COSC363 Assignment 2 – Ray tracing

James Suddaby

#23934915

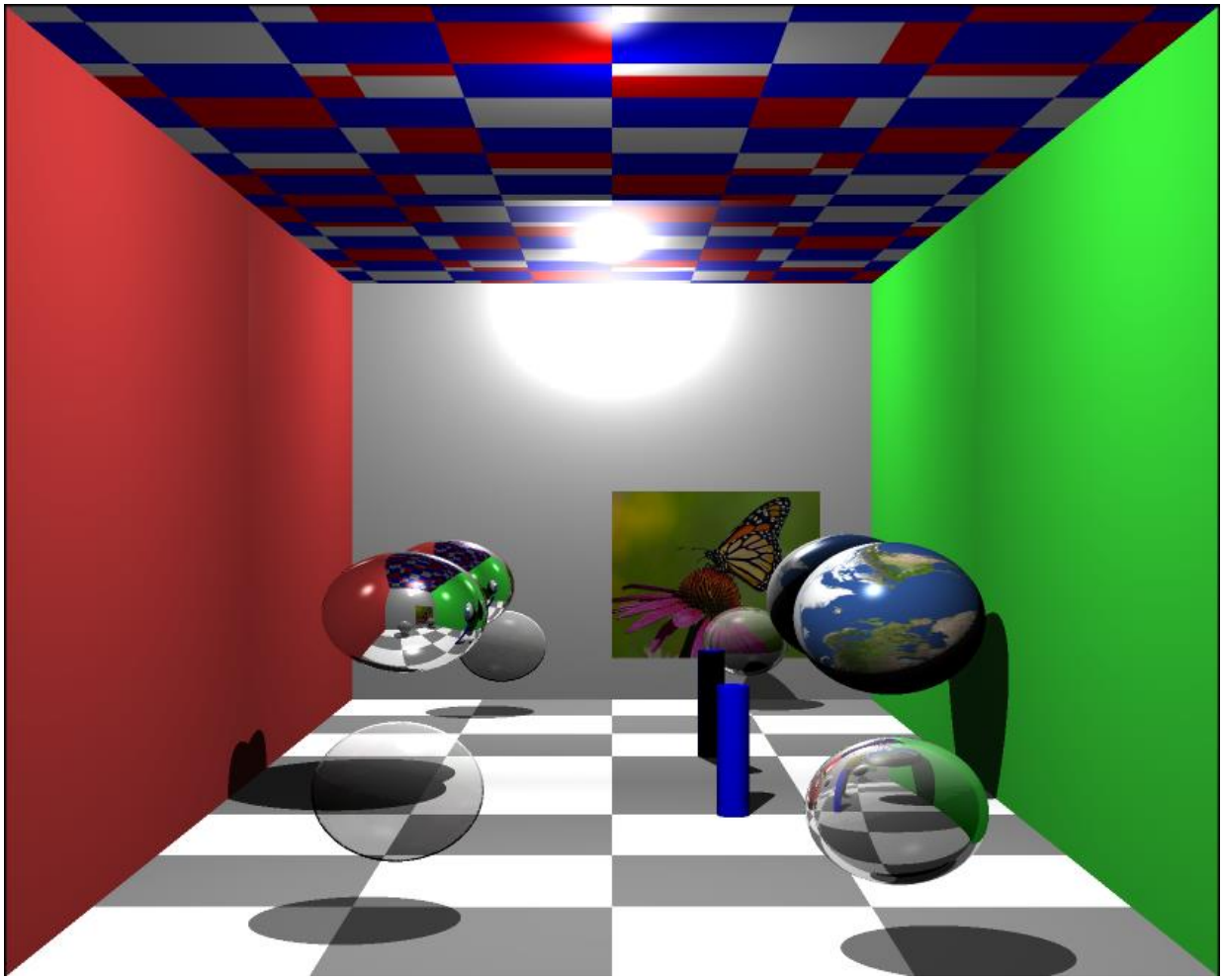


Figure 1: Ray tracer with AA enabled

Declaration
I declare that this assignment submission represents my own work (except for allowed material provided in the course), and that ideas or extracts from other sources are properly acknowledged in the report. I have not allowed anyone to copy my work with the intention of passing it off as their own work.
Name: James Suddaby
Student ID: 23934915
Date: 31/05/2024

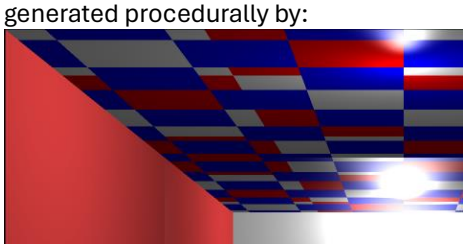
Overview

The Ray Tracer consists of a Cornell box with an assortment of objects/shapes with different properties inside. The floor is coloured with a checker pattern, the back wall is mirror/reflective surface and the roof a procedurally generated pattern. The ray tracer utilizes adaptive anti-aliasing to reduce jaggedness along the edges of shapes/colours.

Extra Features

1. Procedural pattern generated on any surface:

The pattern on the roof of the scene is generated procedurally by:



```
if(ray.index == 1) {
    float blueSize = (30);
    float redSize = (50);

    bool isZBlue = glm::mod(ray.hit.z,blueSize) > blueSize*0.5;
    bool isXBlue = glm::mod(ray.hit.x,blueSize) >blueSize*0.5;
    bool isZRed = glm::mod(ray.hit.z,redSize) >redSize*0.5;
    bool isXRed = glm::mod(ray.hit.x,redSize) >redSize*0.5;

    if ((isXBlue xor isZBlue)){
        color = glm::vec3(0,0,1);
    } else if ((isXRed xor isZRed )) {
        color = glm::vec3(1,0,0);
    } else {
        color = glm::vec3(1.,1.,1.);
    }
    obj->setColor(color);
}
```

2. Objects other than a plane or sphere -> Cylinder

A cylinder is constructed using the coordinates of the center of the base, a parameter for the height of the cylinder and a parameter for the radius of the cylinder.
As the cylinder is a subclass of sceneObject.cpp we need to implement methods for finding the closest point of intersection and the normal vector at any point on the cylinder.
Using the intersection equation given in the lecture notes: $t^2(d_x^2 + d_z^2) + 2t\{d_x(x_0 - x_c) + d_z(z_0 - Z_c)\} + \{(x_0 - x_c)^2 + (z_0 - z_c)^2 - R^2\} = 0$
This is in the quadratic form: $a \cdot t^2 + b \cdot t + c$

So we can take:

$$a = (d_x^2 + d_z^2),$$

$$b = 2 \cdot \{d_x(x_0 - x_c) + d_z(z_0 - Z_c)\},$$

$$c = \{(x_0 - x_c)^2 + (z_0 - z_c)^2 - R^2\}$$

And use the quadratic formula to solve for the intersection points t:

$$t = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

This gives us the two points where the ray intersects with the cylinder, now we must return the closest point. The code for this is given in Figure 2

```
/**
* Cylinder's intersection method.  The input is a ray.
*/
float Cylinder::intersect(glm::vec3 p0, glm::vec3 dir){
   glm::vec3 vdif = p0 - center;

    float a = dir.x * dir.x + dir.z * dir.z;
    float b = 2 * (dir.x * vdif.x + dir.z * vdif.z);
    float c = vdif.x * vdif.x + vdif.z * vdif.z - radius * radius;
    float delta = b * b - 4 * a * c;

    if(delta < 0.001) return -1;

    float t1 = (-b - sqrt(delta)) / (2 * a);
    float t2 = (-b + sqrt(delta)) / (2 * a);

    float pt1 = p0.y + t1 * dir.y;
    float pt2 = p0.y + t2 * dir.y;

    if (pt1 > center.y + height && pt2 < center.y + height)
    {
        return (center.y + height - p0.y) / dir.y;
    }
    if (pt1 > center.y + height || pt1 < center.y)
    {
        t1 = -1;
    }
    if (pt2 > center.y + height || pt2 < center.y)
    {
        t2 = -1;
    }
    return (t1 > t2) ? (t2 >= 0 ? t2 : t1) : (t1 >= 0 ? t1 : t2);
}
```

Figure 2: Cylinder Intersection code.

The other important part of the cylinder is the method that returns the normal vector for any point on the cylinder.
The normal vector can be calculated by the equation: $n = (x_p - x_c, 0, z_p - z_c)$

```
*/
glm::vec3 Cylinder::normal(glm::vec3 p)
{
    const float epsilon = 1e-4f;

    // Check if the point is on the top cap
    if (fabs(p.y - (center.y + height)) < epsilon) {
        return glm::vec3(0.0f, 1.0f, 0.0f);
    }
    // Calculate the normal for the side surface
    glm::vec3 n = glm::vec3(p.x - center.x, 0.0f, p.z - center.z);
    n = glm::normalize(n);

    return n;
};
```

Figure 3: Normal method for cylinder.

Where p is a point which lies on the cylinder, c is the point at the center of the cylinder, n should be normalised to a unit vector. The code for calculating the normal of the cylinder can be taken from **Error! Reference source not found.**

3. Visible end cap on cylinder

The end caps of the cylinder must be implemented separately. In my ray tracer I implemented a cap on the top of the cylinder. The basic idea for modifying the intersection method to account for a cap is to calculate the y coordinate for the points where the ray intersects the cylinder by: $point_y = p_{0_y} + t * dir_y$ and to check if this is bigger or smaller than $c_y + h$ where center is the c is the coordinates of the cylinder and h is the cylinder's height. If that is the case, then we can return the intersection as the top of the cylinder. The code for this is shown in the bottom part of Figure 2

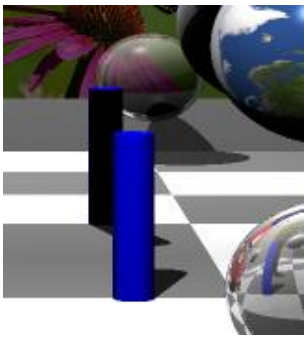
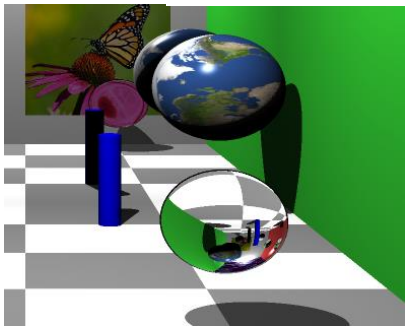


Figure 4: Cylinder with visible cap

The normal method just needs to check if the point is on top cap and return (0,1,0) as the normal vector. The code for this can be seen in the if statement in the code in **Error! Reference source not found.**

4. Refraction of light through an object

Refraction for a sphere was implemented by modelling Snell's law $n_1 \sin\theta_1 = n_2 \sin\theta_2$ and using the equations given in the lecture notes. The refractive indices that I used the refractive sphere in my ray tracer were 1/1.5 which causes rays to be flipped 180deg.



for

5. Fog

Fog is easy to implement and is a result of blending white to the colour of the ray in proportion to the size of the ray. I followed this equation from the lecture notes: $\lambda = \frac{(ray.hit.z) - z_1}{z_2 - z_1}$ and blended using: $color = (1 - \lambda)color + \lambda\, white$

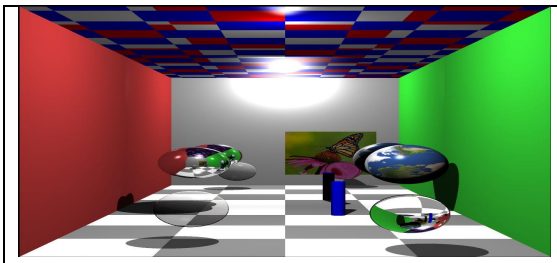


Figure 5: Ray tracer without fog

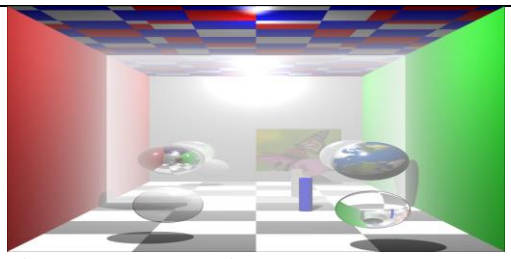


Figure 6: Ray tracer with fog

6. Object Space transformations

The reflective sphere given in **Error! Reference source not found.**7 has been space transformed by the transformation matrix given in Figure 8 8

```
glm::mat4 transform = glm::mat4(1.0);
transform = glm::scale(transform, glm::vec3(1, 1.1, 1));
```

Figure 8: Transformation matrix applied to sphere

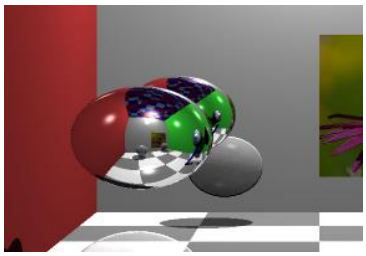
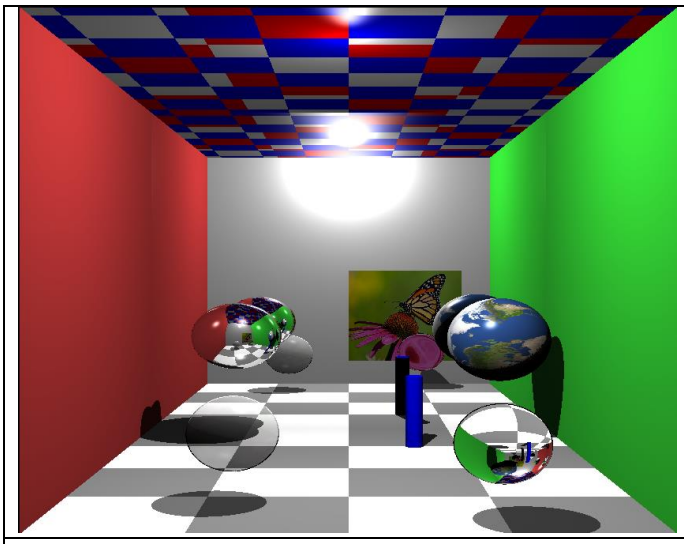


Figure 7: Transformed sphere

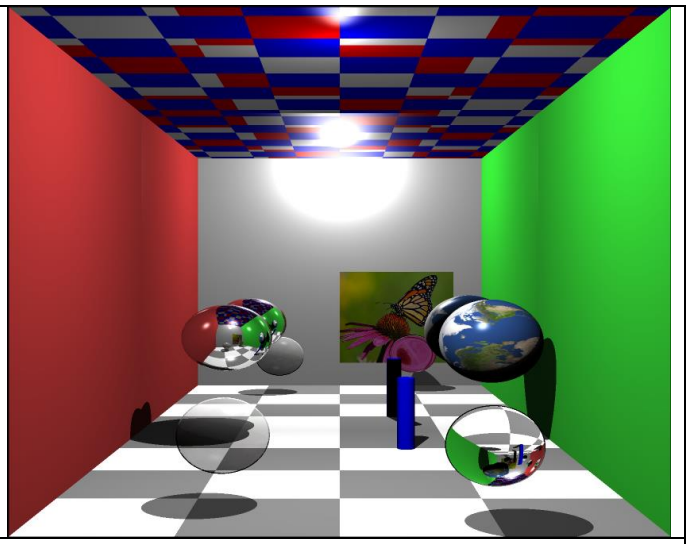
7. Adaptive Anti-Aliasing

Adaptive anti-aliasing is implemented in my ray tracer for reduction in jagged edges and an improvement in rendering speed over basic AA. In my ray tracer, I first iterate through the entire grid, drawing the rays and finding and storing the colour values of each pixel. I then loop through the grid again and check if for each colour value I have stored if it is an edge (if it distinctly different from it's 8 surrounding neighbours. If it is split that pixel into 4 subpixels and return the average colour value between them.

Adaptive anti-aliasing was much faster than basic super-sampling which took 29.68 seconds on the Lab PC compared with 13.8 seconds.



Without AA (8.87 sec on lab pc)



With AA (13.8 sec on lab pc)

8. Texture mapped sphere

My ray tracer includes a sphere which has been texture mapped with an image of the Earth. This was done by mapping the point of intersection on the sphere to the colour of a pixel in the image.

Code for this is shown in Figure 9

```
if (ray.index == 10 )
{
    glm::vec3 normalVec = obj->normal(ray.hit);

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

Figure 9: Code for texture mapping to a sphere

9. Multiple reflections generated using parallel mirror like surfaces

There a two mirror like surfaces in my ray tracer, one is the back wall, and the other is the transformed sphere. Reflections seen in the sphere can be seen in the reflection on the back wall. (Figure 10)

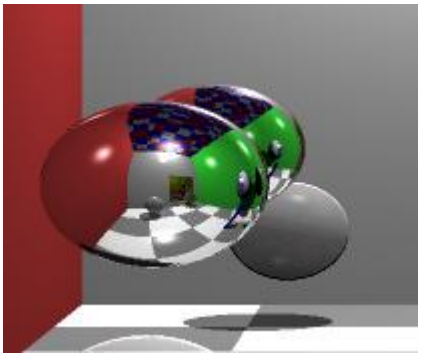


Figure 10: Multiple reflections

10. Image texture mapped to invisible wall

The image of the butterfly that appears to be on the back wall is a reflection of the front wall which is invisible to the rays going in the same direction as the normal of the front wall. I did this by checking the direction of the ray relative to the normal of the plane and return -1 if their dot product is > 0. This in effect lets the rays from the from the eye pass through the wall but not back through it.