Andrew Clifford
19798632

Andrew Clifford - COSC363 Assignment 2 - Ray Tracer

Build

To build the program, navigate to the root directory of the project in terminal and run the following commands:

cmake .
make
./RayTracer.out

Estimated render time with anti-aliasing: 35 seconds
Estimated render time without anti-aliasing: 10 seconds

Description of Ray Tracer

The program is built off the basic Ray tracer implementation from Lab 7 and Lab 8. The basic implementation included a floor plane and a few spheres with shadows and reflections. My Ray tracer has extended this implementation to include refraction through a glass sphere, a transparent sphere, a cylinder, multiple light sources, anti-aliasing, a box constructed using planes. The scene is shown in Figure 1 below.

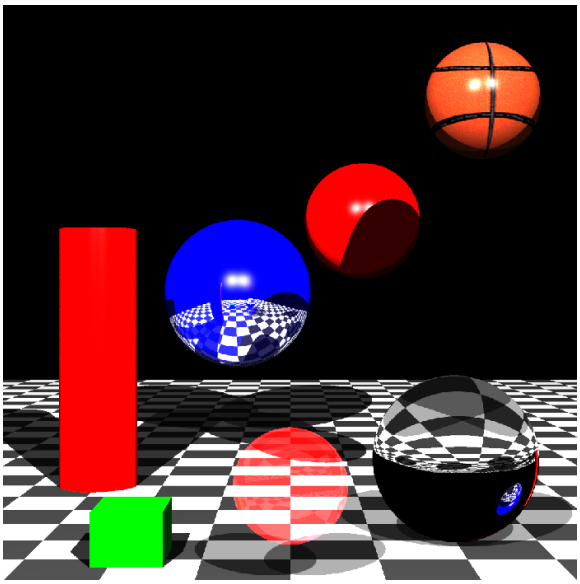

Figure 1 Scene

Minimum Requirements (9 Marks)

Transparent Object

As seen in Figure 1 above, I have implemented a red transparent sphere. This was achieved by recursively tracing rays in the same direction as the ray at the point of intersection with the transparent object.

Shadows

Shadows were generated by casting a shadow ray to each light source from the point of intersection. If the shadow ray intersected another object before hitting the light source, then the pixel was in

shadow. Shadows were made to be 60% lighter for refractive and transparent objects to make the scene more realistic.

Object constructed using a set of planes

A green box was constructed using a set of 6 planes as seen in Figure 1 above.

Chequered Pattern on planar surface

I have implemented a Chequered pattern on the floor plane of the scene. This was done by extending the stripe pattern code given in the labs by introducing a variable in the x direction. The Chequered pattern can be seen in Figure 1 above, while the implementation of this is seen in Figure 2.

```
//Chequered floor
if (ray.index == 4) {
    int modx = (int)((ray.hit.x + checkSize) / 5) % 2;
    int modz = (int)((ray.hit.z + checkSize) / 5) % 2;

    if((modx && modz) || (!modx && !modz)) {
       color = glm::vec3(0.2,0.2,0.2);

    } else {
       color = glm::vec3(1,1,1);
    }
    obj->setColor(color);
}
```

Figure 2 Chequered floor implementation

Extensions (Max 7 Marks)

Cylinder

I implemented a Cylinder class, which includes the functions necessary to calculate the intersection and normal. I used the intersection equation from the lectures (Figure 3) to work out the roots of the equation as seen in Figure 4. The normal function is also shown in Figure 5. An example of a red cylinder is seen in Figure 1 above (Description section).

$$t^2\left(d_x^2 + d_z^2\right) + 2t\left\{d_x\left(x_0 - x_c\right) + d_z\left(z_0 - z_c\right)\right\} + \left\{\left(x_0 - x_c\right)^2 + \left(z_0 - z_c\right)^2 - R^2\right\} = 0.$$

Figure 3 Intersection equation of a Cylinder (From lectures)

```
float x0minusxc = pos.x - center.x;
float z0minuszc = pos.z - center.z;

float a = pow(dir.x, 2) + pow(dir.z, 2);
float b = 2 * ((dir.x * x0minusxc) + (dir.z * z0minuszc));
float c = (x0minusxc * x0minusxc) + (z0minuszc * z0minuszc) - (radius * radius);
float discrim = pow(b, 2) - 4 * (a * c);

//Check if discriminant is valid
if(fabs(discrim) < 0.001 || discrim < 0.0) {
    return -1.0;
}

//Get the roots of the equation
float t1 = (-b - sqrt(discrim)) / (2 * a);
float t2 = (-b + sqrt(discrim)) / (2 * a);
```

Figure 4 Calculating the roots of the intersection equation

```
glm::vec3 Cylinder::normal(glm::vec3 p)
{
    glm::vec3 n = glm::vec3(p.x - center.x, 0, p.z - center.z);
    n = glm::normalize(n);
    return n;
}
```

Figure 5 Normal function for a Cylinder

Refraction

I set sphere sphere3 to be a refractive glass sphere with a refractive index of 1.5. The shadow color of refractive objects is set to be only 30% of the normal shadow color for a more realistic look. The implementation for refractive objects and the result of sphere3 can be seen in Figure 6 and Figure 7 respectively.

```
//Refractive glass sphere
if (obj->isRefractive() && step < MAX_STEPS) {
    float eta = 1 / obj->getRefractiveIndex();
    glm::vec3 g = glm::refract(ray.dir, normalVec, eta);
    Ray refrRay(ray.hit, g);
    refrRay.closestPt(sceneObjects);
    glm::vec3 m = obj->normal(refrRay.hit);
    glm::vec3 h = glm::refract(g, -m, 1.0f/eta);
    Ray refrRay2(refrRay.hit, h);
    glm::vec3 r = trace(refrRay2, step + 1);
    color = 1.0f * r;
}
```

Figure 6 Refraction implementation

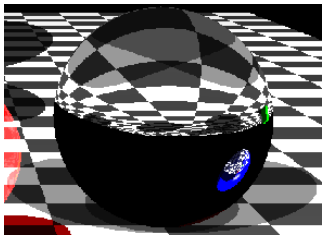

Figure 7 Refraction example

Multiple Light Sources / Multiple shadows

In the scene there are two light sources. This makes the scene a lot brighter and two specular highlights can be seen on the reflective objects. Because there are two light sources, this means that at the point of intersection, two shadow rays were cast (one to each light source) instead of one, resulting in multiple shadows for some objects. Darker shadows were generated where two shadows overlapped for a more realistic look as seen below where (if both shadow rays hit an object) the color value is only half of what it is if only one of the shadow ray hits an object.

```
//Both shadows
if ((shadowRay.index > -1 && shadowRay.dist < glm::length(lightVec)) &&
(shadow2Ray.index > -1 && shadow2Ray.dist < glm::length(lightVec1))) {
    if (shadow2Obj->isRefractive() || shadow2Obj->isTransparent()) {
        color = 0.35f * (obj->getColor());
    } else {
        color = 0.1f * obj->getColor();
    }
}
```

Figure 8 Multiple Shadows implementation

Anti-aliasing

This was achieved by splitting up each pixel in the scene into four subpixels. Primary rays were then traced through the center of each of these subpixels to return a color value. The color value of each of these subpixels is then averaged to give the color value for the original pixel. In doing this, it removed the jaggedness of the edges of both objects and shadows which can be seen in Figure 9 (no anti-aliasing) and Figure 10 (anti-aliasing). The pitfall of this is that the scene now takes a lot longer to load (35 seconds compared to 10 seconds) as there are now four times the primary rays to be traced. The implementation of anti-aliasing is seen in Figure 11 below.

Note: anti-aliasing can be toggled on and off by setting antialiasing = true/false at the top of RayTracer.cpp.

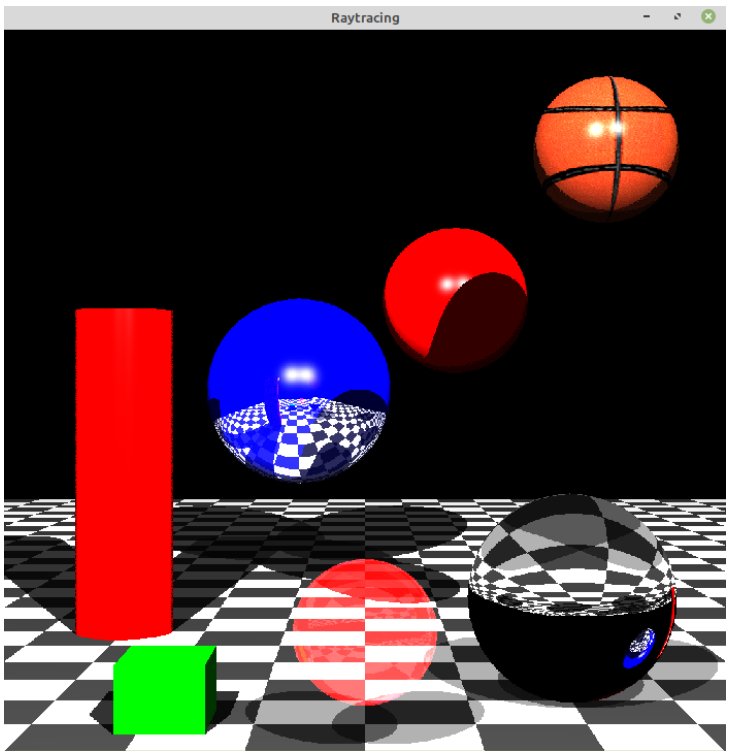

Figure 9 Scene without anti-aliasing

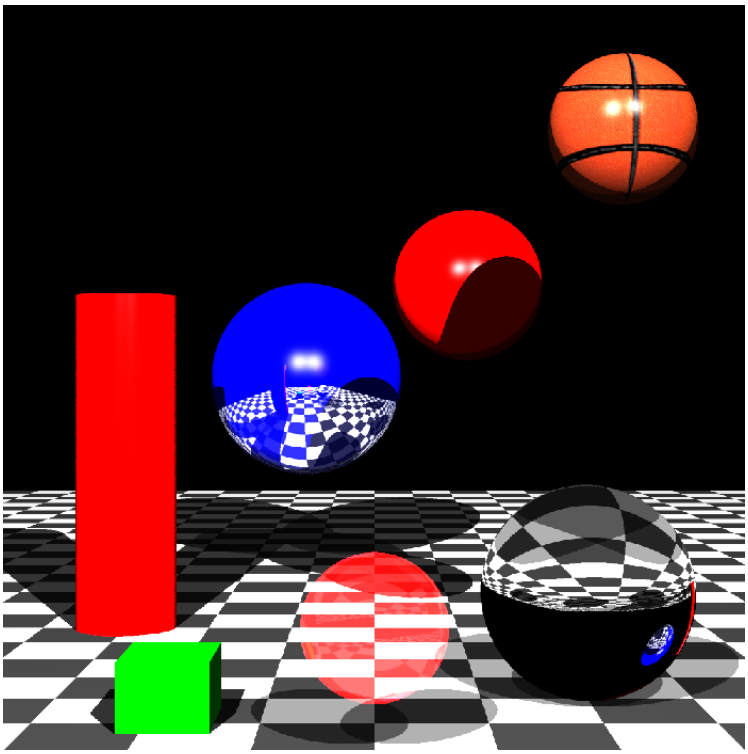

Figure 10 Scene with anti-aliasing

```
if (antiAliasing) {

    //Split the pixel into four subpixels
    glm::vec3 directions[4] {
        glm::vec3(xp+0.25*cellX, yp+0.25*cellY, -EDIST),
        glm::vec3(xp+0.75*cellX, yp+0.25*cellY, -EDIST),
        glm::vec3(xp+0.25*cellX, yp+0.75*cellY, -EDIST),
        glm::vec3(xp+0.75*cellX, yp+0.75*cellY, -EDIST)
    };

    glm::vec3 colorSum = glm::vec3(0, 0, 0);

    for (int i = 0; i < 4; i++) {
        Ray ray = Ray(eye, directions[i]);
        //ray.normalize();
        colorSum += trace(ray, 1);
    }
    //Now take the average of the colorSum to decide the color of the pixel
    col = colorSum * glm::vec3(0.25);

} else {
    Ray ray = Ray(eye, dir);
    col = trace (ray, 1); //Trace the primary ray and get the colour value
}
```

Figure 11 Anti-aliasing implementation

Non-planar object textured

The file ball.bmp (a basketball texture) is mapped to a sphere as seen in Figure 12. I achieved this through the use of some equations I found online with respect to UV mapping which are shown in Figure 13.

Figure 12 Texture mapped ball

```
//Texture map basketball onto sphere
if(ray.index == 1) {
    glm::vec3 c(10.0, 10.0, -60.0); //Centre of the sphere
    glm::vec3 d = glm::normalize(ray.hit - c); //Unit vector

    float texcoords = 0.5 + (atan2(d.x, d.z) / (2 * M_PI));
    float texcoordt = 0.5 - (asin(d.y) / M_PI);
    if(texcoords > 0 && texcoords < 1 &&
       texcoordt > 0 && texcoordt < 1)
    {
        color = texture.getColorAt(texcoords, texcoordt);
        obj->setColor(color);
    }
}
```

Figure 13 Sphere texturing implementation

Successes and Failures

- A success I had during the assignment was texturing non-planar objects. The Wikipedia page on UV mapping was very helpful and made it easy to understand the mapping process.
- Antialiasing was also very rewarding to implement as the quality difference was very noticeable.

- I struggled to implement the Cylinder and Cone classes. This was mainly due to not having a good understanding of how the intersect methods should be implemented. Although I did manage to implement the Cylinder in the end, I didn't manage to implement the Cone which was disappointing.
- Another failure was that the way I implemented anti-aliasing wasn't 'smart'. The best way to implement this would have been to only split the pixel into subpixels where necessary (i.e. at the edges of objects), however I didn't manage to implement it this way. If I had managed to do this, it would have allowed the scene to render much quicker.

References

Basketball texture - https://www.uihere.com/free-graphics/vector-abstract-basketball-ball-texture-ai-file-773/download

Wikipedia UV Mapping - https://en.wikipedia.org/wiki/UV_mapping