

COSC363 Assignment 2: Ray Tracing

Harry Dobbs 89030703

Overview:

A ray tracing algorithm has been implemented to create a realistic scene. The scene includes a wide variety of shapes and effects. The scene accurately utilizes features such as transparency, refraction, shadows, rotation, reflection, textures and specular highlights. These features combined with the use of various shapes create a visually spectacular scene. Fog is turned on by pressing the key 'f'. Fog is turned off by pressing the key 'c'. There is a delay when switching between the two states.

Build Command : `g++ -Wall -o "%e" "%f" Plane.cpp Ray.cpp SceneObject.cpp Sphere.cpp Cylinder.cpp Cone.cpp TextureBMP.cpp -lGL -lGLU -lglut`

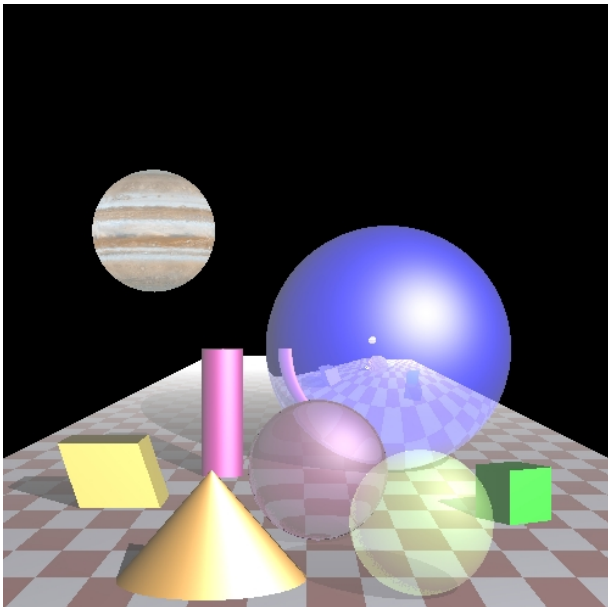


Figure 1: Ray tracing scene with fog turned on.

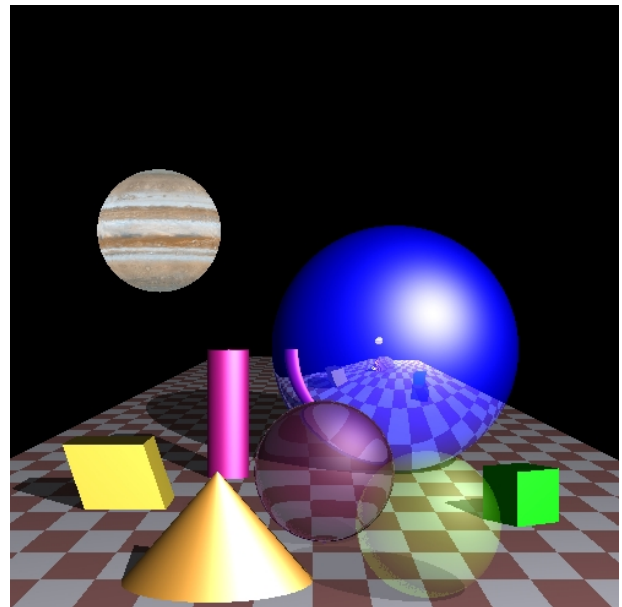


Figure 2: Ray tracing scene with fog turned off

Standard Features (9 marks):

- Contains one light source as well as diffuse, and specular reflections generated by the source.
- Contains shadows for every object. Transparent objects create a lighter shadow.
- The scene contains one large blue reflective sphere.
- The scene contains two boxes. One is skewed and one is regular.
- The scene contains a chequered floor.

Additional Features (7-9 marks):

Primitive Cone (1 mark) :

To be able to display a cone within my assignment, each ray must be able to calculate:

- If it has intersected with the cone
- The normal vector at the point of intersection

These two requirements were constructed as functions. To be able to implement these functions, geometric equations were required. The equations for a cone, and cylinder accompanied with labelled diagrams were provided from the lecture notes (Mukundan ,2019). The points of intersection are obtained by substituting the ray equations (Equation 2) into the general point equation (Equation 1). The resulting equation is then solved for t (which can then be used in ray tracing equations). An issue I had with the original implementation was that a second cone was generated on top of the cone I wanted to display (Figure 3). This issue was resolved by substituting the value of t into the Y ray equation (Equation 2) and ensuring the height was less then the summation of the cone centre plus the height of the cone (Figure 5). A similar method was used to constrain the minimum height of the cone (Figure 5). The normal vector is important to ensure the object is lit correctly. To obtain the normal vector Equation 3 was implemented.

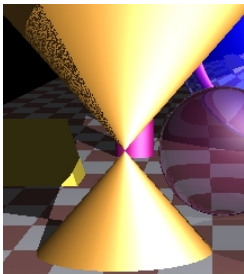


Figure 3: Image of incorrectly implemented cone.

$$(X - X_c)^2 - (Z - Z_c)^2 = \left(\frac{R}{h}\right)^2 (h - Y + Y_c)^2$$

Equation 1: Every point on cone must satisfy this equation.

$$X = X_o + d_x * t \quad Y = Y_o + d_y * t \quad Z = Z_o + d_z * t$$

Equation 2: Ray equations.

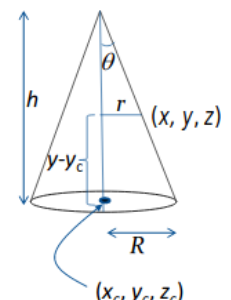



Figure 4: Diagram illustrating the location of each point.

```
if(center.y + height > posn.y + (dir.y * t) && center.y < posn.y + (dir.y * t))
{
    return t;
}
else
{
    return -1;
}
```

Figure 5: Equation implementation to ensure cone is contained with certain values of Y.

Surface normal vector (normalized):


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

where $\alpha = \tan^{-1} \left(\frac{X - X_c}{Z - Z_c} \right)$

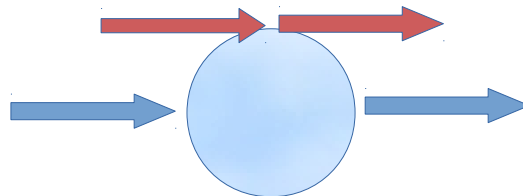
Equation 3: Surface normal vector equation for cone



Figure 6: Image Capture of Cone Implementation

Transparent Object (1 mark) :

When an object is transparent an additional ray is generated in the same direction as the ray that intersected with the transparent object. However this ray's origin is on the other side of the transparent object. The pixel colour that is displayed to us is a result of the transparent object colour multiplied by some transmission factor summed with the colour returned from the additional ray. An issue I had with the implementation is that sometimes the origin of the additional ray is the at the same point the original ray contacts the object (as shown by red arrow in Drawing 1) as opposed to being on the other side of the object (as shown by blue arrow in Drawing 1) . This issue was relatively to simple to resolve and once resolved the transparent object edges looked cleaner. This issue was resolved by testing whether the additional ray intersected the sphere or another part of the scene. Depending on whether it hit the sphere or another part of the scene it would either generate another ray or return the colour value of the original ray.



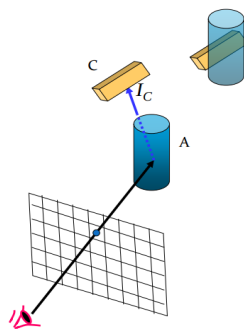
Drawing 1: An Illustration demonstrating two scenarios when dealing with transparent objects.

Refraction of Light (1 mark) :

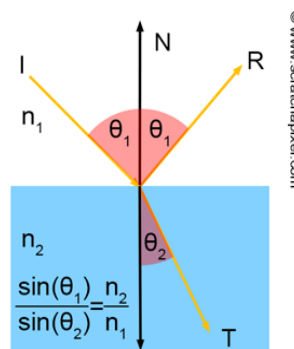
Refraction occurs when light travels through a transparent medium to another medium and the light is bent around a boundary. Light travels at different speeds in different mediums. Different mediums have different indexes of refraction. GLM has a function that takes the direction vector, normal vector and the ratio refractive indices. This function will return a vector that is pointed in the refracted direction (I_c) . This vector is then traced. The colour of the pixel of given by Equation 4 where I_c is the refracted vector, p_t is the scale factor (<1) and I_a is the ray that hits the refractive medium.

$$I = I_a + P_t I_c$$

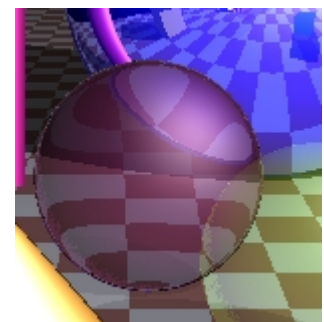
Equation 4:
Pixel colour
equation for
refraction



*Figure 7:
Refraction
example from
COSC363 Lecure
Slides (Dr. R.
Mukundan ,2019)*



*Figure 9: Image
demonstrating
Refraction (Scratchpixel
2014)*



*Figure 8: Refractive
Sphere in scene.*

Rotation or Shear Transformation (1-2 mark) :

The scene implements the rotation of one cube and a shear transformation on another. The Implementation of a shear transformation on a cube is rather trivial. I created a shear cube along the X direction. This was done by shifting all the upper points of the cube, along the X axis in the negative direction. To implement this shift I used basic trigonometry to shift the upper vertices by an amount dependant on the shear angle (Equation 5). Where H is the height of the cube and Phi is desired shear angle.

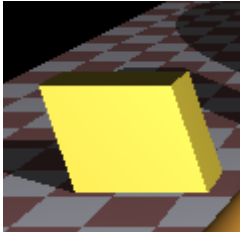


Figure 10: Shear Transformation of a Cube

$$X_{New} = X_{old} + H_{cube} * \tan(\phi_{(shear\ angle)})$$

Equation 5: Shear Equation for Cube

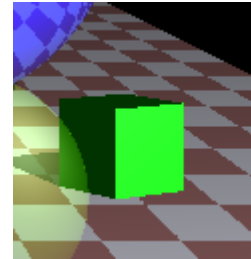


Figure 11: A cube rotated 45 degrees around the Y axis.

Another cube was rotated 45 degrees around the Y axis. In order to achieve this the cube was generated at the origin of the scene. Every vertex underwent a transformation as described in the lecture slides (Dr. R. Mukundan ,2019) . This transformation involves every vertex being multiplied by a rotation matrix. This results in every point undergoing a transformation. The effect of multiplying by a rotation matrix has the same effect as applying Equation 6 to each vertex. Once each point has been rotated around the origin, each point undergoes a translation which is done by adding a position vector to each vertex.

Rotation about the y-axis:

$$(x, y, z) \rightarrow (x \cos\theta + z \sin\theta, \quad y, \quad -x \sin\theta + z \cos\theta)$$

Equation 6: Rotation around Y-axis transformation formula. Theta determines how much the object rotates.

Non-planar Object Texturing (1 mark) :

I implemented the texturing of Jupiter onto a sphere. The image of Jupiter was found online from an open-source website (Solar Textures, 2019). To display an image on a sphere the sphere must have its longitude and latitude coordinates mapped to the x and y coordinates of the image. To implement this texturing first a unit vector is made between where the ray hits the sphere and the centre point of the sphere. This unit vector's components are then substituted into the following equations.

$$X = 0.5 + \frac{\arctan 2(d_z, d_x)}{2\pi}$$

Equation 7: Image X Coordinate Equation

$$Y = 0.5 - \frac{\arcsin(d_y)}{\pi}$$

Equation 8: Image Y Coordinate Equation

These equations return the point on the texture that will be mapped to the point the ray contacted.

Fog Implementation (1-2 mark) :

I have implemented fog which can be toggled on and off (Figure 1, Figure 2). The amount of fog is equivalent to the fog density multiplied with the distance. As the amount of fog increases the pixel colour gets closer to white. When the distance at a certain point is equal to the maximum distance it can be observed that the equations will equate to a value of 1 (white). When the distance is very small the equations will equate to an equation that is very close to the original colour

$$r' = \frac{d}{d_{max}}(1-r)+r \quad g' = \frac{d}{d_{max}}(1-g)+g \quad b' = \frac{d}{d_{max}}(1-b)+b$$

Equation 9:
Implementation of
fog on the red
component

Equation 10:
Implementation of
fog on the blue
component

Equation 11:
Implementation of
fog on the green
component

```
float fogDist = -ray.xpt.z - fogstart;
objectColor.x += (fogDist/fogmax)*(1 - objectColor.x);
objectColor.y += (fogDist/fogmax)*(1 - objectColor.y);
objectColor.z += (fogDist/fogmax)*(1 - objectColor.z);
return objectColor;
```

Figure 12: Code snippet of fog implementation

Primitive Cylinder (1 mark) :

A cylinder object was implemented into the scene. The intersection equation (Equation 8) was available from the lecture slides (Dr. R. Mukunda ,2019). This equation was solved for t to find the points of intersection. Just solving for t will give a cylinder of infinite height. This problem was resolved by constraining the t values based on their predicted heights (similar to the sphere). One issue I had with displaying the sphere was that the top of the sphere was not visible. This was resolved by rearranging equation the ray equations in order to calculate the predicted height of the cylinder.



Figure 13:
Cylinder
object in
scene

Intersection equation:

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

Equation 8: Intersection equation for a Cylinder

$$\mathbf{n} = ((x - x_c)/R, \quad 0, \quad (z - z_c)/R)$$

Equation 9: Normal Equation for cylinder

References :

Dr. R. Mukundan . (2019) . Computer Graphics COSC363 Lec09_RayTracing [PowerPoint Slides]. Retrieved from <https://learn.canterbury.ac.nz/mod/resource/view.php?id=803416>

Scratchapixel. (2014, August 15). Retrieved May 30, 2019, from <https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel>

Solar Textures. (n.d.). Retrieved May 29, 2019, from <https://bit.ly/2Z2xlxl>
Jupiter Texture