Computer Graphics Raytracer

Bogdan Budura

May 2024

1 Introduction

This report summarizes my work over the first half of the semester, which was dedicated to building a pathtracer. I will introduce each feature I have implemented step by step, accompanied by images and execution times. Additionally, I will discuss the challenges I faced while implementing these features and the lessons I learned along the way.

2 Difuse and mirror surfaces

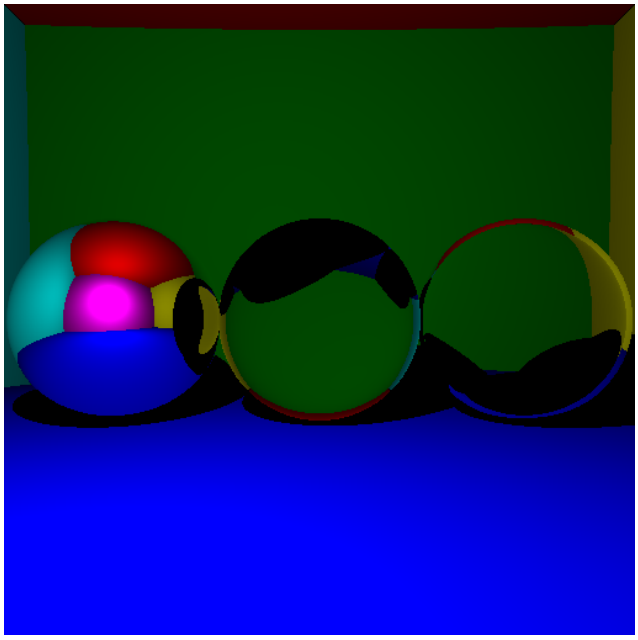
We begin by introducing a simple scene composed of six walls, each painted a different color: green, blue, red, yellow, cyan, and purple. Additionally, we add three spheres to this scene: one reflective and two refractive (one filled and one hollow). For the first test, we will examine how light bounces off their surfaces and interacts with the surrounding scene without any special effects. For this part, we will use 6 light bounces and one sample per pixel. This test was executed in parallel with optimization flags set to `-O3`, achieving an execution time of 35ms.

The initial results are impressive for a starting point, and I must admit I was genuinely amazed at first. However, it is time to enhance the scene further. For the next part, we will augment the pathtracer by adding the Fresnel Law, which makes our surfaces partially reflective and refractive. To simulate this, we sample rays for each pixel and uniformly absorb rays larger than a specified parameter RR; otherwise, we reflect them. By averaging this sampling over 1000 rays, we obtain the image presented in Figure[1b], rendered in approximately 13 seconds.

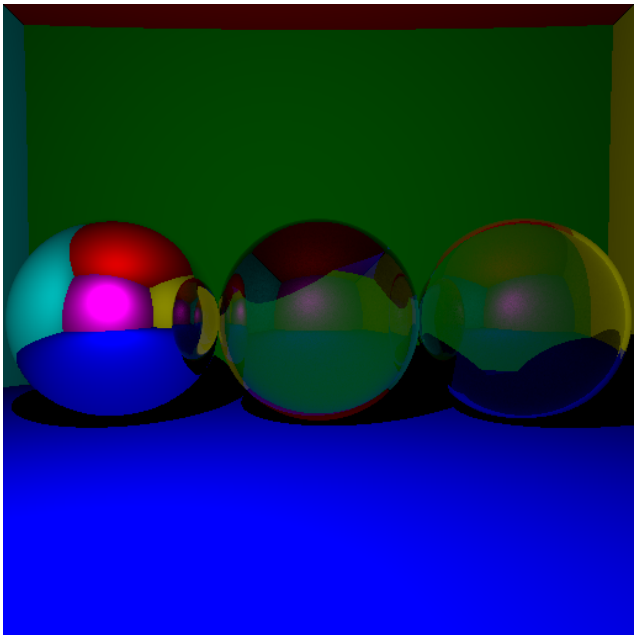
3 Indirect Lightning

For the next part of my raytracer, I implemented indirect lighting, which gives the raytracer its true raytracing capabilities. Building on the idea of the Fresnel law, we must randomly sample directions to account for lighting. In practice, any diffusive surface will reflect some amount of light, which we need to account for. However, there are an infinite number of incoming directions. Therefore, we must sample some directions and then recurse over each direction to compute its light incidence until we have exhausted our light bounces. By sampling again over 1000 rays per pixel, we obtain the image in Figure[2a] in 58.5 seconds.

Although raytracing looks incredible at lower resolutions, we can begin to observe slight visual aberrations called aliasing. This discrepancy arises because pixels are not literal points but small rectangles that are intersected at their centers. Consequently, two consecutive pixels might have a significant distance between their centers, resulting in different colors, which in turn looks unpleasant. The easiest workaround is to sample the intersection point within the entire pixel surface. By doing so, we can



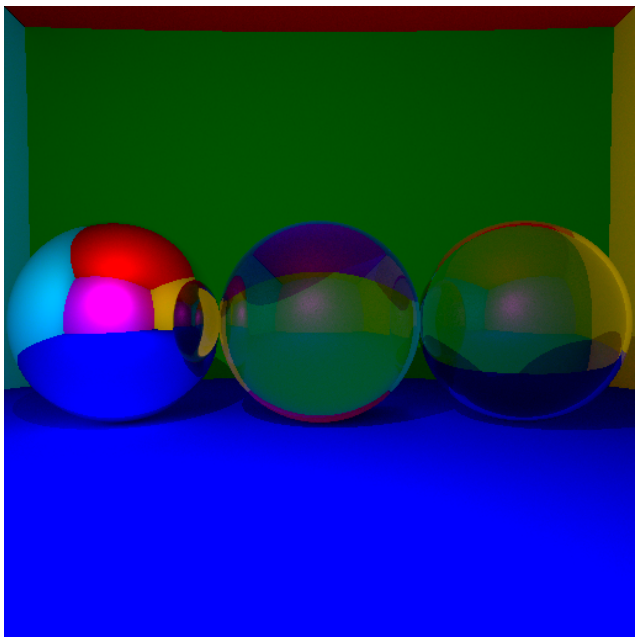
(a) Three Mirrors



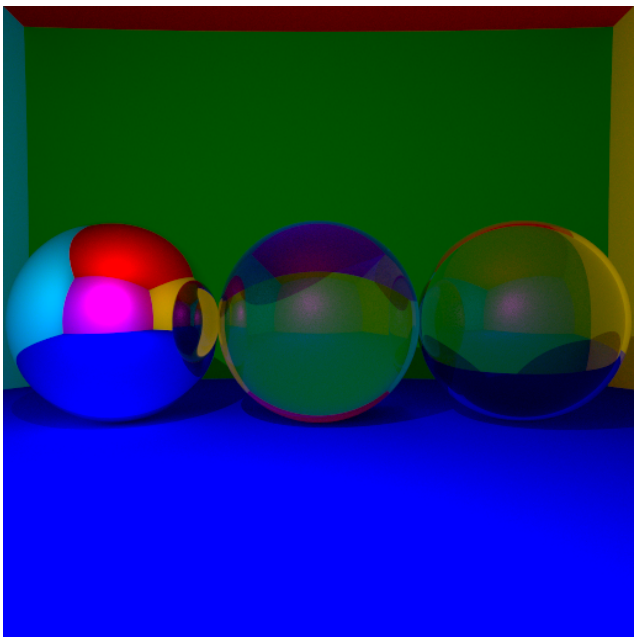
(b) Fresnel Law

Figure 1: Reflective and Refractive Surfaces

average out the resulting color and allow for smoother transitions between pixels. Figure[2b] shows the result of indirect lighting combined with anti-aliasing, sampled over 1000 rays and rendered in 1 minute and 5 seconds.



(a) Raytracing

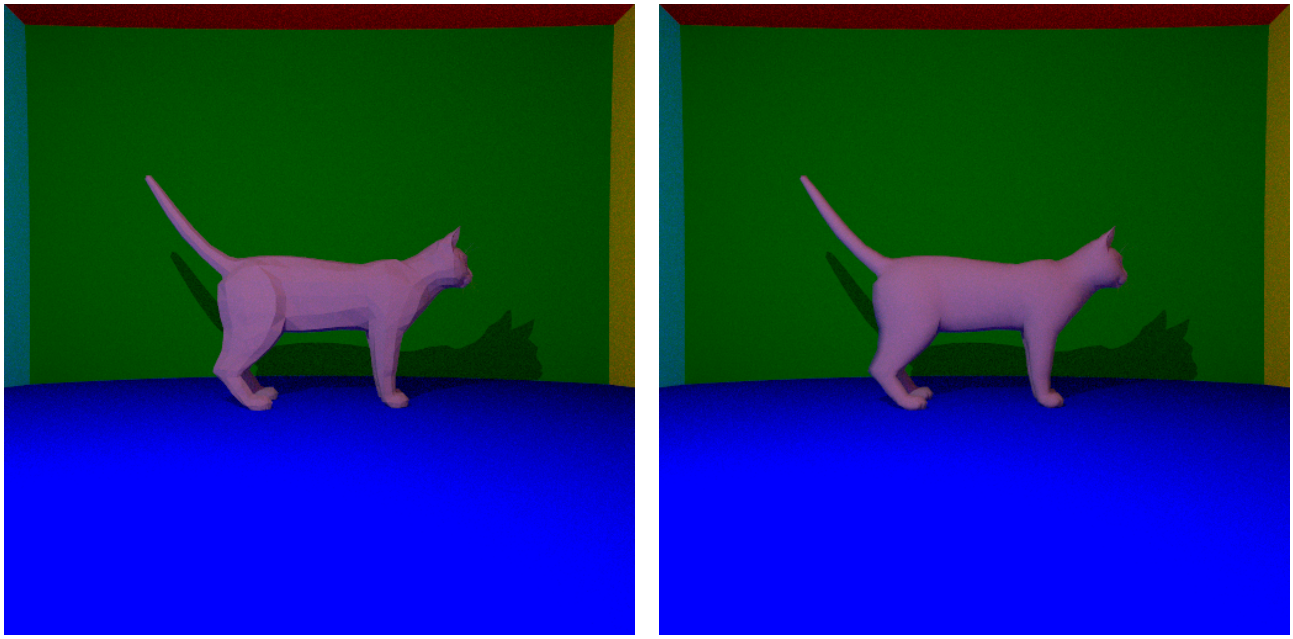


(b) Anti-Aliasing

Figure 2: Indirect Lightning

4 Cat mesh

Time to run 22.5s 64 samples per pixel 6 light bounces The initial setup with spheres was a pleasure to work with, but it has its limitations. There is only so much you can do with simple spheres. Thus, the next phase of my raytracer involves handling more complex objects, such as animals. However, such complex objects cannot be represented directly, so we use approximation structures like triangle meshes. To achieve a clear and crisp representation of an object, the approximation must be very detailed, which is computationally intensive. For instance, rendering the object shown in Figure [3a] without any auxiliary structures took over 40 minutes at 64 samples per pixel, which is both impractical and frustrating to test and work with. This test, with 64 samples per pixel and 6 light bounces, took 22.5 seconds to run.



(a) Cat mesh. No Interpolation

(b) Cat mesh. Normal interpolation

Figure 3: Triangle Mesh

Nevertheless, there are *acceleration structures* that can be extremely useful when working with complex objects. The first and most immediate approach is to enclose the object within a large rectangular box called a *bounding box*, which is straightforward to represent in a program. Furthermore, if a ray does not intersect this bounding box, it will never intersect the object inside it. By employing this optimization, we reduced the initial staggering runtime from 40 minutes to just 10 minutes.

However, we can achieve even better performance. The next step involves using a structure called a Bounding Volume Hierarchy, or BVH. The idea behind BVH is to first enclose the main object in a bounding box and check for ray intersections. If the ray intersects this bounding box, we then create two smaller bounding boxes that attempt to split the object in half. We check for ray intersections with these smaller bounding boxes. This process is recursively applied, deferring the computationally expensive intersection tests with the actual triangles to much simpler intersection tests with progressively smaller bounding boxes.

Although there is an initial overhead to create the BVH, the benefits are substantial. I have presented two images in Figure[3]. Both represent a cat; however, the one in Figure[3b] uses normal

interpolation to smooth out rough edges. Using BVH, I was able to generate these images in 9.65 seconds with 64 samples per pixel, drastically reducing the previous runtime of 10 minutes.

5 Conclusion

The raytracer project was an immensely enjoyable though it came with its fair share of challenges. The initial hurdle was adapting to a completely new programming environment, but the most significant difficulties arose during debugging. A single computational error could cause inexplicable issues in entirely different parts of the code, making the debugging process both intricate and time-consuming.

The most challenging aspect was implementing the BVH. Initially, my BVH was performing worse than having no BVH code at all. I spent countless hours debugging and running very slow code, only to realize that I had not initialized the minimum and maximum bounds within the BVH structure. This oversight resulted in completely erroneous bounding boxes. Additionally, I encountered inexplicable segmentation faults due to not initializing the BVH pointers to null, and the list goes on.

However, despite these setbacks, the project provided invaluable lessons in both programming and problem-solving. The satisfaction of overcoming these challenges and achieving efficient, high-quality renders was well worth the effort.