CSE306 - Ray Tracer - Project 1

Amine Lamouchi

May 2023

1 Introduction

In this report, we present our raytracer that supports diffuse and mirror surfaces, direct and indirect lighting for point-light sources, shadows, antialiasing, and efficient ray-mesh intersection using a bounding volume hierarchy (BVH). The code is executed on a machine equipped with a `2.6 GHz 6-Core Intel Core i7 processor`. All images have a resolution of 512×512 pixels. Recursion depth (light bounces) was limited at 5 and we used 64 rays per pixel. We start by a high-level overview of the classes composing the raytracer.

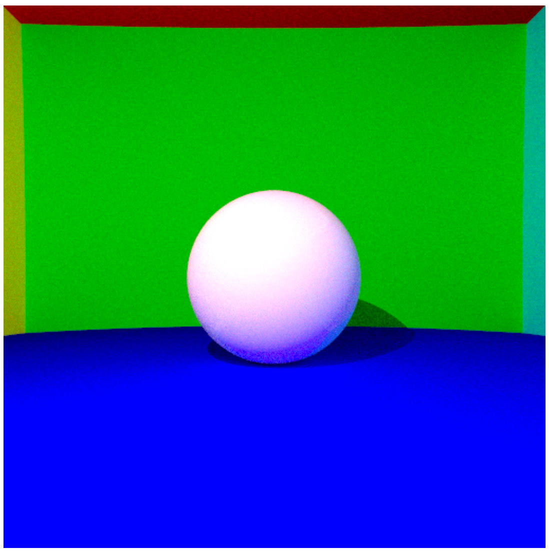
2 Main Classes

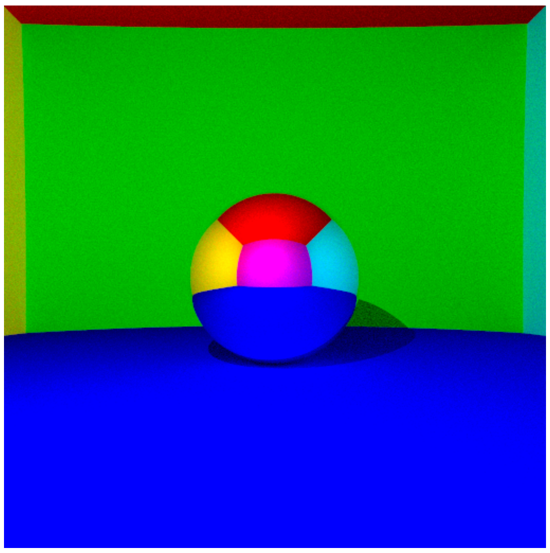
- **Object:** The base class for any object that can be placed in the scene. It contains a virtual function intersect to check if a ray intersects with the object. This class also includes properties for color (rho) and mirror reflection (mirror).
- **Sphere** and **TriangleMesh**: These are subclasses of the Object class and provide specific implementations for the intersect function. The Sphere class represents a sphere in 3D space, while the TriangleMesh class represents a mesh of triangles that can form any arbitrary shape. The TriangleMesh class also includes methods for building and intersecting a bounding volume hierarchy (BVH).
- **Scene:** This class includes a list of all objects in the scene and methods for adding objects to the scene, checking for ray-object intersections, checking for visibility between two points, and computing the color of a pixel by tracing a ray through the scene.
- **Intersection:** This class represents the intersection of a ray with an object. It includes the point of intersection (P), the normal at the intersection point (N), the distance from the ray origin to the intersection point (t), and the color at the intersection point (rho).
- **BoundingBox** and **BVHNode**: The former represents a 3D bounding box (that we keep track of using two corner vectors). It includes methods for checking if a ray intersects with the bounding box and for expanding the bounding box to include a new point. The latter represents a node in the BVH. It includes a bounding box and pointers to the left and right child nodes in the hierarchy as well as start and end indices for the bounds of relevant triangle indices. **I received the help of my classmate Lasha Koroshinadze for the integration of BVHs (both build and intersect).**

The **main** function of the program constructs a **Scene** object, adds several **Sphere** and **TriangleMesh** objects to the scene, and then traces rays through each pixel of the image to compute its color. We will now go over the main features supported in our Ray Tracer and showcase relevant images.

3 Diffuse and Mirror Surfaces

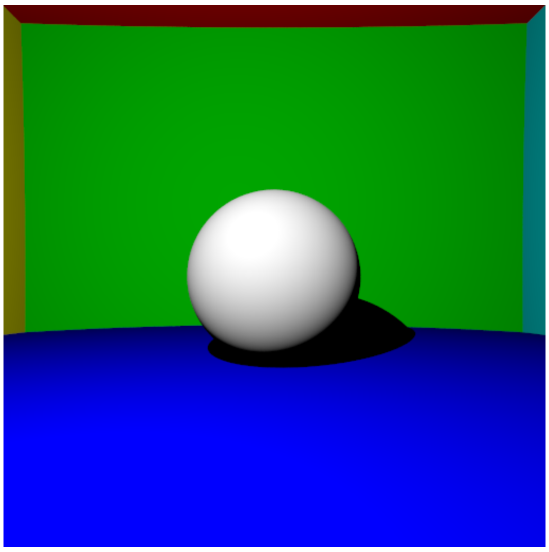
For mirrors, we computed the reflected direction and recursively traced the reflected path up to five bounces. For diffuse surfaces, we computed the color by considering the visibility term, surface reflectance, and the dot product between the surface normal and the light source vector as indicated in the lecture notes.

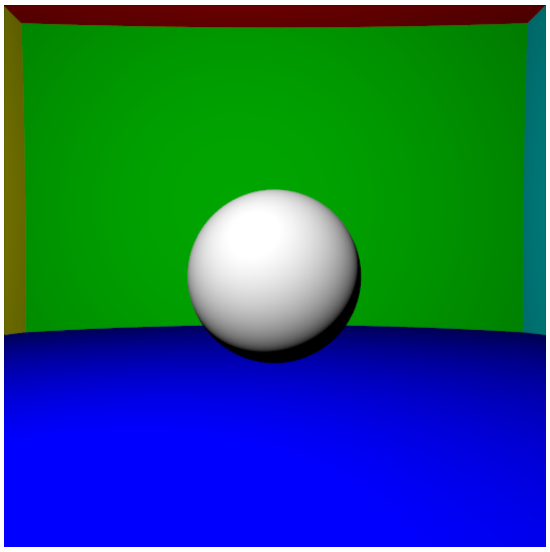

(a) Diffuse surfaces: 4.42 s


(b) Mirror surfaces: 4.32 s

4 Direct Lighting and Shadows

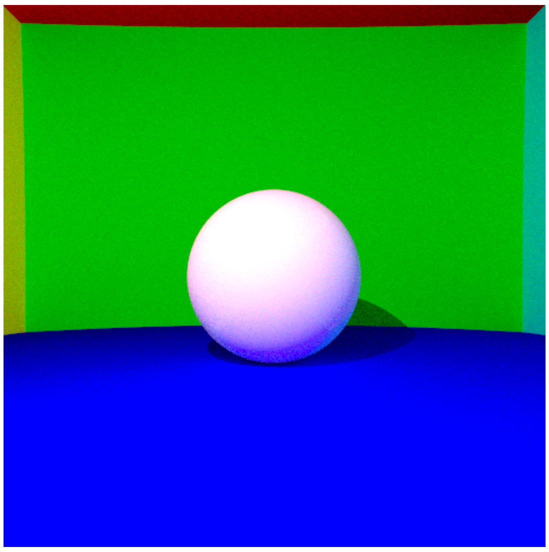
Shadows were implemented using the `visibility` helper function that casts a ray towards the light source starting from the point of intersection. By checking for intersections with scene objects, it determines if a point is in shadow (visibility value of 0) or not (visibility value of 1) which is taken into account in the `getColor` function.

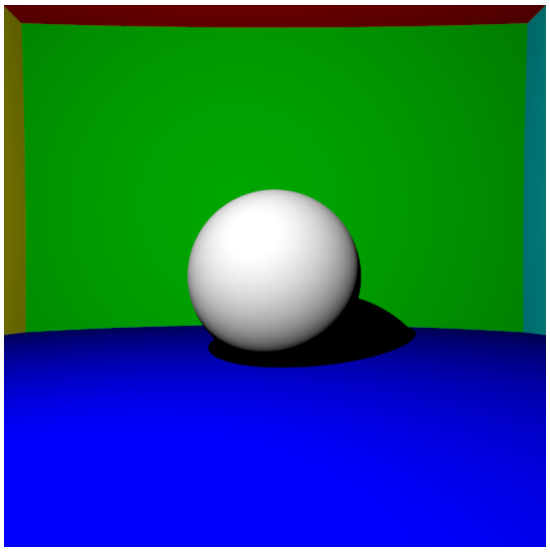

(a) With shadows: 0.51 s


(b) Without shadows: 0.32 s

5 Indirect Lighting

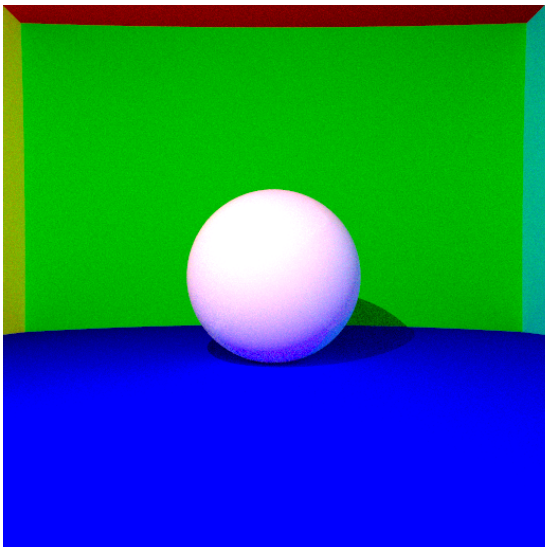
To implement indirect lighting we first generate a random direction vector in a hemisphere oriented around the surface normal at the intersection point then cast a ray from there. The color from this ray is computed recursively by calling the `getColor` function and its contribution is added to the direct illuminatioin (`color`).

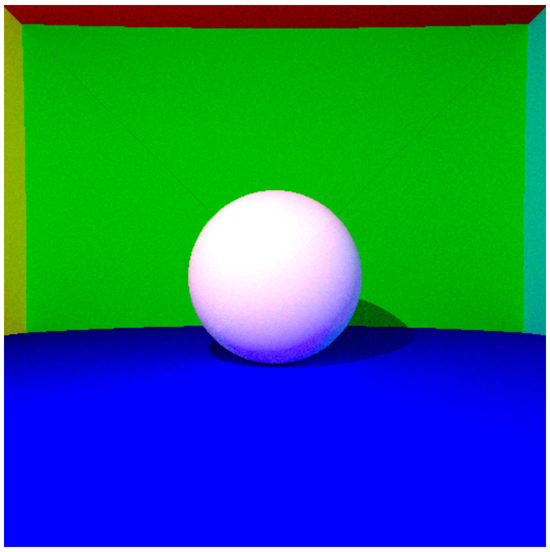

(a) With indirect lighting: 4.42 s


(b) Without indirect lighting: 0.51 s

6 Anti-aliasing

For every pixel in the scene multiple rays are cast and each ray is slightly offset within the pixel area by a random amount, ensuring diverse sampling within the pixel. We can observe that without anti-aliasing, the regions where the walls meet is very rough and pixelated moreover we see two diagonal lines across the screen. Both of these issues are solved after integrating anti-aliasing.


(a) With anti-aliasing: 4.42 s


(b) Without anti-aliasing: 4.21 s

7 Mesh support and BVH

After a mesh is loaded, we first construct a BVH. This is done by recursively dividing the mesh's bounding box into smaller boxes, each containing a predefined minimum number of triangles. The resulting structure is a binary tree with leaf nodes containing a small number of triangles. The construction of the BVH is implemented in the `build_bvh` function. When testing for ray intersections, instead of checking the ray against every triangle in the mesh, we traverse the BVH from the root, intersecting the ray only with the bounding boxes of the child nodes. If a ray intersects a bounding box, we continue to traverse the tree, eventually reaching the leaf nodes that contain the triangles.

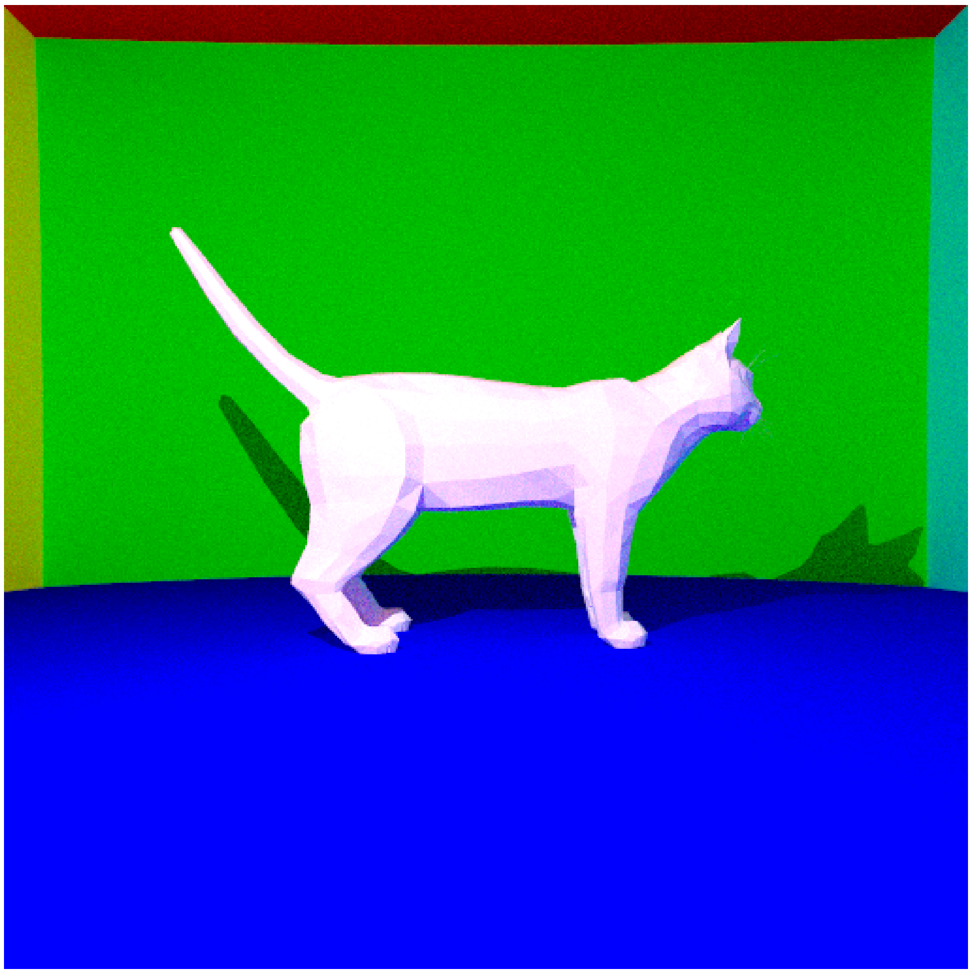


Figure 5: Resolution: 512×512 - Rendering time: 15.18 s - Rays per pixel: 64 - Rec. depth: 5