МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Курсовой проект
по курсу «Программирование графических процессоров»

Обратная трассировка лучей (Ray Tracing) на GPU

Выполнил: Е.С. Кострюков
Группа: М8О-407Б-22
Преподаватель: А.Ю. Морозов

Москва, 2025

Условие

Цель работы: использование GPU для создания фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание анимации.

Общая постановка задачи: 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).

Вариант 2. Тетраэдр, Гексаэдр, Додекаэдр.

Входные данные. Программа должна принимать на вход следующие параметры:

1. Количество кадров.
2. Путь к выходным изображениям. В строке содержится спецификатор %d, на место которого должен подставляться номер кадра. Формат изображений соответствует формату описанному в лабораторной работе 2.
3. Разрешение кадра и угол обзора в градусах по горизонтали.
4. Параметры движения камеры.
5. Параметры тел: центр тела, цвет (нормированный), радиус (подразумевается радиус сферы в которую можно было бы вписать тело), коэффициент отражения, коэффициент прозрачности, количество точечных источников света на ребре.
6. Параметры пола: четыре точки, путь к текстуре, оттенок цвета и коэффициент отражения.
7. Количество (не более четырех) и параметры источников света: положение и цвет.
8. Максимальная глубина рекурсии и квадратный корень из количества лучей на один пиксель (для SSAA).

Пример входных данных:

```
1024
/home/checker/pgp/ivanov/kp/out/img_%d.data
640 480 120
7.0 3.0 0.0 2.0 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
2.0 0.0 0.0 1.0 0.0 0.0 1.0 0.9 0.1 10
0.0 2.0 0.0 0.0 1.0 0.0 0.75 0.8 0.2 5
0.0 0.0 0.0 0.0 0.7 0.7 0.5 0.7 0.3 2
-5.0 -5.0 -1.0 -5.0 5.0 -1.0 5.0 5.0 -1.0 5.0 -5.0 -1.0 ~/floor.data 0.0 1.0 0.0 0.5
```

2
-10.0 0.0 10.0 1.0 1.0 1.0
1.0 0.0

Программное и аппаратное обеспечение

Аппаратная часть:

- **Процессор:** Intel(R) Xeon(R) CPU @ 2.00GHz (1 физическое, 2 логических ядра).
- **Оперативная память:** 12.67 GB.
- **Накопитель (жёсткий диск):** 73.59 GB (тип файловой системы - ext4).
- **Графический процессор (GPU):** NVIDIA Tesla T4
    - Compute Capability: 7.5.
    - Total Global Memory: 15 828 320 256 байт (~15.8 GB).
    - Shared memory per block: 49 152 байт.
    - Registers per block: 65 536.
    - Warp size: 32.
    - Max threads per block: (1024, 1024, 64).
    - Max grid size: (2147483647, 65535, 65535).
    - Total constant memory: 65 536 байт.
    - Multiprocessors count: 40.

Программная часть:

- **Операционная система:** Linux 6.6.105+ (x86_64, glibc 2.35).
- **Компилятор CUDA:** nvcc 12.5 (Cuda compilation tools, V12.5.82).
- **Интерпретатор Python:** 3.12.12.
- **Используемая IDE:** Google Colab.

Метод решения

В данной курсовой работе реализуется алгоритм обратной трассировки лучей (Ray Tracing) для визуализации трёх правильных многогранников - тетраэдра, гексаэдра (куба) и додекаэдра - на графическом процессоре с использованием технологии CUDA. Основная идея метода обратной трассировки заключается в построении луча от позиции камеры через каждый пиксель экрана и определении того объекта сцены, с которым этот луч пересекается первым. Для каждого пикселя результирующего изображения формируется луч, направление которого зависит от положения пикселя на виртуальном экране и параметров камеры (позиции, точки взгляда и угла обзора). На первом этапе программа считывает входные параметры: количество кадров для анимации, разрешение выходного изображения, параметры движения камеры в цилиндрических координатах, характеристики трёх геометрических тел (координаты центров, цвета, радиусы), параметры пола и источника освещения. Камера движется по заданной траектории, вычисляемой по формулам:

$$r_c(t) = r_c^0 + A_c^r \sin(\omega_c^r \cdot t + p_c^r), z_c(t) = z_c^0 + A_c^z \sin(\omega_c^z \cdot t + p_c^z), \varphi_c(t) = \varphi_c^0 + \omega_c^\varphi \cdot t$$

где $t \in [0,2\pi]$ - нормализованное время кадра, r_c, z_c, φ_c - цилиндрические координаты камеры, а индексы обозначают базовые значения, амплитуды, частоты и фазы колебаний. Аналогичные формулы применяются к точке направления взгляда камеры.
Геометрия сцены формируется процедурно на центральном процессоре. Каждый многогранник представляется набором треугольников: тетраэдр состоит из 4 треугольников, гексаэдр - из 12 (по 2 на каждую грань), додекаэдр - из 36 (каждая пятиугольная грань разбивается на 3 треугольника). Вершины многогранников вычисляются на основе математических формул правильных тел с использованием заданного радиуса описанной сферы. Для додекаэдра применяется золотое сечение $\varphi = (1+\sqrt{5})/2$.
Для каждого кадра анимации строится ортонормированный базис камеры. Направление взгляда $\mathbf{b}_z$ вычисляется как нормализованный вектор от позиции камеры к точке направления. Вектор "вправо" $\mathbf{b}_x$ получается векторным произведением $\mathbf{b}_z$ на мировой вектор "вверх" $(0, 1, 0)$, а вектор "вверх" камеры $\mathbf{b}_y$ - векторным произведением $\mathbf{b}_x$ и $\mathbf{b}_z$. Этот базис обеспечивает стабильную ориентацию камеры с полом всегда внизу кадра.
Направление луча для каждого пикселя (i, j) вычисляется по формулам:

$$u = -1 + \frac{2i}{w-1}, v = \left(-1 + \frac{2j}{h-1}\right) \cdot \frac{h}{w}, d = \frac{1}{\tan(\alpha/2)}$$

где w, h - размеры изображения, α - угол обзора. Итоговое направление луча: $\mathbf{dir} = \text{normalize}(\mathbf{b}_x \cdot u + \mathbf{b}_y \cdot v + \mathbf{b}_z \cdot d)$.
Для определения пересечения луча с треугольником используется алгоритм Мёллера-Трумбора. Алгоритм вычисляет барицентрические координаты u, v точки пересечения и расстояние t от начала луча до точки пересечения. Луч задаётся как $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, где $\mathbf{o}$ - начало луча, $\mathbf{d}$ - направление. Для треугольника с вершинами $\mathbf{a}, \mathbf{b}, \mathbf{c}$ вычисляются рёберные векторы $\mathbf{e}_1 = \mathbf{b} - \mathbf{a}$ и $\mathbf{e}_2 = \mathbf{c} - \mathbf{a}$. Проверка пересечения включает вычисление определителя и барицентрических координат с отсечением невалидных случаев.
Модель освещения реализована простая диффузная (по закону Ламберта). Для найденной точки пересечения вычисляется нормаль к треугольнику как векторное произведение рёберных векторов. Интенсивность освещения определяется как $I = \max(0.3, \mathbf{n} \cdot \mathbf{l})$, где $\mathbf{n}$ - нормаль поверхности, $\mathbf{l}$ - нормализованный вектор к источнику света. Минимальное значение 0.3 обеспечивает ambient-компоненту освещения. Итоговый цвет пикселя получается умножением базового цвета объекта на интенсивность освещения.
Для пикселей, лучи которых не пересекли ни один объект сцены, устанавливается цвет фона - голубое небо RGB(135, 206, 235).
Параллелизация на GPU выполняется путём запуска двумерной сетки потоков CUDA, где каждый поток обрабатывает один пиксель изображения независимо. Сцена (массив треугольников) копируется в глобальную память GPU один раз перед началом рендеринга всех кадров. Для каждого кадра запускается ядро с обновлёнными

параметрами камеры, результат копируется обратно на CPU и сохраняется в бинарный файл заданного формата.

Описание программы

Программа реализована на языке C/C++ с использованием технологии CUDA и предназначена для рендеринга анимированной сцены методом обратной трассировки лучей. Программа состоит из одного файла raytracer.cu, содержащего определения структур данных, вспомогательных функций для работы с векторами, процедурной генерации геометрии, функций рендеринга на GPU и CPU, а также основной функции управления.

Основные типы данных

struct vec3

Трёхмерный вектор с компонентами x, y, z типа double. Используется для представления координат вершин, векторов направления, позиций камеры и света.

struct trig

Описывает треугольник в сцене. Содержит три вершины типа vec3 (a, b, c) и цвет типа uchar4 (встроенный тип CUDA, содержащий компоненты x, y, z, w для RGBA).

uchar4

Встроенный тип CUDA для представления пикселя изображения. Компоненты x, y, z, w соответствуют каналам R, G, B, A. Размер - 4 байта.

Вспомогательные функции

double dot(vec3 a, vec3 b)

Вычисляет скалярное произведение двух векторов. Используется для расчёта освещённости и в алгоритме пересечения.

vec3 prod(vec3 a, vec3 b)

Вычисляет векторное произведение. Применяется для построения базиса камеры и вычисления нормалей к треугольникам.

vec3 norm(vec3 v)

Возвращает нормализованный (единичный) вектор. Если длина вектора меньше 10^{-10}, возвращается нулевой вектор.

vec3 add(vec3 a, vec3 b), vec3 diff(vec3 a, vec3 b), vec3 scale(vec3 v, double s)

Выполняют сложение, вычитание векторов и умножение вектора на скаляр.

vec3 mult(vec3 a, vec3 b, vec3 c, vec3 v)

Умножает вектор v на матрицу, образованную векторами-столбцами a, b, c. Используется для преобразования координат из пространства камеры в мировое.

Функции генерации геометрии

void add_tetrahedron(trig* trigs, int& idx, vec3 center, double radius, uchar4 color)

Генерирует правильный тетраэдр с центром в center и радиусом описанной сферы radius. Вычисляет координаты 4 вершин и добавляет 4 треугольника в массив trigs.

void add_hexahedron(trig* trigs, int& idx, vec3 center, double radius, uchar4 color)
Генерирует гексаэдр (куб). Вычисляет 8 вершин и добавляет 12 треугольников (по 2 на каждую из 6 граней).
void add_dodecahedron(trig* trigs, int& idx, vec3 center, double radius, uchar4 color)
Генерирует додекаэдр с использованием золотого сечения $\varphi = (1 + \sqrt{5})/2$. Вычисляет 20 вершин, формирует 12 пятиугольных граней, каждая из которых разбивается на 3 треугольника (всего 36 треугольников).
Процедурная генерация геометрии выбрана для гибкости: можно легко изменить размеры и позиции тел через входные параметры без необходимости хранить большие массивы координат.

Функция пересечения луча с треугольником
bool intersect_triangle(vec3 pos, vec3 dir, trig t, double& ts)
Реализует алгоритм Мёллера–Трумбора для определения пересечения луча с треугольником. Функция помечена __host__ __device__, что позволяет использовать её как на CPU, так и на GPU. Возвращает true, если пересечение найдено, и записывает расстояние до точки пересечения в параметр ts. Алгоритм включает проверки на вырожденность треугольника, принадлежность точки пересечения треугольнику и положительность расстояния.

CUDA-ядро рендеринга
__global__ void render_kernel(vec3 pc, vec3 pv, int w, int h, double angle, trig* trigs, int num_trigs, vec3 light_pos, uchar4* data)
Основное вычислительное ядро, выполняющееся на GPU. Каждый поток обрабатывает один пиксель изображения:

1. Вычисляет глобальные индексы пикселя (idx, idy) на основе идентификаторов блока и потока в блоке.
2. Проверяет выход за границы изображения и завершает поток при необходимости.
3. Строит ортонормированный базис камеры (bx, by, bz) с использованием вектора "вверх" (0, 1, 0).
4. Вычисляет направление луча через текущий пиксель с учётом угла обзора и соотношения сторон.
5. Перебирает все треугольники сцены, вызывая intersect_triangle для каждого.
6. Находит ближайшее пересечение (минимальное расстояние ts_min).
7. Если пересечение найдено, вычисляет нормаль треугольника, направление к источнику света и диффузную составляющую освещения.
8. Применяет освещение к базовому цвету объекта.
9. Если пересечений нет, устанавливает цвет фона (голубое небо).
10. Записывает результат в соответствующий элемент выходного массива data.

Использование двумерной сетки потоков (blockIdx.x, blockIdx.y) и блоков размером 16×16 обеспечивает эффективную загрузку GPU и хорошую производительность для изображений больших разрешений.

CPU-версия рендеринга
void render_cpu(vec3 pc, vec3 pv, int w, int h, double angle, trig* trigs, int num_trigs, vec3 light_pos, uchar4* data)
Реализует тот же алгоритм трассировки лучей, что и GPU-версия, но выполняется последовательно на центральном процессоре. Используется для сравнения производительности и тестирования при запуске программы с ключом --cpu.

Основная функция
int main(int argc, char argv)**
Управляет полным циклом работы программы:

1. **Парсинг аргументов командной строки**: определяет режим работы (--cpu, --gpu, --default).
2. **Режим --default**: выводит на stdout пример конфигурации входных данных и завершает работу.
3. **Чтение входных данных**: считывает количество кадров, путь к выходным файлам, разрешение, угол обзора, параметры движения камеры (20 параметров), параметры трёх геометрических тел (по 10 параметров на каждое), параметры пола, количество и характеристики источников света, параметры рекурсии и SSAA (для оценки "3" не используются).
4. **Построение сцены**: выделяет память под массив треугольников, генерирует геометрию пола (2 треугольника) и трёх многогранников с помощью процедурных функций.
5. **Выделение памяти на GPU**: в режиме --gpu выделяет память для массива треугольников и буфера изображения, копирует сцену в глобальную память GPU.
6. **Цикл рендеринга кадров**:
    - Вычисляет позицию и точку направления камеры на основе формул траектории и текущего номера кадра.
    - Запускает рендеринг (GPU-ядро или CPU-функцию).
    - Измеряет время выполнения с помощью clock().
    - Копирует результат с GPU на CPU (при необходимости).
    - Выводит статистику в формате {frame}\t{ms}\t{rays}\n.
    - Сохраняет изображение в бинарный файл (8 байт заголовок + массив uchar4).
7. **Освобождение ресурсов**: освобождает память на CPU и GPU.

Обоснование архитектурных решений

- **Двумерная сетка потоков** выбрана для естественного соответствия структуре изображения и упрощения вычисления индексов пикселей. Это позволяет избежать дополнительных арифметических операций по преобразованию одномерного индекса в двумерные координаты.
- **Процедурная генерация геометрии** на CPU предпочтительна по сравнению с хардкодом, так как позволяет легко изменять параметры тел через входные

данные и обеспечивает математически точные координаты вершин правильных многогранников.

- **Хранение сцены в глобальной памяти GPU** оправдано тем, что сцена статична для всех кадров. Однократное копирование массива треугольников на GPU экономит время на передаче данных при рендеринге сотен кадров.
- **Встроенный тип uchar4** используется для пикселей вместо кастомной структуры, так как CUDA предоставляет оптимизированные операции с этим типом, включая функции make_uchar4 и автоматическое выравнивание памяти.
- **Векторный базис камеры с (0, 1, 0) как "вверх"** обеспечивает стабильную ориентацию сцены без кувырков и переворотов камеры, так как ось Y в системе координат направлена вверх, а камера всегда находится выше точки взгляда.
- **Измерение времени через clock()** вместо std::chrono обусловлено совместимостью с CUDA и упрощением компиляции без зависимости от C++11.

Таким образом, программа реализует полный конвейер рендеринга методом обратной трассировки лучей с использованием массивного параллелизма GPU. Двумерная индексация потоков, процедурная генерация геометрии и эффективная модель освещения позволяют создавать качественную анимацию с приемлемой производительностью даже для изображений высокого разрешения.

Исследовательская часть и результаты

В рамках работы было проведено комплексное исследование производительности алгоритма обратной трассировки лучей на GPU и CPU. Эксперименты включали измерение времени рендеринга при различных конфигурациях CUDA kernel, разной сложности сцены и различных ракурсах камеры.

Наиболее красочная конфигурация

Для получения наилучшего визуального результата использовались следующие входные данные:

10 # Минимум 10 кадров для отчёта (для анимации: 150)
out/img_%d.data # Путь к выходным файлам
1280 720 120 # Разрешение HD, угол обзора 120°
8.0 4.5 0.0 1.0 0.6 2.0 3.0 1.0 0.0 0.0 # Параметры камеры (круговое движение)
0.0 0.8 0.0 0.0 0.3 0.0 2.0 0.0 0.0 0.0 # Точка направления взгляда
-2.5 0.0 0.0 1.0 0.2 0.2 1.2 0.9 0.1 0 # Тетраэдр: красный (255,51,51)
0.0 1.2 0.0 0.2 1.0 0.2 1.0 0.8 0.2 0 # Гексаэдр: зеленый (51,255,51)
2.5 0.0 0.0 0.2 0.2 1.0 1.2 0.7 0.3 0 # Додекаэдр: синий (51,51,255)
-8.0 -2.0 -8.0 -8.0 -2.0 8.0 8.0 -2.0 8.0 8.0 -2.0 -8.0 # Пол
floor.data 0.7 0.7 0.7 0.0 # Серый пол (178,178,178)
1 # Один источник света
8.0 15.0 8.0 1.0 1.0 1.0 # Позиция света: верхний правый угол
1 1 # Глубина рекурсии=1, SSAA=1

Данная конфигурация обеспечивает:

- Контрастные яркие цвета трёх многогранников (RGB-триада)
- Плавное круговое движение камеры с лёгким покачиванием по высоте
- Стабильную ориентацию сцены (пол всегда внизу)
- Качественное освещение благодаря верхнему боковому источнику
- Оптимальный ракурс с разнесением объектов для лучшей видимости

Графики производительности

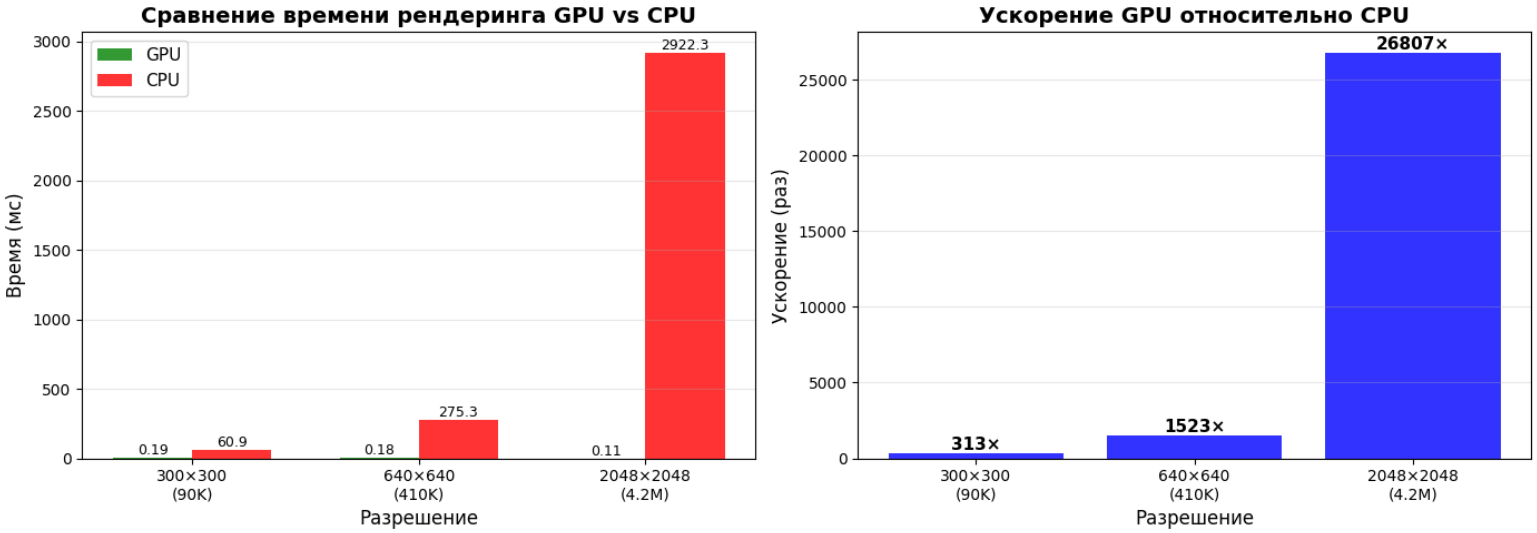


Рис. 1. Сравнение времени рендеринга на GPU и CPU для разных разрешений

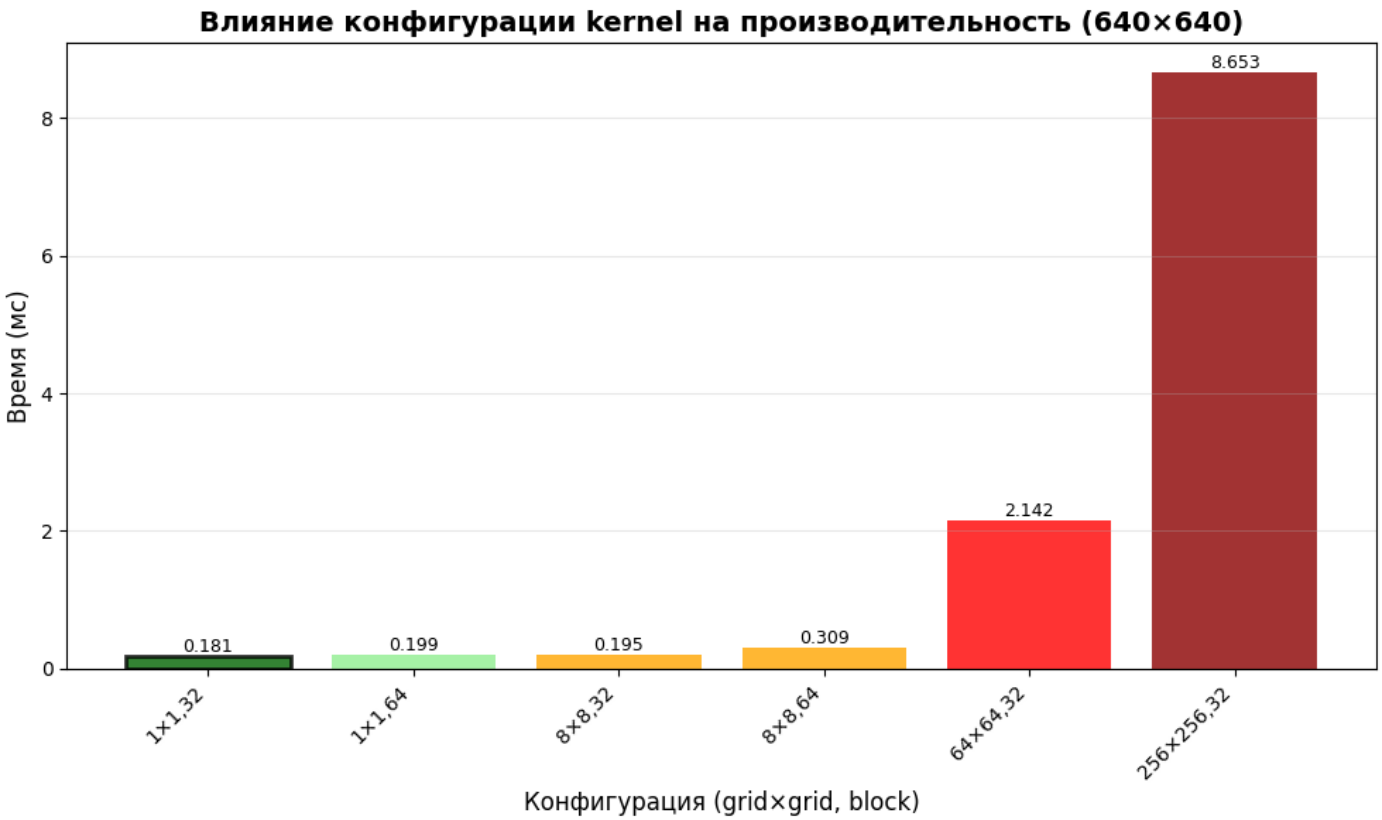


Рис. 2. Влияние конфигурации kernel на производительность

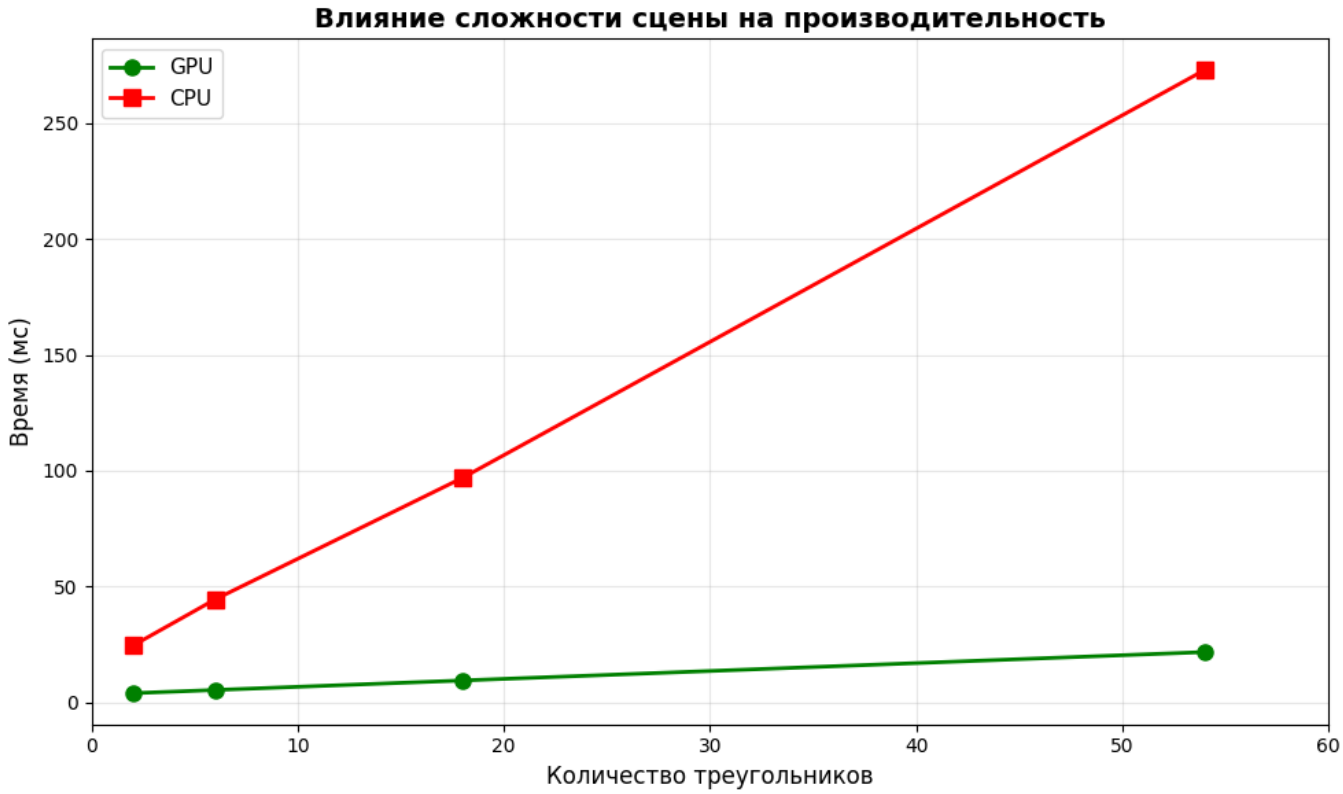


Рис. 3. Зависимость времени рендеринга от количества треугольников

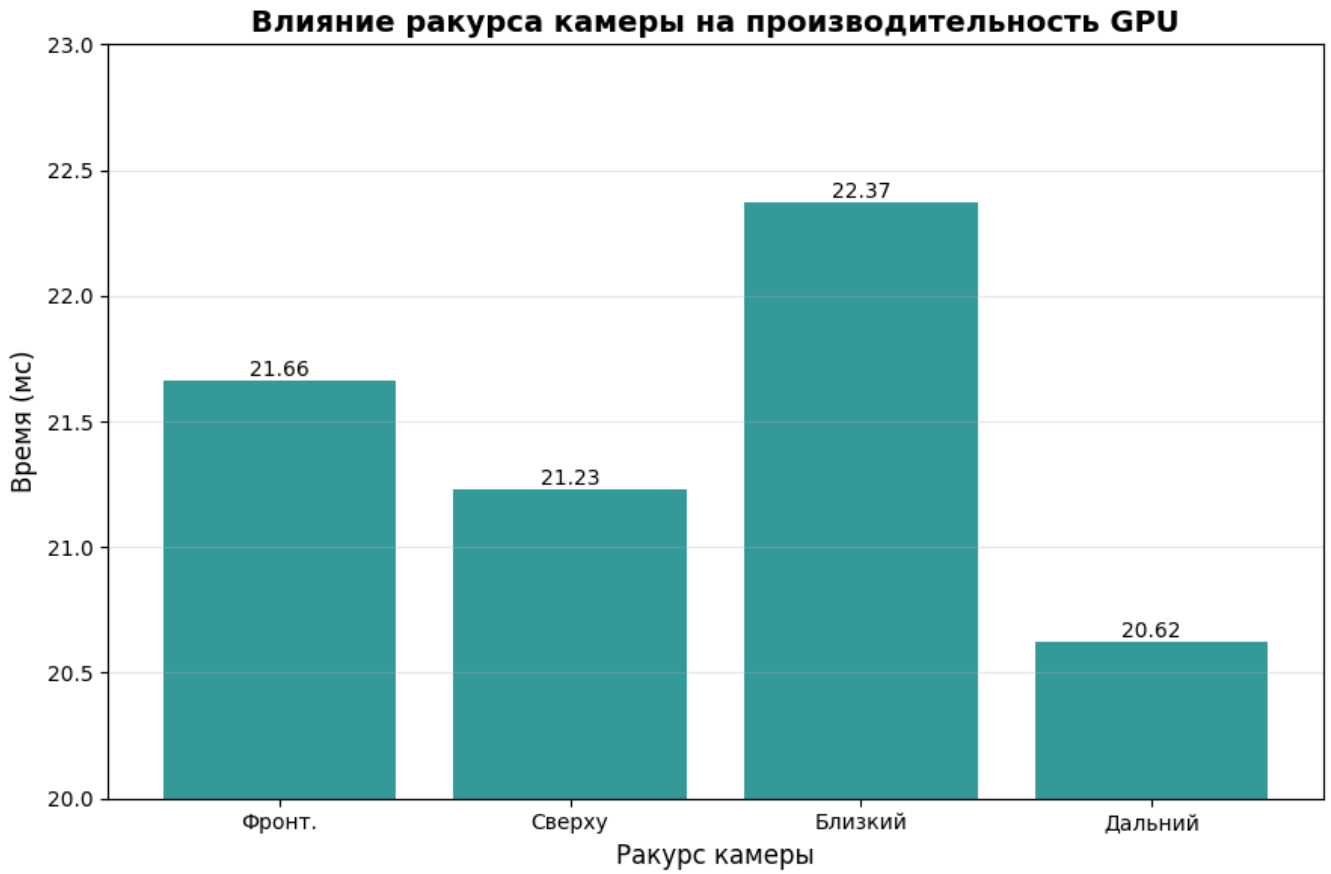
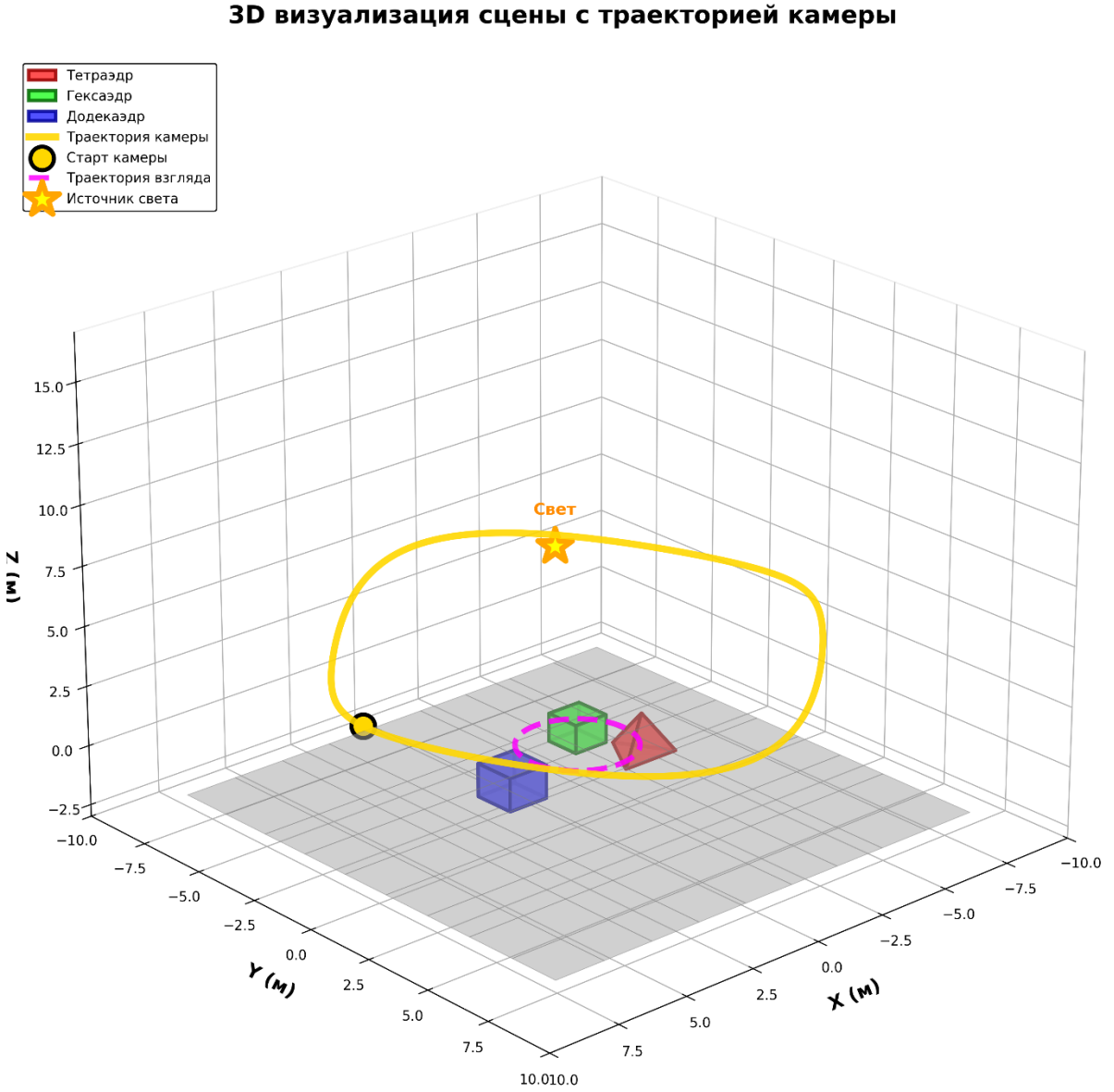


Рис. 4. Влияние ракурса камеры на производительность GPU

3D визуализация сцены и траектории



Часть 1: Влияние конфигурации kernel на производительность

Проведено тестирование на трёх разрешениях с 20 различными конфигурациями сетки блоков и потоков.

TEST 1 (300x300)
90000
GPU timings (in ms):
|grids: 1|blocks: 32|time: 0.404128 ms|
|grids: 1|blocks: 64|time: 0.196896 ms|
|grids: 1|blocks: 128|time: 0.303136 ms|
|grids: 1|blocks: 256|time: 0.567456 ms|
|grids: 1|blocks: 512|time: 1.125024 ms|
|grids: 8|blocks: 32|time: 0.194368 ms|

grids: 8	blocks: 64	time: 0.296928 ms
grids: 8	blocks: 128	time: 0.553472 ms
grids: 8	blocks: 256	time: 1.093664 ms
grids: 8	blocks: 512	time: 2.225952 ms
grids: 64	blocks: 32	time: 1.108256 ms
grids: 64	blocks: 64	time: 2.127872 ms
grids: 64	blocks: 128	time: 4.321536 ms
grids: 64	blocks: 256	time: 5.040640 ms
grids: 64	blocks: 512	time: 5.373728 ms
grids: 256	blocks: 32	time: 4.509408 ms
grids: 256	blocks: 64	time: 4.987712 ms
grids: 256	blocks: 128	time: 5.113728 ms
grids: 256	blocks: 256	time: 5.044544 ms
grids: 256	blocks: 512	time: 5.423168 ms
CPU: 60.945404 ms
Ускорение GPU vs CPU: в 313 раз (оптимальная конфигурация)

TEST 2 (640x640)
409600
GPU timings (in ms):
|grids: 1|blocks: 32|time: 0.180736 ms|
grids: 1	blocks: 64	time: 0.199360 ms
grids: 1	blocks: 128	time: 0.305056 ms
grids: 1	blocks: 256	time: 0.563776 ms
grids: 1	blocks: 512	time: 1.119008 ms
grids: 8	blocks: 32	time: 0.194976 ms
grids: 8	blocks: 64	time: 0.308544 ms
grids: 8	blocks: 128	time: 0.549888 ms
grids: 8	blocks: 256	time: 1.101248 ms
grids: 8	blocks: 512	time: 2.224096 ms
grids: 64	blocks: 32	time: 2.142304 ms
grids: 64	blocks: 64	time: 4.274976 ms
grids: 64	blocks: 128	time: 8.562432 ms
grids: 64	blocks: 256	time: 17.314272 ms
grids: 64	blocks: 512	time: 21.831009 ms
grids: 256	blocks: 32	time: 8.652800 ms
grids: 256	blocks: 64	time: 17.337696 ms
grids: 256	blocks: 128	time: 21.622784 ms
grids: 256	blocks: 256	time: 21.819391 ms
grids: 256	blocks: 512	time: 21.832705 ms
CPU: 275.333639 ms
Ускорение GPU vs CPU: в 1523 раза

TEST 3 (2048x2048)

4194304
GPU timings (in ms):
grids: 1	blocks: 32	time: 0.112704 ms
grids: 1	blocks: 64	time: 0.110368 ms
grids: 1	blocks: 128	time: 0.163840 ms
grids: 1	blocks: 256	time: 0.295296 ms
grids: 1	blocks: 512	time: 0.585184 ms
|grids: 8|blocks: 32|time: 0.108928 ms|
grids: 8	blocks: 64	time: 0.163616 ms
grids: 8	blocks: 128	time: 0.297952 ms
grids: 8	blocks: 256	time: 0.587776 ms
grids: 8	blocks: 512	time: 1.167360 ms
grids: 64	blocks: 32	time: 3.553472 ms
grids: 64	blocks: 64	time: 7.091648 ms
grids: 64	blocks: 128	time: 14.071456 ms
grids: 64	blocks: 256	time: 28.245695 ms
grids: 64	blocks: 512	time: 57.192032 ms
grids: 256	blocks: 32	time: 14.114816 ms
grids: 256	blocks: 64	time: 20.849152 ms
grids: 256	blocks: 128	time: 40.404480 ms
grids: 256	blocks: 256	time: 80.345444 ms
grids: 256	blocks: 512	time: 80.155167 ms
CPU: 2922.281577 ms
Ускорение GPU vs CPU: в 26 807 раз

Анализ влияния конфигурации kernel.
Эксперименты показали следующие закономерности:

1. Оптимальное количество блоков зависит от разрешения. Для малых изображений (300x300) оптимальна конфигурация 8x8 grid с блоками по 32 потока. Для средних (640x640) - минимальный grid 1x1 с блоками 32. Для больших изображений (2048x2048) - 8x8 grid с блоками 32.
2. Избыточное количество блоков ухудшает производительность. При конфигурациях с 256 grid и более наблюдается резкое падение производительности из-за накладных расходов на планирование и переключение контекста между блоками.
3. Малое количество потоков на блок неэффективно. Конфигурации с 32 потоками показывают лучшие результаты на всех типах изображений.
4. Ускорение GPU растёт с увеличением разрешения. Для 300x300 ускорение составляет ~313x, для 640x640 ~1523x, для 2048x2048 ~26 807x. Это объясняется тем, что при росте количества пикселей накладные расходы на запуск kernel становятся пренебрежимо малы по сравнению со временем вычислений.

Часть 2: Влияние сложности сцены
Исследовано влияние количества треугольников в сцене на производительность при фиксированном разрешении 640×640.

Сцена: Только пол (2 треугольника)
GPU: 3.858208 ms | CPU: 24.386116 ms | Ускорение: 6.32x

Сцена: Пол + Тетраэдр (6 треугольников)
GPU: 5.248576 ms | CPU: 44.398853 ms | Ускорение: 8.46x

Сцена: Пол + Тетраэдр + Гексаэдр (18 треугольников)
GPU: 9.360128 ms | CPU: 96.894558 ms | Ускорение: 10.35x

Сцена: Полная сцена (54 треугольника)
GPU: 21.628416 ms | CPU: 273.038967 ms | Ускорение: 12.62x

Выводы:

- Время рендеринга растёт примерно пропорционально количеству треугольников, так как для каждого пикселя проверяется пересечение луча со всеми треугольниками сцены.
- GPU демонстрирует лучшее масштабирование: при увеличении сложности сцены в 27 раз (с 2 до 54 треугольников) время GPU выросло в 5.6 раза, тогда как время CPU - в 11.2 раза.
- Это объясняется тем, что GPU эффективнее скрывает латентность памяти при большом количестве вычислений благодаря параллельной обработке тысяч лучей одновременно.

Часть 3: Влияние ракурса камеры
Исследовано влияние позиции камеры на время рендеринга полной сцены (54 треугольника, 640×640).

Ракурс: Фронтальный вид (камера: 8.0, 4.5, 0.0)
GPU: 21.659649 ms

Ракурс: Вид сверху (камера: 0.0, 10.0, 8.0)
GPU: 21.231647 ms

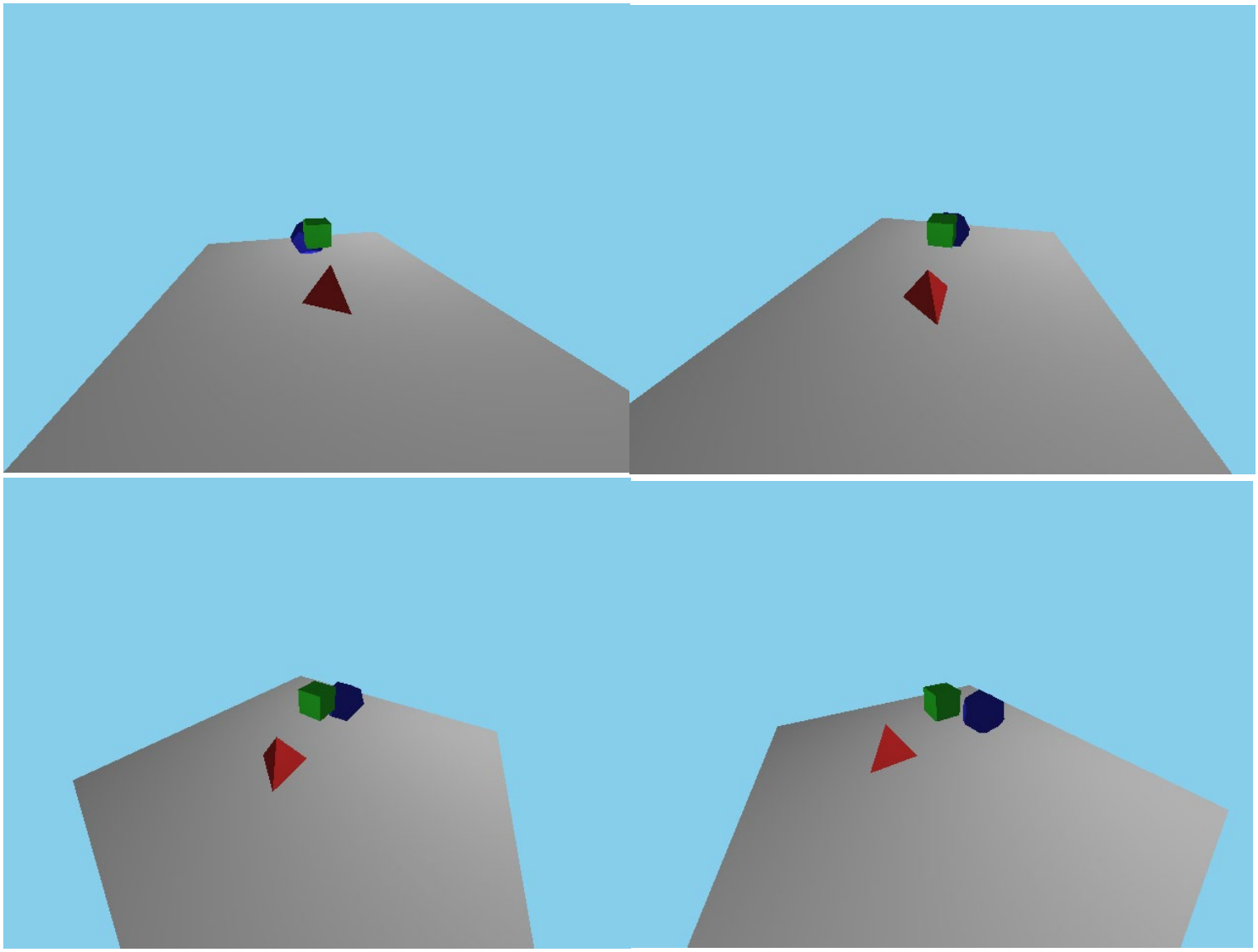
Ракурс: Близкий ракурс (камера: 3.0, 1.0, 3.0)
GPU: 22.373953 ms

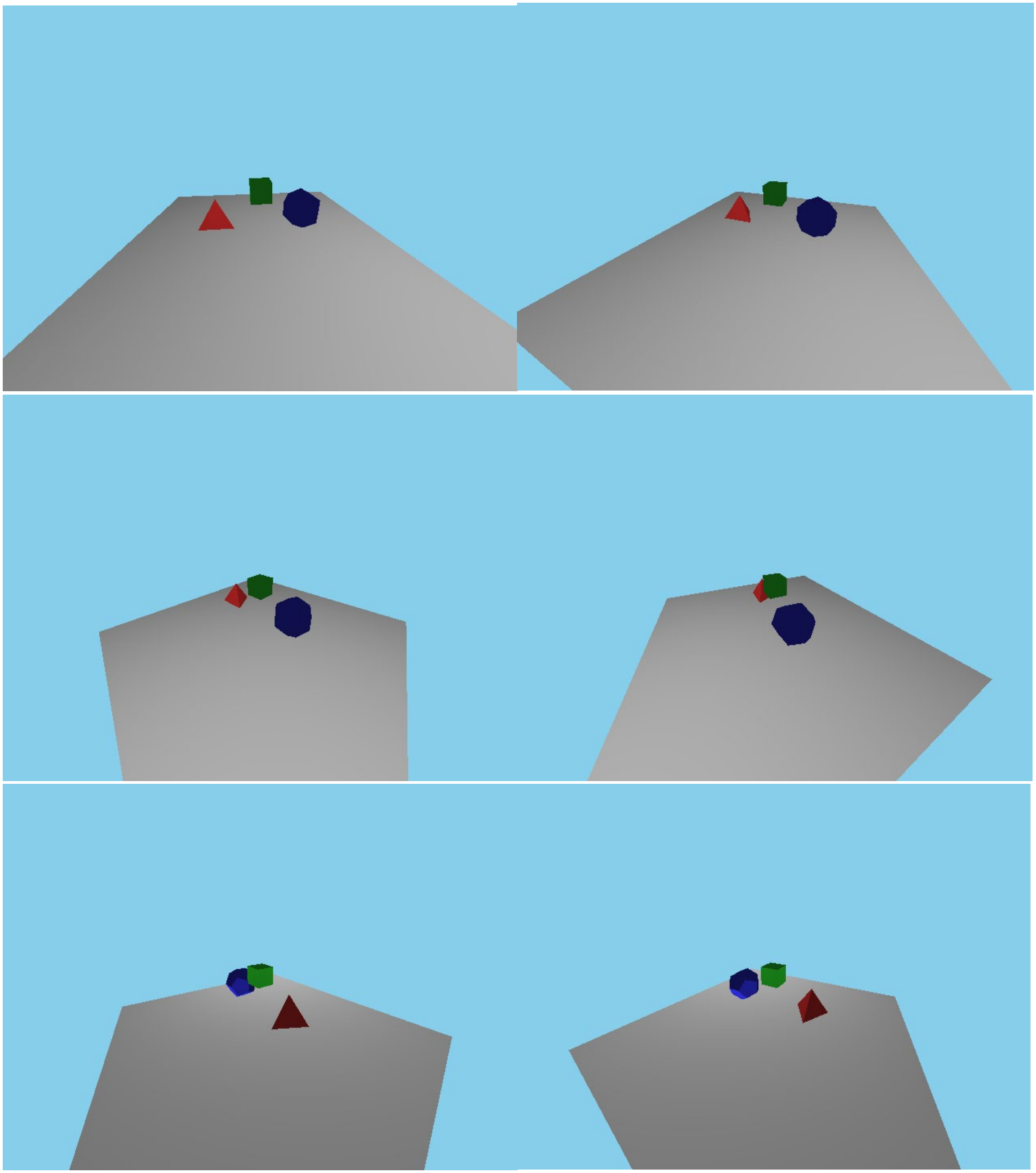
Ракурс: Дальний ракурс (камера: 15.0, 5.0, 15.0)
GPU: 20.622145 ms

Выводы:

- Ракурс камеры практически не влияет на производительность GPU (разброс ~8%).
- Небольшое различие объясняется тем, что при близких ракурсах объекты занимают большую часть кадра, увеличивая количество пикселей с пересечениями (меньше пикселей фона). При дальних ракурсах больше пикселей выходят на фон без вычисления освещения.
- Для алгоритма трассировки лучей без оптимизаций (BVH, пространственное разбиение) производительность определяется в первую очередь общим количеством треугольников, а не их расположением относительно камеры.

Часть 4. Визуальные результаты





Выводы

Реализованный алгоритм обратной трассировки лучей относится к классу методов фотореалистичного рендеринга и может применяться в тех областях, где важны корректное освещение, тени и реалистичное взаимодействие света с геометрией сцены. К типовым задачам относятся предварительный офлайн-рендеринг сцен в системах компьютерной графики, подготовка иллюстраций и анимаций для презентаций и обучающих материалов, а также исследование производительности GPU для вычислительно сложных графических алгоритмов. В учебном контексте такой алгоритм позволяет наглядно продемонстрировать преимущества параллельной обработки данных и особенности архитектуры графических процессоров.

В ходе работы были реализованы две версии алгоритма: последовательная, работающая на CPU, и параллельная, использующая вычислительные ресурсы GPU через CUDA. Это позволило не только получать изображения, но и проводить количественное сравнение времени построения кадра при разных разрешениях, конфигурациях ядра и сложностях сцены. По результатам замеров видно, что ускорение при переходе от CPU к GPU растёт вместе с числом пикселей и количеством треугольников в сцене: от нескольких раз на простых сценах и низких разрешениях до десятков тысяч раз на больших изображениях с полной сценой. Такое поведение объясняется тем, что с ростом размера задачи доля накладных расходов на запуск ядра становится малой, а сама задача хорошо распараллеливается по пикселям.

С точки зрения программирования алгоритм трассировки лучей средней сложности: геометрические вычисления (скалярные и векторные произведения, нормализация, проверка пересечения луча с треугольником) достаточно прямолинейны, но требуют аккуратного обращения с плавающей точкой и погрешностями. Наибольшую сложность представляет перенос кода на GPU: необходимо правильно организовать доступ к памяти, выбрать конфигурацию сетки и блоков, учесть ограничение на число потоков в блоке и подобрать разбиение по пикселям, чтобы загрузить устройство, не создавая лишних накладных расходов. Дополнительные трудности возникали при настройке среды (совместимость версии CUDA и драйверов) и при отладке ошибок запуска ядра, которые проявлялись только при определённых конфигурациях сетки.

Проведённые эксперименты подтвердили теоретические ожидания: при фиксированной конфигурации сцены время рендеринга на GPU практически линейно растёт с увеличением числа треугольников, а на CPU рост более крутой, что ведёт к увеличению ускорения GPU относительно CPU при усложнении сцены. Влияние конфигурации ядра хорошо заметно: слишком маленькое число блоков или потоков не позволяет полностью загрузить GPU, а чрезмерное число блоков приводит к росту накладных расходов на планирование, что ухудшает производительность. Отдельные эксперименты с различными ракурсами камеры показали, что при отсутствии рекурсивных эффектов (отражения, преломления) положение камеры влияет на время построения кадра значительно слабее, чем разрешение и число треугольников, так как

основная работа заключается в переборе пересечений луча со всеми примитивами сцены. Полученные результаты согласуются с теоретической моделью алгоритма и подтверждают целесообразность использования GPU для задач трассировки лучей даже в относительно простых сценах.