CSE306 Assignment 1 - Raytracer

Maria Barbulescu

May 2020

1 Introduction

The purpose of this project is to implement a ray tracer from scratch. This report presents the different steps and their outputs as well as explains the code behind the results.

The C++ code is organized in four different files: `path_tracer.cpp`, `tools.cpp`, `classes.cpp`, `vector.cpp`. The main function is in the first file, while additional classes and functions are in the other three. The `Vector` class is in a separate file for convenience (it can be used for many different projects and this makes it easier to locate). The `classes.cpp` file contains the `Intersection`, `Sphere`, `Ray`, and `Scene` classes. Each `Ray` has an origin vector `O` and a direction unit vector `u`. A Sphere has a center `C`, a radius `r`, an `albedo` vector that is an RGB color vector taking values between 0 and 1, and a `type` string that stands for `"diffuse"`, `"mirror"`, `"transparent"` or `"light"` as needed later on. It also has an `intersect(ray R)` function which returns an `Intersection` object representing the ray's intersection with the sphere. The `Intersection` class objects have attributes such as a `bool exists` that returns `true` if an intersection exists, `t1, t2, t` that represent the distances to the first and second intersection and the closest one of the two respectively, an intersection vector `P` and its unit normal `N`. A `Scene` object is defined by a `vector` of `Spheres` and has three methods. The `closest_intersect(Ray r)` method returns the index of the `Sphere` that intersects `r`, while the `intersection(Ray r)` returns the respective `Intersection` object. Finally, the `get_color(...)` method returns the color for each pixel and will be further explained later on.

2 Rendering basic spheres: diffuse objects

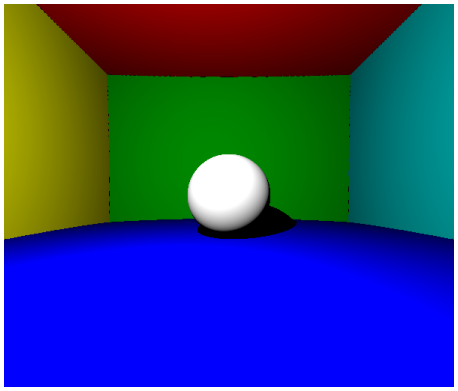

Figure 1: Diffuse object rendering

After implementing the different classes, we set up the scene in the `path_tracer.cpp` file. Initially, there are seven spheres, six of which represent the walls, floor and ceiling. They are all diffuse objects and have different colors. The main object is the seventh sphere which is significantly smaller than the others, and its color is white. For now, the light source is a `Vector`. The camera is set up at `Vector Q`, the width and height of the output image at 512 (this value

changes slightly throughout the project hence the different sizes of the images shown), and the field of view angle at $\frac{\pi}{3}$. To compute the coordinates of each pixel `V`, we iterate on the height and

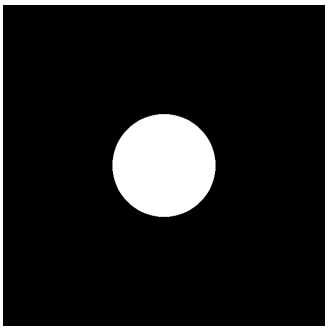

Figure 2: Sphere intersection test

width of the image. Then, we send a `Ray` from `Q` with a normalized ray direction `n` obtained from `V - Q`. A simple initial test for the `Sphere intersect` method is done by setting each pixel to white if the intersection exists, and black otherwise, only considering our main object. The result of this test can be seen in Figure 2.

Adding the other spheres to the scene and implementing the `Scene intersection` allows us to get started on the 3D components of our image, namely the shading and shadows. We implement an `intensity` function to account for the Lambertian model which leads to the first rendering of basic spheres as shown in the first image of Figure 3. The noise is corrected by slightly offsetting the ray origin (second image). The next correction is color oriented, namely a gamma correction, which consists in replacing the pixel color by its value to the power of $\frac{1}{2.2}$. Lastly, we increase the intensity to a value around 10^6 and we obtain the last image in Figure 3.

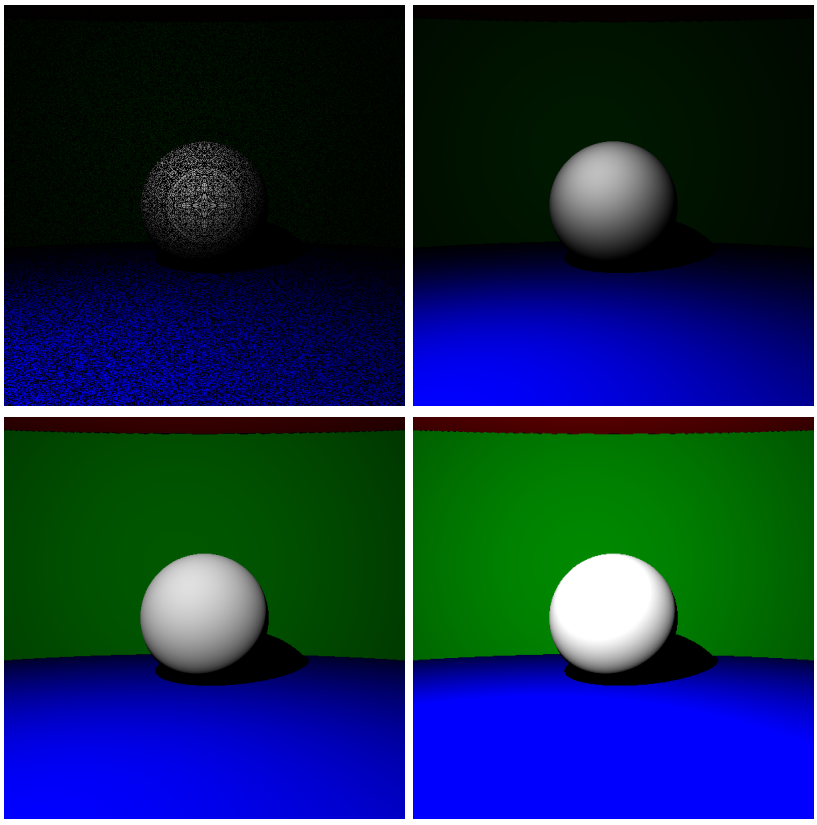

Figure 3: First renderings of a diffuse object

3 Mirrors and transparent objects

To simplify the process for several types of spheres, we implement the `Scene get_color` method. For now, this method keeps the Lambertian model for diffuse surfaces, but applies reflection and refraction according to physical laws. These are the first paths in the code, since the method needs to be recursive. When implementing mirrors, we get a similar amount of noise as with diffuse objects, and we correct that by once again offsetting the ray origin (Figure 4).

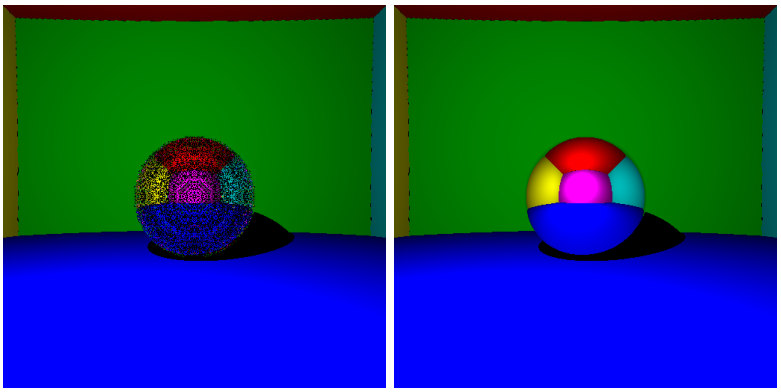

Figure 4: Reflection before and after origin offset

After implementing refraction with `n1 = 1` and `n2 = 1.5` and offsetting the origin accordingly, we can obtain results such as Figure 5, which took under 4 seconds to compute without parallelization. While the middle sphere in Figure 5 is completely transparent, it is not very realistic

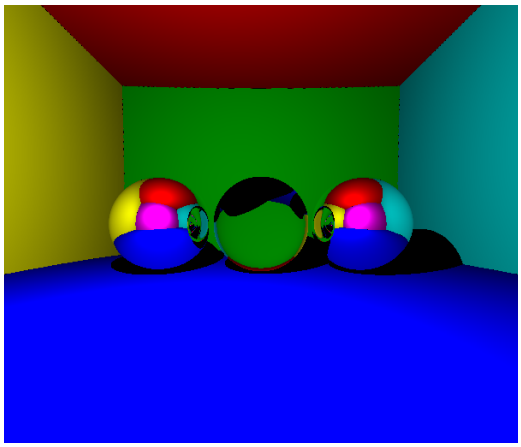

Figure 5: Reflection and refraction

because it does not reflect any rays. To improve this, we change the code according to Fresnel's law. Iterating over pixels, we send k rays to each pixel, resulting in k paths, each obtaining a different color. We then average the value of this color for every pixel. This completes this section and the output can be observed in Figure 6 (the images use 500 rays per pixel and take under 2 minutes with parallelization).

4 Indirect lighting and anti-aliasing

We add indirect lighting to the ray tracer by adding a second component to the diffuse objects in the `get_color` method. For that, we implement a function `random_cos(Vector N)` that takes a vector and produces a new random direction that will be used in sending random rays originating

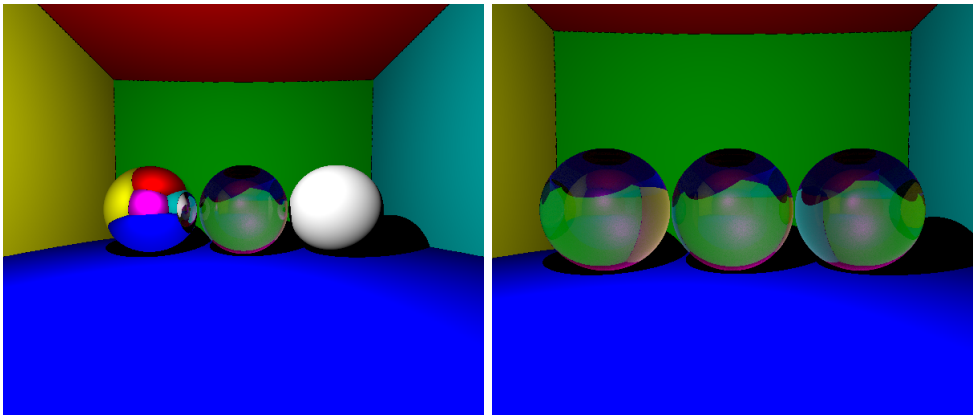

Figure 6: Fresnel law

at an intersection point `P`. An initial result can be seen in Figure 7, which uses 150 rays per pixel. The next step is to implement antialiasing. To do that, we use the `boxMuller` function which provides 2 Gaussian randoms, which are added to the x, y components of the main `Vector V` (pixel vector). As it can be seen in Figure 7, the antialiasing process smooths out the sharp edges even with a relatively small amount of rays per pixel.

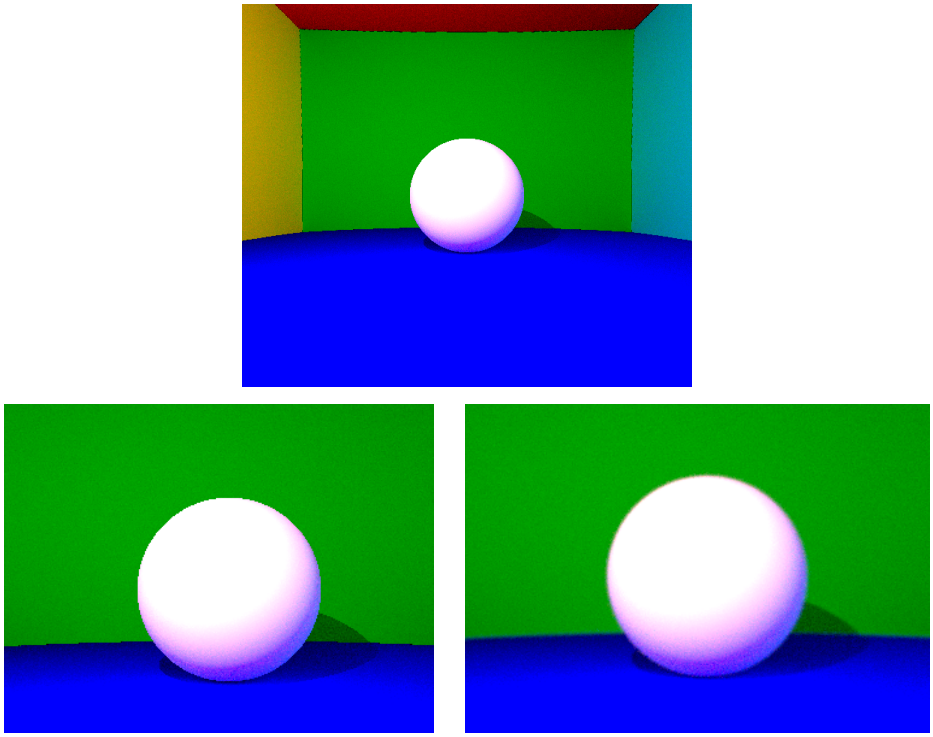

Figure 7: top: indirect lighting (150 rays), bottom: before and after antialiasing (150 rays)

5 Spherical light sources and depth of field

We now introduce spherical light sources. This consists in significantly modifying the `get_color` method to account for spheres which are sources of light and to modify the behaviour of the other types of spheres (diffuse objects especially). To implement the depth of field, modifications are made in the `path_tracer.cpp` file to account for camera shifts when sending out rays. It is worth

noting that from this point on I switched to a machine with 8 cores when computing the images.

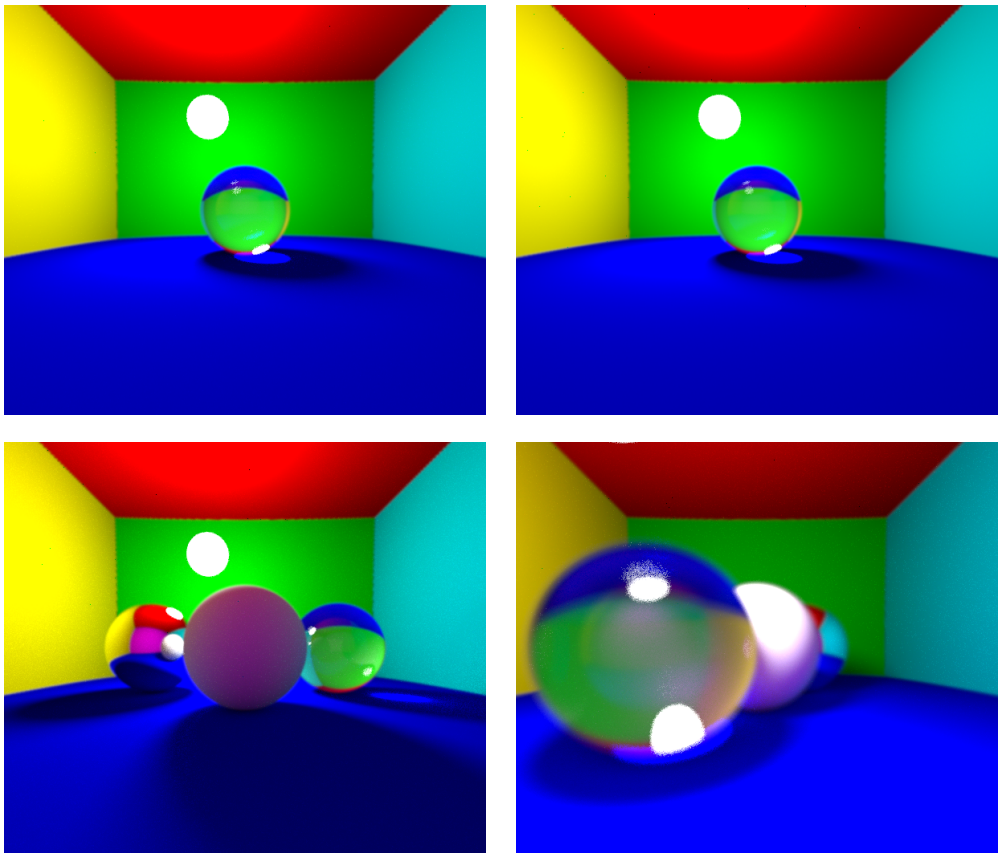

Figure 8: top: 2000 rays (under 12 minutes) and 6000 rays (32 minutes), bottom: 3000 rays (left, 18 minutes) and DoF with 2000 rays (right)

6 Mesh intersection

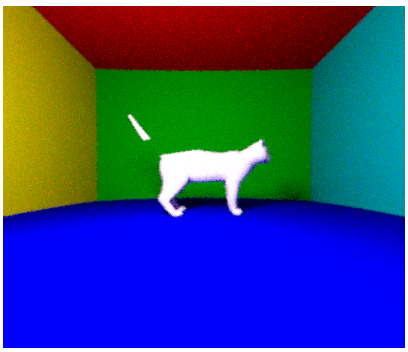

Figure 9: Ray-mesh intersection using 4 rays per pixel

To add meshes as well as spheres to the scene, we implement an abstract class `Geometry` which will be a parent class for both `Sphere` and `TriangleMesh` objects. After implementing the `intersect` method for the `TriangleMesh` class, we obtain the output in Figure 9. The cat is sadly missing half of its tail due to an unidentified bug.