МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Курсовая работа
по курсу «*Программирование графических процессоров*»

Обратная трассировка лучей (Ray Tracing) на GPU

Выполнил: *М.С. Корнев*
Группа: *М8О-412Б-22*
Преподаватель: А.Ю. Морозов

Москва, *2025*

Условие

Описание задачи:

1. **Цель работы.** Использование GPU для создание фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание анимации.
2. **Вариант.** 4. Тетраэдр, Октаэдр, Додекаэдр

Программное и аппаратное обеспечение

Аппаратное обеспечение

Графический процессор (GPU):

- Модель: NVIDIA Tesla T4
- Архитектура: Turing
- Compute Capability: **7.5**
- Графическая память: **16 ГБ GDDR6**
- Пропускная способность памяти: ~320 ГБ/с
- Количество мультипроцессоров (SM): **40**
- Максимальное число потоков на один SM: 1024
- Максимальное число потоков в блоке: 1024
- Максимальное число регистров на блок: 65 536 (по 32 бита)
- Разделяемая память на блок: до **64 КБ**
- Константная память: **64 КБ**

Процессор (CPU):

- Виртуальная машина Colab: 1 × Intel Xeon (Google Cloud)
- Частота: ~2.0–2.2 ГГц
- Количество ядер: обычно 2 доступных потока

Оперативная память (RAM):

- В Google Colab стандартно: **12–13 ГБ** (доступные пользователю)

Жёсткий диск (HDD/SSD):

- Виртуальный диск Colab: ~70–80 ГБ (SSD Google Cloud)

Программное обеспечение

- Операционная система: **Linux (Ubuntu 20.04 LTS, Google Colab среда)**
- Драйвер CUDA: **550.54.15**
- CUDA Toolkit: **12.5** (с nvcc 12.5.82, поддержка до Compute Capability 9.0)
- Язык программирования: **C++ (CUDA C)**
- Компилятор: nvcc (NVIDIA CUDA Compiler Driver)
- IDE / среда разработки: **Google Colaboratory (Jupyter Notebook web-IDE)**
- Дополнительно: доступ к стандартным библиотекам C (cstdio, cstdlib, cmath), а также к CUDA Runtime API

Метод решения

Для визуализации сцены используется классический подход обратной трассировки лучей. Камера рассматривается как источник лучей: для каждого пикселя экрана рассчитывается направление взгляда в мировом пространстве, после чего из позиции камеры в этом направлении выпускается луч. Задача сводится к поиску ближайшего пересечения этого луча с объектами сцены, которые аппроксимируются набором треугольников. В качестве базового примитива выбрана треугольная грань, а сложные тела - тетраэдр, октаэдр и додекаэдр - задаются как фиксированные наборы вершин и треугольников, вычисленных один раз на стороне процессора.

Пересечение луча с треугольником реализовано по широко используемой схеме, аналогичной алгоритму Мёллера-Трумборе. Суть метода в том, что луч представляется полупрямой, а треугольник - параметрической поверхностью внутри своей плоскости. Решается система уравнений на параметры, описывающие положение точки на луче и внутри треугольника. Если найденное значение параметра вдоль луча положительно, а барицентрические координаты лежат внутри треугольника, пересечение считается корректным. Из всех треугольников выбирается тот, для которого точка пересечения находится ближе всего к камере.

После нахождения ближайшей точки пересечения рассчитывается освещение. В рамках варианта на три балла используется упрощённая модель без отражений, прозрачности и рекурсивных эффектов. Цвет точки определяется цветом грани и цветом источника света без учёта сложных компонентов, таких как затухающая интенсивность или блеск. Реализованы тени от объектов: от точки пересечения в сторону источника света запускается дополнительный луч. Если по пути к источнику он пересекает любую другую грань, точка считается затенённой и её цвет затемняется. Таким образом реализуется простое затенение с мягким «приглушённым» цветом в тени.

Для уменьшения «лесенок» по контуру объектов применяется суперсэмплинг. Вместо одного луча на пиксель генерируется несколько лучей, слегка смещённых внутри площади пикселя. Цвет итогового пикселя вычисляется как среднее значение по всем лучам. Это даёт более плавные границы объектов, особенно на диагональных и тонких деталях. Движение камеры организовано по заданным законам в цилиндрических координатах: радиус, высота и угловая координата зависят от времени по гармоническим законам, что создаёт плавный облет сцены.

При подготовке решения опирался на материалы сайта ray-tracing.ru.

Описание программы

Программа реализована на языке C++ с использованием CUDA и оформлена в одном файле с расширением .cu. В основе кода лежит несколько простых структур данных: трёхмерный вектор с компонентами по осям, треугольник с тремя вершинами и цветом, описание сцены с массивом треугольников и параметрами источника света, а также структура камеры с позицией, точкой, в которую она смотрит, и углом обзора.

Логика расчёта разбита на несколько уровней. На самом низком уровне находятся вспомогательные векторные операции: скалярное и векторное произведение,

нормализация и преобразование координат из базиса камеры в мировое пространство. Чуть выше располагаются функции пересечения луча с треугольником и поиска ближайшего пересечения по всему массиву граней. Ещё один слой отвечает за проверку наличия тени, то есть за проверку того, закрывает ли какой-либо объект путь от точки на поверхности к источнику света. Над этим построена функция, которая для одного луча возвращает конечный цвет, всё вместе фактически реализует один шаг трассировки без рекурсивных отражений.

Отдельно выделена функция, которая рендерит один пиксель с учётом суперсэмплинга. В ней формируется базис камеры, вычисляется направление нескольких лучей внутри пикселя, каждый луч передаётся в функцию затенения, а затем результаты усредняются. Эта же функция используется и на CPU, и в ядре CUDA, что позволяет избежать дублирования логики: она помечена как выполняемая и на хосте, и на устройстве.

Построение геометрии сцены вынесено в отдельные функции: пол, тетраэдр, октаэдр и додекаэдр заполняют один общий массив треугольников. Это решение выбрано вместо более сложных иерархических структур, так как в задании достаточно фиксированного небольшого числа объектов. Работа с простым линейным массивом хорошо ложится на модель CUDA и даёт предсказуемый доступ к памяти. Перед рендерингом на GPU этот массив один раз копируется на устройство, а сцена передаётся в ядро через структуру с указателем на массив и числом треугольников. На каждый кадр вычисляется положение и направление камеры по времени, строится структура камеры, после чего запускается либо CPU-рендер, либо CUDA-ядро, в зависимости от режима работы. Полученный кадр записывается в бинарный файл в формате.

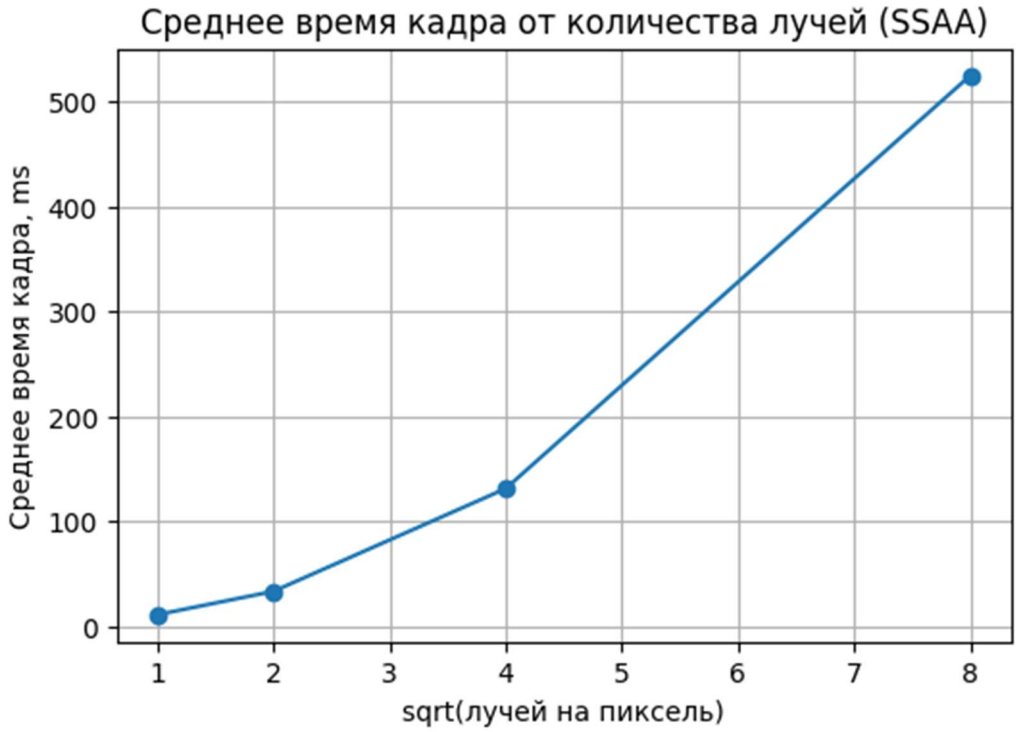
Исследовательская часть и результаты

Таблица результатов среднего времени кадра в зависимости от конфигураций

№	Конфигурация ядра	Среднее время кадра
1	block(8, 8), grid(16, 16)	14.228544
2	block(16, 16), grid(4, 4)	11.912320
3	**block(16, 16), grid(8, 8)**	**9.006912**
4	block(32, 32), grid(4, 4)	12.834624
5	block(32, 8), grid(8, 8)	10.672256
6	block(32, 16), grid(8, 8)	10.914368
7	block(64, 64), grid(32, 32)	17.442816
8	**CPU**	143.226624

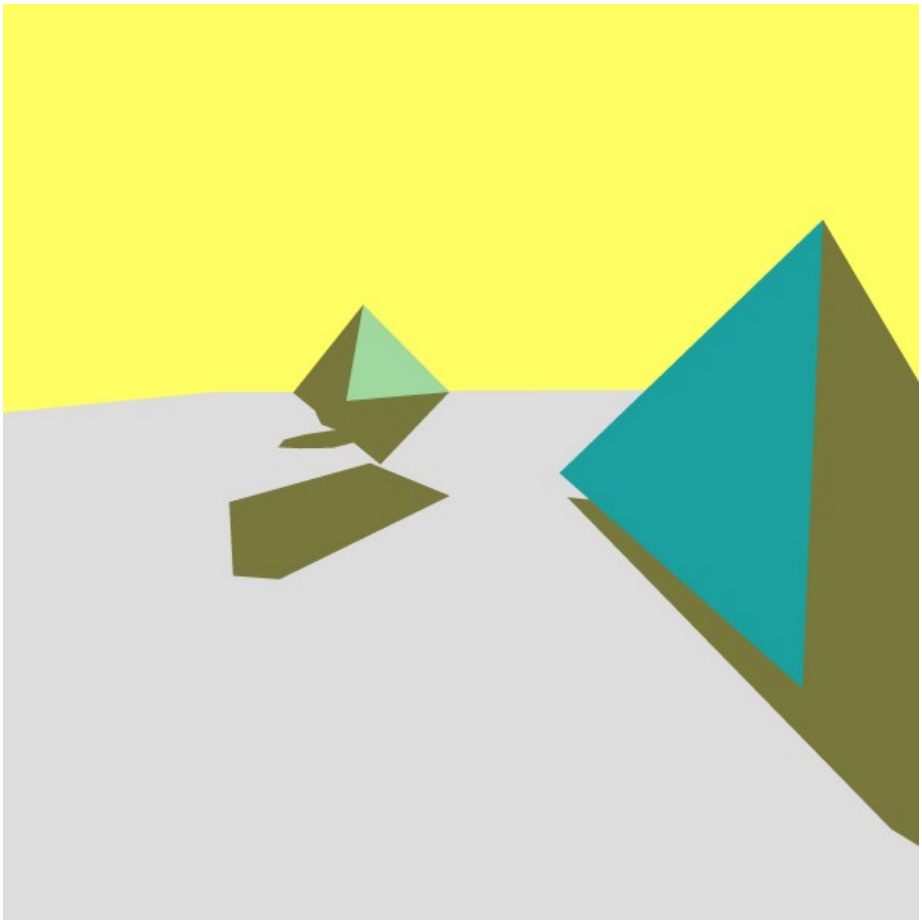
Таким образом, использование CUDA обеспечивает примерно **16-кратное ускорение** по сравнению с последовательной CPU-версией. Такой результат объясняется тем, что рендеринг каждого пикселя является независимой задачей, идеально подходящей для массового параллелизма GPU. В то время как CPU обрабатывает пиксели последовательно, видеокарта запускает тысячи потоков одновременно, что и приводит к столь существенному приросту производительности.

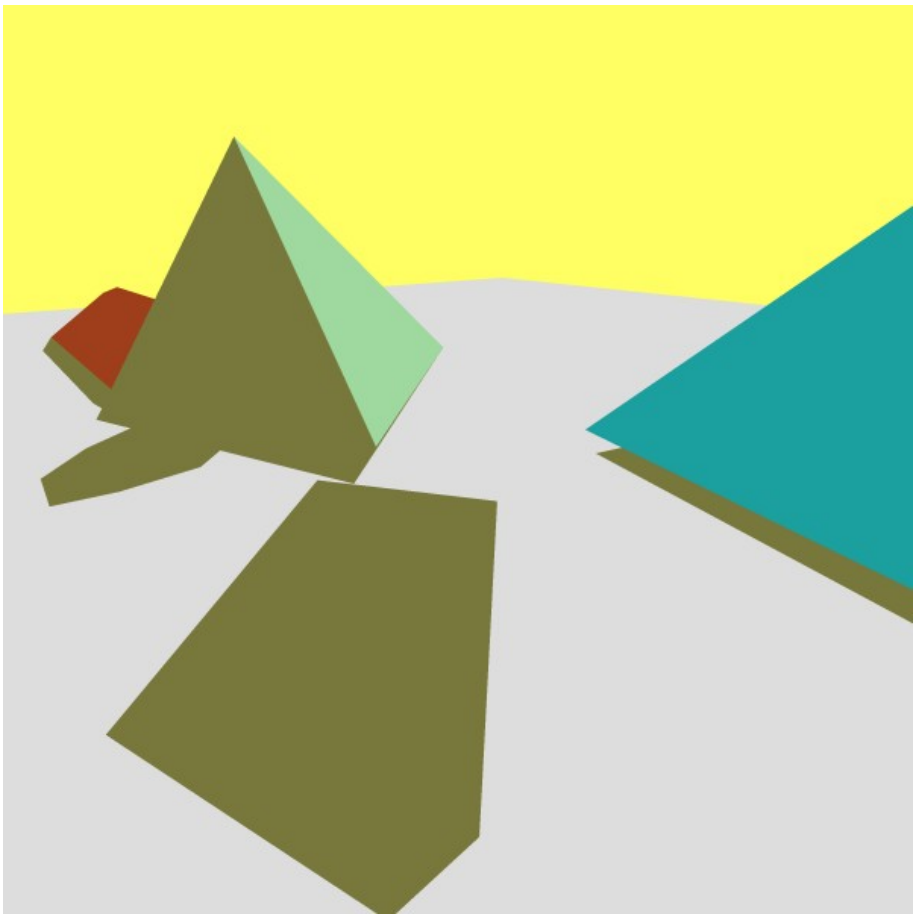
На графике зависимости времени построения кадра от параметра сглаживания SSAA видно, что увеличение числа лучей на пиксель приводит к почти линейному росту времени рендеринга. Это ожидаемо, поскольку при использовании SSAA каждый пиксель вычисляется как среднее нескольких трассировок, а значит стоимость обработки возрастает пропорционально количеству суб-лучей. При sqrt=1 время минимальное, при sqrt=2 — возрастает примерно вдвое, при sqrt=4 — примерно в четыре раза, а при sqrt=8 рост становится уже весьма значительным, что соответствует 64 лучам на один пиксель. На втором графике, показывающем среднее время в зависимости от количества лучей, этот эффект читается ещё яснее: зависимость почти линейная и демонстрирует, насколько сильно SSAA влияет на производительность GPU.

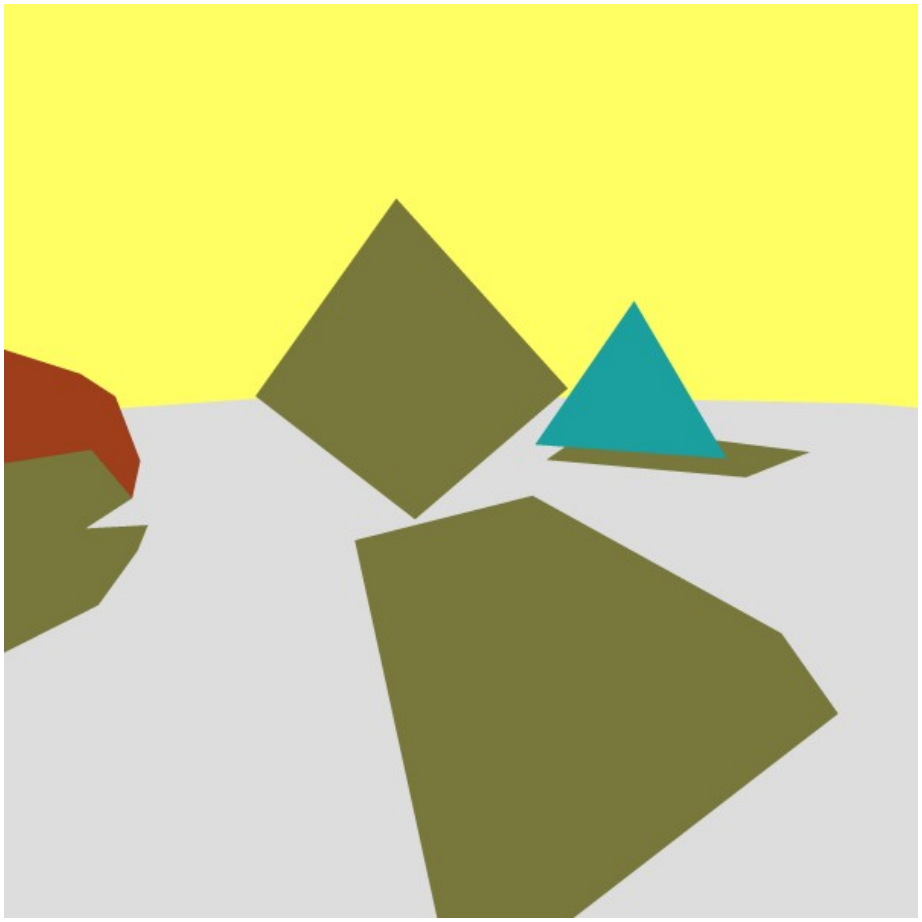


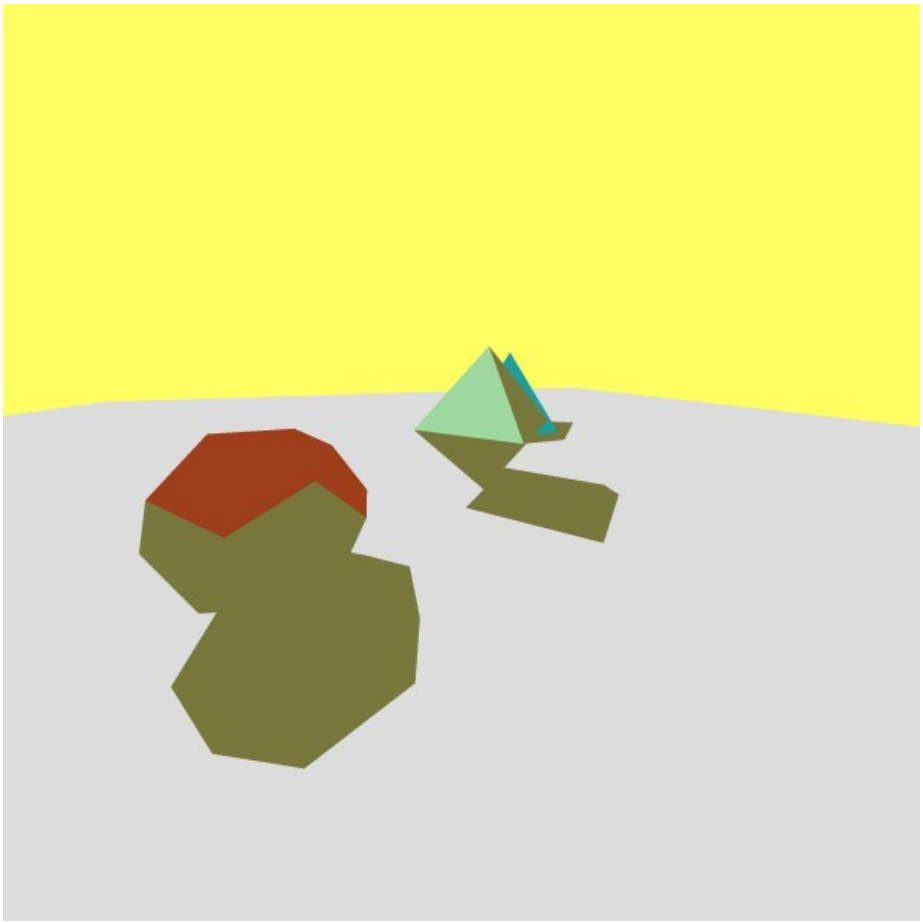
Далее продемонстрирую некоторые кадры, которые получились при следующих входных данных:

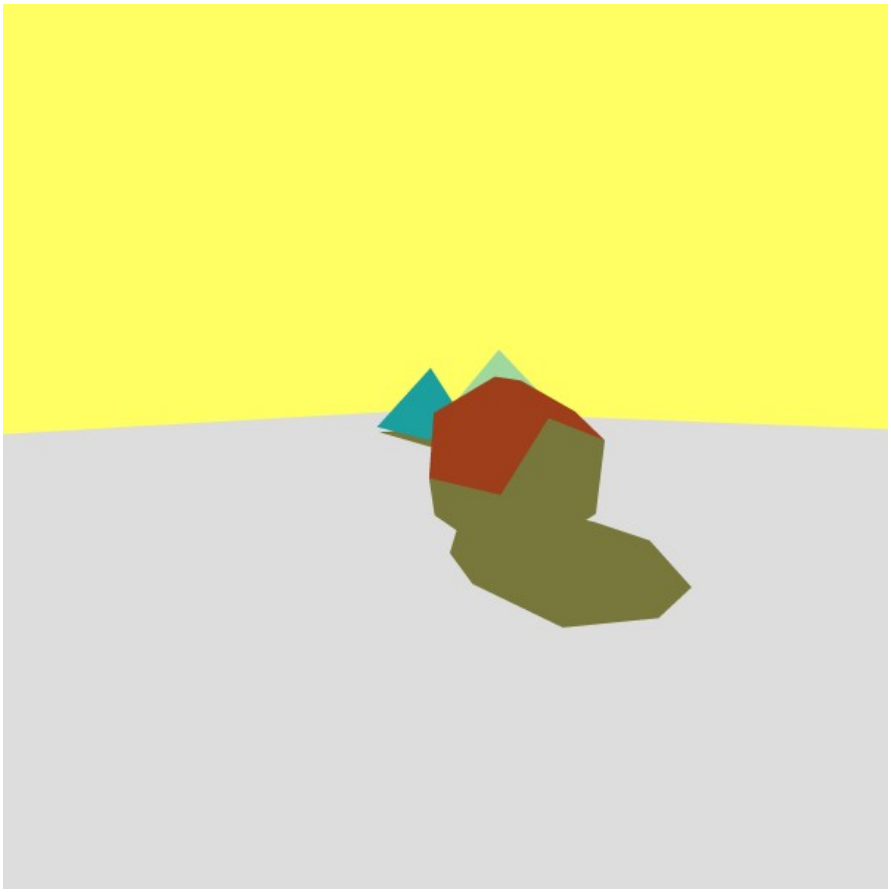
```
100
res/%d.data
600 600 120

10.0 3.0 0.0  4.0 1.0  2.0 6.0 1.0  0.0 0.0
2.0 0.0 0.0  0.5 0.1  1.0 4.0 1.0  0.0 0.0

3.0 3.0 0.5   1.0 0.2 0.2   2.0   0.0 0.0  0
0.0 0.0 0.7   0.2 0.9 0.2   1.75  0.0 0.0  0
-3.0 -3.0 0.0 0.2 0.4 1.0   1.2   0.0 0.0  0

-40.0 -40.0 -1.0  -40.0 40.0 -1.0  40.0 40.0 -1.0  40.0 -40.0 -1.0 ~/floor.data  0.6 0.6 0.6  0.0

1
-10.0 0.0 15.0  0.9 0.9 0.9

1 4
```

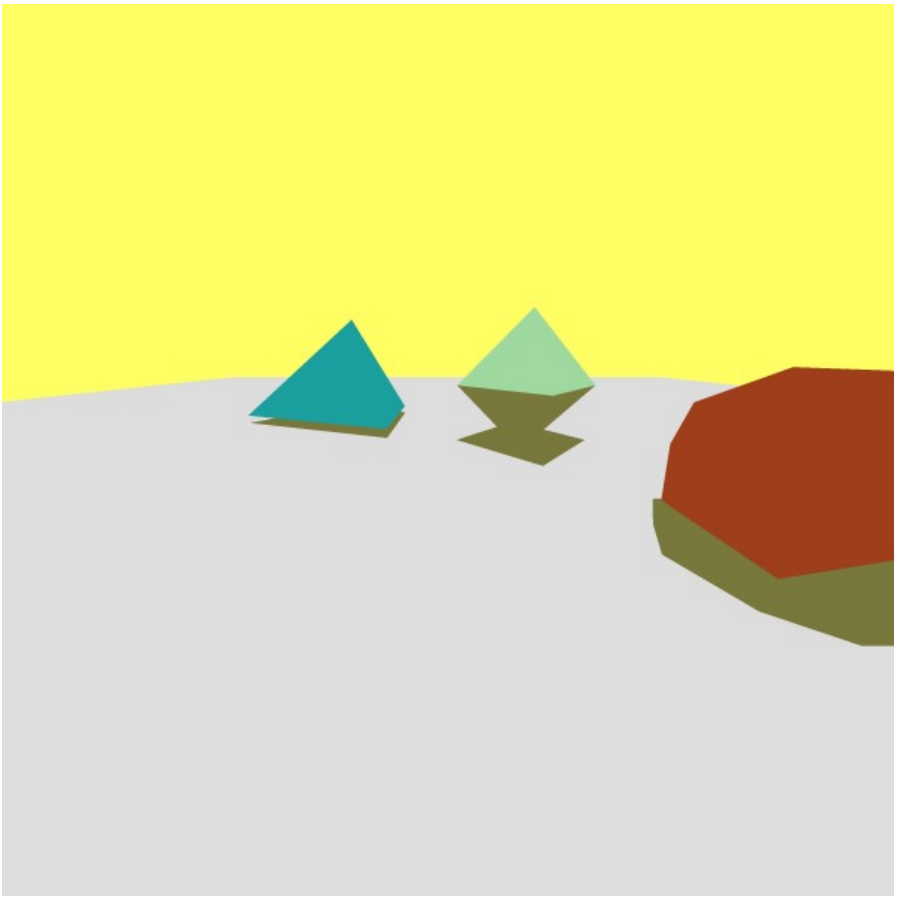


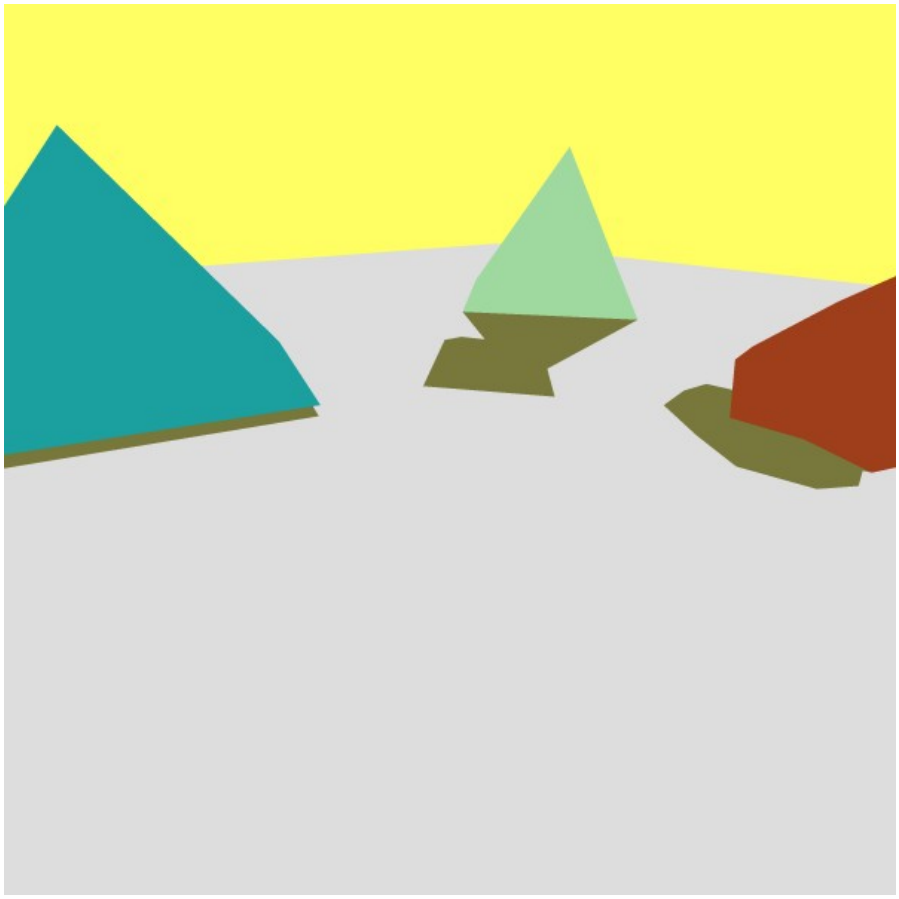


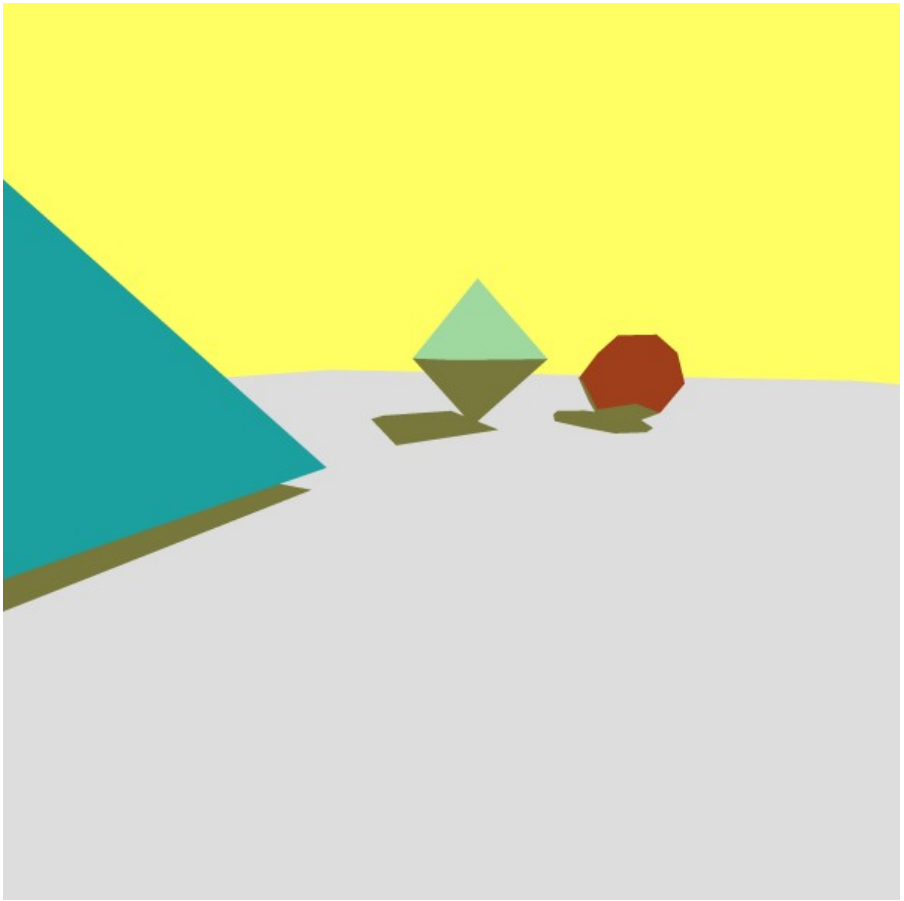


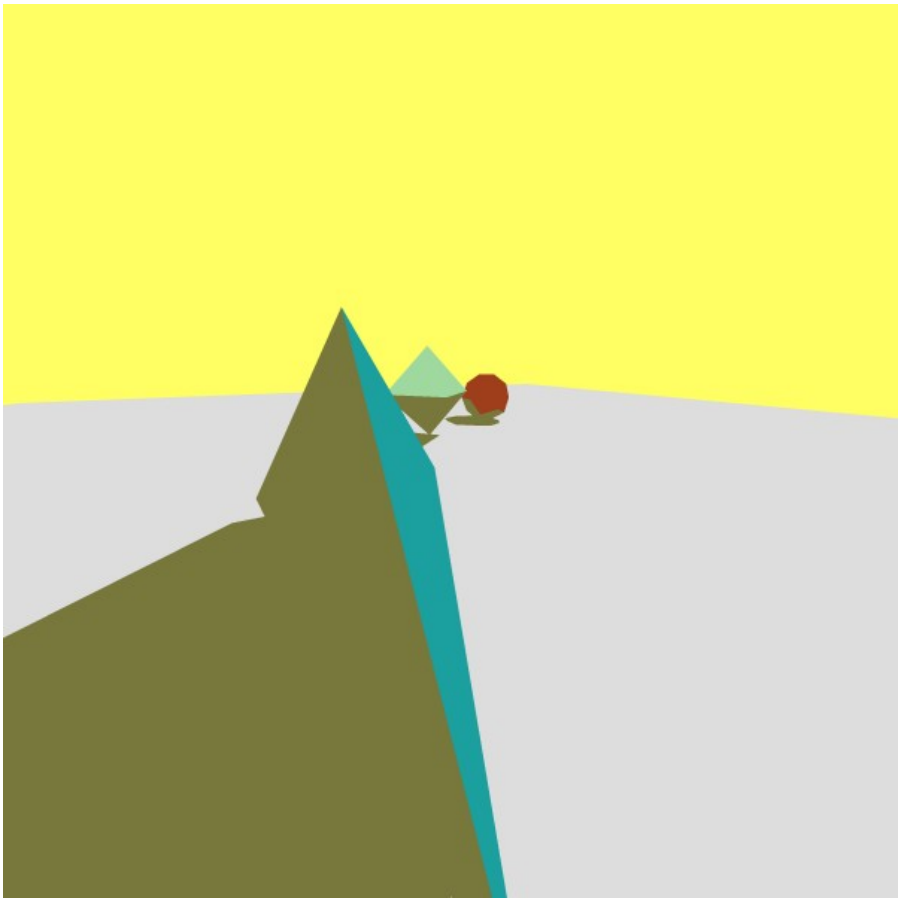


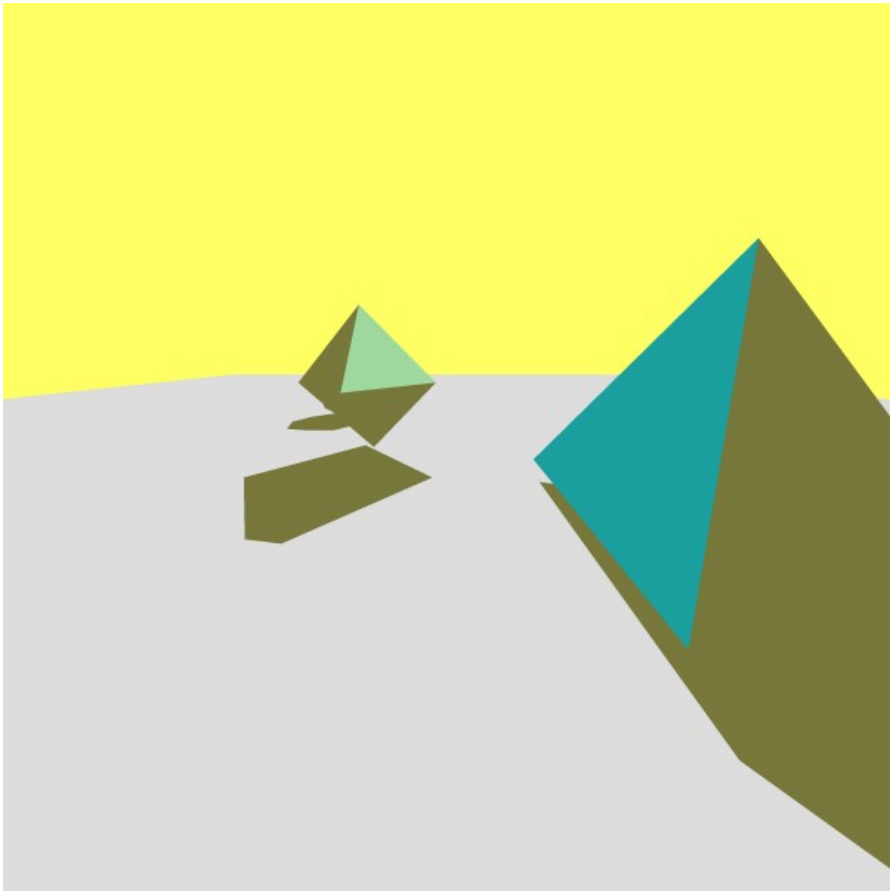


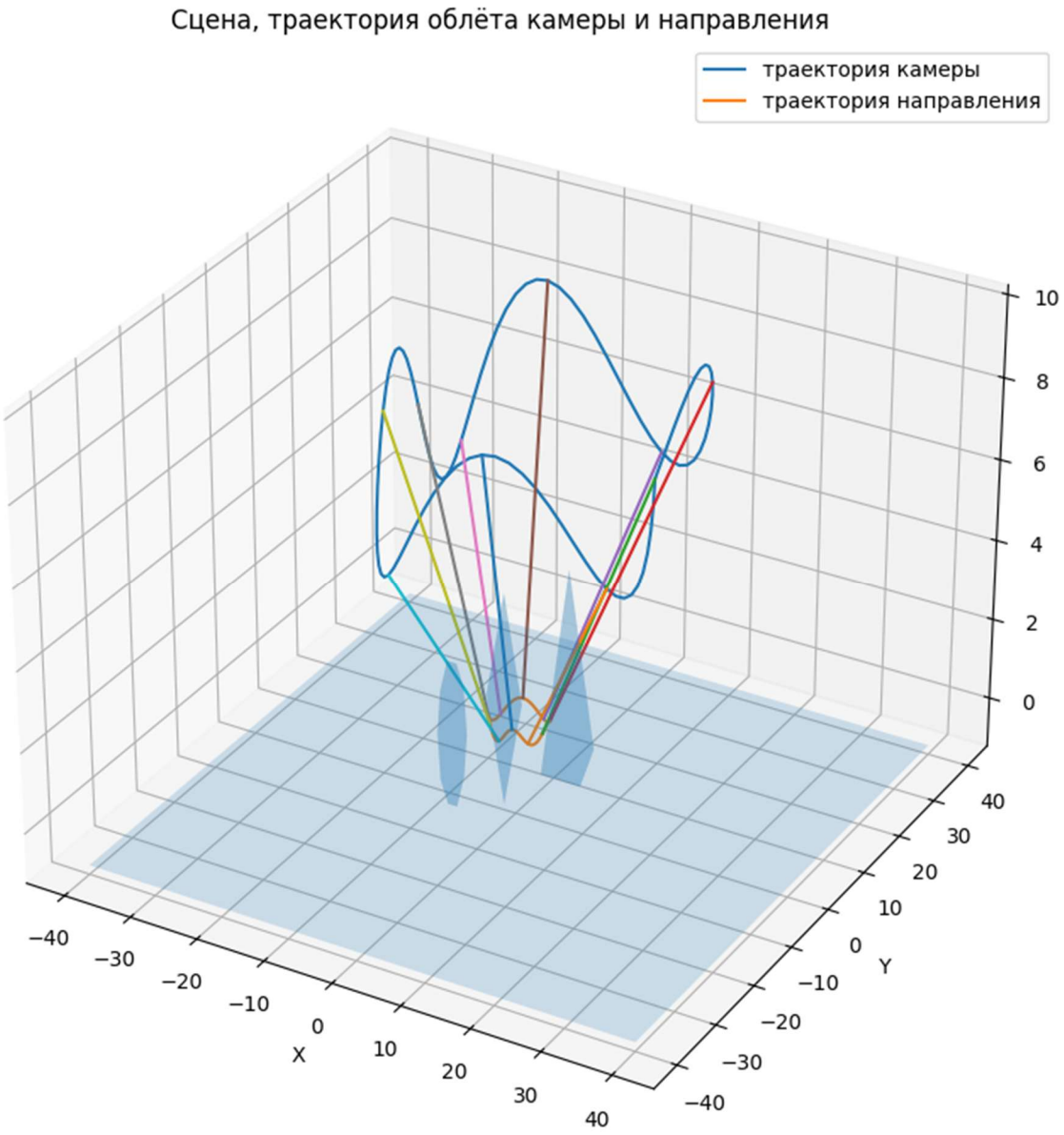












На рисунке представлена трёхмерная схема сцены, включающая геометрию всех используемых объектов и траектории движения камеры. Пол и тела отображены в виде полупрозрачной каркасной модели, что позволяет визуально оценить структуру сцены и относительное расположение объектов. Синяя линия показывает путь, по которому камера движется в процессе анимации, а оранжевая линия отражает траекторию точки, на которую камера направлена в каждый момент времени.

Выводы

Реализованный алгоритм обратной трассировки лучей применим в задачах визуализации трёхмерных сцен, моделировании освещения, генерации теней и создании анимации с произвольным движением камеры; такие методы используются в офлайн-рендеринге, инженерной графике и обучающих системах. В процессе разработки возникли трудности, связанные с корректной реализацией пересечения луча с треугольником, переносом вычислений в CUDA-среду и отладкой параллельных ядер, где ошибки проявляются неявно. Сравнение CPU- и GPU-версий показало закономерное преимущество графического процессора благодаря естественной параллельности задачи: вычисление цвета каждого пикселя изолировано и хорошо распараллеливается. Полученные результаты подтверждают корректность алгоритма и демонстрируют, что даже упрощённая модель освещения и геометрии позволяет эффективно визуализировать сцену и обеспечивать значительный прирост производительности при использовании GPU.

Литература

1. https://docs.nvidia.com/cuda/
2. http://www.ray-tracing.ru/