МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра №806 «Вычислительная математика и программирование»

Курсовой работа
по курсу «Параллельная обработка данных»

Обратная трассировка лучей (Ray Tracing) на GPU

Выполнил: Н.И. Лохматов
Группа: 8О-406Б
Преподаватель: А.Ю. Морозов

Москва, 2024

Условие

1. Цель работы: использование GPU для создание фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание анимации
2. 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 счёт многократного переотражения лучей внутри тела, возникает эффект бесконечности. Камера. Камера выполняет облет сцены согласно определенным законам. В илиндрических координатах (r, φ, z), положение и точка направления камеры в момент времени t определено формулами. Требуется реализовать алгоритм обратной трассировки лучей с использованием технологии CUDA. Выполнить покадровый рендеринг сцены. Для устранения эффекта «зубчатости», выполнить сглаживание (например с помощью алгоритма SSAA). Полученный набор кадров склеить в анимацию 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).
3. Вариант 8: Гексаэдр, Октаэдр, Икосаэдр

Программное и аппаратное обеспечение

1. Графический процессор: Nvidia GeForce RTX 3050 Mobile
    a. Количество потоковых процессоров: 2560
    b. Частота ядра: 1552 МГц
    c. Количество транзисторов: 8.7 млрд
    d. Тех. процесс: 8 нм
    e. Энергопотребление: 80 Вт
2. ОС: Ubuntu 22.04
3. Текстовый редактор: VS Code
4. Компилятор: nvcc

Метод решения

Алгоритм построения фотореалистичной сцены с использованием обратной трассировки лучей начинается с расчёта взаимодействия лучей света с элементами сцены. Основной принцип состоит в том, что из виртуальной камеры через каждый пиксель изображения выпускается первичный луч, определяющий точку пересечения с объектами сцены. В данной работе сцена представлена треугольниками (точечные источники света учитываются отдельно). Эти первичные лучи анализируют видимость объектов с точки зрения камеры.

На этапе освещения проверяется, попадает ли пересеченная точка в область действия каждого из точечных источников света. Для этого из найденной точки к каждому

источнику выпускается теневой луч. Если на пути теневого луча есть объект, который перекрывает источник света, то точка считается находящейся в тени. Однако, в данном случае, освещение модифицируется за счет прозрачности материалов объектов, расположенных на пути теневого луча: коэффициенты прозрачности домножаются на итоговое освещение.
Далее, для расчета локального освещения применяется модель освещения Фонга. Итоговый световой вклад каждого видимого источника света аккумулируется.
Если материал поверхности обладает отражающими свойствами, из точки пересечения выпускается отраженный луч. Аналогично, если материал обладает свойствами преломления, запускается преломленный луч. Для обоих лучей трассировка выполняется рекурсивно. Их вклад в итоговое освещение учитывается с учётом соответствующих коэффициентов отражения и преломления. Этот процесс повторяется до заданной глубины рекурсии или до тех пор, пока интенсивность луча не станет незначительной.
Для получения эффекта многократных отражений и преломлений внутри тел, следует учитывать все внутренние переотражения, пока луч не выйдет за пределы тела или не затухнет. Такой подход создаёт эффект бесконечности, когда световые лучи продолжают взаимодействовать с зеркальными и прозрачными поверхностями.
Дополнительно, для устранения «зубчатости» изображения используется алгоритм сглаживания SSAA, при котором каждый пиксель изображения разбивается на несколько субпикселей, и результат усредняется.
После выполнения всех расчетов сцена визуализируется покадрово. Итоговые кадры объединяются в анимацию с использованием стороннего программного обеспечения. Также проводится сравнительный анализ производительности рендеринга на GPU с CUDA и CPU.

Описание программы

Опишем архитектуру проекта:

1. Директория converter. Тут находятся два скрипта: для конвертации изображений и для создания gif-анимации
2. Директория data. Здесь будут созданы изображения в формате .data
3. Директория images. Сюда сохраняются изображения после конвертации
4. Директория libs. Тут находятся 4 вспомогательных файла: helpers.cuh – полезные функции и макросы, ray_tracing.cuh – основной функционал алгоритма рендеринга и функционал для очистки «мертвых» лучей после каждого вызова рекурсии, ssaa.cuh – алгоритм SSAA для GPU и CPU, structures.cuh – файл с полезными структурами
5. Директория textures. Здесь находится текстура пола в формате .data
6. Файл с лучшей конфигурацией – best_config.txt
7. kp.cu – основной файл с созданием сцены, генерацией фигур и настройкой освещения
8. run.sh – скрипт для быстрого запуска

По сути, можно сказать, что основная работа выполняется в файлах kp.cu и ray_tracing.cuh. Рассмотрим ray_tracing.cuh:

Функция initRays вызывается перед началом рендеринга и отвечает за определение начальных параметров каждого луча, который проходит от наблюдателя через пиксели экрана. Она устанавливает начальные цвета изображения в черный и рассчитывает направление лучей. Позиция каждого пикселя определяется в локальной системе координат, где наблюдатель является точкой отсчета. Затем эти координаты преобразуются с использованием матрицы трансформации, зависящей от положения камеры в пространстве.
Основной метод трассировки лучей — rayTrace — реализует пошаговый расчет для каждой итерации рекурсии:

1. Каждый луч, ассоциированный с конкретным пикселем, проверяется на пересечение с ближайшим объектом сцены с помощью функции intersectionSearch. Если пересечения нет, луч игнорируется.
2. При обнаружении пересечения вычисляются нормаль поверхности и свойства материала:
    - Для нормали используется метод normal, который рассчитывает векторное произведение сторон треугольника. Направление нормали корректируется, чтобы она была направлена к наблюдателю.
    - Для материала используются свойства треугольника, дополненные цветом текстуры в точке пересечения, если объект является текстурированным.
3. Луч переносится в точку пересечения, после чего в цикле обрабатываются теневые лучи ко всем источникам света. Результирующее освещение складывается из вклада каждого источника.

Для каждого источника света: 4. Если источник света находится на обратной стороне поверхности треугольника, он игнорируется (можно доработать с учетом прозрачности). 5. Функция radioCompute рассчитывает общий коэффициент поглощения света на пути теневого луча, что влияет на итоговую интенсивность освещения. 6. Модель освещения Фонга применяется для расчета вкладов от точечных источников света. Коэффициент бликов в данном случае связывается с прозрачностью материала.
После цикла: 7. Итоговая сумма освещений записывается в глобальную память. Из-за возможного конфликта данных при глубокой рекурсии используется atomicAdd для безопасной записи в память. 8. Лучи делятся на преломленные и отраженные, их интенсивность корректируется в зависимости от коэффициентов прозрачности и отражения. Сила лучей также используется для фильтрации слабых ("мертвых") лучей. 9. Завершается текущая итерация рекурсии.
Метод triangleIntersection применяет барицентрический тест для проверки пересечения луча с треугольником. В случае успеха возвращается расстояние до пересечения, иначе — отрицательное значение. Метод intersectionSearch использует triangleIntersection для поиска ближайшего треугольника и записывает результат (идентификатор треугольника и расстояние) в специальный контракт.

Функция radioCompute схожа с intersectionSearch, но вместо ближайшего треугольника она рассчитывает общий показатель поглощения света на пути теневого луча, проверяя все треугольники, расстояние до которых меньше, чем до источника света.
Для получения цвета текстуры в точке пересечения используется функция intersectionTex. На GPU барицентрические координаты пересечения применяются в функции tex2D, которая аппаратно интерполирует значения цвета на основе нормализованных координат.
Функции для очистки «мертвых» лучей:

- gpuCleanRays: запускает обработку на GPU. Функция binCompute определяет, какие лучи слабее заданного порога. Затем их индексы упорядочиваются с использованием функции scan, после чего binsSort перемещает активные лучи в начало массива. Итоговый счетчик возвращает количество оставшихся активных лучей.
- cpuCleanRays: работает на CPU. Лучи сортируются по мощности, и первые слабые (меньше порога) исключаются. Функция возвращает количество активных лучей.

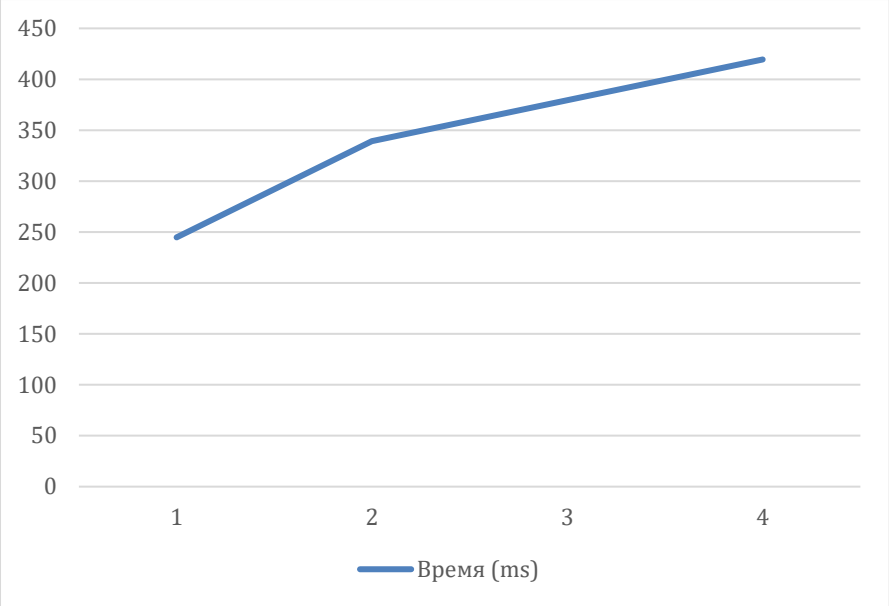
Исследовательская часть

Наиболее красочный результат получился для данных:

```
1024
data/%d.data
720 480 120
4.0 3.0 0.0 2.0 1.0 2.0 6.0 1.0 0.0 0.0
1.0 0.0 0.0 0.5 0.1 1.0 4.0 1.0 0.0 0.0
0.0 -2.5 1.0 0.1 0.23 1.0 0.85 0.9 0.5 10
0.0 2.0 1.0 0.98 0.1 0.55 0.85 0.9 0.5 10
0.0 0.0 1.0 0.61 0.15 1.0 0.85 0.9 0.5 10
-5.0 -5.0 -1.0 -5.0 5.0 -1.0 5.0 5.0 -1.0 5.0 -5.0 -1.0 textures/floor.data 0.5 0.5 0.5 0.7
3
-10.0 0.0 5.0 1.0 0.0 0.0
-5.0 5.0 3.0 1.0 0.0 1.0
1.0 0.0 5.0 0.0 0.0 1.0
6 2
```

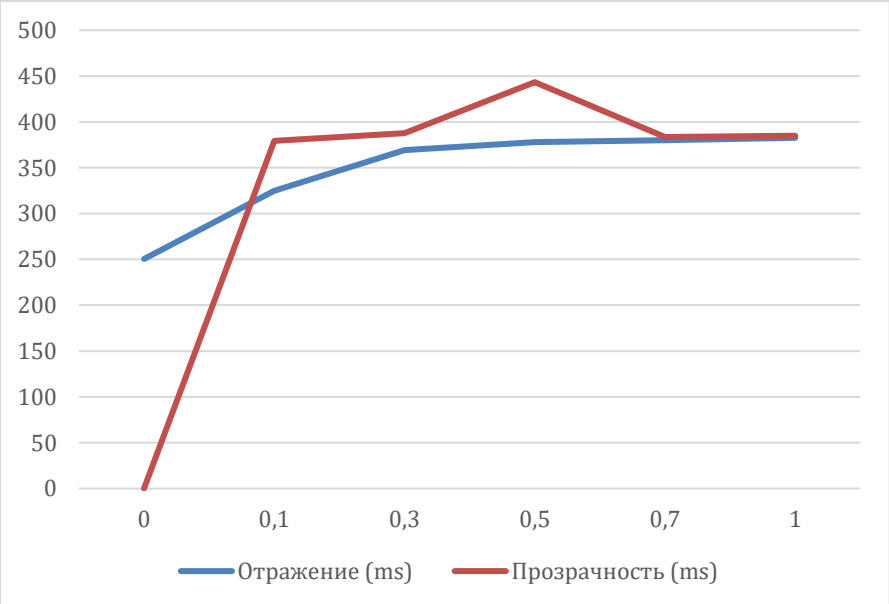
На этих данных я буду проводить все тесты, если не дано других пояснений. Стандартный размер сетки для трассировки – 64×64, для SSAA – 32×32×32×32.

График, показывающий среднее время на генерацию одного кадра с различным количеством источников освещения:



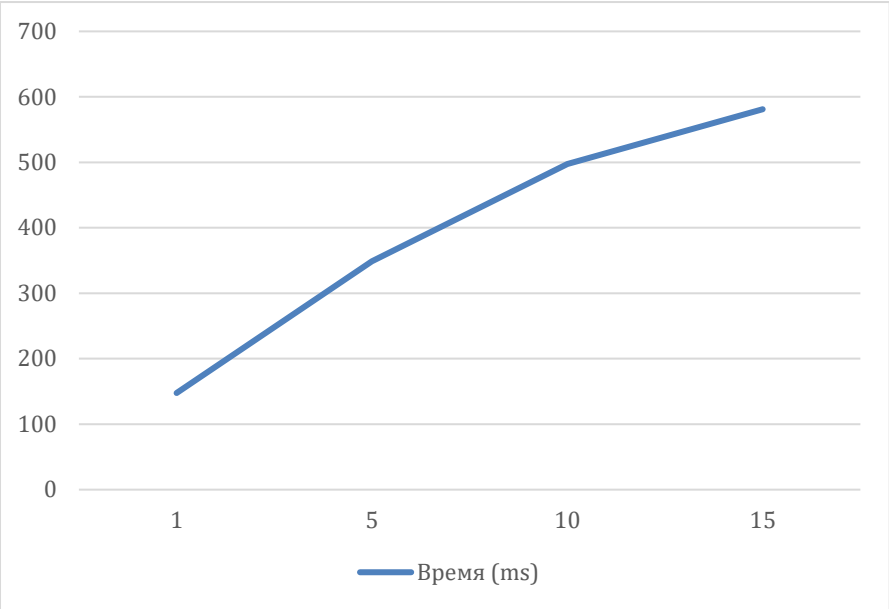
По графику видно, что зависимость времени от количества источников света практически линейная (при прочих неизменных параметрах).

Графики, показывающие зависимость времени от коэффициентов отражения и прозрачности:



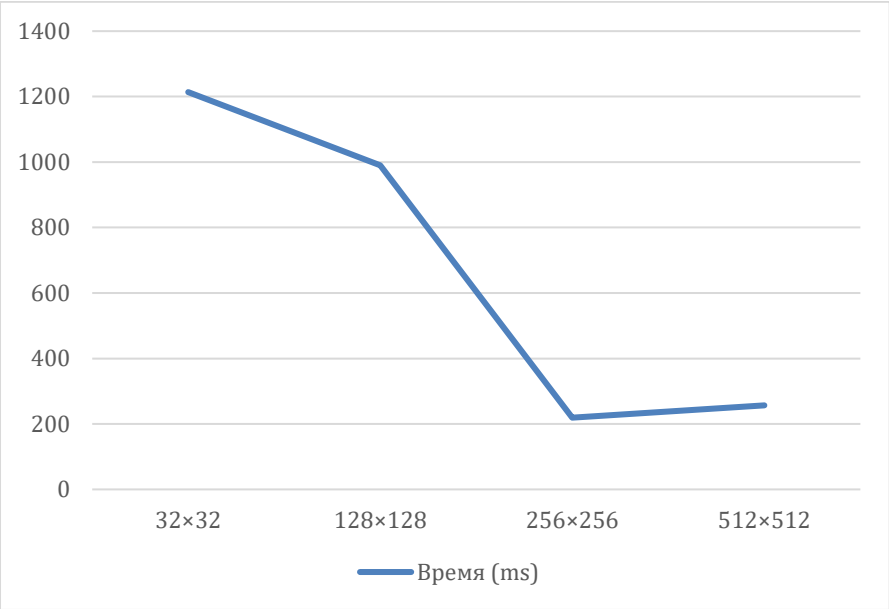
По графику видно, что при увеличении коэффициентов скорость растёт, однако в конце становится практически неизменной.

График, показывающий зависимость времени от максимальной глубины рекурсии:



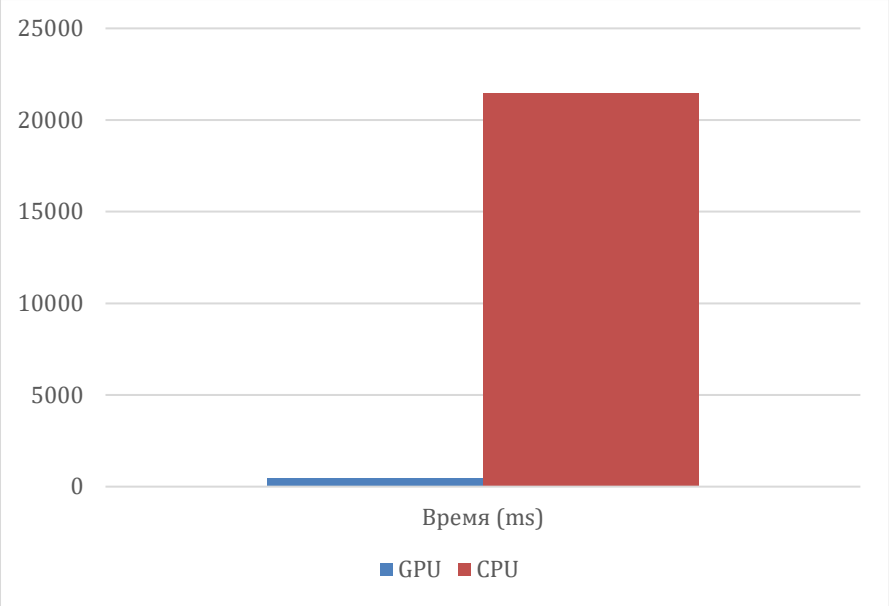
Вполне очевидный результат.

Графики, показывающие зависимость времени от количества потоков:

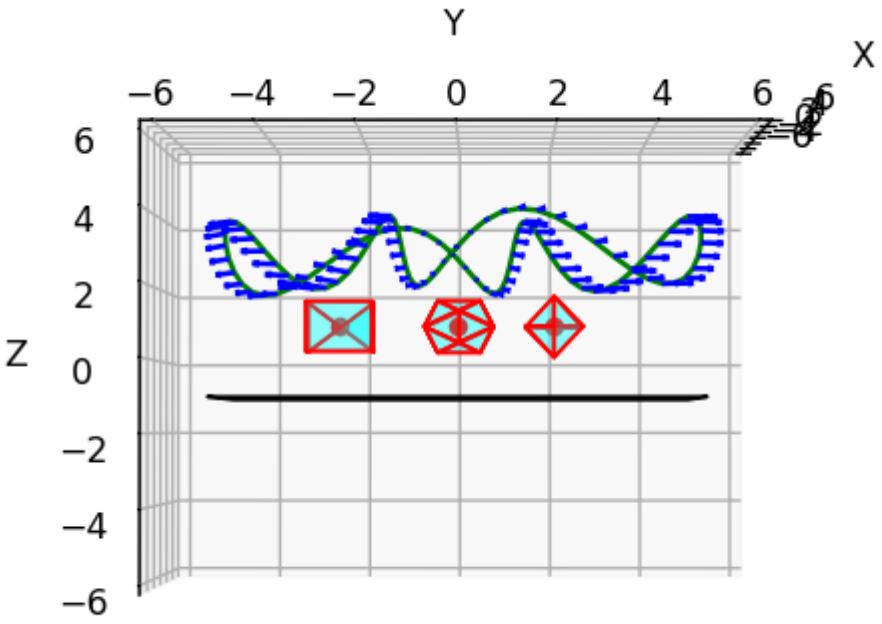


Можно сделать вывод о том, что на моей видеокарте оптимальным количеством потоков является 256×256.

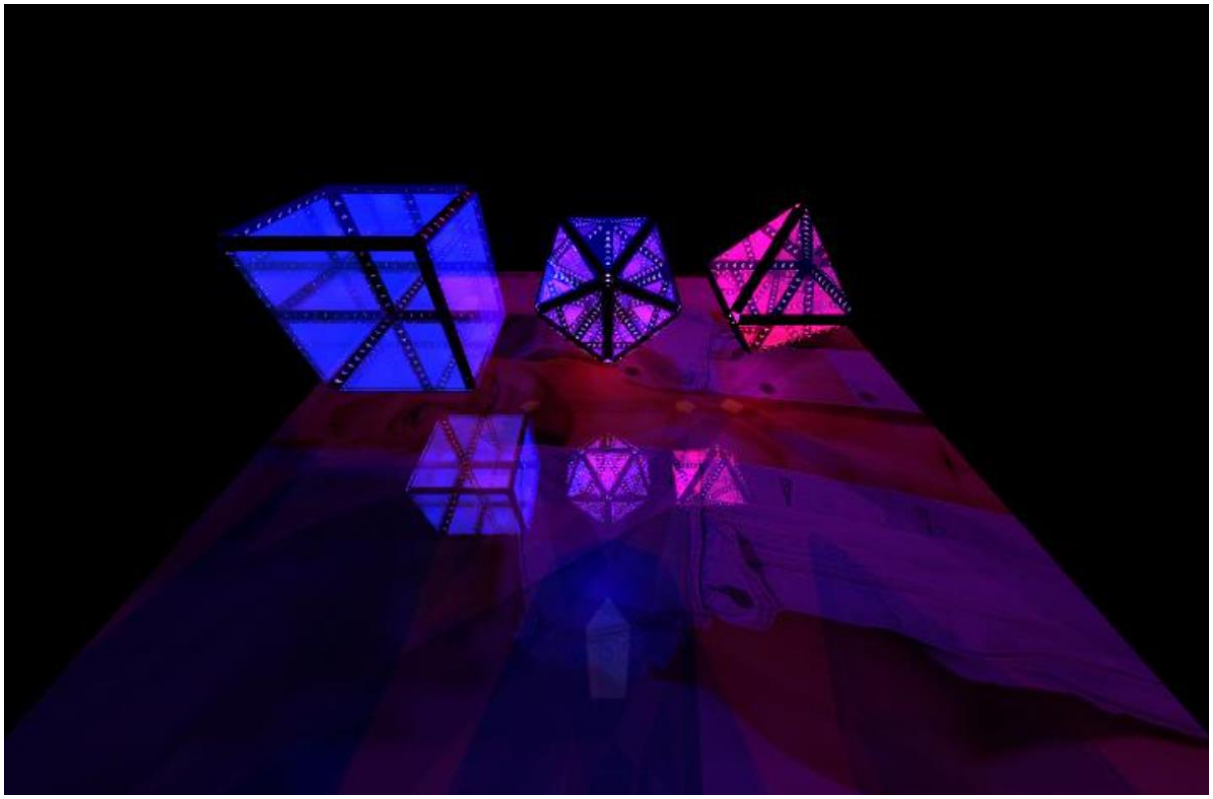
Сравним время генерации одного кадра на GPU (64×64) с CPU:

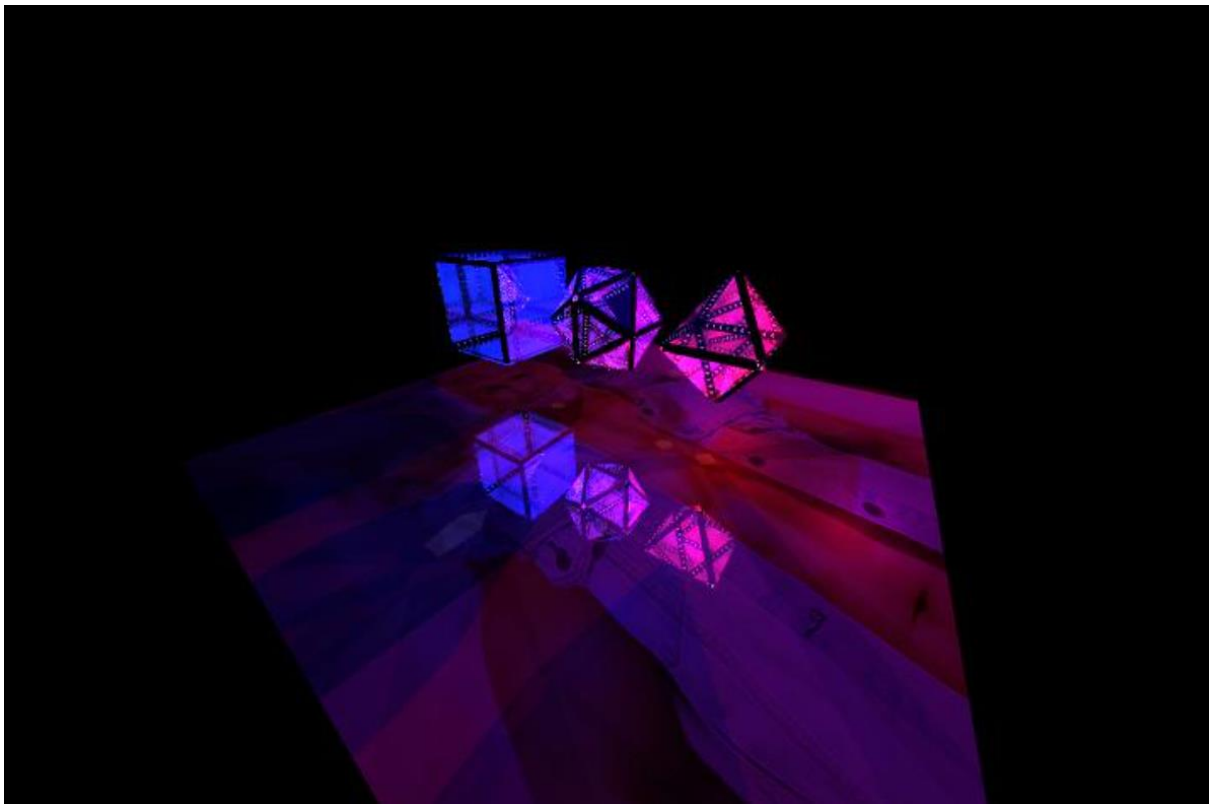


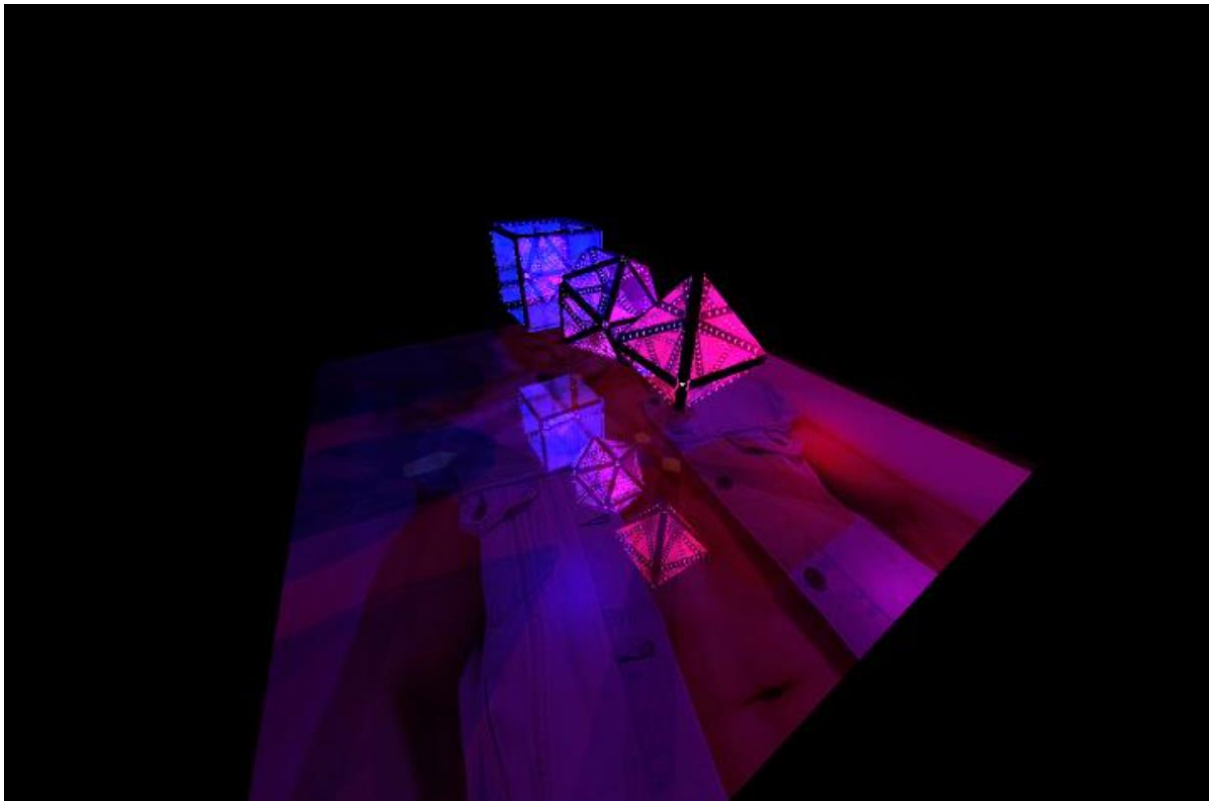
По графику можно сделать вывод о том, что для рей трейсинга GPUподходит гораздо лучше.

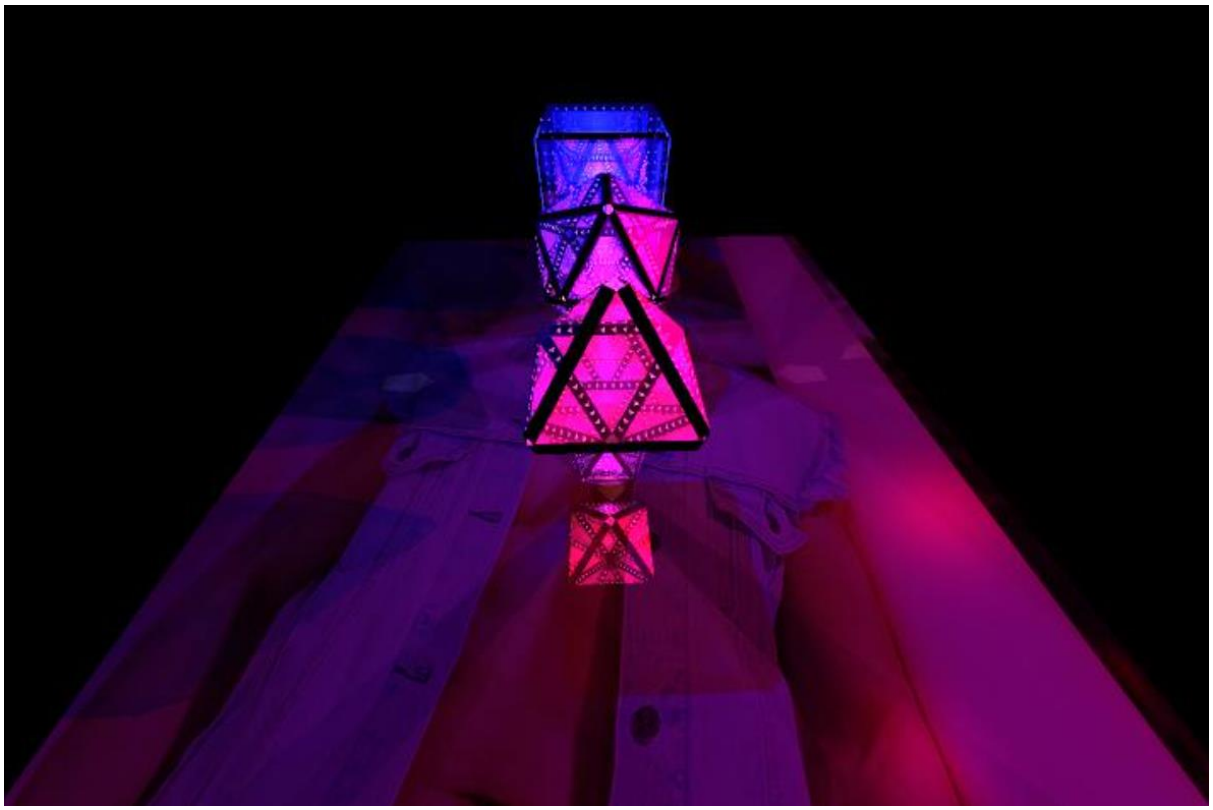


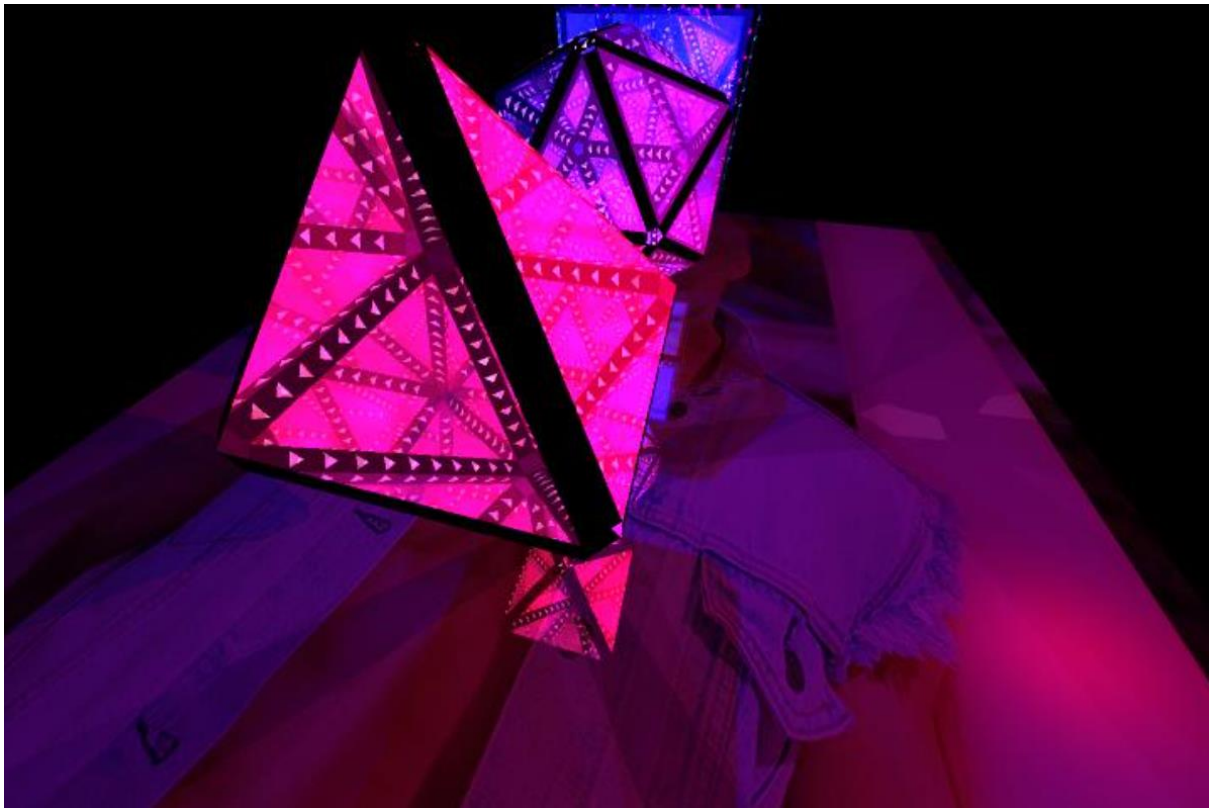
Результаты

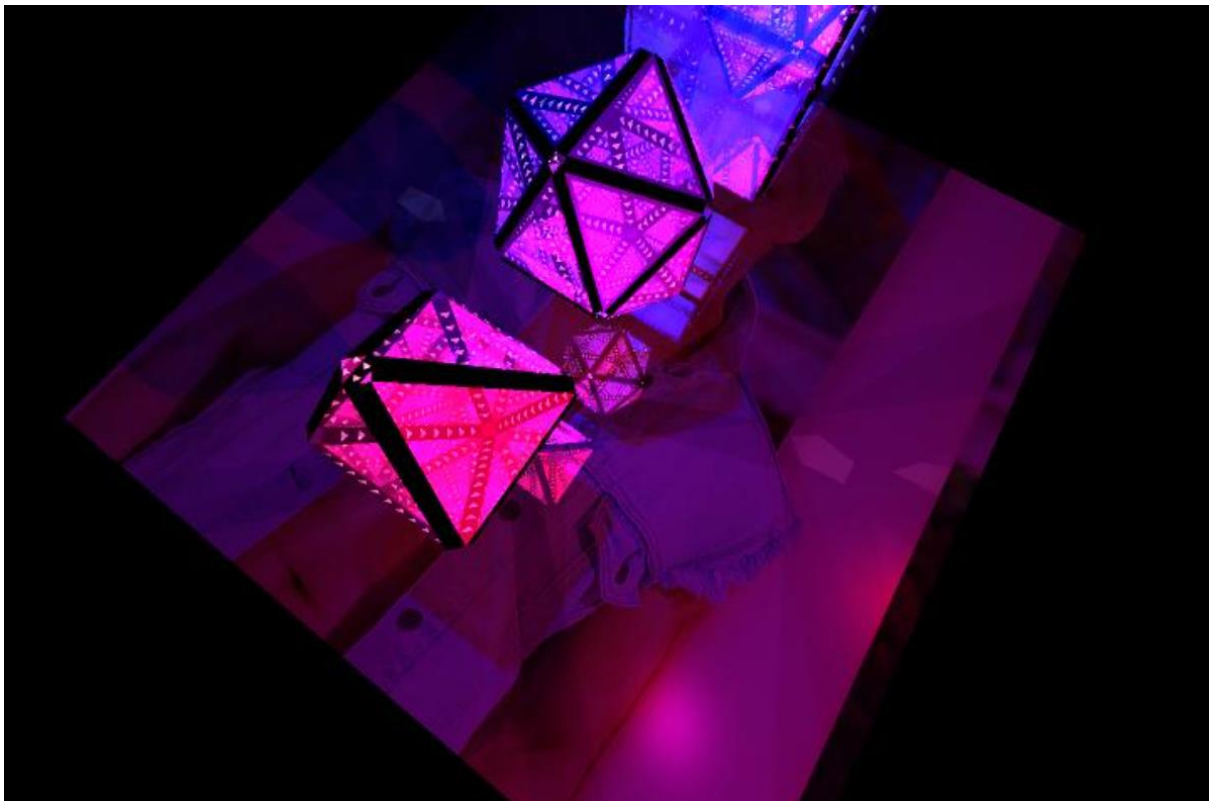


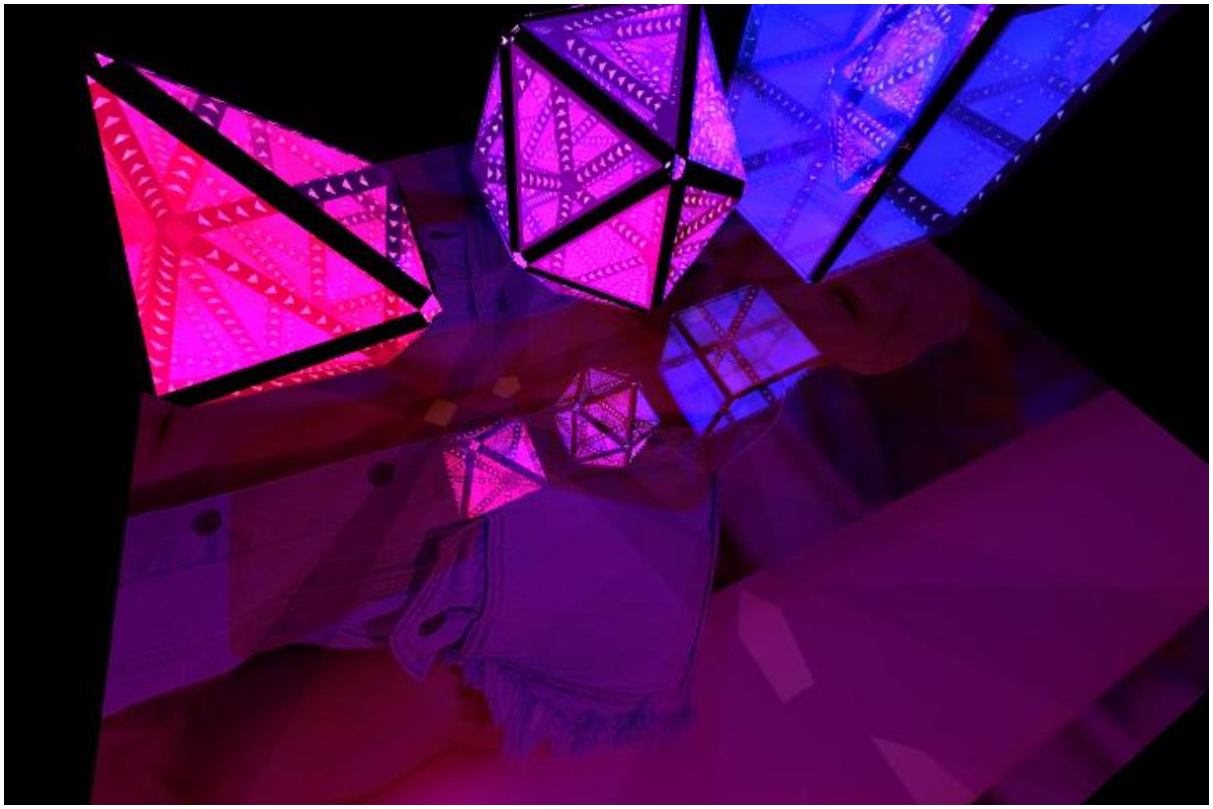


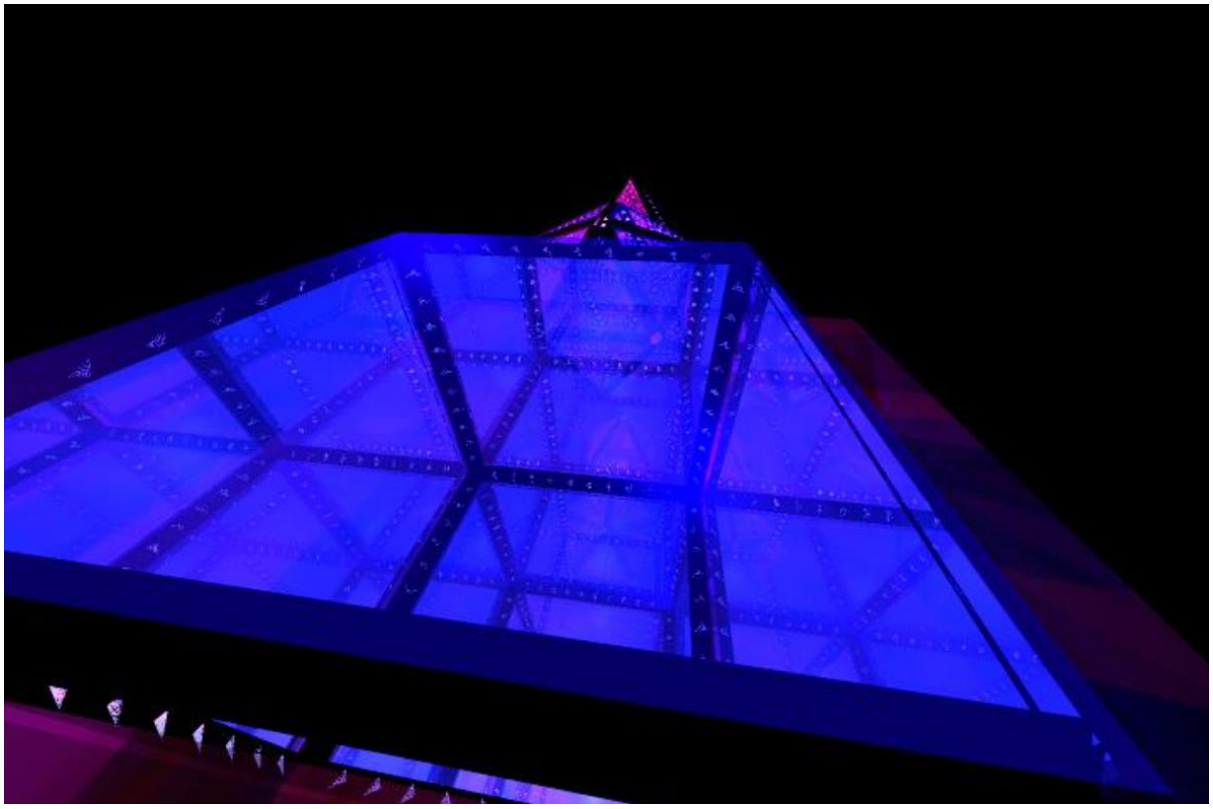


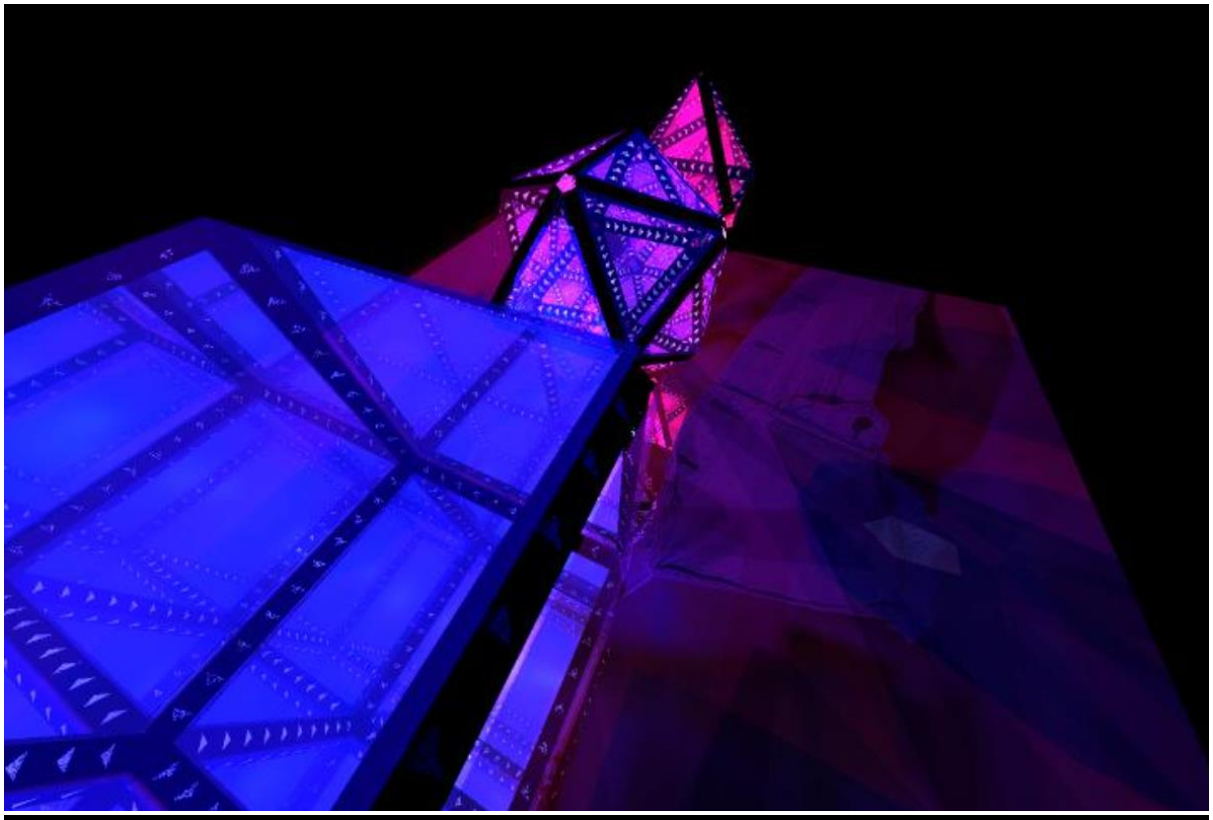


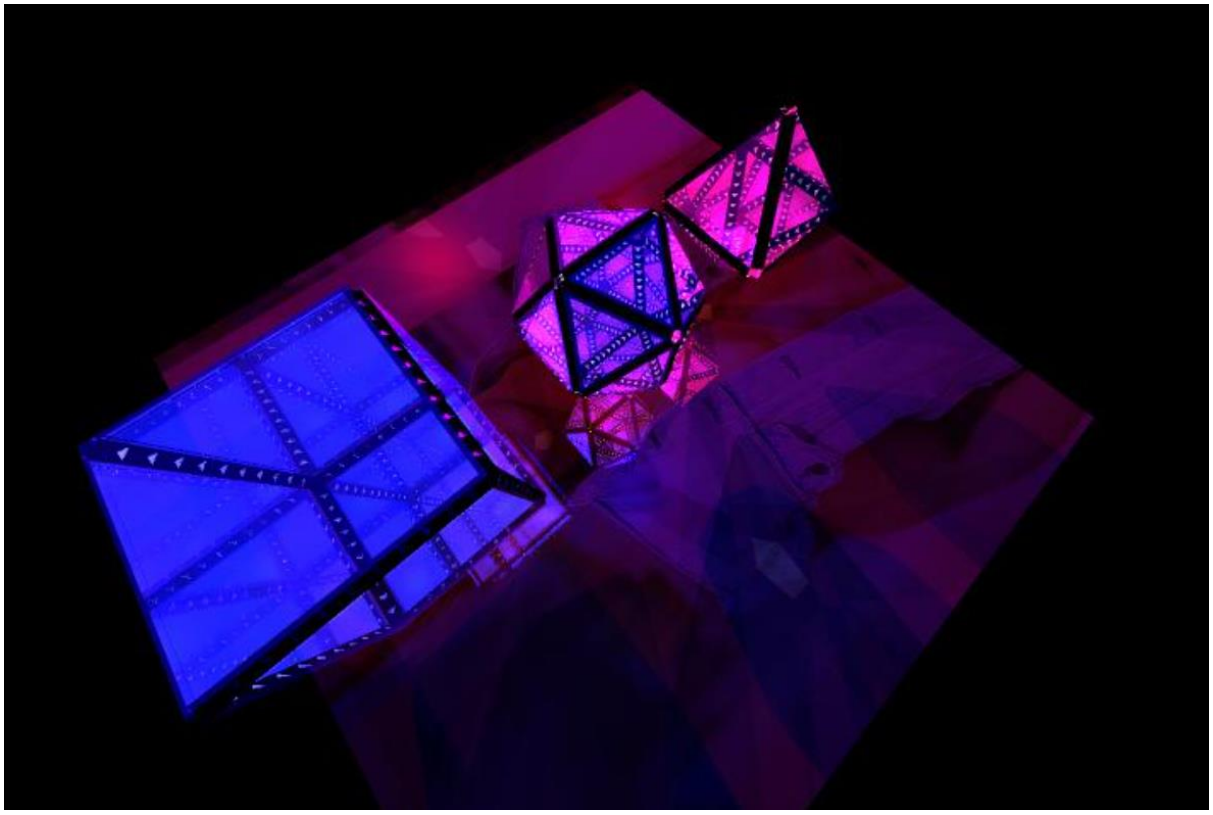












Выводы

Выполнив курсовую работу по теме «Параллельная обработка данных», удалось глубже изучить алгоритм Ray Tracing и особенности его реализации. Этот метод, несмотря на свою ресурсоёмкость, обладает огромным потенциалом для применения в игровой

индустрии и кинопроизводстве. С ростом производительности современных видеокарт, особенно от Nvidia, возможности Ray Tracing становятся все более востребованными. Алгоритм, из-за своей природы неравномерного доступа к памяти, не всегда идеально использует ресурсы GPU, но его реализация на центральных процессорах практически нецелесообразна, а выполнение в реальном времени — невозможно. Именно в этом проекте раскрылись преимущества использования параллельных вычислений на графических процессорах для обработки сложных задач.

Литература

1. http://ray-tracing.ru/articles213.html
2. http://www.ray-tracing.ru/articles164.html
3. http://math.hws.edu/graphicsbook/c8/s1.html
4. https://www.thanassis.space/cudarenderer-BVH.html#recursion