МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Курсовой проект
по курсу «Программирование графических процессоров»

Обратная трассировка лучей (Ray Tracing). Технологии MPI, CUDA и OpenMP

Выполнил: И.Д. Черненко
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2022

Условие

Цель работы: Совместное использование технологии MPI, технологии CUDA и технологии OpenMP для создание фотореалистической визуализации. Создание анимации.

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 счет многократного переотражения 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 результат. Провести сравнение производительности gpu и cpu (т.е. дополнительно нужно реализовать алгоритм без использования CUDA).

Вариант 3: Тетраэдр, Гексаэдр, Икосаэдр

Программное и аппаратное обеспечение

Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
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 : 5

OS:Linux
compiler:nvcc
IDE:VS Code
Editor:nano

Метод решения

Все фигуры в сцене заданы с помощью треугольников, в качестве основной идеи используется вариант пересечения луча и полигона, при этом используется оптимизированный вариант из лекций

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{dot\ (P, E1)} * \begin{bmatrix} dot\ (Q, E2) \\ dot\ (P, T) \\ dot\ (Q, D) \end{bmatrix}$$

$$E1 = v1 - v0$$

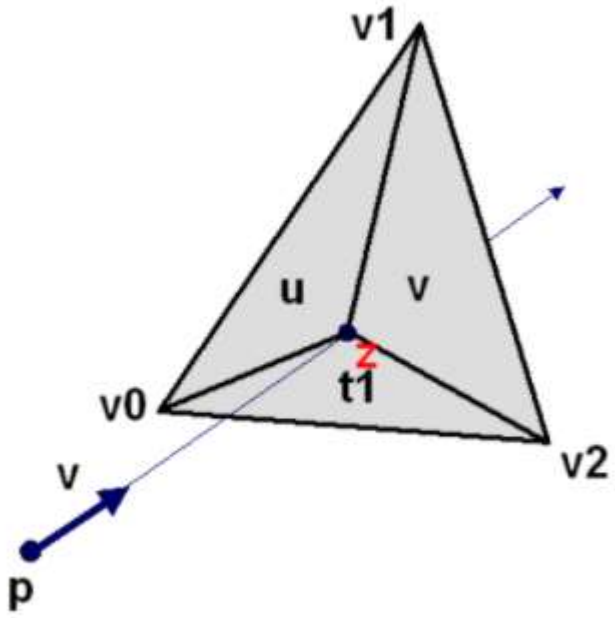
$$E2 = v2 - v0$$

$$T = p - v0$$

$$P = cross\ (D, E2)$$

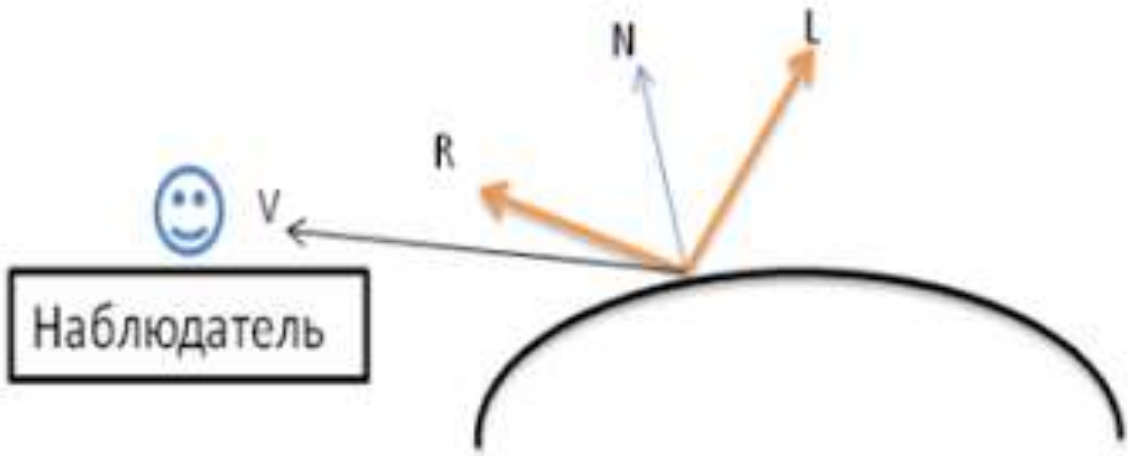
$$Q = cross\ (T, E1)$$

$$D = v$$



Тени

Для добавления в работу теней была использована модель освещения Фонга. При вычислении цвета точки выпускаем луч в направлении источника света, если на пути назад луч встретит объект, значит источник является тенью.



Отражения и прозрачность

Для добавления отражений и прозрачности используется примерно одинаковая логика для получения отражения/прозрачности нужно возвращать при встрече луча с объектом кроме цвета треугольника умноженного на коэффициент отражения/преломления, нужно ещё добавить цвет отраженного луча, который получается созданием ещё одного луча, но с направлением отражения, само отражение вычисляется через нормаль к треугольнику и направление базового луча.

Геометрические объекты

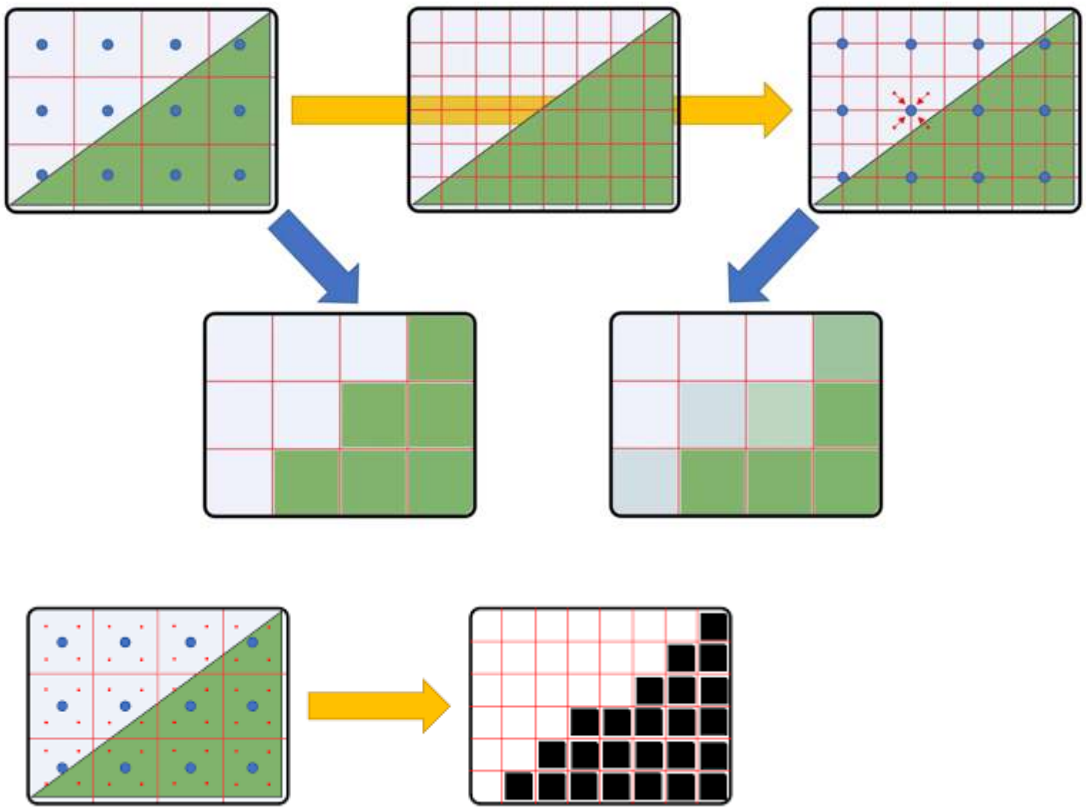
Реализация фигур далась очень легко так как их все изначально можно задать радиусом описанной окружности, поэтому точки всех фигур очень легко вычислялись исходя из формул с той же википедии. В целом каждая из координат точек каждой из фигур легко вычисляется по формуле:

$$\pm a * r/\sqrt{3} + center.x$$

Где a – возможный коэффициент, r – радиус, center - центр

SSAA

Для реализации сглаживания был использован один из простейших вариантов сглаживания, с помощью заданного коэффициента мы рендерим более обширную область картинки, а после этого значения пикселей усредняются по цвету. Есть различные вариации этой техники рознятся они только по методу выделения области



Описание программы

Вся программа была реализована в одном файле main.cu.

Рендер

```
__global__ void render_gpu(vec3 pc, vec3 pv, triangle* triangles, uchar4*
points, int width, int height, double angle, vec3 light_source, vec3
light_shade, int n, int recursion_step) {
    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 dw = 2.0 / (width - 1);
    double dh = 2.0 / (height - 1);
    double z = 1.0 / tan(angle * M_PI / 360.0);
    vec3 bz = norm(pv - pc);
    vec3 bx = norm(prod(bz, {0.0, 0.0, 1.0}));
    vec3 by = prod(bx, bz);
    for (int i = idx; i < width; i += offsetx) {
        for (int j = idy; j < height; j += offsety) {
            vec3 a = {-1.0 + dw * i, (-1.0 + dh * j) * height / width, z};
            vec3 dir = norm(mult(bx, by, bz, a));
            points[(height - 1 - j) * width + i] = ray(pc, dir, triangles,
light_source, light_shade, n, true, recursion_step); //меняем индексацию чтобы
не получить перевернутое изображение
        }
    }

}
```

Трассировка

```
__host__ __device__ uchar4 ray(vec3 pos, vec3 dir, triangle* triangles, vec3
light_source, vec3 light_shade, int n, bool shadows_reflections, int
recursion_step) {
    int k_min = -1;
    double ts_min;
    pos = pos + dir * 0.01; //фикс зернистости
    //uchar4 color_min = {0, 0, 0, 0};
    for (int i = 0; i < n; i++) {
        vec3 e1 = triangles[i].b - triangles[i].a;
        vec3 e2 = triangles[i].c - triangles[i].a;
        vec3 p = prod(dir, e2);
        double div = dot(p, e1);
        if (fabs(div) < 1e-10)
            continue;
        vec3 t = pos - triangles[i].a;
        double u = dot(p, t) / div;
        if (u < 0.0 || u > 1.0)
            continue;

        vec3 q = prod(t, e1);
        double v = dot(q, dir) / div;
        if (v < 0.0 || v + u > 1.0)
            continue;

        double ts = dot(q, e2) / div;
        if (ts < 0.0)
            continue;

        if (k_min == -1 || ts < ts_min) {
            k_min = i;
            ts_min = ts;
        }

    }

    if (k_min == -1) {
        return {0, 0, 0, 0};
    }

    if (shadows_reflections) {
        vec3 pos_tmp = dir * ts_min + pos;
        vec3 new_direction = norm(light_source - pos_tmp);
        for (int i = 0; i < n; i++) {
            vec3 e1 = triangles[i].b - triangles[i].a;
            vec3 e2 = triangles[i].c - triangles[i].a;
            vec3 p = prod(new_direction, e2);
```

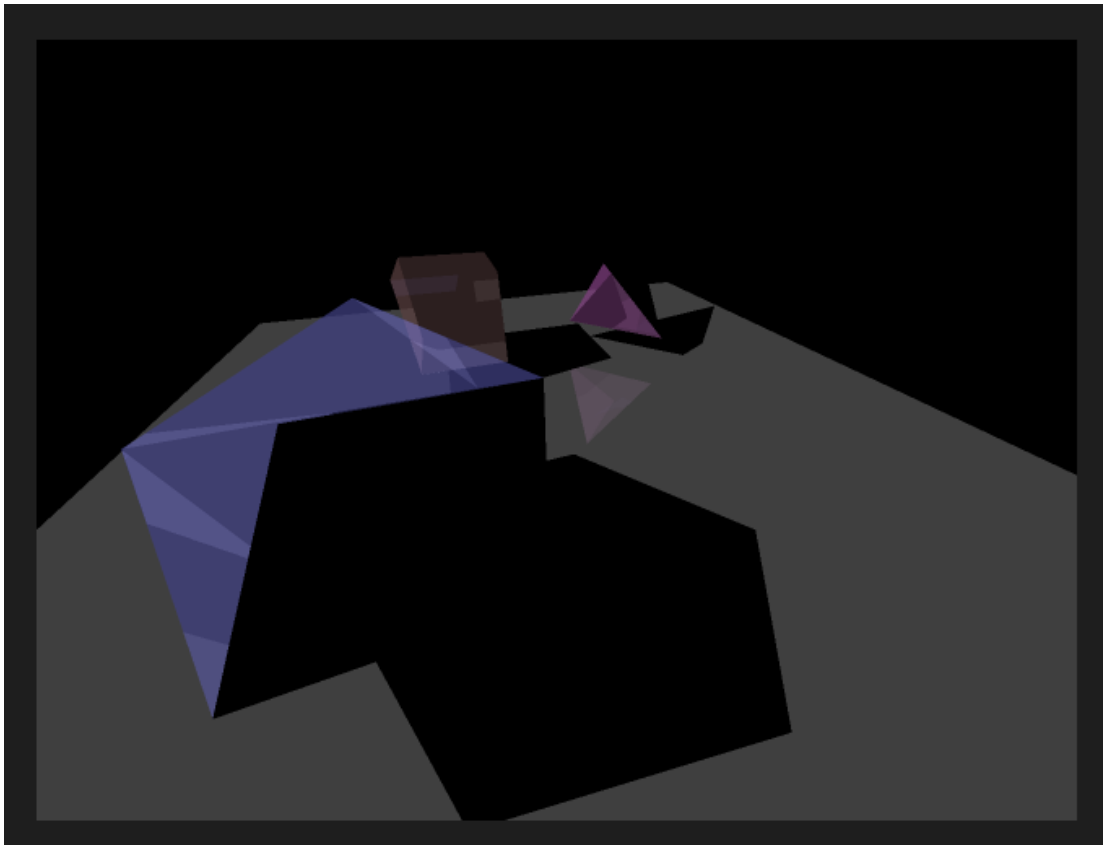
```
            double div = dot(p, e1);
            if (fabs(div) < 1e-10)
                continue;

            vec3 t = pos_tmp - triangles[i] .a;
            double u = dot(p, t) / div;

            if (u < 0.0 || u > 1.0)
                continue;
            vec3 q = prod(t, e1);
            double v = dot(q, new_direction) / div;
            if (v < 0.0 || v + u > 1.0)
                continue;
            double ts = dot(q, e2) / div;
            if (ts > 0.0 && ts < vector_length(light_source - pos_tmp) && i !=
k_min) {
                return {0, 0, 0, 0};
            }

        }
        uchar4 color_min = {0, 0, 0, 0};
        vec3 result = triangles[k_min].color;
        vec3 reflections = triangles[k_min].color;
        }
        if (recursion_step > 0) {
            vec3 reflection_dir = reflect(dir, norm(prod(triangles[k_min].b -
triangles[k_min].a, triangles[k_min].c - triangles[k_min].a)));
            double reflection_scale = 0.5;
            double transparency_scale = 0.5;
            reflections = (reflections * (1.0 - reflection_scale) +
to_vec3(ray(pos_tmp, reflection_dir, triangles, light_source, light_shade, n,
true, recursion_step - 1, texture, with_texture)) * reflection_scale);
            result = (reflections * (1.0 - transparency_scale) +
to_vec3(ray(pos_tmp, dir, triangles, light_source, light_shade, n, true,
recursion_step - 1, texture, with_texture)) * transparency_scale);
        }
            color_min.x += result.x* light_shade.x;
            color_min.y += result.y* light_shade.y;
            color_min.z += result.z* light_shade.z;
        }
        color_min.w = 0;
        return color_min;
    } else {
        return {220, 220, 220};
    }
}
```

Сглаживание

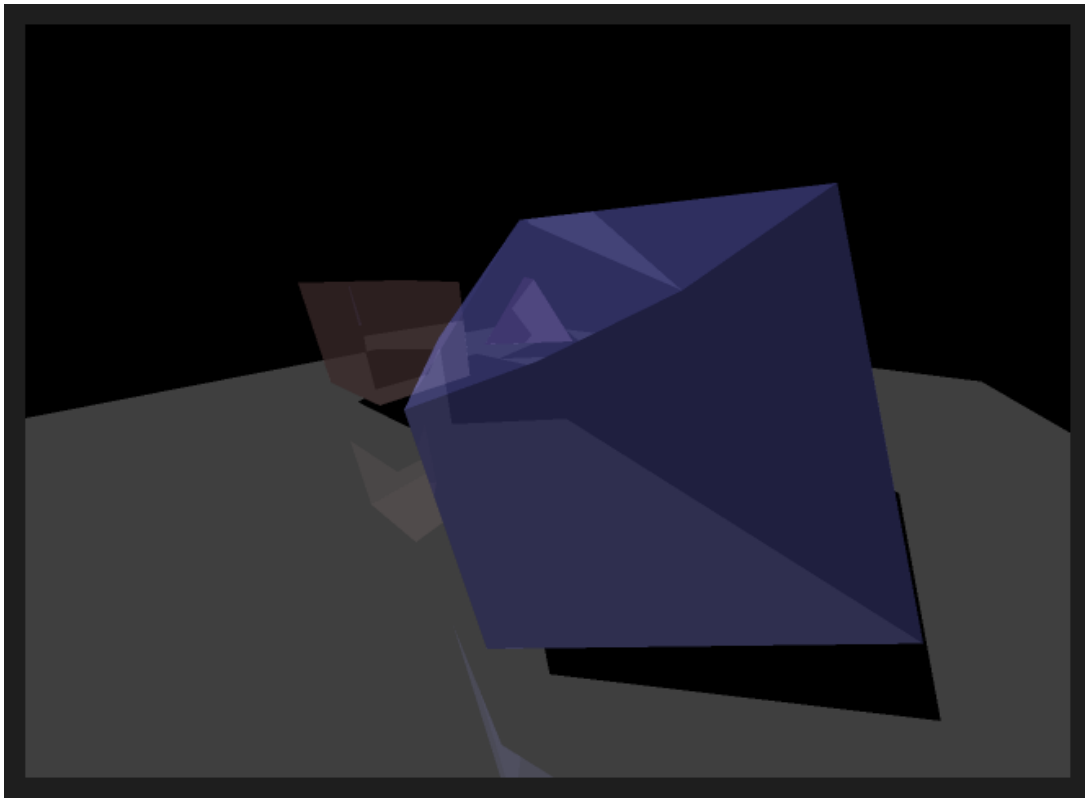
```
__global__ void smoothing_gpu(uchar4* points, uchar4* smoothing_points, int
width, int height, int multiplier) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int offsetx = blockDim.x * gridDim.x;
    int offsety = blockDim.y * gridDim.y;
    int multiplier2 = multiplier * multiplier;
    for (int y = idy; y < height; y += offsety) {
        for (int x = idx; x < width; x += offsetx) {
            vec3 mid = {0, 0, 0};
            for (int j = 0; j < multiplier; j++) {
                for (int i = 0; i < multiplier; i++) {
                    mid = mid + to_vec3(smoothing_points[i + j * width *
multiplier + x * multiplier + y * width * multiplier2]);
                }
            }
            points[x + width * y] = to_uchar4(mid / (multiplier2));
        }
    }
}
```

Результаты

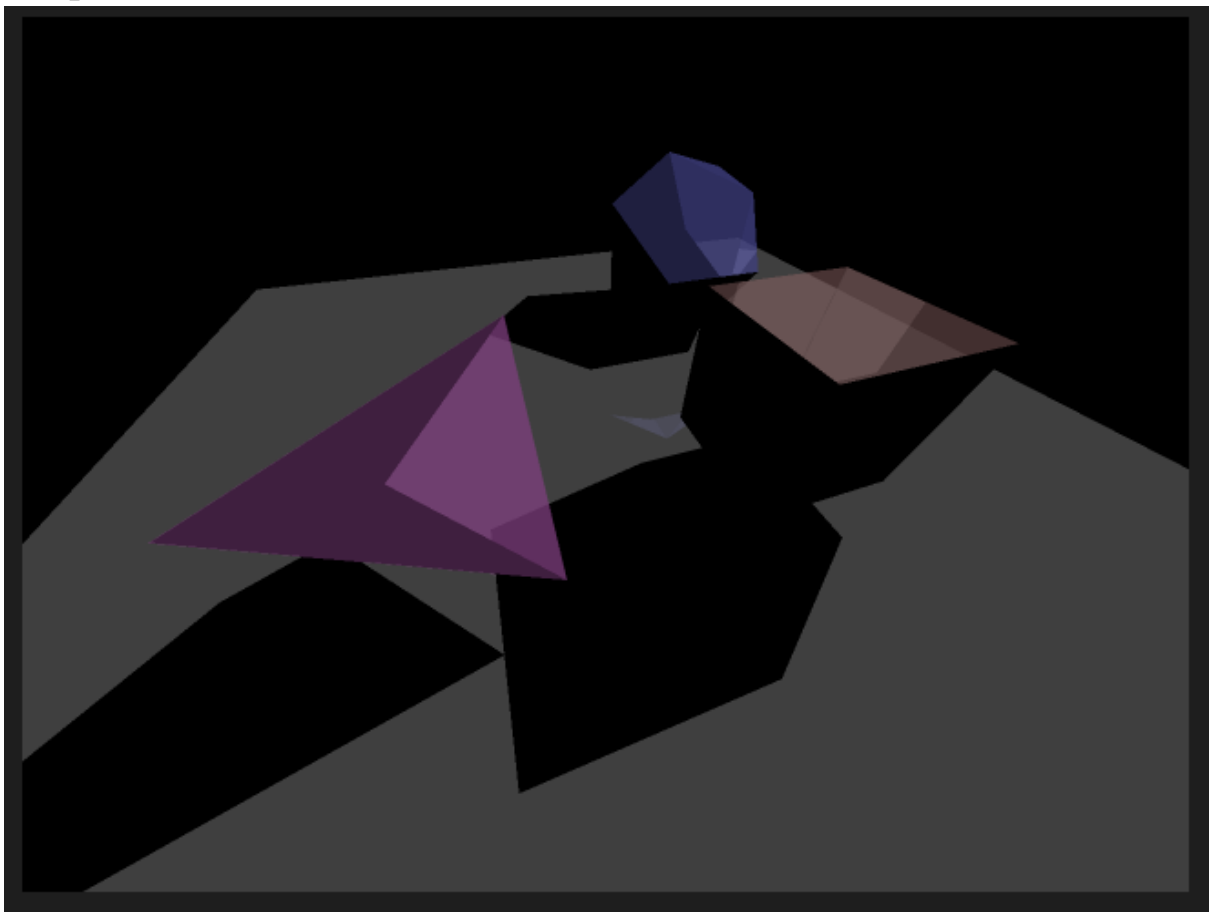
Кадр 1



Кадр 2



Кадр 3



Сравнение по времени

Количество лучей / SSAA	GPU 20	CPU 20	GPU 100	CPU 100
307200 / 1	4172ms	31313ms	21139ms	154658ms
2764800 / 3	21663ms	268928ms	109221ms	1.36534e+06ms

Количество лучей / SSAA	OpenMP + MPI 2, 20	OpenMP + MPI 4 , 20	MPI 2 + GPU, 20	MPI 4 + GPU, 20
307200 / 1	17108ms	9715ms	4130ms	4028ms
2764800 / 3	137217ms	71901ms	21450ms	20305ms

Количество лучей / SSAA	OpenMP + MPI 2, 100	OpenMP + MPI 4, 100	MPI 2 + GPU, 100	MPI 4 + GPU, 100
307200 / 1	83672ms	46542ms	20929ms	19917ms
2764800 / 3	700242ms	368535ms	108382ms	99087ms

Как можно заметить вычисления с технологией MPI дали значительный прирост производительности на GPU, примерно в 5 раз. При применении MPI и OpenMP прирост производительности на CPU заметен не так сильно, что на больших данных приводит к уменьшению времени в 3 - 4 раза.

Выводы

Технология трассировки лучей позволяет строить различные изображения с трехмерными объектами с реализацией отражений и теней благодаря распространения лучей “обратно” от объекта к источнику. При этом у данной технологии есть множество достоинств и недостатков, основным достоинством является возможность эффективного распараллеливания вычислений, что отчетливо видно в тестах производительности, но при этом одним из важных недостатков является перерасчет цвета пикселя для каждого луча что сильно бьет по производительности техники в целом. При этом конкретно в данном проекте, благодаря использованию MPI и OpenMP, удалось достичь значительного улучшения в производительности причем как на GPU, так и на CPU, благодаря использованию в том числе нескольких “процессов”.