Raytracer project report (CSE306 - Computer Graphics)

Pablo Bertaud-Velten

1 Summary

This project was conducted in the CSE306 Computer Graphics course. It implements some of the main building blocks of a raytracer. This project includes the following features:

- Basic geometry (in (x,y,z) coordinates), vectors, etc
- Direct lighting
- Shadows
- Antialiasing
- Indirect lighting for point light sources
- Diffuse and mirror surfaces (a mirror ball, a transparent ball, and a hollow ball)
- Ray-mesh intersections

2 Process and code

All of the code for the raytracer is in the file *raytracer.cpp*.
The code begins by a Vector class, which implements all of the vector definitions and operations we will be needing later on (scalar, vector products, as well as additions, subtractions, etc).
We then define an Intersection structure, which will store all of the information about intersections between rays of light and objects of the scene. For example, some of the attributes of an intersection are whether the object is reflective, its color (in RGB), the normal vector to the object on that intersection point, among other attributes and methods.

We define the Ray class, which represents the rays of light that are "generated" by the camera, and having a direction and an origin.
The Geometry class is used as a basis to all objects on our scene.
The TriangleMesh class defines a mesh of triangles as a list of vertices and indices. This mesh is used to create complex surfaces (such as a cat in our case, see below). The BoundingBox class is here used to speed up the computations of intersections between the rays of light and all of the triangles of the mesh.

To view our objects, we define a Scene class, in which we will add objects (inheriting from the Geometry class). Then, we will generate our image by iterating over all the rays, finding the N-closest intersections of that ray (N being the maximal number of bounces of our ray around the scene, usually 5 in our case). When the ray intersects a surface, the color of the object at the intersection is computed using the various particularities of the object, such as its refractive index, its albedo (RGB color), or its reflectiveness.

We also parallelize the execution of the code, since large scenes would take a while to be rendered.

Our output is an image, for which each pixel is the color computed above by launching the rays. This image is written using the std_image_write library, allowing to view our output as a .png image.

3 Features

Here is a picture of our output, depicting (from left to right) a hollow sphere, a refractive sphere, and a reflective sphere, using 64 rays per pixel, and a maximal depth (max. number of bounces) of 5, with gamma correction of 2.2 and an antialiasing with a standard deviation of 0.5. The rendering of this image took about 10 seconds. Note that the background, sides and ceiling are done using really large spheres for simplicity.

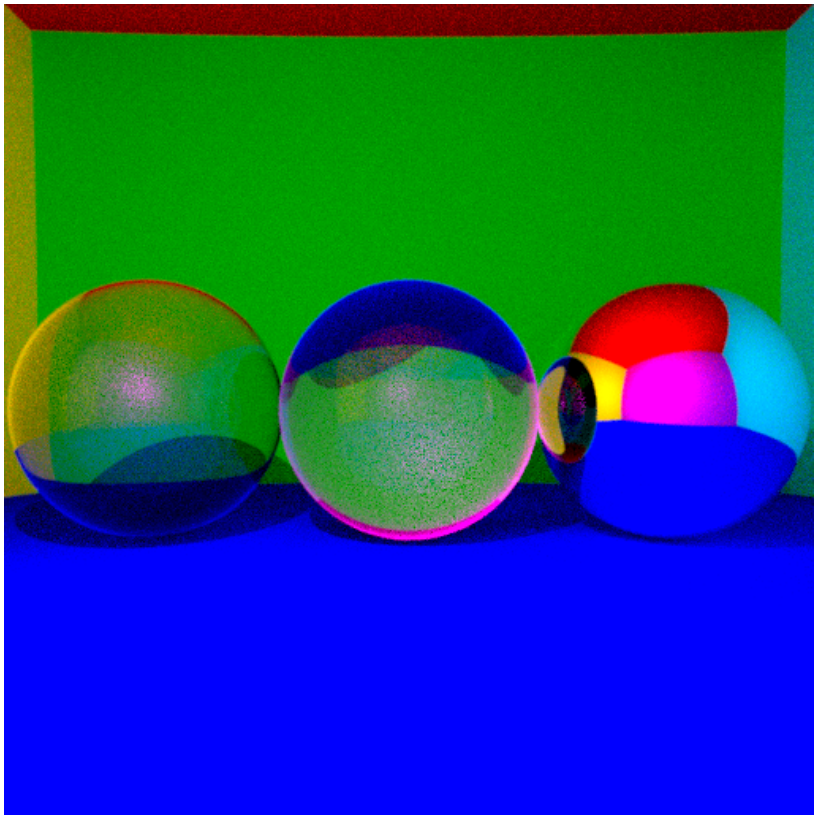

Figure 1: The three spheres

4 Antialiasing (see next page)

Here we can see the slight difference between using aliasing or not: the boundaries of the spheres are a bit more jagged without antialiasing. Antialiasing allows to smoothen the edges out, as can be seen on the two pictures below.

(a) Without antialiasing

(b) With antialiasing

Figure 2

5 Using triangle meshes

In this section, we now use a triangle mesh to render an image of a cat (using a list of vertices and indices), from the .obj file reader provided by Nicolas Bonneel. We are also using BVH to reduce the computation time for looking for intersections of the rays with the mesh. We had 64 rays per pixel. This rendering took 52 seconds.

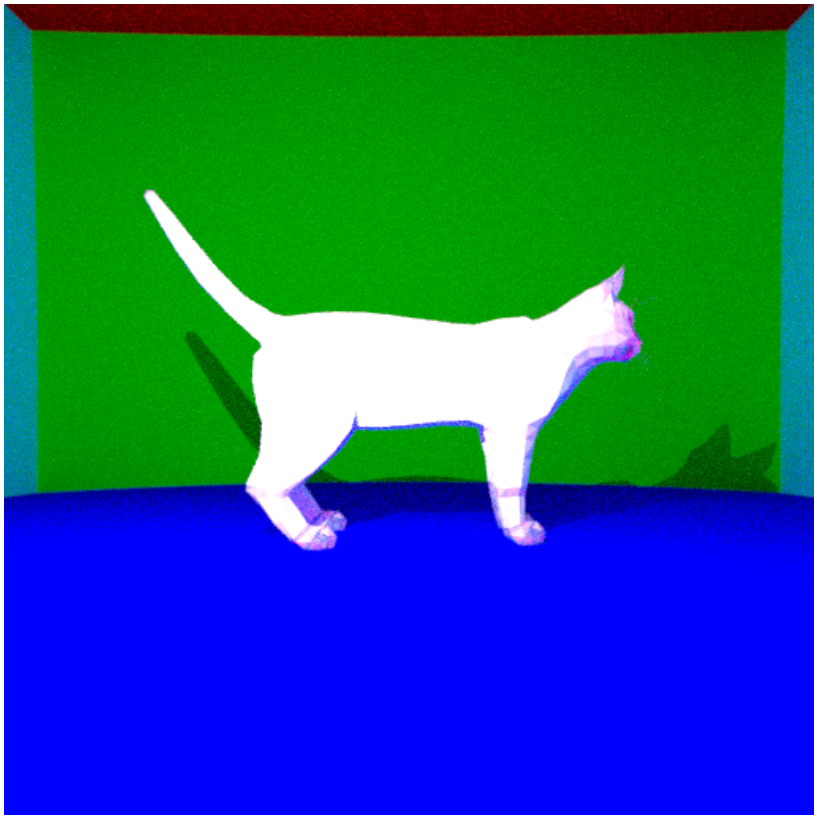

Figure 3: The cat rendering using BVH