CSE306 Assignment 1

pedro.cabral

May 2023

1 Introduction

We provide a Raytracer project in C++ which aims to create realistic 2D images from 3D scenes. In Section 2 we do an overview of the features supported by our Raytracer. We reproduce images to demonstrate examples of each feature. Figure 1 shows multiple features of our Raytracer working together.

Our goal is to make all image examples given in this report reproducible. Therefore, we created a file in `/scenes/gx_scenes.h` containing the code for each of the scenes shown in this report. To generate Figure i of this report, run the `render` program with $i - 1$ as a command line argument. Moreover, all figures can be found in the `/images` folder.

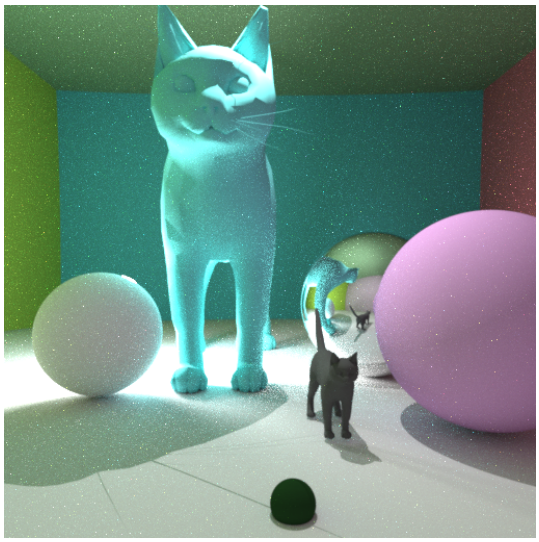

Figure 1: Image of two cats and five spheres. One of the spheres is a light source and it is hidden behind the white sphere. This image took 10 minutes to render with 1000 rays per pixel and 5 bounces per ray.

2 Features

2.1 Vector

The most fundamental files are `gx_vector.h` and `gx_vector.cpp`. It implements the Vector class, used to represent positions, vectors, and colors in our raytracer. It also implements the `Matrix` class, which is a useful tool for the rotation of objects.

2.2 Object

We have the `gx_geometry.h` file that defines the `Object` virtual class and the classes that implement it. In the file `gx_geometry.cpp`, we implement the `Sphere` and `TriangleMesh` classes. In our raytracer, we will render these two types of objects.

When a ray hits an object, we generate an `ObjectHit` object. It contains important information about the hit:

- `Vector P`: The point where the ray hit the object
- `Vector N`: The normal of the object at `P`
- `double tHit`: The distance that `P` is from the ray origin
- `Object *object_ptr`: A pointer to the object

One of the key components of the raytracer is the intersection logic. We use update semantics with the `ObjectHit` object to simplify the code and avoiding make unnecessary copies of variables. For all objects in the scene, we call the `bool Object.updateIntersect(const Ray &ray, ObjectHit &hitInfo)` method by passing the current `hitInfo` variable by reference. Then, `hitInfo` will only be updated if the ray intersects the object before `hitInfo->tHit`. The method returns `true` or `false` depending on whether an update happened.

For the `TriangleMesh` class, we implemented the BVH (bounding volume hierarchy) algorithm. We keep the BVH tree in a templated `binary_tree::BinaryTree<dataBVH>` class. The `BinaryTree` class is implemented independently in the file `binary_tree.h` in its own namespace. We build the `treeBVH` object in the `TriangleMesh` constructor. (Figure 8).

For both `TriangleMesh` and `Sphere` we implemented geometric transformations. These are: `transformTranslate`, `transformScale`, and `transformRotate`. They do exactly what their name specify. For the `TriangleMesh` class, we rebuild the `treeBVH` after every transformation.

2.3 Material

We keep the properties of an `Object`'s material in the `Material` class, defined in `gx_material.h`. This simple class contains information such as albedo, mirror, transparent, and light. We implement diffuse, mirror, and transparent materials. (Figures 2, 3 and 8)

2.4 Camera Models

The files `gx_camera.h` and `gx_camera.cpp` define and implement three different Camera models: `AliasedCamera`, `PinholeCamera`, and `ProjectiveCamera`. Each camera model implements a `generate_ray` method, that generates a (possibly random) ray from the camera center to the specified pixel. The `AliasedCamera` is the first camera model explained in the lecture notes. It does not perform antialiasing: the camera shoots a ray to the center of the pixel (Figure 5). The `PinholeCamera` implements `generate_ray` by shooting a ray distributed according to a gaussian

distribution centered at the pixel center (Figure 6). This camera is able to produce much smoother edges than `AliasedCamera`. Finally, `ProjectiveCamera` implements Depth of Field (DoF) by taking two extra parameters in its constructor: the focal distance and the radius of the aperture (Figure 7).

2.5 Rendering a Scene

We implement both direct and indirect lighting for our raytracer. Moreover, our raytracer supports both point and spherical light sources (Figure 4). In all the examples below, we used 5 bounces per ray.

3 Examples for each feature

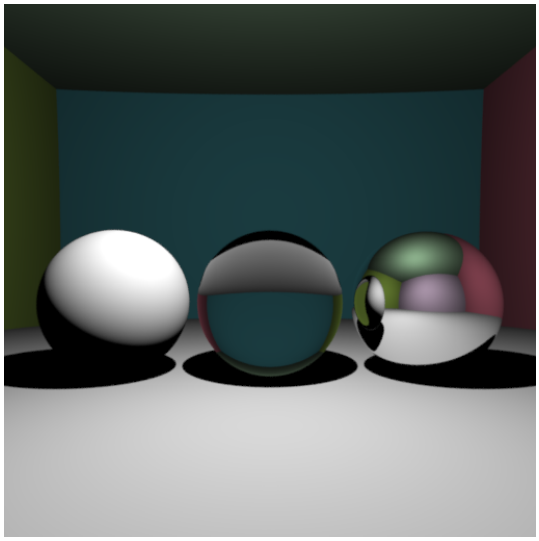

Figure 2: **Diffuse, Mirror, and Transparent surfaces.** Three different spheres: Diffuse (to the left), Transparent (in the middle), and Mirror (to the right). It took 1.9s to render using 100 rays per pixel and no indirect lighting.

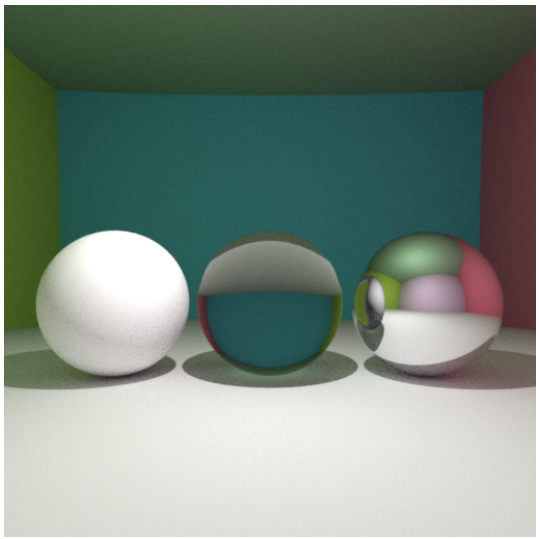

Figure 3: **Indirect Lighting** The same setting as Figure 2 but with indirect lighting and 5 bounces per ray. It took 10.5s to render with 100 rays per pixel.

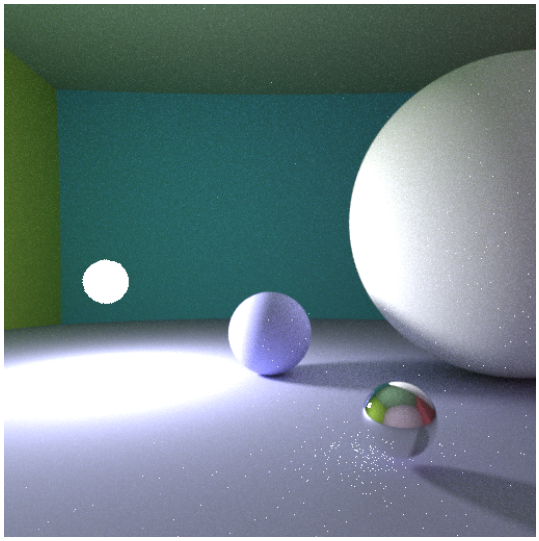

Figure 4: **Spherical Light Soruce.** This image shows the effect of a sphere light source. Because of the noise in indirect lighting, we shoot 400 rays per pixel. It took 56.6s to render.

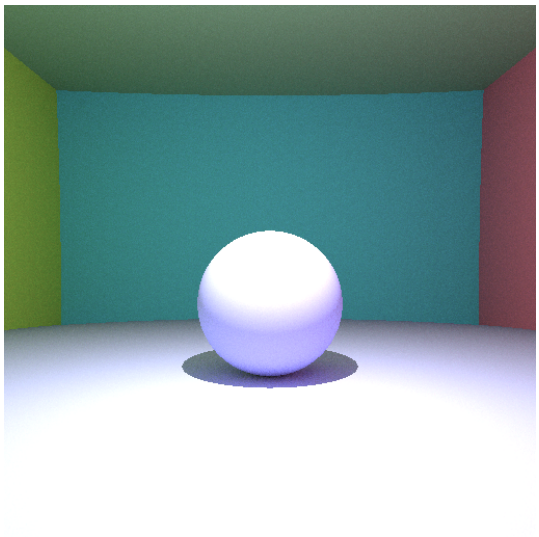

Figure 5: **Aliased Camera.** Image produced by an instance of `AliasedCamera`. This image takes 6.5s to render with 100 rays per pixel. Notice that the edges are not smooth.

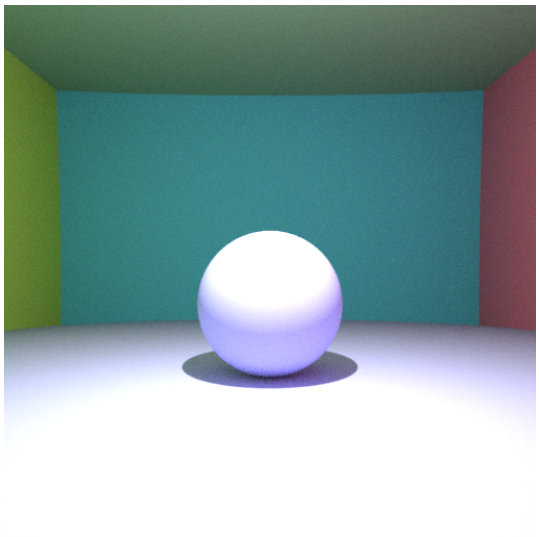

Figure 6: **Pinhole Camera with Antialiasing.** The same image produced by an instance of `PinholeCamera`. This image takes 7s to render with 100 rays per pixel. Notice that the edges are much smoother than in the previous image.

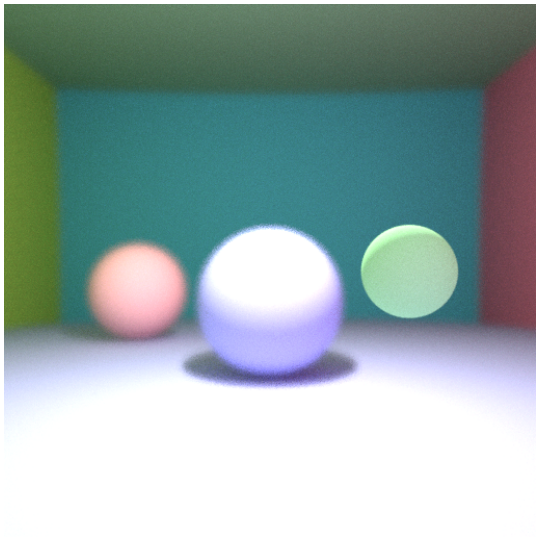

Figure 7: **Projective Camera and Depth of Field.** Image of three spheres at different depths produced by `ProjectiveCamera`. Notice that the farther spheres are out of focus. This image took 7.8s to render with 100 rays per pixel. The focal distance was 10 and the radius of the aperture was 5E-2. The spheres are approximately 10, 45, and 70 units of distance away from the camera. Notice how the farther spheres are blurred while the closest one is sharp.

Figure 8: **Triangle Meshes (with BVH).** Three different cats. The left cat is a mirror, the middle cat is diffuse, and the right cat is transparent. Notice that the three cats have different orientations and sizes. The image takes 32.4s to render with 100 rays per pixel.