

OCR

OpenGL Real-Time Raymarching

Coursework

Ben Barton-Foskett

Contents

Analysis	3
Introduction.....	3
Reduction to Triangles	3
Research	3
Application	9
Signed Distance Function (SDF) Rendering.....	12
Research	12
Application	19
Lighting and Reflections.....	23
Research	23
Application	26
Rigid Body Dynamics	31
Research	31
Application	32
Modelling	33
Introduction.....	33
Scenes.....	33
The Graphical User Interface	34
Objectives	35
Design	36
Introduction.....	36
External Libraries	37
System Overview	37
Algorithms and Modules	42
Classes.....	42
Other Files	58
Testing.....	65
Iterative Testing	65
System and Unit Testing.....	66
Evaluation.....	70
Overall Evaluation	70

Evaluation Specifics	70
Objective Evaluation	70
Individual Further Improvements.....	71
User Feedback.....	72
Further Improvements Based on User Feedback.....	73

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. Furthermore, I want the resulting simulation to be intuitive to interact with.

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);
```

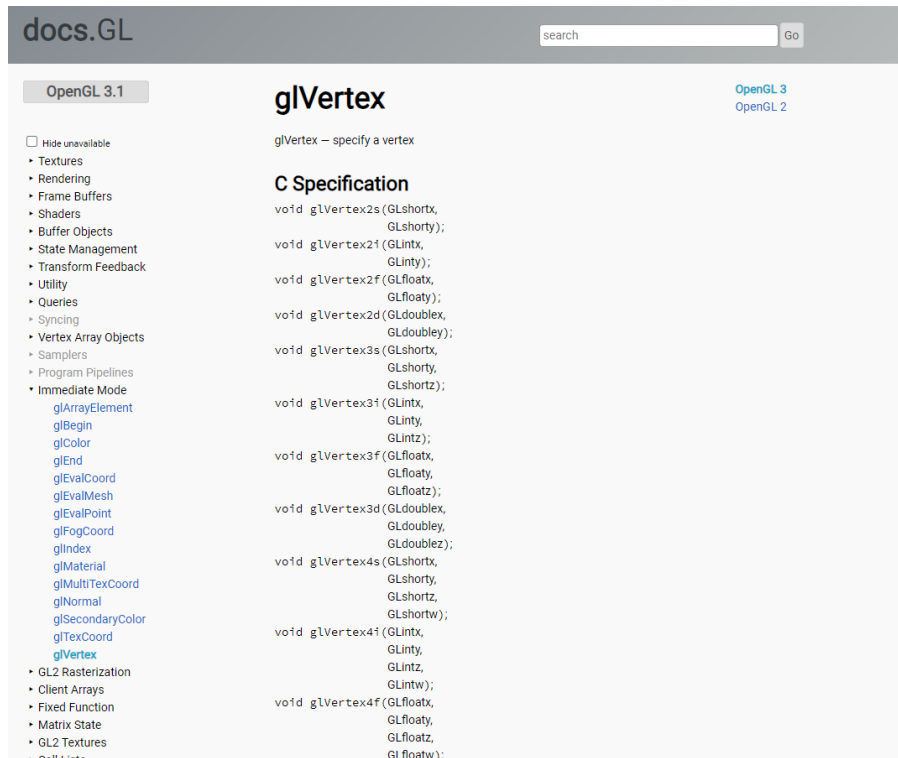
Since we are working in a windowing system most of the interaction with the running program occurs via event callback functions. GLUT takes care of interacting with the underlying windowing system and provides us with a few callback options. Here we use just one - a "main" callback to do all the rendering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

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

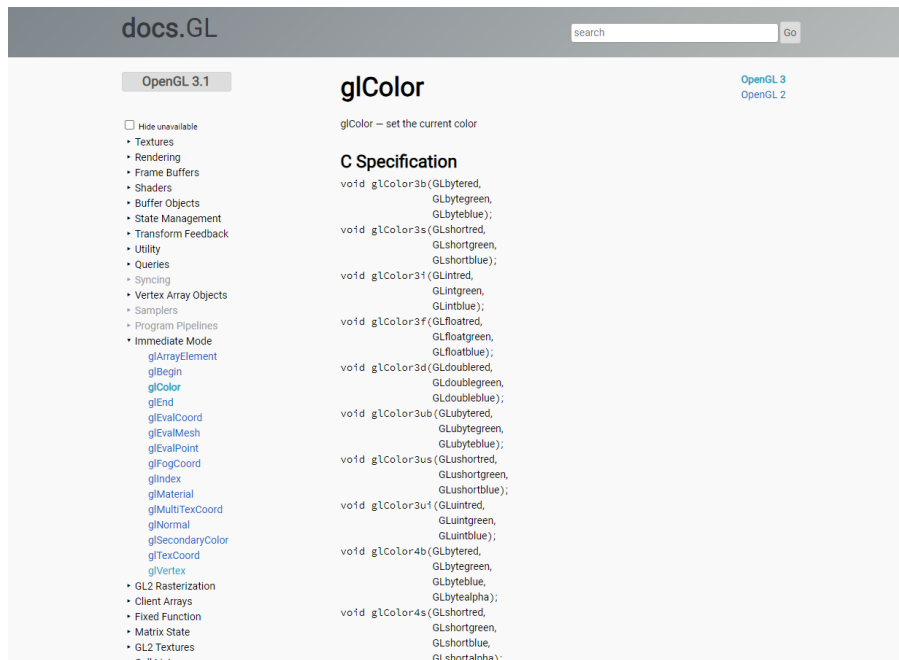
This is our first encounter with the concept of state in OpenGL. The idea behind state is that rendering is such a complex task that it cannot be treated as a function call that receives a few parameters (and correctly designed functions never receive a lot of parameters). You need to specify shaders,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



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:

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

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

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

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:

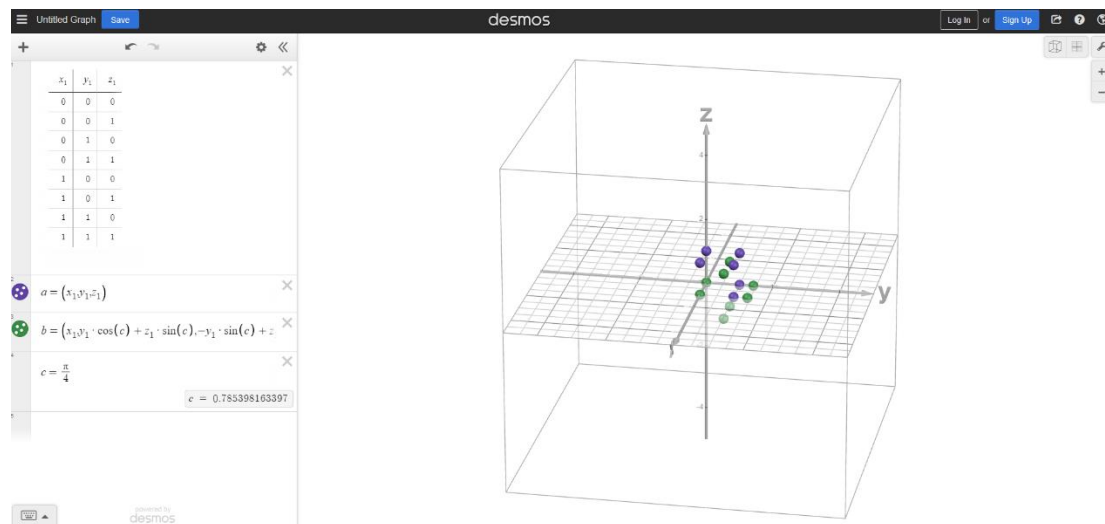
$$\begin{pmatrix} x \cdot \cos(\theta) - z \cdot \sin(\theta) \\ y \\ x \cdot \sin(\theta) + z \cdot \cos(\theta) \\ 1 \end{pmatrix}$$

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

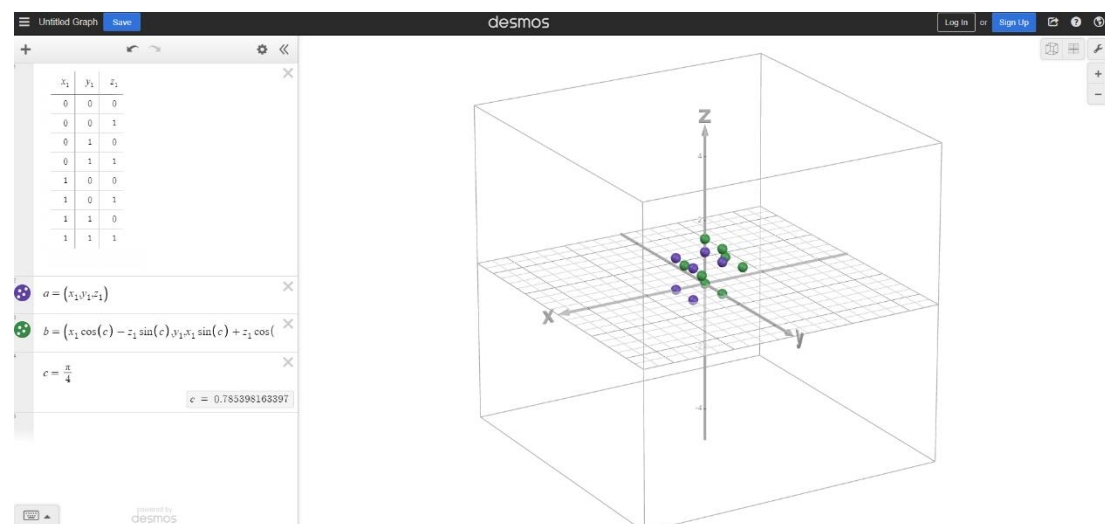
Applying this to a unit cube in Desmos gives the following results:

x-axis:

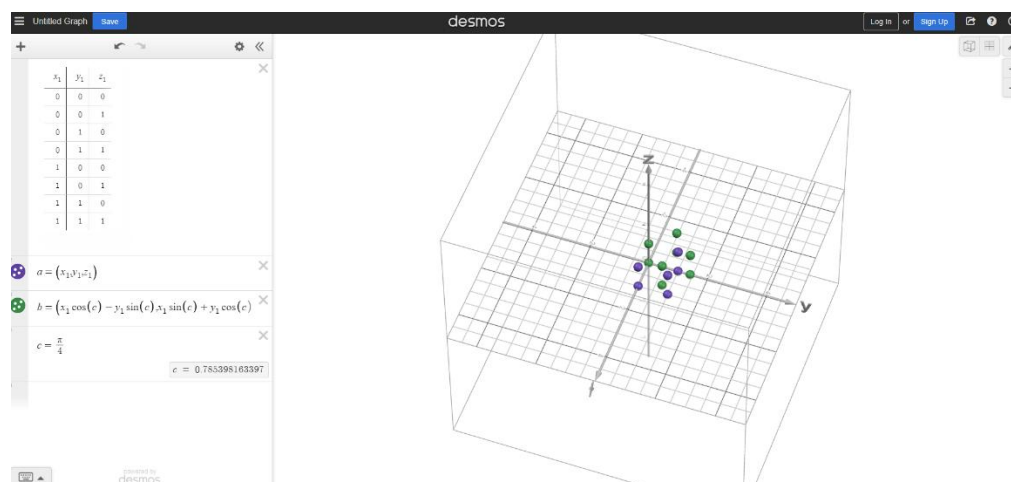
OpenGL Real-Time Raymarching



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.

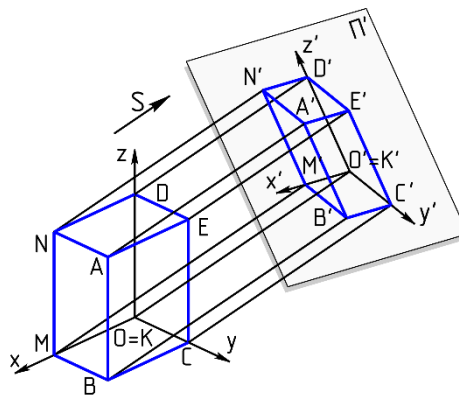


Fig. 1

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

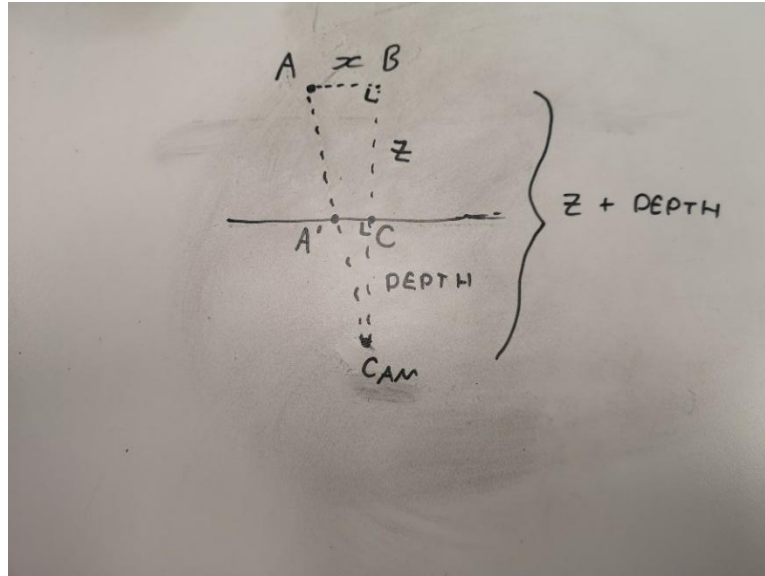


Fig. 2

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.

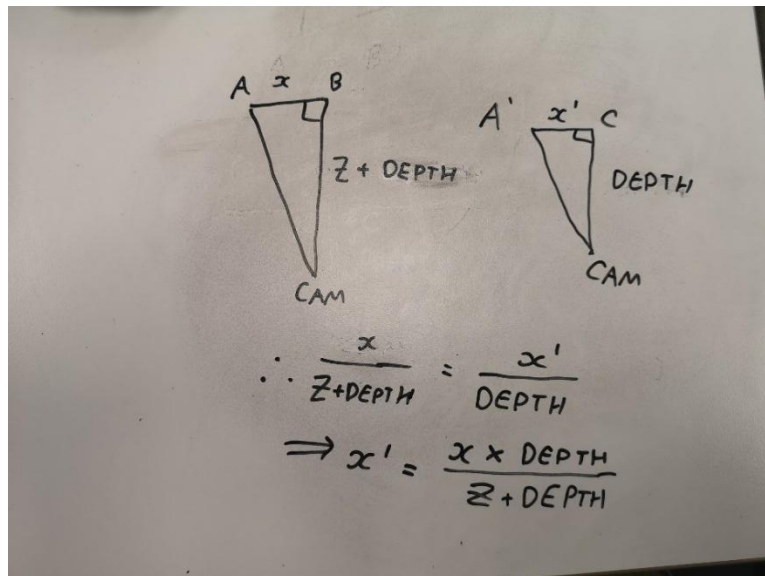


Fig. 3

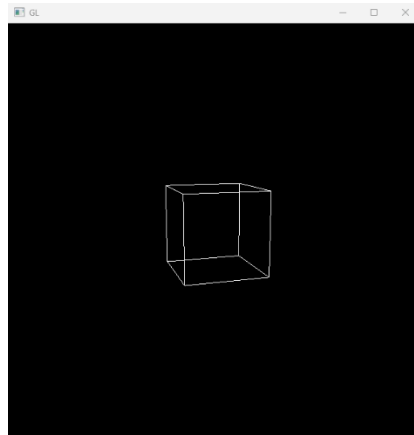
This gives that the equation of a projected coordinate from 3D space to $z = 0$ is:

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

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:



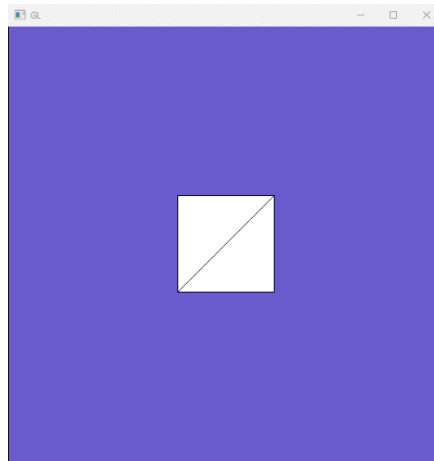
So, as we can see depth, the perspective projection is working.

Rotation Matrices

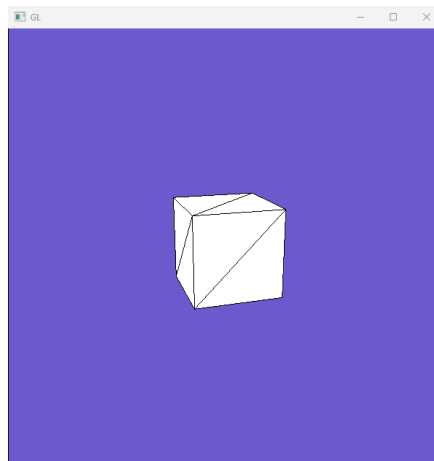
The rotation matrices mentioned in the research section can be applied to this projection now to allow for rotations in real time. Below is an example header class for an affine matrix class in C++:

```
#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

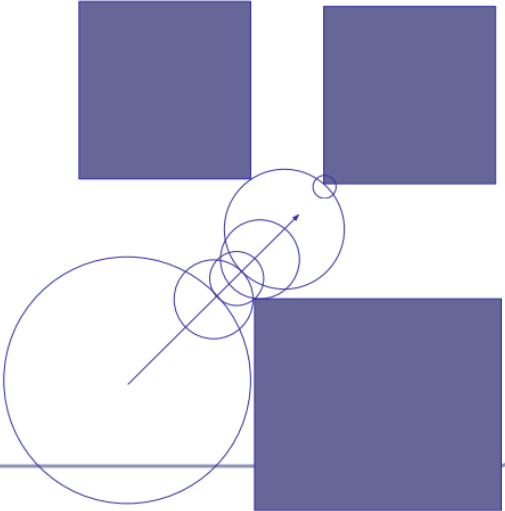
across a University of Cambridge PowerPoint on signed distance functions (SDFs)⁴, where a slide of this is shown below.

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:

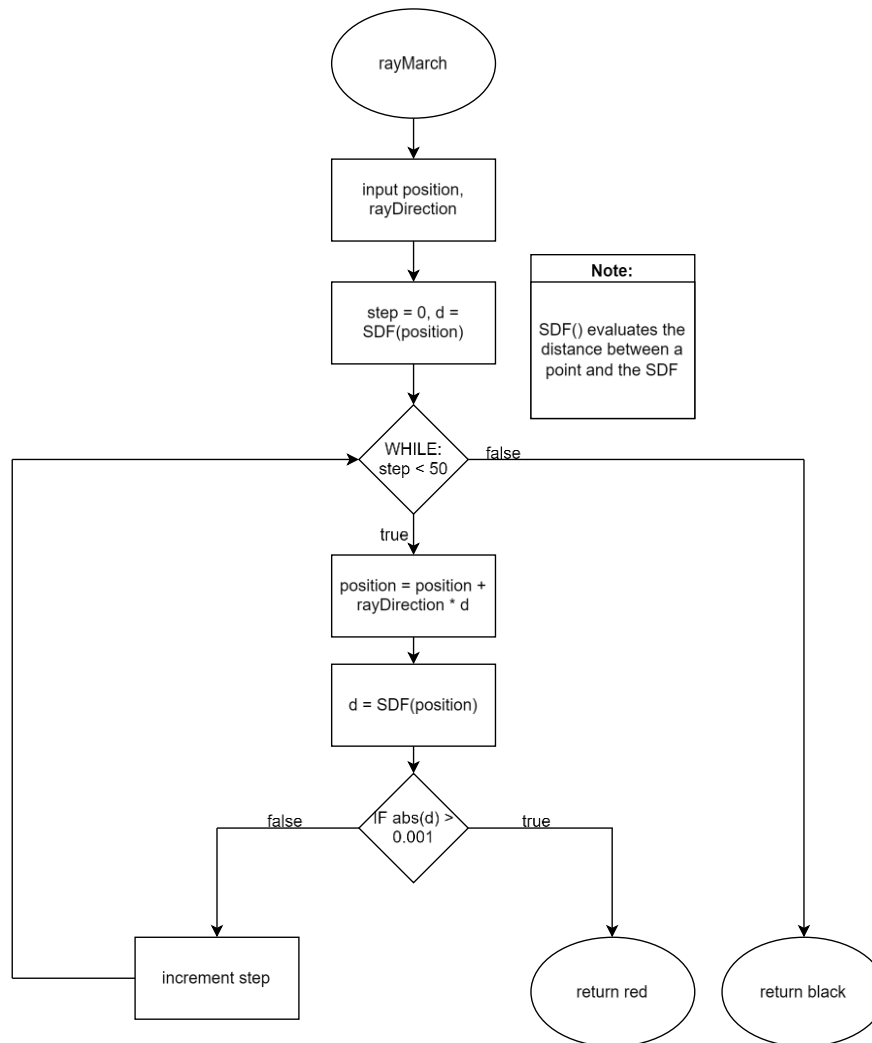


Fig. 4 below gives a nice illustration of what this algorithm is doing.

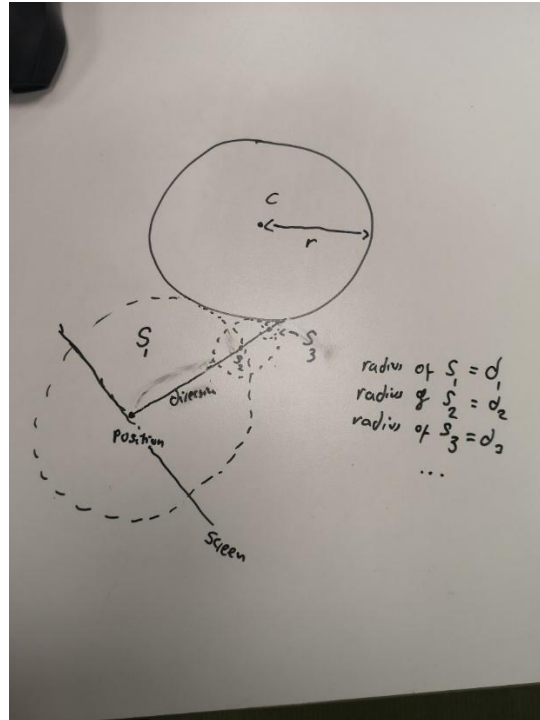


Fig. 4

SDF Arithmetic

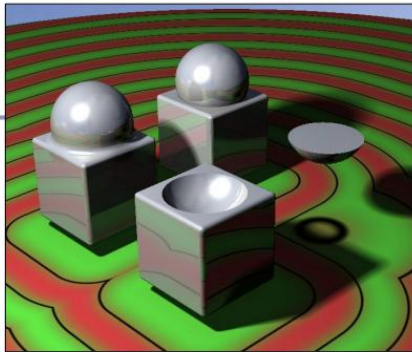
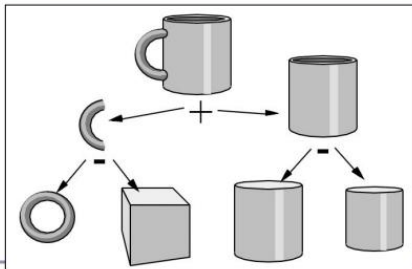
Next, to render more complicated SDFs and render multiple at once, I researched SDF arithmetic, which also lead me back to the Cambridge PowerPoint with the slide shown below.

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:

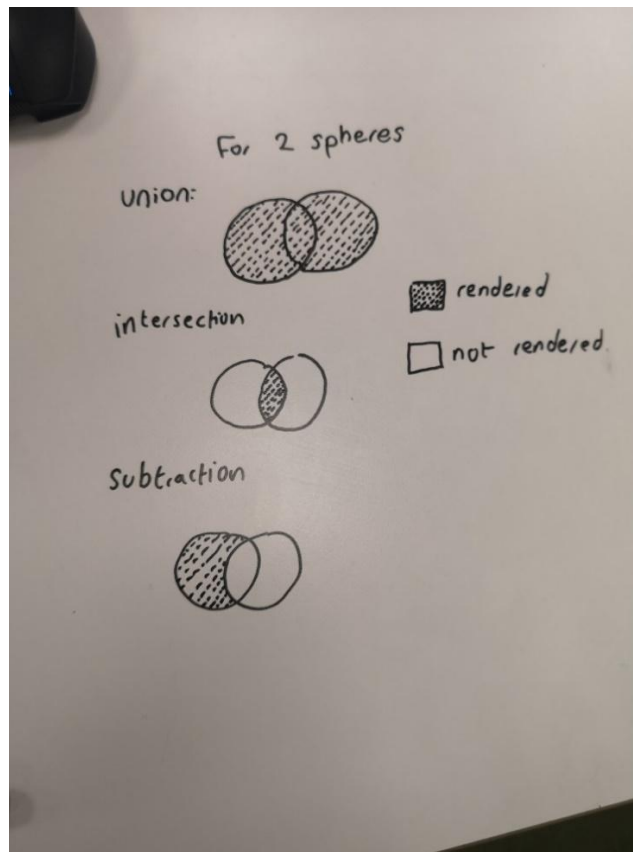


Fig. 5

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 GLFW⁷ environment and another video for GLEW⁸. 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.
- `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.

⁷ 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].

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

- `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 GLSL⁹, 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 maths¹⁰. 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:

```
int main() {  
    glfwInit()  
    auto window = glfwCreateWindow(WINDOW_WIDTH, WINDOW_HEIGHT, "openGl",  
    NULL, NULL);  
    glfwMakeContextCurrent(window);  
    glewInit()  
    Shader vertexShader(Shader::Type::VERTEX, "./vertex.glsl");
```

⁹ 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].

¹⁰ 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].

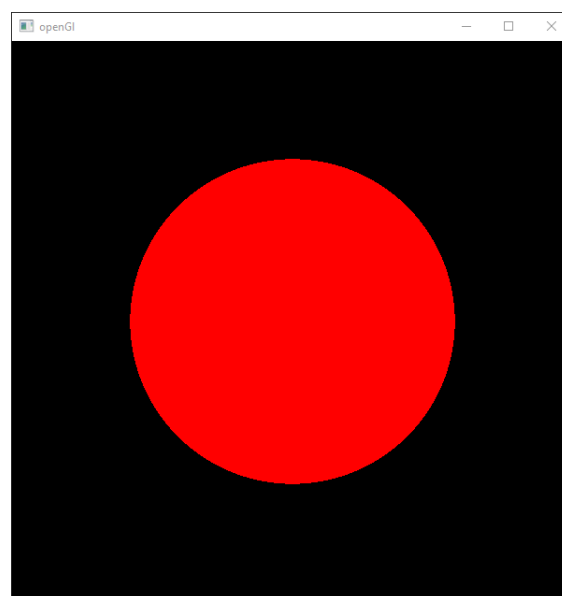
```

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);

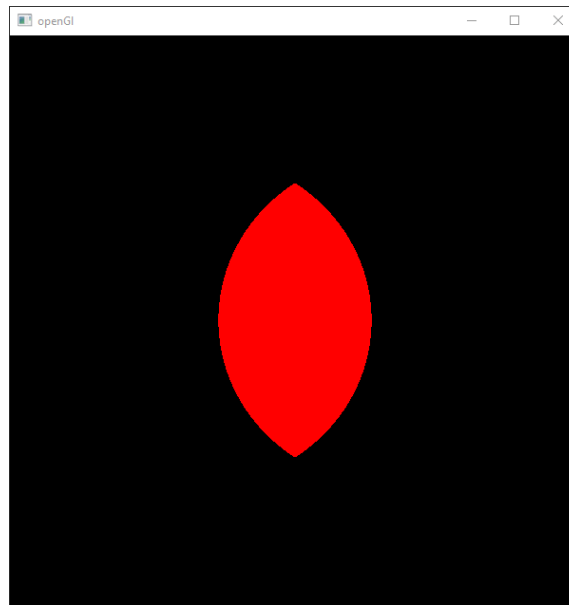
    p.activate();
    glDrawElements(GL_TRIANGLES, iB.number(), GL_UNSIGNED_INT, nullptr);
    glfwSwapBuffers(window);
    glfwPollEvents();
}
glfwTerminate();
return 0;
}
- vertex.glsl
#version 440 core
layout(location = 0) in vec2 vertexPosition;
void main()
{
    gl_Position = vec4(vertexPosition, 0.0,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

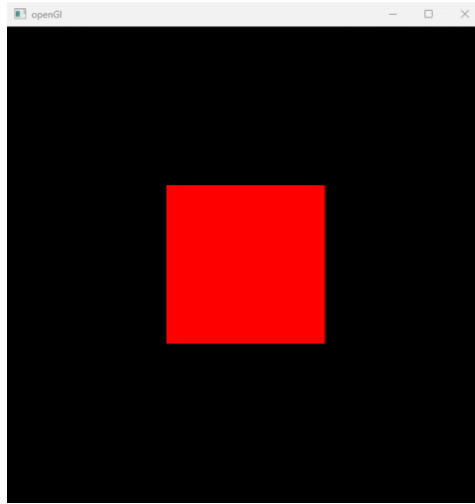
Rendering an SDF is good, but I need a way to rotate these SDFs, so using the same perspective rotation matrices as before and applying the inverse to every point p provided, we can implement this method in GLSL:

```
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;
}
```

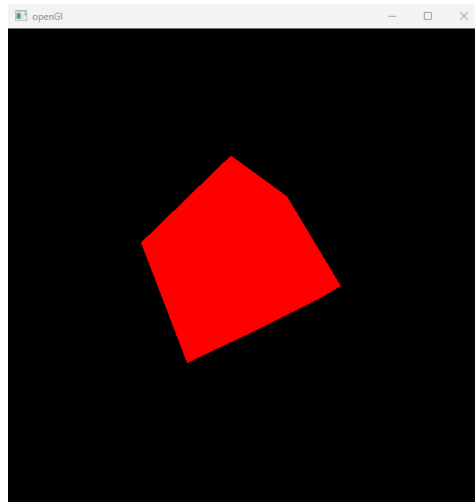
And, to see this in action, the SDF for a box in GLSL is:

```
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°) 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 α , β , and γ 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 4×4 to 3×3 .

Lighting and Reflections

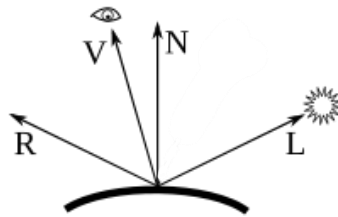
Research

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 equation¹¹:

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

Where k_a , k_d , k_s , and α 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^{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^{th} light, and $i_{m,d}$ is the colour of the m^{th} light. Finally, i_a is the colour of the object, or the “ambient” colour.

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 SDFs¹², and found this had applications for 3D SDFs as well, as screenshot of which is shown below:

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

¹² 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].

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;
}
```

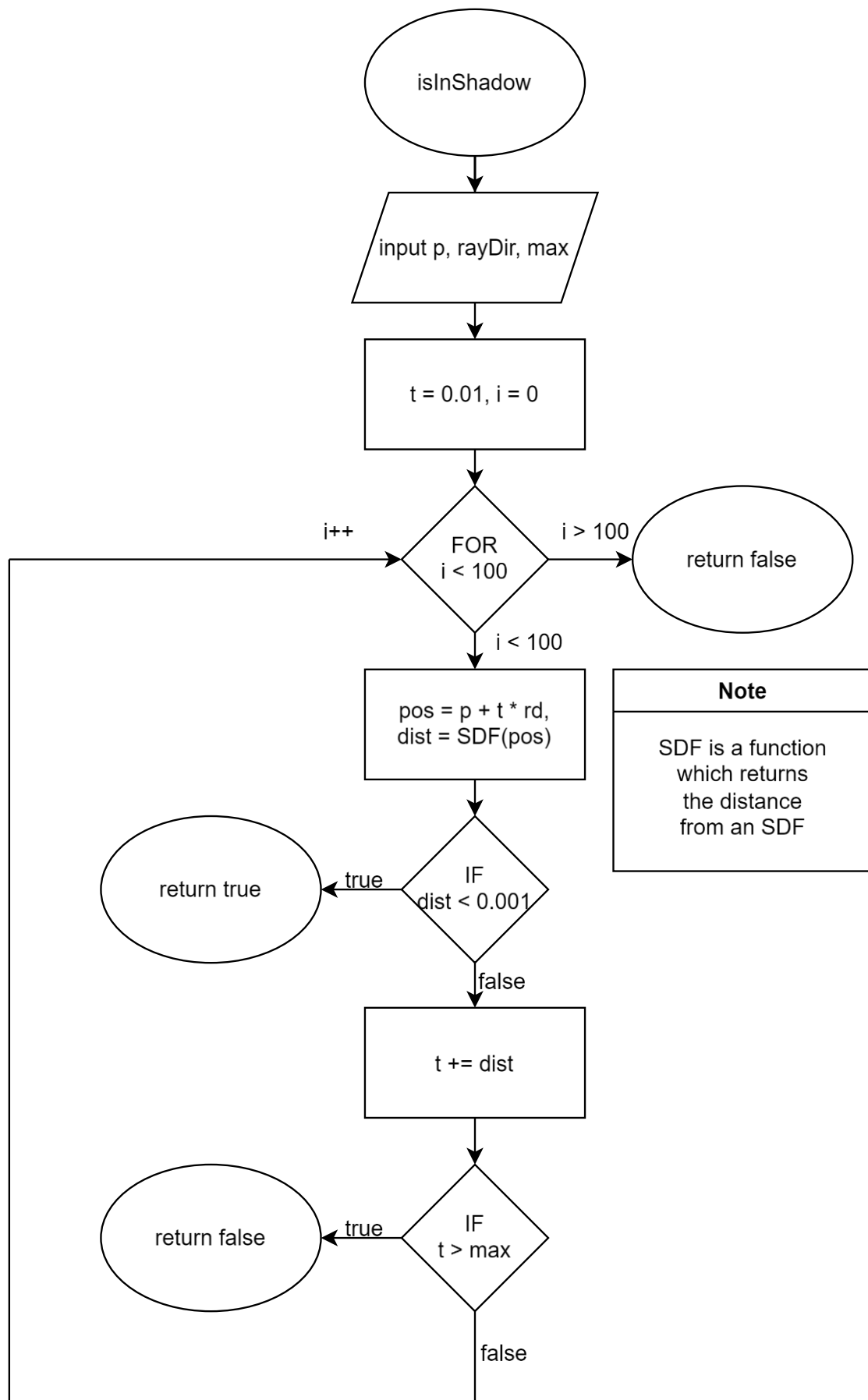
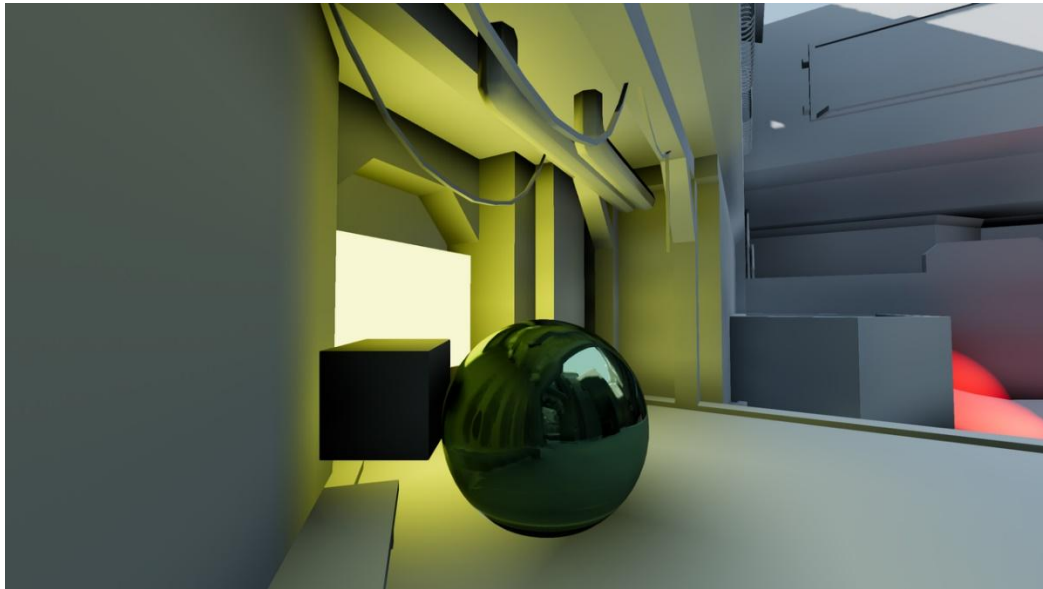


Fig. 6

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 time¹³. 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 research¹⁴, 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

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

¹⁴ Recursive Ray Tracing. (n.d.). Available at: 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 $f(p)$ 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
α	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:

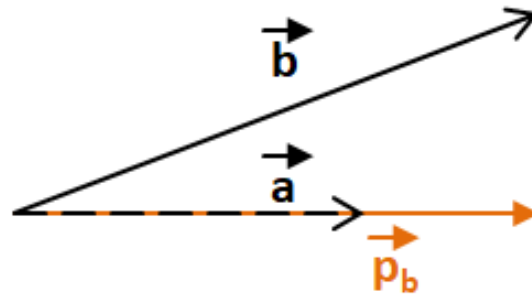


Fig. 7

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:

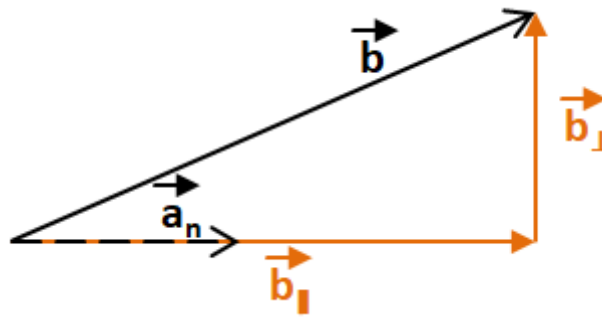


Fig. 8

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:

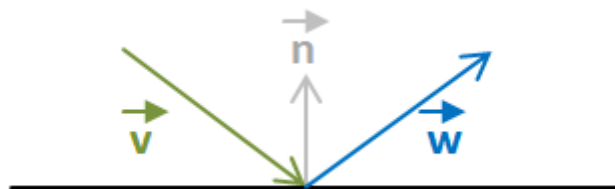


Fig. 9

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

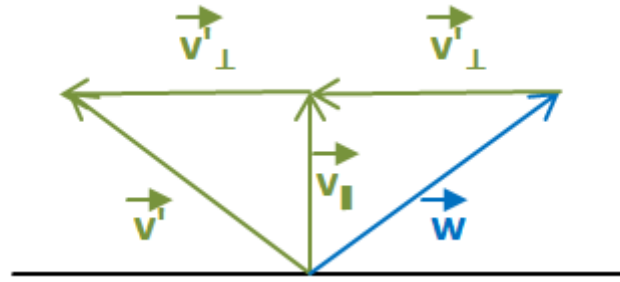


Fig. 10

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 Ia = vec3(1.0);
            vec3 surfaceC = getCol(Ia, 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 dynamics¹⁵, 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, ρ , 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 $y = \sqrt{r^2 - x^2} \sin \theta$, $dy = \sqrt{r^2 - x^2} \cos \theta d\theta$

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

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

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

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

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

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

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

¹⁵ Chapter 6 Rigid Body Dynamics. (n.d.). Available at:

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 (Δp) 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

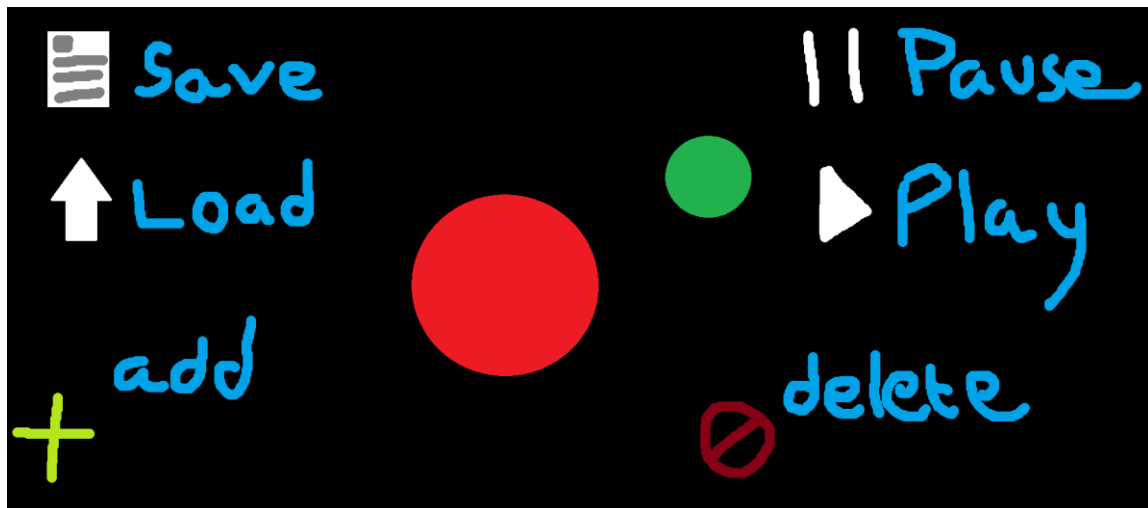
Introduction

Lots of 3D rendering software is pernickety 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

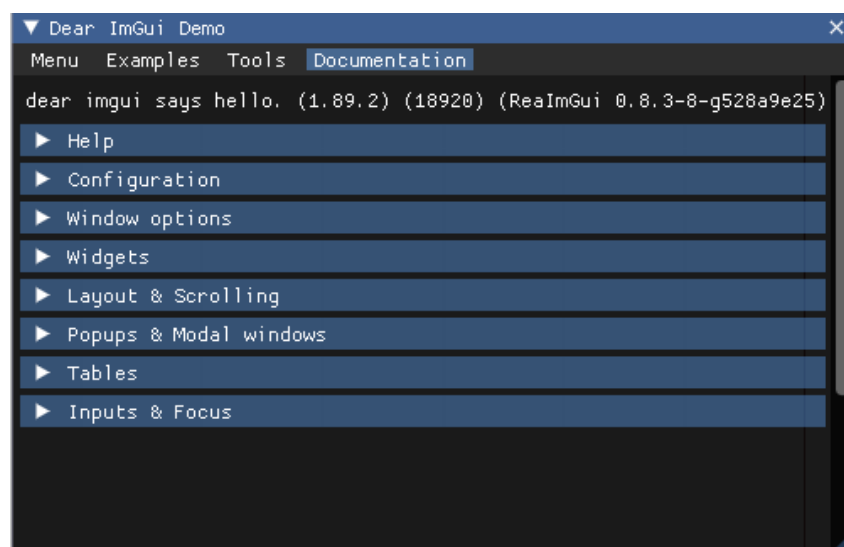
If a user were to right click on an existing object, a menu would come up displaying the objects attributes and would allow the user to edit them.

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`¹⁶ which allows the creation of a GUI inside of an OpenGL3 GLFW environment. The general `imgui` interface looks like this:



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

Objectives

My main objective for the project is to create an interactive and realistic physics simulation which can be altered in real time. One of the main features is that it will be able to be raymarched, improving the realism due to a lack of polygons in rendering shapes like spheres. It should be easy for a user to know how to interact with the software as well, which means designing it in such a way so that it is intuitive, and a bit educational too. Here is a list of things that should be completed to make this successful:

1. The GUI.
 - a. The user interface should be easy to understand and well divided into constituent sections. Some examples of these sections are:
 - i. File, which can be further sectioned into:
 1. Open, a button or menu to open a file if such a file exists.
 2. Save, a button or menu to save the current state of the screen.
 3. Exit, a button or menu to exit the window safely, and could have a key bind attached.
 - ii. Edit, which can be further sectioned into:
 1. Objects, a menu for adding an object into a set place in the scene.
 2. Materials, a menu linked to an aspect of object creation describing how an object can look.
 3. Lights, a menu for viewing information around the lights in the scene, and possibly the ability to add more lights.
 - iii. Settings, which can be further sectioned into:
 1. Controls, a menu or information tab telling the user how to interact with the scene.
 2. Debug, a menu or information tab telling the user about some data in the scene, and how well it is performing on their device. This may be used as a menu of interest for a user that wants to have an idea of how data is structured within the code.
 3. Constants, a menu allowing the user to edit certain global constants within the scene, such as the value for acceleration due to gravity.
 - b. The GUI should be easy to interact with and be able to be laid out in a way where each user is comfortable with the layout. This may be achieved by adding movable menus.
2. The scene's appearance.
 - a. The scene should be clear in how it appears and not have jagged ends to the main floor. This may be achieved by adding distance fog, and a checkered floor to give an idea of size.

- b. Any objects in the scene should be able to be distinguished from any other object when the camera can see more than one object at a time.
- c. The scene should have some initial conditions that allow a user to interact with it without having to create things other than objects. For example:
 - i. The scene should have one or more fixed lights which allows the scene to be viewed and objects to be seen.
 - ii. The scene should have some set materials for the user to pick from when summoning an object.
- 3. Scene interaction.
 - a. The scene must be rotatable so that objects can be viewed from any angle.
 - b. User summoned objects should interact realistically, including when summoned in the same place.
 - i. This means that if two spheres are summoned in the same place, they will not sit directly on top of each other, since even though in theory this is possible, it doesn't tend to happen in real life.
- 4. Objects.
 - a. All objects should be defined as a rigid body with some predetermined values which will allow the object to interact with others in the scene.
 - b. Each object collision between objects must be resolved by the laws of rigid body dynamics.
 - i. When a collision is detected, it calculates the impulse and applies it to each object.
- 5. Materials.
 - a. Each material (whether added or as an initial material) must allow the object to interact with the scene lighting and be identifiable.
 - b. Having a value to determine how shiny the object is.
 - i. Achieved by giving the object a colour, for example and RGB value.
- 6. Lighting.
 - a. The scene should be lit by light sources that follow the Phong lighting model, and should be able to be any colour, and in any position which will allow the model to work.

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

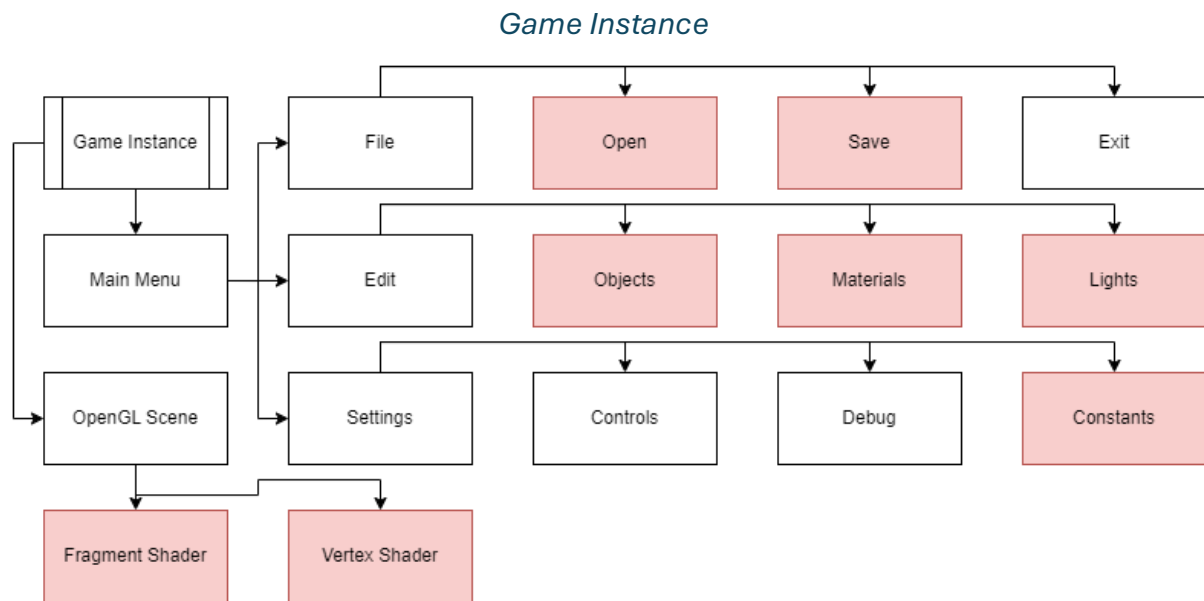
My program consists of many different menus and options, and this section will give an overview of what each menu does, and how the user interacts with the program.

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.



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.

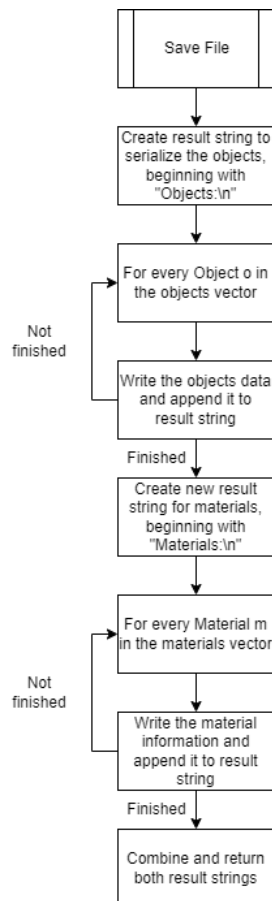
File

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

```

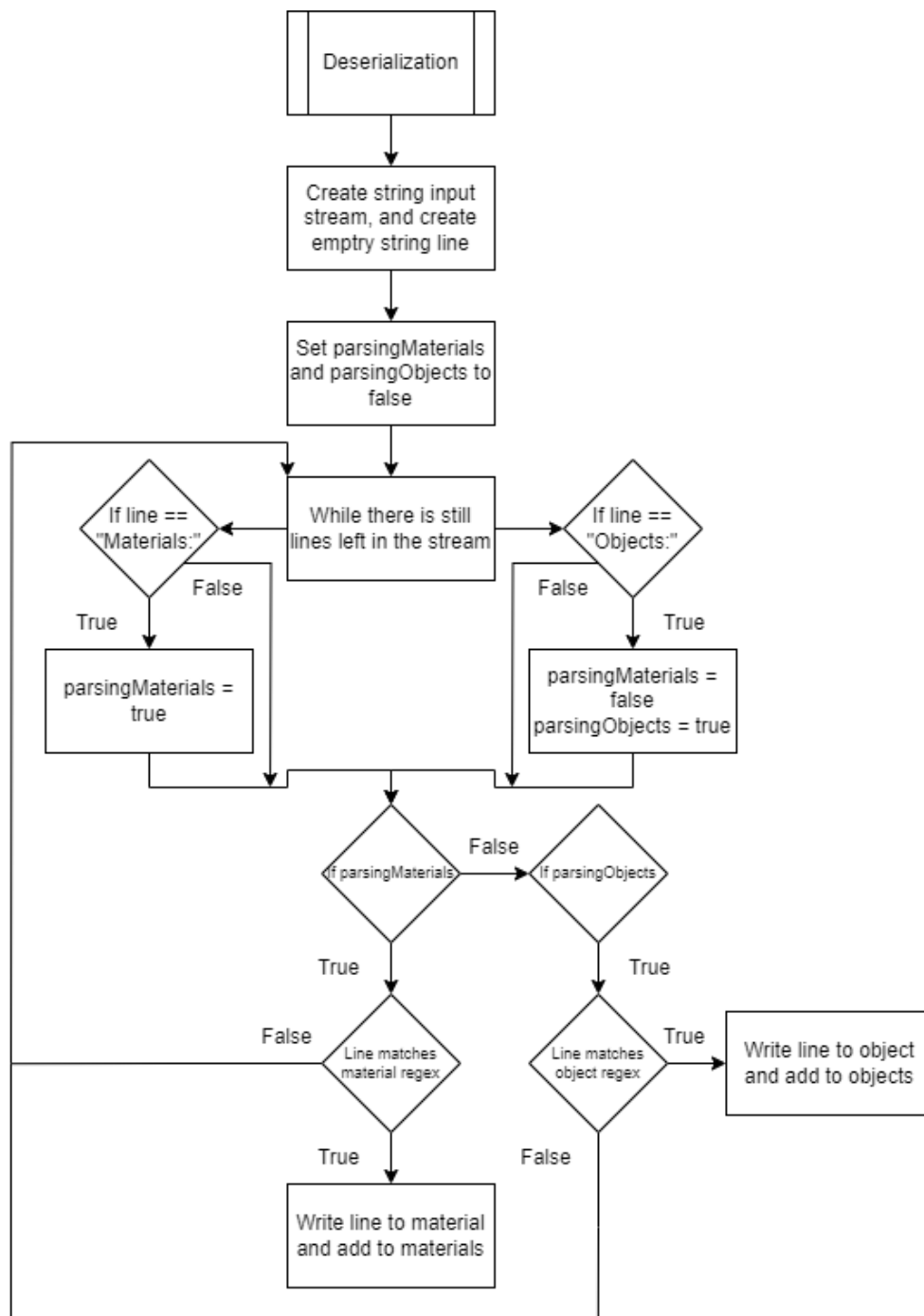
The deserialization function is quite long and is about matching regular expressions. The regular expression used for the materials is as follows:

- 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.

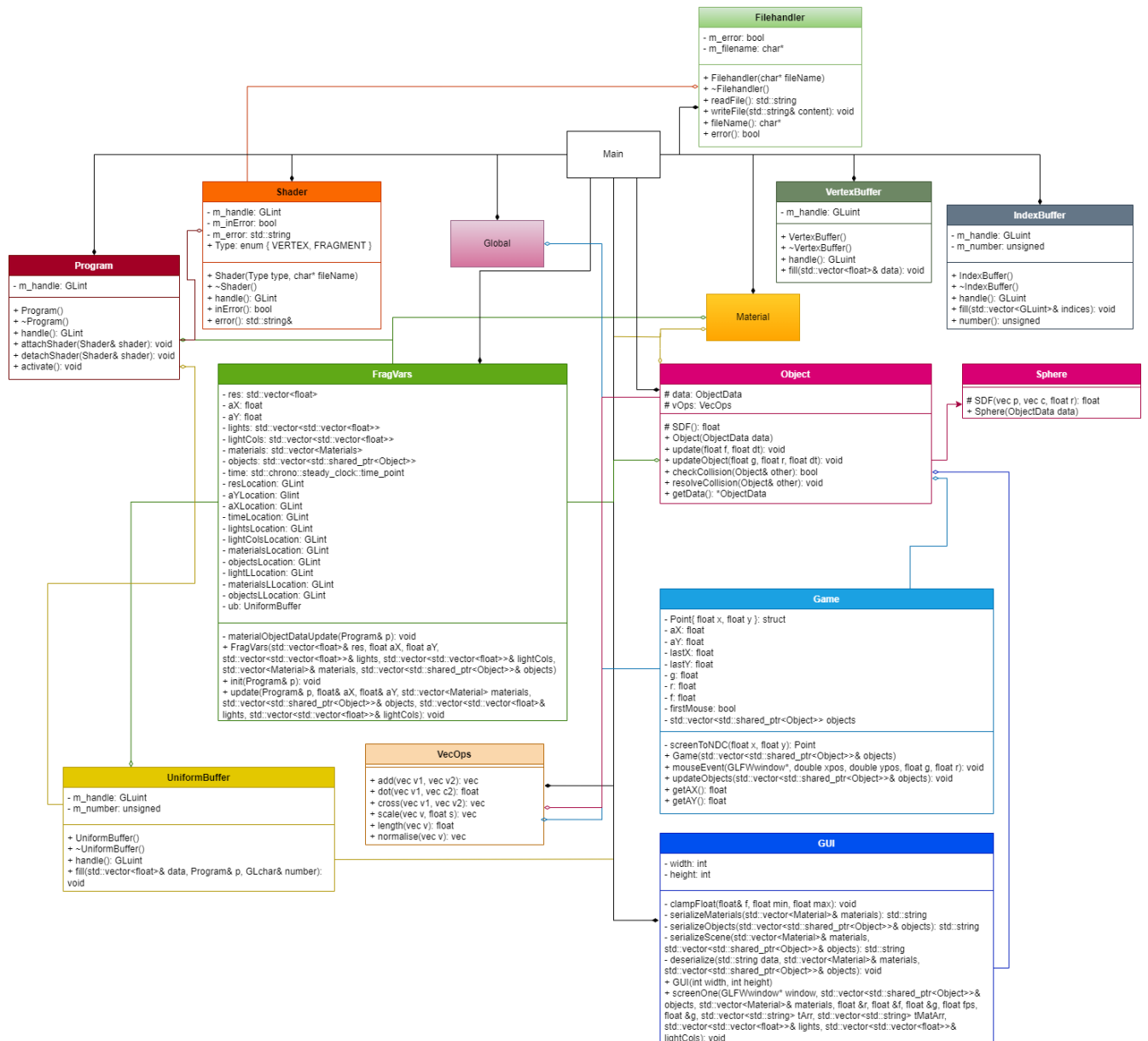
Algorithms and Modules

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.

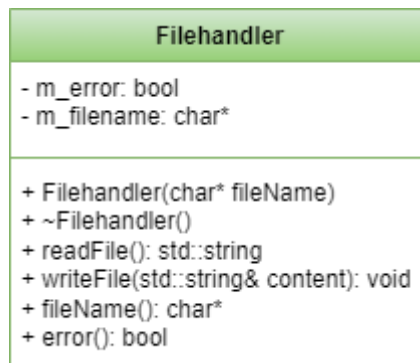
Classes

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 there has been an error.

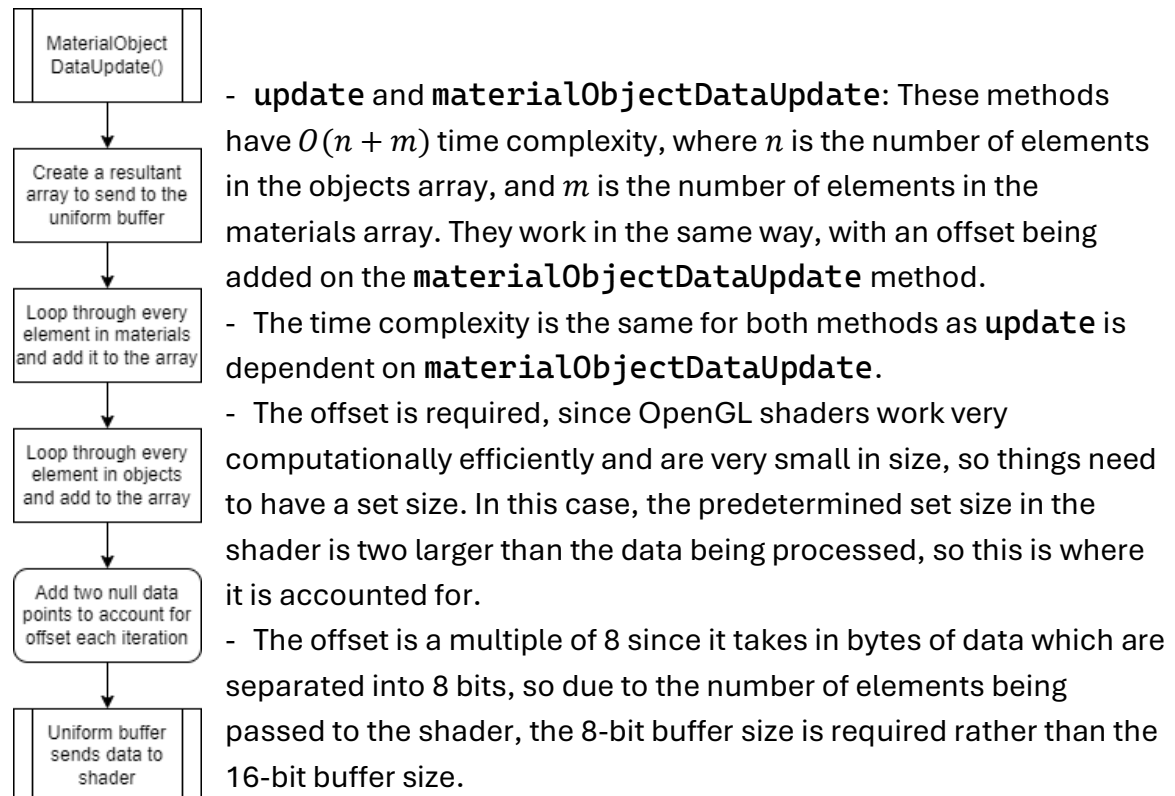
FragVars

FragVars
<pre> - res: std::vector<float> - aX: float - aY: float - lights: std::vector<std::vector<float>> - lightCols: std::vector<std::vector<float>> - materials: std::vector<Material> - objects: std::vector<std::shared_ptr<Object>> - 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& p): void + FragVars(std::vector<float>& res, float aX, float aY, std::vector<std::vector<float>>& lights, std::vector<std::vector<float>>& lightCols, std::vector<Material>& materials, std::vector<std::shared_ptr<Object>>& objects) + init(Program& p): void + update(Program& p, float& aX, float& aY, std::vector<Material> materials, std::vector<std::shared_ptr<Object>>& objects, std::vector<std::vector<float>>& lights, std::vector<std::vector<float>>& 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"> - Point{ float x, float y }: struct - aX: float - aY: float - lastX: float - lastY: float - g: float - r: float - f: float - firstMouse: bool - std::vector<std::shared_ptr<Object>> objects
<ul style="list-style-type: none"> - screenToNDC(float x, float y): Point + Game(std::vector<std::shared_ptr<Object>>& objects) + mouseEvent(GLFWwindow*, double xpos, double ypos, float g, float r): void + updateObjects(std::vector<std::shared_ptr<Object>>& objects): void + getAX(): float + getAY(): float

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
ENDIF
IF (firstMouse):
    lastX = xPos, lastY = yPos
    firstMouse = false
ENDIF
xoffset = xPos - lastX // Get the x and y offsets
yoffset = yPos - lastY
lastX = xPos, lastY = yPos
aX += xoffset, aY += yoffset // Apply them
```

ENDFUNCTION

- This function also limits `aX` to be between $\frac{\pi}{4}$ and $-\frac{\pi}{4}$ radians. The `firstMouse` Boolean variable is used as otherwise there would be jumps in rotation if the user moved the mouse while it wasn't being clicked.
- `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

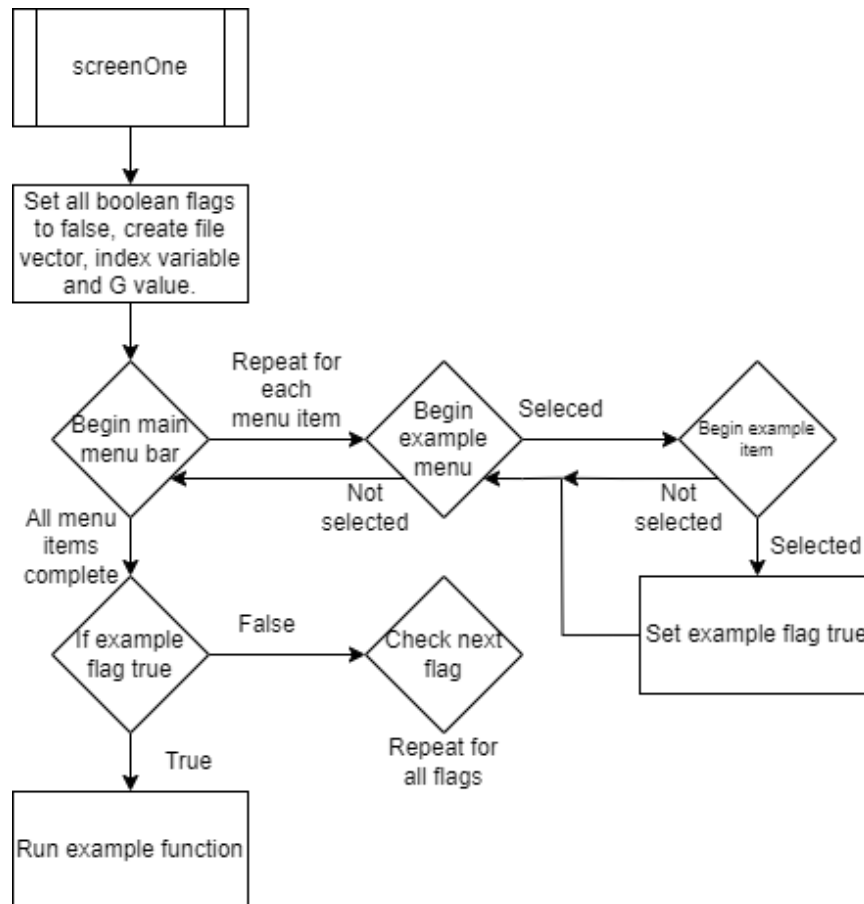
GUI
<ul style="list-style-type: none"> - width: int - height: int
<ul style="list-style-type: none"> - clampFloat(float& f, float min, float max): void - serializeMaterials(std::vector<Material>& materials): std::string - serializeObjects(std::vector<std::shared_ptr<Object>>& objects): std::string - serializeScene(std::vector<Material>& materials, std::vector<std::shared_ptr<Object>>& objects): std::string - deserialize(std::string data, std::vector<Material>& materials, std::vector<std::shared_ptr<Object>>& objects): void + GUI(int width, int height) + 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. Without serialization, the data would be very hard to tell where one data point ends and the next one begins.
- **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 - m_number: unsigned
+ IndexBuffer() + ~IndexBuffer() + handle(): GLuint + fill(std::vector<GLuint>& indices): void + 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
<pre># data: ObjectData # vOps: VecOps # SDF(): float + Object(ObjectData data) + update(float f, float dt): void + updateObject(float g, float r, float dt): void + checkCollision(Object& other): bool + resolveCollision(Object& other): void + getData(): *ObjectData</pre>

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:
 - i. Its type.
 - ii. Its material.
 - iii. Its position.
 - iv. Its defining length.
 - v. Its mass.
 - vi. Its velocity.
 - vii. 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.

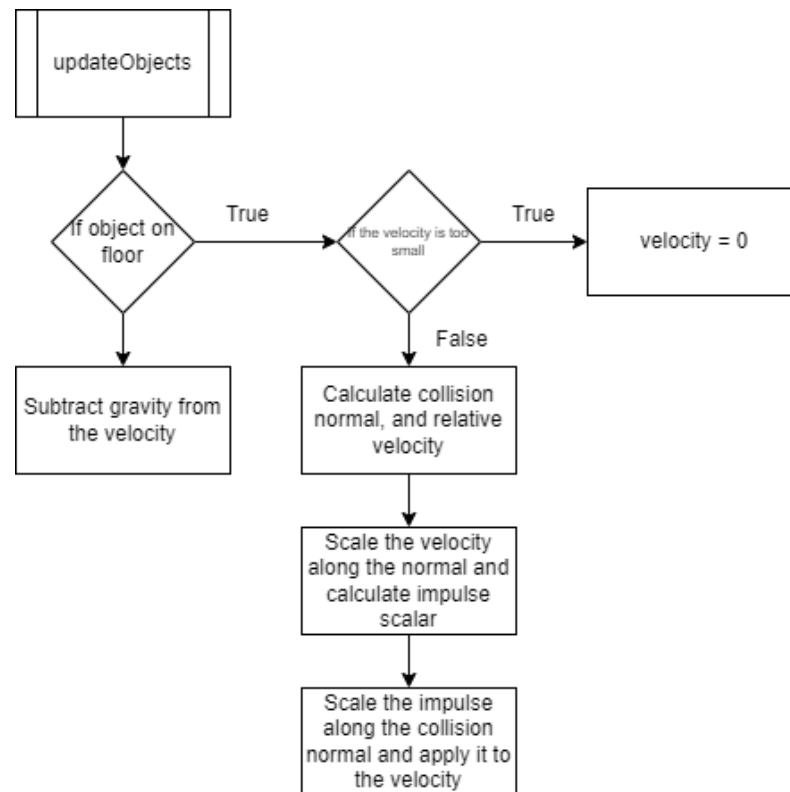
Subclass Sphere

Sphere
<pre># SDF(vec p, vec c, float r): float + Sphere(ObjectData data)</pre>

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 returns other values through polymorphism in subclasses.
- **Object(data)**. This is the constructor for the class and initialises the protected variable data with values specified in the argument.
- **update(f, dt)**. 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. All updates are multiplied by delta time to account for different frame rates.
- **updateObject(g, r, dt)**. 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):



- All updates are multiplied by delta time to account for different frame rates.
- **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:

```

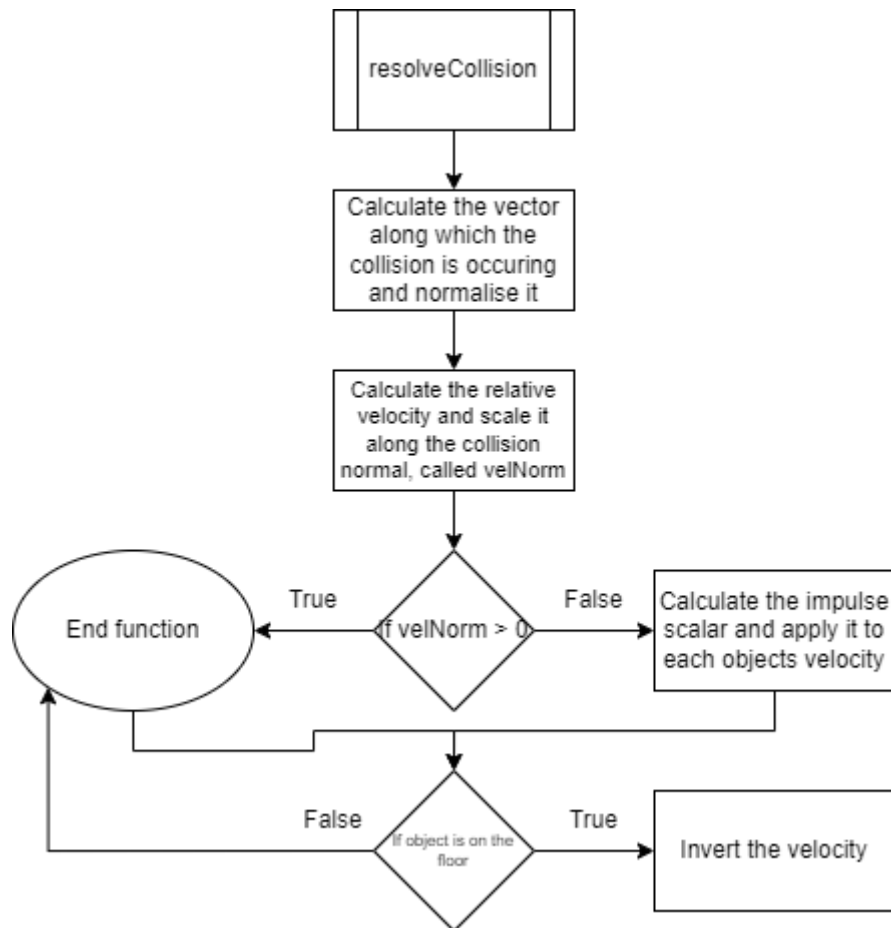
if (other == this object):
    return FALSE END FUNCTION

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 a polymorphic 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() + ~Program() + handle(): GLint + attachShader(Shader& shader): void + detachShader(Shader& shader): void + 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</code> : GLint - <code>m_inError</code> : bool - <code>m_error</code> : std::string + <code>Type</code> : enum { VERTEX, FRAGMENT }
+ <code>Shader(Type type, char* fileName)</code> + <code>~Shader()</code> + <code>handle()</code> : GLint + <code>inError()</code> : bool + <code>error()</code> : std::string&

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 deconstructor 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
- m_handle: GLuint - m_number: unsigned
+ UniformBuffer() + ~UniformBuffer() + handle(): GLuint + fill(std::vector<float>& data, Program& p, GLchar& number): void

The UniformBuffer class is used for binding a large chunk of data to allocated 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 deconstructor 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> + <code>dot(vec v1, vec c2): float</code> + <code>cross(vec v1, vec v2): vec</code> + <code>scale(vec v, float s): vec</code> + <code>length(vec v): float</code> + <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:

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

Which is:

$$\begin{pmatrix} v_{1z}v_{2z} - v_{1z}v_{2y} \\ v_{1y}v_{2x} - v_{1x}v_{2z} \\ v_{1x}v_{2y} - v_{1y}v_{2x} \end{pmatrix}$$

- **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() + ~VertexBuffer() + handle(): GLuint + 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 (Δ) 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(Ia, m, Ii, 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)

 // Calculate diffuse

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

 // Calculate specular

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

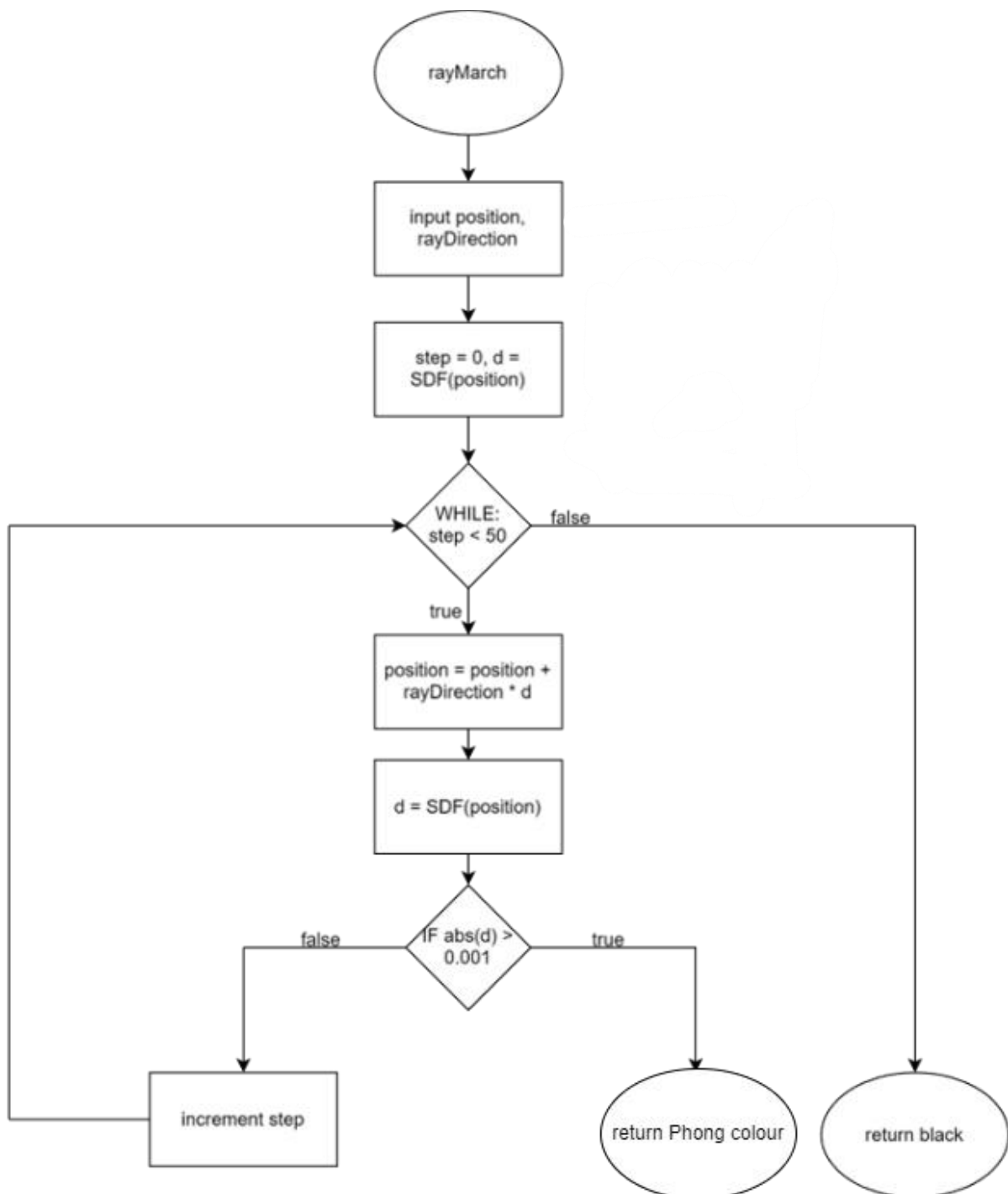
 ENDIF

 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:

- materials**. This is the vector of materials which gets sent to the fragment shader.
- 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() // Get monitor info

```



```

IF (mode == NULL): // Check it has worked
    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 of the two triangles
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
```

Testing

Iterative Testing

Here is a list of issues I faced while making this coursework:

- The first issue I encountered was learning about how to use pointers and referencing to pass data between methods and classes and realising that each class in C++ requires two files: a header and the .cpp class – the source class.
- The second issue I faced was trying to render many objects without the use of OpenGL shaders, meaning the rendering capabilities were extremely slow. This was all the maths related to the “Reduction to Triangles” section of the analysis.

- I then faced an issue of creating objects of different colours, and how I transmitted data to a shader from the code. On the back of this I learnt about uniform buffers.
- The next issue I faced was passing data into the uniform buffer. I had to realise that the format that the shader wanted the code in was very specific and had to be laid out in a set way for it to be sent.
- My next issue was to do with working out how to handle recursion in GLSL, as it does not support recursion. I worked around this by simulating recursion in a for-loop.
- After this, I had a problem where editing object data to create new objects was messing with the materials and object data being passed to the shaders, which was again to do with the uniform buffer.
- The next problem was with rigid body dynamics and figuring out how to correctly resolve collisions in 3D space, which was resolved by implementing a vector class.
- I had a large problem with performance which was bypassed by adding a method in the shader for intersecting a bounding volume and adding a dynamic step size.
- After this, there was a problem with the user interface not keeping changed variables, which was fixed by adding static variables inside of methods.
- The last issue was with delta time, where I was only applying delta time to velocity, instead of velocity and acceleration, meaning that the simulation didn't quite look correct and altering global constants didn't have intended effects.

System and Unit Testing

- Link to testing video: <https://www.youtube.com/watch?v=ktiEpQe9pSY>.

Test Number	Objective number(s)	Timestamp	Expected result	Actual result
1	1.a.i.1 and 1.a.i.2	00:00 – 04:20	<p>A file will be able to be saved. It can then be seen that there is a file, and will then be able to be opened from the dropdown menu. Another file will be saved to show both appear in the dropdown menu. The file format will be accepted by the following regexes:</p> <ul style="list-style-type: none"> - r:([0-9\\.]+) g:([0-9\\.]+) b:([0-9\\.]+) Ka:([0-9\\.]+) Kd:([0-9\\.]+) Ks:([0-9\\.]+) 	Pass – all files were saved and loaded correctly including when the program was quit and reloaded, and the regexes matches.

			Kr:([0-9\\.]+) c:([0-9\\.]+) - type:([0-9]+) material:([0-9]+) pos:(\\-?[0-9\\.]+),(\\-?[0-9\\.]+),(\\-?[0-9\\.]+) radius:([0-9\\.]+) mass:([0-9\\.]+)	
2	1.a.i.3	04:20 – 04:52	Upon pressing “Esc”, or Exit in the file menu, the program will quit.	Pass – the program quit when the “Esc” button was pressed, and when the Exit menu button was pressed.
3	1.a.ii	04:52 – 08:53	The edit menu will allow objects to be added to the scene, materials to be viewed and added, and lights to be viewed and added.	Pass – all submenus in the edit menu worked.
4	1.a.iii	08:53 – 11:39	The settings menu will give information about the program’s controls, object data and universal constants.	Pass – all submenus in the settings menu worked and constants were correctly altered.
5	1.b	09:23 – 09:36	Any menus that appear will be able to move and be resized.	Pass – menus can be moved and resized.
6	2.a	11:39 – 12:40	The floor will appear checkered, not be sheered from any angle, and fade out gently via use of distance fog.	Pass – with the scene being rotated the floor reacts correctly and distance fog is applied.
7	2.b	12:40 – 13:55	When an object is in front of another object, it is clear to	Pass – objects were

			see that they are different objects in different locations with no overlaps.	proved to be distinct.
8	2.c	13:55 – 14:40	In the menus, there will be some initial lighting and materials allowing objects to be added straight away.	Pass – there is initial conditions so that objects can be added straight away.
9	3.a	14:40 – 15:12	The scene will be able to be rotated by the mouse, and only when the left mouse button is pressed.	Pass – the scene is only rotated when the left mouse button is pressed.
10	3.b	15:12 – 15:44	When one object is summoned on top of another, a small random offset is applied to its position to account for realistic interactions.	Pass – Objects do not spawn directly on top of each other.
11	4.a and 4.b	15:44 – 39:40	It can be proved that an object interacts as a rigid body, following rigid body dynamics by the following. <ol style="list-style-type: none"> 1. An object on its own with no external forces should only experience gravity and should fall at a constant acceleration rate. 2. When two objects collide, their collisions will be resolved by the laws of rigid body dynamics. 	Pass – all calculated values were within a small absolute error that can be accounted for by rounding and floating-point arithmetic.
12	5.a	39:40 – 41:08	When two objects are added, they must be distinguishable in colour based on the material, and another material added with random values to show that	Pass – added materials work when used on objects.

			an object can be any material.	
13	5.b	41:08 – 43:23	An object that is shiny must have real-time reflections visualised, with the step count shown to have reflections in reflections (recursion).	Pass – there is reflections and reflections inside of reflections via use of recursion.
14	6.a	43:23 – 59:46	Any given object must have a clear ambient, diffuse and specular component to show the Phong lighting model works. The ambient and diffuse should give the colour and spread of colour on the object, and the specular component should give bright white spots where lots of light is reflecting towards the camera.	Pass – the calculations have proved that the lighting works as expected. However, it was discovered after the video that a small random offset was still applied to the spawned position, slightly decreasing the value, meaning the calculation would have been more precise. See evaluation for further explanation.

Evaluation

Overall Evaluation

Overall, I think my project has been a success. I have successfully managed to make an intuitive physics engine with real time ray-marched lighting, and objects of different materials. I have met many of my objectives, and all tests passed, despite the issue with the last test.

Evaluation Specifics

Objective Evaluation

In this section, I will consider each objective and decide whether I have met it:

- 1.a.i – This objective was met as all buttons in the file menu were able to be opened, and files saved and loaded successfully with the correctly matching regexes. This can be seen through tests 1 and 2.
- 1.a.ii – This objective was met as the objects, materials and lights menus all worked correctly, and objects can add more objects along with removing them, materials can add more materials, and lights can add more lights. This can be seen in test 3.
- 1.a.iii – This objective was initially met as there was menus inside the settings menu which informed the user of the controls, allowed them to successfully edit universal constants and to see the data relating to what is happening in the scene. This can be seen in test 4.
- 1.b – This objective was met since throughout the testing video, I had no problem interacting with the GUI, and menus are moveable and resizable to fit where each user would want the menu. This can be seen throughout the testing video, but specifically in test 5.
- 2.a – This objective was met since the floor did not have any jagged edges, there was a distance fog, and the floor is checkered to give an idea of size. This can be seen throughout the testing video, but specifically in test 6.
- 2.b – This objective was met, as every object was proved to be distinct from any other given object in the scene. This can be seen in test 7.
- 2.c – This objective was met because when the simulation was first loaded, there were initial conditions hard-programmed in that allowed the user to only have to press the create button to begin using the simulation. This can be seen throughout the testing video, but specifically in test 8.
- 3.a – This objective was met as the scene is fully rotatable without any errors occurring during rotation. This can be seen throughout the testing video, but specifically in test 9.

- 3.b – This objective was met as when two objects are spawned in the same location, a random offset was correctly applied meaning that there was never an instance of two objects sitting on top of one another. This can be seen in test 10.
- 4.a and 4.b – This objective was met as it was proved through hand tracing the algorithm that caused the collisions, then showing that the output of this algorithm is as predicted. This was proved for floor-object collisions and object-object collisions. This can be seen mainly through tests 11-1 and 11-2.
- 5.a – This objective was met as when a material was added (or an initial one used), the object to which the material was applied correctly in terms of interacting with the scene and the scene lighting. This can be seen in many points in the testing video, but mainly in test 12.
- 5.b – This objective was met as when an object used a shiny material, or a reflective material, it correctly had reflections on its surface to show the rest of the scene in the correctly distorted manner based on the curvature of the object. This can be seen in test 13.
- 6.a – This objective was met but was not tested well. It was met because the lighting was proved to work correctly through hand tracing of the algorithm, and throughout the testing video it is evident that the Phong model is being applied. However, when proving this for testing, I forgot to disable the initial random offset required for objective 3.b. This meant that my initial assumption of the object being at (0,0,0) was incorrect, and it was in fact at a small offset from this point, which accounted for the larger error in the values than normal. However, if the object was at (0,0,0), the value obtained is very close to the true value, so I still consider this objective to be a success. This can be seen in test 14, and throughout the testing video.

Individual Further Improvements

Whilst I am extremely pleased with the outcome of my project, there are many features I would have liked to add which I unfortunately did not get to, due to time and complexity in transferring data to OpenGL shaders. Here are a few:

- Adding angular velocity to improve realism in collisions and friction. Based on research, this would involve calculations using inertia and conservation of linear and angular momentum.
- Adding objects of different shapes. This would make the software much more interesting to use and give it another major feature. This was initially a feature, but upon adding physics to the system, a lot of the code surrounding these additional objects (such as cubes and tori) broke down. This is why the Sphere class inherits from a parent Object class, because there was going to be other children of the Object class.

- Change of rendering style. This means that I would make further use of the vertex shader to not have the whole scene be rendered by the fragment shader, which would drastically improve performance. This would mean researching model, projection and view matrices, which I did not get to do due to time constraints.

User Feedback

Ben Warner feedback:

- Intuitive and fun to play with.
- Generally stable in normal use-cases (i.e. not stress testing).
- Objects at the edge of the scene don't have fog applied, leading to a break in realism.
- Objects don't jump when the coefficient of restitution is set to 0 – only a sideways velocity is applied.
- Large spheres overlap light sources in some positions, leading to shadows and weird light changes.
- Material colour previews don't account for ambient, diffuse and specular coefficients, so it's difficult to predict what colour an object will appear to be.

Cameron Dennis feedback:

- Feels like I can move camera with keys but cannot.
- Indicators for direction needed.
- UI overall intuitive.
- Floor grid pattern provides nice contrast for visibility.
- Ability to remove materials and lights.
- Label materials.
- It was a stable simulation.

Jamie Whitehair feedback:

- The way of adding objects is simple and easy to do.
- The addition of a zoom feature would add to the completeness.
- You can have many objects present without heavy lag to the program.
- The debug menu is very intuitive and helpful to the user for checking which objects are present and their specific properties.
- The constants menu is a lot of fun to adjust.
- Sliders could be used instead for constant values to allow easier slight adjustments.
- With restitution set to 1, bouncing mechanism sometimes breaks.

Sam Drake feedback:

- Evaluating the Physics aspects of the simulation, the adjustable values for gravitational field strength, restitution, and friction responded as expected, including the extremes of both 0 values, and maximum values.
- The upper limit for gravitational field strength was at a high enough value to be of use and interest for reasonable real life physics interactions.
- The simulation was clearly visible and lighted from opening, with the clear ability to add additional light sources, which interacted with the spherical objects as I would expect.
- The boundaries of the scene were clean, and interacted with the spawned objects as I would expect – When setting friction to 0, objects should continue with their motion until they are out of visible range. This was the case.
- The menus to adjust various physics constants were clear and easily interactable by following the controls in the settings menu – I would suggest having these visible upon startup.
- The possibility of including sliders, or small incremental clickable button changes to the Physics variables might be nice – to add to the live interaction of the objects while they are still moving – Although they would change their behaviour as values were type in/changes – Sliders would make gradual changes easier and more intuitive.
- Changing object sizes, and causing object collisions responded well alongside physics expectations, based on the value of restitution entered. Impulses must have been calculated to provide this visual response.
- Changing the spawning positions of new objects, before causing them to interact with one another, provided different interactions depending on how much Kinetic energy the object would have gained from falling from a higher height – once again providing realistic object interaction.

Further Improvements Based on User Feedback

Addressing feedback:

- Ben Warner:
In this evaluation, Ben acts as the end user which has seen this project through its development so has a good knowledge of where bugs may have arisen, as he has been a regular testing guinea pig.
 1. It is “intuitive and fun to play with” – This is a good response as the main goal of this project was for it to be intuitive and easy to use, so this is perfect feedback.
 2. It is “generally stable in normal use-cases” – This is good feedback, as it means the project is running efficiently in the way it is intended to be used.
 3. “Objects at the edge of the scene don’t have fog applied” – Upon further testing, I have found this error does occur. I had not noticed it as I had been

working with balls of radii in the range of around 0.1 to 0.5, and it is much more noticeable with large objects, that were being used in Ben's test. I would need to look at how I apply the fog to repeat adding the fog to objects.

4. "Objects don't jump when coefficient of restitution is set to 0" – I was not aware of this bug, but I believe it occurs in `object.cpp` in the `updateObject` method. Specifically it is to do with the recent addition of the `lastBounceSpeed` variable in a given object's `objectData`.
 5. "Large spheres overlap light sources in some positions" – As mentioned, I tended to test with small objects, so the hard limit I set on light positions was supposed to prohibit this error. This can be fixed by changing the value in `clampFloat` under the `newLightFlag` to a larger value based on the maximum value of the radius.
 6. "Material colour previews don't account for ambient, diffuse and specular coefficients, so it's difficult to predict what colour an object will appear to be" – Implementing this would involve running a new instance of the lighting algorithm inside each colour button, which is not something I have worked out how to do yet within `ImGui`, but it is something I would like implemented.
- Cameron Dennis:

In this evaluation, Cameron is the user that has been using the software for a while, since he downloaded it from GitHub and had been playing around with it prior to the formal end-user evaluation. He may provide insights into bugs that I hadn't identified.

1. It "feels like I can move [the] camera with keys but cannot" – Here, Cameron is referring to being able to use the W, A, S and D keys to move the perspective camera around the scene. This is something I would like to implement, but I didn't get around to it due to time constraints.
2. "Indicators for direction needed" – Cameron is referring to when objects are spawned in, it is not clear where exactly they are going to be spawned. This is something I considered implementing, I was going to add an opaque object at the position it should be spawned in, but I did not get around to this due to time constraints.
3. The "UI [is] overall intuitive" – This is very good feedback, as with Ben's feedback, this proves that the UI is very easy to use.
4. The "floor grid pattern provides nice contrast for visibility" – This is further good feedback, especially as it contributes towards my objectives relating to the scene being very clear and distinct.
5. There needs to be an "ability to remove materials and lights" – The lack of this feature was an oversight, as it is very easy to implement the ability to add and remove materials and lights, and it would add to the overall experience of the software.

6. The need to “label materials” – Cameron is referring to the materials slider in the Objects menu; he suggested that I could add another colour button to show which material is currently being selected. This was another oversight, as I believe it could make the user experience more intuitive.
 7. “It was a stable simulation” – This is good feedback, since for the simulation to appear realistic, as per my main objective, it needs to be able to run steadily.
- Jaimie Whitehair:

Jaimie is a student who has not had much experience at all with my coursework, so I thought he would also be good to test how intuitive it is for new users.

 1. “The way of adding objects is simple and easy to do” – This is good feedback, showing that the initial conditions of the software are easy to use, as per my objectives.
 2. “The addition of a zoom feature would add to the completeness” – I had not considered this until Jaimie suggested this, but I fully agree, as it would allow you to see the collisions from a closer angle. I believe this would improve the usability of the software.
 3. “You can have many objects present without heavy lag to the program” – This is very positive feedback, as it shows that the scene can remain realistic with a relatively large number of objects. I don’t think this would have been possible without the intersectsBoundingVolume method in the fragment shader.
 4. “The debug menu is very intuitive and helpful to the user for checking which objects are present and their specific properties” – This is very helpful feedback, as my whole intention adding the debug menu was so that any user can understand what is happening in the scene without knowledge of the code, and it can educate them slightly as to what different values do.
 5. “The constants menu is a lot of fun to adjust” – This is good feedback, as I believe if the program is fun to interact with, that improves user experience and allows them to gain intuition into what each value does by altering it.
 6. “Sliders could be used instead for constant values to allow easier slight adjustments” – Throughout watching the end-users test my coursework, I agree with this. All users seemed to find it easier to alter values with a slider rather than inputting numbers, so that is something I would change if I were to remake this.
 7. “With restitution set to 1, bouncing mechanism sometimes breaks” – This is a similar issue as to the one found by Ben, and I think this is caused by the same bug.
 - Sam Drake

Sam Drake is an A-level physics teacher, who I tasked as the main end user. I asked him to give me an in-depth evaluation as to his experience, and specifically to test if the physics of the simulation seems realistic.

1. “The adjustable values for gravitational field strength, restitution, and friction responded as expected” – This is very positive feedback, since it shows that the physics in the simulation regarding floor-object collisions is working, and that my proof in the testing video applies to real-world physics too, making the overall simulation more realistic.
2. “The upper limit for gravitational field strength was at a high enough value to be of use and interest for reasonable real life physics interactions” – This is particularly helpful feedback, since this gives large insight into the user’s experience. He was able to use the altered values to test what changing certain constants do to the simulation, like halving gravity then seeing what changed, or setting friction to 0 and seeing what changes. This shows that any user can have an intuition about what each variable value should do and be able to easily interact to demonstrate realistic physics principles.
3. “The simulation was clearly visible and lighted from opening” – This reinforces my objectives surrounding initial conditions; all Sam had to do was create an object and he could see how the simulation worked.
4. It had a “clear ability to add additional light sources, which interacted with the spherical objects as [he] would expect” – This also reinforces more of my objectives, since it was easy for him to add light sources and see that the reflections were working correctly.
5. “The boundaries of the scene were clean, and interacted with the spawned objects as I would expect” – This proves that the distance fog was working correctly, and Sam goes on to explain how he varied values of friction until it reached 0 to see that at 0 objects go out of vision. This shows that it is easy for a user with minimal experience in operating the software can come to their own conclusions about whether certain aspects of the program work.
6. “I would suggest having these [the control menu] visible upon startup” – This was a helpful suggestion from Sam that would greatly improve user intuition. While there is a control menu, the user has to go to Settings -> Controls to be able to access it. I believe that it would be easier to interact with if this was open by default and then can be optionally closed. This would be an easy fix, as it would just require inverting a Boolean flag variable.
7. “The possibility of including sliders, or small incremental clickable button changes to the Physics variables might be nice” – This is a very similar point as to number 6 in Jaimie’s section, but I agree also that buttons that increment and decrement values could also make it more intuitive.
8. “Changing object sizes, and causing object collisions responded well alongside physics expectations, based on the value of restitution entered.

Impulses must have been calculated to provide this visual response” – This shows that the collisions in the scene are adhering to the laws of rigid body dynamics, as per my objectives. This can be seen since to Sam, it was clear that the collisions were impulse-based.

9. “Changing the spawning positions of new objects, before causing them to interact with one another, provided different interactions depending on how much Kinetic energy the object would have gained from falling from a higher height” – This is a surprising piece of feedback, but I think it is very positive. I didn’t directly implement energy changes, but I think it is very good that the simulation almost seems to naturally simulate the changes without direct program. I suspect this is a side-effect of using rigid body dynamics, and where the equations from this come from.

[END OF DOCUMENTATION]