МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Курсовой проект
по курсу «Параллельная обработка данных»

Обратная трассировка лучей (Ray Tracing) на GPU

Выполнил: А.В. Скворцов
Группа: 8О-408Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

Условие

Цель работы: использование GPU для создание фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание видеоролика.
Задание: 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.
Вариант: гексаэдр, октаэдр, додекаэдр.

Программное и аппаратное обеспечение

- GPU: Geforce 940MX
  - Compute capability : 5.0
  - Total Global Memory : 2147483648
  - Shared memory per block : 49152
  - Registers per block : 65536
  - Max threads per block : (1024, 1024, 64)
  - Max block : (2147483647, 65535, 65535)
  - Total constant memory : 65536
  - Multiprocessors count : 3
- CPU: Intel Core i5-6200U 2.30GHz
- RAM: 4GB
- Software: Windows 10, Visual Studio Code, nvcc

Метод решения

Обратная трассировка лучей — это метод создания фотореалистичных изображений, при котором мы исследуем взаимодействие отдельных лучей с поверхностями, причем отслеживание траектории лучей мы производим не от источника к экрану, а наоборот, что позволяет сохранить огромное число ресурсов и повысить производительность.
В общем виде алгоритм обратной трассировки выглядит следующим образом:

1. Для каждого пикселя камеры, выпускаем из него луч
2. Ищем пересечения луча с ближайшим к нему объектом. Если есть такое пересечение, переходим к пункту 3, иначе возвращаем цвет фона.
3. Для точки пересечения луча с объектом ищем его освещенность каждым источником света по некоторой локальной модели (например Фонга). Затем суммируем эти освещенности.
4. Если объект обладает отражающими свойствами, отслеживаем цвет отраженного луча и прибавляем его к итоговой освещенности, т.е процедура трассировки рекурсивно повторяется.
5. Если объект обладает преломляющими свойствами, отслеживаем цвет преломленного луча и прибавляем его к итоговой освещенности

Существуют различные усложнения алгоритма для его большей реалистичности и производительности, но в данной работе я реализовал простую модель.
Что касается модели освещения, то я использовал частный случай затенения по Фонгу, в общем случае формула вычисления освещенности выглядит следующим образом:

$$I = K_a I_a + K_d (n, l) + K_s (n, h)^p$$

где:

n — вектор нормали к поверхности
l — направление на источник света
h — направление на наблюдателя
K_a — коэффициент фонового освещения
K_s — коэффициент зеркального освещения
K_d — коэффициент диффузного освещения

В моем конкретном случае коффициенты K_s и K_a равны нулю, то есть я учитываю только диффузную составляющую, это немного портит картинку, когда объект находится близко к источнику освещения, но в целом изображение выглядит неплохо. Также я учитываю уменьшение интенсивности света с расстоянием, а итоговая освещенность от источника света обратно пропорциональна квадрату расстояния до этого источника).
Для вычисления отраженного луча используетя закон отражения, выглядящий следующим образом:

$$R_{refl} = R_{inc} - 2(N, R_{inc}) N$$

Для вычисления преломленного луча можно воспользоваться Законом Шелла, однако он зависит от коэффициентов преломления двух сред, поэтому чтобы не усложнять программу, было принято решения считать их равными единице для всех сред, а итоговая формула приняла следующий вид:

$$R_{refr} = R_{inc}$$

Описание программы

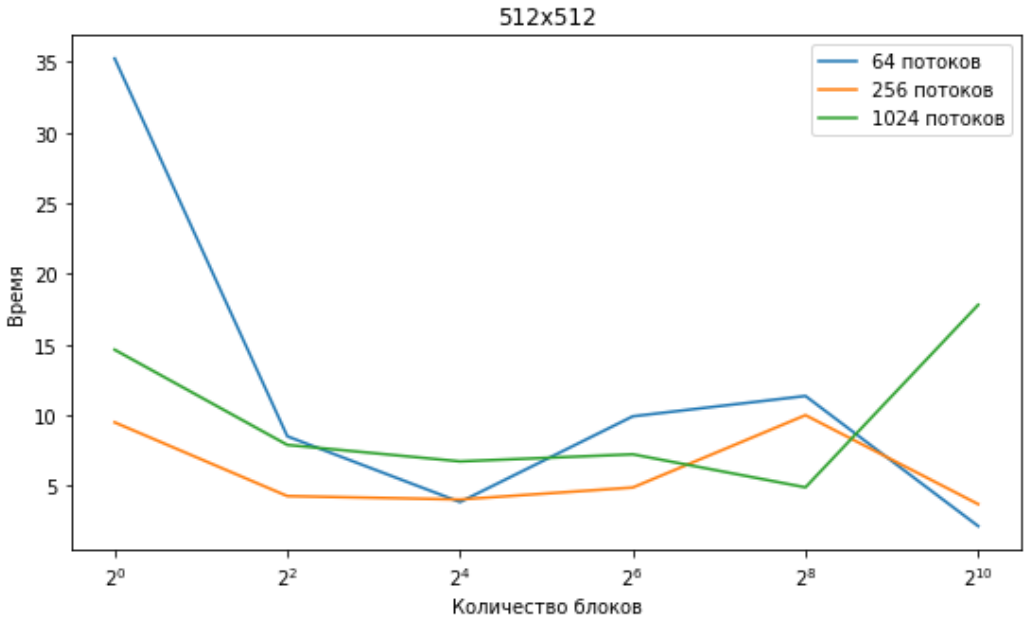
Для удобства организации кода, он поделен на несколько логически завершенных файлов:

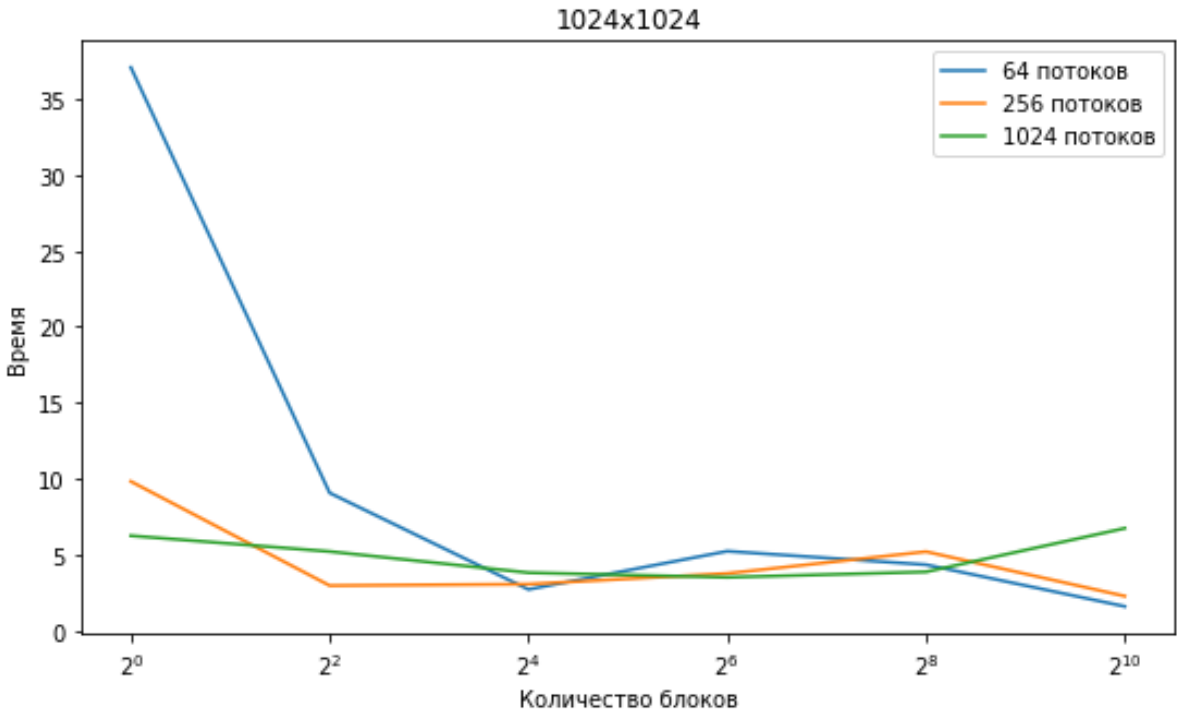
- csc.cuh – макрос для проверки значения cudaError_t
- ray.cuh – файл с классом, представляющим луч (начало луча и его направление)
- camera.cuh – файл с классом, представляющим камеру. Можно задавать ее характеристики: угол обзора, соотношение сторон, положение, направление взгляда.
- vec_math.cuh – файл, перегружающий математические действия с векторным типом float3.
- objects.cuh — файл, определяющий тела, которые могут быть отрисованы с помощью трассировки лучей. Представляет несколько абстрактных классов и их завершенных потомков.
    - Class HittableObject – абстрактный класс, реализующий интерфес всех отображаемых тел.
    - Class Sphere – класс, представляющий шар.

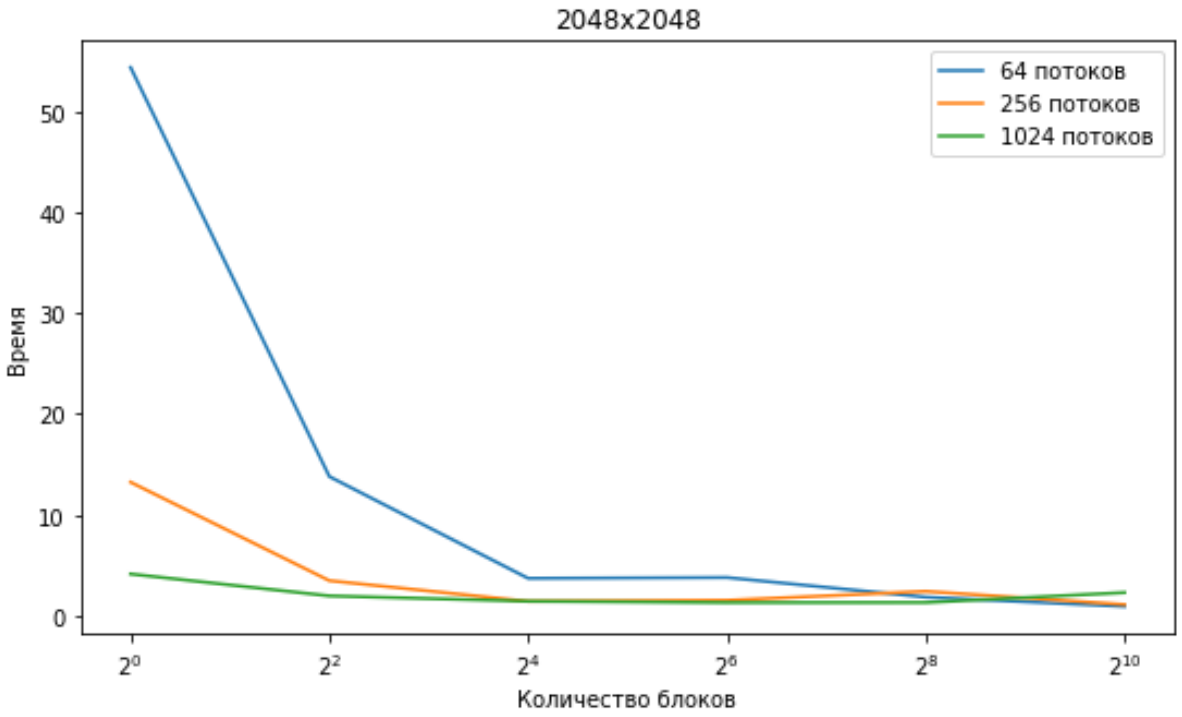
    - PlatonicSolid – абстрактный класс для платоновских тел, служит интерфейсом для их общих методов, наследуется от HittableObject. В его интерфейсе присутствуют виртулаьные функции n_triangles, n_edges и т.д., которые по-хорошему должны быть статик константами, но cuda их, к сожалению не поддерживает.
- ray_tracer.cuh – файл, содержащий наймспейс RayTracer, который содержит версии движка рейтрейсинга для цпу и гпу. Помимо этого в нем есть неймспейс common, который содержит их общие функции. Сами движки имеют одинаковый набор функций, с разной реализацией внутри.
    - void render – генерирует картинку с полученным миром и записывает ее в полученный буфер image.
    - float3 get_color — возвращает цвет заданного луча. В цпу версии работает рекурсивно, в гпу версии стек рекурсии реализован на массиве. Одна такая запись рекурсии хранит в себе текущую инструкцию и набор необходимых переменных
    - void setUpCam – позволяет установить конфигурацию камеры.
- Как уже отмечалось выше гпу версия внутри себя хранит два дополнительных буфера, один для вилеопамяти, второй для рекурсиию
- __global__ void render_kernel – единственное реализованное ядро, которое обрабатывает одним потоком один луч.

Исследовательская часть и результаты

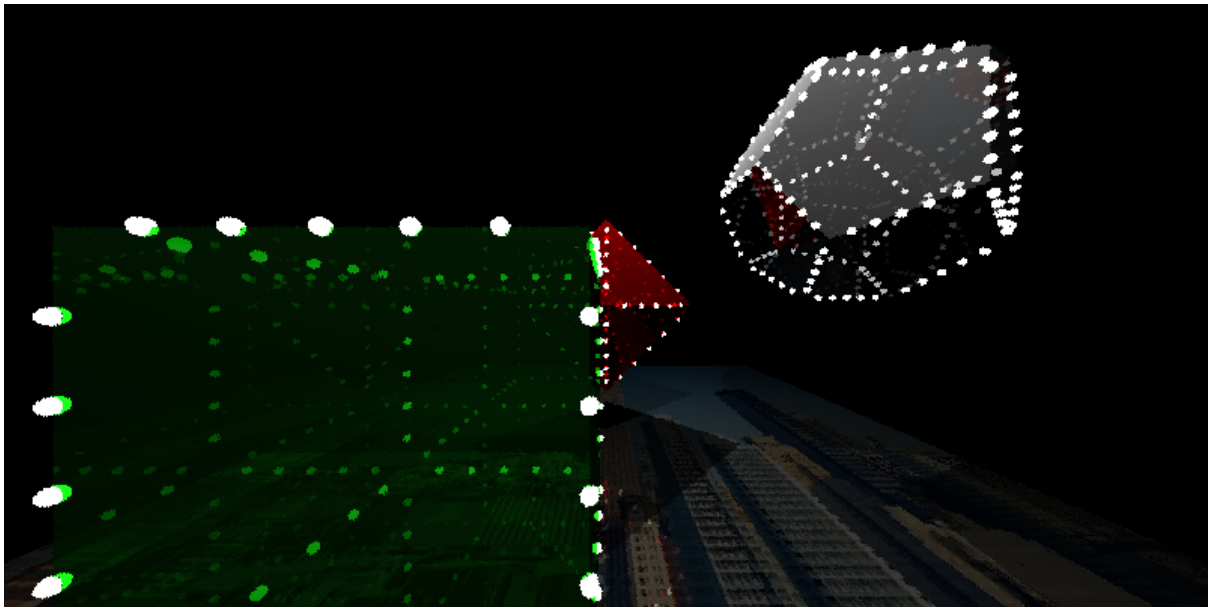
В результате исследования производительности цпу и гпу движков, выяснилось что cuda версия движка работает приблизительно в 130! раз быстрее cpu версии, да и вообще исследование цпу версии достаточно сложное занятие, так как на это уходит невероятно много времени. Если cuda генерирует один кадр за 1.3 секунду, то на процессоре на это уходит 182! секунды, то есть 3 минуты! Из очевидных результатов также можно отметить прямую пропорциональность времени генерации одного кадра количеству сабпикселей ssaa.



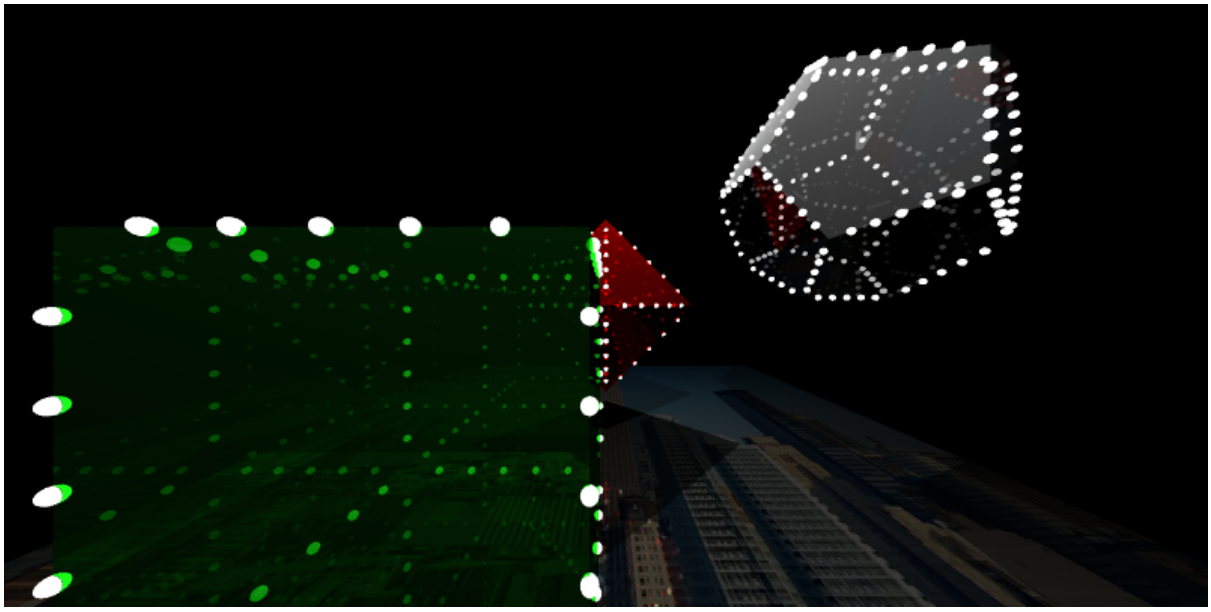

1024x1024
64 потоков
256 потоков
1024 потоков
Время
0
5
10
15
20
25
30
35
2^0
2^2
2^4
2^6
2^8
2^{10}
Количество блоков

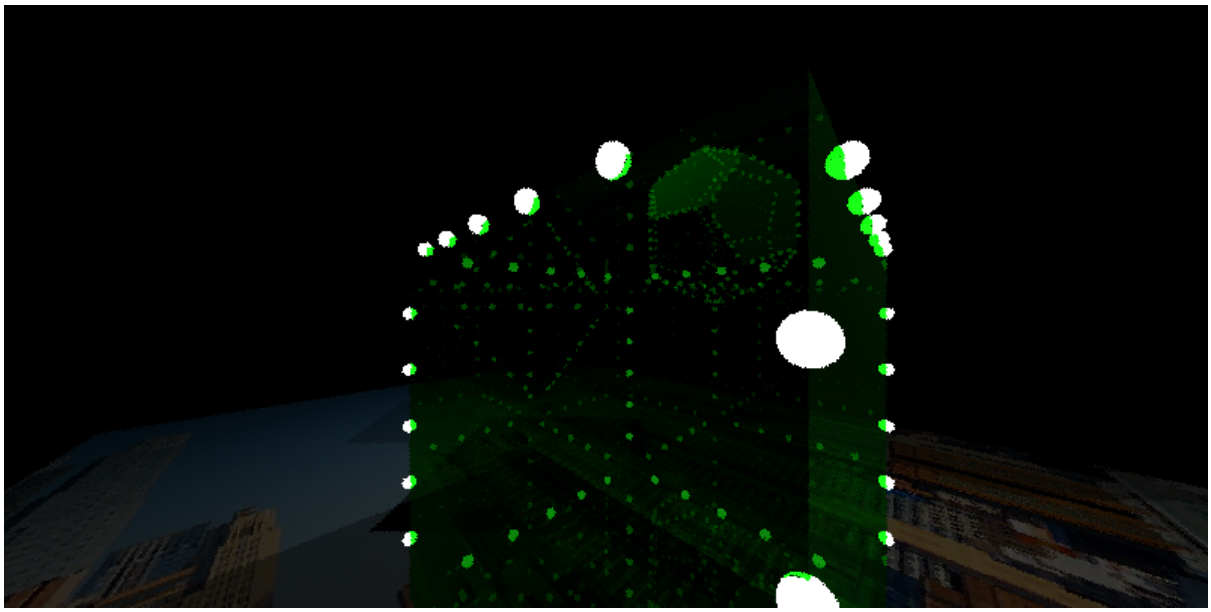

2048x2048
64 потоков
256 потоков
1024 потоков
Время
0
10
20
30
40
50
2^0
2^2
2^4
2^6
2^8
2^{10}
Количество блоков

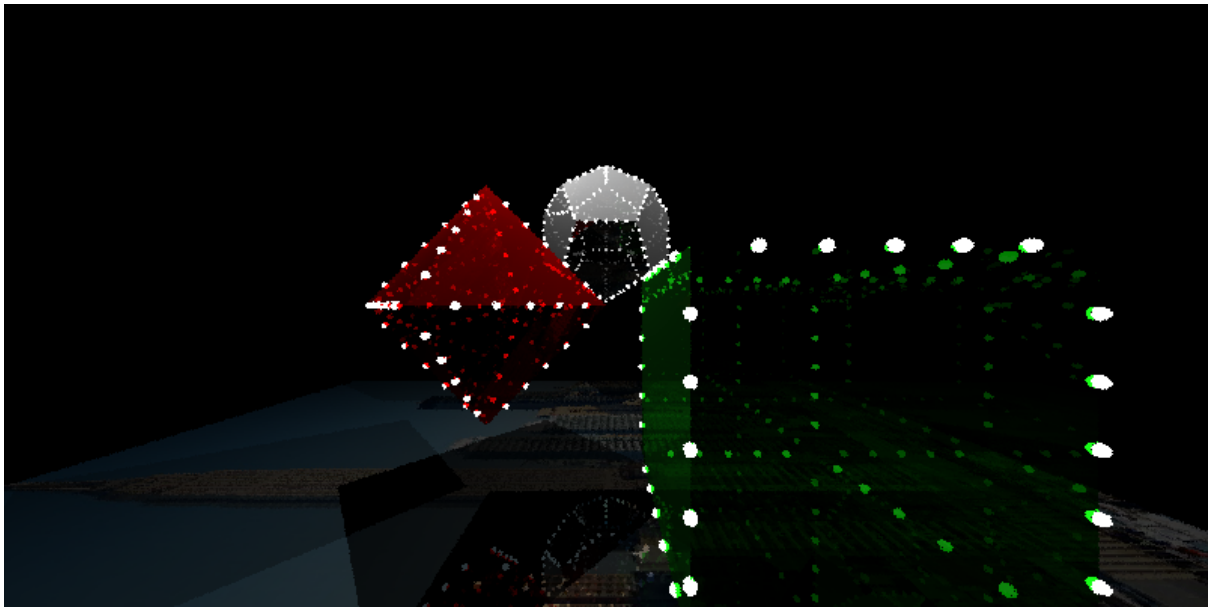
Изображение без сглаживания

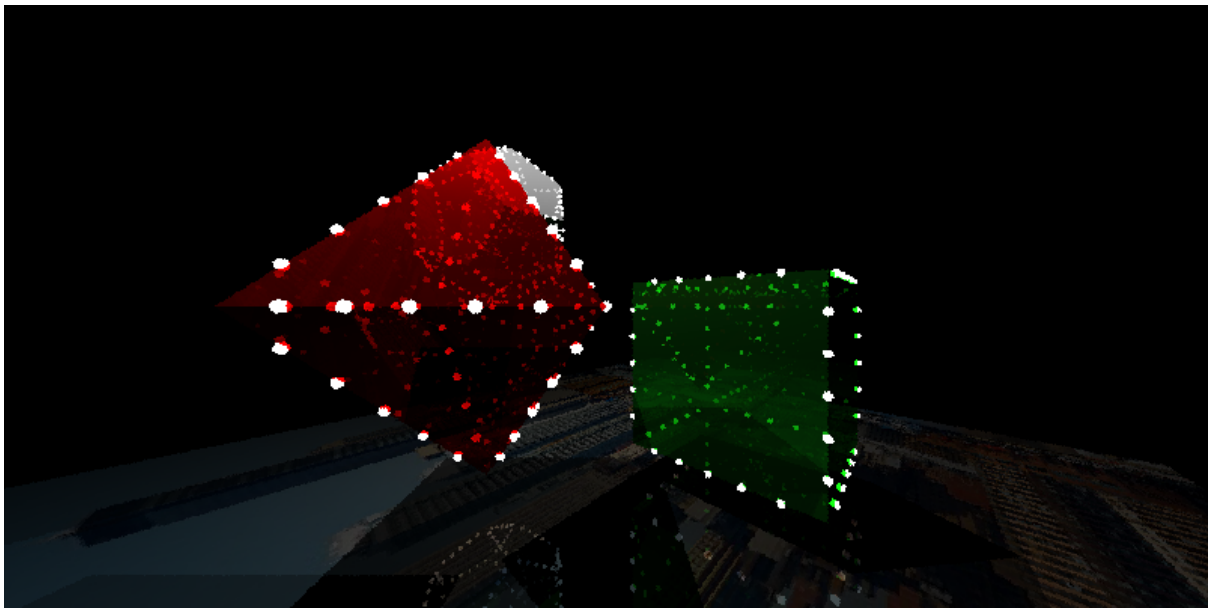


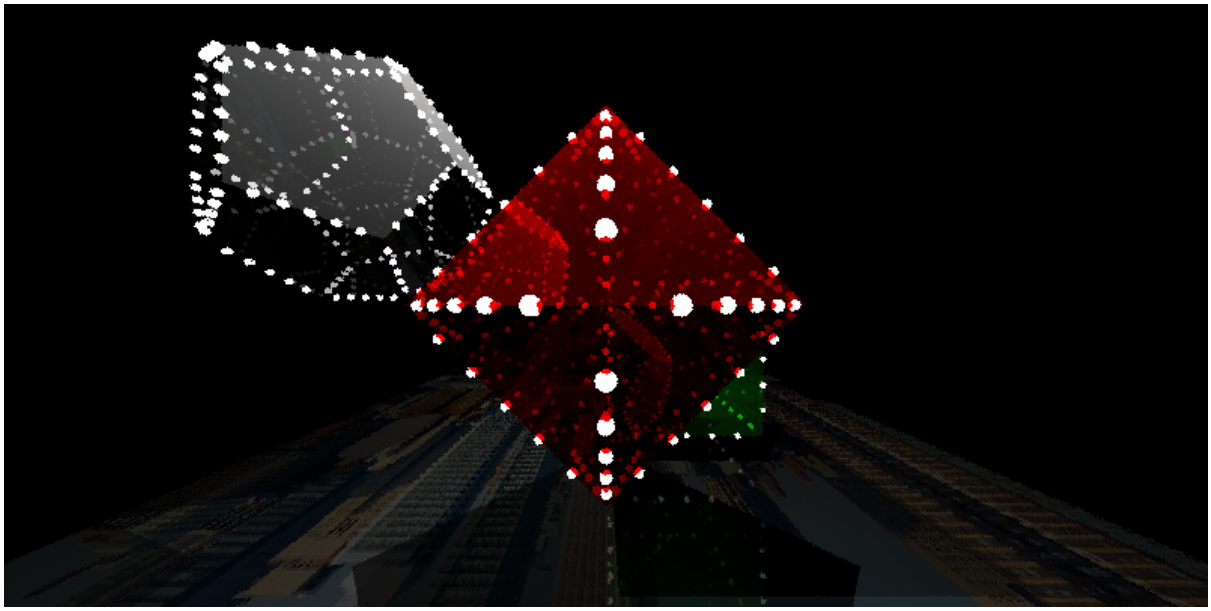
Сглаживание с 16ю сабпикселями

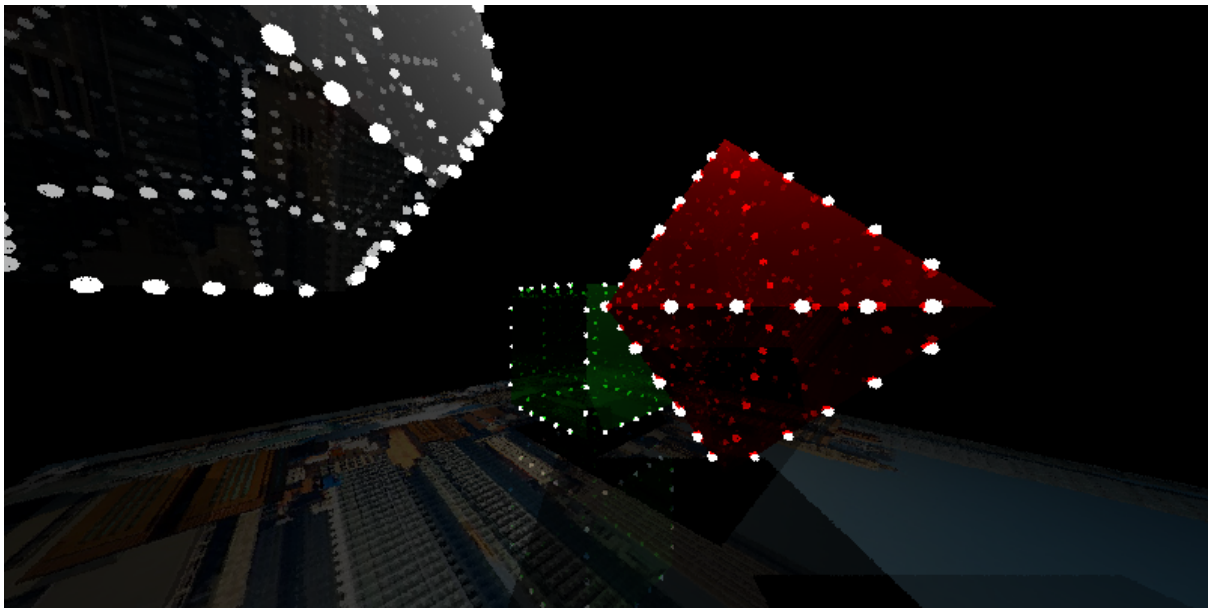


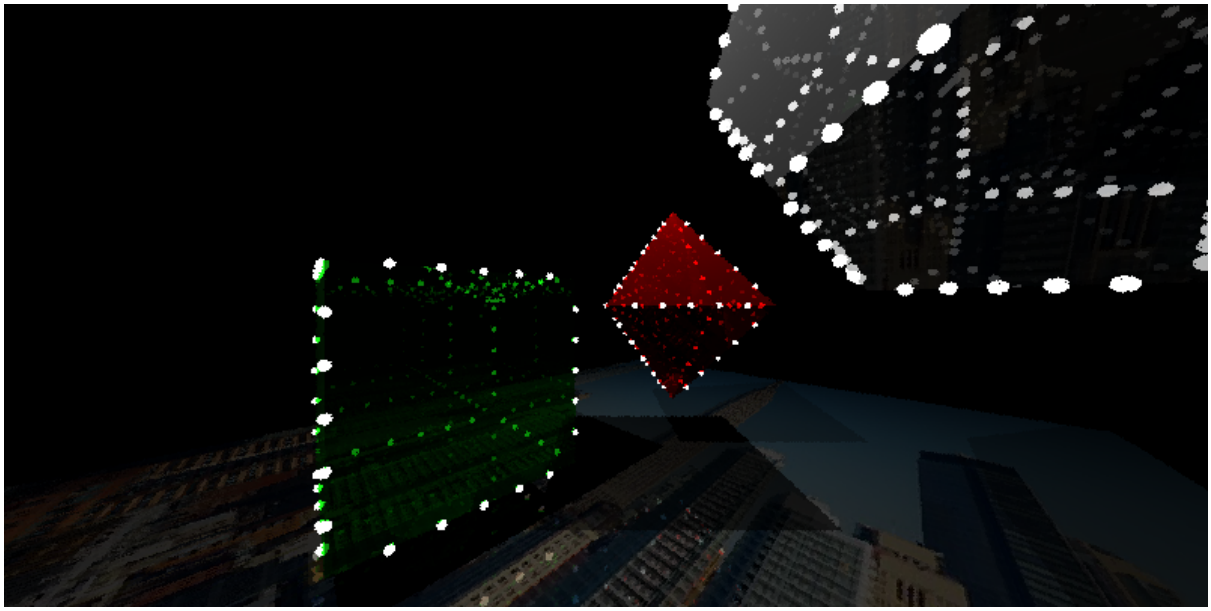












Выводы

Алгоритм обратной трассировки лучей при правильной реализации позволяет добиться таких результатов, которые сложно достичь другими техниками рендеринга. Единственным его недостатком и, причем существенным, является огромная прожорливость, он потребляет значительное число вычислительных мощностей. Именно поэтому применяется он при создании продуктов, где время не является критическим фактором, например при создании фильмов/мультфильмов. С другой стороны он легко поддается распараллеливанию, а отсечение невидимых поверхностей и наличие перспективы являются логическим следствиям алгоритма. Что касается сложности программирования, то мне показалось несложным написать непосредственно сам рейтресинг, однако подгон его под требования курсовой работы, попытки объединить интерфейсы гпу и цпу движков и в целом продумывание архитектуры отняло у меня очень много времени.

Литература

1. www.scratchpixel.com
2. raytracing.github.io/books/RayTracingInOneWeekend