I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Signature – Ruari McGhee

Coursework

Graphics Programming – m3i622944

Ruari McGhee – S1432402

Table of Contents

[Introduction 2](#_Toc40260810)

[Background 2](#_Toc40260811)

[Compute Shaders 2](#_Toc40260812)

[Compute Shader Setup 2](#_Toc40260813)

[Texture 2](#_Toc40260814)

[Shader 3](#_Toc40260815)

[RayMarching Shader 4](#_Toc40260816)

[Raycasting 4](#_Toc40260817)

[Distance Functions 4](#_Toc40260818)

[Constructive Solid Geometry 5](#_Toc40260819)

[Lighting 6](#_Toc40260820)

[Normals 7](#_Toc40260821)

[Rendering 9](#_Toc40260822)

# Introduction

## Background

For this project’s third shader the aim is to experiment with implementing raymarching into the scene. Raymarching works with primitive shapes to combine them and produce new and interesting shapes based on the unions, subtractions and intersections that can occur when these primitives overlap each other.

Raymarching works differently to how the other objects in the project are being handled. Instead of taking the object’s vertexes, transforming them to screen space having the rasterization step check what pixels are being covered. The process instead aims to fire out rays through the scene for each pixel on the screen to check if that pixel position will be covering a viewable object.

To enable this process to work raymarching instead uses compute shaders as it writes its output directly to a texture that can be rendered onto the screen by some other means.

## Compute Shaders

Unlike the other shaders used in the labs and the other parts of the coursework, Compute shaders are not part of the regular shader pipeline, they are a separate shader stage that is largely defined by the user through the binding of their input information and the binding of the output information (usually handled by writing to textures). Due to this there is a lot of freedom in what they can do.

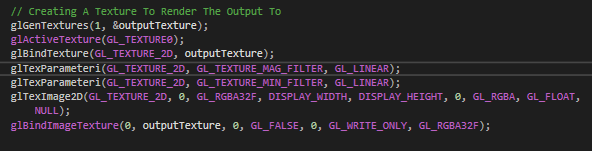
Compute shaders work in groups that are defined by the user. When being instructed to run the user sets how many work groups that they want to happen. Since this runs on the graphics card by having a work group that is the same size as the window/ output texture size, each pixel being calculated can be handled in parallel and thus the output creation can be much faster.

- Note – The use of compute shaders requires OpenGL 4.3 or above to be used.

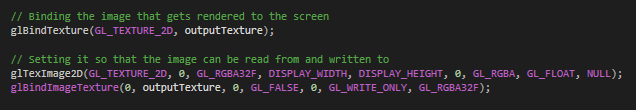
## Compute Shader Setup

### Texture

Instead of rendering a model to the screen the compute shader instead calculates if a pixel on the screen is overlapping an object in the scene. To store this output a texture is created at runtime. This texture is created to have the same size as the game window so that each pixel in the game window has its own location to write an output to from the compute shader.



Unlike the other textures used in the project, when getting bound the texture is set to be an Image2D datatype. This is so that the texture can be both read from and written to.



### Shader

The compute shader is loaded up like any other shader in the project, however for the shader to properly load the compute shader must be the only shader in the program. It cannot be part of another fragment and vertex shader’s program (or any other shader that is part of the regular shader pipeline).

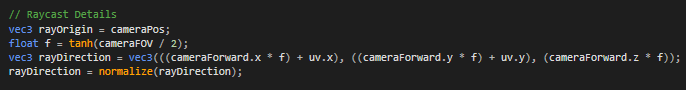
# RayMarching Shader

## Raycasting

To ensure that the rays are being fired out in the correct direction the camera has a direction vector hardcoded into itself that signifies the way that it is facing. For this project it represents that the camera is tilted approximately 20 degrees downwards. This is passed to the compute shader along with the camera’s position in the scene.

This is because each ray fired originates from the camera’s position however the direction that the ray Is fired out needs to be slightly altered depending on what pixel on the screen it is representing. If each ray was just fired out with the camera’s forward direction, then the raycasting and intersection testing would work but the results would give back an orthographic view of the world. To get the rays firing out with the world’s perspective view a small offset needs to be added to each ray direction.

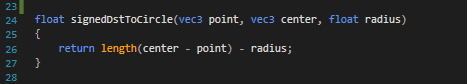
OpenGL makes this calculation easy since the window coordinates ranges from (-1, -1) to (1, 1). By passing through the camera’s FOV the needed offset becomes a simple trigonometry equation.

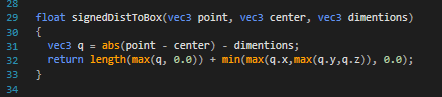


– Note – Originally the FOV given to the camera is 70 degrees but for OpenGL to correctly initialise the perspective projection for the camera this must be converted into radians

## Distance Functions

To calculate if a ray has hit an object in the scene the ray can get the distance to an object’s surface. To calculate we use distance functions. Each type of primitive shape have their own distance function. Some basic examples can be shown by a sphere’s distance function and a cube’s distance function.

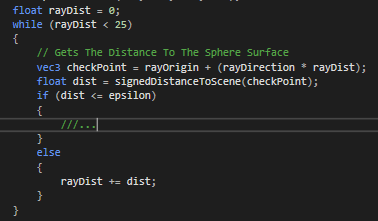




An expanded list of distance functions for other primitive shapes can be found at: <https://iquilezles.org/www/articles/distfunctions/distfunctions.htm>.

The result of the distance function shows if the checked point is inside or outside of the shape. If the result of the distance function is greater than 0 then it means that the point is outside of the object, if it is equal to zero then the point lies on the surface of the object and if it is negative then the point exists inside of the checked object.

If the result is greater than 1 then the ray marches forward using its direction. The amount it marches forward is defined by what was the closest hit surface from the distance function. A max march distance also needs to be defined so that rays do not continue to infinity.



## Constructive Solid Geometry

When there are multiple objects in the scene it is possible to use the distance functions for each primitive to combine geometry. This can easily be done for primitive union, intersection, and subtraction. Instead of checking one object at a time for the ray’s intersection by checking all of them at the same time and manipulating the returned distance value these different effects can be achieved.

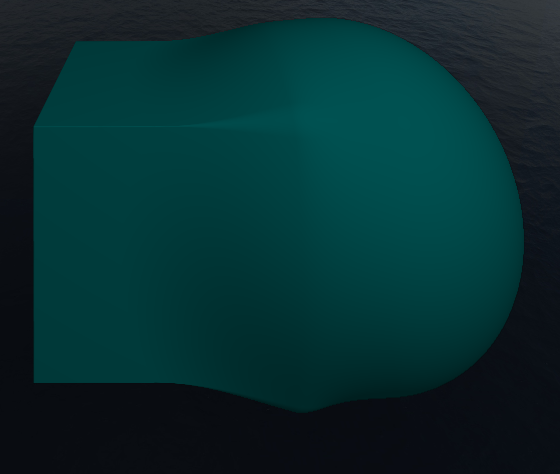
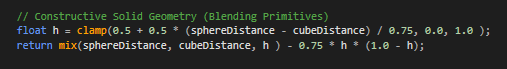
#### Union Example

#### Subtraction Example



#### Intersection Example

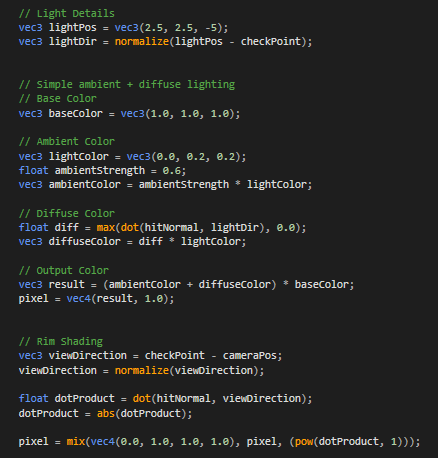


However, by messing around with the distance function comparison between the two primitive objects some more interesting outputs can be generated. The following example is similar to primitive union however the output appears a more organic/ blended version.

## Lighting

If the returned distance from the distance functions is less than a required value, then that pixel on the screen can be assumed to be covering the object. This can be handled very similarly to how fragment shaders work except that instead of setting the screen’s pixel color it is setting the texture’s pixel color.

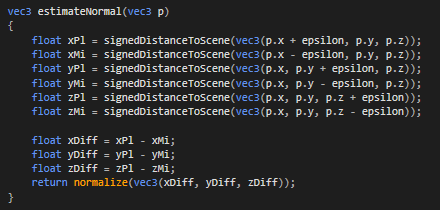
Due to this it is very easy to implement some of the graphical techniques from the previous lab shaders. For example, by implementing the same logic from the rim shader lab a light hue can be given to the output’s edges and simple ambient and diffuse lighting can be created on the output by giving a light location.



## Normals

The trickiest part of this lighting implementation comes from getting the hit normal. Unlike the regular mesh rendering process where the normals are stored in the model this method of normal calculation relies on approximating the gradient of the hit surface. This is mostly due to the fact that the constructive solid geometry can produce results that would exclude the use of regular primitive surface normal calculations

This is done by finding out where the hit surface point is, approximating where along the raycast the point would be outside of the object and then approximating where along the raycast the point would be inside of the object. Or in other words getting three points where the distance function would return values approximately equal to zero, a value greater than zero and a value less than zero.

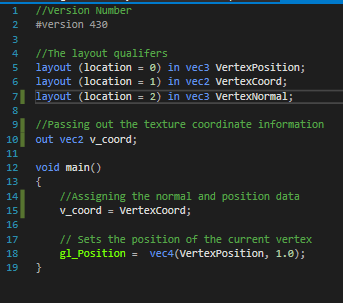
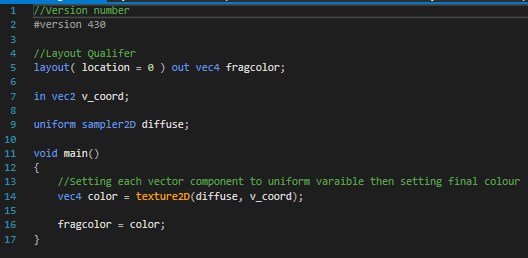


These three points can be used to calculate a direction vector that when normalized can be used as the hit surface’s normal.

#### Example Final Output

# Rendering

Since the output is saved as a texture this can be rendered to the screen on a simple plane in front of the camera. A plane model is loaded in from the project’s resources like how other models in the scene are loaded in. This plane has its vertexes resemble the coordinate system that OpenGL uses for its window, the top right corner is at (1, 1) and the bottom left corner is at (-1, -1).

Because of this no transformation needs to take place in a vertex shader and the fragment shader only needs to assign the pixel color from the texture to the same pixel on the screen.