МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Курсовая работа
по курсу «Параллельная обработка данных»

Обратная трассировка лучей (Ray Tracing) на GPU.

Выполнил: А.С. Федоров
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2022

Условие

Цель работы. Использование GPU для создание фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание анимации.
Вариант задания. 5. Тетраэдр, Октаэдр, Икосаэдр.

Программное и аппаратное обеспечение

Графический процессор
Название: NVIDIA GeForce GTX 1050
Compute capability: 6.1
Объем графической памяти: 2147352576 байтов
Объем разделяемой памяти на блок: 49152 байтов
Объем регистров на блок: 65536
Размер варпа: 32
Максимальное количество потоков на блок: (1024, 1024, 64)
Максимальное число блоков: (2147483647, 65535, 65535)
Объем постоянной памяти: 65536 байтов
Число мультипроцессоров: 5

Процессор
Название: i5-8250U
Базовая тактовая частота процессора: 1,60 ГГц
Количество ядер: 4
Количество потоков: 8
Кеш L1: 256 Кб
Кеш L2: 1 Мб
Кеш L3: 6 Мб

Оперативная память
Тип: DDR4
Объем: 11.9 Гб
Частота: 2400 МГц

Програмное обеспечение
ОС: WSL2 (Windows 11)
IDE: Microsoft Visual Studio 2022 (аддон NVIDIA Nsight)
Компилятор: nvcc

Метод решения

Алгоритм обратной трассировки лучей заключается в сборе информации со сцены путем просчета лучей из камеры. На каждый пиксель камеры выпускается луч (или несколько лучей) в направлении от камеры. Сталкиваясь с различными объектами на сцене возможно вычислить цвет поверхности, ее освещенность или затененность, глянцевость, отражающую и пропускающую способность. Все это отражает в себе материал грани, с которой столкнулся луч. При столкновении с гранью луч может породить дочерние лучи. Возможно два варианта дочерних лучей: отраженный, чья позиция – место попадания родительского луча, а направление – отраженное относительно грани направление родительского луча и сквозной, чья позиция - место попадания родительского луча на грань, направление такое-же как и у родителя. Лучи могут порождать дочерние, пока не обнулится счетчик переотражений или же интенсивность отраженного луча будет ниже минимального порога. Лучи, пущенные из камеры, имеют максимальную интенсивность, интенсивности дочерних лучей вычисляются относительно интенсивности родительского и свойств материала. Каждый луч камеры порождает дерево дочерних лучей. Проход по этому дереву и агрегирование информации из детей относительно их цвета и интенсивности вернет в луч камеры результирующий цвет требуемого пикселя. Совокупность таких построенных деревьев и их обход позволяет построить изображение.

Описание программы

Для трассировки лучей было написано две реализации: на GPU и на CPU.

Реализация на GPU

Для реализации параллельной трассировки лучей выполняется следующий алгоритм:

- Трассировка лучей, которые нужно просчитать
- Сжатие массива с лучами
- Перевыделение памяти под новый массив и копирование туда элементов

Данная процедура выполняется до тех пор, пока после трассировки новых дочерних лучей больше не появится. Затем осуществляется обратный проход по массиву лучей с записью цветов детей в родителей. Обратный проход начинается с самых глубоких лучей и за каждую итерацию поднимается на один уровень выше. Данные о лучах камеры копируются на CPU, где происходит формирование и запись в файл изображения.

Ядро трассировки:

```
__global__ void trace_rays_kernel(Ray* rays, bool* rays_bitmap, uint
rays_traced_number, uint rays_to_trace_number, Object* objects, uint objects_number,
PointLight* lights, uint lights_number, Textured_floor* floor) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    while (idx < rays_to_trace_number){
        float min_t = INF;
        Material hit_material;
        vec3 hit_normal;
        bool hit_bevel = false;
        bool no_shadow = false;
        uint hit_object_id;

        for (uint object_id = 0; object_id < objects_number; ++object_id){
            // грани
            for (uint polygon_id = 0; polygon_id < objects[object_id].faces_number;
++polygon_id){
                float t = is_ray_hits_polygon(rays[rays_traced_number+idx],
objects[object_id].faces[polygon_id]);
                if (min_t > t){
                    min_t = t;
                    hit_material = objects[object_id].face_material;
                    hit_normal = objects[object_id].faces[polygon_id].Normal();
                    hit_bevel = false;
                }
            }
            // ребра
            for (uint polygon_id = 0; polygon_id < objects[object_id].bevels_number;
++polygon_id){
                float t = is_ray_hits_polygon(rays[rays_traced_number+idx],
objects[object_id].bevels[polygon_id]);
                if (min_t > t){

                    min_t = t;
                    hit_material = objects[object_id].bevel_material;
                    hit_normal = objects[object_id].bevels[polygon_id].Normal();
                    hit_bevel = true;
                    hit_object_id = object_id;
                }
            }
            // крышки
            for (uint polygon_id = 0; polygon_id < objects[object_id].caps_number;
++polygon_id){
```

```
                float t = is_ray_hits_polygon(rays[rays_traced_number+idx],
objects[object_id].caps[polygon_id]);
                if (min_t > t){
                    min_t = t;
                    hit_material = objects[object_id].bevel_material;
                    hit_normal = objects[object_id].caps[polygon_id].Normal();
                    hit_bevel = false;
                }
            }
        }

        vec3 hit_color;
        vec3 hit_p = add(rays[rays_traced_number+idx].p, mult(min_t,
rays[rays_traced_number+idx].d));

        // bool hit_floor = false;
        if (min_t == INF){
            // столкновение с полом
            uint hit_polygon_id;
            float u, v;
            for (uint polygon_id = 0; polygon_id < 2; ++polygon_id){
                float t = is_floor_ray_hits_polygon(u, v,
rays[rays_traced_number+idx], floor[0].dev_polygons[polygon_id]);

                if (min_t > t){
                    min_t = t;
                    hit_material = floor[0].floor_material;
                    hit_normal = floor[0].dev_polygons[polygon_id].Normal();
                    hit_polygon_id = polygon_id;
                }
            }

            // если попал
            if (min_t != INF){
                // hit_floor = true;
                hit_color = floor[0].Get_color(u, v, hit_polygon_id);
            }
        }
        else if (hit_bevel){
            hit_color = hit_material.color;
            // printf("%d\n", hit_object_id);
            for (uint light_id = 0; light_id < objects[hit_object_id].lights_number;
++light_id){
                if (dot(hit_normal, rays[rays_traced_number+idx].d) > 0.0)
                    hit_normal = mult(-1.0, hit_normal);
```

```
                if (dot(hit_normal, diff(hit_p, objects[hit_object_id].p)) < 0.0
                    && objects[hit_object_id].lights_radius > length(diff(hit_p,
objects[hit_object_id].dev_lights[light_id]))){
                    hit_color = objects[hit_object_id].lights_material.color;
                    hit_material = objects[hit_object_id].lights_material;
                    no_shadow = true;
                }
            }
        }
        else {
            hit_color = hit_material.color;
        }

        // двусторонние полигоны
        if (dot(hit_normal, rays[rays_traced_number+idx].d) > 0.0)
            hit_normal = mult(-1.0, hit_normal);
        if (min_t == INF){
            rays[rays_traced_number+idx].color = BACKGOUND_COLOR;
        }
        else {
            vec3 p = add(rays[rays_traced_number+idx].p, mult(min_t,
rays[rays_traced_number+idx].d));
            for (uint light_id = 0; light_id < lights_number; ++light_id){
                    vec3 l = diff(lights[light_id].p, p);

                    // теневой луч
                        vec3 shadow_d = norm(l);
                        Ray shadow_ray = Ray(add(p, mult(-RAY_OFFSET,
rays[rays_traced_number+idx].d)), shadow_d, 0, 1.0);
                        vec3 shadow_color{1.0, 1.0, 1.0};
                        bool full_shadow = false;

                        if (!no_shadow){
                            for (uint object_id = 0; object_id < objects_number;
++object_id){
                                // грани
                                for (uint polygon_id = 0; polygon_id <
objects[object_id].faces_number; ++polygon_id){
                                    float t = is_ray_hits_polygon(shadow_ray,
objects[object_id].faces[polygon_id]);
                                    if (t != INF){
                                        Material shadow_hit_material =
objects[object_id].face_material;

                                        if (shadow_hit_material.transparency > 0.0){
```

```
                                        vec3 subtractive_color{1.0f -
shadow_hit_material.color.x,
                                                               1.0f -
shadow_hit_material.color.y,
                                                               1.0f -
shadow_hit_material.color.z};
                                        subtractive_color = mult((1.0f-
shadow_hit_material.transparency), subtractive_color);
                                        shadow_color = diff(shadow_color,
subtractive_color);
                                    }
                                    else{
                                        full_shadow = true;
                                        break;
                                    }
                                }
                            }
                            // ребра
                            for (uint polygon_id = 0; polygon_id <
objects[object_id].bevels_number; ++polygon_id){
                                float t = is_ray_hits_polygon(shadow_ray,
objects[object_id].bevels[polygon_id]);
                                if (t != INF){
                                    Material shadow_hit_material =
objects[object_id].bevel_material;

                                    if (shadow_hit_material.transparency > 0.0){
                                        vec3 subtractive_color{1.0f -
shadow_hit_material.color.x,
                                                               1.0f -
shadow_hit_material.color.y,
                                                               1.0f -
shadow_hit_material.color.z};
                                        subtractive_color = mult((1.0f-
shadow_hit_material.transparency), subtractive_color);
                                        shadow_color = diff(shadow_color,
subtractive_color);
                                    }
                                    else{
                                        full_shadow = true;
                                        break;
                                    }
                                }
                            }
                            // крышки
```

```
                                for (uint polygon_id = 0; polygon_id <
objects[object_id].caps_number; ++polygon_id){
                                    float t = is_ray_hits_polygon(shadow_ray,
objects[object_id].caps[polygon_id]);
                                    if (t != INF){
                                        Material shadow_hit_material =
objects[object_id].bevel_material;

                                        if (shadow_hit_material.transparency > 0.0){
                                            vec3 subtractive_color{1.0f -
shadow_hit_material.color.x,
                                                                   1.0f -
shadow_hit_material.color.y,
                                                                   1.0f -
shadow_hit_material.color.z};
                                            subtractive_color = mult((1.0f-
shadow_hit_material.transparency), subtractive_color);
                                            shadow_color = diff(shadow_color,
subtractive_color);
                                        }
                                        else{
                                            full_shadow = true;
                                            break;
                                        }
                                    }
                                }
                            }
                        }



                    // затенение Фонга
                    rays[rays_traced_number+idx].color = vec3{0.0, 0.0, 0.0};
                    if (!full_shadow){
                        float n_l_cos = dot(hit_normal, norm(l));
                        if (n_l_cos < 0.0)
                            n_l_cos = 0.0;
                        rays[rays_traced_number+idx].color = mult(n_l_cos,
hit_color);

                        vec3 r = diff(mult(2*dot(hit_normal, norm(l)), hit_normal),
norm(l));
                        float v_r_cos = dot(mult(-1.0,
rays[rays_traced_number+idx].d), norm(r));
                        if (v_r_cos < 0){
```

```
                            v_r_cos = 0.0;
                        }
                        float specular_sharpness = 8.0;
                        rays[rays_traced_number+idx].color =
add(rays[rays_traced_number+idx].color, mult(pow(v_r_cos, specular_sharpness),
lights[light_id].color));

                        rays[rays_traced_number+idx].color.x *= shadow_color.x;
                        rays[rays_traced_number+idx].color.y *= shadow_color.y;
                        rays[rays_traced_number+idx].color.z *= shadow_color.z;
                    }
                    rays[rays_traced_number+idx].color =
add(rays[rays_traced_number+idx].color, mult(0.1, hit_color));
                    rays[rays_traced_number+idx].color =
add(rays[rays_traced_number+idx].color, BACKGOUND_LIGHT_COLOR);
                }
            if (rays[rays_traced_number+idx].bounses != 0 and
rays[rays_traced_number+idx].intencity > INTENCITY_THRESHOLD){
                // уменьшить интесивность Фонга
                    rays[rays_traced_number+idx].color = mult(hit_material.phong,
rays[rays_traced_number+idx].color);
                // отраженный
                if (rays[rays_traced_number+idx].intencity*hit_material.reflective >
INTENCITY_THRESHOLD){
                    vec3 r_dir = mult(-2.0*dot(rays[rays_traced_number+idx].d,
hit_normal), hit_normal);
                    r_dir = add(r_dir, rays[rays_traced_number+idx].d);
                    rays[rays_traced_number+rays_to_trace_number+idx] = Ray(add(p,
mult(-RAY_OFFSET, rays[rays_traced_number+idx].d)), r_dir,
rays[rays_traced_number+idx].bounses-1,
rays[rays_traced_number+idx].intencity*hit_material.reflective);
                    rays[rays_traced_number+rays_to_trace_number+idx].parent =
rays_traced_number+idx;
                    rays_bitmap[rays_traced_number+rays_to_trace_number+idx] = false;
                }
                // сквозной
                if (rays[rays_traced_number+idx].intencity*hit_material.transparency
> INTENCITY_THRESHOLD){
                    rays[rays_traced_number+2*rays_to_trace_number+idx] = Ray(add(p,
mult(RAY_OFFSET, rays[rays_traced_number+idx].d)), rays[rays_traced_number+idx].d,
rays[rays_traced_number+idx].bounses-1,
rays[rays_traced_number+idx].intencity*hit_material.transparency);
                    rays[rays_traced_number+2*rays_to_trace_number+idx].parent =
rays_traced_number+idx;
```

```
                    rays_bitmap[rays_traced_number+2*rays_to_trace_number+idx] =
false;
                }
            }
        }

        idx += gridDim.x * blockDim.x;
    }
}
```

Основной цикл параллельного подсчета и обмена данными между процессами:

```
while(max_eps > eps){
        MPI_Barrier(MPI_COMM_WORLD);
        // отправка (right. back, up)
            // right
            if (xb < xnb-1){
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        send_left_right_buff[z*ny+y] = data[_i(nx-1, y, z)];
                    }
                }
                MPI_Send(send_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb+1, yb, zb),
id, MPI_COMM_WORLD);
            }
            // back
            if (yb < ynb-1){
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        send_front_back_buff[z*nx+x] = data[_i(x, ny-1, z)];
                    }
                }
                MPI_Send(send_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb+1, zb),
id, MPI_COMM_WORLD);
            }
            // up
            if (zb < znb-1){
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        send_up_down_buff[y*nx+x] = data[_i(x, y, nz-1)];
                    }
                }
                MPI_Send(send_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb+1), id,
MPI_COMM_WORLD);
            }
```

```
        // прием (left, front, down)
            // left
            if (xb > 0){
                // std::cout << id << "left\n";
                MPI_Recv(recv_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb-1, yb, zb),
_ib(xb-1, yb, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        data[_i(-1, y, z)] = recv_left_right_buff[z*ny+y];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        data[_i(-1, y, z)] = bc_left;
                    }
                }
            }
            // front
            if (yb > 0){
                // std::cout << id << "front\n";
                MPI_Recv(recv_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb-1, zb),
_ib(xb, yb-1, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, -1, z)] = recv_front_back_buff[z*nx+x];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, -1, z)] = bc_front;
                    }
                }
            }
            // down
            if (zb > 0){
                // std::cout << id << "down\n";
                MPI_Recv(recv_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb-1),
_ib(xb, yb, zb-1), MPI_COMM_WORLD, &status);
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, -1)] = recv_up_down_buff[y*nx+x];
                    }
                }
```

```
            } else {
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, -1)] = bc_down;
                    }
                }
            }
        MPI_Barrier(MPI_COMM_WORLD);
        // отправка (left, front, down)
            // left
            if (xb > 0){
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        send_left_right_buff[z*ny+y] = data[_i(0, y, z)];
                    }
                }
                MPI_Send(send_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb-1, yb, zb),
id, MPI_COMM_WORLD);
            }
            // front
            if (yb > 0){
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        send_front_back_buff[z*nx+x] = data[_i(x, 0, z)];
                    }
                }
                MPI_Send(send_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb-1, zb),
id, MPI_COMM_WORLD);
            }
            // down
            if (zb > 0){
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        send_up_down_buff[y*nx+x] = data[_i(x, y, 0)];
                    }
                }
                MPI_Send(send_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb-1), id,
MPI_COMM_WORLD);
            }
        // прием (right, back, up)
            // right
            if (xb < xnb-1){
                MPI_Recv(recv_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb+1, yb, zb),
_ib(xb+1, yb, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
```

```
                    for (int y = 0; y < ny; ++y){
                        data[_i(nx, y, z)] = recv_left_right_buff[z*ny+y];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        data[_i(nx, y, z)] = bc_right;
                    }
                }
            }
            // back
            if (yb < ynb-1){
                MPI_Recv(recv_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb+1, zb),
_ib(xb, yb+1, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, ny, z)] = recv_front_back_buff[z*nx+x];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, ny, z)] = bc_back;
                    }
                }
            }
            // up
            if (zb < znb-1){
                MPI_Recv(recv_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb+1),
_ib(xb, yb, zb+1), MPI_COMM_WORLD, &status);
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, nz)] = recv_up_down_buff[y*nx+x];
                    }
                }
            } else {
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, nz)] = bc_up;
                    }
                }
            }
        MPI_Barrier(MPI_COMM_WORLD);
```

```
        // итерация
        double eps = -10000000.0;
        for (int z = 0; z < nz; ++z){
            for (int y = 0; y < ny; ++y){
                for (int x = 0; x < nx; ++x){
                    next_data[_i(x, y, z)] = ((data[_i(x+1, y, z)]+data[_i(x-1, y,
z)])/(hx*hx) +
                                              (data[_i(x, y+1, z)]+data[_i(x, y-1,
z)])/(hy*hy) +
                                              (data[_i(x, y, z+1)]+data[_i(x, y, z-
1)])/(hz*hz)) /
                                              (2.0*((1/(hx*hx))+(1/(hy*hy))+(1/(hz*hz
))));
                    eps = std::max(eps, std::abs(next_data[_i(x, y, z)]-data[_i(x, y,
z)]));
                }
            }
        }

        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Allgather(&eps, 1, MPI_DOUBLE, eps_buff, 1, MPI_DOUBLE, MPI_COMM_WORLD);

        max_eps = -10000000.0;
        for (int i = 0; i < znb*ynb*xnb; ++i){
            max_eps = std::max(max_eps, eps_buff[i]);
        }

        // обмен указателями
        double* tmp = data;
        data = next_data;
        next_data = tmp;
    }
```

Сжатие массива с лучами

Используя информацию о том, где в массиве пусто, а где луч, сожму их, распределив непрерывно, начиная с начала массива. Для этого использую поразрядную сортировку на базе алгоритма scan. Так как в рамках одного разряда эта сортировка стабильна, то информация об индексах родительских лучей, для только что появившихся лучей останется валидной. За разряд приму массив нулей и единиц, где ноль – значит на этом индексе есть луч, а единица – нет.

Функция сжатия:

```
uint compact(Ray* &dev_rays, bool* dev_rays_bitmap, uint size){

        Ray* dev_compact_rays;
        CSC(cudaMalloc(&dev_compact_rays, (size*sizeof(Ray))));

        uint* dev_s;
        CSC(cudaMalloc(&dev_s, ((size+1)*sizeof(uint))));

        s_gen_kernel <<< GRID_SIZE, BLOCK_SIZE >>> (dev_rays_bitmap, size, dev_s);
        CSC(cudaGetLastError());

        scan(dev_s, size+1);

        uint new_rays_number;
        cudaMemcpy(&new_rays_number, dev_s+size, sizeof(uint),
cudaMemcpyDeviceToHost);
        new_rays_number = size-new_rays_number;

        binary_digit_sort_kernel <<< GRID_SIZE, BLOCK_SIZE >>> (dev_rays,
dev_rays_bitmap, dev_s, size, dev_compact_rays);
        CSC(cudaGetLastError());

        std::swap(dev_rays, dev_compact_rays);

        cudaFree(dev_s);
        cudaFree(dev_compact_rays);

        return new_rays_number;
    }
```

Метод рендера изображения у класса камеры:

```
void Render(uint bounces){
            Ray* rays;
            rays = (Ray*)malloc(w*h*sizeof(Ray));

            float dw = 2.0/(w-1.0);
            float dh = 2.0/(h-1.0);
            float z_side = 1.0/tan(fov*(M_PI/360.0));

            vec3 basis_z = norm(diff(d, p));
            vec3 basis_x = norm(prod(basis_z, vec3{0.0, 0.0, 1.0}));
            vec3 basis_y = norm(prod(basis_x, basis_z));
            for (int y = 0; y < h; ++y){
                for (int x = 0; x < w; ++x){
                    vec3 v = vec3{-1.0f+dw*x, -1.0f+dh*y* h / w, z_side};
                    matr3 matr(basis_x, basis_y, basis_z);
                    vec3 dir = norm(mult(matr, v));

                    rays[x*h+y] = Ray(p, dir, bounces, 1.0);
                }
            }

            bool* dev_rays_bitmap;
            // копирование лучей в память gpu
                CSC(cudaMalloc(&dev_rays, 3*w*h*sizeof(Ray)));
                CSC(cudaMalloc(&dev_rays_bitmap, 3*w*h*sizeof(bool)));
                CSC(cudaMemcpy(dev_rays, rays, w*h*sizeof(Ray),
cudaMemcpyHostToDevice));
                free(rays);
                map_rays_bitmap_kernel <<< GRID_SIZE, BLOCK_SIZE >>>
(dev_rays_bitmap, w*h, 3*w*h);

            uint rays_number = w*h;
            uint traced_rays_number = 0;
            uint rays_to_trace_number = w*h;

            std::vector<uint> rays_traced_chunks_shifts(1, 0);

            for (int i = 0; i <= bounces; ++i){
                rays_traced_chunks_shifts.push_back(rays_to_trace_number+rays_traced_
chunks_shifts.back());
                trace_rays_kernel <<< GRID_SIZE, BLOCK_SIZE >>> (dev_rays,
dev_rays_bitmap,
                                                                 traced_rays_number,
                                                                 rays_to_trace_number,
```

```
                                                        dev_scene_objects,
                                                        scene_objects_number,
                                                        dev_scene_lights,
                                                        scene_lights_number,
                                                        dev_scene_floor);
                CSC(cudaGetLastError());

                uint old_traced_rays_number = traced_rays_number;
                traced_rays_number = rays_number;
                rays_number = compact(dev_rays, dev_rays_bitmap,
old_traced_rays_number+(3*rays_to_trace_number));

                rays_to_trace_number = rays_number-traced_rays_number;

                if (rays_to_trace_number == 0){
                    break;
                }

                // перевыделение памяти
                    Ray* new_dev_rays;
                    CSC(cudaMalloc(&new_dev_rays,
(traced_rays_number+(3*rays_to_trace_number))*sizeof(Ray)));
                    CSC(cudaMemcpy(new_dev_rays, dev_rays, rays_number*sizeof(Ray),
cudaMemcpyDeviceToDevice));
                    cudaFree(dev_rays);
                    dev_rays = new_dev_rays;

                    cudaFree(dev_rays_bitmap);
                    CSC(cudaMalloc(&dev_rays_bitmap,
(traced_rays_number+(3*rays_to_trace_number))*sizeof(bool)));
                    map_rays_bitmap_kernel <<< GRID_SIZE, BLOCK_SIZE >>>
(dev_rays_bitmap, rays_number, (traced_rays_number+(3*rays_to_trace_number)));
                    CSC(cudaGetLastError());
            }
            cudaFree(dev_rays_bitmap);

            // сжать результат
                Ray* new_dev_rays;
                CSC(cudaMalloc(&new_dev_rays, rays_number*sizeof(Ray)));
                CSC(cudaMemcpy(new_dev_rays, dev_rays, rays_number*sizeof(Ray),
cudaMemcpyDeviceToDevice));
                cudaFree(dev_rays);
                dev_rays = new_dev_rays;

                rays_count = rays_number;
```

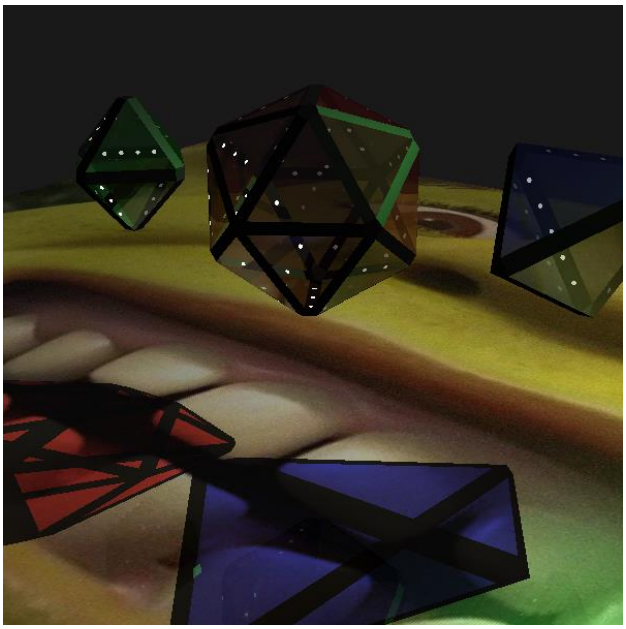
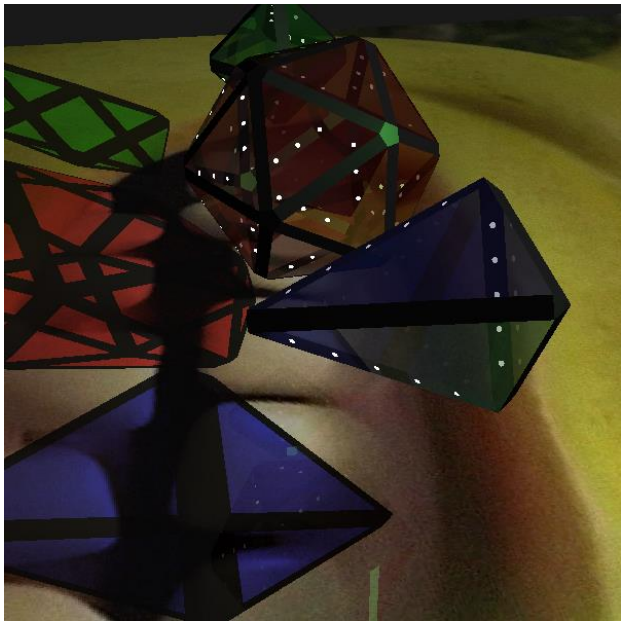
```
            // обратный обход деревьев
                for (uint shift_index = rays_traced_chunks_shifts.size()-2;
shift_index >= 1; --shift_index){
                    back_collection_step_kernel <<< GRID_SIZE, BLOCK_SIZE >>>
(dev_rays, rays_number, rays_traced_chunks_shifts[shift_index],
rays_traced_chunks_shifts[shift_index+1]);
                    CSC(cudaGetLastError());
                }


            Ray* screen_rays = (Ray*)malloc(w*h*sizeof(Ray));
            CSC(cudaMemcpy(screen_rays, dev_rays, w*h*sizeof(Ray),
cudaMemcpyDeviceToHost));

            data = (uchar4*)malloc(w*h*sizeof(uchar4));
            for (int y = 0; y < h; ++y){
                for (int x = 0; x < w; ++x){
                    data[(h-1-y)*w+x].w = 255;
                    data[(h-1-y)*w+x].x =
(uint)(std::clamp(screen_rays[x*h+y].color.x, 0.0f, 1.0f)*255);
                    data[(h-1-y)*w+x].y =
(uint)(std::clamp(screen_rays[x*h+y].color.y, 0.0f, 1.0f)*255);
                    data[(h-1-y)*w+x].z =
(uint)(std::clamp(screen_rays[x*h+y].color.z, 0.0f, 1.0f)*255);
                }
            }

            free(screen_rays);
            cudaFree(dev_rays);
        }
```

Тест быстродействия (в миллисекундах)

	Время рендера одного кадра		
	Минимальное	**Максимальное**	**Среднее**
GPU(CUDA) <32, 32>	405.327	660.686	543.892
GPU(CUDA) <64, 64>	317.949	522.418	431.670
GPU(CUDA) <128, 128>	279.124	**495.294**	376.045
GPU(CUDA) <256, 256>	**275.086**	469.120	**371.771**
CPU	32429.768	43979.532	38346.873

Результат

Изображения, полученные путем рендера демонстрационной сцены:

Выводы

Алгоритм обратной трассировки лучей является фундаментальным для рендер-движков, работающих по принципу трассировки лучей. С помощью него можно добиваться реалистичной или просто красивой картинки в большом числе задач компьютерной графики. Так как математическая модель луча, используемая в данном алгоритме, не подразумевает взаимодействия лучей непосредственно друг с другом, то каждый луч можно считать независимым. Это дает большой потенциал в распараллеливании вычислений, что делает использование GPU для его работы очень выгодным.
Исходя из результатов тестирования быстродействия, реализация обратной трассировки с использованием GPU работает в разы быстрее, чем последовательный просчет на CPU. С ростом качества картинки будет также расти и количество лучей, что еще больше увеличит разницу GPU и CPU реализации. Преимущества видеокарт, в рамках данной задачи, колоссальны.