МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра №806 «Вычислительная математика и программирование»

Курсовой проект

по курсу «Программирование графических процессоров»

Обратная трассировка лучей (Ray Tracing). Технологии MPI, CUDA и OpenMP

Выполнил: М.А. Бронников
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы. Совместное использование технологии MPI, технологии CUDA и технологии OpenMP для создание фотореалистической визуализации. Создание видеоролика/анимации.

Задание.

Сцена. Прямоугольная текстурированная поверхность (пол), над которой расположены три платоновых тела. Сверху находятся несколько источников света. На каждом ребре многогранника располагается определенное количество точечных источников света. Грани тел обладают зеркальным и прозрачным эффектом. За счет многократного переотражения лучей внутри тела, возникает эффект бесконечности.

Камера. Камера выполняет облет сцены согласно определенным законам. В цилиндрических координатах (r, φ, z), положение и точка направления камеры в момент времени t определяется следующим образом:

$$r_c(t) = r_c^0 + A_c^r sin(\omega_c^r \cdot t + p_c^r)$$

$$z_c(t) = z_c^0 + A_c^z sin(\omega_c^z \cdot t + p_c^z)$$

$$\varphi_c(t) = \varphi_c^0 + \omega_c^\varphi t$$

$$r_n(t) = r_n^0 + A_n^r sin(\omega_n^r \cdot t + p_n^r)$$

$$z_n(t) = z_n^0 + A_n^z sin(\omega_n^z \cdot t + p_n^z)$$

$$\varphi_n(t) = \varphi_n^0 + \omega_n^\varphi t$$

где

$$t \in [0, 2\pi]$$

Требуется реализовать алгоритм обратной трассировки лучей (http://www.ray-tracing.ru/) с использованием технологии CUDA. Выполнить покадровый рендеринг сцены. Для устранения эффекта «зубчатости», выполнить сглаживание (например с помощью алгоритма SSAA). Полученный набор кадров склеить в видеоролик любым доступным программным обеспечением. Подобрать параметры сцены, камеры и освещения таким образом, чтобы получить наиболее красочный результат. Провести сравнение производительности gpu и cpu (т.е. дополнительно нужно реализовать алгоритм без использования CUDA).

Вариант:

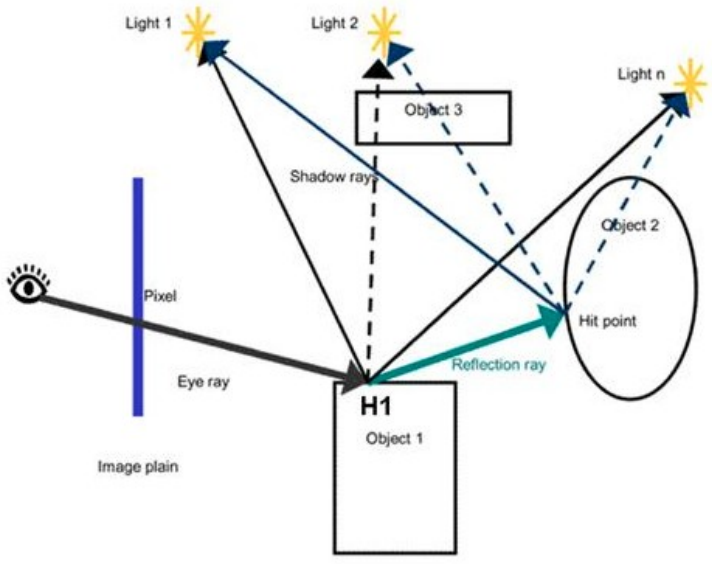
4. Тетраэдр, Октаэдр, Додекаэдр

Программное и аппаратное обеспечение

Device: GeForce GTX 1050 with Max-Q Design
Размер глобальной памяти: 4238737408
Размер константной памяти : 65536
Размер разделяемой памяти: 49152
Регистров на блок: 65536
Максимум потоков на блок: 1024
Количество мультипроцессоров : 5
OS: Linux Ubuntu 18 LTS
Редактор: VSCode
Версии софта: g++ 4.8.4, nvcc 7.0

Метод решения

Алгоритм выглядит следующим образом: из виртуального глаза через каждый пиксел изображения испускается луч и находится точка его пересечения с поверхностью сцены (в данной работе сцена состоит только из треугольников, точечные источники не в счет). Лучи, выпущенные из глаза называют первичными. Далее необходимо определить для каждого источника освещения, видна ли из него найденная точка пересечения(в даннной все источники света точечные). Тогда для каждого точечного источника света, до него испускается теневой луч из точки, который определяет освещается ли данная точка конкретным источником. Если теневой луч находит пересечение с другими объектами, расположенными ближе чем источник света, значит, точка находится в тени от этого источника и освещать ее не надо(но в данной работе мы освещение от источника домножаем на цвета и коэффициенты прозрачности всех преград, которые встретились на пути теневого луча). Далее, считаем освещение по локальной модели (в данной работе это Фонг, но можно выбрать Ламберта). Освещение со всех видимых источников света складывается. Далее, если материал объекта перечечения имеет отражающие свойства, из точки испускается отраженный луч и для него вся процедура трассировки рекурсивно повторяется(результат освещения будет добавлен к общему освещению, посчитанному от источников света с умножением на коэффицент отражения). Аналогичные действия выполненяются, если материал имеет преломляющие свойства.



Описание программы

HL-архитектура разбита по файлам:

1. main.cu — считывание параметров, инициализация сцены и MPI.
2. camera.cuh — класс для работы с камерой, динамически меняющей свое состояние во времени.
3. file_writter.cuh — небольшая обертка для записи рещультата в файл по модификатору, считанному из ввода.
4. structures.cuh - описание основных структур проекта(например float_3 — трехкомпонентный вектор и структура triangle) вместе со стандартными функциями и операторами для работы с ними(произведения векторов, небольшие вспомогательные функции типа перевода цветов из uint32_t в float_3).
5. ray_tracing.cuh — основной функционал алгоритма рендеринга, где приведены основные функии для расчета освещения в сцене(рассмотрим поподробнее далее).
6. figures.cuh — генерация и добавление на сцену фигур тетраедера, октаедера, додекаедера вместе с их диодами на ребрах(все диоды — небольшие белые треугольники и один точечный источник света в середине фигуры)
7. ray_cleaner.cuh — функционал для очистки «мертвых» лучей после каждого вызова рекурсии(как на GPU, так и CPU). Файл основан на 5-ой Л.Р, от куда я взял реализацию скана и часть кода для сортировки лучей.
8. material_table.cuh — небольшая вспомогательная таблица для удобства работы с различными свойствами материалов сцены.(каждый трегольник содержит не сами свойства материала, а индекс из этой таблицы)
9. scene.cuh — основной класс сцены, в котором выделяется память на бэкэндах и вызывается функционал рендеринга. Содержит и управляет текстурами, треугольниками, источниками света и выпущенными лучами и итоговым изображением.
10. mpi_io.cuh — функционал по вводу.выводу для MPI.

Рассмотрим наиболее важный файл: ray_tracing.cu:

Сначала посмотрим на метод init_viewer_back_rays(), который вызывается перед отрисовкой изображения и определяет позицию и направление каждого луча, пущенного от наблюдателя через пиксель и инициализирует изображение черным светом. Для расчета обратных лучей определяется координата пикселя в локальных координатах,где точка отсчета — наблюдатель, после чего координата перемножается на матрицу перехода от одного базису к другому, определенную позицией наблюдателя в пространстве(файл camera.cuh).

Перейдем к основному методу расчета очередного уровня глубины рекурсии отрисовки: ray_trace():

1. Для начала берется очередной луч(каждый луч имеет привязку к конкретному пикселю) и ищется пересечение с ближайшим треугольником сцены с помощью

метода search_triangle_intersection(). Если пересечение не найдено, то луч нас не интересует более, отбрасываем его.
Метод ray_trace принимает на вход начальный индекс и длину шага по массиву лучей, что делает его универсальным как для выполнения на device GPU, так и с помошью OMP CPU.
2. Определяем нормаль и свойства материала пересечения:

a. Для определения нормали используется метод normal_of_triangle(), который ищет нормаль как векторное произведение 2-ух сторон треугольника. После чего направление нормали нужно откорректировать так, чтобы она «смотрела в сторону наблюдателя»(косинус между нормалью и лучом был отрицательный).

b. Для найденного треугольника из таблицы достается свойства его материала, однако к ним добавляется цвет текстуры в точке, если пересеченныц треугольник — текстура.

3. «Перепрыгиваем» к пересечению и «пускаем лучи к источникам света в for цикле. Результат освещения будет суммой освещения от каждого теневого луча.

Для каждого источника света:
4. Если источник света находится за треугольником(его поверхность освещается с обратной стороны), то этот источник нас не интересует(в будущем можно подумать о том, чтобы учитывать и его тоже с домножением на коэффицент прозрачности).
5. Определяется общий коэффицент прозрачности пути теневого луча к свету функцией compute_radiocity_losses(), который влияет на силу освещения светом этой точки.
6. Для расчета освещения от точечного источника используем модель Фонга. В качестве коэффицента бликового освещения я решил использовать коэффицент прозрачности в данной работе.

Вне цикла:
7. Записываем полученную сумму освещений в глобальную память. Необходимо использовать atomic, поскольку при глубине рекурсии > 1 к одному пикселю может быть привязано больше одного луча, который попытается также записать в этот участок. atomicAdd для float_3 я реализовал через 3 встроенных atomicAdd для обычного float.
8. Разветвляем лучи на преломленный и отраженный, используя в качестве мощности этих лучей — мощность текущего луча умноженную на коэффициенты прозрачности и отражения соответственно. (Сила луча нужна для учета того, чтобы определит насколько сильно влияет посчитанное освещение на освещенность пикселя, а также используется при очистке мертвых лучей, если она стала ниже заданного порога)
9. Конец очередной реккурсивной иттерации.

Для расчета пересечения с треугольниками сцены я использую популярный барицентрический тест в методе triangle_intersected(), потому что он более понятен и в интернете полно готовых реализаций этого теста, на основе которых я и реализовал его

у себя, корректируя под общую логику программы. Метод в случае пересечения возвращает расстояние до него(иначе отрицательное число).

Метод поиска search_triangle_intersected() принимает контракт result_contract на поиск ближайшего пересечения треугольника и с помощью метода triangle_intersected ищет наиболее близкий к наблюдателю триугольник. В качестве результата в контракт заносится id треугольника и расстояние до него, а возвращает метод true <==> существует пересечение с треуольником.

Метод compute_radiocity_looses() похож на search_triangle_intersected(), но при проходе определяет общий трехкомпонентный показатель преломления пути от пересечения к источнику света(теневого луча), то есть берет все треугольники, расстояние до которых меньше, чем до источника, но больше 0(EPSILON, чтобы не пересекаться с самим собой).

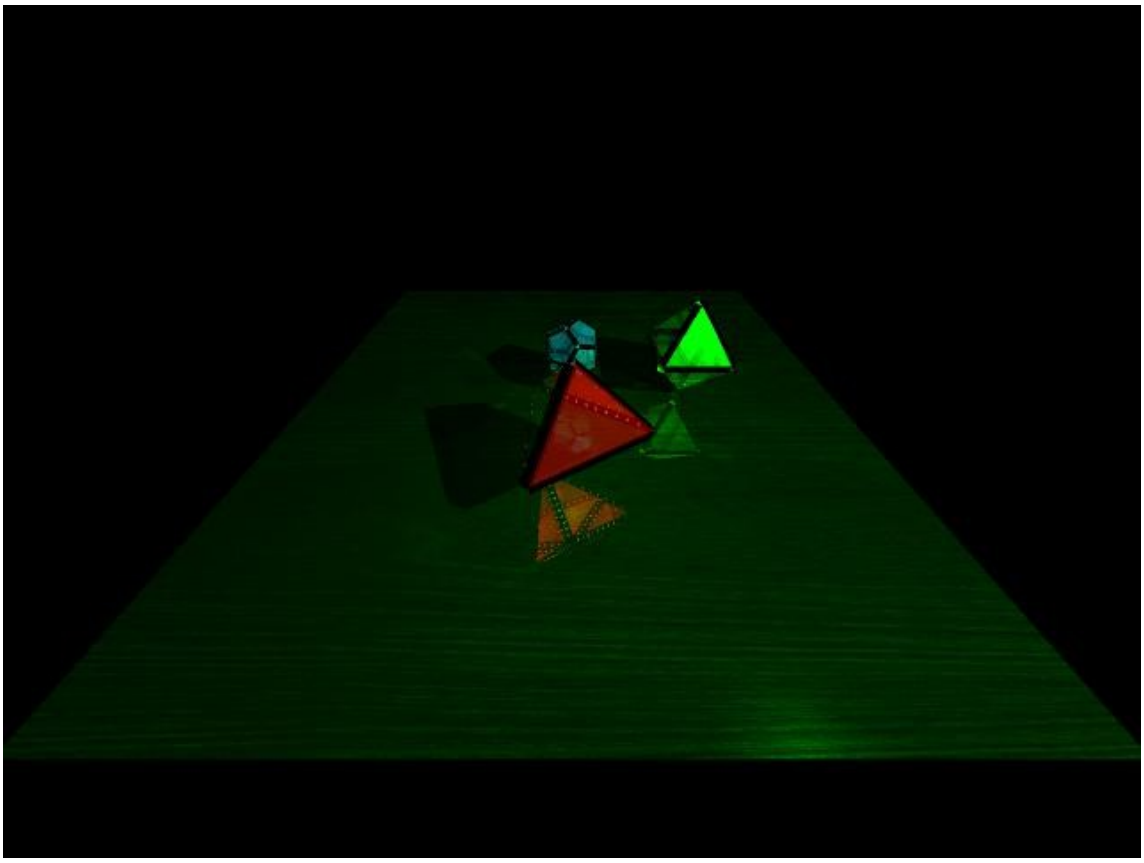
Метод color_of_texture_intersection имеет 2 реализации, зависящих от бэкэнда. Рассмотрим реализацию для GPU!
Для расчета цвета пересечения мы расчитываем барицентрические координаты пересечения с первым из треугольников текстуры. Именно их мы и используем в качестве нормализованных координат в методе tex2D, который позволяет получить также нормализованные и интерполированные координаты света хардварным путем при хранящихся uint32_t данных.

Остальные методы для ray_trace довольно просты и не буду их тут описывать дополнительно. Добавлю только про метод ssaa(), который проходит по изображению квадратным окном заданного размера и в записывает в итоговое изображение среднее значение по окну, приведенное из float_3 к uint32_t.

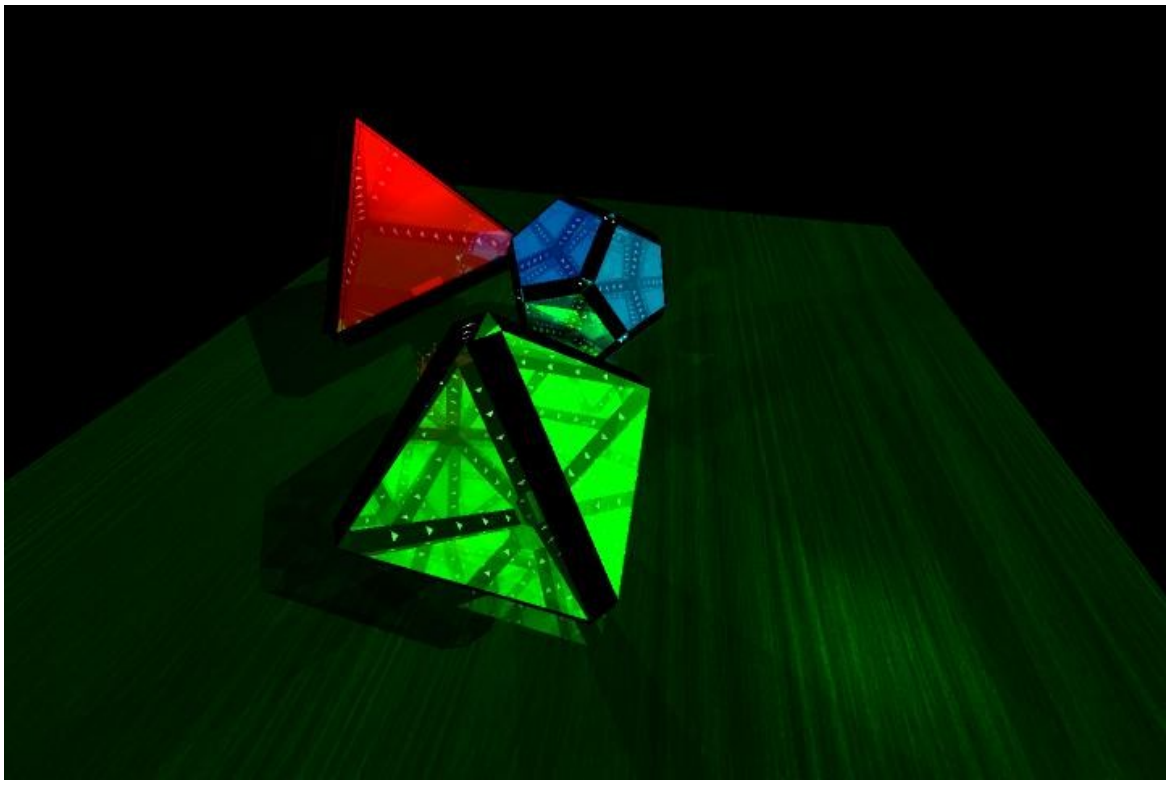
Стоит отметить, что основной функционал, который позволяет использовать технологию MPI реализован в int main(), где инициализируется и определяются параметры много процессорного взаимодействия, однак стоит обратить на то, что для синхронизированного ввода я добавил файл mpi_io.cuh, который предоставляет синхронизированное логировани и чтение входных значений и на изменения, которые коснулись класса камеры, где появилась возможность двигаться с заданным шагом и начальной позицией фрейма.

Результаты

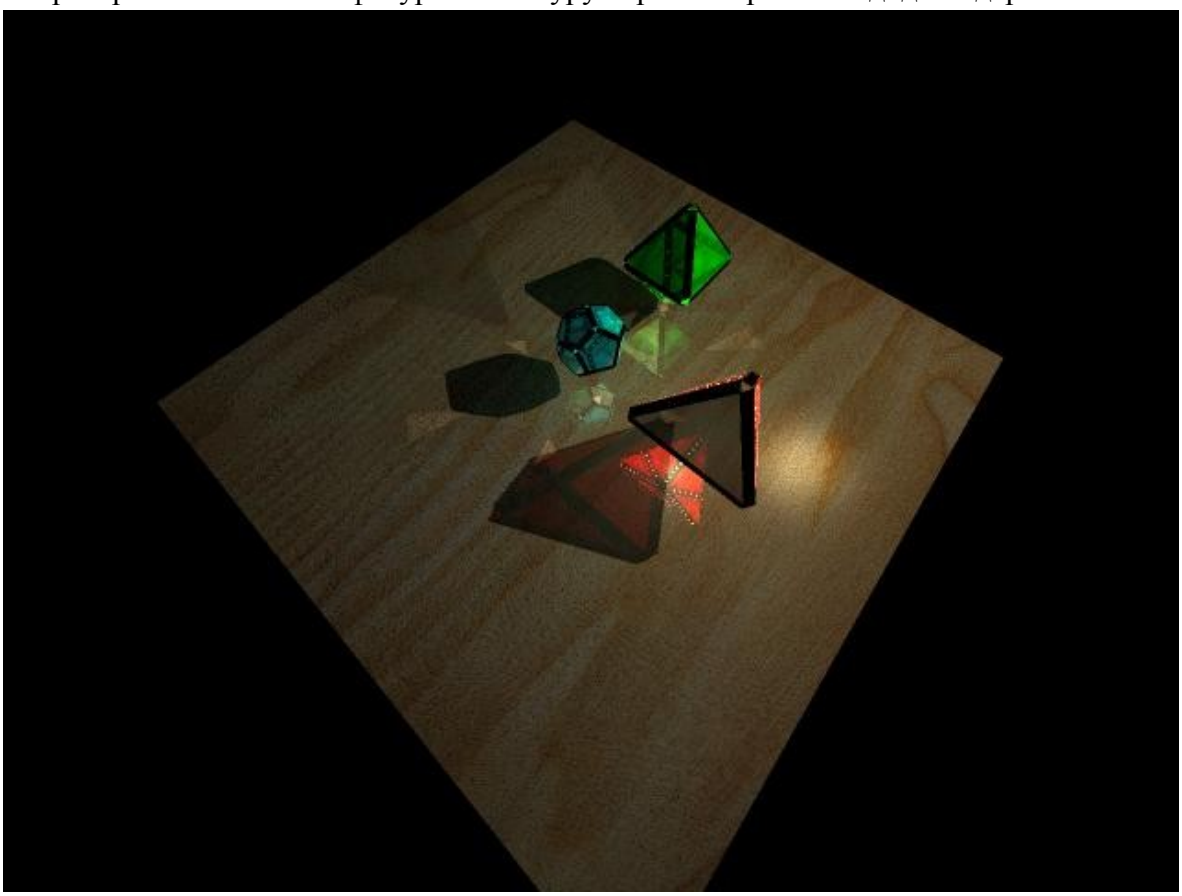
На этом скриншоте отлично виден эффект отражения, наличие текстуры и использование модели Фонга для освещения(блики внизу):



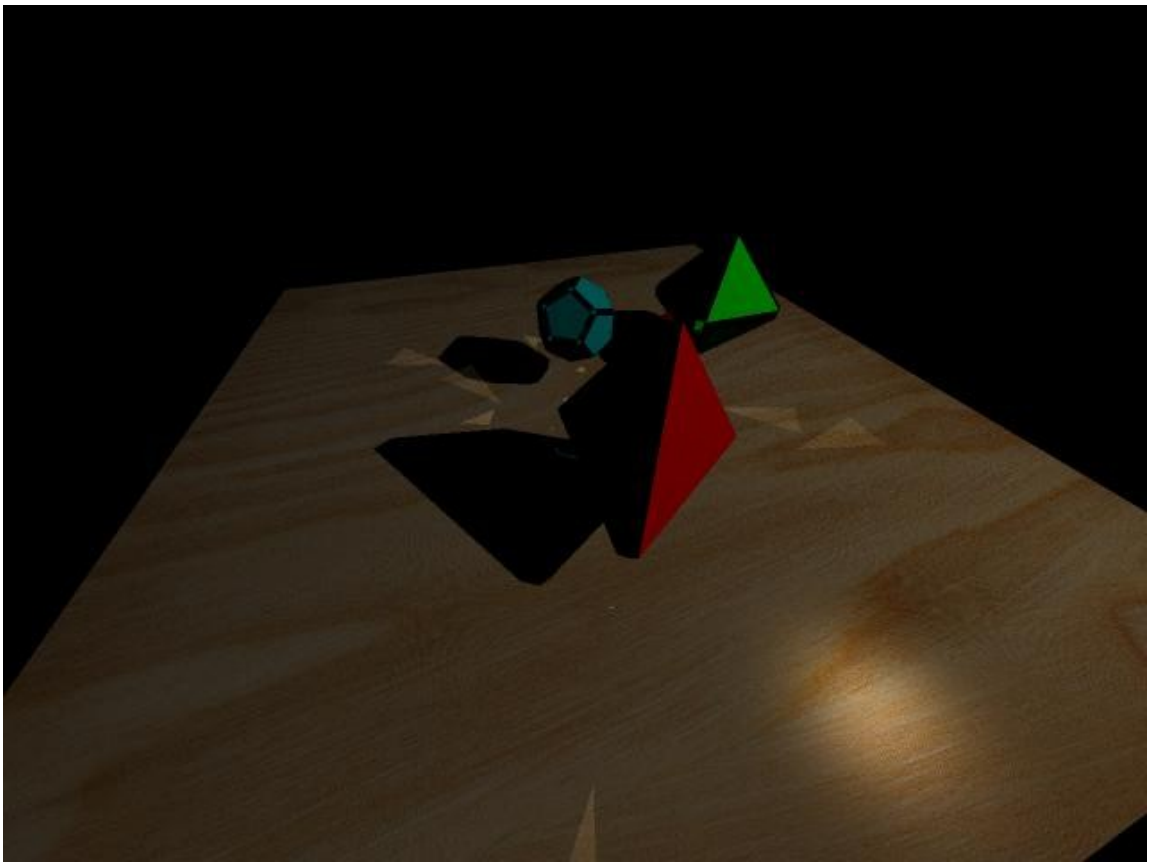
Эффект бесконечности на октаэдере:



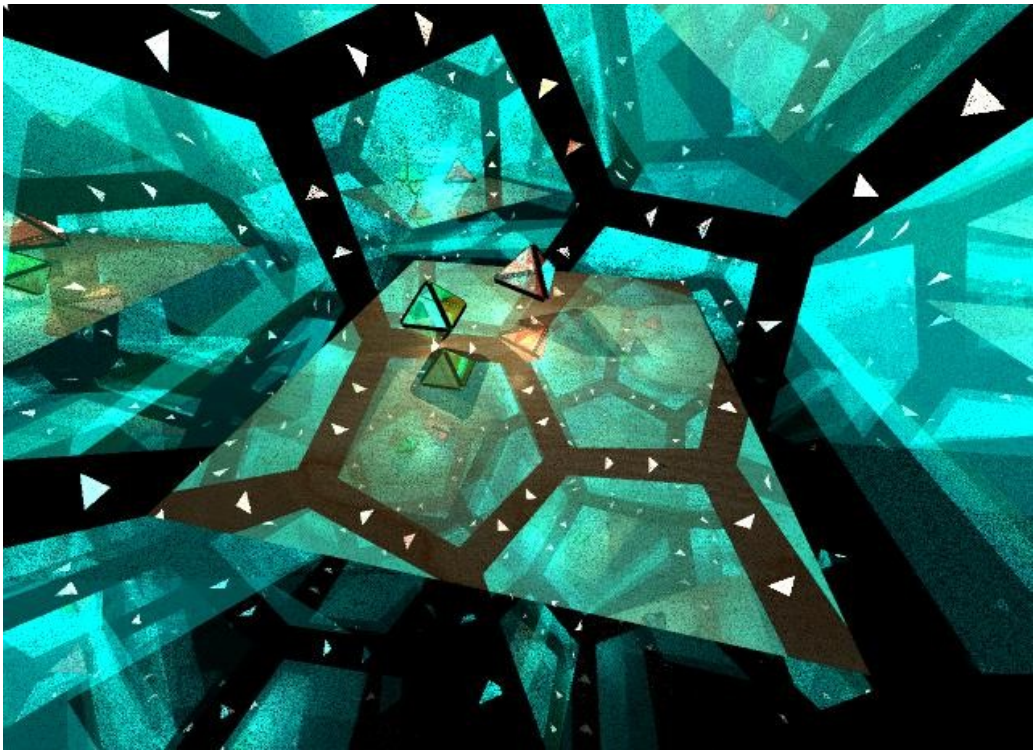
Распространение света из фигур на текстуру через отверстия в додэкаедере:



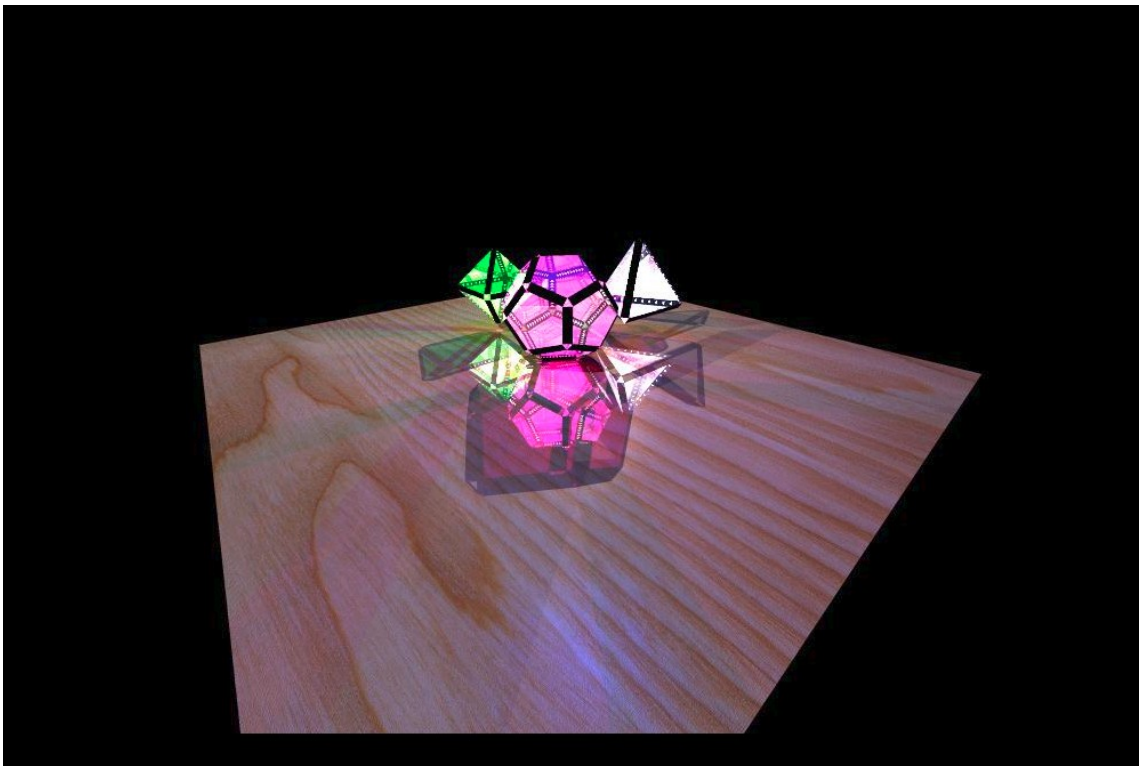
Фигуры без отражений и прозрачности(свет на паркете проходит через отверстия):



Внутри додекаэдера с толстыми ребрами(они зависят от радиуса фигуры) и глубиной рекурсии - 4:

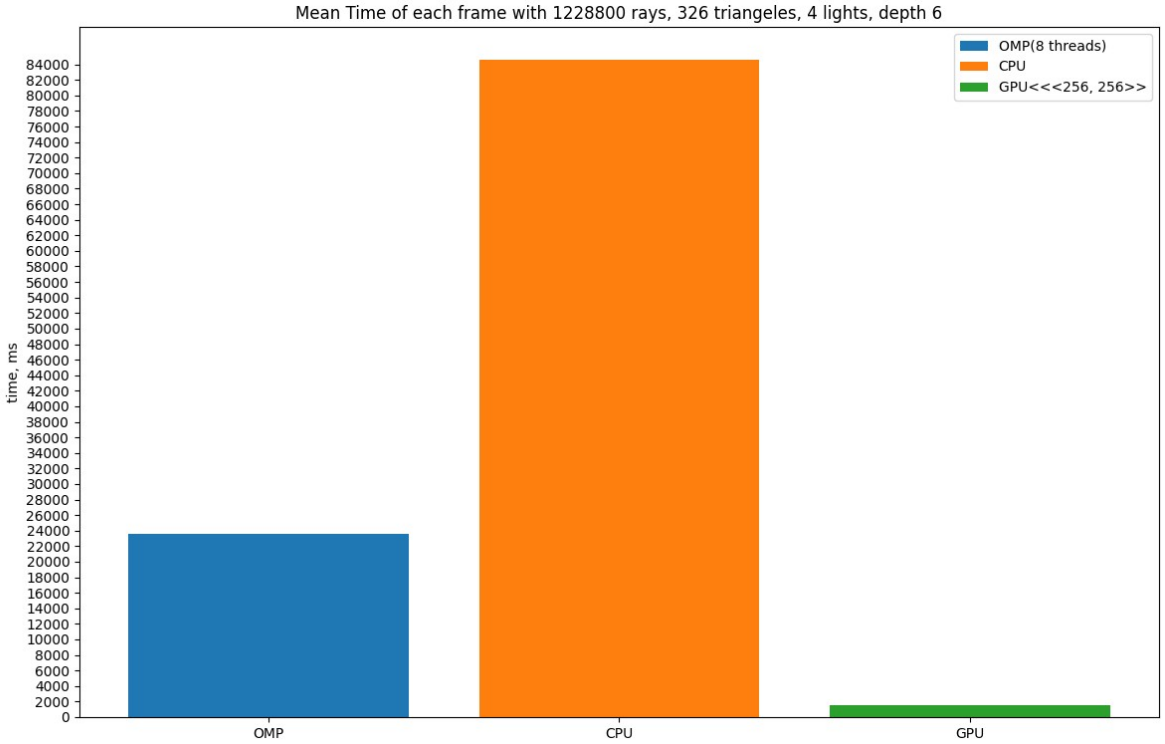


Мой любимый кадр:



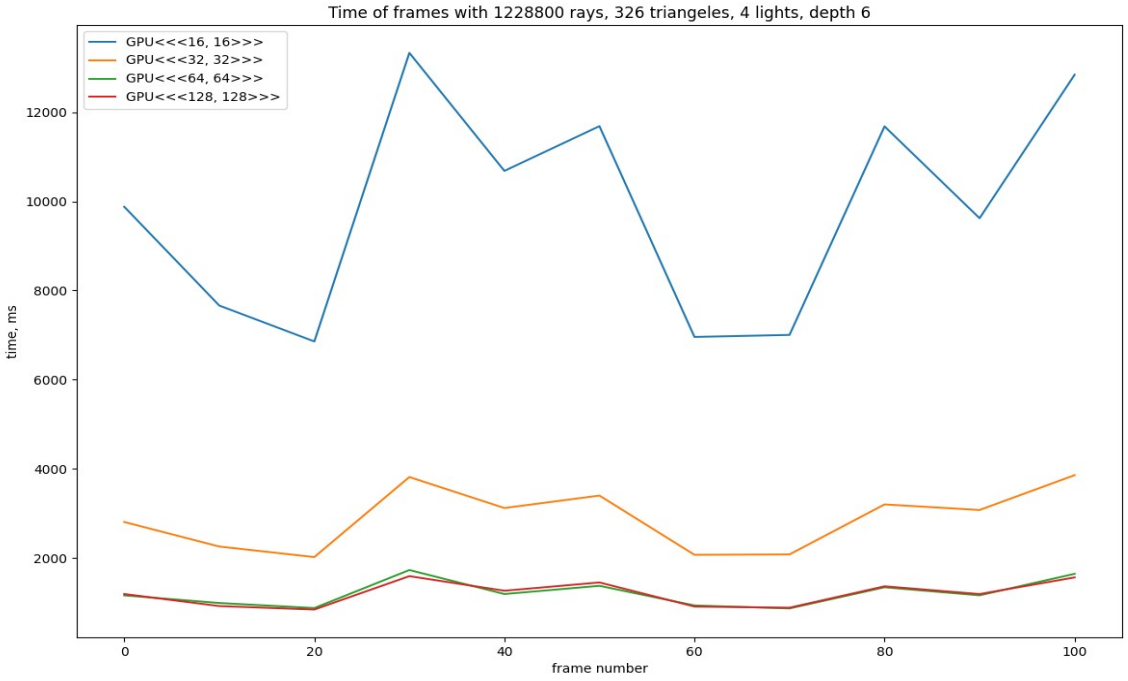
Исследовательская часть

Для начала покажем почему для обратного распространения луча GPU подходит как нельзя лучше. Среднее время генерации одного каждра на GPU(256 блоков, 256 потоков) против одного потока центрального процессора и CPU(без MPI) вместе с OMP:



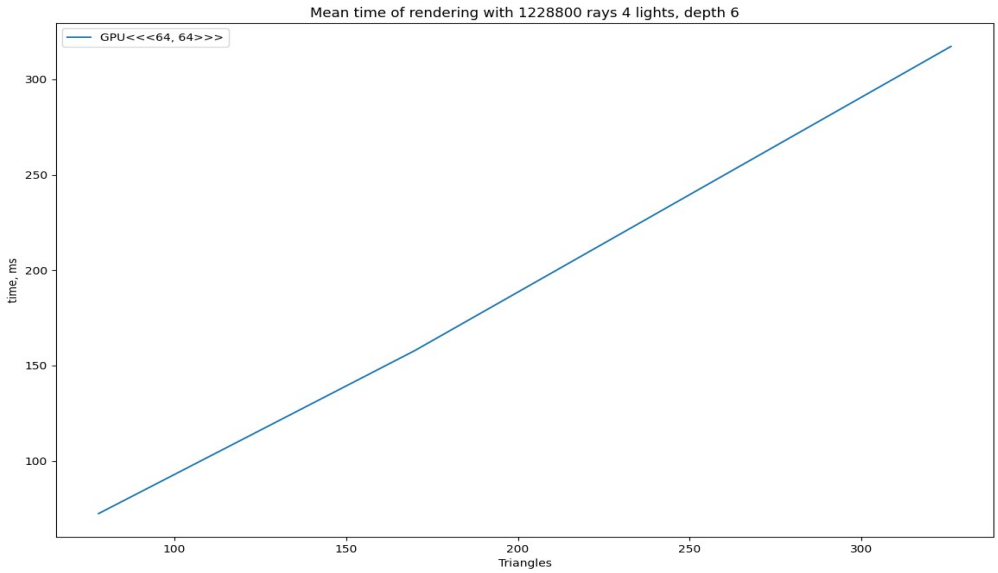
На графике не видно, но среднее время работы GPU получилось 1489.765мс за один фрейм при указанной конфигурации запуска, что сильно опережает результат CPU.

Сравним как влияет на производительность размеры блоков на одной и той же сцене



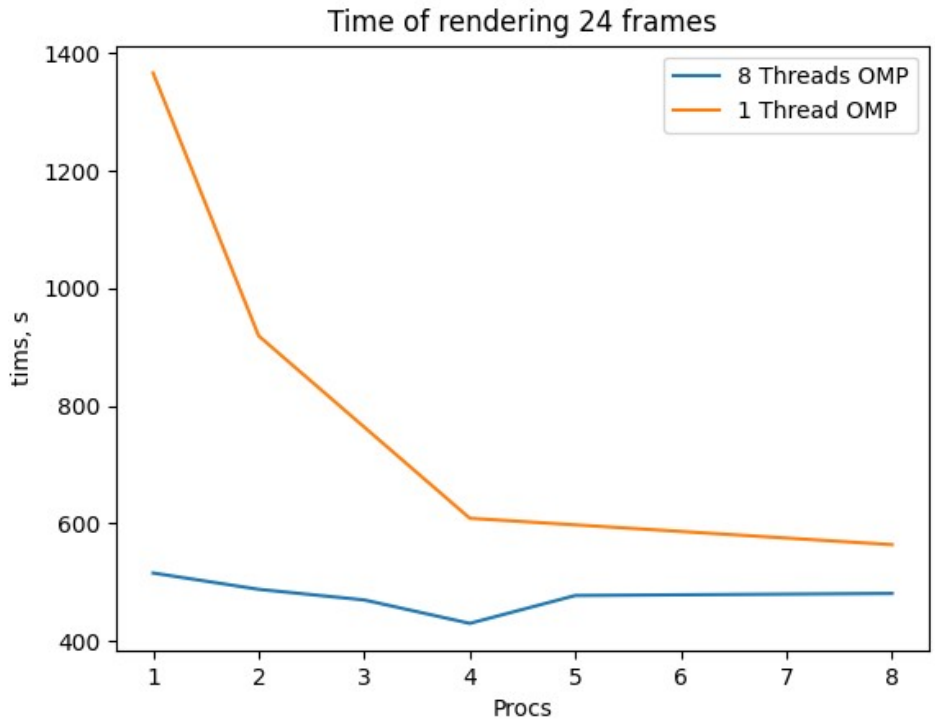
Я объясняю нижний порог около 700мс на фрейм, ниже которого не опускается время рендеринга кадра методикой измерений, поскольку я замеряю полный цикл генерации изображения, включая инициализацию, очистку лучей, ssaa и запись в файл.

Наконец оценим, как количество треугольников влияет на время рендеринга на GPU:



Мы наблюдаем практически линейную зависимость, поскольку каждый поток должен пройтись по всем треугольникам сцены.

Оценим, как влияет количество процессов MPI на время работы программы на CPU.



Данный результат объясняется тем, что у меня в расоряжении был лишь один компьютер с 16 ядрами процессора, что не позволяет раскрыться MPI во всей красе, особенно при дополнительной нагрузке процессора OMP потоками.
Наличие всего одной видеокарты в распоряжении также нивилирует все преимущества MPI, поскольку даже с одним рабочим процессом при RayTracing-е граффический процессор использует свои ресурсы(в особенности регистры) практически на пределе.

Выводы

Выполнив курсовую работу по «Программированию графических процессоров» я узнал как можно распараллелить уже знакомый мне алгоритм рендеринга - RayTracing еще сильнее, используя не только ресурсы граффического процессора, но и центральных процессоров, соединенных в целые кластеры.

В настоящее время этот алгоритм используется не так часто из-за своей прожорливости, однако трудно переоценить его потенциал в игровой индустрии и кинопроизводстве в ближайшие несколько лет при растущей производительности новых видеокарт, в особенности от компании Nvidia. Несмотря на то, что за счет неравномерности доступа к памяти этот алгоритм не является идеальным для получения максимума производительности от видеокарт, его реализация на одном CPU даже с такими технологиями распараллеливания процессов, как OMP, практически бессмысленна, а в реальном времени по просту невозможна, поэтому в этой работе как ни в одной лабораторной расскрылась польза от параллельных вычислений на графических процессорах. Более того, наличие целого кластера из компьютеров с GPU позволяет производить рендеринг последовательных кадров сцены довольно быстро, поскольку видеоряд из кадров обладет не только ресурсом параллелизма в пределах одного кадра(по пиксельно), но и возможностью обрабатывать каждый кадр ряда вне зависимости от остальных. Поэтому даже на текущий момент RayTracing написанный и оптимизированный профессионалами может быть представлен в Real-Time, что делает возможным использование его в различных сферах, где требуется реализм изображения на экране, таких как сфера развлечений и 3D-моделирование.

Не смотря на огромную трудоемкость этой работы, в которой пришлось использовать самые разные знания из области геометрии, параллельной обработки данных и программирования графических процессоров, мне было интересно выполнять эту работу и я горжусь полученным мной результатом.

Литература

1. http://www.ray-tracing.ru/articles164.html