

OCR

OpenGL Real-Time Raymarching

Coursework

Ben Barton-Foskett

Contents

Analysis.....	2
Introduction	2
Reduction to Triangles.....	2
Research	2
Application	8
Signed Distance Function (SDF) Rendering	11
Research	11
Application	18
Lighting and Reflections	22
Research	22
Application	25
Rigid Body Dynamics.....	30
Research	30
Application	31
Modelling.....	32
Introduction.....	32
Scenes	32
The Graphical User Interface.....	33
Design.....	34
Introduction	34
External Libraries	34
System Overview	35
Algorithms and Modules.....	39
Classes	39
Other Files	55

Analysis

Introduction

For my project, I am going to make a 3D raymarching engine, taking some inspiration from how Blender deals with their rendering. When this project is completed, I want to have a 3D simulated environment in which objects can be added and different material properties of the object can be changed in real time, with some physics as well.

To achieve this, I would need to use graphics in such a way that they can be altered without using too much memory. As I am dealing with graphics at a large realistic 3D level, it would be way more efficient to make use of the graphics card to handle rendering rather than normal memory. If I want the objects and their properties to be changed while the program is running, it would be wise to use an object-oriented approach. Some classes that I may use would include:

- Buffer objects, to handle sending data to the graphics card.
- Vertex objects, to handle where on the screen polygon vertices should be rendered.
- Variable handling, to dynamically control and change variables, and communicating with the buffer objects.
- GUI class, to handle user input data and communicate with the variable handling class(es).

Reduction to Triangles

Research

OpenGL Setup

For this method, I researched on how to set up the Open Graphics Library (OpenGL) environment and how to use the GL Utility Toolkit (GLUT) library of OpenGL. I found a website which taught me how to create a basic OpenGL GLUT window. A website on OpenGL¹, a screenshot of which is shown below, explained how to use the initial basic functions for creating a window with GLUT, which are as follows:

- `glutInit(&argc, argv);`
This initialises the GLUT environment. The variables required could be retrieved from the main method of C++ specifying them as arguments.
- `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);`

¹ Ogldev.org. (2022). *Tutorial 01 - Create a window*. [online] Available at: <https://ogldev.org/www/tutorial01/tutorial01.html>.

This defines the mode in which the window opens in that I decided to use GLUT_SINGLE rather than GLUT_DOUBLE because I did not feel I needed double buffering, as it could be harder to work with.

- `glutInitWindowSize(width, height);`
This initialises the window size to a predetermined width and height.
- `glutInitWindowPosition(x, y);`
This specifies where on the monitor the screen should appear.
- `glutCreateWindow("title");`
This renders the window on the screen with a specified title for the argument.
- `glClearColor(r, g, b, a);`
This sets the background colour for the screen, where each RGB component is a float value between 0.0 and 1.0. So, to convert from the traditional 0-255 scale to the 0.0-1.0 scale, you divide by 255.
- `glutMainLoop();`
This allows the OpenGL environment to be run by GLUT.
- `glClear(GL_COLOR_BUFFER_BIT);`
This refreshes the colour buffer, which sends information about pixel colours to the graphics card to render.
- `glutSwapBuffers();`
This calls the refresh of the buffers specified in `glClear()`.

```
glutInit(&argc, argv);
```

This call initializes GLUT. The parameters can be provided directly from the command line and include useful options such as '-sync' and '-gldebug' which disable the asynchronous nature of X and automatically checks for GL errors and displays them (respectively).

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
```

Here we configure some GLUT options. GLUT_DOUBLE enables double buffering (drawing to a background buffer while another buffer is displayed) and the color buffer where most rendering ends up (i.e. the screen). We will usually want these two as well as other options which we will see later.

May-29, 2022: One of the viewers of my youtube channel informed me that he wasn't able to activate the depth test unless GLUT_DEPTH was explicitly added to the above call. On my machine it's working correctly even without this flag but I've decided to add it across the entire code base just in case. We will talk about depth testing later on so for now just add this flag until we get there.

```
glutInitWindowSize(1024, 768);  
glutInitWindowPosition(100, 100);  
glutCreateWindow("Tutorial 01");
```

These calls specify the window parameters and create it. You also have the option to specify the window title.

```
glutDisplayFunc(RenderSceneCB);
```

Since we are working in a windowing system most of the interaction with the running program occurs via event callback functions. GLUT takes care of interacting with the underlying windowing system and provides us with a few callback options. Here we use just one - a "main" callback to do all the rendering of one frame. This function is continuously called by GLUT internal loop.

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

This is our first encounter with the concept of state in OpenGL. The idea behind state is that rendering is such a complex task that it cannot be treated as a function call that receives a few parameters (and correctly designed functions never receive a lot of parameters). You need to specify shaders, buffers and various flags that affect how

Drawing Vertices

Next, I researched how I would go about rendering on the screen and found the methods `glVertex3f()`² and `glColor3f()`² from websites, where some screenshots are shown below. `glVertex3f()` renders a single vertex at a point on the screen, given by three floats: x, y, and z. `glColor3f()` is called before `glVertex3f()`, as it defines the colour of the vertex about to be rendered in 0.0-1.0 RGB float values.

² Docs.gl. (2024). docs.gl. [online] Available at: <https://docs.gl> [Accessed 19 Aug. 2024].

docs.GL

search

Go

OpenGL 3.1

☐ Hide unavailable

- Textures
- Rendering
- Frame Buffers
- Shaders
- Buffer Objects
- State Management
- Transform Feedback
- Utility
- Queries
- Syncing
- Vertex Array Objects
- Samplers
- Program Pipelines
- Immediate Mode
 - glArrayElement
 - glBegin
 - glColor
 - glEnd
 - glEvalCoord
 - glEvalMesh
 - glEvalPoint
 - glFogCoord
 - glIndex
 - glMaterial
 - glMultiTexCoord
 - glNormal
 - glSecondaryColor
 - glTexCoord
 - glVertex
- GL2 Rasterization
- Client Arrays
- Fixed Function
- Matrix State
- GL2 Textures
- Call Lists

glVertex

OpenGL 3

OpenGL 2

glVertex — specify a vertex

C Specification

```

void glVertex2s(GLshortx,
               GLshorty);
void glVertex2i(GLintx,
               GLinty);
void glVertex2f(GLfloatx,
               GLfloaty);
void glVertex2d(GLdoublex,
               GLdoubley);
void glVertex3s(GLshortx,
               GLshorty,
               GLshortz);
void glVertex3i(GLintx,
               GLinty,
               GLintz);
void glVertex3f(GLfloatx,
               GLfloaty,
               GLfloatz);
void glVertex3d(GLdoublex,
               GLdoubley,
               GLdoublez);
void glVertex4s(GLshortx,
               GLshorty,
               GLshortz,
               GLshortw);
void glVertex4i(GLintx,
               GLinty,
               GLintz,
               GLintw);
void glVertex4f(GLfloatx,
               GLfloaty,
               GLfloatz,
               GLfloatw);

```

docs.GL

search

Go

OpenGL 3.1

☐ Hide unavailable

- Textures
- Rendering
- Frame Buffers
- Shaders
- Buffer Objects
- State Management
- Transform Feedback
- Utility
- Queries
- Syncing
- Vertex Array Objects
- Samplers
- Program Pipelines
- Immediate Mode
 - glArrayElement
 - glBegin
 - glColor
 - glEnd
 - glEvalCoord
 - glEvalMesh
 - glEvalPoint
 - glFogCoord
 - glIndex
 - glMaterial
 - glMultiTexCoord
 - glNormal
 - glSecondaryColor
 - glTexCoord
 - glVertex
- GL2 Rasterization
- Client Arrays
- Fixed Function
- Matrix State
- GL2 Textures
- Call Lists

glColor

OpenGL 3

OpenGL 2

glColor — set the current color

C Specification

```

void glColor3b(GLbytered,
              GLbytegreen,
              GLbyteblue);
void glColor3s(GLshortred,
              GLshortgreen,
              GLshortblue);
void glColor3i(GLintred,
              GLintgreen,
              GLintblue);
void glColor3f(GLfloatred,
              GLfloatgreen,
              GLfloatblue);
void glColor3d(GLdoublered,
              GLdoublegreen,
              GLdoubleblue);
void glColor3ub(GLbytered,
              GLbytegreen,
              GLbyteblue);
void glColor3us(GLushortred,
              GLushortgreen,
              GLushortblue);
void glColor3ui(GLuintred,
              GLuintgreen,
              GLuintblue);
void glColor4b(GLbytered,
              GLbytegreen,
              GLbyteblue,
              GLbytealpha);
void glColor4s(GLshortred,
              GLshortgreen,
              GLshortblue,
              GLshortalpha);

```

Affine Matrices

Next, I needed a way to alter the positions of vertices on the screen, so I researched 3D affine matrix transformations. I found a particularly useful website³ which had the matrices I needed. The matrices I needed were the rotation and translation matrices.

The matrix for a rotation about the x-axis is as follows:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which, when applied to a vertex, represented as a position vector $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$, gives the result:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot 1 + y \cdot 0 + z \cdot 0 + 1 \cdot 0 \\ x \cdot 0 + y \cdot \cos(\theta) + z \cdot \sin(\theta) + 1 \cdot 0 \\ x \cdot 0 + y \cdot -\sin(\theta) + z \cdot \cos(\theta) + 1 \cdot 0 \\ x \cdot 0 + y \cdot 0 + z \cdot 0 + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} x \\ y \cdot \cos(\theta) + z \cdot \sin(\theta) \\ -y \cdot \sin(\theta) + z \cdot \cos(\theta) \\ 1 \end{pmatrix}$$

And comparable results come from the other two rotation matrices:

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Respectively giving:

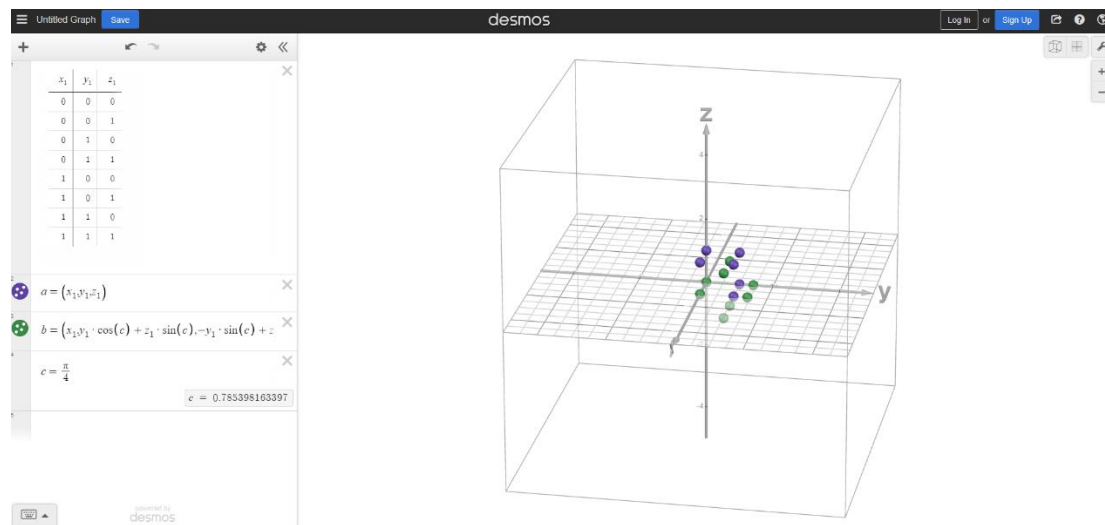
³ www.brainvoyager.com. (n.d.). *Spatial Transformation Matrices*. [online] Available at: <https://www.brainvoyager.com/bv/doc/UsersGuide/CoordsAndTransforms/SpatialTransformationMatrices.html>.

$$\begin{pmatrix} x \cdot \cos(\theta) - z \cdot \sin(\theta) \\ y \\ x \cdot \sin(\theta) + z \cdot \cos(\theta) \\ 1 \end{pmatrix}$$

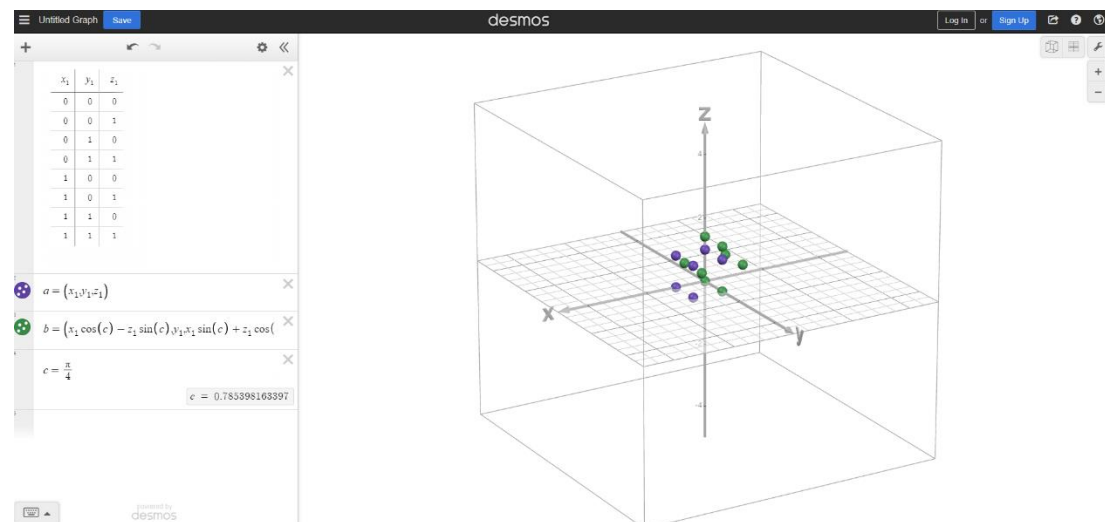
$$\begin{pmatrix} x \cdot \cos(\theta) - y \cdot \sin(\theta) \\ x \cdot \sin(\theta) + y \cdot \cos(\theta) \\ z \\ 1 \end{pmatrix}$$

Applying this to a unit cube in Desmos gives the following results:

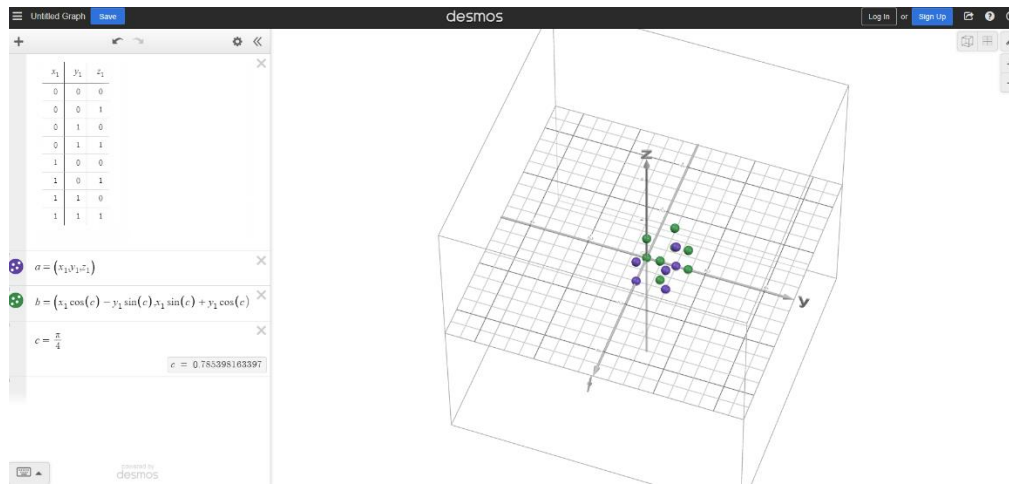
x-axis:



y-axis:



z-axis:



The dots are being rotated about the respective axes, so the rotation matrices are functional.

Finally, due to the rotation matrices, I am limited to a perspective projection rather than an orthographic projection because these matrices are designed to be used in a perspective projection. However, this is not a concern because my initial plan was to use a perspective projection as it seems more realistic than an orthographic projection.

Application

Perspective Projection

My first thought for implementing this would be to use an approach which would simplify a 3D shape into many triangles, which would have three vertices, and could be rendered and dynamically altered by joining up the vertices of the triangles. This would be achievable by representing each vertex as a position vector in 3D space, then applying a perspective projection to it, as shown in Fig. 1.

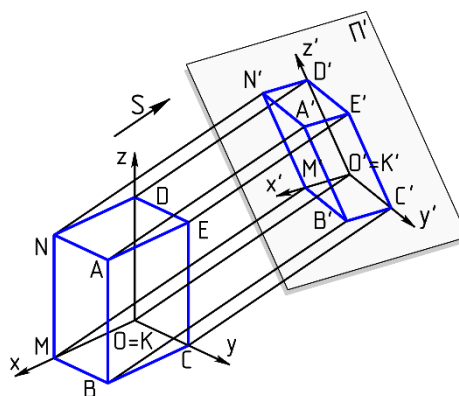


Fig. 1

Calculating this is surprisingly simple. Let a point A in 3D space be defined as the coordinate (x, y, z) , and A' be the projection of A with altered x and y values, and $z = 0$. Let CAM be the position of the camera defined as $(0, 0, -DEPTH)$, where DEPTH is the distance between the camera and the screen, and let C be the point where CAM is

normal to the screen. Looking top down so the only factor being changed is x , we get Fig. 2

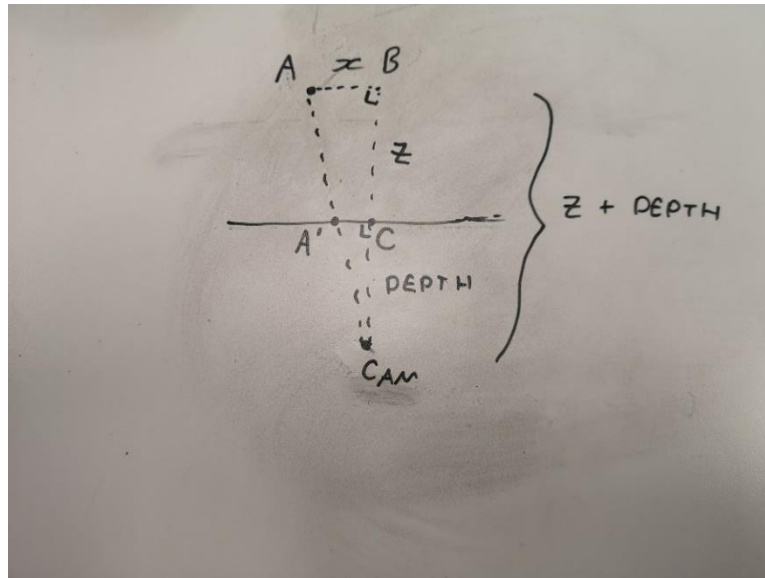


Fig. 2

The length of AB will be x , the length of CAM to C will be DEPTH, and the length of CB will be z , and the angles at B and C will be right angles. This leads to two similar triangles which can be seen in Fig. 3, and the same logic can be applied for the y direction.

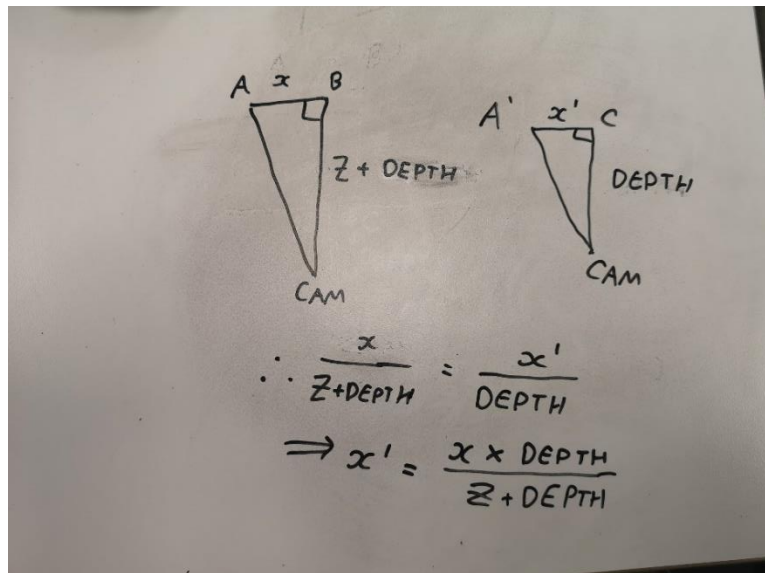


Fig. 3

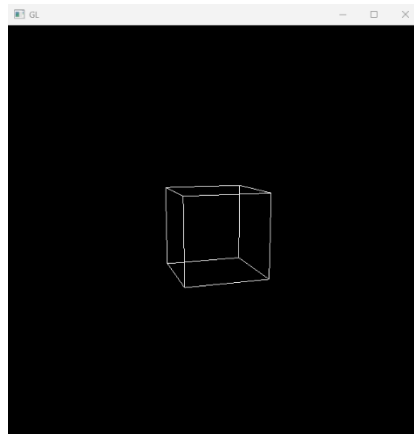
This gives that the equation of a projected coordinate from 3D space to $z = 0$ is:

$$A' = \left(\frac{x \cdot \text{DEPTH}}{z + \text{DEPTH}}, \frac{y \cdot \text{DEPTH}}{z + \text{DEPTH}}, 0 \right)$$

This can be applied to a code extract shown in OpenGL:

```
Vertex Vertex::getProjectedVertex() {
    return Vertex(DEPTH*(x)/(DEPTH+z),DEPTH*(y)/(DEPTH+y),0);
}
```

Here, Vertex is a class with Vertex attributes and the getProjectedVertex() method returns a new Vertex with properties from the previously derived equation. Applying this gives this result:



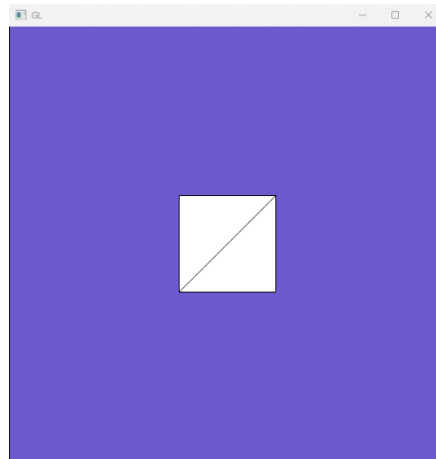
So, as we can see depth, the perspective projection is working.

Rotation Matrices

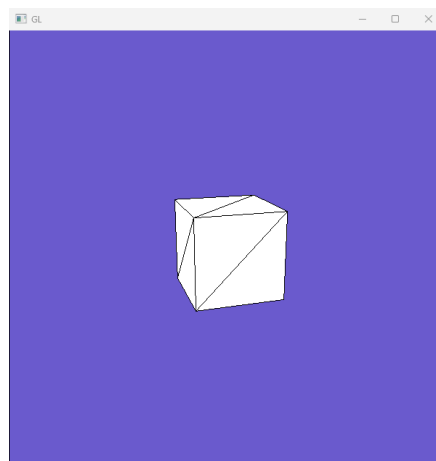
The rotation matrices mentioned in the research section can be applied to this projection now to allow for rotations in real time. Below is an example header class for an affine matrix class in C++:

```
#pragma once
#include <vector>
#include <cmath>
class AffineMatrix {
private:
    std::vector<std::vector<double>> mat;
public:
    AffineMatrix();
    virtual void scale(double scalar);
    virtual AffineMatrix add(AffineMatrix m);
    virtual AffineMatrix multiply(AffineMatrix m);
    virtual AffineMatrix inverse();
    virtual double atPos(int r, int c);
};
```

Applying these transformations gives these results, where the cube is initially facing the screen:



Then, when I move the camera left three times, and up three times, we get this:



So, the matrix transformations have been applied correctly.

However, this method is very memory inefficient, as the inverse of each transform and rotation matrix needs to be applied to the position of CAM every frame, so in practice this method was found to be not the best. But my efforts to go about doing it this way were not wasted, because I gained valuable knowledge surrounding programming in OpenGL, affine matrix transformations and the perspective projection.

Signed Distance Function (SDF) Rendering

Research

Signed Distance Functions

I discovered signed distance functions (also known as signed distance fields) when researching how to add lighting to my scene and found that they were incredibly more efficient than reducing a shape to triangles. I was researching raymarching and came

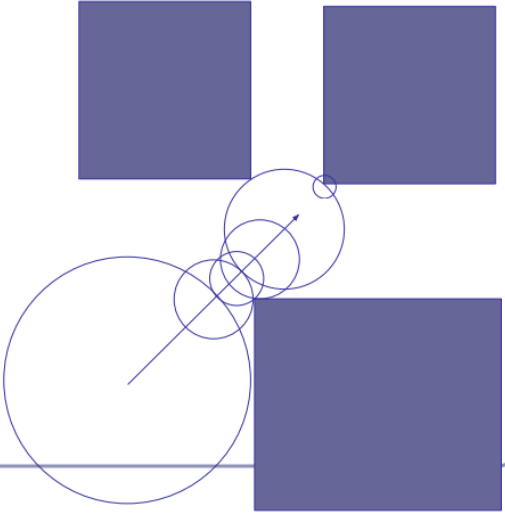
across a University of Cambridge PowerPoint on signed distance functions (SDFs)⁴, where a slide of this is shown below.

GPU Ray-marching: Signed Distance Fields

Ray-marching can be dramatically improved, to impressive realtime GPU performance, using *signed distance fields*:

1. Fire ray into scene
2. At each step, measure distance field function: $d(p) = [\text{distance to nearest object in scene}]$
3. Advance ray along ray heading by distance d , because the nearest intersection can be no closer than d

This is also sometimes called 'sphere tracing'. Early paper: <http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf>



The term “Signed Distance Function” neatly describes exactly their purpose. An SDF returns a positive number if a point is inside the shape, and negative if it is outside, meaning shapes that are harder to represent as triangles can more easily be rendered on the screen, such as spheres. Using a sphere as an example, the derivation is below.

The cartesian equation for a sphere is:

$$\sqrt{x^2 + y^2 + z^2} = r$$

Which, in terms of a position vector \vec{p} is

$$\|\vec{p}\|$$

Therefore letting r be the radius of the sphere, if $\|\vec{p}\| - r < 0$, the point is outside the circle, but if $\|\vec{p}\| - r > 0$, the point is inside the circle. In GLSL, this can be written as:

```
float sphereSDF(vec3 p, vec3 c, float r) {
    return length(c-p) - r;
}
```

⁴ Benton, A. (n.d.). *GPU Ray Marching*. [online] Available at: <https://www.cl.cam.ac.uk/teaching/1718/AdvGraph/5.%20GPU%20Ray%20Marching.pdf>.

Where c is the centre, p is a point and r is the radius.

Raymarching

To render an SDF, a technique called raymarching needs to be used. Again, from the Cambridge PowerPoint, we can generate a flowchart on how raymarching works, generating a red sphere on a black background:

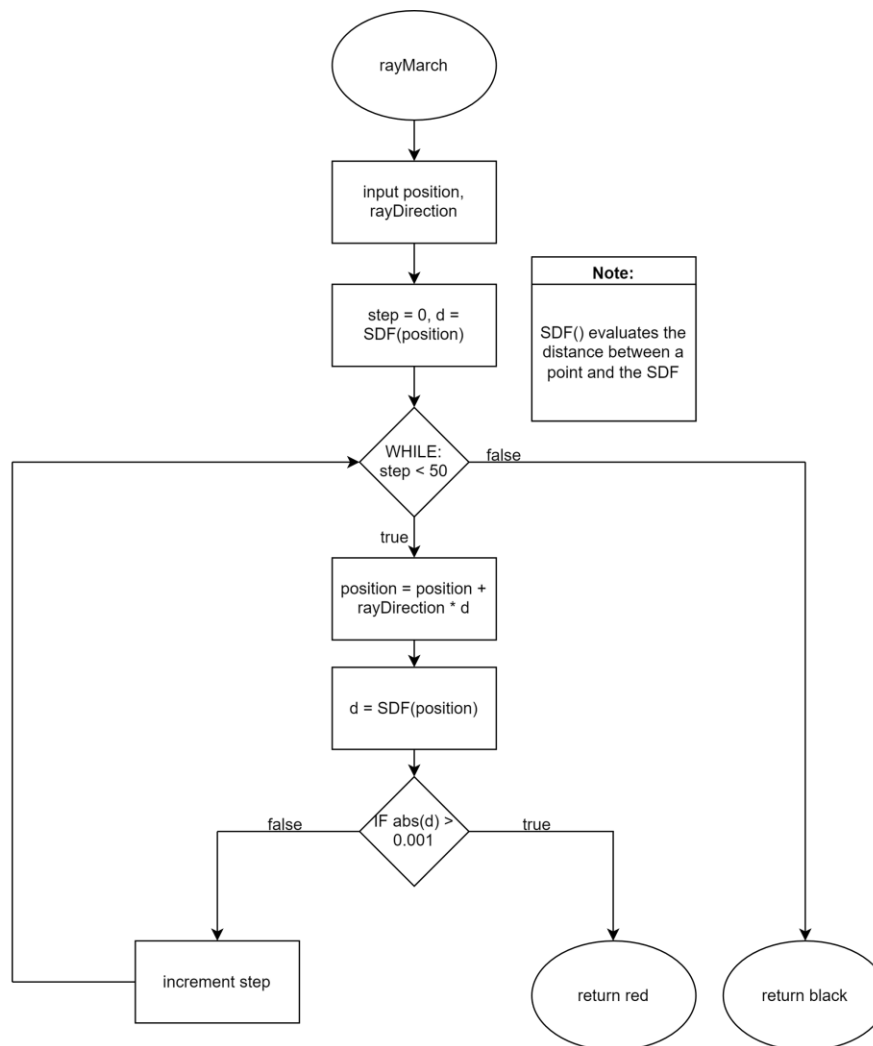


Fig. 4 below gives a nice illustration of what this algorithm is doing.

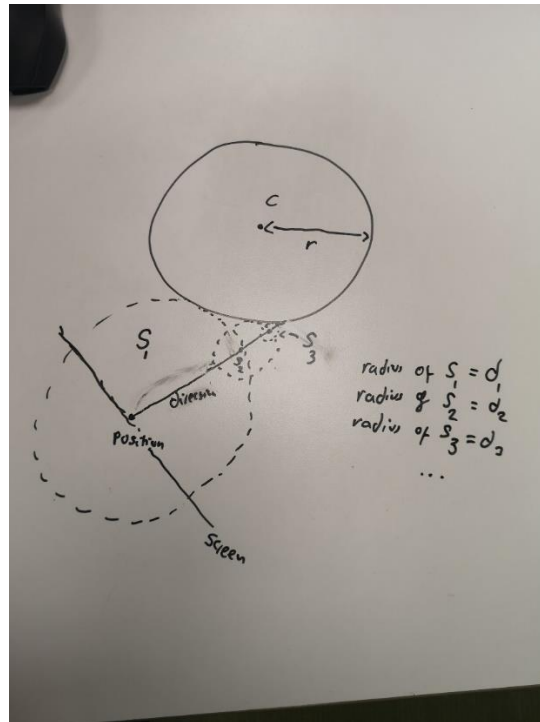


Fig. 4

SDF Arithmetic

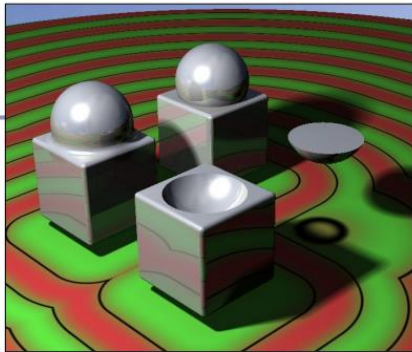
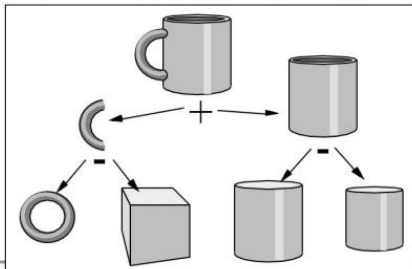
Next, to render more complicated SDFs and render multiple at once, I researched SDF arithmetic, which also lead me back to the Cambridge PowerPoint with the slide shown below.

Combining SDFs

We combine SDF models by choosing which is closer to the sampled point.

- Take the **union** of two SDFs by taking the $\min()$ of their functions.
- Take the **intersection** of two SDFs by taking the $\max()$ of their functions.
- The $\max()$ of function A and the negative of function B will return the **difference** of A - B.

By combining these binary operations we can create functions which describe very complex primitives.

11

I have illustrated what each of these does below:

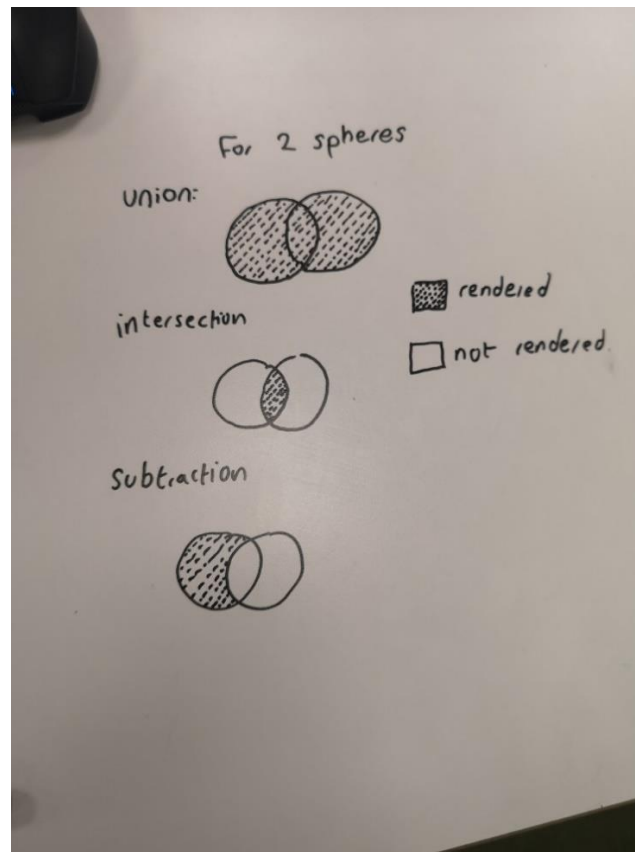


Fig. 5

Therefore, union allows multiple SDFs to be rendered, intersection allows the merging of shapes, and subtraction allows existing shapes to be altered by creating a new SDF that is an area which is not rendered.

Vertex and Fragment Shader Linking with Programs

To render SDFs, I must use vertex and fragment shaders. I discovered that the code in the Cambridge PowerPoint was written in GL Shader Language (GLSL) and I learnt much of the shader syntax online⁵. More on how I used these shaders is continued in the application section. To learn how to link the shaders to the main code, I researched these functions⁶:

- `glCreateShader(glType);`
Creates a new shader, where `glType` is of type `GLenum`. The types I used were either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`, which are abstractions for addresses.
- `glShaderSource(shader, count, string, length);`

⁵ learnopengl.com. (n.d.). *LearnOpenGL - Shaders*. [online] Available at: <https://learnopengl.com/Getting-started/Shaders>.

⁶ registry.khronos.org. (n.d.). *OpenGL 4 Reference Pages*. [online] Available at: <https://registry.khronos.org/OpenGL-Refpages/gl4/>.

Provides the source for a shader given the GLint of the shader, the number of shaders being passed (count), the shader source given as a string pointer, and the length, which can be left as nullptr.

- `glCompileShader(shader);`
Turns the shader into a form which OpenGL will understand.
 - `glGetShaderiv(shader, GL_COMPILE_STATUS, status);`
Returns the compile status of the specified shader and stores it in the passed-by-reference status variable.
 - `glGetShaderInfoLog(shader, bufferSize, length, buffer);`
In case of compilation fail, this returns the error into the buffer. Example code shown below, where `m_handle` is the GLint of the shader:
- ```
const int bufferSize = 1024;
char buffer[bufferSize];
GLsizei length = 0;
glGetShaderInfoLog(m_handle, bufferSize, &length, buffer);
```
- `glDeleteShader(shader);`  
Deletes the specified shader.
  - `glGenBuffers(n, buffers);`  
Generates `n` buffers and stores them in the passed-by-reference buffers.
  - `glBindBuffer(type, buffer);`  
Binds a buffer to a set type. The types I have used are `GL_ELEMENT_ARRAY_BUFFER`, `GL_UNIFORM_BUFFER`, and `GL_ARRAY_BUFFER`.
  - `glBindBufferBase(type, index, buffer);`  
Same as above, including the starting index pointer of an array.
  - `glBufferData(type, size, data, usage);`  
Creates data for a specified buffer type, with a set size (`sizeof(datatype) * sizeof(data)`). The usage describes how the data will be accessed.
  - `glGetUniformBlockIndex(program, "name");`  
Gets the name associated with a "block" in a program.
  - `glUniformBlockBinding(program, index, programIndex);`  
Binds data to a "block" in a specified program, given the starting index pointer of the data, and the starting index inside the program.
  - `glGetUniformLocation(program, name);`  
Gets the name associated with a uniform variable inside a program.
  - `glUniform[int][type](location, [number of data points worth of data]);`  
Adds data to a specified uniform returned from `glGetUniformLocation`. Example syntax is `glUniform3f` which means it is a 3-element array of type float. Any integer is accepted up to and including four, and the types accepted are (f)loat, (i)nt, and "fv" for a 2d array of floats, which are the ones I have used.
  - `glDeleteBuffers(n, buffers);`  
Deletes `n` buffers.
  - `glCreateProgram();`

Creates a program object and returns a GLuint which can be referenced. A program links shaders together.

- glDeleteProgram(program);  
Deletes a specified program.
- glAttachShader(program, shader);  
Attaches a shader to a specified program.
- glDetachShader(program, shader);  
Detaches a shader from a specified program.
- glLinkProgram(program);  
Makes any shaders attached to the program executable by the relevant processors.
- glGetProgramiv(program, GL\_LINK\_STATUS, status);  
Returns the link status of the specified program and stores it in the passed-by-reference status variable.
- glUseProgram(program);  
Runs the program specified.

### *GLFW and GLEW*

In addition to the shaders, to render SDFs, I researched an efficient way to render them, which meant I had to use a new side of OpenGL; the GL Framework (GLFW) and GL Extension Wrangler (GLEW) libraries. I used a video to learn how to set up the GLFW<sup>7</sup> environment and another video for GLEW<sup>8</sup>. Here is an overview of the functions used:

- glfwInit();  
This creates the GLFW environment and returns 1 if the initialisation went as expected.
- glfwCreateWindow(width, height, "title," monitor, share)  
This creates the window at a set width and height and gives the window a specified title. There are also optional parameters for a monitor and window to share resources with, which can just be left as NULL and the function still works. This function returns a GLFWwindow.
- glfwMakeContextCurrent(window)  
This sets the primary focus onto the window given as a parameter, which is of type GLFWwindow.
- glewInit();  
This creates the GLEW environment and returns GLEW\_OK if the initialisation went as expected. GLEW\_OK has an integer value of 0.
- glfwTerminate();  
Safely closes the GLFW environment.

---

<sup>7</sup> mossonthetree (2022). *03 Beginner's OpenGL Make a GLFW window*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=LdPztjrZV5g> [Accessed 19 Aug. 2024].

<sup>8</sup> mossonthetree (2022). *04 Beginner's OpenGL Initialize GLEW*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=jv4T4WFPHFk> [Accessed 19 Aug. 2024].

- `glfwWindowShouldClose(window);`  
Returns a Boolean value. Returns true if the user interacts with the close window button or there is a method set up for the user to press a key to close the window, otherwise returns false.
- `glClear(GL_COLOR_BUFFER_BIT);`  
Clears the buffer which sends colour information to the graphics card.
- `glfwSwapBuffers(window);`  
Refreshes the buffers for the specified GLFWwindow.
- `glfwPollEvents();`  
Processes all pending events, which could be the user clicking on something, or pressing a key.

### GLSL

To program some vertex and fragment shaders to use alongside GLFW and GLEW, I had to learn GL Shader Language (GLSL). I researched to learn the basics of GLSL<sup>9</sup>, which introduced me to the GLSL data types and uniform structures. Most of this felt very intuitive, as it was designed to have similar syntax to C++.

### UV Coordinates

UV coordinates are coordinates that are in the range -1 to 1, and is what OpenGL works well with for coordinate maths<sup>10</sup>. To convert a point in 3D space to be in the -1 to 1 range, the following GLSL code can be used:

```
vec2 uv = gl_FragCoord.xy / res.xy;
uv = uv * 2.0 - 1.0;
uv.x *= res.x / res.y;
```

Where `res` is the screen resolution, and `gl_FragCoord` is each pixel on the screen.

## Application

### Shaders

Based on my research, I could define some basic SDF shapes such as a sphere or a torus and implement methods to find their union, intersection, and to subtract them in GLSL. The fragment shader class is going to need to be a lot bigger than the vertex shader class because the vertex shader only needs to define the screen, which can be implemented as follows:

- `main.cpp`:

---

<sup>9</sup> Cocos.com. (2024). *Introduction to GLSL Syntax | Cocos Creator*. [online] Available at: <https://docs.cocos.com/creator/3.4/manual/en/shader/glsl.html> [Accessed 19 Aug. 2024].

<sup>10</sup> Moss, J. (2024). *How to convert X and Y screen coordinates to -1.0, 1.0 float? (in C)*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/31553325/how-to-convert-x-and-y-screen-coordinates-to-1-0-1-0-float-in-c> [Accessed 19 Aug. 2024].

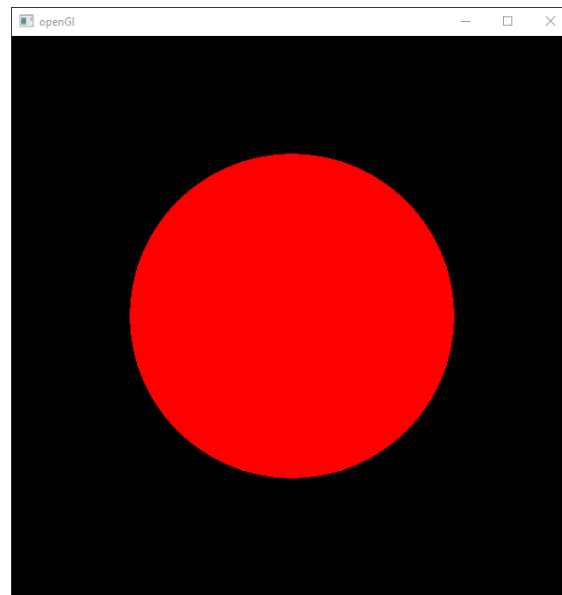
```

int main() {
 glfwInit()
 auto window = glfwCreateWindow(WINDOW_WIDTH, WINDOW_HEIGHT, "openGl", NULL, NULL);
 glfwMakeContextCurrent(window);
 glewInit()
 Shader vertexShader(Shader::Type::VERTEX, "./vertex.glsl");
 Program p;
 p.attachShader(vertexShader);
 std::vector<GLfloat> verts = {
 -1.0f, -1.0f,
 1.0f, -1.0f,
 -1.0f, 1.0f,
 1.0f, 1.0f
 };
 VertexBuffer vB;
 vB.fill(verts);
 std::vector<GLuint> indices = {0, 1, 2, 2, 1, 3};
 IndexBuffer iB;
 iB.fill(indices);
 auto vertexPosition = glGetUniformLocation(p.handle(), "vertexPosition");
 glEnableVertexAttribArray(vertexPosition);
 glBindBuffer(GL_ARRAY_BUFFER, vB.handle());
 glVertexAttribPointer(vertexPosition, 2, GL_FLOAT, GL_FALSE, 0, nullptr);
 while (!glfwWindowShouldClose(window)) {
 glClear(GL_COLOR_BUFFER_BIT);

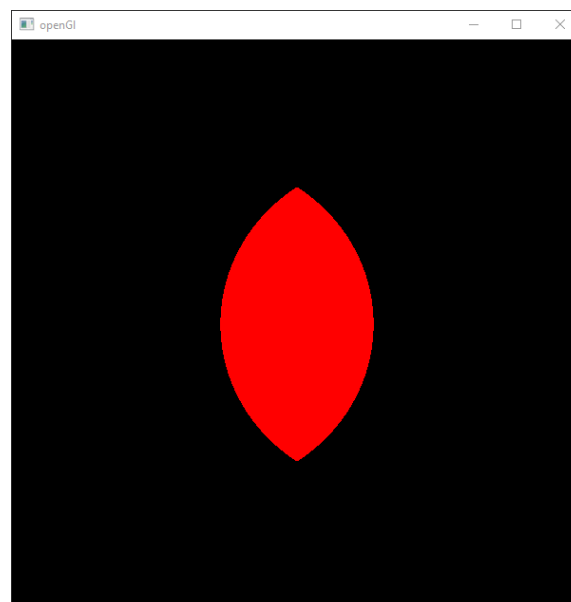
 p.activate();
 glDrawElements(GL_TRIANGLES, iB.number(), GL_UNSIGNED_INT, nullptr);
 glfwSwapBuffers(window);
 glfwPollEvents();
 }
 glfwTerminate();
 return 0;
}
- vertex.glsl
#version 440 core
layout(location = 0) in vec2 vertexPosition;
void main()
{
 gl_Position = vec4(vertexPosition, 0.0, 1.0);
}

```

Then, applying this with a sphere SDF that returns red inside and black outside yields this result:



And, applying the intersection as in Fig. 5:



### *Rotating SDFs*

Rendering an SDF is good, but I need a way to rotate these SDFs, so using the same perspective rotation matrices as before and applying the inverse to every point  $p$  provided, we can implement this method in GLSL:

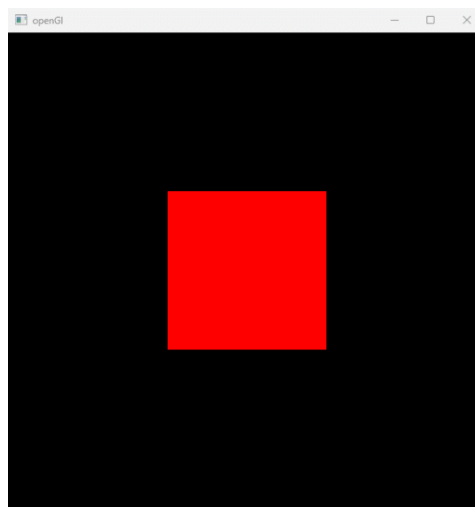
```
vec3 rotateSDF(vec3 p, float x, float y, float z) {
 mat3 rot = mat3(
 cos(y) * cos(z), sin(x) * sin(y) * cos(z) - cos(x) * sin(z), cos(x) * sin(y) * cos(z) +
sin(x) * sin(z),
 cos(y) * sin(z), sin(x) * sin(y) * sin(z) + cos(x) * cos(z), cos(x) * sin(y) * sin(z) - sin(x)
* cos(z),
 -sin(y), sin(x) * cos(y), cos(x) * cos(y)
);
}
```

```
p *= inverse(rot);
return p;
}
```

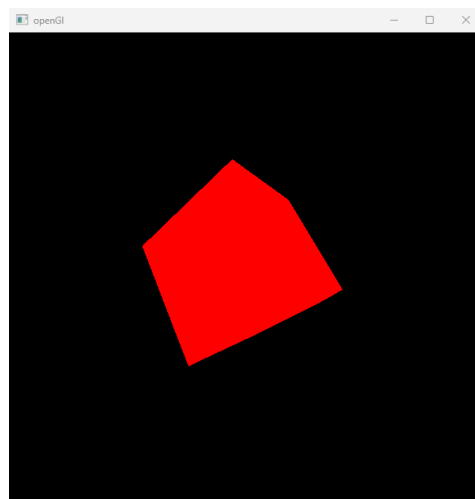
And, to see this in action, the SDF for a box in GLSL is:

```
float boxSDF(vec3 p, vec3 b) {
 vec3 q = abs(p) - b;
 return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
}
```

Where p is the position being checked, and b is the box's dimensions. Initially, the box is facing the camera:



But, after applying a rotation of  $\frac{\pi}{3}$  radians ( $30^\circ$ ) in both the x and y axes, we get this:



So, the rotation matrices can be applied to SDFs. The rotation matrix method looks different to that of the previous ones because the matrix I am using here is:

$$\begin{bmatrix} \cos \beta \cos \gamma & \sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma & \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma \\ \cos \beta \sin \gamma & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & \cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma \\ -\sin \beta & \sin \alpha \cos \beta & \cos \alpha \cos \beta \end{bmatrix}$$

Where the Euler angles  $\alpha$ ,  $\beta$ , and  $\gamma$  are rotations about the  $x$ ,  $y$ , and  $z$  axes, respectively. This matrix comes from the product of the initial matrices, calculated as:

$$\begin{aligned} R(\alpha, \beta, \gamma) &= R_z(\gamma)R_y(\beta)R_x(\alpha) \\ &= \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix} \\ &= \begin{bmatrix} \cos \beta \cos \gamma & \sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma & \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma \\ \cos \beta \sin \gamma & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & \cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma \\ -\sin \beta & \sin \alpha \cos \beta & \cos \alpha \cos \beta \end{bmatrix} \end{aligned}$$

Which are the same rotation matrices as before, just reduced from  $4 \times 4$  to  $3 \times 3$ .

## Lighting and Reflections

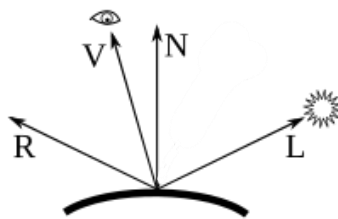
### Research

#### The Phong Reflection Model

If I am going to add lighting to my scene, I would have to research how to implement this. From my research, I found the following equation<sup>11</sup>:

$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

Where  $k_a$ ,  $k_d$ ,  $k_s$ , and  $\alpha$  are the ambient reflection, diffuse reflection, specular reflection, and shininess constants, respectively. Moreover, *lights* is the set of all light sources,  $\hat{L}_m$  is the normalised  $m^{\text{th}}$  light source position vector,  $\hat{N}$  is the normal vector at the given position,  $\hat{R}_m$  is the normalised direction a perfectly reflected ray of light would take, and  $\hat{V}$  is the normalised view vector.



For each light in *lights*,  $i_{m,s}$  is the intensity of the  $m^{\text{th}}$  light, and  $i_{m,d}$  is the colour of the  $m^{\text{th}}$  light. Finally,  $i_a$  is the colour of the object, or the “ambient” colour.

<sup>11</sup> GeeksforGeeks. (2021). *Phong model (Specular Reflection) in Computer Graphics*. [online] Available at: <https://www.geeksforgeeks.org/phong-model-specular-reflection-in-computer-graphics/>.

## Shadows

If I want the scene to look realistic, I am also going to need to know how to add shadows to SDFs. The flow diagram (Fig. 6) shows a simple algorithm to determine whether an SDF is in the shadow of another SDF or not. I found this from a website where the author was exploring how to add shadows to 2D SDFs<sup>12</sup>, and found this had applications for 3D SDFs as well, as screenshot of which is shown below:

Then we can do checks whether we're already at the point where we can abort the loop. If the scene distance of the signed distance function is near 1, we can assume the ray was blocked by a shape and we return 0. If the ray progress is bigger than the light distance, we can assume that we reached the light without any collisions and return a value of 1.

If we didn't return yet, we then have to calculate the next sample position. We do that by adding the distance of the scene to the ray progress. The reason for this is that the scene distance gives us the distance to the nearest shape, so if we add that amount to our ray, we can't possibly cast our ray further than the nearest shape, or even beyond it, which would allow for shadow leaking.

In case we don't hit anything and also don't find the light by the time we used our whole sample budget (the loop ended), we also have to return a value. Because this mainly happens near shapes, shortly before the pixel would count as occluded anyways, we'll use a return value of 0 here.

```
#define SAMPLES 32

float traceShadows(float2 position, float2 lightPosition){
 float2 direction = normalize(lightPosition - position);
 float lightDistance = length(lightPosition - position);

 float rayProgress = 0;
 for(int i=0 ;i<SAMPLES; i++){
 float sceneDist = scene(position + direction * rayProgress);

 if(sceneDist <= 0){
 return 0;
 }
 if(rayProgress > lightDistance){
 return 1;
 }

 rayProgress = rayProgress + sceneDist;
 }

 return 0;
}
```

<sup>12</sup> 2D SDF Shadows (2018). *2D SDF Shadows*. [online] Ronja-tutorials.com. Available at: <https://www.ronja-tutorials.com/post/037-2d-shadows/> [Accessed 19 Aug. 2024].



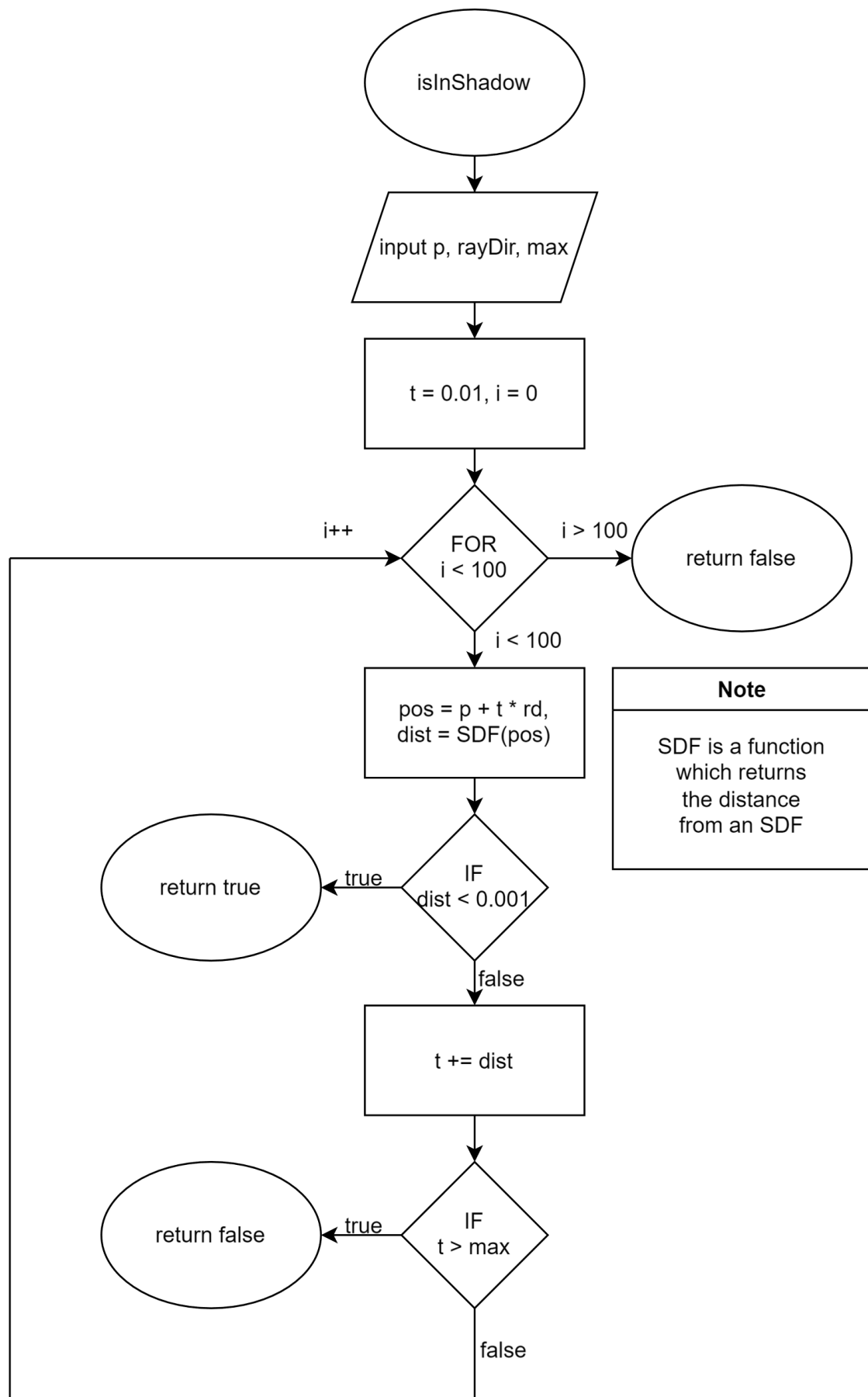
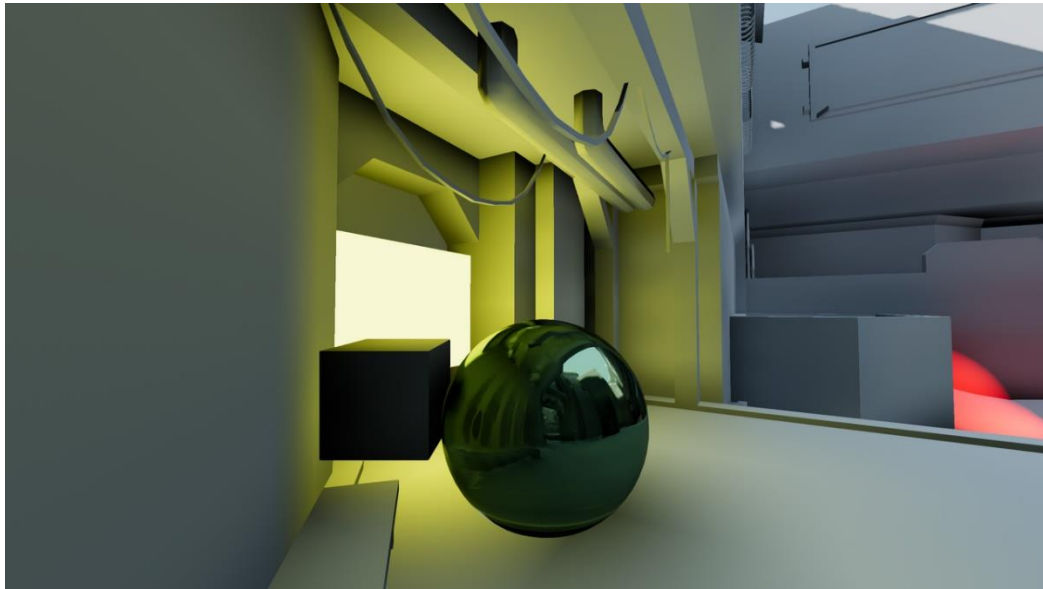


Fig. 6

This algorithm is remarkably close to the raymarching algorithm I discovered in the SDF research section, so it should work very well.

### *Mirror Reflections*

To make my scene more convincing, in real life you can see reflections, so I should have some internal reflections visible in my scene.



For example, in this image, the sphere at the centre of the scene is reflective, so you can see other elements of the scene inside it. From my research, I found an overly complex shader code that showed how reflections could be calculated in real time<sup>13</sup>. This method uses a ray, sent from a certain point throughout the world to see what it hits. I also combined what I learnt from that with more research<sup>14</sup>, which takes a recursive approach at raytracing for reflections.

## Application

### *Lighting*

One of the ways to apply the Phong Reflection Model was to create a method in GLSL which could return the gradient of colour at a given point, as many of the variables in the model can be obtained easily via calculation or user input, as follows:

| Variable Symbol | Variable Name       | How Data is Obtained |
|-----------------|---------------------|----------------------|
| $k_a$           | Ambient coefficient | User input           |
| $i_a$           | Ambient colour      | User input           |
| $k_d$           | Diffuse coefficient | User input           |

<sup>13</sup> Shadertoy.com. (2015). *Shadertoy*. [online] Available at: <https://www.shadertoy.com/view/ldt3Dr> [Accessed 19 Aug. 2024].

<sup>14</sup> Recursive Ray Tracing. (n.d.). Available at: [https://web.stanford.edu/class/cs148/materials/class\\_06\\_recursive\\_ray\\_tracing.pdf](https://web.stanford.edu/class/cs148/materials/class_06_recursive_ray_tracing.pdf).

|           |                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-----------|-------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\hat{L}$ | Light position vector                           | User inputs light position, and the vector is then normalised by dividing by its magnitude                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| $\hat{N}$ | Normal vector                                   | <p>Calculated by taking the slope to the surface at a given position. The definition for this, where <math>f(p)</math> is an SDF function, is:</p> $f'(p) = \nabla f(p) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{pmatrix}$ <p>But this is hard to calculate so instead we can use a formal definition:</p> $f'_1(p) = \begin{pmatrix} f(p + \begin{pmatrix} \varepsilon \\ 0 \\ 0 \end{pmatrix}) \\ f(p + \begin{pmatrix} 0 \\ \varepsilon \\ 0 \end{pmatrix}) \\ f(p + \begin{pmatrix} 0 \\ 0 \\ \varepsilon \end{pmatrix}) \end{pmatrix}$ <p>Which is the same result, since:</p> $\lim_{\varepsilon \rightarrow 0} f'_1(p) = f'(p)$ <p>And the second method is better, as it is more computationally efficient</p> |
| $i_d$     | Light colour                                    | User input                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| $k_s$     | Specular coefficient                            | User input                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| $\hat{R}$ | Direction of a perfectly reflected ray of light | Defined as $\hat{L} - 2(\hat{N} \cdot \hat{L})\hat{N}$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| $\hat{V}$ | View vector                                     | This is the incoming vector from the UV co-ordinates, normalised by dividing by its magnitude                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| $\alpha$  | Shininess coefficient                           | User input                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| $i_s$     | Light intensity                                 | User input                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

The one part of this table that is harder to understand than the other parts is the direction of a perfectly reflected ray of light. This is due to the definition of “reflecting” a vector on a surface. Let  $\vec{v}$  be an incoming vector,  $\vec{n}$  be the normal to the surface, and  $\vec{w}$  be the reflected vector. For notation,  $\vec{v}_{\parallel}$  is a parallel vector to  $\vec{v}$ , and  $\vec{v}_{\perp}$  is a perpendicular vector to  $\vec{v}$ .

For finding a projected vector to a given vector, this method is used:

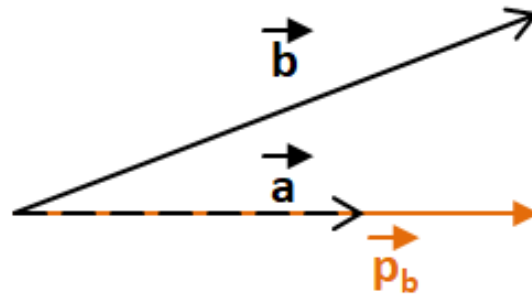


Fig. 7

From Fig. 7,  $\vec{p}_b$  is a projected vector of  $\vec{b}$ . We can use the dot product to calculate  $\|\vec{p}_b\|$ , namely  $\|\vec{p}_b\| = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|}$ . However, if  $\vec{a}$  is normalised, then this becomes  $\|\vec{p}_b\| = \hat{a} \cdot \vec{b}$ .

For splitting a vector into vertical and horizontal components with respect to another vector, the following method is used:

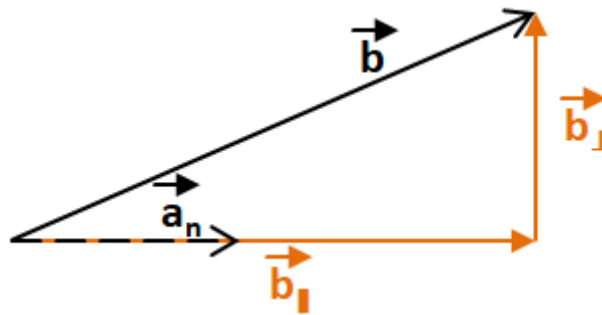


Fig. 8

From Fig. 8,  $\vec{a}_n = \hat{a}$ , so  $\|\vec{b}_\parallel\| = \hat{a} \cdot \vec{b}$ . Therefore, to just get the vector  $\vec{b}_\parallel$ , multiply  $\vec{b}_\parallel$  by  $\hat{a}$ :  $\vec{b}_\parallel = (\hat{a} \cdot \vec{b})\hat{a}$ . Then,  $\vec{b}_\perp$  is calculated from the vector sum of  $\vec{b}$  and  $\vec{b}_\parallel$ :  $\vec{b}_\perp = \vec{b} - \vec{b}_\parallel = \vec{b} - (\hat{a} \cdot \vec{b})\hat{a}$ .

This is applied to reflection as follows:

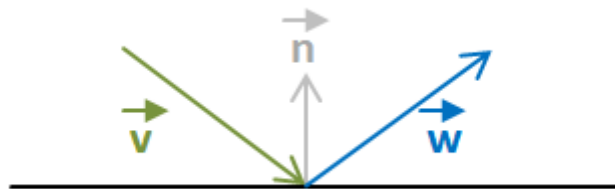


Fig. 9

From Fig. 9, we need to split  $\vec{v}$  into its parallel and perpendicular components. Doing some vector rearrangement gives Fig. 10

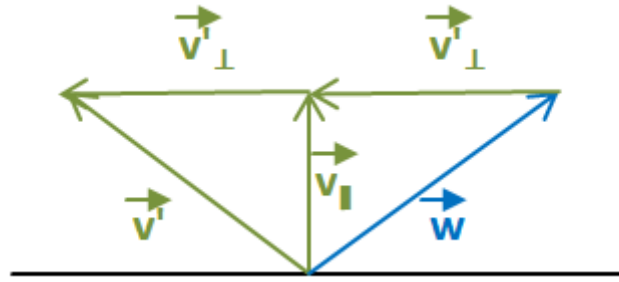


Fig. 10

In Fig 10.,  $\vec{v}'$  denotes  $\vec{v}$  negated. From this diagram, we can form an expression for  $\vec{w}$ . Namely,  $\vec{w} = \vec{v}' + (-\vec{v}'_{\perp}) + (-\vec{v}'_{\perp}) = \vec{v}' - 2\vec{v}'_{\perp}$ . Substituting  $\vec{v}' = -\vec{v}$ , the expression can be simplified:

$$\begin{aligned}\vec{w} &= -\vec{v} - 2(-\vec{v}_{\perp}) \\ \vec{w} &= -\vec{v} - 2(-\vec{v} - (-\vec{v} \cdot \vec{n})\vec{n}) \\ \vec{w} &= -\vec{v} + 2(\vec{v} + (-\vec{v} \cdot \vec{n})\vec{n}) \\ \vec{w} &= -\vec{v} + 2\vec{v} + 2(-\vec{v} \cdot \vec{n})\vec{n} \\ \vec{w} &= \vec{v} + 2(-\vec{v} \cdot \vec{n})\vec{n} \\ \vec{w} &= \vec{v} - 2(\vec{v} \cdot \vec{n})\vec{n}\end{aligned}$$

Which is of the form of  $\hat{L} - 2(\hat{N} \cdot \hat{L})\hat{N}$ , as the dot product is commutative.

### Materials

So that my scene could have multiple objects of assorted colours, and have different coefficients of the ones listed above, I could create a material which stores all the data in a way which can be used. For this data to be changed in real time, I could use a uniform buffer block which is an array in GLSL which data is passed to from the main program. It is defined as follows in this example code:

```
layout(std140) uniform bindPoint {
 Material materials[materialsLen];
};
```

The code outside of GLSL finds the name “bindPoint” and then maps the associated array with the array in GLSL.

### Shadows

To implement the shadow algorithm in the research section, I decided to again use GLSL and have it as a boolean method for ease of use, as shown below:

```
bool isInShadow(vec3 p, vec3 rd, float dist) {
 float t = 0.01;
```

```

for (int i = 0; i < 100; i++) {
 vec3 pos = p + t * rd;
 SDF res = finalSDF(pos);
 if (res.dist < 0.001) {
 return true;
 }
 t += res.dist;
 if (t >= dist) {
 return false;
 }
}
return false;
}

```

### Reflections

The normal way to implement reflections, as I discovered in my research, is by using a recursive function, but GLSL does not support recursion due to it being designed to be lightweight. To get round this, I transferred the recursive method into an iterative method with a maximum depth so that it could function in GLSL. Here is my algorithm:

```

vec3 sortCol(vec3 ro, vec3 rd, float maxDist) {
 vec3 c = vec3(0.0, 0.0, 0.0);
 float Rf = 1.0;
 for (int depth = 0; depth < 3; depth++) {
 vec2 t = rayMarch(ro, rd, maxDist);
 vec3 pos = ro + t.x * rd;
 if (t.x < maxDist && t.y >= 0) {
 vec3 n = calculateNormal(pos);
 vec3 view = normalize(ro - pos);
 vec3 la = vec3(1.0);
 vec3 surfaceC = getCol(la, materials[int(t.y)],
 lightCols, n, lights, view, pos);

 c += Rf * surfaceC;
 Rf *= materials[int(t.y)].Kr;

 rd = reflect(rd, n);
 ro = pos + n * 0.01;
 } else {
 break;
 }
 }
 return c;
}

```

## Rigid Body Dynamics

### Research

#### Object Attributes

From reading a paper on rigid body dynamics<sup>15</sup>, I understood that objects need to have certain key attributes. These attributes are:

- Position,  $\vec{r}$ , the object's position in 3D space.
- Velocity,  $\vec{v}$ , the object's velocity, which increments the position each frame.
- Acceleration,  $\vec{a}$ , the object's acceleration, which increments the velocity each frame.
- Mass,  $m$ , which is a relative number used in determining the outcome of collisions. There is an option to give an object a density,  $\rho$ , then the mass can be calculated by summing each infinitesimally small mass, which is calculated as:

$$m = \int_{\Omega} \rho dV = \iiint \rho dx dy dz$$

Which, for a sphere, is evaluated as follows:

$$m = \int_{\Omega_{sphere}} \rho dV = \int_{-r}^r \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-x^2-y^2}}^{\sqrt{r^2-x^2-y^2}} \rho dx dy dz$$

$$m = \rho \int_{-r}^r \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-x^2-y^2}}^{\sqrt{r^2-x^2-y^2}} dx dy dz$$

$$m = \rho \int_{-r}^r 2 \int_{-r}^r \sqrt{r^2 - x^2 - y^2} dx dy$$

Using the substitution  $y = \sqrt{r^2 - x^2} \sin \theta$ ,  $dy = \sqrt{r^2 - x^2} \cos \theta d\theta$

$$m = \rho \int_{-r}^r 2 \int_{-r}^r \sqrt{r^2 - x^2} \cos \theta \sqrt{-(r^2 - x^2) \sin^2 \theta - x^2 + r^2} dx d\theta$$

$$m = \rho \int_{-r}^r 2(r^2 - x^2) \int_{-r}^r \cos^2 \theta dx d\theta$$

$$m = \rho \int_{-r}^r 2(r^2 - x^2) \left( \frac{\cos \theta \sin \theta + \theta}{2} \right) \Big|_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}}$$

$$m = \rho \int_{-r}^r (r^2 - x^2) \sin^{-1}(1) - (r^2 - x^2) \sin^{-1}(-1) dx$$

$$m = \rho \int_{-r}^r \frac{\pi}{2} (r^2 - x^2) + \frac{\pi}{2} (r^2 - x^2) dx$$

$$m = \pi \rho \int_{-r}^r r^2 - x^2 dx$$

$$m = \frac{4}{3} \pi r^3 \rho$$

<sup>15</sup> Chapter 6 Rigid Body Dynamics. (n.d.). Available at:

[https://www.brown.edu/Departments/Engineering/Courses/En4/Notes/Rigid\\_Bodies\\_1/Rigid\\_Bodies.pdf](https://www.brown.edu/Departments/Engineering/Courses/En4/Notes/Rigid_Bodies_1/Rigid_Bodies.pdf)

Therefore, this mass formula correctly produces volumes to be multiplied by the density to give a mass. There are other formulas which work equally as well.

### Collision Detection

As it turns out, rigid body dynamics is mostly normal physics, so I derived that when two objects collide it should satisfy the following equation, where  $\vec{r}_1$  is the 3D position vector of object 1,  $\vec{r}_2$  is the same for object 2,  $a$  is the defining length of object 1, and  $b$  is the same for object 2:

$$\|\vec{r}_1 - \vec{r}_2\| < a + b$$

### Collision Resolution

Resolving collisions is a bit harder, and using the same paper as before, I interpreted that to resolve a collision between object one and object two, you follow this procedure, using the same notation as in the above paragraph:

- The collision normal,  $\hat{n}$ , is calculated:

$$\hat{n} = \frac{1}{\|\vec{r}_2 - \vec{r}_1\|} (\vec{r}_2 - \vec{r}_1)$$

- The relative velocity,  $\vec{v}_R$ , is calculated:

$$\vec{v}_R = \vec{v}_2 - \vec{v}_1$$

- The relative velocity is dotted against the normal to give it direction:

$$v_N = \vec{v}_R \cdot \hat{n}$$

- The impulse scalar,  $j$ , is calculated with coefficient of restitution  $e$  for dampening ( $m_1$  and  $m_2$  are the respective object masses):

$$j = \frac{-(1 + e)v_N}{\frac{1}{m_1} + \frac{1}{m_2}}$$

- As  $j$  is an impulse, it is equal to a change in momentum ( $\Delta p$ ) which allows each object's velocities to be altered accordingly, as  $j = \Delta p = m\Delta v$ .

## Application

### Objects

Each object will need to have attributes which allow the total control of the entire scene. For this, a C++ struct can be used as follows:

```
struct ObjectData {
 int type, material;
 vec r;
 float l1, mass;
 vec vel, angVel;
 bool down = true, moving = false, floor = false;
};
```



In this, the only relevant parts to rigid body dynamics are:

- $r$ , the position vector of the object.
- $vel$ , the velocity vector of the object.
- $angVel$ , the angular velocity of the object.

### *Detecting and Handling Collisions – SDFs*

When two SDFs collide, a collision can be handled by the collision detection algorithm  $\|\vec{r}_1 - \vec{r}_2\| < a + b$ , where in place of the position vectors, the SDF at a given point can be plugged in, and the intersection can be computed to be greater or less than 0.

## Modelling

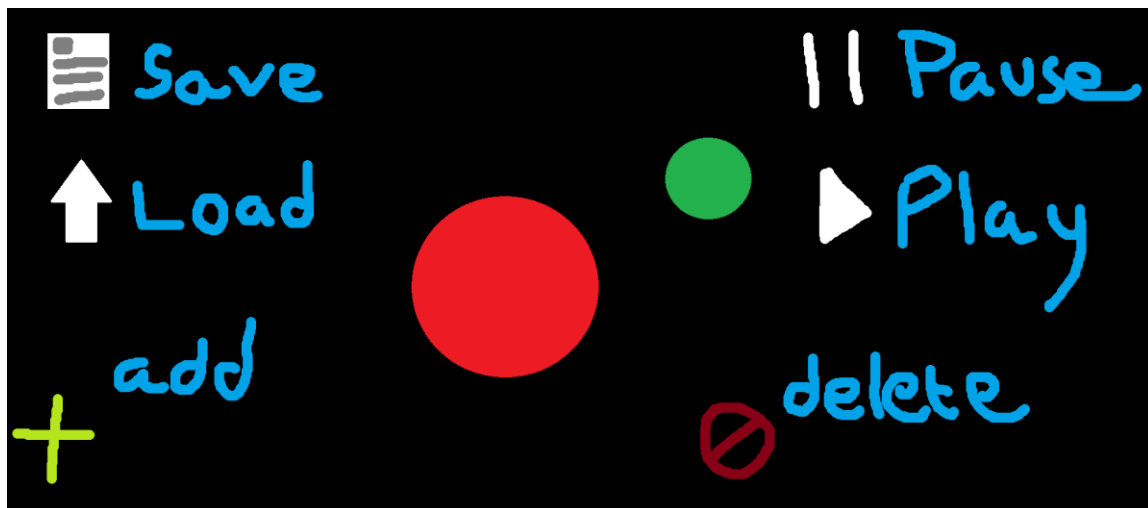
### Introduction

Lots of 3D rendering software is picky to use and relies on a lot of prior knowledge of other software or how 3D rendering works. With my design, I want it to be simple, intuitive, and easy to use. This means including a minimalistic design with not too many technical terms, using sliders rather than text boxes, and an intuitive layout, with things like sidebars and a menu too.

### Scenes

#### *Main Screen*

The main scene you would see would have a few options. In discrete corners you could focus on the scene that you were making in front of you. Here is a simple diagram depicting how this could be achieved:



Here is a run through of the buttons:

- Save  
Saves the current scene in the paused state so that it can be reloaded.
- Load

Loads a previously saved scene.

- Pause

Stops the in-program passage of time meaning nothing can move.

- Play

Resumes the in-program passage of time if paused.

- Add

Adds another object to the scene in another menu.

- Delete

Prompts the user to click on an object and then removes it from the scene.

In addition to these buttons, there would be a feature where if the user pressed a key such as “escape,” the main menu would open.

### *Click Events*

If a user were to right click on an existing object, a menu would come up displaying the objects attributes and would allow the user to edit them.

|                      |     |
|----------------------|-----|
| Colour               |     |
| Clarity              | 0.9 |
| Light spread         | 0.8 |
| Light reflectiveness | 0.6 |
| Shininess            | 32  |

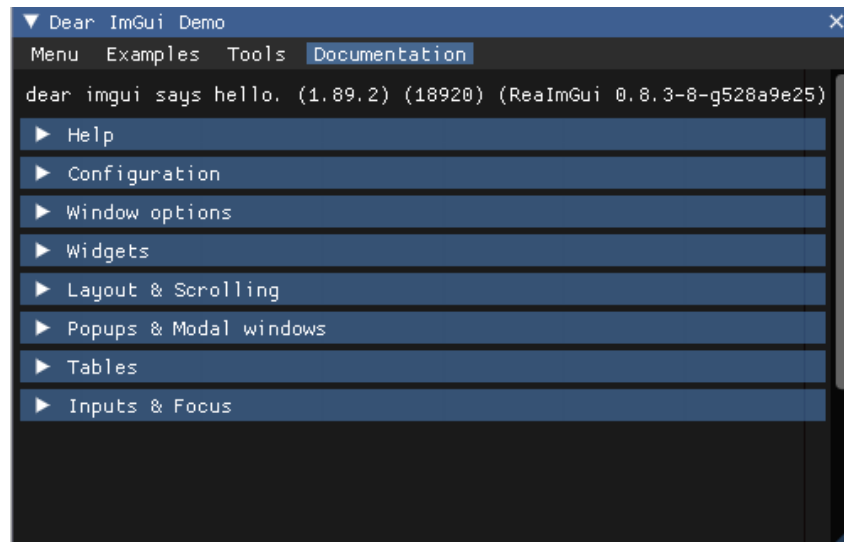
## The Graphical User Interface

### *C++ ImGui*

In C++ programming with OpenGL using OpenGL3 and GLFW, there is limited capability surrounding the creation of a graphical user interface (GUI), so to get around this, I will be using a module called `imgui`<sup>16</sup> which allows the creation of a GUI inside of an OpenGL3 GLFW environment. The general `imgui` interface looks like this:

---

<sup>16</sup> Ocornut (2019). `ocornut/imgui`. [online] GitHub. Available at: <https://github.com/ocornut/imgui> [Accessed 15 Oct. 2024].



## Design

### Introduction

This project is a 3D rendering engine for rendering dynamic physics simulations, with an ease of access in terms of altering global variables. A user has the option to create a new scene or load an existing scene. In a scene, a user can add or remove objects to see how they interact and fill the space when put in specific circumstances, and how different initial conditions can lead to different results. There is also a variety of menus which allow the user to understand how the program is working in simple terms that someone with less of a mathematical background could understand.

### External Libraries

The only external libraries used are some libraries from OpenGL, along with the user interface library Dear ImGui.

#### *OpenGL*

OpenGL: <https://www.opengl.org/>.

GLFW: <https://glfw.org/>.

GLEW: <https://glew.sourceforge.net/>.

GLSL: [https://www.khronos.org/opengl/wiki/Core\\_Language\\_\(GLSL\)](https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)).

#### *Dear ImGui*

Dear ImGui: <https://github.com/ocornut/imgui>.

## System Overview

My program consists of many different menus and options, and this section will give an overview of what each menu does, and how the user interacts with the program.

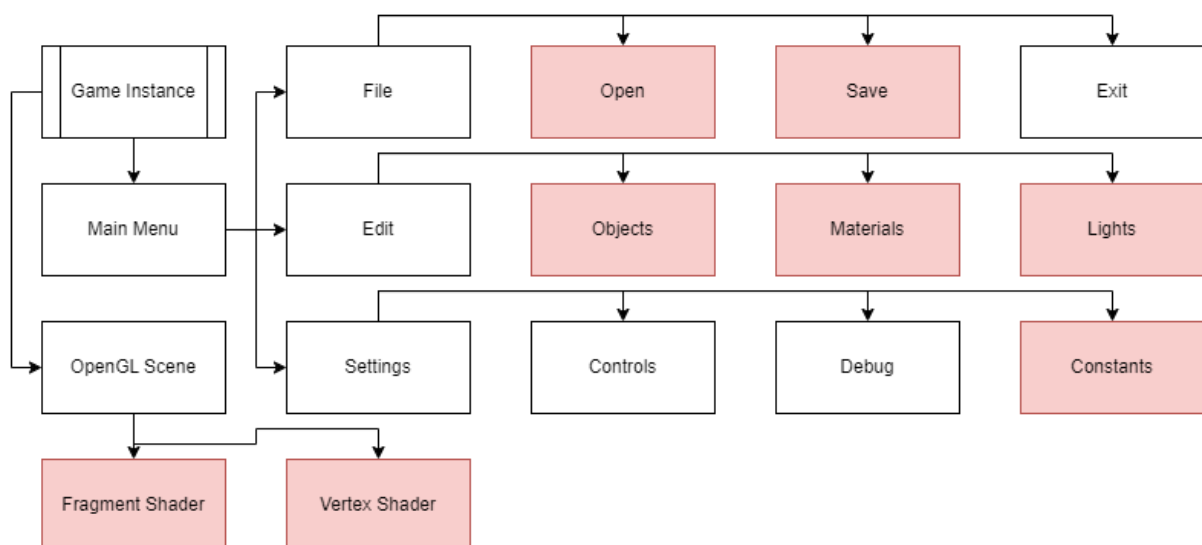
### Startup

When the program begins, the environment needs to be set up, so the various libraries need to be prepared first. For GLFW, it creates the window and assigns the keyboard and mouse to it and sets the context to the current window making it ready for later use. Next, GLEW is also initialised for its functions to be used later in the code. Finally, ImGui is set up with keyboard use to be used with the current window.

The next stage of the setup is to compile the shaders, and the shaders are text files that are interpreted by the GLSL compiler as character arrays. Once these are compiled, the main loop can begin. At this stage, the correct OpenGL and ImGui calls are made, and the data surrounding the objects on the screen is updated and sent to the fragment shader.

At any point, the user can quit by pressing escape.

### Game Instance



In a game instance, there is a menu bar at the top which allows a user to add certain things into the scene, or view aspects of the program. Of these, there is File, Edit, and Settings. Any boxes in red have further user interactions. There also exists the main scene rendered by OpenGL.

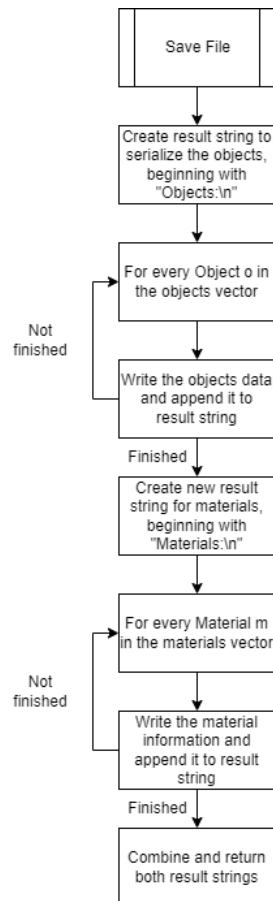
### File

In the File dropdown menu, the user can either select Open, Save or Exit. If the user decides to Exit, the call `glfwSetWindowShouldClose(window, GLFW_TRUE)` is called. This exits the main loop and then further shutdown procedures occur.

*Saving*

If the user decides to Save what they have currently made, another menu appears which prompts the user to name the file. The following procedure then happens:

- The objects and data get serialized:



- The data gets written to a file

FUNCTION save:

Create an empty buffer for a character string

Get the user input for the file name

fName = userInput + ".txt"

data = serialize(objects, materials)

FileHandler f(fName.toCharacterString)

f.writeFile(data)

ENDFUNCTION

The method for writing to files will be explained in the FileHandler section. The reason the file name must be dealt with as a character string is because there is more capability in C++ for character strings, rather than using the string library.

### *Opening*

When a user selects that they want to load a previous file state, the filesystem library is used to find all files that are text files and displays them to the user in a dropdown bar. The user then decides which file they would like to load, causing the data to be read in, deserialized, and displayed on screen as the saved visual state. Here is some pseudocode which describes how this is done, where files is a vector of all text files:

FUNCTION load:

    IF (files.isEmpty()):

        FOR EACH entry in directory:

            IF entry is normal file AND has ".txt":

                files.push\_back(entry.name())

            ENDIF

        ENDFOR

    ENDIF

    file = user get file from dropdown menu

    IF (load button is pressed):

        FileHandler f(files[dropdown index])

        data = f.readFile()

        IF (data.isEmpty() == FALSE):

            deserialize(data)

        ENDIF

    ENDIF

ENDFUNCTION

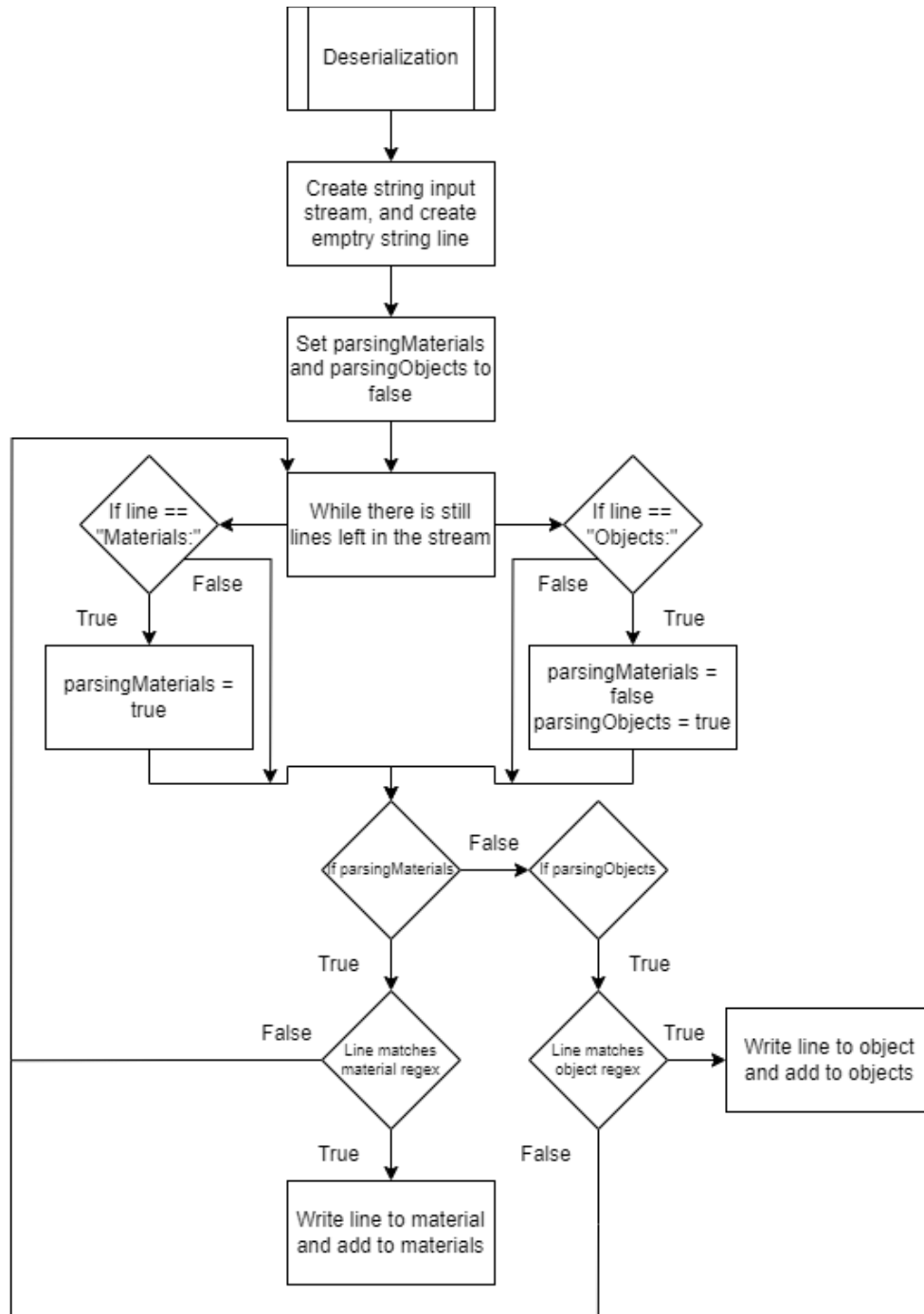
The deserialization function is quite long and is about matching regular expressions. The regular expression used for the materials is as follows:

- r:([0-9\\.]+) g:([0-9\\.]+) b:([0-9\\.]+) Ka:([0-9\\.]+) Kd:([0-9\\.]+) Ks:([0-9\\.]+) Kr:([0-9\\.]+) c:([0-9\\.]+)

And the objects regex is:

- type:([0-9]+) material:([0-9]+) pos:(\\-?[0-9\\.]+),(\\-?[0-9\\.]+),(\\-?[0-9\\.]+)  
radius:([0-9\\.]+) mass:([0-9\\.]+)

Hence, the function which deserializes data can be represented by this flow diagram:



[Edit](#)

The Edit menu has a few submenus, namely Objects, Materials and Lights. The Objects submenu allows the user to add objects, and remove the last object added, or clear all objects. It consists of all of the parameters necessary for an object to be created, and

when the create button is pressed, these aspects are made into `ObjectData` which is used in the constructor of a new `Object`, which is then added to the objects vector. The Materials submenu initially allows the user to view the different materials available, and their respective coefficients in terms of the Phong reflection model. It also previews the colour that is stated as an RGB value. At the bottom of this submenu, there is the option to add other materials, which in this case opens a new submenu with a colour picker and input fields for the Phong coefficients. Finally, the Lights submenu is very similar to the Materials submenu, in which it shows the position of the lights and the colour of the lights, with a colour preview, and a similar button to Materials in that the user can add more lights. This is laid out the same as the “add a material” menu.

### Settings

The final menu, Settings, allows the user to view the Controls, Debug information, and information about the Constants in the simulation. The Controls section purely provides a summary of how to interact with the scene, and what different buttons such as Esc and J do to the scene. The Debug information provides information about the current states of objects in the scene, purely printing their `ObjectData`. The Constants section allows the user to view and vary different global constants in the scene. Namely, the value of gravity (base value of 9.81 which in the code translates to 0.005, this is calculated by scaling the UI version by a value of  $0.005/9.81$ ), the value of the coefficient of restitution, which affects the bounciness of the balls, and the value of the friction, which changes the rate at which the balls slow down when in contact with the ground.

### The Main Scene

The main scene is compiled by the combination of the fragment shader calculating what each pixel on the screens’ colour should be, and then the vertex shader renders two triangles (either side of the screen) to complete the screen.

## Algorithms and Modules

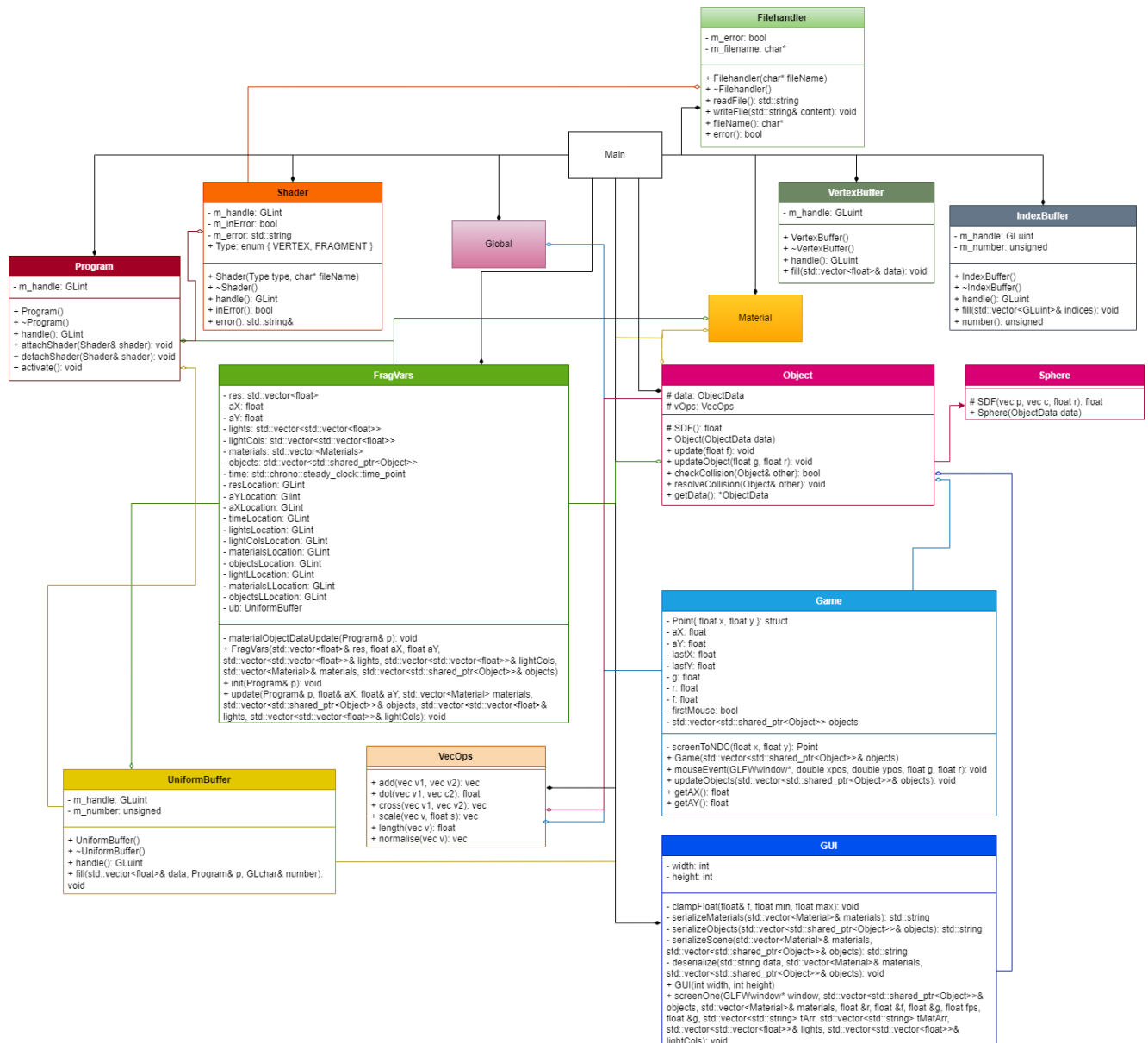
This section will contain an in-depth explanation as to what each of the classes do, and how they interact with each other. It will also contain how each of the key algorithms, among others are used in their respective classes.

### Classes

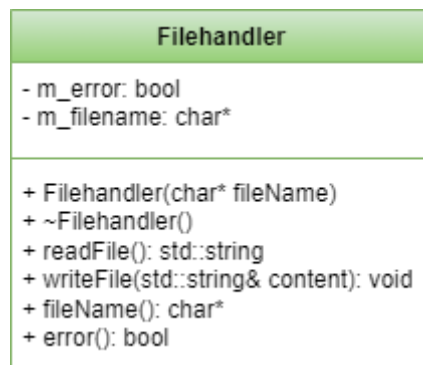
The classes discussed in this section will be discussed in alphabetical order. Any classes which are subclasses will be included in the section of their respective parent class. The “\*” character denotes a pointer variable and the “&” denotes a memory address and is used for passing values by reference.



## All Classes



## Filehandler



Filehandler is a class made for reading files and writing to files. It is used in the main class to read the shader files. Its private variables are `m_error` and `m_filename`. The first

of these stores whether an error has occurred in the file processes. The second of these stores the name of the file being accessed as a C-String. Other important local method variables include:

- `iStream` – A variable of type `std::ifstream` in `readFile()` which controls the input stream from the file.
- `oStream` – A variable of type `std::ofstream` in `writeFile()` which controls the output stream to a file.

### Methods

- `Filehandler(filename)`. This is the constructor for the `Filehandler` class. It begins by setting the value of `m_error` to false. It then gets the length of the `fileName` specified, and casts it to a character array using C-standard memory allocation. If this was successful, it copies the `fileName` into `m_fileName`, which is accessible to the rest of the class, frees used memory, and dereferences pointers. It also identifies if any errors occur.
- `~Filehandler()`. This is the destructor for the `Filehandler` class and is called when an instance of the class is no longer needed. It frees the memory used by `m_filename`.
- `readFile()`. This reads the specified file that was given in the argument of the constructor. It creates an input file stream and stores the result of it into a string stream, of which the contents are put into a string which is returned.
- `writeFile(&content)`. This writes to the specified file that was given in the argument of the constructor. It creates an output file stream, and for each integer `i` that corresponds to a piece of content in that stream, it is written to the file.
- `fileName()`. This is a getter function that returns the name of the file being accessed as a pointer to a character array.
- `error()`. This is a Boolean getter function for whether or not there has been an error.

*FragVars*

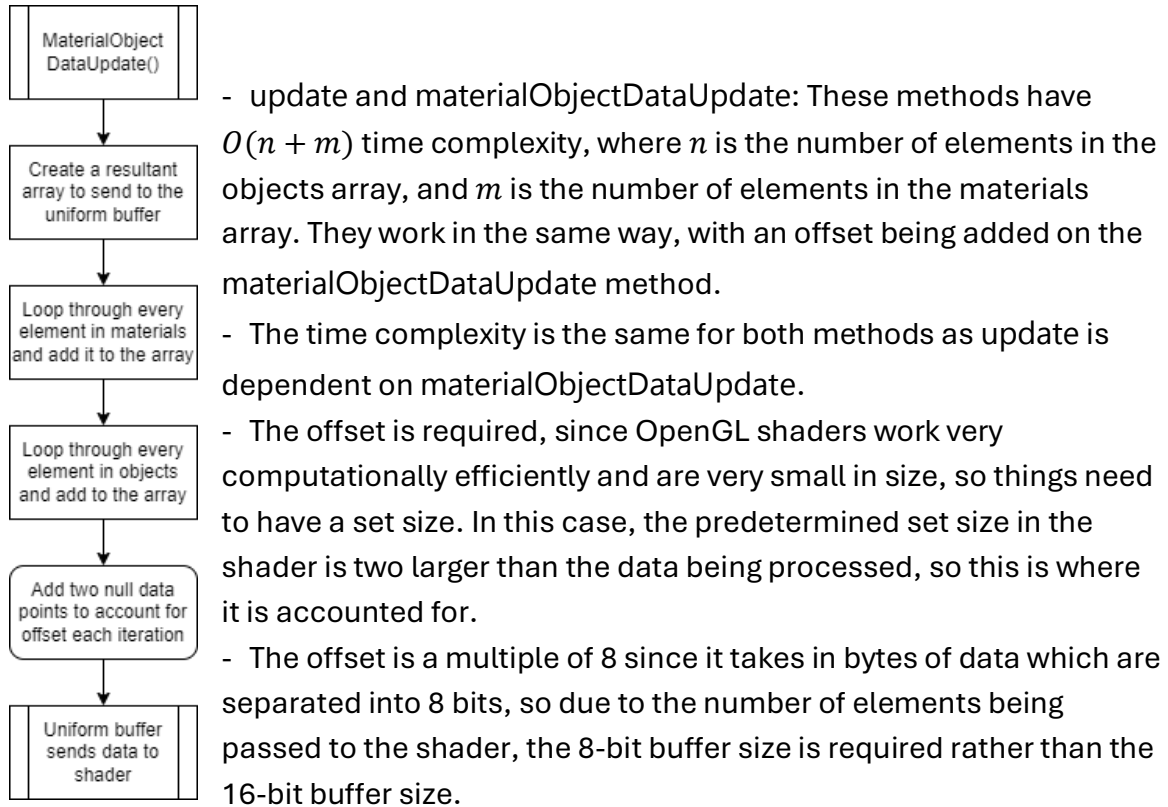
| FragVars                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> - res: std::vector&lt;float&gt; - aX: float - aY: float - lights: std::vector&lt;std::vector&lt;float&gt;&gt; - lightCols: std::vector&lt;std::vector&lt;float&gt;&gt; - materials: std::vector&lt;Material&gt; - objects: std::vector&lt;std::shared_ptr&lt;Object&gt;&gt; - time: std::chrono::steady_clock::time_point - resLocation: GLint - aXLocation: GLint - aYLocation: GLint - timeLocation: GLint - lightsLocation: GLint - lightColsLocation: GLint - materialsLocation: GLint - objectsLocation: GLint - lightLLocation: GLint - materialsLLocation: GLint - objectsLLocation: GLint - ub: UniformBuffer </pre>                                                   |
| <pre> - materialObjectDataUpdate(Program&amp; p): void + FragVars(std::vector&lt;float&gt;&amp; res, float aX, float aY, std::vector&lt;std::vector&lt;float&gt;&gt;&amp; lights, std::vector&lt;std::vector&lt;float&gt;&gt;&amp; lightCols, std::vector&lt;Material&gt;&amp; materials, std::vector&lt;std::shared_ptr&lt;Object&gt;&gt;&amp; objects) + init(Program&amp; p): void + update(Program&amp; p, float&amp; aX, float&amp; aY, std::vector&lt;Material&gt; materials, std::vector&lt;std::shared_ptr&lt;Object&gt;&gt;&amp; objects, std::vector&lt;std::vector&lt;float&gt;&gt;&amp; lights, std::vector&lt;std::vector&lt;float&gt;&gt;&amp; lightCols): void </pre> |

FragVars is a class made for altering variables in the GLSL fragment shader. It is used in the main class which provides it with the necessary setup data, and frame-to-frame data, and passes this data onto the fragment shader. Here is an explanation of its variables, all of which are private:

- res: The screen resolution, represented as a 2D vector of  $\{width, height\}$ .
- aX, aY: The angles about the  $x$  and  $y$  axes for rotational matrix calculations.
- lights: The 3D positions of each of the lights in the scene.
- lightCols: The RGB values of each of the lights in the scene.
- materials: A vector containing information about the possible materials for an object to be made from.
- objects: A vector containing information about the data surrounding each object, such as its position, speed, and velocity.
- time: A quantity used to calculate the time between frames for some calculations.
- [NAME]Location: The location of the named variable in the fragment shader, stored as a GLint.

- ub: A uniform buffer variable used for sending fragment variable data to a uniform in the shader.

The FragVars class has some algorithmically thought-out methods inside it, that I will explain here:



### Methods

- materialObjectDataUpdate(&p). This function has already been described above.
- FragVars(&res, aX, aY, &lights, &lightCols, &materials, &objects). This is the constructor for the FragVars class. It assigns each of the variables in its arguments to a private variable inside of the class.
- init(&p). This class is used for the initialisation of the FragVars class. It calls the function materialObjectDataUpdate and then gets the location inside the shader of each of the relevant variables. It also begins the chrono time clock.
- update(&p, &aX, &aY, materials, &objects). This is a regularly called function which updates the scene state inside of the fragment shader. It begins by updating the variables which it has been provided with by its arguments. It then sends the data to the shader along with calculating the time elapsed between frames. This function also uses materialObjectDataUpdate.

*Game*

| Game                                                                                                                                                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- Point{ float x, float y }: struct</li> <li>- aX: float</li> <li>- aY: float</li> <li>- lastX: float</li> <li>- lastY: float</li> <li>- g: float</li> <li>- r: float</li> <li>- f: float</li> <li>- firstMouse: bool</li> <li>- std::vector&lt;std::shared_ptr&lt;Object&gt;&gt; objects</li> </ul>                                                                    |
| <ul style="list-style-type: none"> <li>- screenToNDC(float x, float y): Point</li> <li>+ Game(std::vector&lt;std::shared_ptr&lt;Object&gt;&gt;&amp; objects)</li> <li>+ mouseEvent(GLFWwindow*, double xpos, double ypos, float g, float r): void</li> <li>+ updateObjects(std::vector&lt;std::shared_ptr&lt;Object&gt;&gt;&amp; objects): void</li> <li>+ getAX(): float</li> <li>+ getAY(): float</li> </ul> |

Game is a class made for handling how the user interacts with the main screen, in terms of mouse operations. Here is an explanation of its variables:

- Point{ float x, float y }. Point is a C++ struct which has two components: x and y. It is useful for describing where on the screen has been clicked, making the data about the click easily accessible.
- aX, aY. These are the angles about the x-axis and y-axis respectively which are responsible for the camera angle to the screen and is passed to the fragment shader which uses rotation matrices to rotate objects and the scene on the screen correctly.
- lastX, lastY. These are used for a class-global record of where the mouse was last frame and is used to calculate an offset to correctly give values of aX and aY.
- g, r, f. These are the values of gravity, the coefficient of restitution, and the coefficient of friction respectively.
- firstMouse. This is a class-global variable representing whether this is the first time that the mouse has clicked the screen in a specific instance.
- objects. This is the vector of objects used all over the code.

**Methods**

- screenToNDC(x, y). This is used for converting a Point from screen coordinates to NDC coordinates using the method specified in the UV-Coordinates section, as NDC coordinates are a form of UV coordinates.
- Game(objects). This is the constructor for the Game class, and it defines the local variable objects as the normal code-wide variable objects.

- `mouseEvent(window, xpos, ypos, g, r)`. This function is used as the OpenGL callback function for all mouse related events. Here is some pseudocode to represent what it does:

FUNCTION `mouseEvent`:

`xoffset = 0, yoffset = 0`

IF (mouse button released):

`firstMouse = true`

ENDFUNCTION

ENDIF

IF (`firstMouse`):

`lastX = xPos, lastY = yPos`

`firstMouse = false`

ENDIF

`xoffset = xPos – lastX`

`yoffset = yPos – lastY`

`lastX = xPos, lastY = yPos`

`aX += xoffset, aY += yoffset`

ENDFUNCTION

- This function also limits `aX` to be between  $\frac{\pi}{4}$  and  $-\frac{\pi}{4}$  radians.
- `updateObjects(objects)`. This function is used to update the private object array inside the function, it is a setter function.
- `getAX()`, `getAY()`. These are getter functions for the variables `aX` and `aY` respectively.

### *Global*

The header file `Global` is responsible for storing global variables that are used across a few files. These variables are:

- `PI`. The recorded float value for `PI` in the code is 3.14159265359. In IEEE 754 floating point binary (sign bit, 11-bit exponent, 52-bit mantissa), this is  
0 10000000000 10010010001100110011001100110011001100110011

. This number is exactly 3.141592653589793, so it is ever so slightly different to the value typed in, making it off by  $6.58 \times 10^{-12}\%$ .

- WINDOW\_WIDTH, WINDOW\_HEIGHT. The first value is 1920, which is true for most monitors, and the second value is 1080.

### GUI

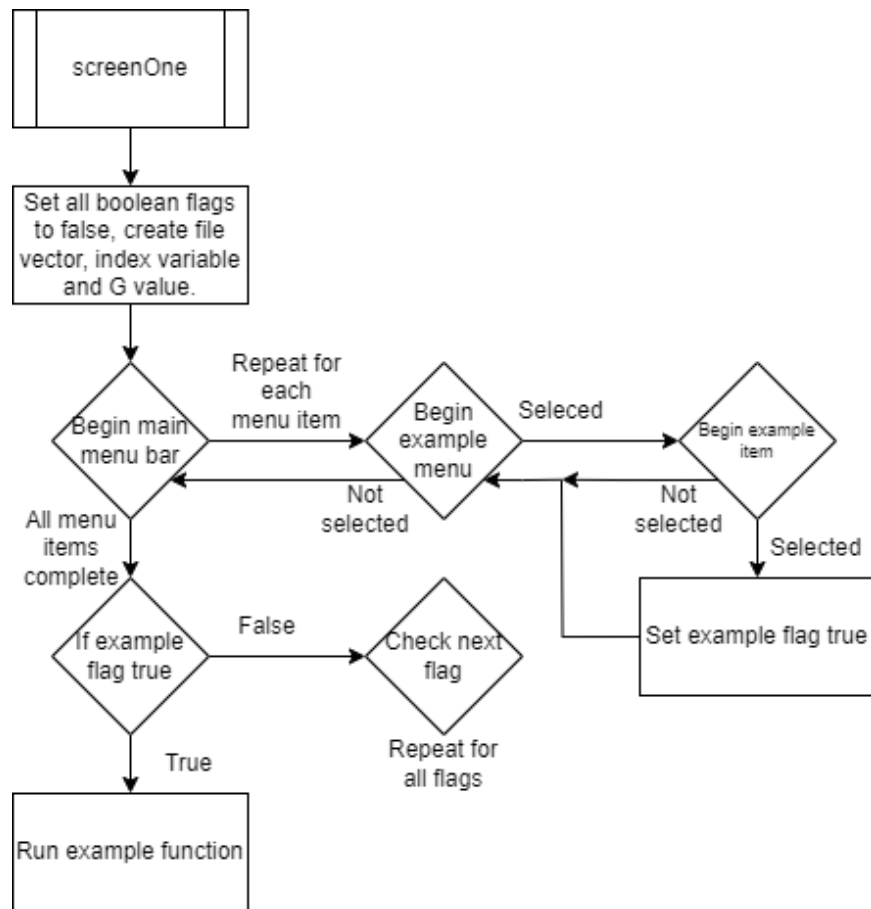
| GUI                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| - width: int<br>- height: int                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| - clampFloat(float& f, float min, float max): void<br>- serializeMaterials(std::vector<Material>& materials): std::string<br>- serializeObjects(std::vector<std::shared_ptr<Object>>& objects): std::string<br>- serializeScene(std::vector<Material>& materials, std::vector<std::shared_ptr<Object>>& objects): std::string<br>- deserialize(std::string data, std::vector<Material>& materials, std::vector<std::shared_ptr<Object>>& objects): void<br>+ GUI(int width, int height)<br>+ screenOne(GLFWwindow* window, std::vector<std::shared_ptr<Object>>& objects, std::vector<Material>& materials, float &r, float &f, float &g, float fps, float &g, std::vector<std::string> tArr, std::vector<std::string> tMatArr, std::vector<std::vector<float>> lights, std::vector<std::vector<float>> lightCols): void |

GUI is the class responsible for the whole interactive user interface used to add objects and materials to the scene. Much of this class has already been talked about while talking about the GUI. The variables width and height are used for the width and height of the screen respectively.

### Methods

- clampFloat(&f, min, max). This function is used to clamp a float value between two specified values. It works by seeing if the float is greater or less than the minimum and maximum values and setting it to the minimum value if it is less than the minimum or setting it to the maximum value if it is greater than the maximum.
- serializeMaterials(&materials). This function, which has been talked about previously, turns the materials vector into a form which can be written to a file.
- serializeObjects(&objects). This function, which has been talked about previously, turns the objects vector into a form which can be written to a file.
- serializeScene(&materials, &objects). This function calls both the serializeObjects and serializeMaterials functions and returns the data.
- deserialize(data, &materials, &objects). This function, which has been spoken about previously, deserializes the data from a file and writes it to the materials and objects vectors.
- GUI(width, height). This is the constructor for the GUI class, which assigns the argument variables width and height to their private equivalents.

- screenOne(window, &objects, &materials, &r, &f, &g, fps, tArr, tMatArr, lights, lightCols). This function is the main method for the GUI. Here is a flow chart representing how it works:



### IndexBuffer

| IndexBuffer                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------|
| - m_handle: GLuint<br>- m_number: unsigned                                                                                      |
| + IndexBuffer()<br>+ ~IndexBuffer()<br>+ handle(): GLuint<br>+ fill(std::vector<GLuint>& indices): void<br>+ number(): unsigned |

IndexBuffer is a class which sends the data surrounding where the triangles from the vertex buffer are located to the graphics card. The variables included in this class are:

- `m_handle`. This is the identifier for the IndexBuffer which is used for binding the buffer data to be sent.
- `m_number`. This is an unsigned integer which represents the number of elements being sent using this buffer.



### Methods

- IndexBuffer(). This is the constructor for the IndexBuffer class. This sets the initial m\_number to 0 and generates a buffer object ready to be bound to.
- ~IndexBuffer(). This is the destructor for the class, it first tests if the handle is non-zero, and if it is it deletes the buffer.
- handle(). This is a getter function for the variable m\_handle.
- fill(&indices). This fills the index buffer. It first sets m\_number to the number of elements in the vector indices and binds the buffer ready to be sent.
- number(). This is the getter function for the variable m\_number.

### Material

Material is a header file which contains the C++ struct which defines the aspects of a material. Namely:

- The RGB value of the material.
- The ambient, diffuse and specular coefficients for Phong lighting.
- The reflective and shininess constants.

### Object and Sphere

| Object                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| # data: ObjectData<br># vOps: VecOps                                                                                                                                                                                             |
| # SDF(): float<br>+ Object(ObjectData data)<br>+ update(float f): void<br>+ updateObject(float g, float r): void<br>+ checkCollision(Object& other): bool<br>+ resolveCollision(Object& other): void<br>+ getData(): *ObjectData |

The object class is used for storing the relevant data for an object and for controlling how they move and checking along with resolving collisions. It has only one subclass for expandability; more subclasses of different object types could be added. This class also has a subclass specifically for spheres. Here are the variables used in this class:

- data. This is a variable that has a type of ObjectData, which is a C++ struct storing all the information about an object at a given point in time. Namely:
  - Its type.
  - Its material.
  - Its position.
  - Its defining length.
  - Its mass.
  - Its velocity.
  - Its impulse.

- viii. Whether it is facing down.
- ix. Whether it is moving.
- x. Whether it is on the floor.
- vOps. This is an instance of the class VecOps, allowing many different vectors in the object class to be manipulated.

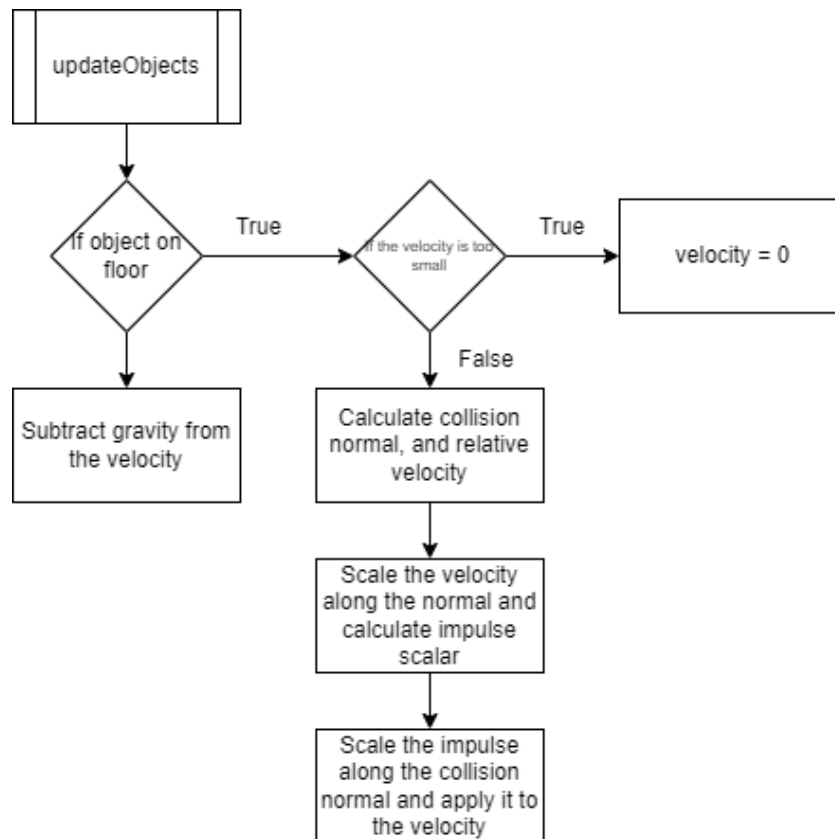
### Subclass Sphere

| Sphere                                                           |
|------------------------------------------------------------------|
| # SDF(vec p, vec c, float r): float<br>+ Sphere(ObjectData data) |

Sphere is the subclass of Object that is concerned with data solely for objects that are spheres. It has no private variables, but it can access the protected variables within Object.

### Object methods

- SDF(). This function returns 0 but is overridden through polymorphism in Sphere.
- Object(data). This is the constructor for the class and initialises the protected variable data with values specified in the argument.
- update(f). This applies friction to an object when it is on the floor, by checking if the centre of the object minus its length is less than the floor height. It then applies the velocity of the object to its position.
- updateObject(g, r). This function serves to apply gravity to the objects. Here is a flow diagram showing how it works (the velocity is set to 0 if it is too small due to floating point errors):



- `checkCollision(&other)`. This method has another object passed into its argument and verifies whether they are currently colliding. Here is some pseudocode showing how it works:

FUNCTION `checkCollision`:

if (`other == this object`):

    return FALSE END FUNCTION

dist = subtract object positions

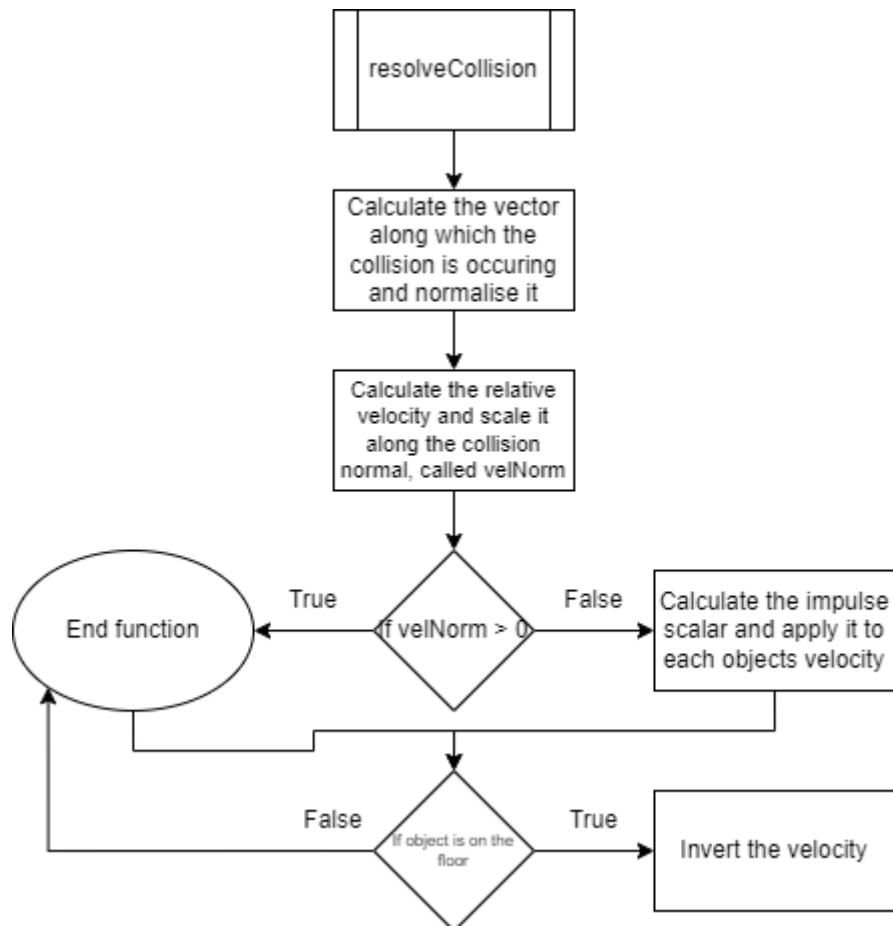
if (`dist < difference between lengths`):

    return TRUE

return FALSE

ENDFUNCTION

- `resolveCollision(&other)`. This function resolves a collision between one object and the object provided in the argument. Here is a flow chart describing how it works, based off the maths in the research section.



- `getData()`. This is a getter function which returns the `ObjectData` as a pointer variable.

### Sphere Methods

- `SDF(p, c, r)`. This is an overridden polymorphed function which takes arguments of the position, centre of the sphere, and the radius of the sphere. It uses the sphere SDF equation referenced earlier in the research section.

### Program

| Program                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| - m_handle: GLint                                                                                                                                      |
| + Program()<br>+ ~Program()<br>+ handle(): GLint<br>+ attachShader(Shader& shader): void<br>+ detachShader(Shader& shader): void<br>+ activate(): void |

The program class is used for attaching and detaching shaders to properly run the main OpenGL scene. It is responsible for the majority of what is scene on screen. The

variable `m_handle` is used as the OpenGL identifier for how to access and use the program inside of OpenGL methods.

### Methods

- `Program()`. This is the constructor for the class. It generates a handle and assigns the returned value to `m_handle`.
- `~Program()`. This is the destructor for the class. It first checks to see if `m_handle` is non-zero, and if it is, it deletes the program and resets the `m_handle`.
- `handle()`. This is a getter function for the `m_handle` variable.
- `attachShader(&shader)`. This runs the OpenGL command to attach a shader to a program handle.
- `detachShader(&shader)`. This runs the OpenGL command to detach a shader from a program handle.
- `activate(&shader)`. This function brings the program, and thus the main scene to life. Here is some pseudocode explaining it:

FUNCTION `activate()`:

link the program

status = ask OpenGL if the program has linked

if (status != TRUE):

create a character string buffer

get a log of what went wrong

ENDIF

tell OpenGL to use the program

ENDFUNCTION

### Shader

| Shader                                                                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| - <code>m_handle: GLint</code><br>- <code>m_inError: bool</code><br>- <code>m_error: std::string</code><br>+ <code>Type: enum { VERTEX, FRAGMENT }</code>                                    |
| + <code>Shader(Type type, char* fileName)</code><br>+ <code>~Shader()</code><br>+ <code>handle(): GLint</code><br>+ <code>inError(): bool</code><br>+ <code>error(): std::string&amp;</code> |

The Shader class is concerned with compiling shaders from a shader file into a form which OpenGL can understand. Here is what each of its variables are for:

- `m_handle`. This is the identifier which uniquely defines a shader.
- `m_inError`. This is a Boolean variable which determines whether there has been an error in compilation.
- `m_error`. This is the string value which stores the error if there has been one in compilation.
- `Type`. This is an enum variable of either VERTEX or FRAGMENT to define to the shader class whether it is dealing with a vertex or a fragment shader.

#### Methods

- `Shader(type, fileName)`. This is the constructor for the class. It begins by setting the initial values of `m_handle`, `m_inError` and `m_error`. It then sets the type based on the enum type specified in the arguments. It then creates the relevant shader based on this type and reads in the shader file source. The shader is then attempted to be compiled, and the status is checked. If any error occurred, `m_inError` is set to true, and `m_error` is set to the error code.
- `~Shader()`. This is the destructor for the class. It first checks to see if `m_handle` is non-zero, and if it is, it deletes the shader and resets the `m_handle`.
- `handle()`. This is a getter function for the variable `m_handle`.
- `inError()`. This is a getter function for the variable `m_inError`.
- `error()`. This is a getter function for the variable `m_error`.

#### UniformBuffer

| UniformBuffer                                                                                                                                                                                                                                                                 |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- <code>m_handle</code>: GLuint</li> <li>- <code>m_number</code>: unsigned</li> </ul>                                                                                                                                                  |
| <ul style="list-style-type: none"> <li>+ <code>UniformBuffer()</code></li> <li>+ <code>~UniformBuffer()</code></li> <li>+ <code>handle()</code>: GLuint</li> <li>+ <code>fill(std::vector&lt;float&gt;&amp; data, Program&amp; p, GLchar&amp; number): void</code></li> </ul> |

The UniformBuffer class is used for binding a large chunk of data to allocated to space in an OpenGL shader. The variables used are:

- `m_handle`. This is the identifier which uniquely defines a UniformBuffer object.
- `m_number`. This is a variable which represents the size of the data in transfer.

#### Methods

- `UniformBuffer()`. This is the constructor for the class. It assigns an initial value of 0 to `m_number` and creates a buffer which is assigned to `m_handle`.

- `~UniformBuffer()`. This is the destructor for the class. It first checks to see if `m_handle` is non-zero, and if it is, it deletes the `UniformBuffer` object and resets the `m_handle`.
- `handle()`. This is the getter function for the variable `m_handle`.
- `fill(&data, &p, &number)`. This function is responsible for sending data to the bind point inside of the fragment shader. It first binds the buffer with the `m_handle`, and then fills the buffer with the necessary data. It finds where the `bindPoint` is in the code, and if it can be found, then it binds the data to it and transmits it.

### VecOps

| VecOps                                                                                                                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| + <code>add(vec v1, vec v2): vec</code><br>+ <code>dot(vec v1, vec c2): float</code><br>+ <code>cross(vec v1, vec v2): vec</code><br>+ <code>scale(vec v, float s): vec</code><br>+ <code>length(vec v): float</code><br>+ <code>normalise(vec v): vec</code> |

The `VecOps` class is responsible for all mathematical vector-based calculations. It has no public or private variables in the class, since it uses an external C++ struct called `vec` which has components `x`, `y` and `z`, which are all floats.

### Methods

- `add(v1, v2)`. This function performs the vector sum operation:  $\vec{v}_1 + \vec{v}_2$ . It adds the constituent `x`, `y` and `z` components.
- `dot(v1, v2)`. This function performs the vector dot product operation:  $\vec{v}_1 \cdot \vec{v}_2$ . It adds the product of the constituent `x`, `y` and `z` components.
- `cross(v1, v2)`. This function performs the vector cross product operation:  $\vec{v}_1 \times \vec{v}_2$ . It is calculated by the determinant of this matrix:

$$\begin{bmatrix} x & y & z \\ v_{1x} & v_{1y} & v_{1z} \\ v_{2x} & v_{2y} & v_{2z} \end{bmatrix}$$

- `scale(v, s)`. This function multiplies a vector by a scalar, increasing only its length.
- `length(v)`. This function returns the length of a vector using the Pythagorean formula. This is useful for getting the speed from a velocity vector for example.
- `normalise(v)`. This function scales a vector by 1 over its length, giving it length 1 but the same direction.

*VertexBuffer*

| VertexBuffer                                                                                          |
|-------------------------------------------------------------------------------------------------------|
| - m_handle: GLuint                                                                                    |
| + VertexBuffer()<br>+ ~VertexBuffer()<br>+ handle(): GLuint<br>+ fill(std::vector<float>& data): void |

VertexBuffer is the last buffer class which is used to send data about the vertices of the triangles to the graphics card. The variable m\_handle is the identifier which uniquely defines a VertexBuffer object.

*Methods*

- VertexBuffer(). This is the constructor for the VertexBuffer class. It generates a buffer with m\_handle and binds it to an array buffer.
- ~VertexBuffer(). This is the destructor for the class. It first checks to see if m\_handle is non-zero, and if it is, it deletes the VertexBuffer object and resets the m\_handle.
- handle(). This is the getter function for the m\_handle variable.
- fill(&data). This function sends the argument data to the graphics card via the created buffer.

*Other Files**Fragment*

Fragment is the fragment shader that deals with the rendering of everything the user can see to do with the main scene. It is an OpenGL shader, so passing data to it is harder than passing data to a regular class with normal functions. This is due to it being much more memory and data efficient.

*Variables*

- fragColor. This is the eventual calculated colour of an area on the screen.
- res. This is the screen resolution.
- lights, lightCols. These are the arrays which describe the light positions and colours for Phong lighting.
- time. This is the time step that is regularly updated.
- aX, aY. These are the angles about the x and y-axes respectively for rotation.
- lightsL, materialsL, objectsL. These are the lengths of all the necessary arrays in the shader. These are necessary due to how precise shaders are with how much memory they are using.



- materials. A vector containing information about the possible materials for an object to be made from.
- objects. A vector containing information about the data surrounding each object, such as its position, speed, and velocity.

### Methods

- intersectsBoundingVolume(ro, rd, c, r). This method determines whether an area of an SDF should be rendered depending on its position. Here is some pseudocode representing how it works:

FUNCTION intersectsBoundingVolume:

oc = rayOrigin – centre

b = dotProduct(oc, rd) // Project oc along ray direction

c1 = dotProduct(oc, oc) – r \* r // Quadratic discriminant

return b \* b – c1 >= 0 // Rest of discriminant for validation on whether it should be rendered.

ENDFUNCTION

- The discriminant ( $\Delta$ ) must be greater than or equal to 0 because that means there is real solutions, so it intersects once or twice, meaning that it must be rendered.
- dynamicStepSize(dist, minStep, maxStep). This method adjusts the step length in raymarching based on a set minimum and maximum, it is a clamping function.
- sphereSDF(p, c, r, i), planeSDF(p, n, h, i). These methods are the signed distance functions for a sphere and a plane. The sphere one has been defined in the research section, but the plane one is defined as follows:

$$\vec{p} \cdot \hat{n} + h$$

- Where  $n$  is the plane normal and  $h$  is its defining length.
- unionSDF(SDF1, SDF2), intersectSDF(SDF1, SDF2), subSDF(SDF1, SDF2). These methods apply the theory talked about in the research section with combining SDFs.
- rotateX(a), rotateY(a). Rotates an SDF, and thus the scene, by a given angle  $a$  about the x and y-axes. These apply the rotation matrices discussed earlier in the research section.
- rotateSDF(p, x, y). Rotates an SDF by given angles  $x$  and  $y$  about their axes, doing the y rotation multiplied by the x rotation, as matrix multiplication is not commutative. It applies the rotation to the position being checked.

- `translateSDF(p, t)`. Translates an SDF by a vector `t` by subtracting `t` from the position being checked.
- `finalSDF(p, rd)`. This method first defines the resultant SDF at a given point to be the floor, and checks if at that point there is another SDF using a 3-choice switch-case statement. Any other SDFs found will have the `unionSDF` applied alongside the initial floor SDF.
- `calculateNormal(p, rd)`. This method calculates the normal at a given point to the `finalSDF`. This is calculated by taking the definition of partial derivative very literally, by adding a very small amount along each axis.
- `isInShadow(p, rd, dist)`. This method determines whether a point `p` is in the shadow of another SDF using raymarching, as explained in the `rayMarch` method below. The only difference is the starting point instead of being the normal `rayOrigin`, will be `p`.
- `getCol(la, m, li, n, li, v, p)`. This method applies the Phong reflection model to a point `p` based on the material properties, the light positions (`li`) and their colours (`li`), along with the line along which it is viewed. The computation calculates the ambient, specular, and diffuse components, and then sums them, as shown in this pseudocode:

FUNCTION `getCol()`:

`a = initialAmbience, d = (0,0,0), s = (0,0,0)`

FOR EACH light `l` IN `lights`:

`light = normalise(l-p)`

`dist = length(light)`

IF (NOT `isInShadow()`):

`r = reflect(-light, n)`

`d += max(n.dot(light), 0) * Kd * l.color * RGB`

`s += Ks * (max(r.dot(v.normalise()), 0)^(c) * l.color`

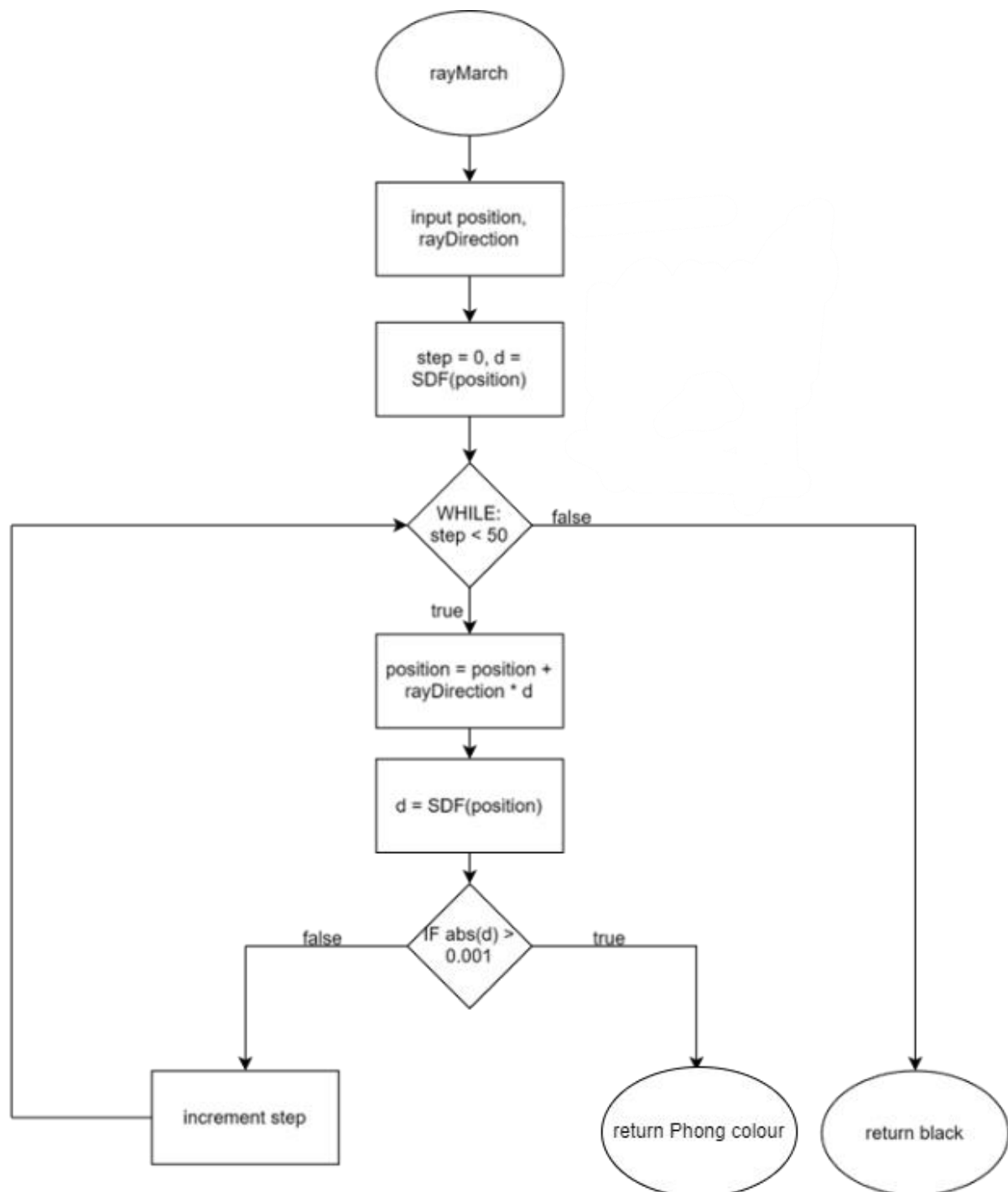
ENDIF

ENDFOR

ENDFUNCTION

- The diffuse and specular are taken as the maximum dot product versus 0 because the light is scaled along the normal, giving the value, but ensuring it isn't negative. If it is negative, it is facing the wrong way.

- rayMarch(ro, rd, maxDistance). This method implements the theory behind raymarching into a GLSL function. Here is a flow diagram describing how it works:



- checkerFloor(p, n). This function is what produces the checkered floor in the main scene. Here is some pseudocode describing how it works:

FUNCTION checkerFloor():

checker = (floor(p.x) + floor(p.y)) MOD 2

```
IF (checker == 0):
 baseColor = white
ELSE:
 baseColor = black
ENDIF

FOR (Light l in lights):
 lightDirection = normalise(l - p)
 distToLight = length(l - p)
 IF (isInShadow(p + n * 0.001, lightDirection, distToLight)):
 RETURN mix(baseColor, grey)
 ENDIF
ENDFOR

RETURN baseColor
ENDFUNCTION
```

- sortCol(ro, rd, maxDist). This method applies rayMarch, getCol, and checkerFloor to produce the resultant colour for an area of the screen.
- main(). This is the main method of the fragment shader, which converts the fragment coordinates to UV coordinates, gets the ray origins and directions, performs matrix rotations, and sorts the colour of areas on the screen, all by calling the relevant methods.

### *Vertex*

Vertex is the vertex shader for the OpenGL environment. It is a very small file, since all it does is sets the position of the vertices to gl\_Position, which allows OpenGL to render them.

### *Main*

This is the main file that gets run when the code begins. It begins by initialising some variables which will be used in a lot of places in the code. Namely:

- i. materials. This is the vector of materials which gets sent to the fragment shader.
- ii. objects. This is the vector of objects, which can be objects of any subtype that gets sent to the fragment shader.

- iii. `game`. This is an instance of the `Game` class, which controls handling screen interactions.
- iv. `frameCount`. This is a variable used for calculating the frame rate.
- v. `prevTime`. This is a variable used for also calculating the FPS. It is not a local variable because it stores data about the previous frame.
- vi. `g`, `r`, `f`. These are the values of gravity, restitution, and friction respectively.
- vii. `fps`. This variable stores the frames per second which is passed to the GUI class.
- viii. `text`, `matText`. These variables store debug information about objects and materials respectively and pass it to the GUI.

### Methods

- `mouse(window, xpos, ypos)`. This method assigns the OpenGL mouse callback function to the method `mouseEvent` in `game`. It only does this if the mouse is not interacting with any part of the `ImGui` GUI.
- `key(window, key, scancode, action, mods)`. This method assigns the escape key to a procedure for closing the window and assigns the J key to a procedure for applying random velocities to the objects.
- `printObjectData(o)`, `printMaterialData(m)`. These methods append the data about an object `o`, or a material `m` to their respective text variables.
- `update()`. This method is responsible for ensuring the physics of the scene is applied to each object, and that the `fps` is calculated. It begins by calculating the frame rate, then clears the debug texts, and removes any objects that are out of bounds. Then, for each object it calls the `updateObject` procedure, and prints the object data and material data. It then iterates through the object array to check for and resolve collisions.
- `updateObjectDatas()`. This method updates the object data of each object after any physics calculations have been computed.
- `main( argc, argv)`. This method is the main method of the code. Here is some pseudocode to describe how the code works and in what order:

FUNCTION `main()`:

    generate random seed

    initialiseGLFW()

    IF (failed to initialise GLFW):

        RETURN -1

    ENDIF

```
mode = getPrimaryMonitorData()
IF (mode == NULL):
 glfwTerminate()
 RETURN -1
ENDIF

retrieve mode information
window = create window
IF (window creation failed):
 glfwTerminate()
 RETURN -1
ENDIF

set current screen to OpenGL window
define keyboard callback as key()
define mouse callback as mouse()
initialise GLEW
IF (GLEW initialisation failed):
 glfwTerminate()
 RETURN -1
ENDIF

initialise ImGui
compile shaders
p = attach shaders to program
define lights, lightCols, materials
fvs = initialise FragVars
vertices = {-1, -1, 1, -1, -1, 1, 1, 1}
fill vertexBuffer with vertices
```

indices = {0, 1, 2, 2, 1, 3}

fill indexBuffer with indices

send data to vertex shader

fvs.init(p)

initialise GUI

WHILE MAIN LOOP:

poll GLFW events

create new ImGui frame

activate program

update()

updateObjectDatas()

fvs.update()

clear OpenGL buffers

draw OpenGL triangles

render GUI

refresh OpenGL buffers

ENDWHILE

destroy ImGui

destroy window

glfwTerminate()

RETURN 0

ENDFUNCTION