Department of Computer Science and Software Engineering

University of Canterbury

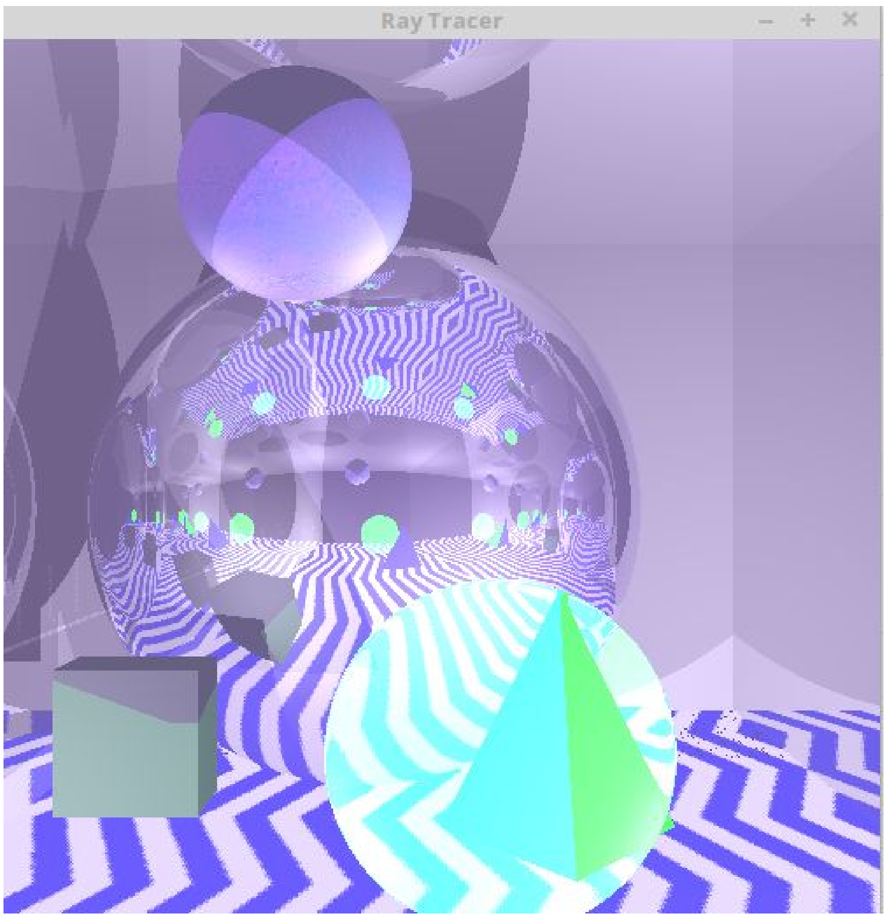
COSC363-18S1 Assignment Two Report

Ray Tracer

by

Robert Loomes (55938778)

Report submitted on 07/06/2018



Build instructions:

g++ -o “%e” Plane.cpp Ray.cpp RayTracer.cpp SceneObject.cpp Sphere.cpp Tetrahedron.cpp TextureBMP.cpp -lGL -lGLU -lglut

Overview:

The ray tracer builds on the exercises developed in the lab, which provided the code skeleton to work around. The functionality of the ray tracer lets the program handle several different types of geometric objects and global illumination features. Beyond the lab functionality, this ray tracer shows lighting, shadows, and reflections.

Run times:

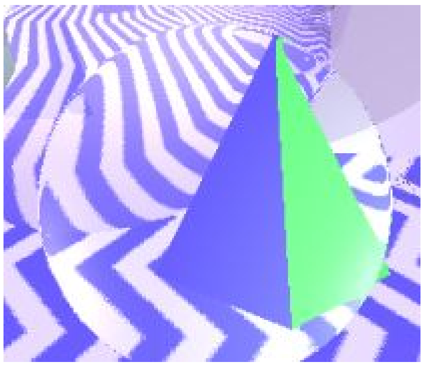
If you limit the window to a 500x500 pixel size, then the image usually renders in around 20 seconds. Rendering full screen can be upwards of 1 minute.

Extra Features:

1. Tetrahedron- a standard tetrahedron is constructed using a series of planes. 4 planes are generated like so:

```
Tetrahedron *tet1 = new Tetrahedron(glm::vec3(10, -15, -75),
glm::vec3(0, -15, -75), glm::vec3(5, -5, -75), glm::vec3(0.0, 0.0,
255.0));
```

This let’s us define each plane of the tetrahedron. After 4 planes have been created, it is just a matter of lining them up to achieve the desired shape.

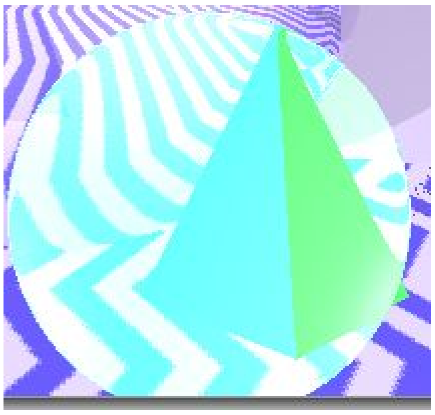


Here I made the transparent sphere clear to make the tetrahedron a little more visible.

2. Multiple light sources-

 2 light sources with specular and ambient parameters are present at 2 different points in the scene. The way they are positioned lets some objects cast 2 shadows at different directions. An object should only be cast 2 shadows if it satisfies the ray-shadow constraints.

3. Transparent object (with minor refraction)-



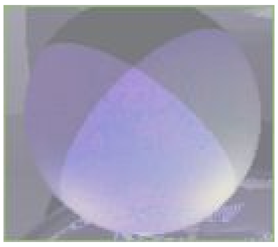
There is a sphere within the scene that displays partial transparency. Also note that refraction is occurring as the rays pass through the sphere, which distorts the floor pattern. Transparency is implemented by multiplying the sphere's RGB value with a transparency percentage e.g.

```
glm::vec3(0.0, 255.0, 0.0) //this denote a fully green colour
```

If we wanted colour to be 50% transparent, then we would simply calculate a new colour value to use by multiplying the 2 together (255 * 0.5).
The above sphere is denoted by the following:

```
transparent_sphere = new Sphere(glm::vec3(2, -9.5, -60.0), 6.0,
glm::vec3(0.5, 100.0, 0.5));
```

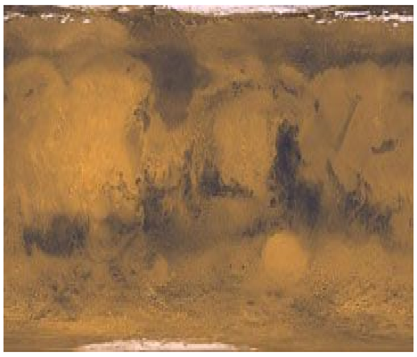
4. A non-planar object textured using an image-



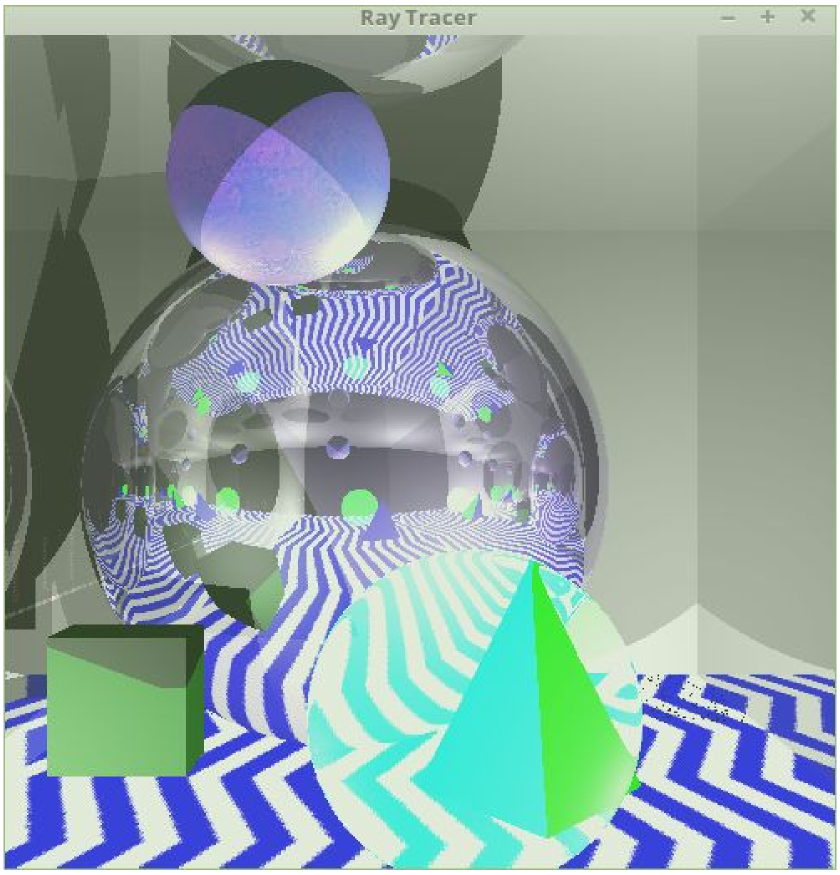
This sphere is actually a transparent sphere with a texture applied to it:

```
txId[0] = TextureBMP((char* const) "planet.bmp");
//for each ray...
col = col + txId[0].getColorAt(pt_s, pt_t);
```

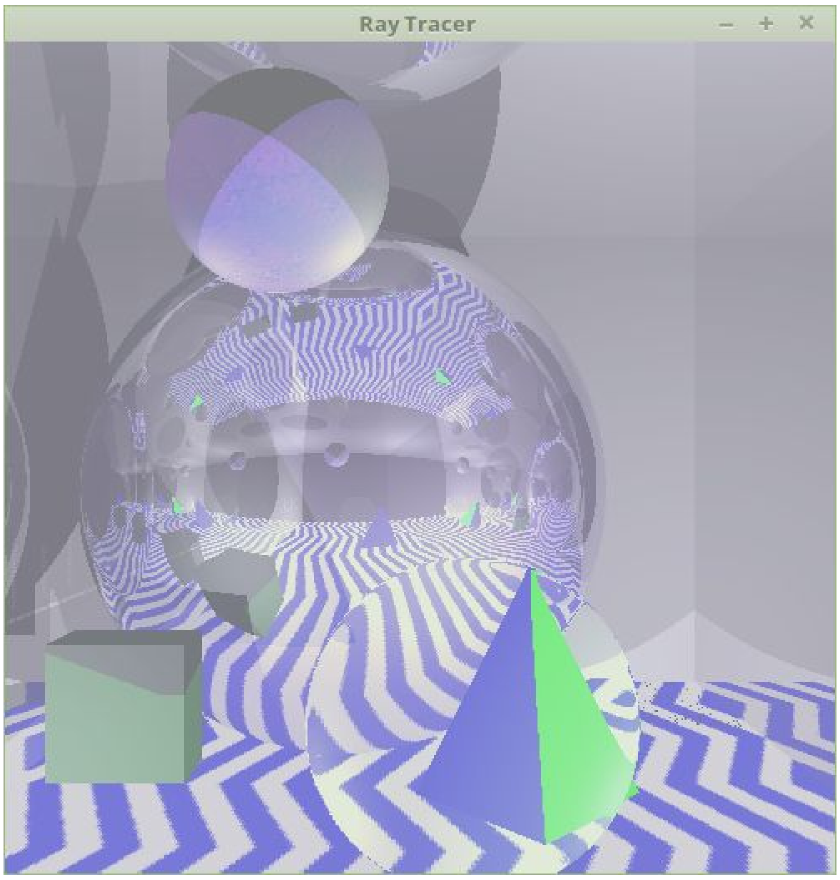
The original textured image is a simple 256*256 pixel square image:



5. Anti-aliasing- Distortion artefacts such as excessive jagged edges of objects and shadows are caused by the ultimately finite amount of rays that are being applied to the image. Each pixel is split into 4 segments and 4 rays are applied to the middle of each segment. After computation, the average colour of these 4 rays is used to colour the pixel. While subtle, anti-aliasing improves the overall image quality.



NO AA



AA- edges slightly smoother.

Successes/failures:

A lot of trial and error was needed as I was lacking a confident understanding of the ray tracing system. However, a lot was learned from the process and I would predict another ray tracing project could be completed in a small fraction of the time it has taken to complete this one.

The anti aliasing system was extremely hard to test and tune, and it was difficult to see the differences between implementations. Also, a higher resolution floor texture could have been used to make the anti-aliasing system more obvious.

References:

Cosc363 2018 lecture slides.

https://www.cs.cmu.edu/afs/cs/academic/class/15462-s09/www/lec/13/lec13.pdf

https://github.com/LWJGL/lwjgl3-wiki/wiki/2.6.1.-Ray-tracing-with-OpenGL-Compute-Shaders-(Part-I)