COSC342 – Ray Tracer

Garth Wales | 4861462

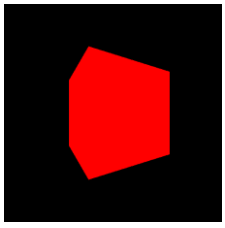
The ray tracer functions by taking each pixel of the screen, and casting rays. Each ray checks each object for an intersection. Each ray has a position and direction. We then find out if you multiply the direction by some positive value does it intersect with the object, in its unit form.

For the plane this is simple as it is a plane in line with the z-axis, and then check if x and y of the hit point is within [-1,1]. For the cube you are checking for 6 different planes located +/- 1 on each axis.

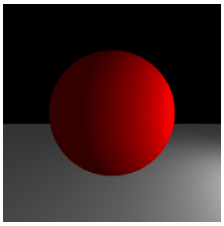
For the cylinder it is more complex. Take the ray point and direction but with the z components set to zero. This allows us to then calculate if the ray intersects a circle infinitely along the z-axis. This intersection either occurs 0, 1 or 2 times. We then check if the intersection is within +/- 1 on the z-axis. Then, as it also has closed ends, if it intersects past -1 or 1 check if it is within a unit circle. Each cap, and its normal, is the opposite sign of its position on the z-axis.

For the octahedron we have quite a different case. The octahedron has 8 faces that are each a triangle. To find an intersection I first created arrays of each vertex (A,B,C) of each face. Each face is looped through where the normal of this face is calculated AB x AC, the plane of intersection is then calculated by taking the dot product of the normal with any point in the plane. I then calculate if the ray intersects the plane of this face. Then, if it does intersect in front of the camera, I must check if the hit point is within the triangle. This is calculated by checking if the cross product of each edge and point has the same direction as the normal. If this is the case, and so each calculation is greater than or equal to 0, then it successfully hit a face of the octahedron.

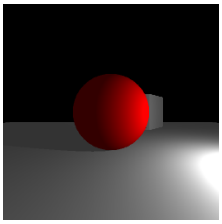
While working on all of these, I would add them to a scene with a different lighting and rotate them in various ways to ensure they work as expected and all the normal vectors are appropriately defined. As such you can see the progress in the following figures:



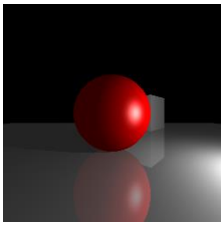
Cube
(Ambient lighting)



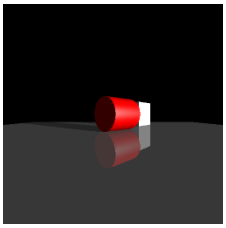
Sphere + Plane
(Diffuse, No shadows)



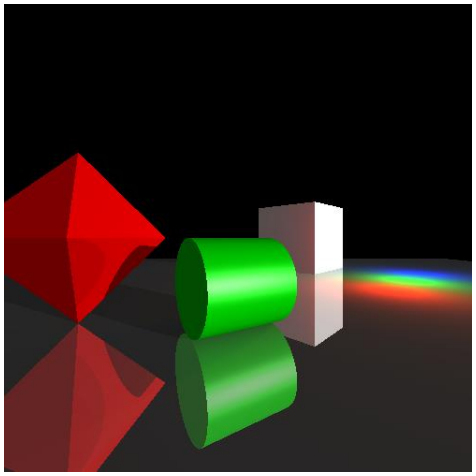
Sphere (Shadows)



Sphere + Plane + Cube
(Mirror + Specular highlights)



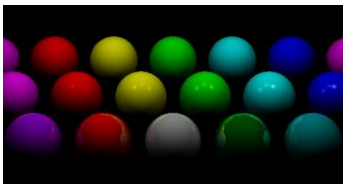
Cylinder + Cube
(Directional light only)



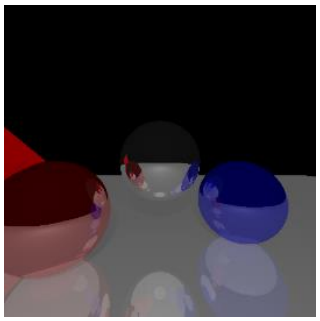
My sample scene features an octahedron, cylinder with specular highlights, cube, plane that has a mirror material as well as a directional light. This showcases some of the more complex elements implemented in this ray tracer.

The scene, sampleScene.txt, is located in the scenes/ folder along with other test scenes.

For each intersection, the computeColour() method is called to calculate the light provided by each light in the scene. This sums the diffuse component, specular component, mirror component and shadow components for each object. For each light, we work out if we travel backwards from the hit point if we reach something closer than the distance to the light. If this is true, then it is in shadow and so this light does not contribute to the colour at this point. Otherwise, diffuse lighting is the unit normal of the hit surface dotted with the unit normal of the light, to give the reflection of light based on the angle of the surface to the light. Specular lighting is the unit view normal dotted with the unit light reflection normal, raised to the power of its specular exponent. This produces bright highlights as the light source itself reflects on the object. To add mirror reflections, it is as simple as calculating the mirror ray direction and then recursively calling computeColour() with the mirror ray, doing this up to some maximum ray depth.

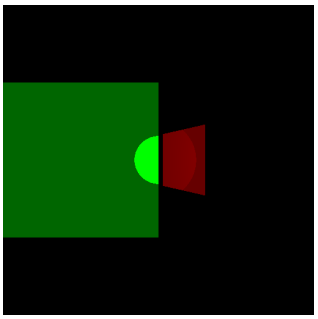


To test general lighting I created a scene, lighting.txt (see above figure), which shows diffuse, specular and mirror on a broad spectrum of different RGB components for each of the material, diffuse, specular and mirror components.



To test mirror reflections I created a scene, mirror.txt, to show four different mirror objects all reflecting each other.

To implement directional lights all that was required was to return the colour of the light for its getIlluminationAt() as these lights provide a constant illumination at an infinite distance. I used these in many of my scenes they clearly worked as intended.



Spotlights are implemented by taking the dot product of two normal vectors. This provides the cosine of the angle between these normal vectors. To test spotlights, I made a scene, spotlight.txt, that shines onto two planes placed at different distances. I tested this originally without any intensity drop-off to ensure the angle cut off worked as intended.

Overall, I am pleased with my progress and have definitely learnt a lot.