Федеральное агентство по образованию Российской Федерации
Государственное образовательное учреждение
высшего профессионального образования
Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики
Кафедра математического обеспечения ЭВМ

Дипломная работа

Трассировка лучей в реальном времени на x64 архитектуре

Работа допущена к защите
Заведующий кафедрой МО ЭВМ
д.ф.–м.н., проф.

_____ Стронгин Р. Г.
Подпись

«____» _____ 2011 г.

Исполнитель:
студент 2 курса магистратуры
факультета ВМК группы 86М1

_____ Морозов А. С.
Подпись

«____» _____ 2011 г.

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

_____ Турлапов В. Е.
Подпись

«____» _____ 2011 г.

Нижний Новгород
2011 г.

Содержание

Введение

В кино индустрии к современной компьютерной графике предъявляются серьезные требования физически корректного моделирования о освещения сцен, состоящая из множества примитивов с различными характеристиками взаимодействия 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 спец эффекты должны выглядеть настолько реалистично, что бы зритель не смог различить, где настоящий актер, а где рисованный двойник. Используя только физически правильные модели и алгоритмы можно обеспечить растущую потребность в более реалистичной трехмерной графике.

С каждым новым фильмом, каждый из нас видит прогресс в компьютерной графике. Картинка становится все красочнее и правдоподобнее, но это все не дается просто так. Естественно, платить за это приходится высокой вычислительной трудоемкостью расчетов. Несомненно, что с каждым годом производительность вычислительной техники растет, но она сразу же «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 десятки, а то и в сотни раз превышает производительность компьютеров того времени. Со временем улучшается и требования к самому изображению. Если несколько лет назад картинка с разрешением 1024x768 считалась излишеством в компьютерной графики, то уже сейчас это слишком мало и все считают де факто FullHD[1] , хотя уже задумываются о еще лучшем качестве. 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 разработка и исследование алгоритма трассировки лучей на архитектуре x64 с применением ускоряющей структуры. 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
- Реализация параллельной версии алгоритма трассировки лучей с использованием OpenMP, TBB
- Сравнение параллельной версии алгоритма трассировки лучей с использованием библиотеки TBB и расширения языка OpenMP
- 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

2. Архитектура центрального процессора

2.1. Архитектура процессоров семейства x86

Рассмотрим основные архитектурные особенности центрального процессора. Одни из базовых понятий для производительности процессора:

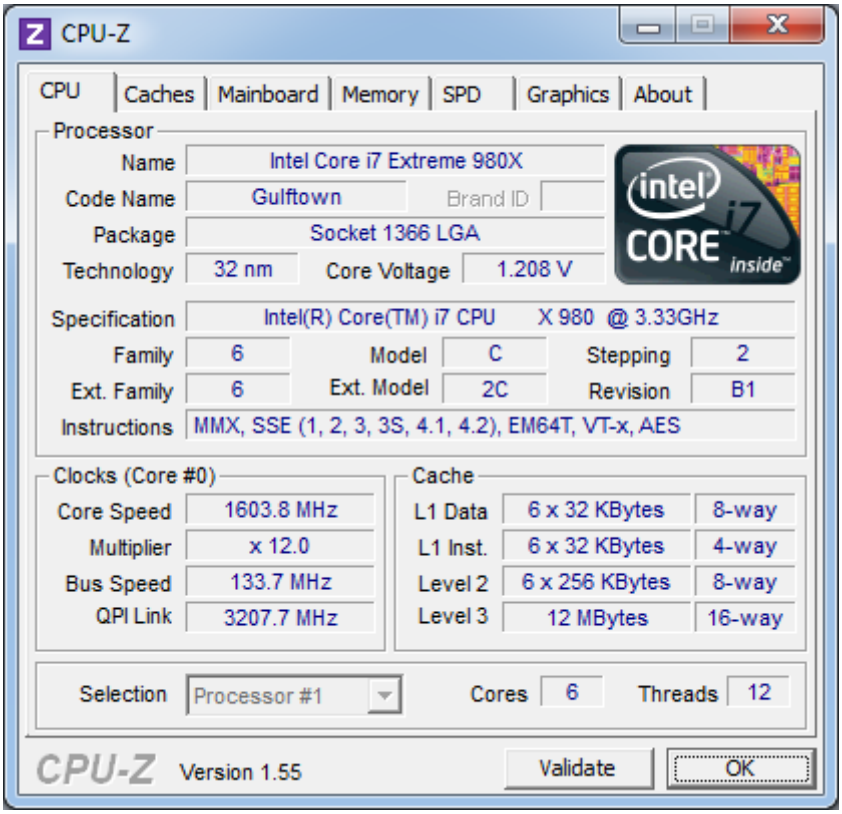
- Latency (Латентность команды) — это число тактов, необходимое для завершения одной команды с момента готовности входных данных команды(выборки их из памяти) и начало ее выполнения
- Throughput (Пропускная способность команды) — это число тактов ожидания, которое требуется процессору перед запуском на выполнение такой же команды

2.2. Архитектура процессоров семейства x64

2.2.1. Архитектура процессора Intel Core i7 980X

`Обзор`

Intel Core i7 — семейство процессоров x86-64 Intel. Core i7 980X это процессор семейства Gulftown вышедшего в 2010 году. Основные характеристики представлены с помощью программы cpu-z:



3. Трассировка лучей

Классический ray tracing [[9]], или метод трассировки лучей, предложен Артуром Аппелем (Arthur Appel) еще аж 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

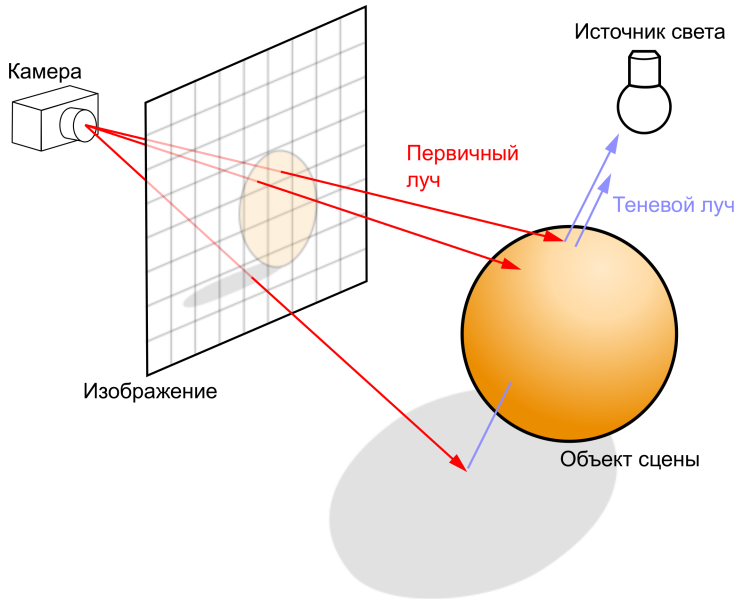
В прямом методе траектории лучей строятся от источника ко всем точкам всех объектов сцены (первичные лучи). Затем проверяется ориентация каждой точки относительно источника, и, если она лежит на стороне объекта, обращенной в противоположную от источника сторону, точка из расчетов освещенности исключается. 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 (заканчивается). 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.

Для построенных таким образом траекторий новых лучей может быть только три исхода. Луч либо выходит за пределы видимой из камеры области сцены, в этом случае все проделанные для него до этого момента расчеты освещенности отбрасываются, поскольку они не принимают участия в формировании изображения. Или луч попадает в камеру, тогда рассчитанная освещенность формирует цвет соответствующего пиксела изображения. Или луч встречает на своем пути новый объект, тогда для новой точки пересечения повторяется расчет освещенности и построения лучей отражения и преломления в зависимости от свойств поверхности объекта (рекурсия). Построение новых траекторий и расчеты ведутся до тех пор, пока все лучи либо попадут в каме-

ру, либо выйдут за пределы видимой области. Очевидно, что при прямой трассировке лучей мы вынуждены выполнять расчеты для лучей, которые не попадут в камеру, то есть, 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(в чистом виде).

3.1.2. Обратный метод трассировки лучей

Обратной метод трассировки лучей, или backward ray tracing. Этот метод расчетов основывается на построении лучей от наблюдателя через плоскость экрана вглубь сцены, а не от источника, то есть – наоборот. 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, количество 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. 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 точка к источнику или от него), наличие объектов, закрывающих точку от источника света, и расстояние до источника света. Если точка обращена в противоположную сторону от источника света или закрыта другим непрозрачным объектом, освещенность от такого источника не рассчитывается, 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 локальной модели освещенности, как сумма освещенностей от всех источников, для которых эта точка не закрыта другими объектами. Этот тип луча получил название ***shadow ray*** (иногда его еще называют illumination ray) – теневой луч. 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 (освещенности или тени) присваивается тому пикселю видового окна, через который проходит соответствующий первичный луч.

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

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

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 transparency ray, т. е. луч преломленный. 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 растространяется 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, если нет – используется интенсивность и цвет фона.

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: неучет вторичного освещения от диффузно отраженного объектами света; низкая скорость и высокая вычислительная стоимость расчетов – в классическом рейтресинге 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"линий и т. д.; дискретность определяющих цвет пиксела первичных лучей – одного первичного луча

недостаточно для корректного определения цвета пиксела, формирующего изображение.

3.2. Модели освещения

В соответствии с принятым в компьютерной 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 освещенности (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 освещенности к настоящему моменту разработано несколько. Самая первая,и самая простая – локальная модель освещенности. Эта модель не рассматривает процессы светового взаимодействия объектов сцены между собой, а только расчет освещенности самих объектов. Вторая, глобальная модель освещенности – Global Illuminations, 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 (ray tracing), рассеянное освещение (radiosity), каустик(caustic) и фотонные карты (photon mapping) и другие.

3.2.1. Глобальные модели освещения

Глобальное освещение (global illumination) — 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 (прямое освещение, англ. direct illumination), но и такие случаи, в которых лучи света от одного и того же источника, отражаются на других поверхностях сцены (непрямая освещенность, англ. 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, однако, 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 (оформление интерьеров, архитектура)

3.2.3. Модель Фонга

Это эмпирическая модель. В самом общем случае, в свете требования фотореалистичности, эта модель учитывает и неявное ambient-освещение. Ambient-освещение, или его еще называют фоновым (background), – это окружающее объект освещение от удаленных источников, чье положение 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. 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, характеризующий 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 что бы точно ориентировать камеру, необходимо указать следующие вектора:

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

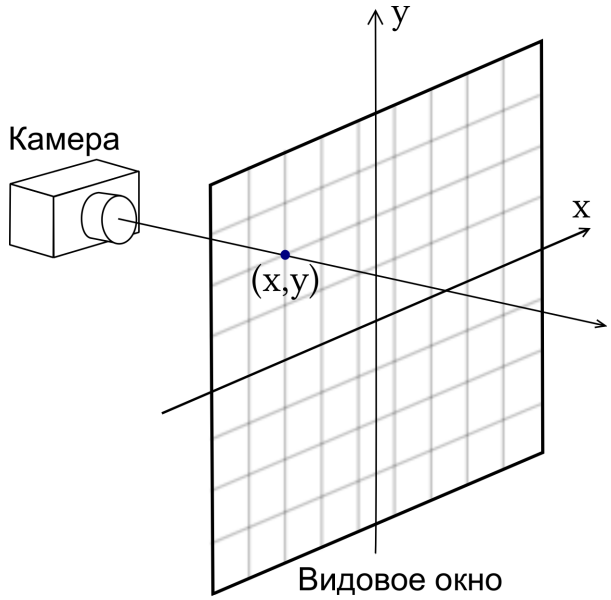
C_p - задает точку в пространстве, определяющую положение камеры (в общей модели координат)

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. Рассчет луча

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



Расчет луча осуществляется достатально просто. Луч определяется 2 векторами: положением и направлением. Положение луча совпадает с положением камеры. А направление вычисляется слудующим образом. Пусть необходимо рассчитать луч, для точки с координатами $(\hat{x}, \hat{y}) : 0 \leq \hat{x} < width, 0 \leq \hat{y} < height$ Для это 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:

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

3.3.2. Алгоритм антиалиасинга

Алгоритм антиалиасинга очень похож на алгоритм расчет луча. После рассчета (x, y) следует уточнить данные координаты: происходит разбиение прямоугольника, образованного точками $(x, y); (x + 1, y); (x + 1, y + 1); (x, y + 1)$ с помощью равномерной сетки $a \times a$. Таким образом, общие количество генерируемых лучей увеличивается в a^2 раз.

3.4. Примитивы

3.4.1. Плоскость

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

Уравнение луча:

$$\begin{cases} x = x_p + t \cdot x_d \\ y = y_p + t \cdot y_d \\ z = z_p + t \cdot z_d \end{cases} \quad (1)$$

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:

$$Ax + By + Cz + D = 0 \quad (2)$$

Для того, что бы найти точку пересечения луча с плоскостью, необходимо подставить уравнение (1) в (2):

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

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, что бы 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 $point = \vec{P} + t \cdot \vec{D}$

3.4.2. Сфера

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

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2 \tag{3}$$

где $\vec{C} = \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix}$ - центр сферы, а r - радиус. Подставим уравнение (1) в (3):

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

раскроем скобки, приведем подобные

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

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

приведем уравнение в виду

$$a \cdot t^2 + b \cdot t + c = 0 \tag{4}$$

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 (4) 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 $a = (\vec{D}, \vec{D}) \neq 0$

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

3.4.3. 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. Легко видеть, что барицентрические координаты не зависят от выбора z. Точка $T(u, v)$, принад-

лежащая треугольнику, может быть записана в виде:

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2 \tag{5}$$

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(1) и треугольником(5), 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:

$$\left[-\vec{D}, \vec{V_1} - \vec{V_0}, \vec{V_2} - \vec{V_0}\right] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \vec{P} - \vec{V_0} \tag{6}$$

Что бы 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 (6), используя метод Крамера:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\vec{D}, \vec{E_1}, \vec{E_2}|} \begin{bmatrix} | & \vec{T}, & \vec{E_1}, & \vec{E_2} & | \\ | & -\vec{D}, & \vec{T}, & \vec{E_2} & | \\ | & -\vec{D}, & \vec{E_1}, & \vec{T} & | \end{bmatrix} \tag{7}$$

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 (7).

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\vec{D} \times \vec{E_2}) \cdot \vec{E_1}} \begin{bmatrix} (\vec{T} \times \vec{E_1}) \cdot \vec{E_2} \\ (\vec{D} \times \vec{E_2}) \cdot \vec{T} \\ (\vec{T} \times \vec{E_1}) \cdot \vec{D} \end{bmatrix} = \frac{1}{(\vec{S}, \vec{E_1})} \begin{bmatrix} (\vec{Q}, \vec{E_2}) \\ (\vec{S}, \vec{T}) \\ (\vec{Q}, \vec{D}) \end{bmatrix} \tag{8}$$

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 (8) 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 мы избавились 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:

$$
\begin{aligned}
\vec{\tilde{E}}_1 &= -\vec{E}_1 = \vec{V}_0 - \vec{V}_1\\
\vec{\tilde{E}}_2 &= \vec{E}_2 = \vec{V}_2 - \vec{V}_0\\
\vec{\tilde{T}} &= -\vec{T} = \vec{V}_0 - \vec{P}\\
\vec{\tilde{N}} &= -\vec{N} = -(\vec{E}_2\times\vec{E}_1) = (\vec{E}_2\times-\vec{E}_1) = (\vec{\tilde{E}}_2\times\vec{\tilde{E}}_1)\\
\vec{\tilde{Q}} &= -\vec{D}\times\vec{T} = -(\vec{D}\times\vec{T}) = -(\vec{D}\times(-\vec{\tilde{T}})) = (\vec{D}\times\vec{\tilde{T}})\\
-\vec{D}\cdot\vec{N} &= (-\vec{D}\cdot(-\vec{\tilde{N}})) = (\vec{D}\cdot\vec{\tilde{N}})\\
\vec{T}\cdot\vec{N} &= (-\vec{\tilde{T}})\cdot(-\vec{\tilde{N}}) = (\vec{\tilde{T}}\cdot\vec{\tilde{N}})\\
\vec{\hat{Q}}\cdot\vec{E}_2 &= (-\vec{D}\times\vec{T})\cdot\vec{E}_2 = (\vec{\tilde{T}}\times\vec{D})\cdot\vec{\tilde{E}}_2\\
-\vec{\hat{Q}}\cdot\vec{E}_1 &= (\vec{\tilde{T}}\times\vec{D})\cdot\vec{\tilde{E}}_1
\end{aligned}
$$

В результате получаем следующие решение:

$$
\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\vec{D}\cdot\vec{\tilde{N}})} \begin{bmatrix} (\vec{\tilde{T}}\cdot\vec{\tilde{N}}) \\ (\vec{\tilde{S}}\cdot\vec{\tilde{E}}_2) \\ (\vec{\tilde{S}}\cdot\vec{\tilde{E}}_1) \end{bmatrix} \tag{9}
$$

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 (9) 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: }
06:
```

Т. о. 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

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: $N! = N \cdot (N-1) \cdot \cdots \cdot 1$

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 мы указало 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. Что бы 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 имеено 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 ни осталось и при уровне оптимизации начиная с O1, компилятор подсчитает выражение и вместо:

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

Подставит, подсчитанное выражение:

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

Проверить это можно дизассемблировав данный пример и увидеть в коде число 5040.

4.2. SIMD инструкции

Применение SIMD инструкций в классе vec4.

4.2.1. Базовые операции в классе vec4

4.2.2. Скалярное произведение векторов

4.3. Ускоряющие структуры

4.3.1. Алгоритм построения BVH

4.3.2. Алгоритм траверса луча через BVH

5. Постановка и результаты экспериментов

Эксперименты проводилимь на 6 ядерном компьютере с процессором Intel Core i7 980x с частотой 3.33GHz, оперативной памятью 12 Гб, ОС - Calculate Linux 11.0 x64.

Для сборки приложения с библиотекой tbb, необходимо, что бы компилятор поддерживал лямбда выражения из нового стандарта c++0x. На данный момент, этот стандарт поддерживает компилятор GCC 4.5.2

5.1. Timer

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

Для очень точной оценки времени работы алгоритмов, был специально написан высокоточный таймер на языке c++, с использованием ассемблера (AT&T - синтаксиса). Таймер выдает время в тактах процессора и включает в себя разные режимы подсчета времени.

5.1.1. Алгоритм работы высокоточного таймера

Основной код, выполняющий замеры времения приведен ниже.

Листинг 8. 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(). Вначале необходимо вызвать инструкцию cpuid, для того что бы 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(), Иснтрукция 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 инструкцией rdtscp[3]. Далее просиходит вычисление разности времения 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.

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.

5.1.2. Эксперименты с высокоточным таймером

5.2. Вектора и Expression templates

5.3. Тестовая сцена

5.4. Эффективность расспараллеливания

5.5. 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.

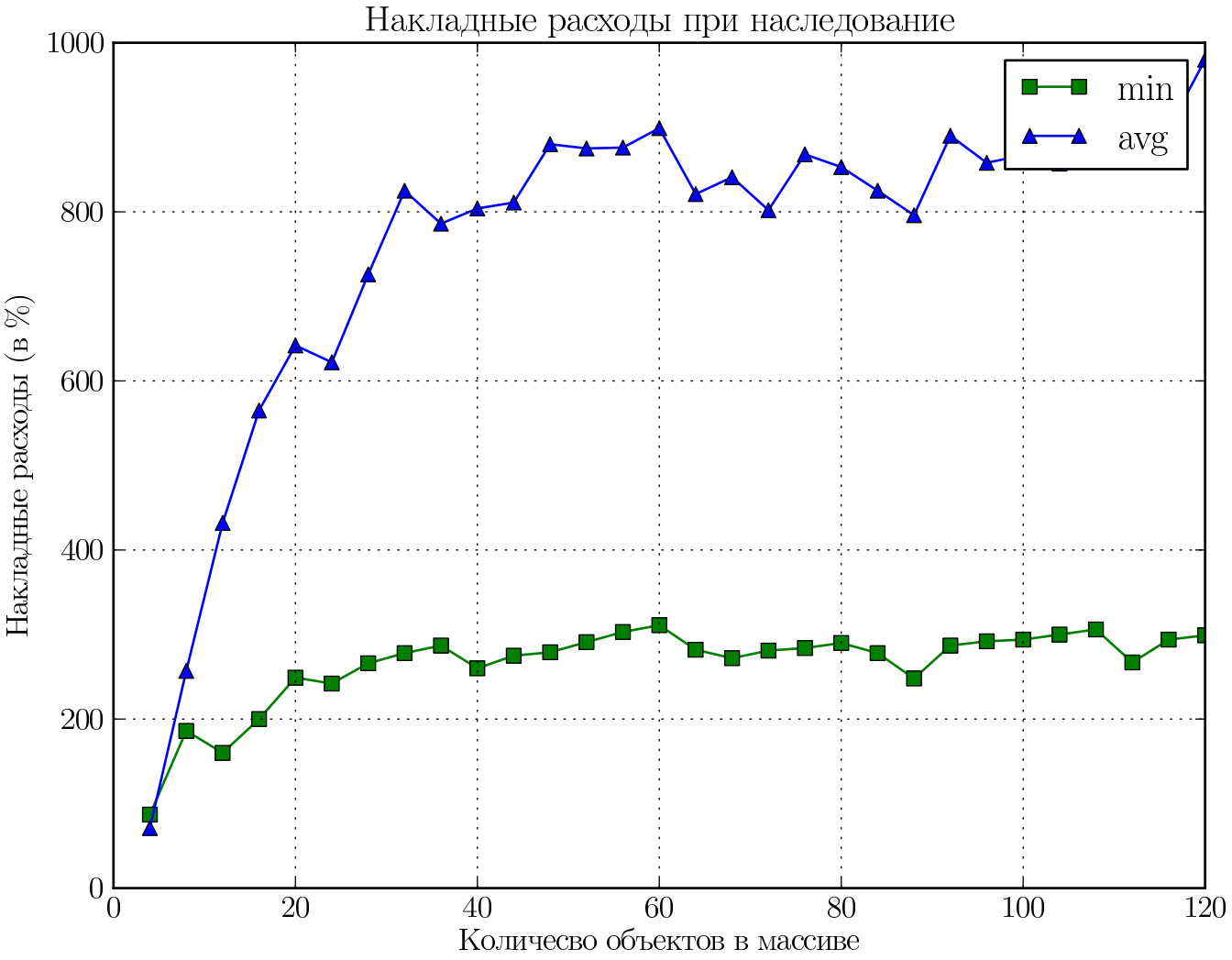
[3]rdtscp - инструкция появилась лишь в процессорах Intel Core i7

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.

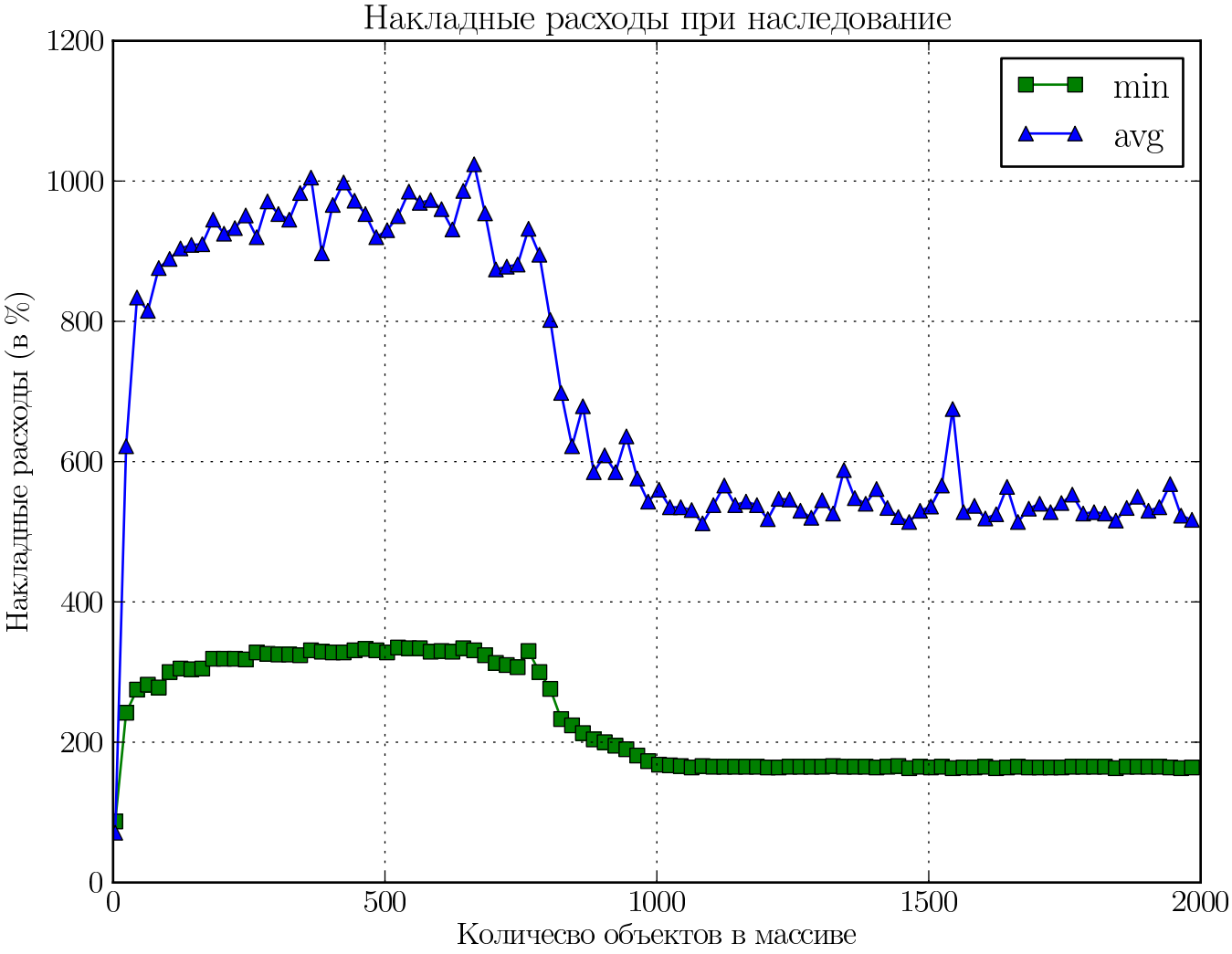
При использование полиморфизма возникают накладные расходы связанные с тем, что при вызове операций у абстрактного класса, необходимо во время испонения определить, к какому классу принадлежит данный объект и вызвать соответствующий метод.

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.



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



5.6. TBB vs OpenMP

5.7. Влияние SIMD инструкций на скорость работы

5.8. Компилирование высокоуровнего кода в ассемблер

Ассемблерные вставки, которые показывают что вычисление отражающего луча это одно SSE.

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 стандарта параллельного OpenMP. Благодаря использованию языка c++ и технику шаблонных выражений, удалось еще повысить производительность программы. Программа продемонстрировала хорошую производительность: используя всего лишь один процессор можно получать изображения в реальном времени.

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

Предметный указатель