Rendering 3D Graphics with a Raytracer

Louis de Benoist

ABSTRACT

Implementation of a C++ ray tracer for 3D computer graphics rendering. Completed as part of the CSE-306 course at École Polytechnique, this piece of software is capable of rendering diffuse and reflective spheres, as well as loading and displaying reflective 3D meshes (.obj files).

Lab 1

This first lab consisted in making a ray-tracer capable of rendering both **diffuse and mirror** spheres. Initially, the ray-tracer only worked with direct lighting and shadows, relying on the archaic Lambertian model. At the end of the first lab, I was able to obtain renderings like Figure 1.

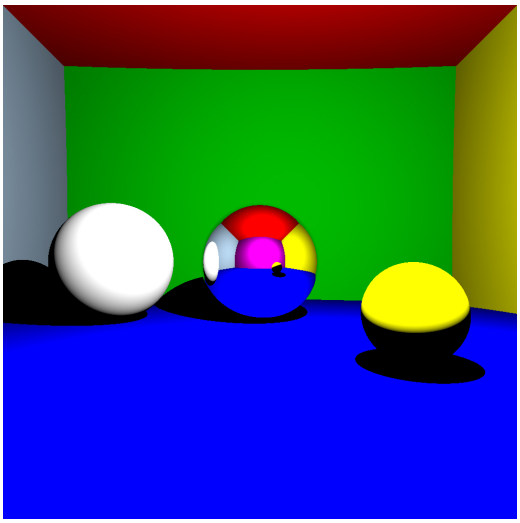

Fig. 1: Rendering at the end of the first lab. 1000×1000 pixels. Rendering time: 0.037 seconds

Figure 1 consists of three spheres, including one that is a mirror. As we can see, the shadows are quite unrealistic, a problem that was fixed in the second lab.

Lab 2

The second lab enabled the spheres, as well as the overall scene, to appear more realistic. This lab saw the introduction of two important new concepts: **indirect lighting and antialiasing**. To obtain indirect lighting, the Lambertian model was replaced with a more accurate rendering equation, thanks to Monte Carlo integration with importance sampling. This results in more realistic shadows, as can be seen in Fig. 2.

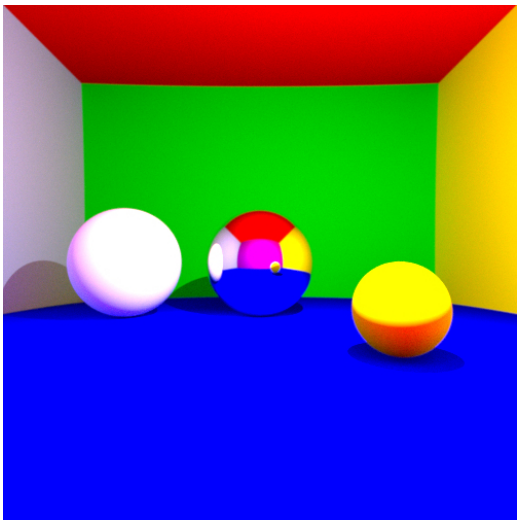

Fig. 2: Indirect lighting with parallelization and antialiasing (500×500 pixels with 1000 rays, max depth of 5, and $\sigma = 0.5$. Rendering time: 127 seconds)

One of the problems with shooting rays through the center of the pixel is that there are sharp, ugly transitions between different objects. To smoothen things out, notably the edges of spheres, antialiasing averages the color obtained from different rays passing through points offset from the center by some Gaussian noise.

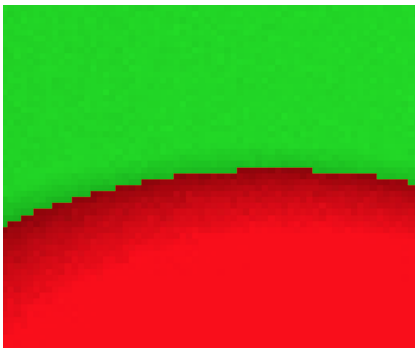

Fig. 3: Figure 2 zoomed before adding the antialiasing. We can see the sharp, ugly edges.

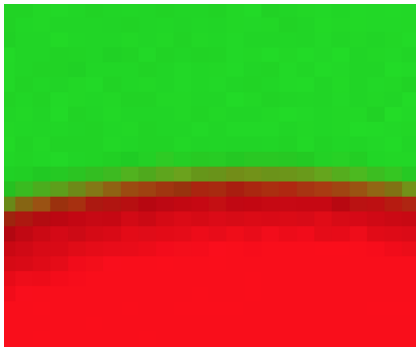

Fig. 4: Figure 2 zoomed in. We see that antialiasing with $\sigma = 0.5$ leads to smoother transitions.

Lab 3

The third lab saw one of the most important upgrades: the ability to load **triangle meshes** (.obj files) and render them. The program can load, resize and move around objects defined by triangle meshes and render them as diffuse or mirror surfaces. Furthermore, **Phong interpolation**, which averages out the normals of the vertices, was added to make a smoother rendering. Later on, textures will be added to obtain actual renderings that are true to the artist's original vision.

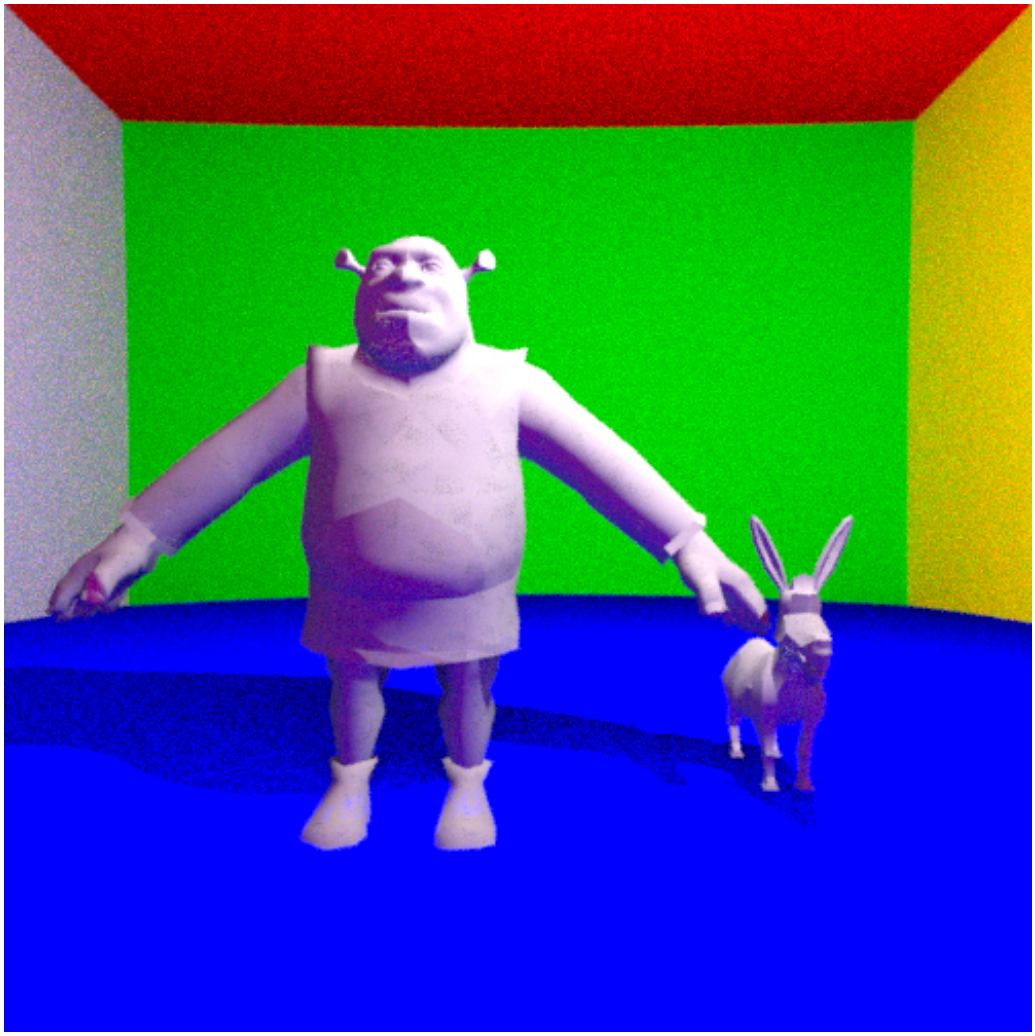

Fig. 5: Shrek (1274 vertices) and donkey (612 vertices) next to each other. 500×500 pixels. 32 paths, max depth of 5, and $\sigma = 0.5$. Rendering time: 243 seconds

To figure out ray-triangle intersection: two methods were implemented. The first was the **naive method**, which simply looped through all of the triangles. The second, which drastically enhanced performance (by a factor of 7 in practice) consisted in using **bounding boxes** to enclose the object.

Ray-bounding box intersection being much faster to test, this enabled me to quickly check whether a given ray would intersect the mesh, preventing me from having to go through each triangle.

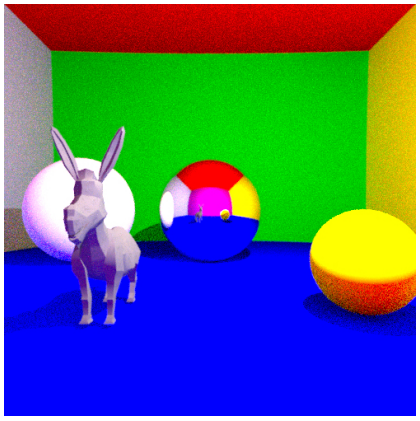

Fig. 6: Donkey (612 vertices) next to some spheres. 500×500 pixels. 32 paths, max depth of 5, and $\sigma = 0.5$. Rendering time: 62 seconds

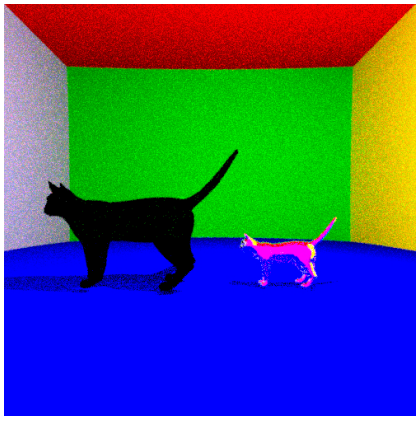

Fig. 7: Two cats (2247 vertices each) of different sizes, one black and the other a mirror. 500×500 pixels. 10 paths, max depth of 5, and $\sigma = 0.5$. Rendering time: 98 seconds

Lab 4

I ended up having more bugs than I thought while implementing Lab 3, leading me to not have enough time to implement BVH, which recursively splits the mesh into bounding boxes, generating a search tree.