 777, состоящего из 360 млн. треугольников в режиме интерактивной визуализации. В качестве метода расчёта глобального освещения используется

Монте-Карло трассировка путей с несколькими уровнями переотражений, исполняющаяся на серийном графическом процессоре с 3 GB памяти. На расчёт одного кадра изображения без видимых шумов в среднем тратится 3 минуты. Интерактивный режим реалистичной визуализации позволяет осуществлять 10 обновлений изображения в секунду, тогда как более ранние методы требовали несколько секунд или минут для расчёта каждого Монте-Карло обновления изображения.

Апробация работы. Результаты работы докладывались и обсуждались на:

- Международной конференции “IEEE/EG Interactive Ray Tracing Symposium 2008”, США, Лос-Анджелес, 2008;
- Международной конференции “Eurographics Symposium on Rendering 2009”, Испания, Жирона, 2009;
- Международной конференции “Eurographics 2010”, Швеция, Норрчёпинг, 2010;
- Международной конференции “Computer Graphics International 2011”, Канада, Оттава, 2011;
- Международной конференции “High Performance Graphics 2011”, Канада, Ванкувер, 2011;
- Международной конференции “SIGGRAPH 2011”, Канада, Ванкувер, 2011;
- Международной конференции “Graphicon 2011”, Россия, Москва, 2011;

Публикации. По результатам работы имеются 7 публикаций, соответствующих требованиям ВАК, из них 3 публикации в научных рецензируемых журналах [2][3][4] входят в библиографическую базу Web of Science и 7 публикаций [1]-[7] входят в библиографическую базу Scopus.

Глава 1

Обзор предметной области

1.1. Конвейер алгоритма реалистичной визуализации

Алгоритм реалистичной визуализации можно представить в виде конвейера, состоящего из нескольких стадий (см. рис. 1.1). В отдельной программе интерактивного моделирования (**стадия А**) создаются данные виртуальной сцены, включающие в себя источники света, параметры камеры наблюдателя, геометрическое представление объектов сцены, текстуры, задающие свойства поверхностей объектов. Также эти данные сцены могут храниться в отдельном хранилище для осуществления оффлайн визуализации необходимых кадров. Все эти данные передаются на конвейер визуализации, состоящий из стадий (Б) – (Е).

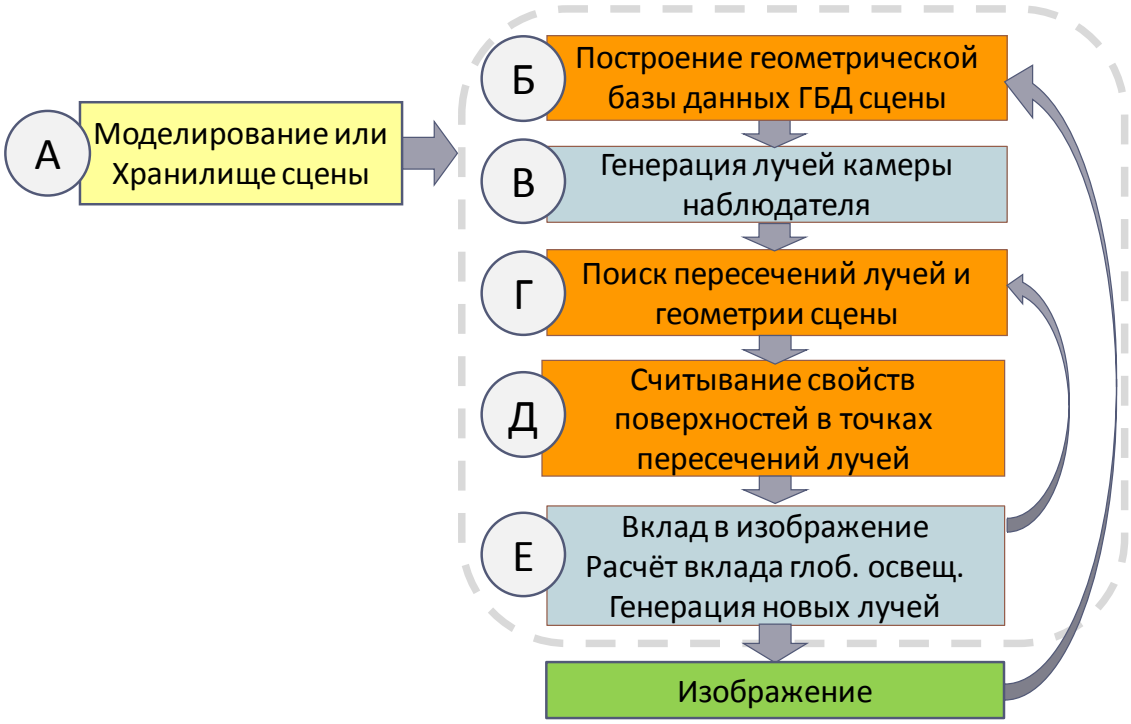


***Рис. 1.1**: Конвейер реалистичной визуализации. Оранжевым цветом отмечены стадии, на повышение эффективности которых направлен фокус настоящей работы.*

Во-первых (**стадия Б**), для каждого кадра анимации перед началом расчёта глобального освещения необходимо 1 раз построить геометрическую базу данных (ГБД), которая организует в поисковой структуре геометрическое представление сцены для быстрого осуществления операций пересечения лучей с объектами сце-

ны и поиска ближайших точек пересечения для каждого луча. В дальнейшем в тексте ГБД может называться ускоряющей структурой (УС).

В качестве выходных данных **стадия В** производит множество лучей, берущих начало в положении наблюдателя, с учётом фокусного расстояния и разрешения изображения. В **стадии Г** осуществляется поиск пересечения лучей и всех объектов сцены с использованием ГБД. В **стадии Д** производится считывание свойств поверхностей геометрических объектов в точках пересечения для каждого луча, для которого пересечение найдено. Свойства поверхностей могут задаваться функциями рассеивания и поглощения света, параметры этих функций могут задаваться текстурами. Текстура является 2D дискретным изображением, задающим варьирующиеся параметры. Текстуры отображаются на геометрическую поверхность с помощью текстурных координат, заданных в программе моделирования для каждой контрольной точки поверхности. **Стадия Е** осуществляет, с учётом свойств поверхности в точке пересечения луча, расчёты прямого и непрямого освещения, попадающего в данную точку. Для расчёта прямого освещения необходимо проверить блокируются ли лучи света, проведённые от источников света к точке освещения. Если не блокируются (проверяется с помощью поиска пересечений), то производится вклад в часть изображения, соответствующее точке. Непрямое освещение учитывает свет, достигающий этой точки посредством нескольких отражений и преломлений через другие объекты сцены. Рассчитанный вклад глобального освещения для каждого пикселя записывается в изображение (**Ё**). Существуют итеративные алгоритмы, позволяющие визуализировать временные результаты расчётов глобального освещения в каждой итерации с прогрессивным накоплением рассчитанного освещения в каждом пикселе, позволяя таким образом производить интерактивную навигацию в сцене и наблюдать постепенную сходимость к конечному результату.

В рамках настоящей работы разработана серия алгоритмов (главы 2 – 5), позволяющих значительно ускорить процесс построения геометрической базы данных (**стадия Б**). Скорость построения ГБД является критичной по времени стадией конвейера для сцен, содержащих сложные динамические объекты, для которых необходимо визуализировать последовательность кадров.

Также в рамках настоящей работы разработан алгоритм поиска пересечений лучей и объектов сцены (**стадия Г**) и считывание свойств поверхностей (**стадия Д**), заданных текстурами (глава 6). Этот алгоритм поддерживает большие сцены, геометрическое представление которых, а также текстуры могут не вмещаться полностью в память графического процессора. Достоинством данного алгоритма является высокая скорость исполнения, обеспечиваемая вычислительной мощностью графического процессора, возможность интерактивной работы в специальных приложениях.

1.2. Расчёт глобального освещения

В данном разделе представлена общая картина процесса реалистичной визуализации на примере алгоритма стохастической Монте-Карло трассировке путей, с помощью которой решается уравнение визуализации [62]:

$$L_0(x, w_0, \lambda, t) = Le(x, w_0, \lambda, t) + \int_{\Omega} fr(x, w_i, w_0, \lambda, t)\, L_i(x, w_i, \lambda, t) \cos(\theta_i)\, dw_i \qquad (1.1)$$

где ***n*** – нормаль в точке ***x***; ***λ*** – длина волны; ***t*** – время;

L_0(x, w_0, λ, t) – излучение заданной длины волны λ, исходящее из точки x вдоль направления w_0 во время t;

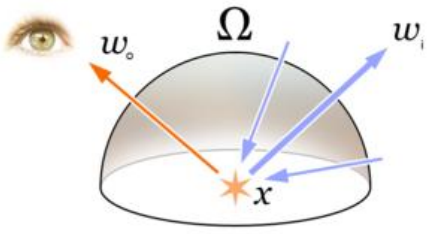
L_i(x, w_i, λ, t) – излучение заданной длины волны λ, попадающее в точку x вдоль направления w_i во время t;

Le(x, w_0, λ, t) – свет, излучаемый в точке;

fr(x, w_i, w_0, λ, t) – двунаправленная функция рассеивания света, значение которой равно доле излучения, отражённого в точке x вдоль направления w_0, пришедшего вдоль направления w_i;

cos(θ_i) – косинус угла между n и w_i, доля затухания падающего излучения в зависимости от угла;

Ω – область определения всех возможных углов w_i падающего излучения, в этой области производится интегрирование излучения.



Одним из методов решения уравнения визуализации является метод Монте-Карло (см. рис. 1.2): выпускается случайный луч из положения наблюдателя из точки *x0* через пиксель *pix* изображений, устанавливается значение переменной *alpha(pix, λ) = 1*. Производится поиск ближайшего пересечения луча с объектами сцены (найдена точка *x1*), после этого в точке *x1* генерируется один случайный луч $w1_i$ (изображён жирной синей линией на рис 1.2) из области определения Ω с плотностью $p1(w1_i)$ распределения вероятностей, пропорциональной значениям функции $fr(x1, w1_i, w_0, \lambda, t)$ в точке *x1*. Далее обновляется значение *alpha*:

$alpha(pix, \lambda)\ *= fr(x1, w1_i, w_0, \lambda, t) / p1(w1_i)$

После этого для луча $w1_i$ производится поиск ближайшей точки пересечения *x2* с геометрией, в которой генерируется новый луч $w2_i$ и т.д. пока не будет найдено пересечение с источником света в точке *xn*. Для пикселя *pix* производится обновление видимого через него освещения:

$intensity(pix) += alpha(pix, \lambda) * Le(xn \rightarrow x4)$

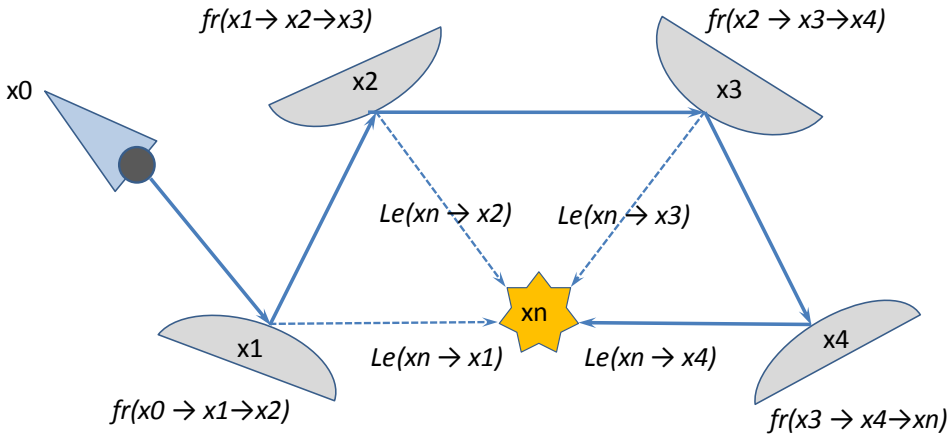


***Рис. 1.2**: Монте-Карло трассировка пути с явными соединениями с источниками света*

Дополнением к уравнению визуализации, повышающим эффективность его решения, является процесс явного соединения точки *x* пересечения луча со случайной точкой (например, *xn*) на источнике света. Явные соединения обозначены прерывистыми линиями на рис. 1.2. Для учёта прямого освещения в точке x методом явного соединения модифицируется уравнение (1.1):

$$L_0(x, w_0, \lambda, t) = Le(x, w_0, \lambda, t) + \int_M fr(x, w_i, w_0, \lambda, t) L_i(xn, x, \lambda, t) G(x \leftrightarrow xn) dA_M \quad (1.2)$$

$$G(x \leftrightarrow xn) = V(x \leftrightarrow xn) \frac{\left|\cos(\theta_i)\cos(\theta_{Li})\right|}{\left|x - xn\right|^2}$$

где $\boldsymbol{V(x \leftrightarrow xn) = 1}$, если линия между точками x и xn не блокируется другими объектами. $V(x \leftrightarrow xn) = 0$, если блокируется.

$w_i = norma(xn - x)$ – нормированный вектор от x до xn.

$cos(\theta_i)$ – косинус угла между w_i и n; $cos(\theta_{Li})$ – косинус угла между $-w_i$ и n_L, нормалью в точке xn. $|x - xn|^2$ – квадрат расстояния между точками x и xn.

$L_i(xn, x, \lambda, t)$ – излучение, попадающее в точку x вдоль направления w_i, выходящее из точки xn на поверхности источника света.

M – область всех точек поверхности источника света (многих источников).

Комбинация двух способов учёта освещения по формулам (1.1) и (1.2) может осуществляться с помощью техники MIS (Multiple Importance Sampling, см. работу [106]). Кроме метода обратной Монте-Карло трассировки путей (от наблюдателя в поисках источников света) существуют другие стратегии расчёта глобального освещения: прямая трассировка (от источников света в поисках наблюдателя), двунаправленная трассировка (создаются частичные пути с обеих сторон, и их крайние точки явно соединяются). Более подробное описание и математические формулировки упомянутых методов, а также их модификаций, представлены в серии работ [106][18][62][10][9][37].

Методы, основанные на применении фотонных карт, в том числе и в комбинации с упомянутыми выше способами прямой и двунаправленной трассировки путей, представлены в работах [40][58]. Методы, основанные на применении фотонных карт, позволяют более эффективно визуализировать каустики.

Существует множество возможных типов двунаправленной функции рассеивания света в точке $fr(x, w_i, w_0, \lambda, t)$. Эта функция позволяет задавать различные свойства поверхности с помощью нескольких параметров, таких как шероховатость и доля отражённого/поглощённого света. На рис. 1.3 представлены изображения, представляющие процесс чёткого отражения лучей, соответствующий зеркальной поверхности; процесс размытого отражения, где отражённые лучи света выходят из точки в узкой области вокруг зеркально отражённого луча; процесс диффузного отражения, где отражённые лучи света выходят из точки во всех направлениях. Аналогичные свойства существуют и у поверхностей, проводящих

свет. Более подробные материалы о двунаправленных функциях рассеивания света могут быть найдены в работах [85][20][118].

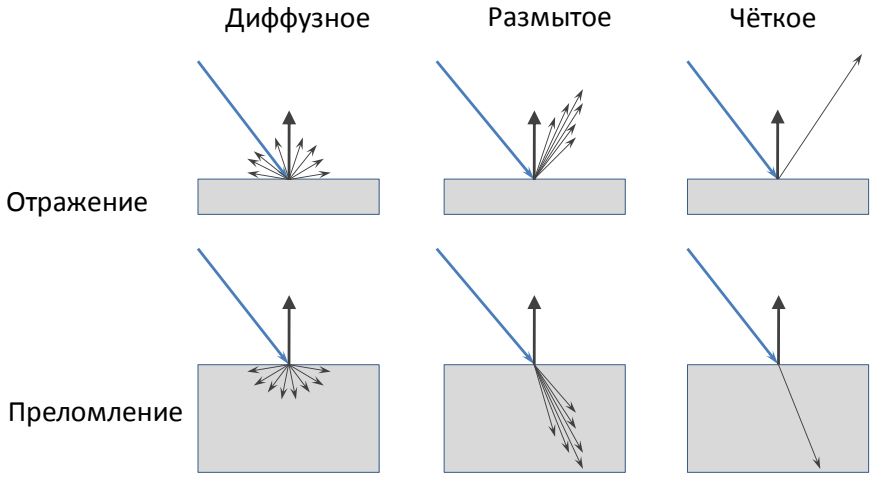


***Рис. 1.3**: Некоторые виды функций отражения или преломления света в точке: диффузное, чёткое или размытое.*

1.3. Представление сцены

В рамках настоящей работы рассматриваются сцены, геометрическое представление которых является полигональным (см. рис. 1.4), являющимся стандартом в подавляющем большинстве программного обеспечения для создания виртуальных сцен. В большинстве случаев объект полигональной сцены задаётся в виде множества треугольников, аппроксимирующих граничные поверхности объекта. Каждый треугольник представлен в виде 3 индексов вершин и индекса материала поверхности. Каждая вершина содержит координату в пространстве сцены, внешнюю нормаль и текстурную координату. Текстурная координата одной вершины содержит 2 числа в диапазоне [0..1], определяя относительное положение точки, соответствующей вершине, в двумерной текстуре, разрешение которой может быть любым. Текстурные координаты вершин треугольников задают функцию отображения любых накладываемых изображений на геометрическую поверхность. Текстурная координата в любой точке треугольника определяется на основе интерполяции текстурных координат в вершинах треугольника.

В материале задаются инструкции о том, какие текстуры накладываются на поверхность, и как они используются. Материал треугольника может определять

свойства поверхности, такие как микро-шероховатость (влияет на свойства отражения), долю отражённого, поглощённого света и другие параметры. На конкретные текстуры указывают ссылки внутри материала, текстуры используются для варьирования различных свойств материала на различных участках поверхности в процессе расчёта глобального освещения. Материал может быть представлен в виде текстуры или программы [101][91].

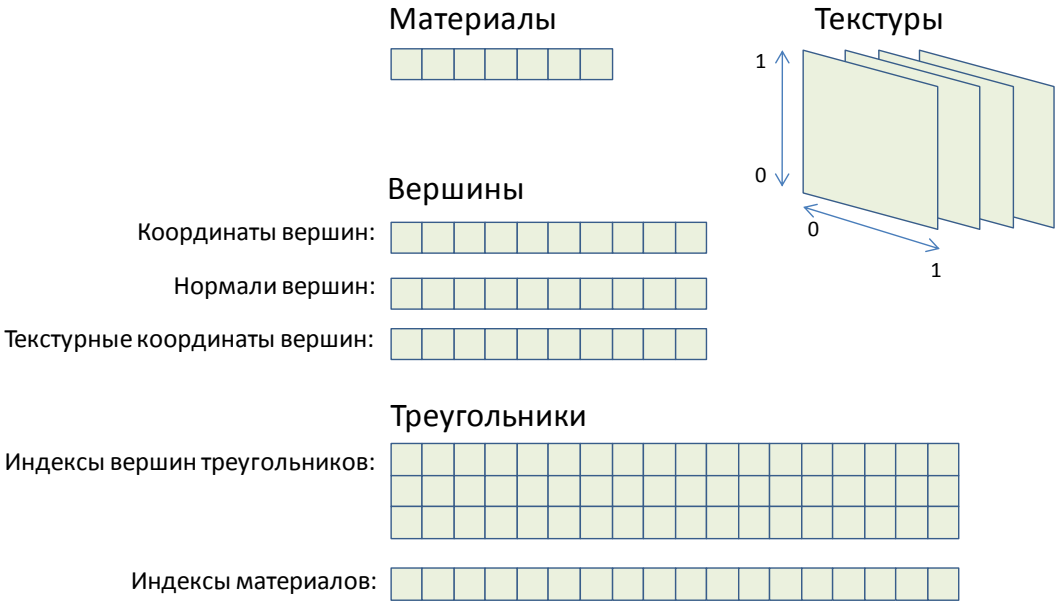


***Рис. 1.4**: Полигональное представление сцены*

Также существуют другие способы задания поверхностей в виде набора криволинейных патчей (например, см. [78]), волосяные примитивы, объёмные и т.д. Однако рассмотрение этих видов представления объектов выходит за рамки настоящей работы.

В рамках настоящей работы в главах 2-5 представлены ортогональные алгоритмы, с помощью которых производится быстрое построение ускоряющей структуры BVH. В главе 6 описан алгоритм поиска пересечения лучей, используемый в расчётах глобального освещения, позволяющий эффективно и интерактивно работать с массивными сценами, состоящими из сотен миллионов треугольников и десятков гигабайт текстурных изображений.

1.4. Построение ускоряющей структуры

Примитивы, из которых состоит 3D объект сцены, в данной стадии должны быть организованы в специальной ускоряющей структуре (другое название: геометрическая база данных, ГБД) для того, чтобы во время поиска пересечений лучей можно было ценой осуществления прохода лучей в узлах ускоряющей структуры уменьшить число тестов пересечения лучей и примитивов сцены. Ускоряющая структура является более качественной (или эффективной), если обеспечивает более высокую скорость поиска пересечений лучей.

Алгоритм построения ускоряющей структуры определяет, как организовать полигоны, из которых состоит объект, в иерархию и вычисляет 3D охватывающие оболочки узлов иерархии. В большинстве алгоритмов построения ускоряющей структуры множество полигонов сортируется в 3D пространстве, после этого данное множество с помощью одной из многочисленных плоскостей разделяется на новые подмножества. Для каждого из новых подмножеств полигонов создаётся узел, являющийся потомком для текущего узла и таким образом создаются связи внутри ускоряющей структуры. Данный процесс разделения продолжается рекурсивно пока каждый узел не будет содержать сравнительно небольшое множество полигонов. Лучшая известная эвристика для выбора оптимальной плоскости разделения это эвристика площадей поверхности **SAH** (Surface Area Heuristic [47][43]). Принятие решения о том, как лучше разделить множество полигонов в пределах узла *N*, основано на функции ожидаемой стоимости прохода луча через этот узел:

$$SAHCost(N) = \frac{SA(N_L)}{SA(N)} \cdot Count(N_L) + \frac{SA(N_R)}{SA(N)} \cdot Count(N_R) \qquad (1.3)$$

где *SA(N)* – площадь поверхности охватывающего шестигранника *AABB(N)* для узла *N* (*AABB(N)* является оболочкой для всех полигонов, принадлежащих узлу N); *Count(N)* – число полигонов, принадлежащих *N*; N_L и N_R – потомки *N*. Также в формуле (1.3) могут быть использованы различные коэффициенты [47].

Алгоритм построения оценивает различные способы разделения геометрии, используя эту эвристическую функцию. Выбирается оптимальное разделение, и далее производится разделение для новых узлов. Оптимальной плоскости разделения соответствует минимальная ожидаемая стоимость прохода луча, т.е. мини-

мальное значение функции (1.3). Способ разделения геометрии методом перебора всех возможных комбинаций подмножеств полигонов является трудоёмким. Существуют различные методы уменьшения временной сложности этой схемы разделения, временная сложность которых является суб-квадратичной [113][112] или линейной [54]. Такие методы могут и не выбрать способ разделения с минимальной оценочной стоимостью (1.3), но они выбирают за меньшее время способ разделения, близкий к минимальному значению (1.3). Ниже приведены схемы устройства 3-х основных видов ускоряющих структур.

Однородные 3D-сетки. Создаётся однородная 3D-сетка внутри всего 3D-пространства сцены. Каждый полигон вставляется в ячейки сетки, пересекающие его (см. рис. 1.5).

Преимущества:

1. Построение сетки быстрое: *O(n)*, где *n* – количество треугольников.
2. Простая процедура прохода луча в ячейках сетки в поисках пересечения с содержащимися полигонами (3D-растеризация луча).
3. Строго упорядоченное в 3D-пространстве перечисление ячеек вдоль луча во время поиска пересечения для луча.

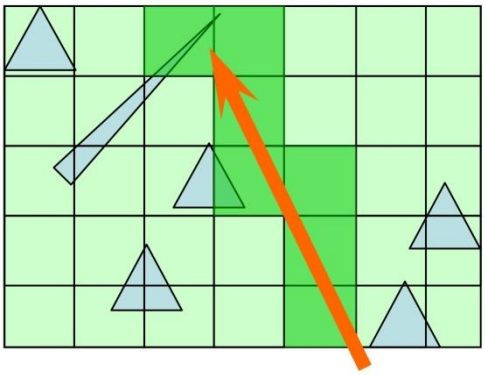


Рис. 1.5: *Однородная 3D-сетка*

Недостатки:

1. Потребление большого объёма памяти (расход памяти на пустые ячейки), если треугольники распределены неравномерно в 3D-пространстве. Сюда можно отнести и ‘холостое’ пересечение многих пустых ячеек, попадающихся вдоль направления луча.

Комментарий. Для устранения недостатка возможно применение иерархических (вложенных друг в друга) 3D-сеток, но при этом значительно усложняется процедура прохода луча.

kd-tree. 3D-пространство, занимаемое сценой, рекурсивно разбивается секущими плоскостями вида $X = divx$, $Y = divy$, $Z = divz$ (см. рис. 1.6). Выбор позиций секущих плоскостей *divx*, *divy*, *divz* с помощью алгоритма адаптируется к распределению полигонов в подпространстве, рекурсивно отделяя пустые области пространства от областей, содержащих полигоны. Такие адаптивные деревья называются SAH-деревьями [47][43].

Преимущества:

1. Нет перекрытий в 3D-пространстве между оболочками узлов дерева .
2. Строго упорядоченное в 3D-пространстве перечисление листьев дерева вдоль направления луча в процессе поиска пересечения луча.
3. Эффективное отсечение пустого пространства для SAH-деревьев во время прохода луча.

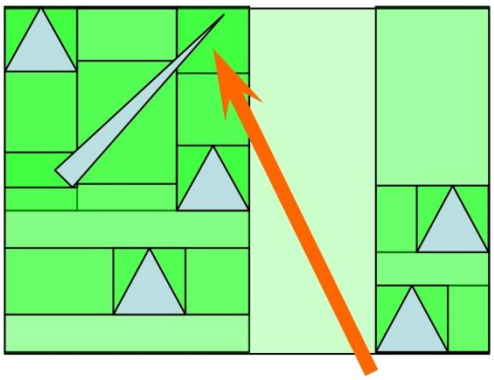


Рис. 1.6: *kd-tree*

Недостатки:

1. Построение SAH-дерева является медленным процессом: временная сложность алгоритма построения оценивается как $O(n * log\ n)$, где n – количество полигонов сцены.
2. До начала процесса построения нет возможности точно определить размер блока памяти, достаточного для хранения kd-tree, т.к. любой полигон может находиться в нескольких листьях kd-tree одновременно.
3. Дерево является ‘высоким’ (узел kd-tree соответствует 1-ой секущей плоскости).

Комментарий. Для kd-tree преимущества 1 и 2 являются преимуществами относительно дерева BVH. Преимущество 3 является преимуществом относительно однородной сетки. Недостатки 2 и 3 являются недостатками относительно дерева BVH.

BVH (bounding volume hierarchy). В процессе построения иерархии охватывающих оболочек рекурсивно сортируются примитивы (полигоны сцены или другие объекты, представленные оболочками в контексте этого процесса) в 3D-пространстве, организуется древовидная ускоряющая структура (см. рис. 1.7). Внутренние узлы BVH являются охватывающими оболочками AABB (axis-aligned bounding box – шестигранник) для оболочек своих потомков. Каждый примитив может находиться только в одном узле дерева. В дереве BVH, построенном с использованием эвристики SAH, структура охватывающих оболочек организована таким образом, чтобы как можно лучше отделить пустые области пространства от областей, содержащих примитивы.

Преимущества:

1. Предсказуемое потребление памяти до начала процесс построения дерева. Для хранения BVH требуется гораздо меньшее количество узлов дерева, чем для хранения kd-tree [48].
2. Быстрое отсечение пустого пространства для SAH-деревьев в процессе прохода луча. Узлы BVH ограждаются от пустого пространства не отдельными секущими плоскостями, а с помощью охватывающих оболочек (шестигранников).

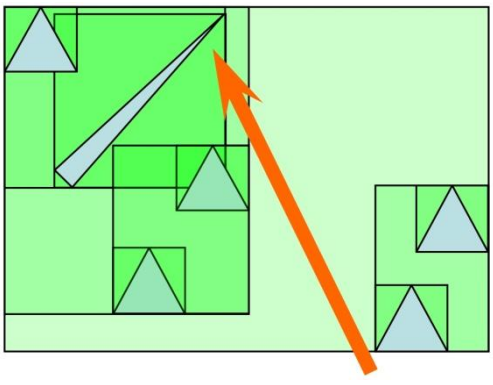


Рис. 1.7: *BVH*

Недостатки:

1. Процесс построения SAH-дерева является медленным: временная сложность алгоритма построения оценивается как $O(n * log\ n)$, где n – кол-во примитивов, используемых в процессе построения дерева.
2. Не строго упорядоченное в 3D-пространстве перечисление листьев дерева вдоль направления луча во время поиска пересечения, т.к. в данном дереве допускаются перекрытия между оболочками узлов BVH.
3. Проблемы с длинными и тонкими треугольниками для AABB узлов (большие объёмы пустого пространства внутри оболочек, см. рис. 1.7).

Комментарий. Ускоряющие структуры kd-tree и BVH имеют общий недостаток 1. Недостатки 3 и 4 для дерева BVH проявляются и для kd-tree (недостаток 2), т.к. длинные и тонкие треугольники могут находиться во множестве узлов kd-tree.

Выбор ускоряющей структуры. В качестве базовой ускоряющей структуры для многих программных продуктов и академических прототипов [12] в задачах реалистичной визуализации часто выбирают дерево BVH, т.к. BVH имеет важные преимущества (особенно важным свойством является предсказуемое потребление памяти). Для недостатка 3 дерева BVH в рамках настоящей работы разработано эффективное решение (см. главу 4). Для ускорения процесса построения дерева (недостаток 1) в рамках настоящей работы разработаны ортогональные методы (главы 2-5), позволяющие для динамических сцен уменьшать множество примитивов в контексте процесса построения дерева (глава 3), использовать для динамических сцен структуру ранее построенного дерева (глава 2), а также задействовать вычислительные мощности графического процессора (главы 4 и 5).

В рамках настоящей работы также использовалась ускоряющая структура типа 3D сетки для организации параллельных прерываний в алгоритме поиска пересечений лучей и сцены на графическом процессоре, где сцена может не вмещаться полностью внутри физической памяти процессора.

Предыдущие алгоритмы построения. С появлением многоядерных архитектур CPU и GPU в академическом сообществе увеличился интерес к разработке новых алгоритмов быстрого и масштабируемого построения ускоряющих структур. В

диссертации [47] содержится подробный анализ и описание свойств, методов построения и применения ускоряющих структур в задачах, основанных на трассировке лучей.

До недавнего времени большая часть исследований фокусировалась на совершенствовании последовательных алгоритмов и оптимизации расхода памяти ускоряющих структур [107]. С изменением парадигмы построения процессорной архитектуры от одного высокочастотного мощного процессорного ядра к архитектуре нескольких ядер более низкой частоты появилось большое количество новых алгоритмов. В работе [36] представлен алгоритм построения и обхода kd-tree на графическом процессоре. Большое количество значительных улучшений алгоритма построения предложено в работах [88][100][123]. Эффективная масштабируемая реализация алгоритма построения kd-tree на многоядерном процессоре была предложена в работе [27]. В работе [113] представлен биновый (в некоторых статьях также переводится как «ячеечный») алгоритм быстрого построения дерева BVH с использованием эвристики SAH, исполняющийся на многоядерном процессоре. С использованием схожих принципов бинового построения разработан алгоритм ещё более быстрого построения BVH, исполняющийся на графическом процессоре [102].

В работах [60][61] представлены алгоритмы построения однородных сеток, а также иерархии сеток на графическом процессоре. Уже упоминались недостатки алгоритмов построения ускоряющих структур применительно к длинным и тонким (т.е. протяжённым) полигонам. В работе [104] представлен алгоритм построения BVH, решающий эту проблему. Полученные в результате работы этого алгоритма деревья BVH обеспечивают самую быструю трассировку лучей среди всех последних опубликованных академических статей. Однако этот алгоритм построения BVH спроектирован для CPU архитектуры и занимает большое количество времени по сравнению с другими.

Алгоритмы LBVH [71], HLBVH [82], реализующие построение BVH на графическом процессоре с использованием Мортон-кодов, более подробно рассмотре-

ны в главе 5 в сравнении с новым более простым и быстрым алгоритмом, разработанным в рамках настоящей работы.

1.5. Поиск пересечения лучей и объектов сцены

Поиск пересечения луча с использованием BVH. В параллельной многопоточной программе в потоке производится поиск пересечения 1 луча и геометрии объектов сцены. Если луч не пересекает оболочку корневого узла *N* дерева BVH, то пересечения для луча нет.

Если луч пересекает оболочку узла *N*, то производятся тесты пересечения луча и оболочек узлов потомков, *NL* и *NR*:

1) Если луч не пересекает оболочки *NL* и *NR*, то из стека вынимается узел, и его указатель записывается в переменную *N*.
2) Если луч пересекает оболочку только узла *NL* (только *NR*), то устанавливается *N* = *NL* (*N* = *NR*).
3) Если луч пересекает оболочки *NL* и *NR*, и при этом пересечение с оболочкой *NL* ближе к основанию луча, чем пересечение с оболочкой *NR*, то устанавливается *N* = *NL*, а узел *NR* записывается в стек.
4) После этого выполняются эти же инструкции для нового значения узла *N*.
5) Если узел *N* является листом BVH, который содержит примитивы сцены, то производятся тесты пересечения луча со всеми примитивами. Точка ближайшего пересечения записывается в контексте луча.

Обзор ранних работ. В академических статьях были представлены методы пакетной трассировки лучей (для собранного пакета лучей производится один обход дерева) и SIMD лучей (один обход дерева производится для пакета из 4 и более лучей): [111][35] для исполнения на CPU.

На CPU часто применяются SIMD инструкции для одновременных произведения тестов пересечений с 4 лучами одновременно, для AVX инструкций – 8 лучей одновременно. На GPU производства NVIDIA векторные вычисления обеспечивают физическую работу одной инструкции для 32 потоков (32 лучей). Пробле-

мой алгоритмов трассировки лучей, при использовании векторных вычислений, является отсутствие когерентности между лучами, расположенными в когерентных потоках в подавляющем большинстве задач расчёта реалистичного переноса света [13]. Одной из попыток решения этой проблемы была разработка процессора фирмы Caustic Graphics [25], спроектированного специально для обработки запросов поиска пересечений лучей и полигонов сцены. Недостатком такого процессора является слишком узкая специализация (поиск пересечений) тогда как в алгоритмах реалистичной визуализации необходимо производить множество других стадий обработки данных, и для этого нужно часто пересылать большой объём данных на центральный процессор общего назначения.

В работах [3][93][80][92] были представлены алгоритмы, основанные на идее трассировки охватывающих (возможно, усечённых) пирамид, содержащих много лучей и амортизации сложности тестов пересечения. Преимуществом этих алгоритмов является экономия вычислений, несмотря на необходимость дополнительной явной сортировки лучей для распределения когерентных лучей и создания пирамид. Однако недостатком является возможность применения только для ограниченного типа лучей: первичные лучи, лучи используемые для расчетов теней и мягких теней, чётко отражённые и преломлённые лучи.

Также существуют подходы [3], основанные на сортировке массива лучей с целью упорядочить когерентные лучи рядом друг с другом и тем самым повысить эффективность обработки на векторной параллельной архитектуре GPU. Пока продемонстрировано преимущество (накладной расход, связанный с сортировкой меньше выигранного времени) такого подхода только для первичных лучей, теневых от протяжённых источников света и зеркально отражённых. Результаты самой быстрой на сегодняшний день трассировки лучей на GPU с применением BVH для сцен, полностью вмещающихся в физической памяти GPU, опубликованы в работе [12].

Генерация огромного количества лучей (десятки миллионов) и их сортировка может позволить создавать большие группы когерентных лучей, для которых в процессе решения задачи переноса света будет осуществляться доступ к когерент-

ным данным сцены. Если необходимо доставлять элементы массивной сцены через медленные каналы доставки (например, при загрузке с диска или удалённого хранилища через сеть рендер-фермы), то подобный подход способен давать большие выигрыши в общем времени реалистичной визуализации [34].

1.6. Архитектура графического процессора

Современные графические процессоры (GPU) являются вычислительно мощными устройствами, устанавливаемыми на многие современные серверные, настольные и мобильные рабочие станции. Наибольшую эффективность GPU обеспечивает для алгоритмов, в которых производятся когерентные вычисления с большим количеством параллельных элементов данных. GPU состоит из нескольких независимых вычислительных ядер [79][39]. Общая вычислительная мощность современного настольного GPU варьируется в диапазоне 1 – 5 Teraflops (триллионов операция с плавающей точкой в секунду), размер физической внутренней памяти GPU варьируется в диапазоне 1 – 6 GB (и 12 GB для самых дорогих версий), пропускная способность доступа к данным памяти GPU составляет не менее 100-150 GB / sec. Хост-процессор (CPU) инициализирует параллельные программы (называемые в английской литературе kernels) для выполнения на GPU, а также управляет логикой запуска различных параллельных программ на GPU в различное время.

Для эффективного использования всей вычислительной мощности GPU необходимо запускать параллельные программы, состоящие из нескольких тысяч (или миллионов) параллельных потоков. Потоки параллельной программы логически разбиваются на блоки (каждый блок потоков физически обрабатывается на единственном ядре GPU). Каждый блок потоков логически разбивается на несколько векторов (в английской литературе называется warp/wave для GPU производства NVIDIA/AMD), где каждый вектор представляет группу из 32 потоков, для которых физически выполняется 1 инструкция (с 32 разными элементами данных) в каждый момент времени на ядре GPU. Если для этой инструкции необходимо длительное ожидание результата (например, чтение данных, находящихся не в кэше ядра), то контекст регистров для текущего вектора сохраняется, результат выпол-

нения операции для этого вектора ожидается, выполнение дальнейших операций вектора откладывается. После этого выбирается другой вектор потоков внутри этого же блока для выполнения других операций, возможно, менее затратных.

Для эффективного исполнения на GPU в любом алгоритме необходимо разбивать работу на подзадачи небольшого размера, выполнение которых будет отображено на блоки потоков параллельной программы, осуществлять когерентный доступ к памяти в рамках одного вектора потоков, обеспечивать загрузку GPU достаточным количеством параллельных потоков.

1.7. Визуализация массивных сцен

Большое количество академических публикаций было посвящено простой (не реалистичной) визуализации массивных (высоко-детализированных) полигональных объектов. В работах [42][31][108] представлены обзоры различных стратегий для визуализации массивных сцен. Во многих алгоритмах использовался подход о прогрессивном упрощении геометрии и построении уровней детализации [22]. Такой подход имеет недостатки при использовании с трассировкой лучей, т.к. требует осуществления большого количества предварительных вычислений, к тому же переходы между уровнями детализации во время визуализации могут не быть бесшовными. В работе [122] используется метод сжатия геометрического представления объектов. В данной работе продемонстрированы высокие коэффициенты сжатия, однако сжатые геометрические данные являются неприменимыми напрямую в процессе трассировки лучей (необходимо декодировать, что требует большого объёма накладных расходов). В работах [69][70] также представлены методы сжатия полигональных объектов, которые могут быть использованы в алгоритмах трассировки лучей, но т.к. кэширования эти алгоритмы не предусматривает, то данные сжатой сцены должны полностью находиться в памяти процессора.

В работе [81] представлена out-of-core[1] система предварительного расчёта текстур мягких теней, называемая PantaRay. Система PantaRay позволяет сохранять долю освещения и затенения на поверхностях геометрических объектов в специ-

[1] Термином out-of-core помечаются алгоритмы, обрабатывающие объёмы данных, которые не вмещаются полностью и одновременно в памяти процессора.

альных текстурах, которые на последующих стадиях могут использоваться для быстрой навигации в сцене. В данной системе разработан механизм кэширования и планирования загрузки больших объёмов данных высоко-детализированных сцен кинофильмов в память GPU.

В настоящей работе (глава 6) использованы некоторые идеи из PantaRay, однако в настоящей работе фокус сделан на применении методов кэширования данных высоко-детализированных на GPU для трассировки лучей общего назначения, без использования предварительных расчётов. Преимуществом настоящей работы является применимость для использования с любыми алгоритмами расчёта переноса света, в том числе и с интерактивными алгоритмами.

В работах [86][24] кроме кэширования используется сложная система упорядочивания различных областей сцены и явное накопление очередей лучей, где каждая очередь соответствует одной области сцены. После заполнения каждой очереди для содержащихся в ней лучей производится поиск пересечения лучей с геометрией, содержащейся в области. В работе [24] также производится распределение очередей задач для выполнения на различных вычислительных ресурсах, содержащихся в компьютере (на ядра CPU и GPU). Подобные системы позволяют реалистично визуализировать очень большие сцены, значительно превышающие объёмы имеющейся памяти процессоров. Однако в рамках настоящего проекта (глава 6) целью является разработка алгоритма трассировки лучей, применимого для интерактивных приложений.

В работах [120][108] представлены алгоритмы out-of-core трассировки лучей на CPU для массивных сцен. Был использован подход, основанный на создании уровней детализации геометрического представления сцен, подсистеме менеджера виртуальной памяти операционной системы. Эти алгоритмы предназначены для простой визуализации сцен без расчётов реалистичного глобального освещения. Также они плохо масштабируются на многоядерных CPU и требуют произведения большого объёма предварительных вычислений.

1.8. Заключение

В данной главе представлен обзор существующих достижений и проблем в реализации основных стадий конвейера реалистичной визуализации. В рамках настоящей работы фокус направлен на разработку алгоритмов, позволяющих:

1. Ускорить процесс построения эффективной ускоряющей структуры (главы 2-5**)**, обеспечивающей высокую скорость поиска пересечения лучей. Ускорение этого процесса является актуальным для анимированных массивных сцен в условиях постоянного роста требований в детализации виртуальных сцен и сложности моделей.
2. Обеспечить решение задачи быстрого поиска пересечения лучей для массивных сцен (содержащих сотни миллионов полигонов) на графическом процессоре (глава 6), имеющем преимущество мощных параллельных вычислений и недостаток в малом размере физической памяти. Решение этой задачи актуально для всех реалистичных алгоритмов визуализации, т.к. в них используется поиск пересечений лучей, и эта стадия занимает существенную долю (25-75%) времени общего конвейера визуализации. Особенно актуальным решение этой задачи является в условиях роста требований в детализации сцен и сложности моделей.

Глава 2

Алгоритм обновления ускоряющей структуры с учётом ранее построенной ускоряющей структуры

В данной главе описан разработанный автором алгоритм эффективного обновления существующей иерархии охватывающих оболочек (сокр. BVH, см. раздел 1.4) на основе структуры ранее построенной иерархии BVH (впервые опубликован в статье [1]). Данный алгоритм применяется для организации полигонов внутри дерева BVH различных типов динамических сцен и позволяет производить деревья, обеспечивающие эффективную трассировку лучей. Разработанный алгоритм объединяет достоинства нескольких более ранних методов, адаптивно уменьшая сложность операций перестроения BVH в каждом кадре в процессе анимации сцены. Минимизация сложных операций перестроения BVH в каждом кадре анимации осуществляется благодаря использованию групп треугольников, которые когерентно перемещаются в сцене. Таким группам соответствуют поддеревья BVH, и в процессе обновления структуры BVH переставляется корень всего поддерева с целью минимизации ожидаемого времени трассировки лучей, обеспечиваемого новой структурой BVH. Таким образом, алгоритм эффективно обрабатывает структурированное движение. Также данный алгоритм определяет группы «взорвавшихся» треугольников (т.е. подвергнувшиеся хаотичному движению) для произведения операций перестроения структуры внутри поддерева BVH, соответствующего группе. Эффективная локализация операций перестроения топологии на несколько независимых, непересекающихся поддеревьев BVH в значительной мере уменьшает вычислительную сложность операций перестроения по сравнению с алгоритмом перестроения всей топологии BVH.

2.1. Алгоритм обновления BVH

На входе алгоритм обновления принимает дерево BVH, построенное для предыдущего кадра анимации и новые координаты вершин треугольников. Алгоритм обновления состоит из следующих трёх стадий:

1) стадия адаптации (обновление координат охватывающих оболочек BVH без обновления топологии дерева);
2) стадия перестроения (полное перестроение структуры поддерева для всех найденных на стадии адаптации поддеревьев, в которых необходимо произвести перестроение структуры);
3) стадия миграции (перестановка поддеревьев, соответствующих группам когерентно движущихся треугольников, в новые позиции в дерева).

«***Взорванным узлом***» дерева BVH называется узел дерева, являющийся корнем поддерева BVH, которое соответствует группе хаотично движущихся треугольников. «***Мигрирующим узлом***» дерева BVH называется тот узел дерева, являющийся корнем поддерева BVH, которое соответствует группе когерентно движущихся треугольников. Узлы дерева определяются как взорванные или мигрирующие во время стадии адаптации с применением эвристических оценок, и обрабатываются в стадиях перестроения и миграции соответственно.

Эвристическое определение мигрирующих узлов дерева. Внутри некоторого узла *N* оценка степени расхождения прямых потомков *NL* и *NR* осуществляется с помощью вычисления значения SAH функции для узла *N* (см формулу 1.3). Изначальное значение SAH функции (1.3) рассчитывается для каждого внутреннего узла *N* и сохраняется в поле *SavedSAHCost(N)*. Дополнительно, количество всех треугольников, объединённых узлом *N*, сохраняется в поле *Count(N)*. Значение *SAHCost(N)* (1.3) вычисляется для каждого внутреннего узла *N* дерева BVH после обновления охватывающей оболочки *BV(N)* во время каждого шага стадии адаптации. Если значение *SAHCost(N)* становится меньше сохранённого ранее значения *SavedSAHCost(N)*, то узлы *NL* и *NR* определяются как мигрирующие узлы и поддеревья, соответствующие NL и NR являются кандидатами на смену позиции в структуре BVH:

$$\frac{SAHCost(N)}{SavedSAHCost(N)} < 1 - dM \qquad (2.1)$$

где *dM* – значение в интервале от 0 до 1, устанавливаемое пользователем для контроля множества поддеревьев, которые необходимо переставить.

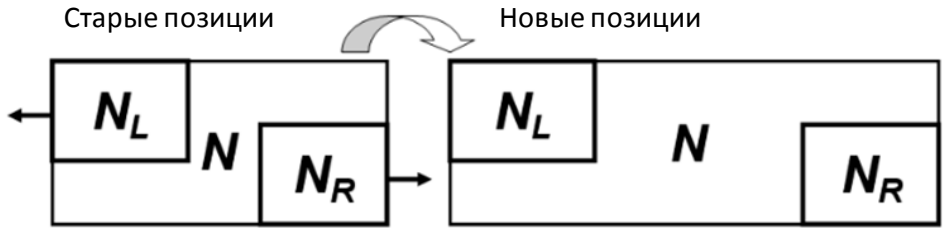


***Рис. 2.1**: Расхождение узлов NL и NR (кадр «до» и «после» движения), являющихся непосредственными потомками узла N. Узлы NL и NR определяются как мигрирующие относительно друг друга с помощью вычисления функции (2.1).*

Пояснение на примере. Пусть даны два поддерева с корнями *NL* и *NR* соответственно, которые соответствуют двум различным объектам, движущихся в противоположных направлениях (см. рис. 2.1), причём *NL* и *NR* являются непосредственными потомками *N*. Охватывающая оболочка *BV(N)* при этом увеличивается, это отражается в увеличении *SA(N)* по сравнению с суммой *SA(NL)* и *SA(NR)* (см. формулу 1.3). Если значения *Count(NL)* и *Count(NR)* остаются постоянными в течение времени, то движение объектов, соответствующих *NL* и *NR* отражается в уменьшении значения *SAHCost(N)*. Если объекты, соответствующие *NL* и *NR*, движутся в противоположных направлениях, то они могут приблизиться к другим объектам и охватывающая оболочка их родительского узла *BV(N)* может пересекать оболочки других объектов. В этом случае, если не изменить структуру BVH, то скорость трассировки лучей может снизиться.

Пусть значение *Count(NL)* или *Count(NR)* уменьшается после использования для вычисления значения *SavedSAHCost(N)*, это означает, что количество примитивов уменьшается внутри поддеревьев *NL* или *NR*. Такая ситуация возможна, только если оболочки некоторых узлов потомков для *NL* или *NR* были подвергнуты операции адаптации (см. ниже) после осуществления миграции их узлов потомков. Но миграция узлов внутри дерева, произведённая несколько раз, может повлечь за собой эффект «утоньшения» дерева: высота дерева может стать намного больше, например, в 2 раза. Такая ситуация внутри поддерева, соответствующего узлу *N*, может быть обнаружена, если неравенство (2.1) верно. Значения *Count(NL)* и *Count(NR)* могут увеличиваться, только если процесс вставки какого-либо поддерева спускается глубже в дереве, однако в этом случае значение *SavedSAHCost(N)* обновляется.

Определение взорванных узлов дерева. Взорванным регионом дерева считается поддерево BVH с корнем *N*, внутри которого существует большое количество перекрывающихся охватывающих оболочек между всевозможными узлами-потомками *N* (см. рис. 2.2).

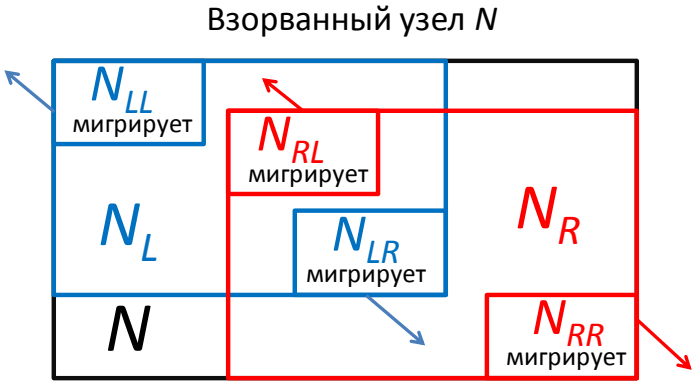


***Рис. 2.2**: Пример перекрытия оболочек узлов NL и NR, для которого узел N определяется как взорванный. Также показаны потомки узлов NL и NR, которые определены как мигрирующие в противоположные направления.*

Доля перекрытия между узлами *NL* и *NR*, потомками узла *N*:

$$MaxOverlap(N) = \frac{SA(OverlapBV(NL, NR))}{\min(SA(NL), SA(NR))} \tag{2.2}$$

где *OverlapBV(NL,NR)* – охватывающая оболочка пересечения между *узлами NL* и *NR*. *MaxOverlap(N)* принимает значения в интервале от 0 до 1. Узел N считается взорванным, если *MaxOverlap(N)* имеет высокое значение (например, большее, чем 0.3) и если выполняется условие:

$$\frac{SAHCost(N)}{SavedSAHCost(N)} > 1 + dE \tag{2.3}$$

где *dE* – значение в интервале от 0 до 1, устанавливаемое пользователем для контроля над множеством поддеревьев, внутри которых будет производиться полное перестроение структуры.

2.2. Стадия адаптации

Данная стадия обновляет охватывающие оболочки всех узлов дерева BVH в соответствии с новыми координатами вершин полигонов, для которых необходимо сформировать дерево BVH. Структура связей дерева при этом не меняется. Начиная с корневого узла, все узлы дерева BVH обходятся рекурсивно. В порядке сни-

зу-вверх (после возврата управления рекурсивного вызова), охватывающие оболочки *BV(N)* узлов обновляются в соответствии с охватывающими оболочками непосредственных узлов потомков, для которых охватывающие оболочки уже были рассчитаны на предыдущем шаге рекурсии. Для каждого листового узла охватывающая оболочка рассчитывается в соответствии с координатами всех треугольников, принадлежащих узлу.

```
void adapt_tree(node N)
{
  if(is_leaf(N))
  {
    BV(N) = Union(BV(all leaf polygons))
  }
  else
  {
    adapt_tree(N.NL);
    adapt_tree(N.NR);
    BV(N) = Union(BV(N.NL), BV(N.NR));
    Detect_migrated_or_exploded_node(N);
  }
}
```

Кроме обновления охватывающих оболочек во время стадии адаптации каждый внутренний узел определяется как мигрирующий или взорванный или ни тот ни другой. Для этого используются формулы (2.1), (2.2) и (2.3). Указатели на мигрирующие или взорванные узлы добавляются в соответствующие массивы: *MigrateBuffer* или *RebuildBuffer*.

Если узлы *NL* и *NR* (прямые потомки *N*) определены как мигрирующие узлы относительно друг друга, то в *MigrateBuffer* добавляется узел *N*. Если узел *N* определён как взорванный, то он добавляется в массив *RebuildBuffer*.

Затем для каждого взорванного узла *N* производятся следующие 3 действия:

1. Узел *N* помечается флагом блокировки: *Locked(N) = true*. Это делается для того, чтобы использовать оболочку узла *N* как неделимый примитив в контексте возможного перестроения поддерева, если такое потребуется выполнить для какого либо узла предка *N*, определённого взорванным в более поздних шагах рекурсии процесса адаптации. Это действие позволяет производить операции независимого перестроения для всех поддеревьев, представленных блокированными корнями.

2. Оболочка *BV(Ni)* каждого узла *Ni*, являющегося потомком *N*, который был добавлен в массив *RebuldBuffer*, проверяется на пересечение с *OverlapBV(NL,NR)* – охватывающей оболочкой пересечения оболочек узлов *NL* и *NR* (являющихся прямыми потомками *N*). Если *BV(Ni)* пересекает *OverlapBV(NL,NR)*, то флаг блокировки для *Ni* снимается: *Locked(Ni) = false*. Т.е. оболочка узла *Ni* не будет использоваться как единый примитив в контексте операции перестроения для поддерева, начинающегося с *N*. Использование *Ni* как неделимого примитива в контексте операции перестроения послужило бы большим препятствием в удалении перекрытий в поддереве, начинающегося с *N* (см рис. 2.3). Другие узлы, оболочки которых не пересекают *OverlapBV(NL,NR)* не изменяют значения блокировки *Locked*, т.к. их влияние на процесс устранения перекрытия в поддереве *N* является минимальным.

3. Все потомки узла *N* удаляются из массива *MigrateBuffer*, если они были занесены в него ранее, т.к. они будут реструктурированы в поддереве с помощью операции перестроения.

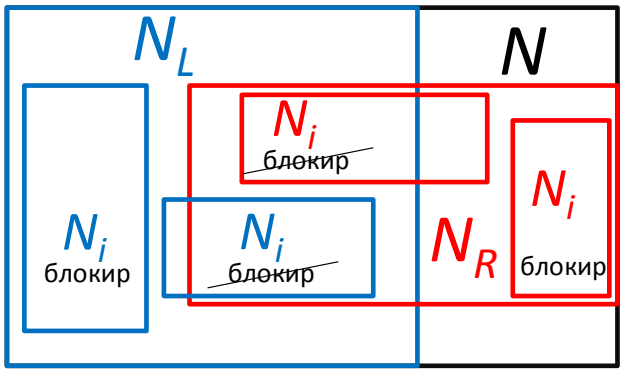


***Рис. 2.3**: Если N определён как взорванный узел, то для каждого его потомка, оболочка которого пересекает оболочку OverlapBV(NL, NR), блокировка снимается. Целью перестроения поддерева, начинающегося с N, является удаление перекрытия OverlapBV(NL, NR) и любых других перекрытий в поддереве.*

2.3. Стадия перестроения топологии поддеревьев

Процесс перестроения осуществляется для каждого поддерева BVH, корневой узел *Ni* которого был определён как взорванный на стадии адаптации. Каждый такой узел *Ni* записан в массиве *RebuildBuffer* и отмечен символом *L* на рис. 2.4 (сокращённо означает установленный флаг *Locked(Ni)=true*).

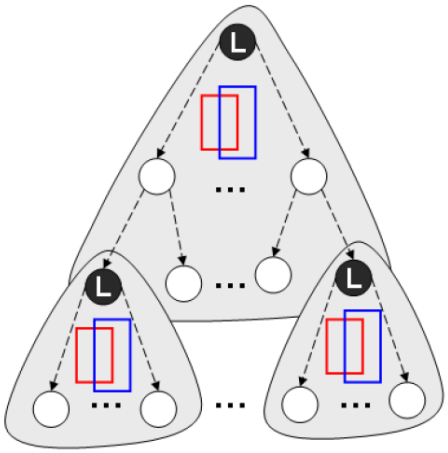


***Рис. 2.4**: Представлены поддеревья BVH, начинающиеся с узлов Ni, отмеченные символом L (т.е. имеющие флаг блокировки Locked(Ni)=true), и ограниченные либо листовыми узлами BVH, либо другим внутренними узлами, отмеченным символом L. Внутри этих поддеревьев производится полное перестроение структуры. На данном изображении каждое поддерево очерчено фигурой с серой границей.*

Процесс полного перестроения топологии для каждого поддерева BVH с корнем *Ni* использует в качестве примитивов охватывающие оболочки листовых узлов BVH или охватывающие оболочки заблокированных внутренних узлов BVH, имеющие флаг *Locked=true*. Элементы множества оболочек, используемых в процессе перестроения поддерева, собираются с помощью рекурсивной функции, обходящей все узлы поддерева сверху вниз, начиная с корня поддерева *Ni* и заканчивая либо листовым узлом BVH, либо заблокированным внутренним узлом *Nj*, имеющим флаг *Locked(Nj)=true* (см. рис. 2.4).

Процесс перестроения структуры каждого поддерева является рекурсивной операцией разделения множества оболочек, где на каждом шаге процесс разделения принимает узел *N* и множество оболочек *S*, которые должны быть разделены в границах оболочки *BV(N)*. Множество *S* разделяется на два непересекающихся подмножества *SL* и *SR*, так чтобы минимизировать возможные перекрытия между охватывающими оболочками подмножеств *SL* и *SR* соответственно, где двум новым подмножествам соответствуют новые узлы *NL* и *NR* соответственно. Далее, по рекурсии, множества оболочек *SL* и *SR* разделяются на новые подмножества в границах оболочек узлов *NL* и *NR* соответственно. Любой алгоритм полного построения дерева BVH (например, [113], см. обзор в разделе 1.4) может использоваться

как основа для поиска оптимального разделения множеств оболочек на 2 части и генерации новой структуры поддерева. Значения *Count(Nk)* и *SavedSAHCost(Nk)* рассчитываются заново для всех узлов поддерева.

2.4. Стадия миграции поддеревьев

Для каждой пары узлов, которые были определены как мигрирующие (см. рис. 2.2), только их узел предок сохраняется в массиве *MigrateBuffer*. Все элементы массива *MigrateBuffer* обрабатываются в 4 цикла (в каждом цикле узлы из массива *MigrateBuffer* обрабатываются в порядке от конца к началу, чтобы сначала обработать поддеревья, находящиеся на нижних уровнях дерева BVH):

1. Отсоединить каждый узел *N* из массива *MigrateBuffer* от дерева BVH.
2. Адаптировать оставшуюся часть дерева (пересчитать охватывающие оболочки) в порядке снизу-вверх, начиная с узла-родителя отсоединённого узла *N*.
3. Для узла *N* найти узел *X* в дереве, начиная с которого можно начинать вставлять *N* (в адаптированном дереве оболочка *N* должна полностью входить в область ограниченную оболочкой *X*; поиск такого узла *X* начинается с места отсоединения узла *N* и заканчивается корнем дерева).
4. Вставить прямые потомки узла *N* в адаптированное дерево.

Адаптация дерева после отсоединения поддерева, начинающегося с узла *N*, представляет собой процесс обхода узлов дерева, начиная с места отсоединения *N* и заканчивая корнем дерева. Применяется переход на родительские узлы (ссылка на узел-родитель хранится для каждого узла *X* в поле *Parent(X)*) и обновление оболочки *BV(X)* и счётчика *Count(X)* вышестоящего узла на основе рассчитанных ранее данных их потомков. Если в результате обновления оболочка *BV(X)* не изменяется, то узел *X* является узлом, начиная с которого можно вставить отсоединённое поддерево с корнем *N*. В данном случае последовательность шагов адаптации дерева заканчивается на узле *X*. Необходимо вставить поддерево с корнем *N* обратно в дерево таким образом, чтобы избежать ухудшения ожидаемого времени трассировки лучей, обеспечиваемое новой структурой дерева, т.е. чтобы минимизировать количество перекрытий среди оболочек узлов.

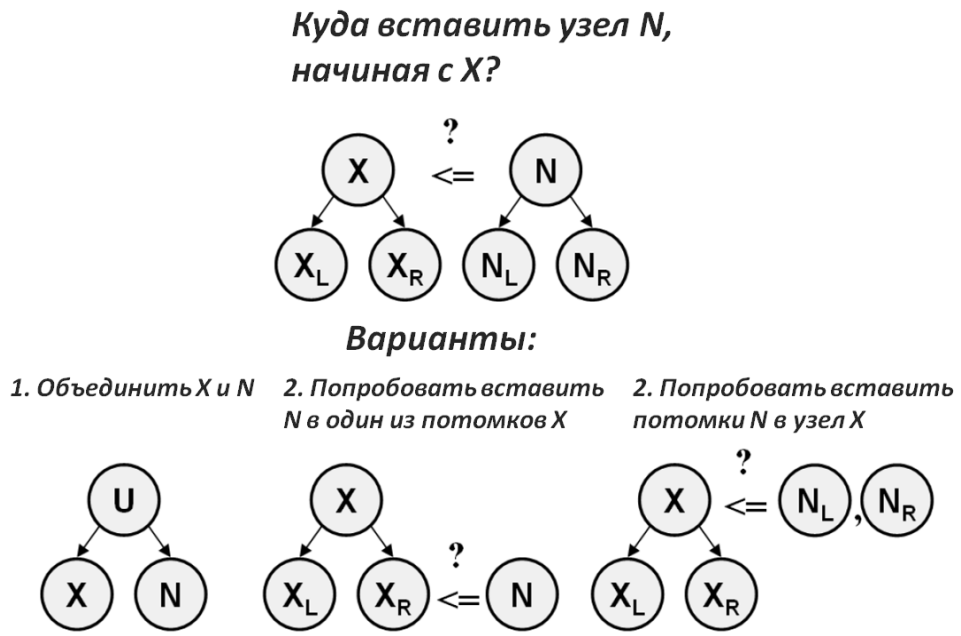


***Рис. 2.5**. Когда поддерево, представленное узлом N, необходимо вставить в дерево, начиная с узла X, то один из следующих вариантов решения принимается: 1) объединить узлы X и N, окончить рекурсию; 2) попробовать вставить узел N в дерево, начиная с узла XL или XR, являющиеся прямыми потомками X; 3) разделить узел N на два его потомка NL и NR, и вставить их в дерево, начиная с X.*

Процесс вставки поддерева с корнем *N* в дерево, начиная с узла *X* в дереве, является рекурсивным и представлен на рис. 2.5. Каждый шаг процесса вставки использует следующие входные параметры: *X* – узел с которого начинается вставка в принимающем дереве, *N* – корневой узел поддерева, которое нужно вставить в некоторое место принимающего дерева под узлом или на уровне *X*.

На каждом шаге процесса вставки принимается один из вариантов решения:

1) Объединить узлы *X* и *N* под новым узлом *U* и остановить рекурсивный процесс вставки.
2) Понизить решение о вставке поддерева *N* на уровень ниже, т.е. пытаться вставить *N*, начиная с прямых потомков *XL* и *XR* узла *X*.

Эти варианты решения ассоциируются с соответствующим значением SAH функции (см. формулу 1.3), которое вычисляется для рассматриваемых конфигураций узлов. Применяется тот вариант решения задачи вставки, который соответствует наименьшему значению SAH функции (1.3). Если выбран 2-ой вариант и при

этом образовалось (или увеличилось) перекрытие оболочек между *XL* и *XR*, прямыми потомками *X*, то он заменяется на 3-ий вариант решения задачи вставки:

3) Разделить узел *N* на его прямые потомки *NL* и *NR* и вставить каждый из них в дерево независимо, начиная с узла X.

На каждом шаге процесса вставки значения *Count(X), SavedSAHCost(X), BV(X)* обновляются. Эвристический метод обнаружения мигрирующих узлов определяет даже случаи, когда один объект статичен, а соседний объект движется в сторону от первого. Поддеревья, соответствующие обоим объектам будут переустановлены в другие места дерева BVH, в результате чего повысится ожидаемая оценка скорости трассировки лучей, обеспечиваемая построенным деревом.

2.5. Менеджер памяти

Стадии перестроения топологии и миграции поддеревьев производят интенсивные обновления указателей, связывающих узлы в дереве. Если не предпринимать никаких дополнительных мер в работе с менеджером памяти, выделяющим и освобождающим память для каждого узла дерева, то в результате нескольких шагов реструктуризации структуры дерева (перестроение и миграция) может значительно снизиться эффективность дерева для трассировки лучей. Это происходит, потому что прямые потомки узлов и их предки могут находиться в некогерентных участках памяти и расстояние между ними может превышать размер кэш-линии процессора. Потребление памяти для дерева типа BVH является ограниченным (требуется не более *2n-1* узлов для организации *n* примитивов). Количество освобождаемых узлов во время отсоединения поддеревьев равно количеству создаваемых узлов во время вставки поддеревьев в другие места дерева.

Расстановка узлов дерева BVH в порядке поиска в глубину (когда узел и один из его потомков располагаются в памяти рядом) или в порядке поиска в ширину (когда пара прямых потомков любого узла располагается в памяти рядом) является оптимальной и эффективной для обеспечения быстрой трассировки лучей [49]. Менеджер памяти для такой задачи строится следующим образом:

1) Выделяется линейный массив узлов дерева BVH. Над ним строится поисковая структура MEM в виде сбалансированного дерева высокой арности (например, октарное).
2) Каждый узел дерева менеджера памяти MEM соответствует блоку узлов BVH и содержит только счётчик количества свободных узлов BVH. Ссылки на узлы потомки в дереве MEM хранить не нужно, т.к. оно сбалансированное.
3) При запросе на выделение узла-потомка для ранее выделенного узла-предка *Ni* в менеджере памяти MEM производится поиск свободного узла. Во время поиска из всех свободных узлов выбирается узел, у которого адрес расположения в памяти ближайший к *Ni*.
4) Во время выделения узла соответствующие счётчики свободных узлов поисковой структуры MEM уменьшают свои значения. Во время освобождения узлов счётчики увеличивают свои значения.

Подобная поисковая структура MEM позволяет генерировать деревья BVH, где узлы-предки и потомки располагаются в памяти когерентно, и, несмотря на частые операции отсоединения и присоединения поддеревьев в разных местах BVH, общая эффективность BVH остаётся высокой для обеспечения быстрой трассировки лучей.

2.6. Результаты

Реализованный алгоритм обновления ускоряющей структуры BVH испытан на компьютере с CPU Intel Core 2 Quad 2.4 GHz, 8Mb of L2 Cache, 1 GB of 800MHz DIMM DDR2 RAM, GPU GeForce 8800 GTX. Подпрограмма обновления ускоряющей структуры BVH написана на языке C++, использует только один поток (одно ядро процессора) и скомпилирована с помощью Intel Compiler 10.1.

В качестве тестовых сцен используются три динамические сцены из хранилища сцен Университета Северной Каролины [105]. В используемых сценах представлено три основных типа движения: взрыв и изменение топологии объекта, движение твёрдых тел, деформация объекта (см. рис. 2.6).

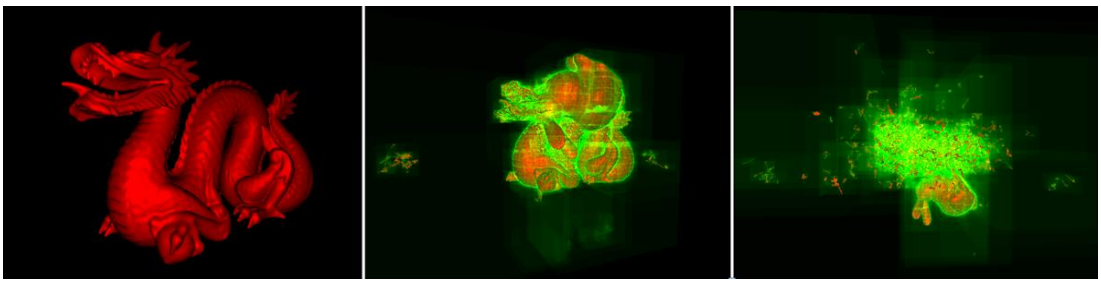


Рис. 2.6-а: *Взрывающийся дракон. На этом иллюстрации показаны 3 кадра из динамической симуляции взрывающегося дракона (252K треугольников). Кролик разбивает дракона на кусочки, это влечёт за собой сильные изменения топологии сцены. BVH узлы подсвечены на среднем и правом изображении зелёным цветом.*

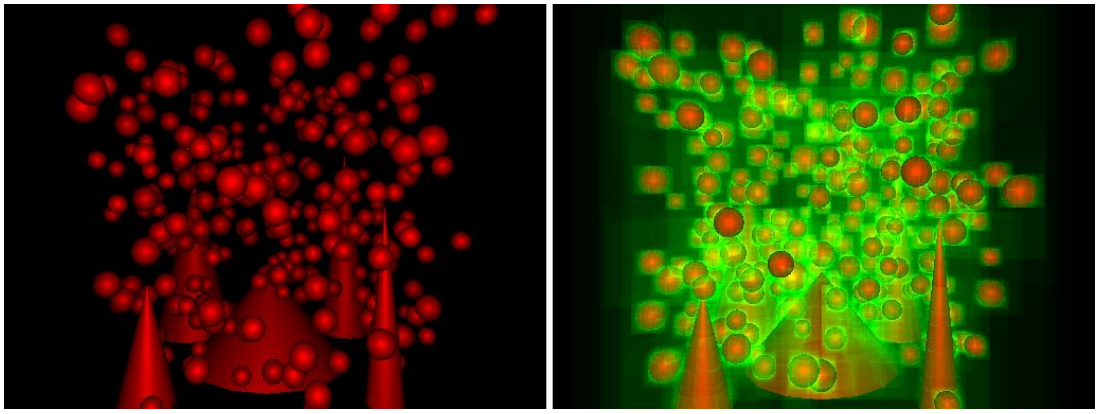


Рис. 2.6-б: *Симуляция хаотичных столкновений N тел (146K треугольников).*

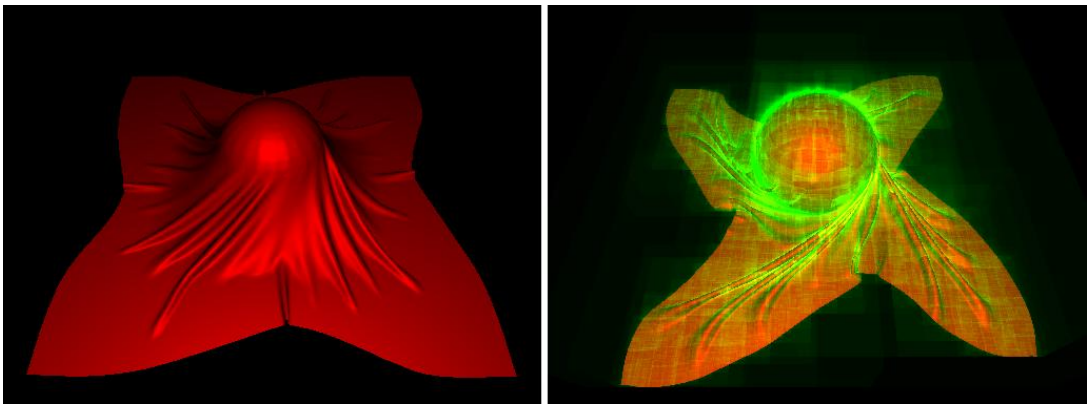


Рис. 2.6-в: *Симуляция движения ткани на шаре (92K треугольников).*

В каждой из 3 тестовых сцен динамические изменения положения объектов производятся в течение моделируемых 3.33 секунд анимации сцены, соответствующих 100 кадрам видеопоследовательности. Используются следующие значения параметров разработанного алгоритма обновления BVH:

dM = 0.05 – для определения мигрирующих узлов (см. формулу 2.1).

MaxOverlap(N) > 0.5 – большая доля перекрытия (см. формулу 2.2).

dE = 0.01 – для определения взорванных узлов (см. формулу 2.3).

Для вышеперечисленных сцен суммарное время обновления дерева BVH (см. рис. 2.7) зависит, в основном, от времени стадии перестроения. В моментах значительных (взрывных) динамических изменений возникает необходимость производить множество операций перестроения поддеревьев (см. рис. 2.8). Практика показывает, что в случаях значительных (взрывных) динамических изменений в сцене имеется несколько сотен поддеревьев, структуру которых необходимо перестроить (см. рис. 2.9). Эти поддеревья можно перестраивать независимо с применением многоядерного процессора, имея при этом хорошую загрузку для всех вычислительных ядер. Стадия миграции поддеревьев является вычислительно недорогой для тестируемых сцен, однако благодаря этой стадии сокращается число дорогих операций перестроения поддеревьев BVH. Эта стадия является особо выигрышной для сцен с элементами когерентного движения.

Например, на начальных этапах анимации сцены столкновений N тел производится множество операций перестроения поддеревьев (см. рис. 2.8), потому что для первого кадра данной симуляции все тела сконцентрированы в пространстве в плотную кучу, а потом тела хаотично двигаются в разные стороны, соударяясь между собой, но не подвергаясь разрушениям (т.е. топология геометрических примитивов внутри этих тел сохраняется). Поэтому на начальных этапах данной сцены алгоритм построения BVH не может ассоциировать отдельные тела с отдельными поддеревьями BVH. В этой сцене после 30-го кадра (см. рис. 2.8) число операций перестроения приближается к нулю. Даже после прокрутки симуляции до 100-го кадра, потом прокрутки назад до 1-го кадра и повторной прокрутки вперёд до 100-го кадра количество дорогостоящих операций перестроения поддеревьев остаётся близким к нулю (на уровне 1%), при этом стадия миграции выполняет почти всю работу по обновлению структуры BVH. Подобный эффект наблюдается, т.к. с течением времени алгоритм обновления BVH, который учитывает характер движения и ранее построенную структуру BVH, сопоставляет каждому телу поддерево BVH и производит вычислительно дешёвые операции миграции поддеревьев для обеспечения эффективной трассировки лучей.

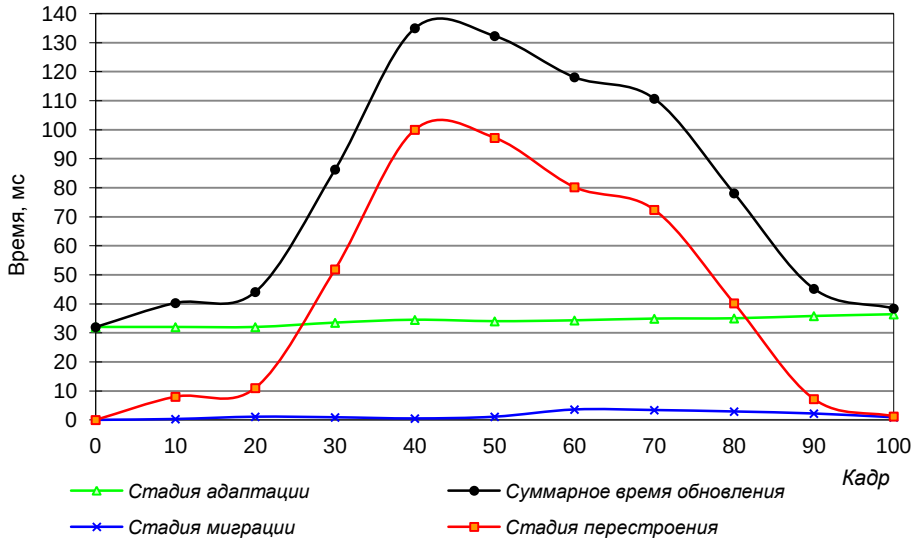


***Рис. 2.7-а**: Замеры времени симуляции взрыва дракона*

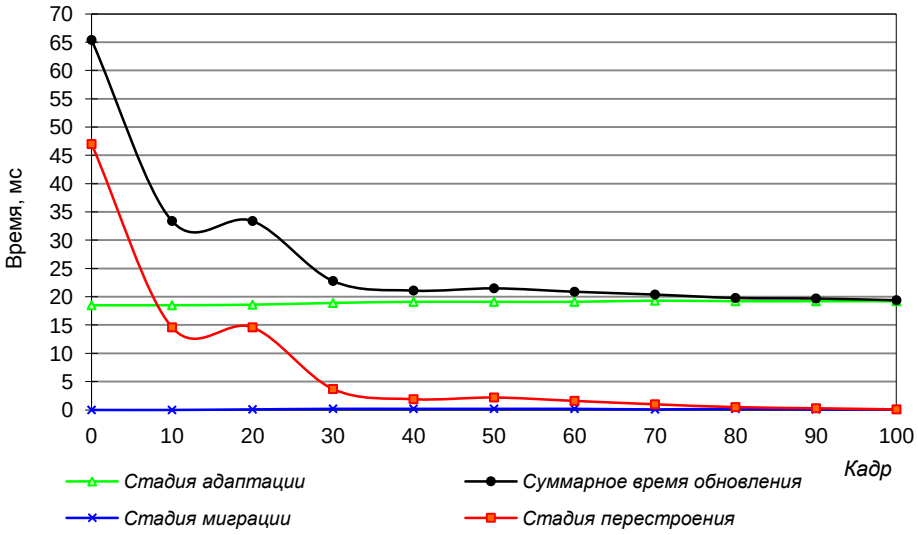


***Рис. 2.7-б**: Замеры времени симуляции столкновений N тел*

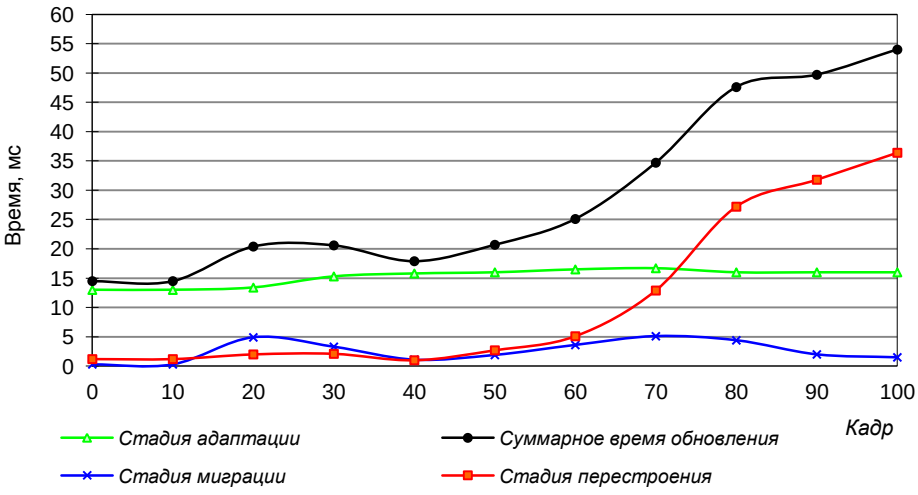


***Рис. 2.7-в**: Замеры времени симуляции ткани на шаре.*

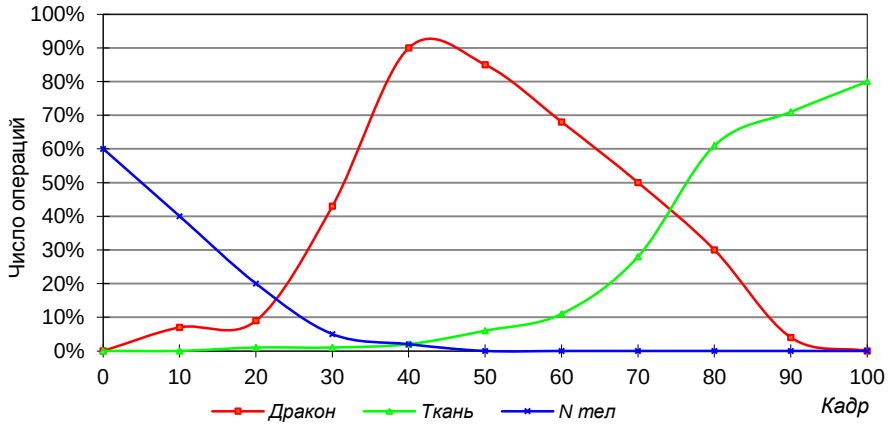


Рис. 2.8: *Число операций перестроения в разработанном комбинированном алгоритме, учитывающем структуру ранее построенного BVH, по отношению к числу операций в алгоритме перестроения BVH, не учитывающего структуру ранее построенного BVH.*

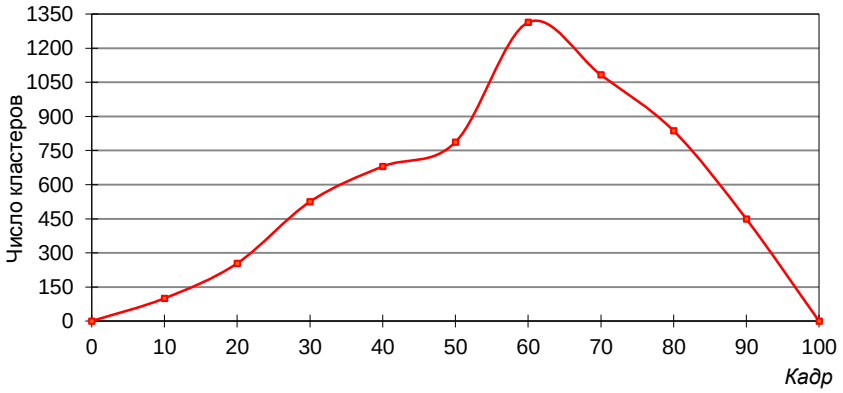


Рис. 2.9*: Число поддеревьев BVH, структуру которых необходимо построить заново в стадии перестроения для сцены взрывающегося дракона.*

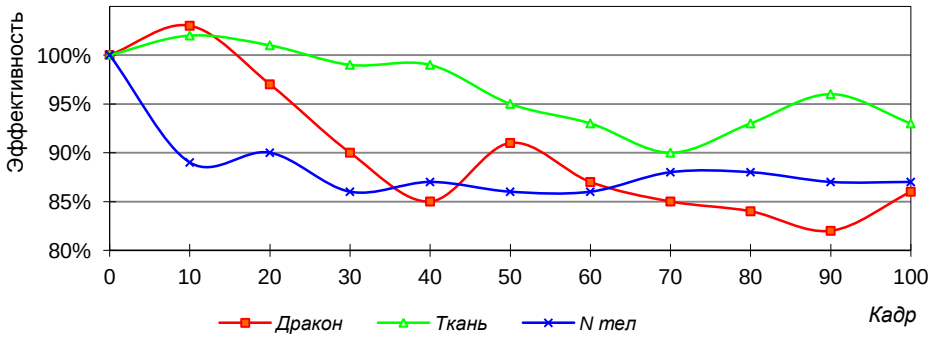


Рис. 2.10*: Эффективность дерева BVH, построенного с помощью алгоритма, использующего структуру ранее построенного BVH по отношению к дереву BVH, построенного с помощью алгоритма перестроения без учёта структуры ранее построенного BVH. В качестве меры эффективности дерева BVH используется скорость трассировки лучей, которая обеспечивается этим деревом.*

Скорость трассировки лучей, которая обеспечивается деревом BVH, созданным новым более быстрым алгоритмом, не более чем на 10-15% ниже скорости трассировки лучей, которая обеспечивается деревом BVH, построенным (медленным) алгоритмом полного перестроения структуры дерева (см. рис. 2.10). Причиной подобного падения скорости трассировки лучей для нового алгоритма является тот факт, что в стадии миграции поиск места для вставки поддеревьев производится путём последовательного перехода от узла предка к потомку, без рассмотрения всего уровня дерева в ширину.

Во время экспериментов с алгоритмом обновления BVH было установлено, что обеспечиваемая скорость трассировки лучей значительно падает (более чем на 40%), если в алгоритме обновления убрать одну из стадий и использовать только стадию перестроения или только стадию миграции. Однако комбинация стадии перестроения и миграции позволяет производить деревья BVH приемлемого качества для визуализации анимированных сцен методом трассировки лучей.

Установленные параметры для алгоритма обновления BVH (*dE=0.05*, *dM=0.01*, *MaxOverlap > 0.5*) являются оптимальными для динамических сцен различного типа. Принимая во внимание характер движения в анимированной сцене, и изменяя параметры алгоритма обновления BVH, можно добиться более высокой скорости обновления BVH для конкретной сцены, но это может привести к дополнительному падению скорости трассировки лучей, обеспечиваемой таким деревом BVH или менее эффективно работать для других анимированных сцен.

2.7. Заключение

В данной главе был представлен алгоритм, который эффективно обновляет дерево BVH. Этот алгоритм производит наименее ресурсоёмкие операции обновления дерева, минимизируя частые вызовы сложных операций перестроения поддеревьев. Для того чтобы уменьшить число сложных операций перестроения, алгоритм комбинирует адаптацию BVH, миграцию поддеревьев BVH и локализованное перестроение структуры поддеревьев BVH.

Обновление BVH с помощью этого алгоритма для тестовых сцен осуществляется в 2-4 раза быстрее по сравнению с алгоритмом полного перестроения BVH, не учитывающего старой структуры BVH.

В анимациях с элементами когерентного движения алгоритм задействует в основном только стадию миграций поддеревьев. Специальный менеджер памяти поддерживает приемлемую эффективность размещения BVH в памяти в условиях существующей кэш-иерархии процессора. Генерация сотен и тысяч независимых поддеревьев для перестроения даже в симуляциях взрывов позволит задействовать множество процессорных ядер для ускорения процесса обновления BVH.

Глава 3

Алгоритм построения ускоряющей структуры с использованием кластеризации входных данных

В данной главе представлен алгоритм быстрого построения дерева BVH для динамических 3D сцен с использованием предположения сохранения связности полигонов в сцене (впервые опубликован в статье [2]).

Во многих случаях анимация сцены осуществляется при помощи изменения координат вершин полигонов, в то время как связи между треугольниками остаются неизменными (т.е. индексы вершин остаются постоянными в структуре данных каждого треугольника). Подобные анимированные сцены называют деформируемыми [110]. Анимированные сцены, содержащие взрывающиеся объекты, можно запрограммировать с сохранением связей треугольников в процессе анимации, если предварительно определить обломки объектов и места разрыва [90].

Если в геометрической модели, представляющей объект отношение *Кол-во Вершин / Кол-воТреугольников < 3*, то среди треугольников модели есть доля связности (т.е. на некоторые вершины ссылаются несколько треугольников). В реальных сценах треугольник не является отдельным объектом – это всего лишь примитив геометрического представления. Даже небольшое число связанных треугольников представляют некоторый объект.

С использованием предлагаемого в этой главе подхода можно ускорить любой алгоритм построения BVH. Ускорение процесса построения BVH достигается при помощи уменьшения количества примитивов, которые учитываются в процессе построения BVH. В качестве примитивов используются кластеры треугольников, ограниченные охватывающими оболочками. В каждом таком кластере содержится несколько связанных треугольников. За счёт этого достигается ускорение стадии построения BVH по сравнению с традиционными алгоритмами построения BVH, где в качестве примитивов, учитывающихся во время построения, используются отдельные треугольники.

Конфигурация кластеров треугольников для каждого динамического объекта сцены предварительно рассчитывается до начала анимации. Полученные кластеры с обновлёнными координатами вершин используются в процессе полного перестроения BVH в каждом кадре анимации сцены. Предполагается, что индексы вершин, на которые ссылаются треугольники, остаются постоянными в процессе анимации. Данное предположение является допустимым для широкого класса анимированных сцен. При этом анимация может быть иерархической, деформируемой или взрывной. Проектирование эвристики, использующейся для генерации кластеров, основано на модели геометрической вероятности и предположении когерентного движения связанных треугольников, принадлежащих одному кластеру.

3.1. Проектирование эвристики генерации кластеров

Геометрическая вероятность. Пусть заданы X и Y – выпуклые области 3D пространства, X включает в себя Y, т.е. $X \cap Y = Y$ (см. рис. 3.1a). Согласно выводам в статье [47] $P(Y \mid X) = SA(Y) / SA(X)$ – условная вероятность, что произвольный луч, пересекающий оболочку области X, пересечёт оболочку области Y, где $SA(X)$ – площадь поверхности охватывающей оболочки области X.

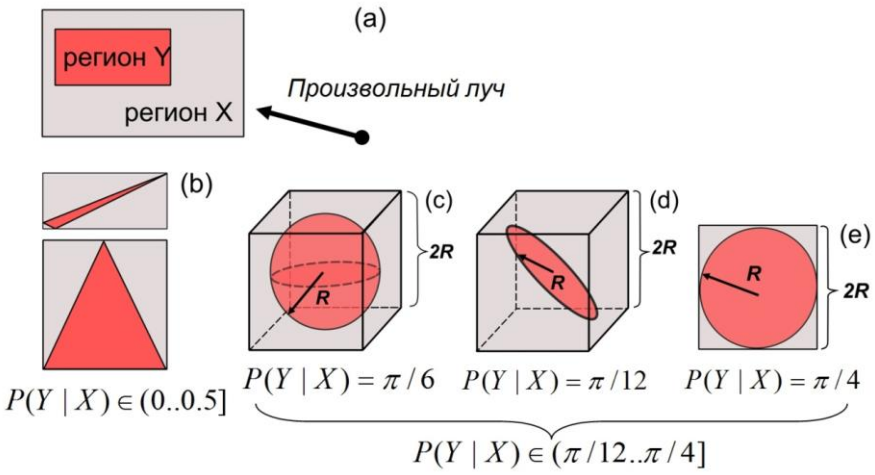


Рис. 3.1: *Геометрическая вероятность, что произвольный луч, пересекающий область пространства X, также пересекает область Y, равна P(Y | X) = SA(Y) / SA(X).*

Когда для произвольного луча во время поиска пересечения с геометрией сцены осуществляется обход дерева BVH, организующего сцену, то скорость обхода BVH выше, если вероятность $P(Y \mid X)$ выше, где область Y соответствует геомет-

рическому примитиву, содержащемуся в листе BVH, область *X* соответствует охватывающей оболочке данного листа.

Максимальное значение вероятности *P(Y | X)* того, что луч, пересекающий AABB-оболочку треугольника, действительно пересекает сам треугольник, равно 0.5 (см. рис. 3.1b). Для треугольника произвольной ориентации и размера значение *P(Y | X)* может быть значительно ниже 0.5.

Если *Y* – сфера радиуса *R*, и *X* – охватывающий куб для сферы *Y*, то $P(Y \mid X) = 4\pi R^2 / (6 \cdot (2R)^2) = \pi / 6$ (см. рис. 3.1c). Если *Y* – произвольно ориентированный плоский диск радиуса *R*, и *X* – охватывающий куб для всех возможных ориентаций диска, то $P(Y \mid X) = 2 \cdot (\pi R^2) / (6 \cdot (2R)^2) = \pi / 12$ (см. рис. 3.1d). Для фиксированной произвольной ориентации диска охватывающая оболочка AABB имеет меньшую площадь поверхности, чем охватывающий куб для всех возможных ориентаций: $P(Y \mid X) \in (\pi / 12..\pi / 4])$ (см. рис. 3.1e).

Во многих случаях значение вероятности *P(Y | X)* выше, если область Y представляет сферу или даже свободно вращающийся диск по сравнению с вероятностью для области Y, являющейся треугольником (см. нижние границы интервалов *P(Y | X)* на рис. 3.1). С учётом этих наблюдений можно заключить, что гораздо выгоднее создавать кластеры связанных треугольников, являющихся аппроксимациями сферы или диска, а затем использовать охватывающие оболочки этих кластеров как неделимые примитивы в процессе построения BVH в каждом кадре анимации (и тем самым ускорять построение BVH). Сфера или диск может вращаться и двигаться свободно в мировых координатах, и в это же самое время значение *P(Y | X)* сохраняет сравнительно высокие значения $(\pi / 12..\pi / 4]$.

Прогноз когерентного движения связанных треугольников. Т.к. кластеры треугольников предварительно рассчитываются до процесса анимации, то кластерная эвристика должна гарантировать, что кластеры останутся неразрывными в процессе анимации. См. рис. 3.2: оба кластера аппроксимируют диск, каждый состоит из 14 треугольников, но кластер (b) имеет более плотную связность (13 различных вершин), чем кластер (a) (16 различных вершин). Нет гарантии, что кластер (a) в

процессе анимации сохранит форму, аппроксимирующую диск: части кластера (a) могут свободно вращаться, нарушая форму, аппроксимирующую диск. Кластерная эвристика должна повышать вероятность генерации кластеров типа (b), которые имеют более плотную связность треугольников (и меньшее значение отношения *КоличествоВершин / КоличествоТреугольников*), которые имеют большую вероятность сохранить аппроксимирующую диск форму в процессе свободного движения и вращения.

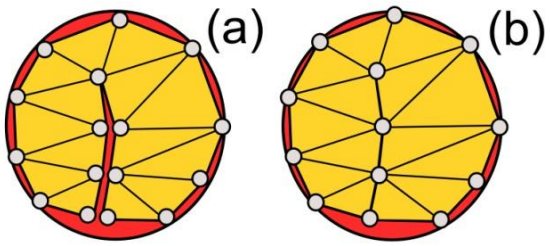


Рис. 3.2: *Кластер* ***b*** *имеет более плотную связность треугольников (и меньшее значение отношения КоличествоВершин / КоличествоТреугольников), чем кластер* ***a****.*

Вывод кластерной эвристики. Любой собираемый кластер треугольников должен иметь форму, похожую на сферу или диск. Пусть есть *k* связанных треугольников и охватывающая сфера *S(k)* радиуса *R(k)*, которая плотно охватывает все треугольники кластера (см. рис. 3.3).

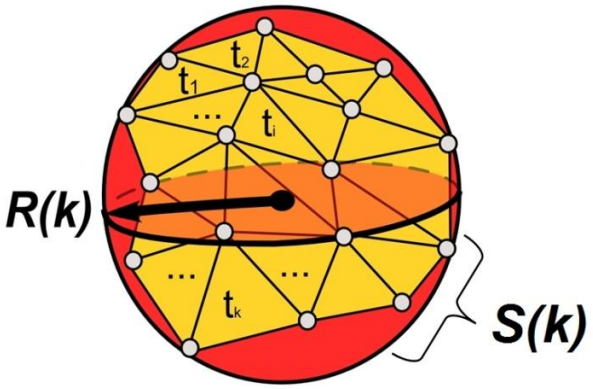


Рис. 3.3: *Кластер треугольников должен иметь форму, похожую на сферу или диск.*

Площадь любого круга внутри сферы *S(k)*:

$$AreaC(k) = \pi \cdot R(k)^2 \tag{3.1}$$

Плоскость *P* с нормалью N_{AVG} единичной длины проходит через центр *S(k)*:

$$N_{AVG}(k) = Normalized(\sum_{i=1}^{k} n_i) \tag{3.2}$$

где n_i – нормаль единичной длины для *i*-го треугольника. Если все нормали в сумме (3.2) аннулируют друг друга, то $N_{AVG}(k) = NaN$ (где $NaN=0/0$) (например, в случае, когда треугольники кластера представляют сферу или куб). В этом случае следует установить любое значение для $N_{AVG}(k)$ (например, наиболее весомое значение среди n_i в зависимости от площади треугольника).

Сумма площадей *k* спроецированных треугольников на плоскость *P:*

$$AreaP(k) = \sum_{i=1}^{k} TriArea_i \cdot abs(dot(N_{AVG}(k), n_i)) \qquad (3.3)$$

где $TriArea_i$ – площадь *i*-го треугольника, *dot(a, b)* – скалярное произведение.

Кластер, состоящий из *k* треугольников, аппроксимируют сферу или диск лучше, если следующая функция имеет более высокое значение:

$$reg(k) = \frac{AreaP(k)}{AreaC(k)} \qquad (3.4)$$

Для локально гладких и плоских (для всех i $dot(N_{AVG}(k),n_i) > 0$) поверхностей $reg(k) \in [0..1)$. Для кривых поверхностей значение *reg(k)* может быть выше 1 (например, для сферы). Замечание: функция *reg(k)* позволяет оценивать аппроксимацию сферы или диска для набора треугольников, т.к. значение *AreaC(k)* вычисляется для охватывающей сферы *S(k)*, плотно облегающей *k* треугольников.

Ограничивающие компоненты для кластерной эвристики. Для контроля связности кластера из *k* соединённых треугольников вводится компонента:

$$regBound(k) = \frac{CountDistinctVertices(k)}{MaxCount} \qquad (3.5)$$

где ***MaxCount*** – приближённое максимальное число вершин в кластере, которое является параметром алгоритма кластеризации. Значение функции *regBound(k)* линейно зависит от числа неповторяющихся индексов вершин внутри кластера *CountDistinctVertices(k)*. *regBound(k)* принимает меньшее значение для кластера с более плотной связностью (с меньшим значением отношения *КоличествоВершин / КоличествоТреугольников*). Ограничивающая компонента используется для составления следующей эвристической функции:

$$Acc\,\mathrm{Pr}e(k) = reg(k) - regBound(k) \qquad (3.6)$$

Значение функции (3.6) имеет большее значение для кластеров, близко аппроксимирующих диск/сферу и имеющих более плотную связность.

Проблема большой протяжённой оболочки кластера не учитывается в эвристике (3.6). Такая проблема может существовать для объектов, составленных из треугольников неоднородных размеров (например, для объектов архитектурного моделирования). В случае, изображённом на рис. 3.4a, оба кластера имеют одинаковое количество треугольников, но один кластер составлен из больших треугольников, а другой из маленьких.

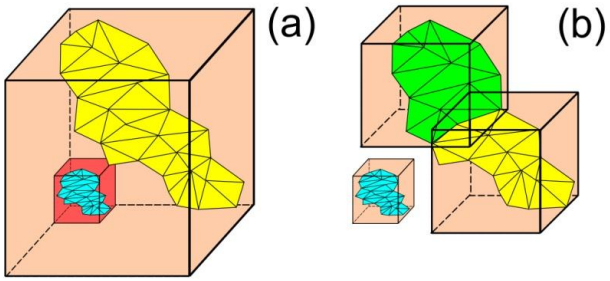


Рис. 3.4: *Проблема большого геометрического размера для кластера: (a) AABB оболочка большего кластера перекрывает AABB оболочку меньшего кластера, что порождает большее число тестов пересечения луча и объектов во время обхода BVH; (b) если кластерная эвристика отсекает составление протяжённых кластеров, тогда области перекрытия между соответствующими оболочками уменьшаются, и также уменьшается количество тестов пересечения луча и оболочек в процессе обхода BVH.*

В таком случае при движении кластеров могут образоваться области больших перекрытий между их оболочками. Подобные перекрытия порождают большое число тестов пересечений луча и оболочек во время обхода BVH, которые можно было бы избежать, если бы кластерная эвристика отсекала создание протяжённых больших кластеров (см. рис. 3.4b). Для решения этой проблемы вводится ‘штрафная’ компонента эвристики:

$$sizeBound(k) = \max(1, \frac{SA(S(k))}{MaxSize \cdot AvgSA}) \qquad (3.7)$$

где *AvgSA* – площадь поверхности охватывающей сферы среднего треугольника объекта, ***MaxSize*** – приближённый максимальный желаемый геометрический размер кластера (*MaxSize* – предустановленный параметр). Значение функции *sizeBound(k)* больше 1 для кластера из *k* треугольников, оболочка которого имеет не-

желательно большой геометрический размер. Компонента *sizeBound(k)* комбинируется с эвристикой (3.6) следующим способом:

$$Acc(k) = reg(k) - sizeBound(k) \cdot regBound(k) \qquad (3.8)$$

Значение функции (3.8) понижается по сравнению со значением функции (3.6), если площадь поверхности охватывающей оболочки кластера больше допустимой площади *MaxSize·AvgSA*. Если значение *Acc(k) > 0*, тогда набор *k* соединённых треугольников считается приемлемым для формирования кластера.

3.2. Генерация множества кластеров

Большее значение *Acc(k)* (формула 3.8) соответствует ‘лучшему’ кластеру выводов подглавы 3.1. Выражение *Acc(k)* представлено в виде суммы положительного и негативного вклада свойств кластера, оценённых выражениями (3.4), (3.5) и (3.7), в общую оценку приемлемости создания кластера. В выражении для *Acc(k)* негативные компоненты скомбинированы в умножении, что позволяет составлять сбалансированные кластеры и по геометрическому размеру и по количеству вершин треугольников внутри кластера (см. рис. 3.7). Функция *Acc(k)* применима для двух алгоритмов генерации кластеров, которые работают в порядке снизу-вверх: итеративное **выращивание** и **склейка** кластеров.

Итеративное выращивание кластеров. На каждом шаге инициализируется пустой новый кластер, и случайный треугольник добавляется в этот кластер. Затем для кластера из *k-1* треугольников вычисляется значение *Acc(k)* для всех треугольников, которые могут быть добавлены в этот кластер (возможный добавочный треугольник должен быть соединён с кластером посредством общих индексов вершин, см. рис. 3.5). Треугольник, соответствующий максимальному значению *Acc(k)*, добавляется в кластер. Процесс итеративного выращивания кластера останавливается, если *Acc(k) < 0* для всех возможных добавлений.

Итеративная склейка кластеров (детали в статье [38]). Этот подход использует двойственный граф, построенный для геометрического модели объекта (см. рис. 3.6а). Двойственный граф определяется отображением каждого треугольника объекта в узел двойственного графа, и соединением 2-ух двойственных узлов этого

графа ребром, если соответствующие треугольники соединены посредством общих индексов вершин. Удаление ребра в двойственном графе склеивает 2 узла в 1 (см. рис. 3.6b). Эта операция соответствует склейке 2-ух кластеров треугольников в один кластер.

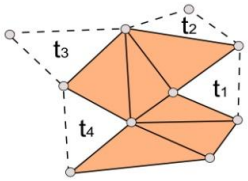


Рис. 3.5: *Acc(k) вычисляется для каждого примыкающего к кластеру треугольника: t_1, t_2, t_3, t_4. Треугольник, соответствующий максимальному значению Acc(k), добавляется в кластер. t_1 и t_4 могут быть добавлены без добавления новых вершин в кластер.*

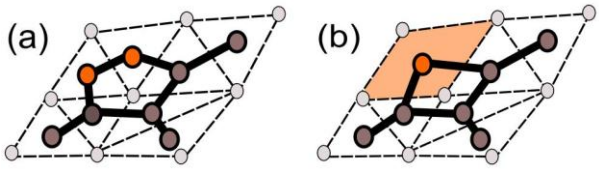


Рис. 3.6: *Удаление двойственного ребра: 2 треугольника (a), соответствующие узлам в двойственном графе, склеиваются в один кластер (b).*

Каждому двойственному ребру соответствует значение *Acc(k)*. Все двойственные рёбра отсортированы в очереди. Очередь реализована с использованием хэш-таблицы, клетки которой отображены на интервал [-2..2] возможных значений функции *Acc(k)*, изменяющейся в этом диапазоне для всех опробованных объектов. Кластерный алгоритм последовательно удаляет двойственные рёбра с максимальным значением *Acc(k)*. После того как 2 двойственных узла склеены в один узел значение *Acc* обновляется для всех двойственных ребёр, соединённых с новым узлом. Этот итеративный процесс останавливается, если текущее максимальное значение *Acc* в очереди меньше 0. После остановки процесса склейки все имеющиеся двойственные узлы, которые могут содержать по нескольку треугольников, представляют множество кластеров, соответствующее параметрам *MaxSize*, *MaxCount*.

Алгоритм выращивания кластеров создаёт кластеры один за другим. Этот алгоритм может создать подмножество отличных кластеров, не оставив “сырья” для других кластеров (см. рис. 3.7, стрелка на изображении посередине).

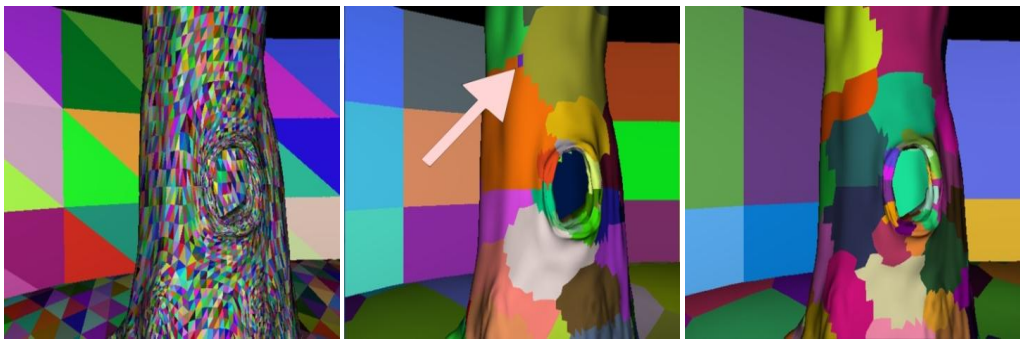


Рис. 3.7: *Слева: треугольники объекта, посередине: кластеры получены итеративным выращиванием, справа: кластеры получены итеративной склейкой.*

Алгоритм склейки кластеров итеративно генерирует все кластеры одновременно. Этот алгоритм учитывает глобальную конфигурацию значений *Acc* и склеивает “лучшую” пару суб-кластеров на каждом шаге. Парное склеивание может создать зубчатые границы между соседними кластерами (см. рис. 3.7, правая картинка). Зубчатые границы могут быть выровнены при помощи вычисления кратчайшего пути по рёбрам треугольников объекта [96].

3.3. Построение ускоряющей структуры на основе множества кластеров

После создания множества кластеров треугольников каждый кластер содержит список индексов треугольников и список неповторяющихся индексов вершин, входящих в собранный кластер. Вершины используются для вычисления охватывающей оболочки для каждого кластера для каждого кадра анимации. Для построения дерева BVH используется любой алгоритм построения (например, предложенный в статье [113]). Но вместо множества треугольников алгоритм построения организует множество оболочек кластеров в иерархию (см. рис. 3.8).

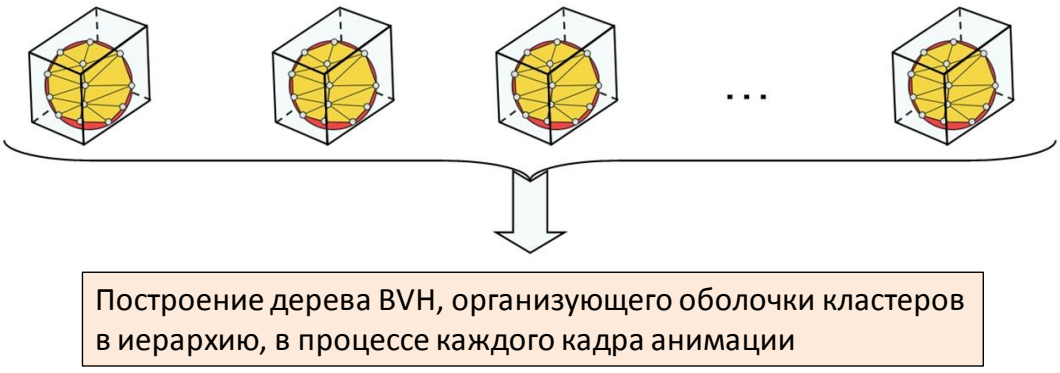


Рис. 3.8: *Построение BVH в каждом кадре анимации. BVH строится на основе охватывающих коробок AABB, вычисленных для каждого кластера треугольников.*

В алгоритме построения BVH множество оболочек кластеров рекурсивно разделяется на 2 непересекающихся подмножества. Процесс разделения заканчивается, если множество кластеров, содержит сумму треугольников, меньшую, чем предустановленное значение L (например, $L = 32$). После остановки создаётся лист дерева со ссылкой на блок индексов кластеров.

3.4. Сравнение с конкурирующей кластерной эвристикой

Разработанная в рамках данной работы кластерная эвристика отличается от эвристики, предложенной в статье [38], по формулировке и цели применения. В [38] итеративно склеиваются кластеры с наименьшим ассоциированным с кластером значением ошибки. Функция ошибки: $Error(k) = E_{fit} + a_1 E_{dir} + a_2 E_{shape}$. Меньшее значение компоненты E_{fit} соответствует “более плоскому” кластеру (т.е. ровной поверхности). Меньшее значение компоненты E_{dir} соответствует “более сонаправленным” нормалям треугольников внутри кластера. Меньшее значение компоненты E_{shape} соответствует “более регулярной” форме границы кластера (“регулярная” граница соответствует границе диска). a_1 и a_2 – установленные пользователем веса для компонент выражения.

Эвристика $Acc(k)$, разработанная в рамках настоящей работы, спроектирована с учётом необходимости обеспечивать высокую скорость поиска пересечений лучей и геометрии в динамических сценах. Строятся кластеры, аппроксимирующие сферы или диски, для построения эффективного дерева BVH и одновременной поддержки свободного движения и вращения кластеров (была учтена модель геометрической вероятности). Эвристика $Acc(k)$ при оценке конфигурации кластеров отдаёт предпочтение кластерам плотной связности, которые имеют меньшую вероятность быть разорванными при свободном движении. Эти свойства $Acc(k)$ позволяют предварительно рассчитывать множество кластеров, которые используются как строительные блоки для построения высококачественных деревьев BVH в каждом кадре анимации (используется предположение о том, что группы связанных треугольников остаются связанными во время анимации).

3.5. Результаты

Установки реализации алгоритма. Алгоритм генерации кластеров и построения дерева BVH реализован на языке C++ для исполнения на CPU с использованием многопоточности. Измерения показателей алгоритма производились на ноутбуке ASUS G1S с процессором Intel Core 2 Duo T5550 (2 ядра на частоте 1.83GHz, L2 cache 2 MB, DDR2 RAM 2 GB, 32-bit Windows Vista).

В качестве базового алгоритма построения BVH для организации кластеров в иерархию используется алгоритм [38]. Для проверки скорости поиска пересечений лучей, обеспечиваемой построенным деревом BVH, используется алгоритм пакетной трассировки (включая пакеты, SIMD-лучи, отсечение по пирамидам, SIMD-отсечение вершин [93]), для трассировки отражённых лучей дополнительно используется программная фильтрация SIMD-лучей (см. [80]). Алгоритм пакетной трассировки лучей использует многопоточность, первоначально разбивает изображение на квадратные блоки пикселей размера 32^2, которые соответствуют пакетам из 1024 лучей, организованных в 256 SIMD-лучей (т.е. используется одна SIMD инструкция для работы с данными, соответствующими 4 лучам одновременно). Изображения в данной главе сгенерированы при разрешении 1024^2.

Несмотря на то, что число кластеров, которые используются в процессе построения BVH, может быть различным (в зависимости от параметров кластерной эвристики), рекурсивный процесс разделения множества примитивов и генерации новых узлов дерева BVH останавливается для создания листа, если в узле содержится несколько кластеров с общей суммой треугольников не более 32. Для динамических сцен кластеры треугольников строятся 1 раз с учётом координат вершин 1-го кадра анимации. В процессе анимации используются одно и тоже множество кластеров треугольников для построения BVH, но при этом обновляются их оболочки. Предполагается, что группы связанных треугольников остаются связанными во время анимации.

Визуализация кластеров. Полученные с помощью использования эвристической функции *Acc(k)* кластеры изображены на рис. 3.9. Для генерации множества кластеров использовался алгоритм итеративной склейки кластеров. Для сцены Thai

statue для генерации кластеров используется только итеративное выращивание кластеров, т.к. не хватает виртуального адресного пространства 32-битной ОС для размещения двойственного графа в памяти для объекта, состоящего из 10 млн. треугольников. Для объектов Cloth и Thai создаются кластеры желаемой дискообразной формы, т.к. все треугольники этих объектов имеют однородные размеры. Для объекта Conference создаются кластеры различной формы, чаще прямоугольной, т.к. этот объект состоит из треугольников неоднородных размеров, чаще длинных и тонких.

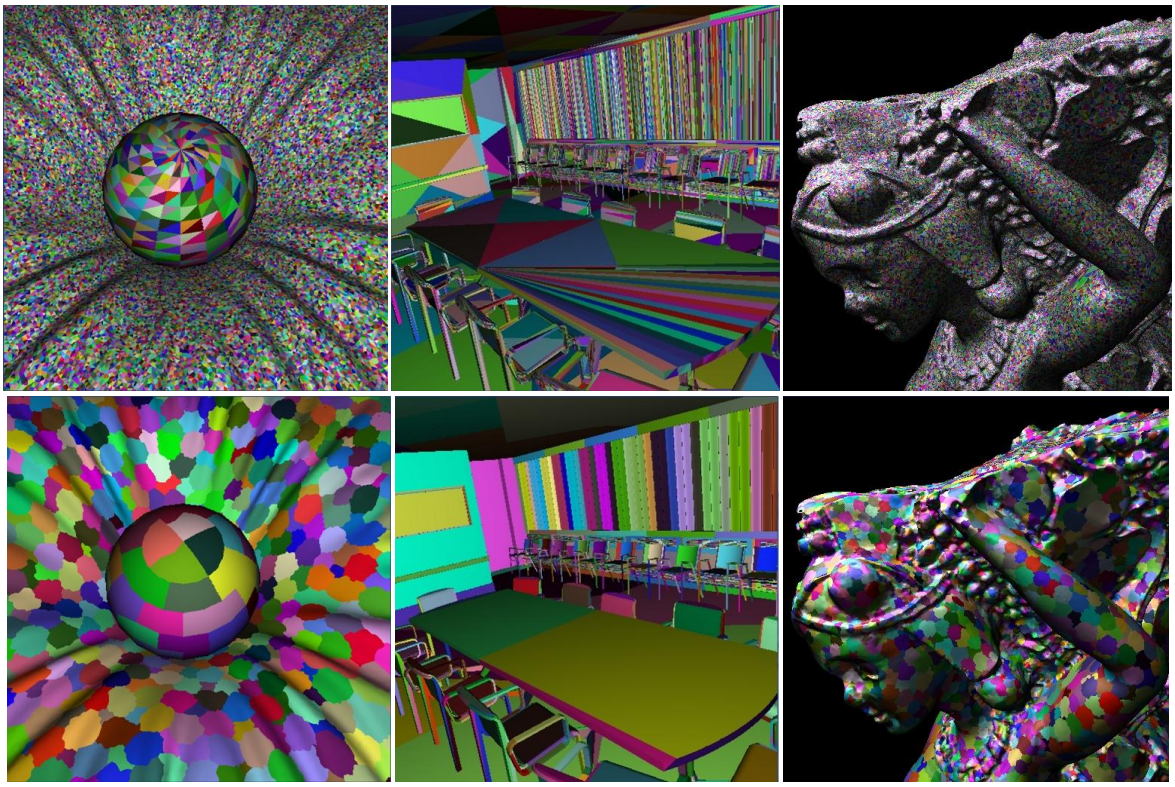


Рис. 3.9: *Слева: динамический объект Cloth Simulation (92K треугольников), посередине: сцена Conference room (282K треугольников), справа: сцена Thai statue (10 млн. треугольников). Верхний ряд: треугольники раскрашены разным цветом. Нижний ряд: кластеры треугольников раскрашены разным цветом, созданы с помощью функции Acc(k) с параметрами MaxSize = MaxCount = 100.*

Используемые сцены. Для тестирования алгоритма генерации кластеров, используется 6 объектов (см. рис. 3.10), представляющих разные классы объектов моделирования: деформируемый объект ткани, симуляция взрыва, симуляция столкновений [105], структурированное движение, модель с множеством длинных и тонких треугольников, большая модель с однородными треугольниками.

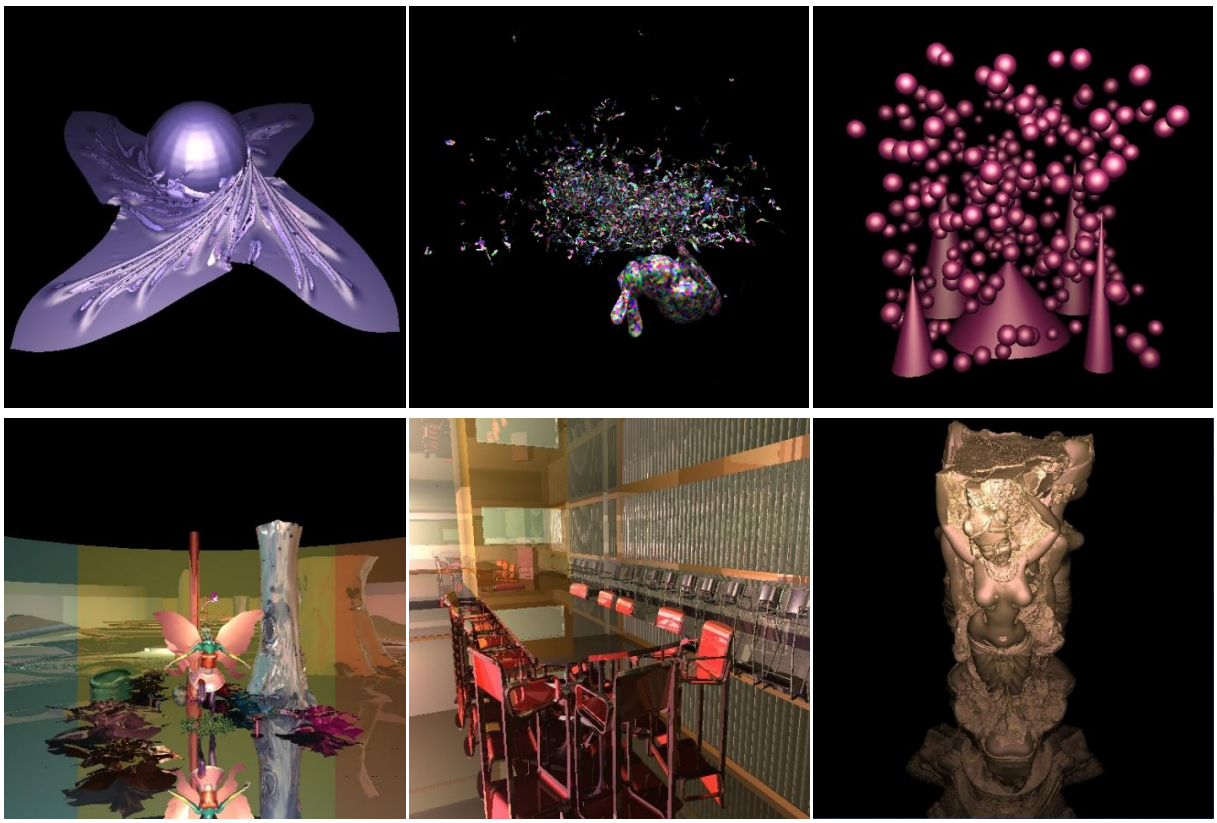


Рис. 3.10: *Используемые сцены (слева-направо, сверху-вниз): Cloth Simulation (92K треугольников), Exploding Dragon (252K), Colliding N body (146K), Fairy Forest (174K), Conference Room (282K), Thai Statue (10 млн.). Первые 4 сцены являются динамическими.*

Исследование параметров кластерной эвристики *Acc(k)*. Измерения различных показателей алгоритма построения BVH на основе кластеров при использовании различных параметров для эвристики *Acc(k)* представлены на рис. 3.11 для сцен Exploding Dragon и Fairy Forest, и в таблице 3.1 более подробно для остальных сцен. В таблице 3.1 и рис. 3.11 параметры алгоритма кластеризации *MaxCount = MaxSize = 0* соответствуют результатам работы алгоритма построения BVH без использования предварительно рассчитанных кластеров треугольников. Для всех рядов *MaxCount > 0*, *MaxSize > 0* построение BVH производится значительно быстрее. Время трассировки лучей не ухудшается по сравнению с использованием кластеров *MaxSize = MaxCount = 0*, пока не используются кластеры большого размера. Наибольшее преимущество от использования алгоритма построения BVH на основе кластеров проявляется для больших сцен, таких как Thai Statue (10 млн. треугольников).

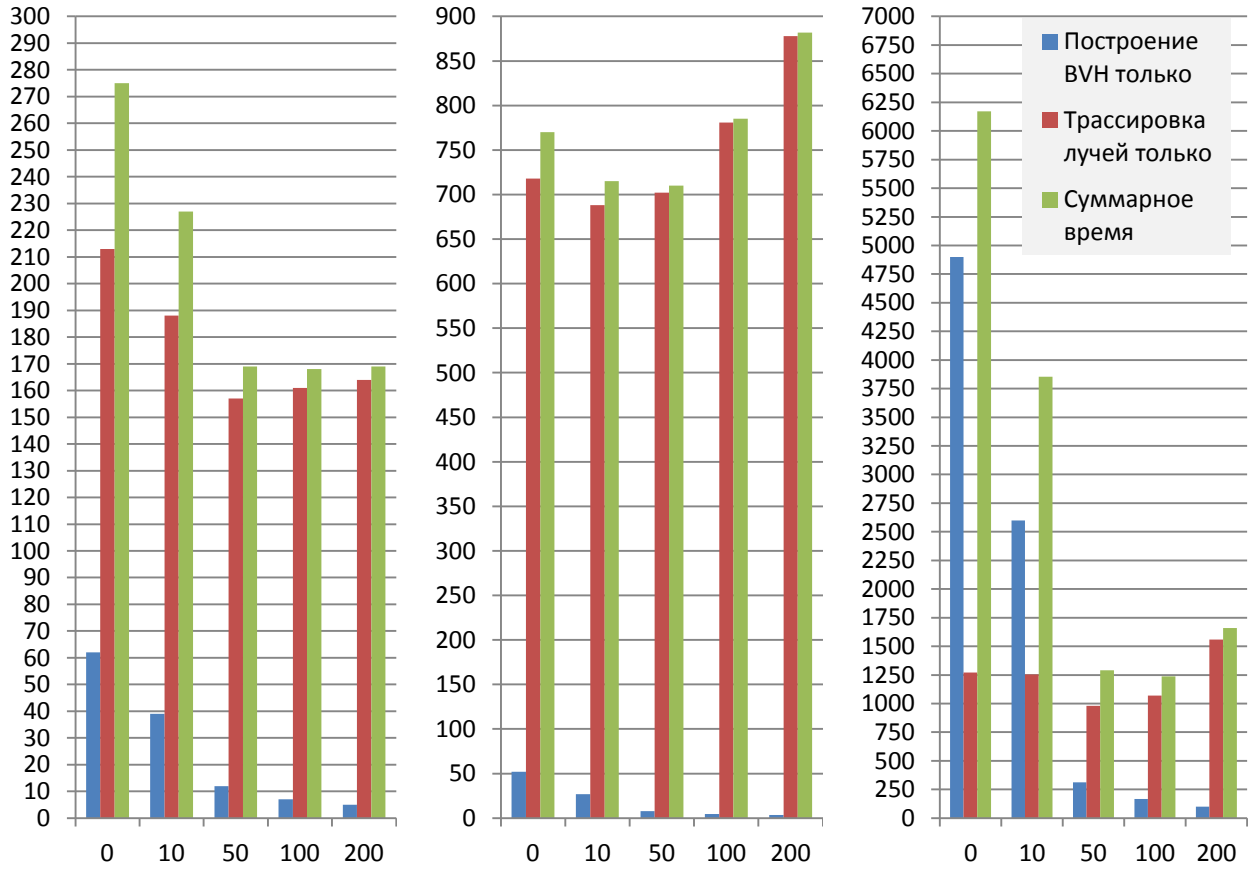


Рис. 3.11: *Зависимость времени (в мс) трассировки отражённых лучей с 2 уровнями переотражений от используемых параметров кластеризации. По горизонтали изменяются параметры MaxSize = MaxCount = X кластерной эвристики Acc(k). По вертикали измеренное время стадии построения BVH и стадии трассировки лучей. График слева: для Exploding Dragon (252K треугольников), посередине: Fairy Forest (174K треугольников), справа: Thai Statue (10 млн. треугольников).*

Параметрами кластерной эвристики *Acc(k)*, обеспечивающими наименьшее суммарное время построения BVH и трассировки лучей, являются *MaxCount = MaxSize = 50* (далее обозначается $Acc(k)_{50,50}$) для всех тестовых сцен: 1) построение BVH быстрее, чем построение без предварительной кластеризации; 2) время трассировки лучей не замедляется по сравнению со временем, соответствующим *MaxCount = MaxSize = 0* (далее обозначается $Acc(k)_{0,0}$). Время трассировки лучей может увеличиваться для дерева BVH, построенного для больших кластеров с параметрами *MaxCount = MaxSize >= 100*, т.к. большие кластеры могут иметь менее плотную связность треугольников, а также иметь большую вероятность образования разрывов внутри кластера во время анимации взрывной симуляции.

	Cloth	Dragon	N-body	Forest	Conf.	Thai
Кол-во треугольников	92K	252K	146K	174K	282K	10M
Кол-во вершин	46.5K	192K	73.9K	97K	166K	5M
MaxCount = 0, MaxSize = 0						
Время кластеризации, мс	0	0	0	0	0	0
Кол-во кластеров	92K	252K	146K	174K	282K	10M
Время построения BVH, мс	23	62.8	37.2	52	82.4	4.9K
Время трассировки только первичных лучей, мс	44.5	32.1	50.2	68	109	250
Время трассировки только 2 уровневых отражений, мс	163	213	533	718	1940	1270
MaxCount = 10, MaxSize = 10						
Время кластеризации, мс	1.9K	2.9K	7.4K	1.2K	3K	165K
Кол-во кластеров	49K	139K	69K	83K	148K	5.1M
Время построения BVH, мс	14.4	39.2	23.6	27.2	43	2.6K
Время трассировки только первичных лучей, мс	42.1	28.2	44.4	63.6	108	240
Время трассировки только 2 уровневых отражений, мс	152	188	448	688	1880	1255
MaxCount = 50, MaxSize = 50						
Время кластеризации, мс	1.5K	1.9K	8.4K	2.8K	5.5K	143K
Кол-во кластеров	5.2K	24.6K	8.2K	17.9K	32K	469K
Время построения BVH, мс	6.7	12.1	5.2	7.8	12.6	311
Время трассировки только первичных лучей, мс	36.6	24.4	42	57.5	110	174
Время трассировки только 2 уровневых отражений, мс	154	157	409	702	1990	980
MaxCount = 100, MaxSize = 100						
Время кластеризации, мс	3.7K	2.3K	9.3K	5.8K	10K	139K
Кол-во кластеров	3.7K	16.4K	4.3K	10.7K	20.4K	194K
Время построения BVH, мс	5.6	7.1	4.1	4.5	9.2	167
Время трассировки только первичных лучей, мс	34.9	25.7	41.9	57.5	109	166
Время трассировки только 2 уровневых отражений, мс	167	161	452	781	2326	1070
MaxCount = 200, MaxSize = 200						
Время кластеризации, мс	11K	3.7K	14K	14K	25K	146K
Кол-во кластеров	2.4K	12.3K	2.5K	6.9K	11.3K	88K
Время построения BVH, мс	4.2	5.1	1.8	3.5	5.9	106
Время трассировки только первичных лучей, мс	37	28.6	41.6	59	104	177
Время трассировки только 2 уровневых отражений, мс	201	164	569	878	2755	1560

Таблица 4.1: *Статистика гибридного алгоритма построения BVH при использовании эвристики Acc(k) с разными параметрами. В таблице приведено время трассировки лучей 6 изображений на рис. 4.2. При трассировке 2-уровневых отражений каждая поверхность всех 6 моделей является отражающей (это стресс тест качества BVH при трассировке многих некогерентных лучей).*

Для некоторых значений параметров *MaxCount > 0*, *MaxSize > 0* время трассировки лучей может быть меньше, чем для $Acc(k)_{0,0}$, благодаря предварительной загрузке всех координат вершин треугольников кластера из глобального массива в кэш вершин во время произведения теста пересечений луча и геометрии кластера. Кроме того, доля кэш попаданий CPU в тесте на пересечение треугольника и пакета лучей повышается, т.к. треугольники ссылаются на небольшой кэш вершин. Подобная оптимизация является возможной при заранее известной структуре связности треугольников внутри кластера (т.е. для предварительно построенных кластеров). Если кластеры не строить заранее, то при использовании обычного метода построения BVH (без кластеров) генерация подобных структур связности занимала бы много времени.

Качество BVH для динамических сцен. На рис. 3.12 представлено сравнение скорости трассировки лучей с 2 уровнями переотражений, обеспечиваемой кластерной эвристикой $Acc(k)_{50,50}$ со скоростью трассировки лучей без использования кластерной эвристики для каждого кадра анимации. Не смотря на то, что кластеры, построенные с помощью $Acc(k)_{50,50}$, генерируются 1 раз с учётом координат вершин, соответствующих первому кадру анимации (с учётом связности треугольников), скорость трассировки лучей остаётся высокой на протяжении всех кадров анимации (выше по сравнению с вариантом без использования кластеров).

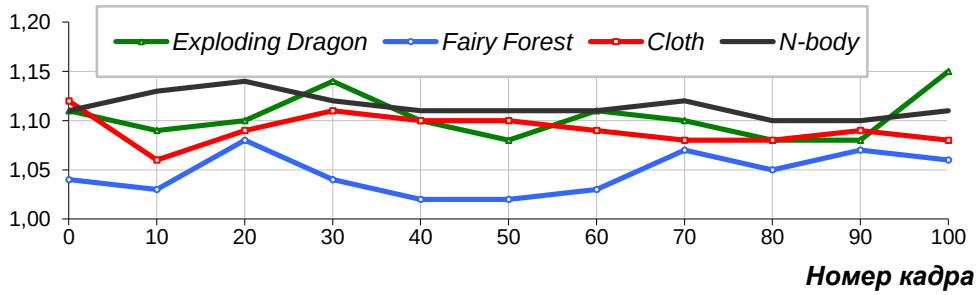


Рис. 3.12: *Графики f = Скорость($Acc(k)_{50,50}$) / Скорость($Acc(k)_{0,0}$) отношения скорости трассировки лучей с 2 уровнями переотражений, обеспечиваемой деревом BVH, построенным на основе $Acc(k)_{50,50}$ (MaxCount=MaxSize=50) к скорости трассировки лучей, обеспечиваемой деревом BVH, построенным на основе $Acc(k)_{0,0}$. Значение f > 1 означает более высокую скорость трассировки лучей, обеспечиваемой $Acc(k)_{50,50}$.*

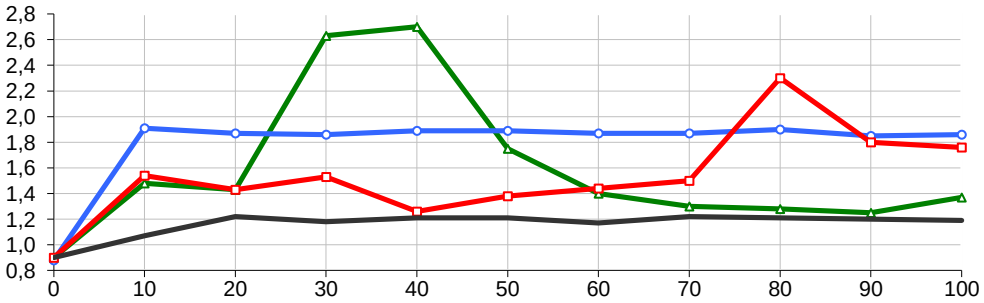


Рис. 3.13: *Графики f = Скорость(Acc(k)$_{50,50}$) / Скорость(Naïve(k)) отношения скорости трассировки лучей с 2 уровнями переотражений, обеспечиваемой деревом BVH, построенным на основе Acc(k)$_{50,50}$ к скорости трассировки лучей, обеспечиваемой деревом BVH, построенным на основе кластеров с использованием эвристики Naïve(k).*

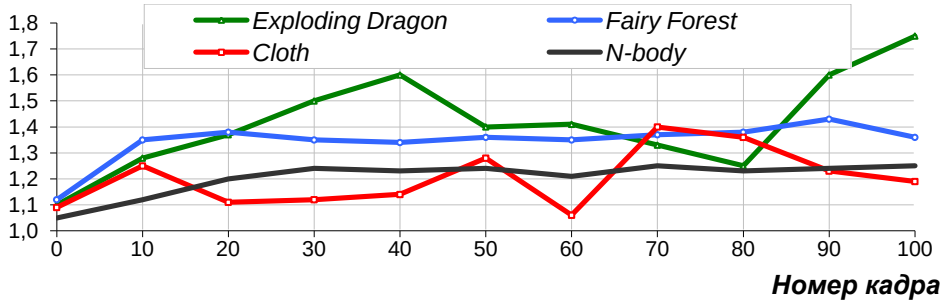


Рис. 3.14: *Графики f = Скорость(Acc(k)$_{50,50}$) / Скорость(Error(k)) отношения скорости трассировки лучей с 2 уровнями переотражений, обеспечиваемой деревом BVH, построенным на основе Acc(k)$_{50,50}$ к скорости трассировки лучей, обеспечиваемой деревом BVH, построенным на основе кластеров с использованием эвристики Error(k), описанной в статье [38].*

Сравнение с аналогами. На графиках рис. 3.13 для 4 анимированных сцен представлено сравнение кластерной эвристики *Acc(k)$_{50,50}$* с SAH эвристикой применительно к построению дерева BVH, используемого для трассировки лучей. Функция эвристики для генерации SAH-кластеров определяется для *k* треугольников следующим образом:

$$Naive(k) = \sum_{i=1}^{k} SA(Треугольник_i) / SA(AABB\ оболочка\ для\ k\ треугольников)$$

где *SA()* – площадь поверхности. Большее значение *Naïve(k)* соответствует кластеру из *k* треугольников, которые плотнее упакованы в области пространства.

На графиках рис. 3.14 для тех же сцен представлено сравнение кластерной эвристики *Acc(k)$_{50,50}$* с кластерной эвристикой *Error(k)*, описанной в работе [38], применяющейся для построения дерева BVH, используемого для трассировки лучей.

Функция *Error(k)* используется с установленными параметрами $a_1 = 0.2$ и $a_2 = 2$ (задаётся требование генерации кластеров с границей регулярной формы, т.е. кластеры должны по возможности иметь дискообразную формую).

Эвристические функции $Acc(k)_{50,50}$, *Naïve(k)* и *Error(k)* реализованы для генерации кластеров в рамках настоящей работы для корректного сравнения. Кластеры для этих эвристик создаются с помощью итеративной склейки с учётом координат вершин первого кадра анимации. Процесс итеративной склейки для *Naïve(k)* и *Error(k)* останавливается, если число уже составленных кластеров равно числу кластеров, составленных с помощью $Acc(k)_{50,50}$. Т.е. для каждой отдельной сцены процесс построения BVH использует одинаковое число кластеров треугольников, составленных на основе $Acc(k)_{50,50}$, *Naïve(k)* и *Error(k)*. Графики на рис. 3.13 и 3.14 демонстрируют преимущество эвристики $Acc(k)_{50,50}$ над другими эвристиками. Ускоряющие структуры BVH, построенные на основе кластеров $Acc(k)_{50,50}$, обеспечивают более высокую скорость трассировки лучей для различных динамических сцен.

Кластеры, созданные с помощью функции *Naïve(k),* обеспечивают высокую скорость трассировки лучей для первого кадра анимации (т.к. построены на основе SAH эвристики). Однако в процессе анимации такие кластеры могут значительно изменить свою форму, и оказаться неэффективными для обеспечения высокой скорости трассировки лучей, т.к. эвристика *Naïve(k)* не требует высокой плотности связности треугольников в кластере, и не отвечает остальным требованиям, наложенным на эвристическую функцию *Acc(k)*. Эвристическая функция *Error(k)* также не учитывает ни высокой плотности связности треугольников, ни умеренного геометрического размера кластера. Но эвристика *Error(k)* (см. результаты на рис. 3.14) обеспечивает более высокую скорость трассировки лучей, чем эвристика *Naïve(k)* для анимированных сцен, т.к. позволяет генерировать кластеры с границами регулярной формы (т.е. более компактные для всех возможных ориентаций кластера в процессе возможного вращения).

3.6. Заключение

Недостатки и ограничения. Для ускорения процесса выполнения алгоритма построения BVH используется предположение, что группы связанных треугольников остаются связанными в процессе анимации. Подобный подход не имеет преимуществ для объектов, не имеющих связанных треугольников (где *Кол-воВершин / Кол-воТреугольников = 3*).

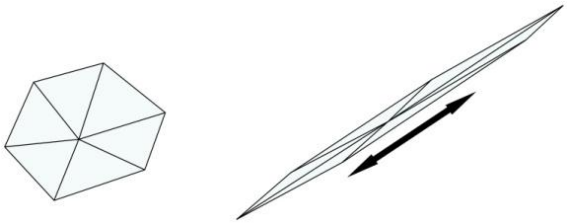


Рис. 3.15: *Если большинство созданных кластеров растягиваются в процессе анимации, то обеспечиваемая такими кластерами скорость трассировки лучей может снизиться.*

Если в процессе анимации большинство созданных кластеров подвергаются сильному растягиванию в произвольных направлениях, и если кластеры трансформируются в длинные и тонкие по форме кластеры (см. рис. 3.15), то тогда скорость трассировки лучей может значительно снизиться, если не генерировать периодически новое множество кластеров с новой конфигурацией. Такой (редкий на практике) случай представляет собой ограничение разработанного гибридного алгоритма: большинство кластеров регулярной формы должны сохранять регулярную форму в процессе анимации. Это ограничение по большей части наследуется от структуры дерева BVH, созданного с помощью иерархии оболочек на основе плоскостей, параллельных базисным координатным плоскостям.

Преимущество. Алгоритм ускорения построения дерева BVH на основе предварительно сгенерированных кластеров треугольников позволяет на порядок ускорить процесс построения BVH, основанный на любом алгоритме построения. Разработанный алгоритм применим для систем визуализации анимированных сцен (деформируемые объекты, анимации взрывов, столкновения, разрушения, структурированное движение), где процедура построения ускоряющей структуры является критической по времени частью задачи (наибольший общий прирост скорости визуализации проявляется для более детализированных объектов).

Глава 4

Алгоритм построения ускоряющей структуры BVH с помощью вспомогательной сетки

В данной главе описан простой в реализации параллельный алгоритм быстрого построения ускоряющей структуры BVH с использованием графического процессора и технологии CUDA [39][79] за линейное время (впервые опубликован в статье [4]). Ускорение процесса построения BVH достигается за счёт использования вспомогательной равномерной 3D сетки. Эта сетка строится внутри охватывающей оболочки сцены, а примитивы (в данном случае, треугольники) распределяются по ячейкам сетки один раз во время построения BVH. Благодаря использованию данной сетки производство каждого узла BVH осуществляется с помощью приблизительной SAH функции (1.3), для расчётов используется ограниченное сверху число вычислительных операций и чтения/записи в память.

Также эффективно решается проблема, связанная с «тонкими и длинными» треугольниками, для которых охватывающая оболочка, выровненная по осям координат (сокр. AABB), может повлечь снижение скорости трассировки лучей. Подобные треугольники разрезаются на несколько частей, где каждый частичный треугольник имеет свою охватывающую оболочку, более мелкую, чем базовая оболочка всего треугольника. Причём параллельный метод разделения треугольников на части устроен таким образом, что суммарное количество всех частичных треугольников сцены не превышает заранее выделенного объёма памяти, решая, таким образом, проблему возможного переполнения памяти.

4.1. Общая схема

По данному алгоритму производятся двоичные BVH иерархии охватывающих оболочек, выровненных по осям координат (сокр. AABB). В работах [113] и [71] предложены алгоритмы бинового построения BVH с использованием CPU и GPU соответственно. Биновый алгоритм рекурсивно распределяет список примитивов (часто используются треугольники) на различные узлы BVH. В описанных ранее биновых алгоритмах в процессе разделения множества примитивов секущей плос-

костью используется временная структура в виде равномерной сетки для уменьшения сложности выбора оптимальной секущей плоскости. Ссылки на примитивы распределяются по ячейкам сетки, после этого SAH функция рассчитывается с использованием ячеек сетки (но не с использованием массива отсортированных примитивов, который может быть большим).

В новом алгоритме, описанном в этой главе, также используются вспомогательные равномерные сетки, которые позволяют строить SAH BVH за время $O(k * n)$, где n – количество примитивов, а k – количество ячеек сетки в одном измерении. В отличие от ранних алгоритмов, в новом алгоритме сеточная структура и распределение треугольников внутри сетки осуществляется всего один раз (а не каждый раз во время генерации нового BVH). Далее создаются уровни детализации этой 3D сетки, где каждый верхний уровень имеет разрешение в 2 раза меньшее в каждом измерении, чем у сетки из предыдущего уровня. Каждая ячейка сетки содержит ссылку на список треугольников, которые содержатся в ячейке (т.е. в пространстве такие треугольники должны пересекать объём соответствующий данной ячейке). SAH функция для генерации корневого узла BVH рассчитывается на основе данных сетки самого верхнего уровня размера k^3 (в описании алгоритма и изображениях часто будет использоваться $k=8$ в качестве наглядного примера): т.е. SAH функция рассчитывается для 21 возможной плоскости разделения (все плоскости между 8 ячейками с учётом 3D), выбирается плоскость с наименьшим значением SAH функции и треугольники разделяются относительно этой плоскости (само разделение запоминается и реально осуществляется на этапе генерации листового узла BVH). Затем оптимальные плоскости разделения находятся для двух списков треугольников и т.д. Листовой узел BVH создаётся, когда очередной создаваемый узел BVH содержит не более m примитивов (во многих работах используется $m=4$). Число треугольников и охватывающая оболочка, соответствующая распределению треугольников относительно каждой тестируемой плоскости при оценке SAH функции рассчитывается с использованием 3D сетки.

4.2. Создание сетки $Grid_0$ распределения треугольников

Сетка нижнего уровня, обозначается *Grid0*, каждая ячейка сетки содержит в адрес и размер списка частичных треугольников сцены, содержащихся в ячейке. Пространственный размер сетки определяется размерами охватывающей оболочки всей сцены. Используется разрешение 1024^3 для *Grid0* (т.е. адрес каждой ячейки может быть закодирован в 30-битном ключе).

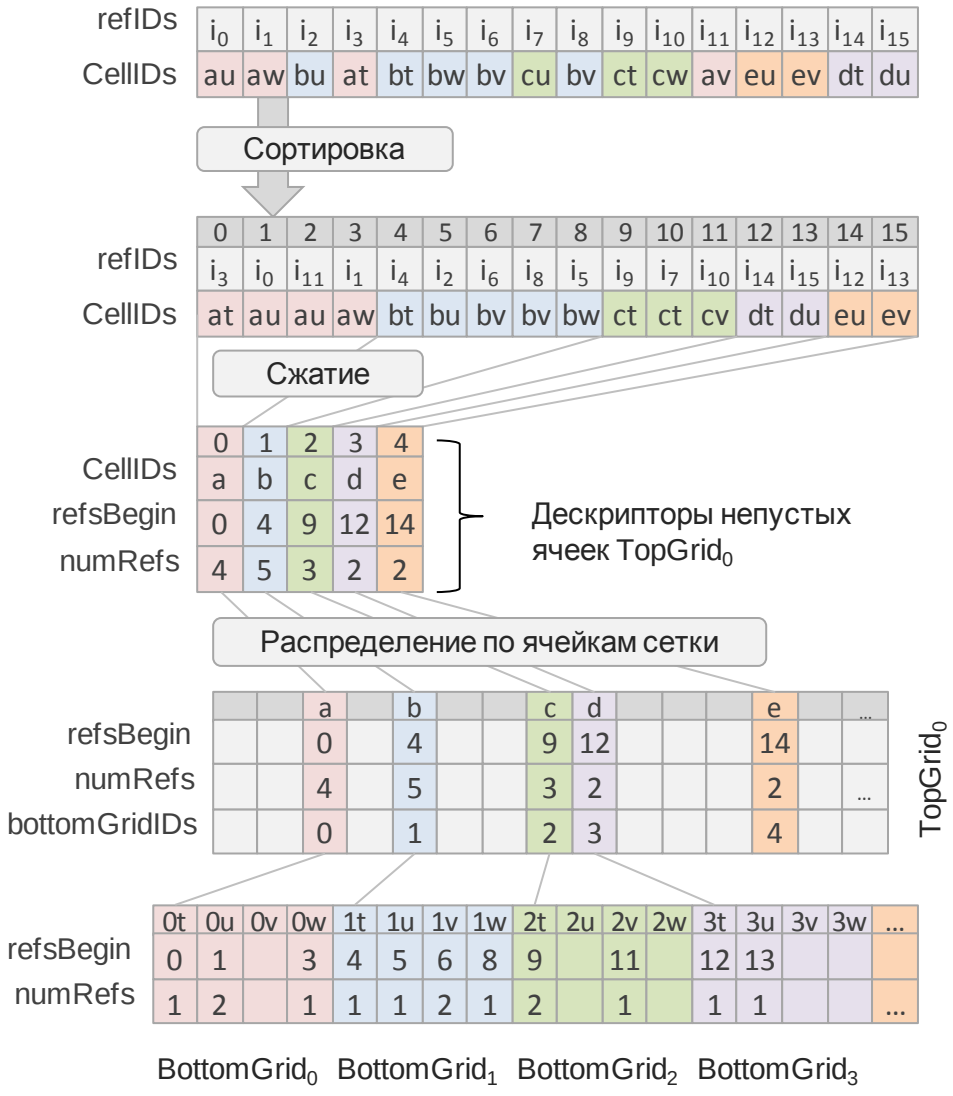


Рис. 4.1: *Процесс распределения частичных треугольников по ячейкам сетки Grid0: Индексы частичных треугольников записаны в массиве refIDs, которые сортируются по возрастанию ключей CellIDs, соответствующих каждому треугольнику. Каждая пара значений $refsBegin_i$ и $numRefs_i$ соответствует ячейке сетки (TopGrid0 или BottomGrid0), содержит адрес сегмента отсортированного массива refIDs, где в сегменте содержатся индексы треугольников, принадлежащие ячейке.*

Т.к. большинство ячеек *Grid0* является пустым, то хранение информации о списках частичных треугольников для всех 1024^3 ячеек является избыточным. Поэтому в памяти *Grid0* хранится в виде 2-уровненвой иерархии одной сетки верхнего уровня *TopGrid0* и малых сеток нижнего уровня *BottomGrid*, созданных для каждой непустой ячейки *TopGrid0*. Практически используется разрешение 128^3 для сетки *TopGrid0* и 8^3 для каждой сетки нижнего уровня.

Каждый частичный треугольник представляет собой индекс треугольника в сцене (его порядковый номер) и охватывающую оболочку AABB данной части треугольника (или всего треугольника, если он не разрезался на части). См. рис. 4.2 как пример разделения треугольника на несколько частей. Массив индексов частичных треугольников *refIDs* изначально инициализируется возрастающей последовательностью целых чисел, начиная от 0 (см. рис.4.1). Для каждого частичного треугольника с использованием разрешения и границ охватывающей оболочки всей сетки *Grid0* вычисляется один 30-битный ключ $ceillID_i$, являющийся индексом ячейки, в которой находится центр оболочки частичного треугольника, и записывается в массив *CellIDs* (см. рис. 4.1). Старшие 21 бит $ceillID_i$ представляют индекс ячейки в *TopGrid0* (например, a, b, c, d, e на рис. 4.1), а младшие 9 бит представляют индекс ячейки в *BottomGrid* (например, t, u, v, w на рис. 4.1).

После вычисления ключей для каждого частичного треугольника, пара массивов *CellIDs* и *refIDs* сортируется в порядке возрастания значений *CellIDs* с помощью радикс-сортировки [29][74]. Процесс сортировки может быть ускорен методом компрессии-сотрировки-декомпрессии [3].

В целях создания *TopGrid0* отсортированный массив *CellIDs* сжимается, используя старшие 21 бит его элементов. Сжатие происходит следующим образом: каждому сегменту массива *CellIDs*, состоящему из подряд идущих элементов, имеющих одинаковые старшие 21 бит, на выходе ставится в соответствие один дескриптор. Каждый такой дескриптор соответствует непустой ячейке *TopGrid0* (равномерная сетка разрешения 128^3) и состоит из 3 чисел: 21-битного ключа, одинакового для всего сегмента частичных треугольников, начала сегмента в отсорти-

рованном массиве *CellIDs* и длины сегмента – числа записываются в полях *CellIDs*, *refsBegin* и *numRefs* соответственно.

21-битный ключ дескриптора используется в качестве индекса ячейки внутри *TopGrid0*, в которую при распределении треугольников по ячейкам (см. рис. 4.2) записывается информация о находящемся в данной ячейке сегменте частичных треугольников (массивы *refsBegin, numRefs* в которых *i-ый* элемент соответствует адресу сегмента треугольников *i-ой* ячейки *TopGrid0*). Дополнительно, для каждой *i-ой* ячейки *TopGrid0* поле $bottomGridIDs_i$ содержит порядковый номер непустой ячейки среди всех непустых ячеек *TopGrid0*.

Аналогично, в целях создания BottomGrid отсортированный массив *CellIDs* сжимается, используя все 30 бит его элементов. В результате сжатия появляется массив дескрипторов, где каждый дескриптор соответствует непустой ячейке *BottomGrid0* (равномерная сетка разрешения 1024^3) и сегменту частичных треугольников попадающих в ячейку.

Массив ячеек *BottomGrid*, содержащих поля *refsBegin* и *numRefs*, размещается в памяти в виде подряд идущих *N* равных блоков. Каждый блок однозначно соответствует одной непустой ячейке сетки *TopGrid0* и представляет собой малую сетку разрешения 8^3. При необходимости получения доступа к данным ячейки *BottomGrid* с 30-битным индексом вида *'aw'* (где *'w'* – слово из младших 9 бит индекса; *'a'* – слово из старших 21 бит индекса) производится чтение порядкового номера непустой ячейки $m=bottomGridIDs_a$ в *TopGrid0*. Обращение к ячейке *BottomGrid* происходит по адресу $m * 8^3 + w$ (см. рис. 4.1) в массиве ячеек *BottomGrids*.

Создание последовательности *M* уровней детализации $Grid_i$ для сетки *Grid0* производится для ускорения параллельных расчётов SAH-функции во время генерации BVH. Каждый уровень детализации $Grid_i$ представляет собой сетку меньшей детализации, в 2 раза меньшего разрешения, чем $Grid_{i-1}$ в каждом измерении *X, Y, Z* (см. рис. 4.2). Каждая ячейка сетки $Grid_i$ $(i > 0)$ содержит одно поле *numRefs*, где каждый элемент хранит количество частичных треугольников, принадлежащих

ячейке. Значение *numRefs* в *k-ой* ячейке сетки $Grid_i$ равно сумме значений в 8 смежных ячейках сетки $Grid_{i-1}$, соответствующих *k-ой* ячейке.

Максимальное разрешение *MR* наименее детальной сетки $Grid_{M-1}$ устанавливается равным 8 для эффективного исполнения алгоритма генерации BVH с применением технологии CUDA. Большие значения *MR* значительно увеличивают время генерации BVH (более чем в ~2 раза при *MR=16*), и незначительно сокращают время трассировки лучей, обеспечиваемое полученным деревом BVH (на ~5% при *MR=16*).

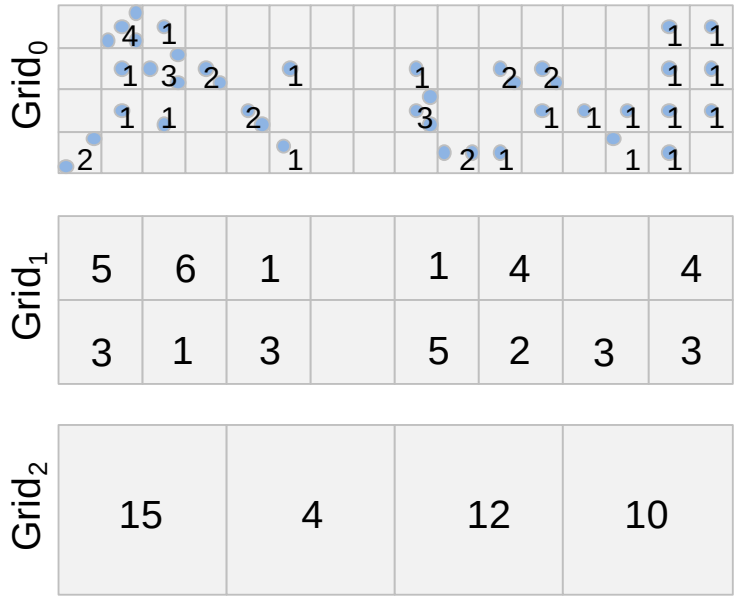


***Рис. 4.2**: 2D проекция уровней детализации Gridi для сетки Grid0. В каждой ячейке значится количество частичных треугольников, принадлежащих ей.*

Генерация иерархии BVH. На этой стадии производится генерация связей между узлами BVH без вычисления охватывающих оболочек. Иерархия BVH строится с использованием упрощённого вычисления SAH-функции в процессе генерации каждого узла при помощи 3D сеток распределения частичных треугольников $Grid_i$ *(i = M-1, M-2, ..., 1, ..., 0)*. Уровни иерархии связей BVH генерируются последовательно друг за другом: сначала генерируется нулевой уровень (1 корневой узел BVH), потом первый уровень (2 потомка корня), затем второй уровень (4 потомка узлов первого уровня) и т.д. Узлы внутри каждого уровня генерируются параллельно, используя ядра графического процессора при помощи CUDA.

```
// Алгоритм генерации иерархии связей бинарного дерева BVH
Алгоритм 4.1 EmitHierarchy( int2 * linkArray,
                            int  * sortedRefIDs,
                            Grid   topGrid,
                            int    numRefs)
 1: Queue SplitQueue[2];
 2:
 3: // Массив границ уровней дерева levelOffset[]
 4: // будет использоваться в процессе вычисления оболочек дерева
 5: int numLevels = 0;
 6: int levelOffset[60];
 7:
 8: int numNodes = 0;
 9: int numQElems = 1;
10: int qin = 0;
11:
12: // Инициализация очереди одним корневым узлом BVH
13: levelOffset[numLevels++] = 0;
14: IBB box = make_ibb(0,0,0, TopGrid.resx, TopGrid.resy, TopGrid.resz);
15: SplitQueue[qin] = SplitQueueInit(numRefs, box);
16: numNodes++;
17:
18: // Построение иерархии связей BVH уровень за уровнем
19: while numQElems > 0 do
20:   LevelOffset[numLevels++] = numNodes;
21:   // Параллельное вычисление SAH-функции для разделения узлов,
22:   // Запись новых задач разделения в выходную очередь SplitQueue[1-qin],
23:   // которая может быть больше входной очереди в 2 раза
24:   SplitQueueProcess(SplitQueue[qin], SplitQueue[1-qin], numQElems );
25:   // Сжатие очереди производится для удаления пустых элементов
26:   int newNumQElems = CompactQueue( SplitQueue[1-qin], 2*numQElems );
27:   // Добавление новых элементов в массив связей BVH
28:   // for each i = [0.. newNumQElems)
29:   //   linkArray[numNodes + i] = SplitQueue[1-qin].NodeInfo[i];
30:   AccumulateLinks(linkArray, numNodes, SplitQueue[1-qin], newNumQElems);
31:   numQElems = newNumQElems;
32:   numNodes += newNumQElems;
33:   qin = 1 - qin;
34: end while
```

Также как и в статье [71] в новом алгоритме содержится очередь задач *SplitQueue[input]* разделения узлов иерархии. Каждый элемент очереди представляет собой созданный узел BVH, границы узла в сетке $Grid_i$, количество треугольников в узле. Параллельно для каждого элемента очереди решается задача разделения: если узел содержит большое число треугольников, то создаётся два новых узла при помощи SAH-функции на следующем уровне иерархии BVH, где новые узлы содержат меньшее число треугольников, также в выходной очереди *SplitQueue[output]* добавляется две новые задачи разделения полученных узлов (см. Алгоритм 4.1). На следующей итерации алгоритма выходная очередь с предыдущей очереди становится входной очередью, и для каждого её элемента параллельно производится разделение узлов с помощью SAH функции.

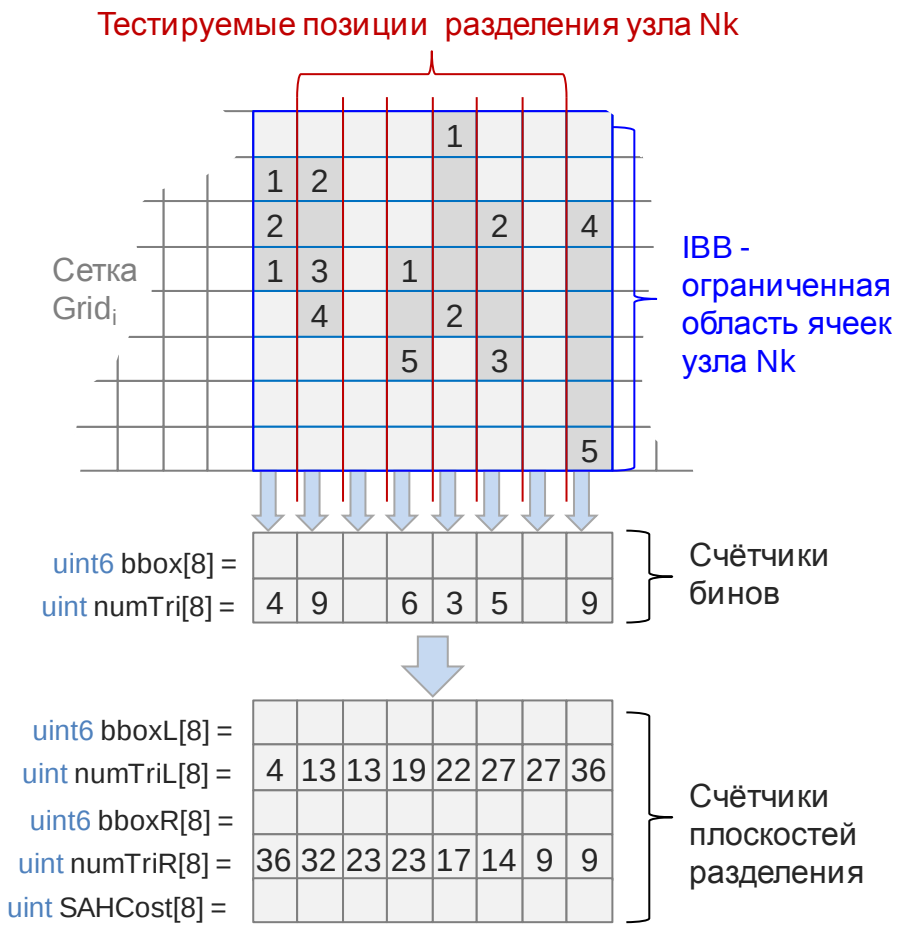


Рис. 4.3: *Оценка SAH функции с помощью сетки для разделения узла BVH.*

Вычисление SAH функции с помощью сеток $\mathbf{Grid_i}$. Каждый элемент очереди *SplitQueue* представляет параметры задачи оценки SAH функции, с помощью которой узел, соответствующий элементу очереди, разделяется на два новых узла. Эти параметры содержат:

- указатель на узел *Ni*

- указатель на текущий уровень детализации сетки $Grid_i$

- область ячеек этой сетки (далее обозначается *IBB*), относящихся к узлу *Ni*; *IBB* задаётся как трёхмерный целочисленный интервал.

Все задачи из одной очереди выполняются параллельно на GPU.

В начале процесса генерации связей дерева (см. Алгоритм 4.1) в очередь *SplitQueue* добавляется только один элемент, соответствующий задаче разделения корневого узла BVH, с указателем на уровень детализации $Grid_{M-1}$ (наименее детализированная сетка с максимальным разрешением 8^3) и также с областью ячеек *IBB*

= (0, 0, 0, $Grid_{M-1}$.resx, $Grid_{M-1}$.resy, $Grid_{M-1}$.resz), охватывающей все ячейки сетки $Grid_{M-1}$. Каждая ячейка сетки $Grid_i$ содержит количество частичных треугольников (см. рис. 4.3), распределённых в неё.

При разделении узла *Nk* для каждого измерения *X, Y, Z* производится вычисление значений SAH функции для 7 плоскостей разделения (см. рис. 4.3: обозначены красными линиями) между ячейками с ограниченной областью *IBB* ячеек узла *Nk*. В данном случае в процессе вычисления SAH функции используется биновый подход (см. [113]), реализованный на CUDA с использованием атомических инструкций: вдоль каждой базисной оси (*X, Y* или *Z*) пространство делится *B-1* плоскостями на *B* бинов, которым соответствуют равные интервалы вдоль этой оси в рамках общей для узла *Nk* границы *IBB*. Параллельно для каждого бина рассчитывается суммарное количество частичных треугольников и объёмлющая трёхмерная область *IBB* всех непустых ячеек, которые содержатся между плоскостями, образующими этот бин.

Используя данные из бинов, счётчики количества частичных треугольников (*numTriL*/*numTriR*) и объёмлющих областей непустых ячеек (*bboxL*/*bboxR*) по левую и правую стороны от каждой плоскости разделения рассчитываются с помощью параллельного префиксного суммирования (см. рис. 4.3). Упрощённая SAH оценка для конкретной плоскости *P[i]* вычисляется с учётом значений счётчиков по левую и правую сторону от плоскости по формуле:

*SAHCost(P[i])=SA(bboxL[i])*numTriL[i] + SA(bboxR[i+1])*numTriR[i+1]*

В качестве плоскости сечения узла *Nk* выбирается плоскость *p* с наименьшим значением *SAHCost*, создаётся два новых узла *NL, NR* в иерархии связей, на которые ссылается *Nk*. Для обоих узлов *NL, NR* добавляются две задачи разделения в выходную очередь *SplitQueue[output]*, где одна ассоциируется с сеткой $Grid_i$ и *IBB=bboxL[p]*, а другая с сеткой $Grid_i$ и *IBB=bboxR[p]*.

Максимальное разрешение наименее детальной сетки $Grid_{M-1}$ равно *MaxUseRes*, используемой для разделения корневого узла. Если для *k-ой* задачи разделения узла максимальная протяжённость области ячеек IBB_k меньше или равно *MaxUseRes /2*, то указатель сетки $Grid_i$ меняется на более детальную $Grid_{i-1}$, а

координаты области ячеек IBB_k умножаются на 2 чтобы составлять область ячеек внутри сетки $Grid_{i-1}$.

Задача разделения не делит узел и образует из него лист BVH со ссылкой на сегмент частичных треугольников, если её область ячеек *IBB* содержит только одну ячейку в самой детализированной сетке $Grid_0$.

Если устанавливается *MaxUseRes=8*, то во время поиска оптимальной плоскости разделения используется до 7 плоскостей и до 8 бинов. Тогда для задачи разделения узла выделяется 8 параллельных потоков графического процессора, где каждый поток независимо рассчитывает счётчики бинов и плоскостей разделения. В CUDA используются 32-элементные векторные вычисления (1 операция с 32 данными), поэтому всякий раз производится работа с 4 задачами разделения в одном векторе.

Пост-обработка после основной генерации иерархии BVH. Если в полученной иерархии некоторые листья содержат много частичных треугольников (например, более 4), то список треугольников тривиально делится на два непересекающихся списка, и соответствующий узел делится на два новых узла.

Вычисление охватывающих оболочек для узлов BVH параллельно для всех узлов каждого уровня BVH, уровень за уровнем, в порядке снизу вверх (сначала вычисляются охватывающие оболочки листьев, затем оболочки их предков и в конце вычисляется оболочка корневого узла). Границы уровней рассчитываются во время генерации иерархии и записываются в массиве *levelOffset* (см. Алгоритм 4.1).

4.3. Разделение треугольников на частичные треугольники

Если в сцене содержатся протяжённые и узкие треугольники, то в BVH могут образоваться перекрытия между охватывающими оболочками AABB нескольких таких треугольников. Наличие подобных перекрытий в BVH является причиной низкой скорости трассировки лучей, обеспечиваемой таким BVH. В алгоритме [104] описано решение такой проблемы в применении к генерации BVH на CPU процессоре, где в процессе генерации BVH используются пространственные разрезания треугольников (подобно kd-tree), если это позволяет уменьшить значение

SAH-функции. Использование пространственных разделений треугольников повышает эффективность BVH (т.е. обеспечиваемую им скорость трассировки лучей) на 20-60% для сцен, содержащих длинные или широкие/тонкие треугольники, и увеличивает потребление памяти. Процесс создания BVH по алгоритму, предложенному в статье [104], является медленным и не применим для интерактивной работы с анимированными сценами. Более того, могут появиться проблемы с переполнением памяти, например для CAD сцен (традиционно содержащих длинные и тонкие треугольники), т.к. общее количество разделений в алгоритме, предложенном в статье [104], не контролируется.

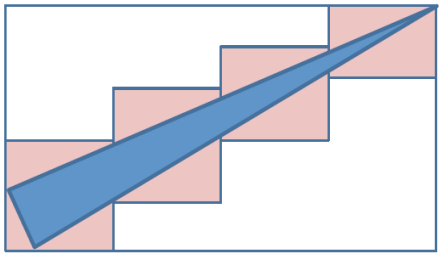


Рис. 4.4: *Узкий протяжённый треугольник разделён на 4 участка, каждый из которых ограничен охватывающей оболочкой. Каждый такой участок вместе с оболочкой и ссылкой на оригинальный треугольник, называется частичным треугольником. С применением разделения на частичные треугольники происходит более эффективное отсечение геометрии треугольника от пустого пространства, а также увеличение потребления памяти.*

В описанном алгоритме генерации дерева BVH в 4.1-4.2 используется 3D сетка с информацией о распределении треугольников в ячейках. Технически, во время распределения треугольников по ячейкам используется объект, называемый «частичным треугольником», содержащий охватывающую оболочку и указатель на треугольник. Такая оболочка может охватывать не весь треугольник, а только его часть, при этом сам треугольник не разрезается на несколько треугольников. Например, на рис. 4.4. изображён один протяжённый узкий треугольник, разделённый на 4 части, ограниченные меньшими охватывающими оболочками. Эти 4 части основного треугольника, ограниченные более мелкими оболочками, являются «частичными треугольниками».

Разделение треугольников сцены на несколько частичных треугольников осуществляется до генерации дерева BVH. Процесс разделения реализован парал-

лельно с применением графического процессора. Выделяемая память под хранение частичных треугольников строго ограничена, поэтому переполнения памяти исключаются полностью. Например, для хранения частичных треугольников может быть выделен блок не более чем на 10% превышающий объём занимаемой памяти блока оригинальных треугольников. При этом некоторые треугольники могут быть разрезаны на много частей, а некоторые могут быть не разрезаны вообще.

Выбирается максимальная протяжённость оболочки частич. треугольника:

$$triangleMaxWidth = sceneExt * sceneDividePortion \qquad (4.1)$$

где *sceneExt* – протяжённость всей оболочки сцены, *sceneDividePortion* – множитель, характеризующий желаемый максимальный размер частичного треугольника по сравнению с размерами сцены.

Изначально все частичные треугольники в массиве *refAABB* инициализируются охватывающими оболочками их оригинальных треугольников.

Параллельно для каждого *i-го* частичного треугольника рассчитывается запрашиваемое количество разрезаний:

$$reqSplits(refAABB_i) = int(refMaxExt_i / triangleMaxWidth)$$

где $refMaxExt_i$ является максимальной протяжённостью оболочки частичного треугольника. Значение $reqSplits(refAABB_i)$ округляется до меньшего целого числа, и представляет собой количество новых частичных треугольников которые необходимо произвести, чтобы протяжённость частичного треугольника не превосходила *triangleMaxWidth*. Значение $reqSplits(refAABB_i)$ масштабируется следующим образом:

$$scaled\_reqSplits_i = \frac{reqSplits(refAABB_i)}{\sum_{j \in all} reqSplits(refAABB_j)} * availableMemory$$

где *availableMemory* представляет количество элементов массива, которые могут быть использованы для хранения новых частичных треугольников. Каждый *i-ый* частичный треугольник $refAABB_i$ разделяется на $scaled\_reqSplits_i + 1$ равных частей, полученных путём равномерного разделения $refAABB_i$ вдоль оси, соответствующей наибольшей протяжённости $refAABB_i$ (см. рис. 4.4). Полученные частич-

ные треугольники в рамках границ плоскостей разделения ограничиваются более тесными оболочками AABB, и новые частичные треугольники записываются в выходную очередь.

Параллельное разрезание на частичные треугольники производится в 2 прохода. Во время первого прохода разрезаются оригинальные треугольники вдоль осей их оболочек, соответствующих наибольшей протяжённости. Во время второго прохода частичные треугольники из выходной очереди первого прохода тоже разрезаются (для широких треугольников может быть использовано другая ось разрезания на втором проходе). Весь процесс генерации частичных треугольников реализован для эффективного исполнения на графическом процессоре. Малые треугольники могут быть разрезаны на 2-3 части, а могут быть не разрезаны вообще. Очень длинные и тонкие треугольники могут быть разрезаны на тысячи мелких частей. Такое разрезание повысит скорость трассировки лучей, обеспечиваемое деревом BVH, т.к. разрезанным большим треугольникам соответствует несколько оболочек AABB, которые лучше ограничивают геометрию треугольника от пустого пространства. Как следствие на разных уровнях BVH образуется меньше перекрытий между узлами BVH.

Общее количество частичных треугольников ограничивается, например, на 110% от числа треугольников сцены. Неоднородное количество разрезаний среди всех треугольников реализовано на параллельной архитектуре графического процессора при помощи использования операций параллельной префиксной суммы и сегментной префиксной суммы [29].

4.4. Результаты

Параметры тестов. Реализован алгоритм построения BVH на основе сеток с применением технологии CUDA [79] для исполнения на графическом процессоре. Все измерения в данном разделе были сделаны с помощью графического процессора NVIDIA GTX 480 (1,5GB памяти). Для измерения характеристик алгоритма использовалось 6 различных сцен (см. рис. 4.5). Сцены Conference и Sponza состоят из многих длинных и тонких треугольников и представляют собой хорошую проверку для разработанного алгоритма параллельного разделения треугольников на

множество частичных треугольников в рамках ограниченного блока памяти. Остальные сцены в исходном виде состоят из мелких треугольников, а сцены Fairy Forest и Exploding Dragon являются анимированными.

Во всех сценах производится разделение треугольников на частичные треугольники так, чтобы протяжённость получаемых частичных треугольников не превышала 1/128-ой протяжённости сцены (*sceneDividePortion = 1/128 см. формулу 4.1).* Количество частичных треугольников строго ограничено максимальным значением 200% от количества оригинальных треугольников конкретной сцены. Во время вычисления значения SAH-функции для генерации узла BVH используется область сетки разрешением не более 8^3. Узлы BVH рекурсивно разделяются на новые узлы, если содержат более 4 частичных треугольников (часто используемое ограничение для многих реализаций решений задачи).

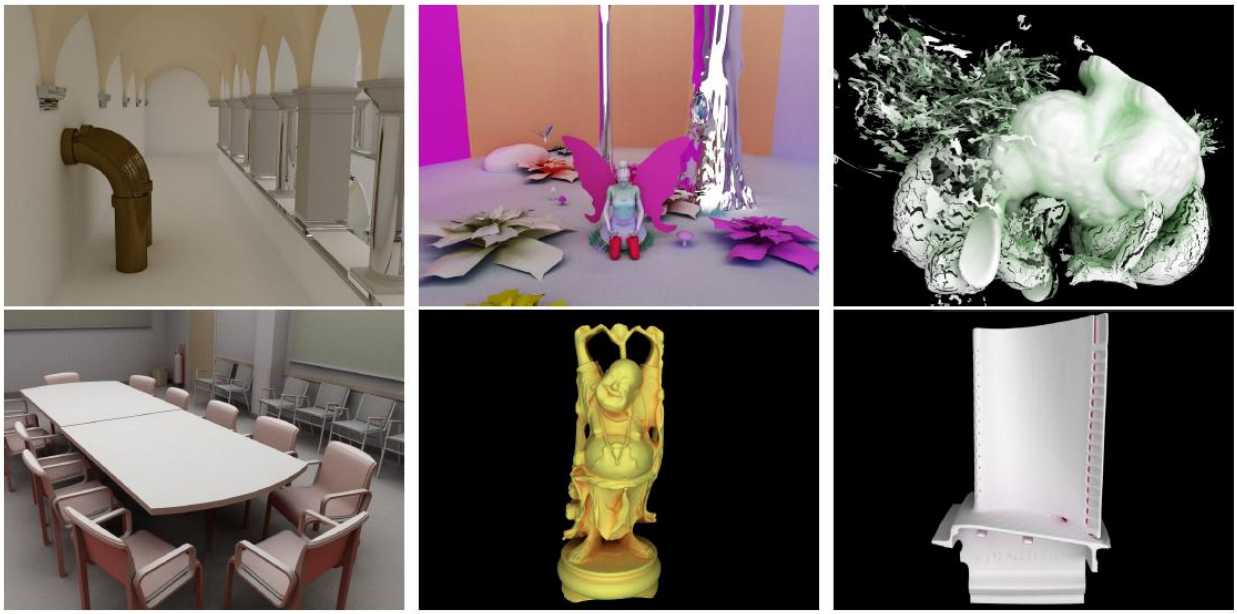


Рис. 4.5: *Изображения 6 сцен (Sponza, Fairy Forest, Exploding Dragon, Conference, Happy Buddha, Turbine Blade) для проверки алгоритма построения BVH на графическом процессоре. Время трассировки лучей с 5 уровнями переотражений: 55мс (45 млн. лучей/сек), 46мс (65 млн. лучей/сек), 35мс (63 млн. лучей/сек), 36мс (65 млн. лучей/сек), 16мс (105 млн. лучей/сек), 19мс (98 млн. лучей/сек).*

Для проверки скорости трассировки лучей, обеспечиваемой для каждой сцены генерируемым BVH, реализовано исполнение на графическом процессоре сто-

хастической Монте Карло трассировки путей [62] с 5 уровнями переотражений при разрешении изображения 1024x768. Трассировка путей реализована с применением диспетчера параллельных задач [12] и стадии сортировки лучей [3]. Это означает, что если в массиве лучей есть когерентные лучи, то во время стадии сортировки они будут расставлены рядом в выходном массиве лучей и далее перейдут к стадии поиска пересечений.

Время построения BVH. В таблице 4.1 представлена статистика времени построения деревьев BVH для тестовых сцен. Для больших сцен время построения увеличивается сравнительно медленно (16.1мс против 5.1мс) относительно роста числа треугольников в сцене (1760K треугольников против 67K), т.к. во время генерации узла и вычисления значений SAH-функции используется ограниченное количество вычислений и чтений из памяти. Для всех тестовых сцен используется одно и тоже разрешение сетки TopGrid, 128^3, составной части $Grid_0$, но количество непустых ячеек TopGrid от сцены к сцене варьируется от 0.1% до 2%.

	кол-во треуг-ов	кол-во частич. треуг-ов	Память, заним. сеткой	кол-во узлов BVH	Время построения BVH
Sponza	67K	186%	116Mb	83K	5.1ms
Fairy Forest	174K	127%	118Mb	140K	7.5ms
Exploding Dragon	252K	100%	27Mb	166K	8.1ms
Conference	282K	148%	115Mb	278K	8.2ms
Stanford Dragon	870K	100%	67Mb	497K	11.1ms
Happy Buddha	1080K	100%	73Mb	590K	11.4ms
Turbine Blade	1760K	100%	80Mb	990K	16.1ms

Разбивка по стадиям	
Разделение треуг-ов	2.7 мс
Сортировка треуг-ов	1.83 мс
Построение сетки	0.9 мс
Генерация BVH	4.67 мс
Вычисление AABB	1.0 мс

Таблица 4.1: *Время построения BVH с помощью нового алгоритма. Вторая колонка представляет количество треугольников сцены, третья – количество частичных треугольников в процентах от количества оригинальных. Четвёртая колонка представляет объём памяти, занимаемый сеткой в процессе построения BVH. Пятая колонка представляет время построения BVH. Выноска представляет разбивку времени построения BVH для сцены Stanford Dragon.*

Качество дерева BVH. Существуют практическая и теоретическая оценки качества BVH, как структуры, используемой для ускорения трассировки лучей. Практическая оценка использует измерения времени трассировки лучей как меру качества дерева BVH. Более быстрая трассировка лучей обеспечивается более ка-

чественным (или эффективным) деревом BVH. Теоретическая оценка использует вычисление значения рекурсивной функции:

Если *N* не является листом дерева:

$$C(N) = nc + C(NL) \cdot \frac{area(NL)}{area(N)} + C(NR) \cdot \frac{area(NR)}{area(N)} \qquad (4.2)$$

Если *N* является листом дерева:

$$C(N) = ntri$$

где *ntri* – количество частичных треугольников в листовом узле; *nc* – константа, характеризующая стоимость операции пересечения луча и оболочки внутреннего узла (часто используется *nc = 2*); *area(N)* – площадь поверхности оболочки узла *N*; *NL* и *NR* – непосредственные потомки узла *N*. Отношение *area(NL) / area(N)* представляет вероятность пересечения случайного луча с оболочкой узла *NL*. Меньшее значение суммы (*area(NL) + area(NR)) / area(N)* означает меньшую вероятность попадания случайного луча в каждый из потомков узла N и, эквивалентно, лучшее отсечение геометрии от пустого пространства. Следовательно, чем более высокое значение имеет *1 / C(Nroot)*, тем более высокое качество имеет дерево, начинающееся с узла *Nroot*.

В таблице 4.2 представлены практические и теоретические измерения качества BVH для 4 разных сцен и 4 разных алгоритмов. Алгоритм, описанной в данной главе в двух вариантах, сравнивается с двумя более медленными алгоритмами, работающими на CPU, генерирующими деревья BVH высокого качества. Все алгоритмы, участвующие в сравнении реализованы в одной программе и запускались для одних и тех же сцен. Для каждого из алгоритмов деревья BVH создавались таким образом, чтобы листья деревьев не содержали более 4 треугольников (или частичных треугольников). Алгоритм SplitBVH (см. статью [104]) производит деревья BVH, обеспечивающие самую быструю трассировку лучей, однако время построения дерева с помощью такого алгоритма для указанных сцен измеряется в секундах или даже минутах. Описанный в данной главе алгоритм GridBvh без стадии разделения треугольников обеспечивает на 10-70% более низкую скорость трассировки лучей по сравнению с SplitBVH для сцен, содержащих протяжённые треугольники, такие как Sponza или Conference. Однако, добавление быстро исполняющейся ста-

дии разделения треугольников позволяют алгоритму GridBvh-split обеспечивать скорость трассировки лучей очень близкую к той, которую обеспечивает алгоритм SplitBvh.

Изменение параметров алгоритма построения. Если для вычисления значений SAH функции использовать область сетки с протяжённостью не более 16 ячеек (вместо 8 описанных ранее), то подобное изменение может повлечь более медленное построение дерева BVH (около 30% медленнее) и более быструю трассировку лучей, обеспечиваемую таким деревом (около 3-5% быстрее). Более высокое значение *sceneDividePortion > 1/128 (см. формулу 4.1)* даёт более низкое потребление низкое потребление памяти на блок частичных треугольников и более медленную трассировку лучей (для сцен с протяжёнными треугольниками типа Sponza или Conference замедление значительно, около 50%).

Сравнение скорости построения с другими алгоритмами построения ускоряющих структур на графическом процессоре. Алгоритмы построения ускоряющих структур (не только BVH, но также 3D сеток и kd-деревьев) появились недавно, и их основным приложением является интерактивная трассировка лучей в анимированных сценах. В таблице 4.3 приведено время построения ускоряющих структур на графическом процессоре из недавних опубликованных работ по данной теме. В таблице 4.3 приведёно время построения BVH наиболее быстрого варианта алгоритма HLBVH [82]. Для сцен, состоящих из треугольников однородных размеров, таких как Buddha и Turbine, алгоритм GridBVH-split обеспечивает трассировку лучей на 10% быстрее, чем аналогичные деревья BVH, построенные по алгоритму HLBVH. В алгоритме HLBVH не решается проблема наличия протяжённых треугольников, поэтому для сцен, содержащих треугольники неоднородных размеров, таких как Conference и Fairy, скорость трассировки лучей, обеспечиваемая деревьями, построенными по алгоритму GridBVH-split, существенно выше (30-50%) по сравнению с другими алгоритмами.

Дерево BVH, построенное по алгоритму LBVH [71], и 3D сетка, построенная по алгоритму Калоянова [60], строятся медленнее, чем GridBVH-split и обеспечивают значительно более медленную трассировку лучей (на 50-100% медленнее).

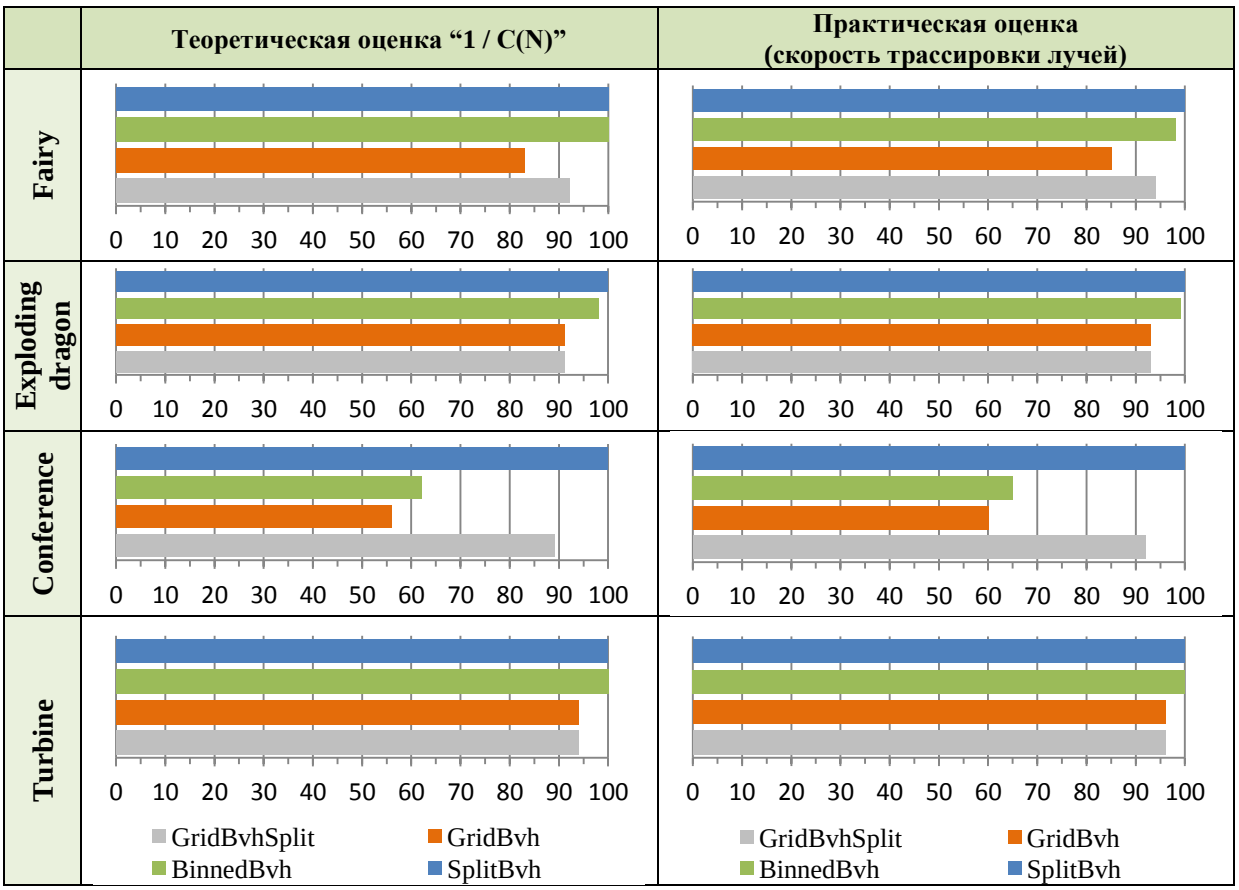


Таблица 4.2: *Оценка качества BVH для 4 алгоритмов: “GridBvhSplit” – описанный в данной главе алгоритм, “GridBvh” – тот же алгоритм, но без разделения треугольников на частичные, “BinnedBvh” – работающий на CPU биновый алгоритм [113], “SplitBvh” – работающий на CPU алгоритм Стича [104]. В левой колонке представлены значения теоретической оценки качества BVH на основе значения функции 1 / C(N), см. формулу 4.2. В правой колонке представлены значения практической оценки качества BVH, т.е. измерена скорость трассировки лучей с помощью алгоритма Монте Карло трассировки пути с 5 уровнями переотражений (кадры соответствующих измерений см. рис. 4.5). На данном графике максимальные значения оценки качества BVH и скорости трассировки лучей нормализованы к 100%, соответствующие алгоритму SplitBVH, который медленнее всех производит BVH, однако обеспечивает самое высокое качество (т.е. скорость трассировки лучей).*

	Измерено на графич. процессоре	Fairy Forest	Exploding Dragon	Buddha	Turbine
GridBvhSplit BVH	GTX 480	8 мс	8 мс	11 мс	16 мс
HLBVH [82] версия быстрого построения BVH, менее эффективное дерево	GTX 280			32 мс (16 мс)	42 мс (21 мс)
LBVH [71] BVH	GTX 280	124 мс (62 мс)	66 мс (33 мс)		
Алг. Калоянова [60] 3D сетка	GTX 280	24 мс (12 мс)	16 мс (8 мс)		

Таблица 4.3: *Время построения ускоряющей структуры для различных конкурирующих быстрых недавно опубликованных алгоритмов, работающих на графическом процессоре:* [82] [71] [60]. *Из 3 конкурирующих статей было выбрано лучшее время построения ускоряющих структур для данных сцен, даже если качество этих ускоряющих структур значительно ниже, чем предлагаемое ускоряющей структурой GridBvhSplit. В скобках представлено в 2 раза меньшее время для сравнения с алгоритмом GridBvhSplit, замеренным на GTX 480. Предполагается, что на GTX 480 параллельные алгоритмы могут работать в 2 раза быстрее (т.к. GTX 480 в 2 раза мощнее, чем GTX 280).*

4.5. Заключение

В данной главе представлен алгоритм построения ускоряющей структуры BVH, исполняющийся на современном графическом процессоре. Описанный алгоритм имеет линейную сложность, является эффективным для обеспечения трассировки лучей в анимированных сценах (быстрое по сравнению с аналогами построение BVH и быстрая трассировка лучей в пределах 10% от скорости наиболее эффективного BVH). Процесс построения BVH ускорен по сравнению с другими благодаря использованию компрессии-сортировке-декомпрессии, использованию сеток с информацией о распределении треугольников во время вычисления SAH-функции. Эффективная трассировка лучей обеспечена также благодаря стадии быстрого разделения треугольников на частичные треугольники, которая даёт лучшее отделение геометрии треугольников от пустого пространства. При этом полностью исключается возможность переполнения памяти в процессе параллельного разрезания треугольников.

Глава 5

Алгоритм построения ускоряющей структуры BVH с помощью бинарного поиска и очередей задач

Разработанный ранее алгоритм Hierarchical Linear Bounding Volume Hierarchies (кратко HLBVH, см статью [82]), работая на графическом процессоре, демонстрировал возможность строить ускоряющую структуру, необходимую для трассировки лучей, в реальном времени даже для сцен, содержащих миллионы динамических треугольников. В данной главе описан более простой в реализации и более мощный вариант алгоритма HLBVH, который назван HLBVH2 (впервые опубликован в статье [6]). В новом алгоритме сложные структуры хранения префиксных сумм, сжатия и частичного поиска в ширину, необходимые для пространственного распределения, были заменены на конвейер, построенный на основе простой концепции очередей задач и бинарного поиска. Новый алгоритм строит идентичное дерево значительно быстрее (5-10 раз), потребляет меньше памяти (в 4 раза) по сравнению с алгоритмом HLBVH.

В основе разработки алгоритма HLBVH2 лежит идея, описанная в работе [71]: область внутри охватывающей оболочки AABB сцены дискретизуется используя n бит (2^n равных интервалов) в каждом из 3 измерений; на треугольнике выбирается одна точка (например, центр), для которой вычисляется мортон-код. Мортон-код имеет $3n$ бит, для точки составляется последовательным чередованием бит дискретных координат в x, y, z измерении (см. рис 5.2). Затем треугольники сортируются в порядке возрастания значений их мортон-кодов. После этого строится дерево BVH, организующее треугольники с использованием мортон-кодов с помощью сложной процедуры двойной сортировки и удаления цепочек узлов, имеющих по одному потомку. В новом алгоритме HLBVH2 данная процедура реализована более эффективно.

5.1. Описание алгоритма

В алгоритме HLBVH2 для центральной точки каждого треугольника вычисляется 30-битный мортон-код, определённый внутри области, ограниченной охватывающей оболочкой сцены. Далее происходит сортировка треугольников в порядке возрастания значений их мортон-кодов. Сортировка всего множества треугольников производится один раз в течение генерации дерева BVH. В отличие от алгоритма HLBVH [82], в котором используется компрессия-сортировка-декомпрессия для ускорения сортировки, на стадии сортировки в HLBVH2 используется метод новой радикс-сортировки на графическом процессоре [74].

Мортон-коды определяют иерархию сеток, где $3n$-битный код идентифицирует один воксель регулярной сетки с дискретизацией на 2^n интервалов в каждом из 3 измерений, где старшие $3m$ бита мортон-кода ($m < n$) идентифицирует родительский воксель более крупоячеечной регулярной сетки с дискретизацией на 2^m интервалов в каждом из 3 измерений. Далее с помощью метода параллельной компрессии создаётся массив кластеров треугольников, где каждый *кластер* представляет интервал в отсортированном массиве треугольников, в этом интервале треугольники имеют одинаковые значения старших $3m$ бит их мортон-кодов. На рис. 5.3 представлен пример 2-битных кластеров в 2D и соответствующие интервалы отсортированных треугольников. На рис. 5.1 изображён пример с 4 кластерами, соответствующими интервалам треугольников разной длины, но имеющим одинаковые старшие $3m$ бит.

После того как все кластеры треугольников получены, производится параллельная генерация дерева BVH для интервала треугольников внутри каждого кластера на основе бинарного поиска с использованием младших $3(n - m)$ бит мортон-кодов треугольников. После этого все кластеры должны быть организованы в дереве BVH верхнего уровня. В процессе генерации такого дерева каждый кластер треугольников используется как один неделимый примитив, представленный оболочкой всего кластера треугольников (см. рис. 5.1). Количество получаемых после компрессии кластеров всегда ограничено максимальным числом 2^{3m}, где 2^{3m} – количество всех ячеек регулярной сетки с дискретизацией на 2^m интервалов в каждом из 3 измерений. На практике, количество получаемых кластеров на 2 порядка

меньше чем 2^{3m}. Поэтому можно выбрать любой алгоритм для построения качественного BVH верхнего уровня, даже работающий на CPU, т.к. количество используемых примитивов в алгоритме построения и время его исполнения ограничено сверху.

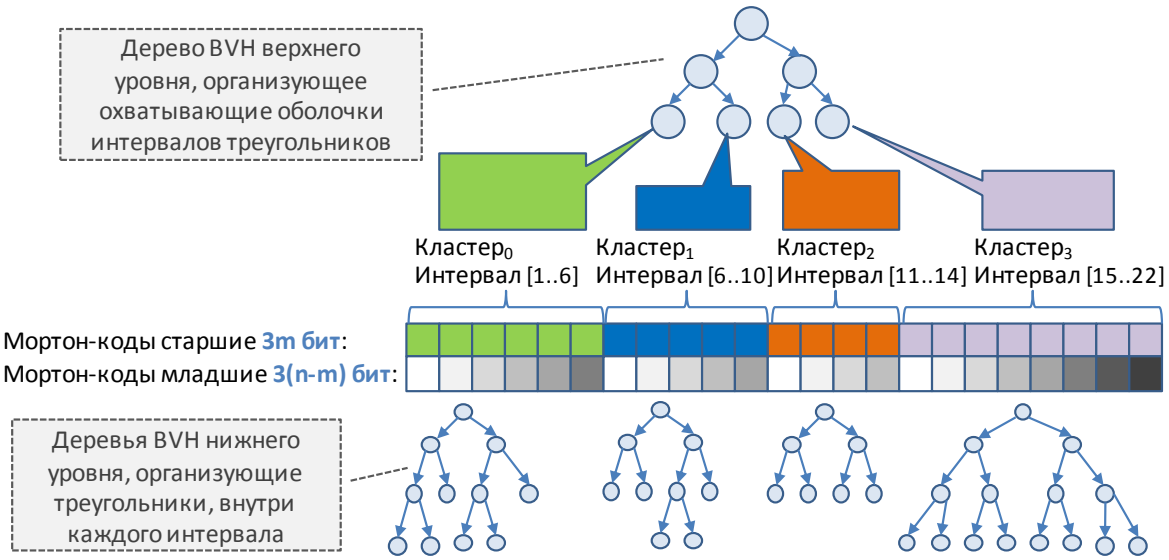


Рис. 5.1: *Построения BVH по алгоритму HLBVH2: для создания поддеревьев BVH на нижнем уровне используется быстрый метод на основе мортон-кодов и бинарного поиска; для создания BVH верхнего уровня, организующего кластеры в иерархию, используется любой метод построения дерева, эффективного для трассировки лучей.*

Очереди задач. В алгоритме LBVH [71] генерация иерархии BVH производится в порядке поиска в ширину, уровень за уровнем, начиная с корня, где генерация всех узлов внутри уровня дерева производится параллельно на графическом процессоре. За одну параллельную итерацию создаётся один уровень дерева, в следующей итерации создаётся уровень дерева, содержащий потомков узлов предыдущего уровня. Очередь задач содержит массив созданных узлов BVH, где каждому узлу соответствует сегмент отсортированных по мортон-кодам треугольников. Каждая задача в очереди заключается в том, чтобы разделить сегмент треугольников на два сегмента и создать два новых соответствующих узла, или не производить разделение и оставить узел листом. По итогам выполнения параллельного разделения узлов создаётся выходной массив *numNewNodes*, где каждый элемент равен 2 или 0, в зависимости от того требуется ли разделение узла и создание двух потомков соответствующему родительскому узлу в очереди задач. После этого для

массива *numNewNodes* параллельно вычисляется префиксная сумма для определения смещений новых узлов-потомков в следующем уровне дерева. При этом необходимо временно записывать в глобальную память все данные, которые необходимо иметь при записи узлов потомков в конкретные позиции выходной очереди. Проблемой LBVH [71] является содержание временных данных, занимающих большие объёмы памяти, запуск дополнительной параллельной подпрограммы для расчётов префиксной суммы и подпрограммы для записи узлов-потомков в конкретные позиции памяти – т.е. из-за необходимости запускать несколько параллельных подпрограмм для создания одного уровня увеличивается время отклика графического процессора. В HLBVH [82] данная ситуация была улучшена, где во время одной параллельной итерации создания иерархии производится 2-3 уровня, однако недостатком является сложная обработка подобного случая в рамках одной итерации.

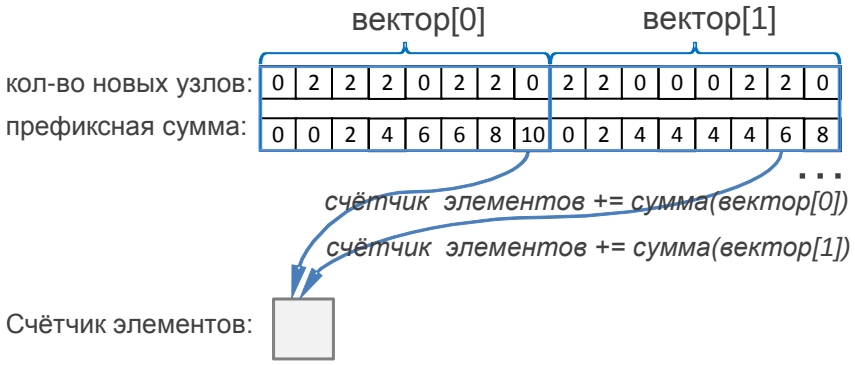


Рис. 5.2: *Параллельная обработка каждого элемента очереди задач может произвести различное количество новых элементов в новой очереди. Используется метод расчёта локальной префиксной суммы новых элементов внутри каждого вектора для расчёта смещений новых элементов в новой очереди относительно общего смещения вектора. Атомическое добавления суммы элементов вектора к глобальному счётчику позволяет вычислить общее смещение всех новых элементов вектора.*

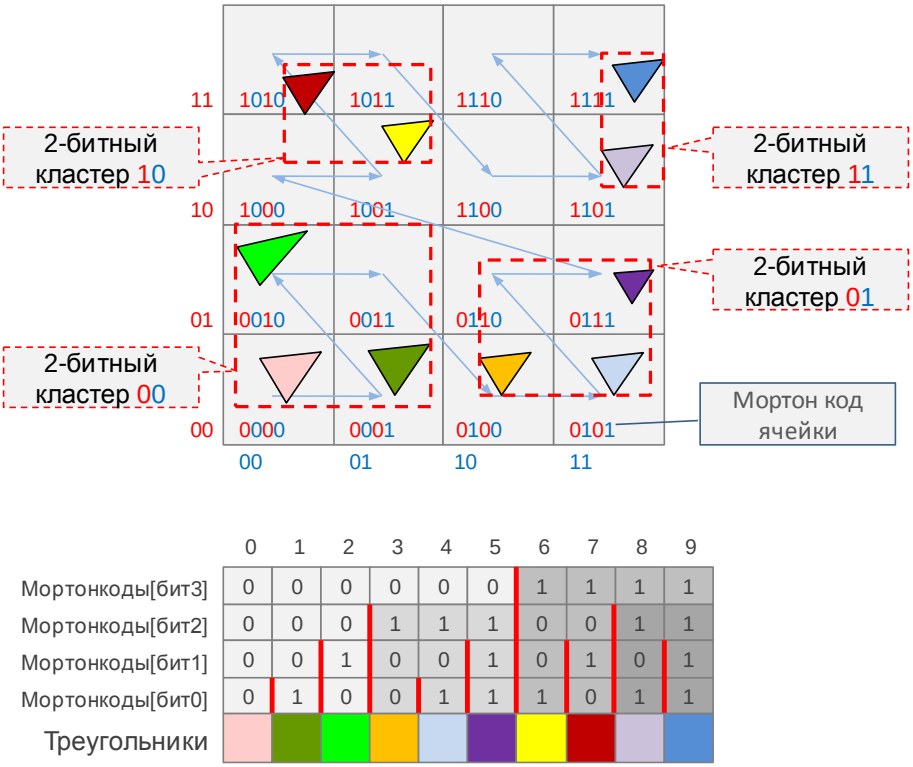
В новом алгоритме HLBVH2 используется другой подход, где каждая индивидуальная задача из очереди задач соответствует обработке в одном потоке одного узла. Во время параллельной обработки задач для каждого вектора потоков считывается вектор задач для обработки из входной очереди с использованием одной атомической инструкции, обновляющей голову очереди, на весь вектор. После того

как каждый поток вектора[2] вычислил количество новых узлов-потомков (0 или 2), все потоки вектора используются для вычисления префиксной суммы локальной внутри вектора для получения смещений новых узлов потомков в выходной очереди задач (и в соответствующем новом уровне дерева) относительно общего смещения вектора. Наконец, первый поток вектора производит одну атомическую операцию добавления суммы новых узлов вектора к глобальному счётчику всех новых узлов. Эта атомическое сложение позволяет для каждого отдельного вектора получать значение общего смещения вектора в выходной очереди новых задач (см. рис. 5.2). Используя в параллельной программе обработки очереди задач подобный подход, становится возможным создавать в этой же программе новые очереди задач и записывать новые элементы в выходную очередь без затрат на временное хранение данных и запуск дополнительных параллельных подпрограмм для вычисления глобальных префиксных сумм.

Генерация иерархии BVH с помощью бинарного поиска. Алгоритм HLBVH2 является производным от алгоритмов HLBVH и LBVH, т.к. производит идентичное дерево BVH методом рекурсивного деления отсортированного массива треугольников на два массива с использованием мортон-кодов. В ранних методах LBVH и HLBVH каждому узлу соответствовал интервал треугольников, отсортированных в порядке возрастания их мортон-кодов, разделение узла на два новых требовало произвести поиск первого элемента в интервале треугольников, чей код отличался от предыдущего элемента. Это реализовывалось в обратном порядке: вместо того, чтобы обрабатывать один узел в одном параллельном потоке, где в каждом потоке должен производиться поиск позиции разделения интервала, в каждом потоке производилась работа с одним значением мортон-кода, и проводилась проверка, отличается ли он от предыдущего мортон-кода в массиве. Если отличался, то номер потока выбирался как позиция разделения родительского узла. Однако, доступ к родительскому узлу требовал заранее рассчитать отображение между отсортированными треугольниками и интервалами узлов, что требовало вычислять дополнительную префиксную сумму [29].

[2] В платформе программирования CUDA вектор называется *warp* (варп), и он содержит 32 параллельных потока, для которых выполняется одна инструкция. В OpenCL вектор называется group (группа).

Вместо этого в более новом алгоритме HLBVH2 подобная сложная обработка упрощена: задача разделения одного узла и соответствующего интервала треугольников решается в одном потоке, который находит плоскость разделения. Вместо того чтобы проверять каждый мортон-код в интервале треугольников конкретного узла, производится бинарный поиск.



	0	1	2	3	4	5	6	7	8	9
Мортонкоды[бит3]	0	0	0	0	0	0	1	1	1	1
Мортонкоды[бит2]	0	0	0	1	1	1	0	0	1	1
Мортонкоды[бит1]	0	0	1	0	0	1	0	1	0	1
Мортонкоды[бит0]	0	1	0	0	1	1	1	0	1	1
Треугольники										

Рис. 5.3: *Примеры: 2D пространство, дискретизированное с использованием 2 бит (2^2 равных интервалов) в каждом из 2 измерений; отсортированный массив треугольников в порядке возрастания их соответствующих 4-битных мортон-кодов; четыре 2-битных кластера треугольников*

В каждой итерации генерации иерархии BVH запускается параллельная программа, в которой параллельные потоки работают над разделением узлов одного уровня дерева, где для каждого узла производится по 2 потомка или он остаётся листом дерева. В случае создания потомков узла они добавляются в иерархию дерева и соответствующие им новые задачи разделения записываются в выходную очередь, которая будет входной очередью на следующей итерации (см. процесс заполнения очередей задач на рис. 5.4, соответствующий массива треугольников, изображённому на рис. 5.3).

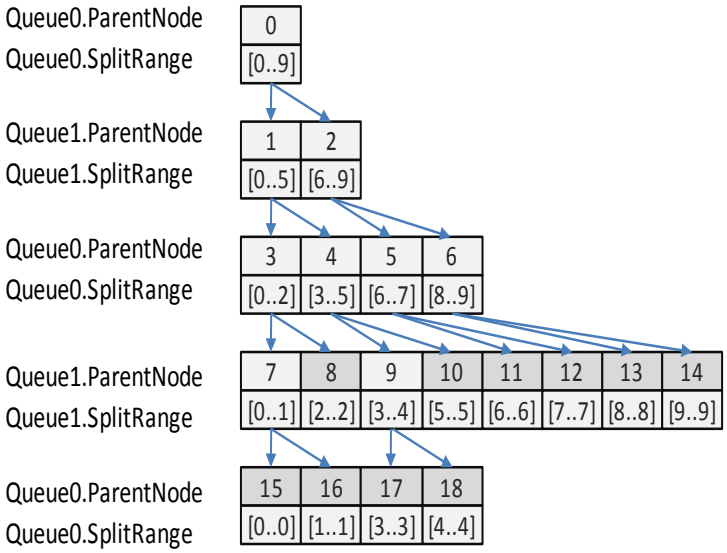


Рис. 5.4: *Процесс рекурсивного деления массива треугольников плоскостью, проходящей посередине оболочки, охватывающей центры треугольников массива.*

Пусть массив треугольников отсортирован в порядке возрастания мортон-кодов, где каждый мортон-код состоит из *3n* бит (см. рис. 5.2). Пусть узлу *N* иерархии BVH соответствует интервал *[a .. b]* массива треугольников и их мортон-кодов. В интервале *[a .. b]* значения мортон-кодов на уровне *k-го* бита упорядочены так, что сначала идут нулевые значения, после этого единичные значения. С помощью бинарного поиска находится позиция *mid* (отмечена красной линией на рис. 5.2) последнего нулевого значения мортон-кода на уровне *k-го* бита, после которого следуют единицы. Тогда первому потомку *NL* узла *N* соответствует интервал треугольников *[a .. mid]*, а второму потомку *NR* узла *N* соответствует интервал треугольников *[mid+1 .. b]*. На следующей итерации генерации иерархии для двух новых узлов *NL* и *NR* в их соответствующих интервалах производится поиск позиции разделения с помощью бинарного поиска границы между нулями и единицами на уровне *(k-1)*-го бита мортон-кода и т.д. Вместо разделения узел дерева остаётся листовым, если интервал треугольников содержит не более *numLeafTri* элементов, где значение *numLeafTri* устанавливается пользователем.

Данный процесс рекурсивного деления интервалов треугольников и создания иерархии с помощью мортон-кодов и бинарного поиска эквивалентен рекурсивному разделению области пространства, в котором находится массив треугольников, вдоль оси Z (или Y или X) секущей плоскостью пополам, распределению

треугольников по обе стороны плоскости и повторению процесса разделения для двух полученных массивов. Причина, по которой этот процесс является эффективным, заключается в том, что для узла с интервалом, состоящим из *ntri* треугольников, плоскость разделения интервала находится с помощью бинарного поиска за *O(log(ntri))* шагов и соответствующих чтений значений мортон-кодов из памяти. Тогда как по более ранним алгоритмам HLBVH/LBVH необходимо было для каждого узла произвести чтение *ntri* мортон-кодов и производить операции префиксных сумм.

Расчёт охватывающих оболочек BVH. После того как иерархия дерева получена запускается процесс расчёта охватывающих оболочек, начиная с нижних уровней дерева, уровень за уровнем. Где на каждом уровне параллельно для всех его узлов рассчитываются охватывающие оболочки. Для листьев охватывающие оболочки рассчитываются на основе содержащихся в них треугольников. Для внутренних узлов на верхних уровнях оболочки рассчитываются на основе оболочек их потомков, рассчитанных ранее.

5.2. Результаты

Параметры тестов. Алгоритм HLBVH2 реализован с применением технологии CUDA для исполнения на графическом процессоре. Все измерения в данном разделе были сделаны с помощью графического процессора NVIDIA GTX 480 (1,5GB памяти). Для измерения характеристик алгоритма использовалось 4 различных сцены (см. рис. 5.5).

Для сортировки треугольников использовались 30-битные мортон-коды ($n = 10$). Для разделения треугольников на кластеры использовались старшие 15 бит мортон-кода (т.е. $m = 5$). В качестве метода построения BVH верхнего уровня, организующего кластеры в иерархию, использован описанный в 4 главе метод GridBvh, в котором для выполнения данной задачи используется только сетка TopGrid, а сетки нижнего уровня BottomGrid не создаются. Таблица 5.1 содержит статистику построения BVH с помощью алгоритма HLBVH2. Применение алгоритма GridBvh является выгодным, ещё и потому что построение BVH полностью осуществляется на графическом процессоре без дополнительных пересылок дан-

ных между графическим и центральным процессорами. Качество полученных деревьев BVH было измерено с помощью формулы оценки качества дерева (см. формулу 4.2) и также с помощью практического измерения скорости трассировки лучей, обеспечиваемой построенным деревом BVH (см. таблицу 5.2).

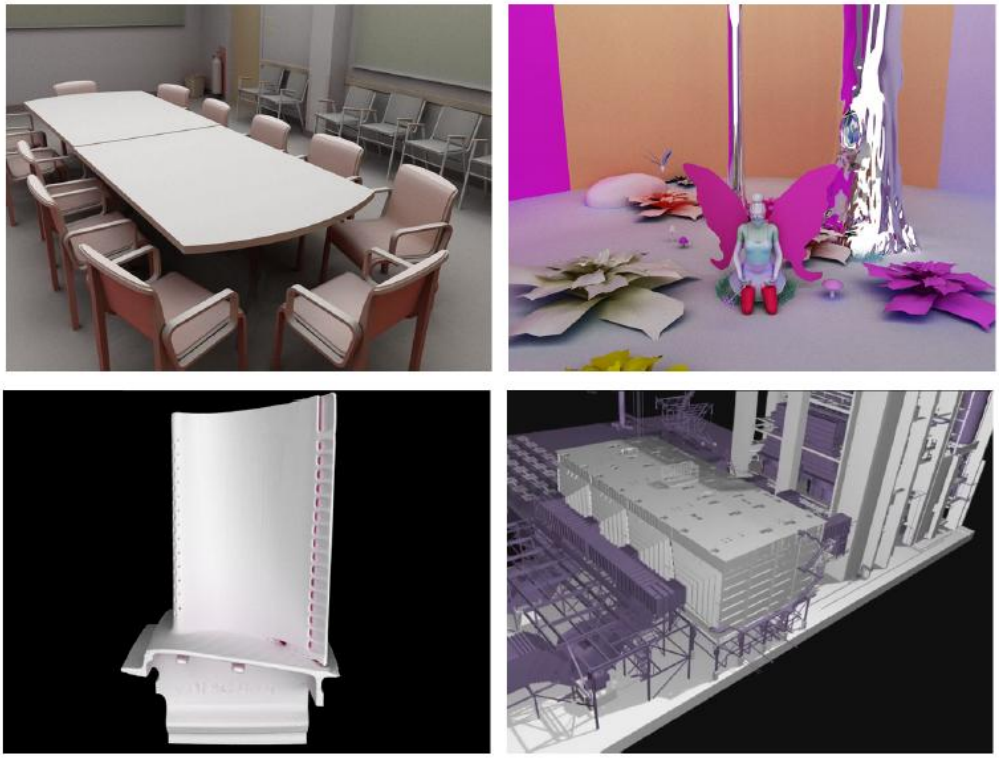


Рис. 5.5: *Тестовые сцены: Conference, Fairy Forest, Turbine Blade, Power Plant*

	кол-во треуг-ов	кол-во 15-бит кластер.	Потребление памяти		Время построения	
			во время построения	готовое дерево	HLBVH	HLBVH2
Fairy Forest	174K	2.4k	33Mb	4Mb	23ms	4.8ms
Conference	282K	2.5k	36Mb	6.5Mb	45ms	6.2ms
Stanford Dragon	870K	2.1k	51Mb	20Mb	81ms	8.1ms
Turbine Blade	1760K	2.3k	75Mb	42Mb	137ms	10.5ms
Power Plant	12700K	2.0k	367Mb	290Mb	-	62.1ms

Разбивка по стадиям, время мс	
Сорт. треуг-ов	1.83
Верх. уров.	2.17
Нижн. уров.	3.1
Вычисл AABB	1.0

Таблица 5.1: *Время построения и потребление памяти алгоритма HLBVH2. В данную таблицу для сравнения также выписано время исполнения раннего алгоритма HLBVH в 2-уровневой версии, обеспечивающего такое же качество BVH, как и 2-уровневая версия HLBVH2. Новый алгоритм позволяет строить BVH на порядок быстрее. Благодаря уменьшенному потреблению памяти в HLBVH2 (в 4 раза по сравнению с HLBVH) стало возможно строить ускоряющую структуру для крупной сцены PowerPlant, состоящей из 12,7 млн. треугольников.*

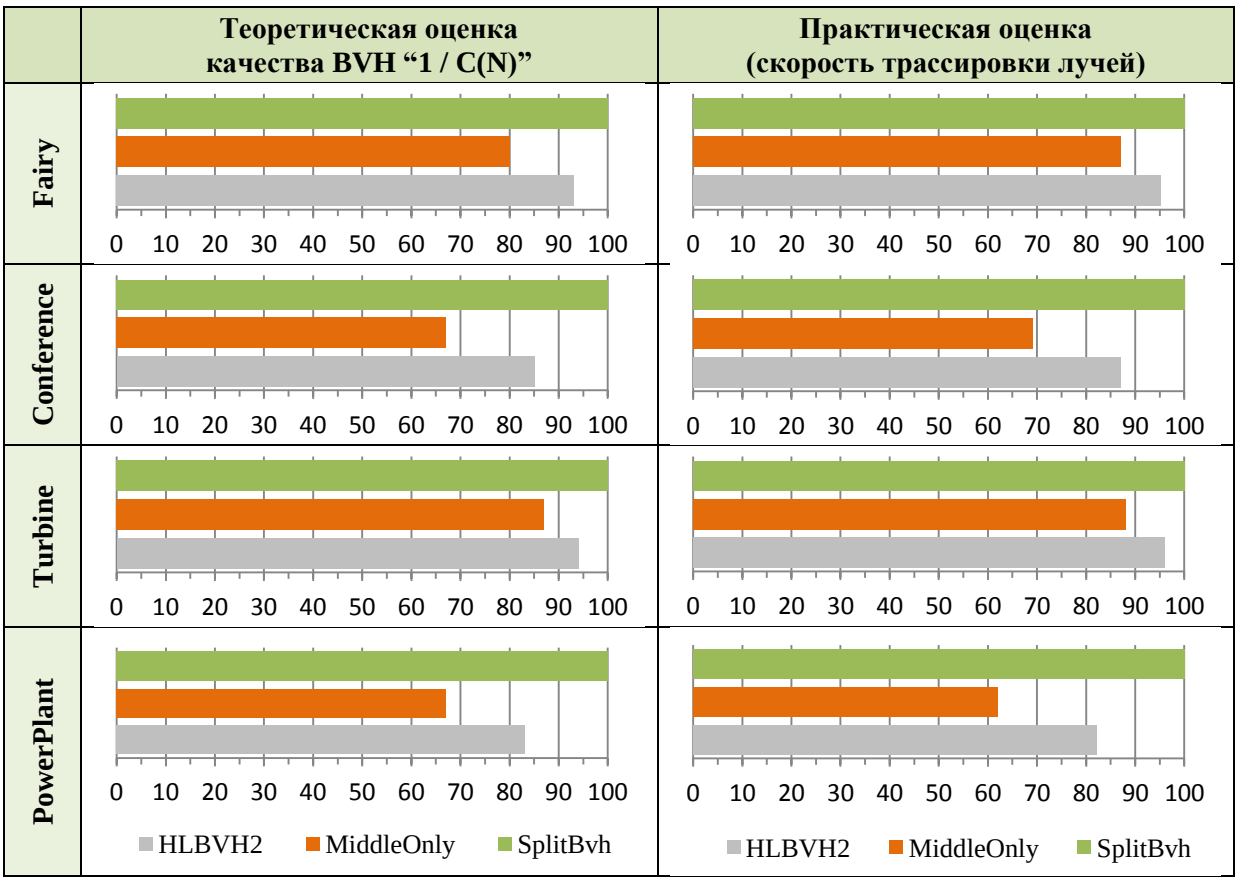


Таблица 5.2: *Оценка качества BVH для 3 алгоритмов: “**HLBVH2**” – двухуровневое построение дерева: соответствующие кластерам треугольников поддеревья BVH нижнего уровня построены на основе мортон-кодов (младшие 3(n-m) биты) и бинарного поиска; BVH верхнего уровня, организующее кластеры, построено по алгоритму GridBVH, описанному в главе 4, использующему вычисления значений упрощённой SAH-функции; “**MiddleOnly**” – сокращённая версия алгоритма HLBVH2, для генерации всех узлов используются мортон-коды и бинарный поиск (т.е. рекурсивное деление оболочки массива треугольников пополам); “**SplitBVH**” – работающий на CPU алгоритм Стича [104]. В левой колонке представлены значения теоретической оценки качества BVH на основе значения функции 1 / C(N), см. формулу 4.2. В правой колонке представлены значения практической оценки качества BVH, т.е. измерена скорость трассировки лучей с помощью Монте Карло трассировки пути с 5 уровнями переотражений (кадры соответствующих измерений см. рис. 5.1). На данном графике максимальные значения оценки качества BVH и скорости трассировки лучей нормализованы к 100%, соответствующие алгоритму SplitBVH, который медленнее всех производит BVH, однако обеспечивает самое высокое качество.*

5.3. Заключение

В данной главе предложен способ реализации универсальных очередей задач, порождающих новые задачи в каждой итерации, и позволяющих строить простые и быстрые параллельные алгоритмы с уменьшенным объёмом используемой памяти. Предложен более простой и эффективный способ построения поддеревьев нижнего уровня методом деления пространства пополам. Разработанные методы позволяют строить деревья BVH высокого качества с помощью нового алгоритма на порядок быстрее, используя в несколько раз меньший объём памяти по сравнению с более ранним алгоритмом HLBVH [82].

Качество деревьев BVH, полученных с помощью алгоритма HLBVH2 на 5-10% ниже (см. таблицу 5.2), чем у деревьев, построенных по алгоритму GridBvhSplit (см. главу 4 таблицу 4.2). Скорость построения BVH по алгоритму HLBVH2 выше на 20-30%. Использование множества частичных треугольников (см. главу 4) в качестве примитивов в процессе построения по алгоритму HLBVH2 повысит эффективность получаемых деревьев. При этом возрастёт время построения дерева до уровня показателей GridBvhSplit. Однако если использовать массив частичных треугольников в качестве примитивов в процессе построения дерева по алгоритму HLBVH2, то это будет более предпочтительным по сравнению с алгоритмом GridBvhSplit ввиду более низкого потребления памяти. В процессе построения дерева алгоритм GridBvhSplit использует вспомогательные разреженные сетки высокого разрешения, тогда как в алгоритме HLBVH2 вместо сеток нижнего уровня используется метод разделения пространства с применением мортон-кодов и бинарного поиска.

После публикации алгоритмов HLBVH2 [6] и GridBvh [4] на их основе была опубликована новая улучшающая работа [14], в которой представлен самый эффективный на сегодняшний день алгоритм построения и обновления BVH, исполняющийся на GPU. Новый алгоритм объединяет модифицированный алгоритм HLBVH2 (глава 5), метод разделения треугольников на частичные (глава 4), а также метод поворотов узлов внутри поддеревьев для улучшения качества существующего дерева BVH (цель идеи схожа с идеей, разработанной в главе 2).

Глава 6

Алгоритм поиска пересечений лучей в массивных сценах на графическом процессоре с ограниченным размером памяти

В данной главе описан разработанный автором эффективный алгоритм трассировки лучей для больших сцен на графическом процессоре. Используются специализированные для конкретной задачи трассировки лучей методы хранения, пересылки и кэширования данных, из которых состоит сцена. Алгоритм поиска пересечений лучей и сцены использует сложную ускоряющую структуру, которая логически организована в виде множества равных страниц данных, которые загружаются в память графического процессора по запросу. В качестве дополнения используется быстрый метод сжатия с потерями геометрических данных с помощью квантования.

Предложенный алгоритм трассировки лучей для больших сцен, реализованный с применением технологии CUDA, применим для различных частных задач, в том числе и для стохастической Монте-Карло трассировки путей, которая порождает лучи с произвольными направлениями, что влечёт за собой необходимость обращаться к произвольным областям памяти в рамках поиска пересечения лучей и сцен для одного пикселя. Подобная ситуация могла бы затруднить эффективное массовое распараллеливание трассировки лучей на графическом процессоре, объём памяти которого строго ограничен. Однако в данной работе продемонстрированы результаты высокой кэш эффективности на графическом процессоре для сцен, объём которых значительно превышает объём памяти графического процессора. Этот алгоритм впервые был представлен на конференции SIGGRAPH 2011 [7], где была продемонстрирована интерактивная трассировка лучей с учётом глобального освещения для массивных сцен, состоящих из нескольких сотен миллионов полигонов, на графическом процессоре ноутбука. Алгоритмы подобного рода в английской литературе имеют приставку out-of-core (т.е. алгоритмы, спроектированные для обработки объёмов данных, не влезающих полностью за один раз в память

процессора). Новый алгоритм является out-of-core по отношению к памяти графического процессора, в данной главе он будет кратко называться ***OOC-IRT*** (out-of-core interactive ray tracing).

6.1. Описание алгоритма

Анализ производительности многих программ обработки графики показывает, что программы, исполняющиеся на GPU, могут превзойти в 10-20 раз по скорости исполнения аналогичные программы, исполняющиеся на CPU. Такое ускорение достигается благодаря большей вычислительной мощности современных графических процессоров (более 1 Tflops) и высокой пропускной способности чтения из внутренней памяти GPU (более 150 Гб/с.). Достигнуть подобного ускорения удаётся, если всё рабочее множество данных алгоритма (например, все текстуры и полигональные модели) полностью вмещается в память GPU. Задачи решения уравнения визуализации, использующие трассировку лучей, для каждого пикселя могут порождать запросы в любые области памяти, где хранится сцена. Поэтому возникает необходимость спроектировать алгоритм решения подобных задач для больших сцен в условиях ограниченного размера физической памяти GPU (см. схему работы с большим объёмом данных на рис. 6.1). Пусть все данные сцены (полигональная модель и текстуры) вмещаются в память хоста (память CPU). Размер кэша ограничен размером физической памяти GPU. На GPU выполняется параллельная многопоточная программа, обрабатывающая большое множество данных, запускается в каждой итерации цикла, называемого ***главным циклом обработки данных*** (также см. рис. 6.1):

```
void out_of_core_data_processing()
{
  int new_data = 1
  while(new_data) {
    // Параллельный многопоточный обработчик данных на GPU
    request_and_process_data_kernel(data)
    // Доставка страниц данных в кэш GPU
    new_data = swap_requested_pages(gpu_data_manager)
  }
}
```

Листинг 6.1: Цикл обработки большого объёма данных.

Если для некоторого потока в кэше GPU нет страниц, соответствующих запрошенным данным, то вычисления для подзадачи, соответствующей этому потоку, откладываются до следующей итерации цикла (см. Листинг 6.1). В той же итерации цикла другая параллельная программа, исполняющаяся на GPU, swap_requested_pages(), производит операции, связанные с политикой пересылки запрошенных страниц данных и обновления соответствующих таблиц страниц. На последующих итерациях цикла запрошенные ранее данные могут находиться в кэше GPU, поэтому возобновление отложенных вычислений для некоторых потоков становится возможным. Процесс запроса, обработки данных и пересылки соответствующих страниц повторяется до тех пор, пока не перестанут появляться новые запросы страниц.

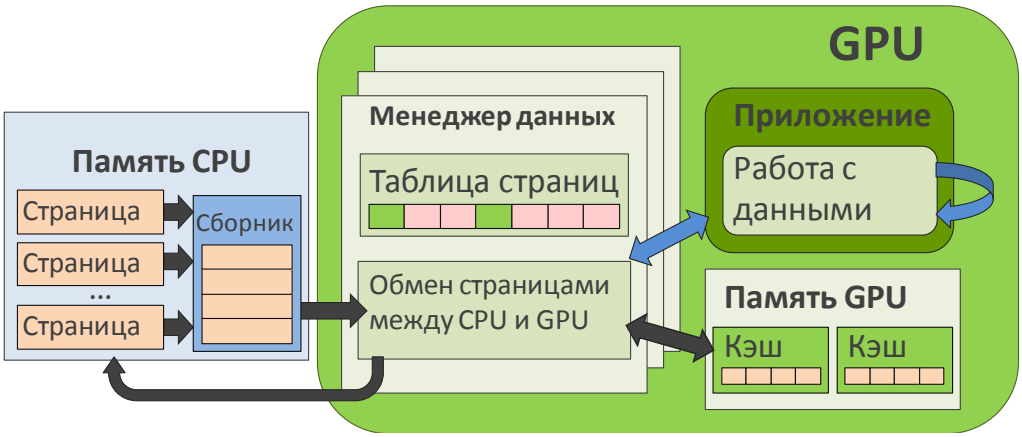


Рис. 6.1: *Менеджер данных, не влезающих в память графического процессора (GPU), однако влезающих в память центрального процессора (CPU). Все данные организованы в виде набора страниц. Приложение, исполняющееся на GPU, обрабатывает данные в цикле и порождает запросы к одному или нескольким менеджерам данных. Если область данных не находится в кэше GPU, то обработчик данных для конкретного потока откладывается на будущее, а менеджер данных доставляет соответствующие этой области страницы данных в кэш, вытесняя другие страницы. CPU получает массив индексов запрошенных страниц, затем все соответствующие страницы собираются в один непрерывный массив для единоразовой пересылки на GPU одного блока данных, состоящего из многих страниц. Внутри GPU поступившие страницы распределяются в необходимые области кэша и для всех ранее прерванных обработчиков данных возобновляется параллельная обработка данных. Данные о местонахождении страниц записываются в специальной таблице страниц и обновляются во время движения страниц. Цикл обработки данных на GPU прекращает исполнение, когда новые страницы не запрашиваются через менеджер данных.*

6.2. Менеджер данных GPU

Главными компонентами менеджера данных являются таблица дескрипторов страниц (таблица страниц), процесс запроса блоков данных из менеджера данных, процесс пересылки из внешней памяти (памяти CPU) и механизм замещения страниц (если кэш данных заполнен).

Дескриптор страницы. Массив данных разделяется на страницы равного размера. Каждой странице ставится в соответствие 32-битный дескриптор (*PageDesc*), которой содержит следующие битовые поля:

[0..23] биты – PageBaseAddress, базовый адрес страницы;

[24..28] биты – TimeUsed, счётчик времени использования страницы;

[29] бит – WasRead, флаг установлен в 1, если производились вычисления с данными страницы во время работы основного цикла обработки данных;

[30] бит – Requested, флаг установлен в 1, если страница запрошена менеджером данных;

[31] бит – OOC, флаг установлен в 1, если страница не находится в кэше GPU. Здесь и далее в тексте страницы, находящиеся в области памяти GPU, выделенной для кэша, называются ***IC-страницами*** (in-core); все остальные страницы (которые не находятся в данный момент в памяти GPU) называются ***OOC-страницами*** (out-of-core).

Таблица страниц состоит из двух массивов: *pageDesc* – массив 32-битных дескрипторов страниц (хранящихся в памяти GPU), *pageHostAddr* – вспомогательный массив указателей на блоки памяти CPU, в которых хранятся страницы. Массив индексов страниц (IC-страниц и OOC-страниц) находится в массиве pageIDs. Этот массив всегда отсортирован таким образом, что первые numCachedPages его элементов всегда указывают на IC-страницы, а остальные элементы указывают на OOC-страницы.

Запросы данных. Когда один из потоков параллельной программы обработки данных *request_and_process_data_kernel()* (см. листинг 6.1) запрашивает доступ к данным, то для начала в дескрипторе соответствующей страницы флаг *Requested* устанавливается в 1 с использованием атомической инструкции *atomicOr*. Затем с помощью атомической инструкции *atomicAdd* увеличивается счётчик *TimeUsed*, который хранит количество обращений к странице в течение выполнения функции *request_and_process_data_kernel()* и помогает в определении порядка замещения страниц при необходимости. 5-битное поле *TimeUsed* обновляется для всех IC-страниц. Если страница была запрошена (*Requested=1*), то *TimeUsed* увеличивается на 1, и новое значение перед сохранением в дескрипторе страницы ограничивается значением 2^5*-1* для предотвращения переполнения.

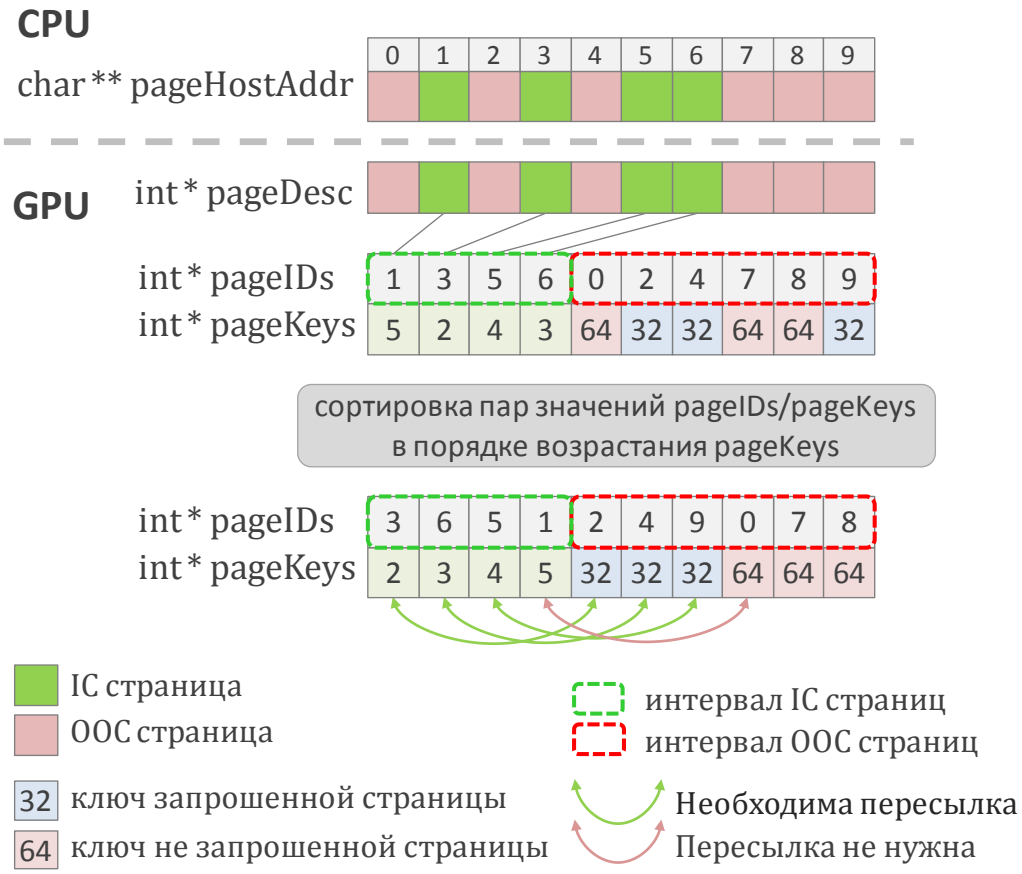


Рис. 6.2: *Алгоритм замещения страниц. Наименее часто используемые IC-страницы, находящиеся в кэше, заменяются на наиболее часто запрашиваемые OOC-страницы, находящиеся не в кэше.*

Также используется обнуление поля *TimeUsed* для всех страниц во время каждого *n-го* вызова (например, во время каждого 10-го вызова) функции управления пересылками страниц *swap_requested_pages()*. Подобное периодическое обнуление придаёт больший вес (меньше шанса быть вытесненными из кэша) для тех страниц, которые являются наиболее популярными и используемыми в течение последних итераций основного цикла обработки данных. Если значения *TimeUsed* для всех страниц равны нулю, то процесс стабильной параллельной радикс-сортировки (выполняющейся на GPU) [74] сохраняет порядок ранее отсортированных IC-страниц. Стабильная сортировка не меняет местами элементы с равными ключами.

Замещение страниц. Когда кэш GPU полон, возникает необходимость удалить некоторые страницы, находящиеся в кэше, и заменить их на запрашиваемые страницы, находящиеся в памяти CPU. Используется политика замещения в первую очередь тех страниц, находящихся в кэше, которые наименее часто используются (Least Frequently Used, LFU). Для определения тех страниц, которые необходимо вытеснить из кэша GPU, значения поля *TimeUsed*, хранящегося в 32-битном дескрипторе IC-страницы, записываются во временный массив *pageKeys* (см. рис. 6.2). Для OOC-страниц, которые были запрошены менеджером данных для загрузки на GPU, соответсвующие значения в массиве pageKeys устанавливаются равными 32, или если OOC-страница запрошена не была, то устанавливается значение 64. Значения 32 и 64 выбраны произвольно, они должны быть больше 5-битных чисел (макс. 31) и значение pageKeys для незапрошенных OOC-страниц должно быть самым большим.

Затем индексы *pageIDs* элементов таблицы страниц сортируются в порядке возрастания соответствующих им ключей *pageKeys* с применением стабильной радикс-сортировки [74]. В итоге, в отсортированном массиве *pageIDs* в интервале *[0..numCachedPages)* оказываются индексы IC-страниц, в интервале *[numCachedPages..numCachedPages+k)* оказываются индексы всех *k* запрошенных менеджером данных OOC-страниц, и в конце отсортированного массива оказываются индексы всех незапрошенных OOC-страниц. Производится корректировка *k*: если *k* > *numCachedPages*, то *k* = *numCachedPages*. Все элементы *pageIDs* в интервале *[0..k)* заменяются на все элементы в интервале

[numCachedPages..numCachedPages+k). Подобная замена индексов в отсортированном массиве *pageIDs* по сути реализует вытесенение наименее часто используемых IC-страниц (см. рис. 6.2).

Пересылки страниц. Выполнение многих операций пересылки страниц небольшого размера из памяти хоста (CPU) в память GPU не позволяет достичь пиковой пропускной способности шины PCI-Express. Поэтому множество запрошенных менеджером данных страниц на стороне CPU собираются в одном сборном массиве (например, размера 1 Мб), вмещающем несколько страниц (см. рис. 6.1). Для каждого такого сборного массива выполняется одна операция пересылки данных из CPU в память GPU. После осуществления пересылки сборного массива на стороне GPU содержащиеся в нём страницы распреедляются в необходимые позиции кэша GPU. Подобный способ пересылки сборных массивов страниц позволяет пересылать данные в память GPU на пиковой пропускной способности шины PCI-Express.

Например, при использовании для соединения GPU шины PCI Express x16 gen 2, пересылка 1 Мб данных в память GPU осуществляется со скоростью 5.6 Гб/с (тесты пропускной способности пересылок доступны в CUDA SDK [79]). Хотя, формирование сборного массива страниц также требует производить копирование запрошенных страниц в сборный массив, – это может осуществляться на различной скорости, в зависимости от используемых планок физической памяти RAM, например на скорости 4 Гб/с. Поэтому суммарная скорость пересылки страниц (включая время сборки страниц, пересылку сборного массива и его распределение в необходимые позиции кэша) может варьироваться в диапазоне 3-3.5 Гб/с. Тем не менее пересылка данных с использованием сборного массива в 2-10 раз быстрее осуществления многих пересылок одиночных страниц, если их размеры менее 0.5-1 Мб.

Использование менеджера данных. Все компоненты менеджера данных GPU выполняются на GPU (кроме пересылок сборного массива страниц). Применяется эффективная параллельная радикс-сортировка [74] и быстрый расчёт префиксной суммы [79]. Эти две функции являются необходимыми для быстрого

обновления таблицы страниц, выполняются на GPU и время их выполнения является незначительным (менее 1%) по сравнению с временем выполнения пересылок страниц.

Наилучший размер страницы выбирается эмпирически в зависимости от задачи. Есть два фактора, которые необходимо принять во внимание: 1) размер таблицы страниц (массив индексов страниц и массив дескрипторов) и 2) общая скорость обработки данных, включая время пересылок на GPU, как функция размера страниц. В настоящей работе менеджеры данных GPU используются для решения следующих задач: 1) поиск пересечения лучей и полигональной геометрии сцены, не вмещающейся полностью в память GPU; 2) осуществления доступа на GPU к атрибутам поверхностей, составляющих сцену; 3) текстурная фильтрация. Метод разделения данных на страницы и выбор размера страницы отличается в каждой из этих 3 задач.

Поле *WasRead* в дескрипторе старинц помогает избегать повторных вычислений внутри страниц, которые остаются в кэше на протяжении нескольких итераций основного цикла обработки данных. Например, задачей параллельной и реентрабельной программы *request_and_process_kernel()* (см. листинг 6.1) может быть поиск минимального значения некоторой функции в каждом потоке (например, поиск ближайшей к основанию точки пересечения луча и геометрии). Флаг *WasRead* показывает производился ли локальный поиск минимума внутри страницы для всех потоков, и если он производился, то запрос на подгрузку подобной страницы или поиск минимума в ней отменяется. Хотя не во всех задачах возможно использовать подобный флаг.

Ограничения. В настоящий момент описана простейшая иерархия памяти “память CPU -> кэш в памяти GPU”. Поэтому размеры данных, обрабатываемых GPU ограничены размером памяти CPU. При возникновении необходимости поддерживать размеры обрабатываемых данных, превышающих размер памяти CPU, потребуется создать уровень менеджера данных, который осуществлял бы пересылки данных между CPU и дисковым пространством или сетью.

Для малых размеров страниц, таблица дескрипторов страниц может стать большой. Одним из решений этой проблемы является увеличение размера страницы, что повлекло бы уменьшение числа страниц, необходимых для учёта.

Описанный в данной работе кэш данных является пригодным только для чтения. Подобное ограничение упрощает алгоритм, т.к. не возникает необходимости синхронизации после модификации данных страниц в многопоточной среде исполнения. Поддержка записи потребовала бы добавления дополнительных полей в таблицу дескрипторов для отслеживания модификаций и пересылки соответствующих страниц обратно в память CPU во время вытеснения из кэша GPU.

6.3. Out-of-core трассировка лучей на графическом процессоре

В этой секции описан алгоритм трассировки лучей на GPU в больших сценах, размер которых превышает размер физической памяти GPU. Во-первых, необходимо использовать ускоряющую структуру данных для поиска пересечений лучей и сцены. Во-вторых, необходимо для каждого луча производить обход ускоряющей структуры и проверять пересечения с полигонами, зная, что геометрия может не находиться в кэше GPU. В-третьих, расчёт глобального освещения может потребовать доступ к произвольному объёму данных (текстуры и атрибуты вершин полигонов), который может превзойти размер геометрической модели сцены.

Сцена загружается из файлов. Объекты сцены могут быть анимированы или деформированы. Поддерживаются клонирующие объекты, которые содержат ссылку на область памяти, содержащую геометрию базового объекта, а также матрицу $M_{instance}$, содержащую информацию о смещении, ориентации и масштабировании относительно базового объекта в мировых координатах. Изображение сцены может быть рассчитано с помощью Монте-Карло трассировки лучей [62], трассировки лучей Кука [28], трассировки лучей Уиттеда [119] и т.д.

6.3.1. Ускоряющая структура

Эффективная трассировка лучей требует использовать ускоряющую структуру для уменьшения числа пересечений лучей и объектов сцены. В настоящей работе спроектирована 3-уровневая ускоряющая структура, концептуально похожая на разработанную в системе PantaRay [81].

На рис. 6.3 представлен процесс обработки геометрического представления объектов сцены и построения ускоряющей структуры. Геометрическое представление объекта конвертируется в модель, содержащую квады, где ***квад*** – это малая полигональная структура из 2 треугольников с одним общим ребром, один квад имеет 4 вершины. Объект, содержаший большое количество квадов, разбивается на несколько ***SAH-ветвей***, где каждая SAH-ветвь является группой, имеющей ограниченное количество квадов (не более *NB*), отделённой от других групп квадов охватывающей оболочкой. Далее, в пространстве каждой SAH-ветви производится сжатие с помощью квантования координат (см. секцию 6.3.1.2). Каждая SAH-ветвь разделяется на более мелкие блоки, содержащие равное количество *NP* квадов. Затем строится 3-уровневая ускоряющая структура:

BVH нижнего уровня (страница): локальная ускоряющая структура BVH строится для каждого блока из NP квадов для всех объектов сцены.

BVH среднего уровня: оболочки страниц, сгенерированных для всех объектов, помещаются в общую 3D сетку, являющейся ***3D сеткой верхнего уровня***, определённую внутри пространства, ограниченного оболочкой сцены. Внутри каждой непустой ячейки этой сетки строится ускоряющая структура BVH, организующая все оболочки страниц, частично пересекающие область этой ячейки.

Для обработки большого количества объектов, которые могут содержать большое количество квадов, каждый объект считывается из файла отдельно, для него генерируется множество страниц, которые сохраняются в файле. Память, выделенная для объекта, освобождается и начинается работа с новым объектом. При этом оболочки страниц всех объектов занимают компактное место в памяти и используются для построения сетки верхнего уровня и BVH среднего уровня.

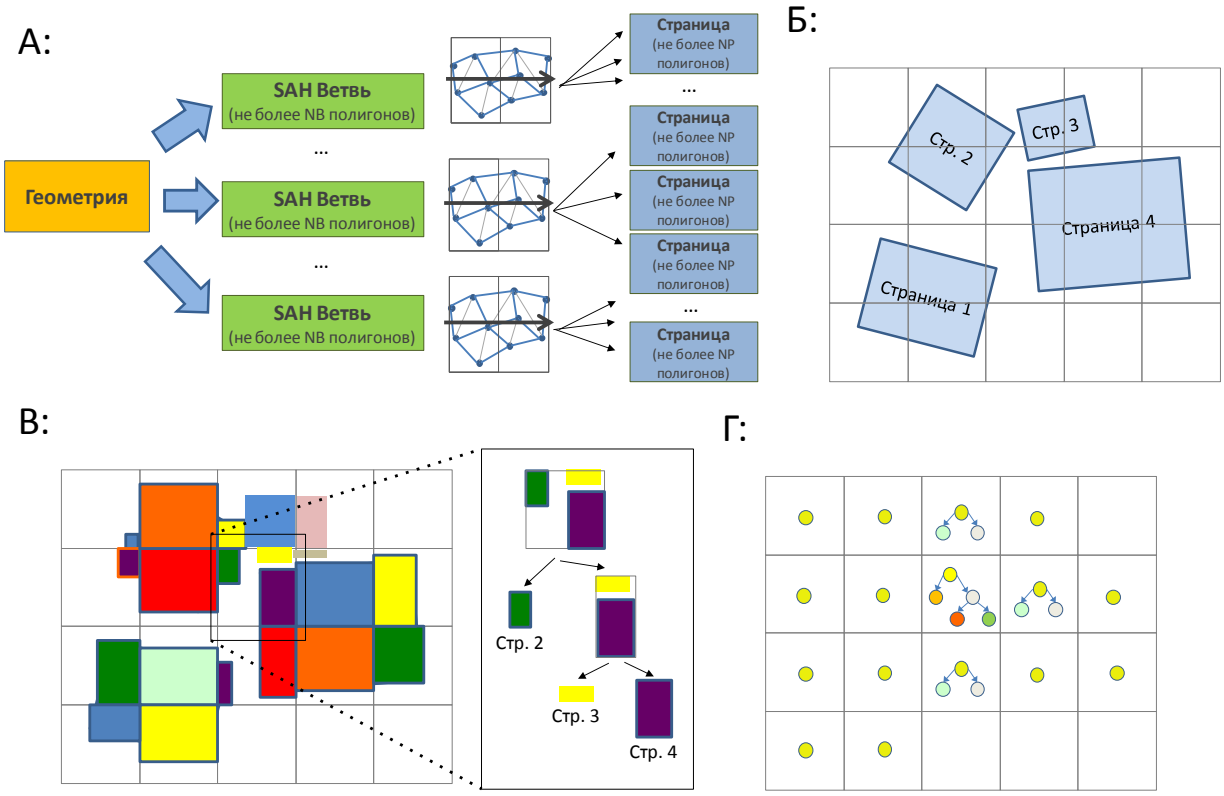


Рис. 6.3: *Формирование ускоряющей структуры данных.* ***А:*** *геометрический объект сцены, если он имеет много полигонов, то разделятся на несколько SAH ветвей (частей объекта с ограниченным количеством полигонов, не более NB, локализованных от пустого пространства с помощью SAH функции (формула 1.3)). Внутри каждой SAH ветви каждому полигону ставится в соответствие мортон-код, и полигоны сортируются в порядке возрастания их мортон-кодов. В каждой SAH ветви все вершины полигонов квантуются в 16-битной 3D сетке, определённой внутри оболочки SAH ветви. В каждой SAH ветви массив отсортированных полигонов тривиально разделяются на несколько равных блоков, состоящих из NP полигонов каждый.* ***Б:*** *Для каждого блока строится BVH, организующее полигоны блока в иерархию – подобная структура здесь и далее называется BVH нижнего уровня (или страницей). Для каждой страницы вычисляется OBB (ориентированная охватывающая оболочка). Все OBB оболочки, представляющие свои страницы, собранные для всех объектов сцены, заключаются внутри общей 3D сетки сцены, являющейся сеткой верхнего уровня.* ***В:*** *Производится вычисление AABB-оболочек пересечений OBB каждой страницы и ячеек сетки.* ***Г:*** *Внутри каждой ячейки сетки верхнего уровня строится BVH среднего уровня, организующее в иерархию все оболочки страниц внутри этой ячейки.*

Каждый объект сцены независимо разделяется на группы квадов, называемые ***SAH-ветвями***, где каждая группа содержит не более *NB* квадов (например, *NB = 64K* квадов). Подобное разбиение объекта на группы квадов можно произвести с помощью построения BVH для объекта, где рекурсивный процесс разделения узла на потомки останавливается, если узел содержит не более *NB* квадов. Целью построения этого BVH является разделение множества всех квадов объекта на нескольких групп (каждый лист BVH содержит группу не более *NB* квадов). После получения списка групп это дерево BVH не используется в дальнейшем. Каждая созданная группа квадов должна быть хорошо отделена от других подобных групп, если в процессе построения BVH использовалось вычисление SAH-функции (формула 1.3, см. главы 2-5). Хотя на практике при использовании подобного метода генерации SAH-ветвей, каждая SAH-ветвь содержит значительно меньшее число квадов, чем *NB*. Для большого числа сцен различного типа среднее число квадов внутри SAH-ветви составляет 60-70% от *NB*. Если использовать SAH-ветвь как основу для создания одной страницы для представления сцены в виртуальной памяти, то будет пустовать 30-40% памяти, выделенной для множества страниц, представляющих сцену, состоящих из не более *NB* квадов. Вместо этого каждая SAH-ветвь дополнительно разбивается на более мелкие компоненты с применением другого метода разбиения.

Внутри каждой SAH-ветви для центральной точки каждого квада рассчитывается 30-битный мортон-код (см. главу 5) и все квады внутри SAH-ветви сортируются в порядке возрастания их мортон-кодов. Массив отсоротированных квадов разбивается на множество блоков равного размера, содержащих *NP* квадов (например, *NP = 4K* квадов). Данный способ разбиения не гарантирует оптимального пространственного разбиения (оболочки блоков квадов могут пересекаться), однако геометрическая когерентность квадов внутри подобных блоков присутствует. Также в результате подобного разбиения среднее число квадов внутри каждого блока составляет 99% от *NP*. Поэтому использование блока из *NP* квадов в качестве основания для создания страницы является эффективным с точки зрения экономного использования памяти.

BVH нижнего уровня (страница). Для каждого блока из NP квадов создаётся BVH, организующее квады этого блока в иерархию. В качестве охватывающих оболочек используются AABB. Для каждого подобного блока из *NP* страниц и соответветствующего BVH выделяется в памяти 1 страница. Для каждой страницы координаты квадов и оболочек узлов BVH сохраняются в локальных координатах. Для каждой страницы выбирается AABB оболочка корневого узла BVH и трансформируется в ориентированную охватывающую оболочку OBB (имеющую координаты в мировой системе координат) с помощью матрицы трансформации $M_{instance}$, соответсвующей объекту сцены, внутри которого создана эта страница. Каждая оболочка OBB имеет ссылку на оригинальную геометрию квадов, координаты которой хранятся в локальных координатах, и матрицу трансформации из мировых координат сцены в локальное пространство страницы. Оболочки OBB каждой страницы являются элементами для построения последующих уровней ускоряющей структуры. Содержание страниц не хранится постоянно в памяти GPU, однако их OBB оболочки хранятся.

BVH среднего уровня. Оболочки OBB для всех страниц, сгенерированных для всех объектов сцены, заключаются внутри 3D сетки, созданной внутри пространства сцены. Разрешение сетки выбирается от 16^3 до 128^3. Для каждой ячейки сетки производится поиск всех OBB, которые пересекают её. Затем вычисляется AABB-оболочка, обозначаемая $isectAABB_i$, охватывающая область пересечения между OBB и AABB-оболочкой ячейки. Каждая непустая ячейка сетки может содержать несколько подобных оболочек $isectAABB_i$. Поэтому для всех таких оболочек создаётся BVH (среднего уровня), организующее в иерархию подобные оболчки пересечений.

3D сетка верхнего уровня. Каждая непустая ячейка сетки содержит единственное дерево BVH среднего уровня. Сетка верхнего уровня, а также BVH среднего уровня постоянно находятся в памяти GPU. Содержание страниц, включая блок квадов и организующее их дерево BVH, в кэше GPU появляется временно, по запросу любого из лучей, для произведения тестов пересечений со всеми лучами.

6.3.2. Сжатие данных

Сжатие используется для увеличения количества полигонов, которые могут находиться одновременно в GPU кэше, и производится на этапе построения 3-уровенвой ускоряющей структуры (подробности представлены в статье [5]).

Входная геометрическая модель, состоящая из множества треугольников, конвертируется в модель, состоящую из квадов, где ***квад*** – это полигональная структура, состоящая из 2 треугольников с общим ребром и 4 вершинами. В процессе конвертации каждый треугольник соединяется с соседним треугольником, если у них есть общее ребро, и они ссылаются на один материал. Если для *i-го* треугольника существует несколько кандидатов, то для объединения с ним используется тот треугольник, с которым в результате получается наименьшая длина сторон квада. Квад содержит в 1.5 раза меньше информации о связях по сравнению с треугольниками: 1 квад имеет 4 ссылки на координаты вершин, тогда как 2 треугольника имеют 6 ссылок.

Предполагается, что объём содержащий квады страницы существенно меньше объёма, содержащего весь объект сцены. Все вершины квада встраиваются в 16-битный куб (координаты изменяются от -2^{15} до 2^{15} в каждом измерении), определённый внутри охватывающей оболочки страницы, и квантуются (т.е. координаты каждой вершины записываются в 3x16=48 бит). Подобная операция производится с помощью умножения координат вершин квада на матрицу $\boldsymbol{M_{encode}}$ (матрица смещения и масштабирования). Для каждого квада все ссылки на координаты вершин заменяются на квантованные координаты, и тогда запись квада занимает 24 байта.

Лучи, для которых необходимо найти пересечение со сценой, задаются в мировых координатах. Во время прохода по ускоряющей структуре данных лучи трансформируются в локальное пространство объекта (с помощью умножения на обратную матрицу объекта $M_{instance}$) для тестов пересечения с геометрией, заданной в локальных координатах.

Поэтому во время построения ускоряющей структуры каждая матрица $M_{instance}$ модифицируется: $M_{instance} = M_{instance} * inverse(M_{encode})$. Тогда для лучей, умноженных на модифицированную матрицу $M_{instance}$, будут даваться правильные результаты при поиске пересечения с геометрией, заданной в локальных координатах и записанной сжато. При этом накладных расходов для декомпрессии геометрии не добавляется, т.к. лучи всё равно должны быть умножены на матрицу.

Подобный алгоритм сжатия с потерями испытывался со сценами различного типа, от CAD-сцен до результатов лазерного сканирования. При этом артефактов и появляющихся дыр между квадами замечено не было. Используется предположение, что охватывающая оболочка страницы, в пространстве которой производится квантование, намного меньше оболочки всего объекта, содержащего страницу. Однако для сцен, в которых оболочки страниц могут иметь большой размер, такое предположение может привести к наличию дыр на стыке страниц.

6.3.3. Поиск пересечений лучей и геометрии сцены

Для осуществления быстрого поиска ближайшей точки пересечения луча и геометрии сцены необходимо использовать ускоряющую структуру данных. Во-первых, верхние два уровня ускоряющей структуры постоянно находятся в памяти GPU, тогда как нижний уровень, состоящий из страниц, может находиться в кэше GPU лишь временно. Поэтому обход всех лучей в узлах ускоряющей структуры осуществляется в несколько итераций, во время которых необходимые страницы запрашиваются менеджером данных (см. листинг 6.1).

В многопоточной программе в каждом потоке осуществляется поиск ближайшего пересечения одного луча и геометрии сцены. Каждый луч начинает производить поиск пересечения внутри 3D сетки верхнего уровня (см. методы обхода узлов сетки [47]). Программа поиска пересечений для всех лучей реентрабельна, т.е. запускается много раз, пока для всех лучей ближайшие пересечения не будут найдены. Поэтому необходимо для каждого луча запоминать ячейку сетки верхнего уровня, с которой для луча начнётся поиск пересечений в следующей итерации/запуске программы поиска пересечений. Для этого множеству лучей ставится в соответствие массив *restart.cell*.

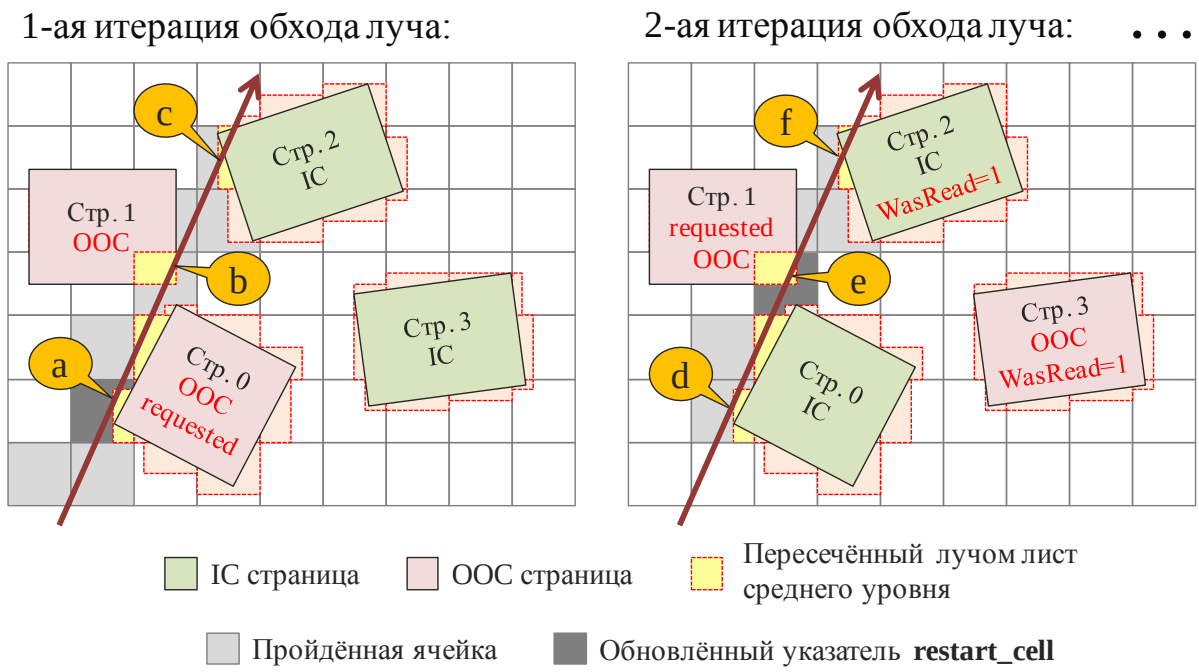


Рис. 6.4: *Многопроходный алгоритм поиска пересечения лучей и сцены. Изображён один отдельно взятый луч и 2 итерации алгоритма. Для луча производится обход всех ячеек сетки верхнего уровня, через которые он проходит, начиная с ячейки на которую указывает ray.restart. Первая итерация: **(a)** Если вдоль направления луча встречается первая OOC страница, то она помечается как запрошенная (requested=1) и указатель ray.restart изменяется на указатель на ячейку, содержащую эту страницу. После этого обход ячеек вдоль направления луча продолжается. **(b)** Вторая и последующие OOC-страницы, встречающиеся вдоль направления луча в текущей итерации, не помечаются запросом, и обход ячеек продолжается. **(c)** Если встречается страница, находящаяся в кэше, то производится поиск пересечения луча и геометрии внутри страницы с помощью обхода BVH нижнего уровня. Если пересечение есть, то информация о нём для луча обновляется. Вторая итерация: **(d)** обход начинается с ячейки, указанной в ray.restart, содержащую ранее запрошенную в менеджере данных Стр.0, в которой на предыдущей итерации не производился поиск пересечения луча и геометрии, а на текущей итерации производится. При отсутствии пересечения обход ячеек вдоль направления луча продолжается, и **(e)** производится запрос первой OOC-страницы для луча в текущей второй итерации. Далее обход луча продолжается, и если встречается **(f)** страница, внутри которой ранее производился поиск пересечений любых лучей и геометрии (если WasRead=1), то в текущей итерации подобный поиск внутри страницы не производится.*

Перед первой итерацией поиска пересечений производится инициализация:

- В массиве *restart.cell* каждый *i-ый* элемент инициализируется указателем на первую ячейку, которую *i-ый* луч пересекает в сетке верхнего уровня.

- Устанавливается флаг *WasRead = 0* для всех дескрипторов страниц в таблице страниц.

Запускается итеративная процедура поиска пересечений (см. листинг 6.1), где функция *request_and_process_data_kernel()* производит обход ускоряющей структуры параллельно для всех лучей, во время обхода некоторые страницы запрашивается через менеджер данных; а функция *swap_requested_pages()* доставляет запрошенные страницы в кэш GPU. Пример двух итераций изображён на рис. 6.4.

Обход в ускоряющей структуре. Во время каждой итерации для луча осуществляется поиск пересечений внутри всех IC-страниц, которые в данной итерации находятся в кэше GPU – даже тех страниц, которые находятся дальше запрошенных OOC-страниц. Поиск пересечения с IC-страницами производится «на всякий случай»: если пересечение с любой IC-страницей будет найдено, то обновится ближайшая точка пересечения для луча, и тогда эти страницы не нужно будет в последующих итерациях запрашивать на загрузку в кэш GPU.

Обход верхнего уровня ускоряющей структуры для каждого луча начинается с ячейки, указатель на который сохранён в *restart.cell*. Обход ячеек производится вдоль направления луча не дальше сохранённой ближайшей к основанию точки пересечения луча и геометрии. Также производится проверка на предмет того, запрашивалась ли для луча в текущей итерации какая-либо OOC-страница. Если ещё не запрашивалась, то запрос производится, если луч пересекает охватывающую оболочку OOC-страницы, которая хранится в GPU памяти постоянно. Но если какие-либо OOC-страницы уже запрашивались для луча в текущей итерации, то обход ячеек сетки продолжается без дальнейших запросов OOC-страниц, тесты пересечения производятся только с геометрией, содержащейся в IC-страницах.

Обход BVH среднего уровня. Если в процессе обхода появляется не пустая ячейка, то необходимо обойти содержащееся в ней дерево BVH среднего уровня.

Обход BVH нижнего уровня. Для каждого листа BVH среднего уровня, охватывающую оболочку которого пересекает луч, производится поиск пересечения луча и геометрии страницы, на которую ссылается лист:

Если в этой странице ранее производился поиск пересечений (флаг её дескриптора *WasRead = 1*), то в этой итерации поиск не осуществляется.

Если это OOC-страница, и в течение текущей итерации для луча не запрашивались другие OOC-страницы, то эта страница запрашивается в менеджере данных GPU на загрузку в кэш (установить флаг *Requested = 1* в дескрипторе страницы), а также указатель $restart.cell_i$, соответствующий лучу, изменяется на эту страницу. Если в текущей итерации для луча запрашивались другие OOC-страницы, то запрос этой страницы пока откладывается.

Если это IC-страница (т.е. соответствующая геометрия находятся в данный момент в кэше), то луч трансформируется матрицей $M_{instance}$ в локальное и закодированное пространство, в котором хранятся координаты геометрии внутри страницы (см. секцию 6.3.2). После этого для луча осуществляется обход дерева BVH нижнего уровня, и при необходимости обновляется информация о ближайшей точке пересечения луча.

После окончания выполнения параллельного обхода 3-уровневой ускоряющей лучей для всех лучей (после завершения выполнения функции *request_and_process_data_kernel()*, см. листинг 6.1) устанавливается флаг *WasRead=1* для всех страниц, которые в данной итерации были в кэше GPU. Запрошенные в данной итерации OOC-страницы загружаются в кэш GPU. В следующей итерации будет производиться поиск лучей и геометрии, содержащейся в этих страницах. Этот процесс (поиск пересечений, запрос, загрузка) продолжается до тех пор, пока не перестанут запрашиваться страницы на загрузку в кэш GPU.

Оптимизация теневых лучей для обхода ускоряющей структуры. Для теневых лучей нет необходимости искать ближайшее к основанию пересечение луча и геометрии, т.к. для таких лучей необходимо проверить, блокируется ли луч на определённом интервале до источника света каким-либо геометрическим объектом или нет.

Поэтому в течение одной итерации обхода луча в ускоряющей структуре для начала проверяются тесты пересечения этого луча и геометрии всех IC-страниц, встречающихся на пути обхода луча. Если пересечение для луча найдено, то для этого луча поиск заканчивается. Если среди IC-страниц пересечения не найдено, то запрашивается первая OOC-страница, встречающаяся во время обхода луча, для загрузки в GPU кэш.

Обоснование использования 3-уровневой ускоряющей структуры. Использование дерева BVH для ускорения трассировки лучей на GPU в сценах, полностью вмещающихся в память, является лучшим решением, обеспечивает самую быструю трассировку лучей по сравнению с сетками или kd-tree [12][60] [123]. Для деревьев BVH расход памяти ниже, чем у других структур. Скорость построения BVH выше, чем у kd-tree и может быть ниже, чем у сетки. Также деревья BVH обеспечивают самую высокую скорость трассировки лучей. Поэтому в качестве ускоряющей структуры для геометрии, содержащейся внутри каждой страницы, выбрано дерево BVH.

Однако если бы для организации оболочек всех страниц использовалось одно дерево BVH верхнего уровня, постоянно находящееся в памяти GPU, то для луча было бы необходимо обходить это дерево каждый раз заново при запуске новой итерации цикла обработки данных (см. листинг 6.1). Такое дерево BVH верхнего уровня может быть достаточно глубоким особенно в сценах, содержащих большое количество клонированных объектов. Обход данного дерева заново во время каждой итерации влечёт за собой значительное увеличение вычислительных расходов. Существует возможность не обходить это дерево заново во время запуска каждой новой итерации, если сохранять стек обхода для каждого луча и использовать его на следующих итерациях. Однако этот подход является очень затратным, если па-

раллельно производится поиск пересечений для 1 миллиона лучей (типично для эффективной загрузки GPU) и если при этом дерево имеет большую глубину.

Использование сетки верхнего уровня позволяет на новой итерации продолжать обход для луча с места остановки, сохранённого на предыдущей итерации. Это позволяет сократить число повторных вычислений на новых итерациях. Использование BVH среднего уровня позволяет решить проблему неоднородного количества страниц, попадающих в ячейки сетки.

Разрешение сетки верхнего уровня выбирается эмпирически для каждой сцены. Использование большого разрешения повлечёт за собой увеличенный расход памяти. Использование маленького разрешения повлечёт за собой большее количество повторных обходов луча в BVH среднего уровня, для которых во время прерывания обхода луча стек не сохраняется для использования на будущих итерациях. Оптимальным разрешением сетки является 16^3-64^3, что даёт относительно небольшое количество накладных расходов, связанных с повторными обходами BVH среднего уровня. Повторного обхода BVH нижнего уровня не происходит, т.к. дескрипторы соответствующих страниц помечаются флагом *WasRead = 1* после попадания в кэш GPU.

6.4. Чтение больших объёмов атрибутов поверхностей

После того как для всех лучей найдены ближайшие точки пересечения с геометрией сцены, каждый луч содержит информацию о ближайшем кваде, который он пересекает: идентификатор объекта сцены *object_id* и идентификатор квада *prim_id* внутри этого объекта.

В процессе расчёта глобального освещения в каждой точке пересечения луча и геометрической поверхности необходим доступ к следующим атрибутам поверхности в точке: нормаль, индекс материала, текстурные координаты, а также другие атрибуты. Объём данных, содержащих указанные атрибуты поверхностей, также как и геометрические координаты может значительно превышать ёмкость памяти графического процессора.

Один экземпляр менеджера данных GPU создаётся для считывания всех атрибутов поверхности для любой точки по адресу *(object_id, prim_id)*. Каждый объект сцены может иметь различное число атрибутов (например, нормали, текстурные координаты или другие данные). Создаётся локальная таблица страниц для нормалей для каждого объекта. Создаётся локальная таблица страниц для текстурных координат для каждого объекта. Точно также создаются локальные таблицы страниц для данных других типов для каждого объекта. После создания всех локальных таблиц они склеиваются в единую глобальную таблицу страниц. Дополнительный массив содержит смещения локальных таблиц внутри глобальной таблицы для каждого типа данных и для каждого объекта. В настоящей реализации используются размеры страниц и размеры типов данных равные степени двойки для того, чтобы избежать пересечения границы страниц одним элементом данных. Предварительная сортировка атрибутов квадов в порядке возрастания мортон-кодов, построенных на основе геометрических координат квадов, повышает когерентность хранящихся данных и компактность страниц атрибутов.

6.5. Фильтрация большого количества текстур на GPU

В современных сценах объёмы текстурных изображений, задающих цвет поверхности и свойства её отражения / поглощения света, могут достигать десятки, сотни гигабайт или даже несколько терабайт данных. Текстура часто представляет 2D изображение произвольного разрешения, натянутое на поверхность. Поверхность на границах содержит текстурные координаты, задающие способ натяжения текстуры на поверхность. Конкретная текстурная координата в точке поверхности вычисляется с помощью интерполяции граничных координат. Для каждой точки пересечения необходимо осуществить фильтрацию текстуры, т.е. считать значения текселей в окрестности точки и вычислить значение текстуры в точке с помощью выбранного метода интерполяции.

Для осуществления доступа к текстурным данным создаётся менеджер данных для текстур на GPU. Все текстуры разрезаются на тайлы (квадраты), где каждый тайл является страницей в контексте менеджера данных. За основу алгоритма чтения данных из текстур и загрузки по запросу взят алгоритм, представленный в работе [84], позволяющий подгружать данные из внешнего хранилища текстур в

оперативную память CPU для осуществления выборки текселя по конкретному адресу. Также используются уровни детализации текстур, позволяющие использовать версию оригинальной текстуры меньшего разрешения, если поверхность с этой текстурой находится далеко от основания луча.

Текстуры разрезаются на тайлы, например размером 16^2 или 32^2. Каждый тексель запрашивается по адресу вида *(object_id, tx, ty)*, где *tx, ty* – текстурные координаты в диапазоне [0..1], отображаемые на конкретное разрешение текстуры. Значения *tx, ty* отображаются на конкретный индекс тайла *tile_id* и смещение запрашиваемого текселя *(dx, dy)* внутри тайла для осуществления доступа к необходимому значению.

Для реализации доступа к текстурам на GPU используется итеративный процесс обработки данных (см. листинг 6.1). Во время каждой итерации, параллельная функция фильтрации *request_and_process_data_kernel()* производит чтение данных текстур для множества запрашивающих потоков (соответствующих точкам пересечения лучей). Если страницы, соответствующие запрашиваемым текстурным данным, находятся в кэше GPU, то считанные тексели сохраняются в локальных буферах потоков. Если необходимые страницы не находятся в кэше, то они подгружаются функцией *swap_requested_data()* и на последующих итерациях соответствующие текстурные данные продолжают заполнять локальные буферы потоков. Как только для каждого потока локальные буферы текстурных данных заполнены, то осуществляется фильтрация текстуры в конкретной точке, и полученные значения передаются в конвейер расчёта глобального освещения.

6.6. Параллельный поиск пересечений для большого массива лучей

Также как и в алгоритме, предложенном [24], в процессе расчёта глобального освещения создаются большие массивы лучей (до 16 млн. лучей) для осуществления трассировки лучей с амортизацией пересылок данных. Большой массив лучей, для которых необходимо найти пересечения, хранится в памяти CPU. Для каждого луча вычисляется 4-байтный ключ (см. [3]), содержащий квантованные координаты основания и направления луча. Все лучи сортируются в порядке возрастания их ключей. Затем отсортированный массив лучей разделяется на несколько

блоков, содержащих до 1 млн. лучей. Внутри одного такого блока благодаря сортировке содержатся когерентные лучи – лучи являются настолько когерентными, насколько позволяет метод генерации квантованного ключа [3]. Сортировка лучей осуществляется на GPU с помощью радикс-сортировки [74], после этого блоки когерентных лучей размером до 1 млн. лучей загружаются в память GPU с помощью функции CUDA zero-copy (загрузка осуществляется параллельно с вычислениями, см. [79]).

Подобный метод сортировки позволяет амортизировать стоимость пересылки страниц данных, т.к. когерентные лучи пересекаются с когерентными геометрическими объектами.

6.7. Результаты

6.7.1. Экспериментальные установки реализации алгоритма

Реализован алгоритм трассировки лучей на графическом процессоре для out-of-core сцен с применением технологии CUDA [79]. Показатели реализации алгоритма были измерены на настольном компьютере с CPU AMD Phenom 2 (4 ядра, 3GHz, 16 GB RAM) и GPU NVIDIA GTX 480 (1.5 GB памяти, 1.3 Tflops). Данные между CPU и GPU передаются по шине PCI Express x16 gen2 с реально измеренной пропускной способностью 5.6 GB/sec.

Детали реализации ускоряющей структуры. В качестве алгоритма построения ускоряющих структур BVH для нижнего и среднего уровней использовался алгоритм HLBVH2 (глава 5) с использованием разрезания длинных и тонких треугольников на частичные треугольники (глава 4). В качестве метода построения сетки верхнего уровня использовался метод построения сетки, описанный в главе 4. Разрешение для сетки выбрано 32^3. Дерево BVH среднего уровня в каждом листе содержит 1 примитив (ссылку на страницу, которая возможно не содержится в кэше графического процессора). Дерево BVH нижнего уровня в каждом листе содержит не более 15 квадов.

Использование сжатия геометрических данных внутри страницы позволяет в среднем использовать 29 байт для хранения одного квада (учитываются данные,

необходимые для хранения геометрических координат квадов и всей 3-уровневой ускоряющей структуры).

Менеджеры данных GPU. Было реализовано 3 менеджера данных:

1) Менеджер данных для поиска пересечений лучей и геометрии: размер кэша 700 MB, размер страницы 4K квадов (или 116 KB).
2) Менеджер данных для осуществления доступа к атрибутам поверхностей: размер кэша 50 MB, размер страницы 1 KB.
3) Менеджер данных для фильтрации текстур: размер кэша 100 MB, размер страницы 1 KB.

Оставшаяся память GPU используется для хранения лучей и временных данных необходимых для различных функций, таких как сортировка лучей и т.п.

Алгоритмы визуализации. Трассировка лучей больших сцен была использована в следующих алгоритмах визуализации:

1) Первичная видимость.
2) Прямое освещение от источников света.
3) Стохастическая Монте-Карло трассировка путей с 3 уровнями переотражений. В тестах, использующих Монте-Карло трассировку путей, для всех объектов сцен установлен диффузный тип рассеивания света на поверхности. В подобной ситуации в каждой точке пересечения луча создаются вторичные отражённые лучи с некогерентными направлениями, которые могут пересекаться с любой частью сцены и запрашивать доступ к любой ячейке памяти. Эта ситуация является стресс-тестом для менеджера данных GPU и для алгоритма поиска пересечений с out-of-core геометрией. В подобных ситуациях для больших сцен может быть сгенерировано множество запросов для пересылки страниц, которые не смогут все одновременно вместиться в кэше GPU.

Во всех 3 случаях использовалось разрешение генерируемого изображения сцены 1024x768, один стохастический луч на пиксель во время каждой итерации

прогрессивного обновления изображения. Т.е. в каждой итерации генерируется 1024x768 параллельных задач, реализующих алгоритм визуализации для своих отдельных пикселей. После завершения *i-ой* итерации генерации изображения на дисплей попадает сумма результатов завершённых итераций, разделённая на количество итераций. Так постепенно удаляется лестничный эффект и Монте-Карло шум в изображении.

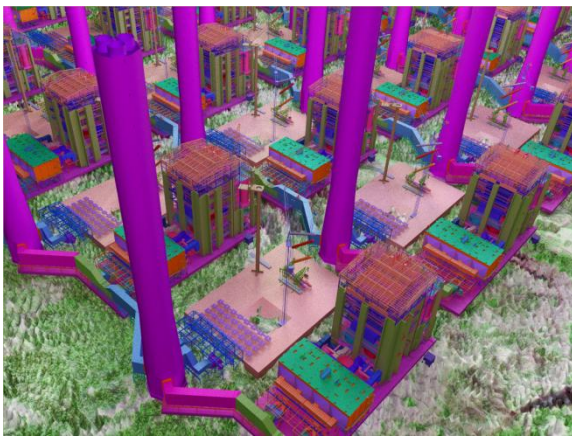
Нет никаких ограничений в применении данного алгоритма визуализации. Вместе с разработанным алгоритмом трассировки лучей возможно применение любых алгоритмов визуализации, основанных на трассировке лучей.

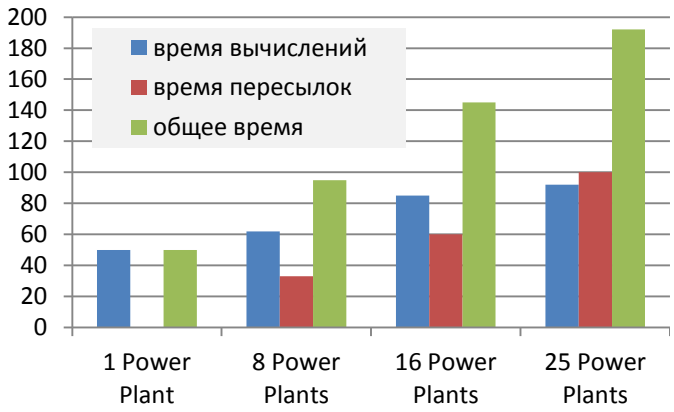
6.7.2. Анализ эффективности

Фиксированный размер кэша GPU, изменяющийся размер сцены. В таблице 6.1 представлен тест с использованием сцены, состоящей из 25 одинаковых моделей Power Plant. Создано несколько реализаций хранения этой сцены. В 1-ом варианте реализации 1 модель Power Plant хранится в памяти и является базовой, остальные 24 модели созданы как клоны базовой модели с необходимыми геометрическими смешениями относительно базы. Во втором варианте реализации в памяти хранится 8 базовых моделей, а остальные 17 моделей являются клонами и т.д.

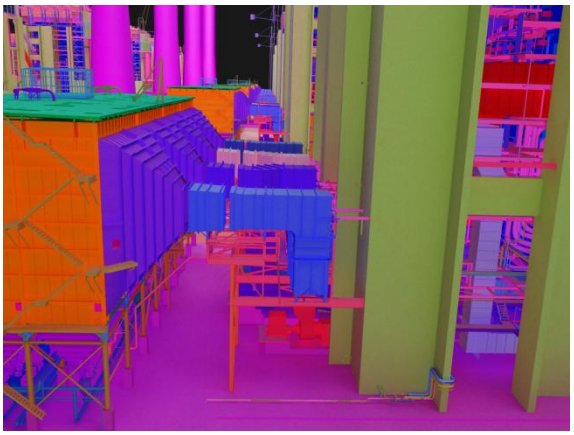
Во всех реализациях хранения этой сцены визуализируется одна и та же сцена, однако объём данных конкретной реализации сцены варьируется от 180 MB до 4500 MB памяти CPU. Во всех реализациях хранения этой сцены необходимо произвести одно и тоже количество вычислений, связанных с поиском пересечений лучей и геометрии. Однако в каждой реализации сцены запрашивается различное количество страниц.

В таблице 6.2 представлен точно такой же тест с другой сценой, состоящей из 30 моделей 3 различных типов с различными реализациями хранения сцены.





Кол-во хранящихся / клонированных треугольников		13 / 325 млн.	104 / 325 млн.	208 / 325 млн.	325 / 325 млн.
Объём данных сцены		180 Мб	1440 Мб	2880 Мб	4500 Мб



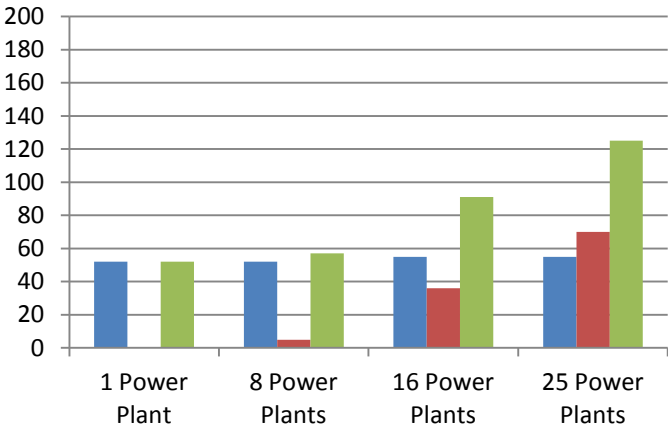
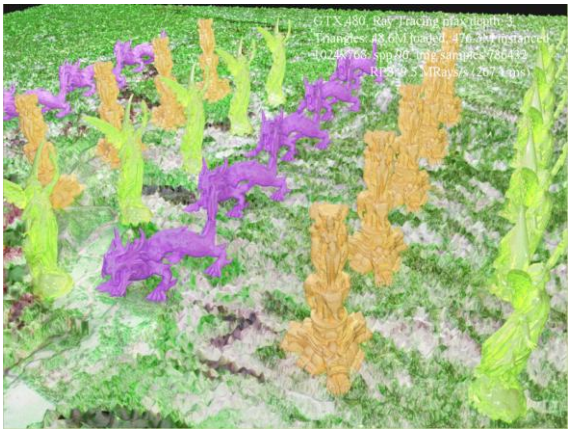
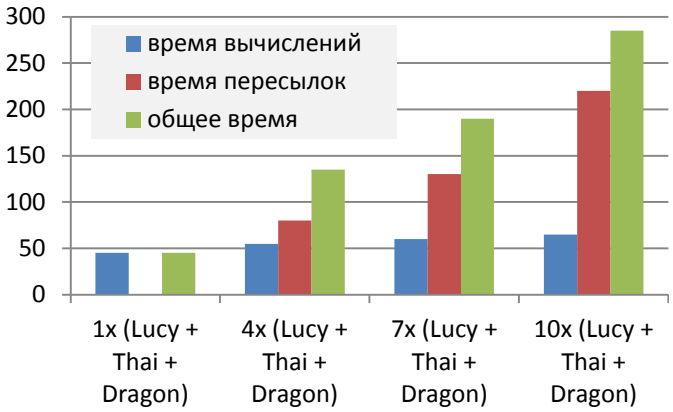


Таблица. 6.1: *Сцена состоит из 25 моделей завода* Power Plant *(каждая модель состоит из 13 млн. треуг-ов), расставленных поверх ландшафта 1.1 млн. треуг-ов и 2.2Гб текстуры, сцена освещена 3 протяжёнными источниками света.* *Вверху* *представлено изображение сцены с видом сверху.* *Внизу* *представлен другой ракурс этой же сцены, когда камера наблюдателя установлена вблизи одной модели* Power Plant. *В правой колонке представлены замеры времени, в мс, Монте-Карло трассировки путей (с 3 уровнями переотражений) для случая, когда в памяти хранится 1 базовая модель* Power Plant, *а все остальные модели созданы с помощью клонирования базовой. Каждый клон состоит из матрицы смещения объекта относительно базы и указателя на область памяти, где хранится* база. *Также представлены замеры времени и объём хранящейся сцены, если в памяти хранится 8 / 16 / 25 базовых моделей* Power Plant, *а остальные созданы с помощью клонирования. Время измерялось при выделении 700 Мб кэша в памяти GPU для временного хранения запрошенных страниц. Время представлено в виде отдельного времени пересылок страниц в кэш GPU, а также отдельного времени вычислений пересечений без учёта пересылок, и общего времени.*





Кол-во хранящихся / клонированных треугольников		45 / 450 млн.	180 / 450 млн.	315 / 450 млн.	450 / 450 млн.
Объём данных сцены		622 Мб	2488 Мб	4354 Мб	6220 Мб

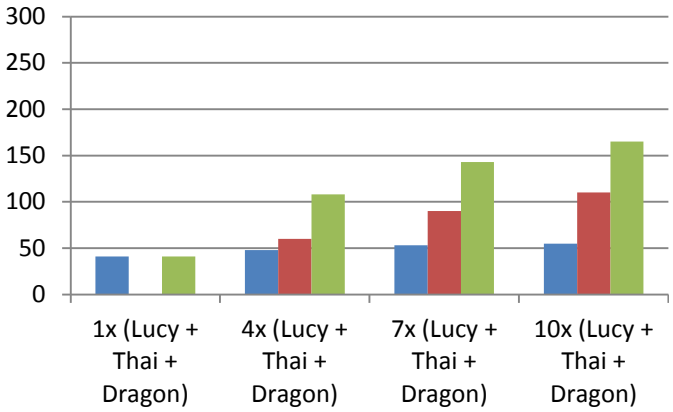


Таблица. 6.2: *Сцена состоит из 10 статуй Thai, 10 статуй Dragon, 10 статуй Lucy, расставленных поверх ландшафта 1.1 млн. треугольников и 2.2Гб текстуры и 3 протяжённых источников света. Вверху представлено изображение сцены с видом сверху. Внизу представлен другой ракурс этой же сцены, когда камера наблюдателя установлена вблизи одной статуи. В правой колонке представлены замеры времени, в мс, Монте-Карло трассировки путей (с 3 уровнями переотражений) для случая, когда в памяти хранится по 1 / 4 / 7 / 10 каждой из трёх статуй, а остальные созданы с помощью клонирования базовых статуй. Каждый клон состоит из матрицы смещения объекта относительно базы и указателя на область памяти, где хранится* база. *Представлены замеры времени и объём хранящейся сцены. Время измерялось при выделении 700 Мб кэша в памяти GPU для временного хранения запрошенных страниц хранящейся сцены и её ускоряющей структуры. Время представлено в виде отдельного времени пересылок страниц в кэш графического процессора, а также отдельного времени вычислений пересечений без учёта пересылок, и общего времени.*

Согласно результатам таблиц 6.1 и 6.2 для реализаций сцен с большим объёмом хранящихся данных, время пересылок страниц занимает большую часть времени. Для Монте-Карло трассировки путей около 50% сцены загружается в кэш GPU во время каждой итерации обновления изображения. Поэтому скорость трассировки лучей для таких сцен ограничивается скоростью передачи данных по шине PCI Express. При этом для увеличения пропускной способности стадии трассировки лучей необходимо увеличить количество лучей, для которых необходимо найти пересечения с геометрией. Для больших сцен, подобных представленным в таблице 6.1 и 6.2, время поиска пересечений для 4 млн. лучей на практике только в 1.5-2 раза больше, чем для 1 млн. лучей.

Для сцен из таблиц 6.1 и 6.2 кэш более эффективен для ближних ракурсов, т.к. в этом случае лучи отражаются в пределах ограниченных областей, страницы пересылаются в кэш GPU редко и удерживаются в кэше долго. Для ракурсов в отдалении от сцены генерируются запросы доступа к страницам любых участков сцены в каждой итерации обновления изображения. Для подобных ракурсов в процессе одной итерации обновления изображения в кэш GPU суммарно может быть загружено больше страниц, чем составляет размер сцены, потому что на каждом поколении отражённых некогерентных лучей все страницы, находившиеся в кэше, могут быть сразу же вытеснены новыми запрошенными страницами.

Фиксированный размер сцены, уменьшающийся размер кэша GPU. В таблице 6.3 представлен тест со сценой размером 700 MB, полностью вмещающейся в максимальный размер кэша GPU. Представлены замеры времени поиска пересечений лучей при уменьшающемся размере кэша. При уменьшении размера кэша в 100 раз время поиска пересечений может упасть в 17 раз для дальнего ракурса сцены и в 7 раз для ближнего ракурса сцены.

В таблице 6.4 представлен тест с уменьшающимся размером кэша GPU для сцены самолёта Боинг 777, содержащего 360 млн. треугольников. При уменьшении размера кэша в 200 раз скорость поиска пересечений лучей снизилась в 10 раз (в целом на графике 6.4 показана логарифмическая зависимость скорости поиска пересечений от размера кэша).

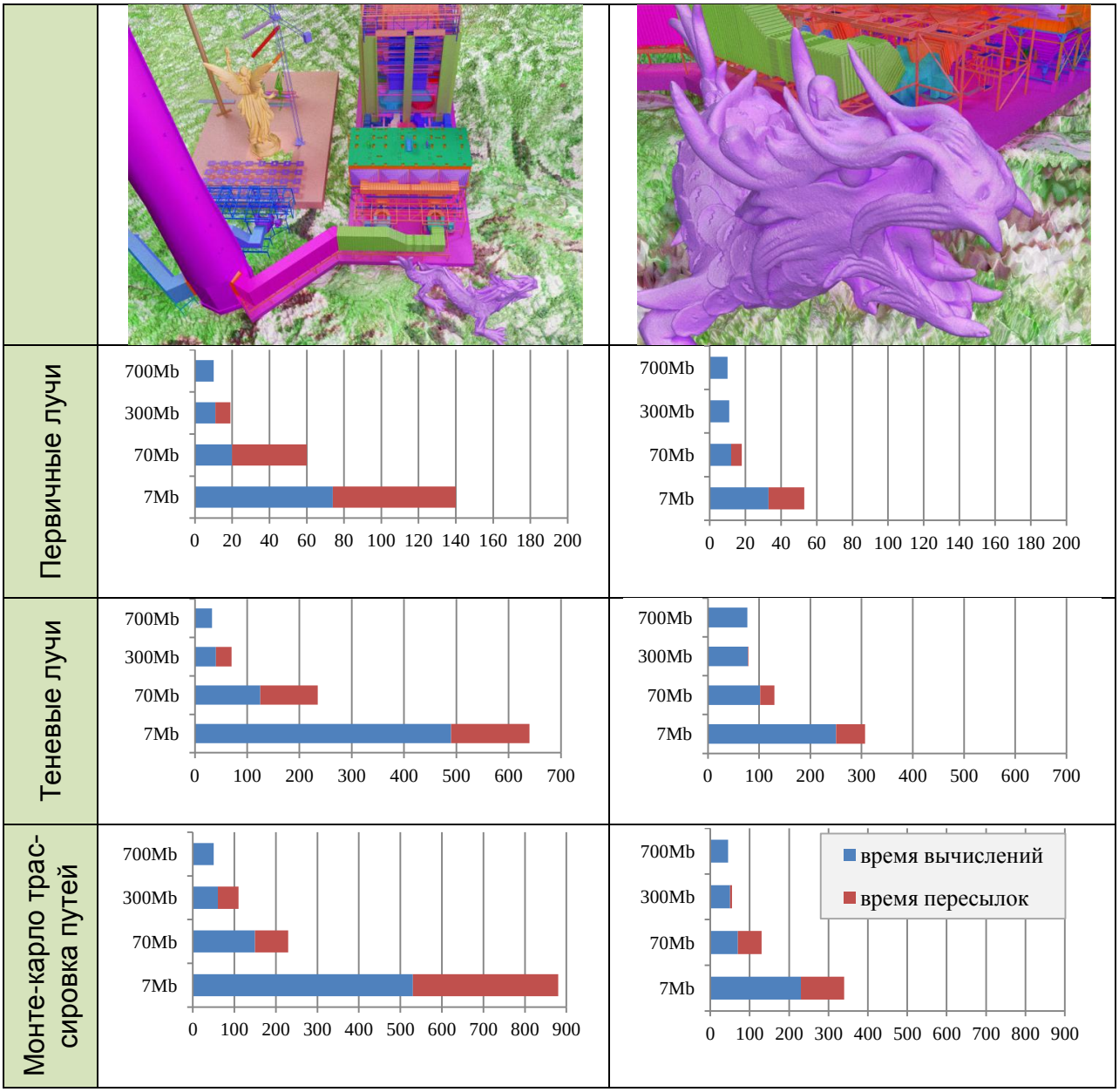
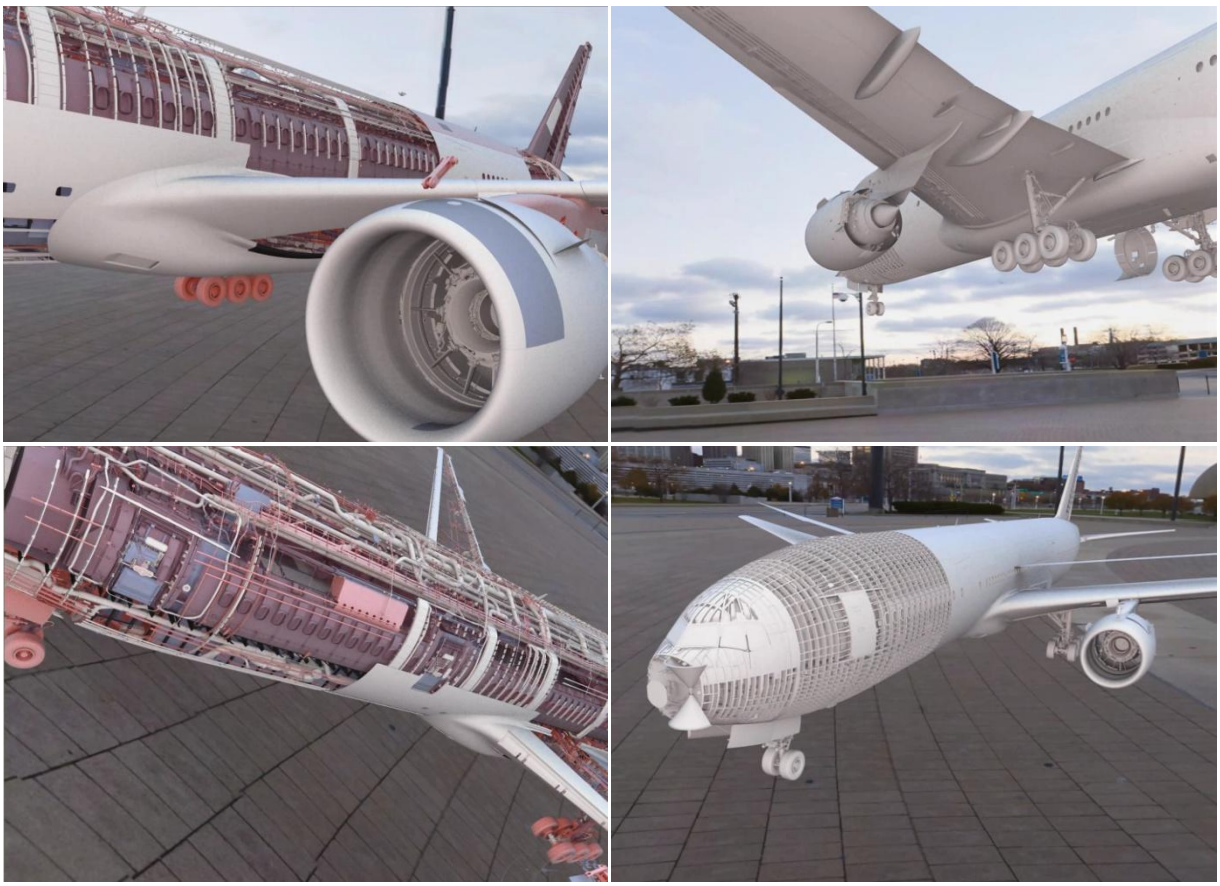


Таблица. 6.3: *Сцена состоит из 1 модели Power Plant (13 млн. треугольников), 1 статуи Lucy (28 млн. треугольников), статуи Dragon (7 млн. треугольников), расставленных поверх ландшафта из 1.1 млн. треугольников и 2.2 Гб текстур и 3 протяжённых источников света. Слева представлен вид сцены сверху, справа представлен вид сцены вблизи статуи Dragon. Суммарно сцена состоит из 49 млн. треугольников, занимающих до 700 Мб в памяти. Внизу для каждого из двух ракурсов сцены представлены замеры времени для 3 видов трассировки лучей: первичная видимость, прямое освещение от 3 протяжённых источников света (теневые лучи) и Монте-Карло трассировка путей с 3 уровнями переотражений. Замеры времени производились для сцены размером 700 Мб при разных размерах кэша на графическом процессоре: 700 Мб (вся сцена может вместиться в кэше), 300 Мб (половина сцены вмещается в кэш), 70 Мб (1/10 часть сцены в кэше), 7 Мб (1/100 часть сцены в кэше).*



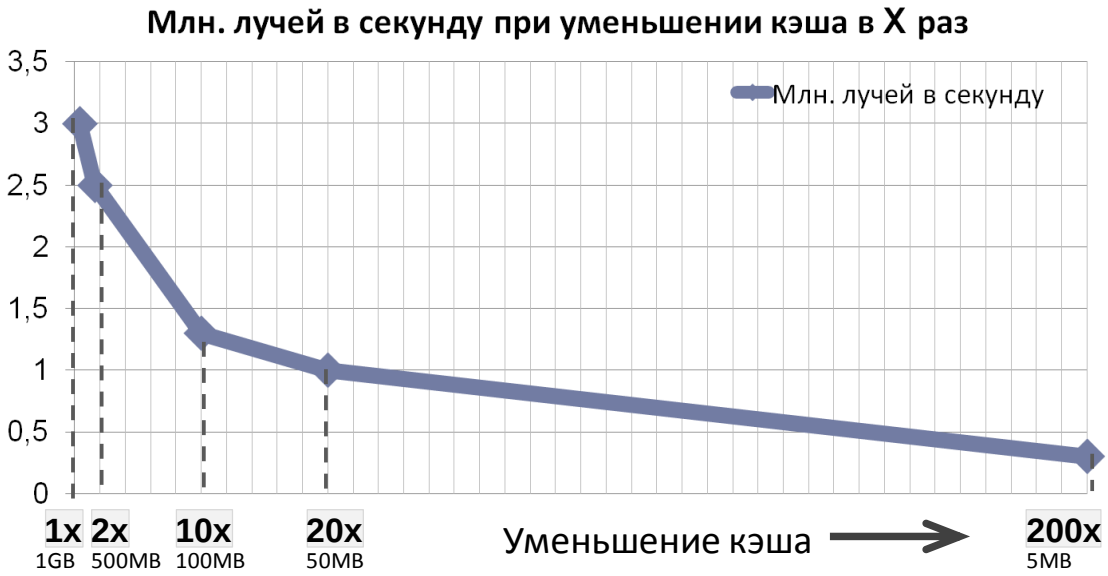


Таблица. 6.4: *Модель самолёта Боинг 777 (предоставлен компанией Боинг, 360 млн. треугольников, занимающих 5Гб в памяти), освещённого картой окружающей среды. На графике представлены замеры скорости трассировки лучей в решении задачи Монте-Карло трассировки путей с 3 уровнями переотражений (стресс-тест для кэша). Замеры времени производились при разных размерах выделенного кэша на GPU: 1 Гб, 500 Мб (кэш в 2 раза меньше максимального), 100 Мб, 50 Мб, 5 Мб. Несмотря на уменьшение размера кэша в 200 раз, скорость трассировки лучей упала всего в 10 раз.*

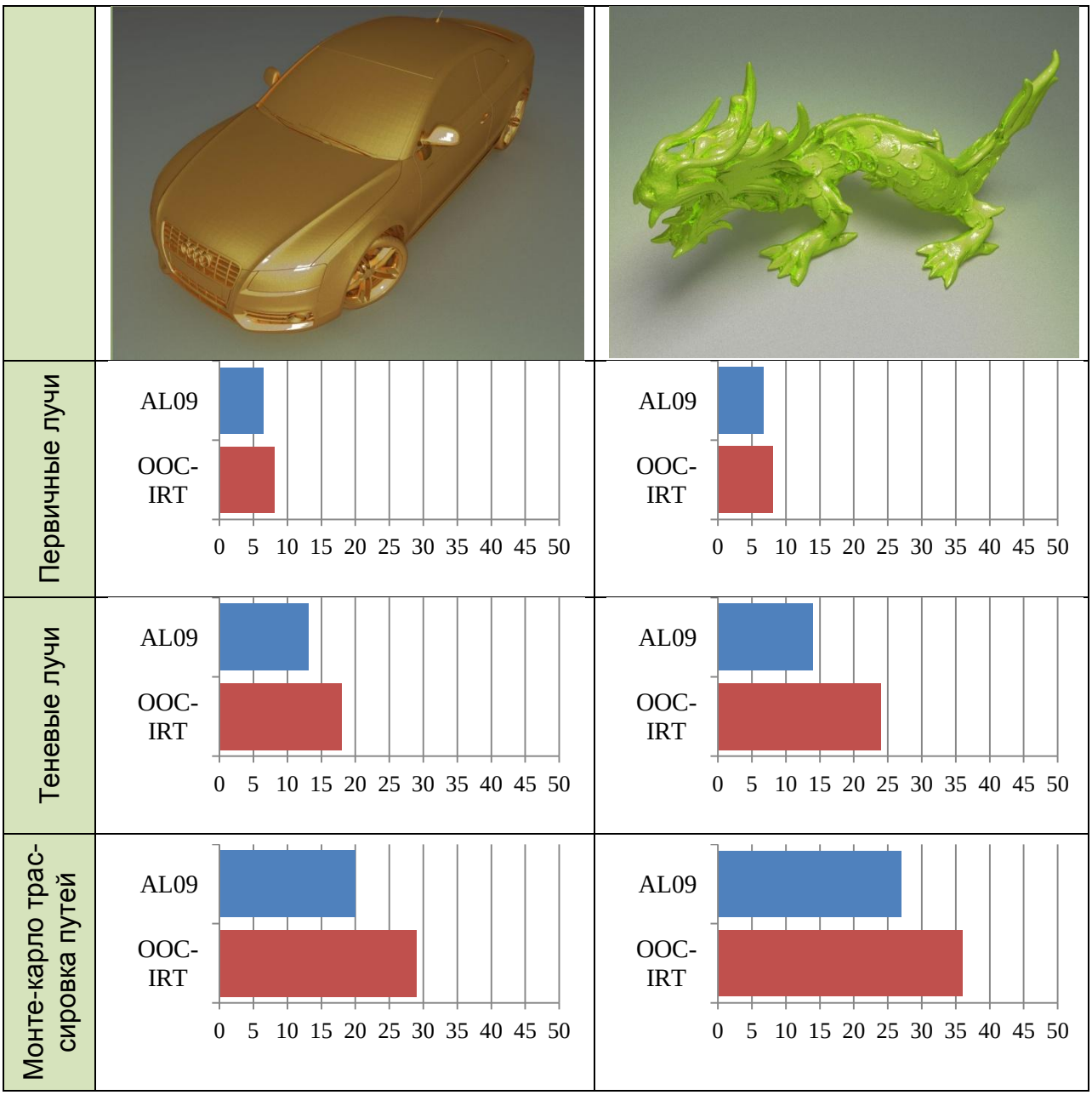
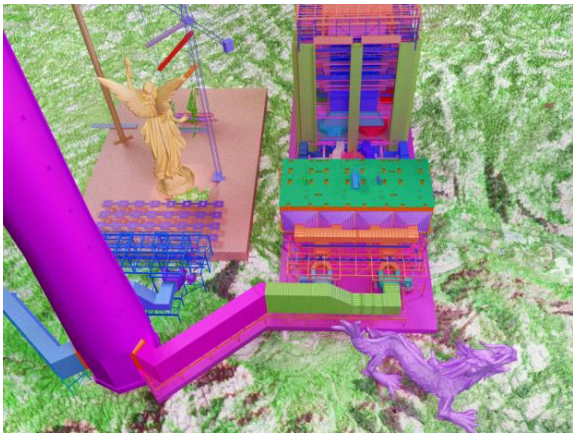
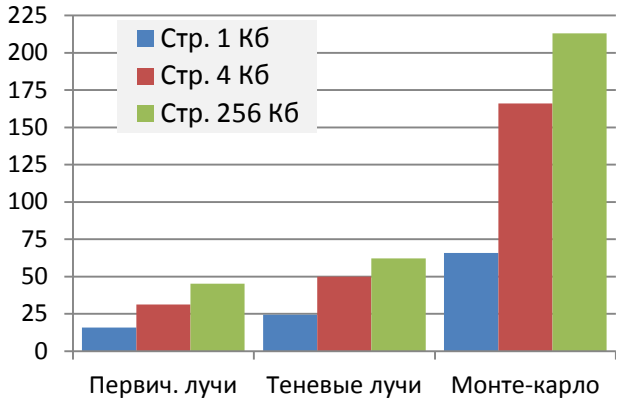


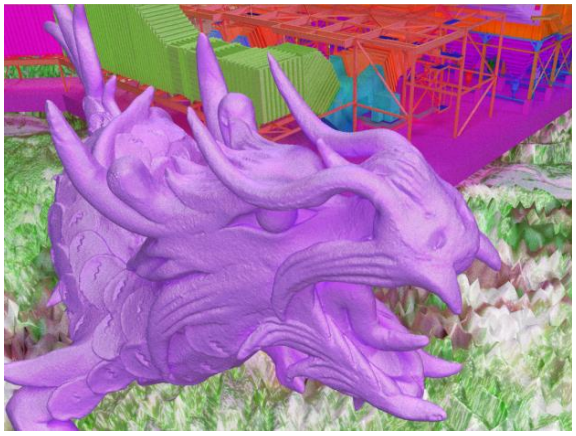
Таблица. 6.5: *Сравнение времени трассировки лучей для двух сцен, состоящих из менее 10 млн. треугольников, полностью вмещающихся в памяти GPU. Сравниваются два алгоритма: OOC-IRT (алгоритм, описанный в данной главе) и AL09 [12] (не поддерживает большие сцены, однако является быстрейшим на данный момент алгоритма поиска пересечений лучей и геометрии, влезающей в память). В данном сравнении выявляется дополнительные временные расходы алгоритма OOC-IRT, связанные с реализацией поддержки больших сцен, по сравнению алгоритмом AL09.*

Тест для малых сцен. В таблице 6.5 представлен тест с небольшой сценой, состоящей из 10 млн. треугольников, полностью вмещающейся в память GPU. Для данной сцены представлены замеры времени поиска пересечений лучей для двух разных алгоритмов: OOC-IRT – алгоритм, описанный в данной главе, поддерживающий трассировку лучей в больших сценах; AL09 – алгоритм, представляющий самую быструю опубликованную реализацию поиска пересечений лучей на GPU [12], реализованный в инфраструктуре настоящего проекта, использующий обычное дерево BVH как ускоряющую структуру, но не поддерживающий большие сцены, превышающие размер памяти GPU. Результаты, показанные в таблице, выявляют долю накладных расходов функций алгоритма OOC-IRT, позволяющих работать с большими сценами. Для сцен, полностью вмещающихся в память GPU, алгоритм OOC-IRT на 10-60% медленнее, чем AL09.

Причинами более медленного поиска пересечений в алгоритме OOC-IRT является использование в дереве BVH нижнего уровня листьев, содержащих не более 15 квадов (экономия памяти, ухудшение скорости поиска), тогда как в алгоритме AL09 дерево BVH имеет листья, содержащие максимум 2-4 квада. Другой причиной более медленной работы OOC-IRT является наличие перекрытий между охватывающими оболочками страниц, принадлежащих одной SAH-ветви. Наличие подобных перекрытий неизбежно в разработанном методе генерации страниц, т.к. страницы создаются с помощью сортировки квадов в порядке возрастания мортон-кодов и тривиального разделения отсортированного массива на равные блоки. Достоинством такого метода является эффективное использование выделенной памяти для подобного способа создания страниц, а также наличие когерентности квадов внутри каждой страницы. При этом генерация SAH-ветвей внутри объекта позволяет избавиться от дополнительных возможных перекрытий и обеспечивать приемлемую скорость поиска пересечений лучей с использованием сложной 3-уровневой ускоряющей структуры.







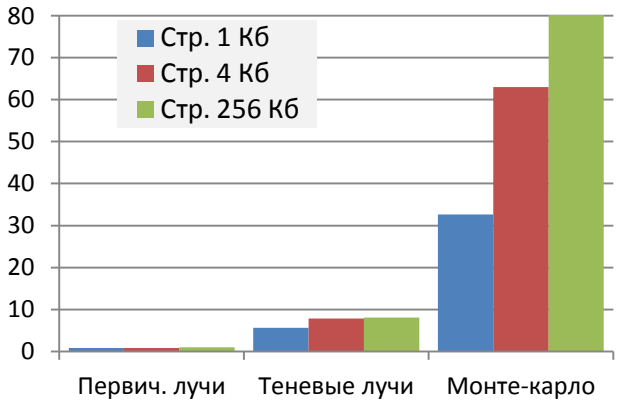


Таблица. 6.6: *Сцена состоит из 1 модели Power Plant (13 млн. треугольников), 1 статуи Lucy (28 млн. треугольников), статуи Dragon (7 млн. треугольников), расставленных поверх ландшафта из 1.1 млн. треугольников и 2.2 Гб текстур и 3 протяжённых источников света. Представлено время текстурной фильтрации, разбитое на время пересылок между CPU и GPU, и время собственно вычислений, связанных с самой фильтрацией, для различных размеров текстурных страниц (тайлов): 1 Кб, 4 Кб или 256 Кб. Данное время рассчитано для 3 разных задач: расчёт первичной видимости, прямого освещения от 3 протяжённых источников света или стохастической Монте-Карло трассировки пути с 3 уровнями переотражений. Во всех тестах наибольшую скорость текстурной фильтрации образуется вследствие использований мелких страниц 1 Кб, т.к. при стохастических обращениях в текстурные ячейки при пересылках мелких страниц по запросу дополнительно пересылается меньше информации по сравнению со случаем использования страниц 256 Кб.*

Скорость фильтрации текстур. В таблице 6.6 представлена скорость текстурной фильтрации с учётом уровней детализации при выборе различного размера страницы (тайла текстуры) и постоянного размера кэша GPU 100 MB. Текстурная фильтрация осуществляется в каждой точке пересечения луча и геометрической поверхности, поэтому скорость текстурной фильтрации зависит от когерентности лучей. Также как и в случае с тестом скорости поиска пересечений лучей в ближнем ракурсе текстурная фильтрация осуществляется быстрее, чем в дальнем, т.к. лучи генерируются в ограниченной области пространства.

Рис. 6.5: *Сцена Times Square New Yourk состоит из 25 млн. треугольников и 10 GB текстур, разрешение кадра 1280x720. Визуализация осуществляется с помощью алгоритма Монте-Карло трассировки путей на GPU GTX580 (3GB памяти) с 3 уровнями переотражений. Частота обновлений изображения в прогрессивном процессе Монте-Карло интеграции: 10 обновлений в секунду. Время генерации финального кадра: 2.5 мин. (1500 обновлений изображения) В процессе каждого Монте-Карло обновления выпускается по одному лучу для каждого пискеля кадра, в точке пересечения луча стохастически создаётся отражённый луч (отражается в зависимости от свойств материала) для учёта вторичного освещения в точке и теневой луч для расчёта прямого освещения в точке. В точке пересечения вторичного луча производится та же операция (глубина учёта вторичного освещения равна трём). Свойства материала определяются на основе значений из текстур, наложенных на поверхность.*

Быстрая скорость текстурной фильтрации для стохастической Монте-Карло трассировки путей обеспечивается минимальным размером страницы (тайла) 4 KB, т.к. в этом случае для некогерентных лучей в кэш GPU загружаются страницы, несущие для каждого запроса текселя меньший объём не нужной информации. Однако выбор маленького размера для страницы означает, что таблица страниц будет большой и, следовательно, операции, связанные с вытеснением страниц из кэша, использующие сортировку, будут замедлены. Следует учитывать, что для некоторых сцен, содержащих большое количество текстур высокого разрешения, таблицы маленьких страниц могут не вместиться полностью в память GPU.

6.7.3. Влияние параметров ускоряющей структуры

Разрешение сетки верхнего уровня. Разрешение 32^3 обеспечивает более высокую скорость поиска пересечений лучей по сравнению с 8^3-16^3 для больших сцен с небольшим кэшем. Большее разрешение 64^3 не обеспечивает большего ускорения для таких сцен. Разрешение сетки 128^3 влечёт за собой создание слишком большого числа ячеек, содержащих растеризованные оболочки страниц, что влечёт высокий расход памяти на сетку, постоянно хранящуюся в памяти GPU.

Разрешение сетки 1^3 обеспечивает на 25% более быстрый поиск пересечений лучей по сравнению с разрешением 32^3 для сцен, у которых все страницы вмещаются в кэш GPU. Разрешение сетки 1^3 обеспечивает в 2-4 раза более медленный поиск пересечений лучей для сцен, занимающих в 10 раз больше памяти, чем кэш графического процессора.

Размер страницы ускоряющей структуры. Если страница ускоряющей структуры содержит более 4K квадов, то поиск пересечений некогерентных лучей (в такой задаче как Монте-Карло трассировка путей) осуществляется более медленно, т.к. каждый запрос страницы может доставить в кэш GPU данные, которые могут быть не использованы. Страницы, содержащие менее 4K квадов, иногда могут обеспечивать более высокую скорость поиска пересечений, однако следует учитывать, что это влечёт за собой больший и для некоторых сцен неприемлемый расход памяти для сетки верхнего уровня и BVH среднего уровня, постоянно хранящихся в памяти GPU.

6.7.4. Доля кэш-попаданий

Доля кэш попаданий в поиске пересечений лучей является низкой (менее 50%), т.к. во время поиска пересечений некогерентных лучей (например, в результате диффузных переотражений) производится большое количество обращений и запросов к некогерентным страницам сцены в рамках одного запуска параллельной программы поиска пересечений. Для диффузных переотражений для одного поколения лучей часто необходимо получить доступ практически ко всем имеющимся в сцене страницам, которые поочерёдно загружаются в кэш GPU, быстро вытесняя ранее находившиеся там страницы. Когда размер кэша является относительно большим, например 20-50% от размера сцены, то доля кэш попаданий для некогерентных лучей более 50%. Когда кэш меньше 10-20% от размера сцены, то страницы быстро вытесняются из кэша, и доля кэш попаданий опускается ближе к 10%.

Однако важным свойством разработанного алгоритма является то, что в первую очередь для загрузки в кэш GPU запрашиваются те страницы, оболочки которых в среднем для миллионов лучей, являются ближайшими к основаниям лучей. Такой порядок загрузки данных повышает эффективность использования загруженных страниц.

6.7.5. Сравнение с аналогами

Модель самолёта Боинг 777 (предоставленная корпорацией Боинг для проведения тестов, содержащая 360 млн. треугольников) является одной из немногих моделей, с помощью которых различные исследовательские группы могут сравнить результаты разработанных алгоритмов визуализации больших сцен. В таблице 6.4 представлены изображения, синтезированные с помощью разработанного алгоритма, исполняющегося на единственном графическом процессоре. Одна итерация прогрессивной Монте-Карло трассировки путей с 3 уровнями переотражений рассчитывается за 60-300 миллисекунд (выпускается 1 путь на пиксель, имеется 1024x768 пикселей). Подобная скорость позволяет осуществлять интерактивный предварительный просмотр глобального освещения и навигацию в очень детализированных сценах на стандартном пользовательском компьютере.

При этом скорость поиска пересечений лучей при использовании 1 пути на пиксель является 2 млн. лучей / сек. Когда увеличивается количество путей на пиксель до 16 в рамках одной итерации, то скорость поиска пересечений лучей увеличивается до 6-7 млн. лучей / сек.

В сравнении с результатами, представленными в конкурирующей работе [24] (использующей 2 более старых графических процессора NVIDIA Quadro), полученный новый результат является в 10-100 раз более быстрым при использовании одного графического процессора NVIDIA GTX 480. Авторы других конкурирующих работ производили попытки визуализировать настолько сложные сцены как Боинг с использованием серверных установок, содержащих десятки узлов. При работе на одном узле такая сложная сцена визуализировалась в течение нескольких десятков часов, а интерактивный предварительный просмотр сложных детализированных ракурсов был невозможен.

6.8. Заключение

В настоящей главе описан out-of-core алгоритм поиска пересечений лучей на графическом процессоре с ограниченной памятью для больших сцен, возможно превосходящих размер памяти графического процессора в десятки раз. Стадия поиска пересечений лучей и геометрии сцены является одной из основных и самых затратных в процессе расчёта глобального освещения сцены или алгоритмах симуляции столкновений. Большое количество алгоритмов визуализации, с использованием произвольно сложных материалов поверхностей, могут быть реализованы с использованием разработанного алгоритма.

В рамках настоящей работы проведённая демонстрация интерактивной визуализации крупных сцен (состоящих из нескольких сотен млн. треугольников) на настольном компьютере является первой в мире.

Тенденцией различных сфер анимационной, инженерной графики, связанных с визуализацией, является постоянный рост сложности сцен. Также существует тенденция увеличения количества ядер и упрощения самих ядер процессоров, разработанных в различных компаниях. При этом увеличение объёма памяти гра-

фического процессора (в частности GDDR5) влечёт к значительному увеличению цены процессора, уменьшая доступность таких процессоров для массовых пользователей. На текущий момент максимальные размеры памяти GPU являются 6 GB для NVIDIA Titan, 12 GB для NVIDIA Quadro K6000, 16 GB для Intel Xeon Phi. При этом размеры сцен могут измеряться в десятках и сотнях гигабайт. Поэтому разработанный алгоритм является актуальным в свете сохранения разницы между запрашиваемой сложностью сцен и имеющейся памятью на графическом процессоре.

Практическая реализация и применение

Алгоритм поиска пересечений лучей в массивных сценах, представленный в главе 6, а также модифицированные алгоритмы построения геометрической базы данных, представленные в главах 2, 4 и 5, были внедрены в программный продукт Сентилео [26] в соответствующие стадии конвейера реалистичной визуализации (см. рис. 1.1). Реализация осуществлена с применением технологии программирования CUDA, занимает порядка 18 тыс. строк программного кода.

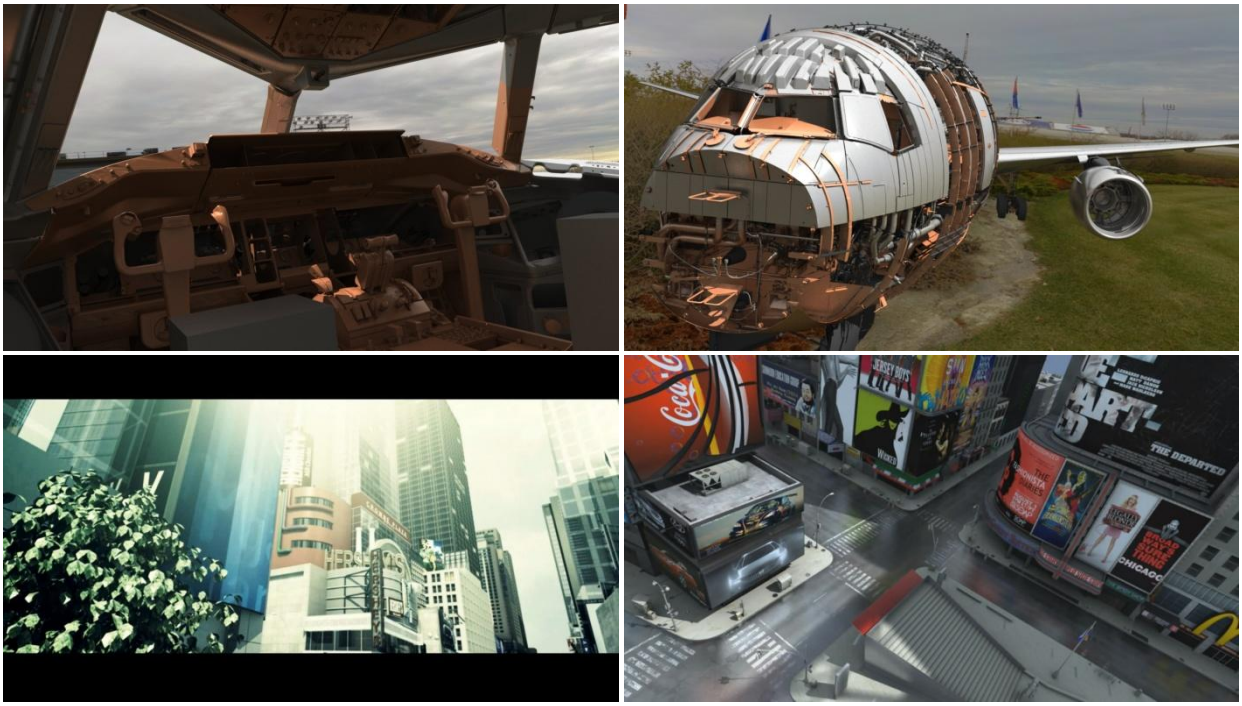


Рис. 7.1: *Пример использования разработанных алгоритмов в программном продукте Сентилео реалистичной визуализации. Визуализация финального кадра сцены Боинга 777 (360 млн. треугольников) и New York Times Square (25 млн. треугольников, 10 Гб текстур) осуществляется за 5-10 мин. Также доступен режим интерактивного просмотра глобального освещения со скоростью 5-10 обновлений изображения в секунду.*

Разработанный продукт Сентилео исполняется на массовых настольных и мобильных компьютерах, оснащённых серийными графическими процессорами, например на ноутбуке с графическим процессором GTX 485M, 780M, настольном компьютере с графическим процессором GTX480, 580, 680, 780 и т.п.

На рис. 7.1 представлены примеры синтезированных изображений высокодетализированных сцен, состоящих из текстур высокого разрешения с применени-

ем одного графического процессора NVIDIA GeForce GTX 580. Представленные результаты на порядок превосходят по скорости визуализации систему, основанную на платформе трассировки лучей OptiX [83], исполняющуюся на профессиональном графическом процессоре Quadro 6000, содержащего 6 GB физической памяти. Следует отметить, что графические процессоры серии Quadro на порядок дороже графических процессоров серии GeForce, также располагают в несколько раз большим объёмом физической памяти и примерно такой же вычислительной мощностью. Других конкурирующих продуктов реалистичной визуализации подобных массивных сцен с применением графических процессоров на данный момент не представлено. Кроме этого система визуализации массивных сцен на графическом процессоре, основанная на алгоритме из главы 6, на порядок быстрее аналогов, исполняющихся на многоядерном центральном процессоре (например [24]).

Программный продукт Сентилео по итогам внедрения разработанных в настоящей работе алгоритмов, был представлен на крупнейшей международной выставке индустрии компьютерной графики SIGGRAPH 2012. В настоящий момент развитие программного продукта продолжается.

Заключение

В рамках настоящей работы разработана серия алгоритмов, позволяющих повысить скорость исполнения тяжёлых стадий конвейера визуализации:

1. Разработаны масштабируемые алгоритмы поиска пересечений лучей и фильтрации текстур, позволяющие с использованием серийных графических процессоров реалистично визуализировать массивные сцены (до 1 млрд. треугольников, до 1 терабайта текстур) на порядок быстрее существующих аналогов, исполняющихся на CPU и на графическом процессоре. Разработанные алгоритмы применимы совместно с любыми алгоритмами расчёта глобального освещения, основанными на трассировке лучей, также применимы для интерактивных систем.
2. Разработанный алгоритм поиска пересечений лучей позволил достичь логарифмической зависимости скорости поиска пересечений от размера кэша графического процессора. Это позволяет использовать кэш память меньшего размера с сохранением приемлемой скорости поиска пересечений.
3. Разработано несколько алгоритмов быстрого построения геометрической базы данных, в том числе алгоритмов, исполняющихся на графическом процессоре с более низким и предсказуемым потреблением памяти и на порядок быстрее существующих аналогов.
4. На основе разработанных алгоритмов реализован программный комплекс реалистичной визуализации, применение которого позволяет привести к существенному повышению производительности труда в архитектурном, инженерном проектировании и киноиндустрии.

Литература

[1] *Garanzha K.* Efficient Clustered BVH Update Algorithm for Highly-Dynamic Models // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing, pp. 123-130 – 2008 [3].

[2] *Garanzha K.* The Use of Precomputed Triangle Clusters for Accelerated Ray Tracing in Dynamic Scenes // Proceedings of Eurographics Symposium on Rendering 2009 / Computer Graphics Forum, Vol. 28, № 4, pp. 1199-1206 – 2009 [3,4].

[3] *Garanzha K., Loop C.* Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing // Proceedings of 31th Annual Conference of the European Association for Computer Graphics Eurographics 2010, Norrköping, Sweden / Computer Graphics Forum, Vol. 29, № 2, pp. 289-298 – 2010 [3,4].

[4] *Garanzha K., Premože S., Bely A., Galaktionov V.* Grid-based SAH BVH construction on a GPU // Proceedings of Computer Graphics International 2011, Ottawa, Canada / The Visual Computer, Vol. 27, № 6-8, pp. 697-706 - 2011 [3,4].

[5] *Garanzha K., Bely A., Galaktionov V.* Simple geometry compression for ray tracing on GPU // Conference proceedings of 21-th International Conference on Computer Graphics and Vision GraphiCon-2011, Moscow State University, pp.107-110 - 2011 [3].

[6] *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, Vancouver, BC, Canada, pp.59-64 - 2011 [3].

[7] *Garanzha K., Bely A., Premoze S., Galaktionov V.* Out-of-core GPU ray tracing of complex scenes // In Technical talk Proceedings 38th International conference on computer graphics and interactive techniques ACM SIGGRAPH 2011, Article No.21, Vancouver, Canada - 2011 [3].

[8] *Боголепов Д., Сопин Д., Ульянов Д., Турлапов В.* Система интерактивного расчета глобального освещения для гибридных сцен // Труды 22-ой международной конференции 184 по компьютерной графике и зрению GraphiCon-2012. – М.: МАКС Пресс - 2012.

[9] *Боголепов Д.* Методы глобального освещения для интерактивного синтеза изображений сложных сцен на графических процессорах // Диссертация на соискание учёной степени кандидата технических наук. – Нижегородский Государственный Университет им. Н.И. Лобачевского - 2013.

[3] Публикация автора, входящая в библиографическую базу Scopus, входящая в перечень ВАК

[4] Публикация автора, входящая в библиографическую базу Web of Science, входящая в перечень ВАК

[10] *Волобой А., Галактионов В., Дмитриев К., Копылов Э.* Двунаправленная трассировка лучей для интегрирования освещенности методом квази- Монте Карло // Программирование. – 2004. – № 5.

[11] *Востряков К.* Когерентные алгоритмы синтеза реалистичных изображений // Диссертация на соискание учёной степени кандидата физико-математических наук. – Институт Прикладной Математики им. М.В. Келдыша РАН - 2009.

[12] *Aila T., Laine S.* Understanding the efficiency of ray traversal on GPUs. // Proceedings of the ACM Symposium on High Performance Graphics - 2009.

[13] *Aila T., Karras T.* Architecture considerations for tracing incoherent rays // Proceedings of the ACM Symposium on High Performance Graphics - 2010.

[14] *Aila T., Karras T.* Fast Parallel Construction of High-Quality Bounding Volume Hierarchies // Proceedings of the ACM Symposium on High Performance Graphics - 2013.

[15] *Amara Y., Marsault X.* A GPU tile-load-map architecture for terrain rendering: theory and applications // The Visual Computer 25 (June) - 2009.

[16] *Antwerpen D.V.* Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU // Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. – 2011.

[17] *Antwerpen D. V.* Recursive MIS Computation for Streaming BDPT on the GPU // Technical report / Delft University of Technology – 2011.

[18] *Antwerpen D.V.:* Unbiased physically based rendering on the GPU // PhD. thesis, Department of Software Technology, Faculty EEMCS, Delft University of Technology in Delft, Netherlands - 2011.

[19] *Arvo J., Kirk D.* Fast ray tracing by ray classification // ACM SIGGRAPH Computer Graphics 21, 4 - 1987.

[20] *Ashikhmin M., Shirley P.* An Anisotropic Phong Light Reflection Model // Journal of Graphics Tools - 2000.

[21] *Bentin C., Wald I.* Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing // Proceedings of the High Performance Graphics - 2009.

[22] *Borgeat L., Godin G., Blais F., Massicotte P., La-hanier C.* Gold: interactive display of huge colored and textured models // ACM Transactions on Graphics 24 (July) - 2005.

[23] *Boulos S., Wald I., Bentin C.* Adaptive Ray Packet Reordering // In Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2008.

[24] *Budge B., Bernardin T., Stuart J. A., Sengupta S., Joy K. I., Owens J. D.* Out-of-core data management for path tracing on hybrid resources // Computer Graphics Forum 28, 2 (April) - 2009.

[25] *Caustic Graphics* // Официальный сайт компании Caustic Professional http://www.caustic.com

[26] *CentiLeo* // Официальный сайт компании CentiLeo http://www.centileo.com

[27] *Choi, B., Komuravelli, R., Lu, V., Sung, H., Bocchino, R.L., Adve, S.V., Hart, J.C.* Parallel SAH k-D Tree Construction // Proceedings of ACM Symposium on High Performance Graphics - 2010

[28] *Cook R., Porter T., Carpenter L.* Distributed Ray Tracing // Computer Graphics (Proceedings of SIGGRAPH 84) 18, 3 - 1984.

[29] *CUDPP*: CUDA data parallel primitives library http://www.gpgpu.org/developer/cudpp/

[30] *Dammertz H., Keller A.* The Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2008.

[31] *Dietrich A., Gobbertti E., Yoon S.-E.* Massive-model rendering techniques // A tutorial / IEEE Computer Graphics and Applications 27 - 2007.

[32] *Ernst M., Greiner G.* Early split clipping for bounding volume hierarchies // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2007.

[33] *Eisemann M., Grosch T., Magnor M., Mueller S.* Automatic Creation of Object Hierarchies for Ray Tracing of Dynamic Scenes // Technical Report 2006-6-1, TU Braunschweig - 2006.

[34] *Eisenacher C., Nichols G., Selle A., Burley B.* Sorted Deferred Shading for Production Path Tracing // In Proceedings of Eurographics Symposium on Rendering - 2013.

[35] *Embree*: Photo-Realistic Ray Tracing Kernels http://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels

[36] *Foley, T., Sugerman, J.* KD-tree acceleration structures for a GPU raytracer // Graphics Hardware - 2005.

[37] *Frolov V., Vostryakov K., Kharlamov A., Galaktionov V.* Irradiance cache for a GPU ray tracer // Proceedings of 22-th International Conference on Computer Graphics and Vision Graphicon - 2012.

[38] *Garland M., Willmott A., Heckbert P.* Hierarchical Face Clustering on Polygonal Surfaces // Proceedings of the Symposium on Interactive 3D Graphics - 2001.

[39] *Garland M., Kirk D. B.* Understanding throughput-oriented architectures // Communications ACM 53 (November) - 2010.

[40] *Georgiev I., Křivanek J., Slusallek P.* Bidirectional light transport with vertex merging // Proceedings SIGGRAPH Asia 2012 / ACM Transactions on Graphics vol 31, 6 - 2012.

[41] *Glassner A.S.* An Introduction to Ray Tracing // Academic Press, London - 1989.

[42] *Gobbetti E., Kasik D., Yoon S.-E.* Technical strategies for massive model visualization // Proceedings of the ACM symposium on Solid and physical modeling, SPM'08 - 2008.

[43] *Goldsmith J., Salmon J.* Automatic creation of object hierarchies for ray tracing // IEEE Computer Graphics & Applications 7, 5 (May) - 1987.

[44] *Gribble C.-P., Ramani K.* Coherent Ray Tracing via Stream Filtering // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2008.

[45] *Gunther J., Popov S., Seidel H.-P., Slusallek P.* Realtime ray tracing on GPU with BVH-based packet traversal // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2007.

[46] *Gunther J., Friedrich H., Wald I., Seidel H.-P., Slusallek P.* Ray tracing animated scenes using motion decomposition // Proceedings of Eurographics 2006 / Computer Graphics Forum 25, 3 (September) - 2006.

[47] *Havran, V.* Heuristic ray shooting algorithms // PhD. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague - 2000.

[48] *Havran V.* About the Relation between Spatial Subdivisions and Object Hierarchies Used in Ray Tracing // Proceedings of conference SCCG, Budmerice, Slovakia - 2007.

[49] *Hanrahan P.* Using caching and breadth first traversal to speed up ray tracing // Proceedings of the Graphics Interface - 1986.

[50] *Hennessy J. L., Patterson D. A.* Computer Architecture A Quantitative Approach, Fourth Edition // Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[51] *Hoberock J., Lu V., Jia Y., Hart J.C.* Stream Compaction for Deferred Shading // Proceedings of the High Performance Graphics - 2009.

[52] *Horn D. R., Sugerman J., Houston M., Hanrahan P.* Interactive k-d tree GPU ray tracing // Proceedings of the symposium on Interactive 3D graphics and games - 2007.

[53] *Hunt W., G. Stoll, W. Mark.* Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic // Proceedings of the IEEE Symposium on Interactive Ray Tracing - 2006.

[54] *Hunt W., Mark W., Fussell D., Stoll G.* Fast and Lazy Build of Acceleration Structures from Scene Hierarchies // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2007.

[55] *Hunt W., Mark W.* Ray-Specialized Acceleration Structures for Ray Tracing // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2008.

[56] *Ize T., Wald I., Robertson C., S. G. Parker.* An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes // Proceedings of the IEEE Symposium on Interactive Ray Tracing - 2006.

[57] *Ize, T., Wald, I., Parker, S.G.* Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures // Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization - 2007.

[58] *Jensen H.W.* Global Illumination using Photon Maps // Eurographics Rendering Workshop - 1996.

[59] *Jensen H.W.* Realistic Image Synthesis using Photon Mapping // A K Peters - 2001. ISBN 156881-147-0.

[60] *Kalojanov J., Slusallek P.* A parallel algorithm for construction of uniform grids // Proceedings of the 1st ACM conference on High Performance Graphics - 2009.

[61] *Kalojanov J., Billeter M., Slusallek P.* Two-level grids for ray tracing on GPUs // Computer Graphics Forum (4) - 2011.

[62] *Kajiya J.T.* The Rendering Equation // Computer Graphics (Proceedings of SIGGRAPH'86) Vol. 20, no. 4 - 1986.

[63] *Kay T., Kajiya J.* Ray tracing complex scenes // Proceedings of SIGGRAPH'86 - 1986.

[64] *Kim T.-J., Moon B., Kim D., Yoon S.-E.* Racbvhs: Random-accessible compressed bounding volume hierarchies // IEEE Transactions on Visualization and Computer Graphics 99 - 2009.

[65] *Lafortune E.P., Willems Y.D.* Bi-Directional Path Tracing // Proceedings of Computer graphics '93, Alvor, Portugal – 1993.

[66] *Laine S., Karras T.* Efficient sparse voxel octrees // Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - 2010.

[67] *Laine S.* Restart trail for stackless BVH traversal // Proceedings of High-Performance Graphics - 2010.

[68] *Lauterbach C., Yoon S.-E., Tuft D., Manocha D.* RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs // Proceedings of the IEEE Symposium on Interactive Ray Tracing - 2006.

[69] *Lauterbach C., Yoon S.-E., Manocha D.* Ray-strips: A compact mesh representation for interactive ray tracing // Proceedings. IEEE/EG Symp. Interactive Ray Tracing - 2007.

[70] *Lauterbach C., Yoon S.-E., Tang M., Manocha, D.* ReduceM: Interactive and memory efficient ray tracing of large models // Computer Graphics Forum 27, 4 (June) - 2007.

[71] *Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.* Fast BVH Construction on GPUs // Proceedings of Eurographics 2009 / Computer Graphics Forum 28, 2 - 2009.

[72] *Mahovsky J. A.* Ray tracing with reduced-precision bounding volume hierarchies // PhD thesis, University of Calgary, Canada - 2005.

[73] *Maury O., Pipony D., Andorra F., Hammack C.* Bending fire with plume, a cuda-based 3d fluid solver and volume renderer // ACM SIGGRAPH Talks 2010.

[74] *Merrill D., Grimshaw A.* Revisiting sorting for gpgpu stream architectures // Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA - 2010.

[75] *Moller T., Trumbore B.* Fast, minimum storage ray triangle intersection // JGT, 2(1): 21-28 - 1997.

[76] *Nickolls J., Buck I., Garland M., Skadron K.* Scalable parallel programming with CUDA. Queue 6, 2 - 2008.

[77] *Novak J., Havran V., Dachsbacher C.* Path Regeneration for Interactive Path Tracing // Proceedings of EUROGRAPHICS (Short Papers). – 2010.

[78] *Niessner M., Loop C., Meyer M., DeRose T.* Feature Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces // ACM Transactions on Graphics (TOG) Volume 31 Issue 1 January - 2012.

[79] *NVIDIA CUDA*. Programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[80] *Overbeck R., Ramamoorthi R., Mark W.-R.* Large Ray Packets for Real-time Whitted Ray Tracing // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2008.

[81] *Pantaleoni J., Fascione L., Hill M., Aila T.* Pantaray: Fast ray-traced occlusion caching of massive scenes // ACM Transactions on Graphics 29, 4 (July) - 2010.

[82] *Pantaleoni J., Luebke D.* HLBVH: Hierarchical LBVH construction for real-time ray tracing // Proceedings of High Performance Graphics - 2010.

[83] *Parker S. G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robinson, A., Stich M.* Optix: A general purpose ray tracing engine // ACM Transactions on Graphics 29, 4 (July) - 2010.

[84] *Peachey D.* Texture on Demand // Pixar Animation, Technical Memo#217 - 1990.

[85] *Pharr M., Humphreys G.* Physically Based Rendering, Second Edition: From Theory to Implementation // Morgan Kaufmann Publishers Inc., San Francisco, CA, USA - 2010. ISBN 0123750792 9780123750792.

[86] *Pharr M., Kolb C., Gershbein R., Hanrahan P. M.* Rendering complex scenes with memory-coherent ray tracing // Proceedings of SIGGRAPH'97 - 1997.

[87] *Popov S., Gunther J., Seidel H.-P., Slusallek P.* Experiences with streaming construction of SAH KD-trees // Proceedings of the IEEE Symposium on Interactive Ray Tracing - 2006.

[88] *Popov S., Gunther J., Seidel H.-P. Slusallek P.* Stackless KD-tree traversal for high performance GPU ray tracing // Computer Graphics Forum 26, 3 (Sept.) - 2007.

[89] *Ragan-Kelley J., Kilpatrick C., Smith B. W., Epps D., Green P., Hery C., Durand F.* The lightspeed automatic interactive lighting preview system // ACM Transactions on Graphics 26 (July) - 2007.

[90] *RayFire Studios.* Официальный сайт компании RayFire Studios. http://rayfirestudios.com

[91] *RenderMan Shaders*
http://renderman.pixar.com/resources/current/rms/tutorialRenderManShaders.html

[92] *Reshetov A., Soupikov A., Hurley J.* Multi-level ray tracing algorithm. ACM Transactions on Graphics, 24(3) - 2005.

[93] *Reshetov A.* Faster ray packets - triangle intersection through vertex culling // Proceedings. of the IEEE/EG Symposium on Interactive Ray Tracing - 2007.

[94] *Ritschel T., Dachsbacher C., Grosch T., Kautz J.* The State of the Art in Interactive Global Illumination // Computer Graphics Forum Vol. 31, no. 1. – 2012.

[95] *Roger D., Assarsson U., Holzschuch N.* Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU // Proceedings of Eurographics Symposium on Rendering - 2007.

[96] *Sander P., Snyder J., Gortler S., Hoope H.* Texture Mapping Progressive Meshes // ACM Transactions on Graphics (Proceedings SIGGRAPH'01) - 2001.

[97] *Satish N., Harris M., Garland M.* Designing efficient sorting algorithms for manycore GPUs // Proceedings 23rd IEEE International Parallel and Distributed Processing Symposium - 2009.

[98] *Segovia B., Ernst M.* Memory efficient ray tracing with hierarchical mesh quantization // Proceedings of Graphics Interface - 2010.

[99] *Sengupta S., Harris M., Garland M.* Efficient parallel scan algorithms for GPUs // NVIDIA Technical Report NVR-2008-003 (December) - 2008.

[100] *Shevtsov M., Soupikov A., Kapustin A.* Parallel kd-tree construction for interactive ray tracing of dynamic scenes // Computer Graphics Forum 26(3) - 2007.

[101] *Sony Pictures Imageworks.* Open Shading Language
http://opensource.imageworks.com/?p=osl

[102] *Sopin D., Bogolepov D., Ulyanov D.* Real-time SAH BVH construction for ray tracing dynamic scenes // Proceedings of the 21th International Conference on Computer Graphics and Vision GraphiCon - 2011.

[103] *Shirley P.* Realistic Ray Tracing // AK Peters, Ltd. - 2000.

[104] *Stich, M., Friedrich, H., Dietrich, A.* Spatial splits in bounding volume hierarchies // Proceedings of High-Performance Graphics - 2009

[105] *University North Carolina.* Dynamic Scenes: http://gamma.cs.unc.edu/RT

[106] *Veach E.* Robust Monte Carlo Methods for Light Transport Simulation // PhD thesis / Stanford University - 1997.

[107] *Wachter C., Keller A.* Instant ray tracing: The bounding interval hierarchy // Proceedings of the 17th Eurographics Symposium on Rendering - 2006.

[108] *Wald I., Dietrich A., Slusallek P.* An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models // Proceedings of the Eurographics Symposium on Rendering - 2004.

[109] *Wald I., Havran V.* On building fast kd-trees for ray tracing, and on doing that in O(N log N) // Proceedings of the IEEE Symposium on Interactive Ray Tracing - 2006.

[110] *Wald I., Mark W. R., Gunther J., Boulos S., Ize T., Hunt W., Parker S. G., Shirley P.* State of the Art in Ray Tracing Animated Scenes // Eurographics 2007 State of the Art Reports, Eurographics - 2007.

[111] *Wald I., Bentin C., Boulos S.* Getting Rid of Packets – Efficient SIMD Single-Ray Traversal using Multi-branching BVH // Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing - 2008.

[112] *Wald I., Boulos S., Shirley P.* Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies // ACM Transactions on Graphics 26, 1 - 2007.

[113] *Wald I.* On fast Construction of SAH based Bounding Volume Hierarchies // Proceedings of the EG/IEEE Symposium on Interactive Ray Tracing - 2007.

[114] *Wald I.* Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture // IEEE Transactions on Visualization and Computer Graphics - 2010.

[115] *Wald I.* Active Thread Compaction for GPU Path Tracing // Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics - 2011.

[116] *Walter B., Bala K., Kulkarni M., Pingali K.* Fast Agglomerative Clustering for Rendering // Proceedings. of the IEEE/EG Symposium on Interactive Ray Tracing - 2008.

[117] *Walter B., Marschner S., Li H., Torrance K.* Microfacet Models for Refraction through Rough Surfaces // Proceedings of the Eurographics Symposium on Rendering - 2007.

[118] *Weidlich A., Alexander W.* Arbitrarily Layered Micro-Facet Surfaces // ACM GRAPHITE'07 Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia – 2007.

[119] *Whitted T.* An improved Illumination Model for Shaded Display // Communications of the ACM 23, 6 - 1980.

[120] *Yoon S.-E., Lauterbach C., Manocha D.* R-lods: fast lod-based ray tracing of massive models // The Visual Computer 22 (September) - 2006.

[121] *Yoon S.-E., Curtis S., Manocha D.* Ray Tracing Dynamic Scenes using Selective Restructuring // Proceedings of the Eurographics Symposium on Rendering - 2007.

[122] *Yoon S., Lindstrom P.* Random-accessible compressed triangle meshes - 2007.

[123] *Zhou K., Hou, Q.,Wang, R., Guo, B.* Real-time KD-tree construction on graphics hardware // ACM Transactions on Graphics 27(5) - 2008.

[124] *Zhou K., Hou Q., Ren Z., Gong M., Sun X., Guo, B.* Renderants: interactive reyes rendering on gpus // Proceedings of ACM SIGGRAPH Asia - 2009.