322COM Reflective report

For my coursework I have been given the task of creating a real-time ray tracer using C++. To accomplish this task I chose to use the SDL library as it allows for easy manipulation of the pixels on screen, as well as methods for receiving user input.

I used multiple resources to help build up my application, with my main starting point being [this article](https://www.programmersranch.com/2014/02/sdl2-pixel-drawing.html). I chose to base my code from this example because it uses an SDL Texture, as opposed to an SDL Surface. SDL Textures are new to SDL2, and are superior to surfaces as discussed in [this SDL to SDL2 migration guide](https://wiki.libsdl.org/MigrationGuide):

Text

Description automatically generated

Figure 1 - SDL Migration Guide

While the performance difference for an application such as mine may be minor, I still wanted to use all the tricks I could to get the best performance from my ray tracer.

A picture containing graphical user interface

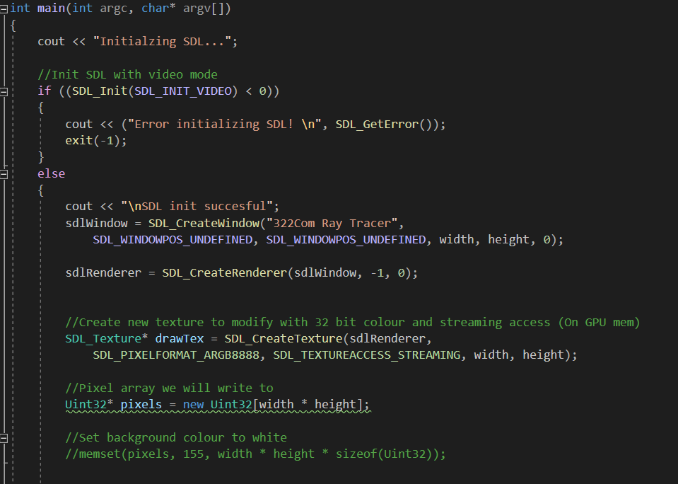
Description automatically generatedCreating an SDL context was simple enough, and it didn’t take long for me to add a sphere class and draw a basic shape to the screen.

Figure 3 - The starting point for my Ray Tracer

Figure 2 - Code for creating an SDL window and texture I can draw to

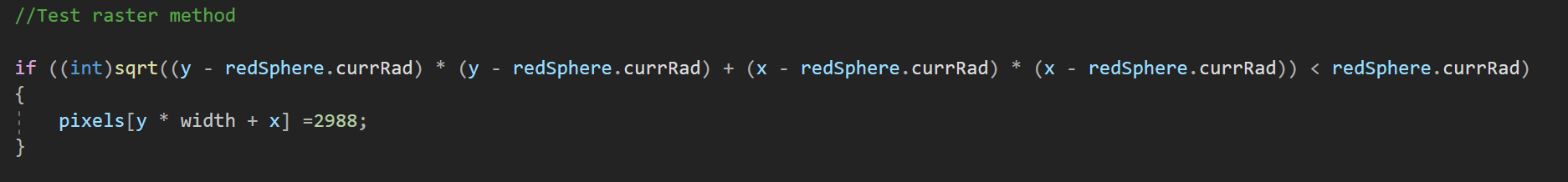
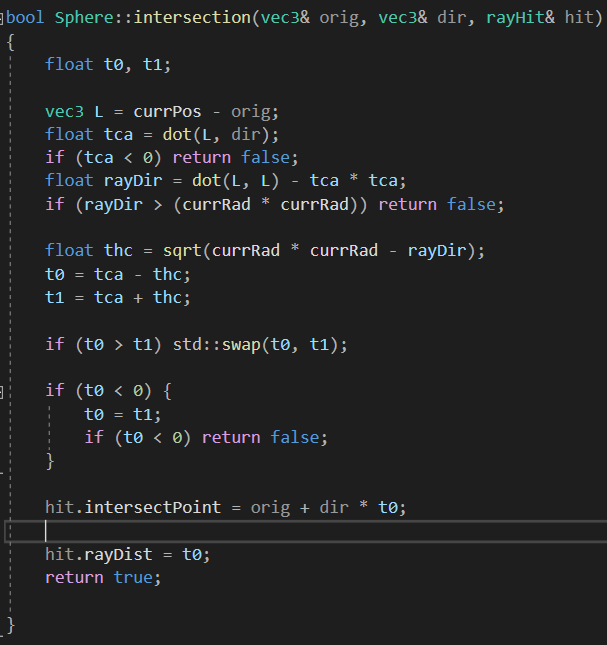


Figure 3A - Simple code I wrote for drawing a circle

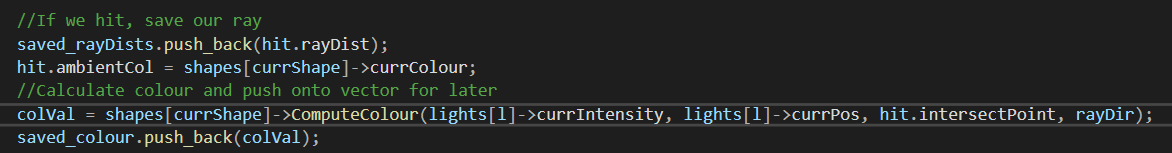
My next step was to implement a ray-sphere intersection method, which will allow me to cast a ray from each pixel of the sphere, and see if it hits my sphere based on its origin, direction and the spheres radius/position.

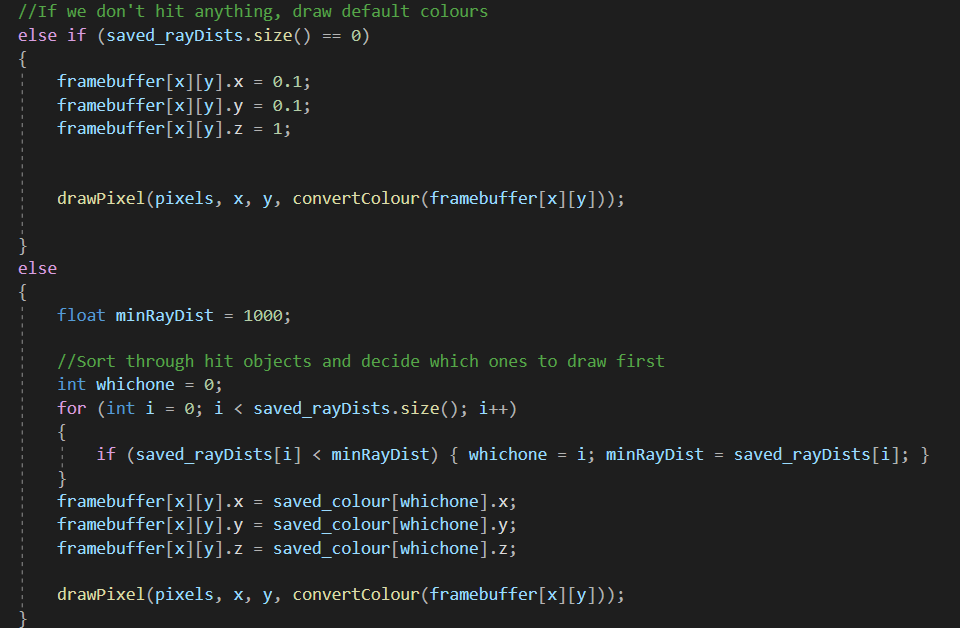
This code is based on the Geometric solution, which allows us to reject the ray early by comparing if the ray direction is greater than the sphere’s radius. The downside to this is that it forces us to calculate the square root of the ray direction, so instead of computing this value I check if the ray direction squared is greater than the radius squared.

The square root operation is very costly so it should be avoided wherever possible.

After computing the intersection I save the intersection point and the ray distance/t value to a ray struct I pass in for later use.

Figure 4 - Sphere intersection code

This code doesn’t contain any logic for sorting depth, so I added it in my main drawing loop:   


Figure 5 - When a shape is hit, I save the ray intersection distance, calculate the colour and then save it to a vector

At the end of the for loop I run through the vector of saved ray distances (t values). If there are none (We didn’t hit anything), then I draw the default background colour, otherwise I sort through the distances and decide which one to draw first.

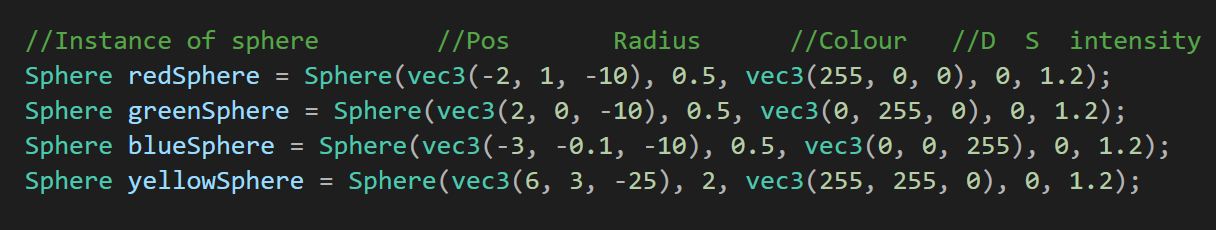


Figure 6 - Multiple instances of my sphere class

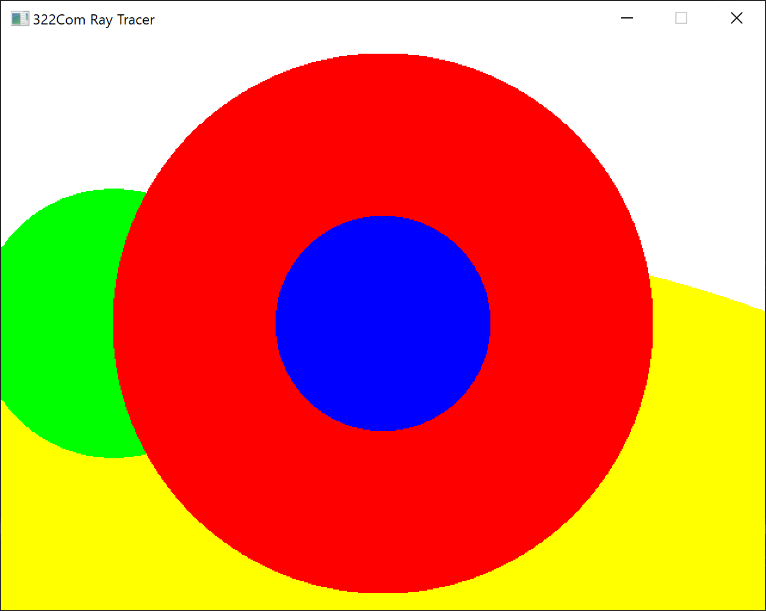
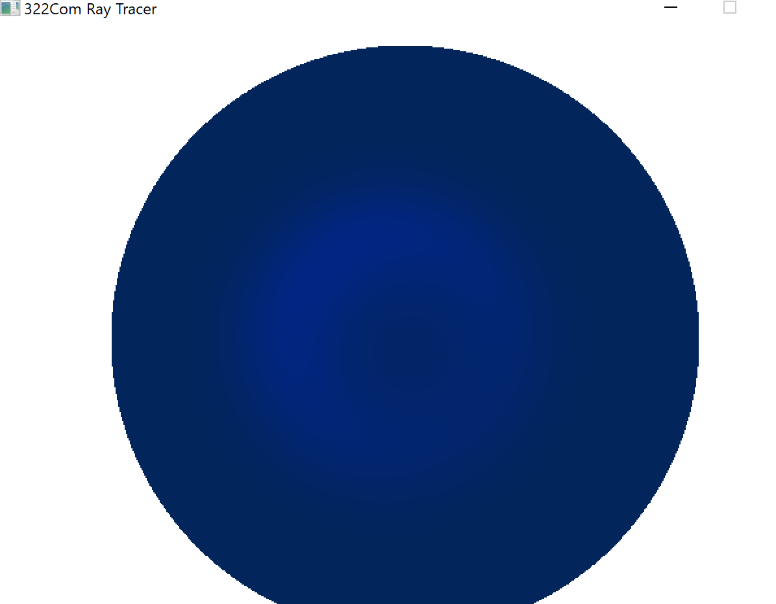
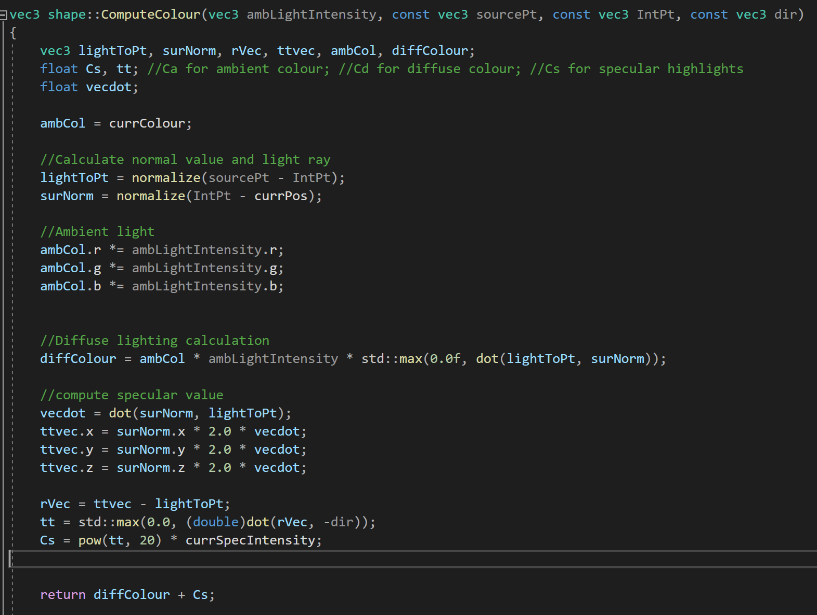
I was able to add multiple spheres at varying depths and have them be drawn correctly.

Figure 7 - My updated function that returns a vec3 RGB value

Figure 8 - My first attempt at computing the colour

Figure 9 - Depth sorting working

My next task was to add diffuse, specular and ambient lighting. Ambient lighting is simple as it just involves adding a value to the shapes base colour. Diffuse and specular are a little more complicated, but I was able to implement them easily enough by looking at their equations:



Figure 10 - Diffuse lighting equation

Figure 11 - Specular Lighting Equation

My first attempt at adding lighting didn’t go to plan. The function I used to compute the colour returned the final colour value as one integer, whereas the rest of my code uses a vec3. I initially tried converting the colour to a vec3 (Which produced the result shown below), but in the end I decided to re-write the function so that it returned an RGB colour value.

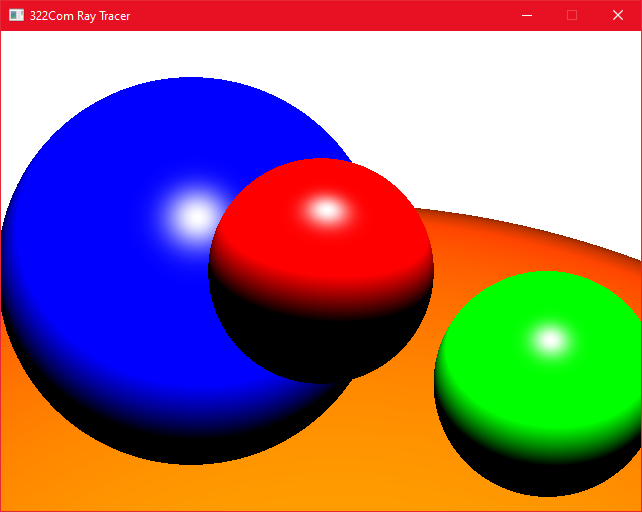
With this updated function the colours/lighting work perfectly, letting me move onto adding more primitives.

Figure 12 - Specular and diffuse lighting

I wanted to add a plane as this would also let me add shadows to my project. To define a plane, you need a function that accepts a point on the plane and then a normal value.

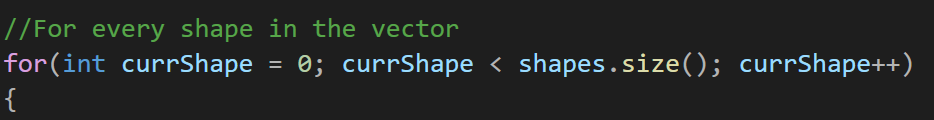
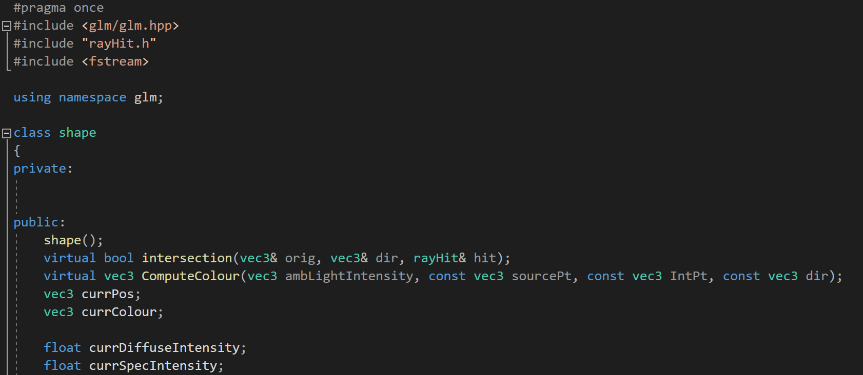
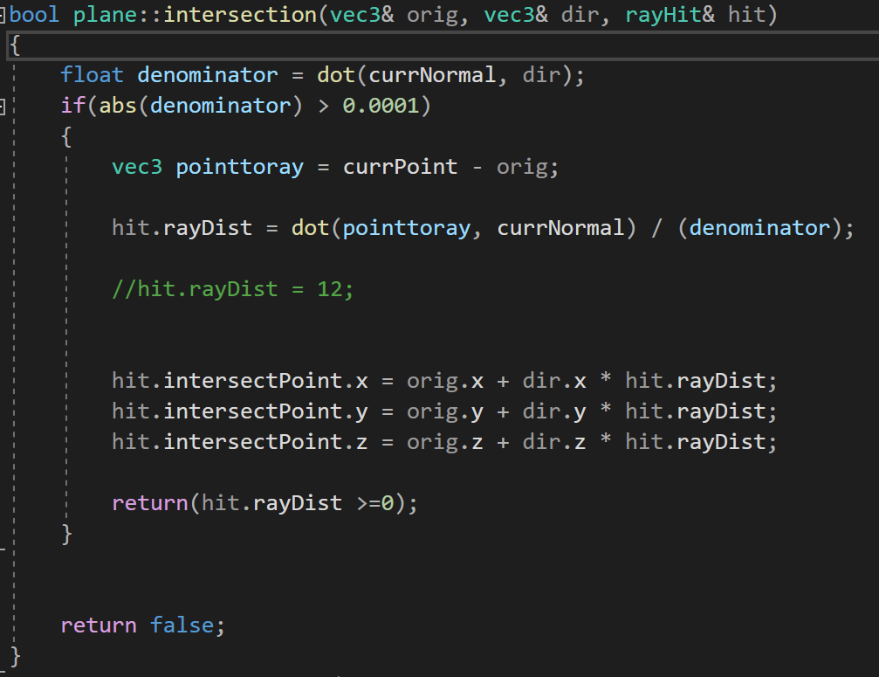
Just like my sphere intersection method, I needed to pass in the origin and direction of my main ray. To keep things simple I decided to split my shapes into classes. I created a base class called Shape, which has virtual functions for computing the intersection and the colour, and then my subclasses such as Plane and Sphere inherit from Shape and override these virtual functions. I am then able to create instances of these shapes in the main method and push them back to a vector of shapes, which I then run through in my loop for each pixel.

Figure 13 - Iterating through each shape

I had to make sure my vector was a vector of pointers, otherwise only the base class functions would be called, rather than the overrides.

This object-oriented design makes it easy to add new shapes to my program. I was able to make a plane class, and override the base intersection function with the following code:

Figure 14 - My shape class

If the denominator is close to 0, the plane and the ray either perfectly coincide (Resulting in an infinity of solutions), or the ray is away from the plane which gives us no intersection. This means we return false if the denominator is less than a small value (0.0001 in my example).

If the ray does intersect then I calculate the distance and point of intersection, like in my sphere class, before saving it to the ray struct I pass in.

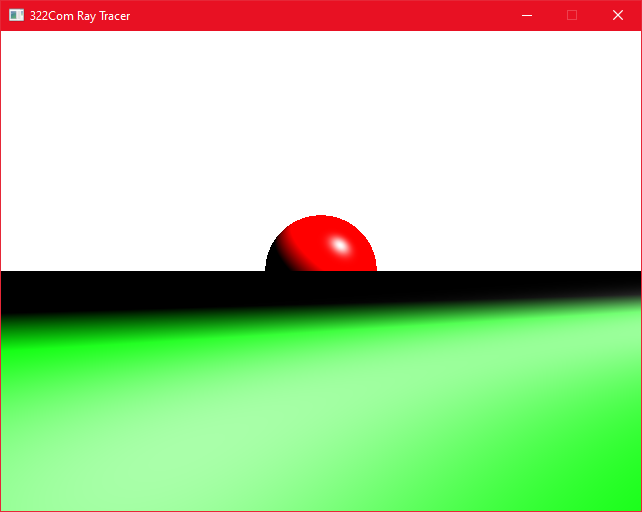
My initial attempt at adding the plane was somewhat broken:

Figure 15A - The broken plane

It took me a little while to fix this, but the problem turned out to be that I had misread the formula, and I was mistakenly using the ray direction value in my calculation of the ray distance when I should have been using the normal value.

This was causing the incorrect value to be returned and therefore meant my depth sorting code was drawing it in-front of the spheres rather than behind.



Figure 16 - my mistake

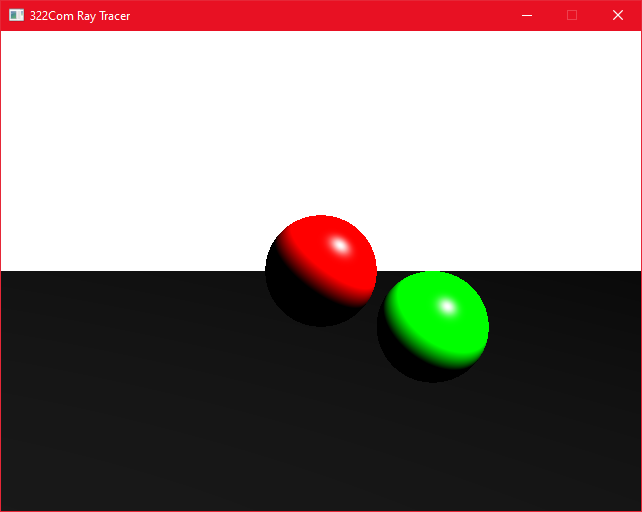
After correcting my mistake my plane looked like this:

Figure 17 - The plane now draws correctly

The next thing I needed to add were shadows. These are calculated by waiting until we intersecting with a shape, and then running through each shape again but this time running the intersection check with the original intersection point as the origin, and the opposite direction to our light source as the direction. If these secondary “shadow rays” hit anything, we then set that colour to black to show that it’s in shadow.

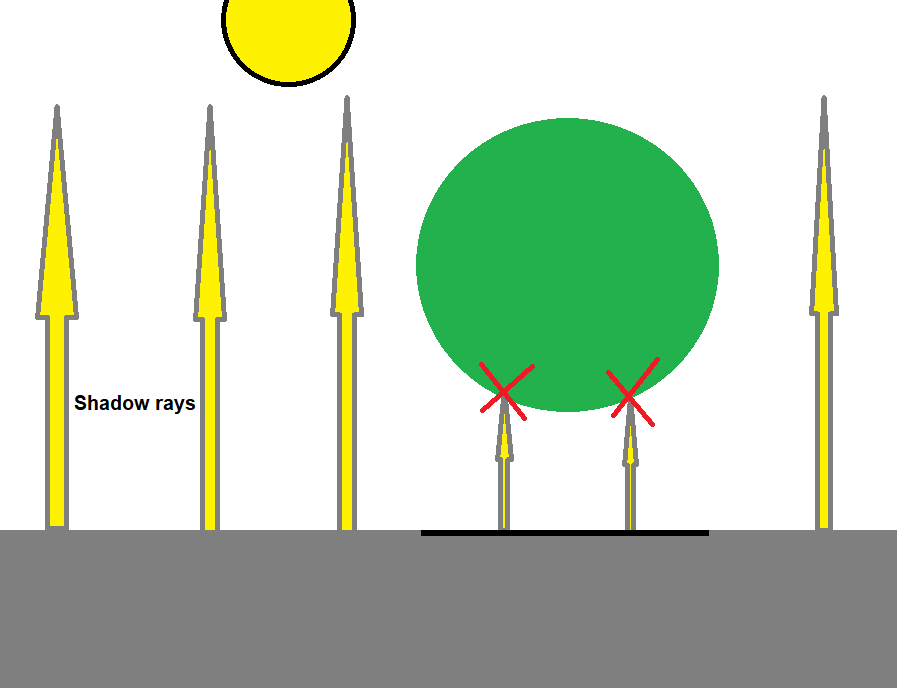


Figure 18 - Diagram showing shadow calculation

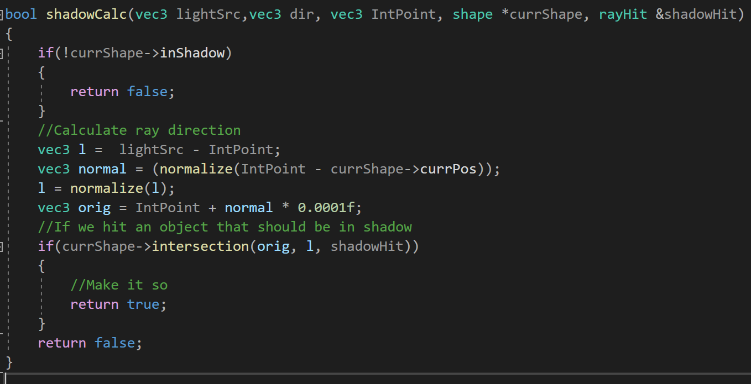
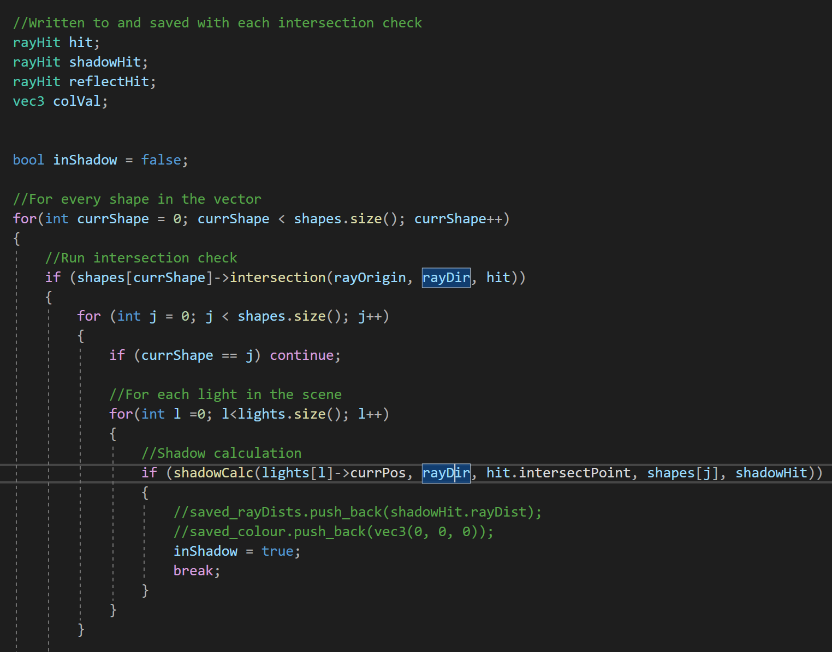
I have implemented the above logic into my code. I start by running the same intersection check for each shape, but now when there is an intersection the code goes through each shape again, firing a ray from the original intersection point. The shadowcalc function called in this section can be seen below:

Figure 19 - I run through the shape array twice to calculate the shadows

Figure 20 - The shadow calculation function

I am reusing the same intersection function for each shape, just with different arguments. If there is a shadow intersection, we break out of the nested loop and set the current pixels colour to black:

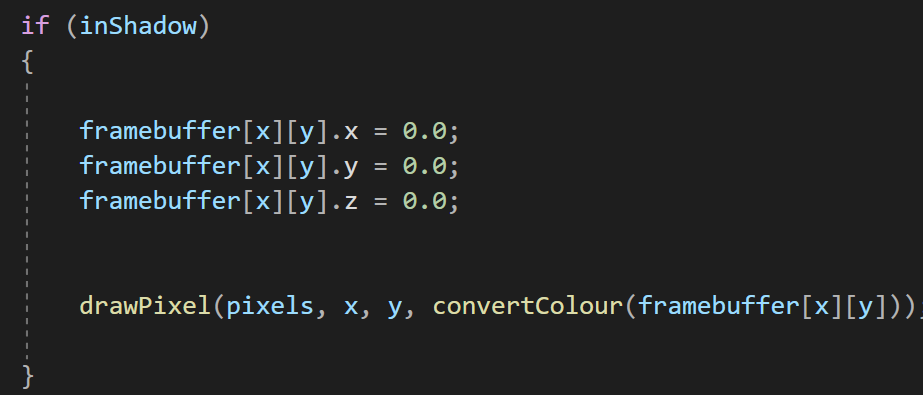
  
  
Otherwise, the pixel is coloured as normal:

Figure 21 - If an object is in shadow its pixels are set to black

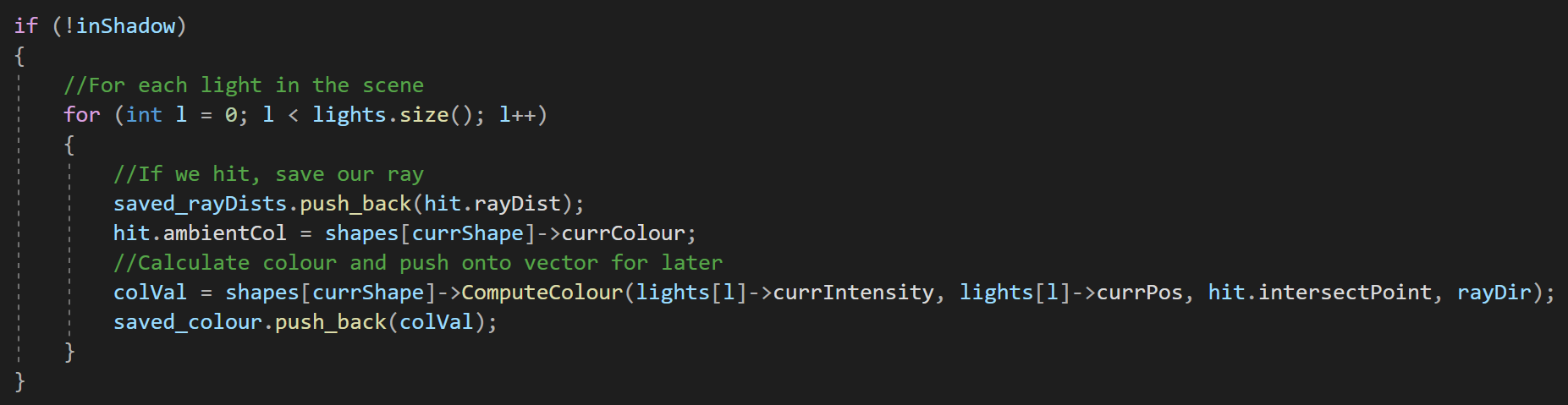


Figure 22 - Otherwise it is calculated normally

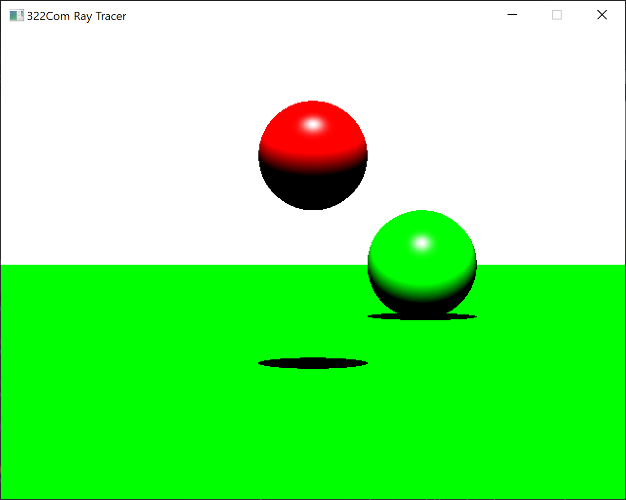


Figure 23 - Working hard shadows

The result from this code is that my shadows are now drawn correctly on the plane.