CSE306
Computer Graphics

**Assignment 1 Report**
**Name:** Marie Huynh

Your first grade will be the result of your raytracer project developed during the first 4 labs. You are expected to have a raytracer with at least:
- Diffuse and mirror surfaces (Lab 1 / pages 17–25 of the lecture notes)
- Direct lighting and shadows, for point light sources (Lab 1 / pages 13-19)
- Indirect lighting for point light sources (Lab 2 / pages 25–31) without Russian roulette
- Antialiasing (Lab 2 / pages 33-34)
- Ray mesh intersection (Lab 3 / pages 42-44) including BVH (Lab 4 / pages 45–47)

Other features are optional though appreciated (notably transparent surfaces, textures, or even area light sources).

Your report is expected to be approximately 4-5 pages long including images. All supported features should be illustrated with images. Please indicate the time it took to render images (and number of mesh triangles if you used other meshes).

Plagiarism: plagiarism will be checked and will not be tolerated. If you were stuck and had to get "inspired" by another code, copied or even adapted code from somewhere, be extremely clear in your report and mention the original sources so that it will not be considered as plagiarism.

# 1 Introduction

The structure and techniques used to implement the raytracer and its following features follow closely the lecture notes and the first steps seen in class during the first lab.

Here is the link to my github repository : https://github.com/marie-huynh/Assignment-1

**The image size for all images is** $512 \times 512$**. The field of view is 60. The intensity is 1e10. Except when told otherwise, we will use 10 paths with a max path length of 5.**

We run the file at each step with the following command : `g++ raytracer.cpp -O3`

# 2 Lab I: Diffuse and Mirror Surfaces

Lambertian surfaces scatter light in all directions whereas purely reflective surfaces only reflect light in a single direction.

We set the `max path length` to 5.

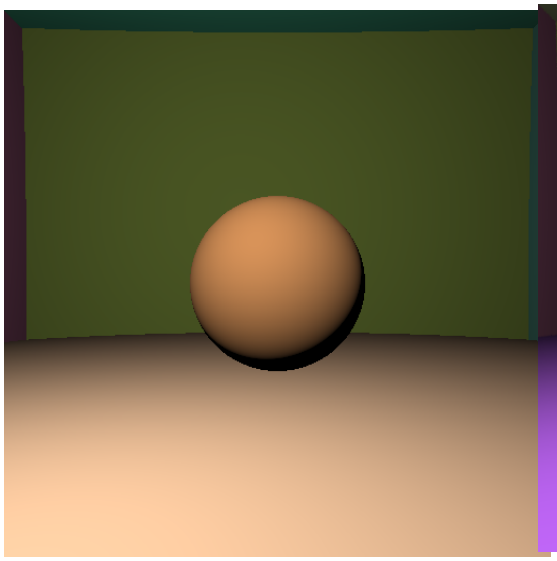

Figure 1: This sphere represents a diffuse surface. The image is computed in 0m4.332s.

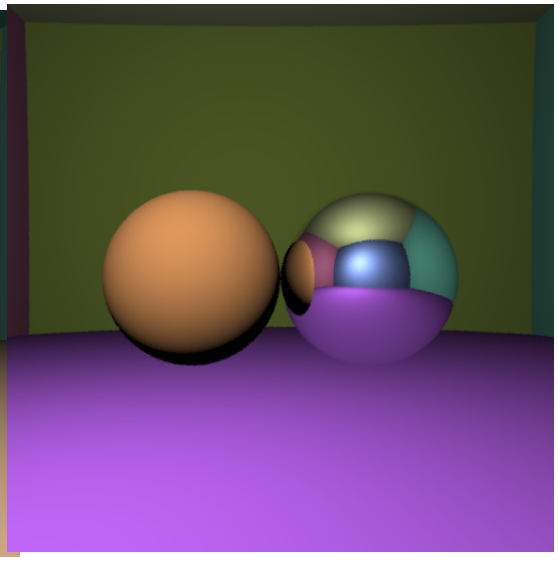

Figure 2: The sphere on the right is a sphere with reflection (mirror surface). The image is computed in 0m5.188s.

## 3 Lab I : Direct lighting and shadows, for point light sources

We render and shade some spheres with direct lighting which only considers light coming from the original source. We display the shadows.

Furthermore, we also add gamma correction so that the scene appears more natural and less contrasted.

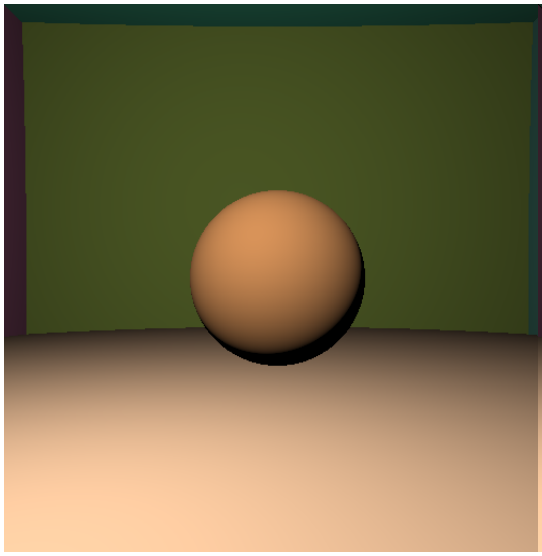

Figure 3: Without shadows. Rendering time : 0m0.621s

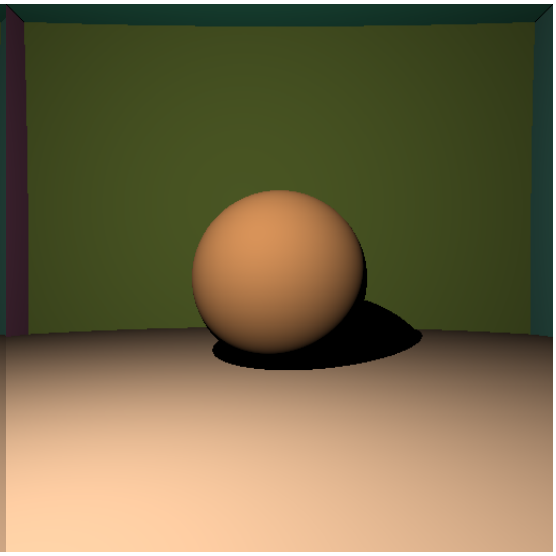

Figure 4: With shadows. Rendering time : 0m1.412s

# 4 Lab II : Indirect lighting for point light sources without Russian Roulette

Indirect lighting is important for realism. Using indirect lighting allows us to not only restrict ourselves to the ray lights coming directly from a light source, and take into account ray lights coming from the same source which are reflected by other objects in the scene. We implement indirect lighting using Monte Carlo Integration.

In the following figures, we see well the smoothing out of the border of the sphere for example.

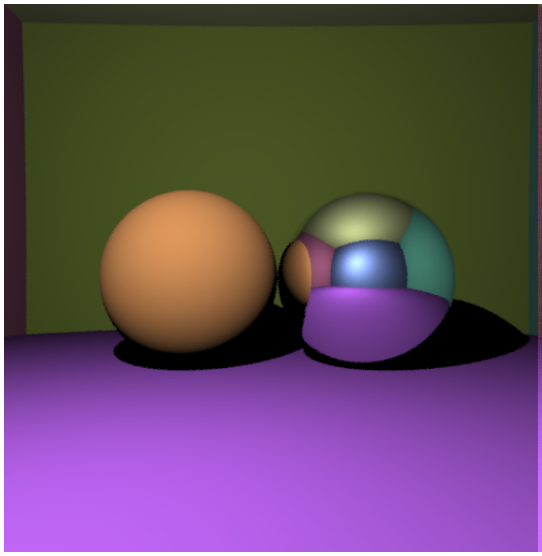

Figure 5: Before Indirect Lighting - Rendering Time : 0m0.929s.

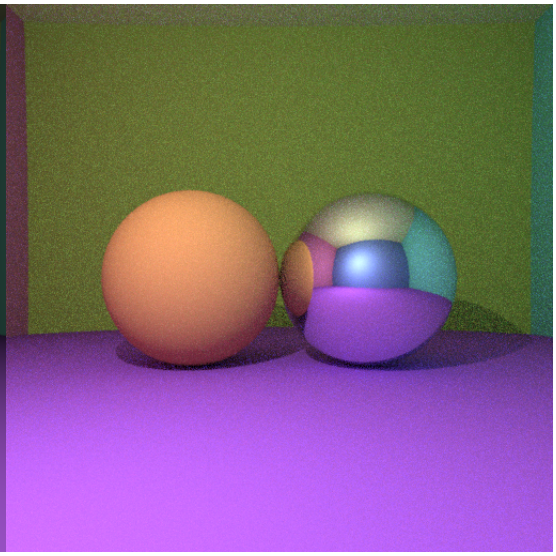

Figure 6: After Indirect Lighting - Rendering Time : 0m3.392s.

## 5 Lab II : Antialiasing

Currently, we sample rays in the middle of each pixels. A ray can intersect the sphere for a pixel and miss it in the next one. This creates a discontinuity between adjacent pixels: we call that phenomenon aliasing.

Anti-aliasing is a technique that allows to "smooth out" those lines and avoid that "stair-like" aspect. We will notably be using a Gaussian filter centered in the middle of pixel i, j to implement it.

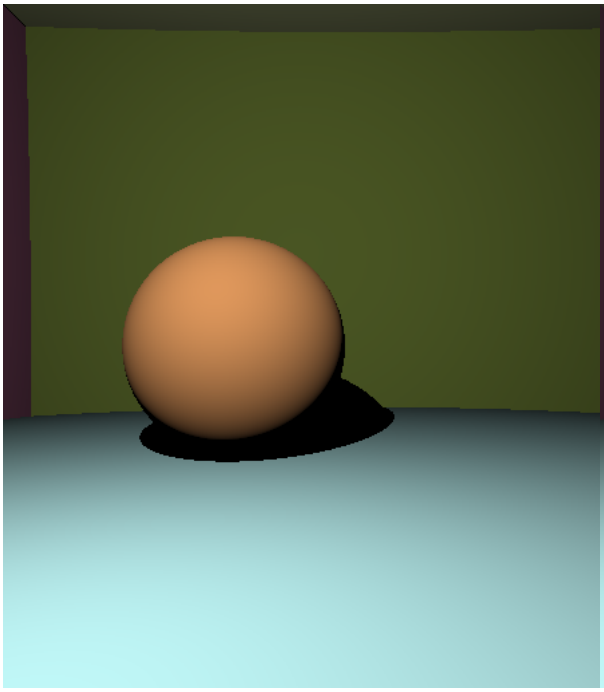

Figure 7: Without antialiasing, Number of Paths = 15, max length path = 5, stdev = 0.5, **Rendering Time : 0m1.084s**

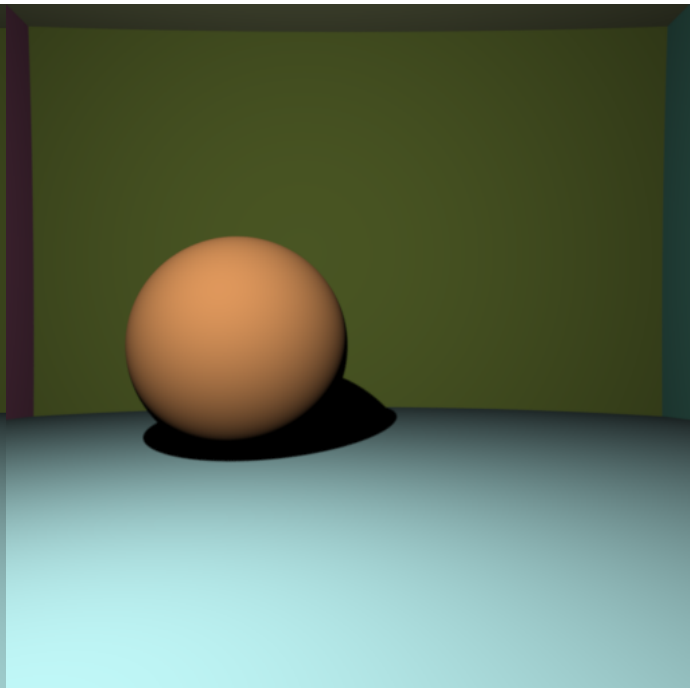

Figure 8: With antialiasing, Number of Paths = 15, max length path = 5, stdev = 0.5, **Rendering Time : 0m1.084s**

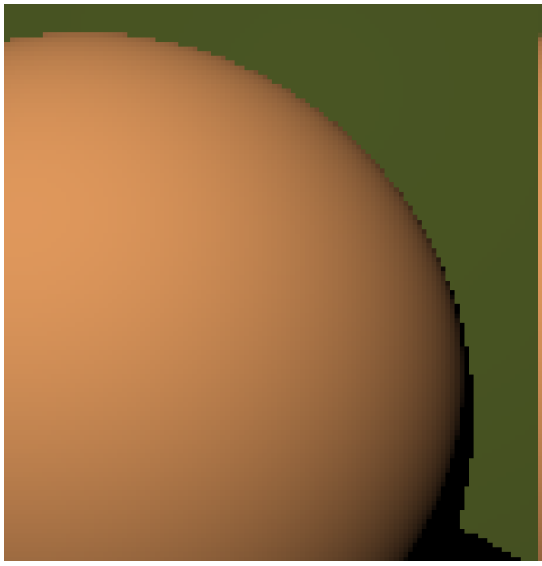

Figure 9: Without antialiasing - zoomed

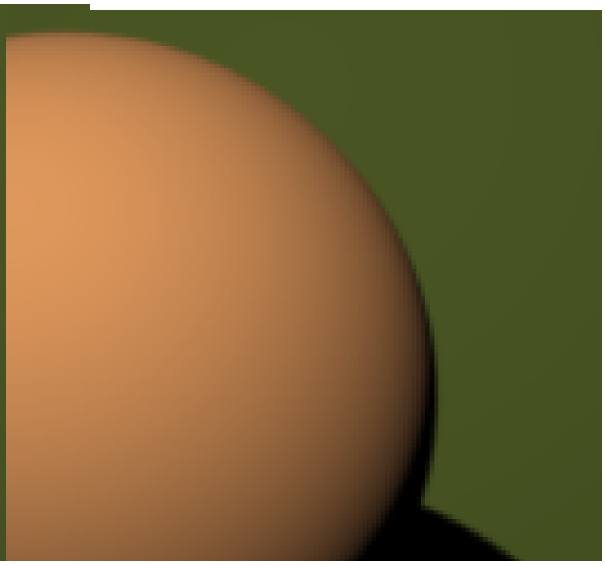

Figure 10: With antialiasing - zoomed

# 6 Lab III and IV: Ray mesh intersection, including BHV

"Triangle meshes are the most popular standard model used to represent polygonal surfaces. Drawing these meshes as a set of independent triangles involves sending a vast amount of information to the graphics system." (source : Belmonte, O., et al. "Efficiently using connectivity information between triangles in a mesh for real-time rendering." Future Generation Computer Systems 20.8 (2004): 1263-1273.)

A mesh is a set of triangles. We traverse all triangles and return the intersection closest to the camera using the Möller–Trumbore intersection algorithm. We inherit the class `Sphere` and the class `TriangleMesh` from a more general Geometry abstract class. The scene is now composed of an array of pointers to Geometry objects.

Checking all the triangles of the mesh makes the rendering time pretty slow as we can see on figure . We can optimize this linear time by testing if the ray intersects the axis aligned bounding box of our cat and then check all triangles if the ray intersects the bounding box.

We can implement the latter recursively with Bounding Volume Hierarchies (BVH). If the ray intersects the bounding box of the mesh, we can check if it intersects the two bounding boxes containing each half of the mesh, then using a quarter etc ...

Following the lectures notes, I implemented BVH building the class `BoundingBox`, the class `Node` and adding functions like `compute bbox`, `build tree`, `intersect`.

As we can see in the following figures, the rendering time is almost divided by 9 when we add BVH to our implementation.

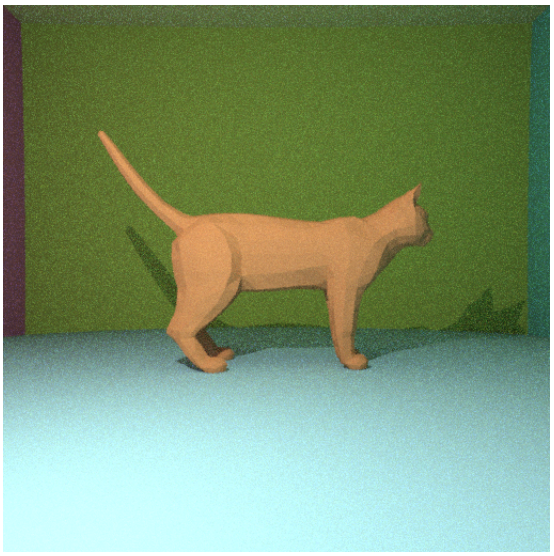

Figure 11: Without BVH, Rendering Time : 27m10.193s.

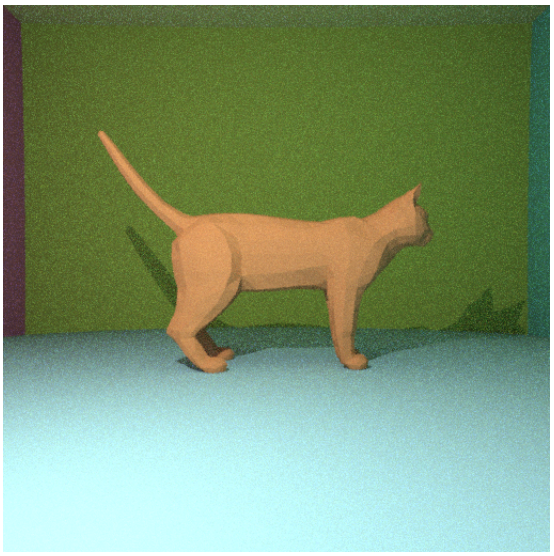

Figure 12: Using BVH, Rendering Time : 3m18.662s.

Figure 13: Ray-Mesh intersection without and with BHV