Assigment

Atodiresei Matei

May 14, 2024

1 Introduction

In this project we created a raytracer capable of simulating rays intersecting with spheres, triangle meshes, by bouncing rays and checking whether or not they colide with the objects. We also utilized anti-aliasing, bounding volumes and BVH.

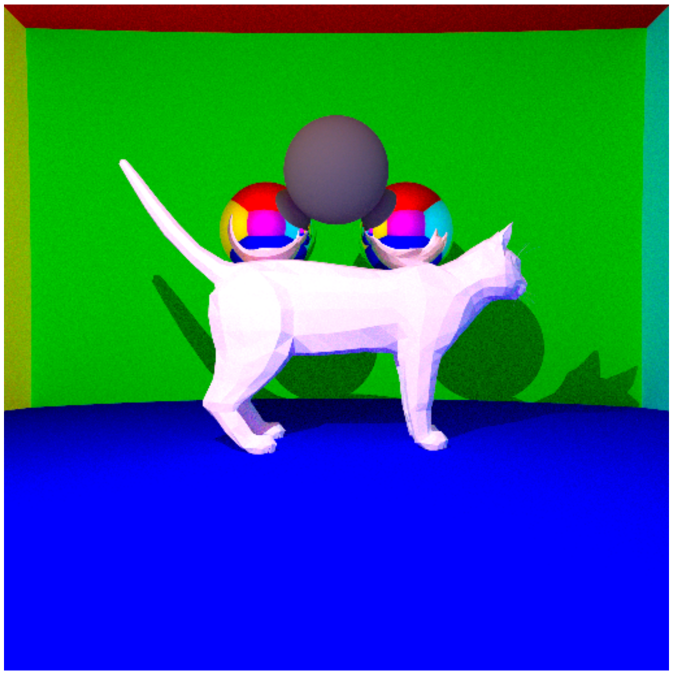


Figure 1: An example of a rendering

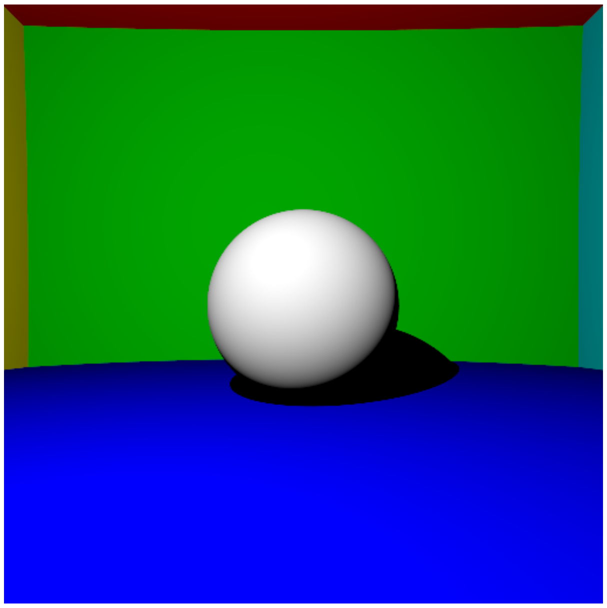
2 Code Structure

All the code is stored in `main.cpp`. We implemented some classes and some structures, as instructed in the lecture notes. I will try to briefly go through the most important features/functions.

The two structures are Intersection and TriangleIndices. They are only used to store multiple values at ones similiar to Vector.

Scene stores all the objects and a couple notable functions like getColor that computes the color of a point. This means that we also handle if an object is a mirror and where we implement the indirect lighting.

The objects, namely the mesh and the sphere have intersection functions.


(a) Simple Picture (with gamma correction)

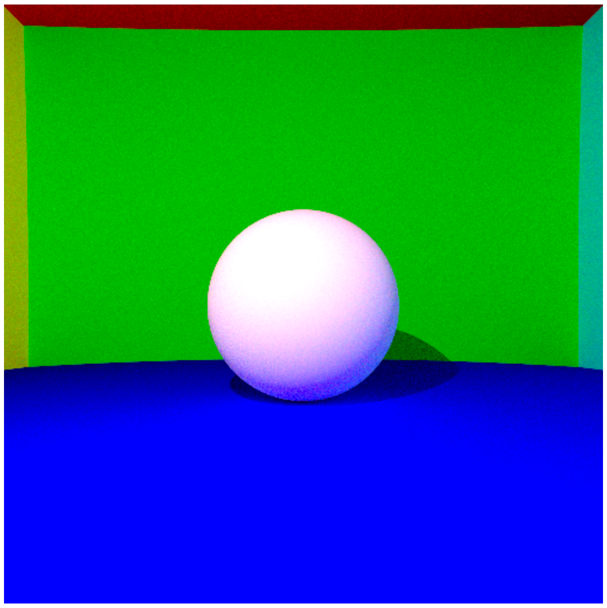

(b) No gamma correction

Figure 2: Caption for both images

3 Diffuse and mirror surfaces

In the first lab we implemented the most basic classes. In the Scene class we place our objects. Initially we just have some spheres that act as walls, and a sphere in the middle with Diffuse Surfaces.

We also implement Mirror surfaces that send the ray bouncing like a mirror. Because we do not want to run the code forever we will need to put a maximum depth on these number of bounces.

Direct lighting and shadows with point light sources

We can also see in this picture that the ball has a shadow. This comes from the compute_visibility() function that send a ray in the direction light came from. This is to see if the path is obstructed.

We can also apply a gamma correction here is to ensure that the intensity of each pixel is accurate on different kind of monitors and devices.

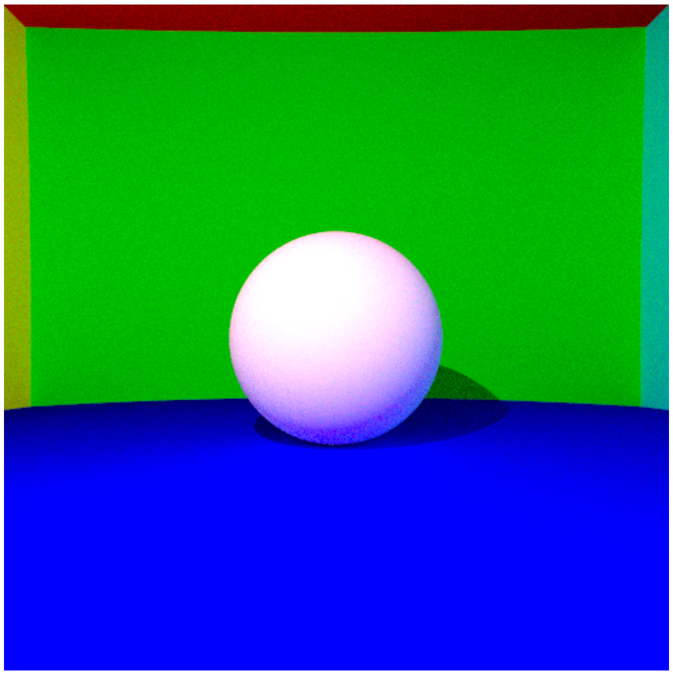
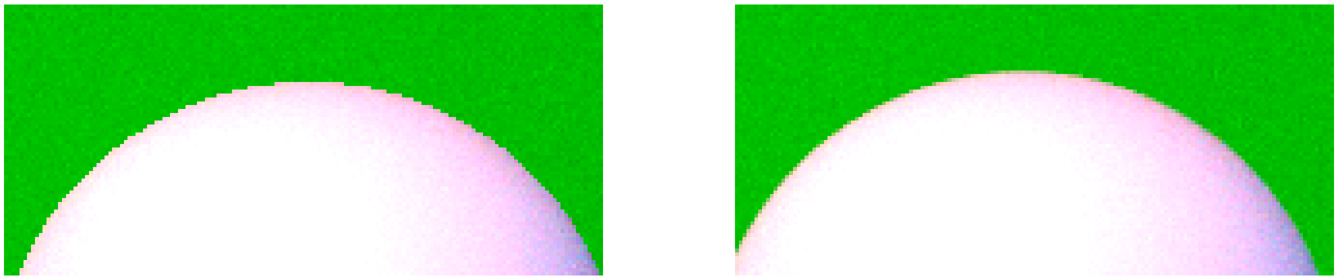


Figure 3: Your caption here

4 Indirect Lighting and Antialiasing

By implementing indirect lighting we stop seeing very dark, unrealistic shadows. This comes from light bouncing from the surrounding objects and onto the would-be shadow place.



(a) Close-up of a sphere

(b) Close-up of a sphere with Antialiasing ON

Figure 4: Caption for both images

As we can see in Figure 4 by sending multiple rays that have a slightly randomized direction we can better capture the edges of the sphere.

5 Triangle Meshes

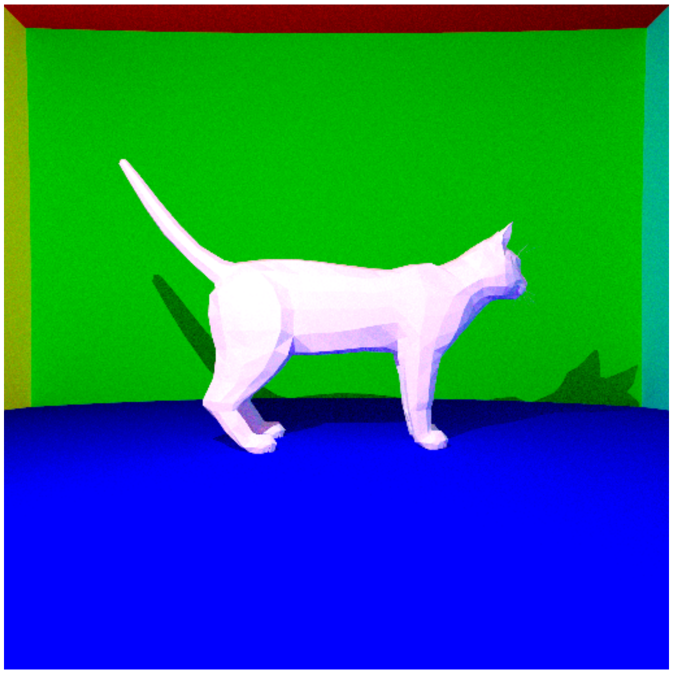


Figure 5: Triangle mesh render of a cat

By placing triangles in space at specific positions we can create an object what can be rendered. In each triangle there are stored the vertices of the triangle, the normals associated with each each vertices, as well as the UV coordinates. These coordinates provide information regarding the texture of the object.

Since we moved from spheres to meshes a new intersect function had to be made. In order to perform the intersection with a triangle we first check if the plane of the triangle is parallel with the ray. Then if it not, we utilize Barycentric coordinates in order to tell if the ray intersects the triangle or not.

6 Bounding Volumes

Since an object is made of a lot of triangles it takes significantly longer to simulate a triangle mesh than a simple sphere. The problem is that when we send a ray we have to check if it intersects all of the objects in the scene. This is why more objects means more computational time.

In order to reduce the resources wasted on computing intersections between objects and rays that are nowhere close, we surround the whole triangle mesh in a Bounding Volume. Now if the ray does not collide with the Volume it will also not intersect the mesh.

This idea can be applied further by utilizing an BVH (Boundary Volume Hierarchy). The Bounding Volume is split into 2,3 sections then those sections are split and so on. Like this we can test intersections between rays and volumes in order to reduce the number of triangles that are needed to be tested for intersections.

This does reduce the computation time significantly: from approximately 5 min to 8 seconds, which is big difference!

7 Parallelization

We also increase our speed by implementing parallelization. By allowing the compiler to compute each iteration of the loop on a different thread. This allows multiple computations to be completed at the same time this allows multiple fold improvement: from 99 seconds to render the mesh cat to only 8 seconds.