# CSE306 Ray Tracer

Vrushank Agrawal

May 2023

## 1 Introduction

This Ray tracer project is implemented in C++ with the following elements: basic spheres, reflection and refractions (mirror, transparent, hollow spheres), indirect lighting, antialiasing, triangle meshes, bounding box and BVH. The rendered images in the report are 512 $\times$ 512 pixels. The maximum ray depth is 5, the gamma correction factor is 2.2, and the camera is positioned at (0, 0, 55). Further, my laptop uses the 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz with 8 logical processors.

## 2 Code

The entire code is available in one file which is the *main.cpp* file. In this file, the Vector class provides basic operations for vectors, including dot and cross products, as well as norm and normalization. The Intersection struct stores information about a point of intersection between a ray (stored as a Ray class object) and a surface, including the point P, normal N, color, refractive index, and reflection information. The Geometry class defines an interface for objects that can be intersected with a ray, and the TriangleMesh class extends Geometry by defining a mesh of triangles. The mesh is stored as a list of vertices and indices in the TriangleIndices class object, and a bounding box (stored as a BoundingBox class object) is computed for the entire mesh to speed up intersection testing.

The program defines a scene by creating instances of the Geometry class, and a Scene class is responsible for computing the rays to be traced for each pixel of the image. The main loop iterates over each pixel computes the corresponding ray and finds the closest intersection with the scene geometry. If an intersection is found, the colour at the intersection point is computed, taking into account the surface properties, such as reflection, refraction, total internal reflection, indirect lighting, and antialiasing. Finally, the colour is accumulated in a buffer, which is written to a PNG file using the stb_image_write library.

The implementation uses OpenMP to parallelize the rendering loop, allowing for faster rendering of large scenes. The TriangleMesh class also includes a bounding volume hierarchy (BVH) to speed up intersection testing.

## 3 Renders

This section will discuss the different features of the project by showcasing specific renders for each feature. All the renders were implemented after an O3 optimization.

### 3.1 Basic Spheres

The figure 1 is a render of three spheres where each sphere is hollow, refractive (index=1.5), and a mirror from left to right respectively. The render took around 133.377 seconds, had antialiasing, indirect lighting, and parallelization switched on and used 1000 rays per pixel.

**Note:** In the rest of the report, we will use *rpp* as an acronym for rays per pixel.

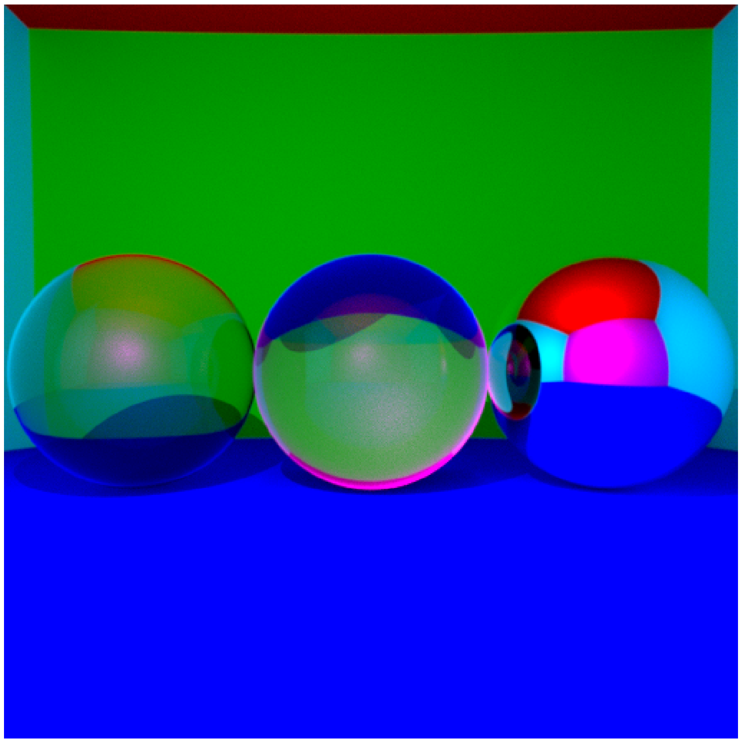
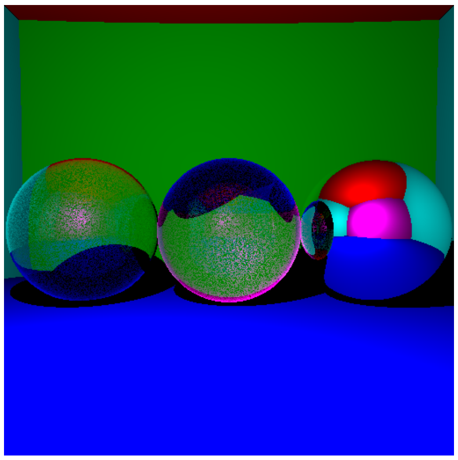


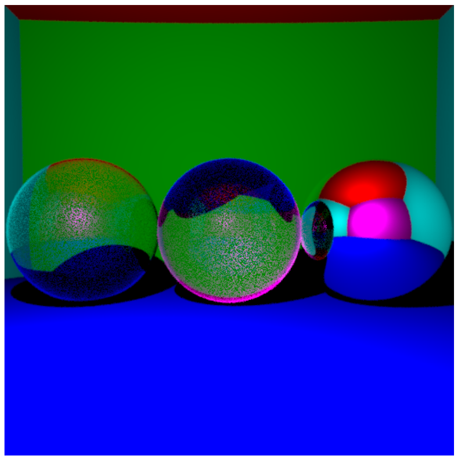
Figure 1: 133.377 Basic spheres, 1000 rpp

### 3.2 Antialiasing

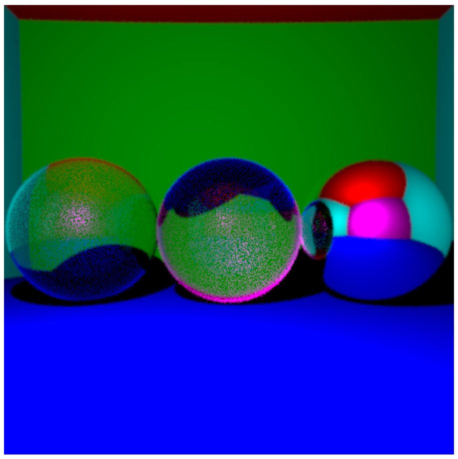
In these renders, parallelization was used but no indirect lighting was implemented and the images were produced with 20 rays per pixel (rpp). The renders were experimented with different standard deviations (STD) for the Box-Muller algorithm specifically with STD=0, 0.5, 1, and 5. Out of these, the smoothest renders seem to come with an STD of 0.5 as seen in figure 2b while a higher STD of 5 in figure 2d produces a lot of blurring. An STD of 0 is the scenario where there is no antialiasing as in figure 2a while an STD of 1 in figure 2c produces a smooth but somewhat blurry image compared to the one rendered for STD of 0.5. All the renders were produced in under 1 second and took a similar time.



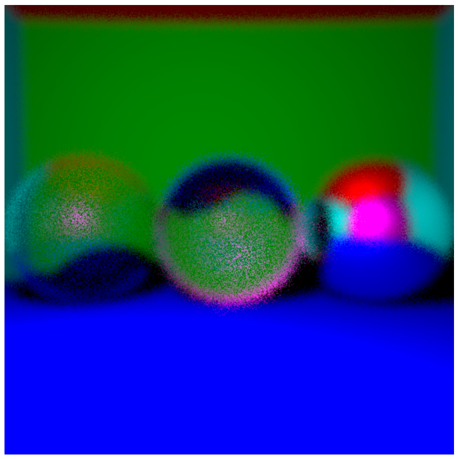
(a) 0.870s STD=0



(b) 0.871s STD=0.5



(c) 0.999s No STD=1



(d) 0.835s STD=5

Figure 2: Antialiasing with different STD for 20 rpp.

### 3.3 Indirect Lighting

In this render, antialiasing was switched on but no parallelization was done. We see that the indirect lighting render in figure 3b takes roughly 9.678 seconds while the render in figure 3a takes only 1.947 seconds for 20 rays per pixel.

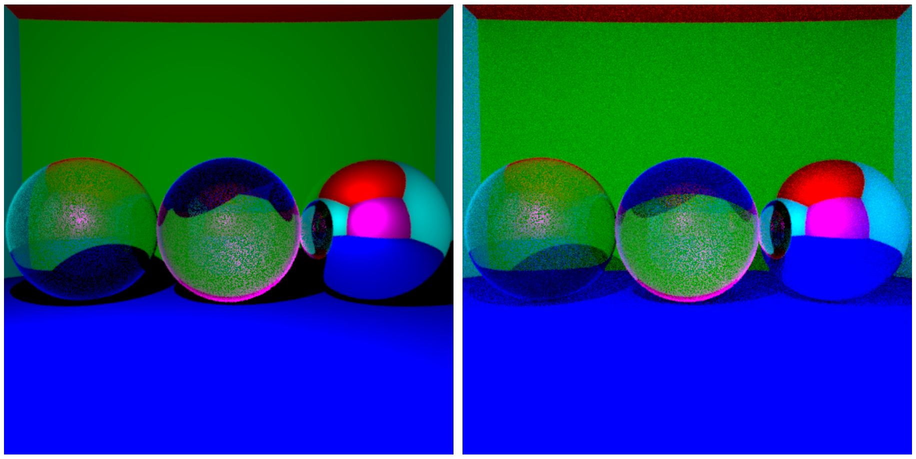

(a) 1.947s Direct Lighting (b) 9.678s Indirect Lighting

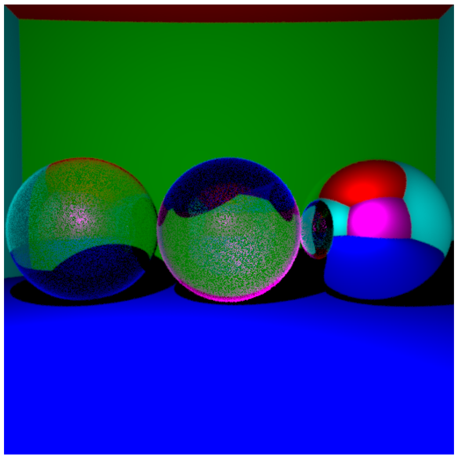
Figure 3: Direct vs Indirect lighting for 20 rpp.

### 3.4 Paralellization

In this render, indirect lighting was switched off and antialiasing was switched on. We used OMP library for parallelization by compiling the code with the *-fopenmp* and *-lpthread* libraries. As a result, we see that the parallel render in figure 4b is completed in 0.691 seconds compared to the render in figure 4a which took 1.947 seconds or roughly three times more for 20 rays per pixel (rpp).

### 3.5 Triangle Mesh

In these renders, antialiasing, indirect lighting, and parallelization were switched on. In figure 5a where only a single bounding box was implemented with 64 rpp, the rendering time was 733 seconds or roughly 12 minutes, whereas, with the BVH in figure 5b, the image took 36.752 seconds to render.


(a) 1.947s Sequential

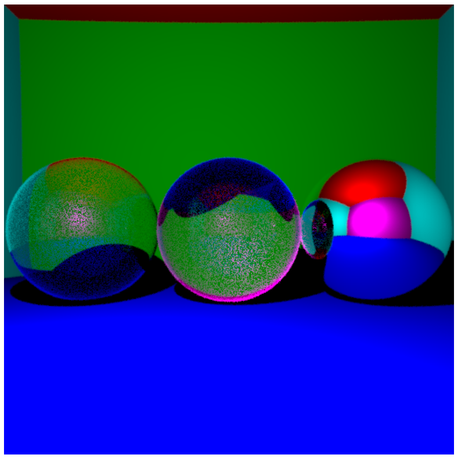
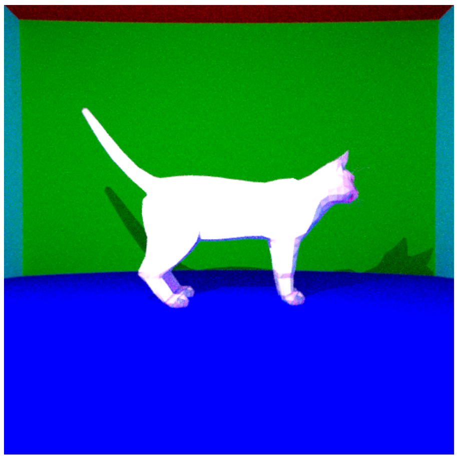

(b) 0.691s Parallel

Figure 4: Parallelization for 20 rpp.


(a) 773.674s without BVH

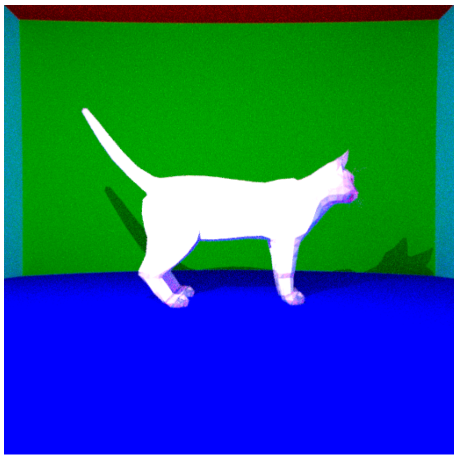

(b) 36.752s with BVH

Figure 5: Cat Mesh for 64 rpp.