Report LAB1 Graphics CSE306

romain.puech

May 2024

1 Introduction

The aim of this project was to code from scratch a ray tracer in C++. The ray tracer handles multiple functionalities such as diffuse and mirror spheres, indirect lighting, antialiasing, mesh intersections, and textures, that we describe in the following sections.

Throughout the report, the rendering times are reported for a 512X512 image, generated with a maximum ray depth of 5. The code is compiled with an *Ofast* flag and run in parallel on a laptop equipped with an Intel Core i7 8th gen.

1.1 Lab 1: Diffuse surfaces and Mirror surfaces

1.1.1 Diffuse spheres

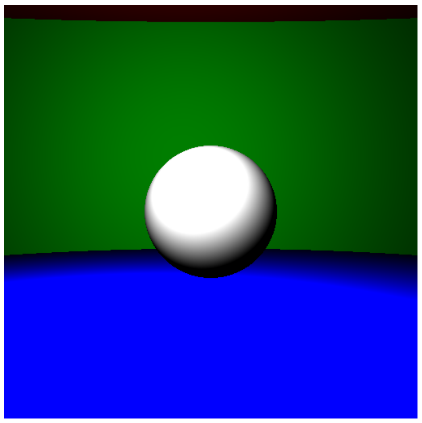
The first lab focuses on rendering both diffuse and mirror spheres. We generate a ray for each pixel of the image, and compute its intersection with the sphere. The resulting color is computed using the albedo of the closest intersected sphere, multiplied by attenuation terms accounting for the distance to the (unique) light source, and the intersection angle. The exact formulas are detailed in the lecture notes, on page 17.

To account for shadows, we multiply the final pixel color by a visibility term. If there is an object between the light source and the point of intersection found, then the visibility is set to 0. Figure 1 displays an image of a diffuse sphere obtained using the ray tracer.

Since our color perception does not scale linearly with the color intensity displayed on a computer screen, we use a power-scale encoding of the color. Typically, one raises the RGB values by a power of $1/\gamma$. This is called gamma correction. We use a value of 2.2 for γ throughout this report.

1.1.2 Mirror surface

Mirror surfaces are handled using recursion. A mirror surface doesn't have an albedo, but rather creates a new ray stemming from the surface, and returns the output of the ray tracing function of this new ray. Figure 2 displays a diffuse


(a) No shadow

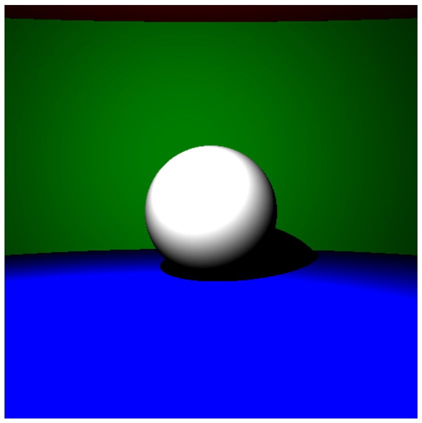

(b) With shadow

Figure 1: Diffuse white sphere. Rendered in about 12 ms.

and a mirror sphere. We also notice the difference in contrast, induced by the use of gamma correction.

1.2 Lab 2: Advanced lighting and camera models

1.2.1 Indirect lighting

Indirect lighting accounts for the light reflected from surrounding diffuse surfaces. To include this contribution, we use Monte-Carlo integration to evaluate the rendering equation. For this, we generate a few rays with random directions stemming from the point of intersection between the camera ray and the sphere and average out their contribution. We cap the number of recursive calls to 5. The exact process is described in the lecture notes on pages 25–31. Figure 3 displays diffuse white spheres. We can easily notice the difference in noise between the two images. Indeed, using more samples per pixel makes our Monte-Carlo approximation more accurate.

1.3 Antialiasing

To correct aliasing (color discontinuity between adjacent pixels), we replace our initial camera rays with perturbed sample rays that are often in the middle of each pixel (described on pages 32–33). The resulting image displayed in figure 4 appears smoothed. We can for example see that the sphere borders now appear blurry.

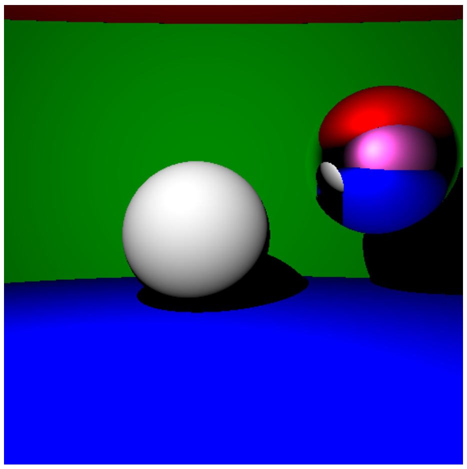
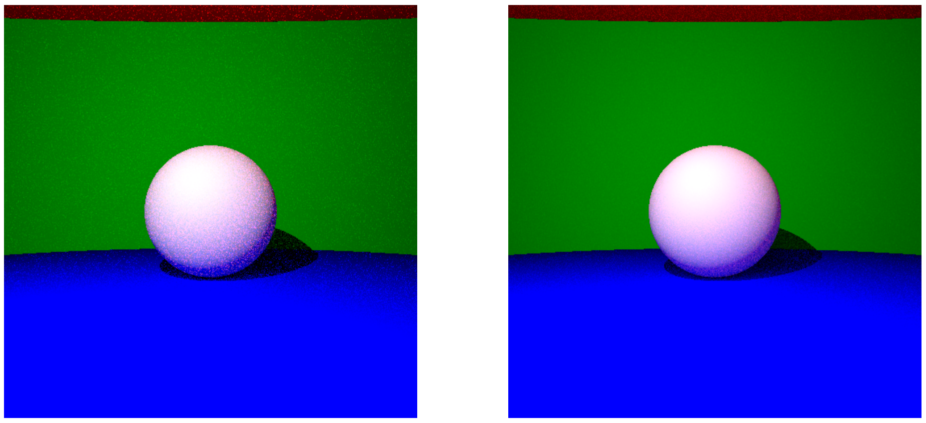


Figure 2: Diffuse and mirror spheres. Rendered in about 15ms.



(a) 1 spp, rendered in 218 ms

(b) 32 spp. Rendered in 8 s

Figure 3: White diffuse sphere with indirect lighting for two different numbers of sample rays per pixel (spp).

1.4 Lab 3: Ray-mesh intersection

1.5 triangle intersections and bounding boxes

We integrated support for triangle meshes, which are collections of triangles composing an object. Similar to our previous approach for calculating ray-scene intersections, determining intersections with a mesh involves traversing all its triangles and returning the closest intersection to the camera. We test our code with a cat mesh, displayed in Figure 5. As the cat mesh is made of millions of triangles, the execution becomes slow. To avoid unnecessary computations,

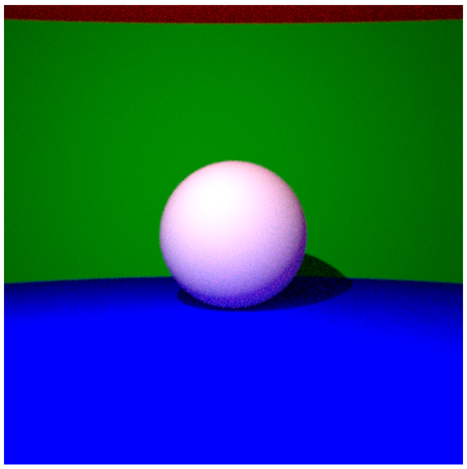


Figure 4: Sphere rendered in 8 with antialiasing and 32 spp.

we employ bounding boxes. We check triangle intersections only when a ray intersects the bounding box surrounding the mesh.

1.5.1 Interpolation of normals and texture

We can control the perceived smoothness of a shape using artist-defined normals that manipulate the shadings. The result is displayed in Figure 6a. We also map triangle vertices to an image texture provided by the artist. A final high-quality (128 spp and less antialiasing) render of the cat mesh with interpolated normals and textures is displayed in Figure 6b.

1.6 Lab 4: Bounding Volume Hierarchies algorithm

We further optimized the computation of the triangle mesh intersections using an algorithm called Bounding Volume Hierarchies. This method involves iteratively applying the bounding box technique. We initially split the entire mesh into two sets of triangles based on their centroid. By repeating this process recursively on each part, we construct a BVH binary tree. We can then use this tree structure to only test intersections with triangles in the most relevant sub bounding box. Using BVH produces a speedup of an order of 10x over the simple bounding box technique.

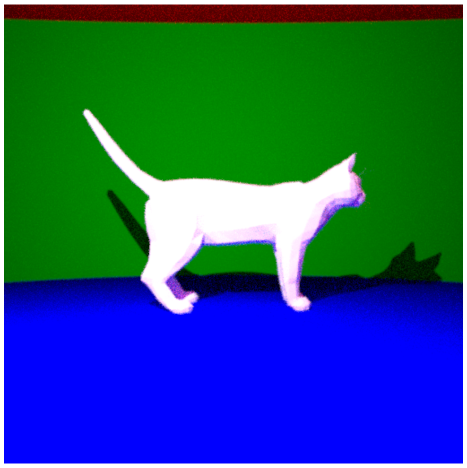
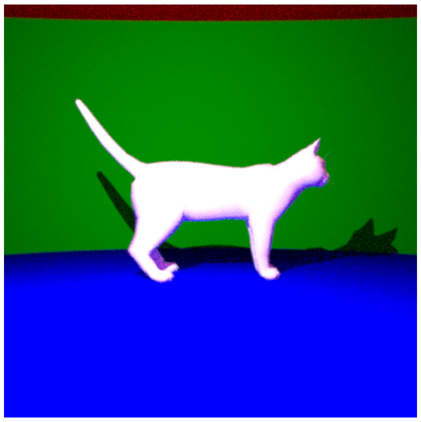
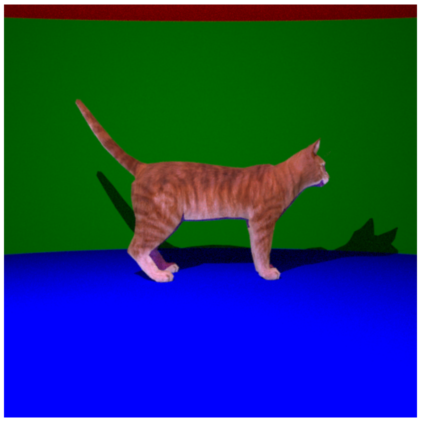


Figure 5: Cat triangle mesh. Rendered in 7 mins with 32 spp and using a bounding box, and about 20 seconds using BVH.



(a) Interpolated normals, 32 spp, rendered in about 20 s with BVH



(b) Interpolated normals and texture, 128 spp, rendered in about 2 min

Figure 6: Cat mesh with extra functionalities.