COSC 363 Assignment 2

Harrison Cook – 52210542

Scene Description:

The scene consists of a floor plane, 5 spheres and a cube. There are two light sources illuminating the scene, casting shadows and creating specular reflections. The floor plane and one of the spheres are textured using a procedural pattern. One sphere has reflections, one sphere is just transparent, one sphere has a map of the world textured onto it and the last sphere is somewhat transparent and refracts light. The cube was created out of 6 planes and has been rotated.

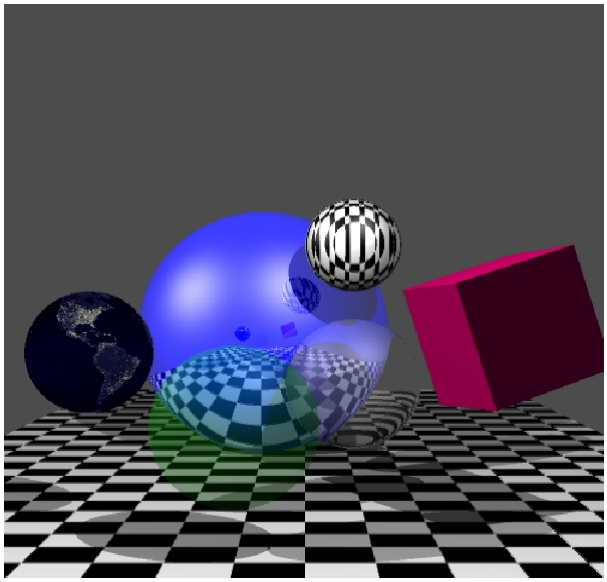

Image 1: The whole scene, seen from the default camera view.

Extra Features:

Multiple Light Sources:

As well as a slightly modified light source from lab 7 & 8 I have created a second light source coming from the negative x direction (left on the screen when viewing the scene) and with the same intensity. New specular reflections and shadows created by the second light source can be seen in Image 1, the new shadows are especially prevalent where the cube shadow intersects with the image-textured sphere's second shadow. In image 2 you see that pixels in shadow from only one light source are lighter than the shadows from the scene with only one light source. This is to be expected and is realistic.

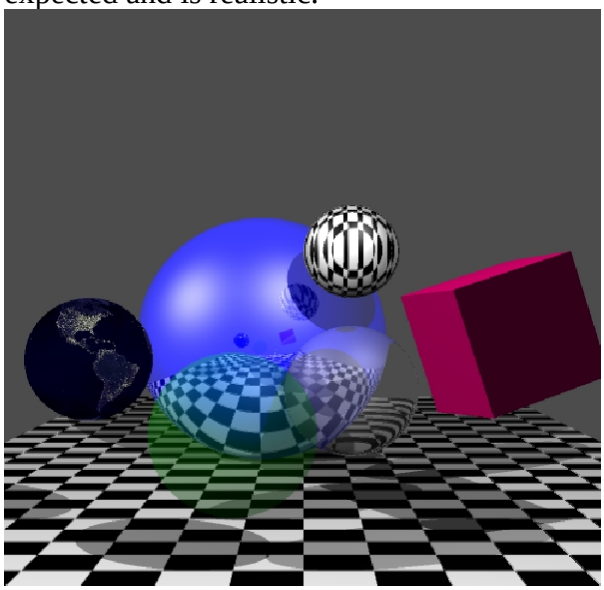

Image 2: Scene with both lights enabled.

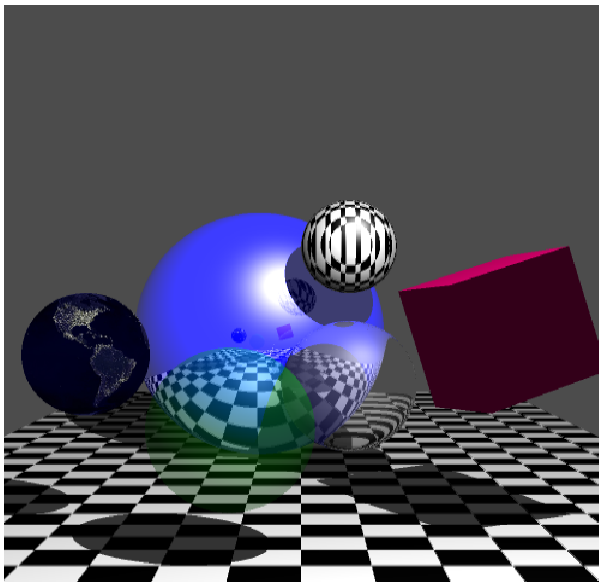

Image 3: Scene with only the original light enabled.

Refraction – Transparent Refracting Sphere:
The sphere in the lower right is transparent and reflective. This is acheived by tracing the rays recursively. In each step of the trace() function, if the current object being traced is the sphere in question, two more rays are traced. The first ray traced is the incident ray, which is then used to trace the second ray, the refracted ray. Transparency is set to 70% (0.3) and is achieved by multipying the current colourSum and the recursively traced refracted colour by the transparency value or inverse transparency value respectively. If either the incident ray or the refracted ray do not intersect with any scene object, the background colour is returned.

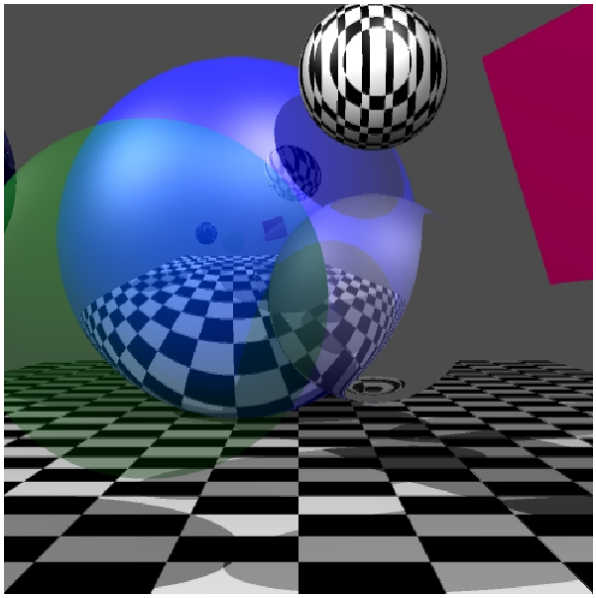
Image 3: eta = 1 / 1.01.

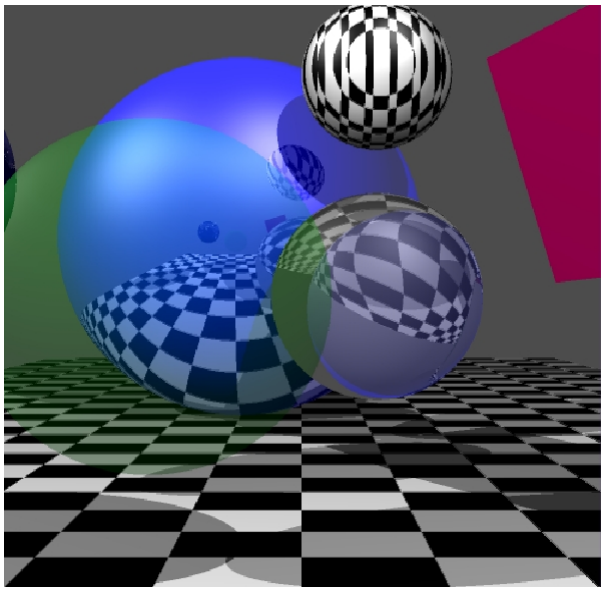
Image 4: eta = 1 / 1.50.

Transparent – Transparent Sphere:
The sphere in the foreground of the scene is transparent. The transparency is done in a very simple, similar way to the refraction. The incident ray is created by using the current ray, then the closest point (where it exits the sphere) is calculated. The outwards ray is calculated using the direction of the incident ray and then finds the closest scene object. The final colour is calculated using the outwards ray closest object's colour and the material colour of the sphere (green). Transparency for this sphere is set to 80% (0.2).

Non-Planar Image-Textured Object – Textured Sphere:
The sphere on the far left is textured by the earth.bmp file. The texturing is done by using the center of the sphere and the incoming ray to find a relative point. This point's x, y and z coordinates are then used to find the corresponding coordinates on the texture image.

```
//--Textured sphere--//
if (ray.xindex == 3) {
    glm::vec3 relativePoint = glm::normalize(ray.xpt - texturedSphereLocation);

    float texY = 0.5 + asin(relativePoint.y) / M_PI; //M_PI is just cmath library's pi
    float texX = 0.5 - atan2(relativePoint.z, relativePoint.x) / (2 * M_PI);
    colourSum = sphereTexture.getColorAt(texX, texY);
}
```

Image 5: The code/calculations used to texture a sphere. See references for source for calculation.

Non-Planar Procedural Patterned Object – Patterned Sphere:
The sphere on the top right is procedurally textured, in a similar way to the floor. I originally had the procedural texture be stripes, but decided to experiment and found I like the chequered look more. The original stripe calculation I used was:

```
(ray.xpt.x + ray.xpt.z) * 3) % 2
```

```
//--Procedurally generated pattern on sphere--//
if (ray.xindex == 1) {
    int xCurrent = (int) ((ray.xpt.x + 200) * 2) % 3;
    int zCurrent = (int) ((ray.xpt.z + 200) * 2) % 3;

    if ((xCurrent && zCurrent) || (!xCurrent && !zCurrent)) { //If both are the same
        sceneObjects[1]->setColor(glm::vec3(1));
    } else { //Else they are different (one 0 and the other 1)
        sceneObjects[1]->setColor(glm::vec3(0));
    }
}
```

Image 6: The code/calculations used to procedurally generate a texture for a sphere.

Anti-aliasing:
To achieve anti-aliasing, I used supersampling, diving each pixel into four segments. Each of the four segments was then traced with a ray to get the colour of each segment. The average colour of the four segments is then used to set the colour of the pixel. As you can see in images 8 & 9, anti-aliasing does make a slight difference in that edges are smoother, especially around the refracting sphere. It does, however, also give a very subtle blur to the whole scene, which is not ideal.

```
//---------Anti Aliasing---------//
glm::vec3 antiAliasing(Ray ray, glm::vec3 eye, float xp, float yp) {
    float pixelSize = (XMAX - XMIN) / NUMDIV;

    Ray firstQuarter = Ray(eye, glm::vec3(xp + pixelSize * 0.25, yp + pixelSize * 0.25, -EDIST));
    Ray secondQuarter = Ray(eye, glm::vec3(xp + pixelSize * 0.25, yp + pixelSize * 0.75, -EDIST));
    Ray thirdQuarter = Ray(eye, glm::vec3(xp + pixelSize * 0.75, yp + pixelSize * 0.25, -EDIST));
    Ray fourthQuarter = Ray(eye, glm::vec3(xp + pixelSize * 0.75, yp + pixelSize * 0.75, -EDIST));

    firstQuarter.normalize();
    secondQuarter.normalize();
    thirdQuarter.normalize();
    fourthQuarter.normalize();

    //Add up all colours and average them
    glm::vec3 average(0.25);
    return (trace(firstQuarter, 1) + trace(secondQuarter, 1) + trace(thirdQuarter, 1) + trace(fourthQuarter, 1)) * average;
}
```

Image 7: The process used to achieve anti aliasing by supersampling.

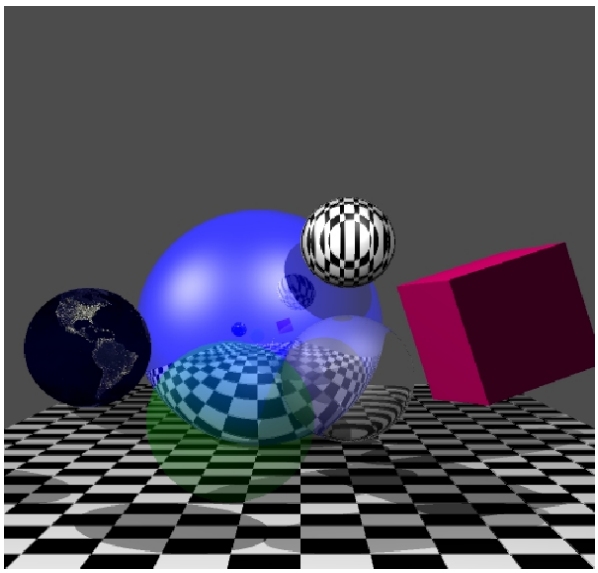

Image 8: Scene with anti-aliasing enabled.

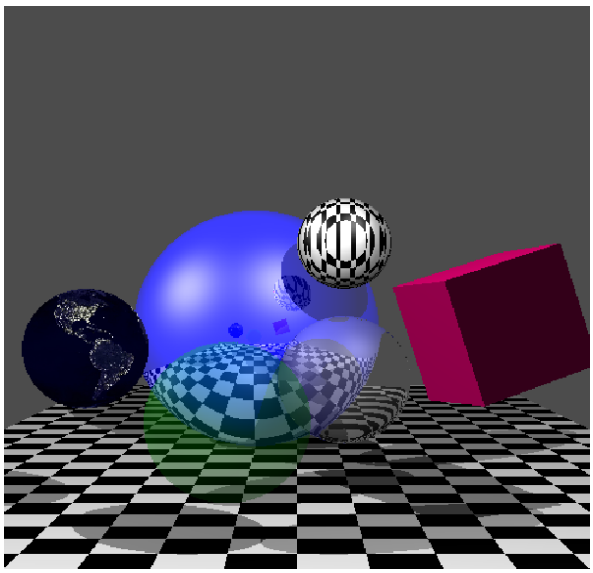

Image 9: Scene with anti-aliasing disabled.

Transformed Object – Rotated Cube:
The cube in the scene has been rotated 25° around the y and z axes. I converted the degrees to radians and then created a glm rotation matrix. I then multplied each point in the cube by the rotation matrix. As I had defined each corner of the cube as a vec3, I had to convert to a vec4 before multiplication and then back to a vec3 to specify a plane coordinate.

```
float angle = glm::radians(25.0f);
glm::mat4 rotation = glm::rotate(glm::mat4(1.0f), angle, glm::normalize(glm::vec3(0, 1, 1)));

glm::vec3 cubeTopA = glm::vec3(-5 + xTransform, -10 + yTransform, -45 + zTransform);
glm::vec3 cubeTopB = glm::vec3(5 + xTransform, -10 + yTransform, -45 + zTransform);
glm::vec3 cubeTopC = glm::vec3(5 + xTransform, -10 + yTransform, -55 + zTransform);
glm::vec3 cubeTopD = glm::vec3(-5 + xTransform, -10 + yTransform, -55 + zTransform);

Plane *topPlane = new Plane(glm::vec3(rotation * glm::vec4(cubeTopA, 1)), //Point A
                            glm::vec3(rotation * glm::vec4(cubeTopB, 1)), //Point B
                            glm::vec3(rotation * glm::vec4(cubeTopC, 1)), //Point C
                            glm::vec3(rotation * glm::vec4(cubeTopD, 1)), //Point D
                            glm::vec3(1, 0, 0.5)); //Colour
```

Image 10: Creating the rotation matrix, defining the top four corners and creating the top plane of the cube.

Other Feature – Camera Movement:
There are two camera views available in my ray tracer, the first default view as seen in Image 1, and the second closer view as seen in Image 11. The second view can be toggled by pressing the ‘c’ key, due to anti-aliasing, the scene usually takes around 10 – 20 seconds to redraw on the lab computers. This second view shows a closer view of the reflecting sphere, the patterned sphere, the transparent sphere and the refractive sphere. The refraction is especially obvious from this view.

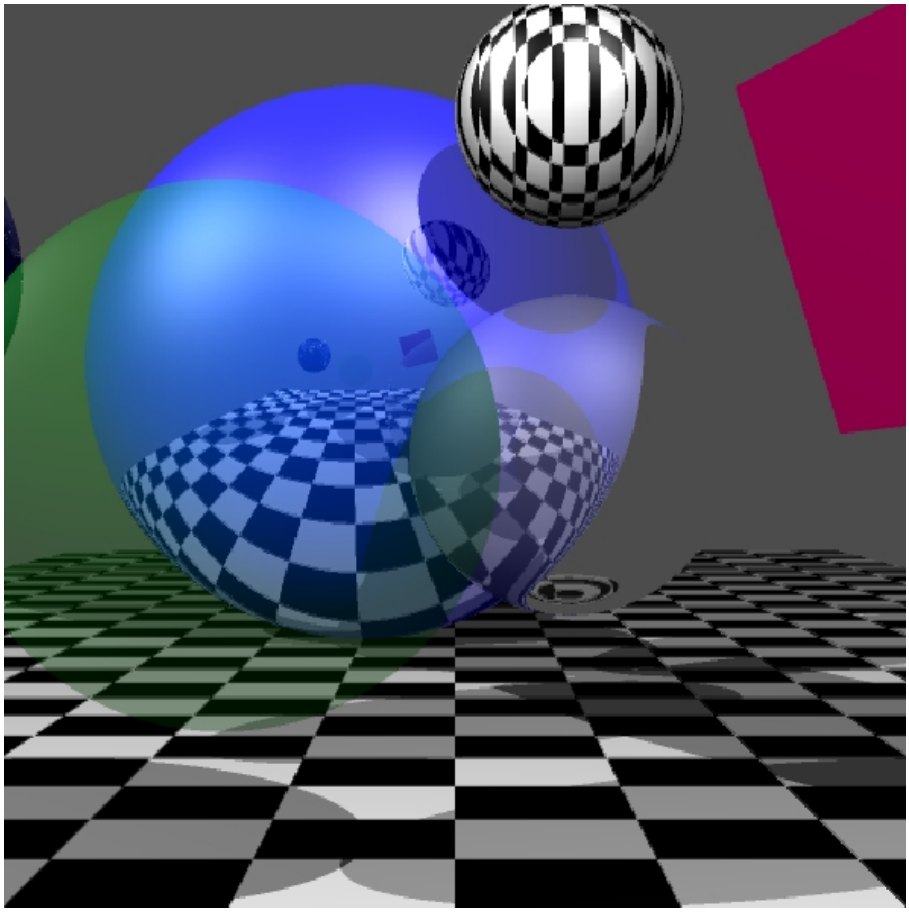

Image 11: The scene from the second camera view. A closeup view of three of the spheres, clearly showing the refraction of the refractive sphere.

Successes and Failures:

Everything I attempted to do I successfully implemented, however I had less trouble with some features than I expected and had more trouble than I expected with others.

All of the texturing, both procedural and image, I thought would give me far more issues than they actually did. The procedural texture generation I never had any issues with at all. The only issue I encountered with the image texturing ended up only taking about 30 minutes to solve, and turned out to be a QtCreator configuration issue with loading images.

Creating the cube (before rotation) was also something I thought would take me longer than it did, but once I figured out how to order the points for a plane, it went quite quickly. Since I put the cube creation into a function I also decided to add the ability to translate the cube using parameters to the function. This was very beneficial since I could move it around the scene much easier when changing my mind about the scene layout.

Rotating the cube, on the otherhand, was possibly the feature that took me the longest. I initially had a lot of issues with the face of the cube I was experimenting with being rotated not around its own center. This was partially due to the way I was creating the planes to begin with and partially to do with me not noticing I was still creating the cube with translation parameters set. Even after noticing this, it still took me a while to get it rotating correctly, and during this time I decided to change my cube generation method to setting the 8 corners of the cube and using those to generate the planes (instead of entering each point of each plane). I was happy with the result in the end, especially since my translation parameters work without distorting the rotation, making it easy for me to rearrange the scene when I want to.

Build Process:

1. Open QtCreator
2. Open the CMakeLists.txt file in the ‘src’ directory as a project
3. Configure the project if necessary
4. To get images to work:
    a) Go to the projects tab in QtCreator
    b) Click on ‘Run’ under ‘Build & Run’ on the left
    c) Browse to change the working directory to the ‘src’ folder
5. Run the project

It usually takes around 20 seconds to fully display the scene.

References:

Earth texture: Taken free from pexels.com - https://www.pexels.com/photo/black-textile-41949/

Image texture mapping adapted from: http://bentonian.com/teaching/AdvGraph1314/3.%20Ray%20tracing%20-%20color%20and%20texture.pdf