# COSC 363 - Assignment 2

Justin Thorby – jth102

**Introduction**

For this assignment I have implemented the ray tracer that is shown below. The scene consists of a floor and background, box, tetrahedron and collection of spheres with different properties.

The extra features I have done are:

- A tetrahedron
- Transparent object
- Refractions
- Anti-aliasing
- Non-planar textured object using an image (rainbow sphere)
- Non-planar textured object using a procedural pattern (striped sphere)

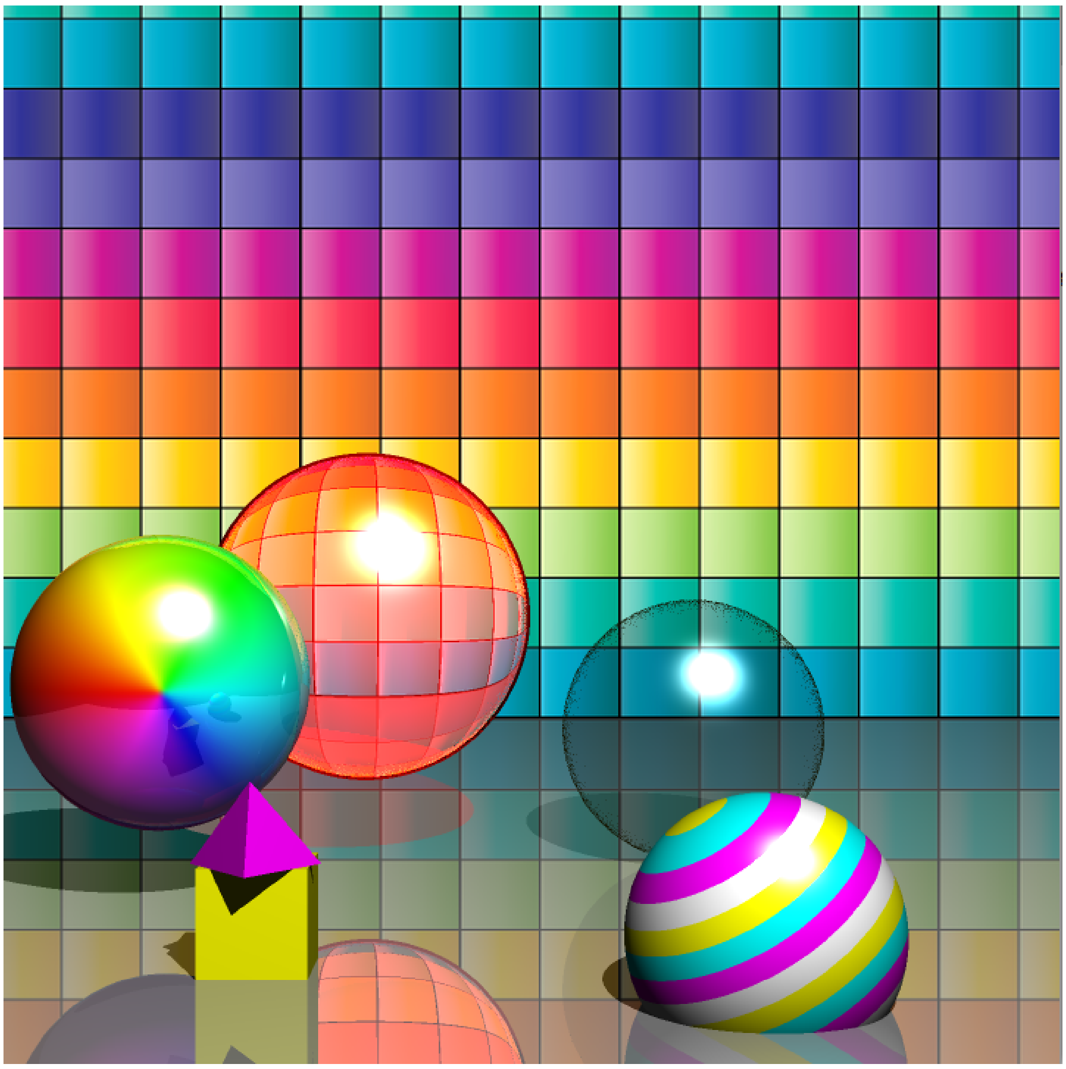


***Figure A** – The full scene*

## 1. The Basic Ray Tracer

### a) Lighting

The ray tracer has a single light source, placed at position (20, 60, -3).
Diffuse reflections have been implemented using Phong’s illumination model. The effect of this is clear in the figure below with the distinction in colour between the two visible planes of the pink tetrahedron.
Specular reflections have also been included and are clearly visible on the 2 spheres also in the figure below. These size of these reflections have been calculated using a Phong’s constant value of 50. The reflections also correctly face the position of the light source as it sits above and to the right of the position of the spheres.

### b) Shadows

Shadows have been produced where appropriate in the scene. This has been done by generating a new ray from the original ray collision point toward the light source and finding whether the new ray (shadow ray) collides with another object before passing the light source. For “standard” (non-transparent) objects the shadow colour is calculated by simply adding the ambient term to the colour of what is in shadow.

### c) Reflections

The scene features two reflective surfaces; the sphere on the left in the figure below and also the floor. Reflections are found by recursively generating new rays (to a limit of 3 steps) in the direction given by glm::reflect(), passing in the previous ray’s direction and the normal vector. We then add the colour values of these rays (with a decreased intensity) together to give a final result.

### d) Box

The scene has a yellow box, as shown in the figure below. The box consists of 6 planes of equal size arranged correctly to create a cube.

### e) Planar textured / patterned surface

The back wall of the scene has been textured using an image where s and t were found using the following functions:

```
s = (ray.xpt.x - backGroundX) / ((backGroundX + backGroundWidth) – backGroundX);
t = (ray.xpt.y - backGroundY) / ((backGroundY + backGroundHeight) - backGroundY);
```

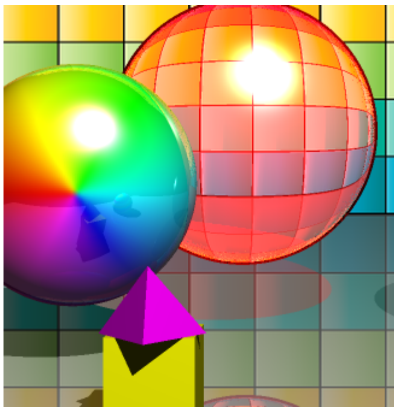
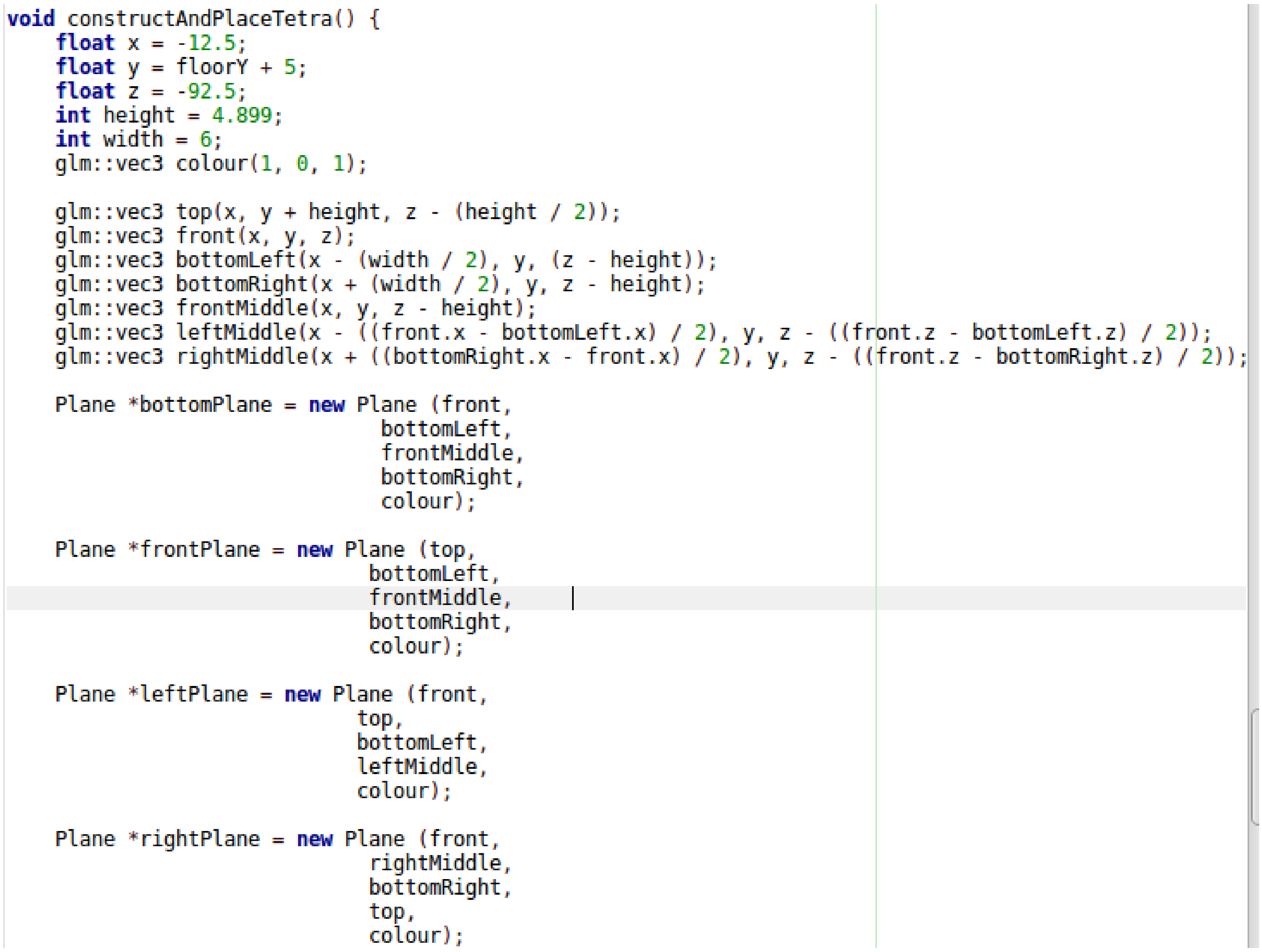


**Figure B** The basics

## 2. Extensions

### a) Tetrahedron

The figure above shows the tetrahedron I have created. It has been constructed by using 4 planes.



```
void constructAndPlaceTetra() {
    float x = -12.5;
    float y = floorY + 5;
    float z = -92.5;
    int height = 4.899;
    int width = 6;
    glm::vec3 colour(1, 0, 1);

    glm::vec3 top(x, y + height, z - (height / 2));
    glm::vec3 front(x, y, z);
    glm::vec3 bottomLeft(x - (width / 2), y, (z - height));
    glm::vec3 bottomRight(x + (width / 2), y, z - height);
    glm::vec3 frontMiddle(x, y, z - height);
    glm::vec3 leftMiddle(x - ((front.x - bottomLeft.x) / 2), y, z - ((front.z - bottomLeft.z) / 2));
    glm::vec3 rightMiddle(x + ((bottomRight.x - front.x) / 2), y, z - ((front.z - bottomRight.z) / 2));

    Plane *bottomPlane = new Plane (front,
                           bottomLeft,
                           frontMiddle,
                           bottomRight,
                           colour);

    Plane *frontPlane = new Plane (top,
                          bottomLeft,
                          frontMiddle,
                          bottomRight,
                          colour);

    Plane *leftPlane = new Plane (front,
                         top,
                         bottomLeft,
                         leftMiddle,
                         colour);

    Plane *rightPlane = new Plane (front,
                          rightMiddle,
                          bottomRight,
                          top,
                          colour);
```

This function (above) shows how the vertex coordinates for each of the four planes of the tetrahedron have been defined. Because a plane requires 4 vertices, each triangle has a point at each of its 3 corners and also another point along one of its edges.

**b, c) Transparent object, refractions**

As shown in the figure below, transparency and refractions have been implemented in 2 of the spheres. The pattern on the wall behind the spheres highlights the effect of the refracting sphere. This has been done by the following process:

- ➔ If the surface is transparent/translucent then generate a secondary ray R1 along the direction of refraction.
  This direction is only computed if the surface is refracting, otherwise the direction remains the same as the previous ray.
  The direction is computed internally by the glm::refract() function which implements the following formulae (where n1, the refraction index, = 1.02):

$$\boldsymbol{g} = \left(\frac{\eta_1}{\eta_2}\right)\boldsymbol{d} - \left(\frac{\eta_1}{\eta_2}(\boldsymbol{d}.\boldsymbol{n}) + \cos\theta_t\right)\boldsymbol{n}$$

$$\cos\theta_t = \sqrt{\left(1 - \left(\frac{\eta_1}{\eta_2}\right)^2\left(1 - (\boldsymbol{d}.\boldsymbol{n})^2\right)\right)}$$

- ➔ If the refracted ray collides with the sphere again, then generate another new ray R2 as in the previous step. Again, if the sphere has no refractive property, the new ray direction remains the same.
- ➔ Recursively call trace() to find the colour value of what ray R2 collides with
- ➔ Find the colour value of the object the original ray collided with. Add a constant value (I used 0.8) to this colour so that it is not so bright.
- ➔ Multiply these colours together to give a result for the refraction / transparency phase of trace().
- ➔ Now the final colour should take on a mixture of the object itself as well as what can be seen through the object.

I have also considered the colour shadows should take when the object producing the shadow is transparent / translucent. If the object is transparent / translucent, I lighten the shadow by multiplying the ambient term by 4. I also colour the shadow by multiplying ¼ of the object producing the shadow's colour by the ambient term. Then the resulting ambient term is added to the overall colour of the pixel. These lighter / coloured shadows can be seen clearly in the figure below.

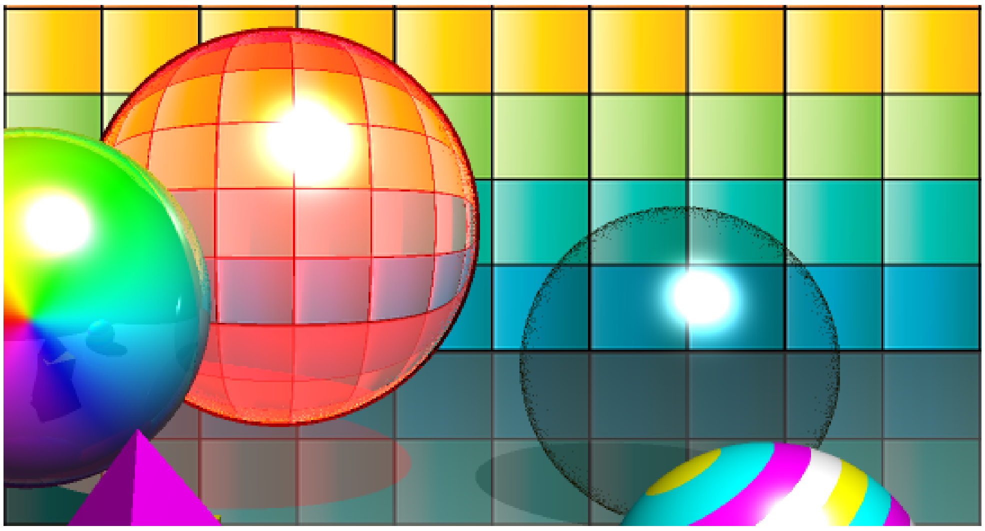


**Figure C** Refraction and transparency

**d) Anti-aliasing**

The intent of anti-aliasing is to remove the distortion artefacts produced by sampling a light field using only a finite set of rays. The significance of this distortion can be reduced by simply using more rays, however will still generally result in some jaggedness. To improve upon this, we can implement an anti-aliasing method such as supersampling, as I have done.
While anti-aliasing will generally give a more pleasant look overall, it does result in more blurred edges. This effect particularly obvious by looking at the tetrahedron in the figure below.

Anti-aliasing has been implemented using supersampling. I used a sampling rate of 4x. Therefore, for each pixel a ray was generated through the center of each of 4 sub-pixels. The colour values of each of these rays were then computed, added together and finally averaged.

This method quite effectively reduces the jaggedness of the edges of both shapes and shadows as shown in the figure below. Note the significant improvement in how smooth the edges of the sphere appear on the left hand side which has supersampling enabled.

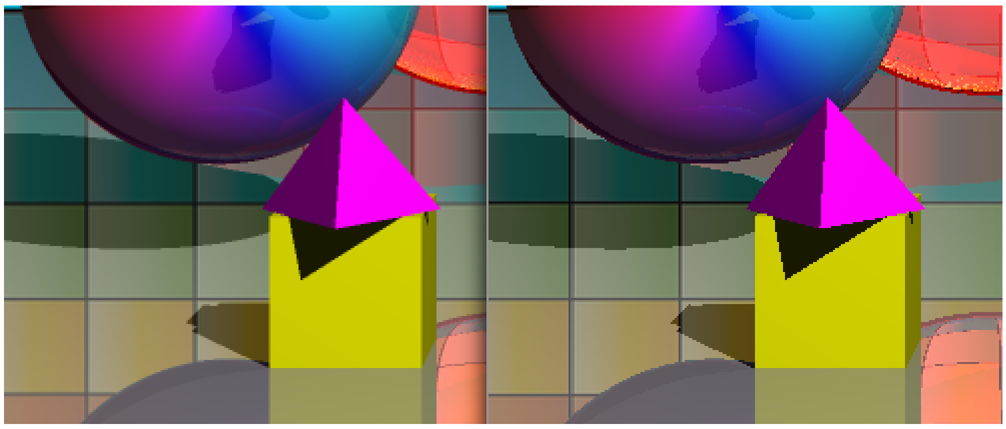


**Figure C – Left:** Supersampling, **Right:** No Supersampling

**e) Non planar textured object using an image**

To texture the sphere shown in the figure below with an image, I implemented the following functions:

Texture s coordinate = asin(normalVectorX) / PI + 0.5
Texture t coordinate = asin(normalVectorY) / PI + 0.5

Then, I set the pixel colour to the colour of the pixel at the given s, t coordinates of the image using the getColorAt(s, t) function

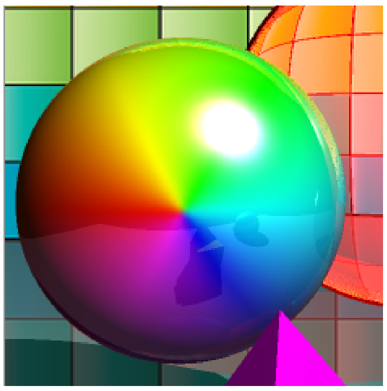


**Figure D** Textured sphere (image)

**f) Non-planar textured object using a procedural pattern**

To procedurally texture the sphere shown in the figure below I used the following function:

$$r = sqrt(x^2 + y^2)$$

I then cast r to an int, modulated the result by 4 and chose a colour for each of the 4 possible values r could take.

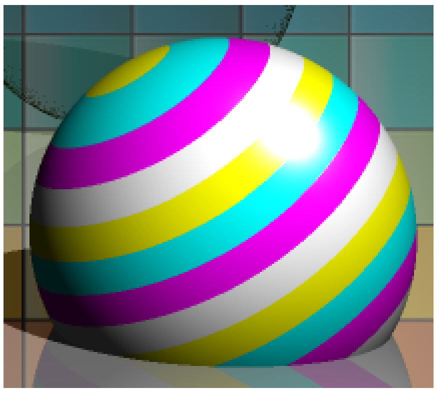


**Figure E** Textured sphere (procedural)

**References**

**Back wall texture: http://maxpixel.freegreatpicture.com/Grid-Texture-Rainbow-Pattern-Squares-Checkered-1758380**

**Sphere texture: https://pixabay.com/p-1911494/?no_redirect**

**procedural texturing : http://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/texture.pdf**

**spherical texturing https://www.mvps.org/directx/articles/spheremap.htm**