Project Report - 2

Ray Tracer Objects & Illuminations

Name: Haider Saeed

ID: 68479094 – (msa280)

Date: 26/05/2023

Email: **msa280@uclive.ac.nz**

Plagiarism Declaration

Name : **M. Haider Saeed (msa280)**

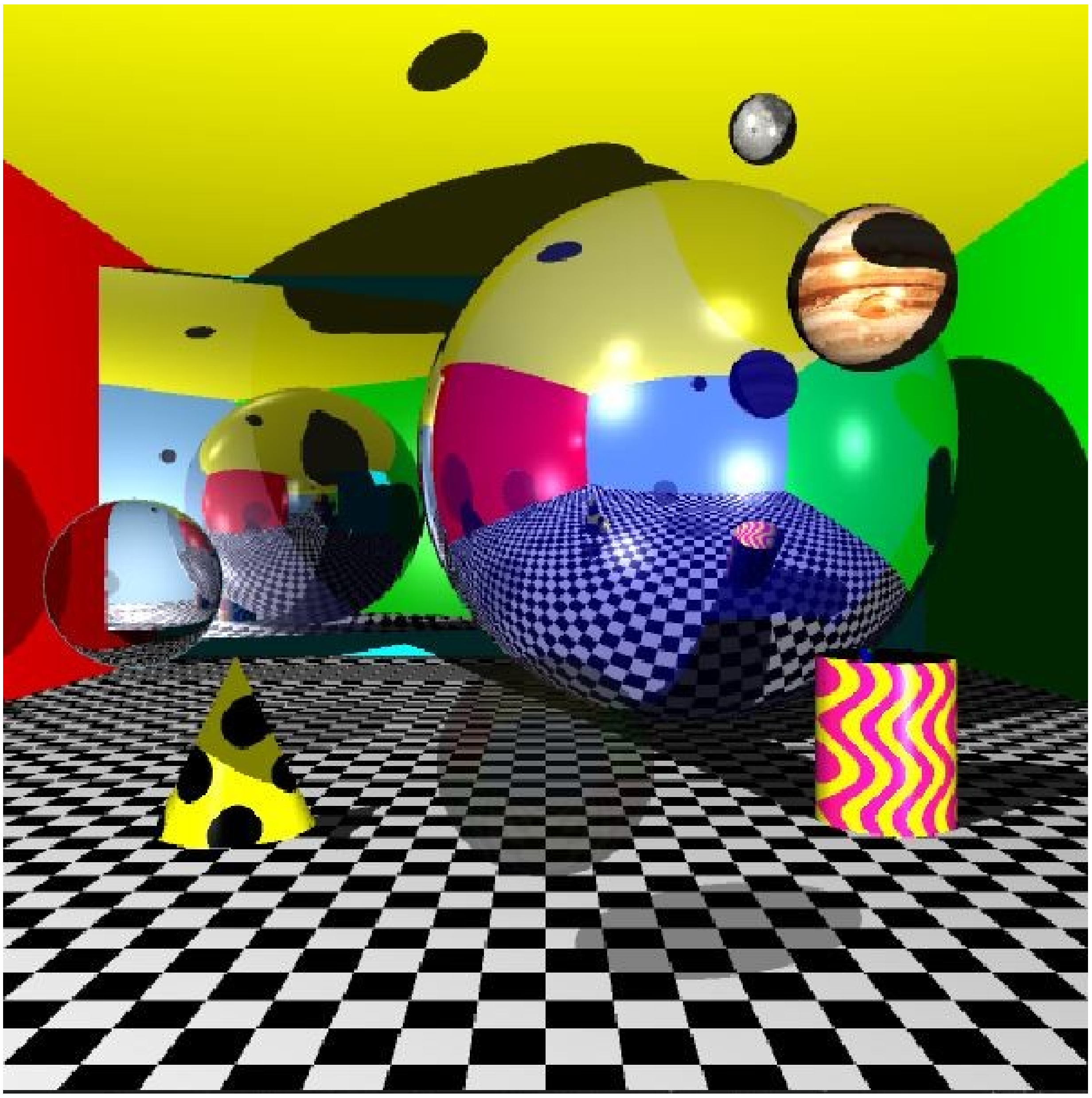
Student ID: **68479094**

Date: **28/03/23**

[illegible]

<u>Contents</u>

Project Success & Ray Tracer Capabilities



Scene 1 - Ray Tracer Output

The results of the ray tracer's implementation are displayed in Scene 1. The scene effectively depicts ray physics, including reflection, transparency, and refraction. Inside a room characterised by five axis aligned planes (the "Front Wall," "Left Wall," "Right Wall," "Floor," and "Ceiling"), features such as various shadows, patterns, and scene objects may be observed in a suitable spatial arrangement. Different kinds of geometric objects and characteristics of global lighting can be handled by this ray tracer. As a result, it has shown that it is capable of improving the visual realism of a produced scene.

Features

Transparent Object – Hollow Sphere

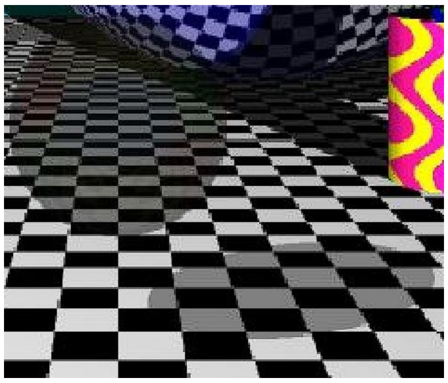


Figure 2 - Hollow Sphere

A hollow sphere can be seen with slightly reflective properties. The reflection of the green, red, and yellow walls can be seen glimmering on the surface of the hollow sphere. The sphere also has a lighter shadow as it is a case of a transparent object.

Reflective Planar Objects – Mirror & Blue Reflective Ball

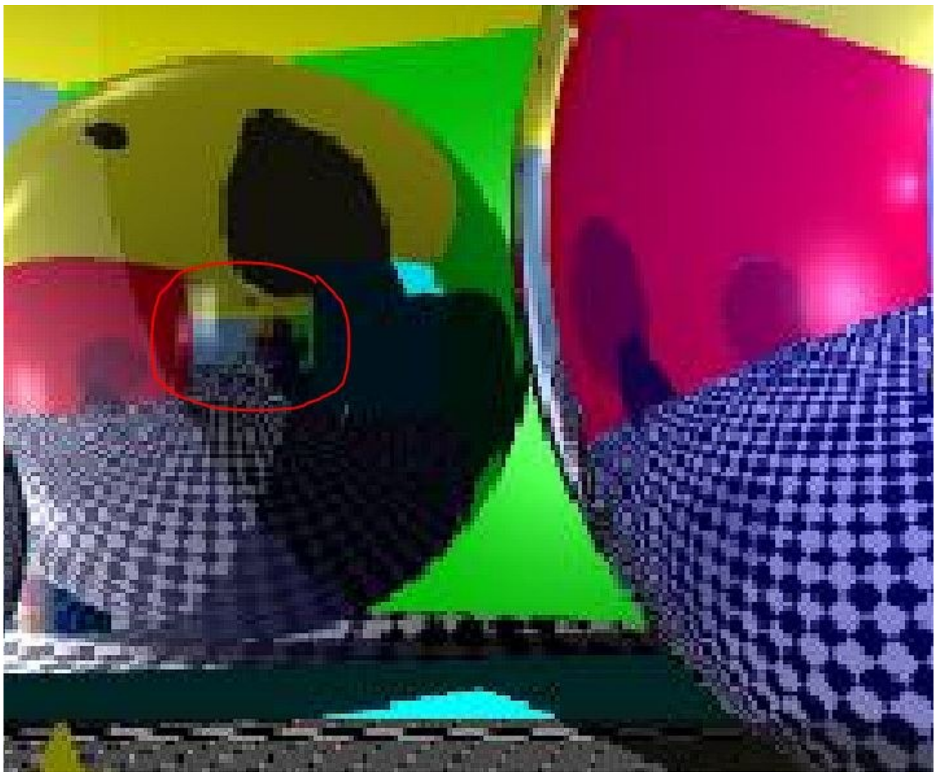


Figure 3 - Mirror and Blue Reflective Ball

Figure 3 shows a planar mirror and a blue reflective sphere with mirror like properties. Both reflective surfaces show mirror-like reflection of their surroundings such as the chequered pattern on the floor. If we observer more closely, we can see multiple reflections being generated inside the mirror. The red circle highlighted shows another mirror with the reflection of another ball inside. This effect is created when the ball's reflective surface is parallel to the mirror's reflective surface generating a recursive pattern of ray tracing. On top of that, the blue colour of the sphere mixes with the reflection colour to produce a hue of blue in the generated reflection.

Cylinder Object

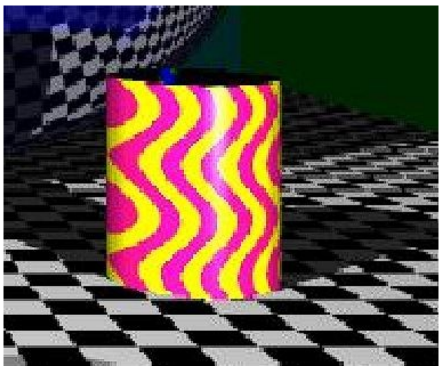


Figure 4 – Cylinder

Figure 4 shows a cylinder with the cosine pattern generated in the y direction. The two shadows from different light sources, pointing in different directions, can be seen on the floor.

Cone Object

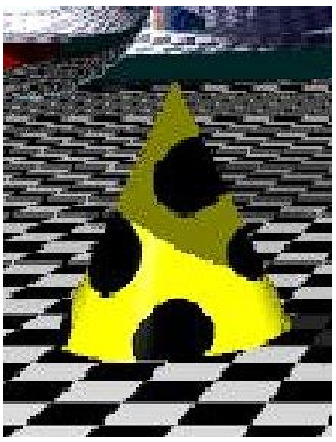


Figure 5 – Cone

Figure 5 shows a cone with large black polka dots. The patterns were generated by adding sin and cos functions. The light shadow of the transparent sphere can also be seen covering the top of the cone.

Refractive Sphere

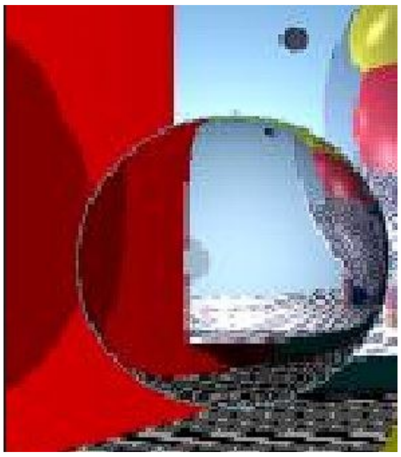


Figure 6 - Refractive Sphere

Figure 6 shows a refractive sphere with the resulting rays being inverted after passing through the sphere. The refraction coefficient is 1.016. The refracted image shows the floor bending on the bottom. The edge of the mirror also changes direction inside the refractive sphere. This sphere is also casting a comparatively lighter shadow onto the wall as it is a case of a refractive surface.

Textured Spheres – Jupiter & Moon

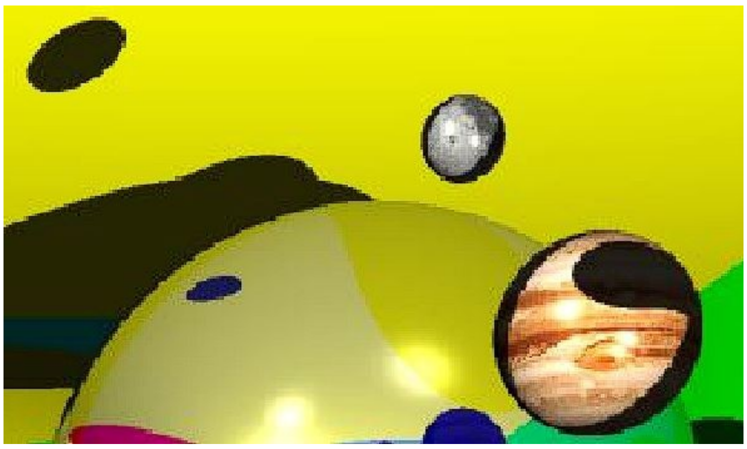


Figure 7 - Jupiter & Moon

Figure 7 shows two texture mapped sphere. One of these spheres is textured to represent the planet Jupiter whereas the other smaller sphere contains the texture of our Moon. The spheres include multiple specular highlights and shadows generated from the two distinct light sources. For example, the shadow of the moon can be seen on both Jupiter and the ceiling. Two white specular spots can be seen on Jupiter and Moon as well.

Basic Anti-Aliasing

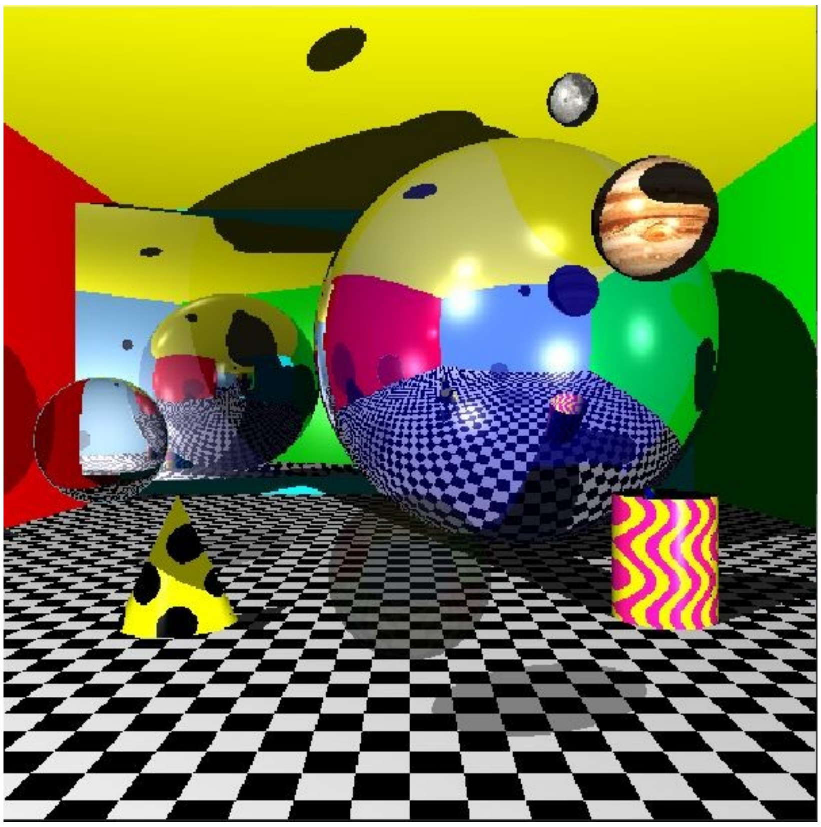


Figure 8 - Anti-aliasing Off

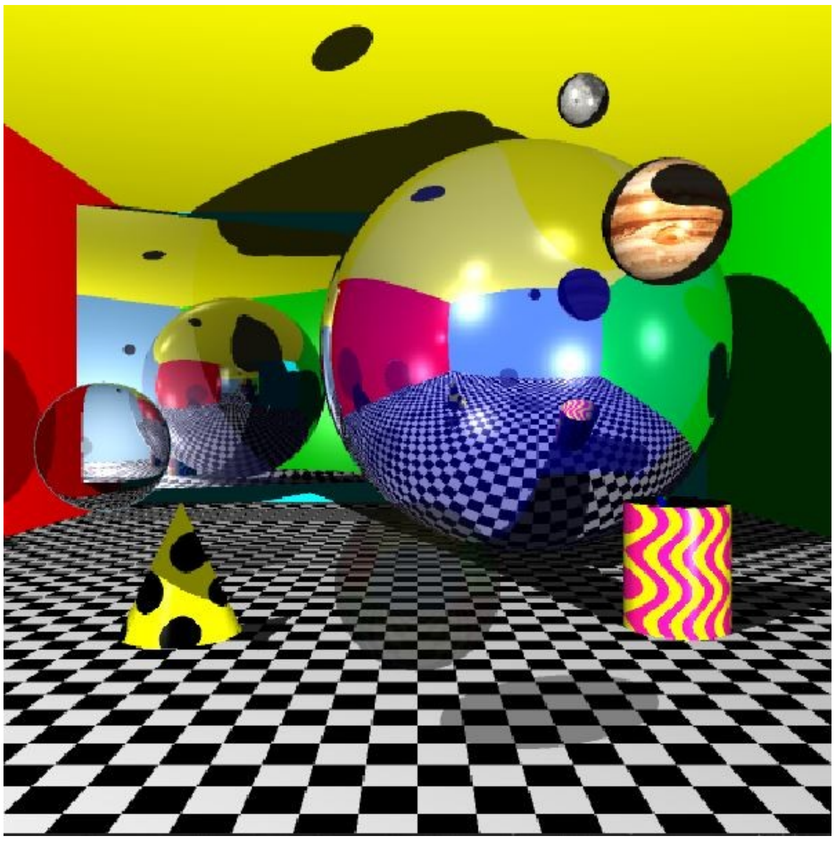


Figure 9 - Anti-aliasing On

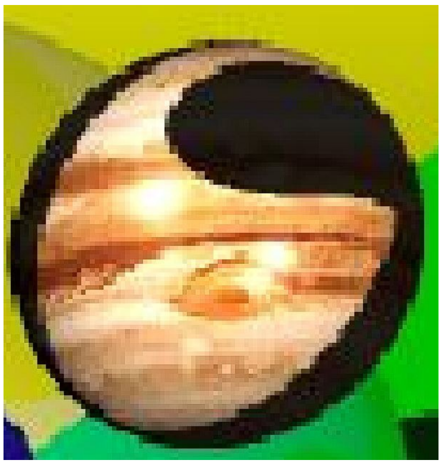


Figure 10 - Jupiter (Anti-aliasing On)

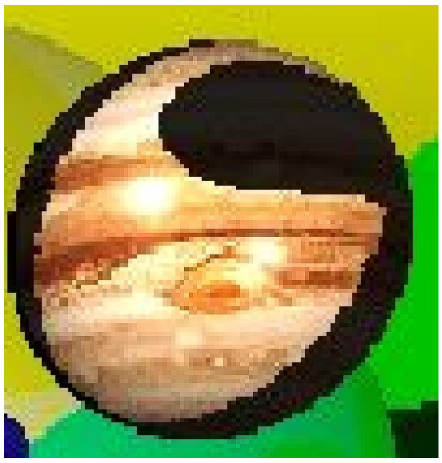
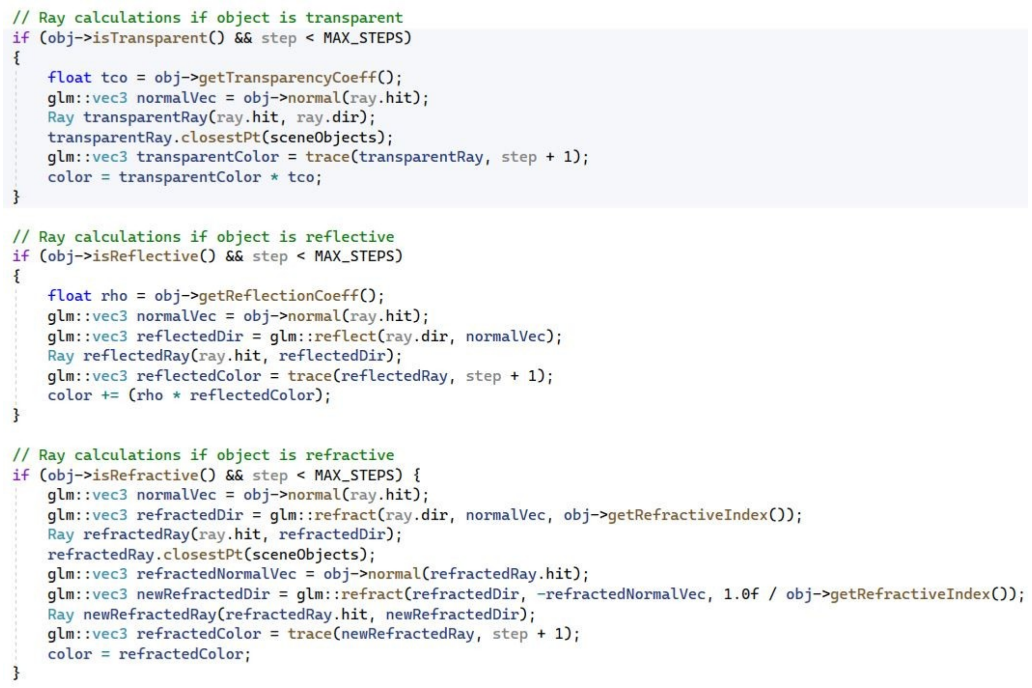


Figure 11 - Jupiter (Anti-aliasing Off)

Figures 8 and 9 show the output of the ray tracer with and without anti-aliasing. In Figure 8, we can see distorted artefacts such as the jaggedness of the objects and shadows. Anti-aliasing helps solve this issue by using super sampling. In super sampling, we generate several rays through each cell and compute the average of the colour values. This results in the object edges changing from that of Figure 10 to the ones in Figure 11. It makes edges appear less blurred and blends colours to enhance the visual realism of the scene.

Calculations & Code Used

Refractions, Reflection & Transparency Handlers

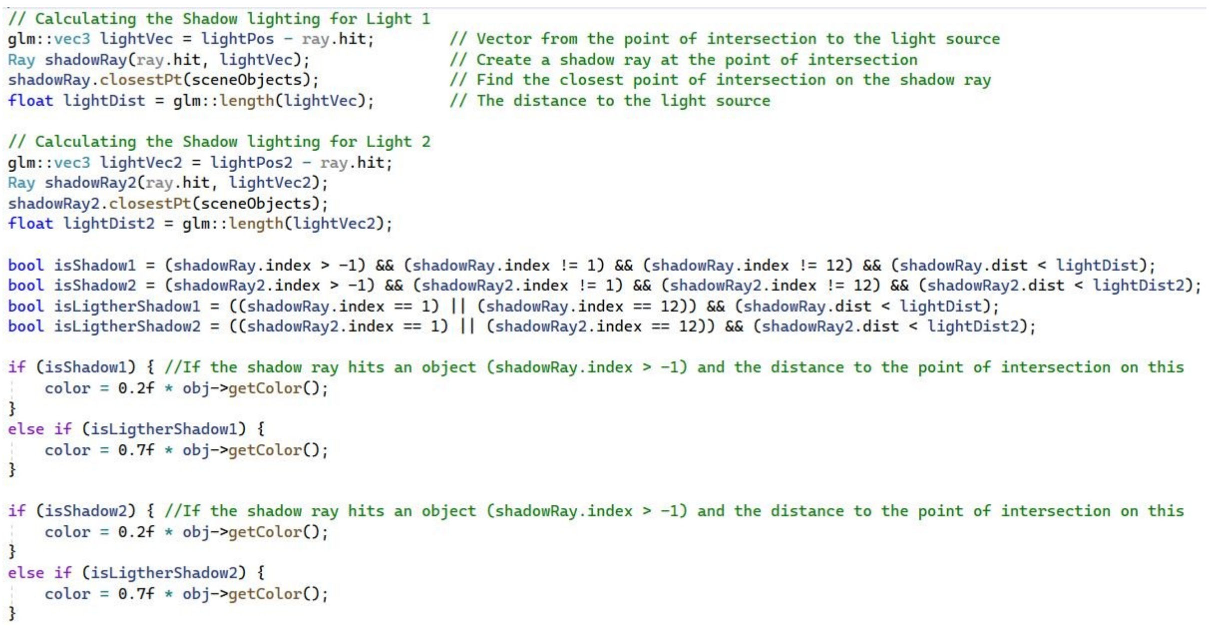


```
// Ray calculations if object is transparent
if (obj->isTransparent() && step < MAX_STEPS)
{
    float tco = obj->getTransparencyCoeff();
    glm::vec3 normalVec = obj->normal(ray.hit);
    Ray transparentRay(ray.hit, ray.dir);
    transparentRay.closestPt(sceneObjects);
    glm::vec3 transparentColor = trace(transparentRay, step + 1);
    color = transparentColor * tco;
}

// Ray calculations if object is reflective
if (obj->isReflective() && step < MAX_STEPS)
{
    float rho = obj->getReflectionCoeff();
    glm::vec3 normalVec = obj->normal(ray.hit);
    glm::vec3 reflectedDir = glm::reflect(ray.dir, normalVec);
    Ray reflectedRay(ray.hit, reflectedDir);
    glm::vec3 reflectedColor = trace(reflectedRay, step + 1);
    color += (rho * reflectedColor);
}

// Ray calculations if object is refractive
if (obj->isRefractive() && step < MAX_STEPS) {
    glm::vec3 normalVec = obj->normal(ray.hit);
    glm::vec3 refractedDir = glm::refract(ray.dir, normalVec, obj->getRefractiveIndex());
    Ray refractedRay(ray.hit, refractedDir);
    refractedRay.closestPt(sceneObjects);
    glm::vec3 refractedNormalVec = obj->normal(refractedRay.hit);
    glm::vec3 newRefractedDir = glm::refract(refractedDir, -refractedNormalVec, 1.0f / obj->getRefractiveIndex());
    Ray newRefractedRay(refractedRay.hit, newRefractedDir);
    glm::vec3 refractedColor = trace(newRefractedRay, step + 1);
    color = refractedColor;
}
```

Figure 12 – Refraction, Reflection & Transparency Code

Figure 12 shows the code used for colour calculations when an object is either transparent or refractive. The methodology and calculation were adopted from Lecture 8 – Ray Tracing (Page 21-27). When an object is transparent, it means that the light passes through the object without changing direction. However, when an object is refractive, the light rays bend and change direction after passing through the object. Therefore, it requires a secondary ray calculation as shown in the code. The secondary ray is then used to trace and get the refracted colour.

Lighting & Shadows

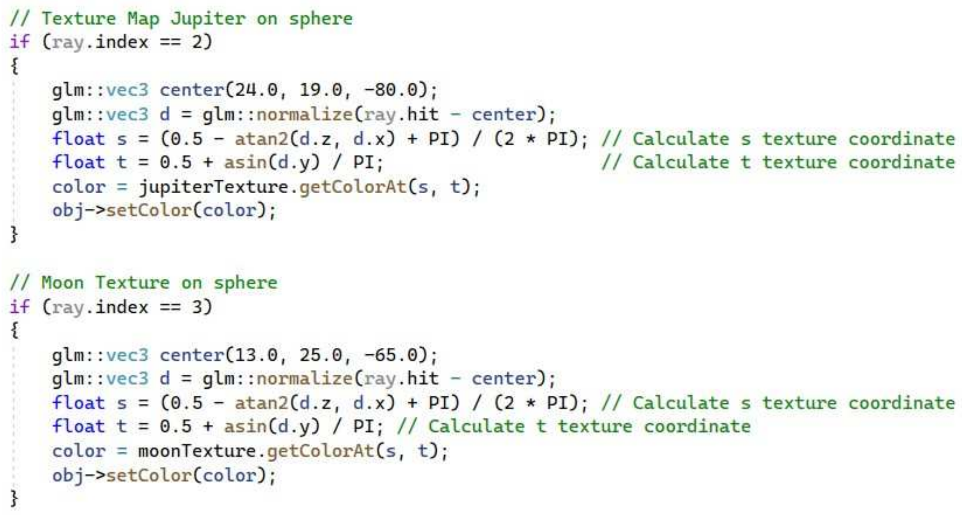


```
// Calculating the Shadow lighting for Light 1
glm::vec3 lightVec = lightPos - ray.hit;        // Vector from the point of intersection to the light source
Ray shadowRay(ray.hit, lightVec);               // Create a shadow ray at the point of intersection
shadowRay.closestPt(sceneObjects);              // Find the closest point of intersection on the shadow ray
float lightDist = glm::length(lightVec);        // The distance to the light source

// Calculating the Shadow lighting for Light 2
glm::vec3 lightVec2 = lightPos2 - ray.hit;
Ray shadowRay2(ray.hit, lightVec2);
shadowRay2.closestPt(sceneObjects);
float lightDist2 = glm::length(lightVec2);

bool isShadow1 = (shadowRay.index > -1) && (shadowRay.index != 1) && (shadowRay.index != 12) && (shadowRay.dist < lightDist);
bool isShadow2 = (shadowRay2.index > -1) && (shadowRay2.index != 1) && (shadowRay2.index != 12) && (shadowRay2.dist < lightDist2);
bool isLigtherShadow1 = ((shadowRay.index == 1) || (shadowRay.index == 12)) && (shadowRay.dist < lightDist);
bool isLigtherShadow2 = ((shadowRay2.index == 1) || (shadowRay2.index == 12)) && (shadowRay2.dist < lightDist2);

if (isShadow1) { //If the shadow ray hits an object (shadowRay.index > -1) and the distance to the point of intersection on this
    color = 0.2f * obj->getColor();
}
else if (isLigtherShadow1) {
    color = 0.7f * obj->getColor();
}

if (isShadow2) { //If the shadow ray hits an object (shadowRay.index > -1) and the distance to the point of intersection on this
    color = 0.2f * obj->getColor();
}
else if (isLigtherShadow2) {
    color = 0.7f * obj->getColor();
}
```

Figure 13 - Lighting & Shadow Code

Figure 13 shows the calculations used in generating the colours for the objects and the shadows. The colours from both lightings are added to give the overall colour to the object. The walls are not considered in this lighting feature as we want some portion of the scene to act as a contrast. The shadow calculation is done by getting the vector from the point of intersection to the light source and then creating a shadow ray at that point of intersection. We then find the closest point of intersection on the shadow ray and calculate the distance to the light source. If the object is not refractive or transparent, we give the shadow colour a darker shade. Otherwise, a lighter shadow shade is used.

Sphere Texturing

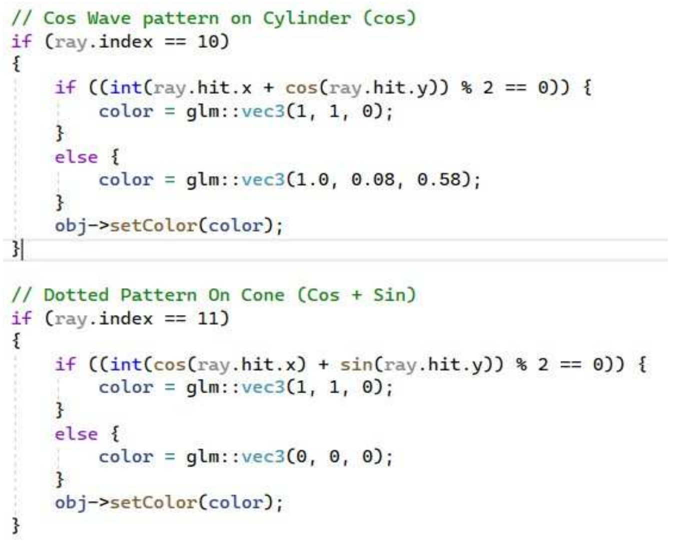


```
// Texture Map Jupiter on sphere
if (ray.index == 2)
{
    glm::vec3 center(24.0, 19.0, -80.0);
    glm::vec3 d = glm::normalize(ray.hit - center);
    float s = (0.5 - atan2(d.z, d.x) + PI) / (2 * PI); // Calculate s texture coordinate
    float t = 0.5 + asin(d.y) / PI;                    // Calculate t texture coordinate
    color = jupiterTexture.getColorAt(s, t);
    obj->setColor(color);
}

// Moon Texture on sphere
if (ray.index == 3)
{
    glm::vec3 center(13.0, 25.0, -65.0);
    glm::vec3 d = glm::normalize(ray.hit - center);
    float s = (0.5 - atan2(d.z, d.x) + PI) / (2 * PI); // Calculate s texture coordinate
    float t = 0.5 + asin(d.y) / PI; // Calculate t texture coordinate
    color = moonTexture.getColorAt(s, t);
    obj->setColor(color);
}
```

Figure 14 - Sphere Texturing

Figure 14 shows the calculations used for texturing the spheres with moon and Jupiter textures. The texture coordinates s and t are calculated by first normalizing the centre position at which the sphere is located and then applying the mapping formula.

Procedural Pattern Calculation

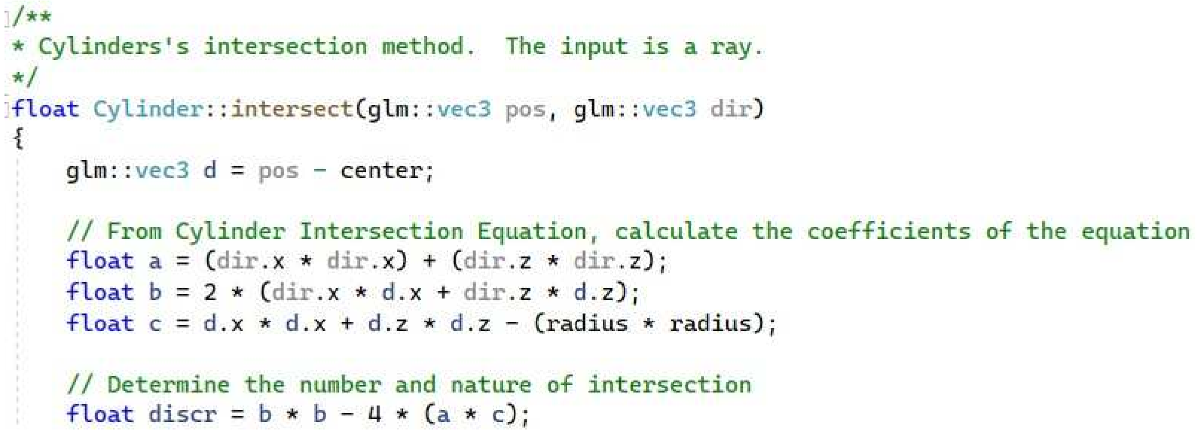


```
// Cos Wave pattern on Cylinder (cos)
if (ray.index == 10)
{
    if ((int(ray.hit.x + cos(ray.hit.y)) % 2 == 0)) {
        color = glm::vec3(1, 1, 0);
    }
    else {
        color = glm::vec3(1.0, 0.08, 0.58);
    }
    obj->setColor(color);
}

// Dotted Pattern On Cone (Cos + Sin)
if (ray.index == 11)
{
    if ((int(cos(ray.hit.x) + sin(ray.hit.y)) % 2 == 0)) {
        color = glm::vec3(1, 1, 0);
    }
    else {
        color = glm::vec3(0, 0, 0);
    }
    obj->setColor(color);
}
```

Figure 15 - Procedural Pattern Formula

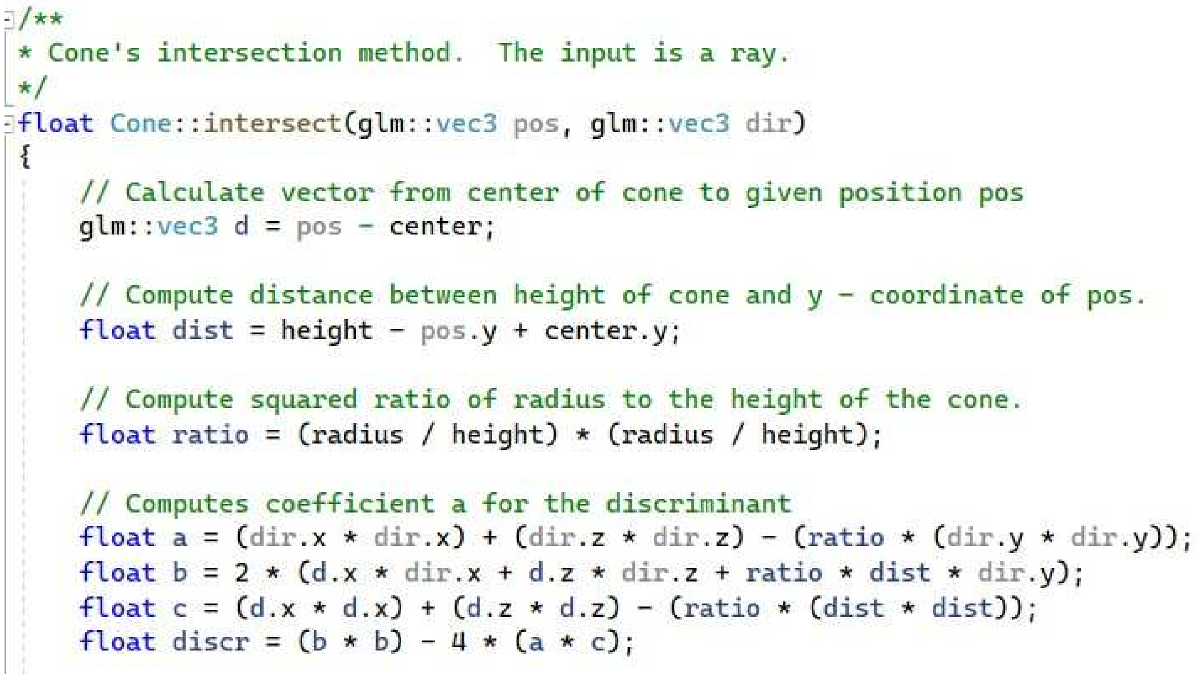
Figure 15 shows the calculations used for generating patterns on the surface of the cone and cylinder. The patterns are generated by getting the x and y values of ray hitting the surface and applying cosine and sine functions to them, switching colours in between.

Cylinder & Cone Intersection Method



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

    // From Cylinder Intersection Equation, calculate the coefficients of the equation
    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);

    // Determine the number and nature of intersection
    float discr = b * b - 4 * (a * c);
```

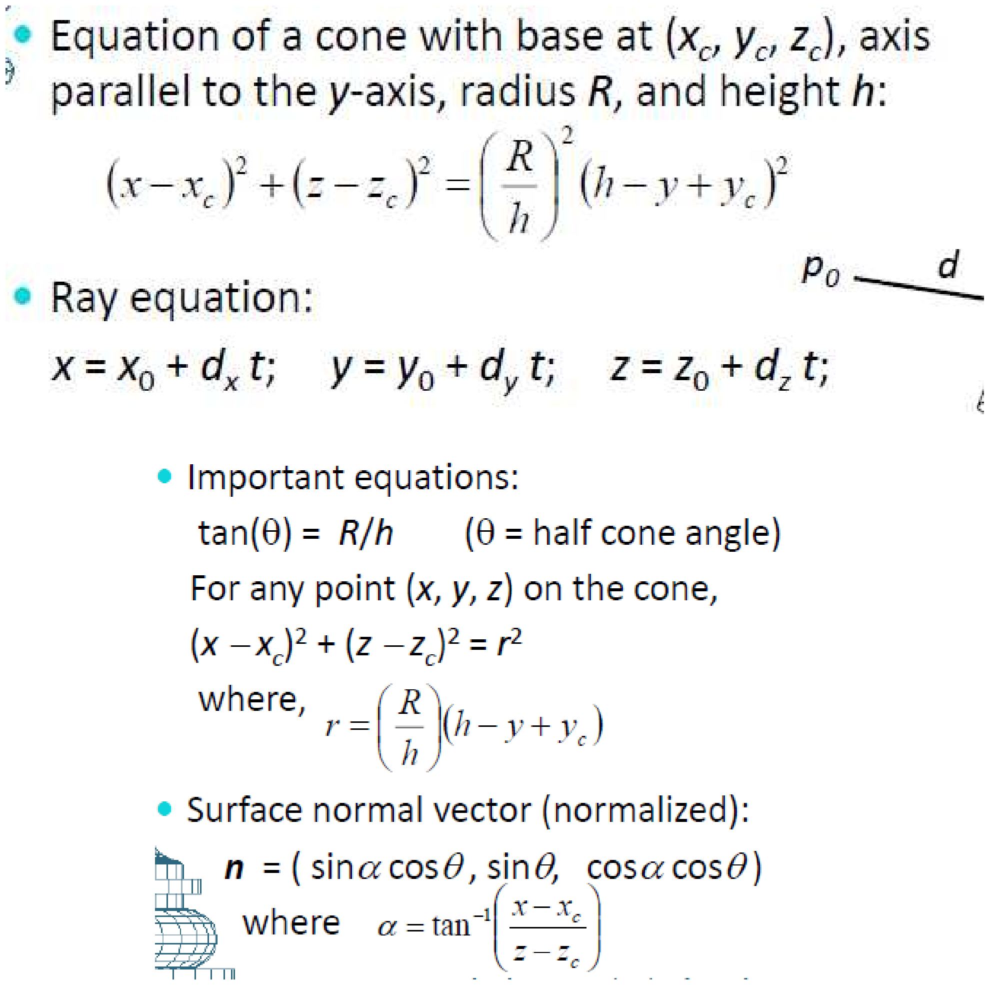
Figure 16 - Cylinder Intersection



```
/**
* Cone's intersection method.  The input is a ray.
*/
float Cone::intersect(glm::vec3 pos, glm::vec3 dir)
{
    // Calculate vector from center of cone to given position pos
    glm::vec3 d = pos - center;

    // Compute distance between height of cone and y - coordinate of pos.
    float dist = height - pos.y + center.y;

    // Compute squared ratio of radius to the height of the cone.
    float ratio = (radius / height) * (radius / height);

    // Computes coefficient a for the discriminant
    float a = (dir.x * dir.x) + (dir.z * dir.z) - (ratio * (dir.y * dir.y));
    float b = 2 * (d.x * dir.x + d.z * dir.z + ratio * dist * dir.y);
    float c = (d.x * d.x) + (d.z * d.z) - (ratio * (dist * dist));
    float discr = (b * b) - 4 * (a * c);
```

Figure 17 - Cone Intersection

Figure 16 and 17 shows the calculations used for ray with cone and cylinder intersection. The formula from lecture notes were used to determine the discriminant and the coefficients of the equation. By implementing the intersect and normal methods, we can display the object in the scene. The following formulae were used:



- Equation of a cone with base at (x_c, y_c, z_c), axis parallel to the y-axis, radius R, and height h:

$$(x-x_c)^2+(z-z_c)^2=\left(\frac{R}{h}\right)^2(h-y+y_c)^2$$

- Ray equation:

$$x = x_0 + d_x t; \quad y = y_0 + d_y t; \quad z = z_0 + d_z t;$$

- Important equations:

  $\tan(\theta) = R/h$ (θ = half cone angle)

  For any point (x, y, z) on the cone,

  $(x - x_c)^2 + (z - z_c)^2 = r^2$

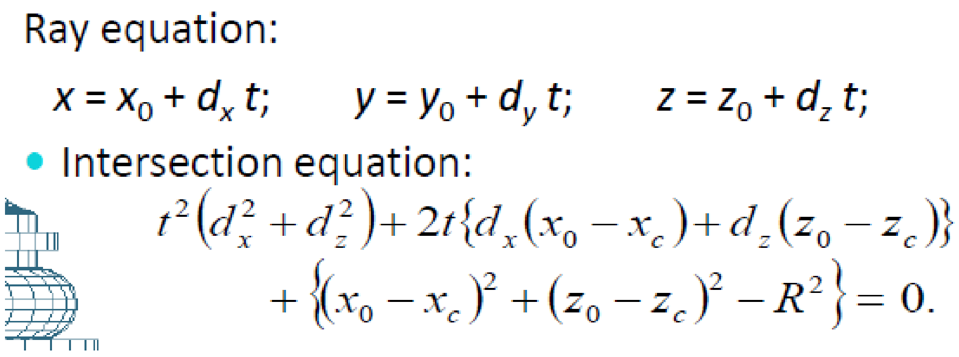
  where, $r=\left(\frac{R}{h}\right)(h-y+y_c)$

- Surface normal vector (normalized):

  $\mathbf{n} = (\sin\alpha\cos\theta, \sin\theta, \cos\alpha\cos\theta)$

  where $\alpha=\tan^{-1}\left(\frac{x-x_c}{z-z_c}\right)$

Figure 18 – Ray Cone Intersection Formula



Ray equation:

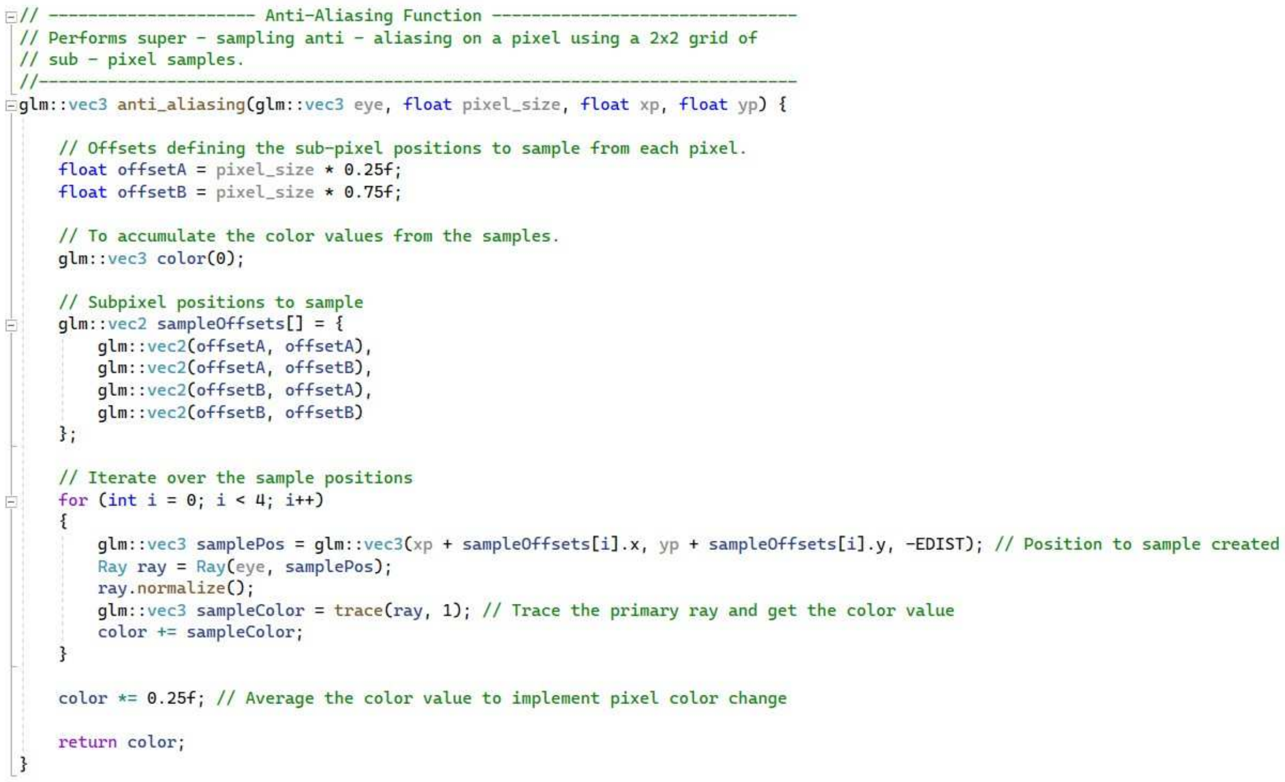
$$x = x_0 + d_x t; \quad y = y_0 + d_y t; \quad z = z_0 + d_z t;$$

- Intersection equation:

$$t^2\left(d_x^2+d_z^2\right)+2t\{d_x(x_0-x_c)+d_z(z_0-z_c)\}+\left\{(x_0-x_c)^2+(z_0-z_c)^2-R^2\right\}=0.$$

Figure 19 - Ray Cylinder Intersection Formula

Anti-aliasing



```
// --------------------- Anti-Aliasing Function ------------------------------
// Performs super - sampling anti - aliasing on a pixel using a 2x2 grid of
// sub - pixel samples.
//---------------------------------------------------------------------------
glm::vec3 anti_aliasing(glm::vec3 eye, float pixel_size, float xp, float yp) {

    // Offsets defining the sub-pixel positions to sample from each pixel.
    float offsetA = pixel_size * 0.25f;
    float offsetB = pixel_size * 0.75f;

    // To accumulate the color values from the samples.
    glm::vec3 color(0);

    // Subpixel positions to sample
    glm::vec2 sampleOffsets[] = {
        glm::vec2(offsetA, offsetA),
        glm::vec2(offsetA, offsetB),
        glm::vec2(offsetB, offsetA),
        glm::vec2(offsetB, offsetB)
    };

    // Iterate over the sample positions
    for (int i = 0; i < 4; i++)
    {
        glm::vec3 samplePos = glm::vec3(xp + sampleOffsets[i].x, yp + sampleOffsets[i].y, -EDIST); // Position to sample created
        Ray ray = Ray(eye, samplePos);
        ray.normalize();
        glm::vec3 sampleColor = trace(ray, 1); // Trace the primary ray and get the color value
        color += sampleColor;
    }

    color *= 0.25f; // Average the color value to implement pixel color change

    return color;
}
```

Figure 20 - Anti-aliasing Implementation

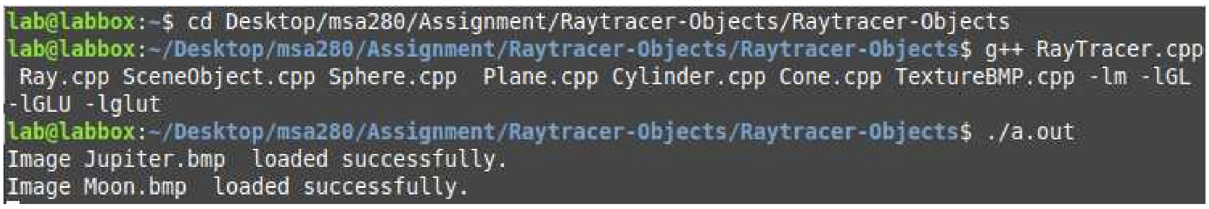
Figure 20 shows the calculations used for anti-aliasing. We generate a set of vector offsets and apply them to different pixel positions. Anti-aliasing is the process of smoothing out the edges of objects and shadows so that they appear less jagged. This is done by taking several pixel offsets around the edges and then returning the averaged final colour. Anti-aliasing is responsible for improving the visual realism of the scene.

Project Failures & Ray Tracer Shortcomings

The project proceeded as planned, however there were a few things that I tried to do but couldn't complete. I attempted to use the circle formula to implement the cylinder and cone's caps, but the cap only showed halfway through. Additionally, when two shadows overlap, the ray tracer does not adjust the colour of the shadow to ambient. Another issue I ran across while working on this project was the fact that the ray tracer takes a while to display the scene and that it is virtually hard to navigate about in the area using the controls I developed. The scene takes around 7.2 seconds to display, and this is done without the anti-aliasing. When the anti-aliasing is turned on, the scene takes 26.8 seconds to load. The blending effect comes at the cost of computing power. However, I had a lot of fun creating this ray tracer scene, and although I wasn't able to include all the extra elements, I will surely attempt to enhance it in the future by including the fog effects along with the depth of field and softer shadows.

Instructions to run project.

1. Download the project zip file and drag it to your desktop.
2. Extract items from the zip. A folder called "msa280" would appear.
3. Go to terminal and change directory by using:
   cd Desktop/msa280/Assignment/Raytracer-Objects/Raytracer-Objects
4. Compile the program using: **g++ RayTracer.cpp Ray.cpp SceneObject.cpp Sphere.cpp Plane.cpp Cylinder.cpp Cone.cpp TextureBMP.cpp -lm -lGL -lGLU -lglut**
5. Run the program using: **./a.out.**



```
lab@labbox:~$ cd Desktop/msa280/Assignment/Raytracer-Objects/Raytracer-Objects
lab@labbox:~/Desktop/msa280/Assignment/Raytracer-Objects/Raytracer-Objects$ g++ RayTracer.cpp
 Ray.cpp SceneObject.cpp Sphere.cpp  Plane.cpp Cylinder.cpp Cone.cpp TextureBMP.cpp -lm -lGL
-lGLU -lglut
lab@labbox:~/Desktop/msa280/Assignment/Raytracer-Objects/Raytracer-Objects$ ./a.out
Image Jupiter.bmp  loaded successfully.
Image Moon.bmp  loaded successfully.
```

Figure 3: Linux Terminal Instructions

Note: All images, files, libraries, documents and the final cpp file are located in the zip folder. They are labelled for clarity.

Extra Features

- Objects other than a Sphere: Cylinder.
- Objects other than a Sphere: Cone.
- Refraction of light through an object: Refractive Sphere).
- Multiple light sources including multiple shadows and multiple specular highlights generated by them.
- Multiple reflections generated using parallel mirror-like surfaces.
- Basic anti-aliasing.
- Textured Sphere: Jupiter & Moon.
- A procedural pattern generated (Cone and Cylinder). The pattern has a more complex structure than simple stripes or checks.

Controls:

↑ Arrow - Move Forwards +7
↓ Arrow - Move Backwards -7
← Arrow - Turn Left -7
→ Arrow - Turn Right +7
F1 – Move Down -7
F2 – Move Up +7

References

R. Mukundan. (2023). COSC363. Lecture Notes 6, 7, 8.

R. Mukundan. (2023). COSC363. Labs 6, 7, 8.

Jupiter Texture: https://www.pinterest.com/pin/494410865316878445/

Moon Texture: https://svs.gsfc.nasa.gov/cgi-bin/details.cgi?aid=4720

PNG to BMP Converter: https://cloudconvert.com/png-to-bmp