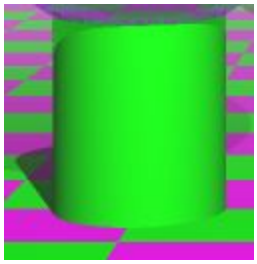
Henry Hickman
32028065

COSC363: Ray Tracer Report

Build Commands: g++ -Wall -o "%e" "%f" Sphere.cpp SceneObject.cpp Ray.cpp Plane.cpp Cylinder.cpp Cone.cpp -lm -lGL -lGLU -lglut

Extra Features

1) Cylinder



The cylinder is constructed by taking the cylinder's height and radius . I calculate the points of intersection using the radius and then check if they fall within the height of the cylinder. I also calculate the surface normal. The equation of the ray is as given in the lecture notes. I calculate a, b, and c to define a quadratic equation.

$$a = (d_x^2 + d_z^2)$$
$$b = (d_x(x_0 - x_c) + d_z(z_0 - z_c))$$
$$c = (x_0 - x_c)^2 + (z_0 - z_c)^2 - r^2$$

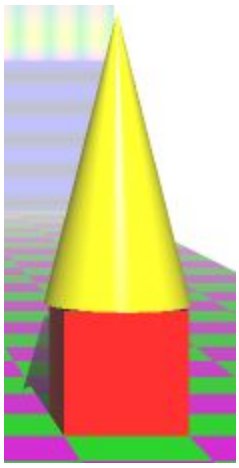
In the above equation (d_x, d_y, d_z) is the direction of the ray, (x_0, y_0, z_0) is the starting point of the ray. (x_c, y_c, z_c) is the center of the cylinder and r is the radius. I use the quadratic formula to find the real solutions to the equation $ax^2 + bx + c = 0$. I then check which of the two solutions are closer. If the solution falls within the range of the cylinder it is displayed, otherwise it is not. The surface normal is calculated by taking a vector n which is given by $(x_0 - x_c, 0, z_0 - z_c)$. The vector is then normalised using the glm::normalize function.

2) Cone

The cone is constructed by taking a given height and radius. I use both of these to calculate the points of intersection, as there is a relationship between height and radius of the cone. I also calculate the surface normal. I calculate a, b, and c to define a quadratic equation.

$$a = (d_x^2 + d_z^2 - d_y^2 tan^2(\theta))$$
$$b = (2((x_0 - x_c)d_x) + 2(z_0 - z_c)d_z + 2(h - y_0 + y_c)d_y)(tan^2(\theta)$$
$$c = (x_0 - x_c)^2 + (z_0 - z_c)^2 - 2(h - y_0 + y_c)^2(tan^2(\theta)$$

Henry Hickman
32028065



In the above equation (d_x, d_y, d_z) is the direction of the ray, (x_0, y_0, z_0) is the starting point of the ray. (x_c, y_c, z_c) is the center of the cone, $tan(\theta)$ is radius/height and h is the height of the cone. I use the quadratic formula to find the real solutions to the equation $ax^2 + bx + c = 0$. I then check which of the two solutions are closer. If the solution falls within the range of the cone it is displayed, otherwise it is not. The surface normal is calculated by taking a vector n which is given by $(x_0 - x_c\ , tan(\theta)\sqrt{(x_0 - x_c)^2\ +\ (z_0 - z_c)^2},\ z_0 - z_c)$. The vector is then normalised using the glm::normalize function. This gives a beautiful cone that my cube wears as a birthday hat.

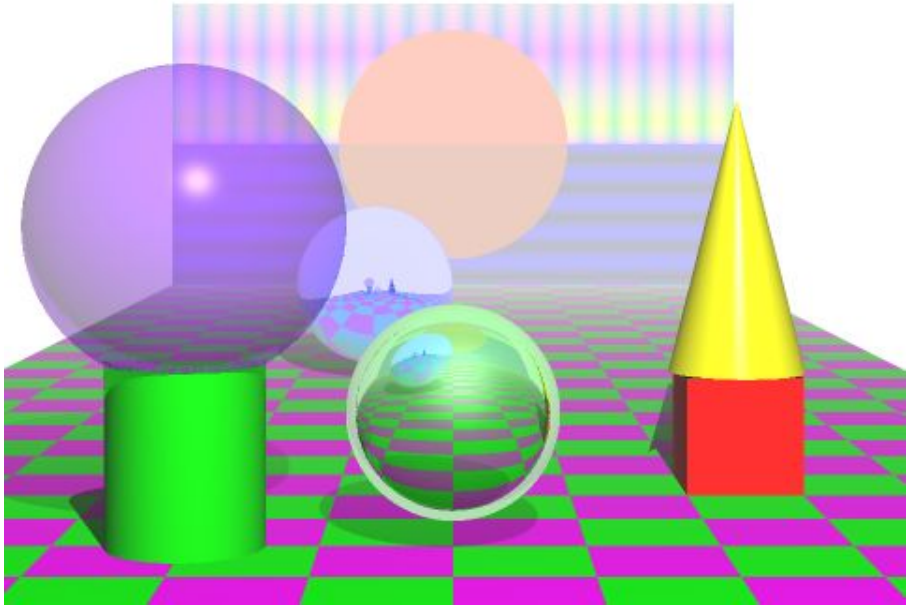
3) Refraction

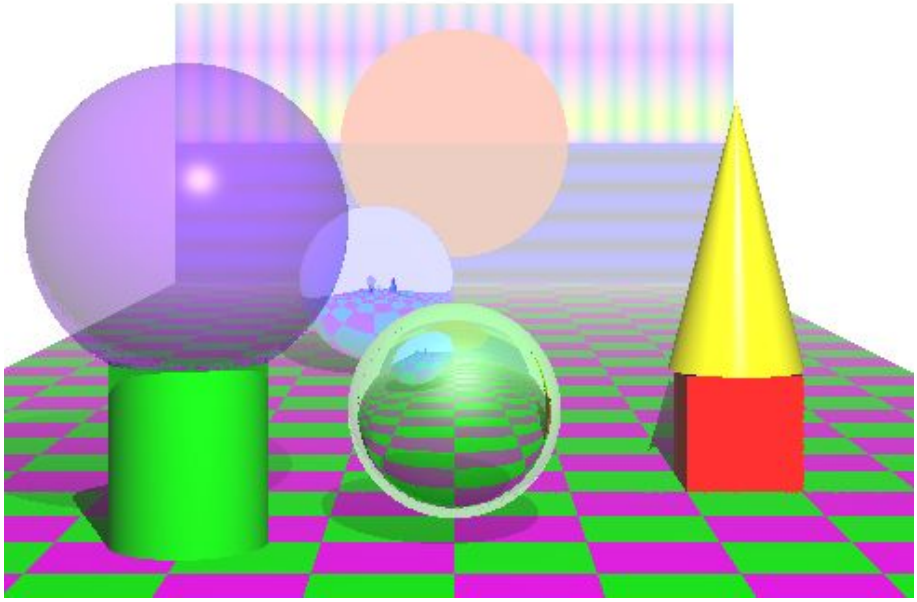
My refractive sphere was made by taking the location the ray hit the sphere and turning it into a normal vector. That vector is then used to make a new vector based on the direction of the ray, the normal vector, and a refraction index (ETA). This second vector, g, is used to create a refractive ray based on where the ray hit and g. A second normalised vector, m, is created from where the refracted ray hits. Another vector, h, is made using the direction or the initial ray, -m, and 1.0f/ETA. The refracted ray is then used to get the color of the refraction. If any of the rays do not hit anything they return a background color. This results in a refractive sphere.

4) Anti aliasing

Each pixel is subdivided into four pixels. Each pixel then has a ray generate the colour value. The colour value is then averaged to get a smoother image. This removes jagged edges on the image making it far more appealing to look at. In order to show the difference between the anti-aliasing I have changed the number of divisions from 500 to 300, as it more clearly demonstrates the impact of the anti aliasing. The top image has used anti-aliasing. The bottom image does not.

Henry Hickman
32028065





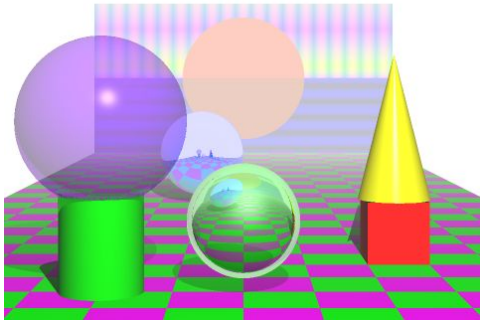
Henry Hickman
32028065

5) Procedurally Generated Patterns

```
if (ray.index == 12 && step < MAX_STEPS) {
    if (ray.hit.x * ray.hit.x + ray.hit.y * ray.hit.y < 150 && ray.hit.y > 0) {
        color = glm::vec3(1, 0.2, 0);
    }
    else if (ray.hit.x * ray.hit.x + ray.hit.y * ray.hit.y < 150 && ray.hit.y < 0) {
        color = glm::vec3(0.7, 0.2, 0);
    }
    else if (ray.hit.y >= 0) {
        color = glm::vec3(fabs(cos(ray.hit.x)), fabs(sin(sqrt(ray.hit.y))), fabs(cos(sqrt(ray.hit.y))));
    }
    else {
        color = glm::vec3(0, 0, fabs(cos(ray.hit.y)));
    }
}
```

The pattern on the back plane of the ray tracer is procedurally generated. The code for the two semicircles are the first two if statements. The remaining pattern is generated by assigning the colour values based on the location of the ray. These formulas were made by plugging things in until it looked pretty.

6) Fog



The fog is generated by setting the background color to white, defining a range for the fog (z_1, z_2). We then find $t = (ray.hit.z - z_1)/(z_2 - z_1)$. We then use t to set the color value to $(1-t)color + t * backgroundCol$. This generates a fog between the range.

Successes and Failures

My ray tracer had colours chosen that were vibrant and made to pop. I believe this made it very eye catching. A failure of my ray tracer was the initial inability to make the procedurally generated pattern more interesting. Initially it was just the top section without the circle, but despite the calculations that went into creating that it still looked like not very complicated stripes. Another failure came from when implementing the anti aliasing. It was not very noticeable at higher resolutions, but became more noticeable as I turned down the number of divisions.

Rendering Time

On my computer with 16GB of RAM and a Ryzen 3600 processor the scene takes approximately 10 seconds to run.

References

Lecture notes from Dr R. Mukundan.