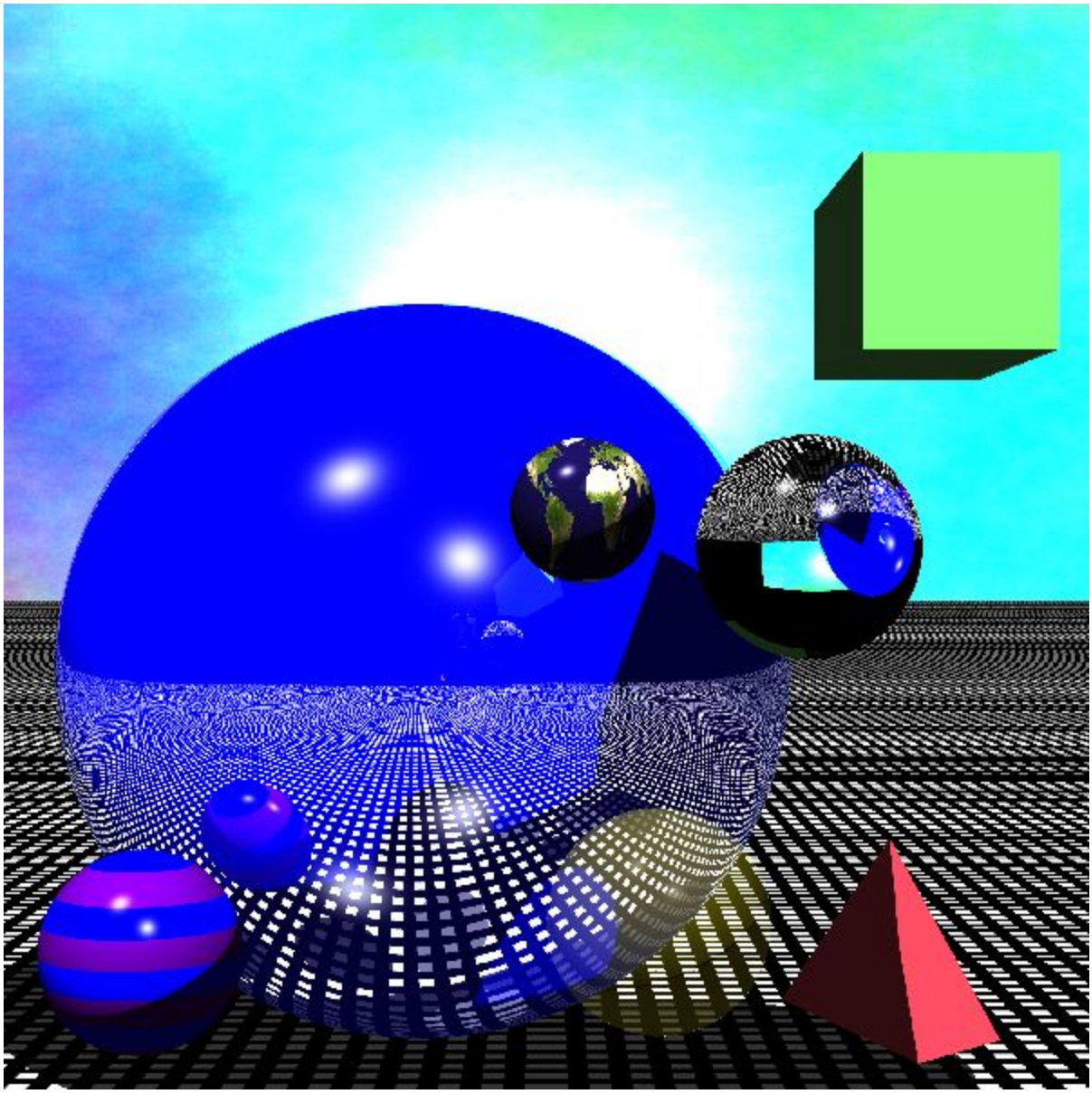
Ray Tracing

Kusal Ekanayake - 97953573 - kre39



Build command:

```
g++ -Wall -o "%e" Assignment.cpp Ray.cpp SceneObject.cpp Sphere.cpp TextureBMP.cpp  Plane.cpp -lm -lGL -lGLU -lglut
```

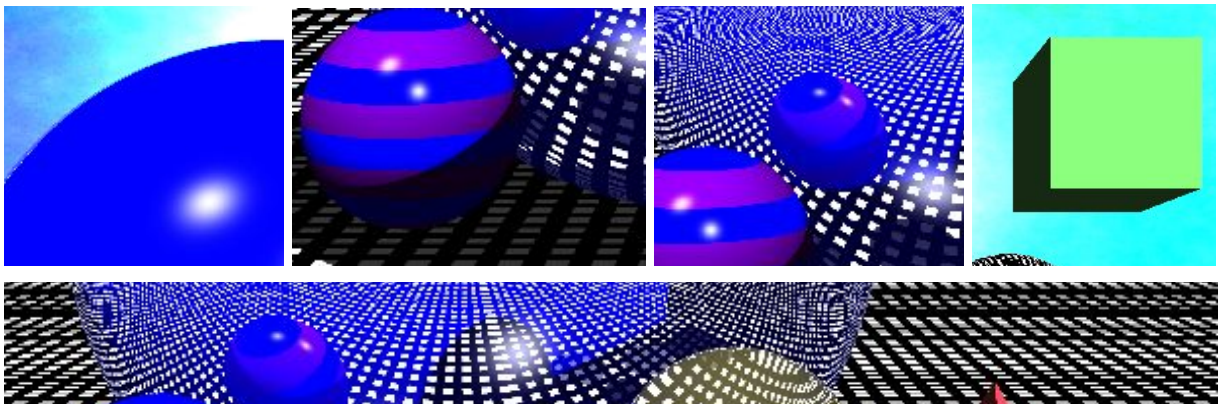
Challenges

This ray tracer built off the work completed within labs 7 & 8. While creating the ray tracer, many challenges were encountered. Many of them were overcome however some had to be worked around. When attempting to create new shapes through separate models, I had difficulty visualising the rays necessary for implementing them. However when creating the tetrahedron, after originally implementing in the way covered further on in the report, I managed nearly implement a technique where I used the plane class to create a 3 vertex plane which would be able to be used as each side of a tetrahedron. When trying to implement this correctly, I continued to run into issues which results in me switching back to the originally implemented method of implementing the tetrahedron.

Features

Basic features

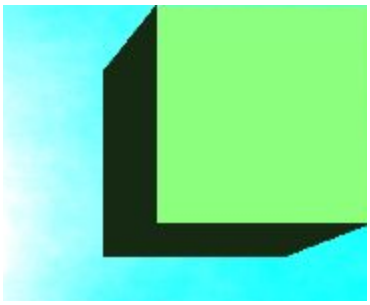
The ray tracer has implemented all of the basic ray tracer features:



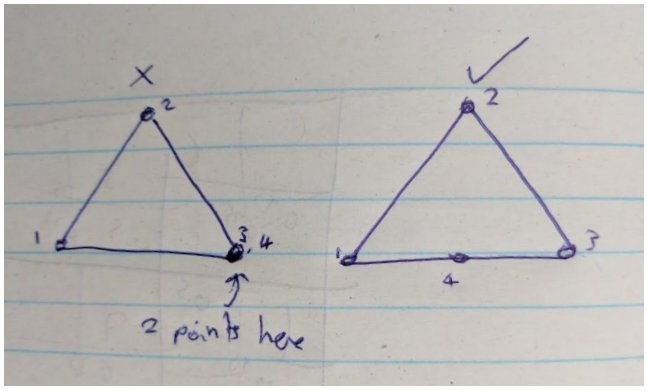
(Top left) - Light source
(Top middle left) - Shadow from light source
(Top middle right) - Reflection off sphere
(Top right) - Box made of 6 separate planes
(Bottom) - patterned plane

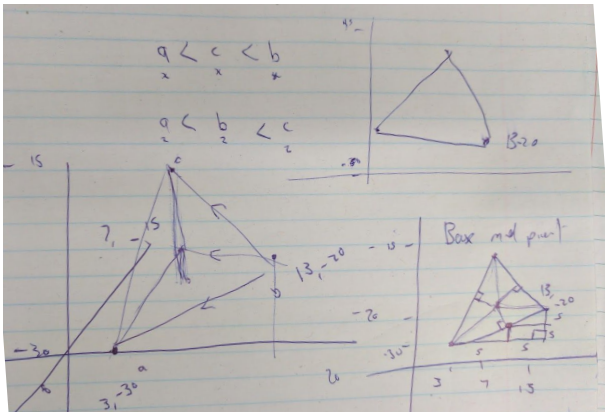
Throughout the course of implementing these features, no major issues arose. The most challenging aspect was making the box out of six planes and making each plane line up with the other planes accurately. Once the box had all of the 6 vertices defined, it was as simple as lining them up with the other vertices correctly when making the planes. After that moving the box was as simple as adding or subtracting the same values to the x, z or y coordinates. There is also a texture colour splash included on the backplane to add brightness to the scene.

Extension Features

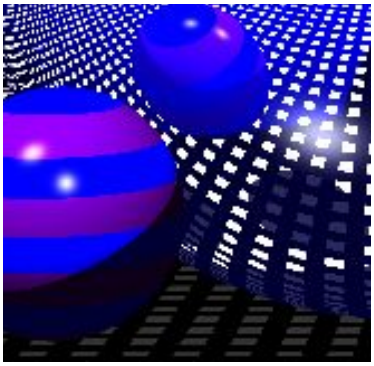


Primitives other than a plane, sphere or box - Tetrahedron.
When looking into implementing a tetrahedron as a primitive shape, originally I attempted to just program the tetrahedron to have two of it's points on each square plane be assigned the same coordinate, however this did not work and instead ended up with the method of setting the 4th point of the quad to placed in the middle of 2 of the other points in the quad. This is shown in the following picture. The original attempt did not require many calculations as it meant I just needed to choose 4 points near each other so I could connect them and created a tetrahedron. When creating the tetrahedron with the second method which did work, I was able to start by choosing 3 points to for a triangle to form the base (have the same y). From there I would be able to calculate one of the mid points between 2 of the points and using trig was able to calculate the midpoint. Using the base to determine the mid peak of the tetrahedron and then more trigonometry to determine a mid point between the peak and the base. This provided me the minimal amount of points (6 points) to construct a tetrahedron. Moving the tetrahedron after constructed was as simple as moving the box shape. Note this tetrahedron is not a regular tetrahedron (this is an irregular tetrahedron).



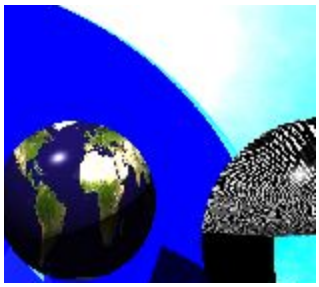


Multiple light sources, including multiple shadows generated by them.
This feature was relatively straightforward to implement but difficult to make work without any errors. It required that I created another light source within the trace method but then duplicate and replace all the existing lines which the original light source effected but replacing the first light source with the second light source. Within the ray tracer, you can see the two light sources been reflected off all the spherical objects and the shadows in the reflection of the large sphere. The two shadows shown are shadows of the same striped pattern sphere.



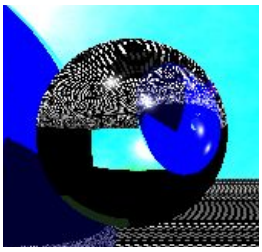
Transparent object.

The transparent object was implemented using the information from the provided lecture notes. Using a large eta of nearly 1, means that the refraction within the sphere is nearly completely transparent. The reason for not using exactly 1 was that it cause segmentation faults which were potentially due to having a sphere that was not able to be properly distinguished from its surrounding. It's worth noting that the shadow of the the transparent sphere has been decreased in intensity to take in account the fact that a lot of the light (but not all) will most likely pass through the sphere. I also had to ensure no seg faults occurred when finding the closest object to the spheres refracted ray.



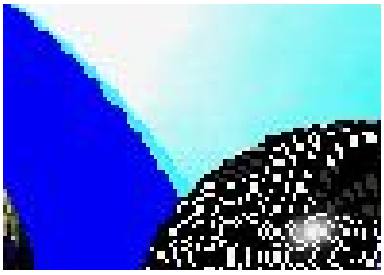
Refractions.

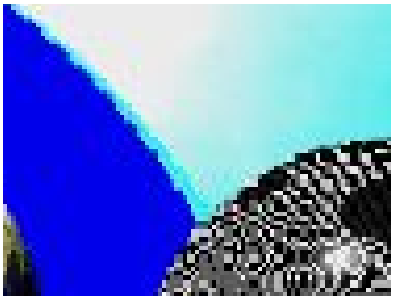
The refracting object within the scene is the glass like sphere pictured here. By using a similar method to the transparent sphere, obtaining a refracting sphere meant I just had to decrease the eta from 1 to 1/1.5. This gave the sphere a solid glass like appearance.



Antialiasing.

In the following images, the top image is the scene rendered with 500 number divisions without anti aliasing and the bottom is 500 divisions with anti aliasing. The difference is fairly obvious within the refraction of the sphere and the clarity of the shadow within it. The checkered floor provides to be a challenge when rendering the small refractions on the cube, however using simple anti aliasing, we can easily make out the defining areas of the sphere. The simple single depth version of anti aliasing was used within this scene which means each ray is subdivided into 4 rays with the average colour calculated for each quarter of the division within the single ray. The colour calculated after wards is equivalent to the average colour over the four rays that split from the original ray.





Not only is this only visible within the refraction, it is also visible within the edge of the large blue sphere having a more even and soft edge. The purpose of anti aliasing is to help more accurately and concisely differentiate all the different squares more evenly and to provide a less jagged outline to the shapes within the scene.

The calculations involved in creating this anti aliasing method was just taking the original ray which was focused in the middle of the division, removing it and replacing it with 4 more rays which were centered in the middle of each quarter of the original division.

Non planar object textured using image.

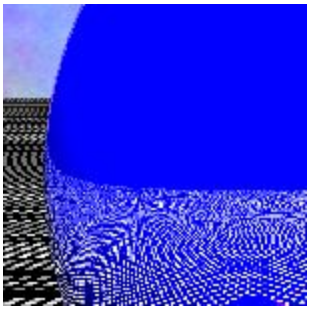
The non planar textured object in this scene is the globe textured sphere located in front of the blue sphere.The formula to calculate the colour of part of the

sphere was calculated by determining the colour at every u (x) and v (y) of the imported texture and mapping that to the same pointed on the sphere. Providing a clear smooth texture sphere. A texture has also been used on the back screen, a generic colour splash to add some brightness into the scene.

```
if (ray.xindex == 1){
    float ucoord = asin(normalVector.x)/M_PI + 0.5;
    float vcoord = asin(normalVector.y)/M_PI + 0.5;
    col = tex1.getColorAt(ucoord, vcoord);
}
```

The code used to calculate this was based off a stackoverflow forum answer from which I altered to work with the scene. This has been referenced in the references,

Non planar object textured using a procedural pattern.



The procedural pattern used with the sphere on the lower left of the scene. I went through by the y coordinates and whenever the y coordinates rounded to an int were divisible by 2, I would colour the level purple, otherwise it would remain the same colour. The same type of method was used to colour the floor in the checkered pattern that it is. This is shown in the code snippet below. For the cube, for the colour to change every interval on y, the ray's y component had to be check. However for the plane which is shown on the bottom half, the initial colour of the plane is black and every time both the x and z component are divisible by 2 when rounded, the square would be painted white, giving the checkered visual.

```
glm::vec3 spherePattern(Ray ray){
    if ((int) (ray.xpt.y) % 2 == 0){
        return glm::vec3(0, 0, 1);
    }else {
        return sceneObjects[ray.xindex]->getColor();
    }
}

glm::vec3 planePattern(Ray ray){
    if ((int) (ray.xpt.z) % 2 == 0 && (int) (ray.xpt.x) % 2 == 0){
        return glm::vec3(1, 1, 1);

    }else {
        return sceneObjects[ray.xindex]->getColor();
    }
}
```

References

https://static.pexels.com/photos/269888/pexels-photo-269888.jpeg - globe picture
https://pixabay.com/p-2219120/?no_redirect - background of scene (pastel colours)
https://stackoverflow.com/questions/22420778/texture-mapping-in-a-ray-tracing-for-sphere-in-c - where equation for texture mapping sphere was derived from