Report for Ray Tracer Assignment

COSC363

Kai Koh

64276083

Hkk18

1 Build Process

- Open and run the CMakeLists.txt using an IDE (eg. QtCreator).
- Ensure working directory is the root directory of the repository or else images and other stuff will fail to load.
  - In this case, the textured sphere will not display the image and just appears black.
- You can choose primary ray tracing or antialiasing by uncommenting either one of the respective lines in the display() method in RayTracer.cpp:
  - glm::vec3 materialCol = antiAliasing(eye, cellX, xp, yp); //Anti-aliasing
  - glm::vec3 materialCol = trace(ray, 1); //Trace the primary ray and get the colour value

2 Failures

I did not manage to implement the spotlight and get the fog working as intended.

3 Extra Features

3.1 Cone

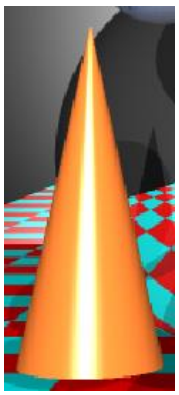
The cone is formed by calculating the surface normal and point of intersection of the ray. The ray equation used here is: $\mathbf{x = x_0 + d_x t, y = y_0 + d_y t, z = z_0 + d_z t}$, where 0 is the origin of the ray, d is direction of ray, t denotes distance between origin to the point on the ray. The intersection and normal have been calculated using the following equations:

```
float Cone::intersect(glm::vec3 pos, glm::vec3 dir)
{
    glm::vec3 d = pos - center;
    float y_change = height - pos.y + center.y;
    float tangent = (radius / height) * (radius / height);

    float a = (dir.x * dir.x) + (dir.z * dir.z) - (tangent * (dir.y * dir.y));
    float b = 2 * (d.x * dir.x + d.z * dir.z + tangent * y_change * dir.y);
    float c = (d.x * d.x) + (d.z* d.z) - (tangent * (y_change* y_change));
    float determinant = b * b - 4 * (a * c);
```

The intersection equation is: $\mathbf{t^2(d_x^2 + d_z^2 – \tan^2 d_y^2) + 2t(d_x(x_0 – x_c) + d_z(z_0 – z_c) + \tan^2(h – y_0 + y_c)d_y) + ((x_0 – x_c)^2 + (z_0 - z_c)^2 – \tan^2(h – y_0 + y_c)^2) = 0}$

```
glm::vec3 Cone::normal(glm::vec3 p)
{
    glm::vec3 d = p-center;
    float r = sqrt(d.x * d.x + d.z * d.z);
    glm::vec3 n= glm::vec3 (d.x, r*(radius/height), d.z);
    n=glm::normalize(n);
    return n;
}
```



Above is a screenshot of the cone that was created using the above equations, with a light source shining straight at the cone explaining the shiny surface of the cone's front view.

3.2 Cylinder and Texture

The cylinder object is also constructed using the point of intersection and normal of the ray. It also uses the same ray equation as the cone. As shown below, the equation for the point of intersection is somewhat similar to that of the cone object, except in the use of tangent squared in the latter and some other calculations.

```
float Cylinder::intersect(glm::vec3 pos, glm::vec3 dir)
{
    glm::vec3 d = pos - center;
    float a = (dir.x * dir.x) + (dir.z * dir.z);
    float b = 2 * (dir.x * d.x + dir.z * d.z);
    float c = d.x * d.x + d.z * d.z - (radius * radius);
    float determinant = b * b - 4 * a * c;
```

The intersection is obtained by solving the quadratic equation for t, where R is the radius of the circular plane of the cylinder: $\mathbf{t^2(d_x^2 + d_z^2) + 2t(d_x(x_0 - x_c) + d_z(z_0 - z_c)) + (x_0 - x_c)^2 + (z_0 - z_c)^2 - R^2) = 0}$

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

3.2.1 Procedurally Textured Pattern

The cylinder has also been textured in a procedural way, using the IF-ELSE condition:

```
if (int(ray.xpt.x - ray.xpt.y) % 2 == 0) {
    materialCol = glm::vec3(0.2, 0.2, 0);
} else {
    materialCol = glm::vec3(1,1,1);
}
```

All these equations ultimately generated the striped cylinder as displayed above. The top part of the cylinder that appears shaded due to one of the shadows being casted by the sheared cube that is located above the cylinder.

3.3 Tetrahedron

I have opted to construct a pyramid-shaped tetrahedron for this project, using planes to generate each face of the tetrahedron. Variables A-F are the points on the object, used altogether to create its faces, which are then pushed together to form a complete tetrahedron. I have set 2 different colours for the different faces of the tetrahedron to distinguish between them.

```
void drawTetrahedron(float x, float y, float z, float length, glm::vec3 color1, glm::vec3 color2)
{
    glm::vec3 A = glm::vec3(x, y, z);
    glm::vec3 B = glm::vec3(x+length, y, z);
    glm::vec3 C = glm::vec3(x+length*0.5, y, z + sqrt(3.0f) * 0.5 * length);
    glm::vec3 D = glm::vec3(x+length*0.5, y+sqrt(6.0f)/3.0f * length, z + sqrt(3.0f) * 0.25 * length);
    glm::vec3 E = glm::vec3(x+length/2, y, z);
    glm::vec3 F = glm::vec3((C.x+D.x)/2, (C.y+D.y)/2, (C.z+D.z)/2);

    Plane *face1 = new Plane(E, B, C, A, color1);
    Plane *face2 = new Plane(B, D, F, C, color2);
    Plane *face3 = new Plane(D, A, C, F, color1);
    Plane *face4 = new Plane(A, D, B, E, color2);

    sceneObjects.push_back(face1);
    sceneObjects.push_back(face2);
    sceneObjects.push_back(face3);
    sceneObjects.push_back(face4);
}
```

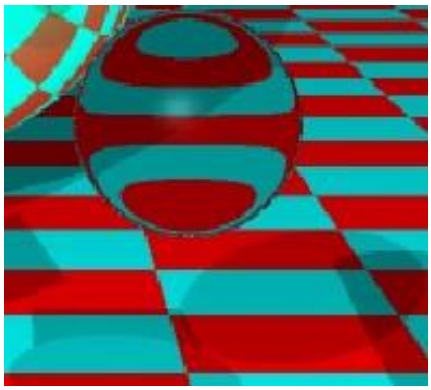
3.4 Multiple Light Sources and Shadows

Two light sources have been implemented with 0.6 and 0.4 light intensity respectively. This also affects the intensities of the diffuse and specular, as well as the brightness of the

entire environment. The two light sources will cast a shadow each in different directions on objects that fulfill the constraints of the ray shadows. Each object satisfying the ray shadow constraints will have two shadows being casted.

3.5 Transparency

One special thing to observe is that for the smaller sphere (non-textured) to the righthand side of the large reflective sphere, the shadows are noticeably lighter than the others.

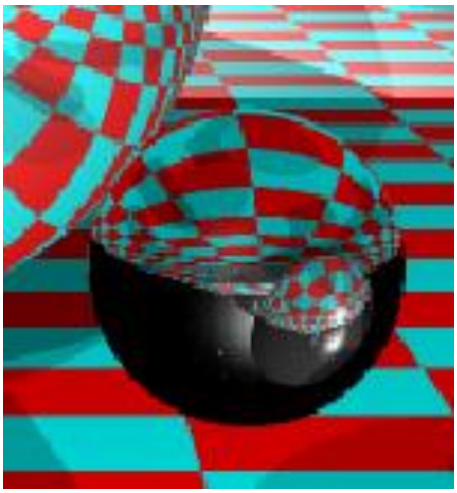


This is due to the sphere being 80% transparent. The sphere's colorSum was multiplied by its transparency level (80% means 0.2 transparency), before adding the refracted colour to the colorSum:

colorSum = colorSum * transparency + refractionColor * (1 - transparency);

3.6 Refraction of Light

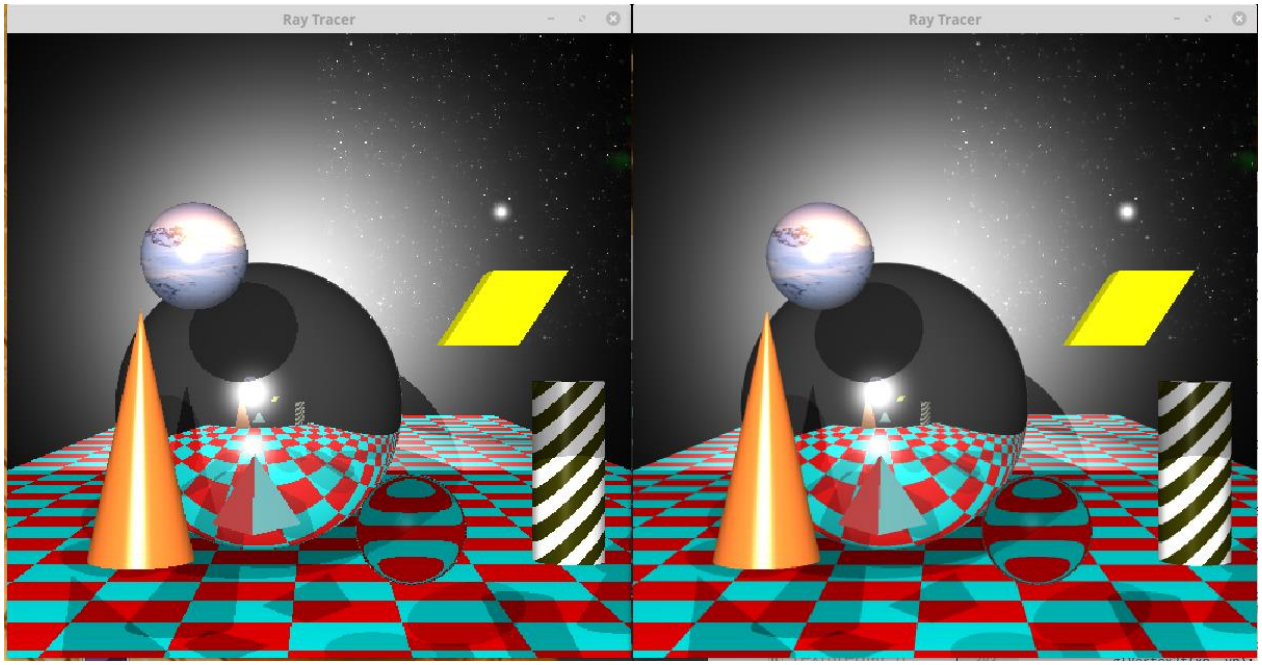
Likewise, the sphere in 3.5 is also refracting light passing through it. I have used ETA = 1.01 to obtain the above refraction in 3.5, while the picture below shows the sphere having an ETA value of 1.5.



3.7 Anti-aliasing Comparison

Super Sampling, a type of Spatial Anti-aliasing, was implemented to reduce the 'jaggedness' of the objects rendered, all in all to produce a much clearer, higher quality rendering of the objects. The 'jaggedness' is caused by finite set of rays that have been generated through the environment, or the output device simply does not have quality

display settings to represent graphics in a smoother way. While it may not be obvious in the screenshot below, due to resolution limitations, the window on the left (without anti-aliasing) has significantly more ‘jaggedness’ on its objects than the window on the right (with anti-aliasing activated).



3.8 Image-textured Object

The small floating sphere at the top left of the large reflective sphere has been textured with the Antarctica image file as sourced from Skybox.zip, Lab 4 – Texture Mapping, using the following calculations to wrap the image around the entire sphere.

```
glm::vec3 center(-7, 5.0, -70.0);
glm::vec3 d = glm::normalize(ray.xpt-center);
float u = (0.5 - atan2(d.z,d.x) + M_PI) /(2 * M_PI);
float v = 0.5 + asin(d.y) / M_PI;
materialCol = textureSphere.getColorAt(u, v);
```

3.9 Object Transformed via Shearing

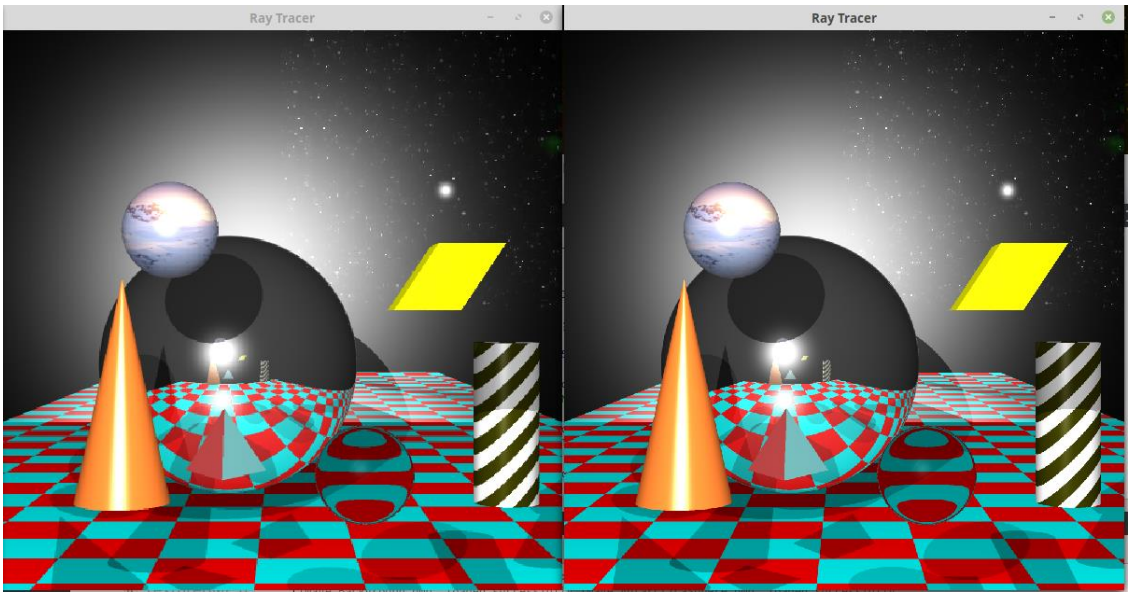
Here, I have chosen to shear the cube that I have implemented. The equations used here are:

```
void drawCube(float x, float y, float z, float length, float width, float height, glm::vec3 color, int shear)
{
    glm::vec3 A = glm::vec3(x, y, z);
    glm::vec3 B = glm::vec3(x+length, y, z);
    glm::vec3 C = glm::vec3(x+length+shear, y+height, z);
    glm::vec3 D = glm::vec3(x+shear, y+height, z);
    glm::vec3 E = glm::vec3(x+length, y, z-width);
    glm::vec3 F = glm::vec3(x+length+shear, y+height, z-width);
    glm::vec3 G = glm::vec3(x+shear, y+height, z-width);
    glm::vec3 H = glm::vec3(x, y, z-width);
```

This gave rise to the cube being sheared to the right, as seen above.

4 Run Time vs Quality

Run time and quality trade-offs can be achieved by changing the value of NUMDIV in RayTracer.cpp. Increasing it will improve the quality of the graphics rendered but consume exponentially more resources (hence loading time), while decreasing it will significantly reduce the quality but finish loading the graphics in a shorter time. The picture on the left is of lower quality but took only 15 seconds, this was rendered using a NUMDIV value of 500. The one on the right took approximately 45 seconds to render but is of higher quality as it was achieved with a NUMDIV value of 1000.



5 References

University of Canterbury COSC363 Computer Graphics Lecture and Lab materials

https://en.wikipedia.org/wiki/Anti-aliasing

https://en.wikipedia.org/wiki/Supersampling

https://en.wikipedia.org/wiki/UV_mapping

https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html#SECTION00023200000000000000