Vector Calculations

To first perform raytracing, vector calculations need to be implemented first. Various operations were implemented, such as addition, subtraction, scalar multiplication, cross product, dot product, scalar division, getting the length of a vector from the origin, getting the square length of it, etc. Vectors are the basis of not just ray tracing, but also colour calculation and point calculation. A simple write function was created to output the colour onto a ppm file. That was simple enough for CoPilot to code for me and it also wrote the tests.

Rays are an extension of vectors.

Basic Raytracer 1: Camera

Initially, the camera was set at position 0,0,0 while looking towards position 0,0,-1. Initially, the camera has a width and an aspect ratio. Image height is calculated from the aspect ratio with the width. Other variables were set concretely, such as focal length, and viewport_height with viewport_width calculated from the viewport_height by multiplying it by the ratio of the width and height. The camera is implemented by drawing from 0,0,0 to pixels on the plane and drawing a ray beyond that.

The centre of the 0,0 pixel is calculated on this flat plane, which is used to represent the camera. It is where the positions of the other pixels are calculated from to be used in the ray calculation. Whatever the ray hits, it will try to return the value of the colour attained from that hit point.

Initially, camera was created in the main file. Later, it was moved to its own camera header file. The camera calls the writeColor function in a function called render, which does the calculations for the colours that are attained by the rays.

After this, the camera was given more features, such as field of view, a positionable position, an orientation given by the direction in which the camera looks at and also an up axis direction to help in the orientation of the camera. This involves some vector calculus to fulfill.

Basic Raytracer 2: Intersection Tests

Three shapes are implemented in this raytracer, spheres, cylinders and triangles. The sphere has a radius and centre position. The cylinder has a radius, axis direction, radius and center. Triangles are defined by the position of their three points. Broadly speaking, when a ray is shot, all the shapes are checked against the shape to see if the ray ever intersects it. The result of the equation for this calculation will result in a quadratic equation. For the triangle however, it is a matter of equating the plane of the triangle with the ray and finding whether the ray is on the triangle.

For a sphere, the calculation of its intersection is already known. CoPilot provided the intersection calculations for cylinder and triangle. Initially, CoPilot didn’t provide the calculations for the caps as cylinder was initially assumed to be an infinite cylinder. Caps will have to be calculated separately from the body. CoPilot provided the cap calculation.

According to CoPilot, the Triangle calculation is based on the Muller-Trumbore Intersection Algorithm to check whether a ray intersects the triangle. Initially, I did not change the calculations but after, I modified them to use half-b calculations for quadratic equations for sphere and cylinders. Some debugging was undergone on the cylinders to get them working properly. At the end, as it was noticed that cylinder calculations were very slow, some rearrangement of the cylinder intersection code was done.

Basic Raytracer 3: Blinn-Phong Shading

CoPilot set up the initial Blinn-Phong shader, which was called on scatter function whenever a ray hits a point. When a ray hits a point, a record is created, noting down the relevant information for further calculations for colour. I initially implemented a Lambertian shader to make sense of the material calculation. At this point, objects do not consider the light contribution as light wasn't implemented yet. At that point, the shader only accounted its own variables.

Later, when lights are implemented, I wrote that for all hit points on a record, after the materials, intersection and t scalar are calculated, a function is called for calculating the total light contribution from all light sources. This account for distance of the light from the point and whether the point is in shadow or not. After the total light contribution is calculated, it is passed on as attenuation to be used as the final colour calculation. This allows for a bounce back effect for light and allows the light to grow weaker the further away it is from the scene.

Basic Raytracer 4: Shadows

My initial plan was to shoot rays from all the light towards the point but CoPilot written out the code to shoot shadow rays towards the light sources. In my mind, there is no difference in both approaches so I went with what CoPilot has written. If the shadow ray is blocked by any other object, then it is decided that the area will only receive, initially, a black colour but later on, an ambient colour derived from the material of the point but scaled to be darker.

This allows for clear, explicit shadows to be defined.

Basic Raytracer 5: Tone Mapping

CoPilot suggested Reinhard Tone Mapping for the tone mapping. For the reader's information, my raytracer was taking multiple samples on a point to calculate an average of the colour output for antialiasing. I was applying reinhard tone mapping on the multiple samples at a time before noticing. With how Reinhard Tone Mapping is done, dividing a colour by itself + 1 on all elements, that produced an almost black picture. I only realised the problem later on. After, Reinhard Tone Mapping was placed after calculating the average colour at the point.

Basic Raytracer 6: Reflection

Instead of defining a unique material for reflective material, I have decided to use the Blinn-Phong model to create reflective material. If the reflectiveness is set to true, then all rays are reflected. However, they are multiplied by a reflectivity value which is from 0 to 1. Various combinations of probability and attenuation was used but it was decided that reflectivity is going to be used to attenuate the value, instead of being used as the probability of getting a reflected ray. Should be noted that non-reflective materials are allowed to reflect some rays but that is instead decided by probability based on the ks value.

Basic Raytracer 7: Refraction

Refractive material are also implemented in the Blinn-Phong materials. A special refract function was created that allows calculation for a ray that takes place in a refractive material. In the refractive section of Blinn-Phong, the Shlick approximation is used. If the ray hits the refractive material at a certain angle, then it will reflect. Otherwise, it will use the refract function. Refractive materials do not actually account for their own variables except its own refractive index.

Pathtracer 1: Pixel Sampling

To anti-alias multiple points were picked in a disc around a pixel. When a random point is picked, that is used in conjunction with multiple other points on the disc to calculate the average amount of light from a point. This is to prevent aliasing in the ray tracer. Initially, I planned for the area to be a unit square around the pixel but CoPilot help write a function to generate a random point around a unit disk instead of a unit square which would help implementing the anti-aliasing feature.

Pathtracer 4: Light Sampling

For the calculation of a shadow, very much the same principle was done as doing pixel sampling is used for the anti-aliasing. Instead of a disc however, the point picked is from around the unit sphere in which the light centre was at. When a shadow ray is shot from an intersection point, it shoots multiple times to multiple different points int he light sphere to help generate soft shadows. Not a lot of shadow rays are needed however. Currently, the raytracer uses only 10 shadow rays but it produces a dramatic effect.