CSE306: Computer Graphics - Project report 1

Mihaila Victor Matei

May 14, 2024

Contents

1 Introduction

In this first project we worked on implementing a ray tracer model that closely resembles naturally occurring images through replicating the behaviour of rays of light while using some mathematical tricks to reduce computational load.

Some of the important features included in this project are the simulation of phenomena such as reflection or refraction, shading and indirect lighting, used on spheres and triangle meshes, with bounding boxes and bounding volume hierarchies to make rendering faster. These features as well as their associated impact on execution time are observable in the renders presented in this report.

The program was executed on a Intel(R) Xeon(R) W-1270P CPU @ 3.80GHz CPU (school computer rooms CPU) and the timings associated to the different renders are specific to this system. That being said, increases or decreases in execution time associated to certain features (for example increase for indirect lighting, decrease for BVH) should occur on any CPU.

2 Code overview

The code for producing the included renders is available at (insert link). The project was done in C++ and compilation with OMP allowed for parallelization which was useful for faster testing of the program.

The code uses several classes that I will briefly present in this section. First, the provided Vector class from tutorial 1 serves as the building block for all components of the render (coordinates, colors and more). This class implements three dimensional vectors and their corresponding operators. Second, the Geometry class and its two sub-classes Sphere and Triangle Mesh. As stated in the introduction, our renders handle spheres and triangle meshes, which are treated similarly as members of a "Scene" but behave differently with respect to how they intersect light. Some of the interesting features of the Sphere class are the possibilities to create reflective, refractive and transparent spheres, solid colored spheres, and even hollow spheres, constructed as two nested spheres as suggested in lecture 1. The TriangleMesh class includes a function provided in tutorial 3 for reading obj file of such meshes, and also functions for determining and intersecting bounding boxes and also the implementation of BVH using an additional Node structure. All sections of the project implement the mathematical formulas indicated in the lecture notes for lectures 1-3. Finally, the Scene class is responsible for linking all elements of the render together. It stores all "geometries" (spheres and meshes) and simulates how light intersects and interacts with them. All scenes initially contain 4 walls, a floor and a ceiling in the form of 4 solid, colored, massive spheres, big enough to resemble the behavior of flat surfaces.

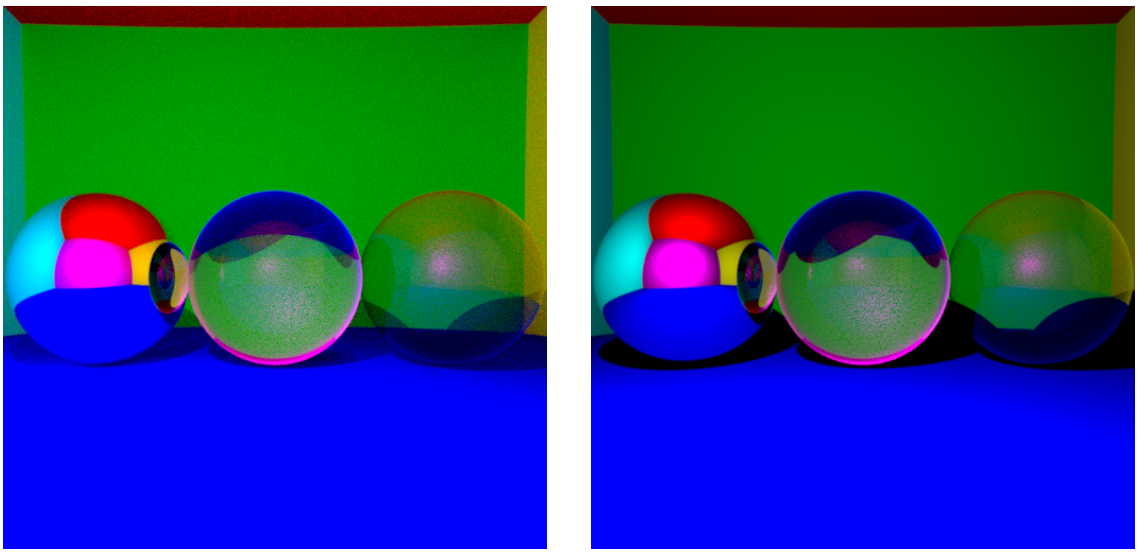
3 Part I: Spheres

In the first part of the project, we were only concerned with handling spheres and phenomena such as reflection, refraction or indirect lighting.

The renders below try to replicate the elements from slide 42 of lecture 1, namely a reflective sphere on the left, a refracting sphere in the middle, and a hollow sphere on

the right, constructed as two nested spheres at the same position, with slightly different radii.

The inclusion of indirect lighting, as expected, introduces a considerable increase in execution time. Renders using indirect lighting or not, and their associated computational time, can be found below. For these renders, I used 1000 rays per pixel and a max ray depth of 5 bounces, with a gamma correction of 2.2 and an angle of 60 degrees.

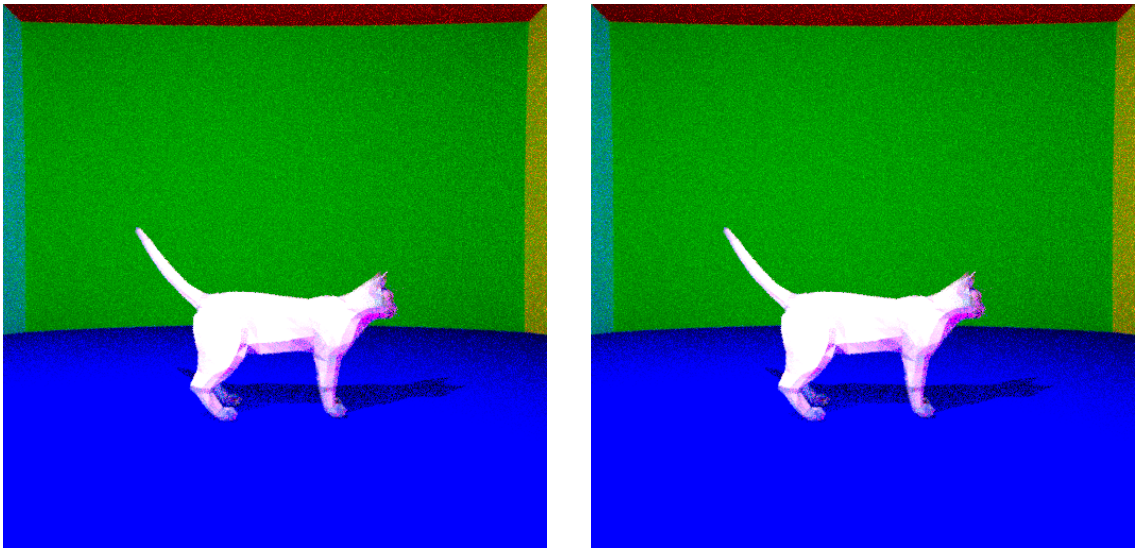


(a) Spheres with indirect lighting, 73103ms (b) Spheres without indirect lighting, 16901ms

Figure 1: Comparison of results and execution times for the rendering of the three spheres with and without indirect lighting, 100 rays per pixel

4 Part 2: Triangle meshes

Tutorials 3 and 4 were concerned with handling triangle mashes, useful for displaying more complex shapes (in our case, a cat). To cope with the increased computational load, we use bounding volume hierarchies(BVH) to store triangles in a tree structure based on where they are located with respect to an axis of choice, which effectively discards parts of the mesh that are too far. This proved to work a lot better than simply intersecting all triangles. Figure 2 shows the comparison of results and execution times between using BVH or not.

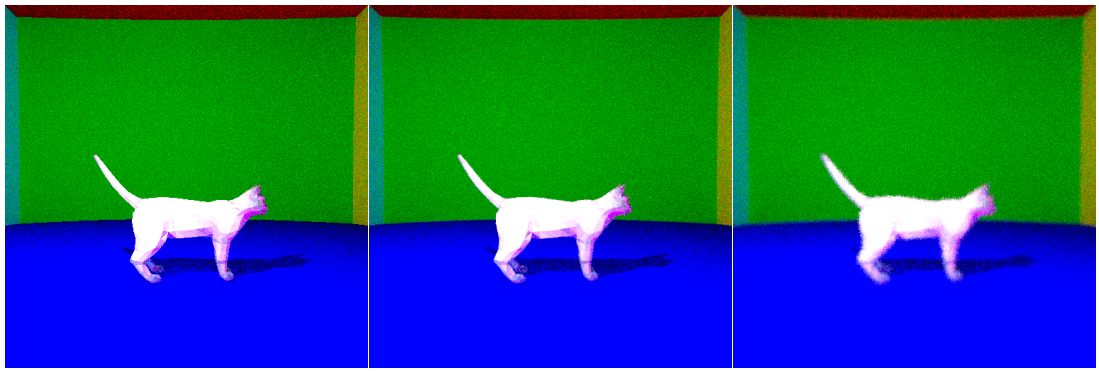


(a) Without BVH, 216537ms (b) With BVH, 8987ms

Figure 2: Comparison of results and execution times for cat mesh, 10 rays per pixel

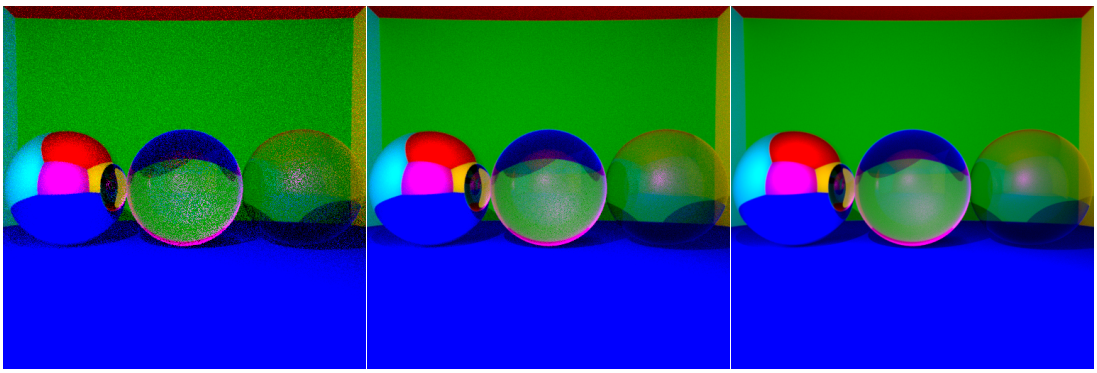
5 Extras: Playing around with parameters

In this section, I wanted to include a comparison of results for different values of the rays per pixel parameter, as well as a comparison of images with different antialiasing parameters. Additionally, I included a render that combines the sphere and triangle mesh components. It is important to note the much higher execution time of more qualitative renders, as it highlights the need to find a good balance between image quality and computational requirements.



(a) $\sigma = 0$ (b) $\sigma = 0.5$ (c) $\sigma = 2$

Figure 3: Comparison of antialiasing for standard deviations from 0 (no antialiasing) to 2 (blurriness)), 10 rays per pixel



(a) 10rpp, 1693ms (b) 100rpp, 73103ms (c) 1000rpp, 904984ms

Figure 4: Comparison of execution times for different numbers of rays per pixel, indirect lighting enabled in all cases

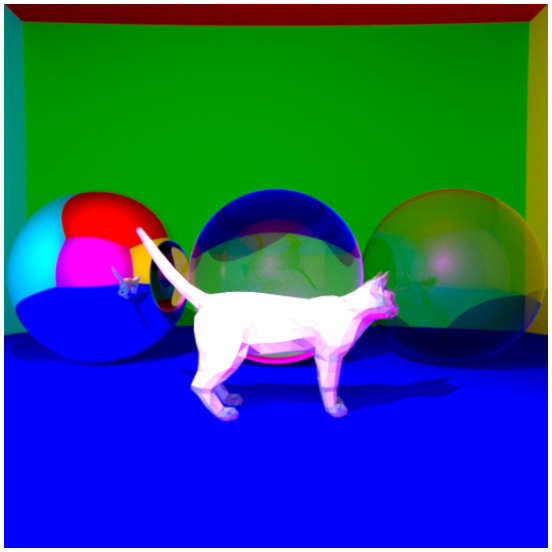


Figure 5: Cat mesh with a reflecting, refracting and hollow spheres, 100 rays per pixel, 151529 ms; The cat reflects nicely on the leftmost sphere