CSE306 - Ray Tracer Project

1 Introduction

1.1 Project Overview

This project implements a handcrafted C++ ray tracer capable of generating geometrical objects and rendering object files along with their texture. My project implements all of the mandatory sections of the assignment (Diffuse and mirror surfaces, direct lighting and shadows, indirect lighting without Russian roulette, anti-aliasing, ray-mesh intersection including BVH), as well as most of the optional sections (transparent surfaces, depth of field, interpolation, texture).

All of the project's code can be found in the present GitHub repository. Files have been split in headers and sources for organisational purposes, and I added a Makefile to simplify compilation. This report explains how I conducted the project, without covering formulas referenced in the textbook. The formulas used in the code are based on the textbook and live session explanations.

To ensure clarity and correctness, the code is also optimised as much as possible (without adding too many level of abstraction which would be counter-productive). To avoid and identify bugs easily, `const` statements were added wherever possible. To achieve the highest precision, floating-point operations were prioritised. To optimise memory, the most efficient types for each variable were used. Nonetheless, some typical concerns such as privacy of attributes have been disregarded as they would have overcomplicated the code with a multitude of getters and setter functions.

1.2 Execution Parameters

This ray tracer runs on a MacBook Pro M1 with C++11. Execution parameters can be set in `main.cpp`. For the empty environment, we recreate the textbook's setup. We make a 512×512 image with the same camera and light placement. We choose the same field of view of $60°$ and light intensity of 10^7. We also copy the textbook's spherical walls to get comparable visual results. In terms of quality, all images are rendered with 128 rays per pixel and a maximum of 5 bounces.

Figure 1: Cat and sphere image with all implemented features - *65.2s rendering time*

2 Diffuse and mirror surfaces

The first step is to define the building blocks of our project. After writing a `Ray` class which models each traced rays, we define a `Sphere` class which represents base surfaces. Using this, we can then write an `intersect` function in charge of determining the closest element the ray intersects and a `getColor` function responsible for determining the colour of that ray.

This way, we can create an empty environment for the picture made of spherical walls and spheres contained in that environment. We can also create options for the surface of our spheres. We notably define mirror surfaces, which essentially works by calling the `getColor` recursively on the ray which goes in the direction normal to the sphere.

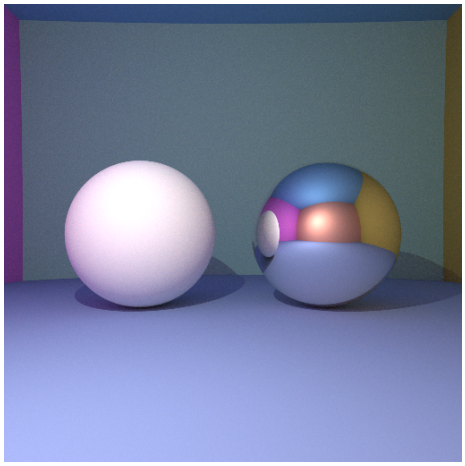

Figure 2: Diffuse and mirror spheres - *9.5s rendering time*

3 Transparent surfaces (optional)

Defining transparent surfaces is an interesting additional task. Rays can continue living by passing through spheres, taking into account the Snell-Descartes law. To make our spheres hollow and transparent, we create a second sphere with the same centre and slightly smaller radius, but with inverted normals. We can then optionally add Fresnel reflection, which relies on Schlick's approximation and randomly launches either a reflection or a refraction ray using the `random` library.

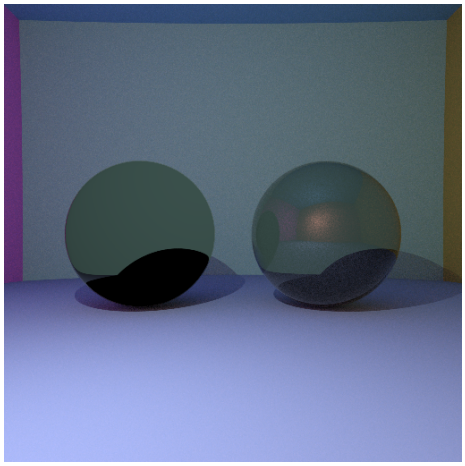

Figure 3: Transparent spheres without (left) and with (right) Fresnel reflection - *9s rendering time*

4 Direct lighting and shadows

After adding direct lighting, we can also add shadows to our spheres. For each pixel, we perform another intersection check to see if any object blocks the path between itself and the light source. If it is the case, we blacken that zone, which brings crucial realism to our image.

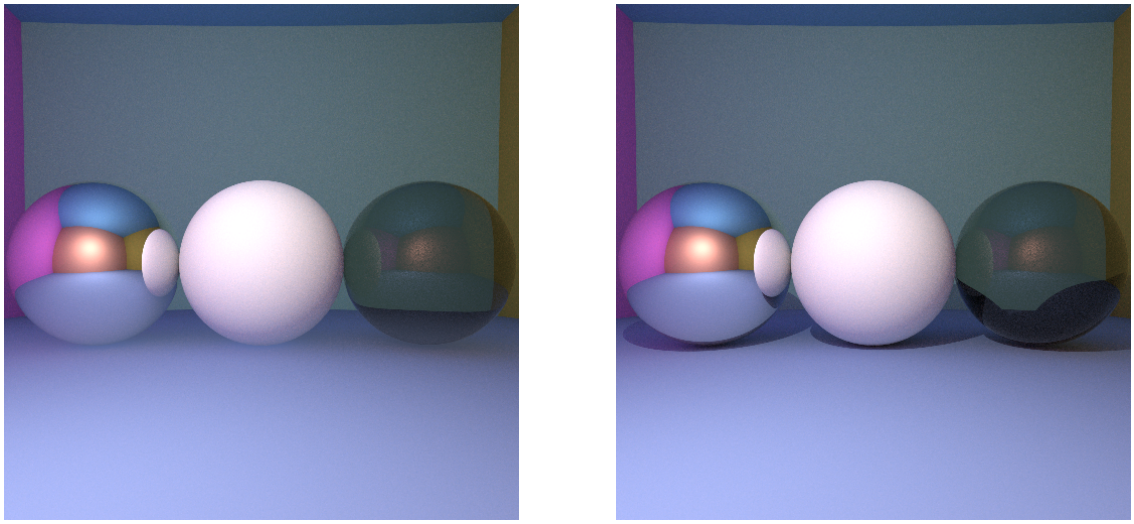

Figure 4: Spheres without shadows (left) vs. control image (right) - *7.4s rendering time*

5 Indirect lighting

The next step is to add indirect lighting (without Russian roulette) to our environment. We implement that feature in our `getColor` function, by fetching the colour of an indirect ray starting at the intersection point and going around the normal, and adding it to the original ray colour. This gives a much more realistic result, since objects that are not pure black reflect visible light.

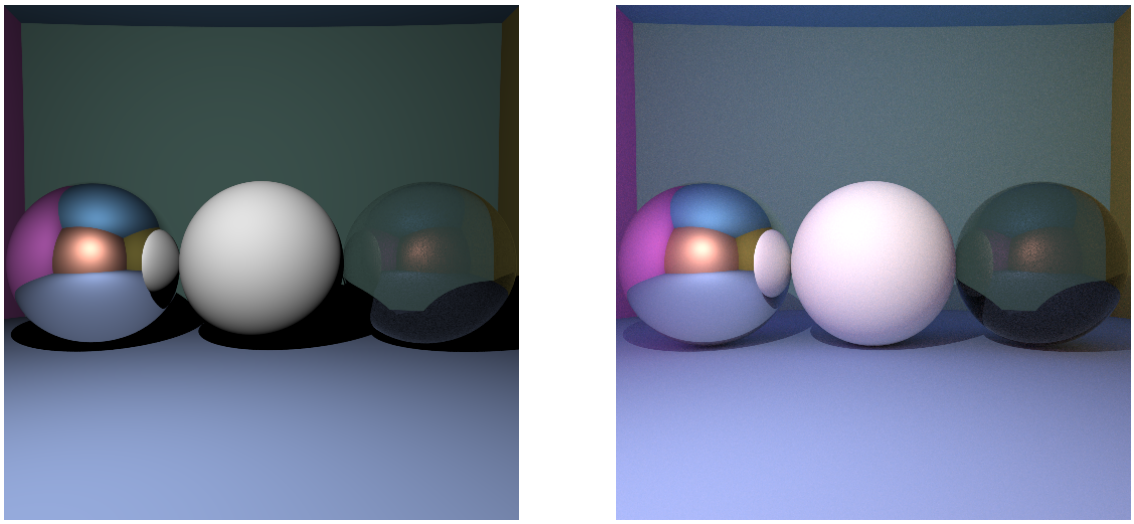

Figure 5: Spheres without indirect lighting (left) vs. control image (right) - *2.1s rendering time*

6 Anti-aliasing

The implementation of anti-aliasing also greatly helps improving realism. In our physical continuous world, edges are sharp. However, on images made up of discrete pixels, maintaining this sharpness is counterproductive, as it creates unrealistic colour jumps on the image. A simple way to tackle that issue is to use pixel jitters, which artificially smoothens edges, drastically improving realism.

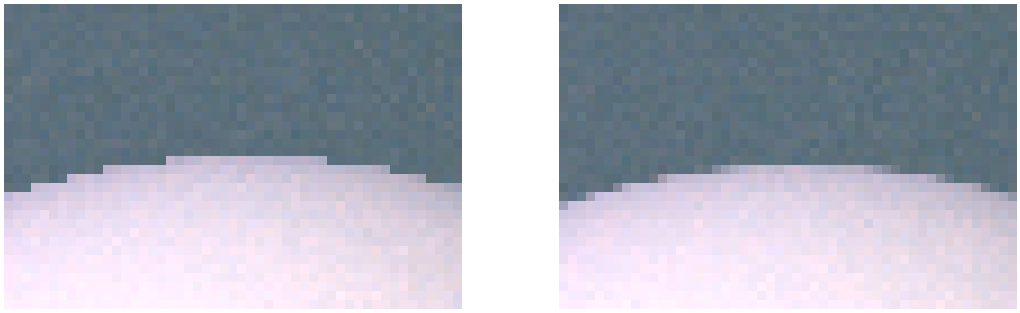

Figure 6: Sphere edge without anti-aliasing (left) vs. control edge (right) - *9.6s rendering time*

7 Depth of field (optional)

Another stylistic feature we can implement is depth of field. Simulating the effect of focus on camera lenses, we define a focus distance and an aperture. Using formulas from optics, we virtually blur out the objects that are further from the focus distance. This allows us to get some nice effects and better showcase the element in the foreground.

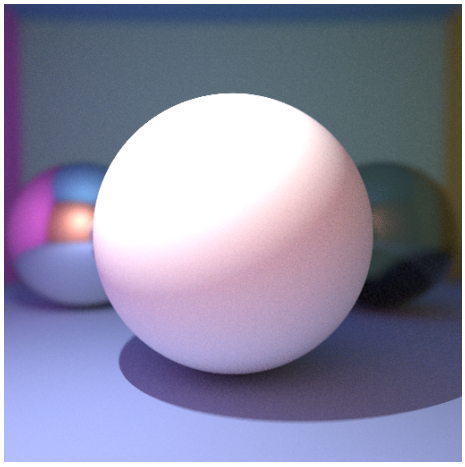

Figure 7: Spheres image with depth of field - *9.9s rendering time*

8 Ray mesh intersection including BVH

Using the given `readOBJ` function, we can define a new sort of geometric shape, `TriangleMesh` which allows us to model any object from an object file. Using many triangles and the Möller–Trumbore intersection algorithm, and accelerating our algorithm with a bounding box and Bounding Volume Hierarchies (BVH), we get an efficient algorithm capable of rendering a cat with high quality (128 rays per pixel) within an acceptable time frame (under 1 min).

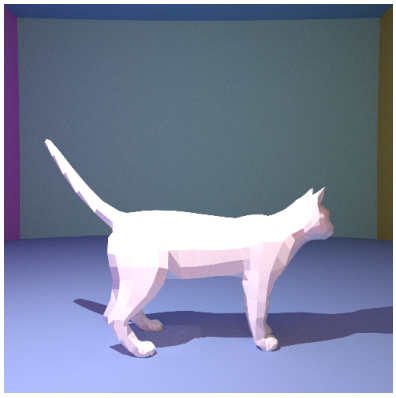

Figure 8: Base cat model - *57.1s rendering time*

9 Interpolation (optional)

Building on the computed barycentric coordinates, we can interpolate values on the mesh to make our cat more realistic by giving it a smoother shape through Phong interpolation.

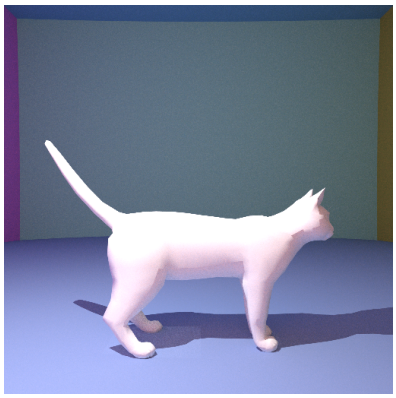

Figure 9: Cat model with interpolation of normals - *52.9s rendering time*

10 Textures (optional)

Vertices can also be associated to a 2D point within a texture map. Building on the given `stb_image.h`, we enable our ray tracer to fetch the albedo of the ray based on an image containing the object's texture. In our case, this approach gives us a cat with a realistic skin.

Figure 10: Cat model with textures - *53.3s rendering time*