Министерство образования и науки Российской Федерации
Государственное образовательное учреждение
высшего профессионального образования
«Нижегородский государственный университет им. Н.И. Лобачевского»
Факультет вычислительной математики и кибернетики
Кафедра математического обеспечения ЭВМ

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Трассировка лучей в реальном времени на x64 архитектуре

Допущена к защите : _____
Заведующий кафедрой МО ЭВМ
д.ф.–м.н., проф.

_____ Стронгин Р. Г.
Подпись

«_____» _____ 2011 г.

Выполнил: студент группы 86М1

_____ Морозов А. С.
Подпись

Научный руководитель:
д. т. н., профессор кафедры МО ЭВМ

_____ Турлапов В. Е.
Подпись

Нижний Новгород
2011 г.

Содержание

Введение

В киноиндустрии к современной компьютерной графике предъявляются серьезные требования физически корректного моделирования освещения сцен. Каждая из них состоит из множества примитивов с различными характеристиками, которые по разному взаимодействуют со светом. Даже малейшие неточности, могут отбросить художественный или анимационный фильм в рубрику любительского кино, и при этом не принести ожидаемой прибыли. Особенные требования предъявляются именно к художественному фильму, т. к. используемые спецэффекты должны выглядеть настолько реалистично, чтобы зритель не смог различить, где настоящий актер, а где рисованный двойник, выполняющий невероятные трюки. Используя только физически правильные модели и алгоритмы, можно обеспечить растущую потребность в более реалистичной трехмерной графике.

С выходом нового фильма каждый из нас видит прогресс в компьютерной графике. Картинка становится все красочнее и правдоподобнее, но все это не даётся просто так. Естественно, платить за это приходится высокой вычислительной трудоемкостью расчетов. Несомненно, что с каждым годом производительность вычислительной техники растет, но она сразу ”расходуются” на новые спецэффекты. Существует наблюдение, которое гласит, что время расчета одного кадра не изменяется. Среднее время расчета полного фильма 15 лет назад занимал около 10-12 месяцев, так и сегодня, тратят столько же времени, хотя при этом, надо заметить, что производительность современных компьютеров в сотни раз превышает производительность компьютеров того времени. С развитием вычислительной техники растут требования к самому изображению. Если несколько лет назад картинка с разрешением 1024x768 считалась излишеством в компьютерной графике, то уже сейчас это слишком мало и все считают де-факто FullHD[1]. В последний год компьютерная индустрия, дабы не потерять зрителя, начала использовать новые технологии – 3D, которая требует еще большей вычислительной мощности.

Именно за последние несколько лет компьютеры стали по настоящему параллельными. Появились многоядерные процессоры. И именно по этому, что 15 лет назад было трудоемкой задачей рендеринга, то сейчас это можно получить почти в реальном времени при том же качестве результата.

[1]FullHD – это разрешение экрана 1920x1080 пикселей

1. Постановка задачи

Главной целью данной работы является исследование и реализация алгоритма трассировки лучей на архитектуре x64. Для решения главной задачи требуется решить ряд следующих подзадач:

- Реализовать высокопроизводительный алгоритм трассировки лучей на центральном процессоре;
- Реализация и исследование оптимизированной версии с использованием векторных расширений архитектуры x64;
- Реализация и исследование специализированного класса векторов для алгоритма трассировки лучей основанного на технологии шаблонных выражений с применением векторных оптимизаций – SIMD[2] инструкции, что должно дать хорошую скорость работы приложения без потери качества восприятия кода;
- Реализация параллельной версии алгоритма трассировки лучей с использованием OpenMP, TBB;
- Сравнение параллельной версии алгоритма трассировки лучей с использованием библиотеки TBB и OpenMP на многоядерном процессоре с технологией HT[3];
- Алгоритмическая оптимизация : реализация ускоряющей структуры;
- Сравнение реализации алгоритма с использованием ускоряющей структуры и без нее.

В качестве основного языка программирования выбирается язык C++, а для отображения результатов — кроссплатформенная библиотека SDL.

[2]Single Instruction, Multiple Data — Одна Инструкция, Много Данных
[3]HT - Hyper-Threading или Гиперпоточность

2. Архитектура центрального процессора

Для того, что бы ответить на вопрос, почему же была выбрана архитектура x86-64 для написание столь сложного и трудоемкого приложения, необходимо рассмотреть её основные особенности.

Одни из базовых понятий для производительности процессора:

- Количество тактов процессора, затрачиваемое на обработку инструкции пока она проходит все стадии в процессоре (Latency)
- Количество тактов центрального процессора, необходимое для принятия следующей инструкции на обработку (Throughput)

В процессоре используется различный уровень параллелизма:

- Суперскалярность — имеется несколько исполнительных блоков;
- SIMD (Single Instruction, Multiple Data) – параллельная обработка по данным;
- Конвеерность;
- SMT (Simultaneous Multi-Threading) – одновременное использование ресурсов несколькими процессами, конвеер испульзоется лучше, вычислительные блоки не простаивают;
- SMP – Symmetric Multi-Processing (многоядерные процессоры).

2.1. Архитектура процессора Pentium 4

Intel Pentium 4 – это одноядерный x86 - совместимый микропроцессор компании Intel, представленный 20 ноября 2000 года.

В основе архитектуры любого процессора лежат несколько обязательных конструктивных элементов: кэш команд и данных, предпроцессор и блоки исполнения команд.

Процесс обработки данных состоит из нескольких характерных этапов. Сначала инструкции и данные забираются из кэша, который разделен на кэш данных и кэш инструкций, – эта процедура называется выборкой. Затем выбранные из кэша инструкции декодируются в понятные для данного процессора примитивы – микроинструкции (uops), и называется данная процедура декодированием. Далее декодированные команды поступают на исполнительные блоки процессора, где и выполняются, а результат записывается в оперативную память.

Процессы выборки инструкций из кэша, их декодирование и продвижение к исполнительным блокам осуществляются в предпроцессоре, а процесс выполнения декодированных команд — в блоке исполнения команд. Таким образом, даже в самом простейшем случае команда проходит как минимум четыре стадии обработки:

- выборка из кэша;

- декодирование;
- выполнение;
- запись результатов.

Указанные стадии принято называть конвейером обработки команд. В простейшем случае конвейер является четырехступенчатым, и каждую из этих ступеней команда должна проходить ровно за один такт. Для четырехступенчатого конвейера на выполнение одной команды соответственно отводится ровно четыре такта. В реальных процессорах конвейер обработки команд может быть более сложным и включать большее количество ступеней. Собственно говоря, отличительной особенностью процессоров семейства Intel Pentium 4 и является их беспримерно длинный конвейер. Так, в процессорах на ядре Northwood длина конвейера составляла 20 ступеней, а в новом процессоре Prescott она увеличена до 31 ступени. Причина увеличения длины конвейера заключается в том, что поскольку многие команды являются довольно сложными и не могут быть выполнены за один такт процессора, особенно при высоких тактовых частотах, то каждая из четырех стадий обработки команд (выборка, декодирование, выполнение, запись) должна состоять из нескольких ступеней конвейера. Кроме того, в конвейер преднамеренно вставляются так называемые пустые ступени (Drive), на которых не происходит обработка инструкции.

Эти пустые (или передаточные) ступени необходимы для того, чтобы при высоких тактовых частотах сигнал успевал во время одного такта распространится от одного исполнительного блока к другому. Напомним, что при частотах свыше 3 ГГц время одного такта составляет менее 3 нс. За столь короткий промежуток времени свет в вакууме успевает пройти расстояние менее 1 см, а поскольку скорость распространения сигналов в кристалле существенно ниже скорости света, то при высоких тактовых частотах неизбежно приходится вводить пустые ступени конвейера для передачи сигнала.

Всякий процессор в конечном счете должен быть сконструирован таким образом, чтобы за минимальное время выполнять максимальное количество инструкций. Именно количество выполняемых за единицу времени инструкций и определяет производительность процессора.

Существует два принципиально различных способа повышения производительности процессора (не считая, конечно, увеличения тактовой частоты). Суть первого состоит в том, чтобы увеличивать количество исполнительных блоков — таким образом реализуется множество параллельных коротких конвейеров. Данный подход позволяет в полной мере реализовать параллелизм на уровне инструкций (Instruction-Level Parallelism, ILP), когда несколько инструкций выполняются одновременно в различных исполнительных блоках процессора. Количество ступеней конвейера здесь невелико, поэтому инструкции выполняются за небольшое количество циклов.

Для реализации параллелизма на уровне инструкций необходимо, чтобы поступающие на исполнительные блоки команды можно было выполнять параллельно. Однако если, к приме-

ру, для выполнения следующей по порядку инструкции требуется знать результат выполнения предыдущей инструкции (подобные инструкции называются взаимозависимыми), то в этом случае параллельное выполнение невозможно. Поэтому препроцессор прежде всего проверяет взаимозависимость команд и переупорядочивает их — не в порядке поступления (Out of Order), а так, чтобы их можно было выполнять параллельно. На последних ступенях конвейера инструкции выстраиваются в исходном порядке.

При коротком конвейере на каждой ступени процессор способен выполнять большее количество работы, однако на прохождение инструкции через каждую ступень конвейера здесь затрачивается больше времени, что ограничивает повышение тактовой частоты процессора. В этой ситуации увеличение числа команд, выполняемых за единицу времени, достигается за счет распараллеливания инструкций и наращивания исполнительных блоков процессора.

При использовании длинного конвейера возможно увеличение тактовой частоты процессора, то есть сам конвейер оказывается более быстрым. Применение длинных конвейеров с высокими тактовыми частотами процессора — это второй способ увеличения производительности процессора, и именно такая идеология длинного конвейера заложена в архитектуре процессора Intel Pentium 4. При использовании длинного конвейера на стадии исполнения инструкций задействуется меньшее количество исполнительных блоков, но каждый из них обладает длинным и соответственно быстрым конвейером. Это означает, что каждый блок исполнения (Execution Unit) имеет больше доступных для выполнения тактов и способен одновременно выполнять довольно много инструкций.

Этот метод имеет, однако, свои подводные камни. Дело в том, что в случае длинного конвейера предпроцессору необходимо обеспечивать ему соответствующую загрузку. Для этого предпроцессор должен обладать довольно большим буфером, способным вмещать достаточное количество инструкций. Если же в кэше отсутствует инструкция или данные для конвейера, то образуются так называемые конвейерные пузырьки (Pipeline Bubbles), которые проходят все ступени конвейера, но ни на одной из них не производятся никакие действия. Наличие Pipeline Bubbles негативно отражается на производительности процессора, поскольку ресурсы процессора просто-напросто простаивают. Избежать возникновения нежелательных простоев в процессорах позволяют различные хитроумные алгоритмы, например Hyper-Threading. Как уже отмечалось выше, новый процессор Prescott имеет необычайно длинный конвейер — 31 ступень, что на 11 степеней больше, чем в процессоре Northwood. При этом архитектура Intel NetBurst, заложенная в процессоре, не претерпела существенных изменений. Структурная схема процессора изображена на рис 1.

При работе процессора инструкции выбираются из кэша L2 и декодируются. Кэш L2 процессоров семейства Pentium 4 под названием Advanced Transfer Cache, имеет 256-битную шину, работающую на частоте ядра, и усовершенствованную схему передачи данных, кэш обеспечивает высочайшую пропускную способность, столь важную для потоковых процессов обработки.

Для выборки команд из кэша L2 и их последующего декодирования в микрооперации от-

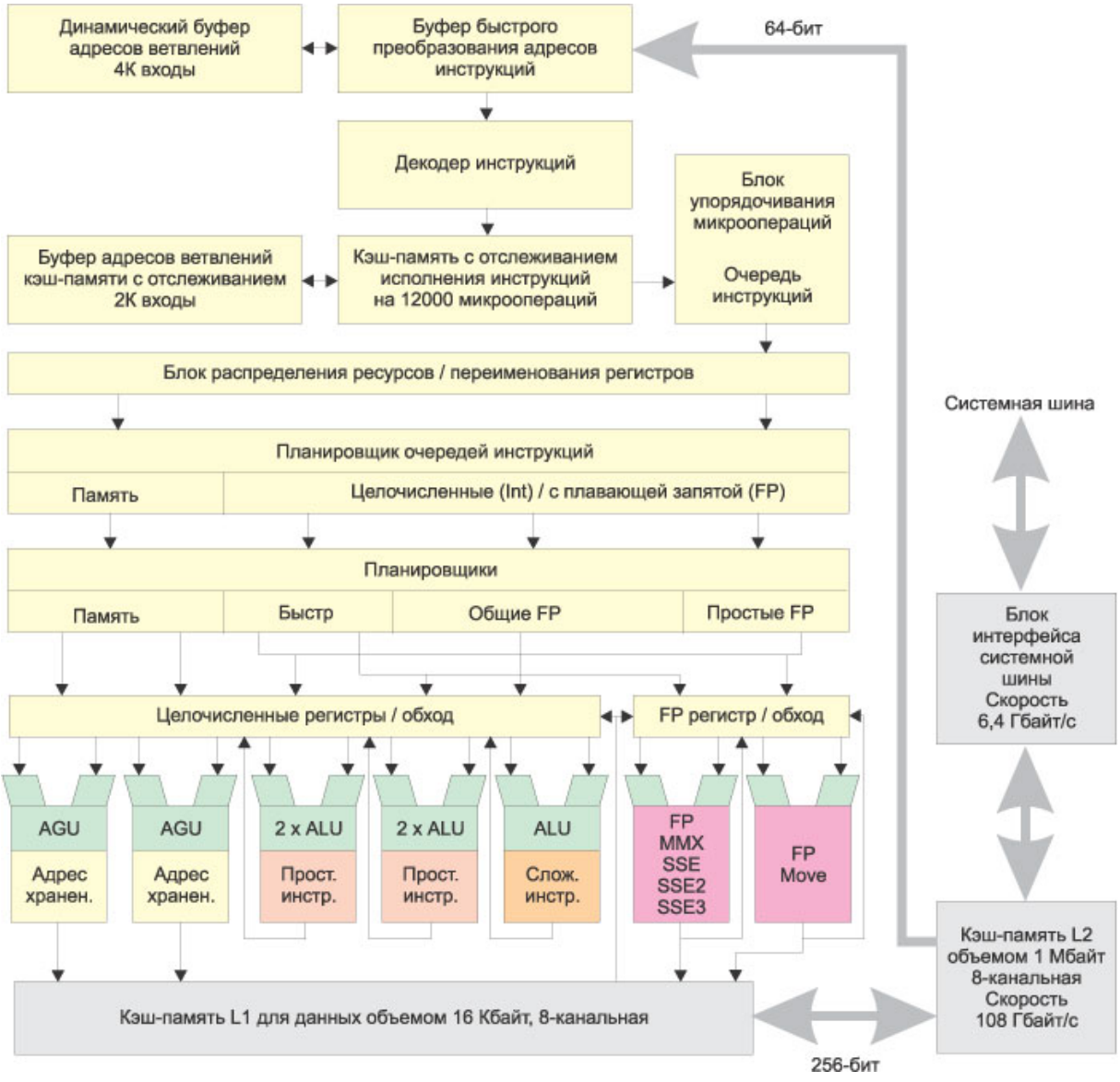

Рис. 1: Архитектура процессора Pentium 4

водится несколько начальных ступеней конвейера. Соответственно при выполнении фрагмента программного кода для декодирования команд будет использовано несколько процессорных тактов. Однако во многих современных (прежде всего мультимедийных) приложениях один и тот же фрагмент кода может повторяться многократно, и было бы нерационально тратить процессорные такты на повторную выборку, транслирование и декодирование. Выгоднее хранить уже готовые к исполнению микроинструкции в специальном кэше L1, где из них формируются мини-программы, называемые отслеживаниями (Traces). Каждая такая программа может содержать до 6 декодированных инструкций uops. Мини-программы формируются из инструкций, которые выполняются последовательно (именно поэтому они и называются отслеживаниями). При этом в самом программном коде указанные инструкции могут не следовать друг за другом, то есть реализуется внеочередное выполнение инструкций (Out-of-Order). При попадании в кэш L1 происходит внеочередное выполнение команд; при этом значительно экономятся ресурсы процессора, так как по своей сути внеочередное выполнение команд подразумевает устранение первых ступеней конвейера, фактическая длина которого в этом случае составляет уже 31 ступень. В кэше с отслеживанием может храниться до 12 тыс. декодированных микрокоманд.

Режим работы процессора при внеочередном выполнении команд (то есть когда происходит попадание в Trace Cache и используются уже декодированные команды) является естественным для процессора Intel Pentium 4. Поэтому, говоря о длине конвейера в 31 ступень, мы имеем в виду длину основного конвейера — без учета первых ступеней, которые используются при необходимости выборки команд, их трансляции, декодирования и сохранения в Trace Cache полученных микрокоманд.

Чтобы обеспечить высокий процент попаданий в кэш L1 с отслеживаниями (Trace Cache) и построение в нем мини-программ, используется специальный блок предсказания ветвлений (Branch Targets Buffers, BTB и Instruction Translation Look-aside Buffers, I-TLB). Этот блок позволяет модифицировать мини-программы, основываясь на спекулятивном предсказании. Так, если в программном коде имеется точка ветвления, то блок предсказаний может предположить дальнейший ход программы вдоль одной из возможных ветвей и с учетом этого спекулятивного предсказания построить мини-программу.

Рассмотрим теперь процесс продвижения микроинструкций по основному конвейеру, то когда процессор работает в режиме внеочередного выполнения инструкций. В течение первых двух тактов в Trace Cache передается указатель на следующие выполняемые инструкции — это две первые ступени конвейера, называемые Trace Cache next instruction pointer. После получения указателя в течение двух тактов происходит выборка инструкций из кэша (Trace Cache Fetch) — это две следующие ступени конвейера. Затем выбранные инструкции должны быть отосланы на внеочередное выполнение. Для того чтобы обеспечить продвижение выбранных инструкций по процессору, используется еще одна дополнительная, или передаточная, ступень конвейера (Drive).

На следующих ступенях конвейера, которые называются Allocate & Rename, происходят переименование и распределение дополнительных регистров процессора. В процессоре Intel

Pentium 4 содержится 128 дополнительных регистров, которые не определены архитектурой набора команд. Переименование регистров позволяет добиться их бесконфликтного существования.

Далее формируются две очереди (Queue) микрокоманд: очередь микрокоманд памяти (Memory uop Queue) и очередь арифметических микрокоманд (Integer/Floating Point uop Queue).

На следующих ступенях конвейера происходит планирование и распределение (Schedule) микрокоманд. Планировщик (Scheduler) — это своего рода сердце ядра процессора — выполняет две основные функции: переупорядочивание микрокоманд и распределение их по функциональным устройствам. Суть переупорядочивания микрокоманд заключается в том, что планировщик определяет, какую из них уже можно выполнять и в соответствии с их готовностью меняет порядок следования. Распределение микрокоманд происходит по четырем функциональным устройствам, то есть формируются четыре очереди. Первые две из них предназначены для устройств памяти (Load/Store Unit) и формируются планировщиком Memory Scheduler из очереди памяти Mem uop Queue. Микрокоманды из очереди арифметических микрокоманд (Integer/Floating Point uop Queue) также распределяются в очереди соответствующих функциональных устройств, для чего предназначено три планировщика: Fast ALU Scheduler, Slow ALU/General FPU Scheduler и Simple FP Scheduler.

Fast ALU Scheduler — это распределитель простых целочисленных операций, который собирает простейшие микроинструкции для работы с целыми числами, чтобы затем послать их на исполнительный блок ALU, работающий на двойной скорости. В процессоре Pentium 4 имеются два исполнительных блока ALU, работающих на удвоенной скорости. К примеру, если тактовая частота процессора составляет 3,2 ГГц, то эти два устройства ALU работают с частотой 6,4 ГГц и в параллельном режиме способны выполнять четыре целочисленные операции за один такт. Такие блоки ALU получили название Rapid Execution Engine (блоки быстрого исполнения). Отметим, что в процессоре Prescott в один из быстрых блоков ALU добавлен блок Shifter/Rotator, исполняющий инструкции типа сдвига и вращения. Благодаря этому такие инструкции теперь исполняются гораздо быстрее, поскольку в предыдущих реализациях Pentium 4 сдвиг и вращение трактовались как сложные инструкции и выполнялись на медленном ALU.

2.2. Архитектура процессора Nehalem

Для того, чтобы понять, как эволюционировал центральный процессор, необходимо рассмотреть более современный процессор.

Intel Nehalem – микропроцессорная архитектура компании Intel, представленная в 4 квартале 2008 года.

x86-64 (также x64/AMD64/Intel64/EM64T) – это 64-битная аппаратная платформа (чипсет, архитектура микропроцессора и команд), разработанная компанией AMD для выполнения 64-разрядных приложений. Это расширение архитектуры x86 с полной обратной совместимостью.

Основные особенности архитектуры x64:

- 16 целочисленных 64-битных регистра общего назначения (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8 — R15);
- 8 80-битных регистров с плавающей точкой (ST0 — ST7);
- 8 64-битных регистров Multimedia Extensions (MM0 — MM7, имеют общее пространство с регистрами ST0 — ST7);
- 16 128-битных регистров SSE (XMM0 — XMM15);
- 64-битный указатель RIP и 64-битный регистр флагов RFLAGS.

По сравнению с серией Pentium процессоры Nehalem продвинулись далеко вперед по интенсивному пути, а именно не увеличивая количества вычислительных блоков (на ядро). С увеличением количества ядер на кристалле возрастала и производительность. Достичь этого удалось благодаря определению оптимальной длины конвейера, а также за счет увеличенных буферов по переупорядочиванию и буфера декодированных инструкций, ко всему прочему, увеличился процент предсказаний ветвлений и за счет подрастания тактовых частот. Увеличенный кэш второго третьего уровня позволил меньше обращаться к оперативной памяти, в результате чего скорость программ значительно увеличилась. Еще одним из не мало важных факторов, который внес существенную часть производительности в процессор – это технология компании Intel, называемая Hyper-threading. Проблема заключалось в том, что процессор, выполняя программы даже после переупорядочивания инструкций все равно простаивал по большей части. Это связано с тем, что данные в программе сильно зависимы. Данная технология помогла решить данную проблему, тем самым еще больше повысив эффективность использования вычислительных блоков(рис. 3). Суть технологии в том, что одно физическое ядро одновременно исполняет 2 потока данных.

Центральный процессор, в первую очередь, ориентировался на быстрое исполнение кода при имеющихся скромных ресурсах, в отличие от GPU, у которых изначально закладывалась мысль о параллельной обработке данных, поэтому у графической карты производительность каждого ядра очень мала, но большое количество ядер позволяют графическим ускорителям иметь большую производительность. Для примера, можно посмотреть на производительность 6 ядерного процессора Intel Core i7 990X Extreme Edition [19], у которого пиковая производительность равна 109 Gflops и графический ускоритель AMD Radeon HD 6990 [20], содержащий 3072 ядер, с пиковой производительностью равной 5100 Gflops. При перерасчете на ядро получаем, что центральный процессор имеет 18,16 Gflops/Core, а графический ускоритель всего 1,66 Gflops/Core. Таким образом, получаем, что одно ядро процессора быстрее одного ядра графической карты в 10,94 раза.

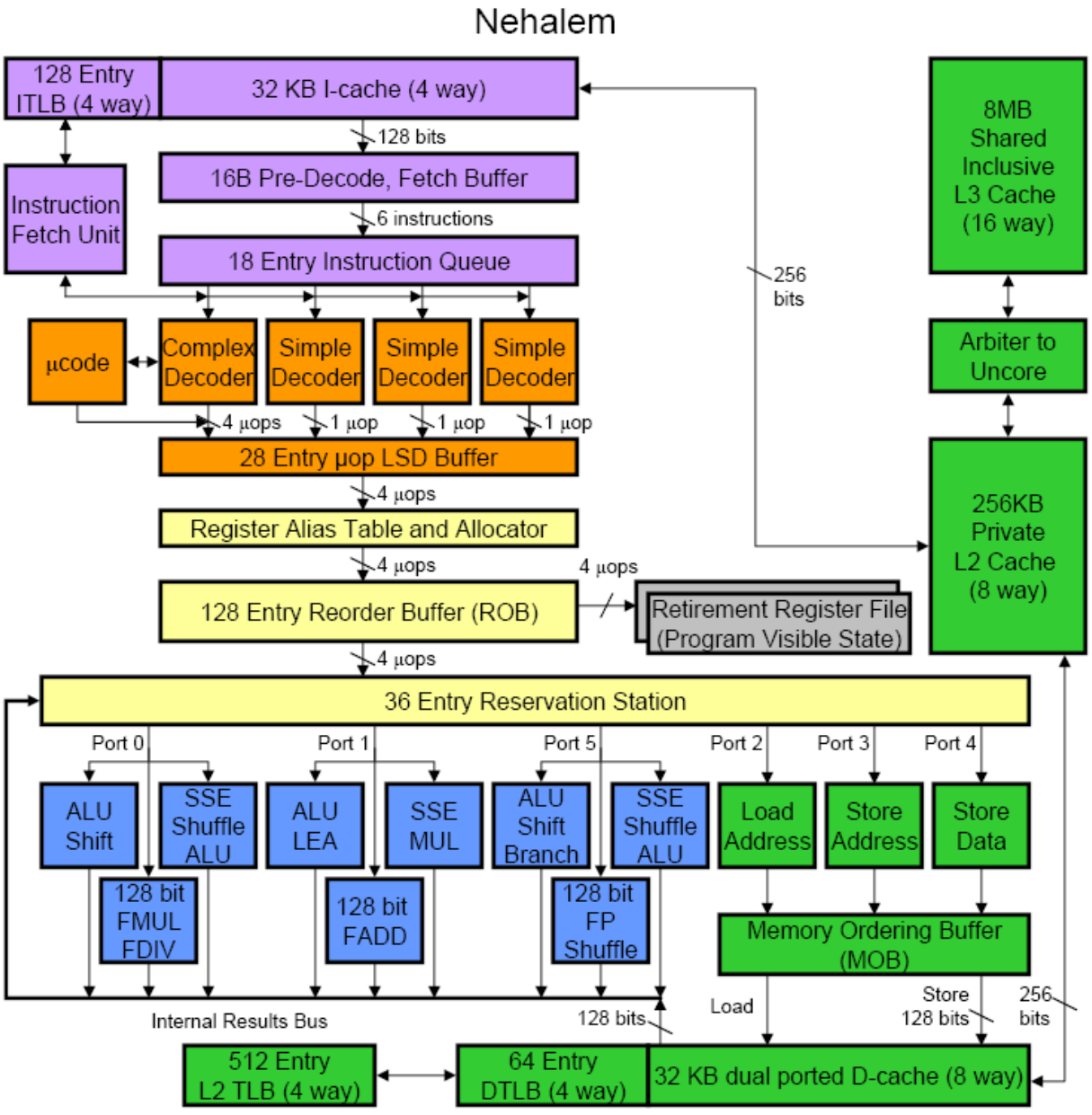

Рис. 2: Архитектура процессора Nehalem

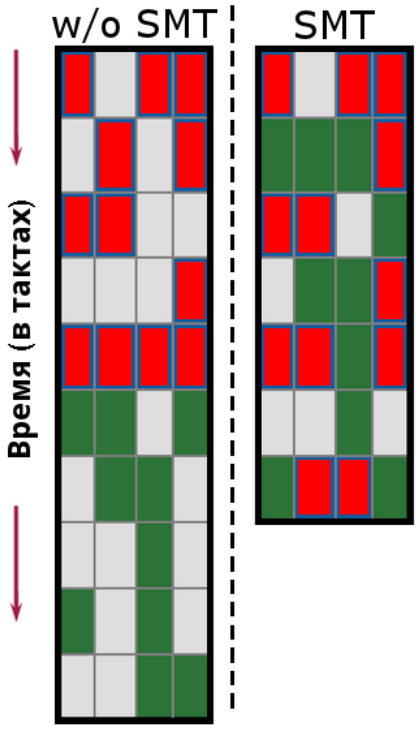

Рис. 3: Технология Simultaneous MultiThreading, улучшенная технология Intel Hyper-Threading

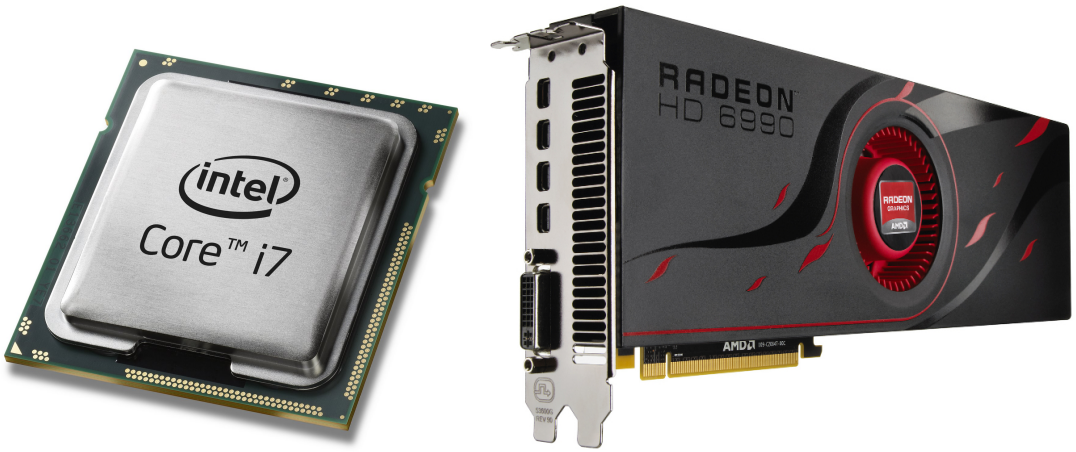

Рис. 4: Самый быстрый центральный процессор Intel Core i7 990X и самая быстрая видеокарта AMD Radeon HD 6990 на Q2 2011

3. Трассировка лучей

Классический ray tracing [9], или метод трассировки лучей, предложен Артуром Аппелем (Arthur Appel) еще в 1968 году и дополнен алгоритмом общей рекурсии, разработанным Whitted в 1980 году. Понадобилось почти 12 лет эволюции вычислительных систем, прежде чем этот алгоритм стал доступен для широкого применения в практических приложениях. Реализация высокопроизводительной версии трассировки лучей уже предпринимаются различными компаниями. О сложности задачи трассировки лучей можно прочитать в соответствующих источниках [10].

3.1. Алгоритмы трассировки лучей

Суть метода заключается в отслеживании траекторий лучей и расчета взаимодействий с лежащими на траекториях объектами, от момента испускания лучей источником света до момента попадания в камеру. Под взаимодействием луча с объектами понимаются процессы диффузного (в смысле модели локальной освещенности), многократного зеркального отражения от их поверхности и прохождение лучей сквозь прозрачные объекты. Таким образом, ray tracing – первый метод расчета глобального освещения, рассматривающий освещение, затенение (расчет тени), многократные отражения и преломления. Различают два основных вида метода трассировки лучей: ***прямой*** – forward ray tracing, и ***обратный*** – backward ray tracing.

3.1.1. Прямой метод трассировки лучей

Прямой метод трассировки лучей или forward ray tracing. В прямом методе траектории лучей строятся от источника ко всем точкам всех объектов сцены (первичные лучи). Затем проверяется ориентация каждой точки относительно источника, и, если она лежит на стороне объекта, обращенной в противоположную от источника сторону, точка из расчетов освещенности исключается. Для всех остальных точек вычисляется освещенность с помощью локальной модели освещения. Если объект не является отражающим или прозрачным, то есть поверхность объекта только диффузно рассеивает свет, траектория луча на этой точке обрывается. Если же поверхность объекта обладает свойством отражения (reflection) и/или преломления (refraction), из точки строятся новые лучи, направления которых совершенно точно определяются законами отражения и преломления.

Построенные лучи таким образом могут иметь только 3 исхода:

- Луч выходит за пределы видимости камеры. Тогда все проделанные для него до этого момента расчеты отбрасываются, поскольку они не принимают участия в формировании изображения;
- Луч попадает в камеру. Тогда рассчитанная освещенность формирует цвет соответствующего пиксела изображения;

- Луч встречает на своем пути новый объект. Тогда для новой точки пересечения повторяется расчет освещения и построения лучей отражения и преломления в зависимости от свойств поверхности объекта.

Построение новых траекторий и расчеты ведутся до тех пор, пока все лучи либо попадут в камеру, либо выйдут за пределы видимой области. Очевидно, что при прямой трассировке лучей мы вынуждены выполнять расчеты для лучей, которые не попадут в камеру, т.е. проделывать бесполезную работу. По некоторым оценочным данным доля таких ”слепых” лучей довольно велика. Эта главная, хотя и далеко не единственная причина того, что метод прямой трассировки лучей считается неэффективным и на практике не используется, по крайней мере в чистом виде.

3.1.2. Обратный метод трассировки лучей

Обратный метод трассировки лучей, или backward ray tracing. Этот метод расчетов основывается на построение лучей от наблюдателя через плоскость экрана вглубь сцены, а не от источника. Этот способ достаточно изящен, что позволяет решить массу проблем, возникающих при прямой трассировке, а сам метод отличается простотой и понятностью. Лучи теперь строятся иначе. А именно, по двум точкам: первая точка, общая для всех лучей – положение камеры (наблюдателя), вторая точка определяется положением пиксела на плоскости видового окна. Таким образом, направление каждого луча строго определено (две точки в пространстве определяют одну и только одну прямую – школьный курс геометрии), и количество первичных лучей также известно – это общее количество пикселей видового окна. Например, если видовое окно имеет 1920 пикселей по ширине и 1200 пикселей по высоте, то количество первичных лучей составит 1920 x 1200 = 2 304 000. Каждый луч вдоль заданного направления продлевается от наблюдателя вглубь трехмерной сцены, и для каждой траектории выполняется проверка на пересечение со всеми объектами сцены и с отсекающими плоскостями (рис. 5). Если пересечений с объектами нет, а есть пересечение только с плоскостью отсечения, значит луч выходит за пределы видимой части сцены, и соответствующему пикселю видового окна присваивается цвет фона. Если луч пересекается с объектами сцены, то среди всех объектов выбирается тот, который ближе всего к наблюдателю. В точке пересечения с таким объектом строится три новых, так называемых вторичных луча.

Первый луч строится в направлении источника света. Если источников несколько, строится несколько таких лучей, по одному на каждый источник. Основное назначение этого луча – определить ориентацию точки (обращена ли точка к источнику), наличие объектов, закрывающих точку от источника света. Если точка обращена в противоположную сторону от источника света или закрыта другим непрозрачным объектом, освещенность от такого источника не рассчитывается, так как точка находится в тени. В случае затеняющего прозрачного объекта интенсивность освещения уменьшается в соответствии со степенью прозрачности. Если точка закрыта от освещения всеми источниками сцены, ей присваивается фоновый (ambient) цвет. В противном случае точка освещена, интенсивность и цвет освещения рассчитываются при помощи локаль-

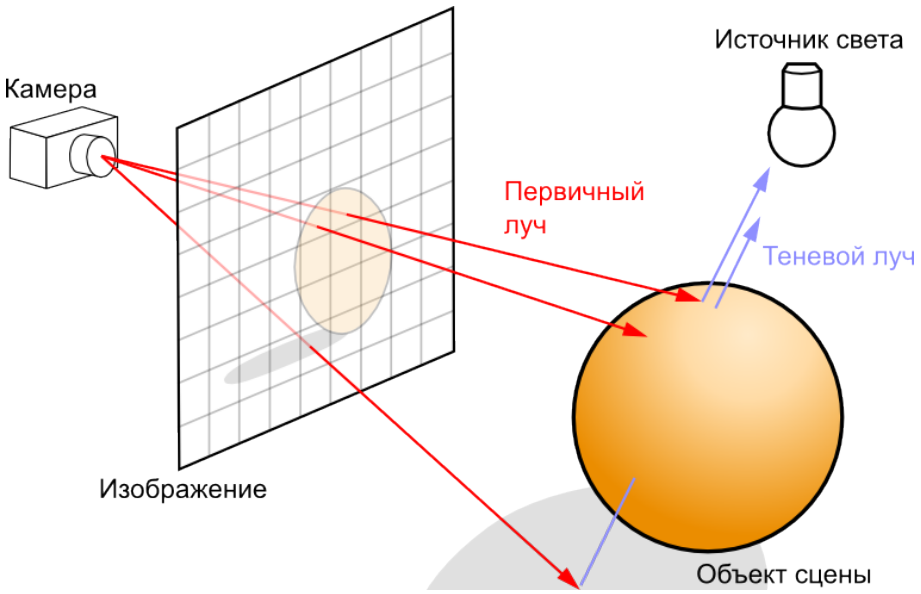

Рис. 5: Обратный метод трассировки лучей

ной модели освещенности, как сумма освещенностей от всех источников, для которых эта точка не закрыта другими объектами. Этот тип луча получил название теневой луч (shadow ray или иногда его еще называют illumination ray). Если поверхность объекта не является отражающей и непрозрачна, теневой луч – единственный тип лучей который строится, траектория первичного луча обрывается (заканчивается), и дальнейшие расчеты не выполняются. Рассчитанный цвет присваивается тому пикселю видового окна, через который проходит соответствующий первичный луч.

Второй луч строится, если поверхность объекта обладает отражающими свойствами, и называется луч отражения (reflection ray) или отраженный луч. Направление отраженного луча определяется по закону:

$$\vec{R} = \vec{I} - 2 \cdot \vec{N}(\vec{N}, \vec{I}) \tag{1}$$

где $\vec{R}$ - отраженный луч, $\vec{I}$ - падающий первичный луч, $\vec{N}$ - нормаль к поверхности в точке соударения. Для отраженного луча проверяется возможность пересечения с другими объектами сцены. Если пересечений нет, то интенсивность и цвет отраженного луча равна интенсивности и цвету фона. Если пересечение есть, то в новой точке снова строится три типа лучей – теневые, отражения и преломления. Третий луч строится, если поверхность объекта прозрачна, и носит название преломленный луч (transparency ray). Направление для преломленного луча определяется следующим образом:

$$\vec{T} = \frac{n_1}{n_2} \cdot \vec{I} - \left[\cos\alpha + \frac{n_1}{n_2} \cdot \left(\vec{N}, \vec{I}\right) \right] \cdot \vec{N}$$

$$\cos\alpha = \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \cdot \left(1 - \left(\vec{N}, \vec{I}\right)^2\right)}$$

где $\vec{T}$ - переломленный луч, n_1 - коэффициент рефракции для первой среды (в которой рас-

пространяется первичный луч), n_2 - коэффициент рефракции для второй среды прозрачного объекта.

Так же, как и в предыдущем случае, проверяется пересечение вновь построенного луча с объектами, и, если они есть, в новой точке строятся три луча.

Таким образом, для каждого первичного луча можно построить древовидную структуру. Если древовидная структура для данного луча построена, то расчет освещенности можно выполнить в следующем порядке. Для каждой ветви дерева спускаемся вдоль древовидной структуры к последнему пересечению вторичного луча и поверхности (будем дальше называть их узлами). Поскольку это последний узел в цепи, то вкладов от преломлений и отражений нет, поэтому, освещенность узла вычисляется при помощи локальной модели освещения с учетом видимости источников света для данного узла. Затем, вычисленная освещенность передается вверх по ветви к следующему ближайшему узлу. Освещенность в этом узле будет вычисляться по формуле:

$$\vec{I}_{total} = \vec{I}_{local} + K_{reflection} \cdot \vec{I}_{reflection} + K_{refraction} \cdot \vec{I}_{refraction}$$

где $\vec{I}_{total}$ - полная освещенность в точке, $\vec{I}_{local}$ - локальная освещенность в точке, вычисленная от источников освещения с помощью одной из локальной модели освещенности, $K_{reflection}$ - коэффициент, определяющий отражающие свойства поверхности, $\vec{I}_{reflection}$ - освещенность предыдущей точки, переданная вдоль ветки отражения, $K_{refraction}$ - коэффициент, определяющий преломляющие свойства поверхности $\vec{I}_{refraction}$ - освещенность предыдущей точки, переданная вдоль ветки преломления

Естественным завершением трассировки лучей является выход всех испущенных вторичных лучей за пределы видимой области и их рассеяние на чисто диффузных объектах. Результат вычислений будет наиболее точным. Но, если сцена достаточно сложна, такой расчет будет очень медленным, а в некоторых случаях и невозможным по причине ограниченности аппаратных ресурсов. Легко увидеть, что вклад освещенности от каждого нового вторичного луча очень быстро уменьшается по той простой причине, что коэффициенты свойств отражения и преломления материалов меньше единицы. Поэтому часто трассировку лучей прекращают, когда вклад от следующего узла ветви становится меньше заданной величины. Это также достаточно точный метод расчетов, который может быть использован для получения качественных результатов при определенных условиях. Наконец, для получения оценочного расчета можно оборвать трассировку лучей после выполнения заданного количества итераций, это самый быстрый и наименее точный расчет.

3.1.3. Достоинства и недостатки

Основные достоинства рекурсивного метода обратной трассировки лучей – расчет теней, многократных отражений и преломлений, значительно повысивших степень реалистичности получаемых изображений.

Основные недостатки:

- Отсутствие учета вторичного освещения от диффузно отраженного объектами света;
- Низкая скорость и высокая вычислительная стоимость расчетов – в классическом алгоритме трассировки лучей необходимо проверять на пересечение каждый луч со всеми объектами сцены, в результате от 70 до 95 процентов всего времени расчетов тратится на вычисление пересечений;
- Резкие границы цветовых переходов тени/подсветок/прозрачности;
- Aliasing – «зазубренность»(ступенчатость) линий;
- Дискретность определяющих цвет пиксела первичных лучей, т.е. одного первичного луча недостаточно для корректного определения цвета пиксела, формирующего изображение.

Однако от большинства недостатков можно избавится.

3.2. Модели освещения

В соответствии с принятым графике подходом, расчет освещенности распадается на две основные задачи. Первая – определить способ расчета освещенности в произвольной точке трехмерного пространства, решается при помощи построения обобщенной математической модели освещения (illuminating model). Вторая задача – применение illuminating model для компьютерных расчетов освещенности трехмерных объектов с конкретной геометрией и свойствами поверхности, решается при помощи так называемой модели затенения (shading model).

Моделей освещения к настоящему моменту разработано несколько. Самая первая и самая простая – локальная модель освещения. Сама модель не рассматривает процессы светового взаимодействия объектов сцены между собой, а только расчет освещенности самих объектов. Вторая – это глобальная модель освещенности (global illuminations model), она рассматривает трехмерную сцену как единую систему и пытается описывать освещение с учетом взаимного влияния объектов. В рамках этой модели рассматриваются такие вопросы, как многократное отражение и преломление света, рассеянное освещение (radiosity), каустик(caustic) и фотонные карты (photon mapping) и другие.

3.2.1. Глобальные модели освещения

Глобальное освещение (global illumination model) – это название ряда алгоритмов, используемых в 3D-графике, которые предназначены для добавления более реалистичного освещения в трёхмерные сцены. Такие алгоритмы учитывают не только свет, который поступает непосредственно от источника света, т.е. прямое освещение(model!illumination!direct), но и такие случаи, в которых лучи света от одного и того же источника, отражаются на других поверхностях сцены, т.е. непрямая освещенность(indirect illumination).

Теоретически, отражение, преломление, тень – примеры глобального освещения, потому что для их имитации необходимо учитывать влияние одного объекта на другие, в отличие от случая, когда на объект падает прямой свет. На практике, только моделирование диффузного отражения или каустики называется глобальным освещением.

Изображения, полученные в результате применения алгоритмов глобального освещения, часто кажутся более фотореалистичными, чем те, в процессе рендеринга которых, применялись алгоритмы только прямого освещения, но для просчета глобального освещения, требуется гораздо больше времени.

3.2.2. Локальные модели освещения

Существующие локальные модели освещения можно разделить на две категории. К первой категории относятся эмпирические модели. Они обычно эффективны в плане быстродействия и некоторые из них дают довольно реалистичную картинку. Они обычно не оперируют такими физическими величинами, как световая энергия или световой поток. Однако эти модели находят довольно широкое применение в областях, где не требуется точная физическая информация об освещении (например, спецэффекты в фильмах, программы для художников и дизайнеров, для рекламных целей).

Ко второй категории относятся модели, базирующиеся на физических представлениях о теории света. Изображения, полученные с использованием этих моделей, очень хорошо соотносятся с экспериментальными данными. Поэтому, эти модели находят применение там, где важна точная имитация поведения света, например, при моделирование распространения света в помещение.

3.2.3. Модель Фонга

Модель Фонга – это эмпирическая модель. В самом общем случае, в свете требований фотореалистичности, эта модель учитывает и неявное ambient-освещение. Ambient-освещение, или как его еще называют фоновым освещением (background), – это окружающее освещение объект от удаленных источников, чьи положения и характеристики не известны. Необходимость учета ambient-освещения, пусть и очень грубо, обусловлена тем, что его вклад может быть достаточно велик – до 50% от общей освещенности. В local illumination считают, что фоновое освещение задает цвет (и его интенсивность) объекта в отсутствии явных источников света или в тени и кроме того не несет никакой информации об объекте, кроме значения простого цвета, равномерно заливающего контур объекта.

Интенсивность такого освещения постоянна и равномерно распределена во всем пространстве, расчет его отражения поверхностью выполняется по формуле:

$$\vec{I}_{amb} = K_a \cdot \vec{I}_a$$

где $\vec{I}_{amb}$ - интенсивность отраженного ambient освещения, K_a - коэффициент, характеризу-

ющий отражающие свойства поверхности для ambient-освещения, $\vec{I}_a$ - исходная интенсивность ambient-света, падающего на поверхность.

Часть света от прямых источников зеркально отражается поверхностью, а остальной свет диффузно рассеивается во всех направлениях. Кроме чисто зеркального отражения, которое имеют идеально отполированные поверхности, различают так называемое glossiness или распределенное зеркальное отражение – отражение в некотором створе углов, а не на один единственный угол. Такое рассеяние света обусловлено микрорельефом (”шероховатостью”) поверхности, то есть поверхность реальных объектов не является идеально гладкой, а состоит из большого количества микровыступов и впадин, которые зеркально отражают падающий свет под разными углами. Результатом glossy-отражения является specular highlight – яркий световой блик, имеющий размер в зависимости от степени шероховатости поверхности.

Интенсивность рассеянного света зависит от угла падающего на поверхность света по закону Ламберта (Lambert):

$$\vec{I}_{diff} = K_{diff} \cdot \vec{I}_d \cdot \cos(\alpha)$$

где $\vec{I}_d$ - интенсивность падающего на поверхность света, K_{diff} - коэффициент, характеризующий рассеивающие свойства поверхности, $\cos(\alpha)$ - угол между направлением на источник света и нормалью поверхности.

Другими словами, поверхность будет освещена больше, если свет падает на нее перпендикулярно ($\alpha = 0$), и меньше, если свет падает под любым другим углом, поскольку в этом случае увеличивается освещаемая площадь. Диффузно рассеянный свет является главным источником визуальной информации о геометрии трехмерных объектов.

Как было уже сказано ранее, свет отражается зеркально в некотором створе углов, и для большинства реальных материалов мы всегда видим зеркальную подсветку в форме светового пятна, а не в форме яркой точки. Поэтому, для расчета интенсивности зеркально отраженного света используется формула, предложенная Фонгом:

$$\vec{I}_{spec} = K_{spec} \cdot \vec{I}_s \cdot \cos^n(\beta)$$

где $\vec{I}_{spec}$ - интенсивность зеркально отраженного света, $\vec{I}_s$ - интенсивность источника света, $\vec{K}_s$ - коэффициент, характеризующий свойства зеркального отражения поверхности β - угол между направлением идеального отражения и направлением на наблюдателя, степень n определяет размер пятна светового блика, чем больше n, тем меньше световой блик, и тем ближе отражающие свойства поверхности к свойствам идеального зеркала.

Формула Фонга – пример компьютерной фикции, поскольку она не имеет физического смысла. Ее используют просто потому, что она дает хорошие практические результаты.

Таким образом, локальная модель освещенности предполагает расчет отраженной фоновой освещенности, диффузного и зеркального отражения от прямых источников:

$$\vec{I}_{local} = K_{amb} \cdot \vec{I}_{amb} + K_{diff} \cdot \vec{I}_{diff} \cdot \left(\vec{L}, \vec{N}\right) + K_{spec} \cdot \vec{I}_{spec} \cdot \left(\vec{R}, \vec{V}\right)^n$$

3.3. Модель камеры

Для того, чтобы точно ориентировать камеру, необходимо указать следующие вектора:

C_d - задает вектор направления, т.е. указывает, куда смотрит камера (в локальной модели координат);

C_p - задает точку в пространстве, определяющую положение камеры (в общей модели координат);

C_u - задает вектор направления, указывая, где у камеры вверх (в локальной модели координат);

C_l - задает вектор направления, указывая, где у камеры лево (в локальной модели координат).

3.3.1. Расчет луча

Для того, чтобы построить исходящий луч, необходимо знать через какую точку (x, y) видового окна пройдет луч.

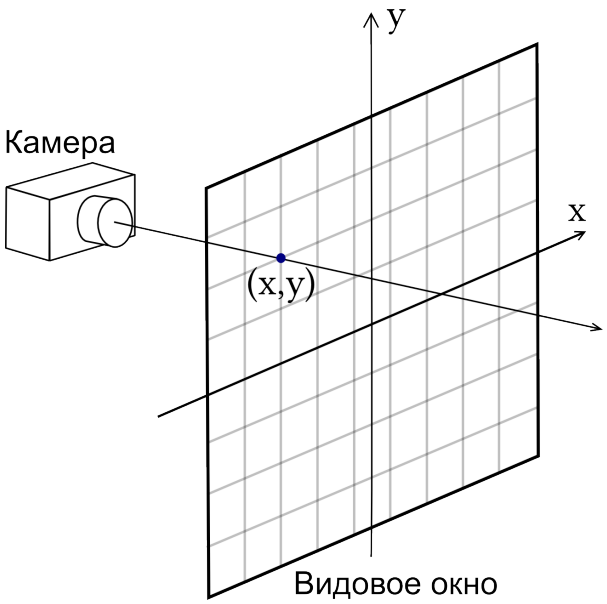

Рис. 6: Модель камеры

Расчет луча осуществляется достаточно просто. Луч определяется 2 векторами: положением и направлением. Положение луча совпадает с положением камеры. А направление вычисляется следующим образом. Пусть необходимо рассчитать луч для точки с координатами

$$(\hat{x}, \hat{y}) : 0 \leq \hat{x} < width, 0 \leq \hat{y} < height$$

Для этого преобразуем координаты $(\hat{x}, \hat{y})$ в $(x, y) : -1 \leq x \leq 1, -1 \leq y \leq 1$

$$x = 2.0 \cdot (\hat{x}/width) - 1.0$$
$$y = 1.0 - 2.0 \cdot (\hat{y}/height)$$

Учитывая соотношение сторон и угол раствора камеры:

$$aspect = width/height$$
$$tan\_av = \text{tg}(angle\_of\_view \cdot rad\_to\_angle)$$
$$x = x \cdot aspect \cdot tan\_av$$
$$y = y \cdot tan\_av$$

Для получения направляющего вектора луча, осталось взять векторы камеры с соответствующими коэффициентами:

$$dir = \text{normalize}(C_d + C_l \cdot x + C_u \cdot y)$$

3.4. Антиалиасинг

Как следует из приставки ”анти”, эта технология призвана бороться с алиасингом, т.е. со ”ступеньками”. Чтобы понять, что такое алиасинг, необходимо понять самый общий принцип вывода изображения на экран монитора. Экран состоит из миллионов очень мелких квадратов (обычно называемых точками или пикселями), примерно как бумага в клетку, только гораздо мельче. Каждый квадрат (точка, пиксель, клетка) может быть закрашена только одним цветом.

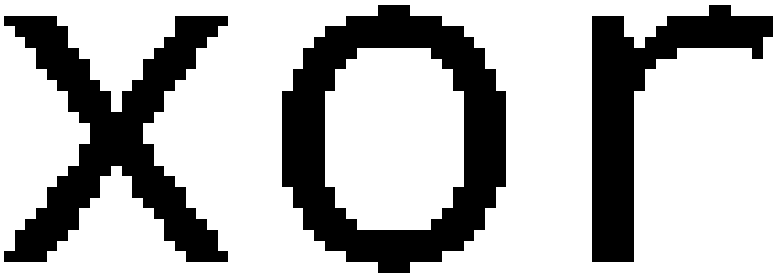

Рис. 7: Пример шрифта без антиалиасинга

Обратите внимание, что рисунок, состоящий из таких больших точек, выглядит странно. Собственно, это и есть алиасинг - на краях букв видны ”ступеньки”. Самое очевидное решение проблемы - уменьшить точки. К сожалению, экран монитора имеет очень существенный недостаток: он не позволяет сделать точки настолько малыми, чтобы взгляд не мог их различить.

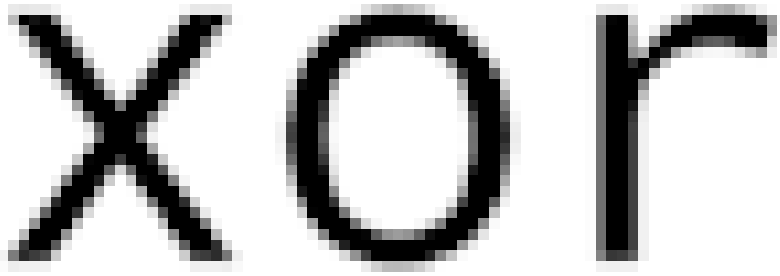

Рис. 8: Пример шрифта с антиалиасингом

Однако, при помощи плавных переходов цветов на изображении, можно очень существенно сгладить ”ступеньки”, т.е. как бы компенсировать недостаток пространственного разрешения цветовым.

Рис. 9: Сравнение шрифта с алиасингом и антиалиасингом

Сглаживание основывается на том, что каждый пиксель разбивается на несколько субпикселей. Цвет каждого пикселя определяется усреднением по какому-либо закону цветов всех субпикселей, которые находятся внутри пикселя. При этом, хотя физически разрешение остается прежним, эффективное разрешение значительно повышается. Именно на таком принципе борьбы с алиасингом основаны все современные методы антиалиасинга, которые уже давно используются в игровых ускорителях.

Два наиболее часто применяемых подхода - это ”суперсэмплинг” и ”мультисэмплинг”. Оба они основаны на том, что цвет каждого пикселя вычисляется путем смешивания цветов субпикселей (сэмплов). Но сэмплы в этих методах генерируются по-разному.

Суперсэмплинг (supersampling) – это самый простой и прямолинейный метод сглаживания. Он заключается в том, что изображение рассчитывается в виртуальном разрешении, в несколько раз превосходящем реальное экранное. После чего оно масштабируется и фильтруется до итогового разрешения. При этом цвет каждого пикселя реального разрешения вычисляется на основе нескольких субпикселей виртуального. Это позволяет значительно повысить качество изображения, но при этом нагрузка на ускоритель возрастает в несколько раз и скорость при этом, соответственно, падает. Вызвано это тем, что вместо одного цвета для пикселя приходится рассчитывать в несколько раз больше.

Мультисэмплинг – гораздо более хитрый и интеллектуальный метод сглаживания. Правильнее это называть даже не метод, а скорее инструмент. Идея заключается в том, что вычислять N субпикселей для каждого пикселя нет необходимости, так как уже рассчитанные субпиксели во многих случаях можно использовать несколько раз для формирования не одного, а нескольких результирующих пикселей. С другой стороны, в некоторых участках изображения

сглаживание не требуется вовсе, поэтому рассчитывать их по нескольким субпикселям не целесообразно. И, наоборот, в других участках нужно очень хорошее качество сглаживания и там можно рассчитать очень много субпикселей. Этот инструмент позволяет не только значительно сэкономить ресурсы ускорителя, но и получить лучшее качество сглаживания.

3.4.1. Supersampling

Несмотря на то, что мультисэмплинг является интеллектуальным методом сглаживанием, который позволяет сэкономить время вычисления, качество изображения может быть недостаточно хорошим. Если рассматривать ”сглаживание” только границ объектов, то это позволяет повысить производительность, за счет того, что не происходит вычислений вспомогательных субпикселей в случае, если это один объект. Но в данном случае, могут иметь место отражения или преломления и тогда они будут выглядить ”ступеньками”. Поэтому был выбран алгоритм суперсэмплинг.

В качестве паттерна для вычисления субпикселов может быть использована равномерная сетка, однако качество получаемого изображения не очень хорошое, поэтому предложен новый паттерн. В соответствие с данным паттерном субпикселы выбирались по кругу внутри пиксела.

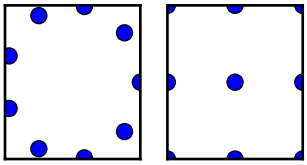

Рис. 10: Паттерны расположения субпикселов

Покажем почему данный шаблон лучше ”стандартного”.

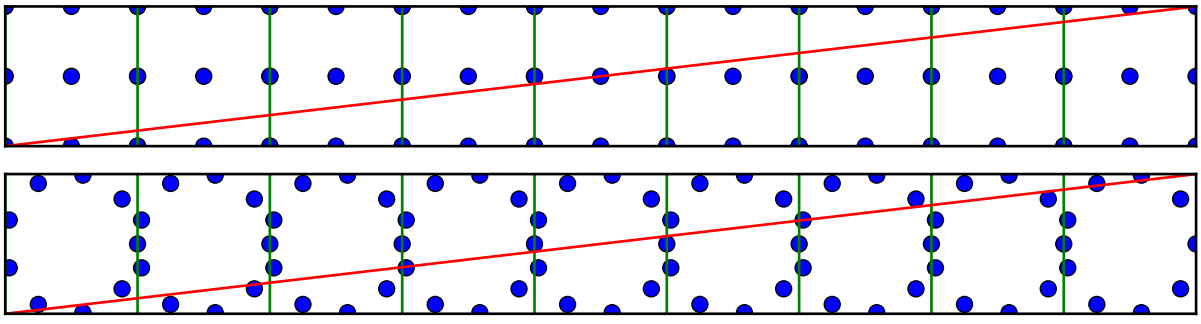

Рис. 11: Сравнение паттернов

Пусть необходимо рассчитать границу объекта, которая проходит ниже линии (см. рис. 11). В случае со стандарным шаблоном, пикселы с 1 по 4 закрасятся одним цветом, т.к. в них оди-

наковое количество субпикселов. В новом паттерне заливка будет происходить более плавно, при равном числе субпикселей.

3.4.2. Результаты работы алгоритмов сглаживания

Представлены результаты различных алгоритмов сглаживания.

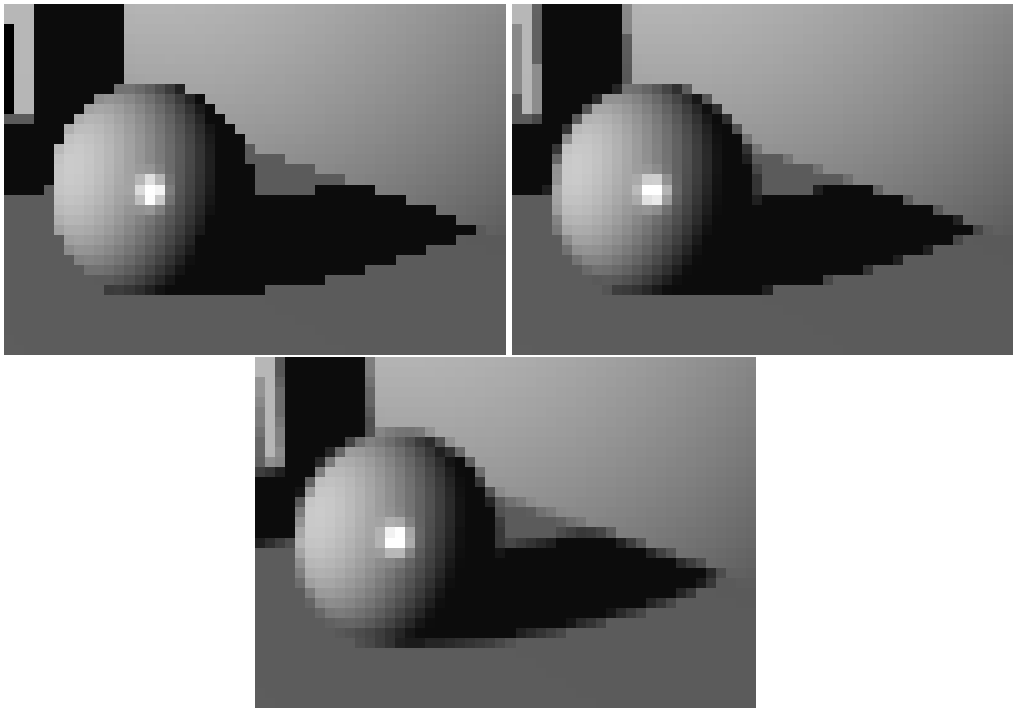

Рис. 12: Результаты сравнения паттернов
Слева на право: без сглаживания; алгоритм с равномерной сеткой; алгоритм с точками по окружности

3.5. Примитивы

3.5.1. Плоскость

Для определения пресечения луча с плоскостью, необходимо найти точку в пространстве, которая будет удовлетворять двум уравнениям: уравнению луча и уравнению плоскости.

Уравнение луча:

$$\begin{cases} x = x_p + t \cdot x_d \\ y = y_p + t \cdot y_d \\ z = z_p + t \cdot z_d \end{cases} \quad (2)$$

или

$$\vec{R}(t) = \vec{P} + t \cdot \vec{D}$$

где $\vec{P} = \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix}$ - начало луча, а $\vec{D} = \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix}$ - направление луча.

Уравнение плоскости задается следующим образом:

$$Ax + By + Cz + D = 0 \tag{3}$$

Для того, чтобы найти точку пересечения луча с плоскостью, необходимо подставить уравнение (2) в (3):

$$A(x_p + t \cdot x_d) + B(y_p + t \cdot y_d) + C(z_p + t \cdot z_d) + D = 0$$

Раскроем скобки и привидём подобные:

$$t(Ax_d + By_d + Cz_d) + Ax_p + By_p + Cz_p + D = 0$$

выразим неизвестную величину t:

$$t = -\frac{Ax_p + By_p + Cz_p + D}{Ax_d + By_d + Cz_d}$$

из уравнения видно, что луч либо пересекает плоскость в какой-то точке, либо нет. Это связано с тем, что если $Ax_d + By_d + Cz_d = 0$, то плоскость и луч параллельны друг другу. Т.к. $\vec{N} = \begin{pmatrix} A \\ B \\ C \end{pmatrix}$ - это нормаль к поверхности, а из геометрии известно, что если $(\vec{D}, \vec{P}) = 0$, то вектора параллельны.

Для того, чтобы найти величину t, необходимо рассчитать всего несколько скалярных произведений:

$$t = -\frac{(\vec{P}, \vec{N}) + D}{(\vec{D}, \vec{N})}$$

при условии, что $(\vec{D}, \vec{N}) \neq 0$.

Алгоритм нахождения точки пересечения луча и плоскости

1 **if** $(\vec{D}, \vec{N}) \neq 0$

2 **then** $t = -\frac{(\vec{P}, \vec{N}) + D}{(\vec{D}, \vec{N})}$

3 $point = \vec{P} + t \cdot \vec{D}$

3.5.2. Сфера

Для сферы необходимо проделать те же выкладки. Уравнение сферы записывается следующем образом:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2 \tag{4}$$

где $\vec{C} = \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix}$ - центр сферы, а r - радиус. Подставим уравнение (2) в (4):

$$((x_0 + t \cdot x_d) - x_c)^2 + ((y_0 + t \cdot y_d) - y_c)^2 + ((z_0 + t \cdot z_d) - z_c)^2 = r^2$$

раскроем скобки:

$$\begin{aligned}(x_p + t \cdot x_d)^2 - 2(x_p + t \cdot x_d) \cdot x_c + x_c^2 + \\ + (y_p + t \cdot y_d)^2 - 2(y_p + t \cdot y_d) \cdot y_c + y_c^2 + \\ + (z_p + t \cdot z_d)^2 - 2(z_p + t \cdot z_d) \cdot z_c + z_c^2 = r^2\end{aligned}$$

$$\begin{aligned}x_p^2 + 2x_p x_d \cdot t + x_d^2 \cdot t^2 - 2x_p x_c - 2x_d x_c \cdot t + x_c^2 + \\ + y_p^2 + 2y_p y_d \cdot t + y_d^2 \cdot t^2 - 2y_p y_c - 2y_d y_c \cdot t + y_c^2 + \\ + z_p^2 + 2z_p z_d \cdot t + z_d^2 \cdot t^2 - 2z_p z_c - 2z_d z_c \cdot t + z_c^2 = r^2\end{aligned}$$

приведем уравнение в виду:

$$a \cdot t^2 + b \cdot t + c = 0 \tag{5}$$

после раскрытия скобок и приведения подобных, получаем:

$$\begin{aligned}a &= x_d^2 + y_d^2 + z_d^2 \\ b &= 2x_p x_d + 2y_p y_d + 2z_p z_d - 2x_d x_c - 2y_d y_c - 2z_d z_c \\ c &= x_p^2 + y_p^2 + z_p^2 - 2x_p x_c - 2y_p y_c - 2z_p z_c + x_c^2 + y_c^2 + z_c^2 - r^2\end{aligned}$$

упростим:

$$\begin{aligned}a &= (\vec{D}, \vec{D}) \\ b &= 2x_d(x_0 - x_c) + 2y_d(y_0 - y_c) + 2z_d(z_0 - z_c) = 2 \cdot (\vec{D}, \vec{P} - \vec{C}) \\ c &= (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2 = 2 \cdot (\vec{P} - \vec{C}, \vec{P} - \vec{C})\end{aligned}$$

перепишем в векторном виде

$$\begin{aligned}a &= (\vec{D}, \vec{D}) \\ b &= 2 \cdot (\vec{D}, \vec{P} - \vec{C}) \\ c &= 2 \cdot (\vec{P} - \vec{C}, \vec{P} - \vec{C}) - r^2\end{aligned}$$

Если уравнение (5) не имеет вещественных решений, то луч не пересекает сферу. Если имеется два решения, то наименьший положительный корень этого уравнения определит на луче ближайшую точку пересечения луча со сферой.

Далее решаем обыкновенное квадратное уравнение, находим корни и получаем значение t, при условие, что $a = (\vec{D}, \vec{D}) \neq 0$

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

3.5.3. Треугольник

Алгоритм пересечения луча и треугольника основан на барицентрических координатах.

Барицентрические координаты – координаты точки n-мерного аффинного пространства A^n, отнесенные к некоторой фиксированной системе из $(n+1)$-ой точки $p_0, p_1, \ldots, p_n$, не лежащих в $(n-1)$-мерном подпространстве. Пусть z есть произвольная точка в A^n. Каждая точка $x \in A^n$ может быть единственным образом представлена в виде суммы:

$$x = z + \alpha_1 \cdot z\vec{p_1} + \alpha_2 \cdot z\vec{p_2} + \cdots + \alpha_n \cdot z\vec{p_n}$$

где $\alpha_1, \alpha_2, \ldots, \alpha_n$ вещественные числа, удовлетворяющие условию:

$$\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1$$

Числа $\alpha_1, \alpha_2, \ldots, \alpha_n$ называются барицентрическими координатами точки x. Не трудно заметить, что барицентрические координаты не зависят от выбора z. Точка $T(u, v)$, принадлежащая треугольнику, может быть записана в виде:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2 \tag{6}$$

где (u, v) – это бариецентрические координаты такие, что $u \geq 0$, $v \geq 0$ и $u + v \leq 1$, а V_0, V_1, V_2 - это точки пространства, образующие треугольник. Вычисление пересечения между лучем (2) и треугольником (6), это решение следующего уравнения:

$$\vec{P} + t \cdot \vec{D} = (1 - u - v)\vec{V_0} + u\vec{V_1} + v\vec{V_2}$$

после очевидных преобразований:

$$\begin{aligned} \vec{P} + t \cdot \vec{D} &= \vec{V_0} - u\vec{V_0} - v\vec{V_0} + u\vec{V_1} + v\vec{V_2} \\ \vec{P} - \vec{V_0} &= -t\vec{D} + u\vec{V_1} - u\vec{V_0} + v\vec{V_2} - v\vec{V_0} \\ -t\vec{D} + u(\vec{V_1} - \vec{V_0}) + v(\vec{V_2} - \vec{V_0}) &= \vec{P} - \vec{V_0} \end{aligned}$$

получаем:

$$\left[-\vec{D}, \vec{V_1} - \vec{V_0}, \vec{V_2} - \vec{V_0}\right] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \vec{P} - \vec{V_0} \tag{7}$$

Чтобы решить задачу, необходимо найти вектор $\begin{pmatrix} t \\ u \\ v \end{pmatrix}$. Обозначив $\vec{E_1} = \vec{V_1} - \vec{V_0}$, $\vec{E_2} = \vec{V_2} - \vec{V_0}$ и $\vec{T} = \vec{P} - \vec{V_0}$, решим уравнение (7), используя метод Крамера:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\vec{D}, \vec{E_1}, \vec{E_2}|} \begin{bmatrix} | & \vec{T}, & \vec{E_1}, & \vec{E_2} & | \\ | & -\vec{D}, & \vec{T}, & \vec{E_2} & | \\ | & -\vec{D}, & \vec{E_1}, & \vec{T} & | \end{bmatrix} \tag{8}$$

Из курса линейной алгебры известно, что: $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$. Принимая во внимания этот факт, перепишем уравнение (8).

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\vec{D} \times \vec{E_2}) \cdot \vec{E_1}} \begin{bmatrix} (\vec{T} \times \vec{E_1}) \cdot \vec{E_2} \\ (\vec{D} \times \vec{E_2}) \cdot \vec{T} \\ (\vec{T} \times \vec{E_1}) \cdot \vec{D} \end{bmatrix} = \frac{1}{(\vec{S}, \vec{E_1})} \begin{bmatrix} (\vec{Q}, \vec{E_2}) \\ (\vec{S}, \vec{T}) \\ (\vec{Q}, \vec{D}) \end{bmatrix} \tag{9}$$

где $\vec{S} = (\vec{D} \times \vec{E_2})$ и $\vec{Q} = (\vec{T} \times \vec{E_1})$

На этом можно остановится, однако можно заметить, что $(E_1 \times E_2)$ – это нормаль к треугольнику, которую можно заранее предвычислить. Запишем решение по другому:

$$\begin{aligned} |-\vec{D}, \vec{E_1}, \vec{E_2}| &= -\vec{D} \cdot (\vec{E_1} \times \vec{E_2}) = -\vec{D} \cdot \vec{N} \\ |\vec{T}, \vec{E_1}, \vec{E_2}| &= \vec{T} \cdot (\vec{E_1} \times \vec{E_2}) = \vec{T} \cdot \vec{N} \\ |-\vec{D}, \vec{T}, \vec{E_2}| &= (\vec{T} \times -\vec{D}) \cdot \vec{E_2} = \vec{\hat{Q}} \cdot \vec{E_2} \\ |-\vec{D}, \vec{E_1}, \vec{T}| &= -|-\vec{D}, \vec{T}, \vec{E_1}| = -(-\vec{D} \times \vec{T}) \cdot \vec{E_1} = -\vec{\hat{Q}} \cdot \vec{E_1} \end{aligned}$$

где $\vec{\hat{Q}} = -\vec{D} \times \vec{T}$.

Тогда вычисление по формуле (9) можно переписать так:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(-\vec{D} \cdot \vec{N})} \begin{bmatrix} (\vec{T}, \vec{N}) \\ (\vec{\hat{Q}}, \vec{E_2}) \\ (-\vec{\hat{Q}}, \vec{E_1}) \end{bmatrix}$$

где $\vec{E_1} = \vec{V_1} - \vec{V_0}$, $\vec{E_2} = \vec{V_2} - \vec{V_0}$, $\vec{T} = \vec{P} - \vec{P}$, $\vec{N} = (\vec{E_1} \times \vec{E_2})$, $\vec{\hat{Q}} = (-\vec{D} \times \vec{T})$.

Таким образом, можно избавится от одной операции векторного умножения (фактически она осталась, но мы можем ее подсчитать заранее). Для того, что бы еще улучшить данный

алгоритм и избавится от знака минус, произведем несколько замен:

$$\vec{\tilde{E}_1} = -\vec{E_1} = \vec{V_0} - \vec{V_1}$$
$$\vec{\tilde{E}_2} = \vec{E_2} = \vec{V_2} - \vec{V_0}$$
$$\vec{\tilde{T}} = -\vec{T} = \vec{V_0} - \vec{P}$$
$$\vec{\tilde{N}} = -\vec{N} = -(\vec{E_2} \times \vec{E_1}) = (\vec{E_2} \times -\vec{E_1}) = (\vec{\tilde{E}_2} \times \vec{\tilde{E}_1})$$
$$\vec{\tilde{Q}} = -\vec{D} \times \vec{T} = -(\vec{D} \times \vec{T}) = -(\vec{D} \times (-\vec{\tilde{T}})) = (\vec{D} \times \vec{\tilde{T}})$$
$$-\vec{D} \cdot \vec{N} = (-\vec{D} \cdot (-\vec{\tilde{N}})) = (\vec{D} \cdot \vec{\tilde{N}})$$
$$\vec{T} \cdot \vec{N} = (-\vec{\tilde{T}}) \cdot (-\vec{\tilde{N}}) = (\vec{\tilde{T}} \cdot \vec{\tilde{N}})$$
$$\vec{\tilde{Q}} \cdot \vec{E_2} = (-\vec{D} \times \vec{T}) \cdot \vec{E_2} = (\vec{\tilde{T}} \times \vec{D}) \cdot \vec{\tilde{E}_2}$$
$$-\vec{\tilde{Q}} \cdot \vec{E_1} = (\vec{\tilde{T}} \times \vec{D}) \cdot \vec{\tilde{E}_1}$$

В результате получаем следующие решение:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\vec{D} \cdot \vec{\tilde{N}})} \begin{bmatrix} (\vec{\tilde{T}} \cdot \vec{\tilde{N}}) \\ (\vec{\tilde{S}} \cdot \vec{\tilde{E}_2}) \\ (\vec{\tilde{S}} \cdot \vec{\tilde{E}_1}) \end{bmatrix} \tag{10}$$

где $\vec{\tilde{E}_1} = \vec{V_0} - \vec{V_1}$, $\vec{\tilde{E}_2} = \vec{V_2} - \vec{V_0}$, $\vec{\tilde{T}} = \vec{V_0} - \vec{P}$, $\vec{\tilde{S}} = \vec{\tilde{T}} \times \vec{D}$, $\vec{\tilde{N}} = (\vec{\tilde{E}_2} \times \vec{\tilde{E}_1})$.

Алгоритм нахождения точки пересечения луча и треугольника

1 Вычисляем по формуле (10) вектор $\begin{bmatrix} t \\ u \\ v \end{bmatrix}$

2 **if** $(u \geq 0)$ && $(v \geq 0)$ && $(u + v \leq 1)$

3 **then** $point = \vec{P} + t \cdot \vec{D}$

Листинг 1. Реализация алгоритма пересечения луча и треугольника

```
01: float Triangle::crossing(const Ray & r)
02: {
03:     vec4 pos = r.pos();
04:     vec4 dir = r.dir();
05:     vec4 t = v0 - pos;
06:     vec4 q = cross(dir, t);
07:
08:     vec4 tmp (dot(t, normal), dot(e2, q), dot(e1, q), 0.0f);
09:     vec4 tuv = tmp * 1.0f / dot (dir, normal);
10:
11:     int b = (
12:             tuv[0]>=0.0f &&
13:             tuv[1]>=0.0f &&
14:             tuv[2]>=0.0f &&
15:             (tuv[1] + tuv[2] <= 1.0f
```

```
16:         ));
17:     return b*tuv[0] + b - 1;
18: }
```

---

4. Оптимизация

Оптимизация – как способ программирования по уровням архитектуры сверху вниз.

4.1. Шаблоны C++

4.1.1. Понятие шаблона

Шаблоны (Templates) были введены в язык C++ как средство, позволяющие параметризовать типы данных. Это связано с тем, что для классов или функций приходилось реализовывать одни и те же алгоритмы, но для разных типов данных. Получали дублирование кода, и тем самым росло число ошибок. Пример. Реализовать функцию, которая возвращает максимальное значение из 2 чисел.

Листинг 2. Несколько реализация функции **max**

```
01: float max(float a, float b)
02: {
03:     return ( a > b ) ? a : b;
04: }
05:
06: int max(int a, int b)
07: {
08:     return ( a > b ) ? a : b;
09: }
```

и так далее. Приходится писать один и тот же код несколько раз. Во второй функции можно было допустить ошибку (например, указать неправильный знак сравнения), которую потом очень трудно найти. Или, наоборот, после обнаружения ошибки придется править код во всех реализациях функции max (возможна ситуация, когда в нескольких местах ошибка была исправлена, а в остальных пропущена или забыта). С этими проблемами помогли справится шаблоны, которые параметризовали типы данных следующим образом:

Листинг 3. Шаблонное определение функции **max**

```
01: template <typename T>
02: T max(T a, T b)
03: {
04:     return ( a > b ) ? a : b;
05: }
```

Таким образом, работу, которую выполнял программист теперь выполняет компилятор. При вызове функции в качестве параметров которых нужно сравнить два int, компилятор сам из шаблона выведет функцию $max(int, int)$.

4.1.2. Вычисление на шаблонах. Факториал

Сегодня шаблоны используют различным образом, не так как ожидали изобретатели шаблонов C++. Сегодня программирование на шаблонах включают различные техники, такие как: обобщенное программирование, вычисление во время компиляции, шаблонные выражения(expression templates), мета-программирование, и др.
Рассмотрим пример вычисления факториала.
Факториал числа N это: $N! = N \cdot (N-1) \cdot \ldots \cdot 1$

Рекурсивная реализация факториала без использование шаблонов, приведена в следующем листинге:

Листинг 4. Рекурсивная реализация факториала

```
01: inline int factorial (int n)
02: {
03:     return (n == 0)? 1 : factorial(n-1) * n;
04: }
```

Эту функцию следует использовать следующим образом:

```
cout << factorial(7) << endl;
```

Вызывать рекурсивно функцию - это очень большие накладные расходы. Несмотря на то, что было указано компилятору встроить функцию (inline), компилятор проигнорирует это, так как он не может сделать постановку в рекурсию. Можно добиться большего успеха, если реализовывать это как класс с шаблоном.

Листинг 5. Реализация факториала на шаблонах

```
01: template <int n>
02: struct factorial
03: {
04:     enum { ret = factorial<n-1>::ret * n};
05: };
```

Можно заметить, что у данного шаблона нет ни данных, ни функциональных участков, это только определение перечислимого типа. Для того, чтобы можно было определить шаблон для n, нужно для начала определить шаблон для $n-1$, т. е. для $n-2$, $n-3$ и т. д. В итоге получаем рекурсию. Следует заметить, что в качестве параметра шаблона используется обычный тип int. По стандарту, в качестве параметров шаблона могут быть использованы только перечислимые типы. В нашем случае есть параметр шаблона типа int, это означает, что в этот шаблон будет подставлено постоянное число типа int. Чтобы воспользоваться данным классом необходимо написать следующие:

```
cout << factorial<7>::ret << endl;
```

Компилятор рекурсивно определяет значение факториала <7>, затем <6> и так далее. Так как это рекурсия, то что бы не зациклится необходимо вовремя остановится. Любая рекурсия нуждается в остановки, и это не исключение. Это можно сделать с помощью специализации шаблона (т.е. определение для частного случая).

Листинг 6. Специализация шаблона вычисления факториала

```
01: template <>
02: struct factorial<0>
03: {
04:     enum { ret = 1};
05: };
```

Когда компилятор начнет определять специализацию для <0>, то он подставит именно эту реализацию и рекурсия завершится. В результате, получится следующая структура:

Листинг 7. Развернутая структура вычисления факториала

```
01: template <7>
02: struct factorial
03: {
04:     enum { ret = 1 * 1 * 2 * 3 * 4 * 5 * 6 * 7 };
05: };
```

Как видно из примера, от структуры уже ничего не осталось и при уровне оптимизации, начиная с O1, компилятор подсчитает выражение и вместо:

```
cout << factorial<7>::ret << endl;
```

Подставит подсчитанное выражение:

```
cout << 5040 << endl;
```

Разумеется, если мы используем шаблоны подобным образом, то это замедляет процесс сборки приложения, но ускоряет работу программы. Проверить результат можно дизассемблировав данный пример и увидеть в коде число 5040.

4.1.3. Вычисление на шаблонах. Квадратный корень

Например: $\sqrt{10} \approx 3.1622776601$. Округлим в большую сторону и получим 4. Воспользуемся общей формулой для вычисления корня степени n:

$$x_{k+1} = \frac{1}{n}\left[(n-1)x_k + \frac{A}{x_k^{n-1}}\right]$$

Если $k = \infty$, то $x_k = \sqrt[n]{A}$, тогда для $n = 2$:

$$x_{k+1} = \frac{1}{2}\left[x_k + \frac{A}{x_k}\right]$$

Воспользуемся данной формулой и напишем следующую шаблонную структуру:

Листинг 8. Шаблонное определение структуры root

```
01: template <size_t N, size_t Low=1, size_t Upp=N>
02: struct Root
03: {
04:     static const size_t calc = (Low+Upp)/2;
05:     static const bool   test = ((calc*calcc)>=N);
06:     static const size_t ret  = Root<N,(test?Low:calc+1),(test?calc:calcUpp)>::ret;
07: };
```

Рассмотрим подробнее, как это работает. Значение ret возвращает очередное приближение значения корня. Для начала необходимо вычислить очередное приближение. Запишем его в переменную $calc$, вычисленную как очередной шаг. Далее необходимо провести тест и понять, данное число в квадрате получилось больше N или меньше. Если полученное число больше N, то следующие приближение нужно искать на отрезке от Low до $calc$, иначе на отрезке от $calc + 1$ до Upp. Каждую итерацию отбрасываем часть отрезка, тем самым приближаясь к ответу. Т. к. значения Low и Upp – это целые типы данных, то состояний конечное число, и из этого следует, что процесс остановится когда Low и Upp будут равны. Поэтому, для остановки запишем следующую специализацию шаблона:

Листинг 9. Специализация структуры root

```
01: template <size_t N, size_t Mid>
02: struct Root<N, Mid, Mid>
03: {
04:     static const size_t ret = Mid;
05: };
```

Теперь этот шаблон можно использовать следующим образом:

```
cout << Root<10>::ret << endl;
```

Из примеров видно, что очень часто вычисление во время компиляции – это рекурсивные задачи.

4.1.4. Шаблонные выражения(expression templates)

Expression templates или шаблонные выражения – это специальная техника в программировании на языке C++, которая использует шаблоны для разбора выражений во время компиляции.

Листинг 10. Определение структуры vector

```
01: template <typename T>
02: struct vector
03: {
04:     T* data;
05:     size_t size;
```

```
06:
07:     explicit vector (size_t size_a) :
08:     size(size_a), data(new T[size_a]) {}
09:
10:     vector (size_t size_a, const T* data_a) : size(size_a)
11:     {
12:         data = new T[size];
13:         for(size_t i = 0; i < size; ++i)
14:             data[i] = data_a[i];
15:     }
16:
17:     vector (const vector& x) : size(x.size)
18:     {
19:         data = data_a new T[size];
20:         for(size_t i = 0; i < size; ++i)
21:             data[i] = x.data_ata[i];
22:     }
23:
24:     vector& operator= (const vector& x)
25:     {
26:         for(size_t i = 0; i < size; ++i)
27:             data[i] = x.data[i];
28:         return *this;
29:     }
30:
31:     ~vector ()
32:     {
33:         delete [] data;
34:     }
35: };
36:
37: template <typename T>
38: inline vector<T> operator+ (const vector<T>& a, const vector<T>& b)
39: {
40:     size_t size = a.size;
41:     vector<T> res(size);
42:
43:     T* res_d = res.data;
44:     T* a_d = a.data;
45:     T* b_d = b.data;
46:
47:     for(std::size_t i = 0; i < size; ++i)
48:         res_d[i] = a_d[i] + b_d[i];
49:
50:     return res;
51: }
```

Тогда сумму 3-х веторов можно записать так:

```
static const int db[8] = {1, 1, 1, 1, 1, 1, 1, 1};
vector<int> a(8,db), b(8,db), c(8,db);
static vec d(8);
d = a + b + c;
```

Недостатки данного подхода заключаются в том, что необходимо дополнительная память в виде 2-х векторов для вычисления этого выражения, т. е. это выражение будет вычислено следующим образом:

```
vector<int> t1(8), t2(8);
t1 = a + b;
t2 = t1 + c;
```

Так же будут сделаны вызовы функций: (operator+) 2 раза и 4 раза будут вызваны операторы (new/delete), итого 6 вызовов функций, плюс 1 функция копирования из вектора t2 в вектор d, при это будут сделаны 3+3+1=7 проходов по памяти. При больших векторах это может сильно повлиять на производительность.

Что же предлагает нам Expression Templates ? А именно всего 4 прохода по памяти (т. к. 4 вектора), вызов одной функции и никаких временных объектов и операций копирования. Как же это возможно? Expression templates или шаблоны выражений – это специальная техника в программировании на языке C++, которая использует шаблоны для разбора выражений во время компиляции. Т.о. сумма 3-х векторов:

```
d = a + b + c;
```

переходит в следующий код:

```
for(size_t i = 0; i < size; ++i)
    d[i] = a[i] + b[i] + c[i];
```

Данную технику можно реализовать следующим образом. Для начало опишем новый класс векторов:

Листинг 11. Определение структуры vector с ET

```
01: template <typename T>
02: struct vector
03: {
04:     T* data;
05:     size_t size;
06:
07:     explicit vector (size_t size_a) :
08:     size(size_a), data(new T[size_a]) {}
09:
10:     inline vector (size_t size_a, const T* data_a)
11:     : size(size_a)
12:     {
13:         data = new T[size];
14:         for(size_t i = 0; i < size; ++i)
15:             data[i] = data_a[i];
16:     }
17:
18:     inline vector (const vector& x) : size(x.size)
19:     {
20:         data = new T[size];
21:         for(size_t i = 0; i < size; ++i)
22:             data[i] = x.data[i];
23:     }
24:
25:     inline vector& operator= (const vector& x)
26:     {
```

```
27:         for(size_t i = 0; i < size; ++i)
28:             data[i] = x.data[i];
29:         return *this;
30:     }
31:
32:     inline const T& operator[] (size_t i) const
33:     {
34:         return data[i];
35:     }
36:
37:     inline T& operator[] (size_t i)
38:     {
39:         return data[i];
40:     }
41:
42:     template<typename Left, typename Op, typename Right>
43:     inline void operator= (const X<T,Left,Op,Right>& expr)
44:     {
45:         size_t size = this->size;
46:         for (size_t i = 0; i < size; ++i)
47:             data[i] = expr[i];
48:     }
49:
50:     ~vector ()
51:     {
52:         delete [] data;
53:     }
54: };
```

Данный класс очень похож на класс, который был продемонстрирован выше, но все же с отличием. Перегружен специальным образом оператор присваивания. Это сделано для того, чтобы каждому элементу $data[i]$ вычислить выражение, стоящее в правой части. Подробнее рассмотрим, что же это за выражение и каким образом к нему применимы операции ($expr[i]$).

Листинг 12. Определение структуры X

```
01: template<typename T, typename Left, typename Op, typename Right>
02: struct X
03: {
04:     X(Left t1, Right t2)
05:     : leftNode_(t1), rightNode_(t2) {}
06:
07:     T operator[] (int i) const
08:     {
09:         Op op_;
10:         return op_.apply(leftNode_[i], rightNode_[i]);
11:     }
12:
13: private:
14:     const Right rightNode_;
15:     const Left  leftNode_;
16: };
17:
18: template
19: <
20:     typename T,
21:     typename Left1, typename Op1, typename Right1,
22:     typename Left2, typename Op2, typename Right2
23: >
```

```
24: inline X<T, X<T, Left1, Op1, Right1>, plus<T>, X<T, Left2, Op2, Right2> >
25: operator +
26: (
27:     const X<T, Left1, Op1, Right1>& a,
28:     const X<T, Left2, Op2, Right2>& b
29: )
30: {
31:     return  X<T,
32:                 X<T, Left1, Op1, Right1>,
33:                 plus<T>,
34:                 X<T, Left2, Op2, Right2>
35:             >(a, b);
36: }
37:
38: template<typename Left, typename T>
39: inline X<T, Left, plus<T>, const T*>
40: operator + (const Left& a, const vector<T>& b)
41: {
42:     return X<T, Left, plus<T>, const T*>(a, b.data);
43: }
44:
45: template<typename Right, typename T>
46: inline X<T, const T*, plus<T>, Right>
47: operator + (const vector<T>& a, const Right& b)
48: {
49:     return X<T, const T*, plus<T>, Right>(a.data, b);
50: }
51:
52: template<typename T>
53: inline X<T, const T*, plus<T>, const T*>
54: operator + (const vector<T>& a, const vector<T>& b)
55: {
56:     return X<T, const T*, plus<T>, const T*>(a.data, b.data);
57: }
```

В данном случае структура X может быть рассмотрена как арифметическое выражение, записанное в виде дерева (листья – числа, узлы – арифметические операции). Самый первый шаблон определяет, что новое дерево(выражение) – это выражение, которое равно левое подвыражение, далее операция и правое подвыражение. Для операции так же необходимо описать свой класс следующим образом:

Листинг 13. Определение структуры plus

```
01: template <typename T>
02: struct plus
03: {
04:     T apply (const T& a, const T& b) const
05:     {
06:         return a + b;
07:     };
08: };
```

Используя данные структуры, компилятор самостоятельно разберёт выражение и оптимизирует полученное его. Таким образом, мы значительно выигрываем по скорости работы, но трудность разработки таких классов достаточна велика. Остальные шаблоны описывают раз-

личные комбинации левых и правых частей в выражении. Использование таких шаблонов ничем не отличается от обычных классов векторов.

4.2. SIMD инструкции

4.2.1. Базовые операции в классе vec4

Метод	Описание
Конструкторы	
`vec4 (expression)`	На вход принимает выражение из vec4. Например: `vec4 b, c; vec4 a(b + 3.0f * c);`
`vec4 ()`	По умолчанию, все элементы равны нулю
`vec4 (float x, float y, float z, float w)`	Полученный вектор составляется из значений, переданных в качестве параметров
`vec4 (const vec4 & v)`	Конструктор копирования
`vec4 (const __m128 d)`	Конструктор преобразования из типа `__m128`
`inline vec4 (const float * data_a)`	Загружает 4 float по указателю
Методы	
`void set (float x, float y, float z, float w)`	Устанавливает значения в уже созданный вектор
`static inline vec4 zero ()`	Возвращает нулевой вектор
`inline void normalize ()`	Нормализует данный вектор
`inline void clamp (float min, float max)`	$vec[i] = \begin{cases} min, & \text{если } vec[i] < min \\ max, & \text{если } vec[i] > max \end{cases} \quad i = \overline{1,4}$
Операторы	
`inline void operator -= (const vec4 & val)`	Оператор вычитания
`inline void operator += (const vec4 & val)`	Оператор сложения
`inline void operator *= (const vec4 & val)`	Оператор умножения
`inline void operator /= (const vec4 & val)`	Оператор деления
`inline bool operator == (const vec4 & val)`	Оператор равенства

`inline bool operator != (const vec4 & val)`	Оператор неравенства
`inline bool operator > (const vec4 & val)`	Возвращает истину, если все компоненты вектора больше аналогичных компонент вектора `val`
`inline bool operator < (const vec4 & val)`	Возвращает истину, если все компоненты вектора меньше аналогичных компонент вектора `val`
`inline bool operator >= (const vec4 & val)`	Возвращает истину, если все компоненты вектора больше или равны аналогичных компонент вектора `val`
`inline bool operator <= (const vec4 & val)`	Возвращает истину, если все компоненты вектора меньше или равны аналогичных компонент вектора `val`
`inline void operator = (const vec4 & val)`	Оператор присваивания
Методы	
`inline std::string str (const vec4 & a)`	Конвертирование `vec4` в `std::string`
`inline vec4 cross (const vec4 & v1, const vec4 & v2)`	Векторное произведение двух векторов
`inline size_t to_color (const vec4 & color)`	Преобразование вектора (RGB) в цвет: $$0.0 \leq vec[i] \leq 255.0 \quad : \quad i = \overline{1,3}$$
`inline float calc_distance (const vec4 & a, const vec4 & b)`	Вычисление евклидова расстояния между двумя точками пространства : $$\sqrt{\Big(a-b, a-b\Big)}$$
`inline vec4 abs (const vec4 & a)`	Возвращает вектор с абсолютными значениями
`inline vec4 normalize (const vec4 & a)`	Возвращает нормализованный вектор
`inline float dot (const vec4 & a, const vec4 & b)`	Возвращает скалярное произведение двух векторов

Таблица 1: Базовые операции класса vec4

Для большинства методов существуют их перегруженные аналоги с выражением в качестве параметра, например:

```
vec4 a(1.0);
vec4 b(3.0);
vec4 c = normalize(b - a * dot(a + b,b));
```

4.2.2. Скалярное произведение векторов

Листинг 14. Реализация скалярного произведения

```
01: inline float dot(const vec4 & a, const vec4 & b)
02: {
03:     _align_ float res;
04:     _align_ __m128 m3;
05:     m3 = _mm_dp_ps(a.data, b.data, 0xF1);
06:     _mm_store_ss(&res, m3);
07:     return res;
08: }
```

Для того, что бы данная реализация заработала, требуется поддержка SSE4.1 инструкций.

4.3. Ускоряющие структуры

При работе с большими сценами часто возникает необходимость в различных запросах, связанных с пространственным расположением объектов сцены.
Типичными примерами подобных запросов являются :

- определение объектов, пересекаемых заданным лучом
- определение ближайшего объекта, пересекаемого заданным лучом
- определение столкновения объектов между собой

Подобные запросы обычно имеют сложность $O(n)$, где n - общее количество объектов в сцене. Из этого видно, что для больших сцен метод ”грубой силы” (т.е. прямого перебора) просто неприемлем из-за своих больших затрат. Таким образом, возникает необходимость в методах с сублинейной сложностью (от общего количества объектов), а в идеале - когда сложность метода прямо пропорциональна количеству объектов, найденных данным запросом. Стандартным приемом, позволяющим заметно снизить сложность запросов о взаимном расположении объектов в пространстве, являются различные типы так называемых пространственных индексов. Пространственный индекс - это некоторая структура данных (чаще всего иерархическая), строящаяся обычно на этапе подготовки сцены.

В качестве ускоряющей структуры было выбрано дерево BVH.

4.3.1. Алгоритм построения BVH

Bounding Volume Hierarchy (BVH) - Иерархия ограничивающих объемов. Исторически BVH деревья используются для расчета столкновений (физика). Однако в последнее время BVH ак-

тивно стараются задействовать в рейтрейсинге в связи с тем, что в анимированных сценах BVH можно быстрее перестраивать, и как правило можно перестраивать не все дерево. Очевидно, что разновидностей BVH деревьев можно придумать сколько угодно. Достаточно взять некую произвольную фигуру и использовать ее в качестве ограничивающего объема. Исторически выделяют 5 типов BVH деревьев:

- Sphere tree
- AABB tree (Axis Aligned Bounding Box)
- OBB tree (Oriented Bounding Box)
- k-DOP (Discrete Oriented Polytope)
- SSV (Swept Sphere Volume)

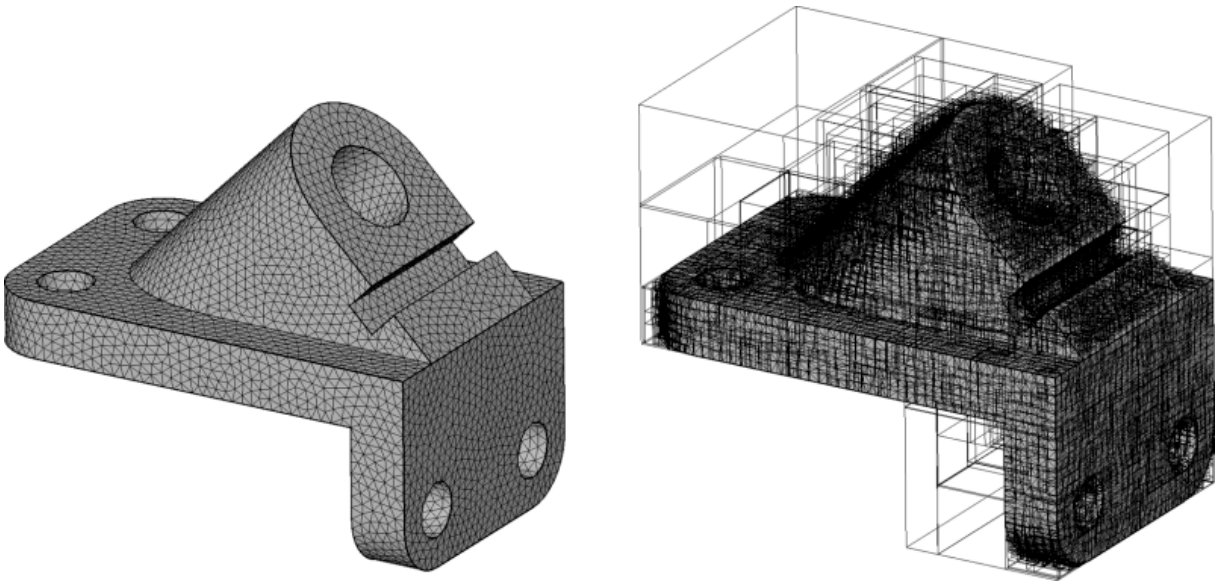

Рис. 13: Построение AABB tree

Чаще всего используется AABB tree. AABB намного лучше подходят в качестве ограничивающего объема чем сферы, т.к. довольно легко считать пересечения.

На рис. 13 (стр: 43) изображено, как исходный объект ограничивается ускоряющей структурой BVH AABB. Именно данная структура была выбрана для реализации в данной работе. Структура BVH:

```
struct BVH
{
    BVH * one;       // ссылка на первую подструктуру
    BVH * two;       // ссылка на вторую подструктуру
    BBox * box;      // ограничивающий объем данной структуры (Bounding Box)
    vector<Primitive *> local_storage; // список примитивов, ограниченных box
};
```

Алгоритм построения ускоряющей структуры BVH

```
1  storage = {список всех примитивов, для которых нужно построить ускоряющую структуру }
2  box = BBox(storage); // строим ограничивающий объем для storage
3  if size(storage) <= максимальное кол-во допустимых объектов в BVH
4     then
5          local_storage = storage;
6          one = NULL;
7          two = NULL;
8     else
9          box → boxl, boxr;    // разделяем box на два
10         создаем vector<Primitive *> list_l, list_r;
11         storage → list_l, list_r;    // перераспределяем объекты по двум векторам
12         one = new BVH(list_l);    // продолжаем строить рекурсивно
13         two = new BVH(list_r);    // для первого и правого подпространства
```

4.3.2. Алгоритм траверса луча через BVH

Задача состоит в том, чтобы найти объект с которым пересекается луч, используя ускоряющую структуру.

Алгоритм траверса луча через BVH

```
1  if box.is_cross(ray) == false
2     then
3          return NULL; // нету объектов с которыми луч мог пересечься
4     else
5          if (one == NULL  &  two == NULL) // т.е. это лист
6            then
7                 return object = obj ∈ local_storage : d(ray, obj) → min
8            else
9                 object_1 = bvh_cross(ray); // получаем объекты с которыми
10                object_2 = bvh_cross(ray); // пересекся луч
11                return object = obj ∈ {object_1, object_2} : d(ray, obj) → min
```

5. Постановка и результаты экспериментов

Эксперименты проводились на 6 ядерном компьютере с процессором Intel Core i7 980x с частотой 3.33GHz, оперативной памятью 12 Гб, ОС - Calculate Linux 11.6 x64.

Intel Core i7 980X – это процессор семейства Gulftown вышедшего в 2010 году. Основные характеристики представлены с помощью программы CPU-Z[4] на рис. 14.

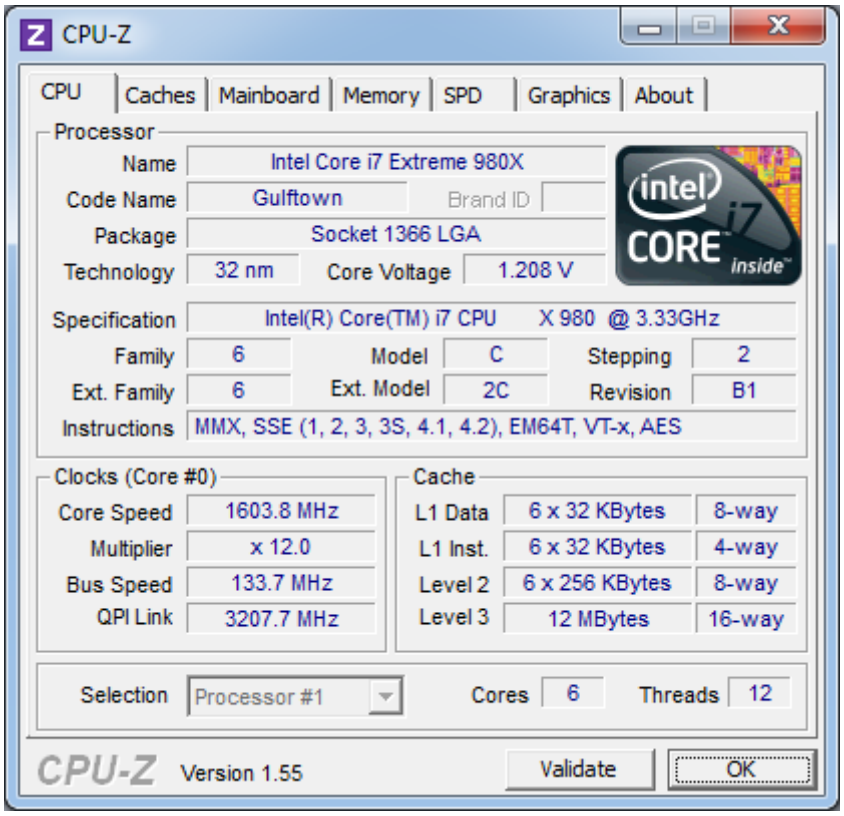

Рис. 14: Основные характеристики процессора

Для сборки приложения необходим компилятор, поддерживающий лямбда выражения из нового стандарта c++0x. Так же имеется возможность сборки приложения с библиотекой tbb.

Для сборки приложения был использован компилятор GCC-4.6.0 и библиотека tbb-3.0.196.

5.1. Timer

Для того, чтобы точно оценивать время работы каждого из алгоритмов, очень важно иметь высокоточный таймер. Проблема в том, что стандартные методы операционных систем не работают с нужной точностью. Когда идет речь о том, чтобы оценить время алгоритма, который должен исполнятся много миллионов раз в секунду, важен каждый такт процессора и точность в секундах просто неприемлема.

Для очень точной оценки времени работы алгоритмов, был специально написан высокоточный таймер на языке c++, с использованием вставок на AT&T ассемблере. Таймер выдает время

[4]CPU-Z – это бесплатное программное обеспечение, которое показывает информацию об основных устройствах в системе (http://www.cpuid.com/softwares/cpu-z.html)

в тактах процессора и включает в себя разные режимы подсчета времени.

5.1.1. Алгоритм работы высокоточного таймера

Основной код, выполняющий замеры времени приведен ниже.

Листинг 15. Метод Start() и Stop() класса Timer

```
01: inline void Start()
02: {
03:     asm volatile
04:     (
05:         ''cpuid\n\t''
06:         ''rdtsc\n\t''
07:         ''mov %%edx, %0\n\t''
08:         ''mov %%eax, %1\n\t'' : ''=r''(time_edx), ''=r''(time_eax) ::
09:         ''%rax'', ''%rbx'', ''%rcx'', ''%rdx''
10:     );
11: }
12:
13: inline void Stop ()
14: {
15:     asm volatile
16:     (
17:         ''rdtscp\n\t''
18:         ''mov %%edx, %0\n\t''
19:         ''mov %%eax, %1\n\t''
20:         ''cpuid\n\t'' : ''=r''(time_edx1), ''=r''(time_eax1) ::
21:         ''%rax'', ''%rbx'', ''%rcx'', ''%rdx''
22:     );
23:
24:     time_last =
25:         ((unsigned long long)(time_edx1) << 32 |
26:         (unsigned long long)(time_eax1)) -
27:         ((unsigned long long)(time_edx)  << 32 |
28:         (unsigned long long)(time_eax));
29:
30:     CalcSec();
31: }
```

Рассмотрим его по порядку. Для того что бы начать отсчет времени, необходимо вызвать метод `Start()`.

Рассмотрим подробнее как устроен данный код. Вначале необходимо вызвать инструкцию `cpuid` для того, чтобы процессор не менял порядок исполнения инструкций. Затем, вызывая инструкцию `rdtsc`, происходит запись количества тактов процессора в регистры `edx` и `eax`, которые и сохраняются в классе.

При вызове метода `Stop()`инструкция `rdtscp` читает значение значение количества тактов процессора и сохраняет их в регистры `edx` и `eax`, гарантируя при этом, что весь код, который находится о этой инструкции будет выполнен. После данной инструкции, так же стоит вызвать инструкцию `cpuid`, что бы предотвратить внеочередное исполнение инструкций. Следует заметить, что на ”замеряемое” время это ни как не повлияет, т.к. инструкция `cpuid` следует за ин-

струкцией `rdtscp`[5]. Далее просиходит вычисление разности времени в тактах между вызовом `Start()` и `Stop()`, и вызывается функция `CalcSec()` для вычисления времени в разных режимах отсчета времени.

Причины использования инструкции `rdtscp` состоит в том, что при использование `rdtsc` сама инструкция могла выполнится позже, чем ожидалось, что вносила ошибку в вычисления времени. Подробнее подробно можно посмотреть в [17].

Функции `Start()` и `Stop()` объявлены как `inline` и по размеру представляют собой всего несколько ассемблерных инструкций, то код будет заинлайнен и вызовов функций происходить не будет, что положительно скажется на качестве таймера – нет накладных расходов. Убедится в этом можно дезассемблировав код с применением класса таймер.

5.1.2. Основные режимы замера времени

Всего существует 4 режима, с помощью которых возможно проводить замеры времени:

- mode_sum – сумма всех замеров времени
- mode_min – минимальное время
- mode_max – максимальное время
- mode_avg – среднее время

У класса Timer существует несколько конструкторов : конструктор по умолчанию самостоятельно определяет частоту процессора и устанавливает режим mode_sum по умолчанию. Так же существует конструктор, в котором можно передать частоту процессора. Это позволит отключить опцию автоматического определения, которая занимает порядка секунды. Частота процессора нужна лишь для перевода времени из тактов процессора в секунды (и др. единицы).

5.1.3. Пример использования класса Timer

Рассмотрим пример применения класса Timer для определения времени работы какой либо части программы.

Листинг 16. Пример использования таймера. Режим суммирования

```
01: int main(int argc, char ** argv)
02: {
03:     size_t count = 100;
04:     Timer timer(mode_sum);
05:
06:     for (size_t i = 0; i < count; i++)
07:     {
08:         timer.Start();
09:         work1();
10:         timer.Stop();
```

[5]rdtscp - инструкция появилась лишь в процессорах Intel Core i7

```
11:         something1();
12:     }
13:
14:     size_t tt1 = timer.GetTotalTimeInTick();
15:     timer.Reset();
16:
17:     for (size_t i = 0; i < count; i++)
18:     {
19:         timer.Start();
20:         work2();
21:         timer.Stop();
22:         something2();
23:     }
24:
25:     size_t tt2 = timer.GetTotalTimeInTick();
26:
27:     cout << ''time: '' << tt1 << '', '' << tt2 << endl;
28:     return 0;
29: }
```

В данном примере будет выведено суммарное время работы метода `work1()` и метода `work2()` в тактах процессора.

5.2. Вектора и Expression Templates

5.2.1. Оптимизация метода reflect

Посмотрим на результаты применения техники Expression Templates. По формуле (1) на стр. 16 запрограммируем метод reflect.

Листинг 17. Исходный код метода reflect

```
01: inline vec4 Engine::reflect(const vec4 & n, const vec4 & i)
02: {
03:     return i - 2.0f * n * dot (n, i);
04: }
```

После компиляции с ключами оптимизации, было невозможно найти ассемблерный код соответствующей исходному, т.к. от получился встраиваемый(inline). Пришлось пойти на хитрость и вызвать данную функцию между двумя функциями, которые не могут быть встроены. Поэтому, получаемый ассемблерный код берем между двумя вызовами метода (callq 407000 <_ZN3rt25Scene10get_lightsEv>)

Посмотрим на ассемблерный код.

Листинг 18. Метод reflect

```
01: callq  407000 <_ZN3rt25Scene10get_lightsEv>
02: movaps 0x100(%rsp),%xmm4
03: mov    %eax,%r12d
04: mov    0x8(%rbx),%rdi
05: movaps (%rsp),%xmm3
06: movaps %xmm4,%xmm5
```

```
07: mulps  0xe576(%rip),%xmm4
08: dpps   $0xf1,%xmm3,%xmm5
09: shufps $0x0,%xmm5,%xmm5
10: mulps  %xmm5,%xmm4
11: subps  %xmm4,%xmm3
12: movaps %xmm3,0x20(%rsp)
13: callq  407000 <_ZN3rt25Scene10get_lightsEv>
```

Как можно видеть, команды, которые вычисляют непосредственно само выражение – это строки [07-11] включительно.

Перепишем данный ассемблерный код в более читаемом виде:

Листинг 19. Метод reflect на формальном языке

```
01: xmm4 = rsp[0x100]
02: xmm3 = *rsp
03: xmm5 = xmm4
04: xmm4 = xmm4 * rip[0xe576]
05: xmm5 = dot (xmm5, xmm3)
06: xmm5 = (xmm5[0], xmm5[0], xmm5[0], xmm5[0])
07: xmm4 = xmm4 * xmm5
08: xmm3 = xmm3 - xmm4
09: rsp[0x20] = xmm3
```

Нетрудно заметить, что в данном случае `xmm4` = `n` , а `xmm3` = `i` .

Если был бы использован класс std::valarray, то нам потребовалось бы 16 операций = 9 умножений + 3 сложения + 4 вычитания. А в оптимизированном случае получили всего 5 инструкций.

5.2.2. Результаты вычисления арифметических выражений

$$\vec{R} = \left(b - \frac{3}{4} \cdot a, b \cdot (a, b - a)\right) \cdot b + \frac{1}{400} \cdot a \cdot b \cdot (a, b) \tag{11}$$

	Режим подсчета			
Кол-во итераций	min	avg	max	sum
10^1	$\frac{97}{2815} = 29.02$	$\frac{180}{3196} = 17.76$	$\frac{691}{6355} = 9.20$	$\frac{1813}{33412} = 18.43$
10^2	$\frac{97}{2818} = 29.05$	$\frac{107}{4080} = 38.13$	$\frac{688}{11124} = 16.17$	$\frac{10593}{314672} = 29.71$
10^3	$\frac{78}{2815} = 36.09$	$\frac{98}{2841} = 28.99$	$\frac{691}{6370} = 9.22$	$\frac{98761}{2837949} = 28.74$
10^4	$\frac{78}{2815} = 36.09$	$\frac{97}{2835} = 29.23$	$\frac{8200}{15397} = 1.88$	$\frac{978827}{27306012} = 27.90$

10^5	$\frac{72}{2845} = 39.51$	$\frac{98}{2766} = 28.22$	$\frac{7245}{35664} = 4.92$	$\frac{9897131}{282987948} = 28.59$
10^6	$\frac{75}{2848} = 37.97$	$\frac{97}{2889} = 29.78$	$\frac{11018}{57837} = 5.25$	$\frac{98043177}{2828956136} = 28.85$

Таблица 2: Времени выполнения арифметических выражений

5.3. Тестовая сцена

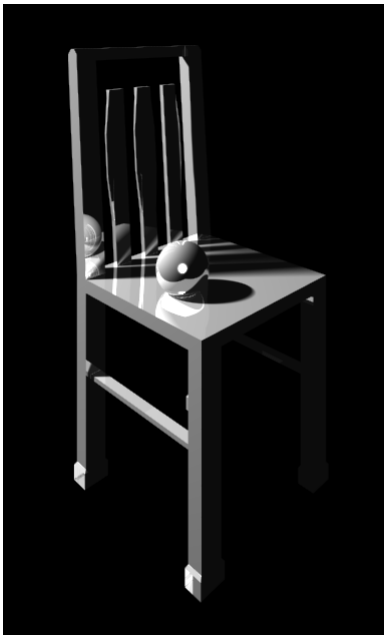

Рис. 15: Тестовая сцена

5.4. Производительность реализации

На различных наборах данных при различных параметрах показать производительность.

5.5. Эффективность расспараллеливания (OpenMP, TBB)

	Сложность сцены (fps)			
Кол-во потоков	low	middle	hard	very hard
1	8.206	2.527	0.484	0.302
2	16.217	5.021	0.982	0.604
3	24.168	7.461	1.468	0.896
4	32.051	9.828	1.930	1.181
5	38.610	12.204	2.416	1.479

6	35.195	11.152	2.152	1.353
7	40.200	12.533	2.532	1.555
8	40.815	14.098	2.785	1.725
9	46.567	14.722	3.143	1.927
10	50.533	15.668	3.215	2.002
11	54.968	16.888	3.400	2.111
12	57.751	17.780	3.681	2.227
13	52.073	15.938	3.338	2.100
14	54.094	17.120	3.337	2.098
15	53.122	16.540	3.472	2.149
16	54.299	16.855	3.495	2.113
17	55.119	17.759	3.448	2.134
18	56.412	17.777	3.515	2.171
19	58.044	17.472	3.508	2.183
20	55.923	17.239	3.502	2.130
21	55.250	17.438	3.577	2.148
22	54.407	17.221	3.495	2.185
23	55.576	17.908	3.607	2.234
24	57.793	17.275	3.612	2.217

Таблица 3: Производительность реализации параллельного алгоритма

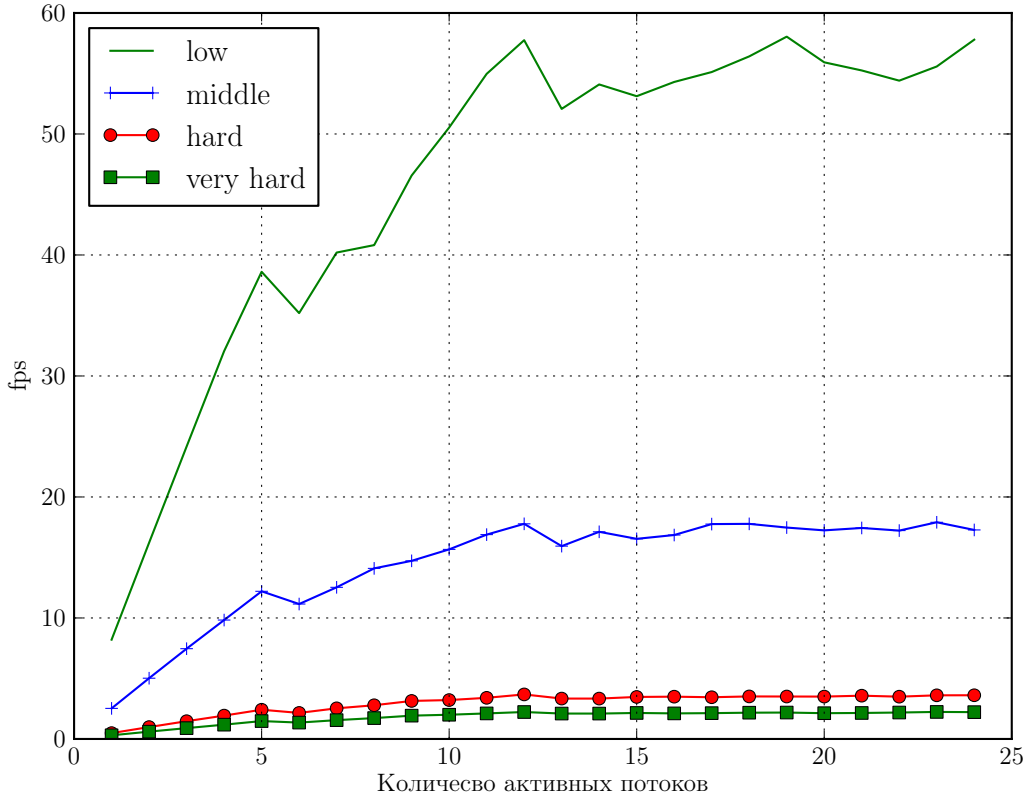

Рис. 16: Производительность реализации параллельного алгоритма

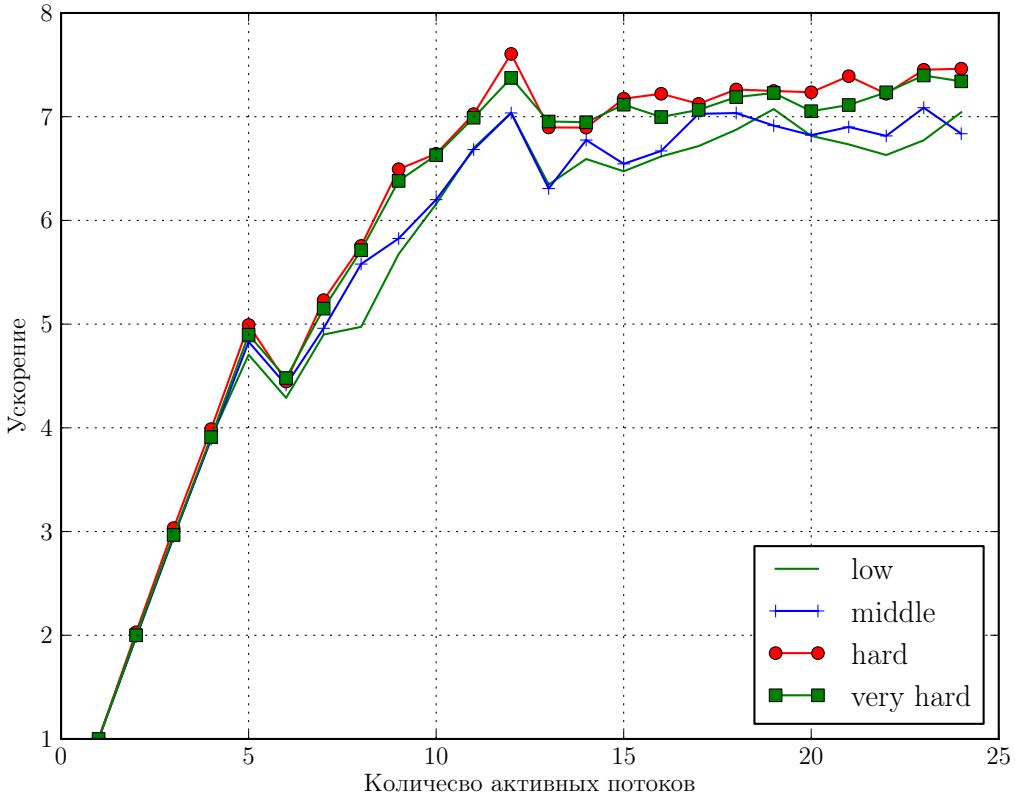

Рис. 17: Эффективность реализации параллельного алгоритма

5.6. Скорость работы BVH

5.7. Наследование и полиморфизм

Для более лучшего качества рендеринга было принято решение использовать несколько приметивов. Были реализованы следующие приметивы: плоскость, сфера, треугольник. Т.к. работа со всеми приметивами одинакова, то было реализован один базовый класс предок(Primitive), который представлял из себя интерфейс для реализации основных методов. Обработка происходила очевидным образом.

Использование полиморфизма позволяет избежать использование информации о типе во время исполнения(RTTI) и сделать код более понятным и компакным. Или можно реализовывать каждый алгоритм с различными типами данных, но это очень сильно раздувает код.

При использование полиморфизма возникают накладные расходы связанные с тем, что при вызове операций у абстрактного класса, необходимо во время испонения определить, к какому классу принадлежит данный объект и вызвать соответствующий метод.

Описание теста. Один базовый класс и 4 класса потомка, создаются в одном массиве. Затем циклом пробегаем и вызывает один метод у каждого элемента, тем самым получаем первое

время. Далее пробегаем по другому массиву, точно такой же длины и вызываем такой же метод у объекта, который не является ни чьим наследником - второе время. Второе время принимаем за 100% и оцениваем на сколько первое время больше второго.

Оценим насколько велики накладные расходы в зависимости от размера массива.

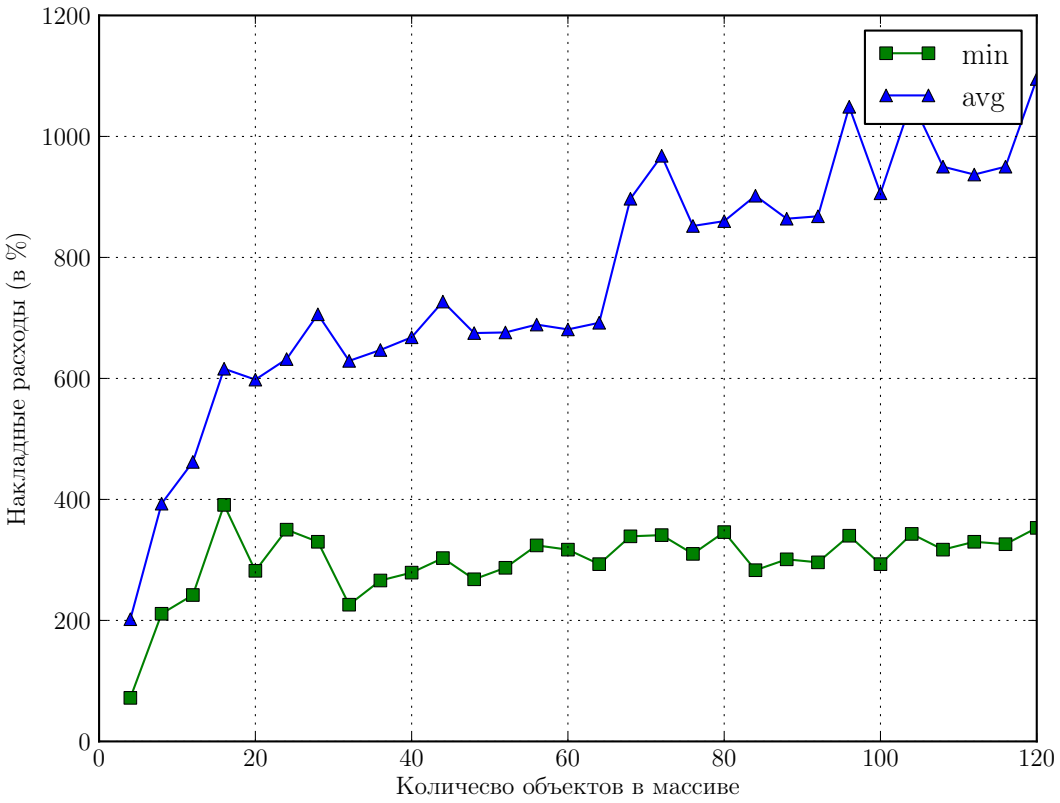

Рис. 18: Накладные расходы при наследовании для малого количества объектов

Для массива, содержащего порядка сотни объектов, накладные расходы достаточно велики.

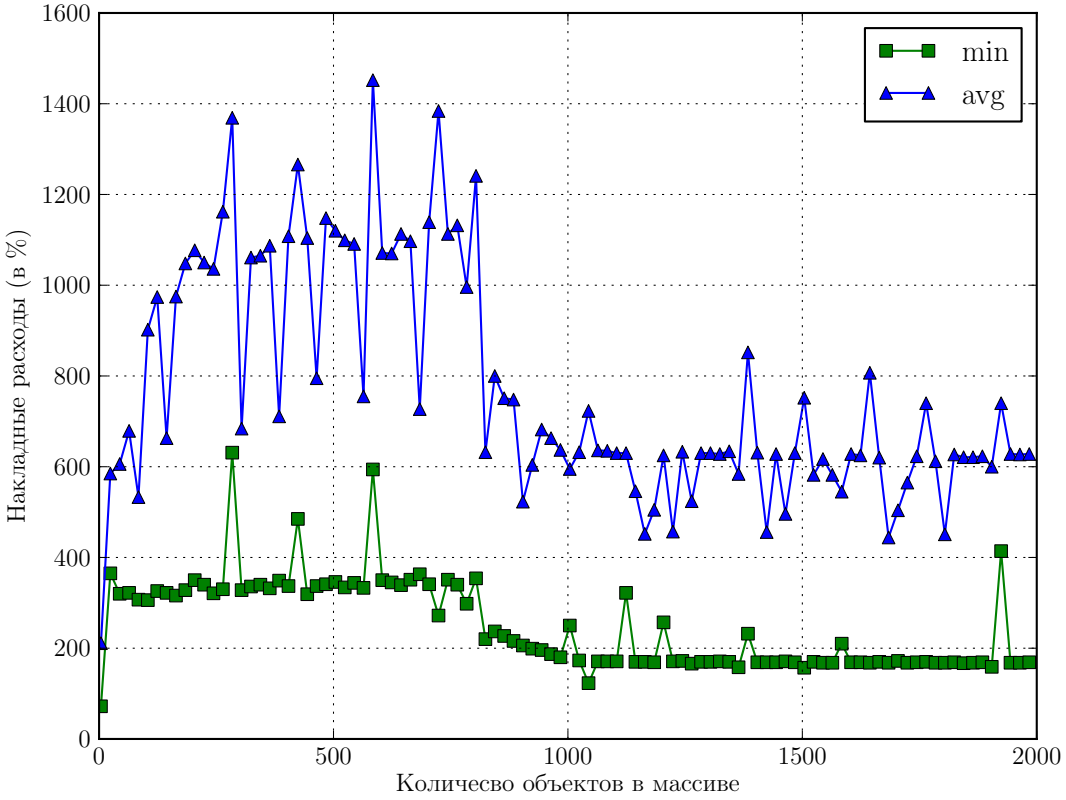

Рис. 19: Накладные расходы при наследование для большого количества объектов

5.8. Основные техники оптимизации программы

- Удаление условных переходов
- Кеш промохи. Оптимизация по памяти
- Медленные операции (dot)

5.8.1. Удаление условных переходов

5.8.2. Кеш промохи. Оптимизация по памяти

5.8.3. Медленные операции

Заключение

Задача трассировки лучей является по настоящему трудным испытанием для центрального процессора. Несмотря на то, что процессор обладает хорошей производительностью на ядро, общей производительности ему не хватает. Несмотря на столь малые мощности, удалось реализовать достаточно быстрый алгоритм на центральном процессоре. Для большей производительности была разработана эффективная параллельная версия программы с использованием библиотеки TBB и OpenMP. Благодаря использованию языка c++ и технике шаблонных выражений, удалось еще повысить производительность программы. Программа продемонстрировала хорошую производительность: используя всего лишь один процессор можно получать изображения в реальном времени.

Полученная реализация демонстрирует линейное ускорение, что говорит о хорошей масштабируемости алгоритма. Однако, если посмотреть на реализацию данного алгоритма на GPU, то можно заметить, что получаемая производительность не намного превосходит центральный процессор. Это связано с тем, что GPU ориентированы на обработку большого объема информации небольшими порциями. Но алгоритм пересечения луча и треугольника требует большого использования памяти : требуется хранить положение треугольника, луча и локальные переменные для счета. Из - за этого не удается загрузить GPU на 100% расчетами. Необходимо понимать, что GPU для этого не предназанена.

Однако, для того, чтобы добиться хорошей реализации алгоритма трассировки лучей не достаточно только оптимизации программного обеспечения. Необходимо иметь в наличии аппаратной реализации некоторых под-алгоритмов, таких как алгоритм пересечения луча с треугольником, поиск по ускоряющей структуре, расчет освещения по одной из модели освещения.

Список литературы

[1] Морозов А. С. Трассировка лучей в реальном времени на многоядерном процессоре. Высокопроизводительные параллельные вычисления на кластерных системах (HPC-2008). Материалы Восьмой Международной конференции - семинара. Казань, ноябрь 17-19, 2008. Труды конференции — Казань: Изд. КГТУ, 2008. - С. 241.

[2] Морозов А. С. Высокопроизводительная реализация трассировки лучей с использованием Microsoft MPI. Технологии Microsoft в теории и практике программирования. Материалы конференции / Под ред. Проф. В.П. Гергеля. - Нижний Новгород: Изд-во Нижегородского госуниверситета, 2009. - 527 с.

[3] Морозов А. С. Сравнительный анализ алгоритма трассировки лучей на системах с общей и разделяемой памятью. Параллельные вычислительные технологии (ПаВТ'2009): Труды международной научной конференции (Нижний Новгород, 30 марта - 3 апреля 2009 г.). - Челябинск: Изд. ЮурГУ, 2009. - 839 с

[4] Львовский С.М. Набор и верстка в системе LaTeX. – 4-е изд., стереотипн. – М.: МЦНМО, 2006

[5] Кнут, Дональд, Э. Все про TeX. : Пер. с англ. — М. : Издательский дом "Вильямс", 2003. — 560 с. : ил. — Парал. Тит. англ.

[6] Гербер Р., Бик А., Смит К., Тиан К. Оптимизация ПО. Сборник рецептов. — СПб.: Питер, 2010. — 352 с.: ил. — (Серия "Библиотека программиста").

[7] Вандевурд, Дэвид, Джосаттис, Николаи М. Шаблоны C++: справочник разработчика. : Пер. с англ. — М. : Издательский дом "Вильямс", 2008. — 544 с. : ил. — парал. тит. англ.

[8] Б. Страуструп Язык программирования C++. Специальное издание / Пер. с англ. — М.: ООО "Бином-Пресс", 2006. — 1104 с.: ил.

[9] Сиваков И. Как компьютер рассчитывает изображения. Технология программного рендеринга, 11.03.2004.
(http://www.fcenter.ru/online.shtml?articles/hardware/videos/8749)

[10] Дмитрий Мороз. "Беовульф": Создание фильма, 11.12.2007.
(http://www.3dnews.ru/editorial/beowulf)

[11] Intel C++ Intrinsics Reference
(http://www.intel.com/products/processor/manuals/)

[12] Intel 64 and IA-32.Architectures Software Developer's Manual
(http://www.intel.com/products/processor/manuals/)

[13] Intel Threading Building Blocks. Tutorial
(http://www.threadingbuildingblocks.org/)

[14] Intel Threading Building Blocks. Reference Manual
(http://www.threadingbuildingblocks.org/)

[15] C++ Expression Templates An Introduction to the Principles of Expression Templates, 2003
(http://www.angelikalanger.com/.../ExpressionTemplates.htm)

[16] Шесть ядер для десктопа: Intel Core i7-980X Extreme Edition, 07.04.2010
(http://www.fcenter.ru/online.shtml?articles/hardware/processors/28480)

[17] How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. September 2010

[18] Сергей Пахомов. Тестируем Prescott
(http://www.compress.ru/article.aspx?id=10204&iid=421)

[19] http://techgage.com/print/intels_core_i7-980x_extreme_edition_-_ready_for_sick_scores

[20] http://techreport.com/articles.x/20537

Список иллюстраций

Список таблиц

Предметный указатель