

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ)

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

Курсовой работа
по курсу «Параллельная обработка данных»

Обратная трассировка лучей (Ray Tracing) на GPU.

Студент: Попов И.

П. Группа:

М8О-406Б-20

Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2023

Условие

Цель работы. Использование GPU для создание фотореалистической визуализации.

Рендеринг полужеркальных и полупрозрачных правильных геометрических тел.

Получение эффекта бесконечности. Создание анимации.

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 перепотражения 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 анимацию любым доступным программным обеспечением. Подобрать параметры сцены, камеры и освещения таким образом, чтобы получить наиболее красочный.

На три балла: без рекурсии, без текстур, без отражений (без эффекта бесконечности), простые модели без отдельных ребер с источниками света, один источник света.

результат. Провести сравнение производительности гри и сри (т.е. дополнительно нужно реализовать алгоритм без использования CUDA).

Вариант 7. На сцене должны располагаться три тела: Гексаэдр, Октаэдр, Додекаэдр.

Входные параметры

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 $r_c^0, z_c^0, \varphi_c^0, A_c^r, A_c^z, \omega_c^r, \omega_c^z, \omega_c^\varphi, p_c^r, p_c^z$ и $r_n^0, z_n^0, \varphi_n^0, A_n^r, A_n^z, \omega_n^r, \omega_n^z, \omega_n^\varphi, p_n^r, p_n^z$.
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).

Программное и аппаратное обеспечение

NVIDIA GeForce GTX 1660 Super:

Compute capability: 7.5

Dedicated video memory: Typically 6 GB (may vary by manufacturer)

Shared memory per block: 49152 bytes

Register per block: 65536 bytes

Total constant memory: 65536 bytes

Max threads per multiprocessor: 2048

Max threads per block: 1024

CPU AMD Ryzen 3600X

Physical cores: 6

Threads: 12

Base clock speed: 3.8 GHz

Boost clock speed: 4.4 GHz

L1 cache: 384KB (per core)

L2 cache: 512KB (per core)

L3 cache: 32 MB (shared)

Chip lithography: 7 nm

16 Гб RAM

1 Тб HDD

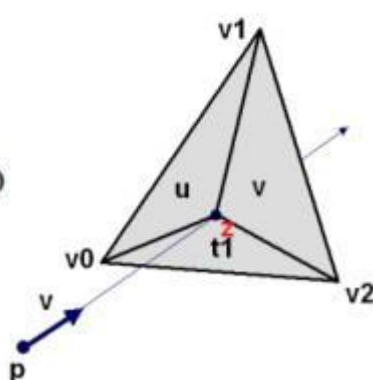
OS – Windows 11 ProWSL, IDE – VS Code, compiler - nvcc

Метод решения

Решение начинается с определения сцены, где, как сцена, так и все присутствующие объекты представлены в виде набора полигонов. Координаты вершин фигур и формирование самих полигонов осуществляются на основе математических формул для каждой из фигур.

Ray tracing - это метод компьютерной графики, который используется для симуляции освещения и создания визуально реалистичных изображений. Основная идея заключается в моделировании пути света, представленного в виде лучей, от источника света до камеры. Этот метод позволяет учитывать различные световые явления, такие как отражение, преломление и тени, что способствует созданию более реалистичных и эффектных изображений.

В контексте задачи, описанной выше, ray tracing применяется для определения того, какие объекты на сцене видны из точки зрения камеры, и каким образом свет взаимодействует с этими объектами. Путем трассировки лучей и их взаимодействия с полигонами на сцене достигается определение цвета каждого пикселя на изображении.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\text{dot}(P, E1)} * \begin{bmatrix} \text{dot}(Q, E2) \\ \text{dot}(P, T) \\ \text{dot}(Q, D) \end{bmatrix}$$
$$\begin{aligned} E1 &= v1 - v0 \\ E2 &= v2 - v0 \\ T &= p - v0 \\ P &= \text{cross}(D, E2) \\ Q &= \text{cross}(T, E1) \\ D &= v \end{aligned}$$


В данной реализации поддерживается лишь один источник освещения. Проверка наличия луча от этого источника до первого полигона для каждого луча определяет, находится ли точка в тени, что влияет на затемнение соответствующего пикселя.

Для сглаживания используется алгоритм SSAA (Super Sampling Anti-Aliasing). Изображение расширяется в несколько раз (например, в 4), происходит трассировка лучей для каждого пикселя расширенной картинки, и затем возвращение к исходному размеру, считая значение пикселя средним значением соседних пикселей на расширенной картинке.

Для повышения эффективности рендеринга и сглаживания применяются параллельные вычисления. Распараллеливание применяется как для этапа сглаживания, так и для рендеринга изображений. Однако стоит отметить, что распараллеливание трассировки лучей может быть сложным из-за необходимости эффективного управления параллельной обработкой нескольких полигонов одновременно.

Описание программы

В данной работе реализованы два ядра для вычислений на видеокарте: одно для параллельного рендеринга и другое для параллельного сглаживания. С целью сравнения производительности разработаны аналогичные функции, выполняемые на центральном процессоре.

В программе представлены следующие функции:

1. Определение полигонов для заданных геометрических тел
2. Обратная трассировка луча с учетом освещения

```
__host__ __device__ uchar4 ray(const vec3& position, const vec3&
direction, const vec3& lightPosition, const uchar4& lightColor, const
mesh* meshes, int meshesCount) {
    int closestIndex = -1;
    double closestIntersectionTime;

    for (int i = 0; i < meshesCount; i++) {
        double intersectionTime = rayPolyIntersect(position,
direction, meshes[i]);

        if (intersectionTime >= 0.0 && (closestIndex == -1 ||
intersectionTime < closestIntersectionTime)) {
            closestIndex = i;
            closestIntersectionTime = intersectionTime;
        }
    }

    if (closestIndex == -1) {
        return make_uchar4(0, 0, 0, 255);
    }

    vec3 intersectionPoint = direction * closestIntersectionTime +
position;

    if (!isPointShadowed(intersectionPoint, lightPosition, meshes,
meshesCount, closestIndex)) {
        return calculateShadedColor(meshes[closestIndex],
lightPosition, lightColor);
    } else {
        return make_uchar4(0, 0, 0, 255);
    }
}
```

3. Рендеринг изображения как на центральном процессоре, так и на графическом процессоре

```
__global__ void kernel_render(vec3 camPos, vec3 camView,
```

```

        int width, int height, double
fieldOfView, uchar4* resultData,
        vec3 lightPos, uchar4 lightColor,
        mesh* meshes, int meshesCount) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offsetx = blockDim.x * gridDim.x;
    int offsety = blockDim.y * gridDim.y;

    double deltaWidth = 2.0 / (width - 1.0);
    double deltaHeight = 2.0 / (height - 1.0);
    double z = 1.0 / tan(fieldOfView * M_PI / 360.0);

    vec3 camBasisX, camBasisY, camBasisZ;
    generateCameraBasis(camPos, camView, camBasisX, camBasisY,
camBasisZ);

    for (int i = idx; i < width; i += offsetx) {
        for (int j = idy; j < height; j += offsety) {
            vec3 rayDir = calculateRayDirection(i, j, width, height,
deltaWidth, deltaHeight, z, camBasisX, camBasisY, camBasisZ);
            resultData[(height - 1 - j) * width + i] = ray(camPos,
norm(rayDir), lightPos, lightColor, meshes, meshesCount);
        }
    }
}

```

4. Сглаживание как на центральном процессоре, так и на графическом процессоре

```

__global__ void kernel_ssaa(uchar4* data, uchar4* ssaaResData, int w,
int h, int sqrtRaysPerPixel) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offsetx = blockDim.x * gridDim.x;
    int offsety = blockDim.y * gridDim.y;

    int totalSamples = sqrtRaysPerPixel * sqrtRaysPerPixel;

    for (int x = idx; x < w; x += offsetx) {
        for (int y = idy; y < h; y += offsety) {
            uint4 accumulatedSamples = make_uint4(0, 0, 0, 0);
            for (int i = 0; i < sqrtRaysPerPixel; i++) {
                for (int j = 0; j < sqrtRaysPerPixel; j++) {
                    uchar4 p = data[w * sqrtRaysPerPixel * (y *
sqrtRaysPerPixel + j) + (x * sqrtRaysPerPixel + i)];
                    accumulatedSamples.x += p.x;
                    accumulatedSamples.y += p.y;
                    accumulatedSamples.z += p.z;

```

```

    }

    }

    accumulatedSamples.x /= totalSamples;
    accumulatedSamples.y /= totalSamples;
    accumulatedSamples.z /= totalSamples;

    ssaaResData[y * w + x] =
make_uchar4(accumulatedSamples.x, accumulatedSamples.y,
accumulatedSamples.z, 255);
    }

}
}

```

5. Вспомогательные математические операции над векторами
6. Получение входных данных для получения наиболее красочного ответа

Результаты

В рамках сравнения времени рендеринга одного кадра с использованием графического процессора и без него, мы будем анализировать одну и ту же сцену, изменяя лишь количество лучей (размерность изображения). Будем фиксировать время, затраченное исключительно на процесс рендеринга, не включая время, затраченное на копирование данных на/с видеокарты.

Этот подход позволит нам оценить эффективность графического процессора в зависимости от размерности изображения, выявив, насколько параллельные вычисления на GPU влияют на ускорение процесса рендеринга по сравнению с выполнением на CPU.

Разрешение изображения (количество пикселей)	Время рендеринга на GPU (мс)	Время рендеринга на CPU (мс)
1500	1.05645123	10.74534344
15000	3.01354681	68.25855345
150000	21.46541616	471.86844611
1500000	237.55463164	5565.53491644
15000000	1922.35468413	46568.03515373

Входные данные

```

123
res/%d.data
640 480 120

6.0 2.5 0.0      2.0 1.5      2.5 6.0 1.5      0.0 0.0
1.5 0.0 0.0      0.5 0.1      1.0 4.5 1.0      0.0 0.0

-0.5 0.0 2.0      1.0 0.1 0.1      1.75

```

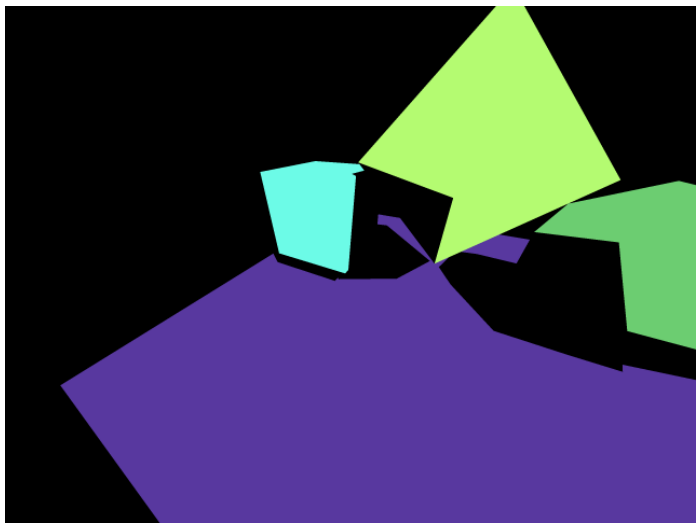
```
0.0 2.8 0.75    0.1 0.1 1.0    1.75
0.0 -2.8 0.5    0.1 1.0 0.1    1.5

-5.0 -5.0 -1.0    -5.0 5.0 -1.0    5.0 5.0 -1.0    5.0 -5.0 -1.0
0.576 0.91 0.467

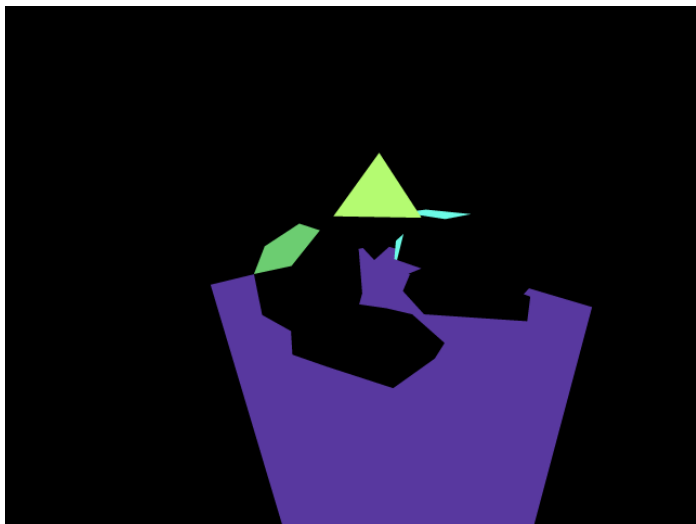
-10.0 -10.0 15.0    0.3 0.2 0.1

4
```

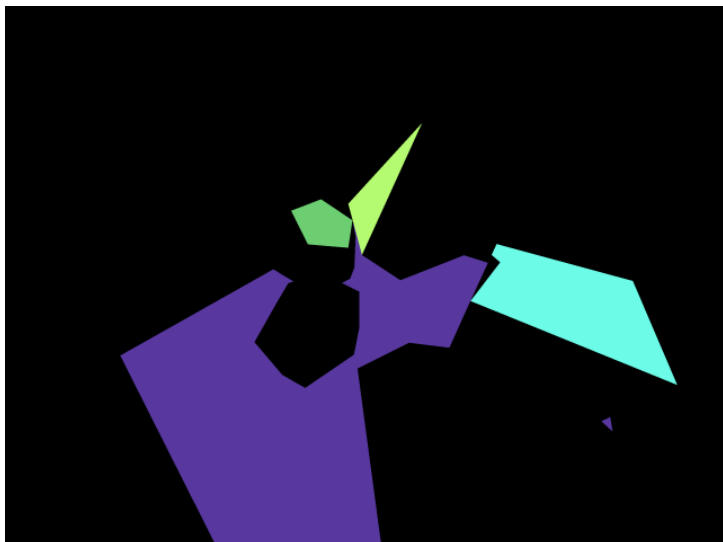
Кадр 41



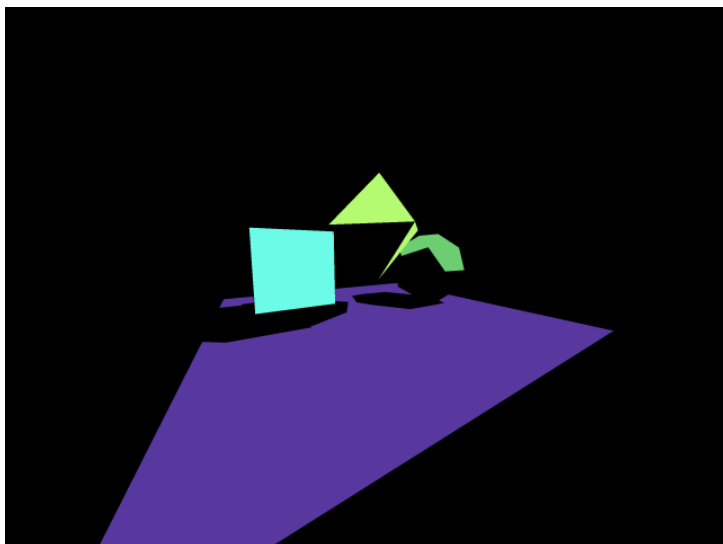
Кадр 71



Кадр 88



Кадр 116



Ссылка на gif-анимацию данной работы

https://github.com/Lunidep/PGP_POD/blob/main/cp/pgp.gif

Выводы

В ходе этой курсовой работы я изучил алгоритм обратной трассировки лучей и предпринял попытку самостоятельной реализации с использованием параллельных вычислений на графическом процессоре.

Этот алгоритм, несмотря на свою простую математику, широко применяется в создании мультимедийного контента, такого как игры и мультфильмы, так как позволяет учесть сложные световые взаимодействия, такие как отражение и преломление света, что существенно повышает качество и реализм визуализации.

В ходе исследования я убедился в значительном ускорении процесса рендеринга изображений при использовании GPU. Для подобных задач видеокарты становятся неотъемлемым инструментом, обеспечивая высокую эффективность вычислений.