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

Выполнил: М.А.Жерлыгин
Группа: 8О-408Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Использование GPU для создания фотореалистичной визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание видеоролика.

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 выполняет облет сцены согласно определенным законам. В цилиндрических координатах (r, φ, z) положение и точка направления камеры в момент времени t определяется следующим образом:

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

Вариант 9: Гексаэдр, Додекаэдр, Икосаэдр

Программное и аппаратное обеспечение

GPU:

1. Compute capability: 7.5;
2. Графическая память: 4294967296;
3. Разделяемая память: 49152;
4. Константная память: 65536;
5. Количество регистров на блок: 65536;
6. Максимальное количество блоков: (2147483647, 65535, 65535);
7. Максимальное количество нитей: (1024, 1024, 64);
8. Количество мультипроцессоров: 6.

Сведения о системе:

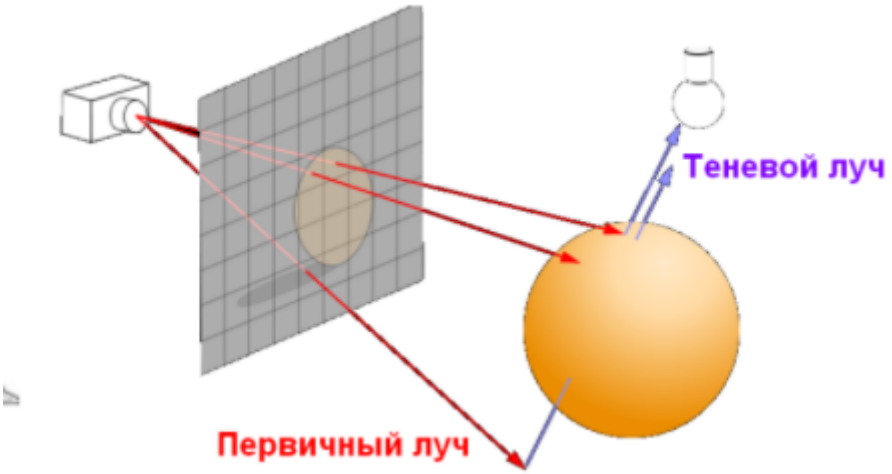
1. Процессор: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
2. Память: 16,0 ГБ;
3. HDD: 237 ГБ.

Программное обеспечение:

1. OS: Windows 10;
2. IDE: CLion 2021.3;
3. Компилятор: nvcc.

Метод решения

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.



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.

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

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

SSAA.
Этот алгоритм работает следующим образом —с помощью коэффициента вычисляется размер новой области и производится рендер этой большой 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 соответствующее место в меньшей области, которое рассчитывается исходя из коэффициента.

Тела.

Икосаэдр.
Эта фигура имеет 12 вершин и 20 граней.
Если выровнять икосаэдр и посмотреть на него под правильным углом, то можно заметить, что две его вершины лежат на одной оси друг под другом, а остальные расположены на двух окружностях.
На каждой окружности их по пять штук, а значит интервал между ними 72 градуса. Смещение между окружностями — 36 градусов. Для выравнивания вершин нам опять понадобится волшебный угол из Википедии: «If two vertices are taken to be at the north and south poles (latitude ±90°), then the other ten vertices are at latitude ±arctan(1/2) ≈ ±26.57°». В переводе на русский это означает, что волшебный угол — арктангенс одной второй.
Затем в зависимости от параметров центра корректируем его положение относительно сцены.

Додекаэдр.
Эта фигура 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.

Гексаэдр.
Для построения куба необходимо было вычислить длину его ребра, которая рассчитывается из радиуса. Так как куб вписанный, длину ребра можно взять из диагонали: $a = 2r\sqrt{3}$. Затем нумеруем вершины, вычисляем их координаты имея центр и диагональ и разбиваем каждую грань на полигоны. После чего в зависимости от параметра центра фигуры осуществляем его корректировку на сцену.

Описание программы

Рендеринг:

```
__global__ void gpu_render(vec3 pc, vec3 pv, int w, int h, double angle,
uchar4* data, vec3 l_position, vec3 l_color,
                           triangle *trigs, int rays_sqrt) {
   int id_x = blockDim.x * blockIdx.x + threadIdx.x;
   int id_y = blockDim.y * blockIdx.y + threadIdx.y;
   int offset_x = blockDim.x * gridDim.x;
   int offset_y = blockDim.y * gridDim.y;

   // из примера с лекций
   double dw = 2.0 / (w - 1.0);
   double dh = 2.0 / (h - 1.0);
   double z = 1.0 / tan(angle * M_PI / 360.0);
   vec3 b_z = normalize(pv - pc);
   vec3 b_x = normalize(prod(b_z, {0.0, 0.0, 1.0}));
   vec3 b_y = normalize(prod(b_x, b_z));
   for (int i = id_x; i < w; i += offset_x)
       for (int j = id_y; j < h; j += offset_y) {
           vec3 v = {-1.0 + dw * i, (-1.0 + dh * j) * h / w, z};
           vec3 dir = mult(b_x, b_y, b_z, v);
             data[(h - 1 - j) * w + i] = ray(pc, normalize(dir), l_position,
l_color, trigs, rays_sqrt);
       }
}
```

Ray tracing:

```
__host__ __device__ uchar4 ray(vec3 pos, vec3 dir, vec3 l_position, vec3
l_color, triangle *trigs, int rays_sqrt) {
   // взято из примера с лекций
   int k_min = -1;
   double ts_min;
   for (int k = 0; k < rays_sqrt; k++) {
       vec3 e1 = trigs[k].b - trigs[k].a;
       vec3 e2 = trigs[k].c - trigs[k].a;
       vec3 p = prod(dir, e2);
       double div = p * e1;
       if (fabs(div) < 1e-10)
           continue;
       vec3 t = pos - trigs[k].a;
       double u = (p * t) / div;
       if (u < 0.0 || u > 1.0)
           continue;
       vec3 q = prod(t, e1);
       double v = (q * dir) / div;
       if (v < 0.0 || v + u > 1.0)
           continue;
       double ts = (q * e2) / div;
       if (ts < 0.0)
           continue;
       if (k_min == -1 || ts < ts_min) {
```

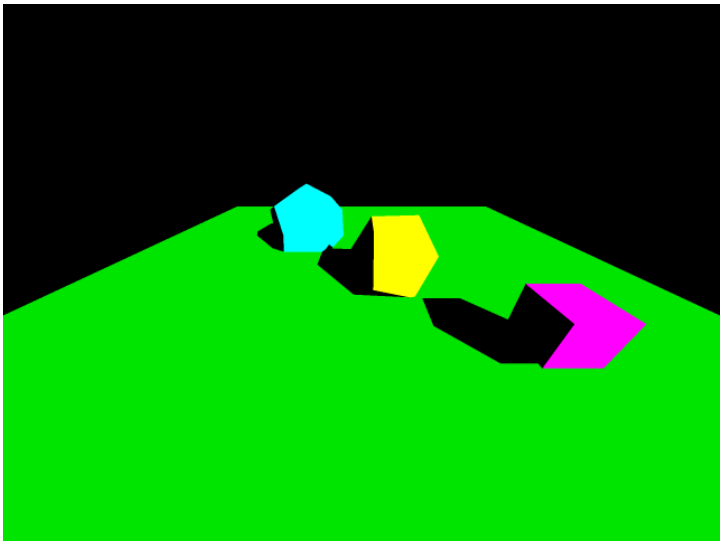
```
            k_min = k;
            ts_min = ts;
        }
    }
    if (k_min == -1)
        return {0, 0, 0, 0};
    pos = dir * ts_min + pos;
    dir = l_position - pos;

    double size = sqrt(dir * dir);

    dir = normalize(dir);
    for (int k = 0; k < rays_sqrt; k++) {
        vec3 e1 = trigs[k].b - trigs[k].a;
        vec3 e2 = trigs[k].c - trigs[k].a;
        vec3 p = prod(dir, e2);
        double div = p * e1;
        if (fabs(div) < 1e-10)
            continue;
        vec3 t = pos - trigs[k].a;
        double u = (p * t) / div;
        if (u < 0.0 || u > 1.0)
            continue;
        vec3 q = prod(t, e1);
        double v = (q * dir) / div;
        if (v < 0.0 || v + u > 1.0)
            continue;
        double ts = (q * e2) / div;
        if (ts > 0.0 && ts < size && k != k_min) {
            return {0, 0, 0, 0};
        }
    }

    uchar4 color_min;
    color_min.x = trigs[k_min].color.x;
    color_min.y = trigs[k_min].color.y;
    color_min.z = trigs[k_min].color.z;

    color_min.x *= l_color.x;
    color_min.y *= l_color.y;
    color_min.z *= l_color.z;
    color_min.w = 0;
    return color_min;
}
```

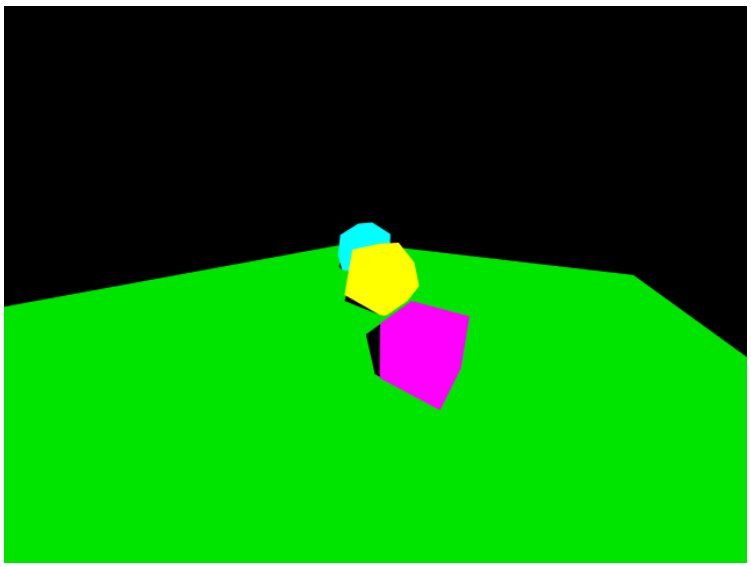
SSAA:

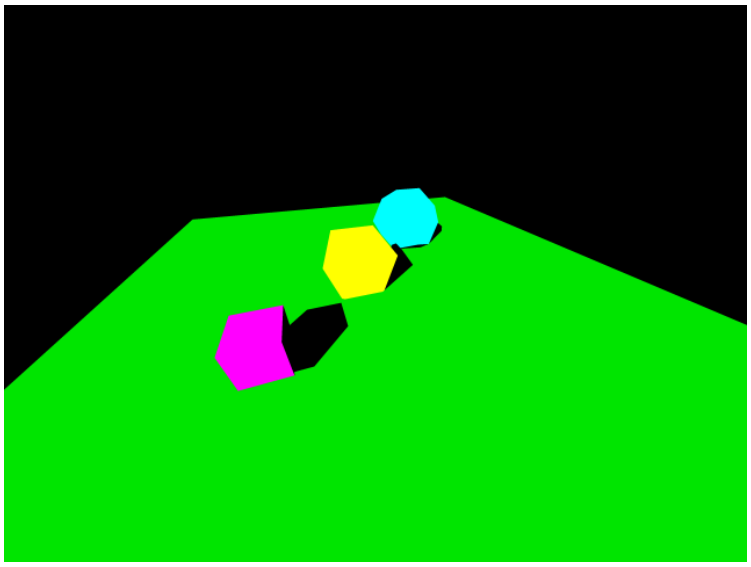
```
__global__ void ssaa_gpu(uchar4 *data, uchar4 *out_data, int w, int h, int k)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int idy = blockIdx.y * blockDim.y + threadIdx.y;
   int offsetX = blockDim.x * gridDim.x;
   int offsetY = blockDim.y * gridDim.y;

   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 < k; j++) {
               for (int i = 0; i < k; i++) {
                   int index = k * k * y * w + k * j * w + k * x + i;
                   mid.x += data[index].x;
                   mid.y += data[index].y;
                   mid.z += data[index].z;
               }
           }

           double div = k * k;
            out_data[x + y * w] = make_uchar4(mid.x / div, mid.y / div, mid.z
/ div, 0);
       }
   }
}
```

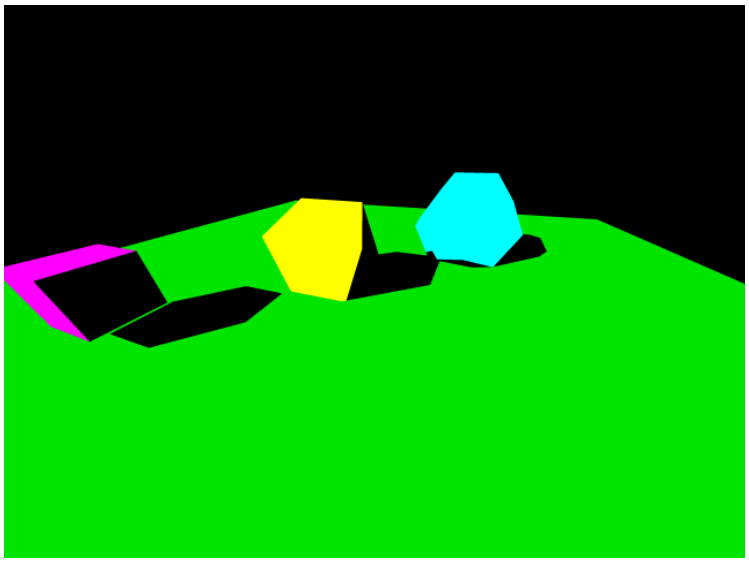
Исследовательская часть и результаты

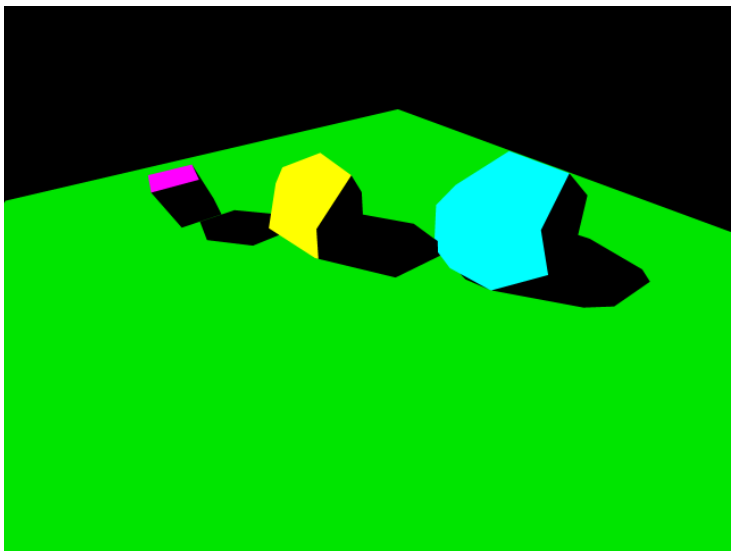
Примеры результата (10 кадров):

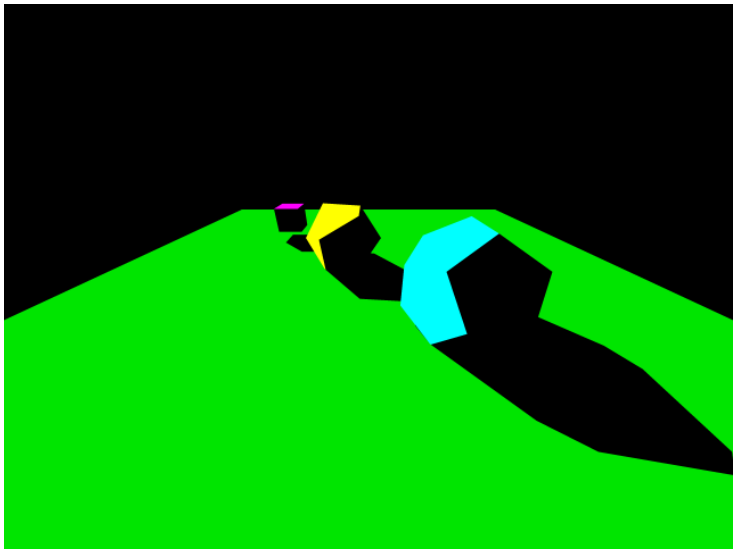


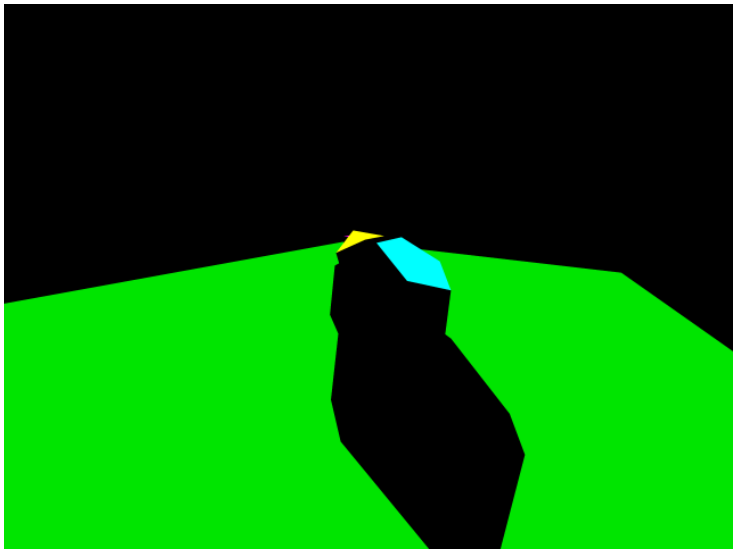


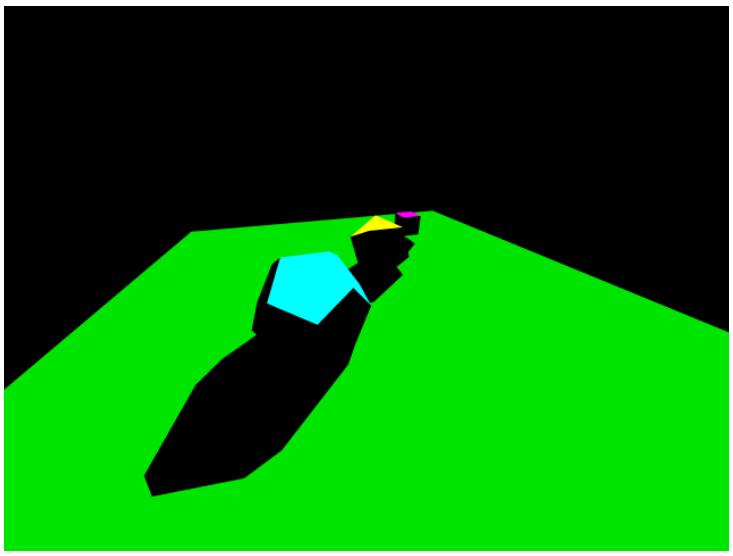


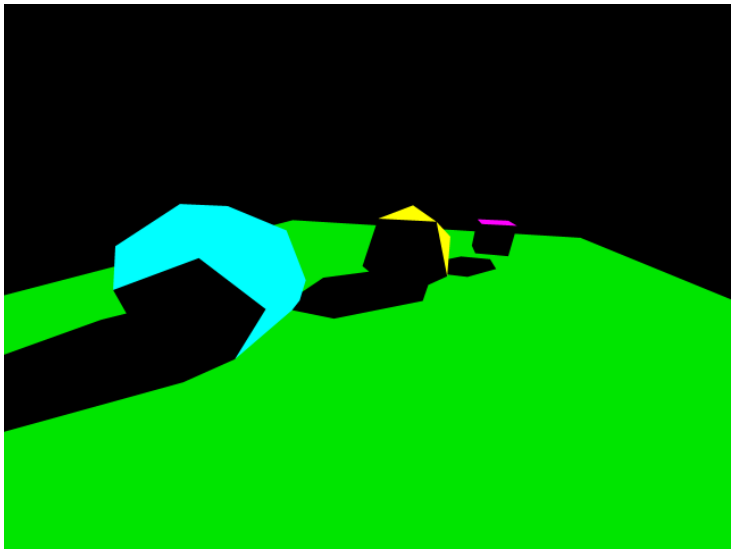


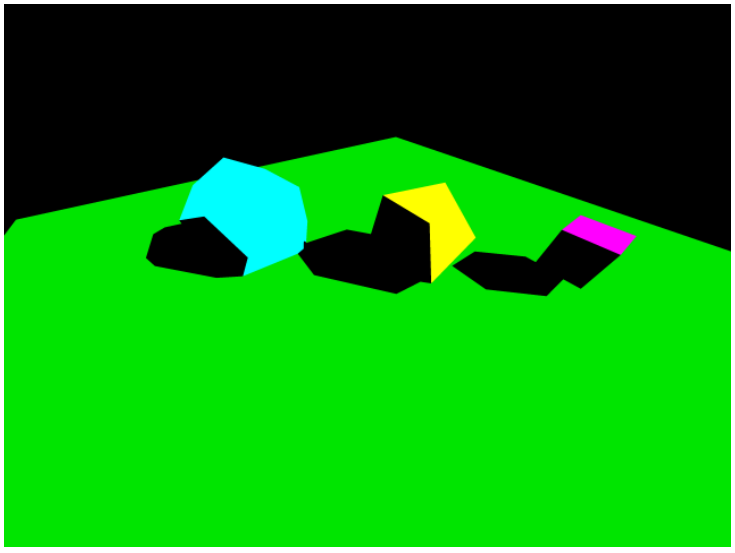












Наиболее красочное изображение достигается при следующих входных данных:
10
./out/img_%d.data
640 480 120
7.0 3.0 0.0 2.0 1.0 2.0 6.0 1.0 0.0 0.0
2.0 0.0 0.0 0.5 0.1 1.0 4.0 1.0 0.0 0.0
4.0 4.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0
1.0 1.0 0.0 1.0 1.0 0.0 1.5 0.0 0.0 0.0
-2.5 -2.5 0.0 0.0 1.0 1.0 1.75 0.0 0.0 0.0
-10.0 -10.0 -1.0 -10.0 10.0 -1.0 10.0 10.0 -1.0 10.0 -10.0 -1.0 temp 0.0 0.9 0.0 0.5
1
100 100 100 1.0 1.0 1.0
1 6

Rays	GPU 10 frames time, ms	CPU 10 frames
640 x 480 x 3 x 3	7822	297197
640 x 480 x 6 x 6	25835	510240
640 x 480 x 9 x 9	60694	-

Исходя из полученных данных в очередной раз можно убедиться, что сложные вычисления лучше всего производить на gpu, так как это гораздо быстрее.

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, она 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.
Также трассировка применяется для создания фотореалистичных изображений (чем мы и занимались в данной работе).

Литература

1. GPU Ray Tracing Vladimir Frolov (Nvidia, MSU, Keldysh Institute of Applied Math)
2. Процедурная генерация трёхмерных моделей (url) - https://habr.com/ru/post/194620/