1 Introduction

The primary goal of this assignment was to develop a raytracer capable of rendering 3D scenes with realistic lighting and shading effects. To achieve this, I implemented various features such as diffuse and mirror surfaces, direct lighting and shadows, indirect lighting, antialiasing, and ray-mesh intersection with Bounding Volume Hierarchy (BVH) acceleration.

The implemented features encompass:

1. Surfaces with diffuse and mirror properties
2. Point light sources that cast direct lighting and shadows
3. Indirect lighting for point light sources, without employing Russian roulette
4. Techniques for antialiasing
5. Accelerated ray-mesh intersection using BVH

ALL of the images below except the cat have the following config:

```
int imageWidth = 1024;
int imageHeight = 1024;
Vector camera = Vector(0, 0, 55);
double fieldOfView = degToRad(60);
double gamma = 2.2;
int maxDepth = 20;
int raysPerPixel = 600;
```

2 Diffuse and Mirror Surfaces

The Phong reflection model served as the basis for simulating diffuse and mirror surfaces in this project. Diffuse surfaces scatter incoming light uniformly in all directions, resulting in a matte appearance, while mirror surfaces reflect light in a single direction, creating specular highlights.

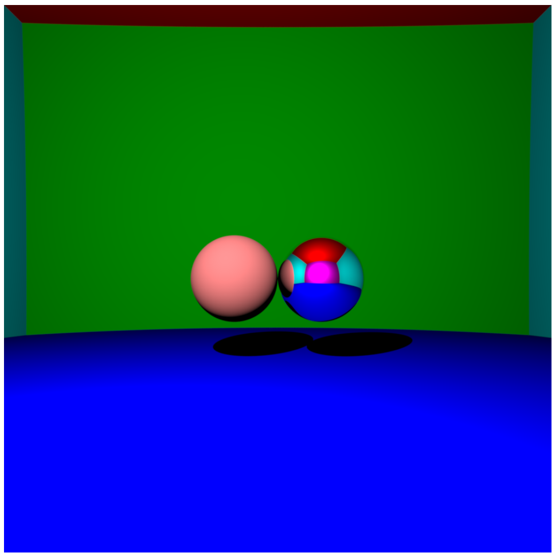


Figure 1: This image took 16734ms to render

For reflective surfaces, a reflected ray is generated and traced recursively to determine the color. In the case of diffuse surfaces, the direct lighting contribution is calculated based on factors such as light intensity, surface color, and the angle between the light direction and the surface normal.

3 Direct Lighting and Shadows

To implement direct lighting and shadows for point light sources, the visibility of the light source from the intersection point is determined by casting a ray from the intersection point towards the light source and checking for any occluding objects.

The direct lighting contribution is computed using the following formula:

$$\text{color} = \frac{\text{lightIntensity}}{4\pi\text{distance}^2} \cdot \frac{\text{surfaceColor}}{\pi} \cdot \text{visibility} \cdot \max(0, \text{dot}(\text{lightDirection}, \text{surfaceNormal}))$$

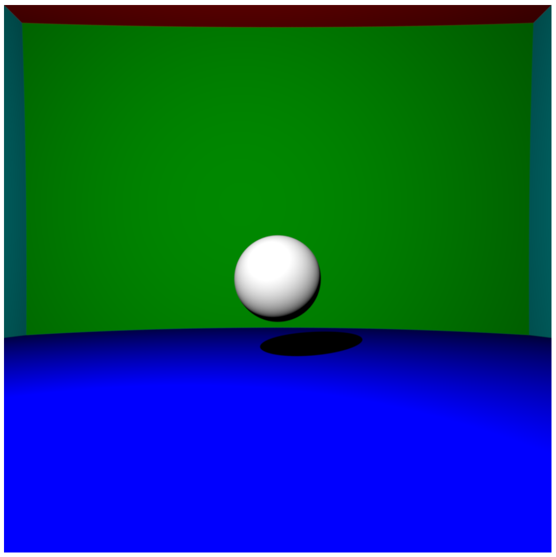
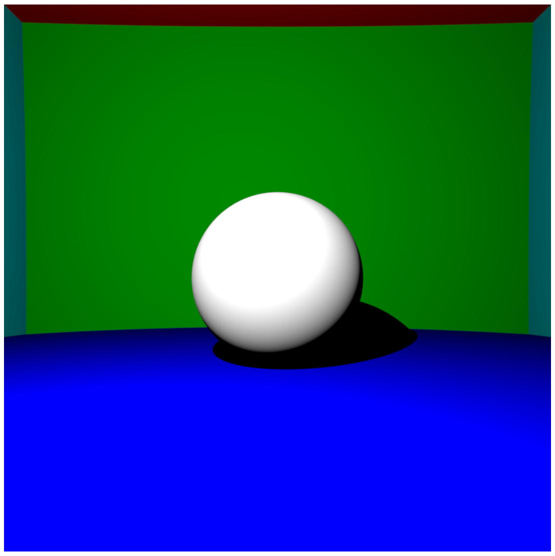


Figure 2: This image took 16635ms to render

Where `lightIntensity` represents the intensity of the point light source, `distance` is the distance between the intersection point and the light source, `surfaceColor` denotes the color of the surface, `visibility` is either 1 (illuminated) or 0 (shadowed), and `dot(lightDirection, surfaceNormal)` calculates the cosine of the angle between the light direction and the surface normal.

4 Indirect Lighting

The bouncing of light between surfaces, resulting in a more realistic and globally illuminated scene, is simulated through indirect lighting. This is achieved by shooting random rays from the intersection point and recursively tracing them to gather incoming light from other surfaces.


(a) This image took 2866ms to render without indirect lighting.

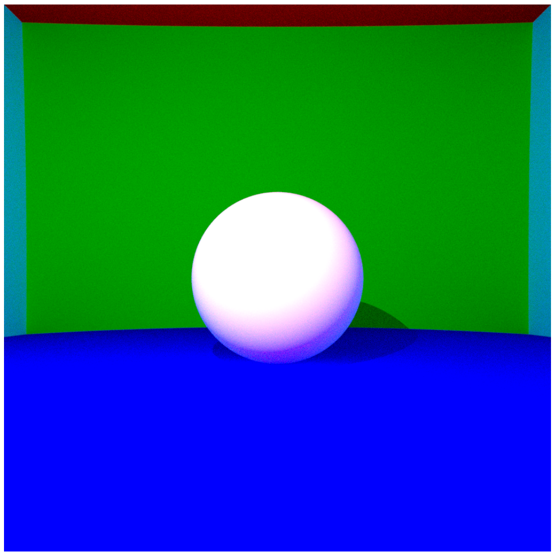

(b) This image took 64851ms to render with indirect lighting.

Figure 3: Comparison of rendering with and without indirect lighting

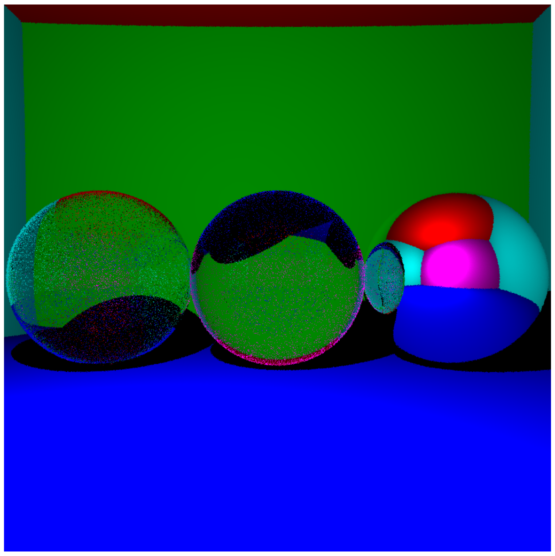
The indirect lighting contribution is obtained by multiplying the surface color with the color obtained from the recursively traced random ray. This process is repeated for a fixed number of bounces, determined by the `depth` parameter, to capture multiple light bounces.

The `randomCosineDirection` function generates a random ray based on the cosine-weighted distribution around the surface normal. Tracing this random ray yields a color that is multiplied with the surface color and added to the pixel color.

5 Antialiasing

Supersampling, where multiple rays are shot per pixel and their colors are averaged, is employed to reduce the appearance of jagged edges (aliasing) in rendered images.

Random offsets for each ray within the pixel's area are generated using the Box-Muller transform, which produces random numbers with a Gaussian distribution.


(a) This image took 61ms to render. 1 Ray

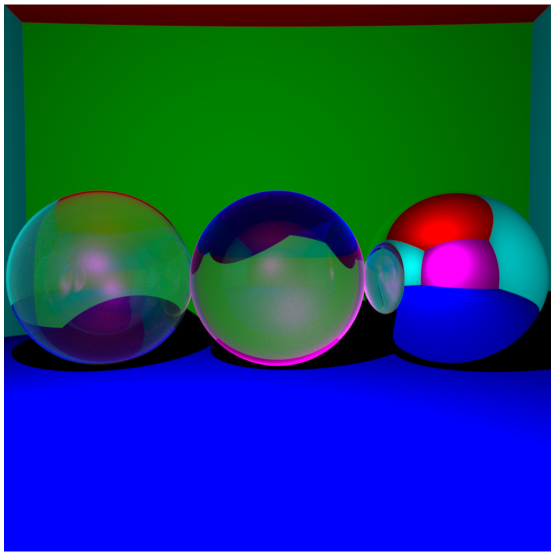

(b) This image took 27711ms to render. 100 Rays

Figure 4: Comparison of rendering with and without antialiasing

The number of rays shot per pixel is determined by `raysPerPixel`. For each ray, random offsets (`offsetX` and `offsetY`) are generated using the Box-Muller transform, and the pixel position is adjusted accordingly. The colors of all the rays are accumulated and averaged to obtain the final pixel color.

6 Ray Mesh Intersection and BVH

The Möller–Trumbore ray-triangle intersection algorithm is implemented to render meshes in the raytracer. This algorithm efficiently determines whether a ray intersects a triangle by solving a system of linear equations and calculating the barycentric coordinates of the intersection point within the triangle.

To accelerate the rendering process for large meshes, a Bounding Volume Hierarchy (BVH) is employed. The BVH is a tree-like structure that hierarchically subdivides the mesh into smaller bounding volumes, with leaf nodes containing subsets of mesh triangles and internal nodes representing the bounding volumes of their child nodes.

The BVH is constructed by recursively splitting the mesh triangles based on their spatial distribution, choosing the splitting plane along the longest axis of the bounding box and partitioning the triangles based on their centroid position relative to the splitting plane.

During the intersection process, the BVH is traversed from the root node, recursively checking child nodes for intersection if a ray intersects the bounding volume of a node. This hierarchical approach allows for quick discarding of large portions of the mesh that are not intersected by the ray, significantly reducing the number of ray-triangle intersection tests.

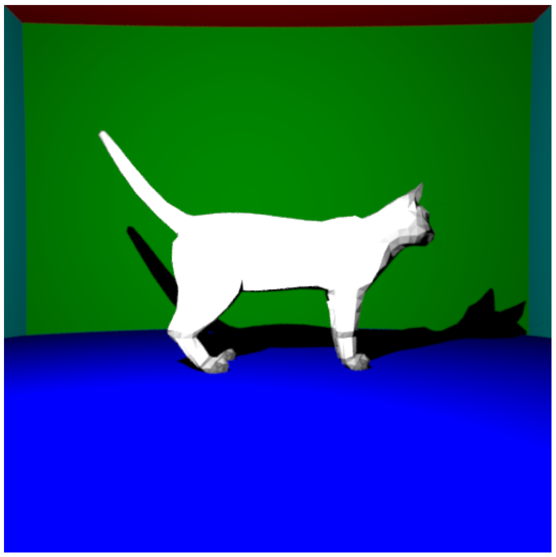


Figure 5: This image took 111199ms to render with: 64 raysPerPixel, 5 maxDepth, 512x512 dimensions.

Compared to testing the ray against every triangle in the mesh, the BVH implementation provided a $\sim 7\times$ improvement in rendering times.

7 Additional Features (Optional)

In addition to the basic features, the following optional features were implemented in the raytracer:

1. Transparent Surfaces: Support for transparent surfaces was added by implementing refraction based on Snell's law and Fresnel equations. When a ray intersects a transparent surface, it is split into a reflected ray and a refracted ray, with the Fresnel equations determining the amount of light reflected and refracted at the surface interface. This allows for rendering objects with varying degrees of transparency and creating realistic glass-like materials.

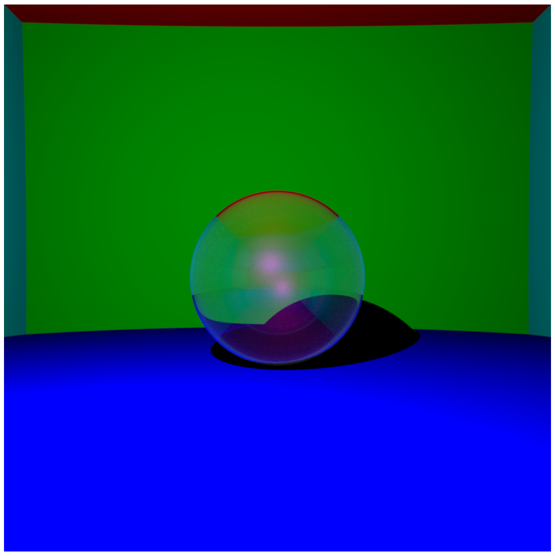


Figure 6: This image took 22388ms to render

8 Conclusion

This assignment involved the successful implementation of a raytracer with various features to render 3D scenes with realistic lighting and shading. Key features included diffuse and mirror surfaces, direct lighting and shadows, indirect lighting, antialiasing, and ray-mesh intersection with BVH acceleration.

Diffuse and mirror surfaces allowed for the simulation of different material properties and the creation of visually diverse scenes. Direct lighting and shadows added depth and realism to the rendered images, while indirect lighting, despite being computationally expensive, provided a more natural and globally illuminated look.

Antialiasing, achieved through supersampling, significantly improved the visual quality of the rendered images by reducing jagged edges and aliasing artifacts. The Möller–Trumbore algorithm enabled efficient ray-triangle intersection tests, and the BVH acceleration structure greatly enhanced the rendering performance for scenes with complex meshes.