## Render times and build process

On my 2017 MacBook Pro (15 inch), the render takes 3min 30sec approximately at 1000x1000 pixels with anti-aliasing turned on.

One part of the build process that I set up specific to my machine was adding `"/usr/local/Cellar/glm/0.9.9.8/include"` to my `include_directories` in CMakeLists.txt. This allowed me to use glm on my machine instead of running the program with a VM. I also added a preprocessor block to include a different version of GLUT on macOS devices. The project appears to render on the UC Lab machines, but if something does not work it is likely to be caused by one of those changes.

## Ray Tracer

The scene rendered in RayTracer.cpp shows a family standing around a crystal ball (with cones representing bodies and cubes representing heads). The front-left family member has a semi-transparent sphere representing its head instead of a cube. I have decided that this is because that family member is an ‘air-head’. The full rendered scene is shown in figure 1.

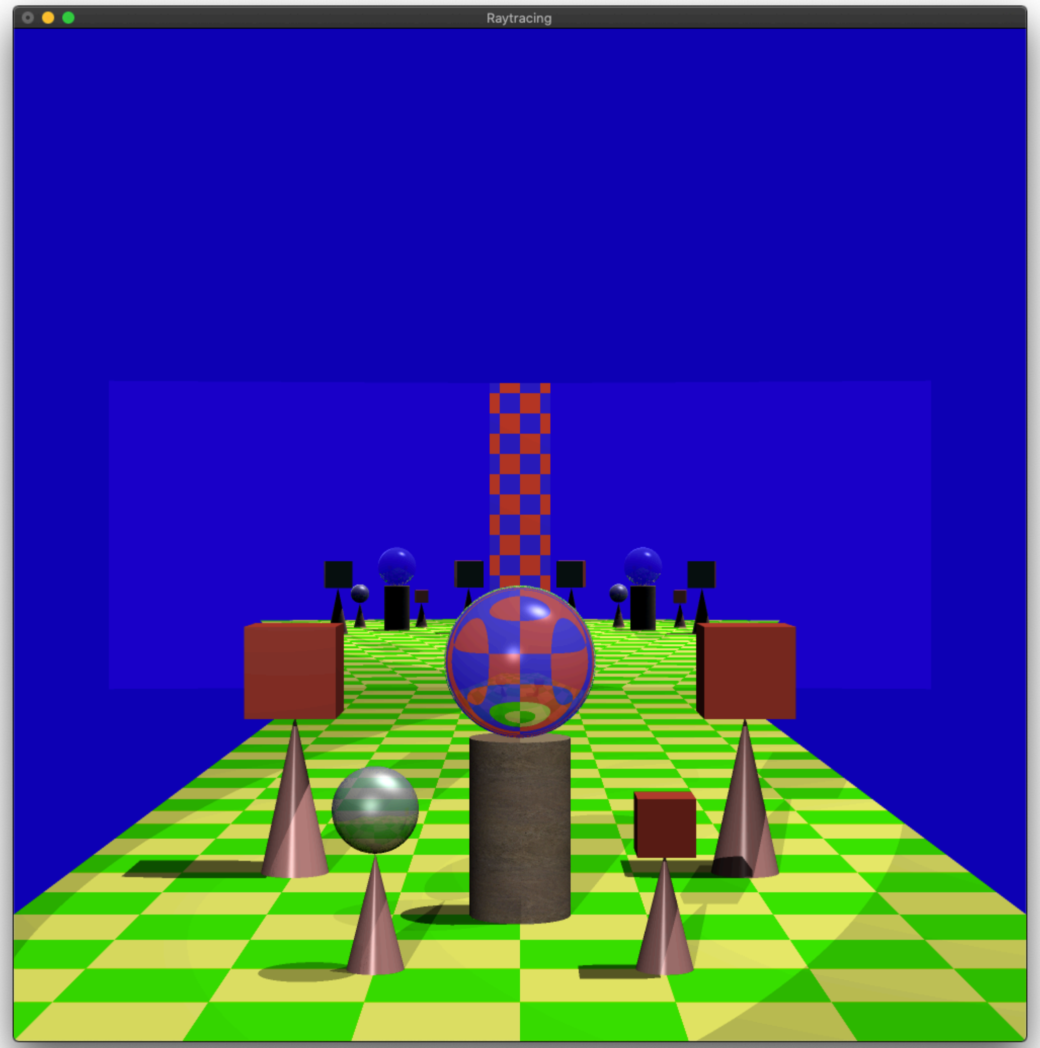


*Figure 1: Full ray traced scene*

A success of my ray tracing implementation is that an object can have more than one property that affects the colour values on its surface. For example, the sphere representing a crystal ball is both refractive and slightly reflective.

One thing I did not manage to fully implement in my application was a Torus. I followed a blog post tutorial to implement the mathematics[1], and took functions to solve 4th order polynomials from a source file from the Graphic Gems repository (which provides implementations of code taken from the textbook of the same name) on GitHub[2], but was unable to have the torus correctly render in positions other than the origin. A comparison is shown in figures 2 and 3.

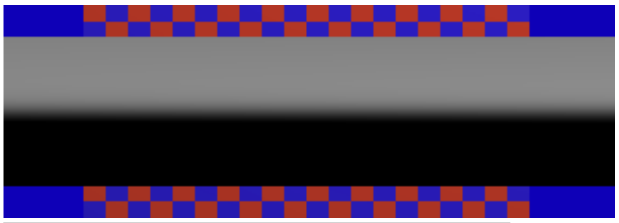


Figure 2: torus rendered around camera (at origin)

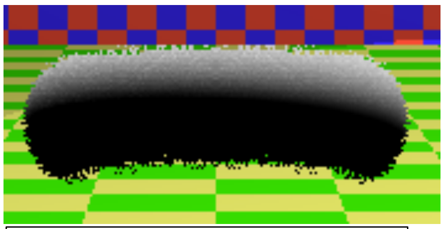


Figure 3: torus rendered outside of origin

## Multiple light sources

Multiple light sources have been used in the scene. The first light appears above and to the right of the family, and the second appears in front of and to the left of the family.
Multiple lighting has been implemented by iterating through a list of all the lights in the scene and performing lighting calculations for each source individually. Additionally, the distance of each light source has been included in the calculation, meaning that further away light sources contribute less light. Figure 4 below shows both light sources appearing reflected on a sphere, and figure 5 shows overlapping shadows caused by the multiple light sources. Note that only points blocked from both sources are in full shadow.

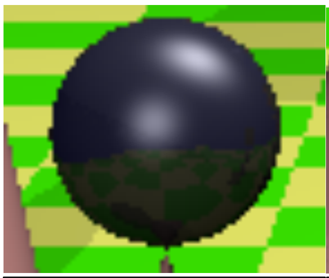


Figure 4: double light on sphere

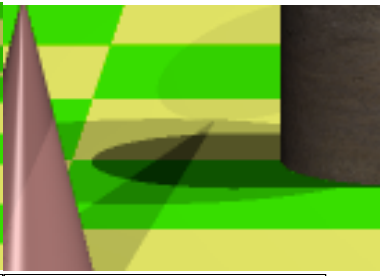


Figure 5: overlapping shadows

## Spotlight

The second light source in the scene has been implemented as a spotlight. When calculating the colour of a point, the shadow colour is used if the light source vector differs too greatly from the spot direction vector.

The calculation is not done for points already in shadow, so that no point is rendered with a "double shadow" from the same light (where points that would both be in shadow and are outside the spotlight appear much darker than regular shadows).

An example of a double shadow is shown in figure 6, with a correct rendering on the right in figure 7.

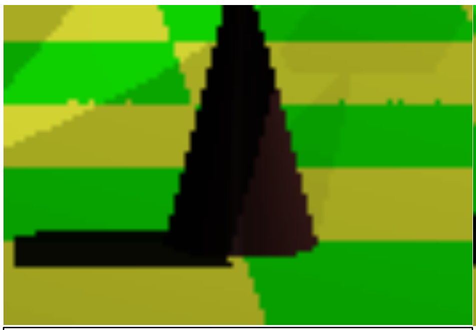


Figure 6: cone with double shadow behind it

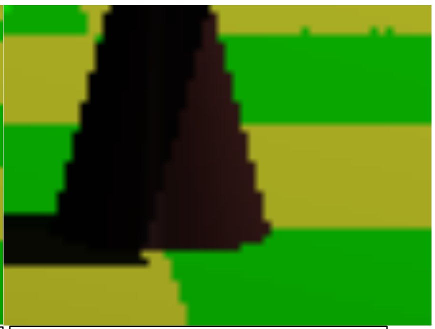


Figure 7: correctly rendered shadow

## Cylinder

Capped cylinders have been implemented as an extension of the SceneObject class. No caps are rendered underneath the cylinder, as the scene I was designing does not require showing the bottom of a cylinder.

## Cone

A cone object has been created by substituting the equation for a ray into the equation given in lectures for points on the surface of a cone, as seen below.

$$(pO_x + td_x - x_c)^2 + (pO_z + td_z - x_z)^2 = \left(\left(\frac{R}{h}\right)(h - pO_y + td_y + y_c)\right)^2$$

$$\begin{aligned} t^2d_x^2 &+ 2td_x(pO_x - x_c) + (pO_x - x_c)^2 \\ &+ t^2d_z^2 + 2td_z(pO_z - x_z)^2 + (pO_z - x_z)^2 \\ &= \left(\frac{R}{h}\right)^2\left((td_y)^2 + 2td_y(h + y_c - pO_y) + (h + y_c - pO_y)^2\right) \\ &= \left(\frac{R}{h}\right)^2(td_y)^2 + t\left(2\left(\frac{R}{h}\right)^2 d_y(h + y_c - pO_y)\right) + \left(\frac{R}{h}\right)^2(h + y_c - pO_y)^2 \end{aligned}$$

$$\begin{aligned} t^2d_x^2 &+ 2td_x(pO_x - x_c) + (pO_x - x_c)^2 \\ &+ t^2d_z^2 + 2td_z(pO_z - x_z)^2 + (pO_z - z_c)^2 \\ &- \left(\frac{R}{h}\right)^2(td_y)^2 - t\left(2\left(\frac{R}{h}\right)^2 d_y(h + y_c - pO_y)\right) - \left(\frac{R}{h}\right)^2(h + y_c - pO_y)^2 = 0 \end{aligned}$$

$t^2$ term: $d_x^2 + d_z^2 - \left(\frac{R}{h}\right)^2 d_y^2$

$t$ term: $2d_x(pO_x - x_c) + 2d_z(pO_z - x_z)^2 - 2\left(\frac{R}{h}\right)^2 d_y(h + y_c - pO_y)$

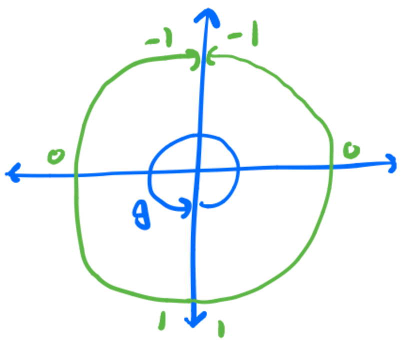
constant term: $(pO_x - x_c)^2 + (pO_z - z_c)^2 - \left(\frac{R}{h}\right)^2(h + y_c - pO_y)^2$

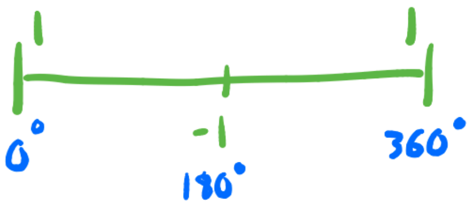
The substitution was then rearranged into a quadratic equation of the form $at^2 + bt + c = 0$ and solved using the quadratic formula for the intersection points t1 and t2. The cone's normal vector is also calculated using the equation given in the lecture notes.

## Textured cylinder

A cylinder has been textured with a wood texture, so that it looks like a wooden pole. The dot product of a pre-defined vector and the cylinder's surface normal at the hit point is used to calculate the texture coordinate.

Since the inverse cosine of two angles can map to the same value on the range [-1, 1], an x-coordinate check has been used to ensure that values on the right side of the cylinder wrap around correctly. This conversion means that each angle from 0-360 degrees maps to a single coordinate. The textured cylinder can be seen in the middle of the family in Figure 1.





## Anti-aliasing

Anti-aliasing has been implemented to smooth sharp changes in colour appearing in the ray-traced image. Each display pixel has been split into 4 quadrants, and rays are traced through the middle of each quadrant. The colour values of the split rays are then averaged to give the resultant pixel colour. This gives the render smoother edges, especially in places where there are sharp changes in colour (e.g. on shadow borders). No recursive splitting of quadrants has been implemented. The effects of anti-aliasing on the render can be seen in figures 8 and 9.

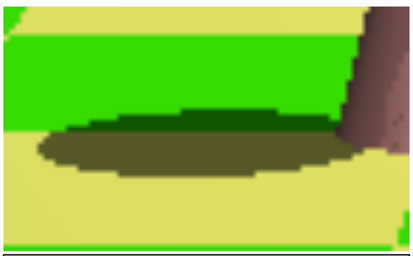


Figure 8: shadow with jagged edges

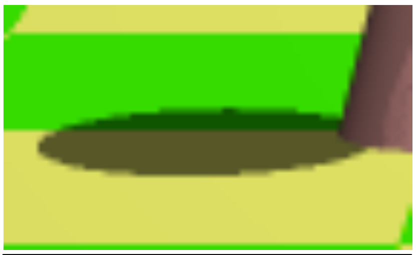


Figure 9: smooth anti-aliased shadow

## Refractive sphere

A sphere has been given refractive properties. An interior refracted ray is calculated using the hit point and the glm::refract method, which uses the object's refractive index to determine how extreme the refraction should appear. This ray is then used to find the exiting ray, which gives the colour to display on the sphere's surface at the original hit point using a recursive trace method call. Refractive objects also have lighter shadows than non-refractive objects. The refractive sphere can be seen on the podium at the centre of Figure 1.

## References

1. Marcin Chwedczuk. (2018, May 06). Ray tracing a torus [blog post]. Retrieved from https://marcin-chwedczuk.github.io/ray-tracing-torus
2. https://github.com/erich666/GraphicsGems/blob/master/gems/Roots3And4.c