COSC363 Assignment 2

Cameron Stevenson
57052612

Compilation/Running

I've included a VM lab box compiled labcompile.out, so run *./labcompile.out* . If that doesn't work try run *make* and then *./assignment2.out*

Minimum Features

This ray traced scene (Figure 1.1) features objects of interest in a line, hovering above a floor plane with a large reflective sphere in the background. A transparent object, the hollow sphere second from the left, is present (Figure 1.1). All objects cast shadows, with the transparent sphere having a lighter shadow (Figure 1.2). The scene features an object constructed using a set of planes, the cube on the left (Figure 1.1). The planar floor features a chequered pattern (Figure 1.1).

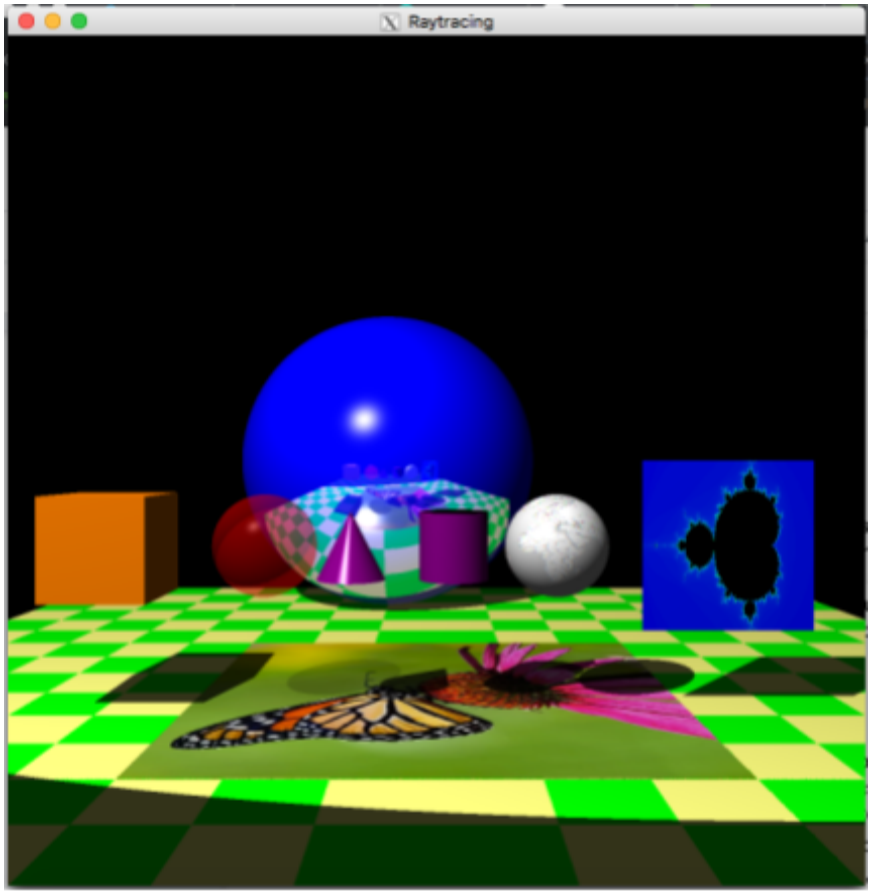


Figure 1.1

Figure 1.2

Extra Features

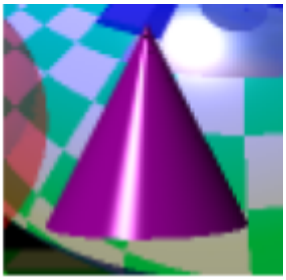


Figure 2.1

Cone (Figure 2.1) - the cone intersection is solved for two constraints. Firstly, the ray intersects with a circular cross section of the cone for some radius r, ($x^2 + z^2 = r^2$). Secondly, the radius of this cross section is dependent on the y coordinate of the intersection ($r = \frac{R}{H}(y - y_{center})$), where H is the height of the cone, R is the radius at the bottom, and y_{center} is the y coordinate of the point of the cone. Substituting in the ray intersect point $(x, y, z) = p0 + t * dir$, gives a quadratic in t which can be solved similarly to

the sphere, with the addition of discarding an intersection point if it exceeds the height of the cone. For normals a diff vector is constructed from the tip of the cone to the point of intersection. The cross product of this diff with the vertical axis of the cone gives a vector on the horizontal plane which is perpendicular to both. Importantly it is perpendicular to the diff and also the expected normal. So the normal is the cross product of the diff with this horizontal plane vector.

Cylinder (Figure 2.2) - for intersection this is a simpler case of the cone, where the radius is a constant instead of dependent on y. For normals, this is a simpler case of the sphere. Since the normal points away from a center vertical axis, it can be treated as pointing away from a center point but with the y component removed, then normalization fixes the length change.

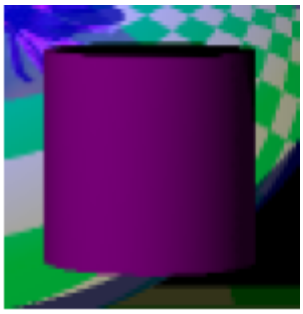
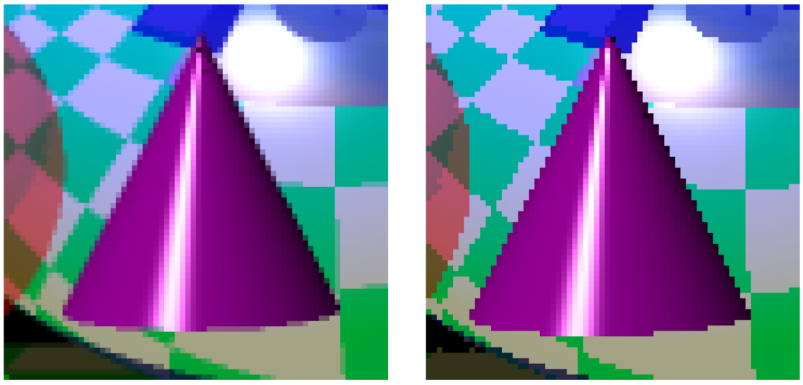


Figure 2.2

Spotlight - I turned the main light into a wide beam spotlight, so it lights all of the objects but you can see where the floor is outside of the beam (Figure 2.3). It is implemented by comparing the spotlight's direction vector to the vector from the light to the ray intersection, with a dot product of these vectors normalized compared against the cosine of the desired spotlight angle.

Figure 2.3

Anti-aliasing (Figure 2.4) - This is a simple constant multi-sampling, sampling each pixel 4 times in the centers of their four quadrants. It smooths out object edges well.



AA No AA

Figure 2.4

Textured non planar object (Figure 2.5) - the sphere has been textured with a globe texture from wikimedia[1]. The vertical axis of the texture corresponds directly to y-value, whilst the horizontal axis is wrapped around the sphere. The ray tracer uses the inverse of this mapping to take an intersection point and get a texture coordinate for sampling.

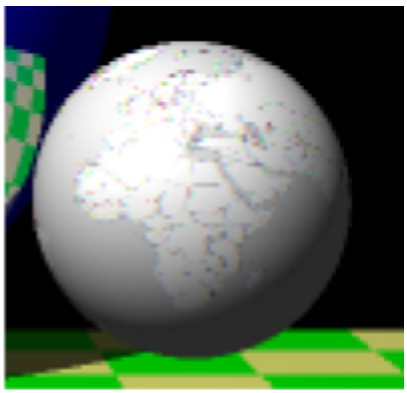


Figure 2.5

Procedural pattern (Figure 2.6) - I wanted to generate a pattern using a fractal, and chose the Mandelbrot set[2]. The Mandelbrot set is a set of points in the complex plane which

[1] https://commons.wikimedia.org/wiki/File:A_large_blank_world_map_with_oceans_marked_in_blue.gif
[2] https://en.wikipedia.org/wiki/Mandelbrot_set

satisfy a certain property. For a point c, take $f(z) = z^2 + c$, and with z starting at zero, repeatedly iterate this function (putting the output of $f(z)$ as the input z of the next iteration). If the output is bounded (there is some value for which the size of the output of $f(z)$ is always less than this value forever), then c is a member of the Mandelbrot set. This can be plotted with the complex plane as a plane in our scene, and each point can be coloured depending on if it is in the set or not. For each ray intersection, the value of c on the plane is determined, the iteration process is applied, and the point is coloured appropriately.

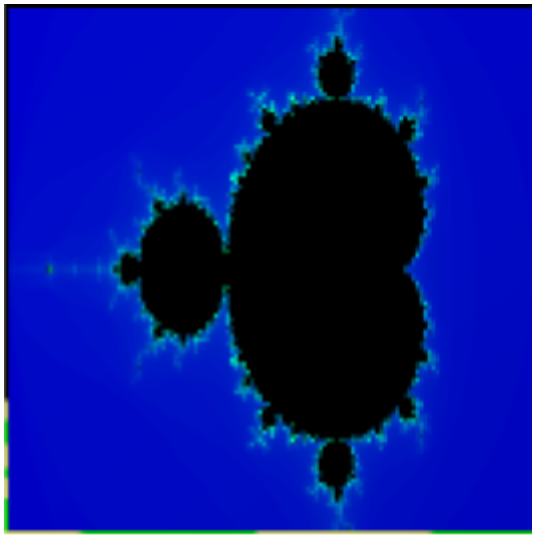


Figure 2.6

From codingame.com[3] I found out that if the size of z ever exceeds 2, then the iteration process will be unbounded and so can be stopped early. Otherwise you can keep iterating till some large maximum number of iterations and then reasonably assume a high probability of boundedness. The site also suggested colouring unbounded points based on how many iterations it took to confirm unboundedness. For a point not in the Mandelbrot set, this gives some measure of how close it is to being in the Mandelbrot set (cyan points in Figure 2.6) by the condition applied (which also happens to correlate to how close the point is to the border of the Mandelbrot set in the complex plane).

Performance

On my laptop this scene takes about 22 seconds to render at 650 by 650 pixels with anti-aliasing. Without anti-aliasing it takes about 6 seconds, which is about the expected time difference as each pixel is sampled 4 times with anti-aliasing.

Successes/Failures

I'm fairly happy with the final render itself, except for the clarity on the earth textured sphere and the Mandelbrot set plane. I tried without anti-aliasing but this substitutes blurriness for artifacting, so I haven't pursued that further. Perhaps if turning off anti-aliasing within the objects did make them clearer, I could make it so that each AA sample returns which object it has hit, and only apply anti-aliasing on object boundaries.

With regards to program structure, there's a few instances where I'd prefer to move logic into the source files for particular objects (such as deriving the texture coordinates from a plane intersection). I did this for sphere texturing, but from that it became clear that to do it properly I would have to rework SceneObject. This didn't seem worth it for the few objects I rendered, when I could just use the ray.index check to manipulate the one object I wanted to for each feature. So for now a lot of logic is in the trace function, but if this ray tracer was used seriously for a bigger scene I would classify all the features and how they overlap and create a proper SceneObject inheritance structure.

[3] https://www.codingame.com/playgrounds/2358/how-to-plot-the-mandelbrot-set/mandelbrot-set