Jules Lahmi BX25

Ray-Tracing Project Report

CS306 - Computer Graphics

1. Introduction

This project implements a basic renderer using path tracing. The original goal is to display objects in a 3-dimensional space, illuminated by a light and seen through a camera (both considered points in the 3D space). Then step by step we integrate multiple additions to the original renderer with the goal of better realism : shading, shadows, gamma correction, reflection, refraction and others.

Two major scenes were used to demonstrate the renderer:

- **A simple box with spheres, demonstrating reflection, refraction and shadows.**
- **A cat model as a triangle mesh.**

2. The basic Ray-Sphere implementation

The project is based on three classes, Rays, Spheres and Scenes. Rays simply have an origin and a direction, while spheres have a center, a radius and an albedo for defining color. Scenes are arrays of spheres. We first define a vector in the 3D space as the camera, and trace all possible rays from the camera towards the scene at a 60° angle. The equations of the ray are found from the pixel we will be writing the final color in. We find that the pixel (x,y) is at coordinates $(Qx + x + 0.5 - W/2, Qy + y + 0.5 - H/2, Qz - W/(2\tan(\alpha/2))$ with (Qx, Qy, Qz) the camera, and it gives us the equation for the ray we will be launching. Then for each ray, using an 'intersect' function looping over the scene, we check if it ever encounters a sphere, and find the closest sphere encountered by the function and its distance from the camera. Finally we find the albedo of the closest sphere, and return it to the pixel we are iterating over.

We can then easily add shadows and gamma correction using formulas given in the Lecture Notes. The result of these computations only depend on the previously mentioned parameters, as well as the light vector and its intensity.

3. Reflections, Refractions and Fresnel Law

This was a very hard part for me, especially refraction, which I somehow failed since my hollow ball is not working properly.
For pure reflection, we simply use Snell-Descartes Law and launch a ray bouncing off the mirror we encountered, and repeat the operation recursively until we either meet recursion depth limit or hit a non-mirror surface. We simply add a mirror parameter to our sphere class to check for this.
Refraction was harder, and also uses Snell-Descartes Law to bounce off the ray, but part of the ray continues its trajectory. I struggled a bit with both refraction and reflection on the sign of the offset I was supposed to add, so it took a lot of trial and error to get something correct. I also tried to implement

the hollow ball using the trick described in the Lecture Notes with the inverted normal. It looks decent but it is not exactly what it should be. (See figure 1).

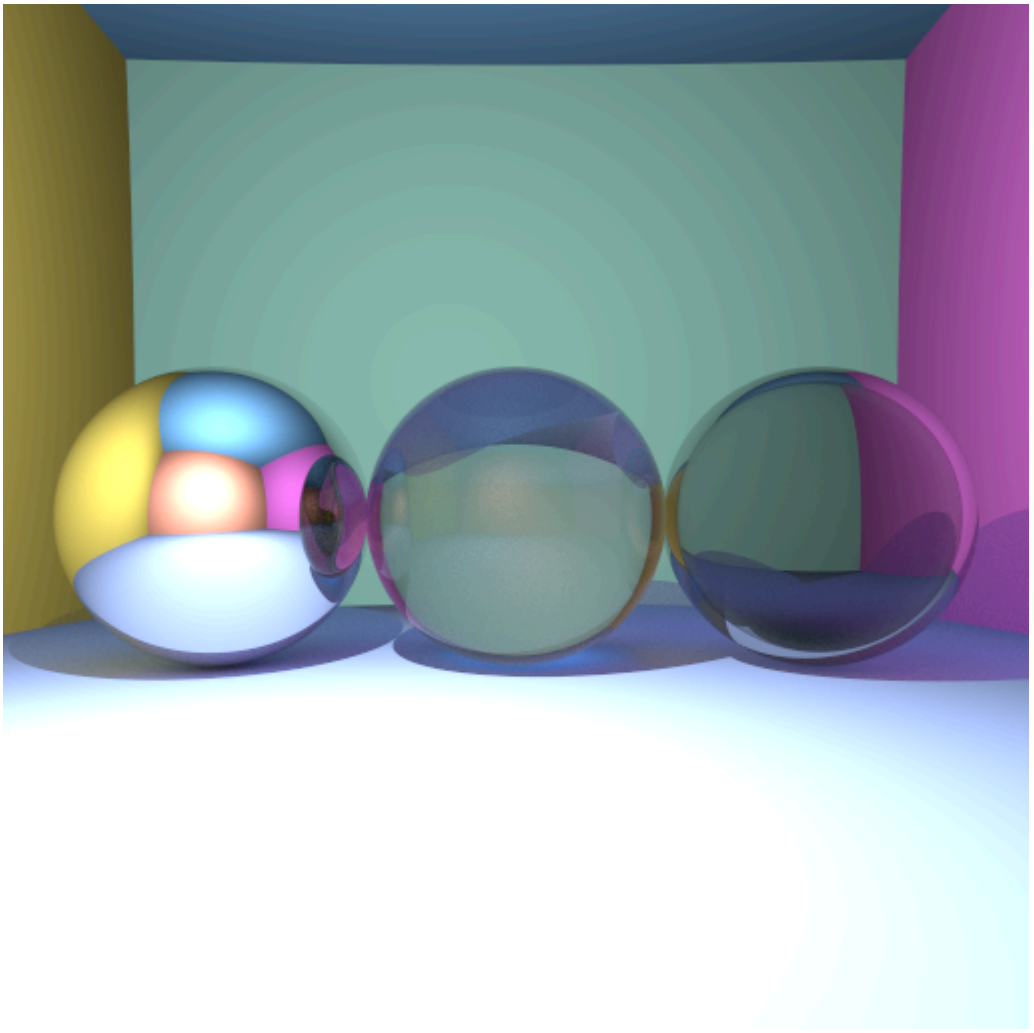

Figure 1 : Reflection, Refraction and Hollow Ball with Fresnel Law and Indirect Lighting

Indeed, I had to tweak both sphere's refraction indexes and sizes to obtain something about right, and after implementing Fresnel law, it doesn't work on the hollow sphere (See Fig 1).
For Fresnel Law, it affects refraction, and instead of only launching a refraction ray, we launch a large number of rays that are either refraction or reflection randomly (depending on Schlick's approximation). This gives us a partial reflection inside the refractive sphere of index 1.5.
I also struggled with Fresnel approximation as I had not implemented parallelization yet and the code took multiple minutes to run, and I thought I was making way too much computations for a long time. I debugged for a long time trying to reduce computations before realizing it was working fine.

4. Indirect Lighting and Antialiasing

Regarding indirect lighting, although the maths as they are described in the lecture notes are very complicated and unintuitive, they simplify quite nicely and the final implementation is relatively easy, as

described in equations 2.6 to 2.9, only needing to create two random variables and generate a random vector from the normal. We just do this for all points that are not in direct lighting. Then as we have implemented large sampling for Fresnel implementation, we can here reuse the code and launch multiple rays from the point, and average them out.
Since we are already launching multiple rays, anti-aliasing was also not that complicated : inside the main function, we simply launch multiple rays around the pixel with a small offset and average them out.

5. Triangles and Meshes

This part was very complicated since it meant a complete restructuring of the code. Although the Trianglemesh class was given, the Sphere and Scene Class still needed restructuring to be part of the virtual Geometry class, which includes triangles and spheres. This means that Scene is now a vector of pointers to Geometry objects, requiring a bit more careful handling when accessing the object (since we don't know whether it's a sphere or a triangle). Similarly, this also means that I had to say that mirror, refraction index and albedo were part of the geometry class and not the sphere class, since we can assume that some triangles will need refraction and reflection later (I spend some time implementing this while it was in fact never used, we only actually need the virtual intersect function from the Geometry class).
And the tough part was creating the intersect function for the triangle class using the maths given in the Lecture Notes.
The function also needed to be modified in Sphere, since I decided at this point that the function intersect would also have parameters P and N, and that they would be computed inside the function instead of after the function in Get_Color. This did not change much for the function in Sphere, but the maths for triangles were more complex so it made get_color a lot clearer, and reduced computation time by not calculating the distance t twice.
I will go over the new intersect function for TriangleMesh on the part on BVH since it requires iterating over the BVH tree.

6. Bounding box and BVH

I first implemented a very simple bounding box which was working using a BoundingBox class where we compute the min and the max of a Trianglemesh for each vector, and create a ray-bbox intersect function, and we can entirely skip any calculations if the ray does not intersect with the Trianglemesh.
Regarding the BVH, both building it at the beginning and iterating through it is very tricky.
To create the BVH, we first create a Tree BVH class to store it, as this will make it more simple to store.

I will first go over the code of the creation of the tree. First over the whole space we create a bounding box as we did earlier. We then find the longest axis of this large bounding box, and try to split the triangles with their barycenter regarding this axis. We create a get barycenter auxiliary function, and find the parameter of the barycenter corresponding to the axis we are splitting on. Then we simply apply the quicksort algorithm given in the Lecture Notes, and it works out very well, giving us a complete Tree over the whole 3D space.

Then our final piece of the puzzle for this very efficient optimization is to construct the intersect function between the ray from the camera and the BVH tree. First from the original intersect function we add the BVH, and we check recursively.
The easier part of this setup is simply the recursive part because we don't have to do the maths for Ray-Triangle intersections yet. We simply have to compute both sides of the BVH and recursively call intersect for the two parts, and return the one with the smaller distance.
The maths parts were very very tricky, even while copying from the Lecture Notes (equations 2.10 to 2.14). Our final goal is simply to return the closest intersection with each of the triangles of the BVH, and to compute P and N once we found it, but the maths to find check that we are well within the triangle (and not parallel) are a bit tricky.

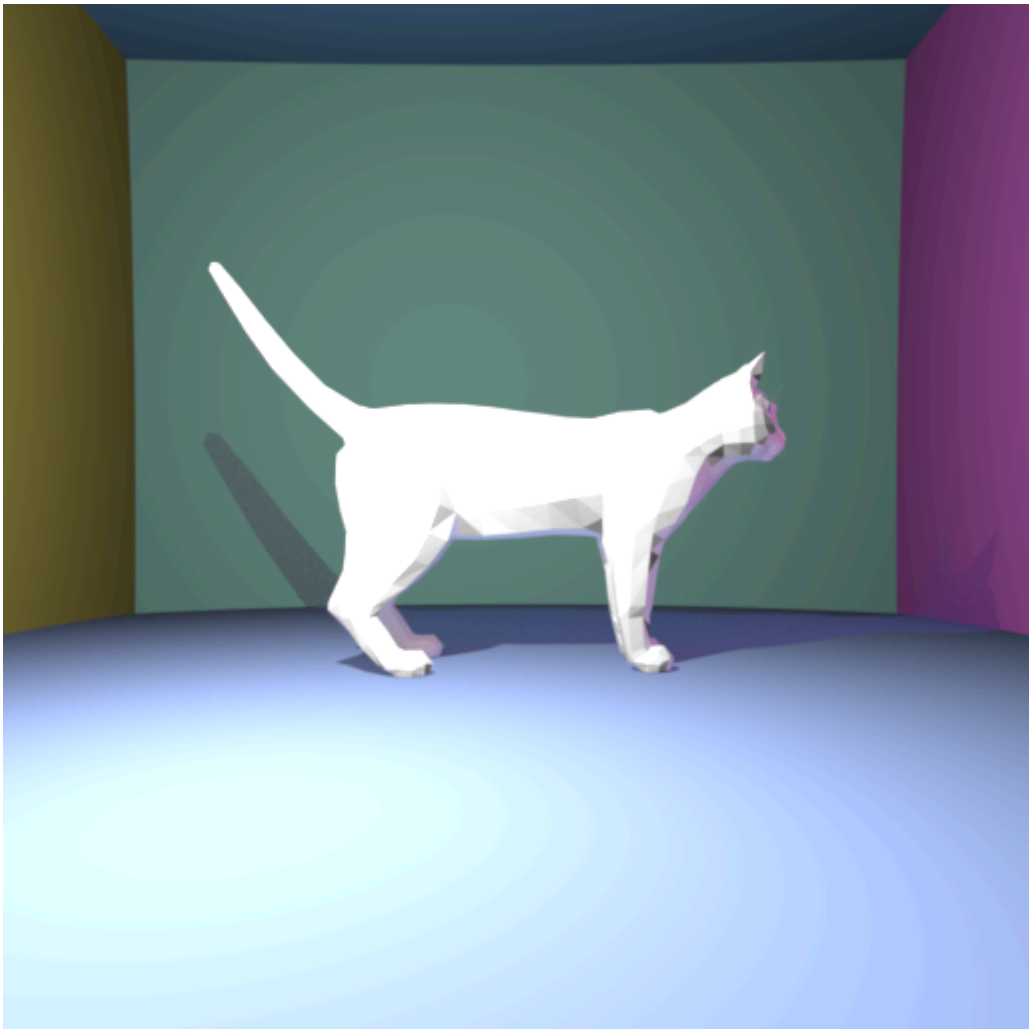

Figure 2 : Cat with indirect light and BVH tree implemented. (2 minutes runtime for 1000 rays per pixel and depth recursion 8)

6. Conclusion

To conclude, we manage to obtain balls with refraction and reflection, indirect light, antialiasing and Fresnel Law working, although the hollow ball is a bit dysfunctioning. Similarly, the cat without texture

looks very good, and all other parts of the code are still working. We have not implemented textures and soft shadows. Regarding computation time, in general, we are about 10-30% slower than the number written in the Lecture Notes. I assume there are some slight optimizations I did not do, or maybe I am missing some compiler optimizations. BVH has indeed greatly improved runtime, reducing it to an acceptable time and decent performance for 1000 rays per pixel and depth 8. I have also below added an image of the cat and a ball with refraction on the same frame to check both worked at the same time.

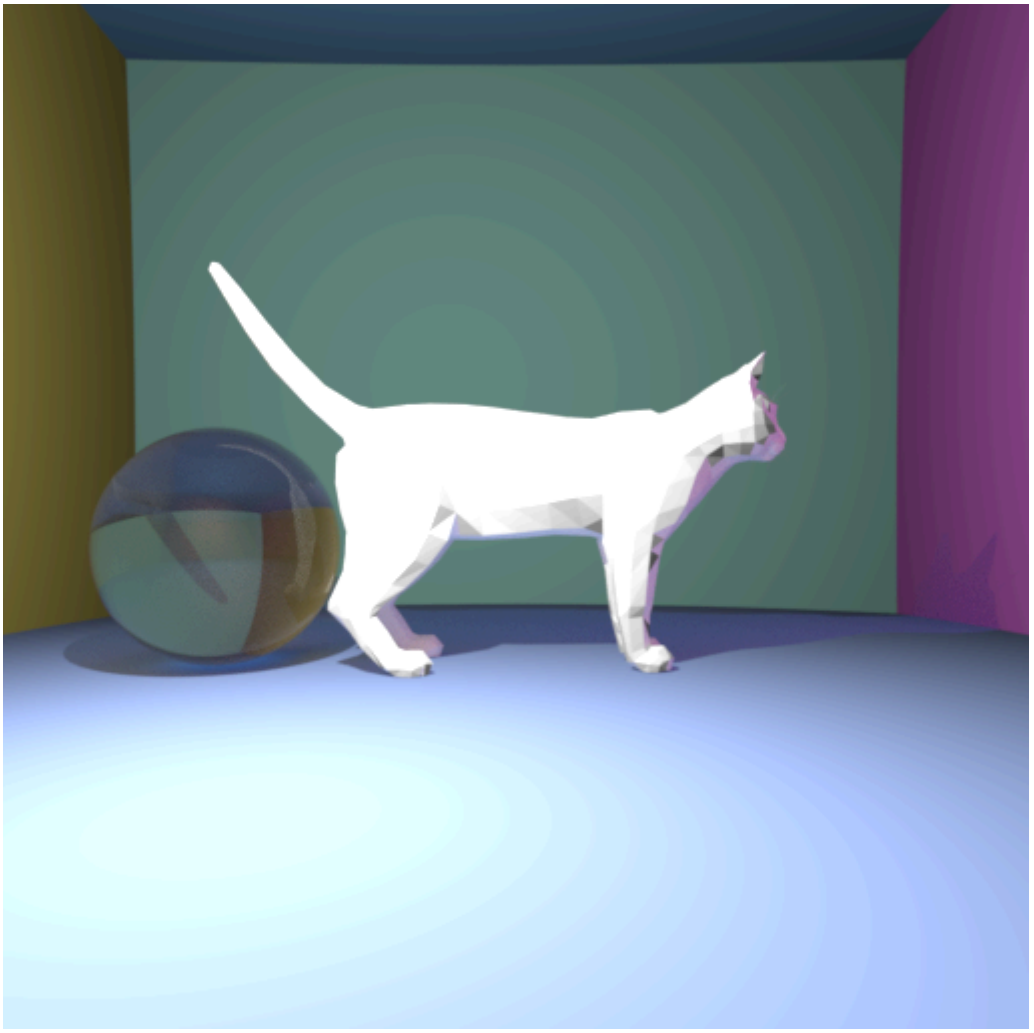

Figure 3 : Cat and refraction ball at the same time