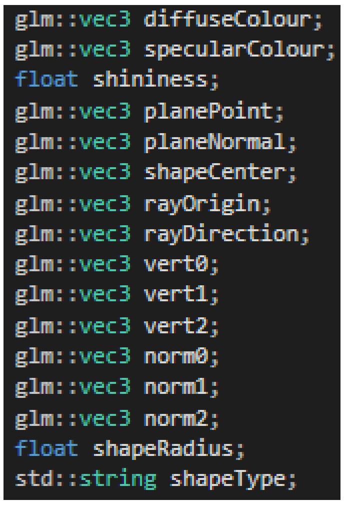
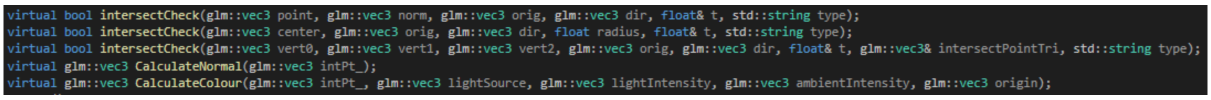
For my ray caster program, I made use of object-oriented programming to create different objects. The three different primitives I have are planes, spheres and triangles. These are all separate classes, which inherit from a base shape class. This shape class contains all the properties that make up an object, such as the centre and radius of a sphere or point and normal of a plane.



```
glm::vec3 diffuseColour;
glm::vec3 specularColour;
float shininess;
glm::vec3 planePoint;
glm::vec3 planeNormal;
glm::vec3 shapeCenter;
glm::vec3 rayOrigin;
glm::vec3 rayDirection;
glm::vec3 vert0;
glm::vec3 vert1;
glm::vec3 vert2;
glm::vec3 norm0;
glm::vec3 norm1;
glm::vec3 norm2;
float shapeRadius;
std::string shapeType;
```

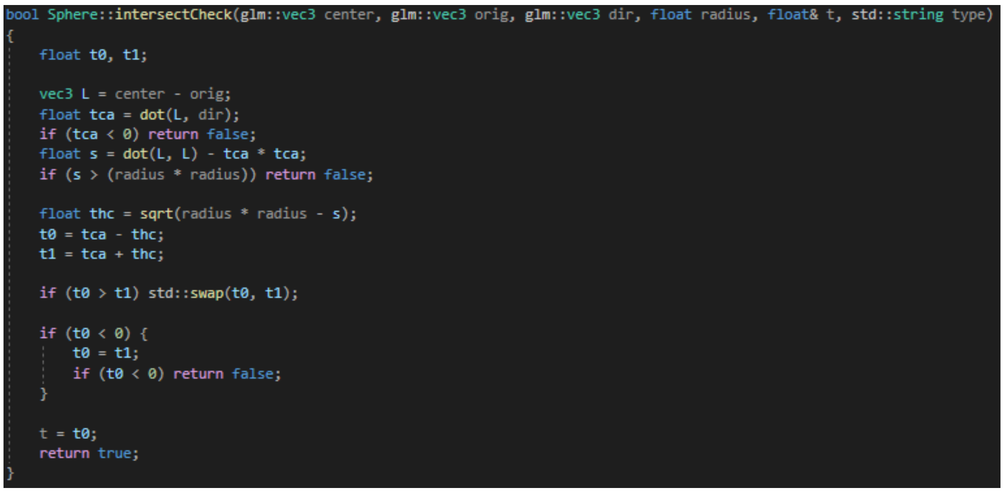
In the future, I will make sure to only have these in the individual shape classes, as it is a waste of memory to initialize many variables that won't be used on the other shapes. The class also contains constructors for the different shapes, as well as functions for intersections tests, calculating normal and calculating colour.



```
virtual bool intersectCheck(glm::vec3 point, glm::vec3 norm, glm::vec3 orig, glm::vec3 dir, float& t, std::string type);
virtual bool intersectCheck(glm::vec3 center, glm::vec3 orig, glm::vec3 dir, float radius, float& t, std::string type);
virtual bool intersectCheck(glm::vec3 vert0, glm::vec3 vert1, glm::vec3 vert2, glm::vec3 orig, glm::vec3 dir, float& t, glm::vec3& intersectPointTri, std::string type);
virtual glm::vec3 CalculateNormal(glm::vec3 intPt_);
virtual glm::vec3 CalculateColour(glm::vec3 intPt_, glm::vec3 lightSource, glm::vec3 lightIntensity, glm::vec3 ambientIntensity, glm::vec3 origin);
```

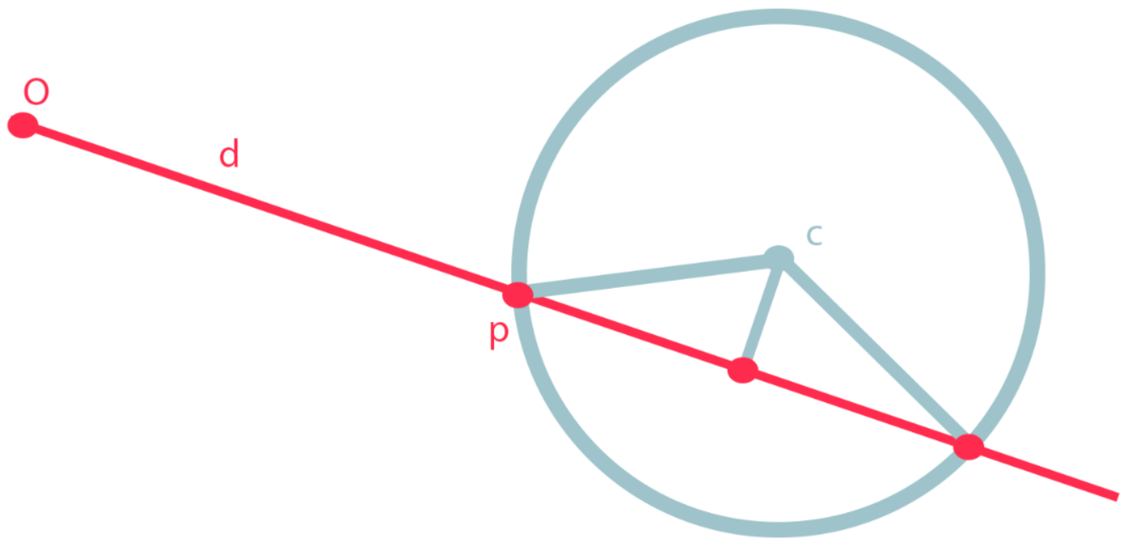
The calculation functions are better than the intersection check functions as all shapes are capable of using the same function. When I first wrote the code, it was necessary for them to use different functions due to different inputs, however they could easily be changed to use the same inputs in the same way as the calculate normal and colour functions do. This would be a small way of improving the efficiency of the program.

The sphere class holds all of the information of the sphere, such as its centre, radius and colour. This allows me to create as many sphere objects as I want with one line of code rather than having to copy and paste the same code several times. It is also capable of calculating an intersection between the sphere and a ray, the normal of the sphere, and phong shading on the sphere.



```
bool Sphere::intersectCheck(glm::vec3 center, glm::vec3 orig, glm::vec3 dir, float radius, float& t, std::string type)
{
    float t0, t1;

    vec3 L = center - orig;
    float tca = dot(L, dir);
    if (tca < 0) return false;
    float s = dot(L, L) - tca * tca;
    if (s > (radius * radius)) return false;

    float thc = sqrt(radius * radius - s);
    t0 = tca - thc;
    t1 = tca + thc;

    if (t0 > t1) std::swap(t0, t1);

    if (t0 < 0) {
        t0 = t1;
        if (t0 < 0) return false;
    }

    t = t0;
    return true;
}
```

This is the function to check for an intersection between a ray with origin ‘orig’ and a direction ‘dir’. This is used for both camera and shadow calculations. Working out the intersection point can be done by using all of the known points to imagine a number of triangles and working out all of their points.



L is the distance from the origin to the centre of the sphere.

To project that onto the ray direction vector, we take the dot product of both. (tca)

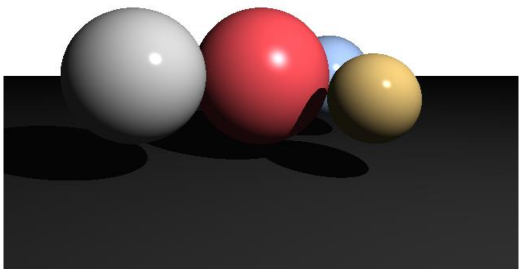
If that value is negative, that means the ray is pointing in a different direction, therefore we return false as there cannot be an intersection.

‘S’ is the distance from the centre of the sphere to the ray, which can be found through the Pythagorean theorem. The same can be done with ‘thc’, which is from that point to the intersection point.

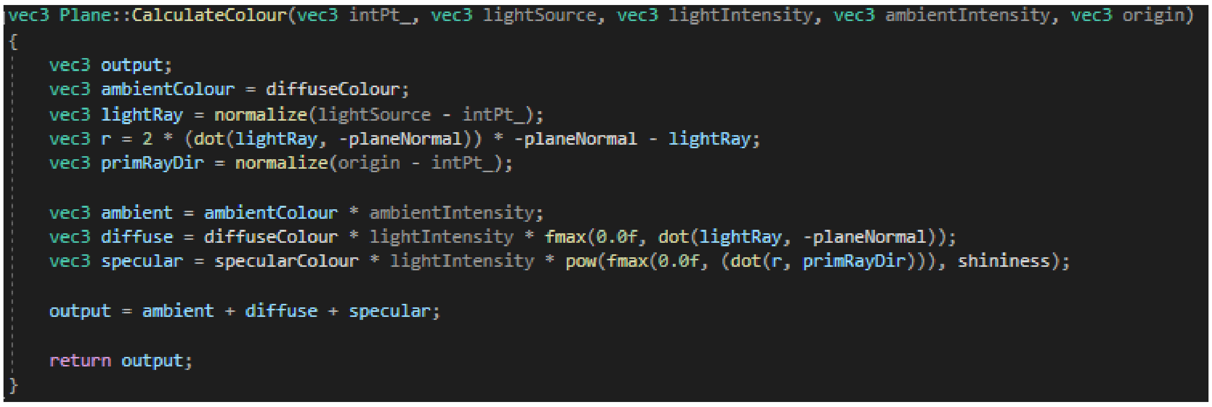
If ‘s’ is larger than ‘r’ (from the centre of the sphere to the last intersection point), then the ray must have missed the sphere and therefore there would be no intersection.

Otherwise, we can work out both intersection points using the formula “t = tca – thc” and “t = tca + thc”, before finally comparing the two intersection points in order to see which one has occurred first.

This ‘t’ value is then used in the main part of the program to figure out which shape should be rendered first.

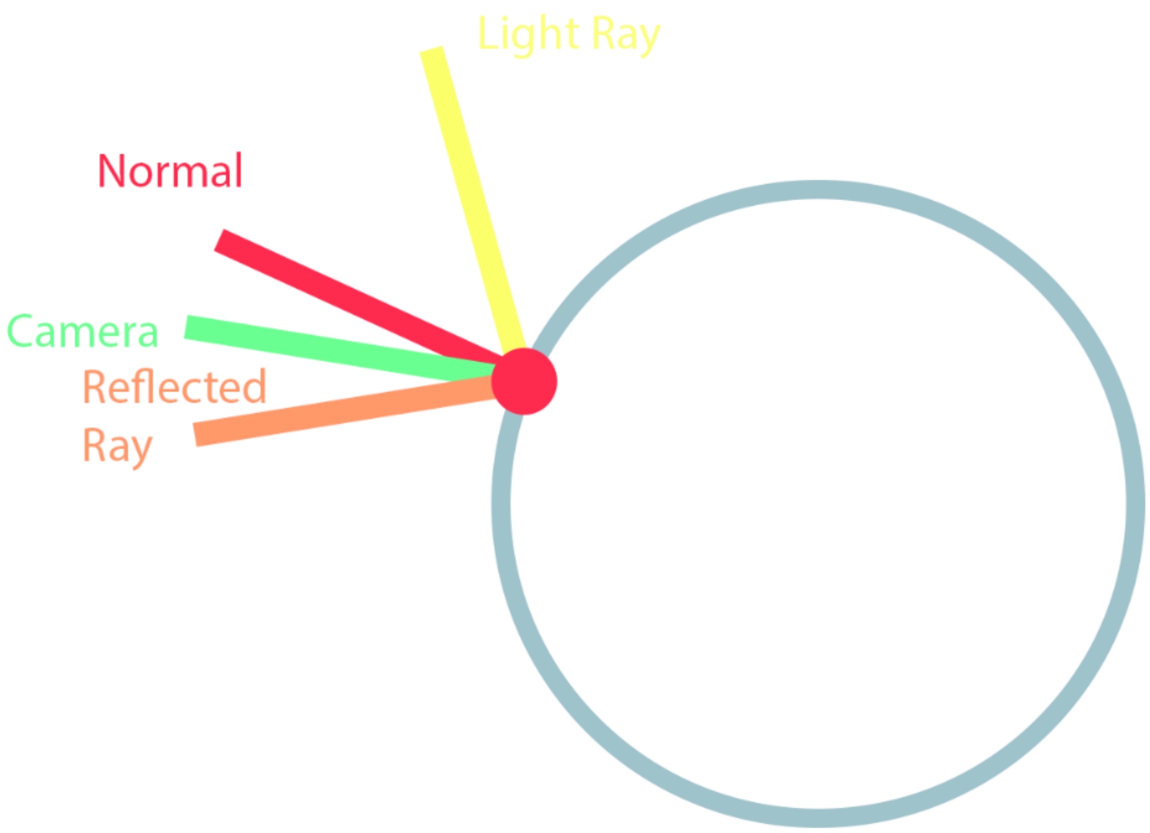


The colour calculation involves phong shading, where the ambient, specular and diffuse intensities are used to figure out how the shape should be shaded. All of the objects use the same calculation, the only difference being normals which are also calculated in each shape class.

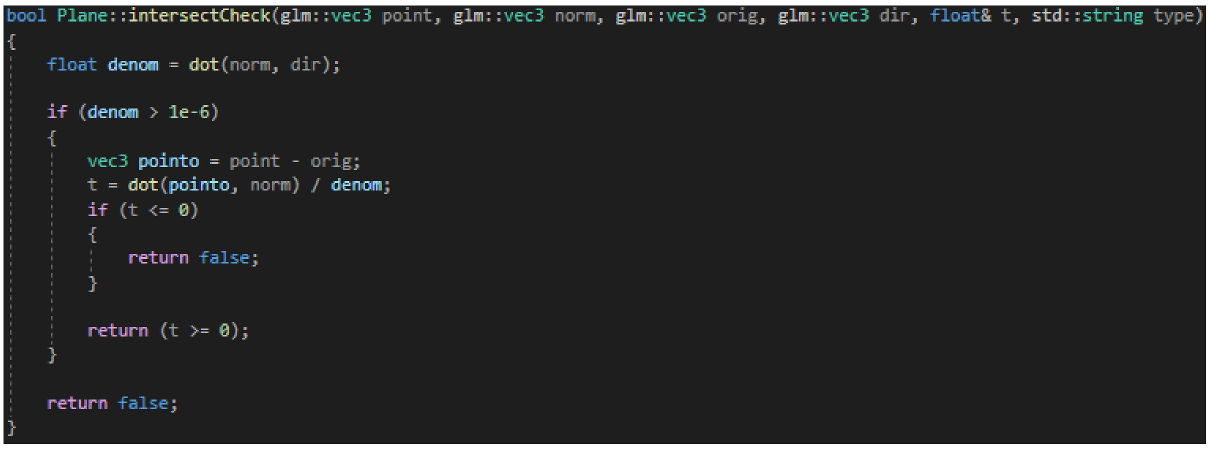


```
vec3 Plane::CalculateColour(vec3 intPt_, vec3 lightSource, vec3 lightIntensity, vec3 ambientIntensity, vec3 origin)
{
    vec3 output;
    vec3 ambientColour = diffuseColour;
    vec3 lightRay = normalize(lightSource - intPt_);
    vec3 r = 2 * (dot(lightRay, -planeNormal)) * -planeNormal - lightRay;
    vec3 primRayDir = normalize(origin - intPt_);

    vec3 ambient = ambientColour * ambientIntensity;
    vec3 diffuse = diffuseColour * lightIntensity * fmax(0.0f, dot(lightRay, -planeNormal));
    vec3 specular = specularColour * lightIntensity * pow(fmax(0.0f, (dot(r, primRayDir))), shininess);

    output = ambient + diffuse + specular;

    return output;
}
```

'PrimRayDir' is a vector towards the camera and 'r' is a reflection of the direction of the light source. Ambient light is the global illumination which can be seen across all objects and consists simply a set intensity value and the standard colour of the object. Diffuse colour is how the angle of light affects the illumination of the object, which is done using the normal. If the normal is facing the light source, it will have maximum illumination, whilst values facing away have none. Any value in between is simply proportional to the cosine of the angle between the light and normal. Finally, specular shading is what causes the bright spot on the surface of the sphere, and uses the dot product of the reflected ray and the camera ray to the power of the shininess value of the object, which is set when the object is created. The overall colour is simply all of these intensities added together.

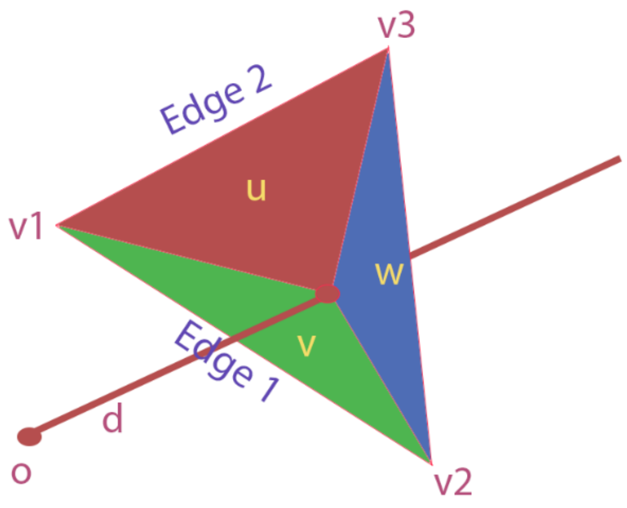


The plane and triangle classes are the same as the sphere class, except they use different intersection methods. The plane intersection method is simple, as a vector that passes through the plane will be perpendicular to its normal.  Therefore, the dot product of the ray vector taken away from the point on the plane and the normal divided by the dot product of the ray and normal is the intersection point on the plane. If this value is close to zero, the ray and the plane must be parallel and therefore there is no intersection.

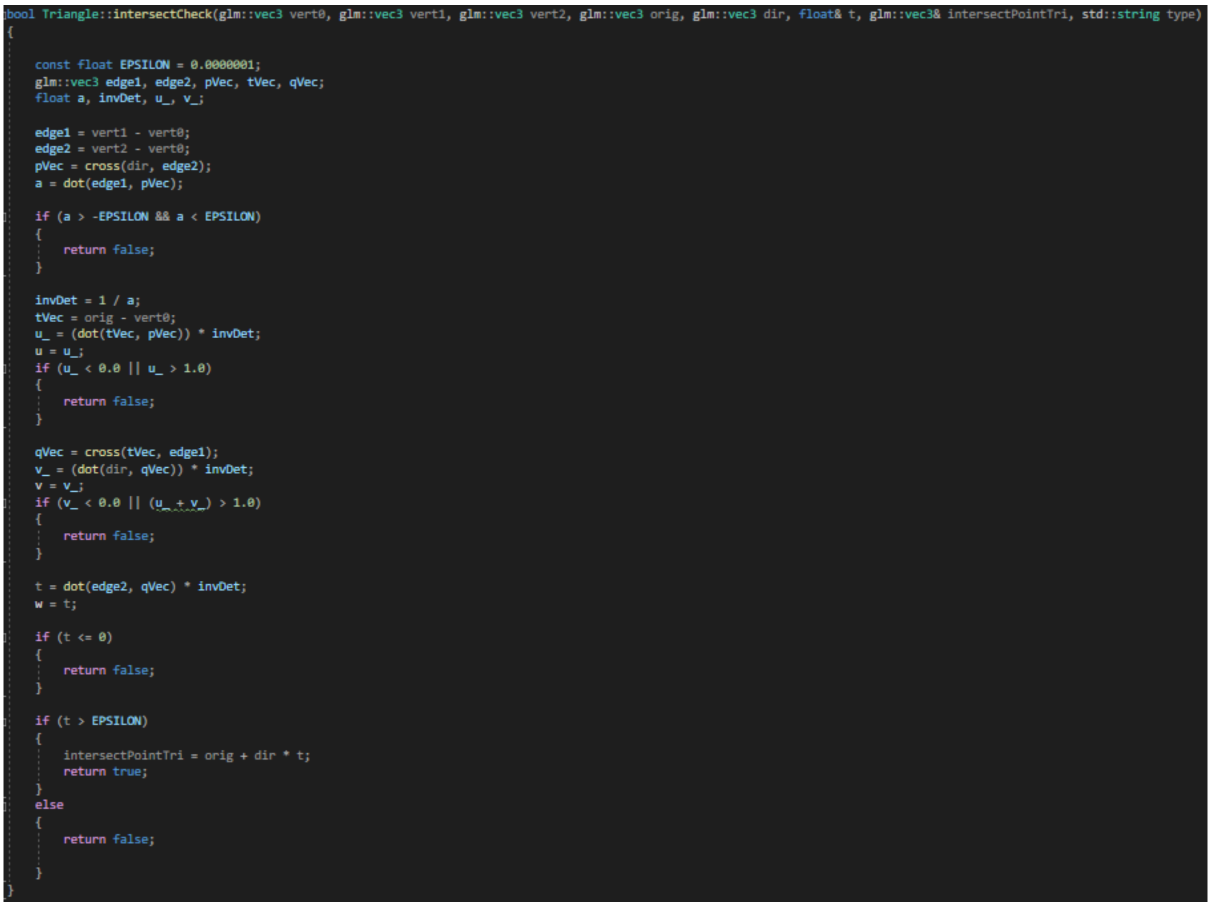


```
bool Plane::intersectCheck(glm::vec3 point, glm::vec3 norm, glm::vec3 orig, glm::vec3 dir, float& t, std::string type)
{
    float denom = dot(norm, dir);

    if (denom > 1e-6)
    {
        vec3 pointo = point - orig;
        t = dot(pointo, norm) / denom;
        if (t <= 0)
        {
            return false;
        }

        return (t >= 0);
    }

    return false;
}
```

Triangle ray intersection is more complicated, and I'm using the Moller-Trumbore method which uses barycentric coordinates and is a fast way of calculating whether an intersection has occurred.



$$\text{Edge 1} = v2 - v1$$
$$\text{Edge 2} = v3 - v1$$
$$u = \frac{(o - v1) \cdot (d \times \text{edge 2})}{\text{edge 1} \cdot (d \times \text{edge 2})}$$
$$v = \frac{d \cdot ((o - v1) \times \text{edge 1})}{\text{edge1} \cdot (d \times \text{edge 2})}$$
$$t = \frac{\text{edge 2} \cdot ((o - a) \times \text{edge1})}{\text{edge 1} \cdot (d \times \text{edge 2})}$$

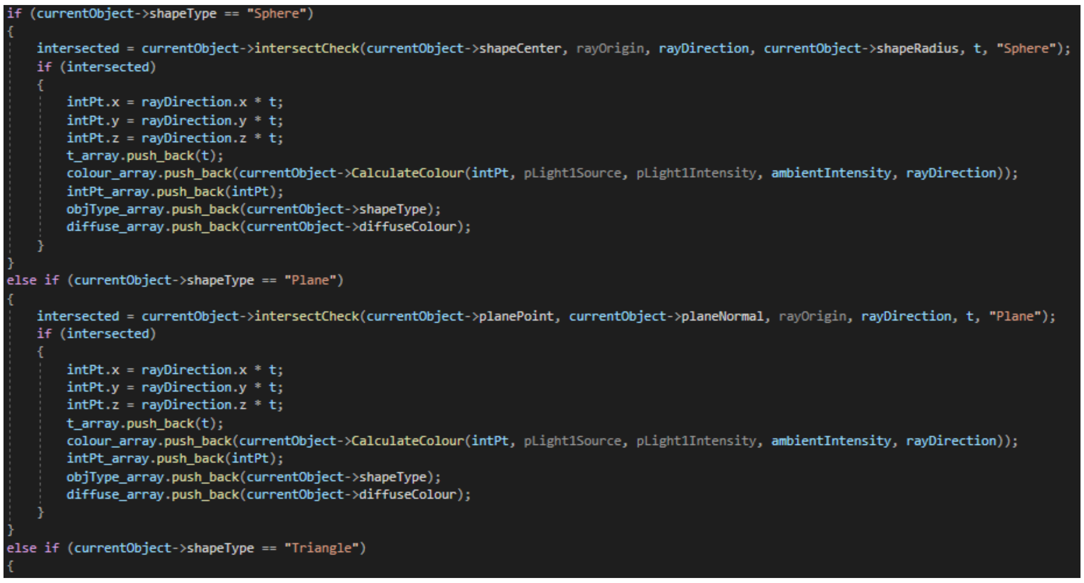


```cpp
bool Triangle::intersectCheck(glm::vec3 vert0, glm::vec3 vert1, glm::vec3 vert2, glm::vec3 orig, glm::vec3 dir, float& t, glm::vec3& intersectPointTri, std::string type)
{

    const float EPSILON = 0.0000001;
    glm::vec3 edge1, edge2, pVec, tVec, qVec;
    float a, invDet, u_, v_;

    edge1 = vert1 - vert0;
    edge2 = vert2 - vert0;
    pVec = cross(dir, edge2);
    a = dot(edge1, pVec);

    if (a > -EPSILON && a < EPSILON)
    {
        return false;
    }

    invDet = 1 / a;
    tVec = orig - vert0;
    u_ = (dot(tVec, pVec)) * invDet;
    u = u_;
    if (u_ < 0.0 || u_ > 1.0)
    {
        return false;
    }

    qVec = cross(tVec, edge1);
    v_ = (dot(dir, qVec)) * invDet;
    v = v_;
    if (v_ < 0.0 || (u_ + v_) > 1.0)
    {
        return false;
    }

    t = dot(edge2, qVec) * invDet;
    w = t;

    if (t <= 0)
    {
        return false;
    }

    if (t > EPSILON)
    {
        intersectPointTri = orig + dir * t;
        return true;
    }
    else
    {
        return false;

    }
}
```

Finally, the triangle class also stores three vertices and three normals, which are used either when creating a new triangle or when making a complex object using the 'objloader'.

The main program itself consists mostly of creating shapes, running the SDL window, checking for keyboard input and running the raycast function. The actual calculations happen in the function 'raycaster()'.

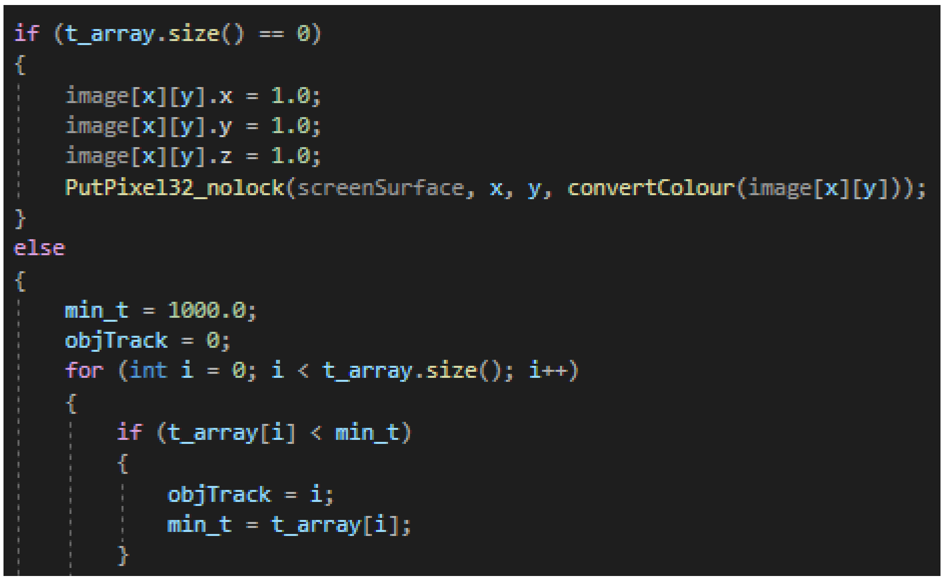
This function is responsible for finding intersections between objects, figuring out which objects go directly in front, calculating whether there are any shadows, and finally computing the colour of the corresponding pixel based on those conditions. This is also the function that is repeated when the user interacts with the light or adds in a new object. All of the intersections are checked by running through an array of shapes and checking for intersections with the current ray. I would have liked to implement bounding boxes to make this check much more effective, as instead of going through every single object in every single pixel I could have simply ran a ray box intersection per object and if a collision was detected then I could run the more complicated intersection test.



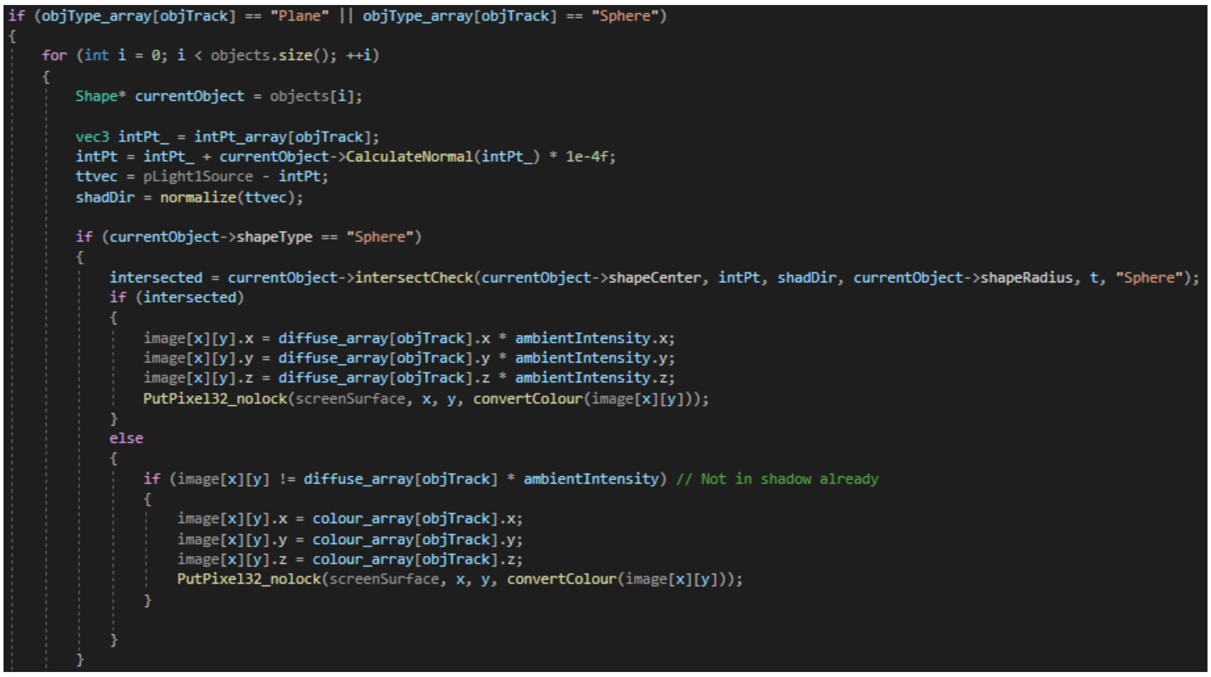
```cpp
if (currentObject->shapeType == "Sphere")
{
    intersected = currentObject->intersectCheck(currentObject->shapeCenter, rayOrigin, rayDirection, currentObject->shapeRadius, t, "Sphere");
    if (intersected)
    {
        intPt.x = rayDirection.x * t;
        intPt.y = rayDirection.y * t;
        intPt.z = rayDirection.z * t;
        t_array.push_back(t);
        colour_array.push_back(currentObject->CalculateColour(intPt, pLight1Source, pLight1Intensity, ambientIntensity, rayDirection));
        intPt_array.push_back(intPt);
        objType_array.push_back(currentObject->shapeType);
        diffuse_array.push_back(currentObject->diffuseColour);
    }
}
else if (currentObject->shapeType == "Plane")
{
    intersected = currentObject->intersectCheck(currentObject->planePoint, currentObject->planeNormal, rayOrigin, rayDirection, t, "Plane");
    if (intersected)
    {
        intPt.x = rayDirection.x * t;
        intPt.y = rayDirection.y * t;
        intPt.z = rayDirection.z * t;
        t_array.push_back(t);
        colour_array.push_back(currentObject->CalculateColour(intPt, pLight1Source, pLight1Intensity, ambientIntensity, rayDirection));
        intPt_array.push_back(intPt);
        objType_array.push_back(currentObject->shapeType);
        diffuse_array.push_back(currentObject->diffuseColour);
    }
}
else if (currentObject->shapeType == "Triangle")
{
```

This code could be reduced by a big margin as I could use the same intersect function and therefore only had one if statement instead of one per type of object. However, when I first wrote the object classes this required me to have a separate function for each shape and therefore there are three ifs per object. This could have been a good way of speeding up the program.

I then compare the intersection points, which are stored in an array, and the smallest value is the one used as the object that pixel can see. This is where the shadow check is run, which does the same intersection test again but using a shadow ray instead of the camera ray. Pixels in the shadow use the ambient light whereas those not in a shadow use phong shading.



```cpp
if (t_array.size() == 0)
{
    image[x][y].x = 1.0;
    image[x][y].y = 1.0;
    image[x][y].z = 1.0;
    PutPixel32_nolock(screenSurface, x, y, convertColour(image[x][y]));
}
else
{
    min_t = 1000.0;
    objTrack = 0;
    for (int i = 0; i < t_array.size(); i++)
    {
        if (t_array[i] < min_t)
        {
            objTrack = i;
            min_t = t_array[i];
        }
```



```
if (objType_array[objTrack] == "Plane" || objType_array[objTrack] == "Sphere")
{
    for (int i = 0; i < objects.size(); ++i)
    {
        Shape* currentObject = objects[i];

        vec3 intPt_ = intPt_array[objTrack];
        intPt = intPt_ + currentObject->CalculateNormal(intPt_) * 1e-4f;
        ttvec = pLight1Source - intPt;
        shadDir = normalize(ttvec);

        if (currentObject->shapeType == "Sphere")
        {
            intersected = currentObject->intersectCheck(currentObject->shapeCenter, intPt, shadDir, currentObject->shapeRadius, t, "Sphere");
            if (intersected)
            {
                image[x][y].x = diffuse_array[objTrack].x * ambientIntensity.x;
                image[x][y].y = diffuse_array[objTrack].y * ambientIntensity.y;
                image[x][y].z = diffuse_array[objTrack].z * ambientIntensity.z;
                PutPixel32_nolock(screenSurface, x, y, convertColour(image[x][y]));
            }
            else
            {
                if (image[x][y] != diffuse_array[objTrack] * ambientIntensity) // Not in shadow already
                {
                    image[x][y].x = colour_array[objTrack].x;
                    image[x][y].y = colour_array[objTrack].y;
                    image[x][y].z = colour_array[objTrack].z;
                    PutPixel32_nolock(screenSurface, x, y, convertColour(image[x][y]));
                }

            }
        }
```

The things I would have liked to have added are reflections, soft shadows and bounding boxes. Reflections would require a recursive function which would take a ray as an input and pass that ray on further to figure out where the reflected ray is going. Meanwhile, soft shadows require an area light instead of a point light like the one I have, which has several points coming from the light to generate a penumbra on either side of the shadow in order to make it smoother. In the end, I was only limited by how much time I gave myself to work on the project however I am generally pleased with the results that I have achieved.