CSE 167 Final Project: Ray Tracing

Wenxiao Li

Index:

1. Intro

Ray tracers can produce some of the most impressive renderings with high quality shadows and reflections. This project constructs a ray tracer **with global illumination**. The components of this project include: ppm image files I/O, basic vec3 functions, rays, a movable camera, geometries including sphere and triangle, materials including Lambertian, metal and dielectric.

Some out-class materials that I referenced in building the project:

https://www.realtimerendering.com/ *Ray tracer in one weekend*.

https://github.com/RayTracing/raytracing.github.io

https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm This one is for implementing triangle geometries.

You can open the FinalProject folder, and run FinalProject.exe in Microsoft Visual Studio. The project should be able to compile. If the code somehow does not compile, contact wel032@ucsd.edu, and I am very happy to do a showcase via zoom.

2. PPM Image files I/O

PPM image files stores (R, G, B) channels of a pixel in every single line. The pixels are read/written from left to right, from top to bottom. Here is an example from Wikipedia:

In Finalproject.cpp, at the begin of my main function I create an output file stream, and pass in a ppm file. At the end of main, after a color for a pixel is computed, I write the color to the ppm file. To read the ppm file so you can see it as a real image, go to https://www.cs.rhodes.edu/welshc/COMP141_F16/ppmReader.html.

3. Vec3 Library

In vec3.h, a vec3 class is constructed and lots of useful vec3 functions are also included. I get this function library from https://github.com/RayTracing/raytracing.github.io. Therefore, I do not need to use build-in glm classes. And, all the implementations are done via vec3 and scalar operations, not via matrices.

4. Rays

Rays can be represented as R(t) = p0 + t*d. In this scenario, p0 is the origin point of the ray, and d is the direction. In my ray.h file, I defined a ray class, with A and B as pass in parameters. A represents the original point of ray, and B represents the direction.

Here is my Ray class:

```cpp
#ifndef RAYH
#define RAYH
#include "vec3.h"

class ray {

    public:
        ray() {}
        ray(vec3& a, vec3& b) {
            A = a;
            B = b;
        }

        vec3 origin() {
            return A;
        }

        vec3 direction() {
            return B;
        }

        vec3 point_at_parameter(float t) {
            return A + t * B;
        }

        vec3 A;
        vec3 B;
};
#endif // !RAYH
```

5. Camera

Vfov(theta) is the angle that we see through the portal. Therefore, the angle between the horizonal axis and one of vfov line is theta/2. By convention z = 1, so half height will be tan(theta/2). Since retina plane maintains the ratio of actual height and size, we pass in a ratio parameter aspect, and get half-width. The eye of the camera is exactly the vec3 look from. Then, we try to set up a basis. w is pointing opposite to the retina plane. And, u is determined by cross product of w and another pass in vector vup. After u, w are set, v is determined by cross product of u and w. Last, we place the retina plane in the correct position by specifying its lower-left corner. Here is my camera class:

```cpp
class camera {
    public:

        camera() {
            lower_left = vec3(-2.0, -1.0, -1.0);
            horizontal = vec3(4.0, 0.0, 0.0);
            vertical = vec3(0.0, 2.0, 0.0);
            origin = vec3(0.0, 0.0, 0.0);
        }

        camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect) {
            vec3 u, v, w;
            float theta = vfov * M_PI / 180;
            float half_height = tan(theta / 2);
            float half_width = aspect * half_height;
            origin = lookfrom;
            w = unit_vector(lookfrom - lookat);
            u = unit_vector(cross(vup, w));
            v = cross(w, u);

            horizontal = 2 * half_width * u;
            vertical = 2 * half_height * v;
            lower_left = origin - half_width * u - half_height * v - w;
            //lower_left = vec3(-half_width, -half_height, -1.0);
            //horizontal = vec3(2.0 * half_width, 0.0, 0.0);
            //vertical = vec3(0.0, 2.0 * half_height, 0.0);

        }

        ray get_ray(float u, float v) {
            return ray(origin, lower_left + u * horizontal + v * vertical - origin);
        }

        vec3 origin;
        vec3 lower_left;
        vec3 horizontal;
        vec3 vertical;
};
```

The get ray function is an essential function RayThruPixel. Basically, given a ray's origin and a point it passes, it will generate the right ray in world frame.

6. Geometries

We use the hit function to find if there are intersections between a certain object and a ray. Since different geometries hit differently, I make this function virtual to ensure robustness. The hit_record struct will be passed as reference all the time, to make sure the hit information will be updated along the way. In hit_record, t keeps track of the distance between the last hit point (this point is ray origin initially) to the current one. P keeps track of the intersection point. And, normal records the current normal vector. Here is my abstract Geometry class:
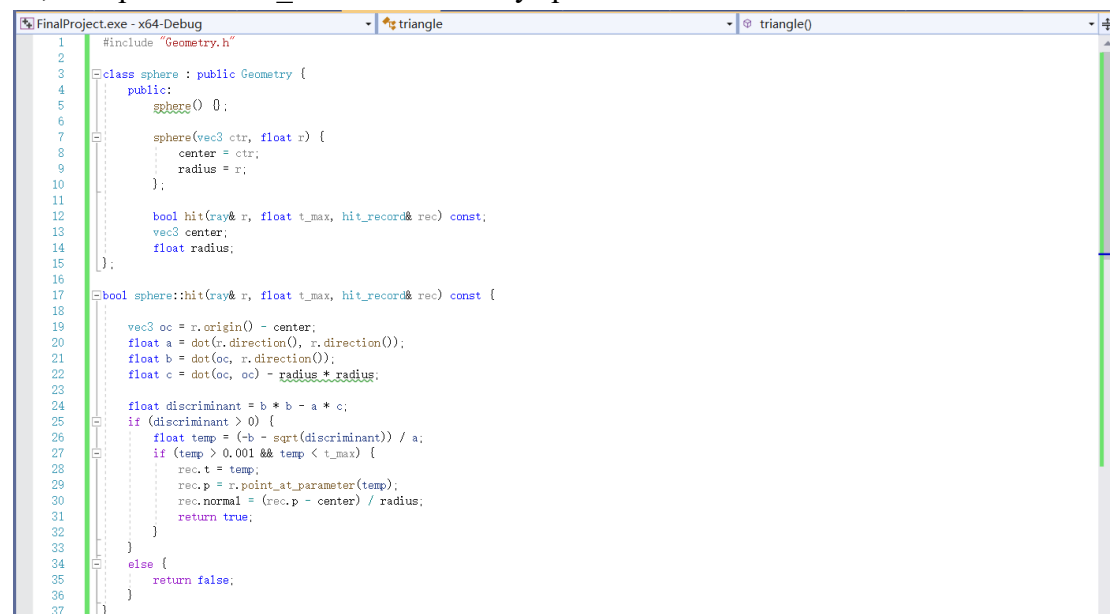
```cpp
#include "ray.h"

class material;

struct hit_record {
    float t = 0; // hit distance
    vec3 p;      // intersection
    vec3 normal;
};

class Geometry {
    public:
        virtual bool hit(ray& r, float t_max, hit_record& rec) const = 0;
};
```

6.1 Spheres

Spheres can be determined by specifying its center and radius. In constructor, I pass in these two parameters. In hit function, to check whether a ray intersects with a sphere, we need to solve a quadratic equation with respect to t. The equation is:
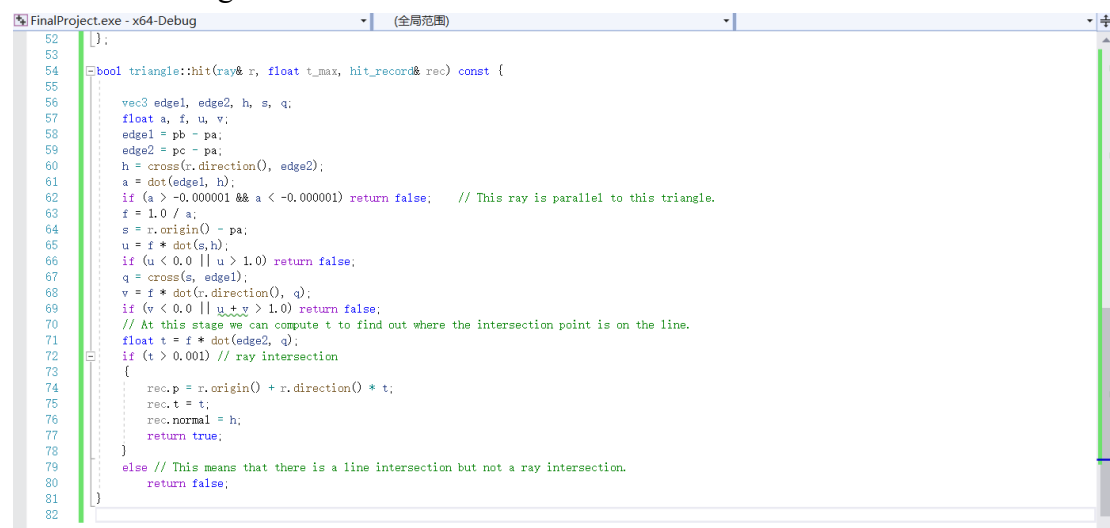
$$d^2t^2 + 2d(p_0 - c)t + (p_0 - c)^2 - r^2 = 0$$

In my implementation, oc $=(p_0 - c)$, a $= d^2$, b $= d(p_0 - c)$, c $= (p_0 - c)^2 - r^2$. And, the discriminant to check will be $b^2 - ac$, since I do not multiply 2 to b previously. If the discriminant > 0, we get the nearer intersection by taking the sign negative. Then, we check if this distance is the nearest intersection we have seen from the beginning. If so, we update the hit_record. Here is my sphere class:

```cpp
#include "Geometry.h"

class sphere : public Geometry {
    public:
        sphere() {};

        sphere(vec3 ctr, float r) {
            center = ctr;
            radius = r;
        };

        bool hit(ray& r, float t_max, hit_record& rec) const;
        vec3 center;
        float radius;
};

bool sphere::hit(ray& r, float t_max, hit_record& rec) const {

    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = dot(oc, r.direction());
    float c = dot(oc, oc) - radius * radius;

    float discriminant = b * b - a * c;
    if (discriminant > 0) {
        float temp = (-b - sqrt(discriminant)) / a;
        if (temp > 0.001 && temp < t_max) {
            rec.t = temp;
            rec.p = r.point_at_parameter(temp);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
    }
    else {
        return false;
    }
}
```

6.2 Triangles

For triangles, we only need to specify its 3 vertices. To check if a ray hit the triangle, I calculate the barycentric coordinates of 3 vertices, and check if they are between 0 and 1. Notice before that, to avoid numerical issues, I first check where the ray is parallel to the triangle. If so, the cross product of direction of r and an edge (pass into variable h) will be orthogonal to the surface. And, I just check the dot product of h and another edge to see if they are orthogonal to each other. Then, I proceed to check the barycentric coordinates. In lecture slides, we get them by constructing a matrix. However, since I do not use matrix libraries, I utilize the Möller–Trumbore intersection algorithm from Wikipedia. In the implementation, u and v are barycentric coordinates, and I check their range. If they satisfy the requirements, I calculate the intersection and update hit_record. Here is the triangle class:

```cpp
bool triangle::hit(ray& r, float t_max, hit_record& rec) const {

    vec3 edge1, edge2, h, s, q;
    float a, f, u, v;
    edge1 = pb - pa;
    edge2 = pc - pa;
    h = cross(r.direction(), edge2);
    a = dot(edge1, h);
    if (a > -0.000001 && a < -0.000001) return false;    // This ray is parallel to this triangle.
    f = 1.0 / a;
    s = r.origin() - pa;
    u = f * dot(s,h);
    if (u < 0.0 || u > 1.0) return false;
    q = cross(s, edge1);
    v = f * dot(r.direction(), q);
    if (v < 0.0 || u + v > 1.0) return false;
    // At this stage we can compute t to find out where the intersection point is on the line.
    float t = f * dot(edge2, q);
    if (t > 0.001) // ray intersection
    {
        rec.p = r.origin() + r.direction() * t;
        rec.t = t;
        rec.normal = h;
        return true;
    }
    else // This means that there is a line intersection but not a ray intersection.
        return false;
}
```

I implement the triangle class in the same file as the sphere class. This is because the compiler complains the hit functions issues if I put them into separate files.

7. Materials

In the abstract class material, we have a function scatter to take care of both the diffuse light and the specular light. In the implementation with global illumination, Lambertian materials will only diffuse colors, while metal materials will always reflect color. Some in-the-middle materials can be created by taking some proportion of diffuse, and some proportion of reflect. But I only implemented the two separate cases since I believe they are representative. Materials will absorb energy in each bounce, but in my implementation, it is not taken care of in the material classes.

7.1 Lambertian

Attenuation of a material will determine the color of it. To make the out ray diffuse to all the directions, I pick a point in the sphere at random, to construct a random out

direction. (Rec.p + rec.normal) represents the center of the sphere that I am going to generate a direction. And, a random point in the sphere determines the direction, which is target – rec.p. And, the out ray has intersection point as the origin, target – rec.p as direction. Here is my Lambertian class:

7.2 Metal

For metal materials, I calculate the reflected vector the same method as HW3. Besides, the fuzziness of the metal is also implemented. To implement fuzziness, I added some randomness to the reflected vector. The index fuz will determine how fuzzy the material is, by controlling the proportion of randomness. Here is my metal class:

*7.3 Dielectric

A glass material is implemented in this class. We did not cover it in class. I get this from https://github.com/RayTracing/raytracing.github.io just to make the result look fancier.

8. Raytracing details

At the start of the program, I set up the image width, height (nx, ny), number of times to take random results to produce global illumination (ns), the camera position, and constructing the world scene with materials. Here is a code snippet that achieves it:

```cpp
int main() {
    ofstream outFile;
    outFile.open("out_camera_triangle.ppm");

    int nx = 900;
    int ny = 600;
    int ns = 25;

    outFile << "P3\n" << nx << " " << ny << "\n255\n";

    camera cam(vec3(13, 2, 3), vec3(0, 0, 0), vec3(0,1,0), 20, float(nx) / float(ny));

    //camera cam;
    vector<Geometry*> world;
    vector<material*> materials;

    world.push_back(new sphere( vec3(0,0,-1), 0.5));
    materials.push_back(new lambertian(vec3(0.1, 0.2, 0.5)));
    world.push_back(new sphere(vec3(0, -100.5, -1), 100));
    materials.push_back(new lambertian(vec3(0.8, 0.8, 0.0)));
    world.push_back(new sphere(vec3(1, 0, -1), 0.5));
    materials.push_back(new metal(vec3(0.8, 0.6, 0.2), 1.0));
    world.push_back(new sphere(vec3(-1, 0, -1), 0.5));
    materials.push_back(new dielectric(1.5));
```

Then, for every pixel of the image (controlled by i,j loops), I compute the color of the pixel. To compute the color, I shoot a ray to that pixel. I can get a point that the ray passes. Therefore, I am able to compute the ray in world frame. I do this procedure ns times to better accommodate the randomness that I introduced to materials' color diffuse and fuzziness. After ns times, I average the color. Notice that I mentioned earlier rays lose energy in each bounce. In my implementation I make the loss rate 50%. So, to cleverly address the loss, I just take the square root of the result color, which is called "gamma 2 approximation" by *Ray tracer in one weekend*. Here is the code:

```cpp
    materials.push_back(new metal(vec3(0.8, 0.6, 0.3), 0.2));

    for (int j = ny - 1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(0, 0, 0);
            for (int s = 0; s < ns; s++) {
                float u = float(i) / float(nx);
                float v = float(j) / float(ny);

                ray r = cam.get_ray(u, v);

                col += calColor(r, world, materials, 0);
            }
            col /= float(ns);
            col = vec3(sqrt(col[0]), sqrt(col[1]), sqrt(col[2]));
            int ir = int(255.99 * col[0]);
            int ig = int(255.99 * col[1]);
            int ib = int(255.99 * col[2]);

            outFile << ir << " " << ig << " " << ib << "\n";

        }
    }
```

Then, I will go through the details of computing a color. First, I will introduce the light source. There is no explicit light source constructed in the project, but a sun that is infinitely far away can be assumed. The reason for not implementing artificial light sources is the computational cost. In current scenario, the ray tracer takes several minutes to operate.

```cpp
    else {
        vec3 unit_dir = unit_vector(r.direction());
        t = 0.5 * (unit_dir.y() + 1.0);
    }
    return (1.0-t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
}
```

This part constructed a mix of blue and white background that can mimic a blue sky as a real-world scene. And, whenever a bounced ray shoots to the sky, it will get a color mixed by white and blue.

In the i loop, we go through all the geometries to find the nearest hit, and update

hit_record along the way. Then, if there is a hit, we calculate the bounced ray, and make recursive calls to compute the color. Otherwise, the ray shoots to the sky. In my implementation, I set the max recursive depth to 5. The recurse will stop when depth=5.
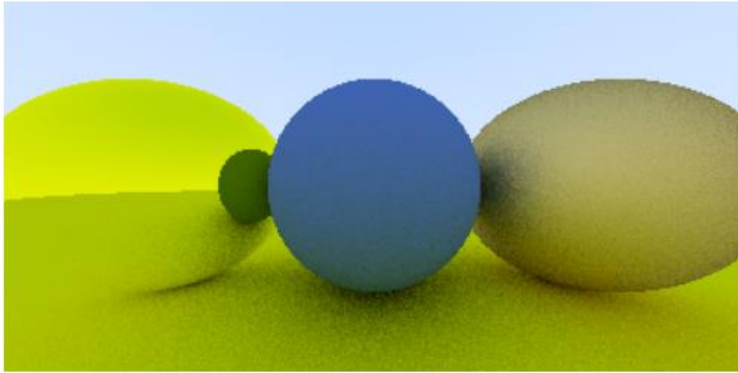

9. Result demonstration

There will be multiple ppm files available. These PPM files were created when I gradually developed my project. Here are image views of them for you to check:
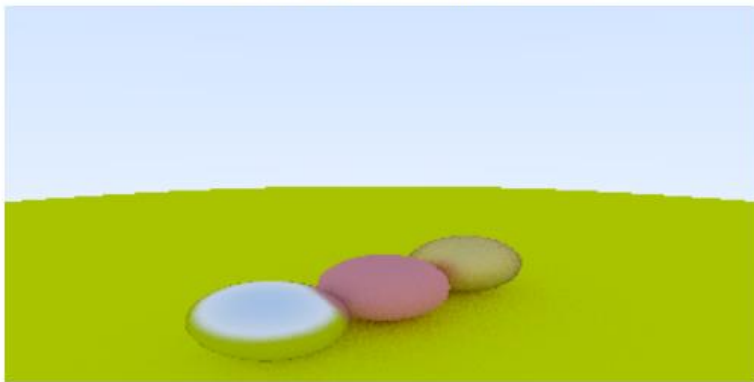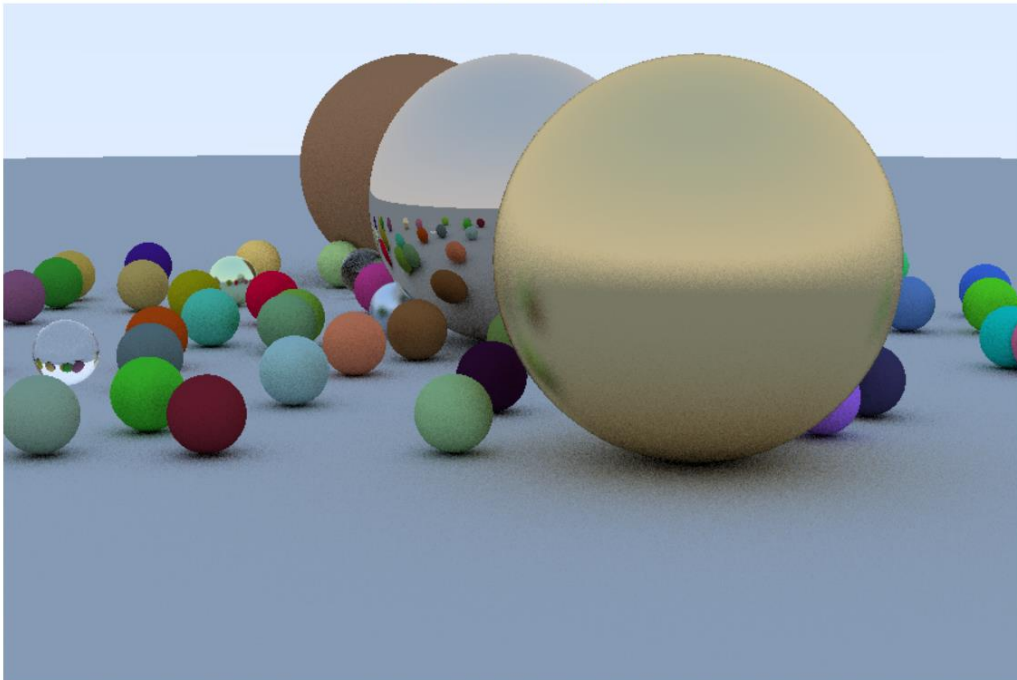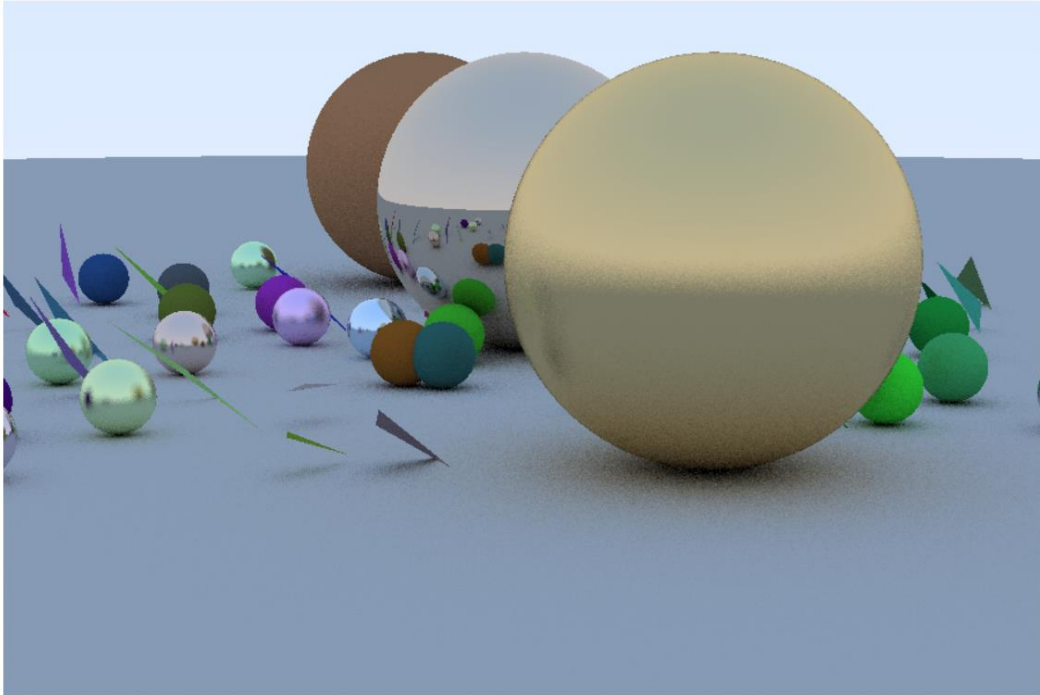

File: out.ppm
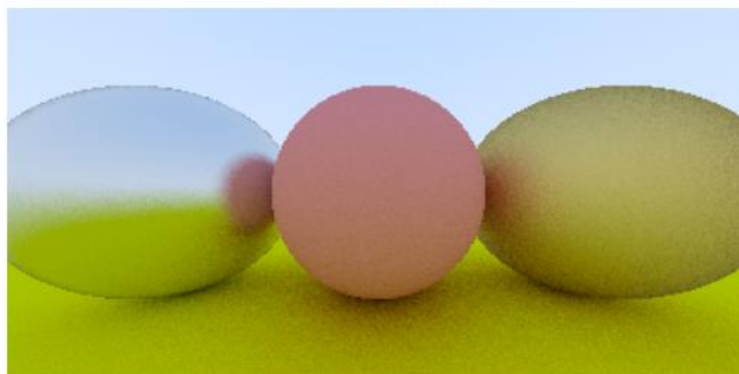
File: out2.ppm

File: out_camera.ppm

File: out_camera_spheres.ppm

File: out_camera_megaRandom.ppm

When you run FinalProject.cpp, you should generate a megaRandom scene. It is not guaranteed that the scenes we get are exactly the same, since it involves lots of randomness in constructing the scene.