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*). Технологии *MPI*, *CUDA* и *OpenMP*

Студент: Ваньков Д. А.
Группа: М8О-407Б-17
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы. Совместное использование технологии *MPI*, технологии *CUDA* и технологии *OpenMP* для создания фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание видеоролика.

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:

$$r_c(t) = r_c^0 + A_c^r \sin(\omega_c^r \cdot t + p_c^r);$$
$$z_c(t) = z_c^0 + A_c^z \sin(\omega_c^z \cdot t + p_c^z);$$
$$\varphi_c(t) = \varphi_c^0 + \omega_c^\varphi t;$$
$$r_n(t) = r_n^0 + A_n^r \sin(\omega_n^r \cdot t + p_n^r);$$
$$z_n(t) = z_n^0 + A_n^z \sin(\omega_n^z \cdot t + p_n^z);$$
$$\varphi_n(t) = \varphi_c^0 + \omega_c^\varphi t,$$

где $t \in [0, 2\pi]$.

Требуется реализовать алгоритм обратной трассировки лучей (http://www.ray-tracing.ru/) с использованием технологии *CUDA* и *OpenMP*. Выполнить покадровый рендеринг сцены. Для устранения эффекта «зубчатости», выполнить сглаживание (например, с помощью алгоритма *SSAA*). Полученный набор кадров склеить в видеоролик любым доступным программным обеспечением. Подобрать параметры сцены, камеры и освещения таким образом, чтобы получить наиболее красочный результат. Провести сравнение производительности *cpu* (*OpenMP*) и *gpu* (*CUDA*).

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

Программное и аппаратное обеспечение

Graphics card: GeForce 940M

Размер глобальной памяти: 4242604032

Размер константной памяти: 65536

Размер разделяемой памяти: 49152

Максимальное количество регистров на блок: 65536

Максимальное количество потоков на блок: 1024

Количество мультипроцессоров: 3

OS: Linux Ubuntu 18.04

Редактор: CLion, Atom

Машины в кластере:

1. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 1050, 2 Gb

2. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GT 545, 3 Gb

3. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 650, 2 Gb

4. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 12 Gb, GeForce GT 530, 2 Gb

5. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 8 Gb, GeForce GT 530, 2 Gb

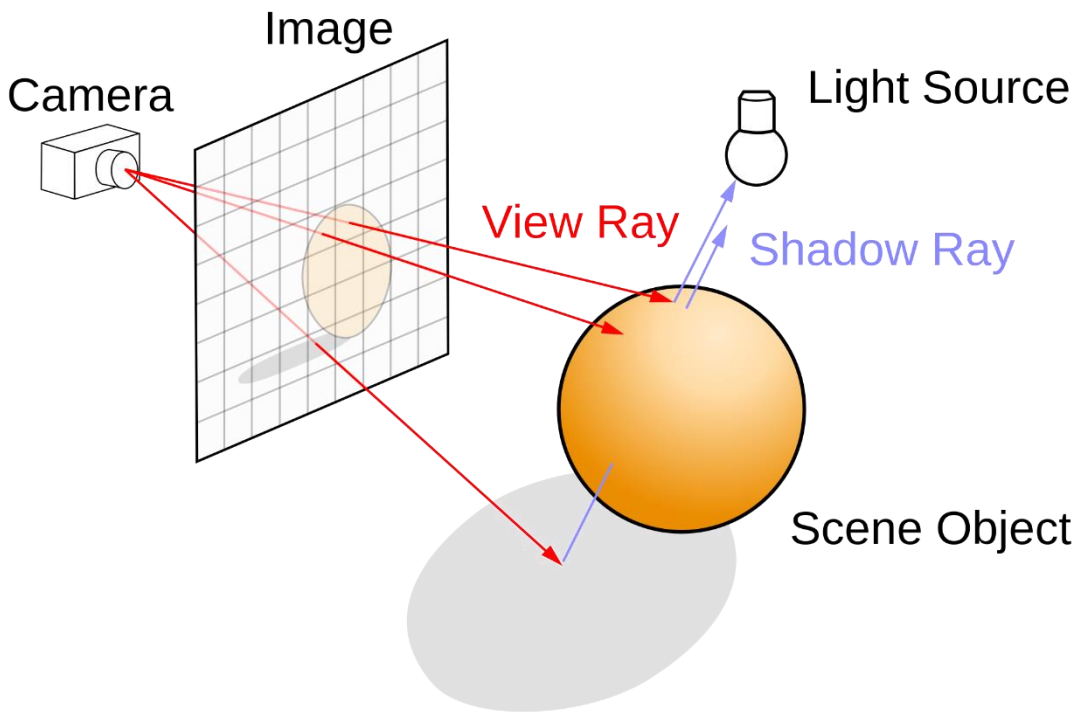
Все машины соединены гигабатным ethernet и находятся в подсети 10.10.1.1/24.

Версии софта: mpirun 1.10.2, g++ 4.8.4, nvcc 7.0

Алгоритм решения

Алгоритм трассировки

Количество лучей равно размеру экрана, умноженного на квадрат коэффициента алгоритма SSAA. Так как у меня уровень рекурсии нулевой, то есть, лучи не отражаются после попадания, все лучи, запускаются из проекционного экрана на построенную сцену и затем происходит проверка на пересечение полигонов объектов лучами. Если луч пересекает полигон на сцене отображается фигура. Тень от объекта вычисляется следующим образом: происходит проверка на пересечение луча, запущенного из точки в камеру, с фигурой. Если такое пересечение есть, данная точка считается тенью. Пример работы алгоритма визуально:



Модель освещения

В качестве модели освещения используется модель Фонга. Затенение по Фонгу — это модель расчёта освещения трёхмерных объектов, в том числе полигональных моделей и примитивов, а также метод интерполяции освещения по всему объекту.

Расчёт освещения по Фонгу требует вычисления цветовой интенсивности трёх компонент освещения: фоновой (ambient), рассеянной (diffuse) и глянцевых бликов (specular). Фоновая компонента — грубое приближение лучей света, рассеянных соседними объектами и затем достигших заданной точки; остальные две компоненты имитируют рассеивание и отражение прямого излучения. В виде формулы это выглядит следующим образом:

$$I = K_a I_a + K_d(\vec{n}, \vec{l}) + K_s(\vec{n}, \vec{h})^p$$

где

$\vec{n}$ — вектор нормали к поверхности в точке

$\vec{l}$ — падающий луч (направление на источник света)

$\vec{h}$ — отраженный луч (направление идеально отраженного от поверхности луча)

$\vec{h} = 2(\vec{l} * \vec{n})\vec{n} - \vec{l}$

K_a — коэффициент фонового освещения

K_s — коэффициент зеркального освещения

K_d — коэффициент диффузного освещения

Построение тел

Гексаэдр

Для построения куба необходимо было вычислить длину его ребра, которая рассчитывается из радиуса. Так как куб вписанный, длину ребра можно взять из диагонали: *a = 2 * r * sqrt(3)*. Затем нумерую вершины, вычисляю их координаты имея центр и диагональ и разбиваю каждую грань на полигоны. После чего в зависимости от параметра центра фигуры осуществляю его корректировку на сцену.

Октаэдр

Нумеруем вершины этого многогранника в системе координат с центром в этой фигуре описываются согласно:

$$(\pm 1,0,0),$$
$$(0,\pm 1,0),$$
$$(0,0,\pm 1).$$

Далее, необходимо умножить каждую из этих координат на коэффициент масштабирования или радиус описанной сферы. После чего в зависимости от параметра центра фигуры осуществляю его корректировку на сцену.

Додекаэдр

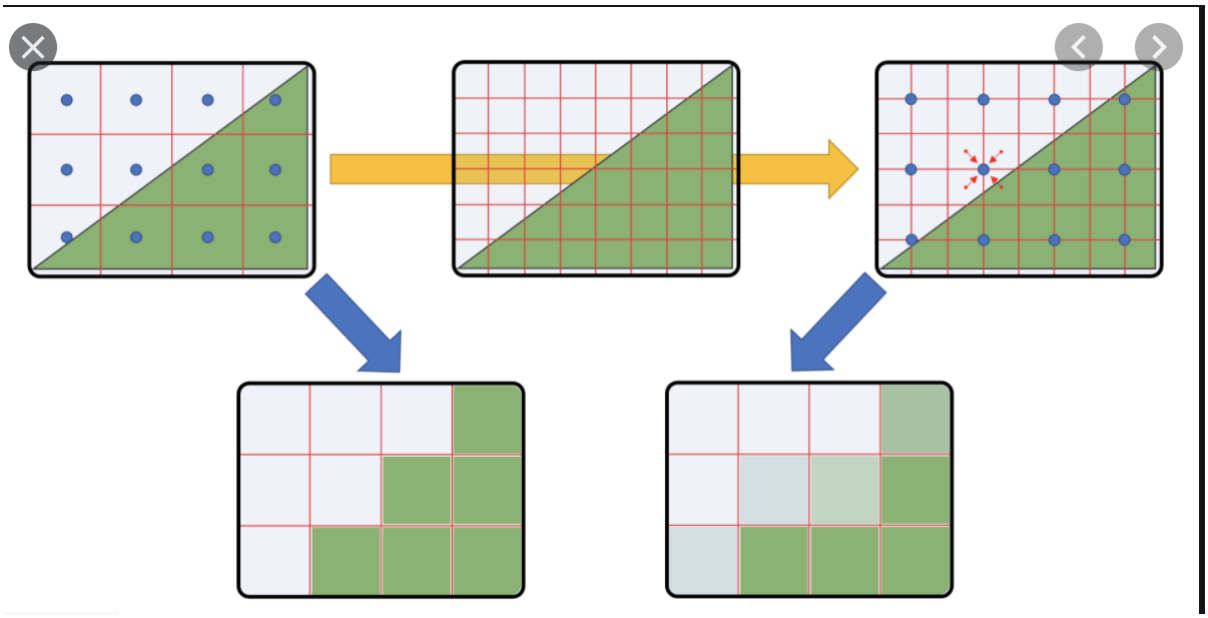
Эта фигура оказалась самой сложной из-за большого количества полигонов и вершин. Она имеет 20 вершин и 12 граней -- пятиугольников, которые необходимо разбить на 3 треугольника. Для этого вычисляется длина ребра по формуле:

$$R = \frac{a}{4}(1+\sqrt{5})\sqrt{3},$$

Где a — это сторона, а R — радиус. Затем нумеруются вершины согласно правилу обхода и в цикле для каждой грани осуществляется обход фигуры. После чего в зависимости от параметра центра фигуры осуществляю его корректировку на сцену.

Алгоритм SSAA

Этот алгоритм работает следующим образом — с помощью коэффициента вычисляется размер новой области и производится рендер этой большей области. Затем, так как необходимо получить область заданного параметра мы проходим по уже отрендеренной большей области и берем среднее значения цветов пикселей и ставим их в соответсвующее место в меньшей области, которое рассчитывается исходя из коэффициента. Пример работы визуально:



Данный алгоритм позволяет уменьшить эффект альясинга или зубчатости на границах фигур, то есть происходит сглаживание.

Работа с MPI и OpenMP

Для распараллеливания с использованием технологии OpenMP я выбрал функцию рендеринга объектов на cpu, а также алгоритм SSAA. Технология MPI используется для того, чтобы распараллелить рендеринг кадров если выбран режим gpu. В этом случае я считываю данные на 0-ом процессе, затем с помощью функции MPI_Bcast() передаю информацию всем остальным процессам. Каждый из процессов берет себе определенное количество кадров и рендерит их независимо от остальных используя уже написанное ядро для рендера на CUDA. Затем каждый из них записывает отредеренный кадр в существующую дирректорию.

Описание программы

Рендеринг

```
__global__ void render_gpu(vector_cords p_c, vector_cords p_v, int w, int h, double fov, uchar4* pixels,
                           vector_cords light_pos, vector_cords light_col, polygon* polygons, int n) {
    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 = (double)2.0 / (double)(w - 1.0);
    double dh = (double)2.0 / (double)(h - 1.0);
    double z = 1.0 / tan(fov * M_PI / 360.0);
    vector_cords b_z = norm(p_v - p_c);
    vector_cords b_x = norm(crossing(b_z, {0.0, 0.0, 1.0}));
    vector_cords b_y = norm(crossing(b_x, b_z));
    for (int i = idx; i < w; i += offsetX)
        for (int j = idy; j < h; j += offsetY) {
            vector_cords v;
            v.x = (double)-1.0 + dw * (double)i;
            v.y = ((double)-1.0 + dh * (double)j) * (double)h / (double)w;
            v.z = z;
            vector_cords dir = multiply(b_x, b_y, b_z, v);
            pixels[(h - 1 - j) * w + i] = ray_aux(p_c, norm(dir), light_pos, light_col, polygons, n);
        }
}
```

Ray Tracing

```
__host__ __device__ uchar4 ray_aux(vector_cords pos, vector_cords dir, vector_cords light_pos,
                                   vector_cords light_color, polygon *polygons, int n) {
    int min_value = -1;
    double ts_min;
    for (int i = 0; i < n; ++i) {
        vector_cords e1 = polygons[i].p2 - polygons[i].p1;
        vector_cords e2 = polygons[i].p3 - polygons[i].p1;
        vector_cords p = crossing(dir, e2);
        double div = scal_mul(p, e1);

        if (fabs(div) < 1e-10)
            continue;

        vector_cords t = pos - polygons[i].p1;
        double u = scal_mul(p, t) / div;
        if (u < 0.0 || u > 1.0)
            continue;

        vector_cords q = crossing(t, e1);
        double v = scal_mul(q, dir) / div;
        if (v < 0.0 || v + u > 1.0)
            continue;

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

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

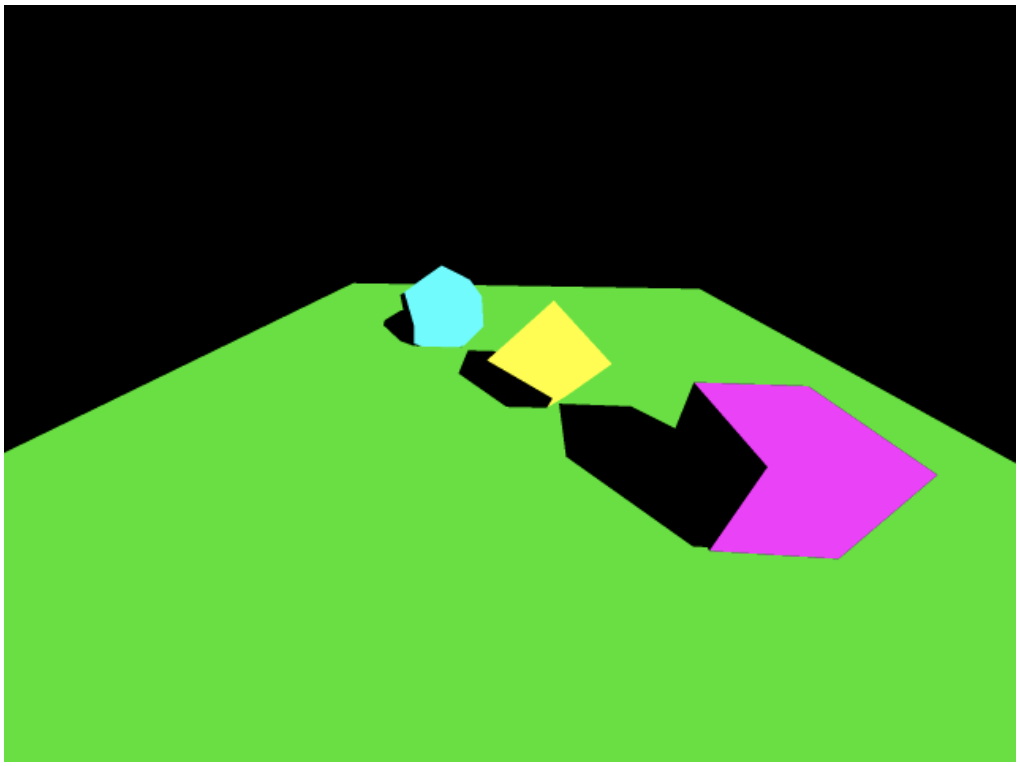
```
for (int i = 0; i < n; i++) {
    vector_cords e1 = polygons[i].p2 - polygons[i].p1;
    vector_cords e2 = polygons[i].p3 - polygons[i].p1;
    vector_cords p = crossing(dir, e2);
    double div = scal_mul(p, e1);

    if (fabs(div) < 1e-10)
        continue;

    vector_cords t = pos - polygons[i].p1;
    double u = scal_mul(p, t) / div;

    if (u < 0.0 || u > 1.0)
        continue;

    vector_cords q = crossing(t, e1);
    double v = scal_mul(q, dir) / div;

    if (v < 0.0 || v + u > 1.0)
        continue;

    double ts = scal_mul(q, e2) / div;

    if (ts > 0.0 && ts < length && i != min_value) {
        return {0, 0, 0, 0};
    }
}

uchar4 color_min;
color_min.x = polygons[min_value].color.x;
color_min.y = polygons[min_value].color.y;
color_min.z = polygons[min_value].color.z;

color_min.x *= light_color.x;
color_min.y *= light_color.y;
color_min.z *= light_color.z;
```

SSAA

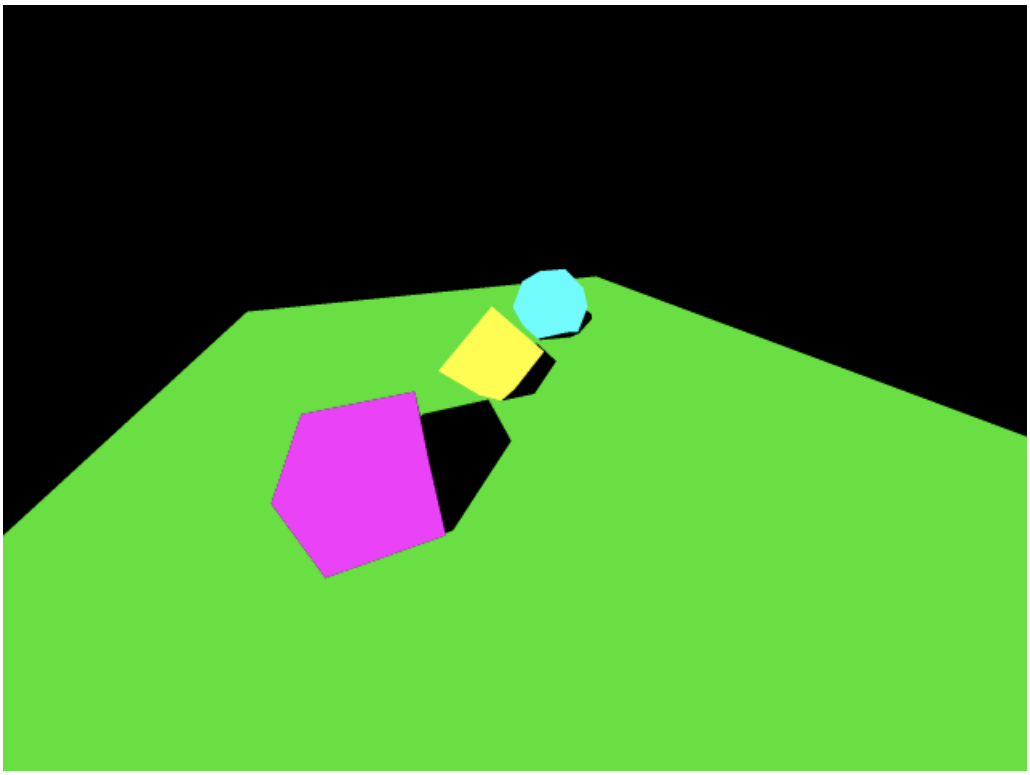
```
__global__ void ssaa_gpu(uchar4 *pixels, int w, int h, int coeff, uchar4 *ssaa_pixels) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int offsetX = blockDim.x * gridDim.x;
    int offsetY = blockDim.y * gridDim.y;

    for (int y = idy; y < h; y += offsetY) {
        for (int x = idx; x < w; x += offsetX) {
            int4 mid = { 0, 0, 0, 0 };
            for (int j = 0; j < coeff; j++) {
                for (int i = 0; i < coeff; i++) {
                    int index = y * w * coeff * coeff + x * coeff + j * w * coeff + i;
                    mid.x += ssaa_pixels[index].x;
                    mid.y += ssaa_pixels[index].y;
                    mid.z += ssaa_pixels[index].z;
                    mid.w += 0;
                }
            }
            pixels[y * w + x].x = (uchar)(mid.x / (coeff * coeff));
            pixels[y * w + x].y = (uchar)(mid.y / (coeff * coeff));
            pixels[y * w + x].z = (uchar)(mid.z / (coeff * coeff));
            pixels[y * w + x].w = 0;
        }
    }
}
```

Результат работы

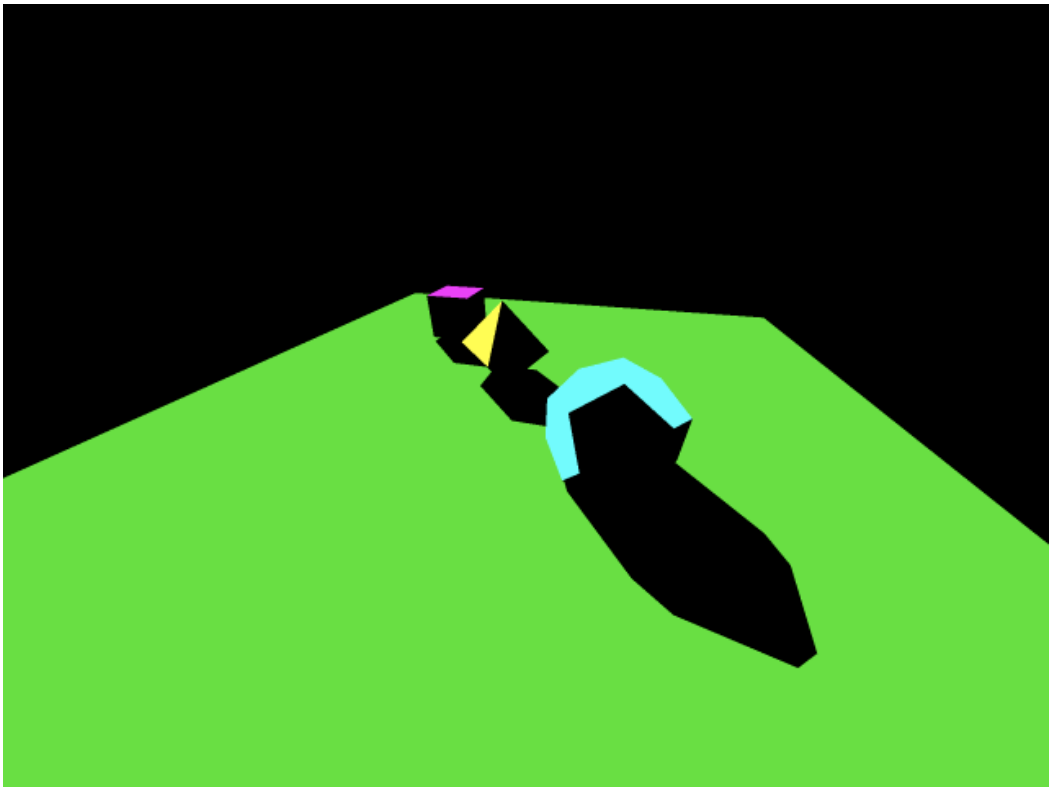
Кадр 1



Кадр 19



Кадр 54



Результаты сравнений

100 frames

Rays number	gpu	cpu	gpu + mpi 1 proc	gpu + mpi 2 proc	gpu + mpi 5 proc	gpu + mpi 10 proc
640x480x3x3	57335.9 ms	962612.9 ms	57167ms	57094ms	56535.6ms	57271.1ms

10 frames

Rays number	gpu	cpu	gpu + mpi 1 proc	gpu + mpi 2 proc	gpu + mpi 5 proc	gpu + mpi 10 proc	Omp
640x480x6x6	21829 ms	367832 ms	22019.1 ms	21814.45 ms	21887.58 ms	21943 ms	322829 ms

Исходя из полученных данных можно заметить, что присутствует рост производительности, однако после количества процессов равного 10 -- рост замедляется. Также можно заметить прирост в производительности с

использованием OpenMP относительно простого запуска cpu. Полученные результаты отличаются не очень сильно из-за того, что набор данных был не большим. Такие сравнения необходимо проводить на данных оченб большого размера.

Выводы

Трассировка лучей — это технология построения изображения трёхмерных моделей в компьютерных программах, при которых отслеживается обратная траектория распространения луча (от экрана к источнику). Данная технология имеет как свои достоинства, так и недостатки. Во-первых, он дает возможность рендеринга гладких объектов без аппроксимации их полигональными поверхностями (например, треугольниками). Во-вторых, вычислительная сложность метода слабо зависит от сложности сцены и также высокая алгоритмическая распараллеливаемость вычислений — то есть можно параллельно и независимо трассировать два и более лучей, разделять участки (зоны экрана) для трассирования на разных узлах кластера.
Одним из серьезных недостатков метода обратного трассирования является его производительность. Данный метод трассирования лучей каждый раз начинает процесс определения цвета пикселя заново, рассматривая каждый луч наблюдения в отдельности.

Трассировка сейчас применяется для создания компьютерных игр, если быть точнее для создания реалистичного освещения, отражений и теней, обеспечивающее более высокий уровень реализма по сравнению с традиционными способами рендеринга. Однако игры, поддерживающие трассировку лучей, требуют очень хорошего железа, так как происходит очень много вычислений.

Объединение технологий MPI, CUDA и OpenMP позволяет добиваться лучшей производительности, однако нужно осторожно пользоваться ими. Из-за того, что CUDA использует копирование данных с devic-а на host, время работы существенно замедляется, необходимо понимать, что будет выгоднее рассчитать на cpu или осуществить 2 этапа копирования и произвести расчеты на gpu. Также нельзя бездумно увеличивать количество процессов для MPI, так как они попросту начнут мешать друг-другу считать. При использовании OpenMP не нужно пытаться распараллелить

все циклы, а постараться проанализировать необходимость распараллеливания.