Creative Computing

A Study on the Development of 3D Graphical Software

by

Jan Pantic

N00222591

Supervisor: John Dempsey

Second Reader: Joachim Pietsch

Year 4 2025/2026

DL836-BSc (Hons) Creative Computing

30/04/2026

Acknowledgements

I would like to thank the lecturers at IADT for the excellent education and learning experience I had through this 4-year program.

I would also like to extend my gratitude to my colleagues at IADT, who were always there to help with any kind of project or issue, no matter how complicated it seemed at first, and who were always open to complex discussions on any computer science topic. This thesis wouldn't have been possible without their constant support.

Additionally, I would like to mention my project supervisor John Dempsey, who believed in the project from day 1, and who's experienced guidance made this project and thesis achievable and manageable.

Lastly, I must mention my family, as they were there to support me in finishing this project, and were always willing to use their free time to help me test my projects and proof-read my thesis.

Table of Contents

Introduction and Project Context

The aim of this project is to make a piece of 3D graphical software from scratch using the C++ language and various libraries. This document will detail the extensive technical research undertaken to achieve this goal. It will also cover how the 3D engine attached to this thesis was developed.

Scope:

Originally, the scope of the project was set way too high; the intent was to create a simple 3D game engine, which had capacity for rendering, custom scripting, audio and so on. However, the scope of the project kept creeping up, as simply implementing 3D rendering was proving to be a massive undertaking. The scope has since been modified and now the objective is to create a 3D engine, such as a model viewer. The below objectives are more in-line with the current scope.

Objectives:

1. *Build Dev Environment*

   The first objective is to use CMake to build a Visual Studio project that can be used as a dev environment. This project will use the Visual Studio C++ compiler.

2. *Window*

   The second objective is to create a window which would display the result of the rendering on the GPU. This window would be opened using the SDL library.

3. *2D Rendering*

   The next step is rendering a simple shape in 2D. To do this, vertex data must be passed to the GPU.

4. *3D Rendering*

   Create a 3D environment and test models, display the 3D objects on the screen.

5. *Object Representation*

   The 3D objects on the screen must have a class to represent them. These objects will have properties, such as position, rotation, scale, vertex count, and so on. The project will use an OOP approach for this purpose.

6. *Camera + Camera Control*

   The 3D objects will need to be viewed from many different angles, so the goal is to implement a camera that can be moved freely across the 3D environment.

7. *GUI (Graphical User Interface)*

   This interface will be used to view object details, create new objects, manipulate objects, and so on. This UI will be created using an external library called ImGUI.

8. *Object Selection and Importing*

This engine will include object importing using the external Assimp library. Any 3D file extension will be possible to import. Objects will also be selectable on the screen.

9. *Lighting*

The lighting will be done using a simple Diffuse Lighting method, which will give the objects shadows relative to the light source in the 3D engine.

10. *Comprehensive Unit and Feature Testing*

The project will include unit and feature tests that will be run using CMake.

Success Criteria:

1. Develop an engine capable of 3D rendering
2. Custom object importing
3. Object movement in 3D space
4. Lighting
5. Simple GUI

Research and Background

Game Engine Architecture

A game engine is a piece of software developed specifically for the purpose of creating video games. A game engine is made up of multiple parts, some of which include; A rendering engine, a sound engine, an asset manager and a development environment using a scripting language.

The research on this thesis originally began as a study on the development of game engines, and the intended outcome was to create a simple game engine which met all the criteria listed above.

The problem with making the thesis about game engines is the incredible complexity of each individual part of a game engine. Just creating 3D rendering from scratch took up many months, not to mention the research required to make it work well and behave correctly. The thesis has now pivoted to 3D rendering solely, and the system architecture of creating 3D software.

Graphics API

This section will go over a quick introduction to rendering, the basics of OpenGL, the differences between industry standard graphical APIs and using APIs to communicate to the GPU.

Introduction to Rendering

Rendering is the concept of displaying information in the form of images on the screen. The computer screen (or display) is in essence a collection of pixels. A pixel is a point on the screen that can display either a red, green or blue color. These colors, when mixed at different intensities, can form a complex image. An average computer screen has 2073600 pixels, or a resolution of 1920 x 1080. The whole process of rendering deals with turning some kind of computer-generated image from information to each pixel on the screen.

The CPU (Central Processing Unit) is very good at processing complex tasks; however, rendering an image to the screen is more than a complex task. It's a task that requires many steps to be processed simultaneously. Each pixel needs to be updated extremely quickly, and it would simply be inefficient to do this process sequentially using a CPU. This is where the GPU (Graphics Processing Unit) comes in. The GPU can run many tasks simultaneously, or in parallel. The GPU is the main processing unit used in rendering graphics to the screen in modern computing.

Basics of OpenGL

To start this part of the technical research section, it's important to talk about the graphics API that will be used in this project. An API (Application Programming Interface) is a block of code or piece of software that the programmer uses to gain pre-made functionality that they can make use of in their own application. The graphics API used for this project is OpenGL.

OpenGL is a state machine that is used as a communication layer to the GPU. This definition seems at first glance to be quite complex, but it is important for the rest of this paper to describe this definition's main components.

For the first section of the aforementioned definition, Shreiner et al (2013) described OpenGL as follows: "OpenGL is designed as a streamlined, hardware-independent interface that can be implemented on many different types of graphics hardware systems, or entirely in software (if no graphics hardware is present in the system) independent of a computer's operating or windowing system". They then go on to further elaborate that OpenGL doesn't provide any windowing (opening a software window) functionality or functionality for describing three-dimensional objects; it just provides functions which allow for the rendering of said object **(p. 9)**. The important thing to take away from this is that OpenGL is at its core an interface whose state can be changed by the programmer's commands. The state is a collection of options the programmer gives as input to OpenGL, and is kept in the OpenGL context, which is essentially a container for this type of state data (p. 15). For example, the programmer may choose to use any different version of OpenGL, which can house different versions of the 3D rendering functions. This would count as changing the **state**.

The second part of the definition is that mention of a communication layer to the GPU. OpenGL gives you many functions which provide an access point to some functionality of the GPU. This functionality is then executed per the programmer's request, and the GPU returns some output to the screen that matches your given instructions. For example, you may tell OpenGL to use the color of your choosing. OpenGL will take in this color, perform some operations on the GPU, and then use the GPU to display the color back to the screen. The important part to take away from this is that the programmer doesn't need to know low-level GPU code; they just need to know the functions in OpenGL that make that functionality happen. This is the essence of what is meant by the concept of a communication layer to the GPU.

So far OpenGL seems like it can do essentially everything required to create a 3D scene for this thesis' project. The two things that it can't do, however, are **defining the 3D shapes and creating a window.** OpenGL is mostly concerned with something called a framebuffer, which for now will be simply defined as pixel data storage on the GPU. It then sends this data to the display, but the display itself will not simply render this data on your screen. This data needs to be shown in a window, and that's where a library such as SDL comes in. On his GitHub page, Sam Lantinga (The creator of SDL) gives a quick definition: "Simple DirectMedia Layer (SDL for short) is a cross-platform library designed to make it easy to write multi-media software, such

as games and emulators" (Lantinga, S., n.d.). SDL will be used to handle the window creation that will allow OpenGL to render to the screen. This is done by creating an SDL + OpenGL window.

Combining SDL and OpenGL already builds the foundation for 3D graphics and software development for the project.

Vulkan and DirectX vs OpenGL

Vulkan and DirectX are two other APIs used by many companies in the industry. DirectX is Microsoft's general game API that covers not only 3D graphics, but also the other tools required for the process of making a game. This means that it comes with more features that might be necessary for a simple game engine. Vulkan is NVIDIA's graphical API that provides more low-level access to the GPU, which could potentially provide greater performance given programmer skill.

Comparing these two to OpenGL was the first step of picking a graphical API to use for this project. OpenGL was chosen because it's the easiest one to learn and use out of the two. It's also very stable and time-tested.

Communicating with the GPU (Graphics Processing Unit)

The GPU is the chip responsible for all graphical tasks on the computer. The important difference between the GPU and CPU (Central Processing Unit) is that the GPU works **simultaneously**, whereas the CPU works **sequentially**. The GPU can execute many tasks at once, but the CPU executes a complicated task one step at a time. The job of OpenGL is to be that layer that communicates with the GPU, has it execute many tasks quickly, display the result and then go back to listen for more instructions (Code snippets in appendix).

OpenGL being a state machine makes sense for this purpose. Shreiner et al. (2013) compare it to a railway track: the trains can only go one way, but you can flip the switch to change the rails they go down. OpenGL will run its functionality (the trains), but the tracks (or the execution) will be different based on the options you provide (p. 18). This is particularly useful for interfacing with the GPU, because you don't have to wait for OpenGL; you always know it will either run the train and give a return or error out.

OpenGL also has many inbuilt optimizations for the GPU that can be turned on, which saves the programmer the hassle of optimizing GPU memory usage themselves. In general, OpenGL can be thought of as the "communication protocol" that will be used for this project.

Full Graphics Pipeline and Implementation

The first heading is the basic rendering pipeline steps required to render any object in OpenGL. The next heading, shader deep dive, will go into a deep dive into OpenGL's shaders and how they're structured. After these two headings, the OpenGL implementation heading will explain how these aforementioned headings come together as a whole in technical implementation.

The VAO, VBO, and EBO heading will cover the data transfer pipeline and what these data structures represent. Lastly, the frame buffer heading explains where all the data collected and transformed by the shaders is stored, and how this is displayed on the screen.

Rendering Pipeline Steps

To display objects on screen, OpenGL runs some input data through a layered process of individual steps that sequentially build the object's representation on screen. Before explaining the individual steps, there are some prior knowledge and definitions required to understand individual parts of each step.

The term "shader" will be used a lot in the rest of this thesis. A shader is simply a process that takes in an input, transforms the input in some way, and then outputs a value. Shreiner et al. (2013) describes shaders this way: "Shaders, [...], are like a function call---data are passed in, processed, and passed back out". OpenGL shaders are written in a C-like language called GLSL.

Another important term to get acquainted with is the concept of a vertex. A vertex is a point location in 3D space. It holds three values; X, Y, and Z. A flat triangle floating in 3D space would have 3 vertices.

There are 6 steps in the shader process De Vries (2014). These 6 steps are described below;

1. A vertex shader that takes 3D vertex data. This step is important for turning a collection of coordinate data into a representation of 3D shapes, with depth, perspective, and so on.

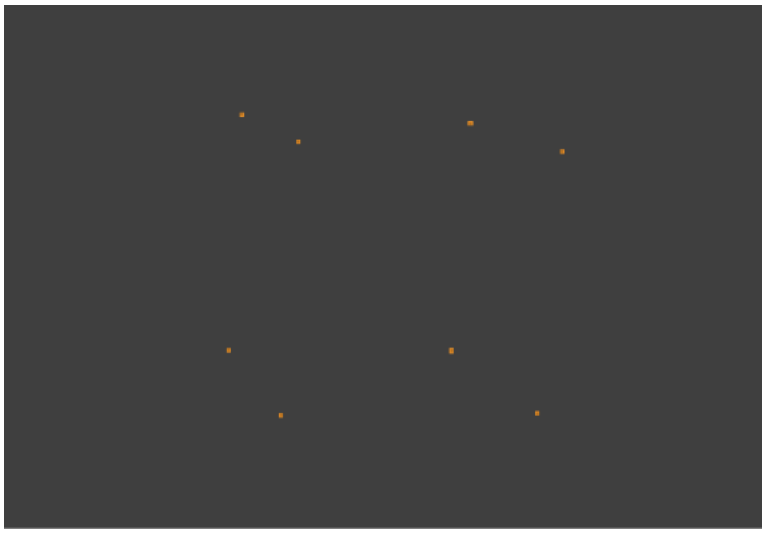

Figure 1, vertices are marked as yellow.

2. The geometry shader takes as input the output of the vertex shader. It generates the shape and can also generate more shapes (for example if it's given a square shape, it will turn that square into two triangles).

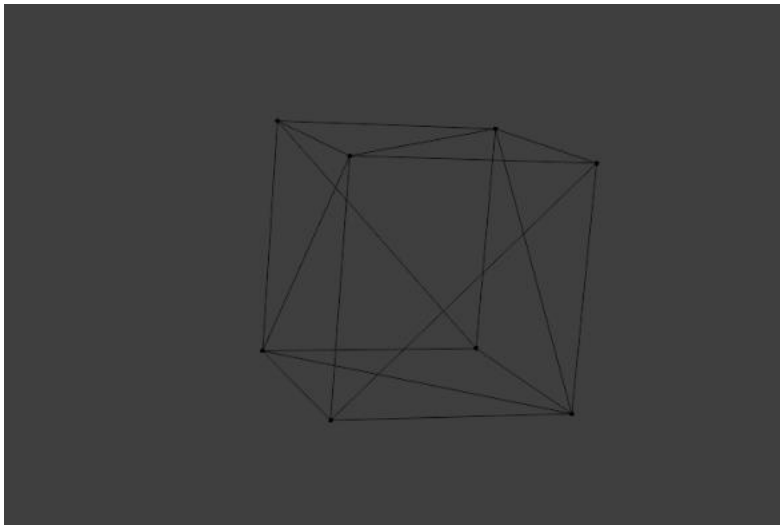

Figure 2, The cube is made of triangles that connect to vertices.

3. The primitive assembly shader takes as input the vertices in the form of triangles (primitives) from the output of the geometry shader step and assembles these triangles into the given shape.
4. The rasterization stage takes as input all the triangles and matches them to pixels on the screen. This action generates "fragments", which are stores of data per pixel. These fragments can hold important information, such as a depth value or color values.

5. The fragment shader, as the final step, takes the fragments generated by the rasterization stage and calculates the final color of the fragment pixel. This color can be influenced by lighting, color, texture, etc.
6. (EXTRA) This output is then passed through some depth checks and blending checks just to see if the pixel is hidden behind another object. This creates depth.

This array of steps will display everything you see on your screen if the software you're running uses OpenGL. For the purposes of the thesis project, the engine will use simple diffuse lighting.

Shader Deep Dive

This section will be used to explain shaders in further detail. The actual code won't be covered, but the function of the shaders used will be described in a visual way. Let's take the knowledge from the previous section and apply it here. A shader needs an input to process into an output. How does a shader in OpenGL know what input to take in?

Most of OpenGL's functionality is tied to a structure called OpenGL Objects. These objects hold information and implementation about the functionality you're trying to access. For example, in order to create working shaders, they must first be attached to an OpenGL shader program. The shader program is one of those OpenGL objects that can be created and bound to the current OpenGL context.

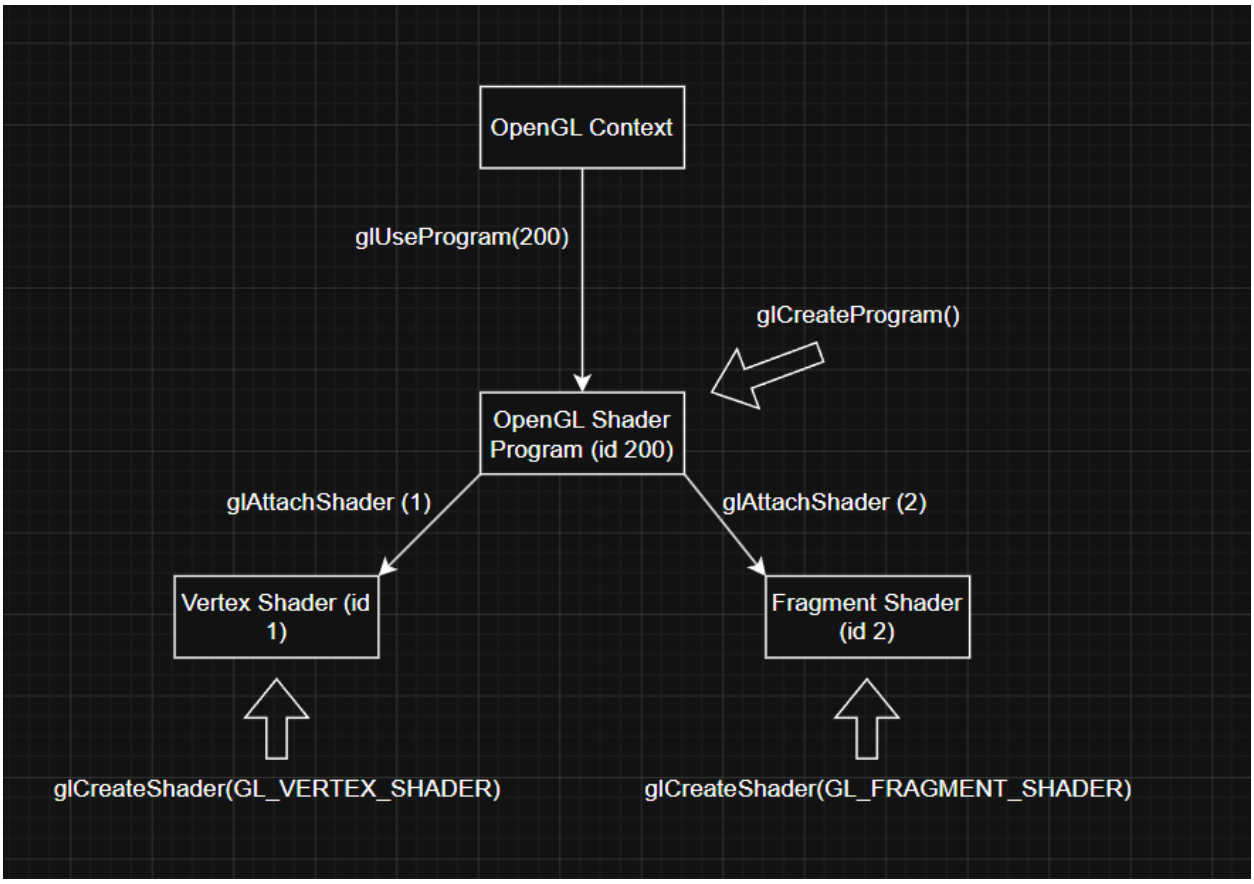

Figure 3, simplified graph: the first and second shaders are created and bound to the overarching shader program, then that shader program is used by the context in rendering

As can be observed in the figure above, the OpenGL shader program is created with an ID, and then both of the shaders are created with their own IDs. The shaders are then attached to the shader program, and then the shader program is used by the current OpenGL context, which results in the program being used in the rendering loop.

The shaders under the Shader program are created by using glCreateShader(GL_VERTEX_SHADER) and glCreateShader(GL_FRAGMENT_SHADER). By creating the shaders this way, the shaders are set up perfectly for the next step, which is glLinkProgram(id). If there are valid vertex and fragment shaders attached to the vertex and fragment slot of the program, the linking step will combine them into an executable program. This means that the flow of information in the shader program is now clear; the vertex shader's output will be taken as the fragment shader's input (consult the shader steps if this set of steps is unclear).

GLSL, the programming language used to create individual shaders, is a language close to C that has many variable types that are used to hold and pass information used for processing.

Shreiner et al. (2013) lists and describes the different basic GLSL data types, such as

floats, ints, doubles, uints and bools. These are found in many programming languages, but the types that will be the focus of this section are GLSL's aggregate types. A float, or a floating-point number, is essentially a number with a decimal point. The vec2 type is a vector type that contains two floating point numbers. This vector type goes up to a vec4. Each of the simple types listed above have their "vector" variants. There are also matrix types for floats and doubles. Representing numbers in these kinds of "groups" is extremely important for 3D computing and rendering, which will be explained further in the upcoming mathematics section.

Storage qualifiers are statements made before a variable definition that modify the variable's behavior. The storage qualifiers that will be required for this paper are the in, out, and uniform qualifiers.

The "in" qualifier designates the input variable. It can be used to declare two types of inputs: 1. The previous shader output variable or 2. A vertex input. The vertex input can only be declared for a vertex shader.

The "out" qualifier designates the output variable of a shader stage. For example, you can say "out vec3 someOutput", and the shader will output what you've described. In the next related shader, you will have to type "in vec3 someOutput".

The "uniform" qualifier, when written before a variable declaration, states that the variable will be given to the shader by the application before the shader is executed. These variables are global across all shader steps and can be accessed in the OpenGL C++ code and manipulated. For example, a shader can receive a new uniform vec3 color variable every time it's executed, because the color variable is sent by the application. (Shreiner et al., 201

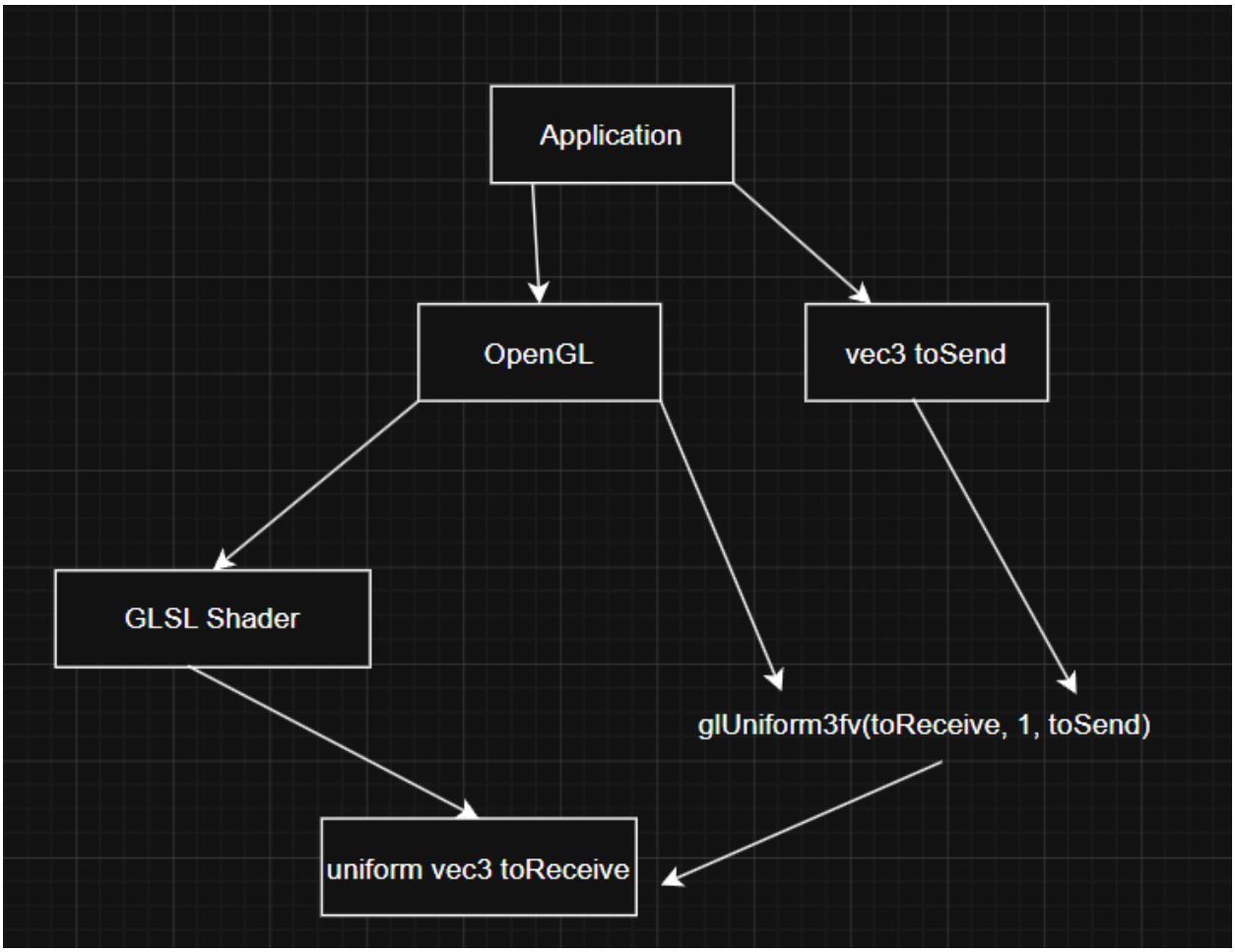

Figure 4, simplified graph: The application sends its processed toSend variable to the OpenGL glUniform3fv function, which updates the shader's toReceive variable with the data from the toSend variable before rendering

OpenGL Implementation

Now that OpenGL's core functionality and shader concepts have been introduced, it's time to consider how this can be implemented into an OpenGL + SDL application. This section, and the sections under it, will go into further detail about OpenGL, and may include code snippets and graphs to illustrate complex ideas. This section specifically will cover a quick overview of the OpenGL setup in the project, outlined by De Vries, J. (2014).

First, the C++ programming language will be used for all the application code. There are a few options for a C++ compiler, but considering the project's complexity it was decided that the Visual Studio compiler + IDE suits the project's needs. Considering the application will implement several C/C++ libraries, CMake will be used as a build tool to create a CMakeFile that will handle building the project.

SDL will then be used to create an event loop and an OpenGL context. This loop will handle all "runtime" code (such as rendering instructions, button events, shader updates etc.). SDL will also be used to open a window with an OpenGL context, which will allow the GPU to display its rendering output to the screen.

Lastly, all external files will be dropped into their sorted directories/folders and will be searched for by SDL at runtime. For example, the final build will have a model, texture, and shader directory, which the user will be able to upload files to.

VBO, VAO and EBO

VBO, VAO, and EBO are OpenGL data structure objects that are the key to all rendering and application to GPU communication.

The VBO, or Vertex Buffer Object, is a buffer that holds vertex data to be stored on the GPU's memory. The VBO itself holds no information on how to arrange the vertices. The VBO is then put into a vertex buffer (storage), and the array of vertices that will be used is then inserted into the buffer. A programmer can set it up so that their application uses one giant VBO for everything in their scene, or they can make it so that each small object has its own VBO. This project will use the latter approach.

The VAO, or Vertex Array Object, stores attribute calls that are related to drawing the vertex data. In other terms, it stores the information OpenGL requires to make sense of the VBO that's currently bound. It can be thought of as a storage of information on how to differentiate the vertices in the VBO. For example, if the VBO has data such as [0.2, 2.0, 0.3, 2.0, 0.3, 0.4], the VAO could be programmed to say: "The vertex begins at the start of the array and ends at the end of the third number. So, this array holds 2 vertices, (0.2, 2.0, 0.3) and (2.0, 0.3, 0.4)." The VAO essentially stores metadata about the layout of the vertices. The VAO can also be used to differentiate between different vertex types, for example the first vertex (0.2, 2.0, 0.3) can be a "positional vertex" and (2.0, 0.3, 0.4) can be a "color vertex".

The EBO, or Element Buffer Object, is a buffer that generally stores the indices of each vertex. These indices are used by OpenGL to determine which vertices in the VBO it should draw. A cube, created in 3D, can have vertices that perfectly overlap, so the EBO is used to filter the vertices that are unique and don't repeat. This is a major optimization because the GPU doesn't have to hold duplicate vertex data.

Combining all these buffer objects makes the process of storing and passing data required for 3D rendering possible.

Frame Buffers

In computer rendering, the pixel output from various types of rendering operations is written to screen buffers. Screen buffers are simply data types that store per-pixel information for the rendering outcome.

De Vries (2014) writes: "The combination of these buffers is stored somewhere in GPU memory and is called a framebuffer". All these previously mentioned rendering operations get grouped into the base frame buffer, and the output to the screen is based on that frame buffer.

There can be multiple frame buffers in one OpenGL context. In the implementation section down the line, there will be mention of an extra frame buffer only used for screen object selection

Mathematics for 3D rendering

Coordinate Systems and their Transformations

In OpenGL computer rendering, there are two types of coordinate systems that are visible: the 2D coordinate system existing on a 2-Dimensional plane, and the 3D coordinate system existing in a 3-Dimensional world.

All 3D graphics are made of 2D shapes called polygons. These are usually triangles and are also known as primitives. To render a 2D triangle on the screen, as mentioned before, it's required to send an array of vertices to the OpenGL shader pipeline. A 3D shape is a collection of primitives. If OpenGL is told to render a cube, it will construct primitives (triangles) out of the vertex data given to it. These primitives create the faces of the 3D object.

For the purposes of rendering these shapes to the screen, de Vries (2014) states that OpenGL expects all vertex coordinates to be normalized to a range of –1.0 to 1.0. All vertices outside of this range are not rendered on the screen. The rasterizer (from the shader steps) requires these NDC (Normalized Device Coordinates) to represent objects as pixels on the screen. The vertices are taken through multiple transformations before the rasterizer can turn them into NDC to represent the object as actual pixels on the screen.

These transformations are used to take the vertices from one "space" to another. There is the local space, world space, view space, clip space and finally screen space. Each of these spaces is their own unique 3D coordinate space, or coordinate system if you will.

This is a list of every space type and the sort of transition required to move to the next space, de Vries (2014);

Local space is the space the object arrives in when imported into the pipeline. The simplest way to explain local space is that all the vertices are positioned relative to the whole object's center, or "origin". In other words, the (0,0,0) position is in the center of the object, and all the object's vertices exist in that reference frame. To move from one space to another, a transformation matrix is used in each step. These matrix operations all have names, and the first one is called the Model Matrix. Each vertex in the object is multiplied by this model matrix.

World space is the result of the Model Matrix transformation from the previous step. What changed is that now all the object's vertices exist relative to the 3D world's origin. In other words, (0,0,0) is at the center of the world, not the object. To move to the next step, the View Matrix is used.

View space is the result of the View Matrix transformation, and it positions each vertex from the perspective of the camera. With this step, the process is almost complete. The next step is to transform each of these vertices with the Projection Matrix into the clip space.

Clip space firstly normalizes the vertices into the previously mentioned NDC (Normalized Device Coordinates) and then applies a perspective if the programmer chooses to include a perspective view. This distorts the vertex positions in such a way to mimic real life perspective. Finally, a Viewport Transform is applied, which transforms each NDC into its actual screen position (for example a 1920x1080 window). This is called screen space, and it's what is rendered as the output of the rasterization step. Below is a quick diagram created in blender that explains how the camera sees an object after it's transformed to view space and moved into screen space.

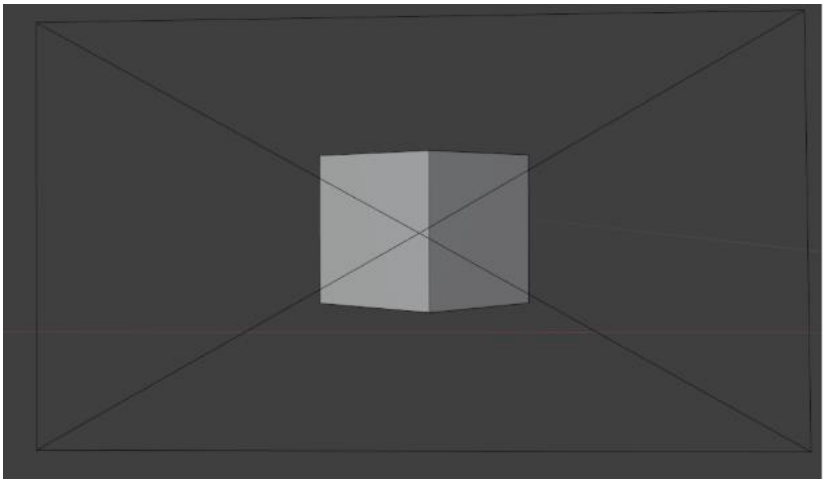

Figure 5, the camera's point of view

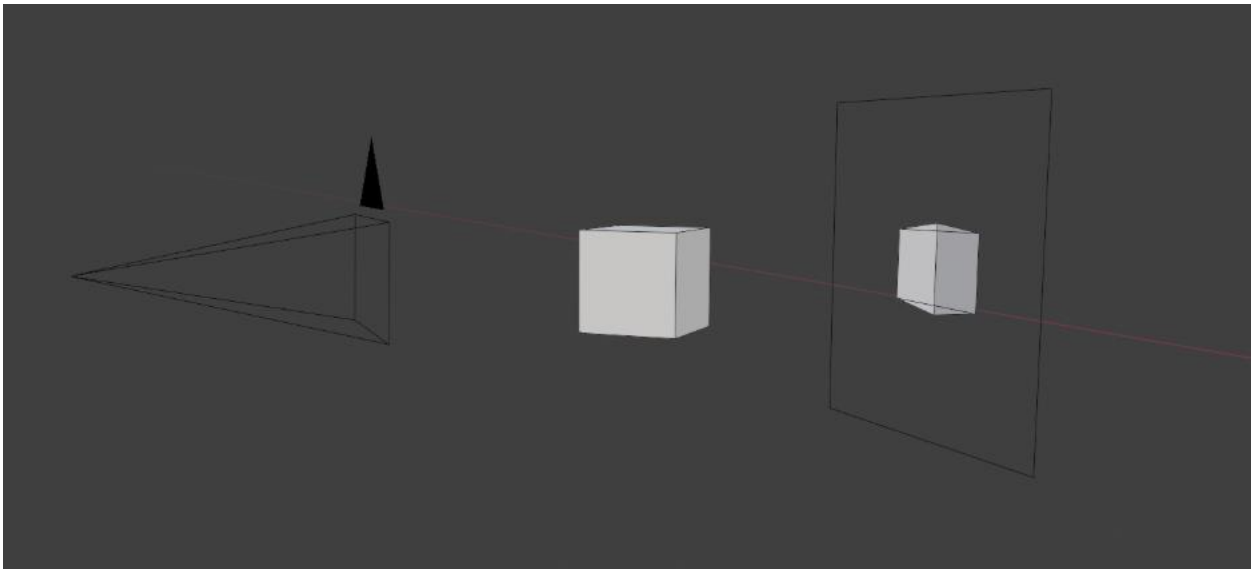

Figure 6, what the camera is looking at, and what the actual final product is on the screen (screen space with a foreshortened cube). The foreshortening is an outcome of the projection of the cube.

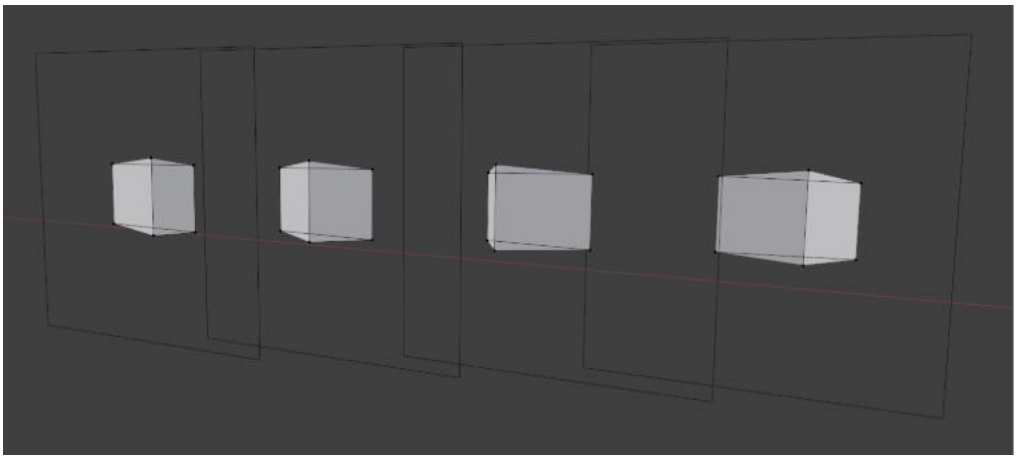

Figure 7, the individual frames of the action of the camera rotating around the cube

Implementation of Mathematics concepts

The previous step to rasterization was theoretically sound, but how are the transformations applied in real 3D software? The glm library for C++ includes in it the transformation matrices for all the steps from local space to screen space. All that's required is to create the matrices and pass them to the vertex shader so they may be multiplied. The glm library also includes definitions for mathematical vectors which will be used to define collections of vertices. For example, a 3D object will house a list of vec3 vertices.

Systems Engineering

The systems engineering section of this document explains the software development of the engine itself as a product. This section won't interact with the previous sections and will explain the lower-level concepts required to make a piece of software interact well with the computer and the user.

For example, the user needs to be able to use their keyboard and mouse to perform actions in the game engine overlay. They also need to have the engine store the keyboard and mouse data to later be used by the scripting language to create game functionality. The game engine needs systems that provide the user with all the tools they need to work on their game.

Object Representation

Objects in 3D engines are essentially containers that hold information. The way that these objects are represented can vary, but this document will discuss the way that two game engines accomplish this task. Even though the focus of this paper isn't game engines, it's still useful to take a look at how 3D software such as game engines handle objects.

The Unity engine implements an object-oriented GameObject system. A GameObject in Unity is a container for components, and each component describes the behavior of a GameObject. For example, the cube would have a Mesh filter component and a Mesh Rendered component, which both tell the rendered what kind of shape the cube is an how to render it. In Unity, when a user saves a custom-made script and attaches it to a GameObject, the script becomes a component. There are also pre-made Unity engine components, so the idea for game development reusability is for users to create their own scripts, while interfacing with pre-made Unity components. Importantly, every GameObject starts with a Transform component, which cannot be removed.

This component facilitates the GameObject's position, rotation and scale in 3D space. It's also important to mention that each GameObject with a transform can have a "parent" and a "child" object, which will inherit the object's location in 3D space. (Unity Technologies., 2023)

This very "object-oriented programming" implementation of Object representation is simple and effective. It compartmentalizes all functionality and behaviors as components, which are reusable and can be attached to multiple objects.

Input and Output

The input and output for a game engine are how the user interacts with the software, and what information the software throws back to the user. The game engine needs to be responsive to keyboard commands, and it needs to respond to mouse clicks. These are the bare minimum requirements.

The keyboard, for example, must be read and stored in memory, to be later re-used by the game engine user to program their game.

For a simple game engine, it's possible to simply use a development library to provide a pre-created I/O system that can be implemented into the game engine code. An example of this type of development library is Simple DirectMedia Layer. SDL3, to be exact, provides all of the basic functionality that a piece of software needs to be usable for the average user. It provides Input and Output, Audio, a File System API and so on. (Lantinga, S., 1998).

Audio

Commercial game engines such as Unity use their own proprietary audio code and implementation, which is extremely complex and robust, taking into account audio mixers, audio playback etc.

This is where Simple DirectMedia Layer can be used to help create an audio system for a simpler game engine. SDL has the concept of an SDL_AudioStream, which lets the programmer create a system that can stream, mix, play, and record audio. For a simple game engine, all that is required is playback of audio and audio mixing. Using SDL, it's possible to have the cube play audio when a specific event is triggered.

File System

Game engines use operating system calls to manage files in the file system, but they don't usually directly open the system's file explorer. Usually, game engines wrap the host OS' file system in their own implementation, which is done to provide a generalized file system that works the same on all operating systems. The OS' file system implementation needs to do everything that a regular file system can do, such as changing file names, opening and editing different files, scanning directories etc (Gregory, 2018).

All commercial game engines have a file system, and it's usually programmed for the specific game engine from the ground up. For a simple game engine, using SDL makes implementing file systems easier for the programmer. SDL provides a CategoryFilesystem API, that provides all the basic functions required to create a file system wrapper for the game engine. SDL also provides an IO Stream implementation (an IO Stream is a software component that facilitates data flow between a piece of software and a destination). (Lantinga, S, 1998).

The actual asset importing is done using the Assimp library, which has its own data structure called "nodes". An .obj file can be imported, and the object will be extracted into a scene node, which can then be parsed in the engine and the vertex data assigned to an object. This object then creates a VBO, VAO and EBO and OpenGL's rendering steps then act on it.

Resources and Memory

Resources are essentially a designation for all media types that a game engine supports for use in creating games. They can be things such as meshes, sounds, animations etc. Commercial game engines have robust resource managers that have support for multiple file formats.

Requirement Analysis

Similar Applications

The applications most like this project's 3D engine are; Blender, Tinkercad and Windows 3D Viewer. Blender is a complex 3D modelling software whereas Tinkercad and 3D viewer are more in the scope of this project. Below is a quick breakdown of these applications and what lessons should be taken for this project.

3D Viewer

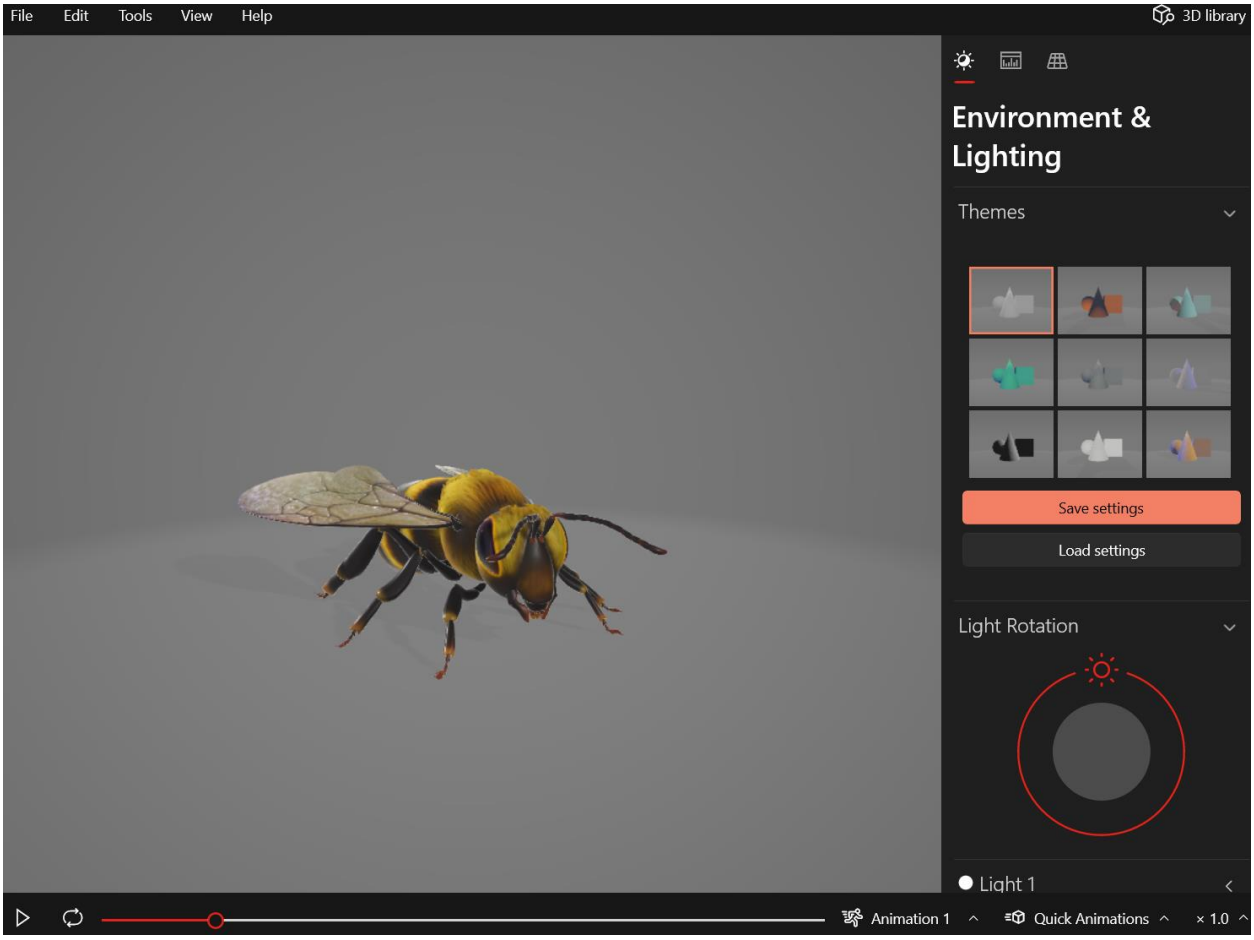

Figure 8, Microsoft's 3D Viewer

3D Viewer is Microsoft's 3D model viewer. In the figure above it's obvious what the most important elements are; the 3D viewport, a model with capacity to display materials, an environment & lighting config on the right, and an animation playback segment on the bottom. This is a very effective design layout as the model is clearly visible and the camera can be rotated.

The thesis project's 3D engine should, like 3D Viewer, have a model viewport, a rotatable camera, variable light sources, object importing, and a simple UI. These are the basic requirements for a functional 3D model viewer.

Tinkercad

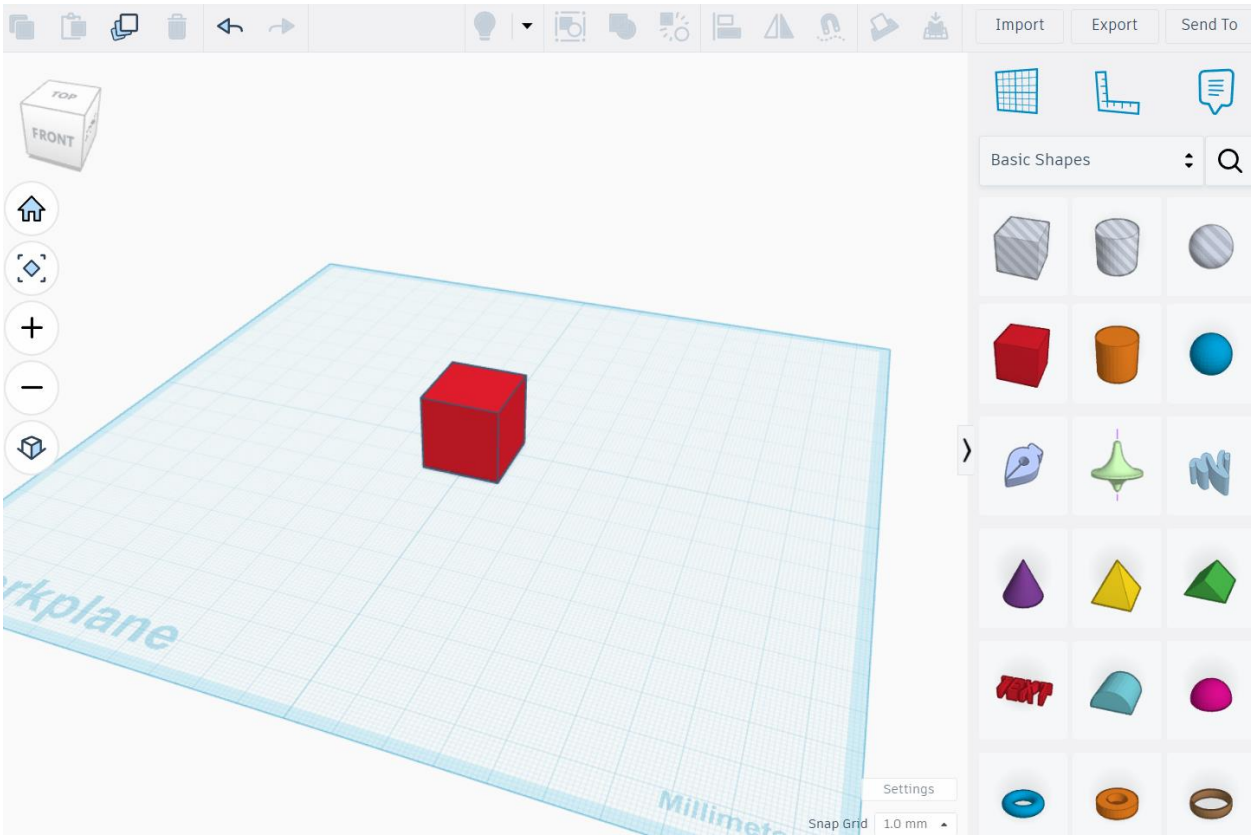

Figure 9, Autodesk's Tinkercad

Tinkercad is Autodesk's simple 3D modelling tool. Like 3D viewer, there is a 3D viewport, but the UI is more focused on quick access to different kinds of models. These models can be dropped into the scene and moved in different ways relative to the base plane. This software has modelling functionality, and models can be imported/exported.

The takeaway for this project is to include a way to create simple shapes quickly, so that the user can get to edit their scene as fast as possible. There should also be a grid or base plane that shows how the objects are positioned.

Blender

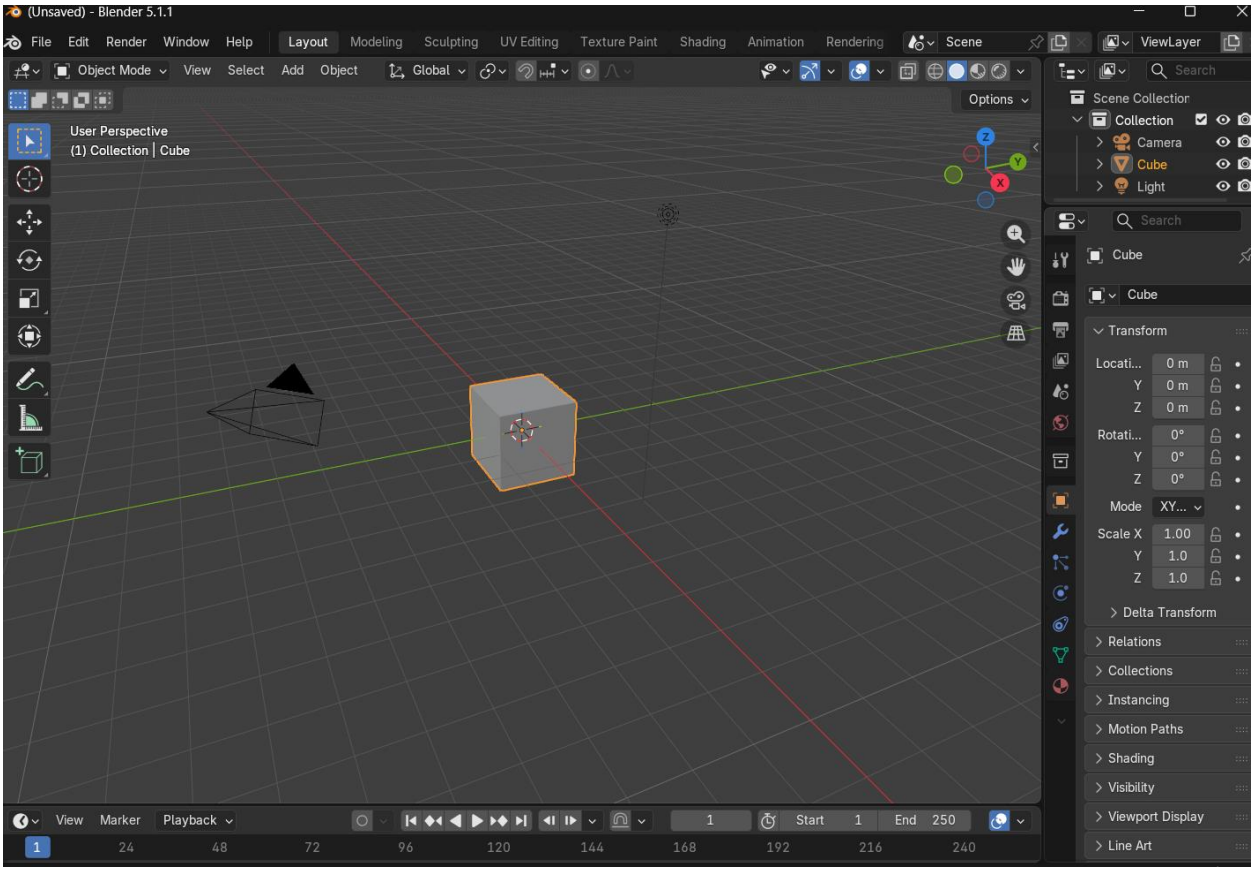

Figure 10, Blender

Blender is an open-source 3D modelling software that comes pre-packaged with fully feature rich functionality. The features are too many to list here, but the important takeaway from this application is the object system. Each object can have a parent object, and different objects have different “components”. For example, every object has a 3D position, rotation, and scale, but a “light source” object has different properties to a “cube” object. The light source can emit light, where a cube is simply a mesh receiving that light. Objects can also be selected and moved using the viewport.

Key Features

Based on the application review in this section, the thesis’ 3D engine should have the following key features to make it a piece of simple 3D software;

- 3D rendering
- Viewport to display rendering
- Editable Shaders
- Lighting
- Color
- Object importing
- Object movement, scaling, rotating

Implementing the above features will ensure the engine is at least a usable 3D model viewer, which is already a large undertaking in software engineering.

Implementation

This section of the document will explain in detail the development process and methodology used in creating the 3D engine. These steps are in chronological order and appear in the same order as the issues on the GitHub page.

Build Dev Environment

The most important part of this project, and the part that will most impact the further development of the project, is the development environment. The IDE (Integrated Development Environment) of choice is Visual Studio 2022. Why was Visual Studio 2022 chosen over other free options? Visual Studio has its own C++ compiler, and once it's installed IntelliSense (Visual Studio's code comprehension tool) will help greatly with understanding the C++ code. Visual Studio also offers a robust set of features to make development and debugging easier.

With Visual Studio installed, the next step was to create a CMake project in Visual Studio, which will allow the use of CMake when building the project. CMake is a cross-platform tool for building C/C++ projects, and this is very applicable for this thesis project as the build process can be modified by adding a CMakeFile. This CMakeFile allows the programmer to fully control all the steps taken in building and compiling the project, which allows for easy integration of external libraries. The CMakeFile has been upgraded with every major pull request, and it is set up to generate a Windows compatible executable file, and all the other data required to run the project in its release state. When a new feature is mentioned in this document, it's safe to assume the CMakeFile was modified in some way to accommodate it.

The basic flow of the dev environment then becomes; Make a change in the code, run the project in Visual Studio (which then runs the CMake build), debug the code using Visual Studio tools, and finally push the code to GitHub.

```
cmake -S . -B build
```

Figure 11, command that configures the project into the build directory

```
cmake --build build
```

Figure 12, command that builds the project into the build directory

```
cmake --build build --config Release
```

Figure 13, command that builds the project into the build directory as a Release build

Window

As mentioned in the Technical Review section, a window is required for the GPU to be able to render any output to the screen. In OpenGL, there is no way to open a window, but the external SDL library can open a window with an OpenGL context (or OpenGL "instance").

The SDL subsystems required for the project must first be initialized. The video and events systems are initialized, as these are the systems required for handling the update loop, the rendering, and so on. After that, the SDL OpenGL window attributes are set, and the version is set. The window is then created with a set resolution. The context is created shortly after. Both of these objects, the window and the context, are stored in memory.

```
// Initialize SDL3
if (!SDL_Init(SDL_INIT_VIDEO)) {
    printf("SDL could not initialize video! SDL_Error: %s\n", SDL_GetError());
}

if (!SDL_Init(SDL_INIT_EVENTS)) {
    printf("SDL could not initialize events! SDL_Error: %s\n", SDL_GetError());
}
```

Figure 14, initializing SDL subsystems

```
//Setting GL window attributes
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);

// Create an SDL window with OpenGL ES context
window = SDL_CreateWindow("Jan's Game Engine", (int)resolution.x, (int)resolution.y, SDL_WINDOW_OPENGL);
```

Figure 15, creating window

The next objective to get OpenGL running correctly is to use a library called glad. OpenGL is installed slightly differently on each GPU, and glad works to provide easy access to all OpenGL functions across all systems.

OpenGL's functions and states can now be accessed. The last step is to create a loop using SDL events. This loop will run forever until the SDL_EVENT_QUIT is triggered, at which point the loop will fully terminate, and the window and context will be deleted, quitting the application. It is important to fully delete all memory being used, so as to avoid memory leaks.

```
while (running) {
    while (SDL_PollEvent(&event)) {
        switch (event.type) {
        case SDL_EVENT_QUIT:
            running = false;
            break;

        default:
            break;
```

Figure 16, a quick snippet of the loop, with the sdl event polling and quit case.

```
SDL_GL_DestroyContext(context);
SDL_DestroyWindow(window);
SDL_Quit();
exit(1);
```

Figure 17, the set of functions that run when the quit event is called.

2D Rendering

Rendering a 2D shape is quite a big jump from simply opening a blank SDL_GL window. The first step was to render a color to the screen. The way OpenGL renders is called double-buffer rendering. As mentioned before, a frame buffer is essentially a collection of all pixel data to be displayed on the screen. The double-buffer rendering method deploys a front and back buffer. The front buffer is displayed to the screen, and the back buffer is generated while the front buffer is still being displayed. The back buffer then becomes the front buffer, and the process repeats. To give the window color, the front buffer must be cleared first and then painted. OpenGL is given a color to clear with and then instructed to clear the screen with said color and then swap the buffers. When this is put into a loop, it means that the screen is constantly refreshed with a color, effectively painting the screen. It can be thought of as the GPU throwing paint onto a canvas, and then quickly scrambling to throw paint onto a new canvas which will replace the old one.

```
glClearColor(1.0f, 0.5f, 0.3f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
SDL_GL_SwapWindow(window);
```

Figure 18, choosing a clear color, clearing the screen with said color, and swapping the "canvas"

An important concept to re-iterate is the nature of OpenGL's state machine. In figure 14, the glClear isn't taking in the glClearColor in the traditional sense; it's instead looking for the color in the state's memory.

Now that the window is displaying a color, the 2D shape is ready to be implemented into the project. The VBO is created, bound to the context, and then the vertexArray is inserted into the VBO.

```
float vertexArray[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};

unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(VBO, sizeof(vertexArray), vertexArray, GL_STATIC_DRAW);
```

Figure 19, creating and using a VBO

At this point OpenGL's shader pipeline comes into play. The shaders are written as string variables, compiled and then passed into a shader program and linked.

```
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);

unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);

unsigned int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);
```

Figure 20, creating vertex and fragment shaders from source code, compiling shaders, creating a shader program, and attaching shaders. The shaders are then linked together, and the program is used in the project (Refer to figure 3 for graph and clarification).

With the shader program created, the next step is to create the VAO and give it an attribute pointer. This attribute pointer will determine which data from the vertex array represents which vertices. After this, down in the render loop, the shader program is used, the VAO is bound, and the glDrawArrays function is called. This little loop is running constantly, and when the GPU splashes the canvas with color, it now quickly draws a triangle before clearing the screen with the same color again.

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(VBO, sizeof(vertexArray), vertexArray, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Figure 21, the VAO is added to the code, and it now denotes how the VBO will be used

```
glUseProgram(shaderProgram);

glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Figure 22, in the render loop, these functions draw the triangle every time the screen is cleared

The above steps are the base foundation of the project, and they are the first steps undertaken to render anything. In the future, these steps will be refined and an EBO will be introduced.

Before moving on to the next steps of implementation, it's important to mention that this is a simplified version of the process. This is done for the sake of brevity and to condense the complicated code into the most important parts.

Figure 23, an example of what the screen would look like right now.

3D Rendering

With the rendering process now implemented, the next step was to clean up the shader importing and then push the cube into the third dimension.

The shaders were being imported by parsing a string into the shader program from the main file. This was not efficient for long-term shader programming clarity, so the next step was to create a shader class which will read the shader.

This is where the project took a general step into OOP principles, with new implementation having its own class assigned to it. For example, shader implementation was written into a shader class.

```
class Shader {
public:
    // ID Of shader program
    unsigned int ID;

    //Constructor that takes in a vertexPath string and a fragmentPath string for building the shader
    Shader(const char* vertexPath, const char* fragmentPath);

    //Use the shader
    void use();
};
```

Figure 24, the shader class. The constructor of the class takes a path to the shader files. The use method just calls glUseProgram(ID)

The shader constructor reads the vertex and fragment shader files from their file locations and then converts them to strings. The regular vertex and fragment shader creation steps are then taken, using the file contents as the shader code. The shader program is then created and linked. So now the code that was floating in the main file is now in its own file, with the shader source code being separated into files.

To use 3D mathematics (such as 3D+ matrices and vectors), it's possible to write custom implementation, but for the sake of time this project will use the glm library. This library comes with all the mathematical functions required to represent 3D objects. Functions such as glm::transform, glm::rotate, glm::scale and data structures such as gm::vec3 and glm::mat4 are crucial for representing 3D objects. Since OpenGL can't represent objects mathematically, using a library like this is crucial.

Representing 3D objects (meshes) requires a bit of a setup. First, a mesh needs to be represented using a vector (array) of vertices. The vertex is more than just a set of locations, however. In this project, the vertex is defined as a struct which contains a position vector, a normal vector, and a texture coordinate vector.

The position vector marks the points in 3D space where the vertex is located. The normal vector is a vector that is perpendicular to a surface, which is required for lighting calculations and much more. The texture coords 2D vector stores the uv coordinate data for each vertex, which will come in handy later when textures are implemented.

```
struct Vertex {
    //Laying out the
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TextureCoords;
};
struct TextureData {
    unsigned int id;
    std::string type;
    aiString path;
};
```

Figure 25, structs used in mesh, vertex and texturedata. Note the glm vector data type

```
class Mesh {
private:
    unsigned int VAO, VBO, EBO;

    void setupMesh();
public:
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<TextureData> textures;
    void draw(Shader shader);

    Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices, std::vector<TextureData> textures);

};
```

Figure 26, mesh class using vertex and texturedata structs

```
void Mesh::setupMesh()
{
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), vertices.data(), GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
        indices.data(), GL_STATIC_DRAW);
```

Figure 27, setupMesh method which generates VAO, VBO and EBO

```
// vertex positions
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
// vertex normals
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
// vertex texture coords
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TextureCoords));
```

Figure 28, vertex attributes are set so the shader knows which part of the VBO belongs to vertices, normals and texture coords.

As pictured in above figures, each mesh generates, binds and applies data to its own VAO, VBO and EBO. The glBufferData function takes in the vector of vertices as data and passes that data to the mesh's VBO. The vertex attribute pointers are then set. Each mesh has its' own draw method, which is called each frame by the renderer. Before that, however, there is a missing step to achieving 3D. The key is in the vertex shader.

The vertex shader, in the 2D implementation, simply takes in the position of a vertex and positions it on the screen (in clip space specifically). However, to reach 3D, the vertex positions

much be multiplied by 3 transformation matrices; projection, view and model. These matrices are created using glm::mat4 and then passed into the vertex shader.

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
```

Figure 29, model, view and projection matrices come in from the data in the application, and then everything matrix is multiplied together for each vertex

```
glm::mat4 model = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(45.0f),
    (float)stateHandler.getResolution().x / (float)stateHandler.getResolution().y, 0.1f, 100.0f);
Camera mainCamera;
//----------------------------
```

Figure 30, the model, view and projection are calculated in the application, then updated in the update loop and passed into the shader's uniform variables

Each vertex is then multiplied by the projection, view, and model matrices. These matrices are created using the glm library, which is essential here due to its perspective function. They are then updated in the update loop and passed into the shader. The camera is a very complex object from which all the 3D space can be seen, but crucially, here it provides the view matrix for the calculation in the shader. This means that wherever the camera goes, and no matter the direction it points at, it influences the vertices rendered on the screen. This, combined with projection, creates a feeling of a true 3D space.

Now the last step to address is the renderer. The renderer passes up-to-date model view and projection matrices to the main shader from the update loop. It then tells the meshes to draw themselves on screen. With that, the 3D conversion is complete.

```
renderer.modelUniform(lightingShader, model);
renderer.viewUniform(lightingShader, mainCamera);
renderer.projectionUniform(lightingShader, projection);
gui.renderUi(stateHandler, lightingShader);
renderer.renderPass(myTexture, pickingTexture, lightingShader, stateHandler.getSimulatedObjects(), stateHandler);
//----------------------------------
```

Figure 31, renderer object.

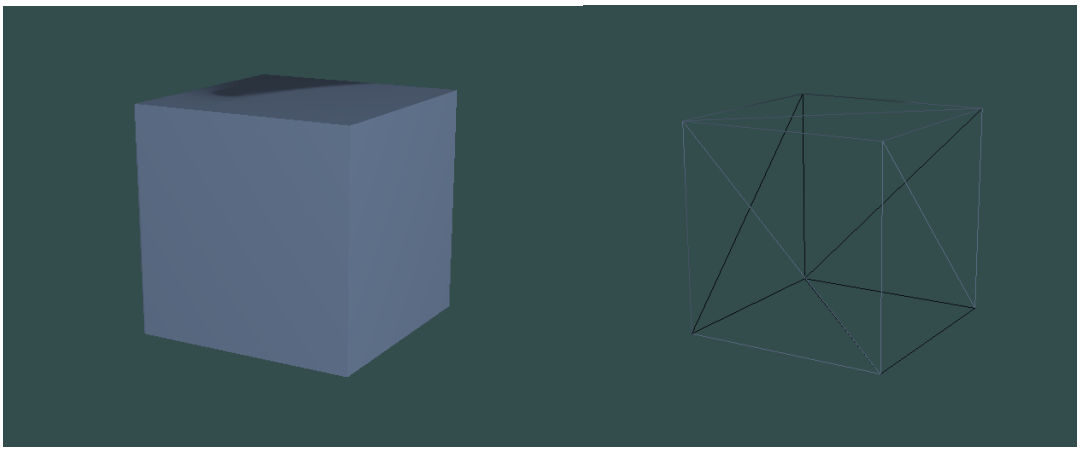

Figure 32, cubes, one is fully rendered and one is only rendered at the edges. Note the individual vertices that form the edges, and how they all form triangles.

Object Representation

Now that the meshes can be rendered, the question becomes; how is all this data managed in an accessible way? The answer lies in SimulatedObjects. These objects are containers that can store any useful type, defined by the user. The user can create a model type that holds meshes and then create a SimulatedObject and give it this type. The model becomes a component attached to the SimulatedObject. This paradigm is structured like the Unity component system.

```
class SimulatedObject {
private:
    bool enabled;

    std::unordered_map<std::type_index, void*> *components;

    static std::vector<const char*> allowedComponents;


public:
    glm::vec3 position;
    glm::f32 rotation;
    glm::vec3 scale = glm::vec3(1.0f, 1.0f, 1.0f);
    glm::vec3 axis;
```

Figure 33, a snipped of the SimulatedObject class.

The above figure shows a small snippet of the SimulatedObject class. Note that this object has a hash map of types and void pointers. This is the way to store any type and access its component. Also note the position, rotation, and scale variables. This object can be moved in the 3D world by editing these values.

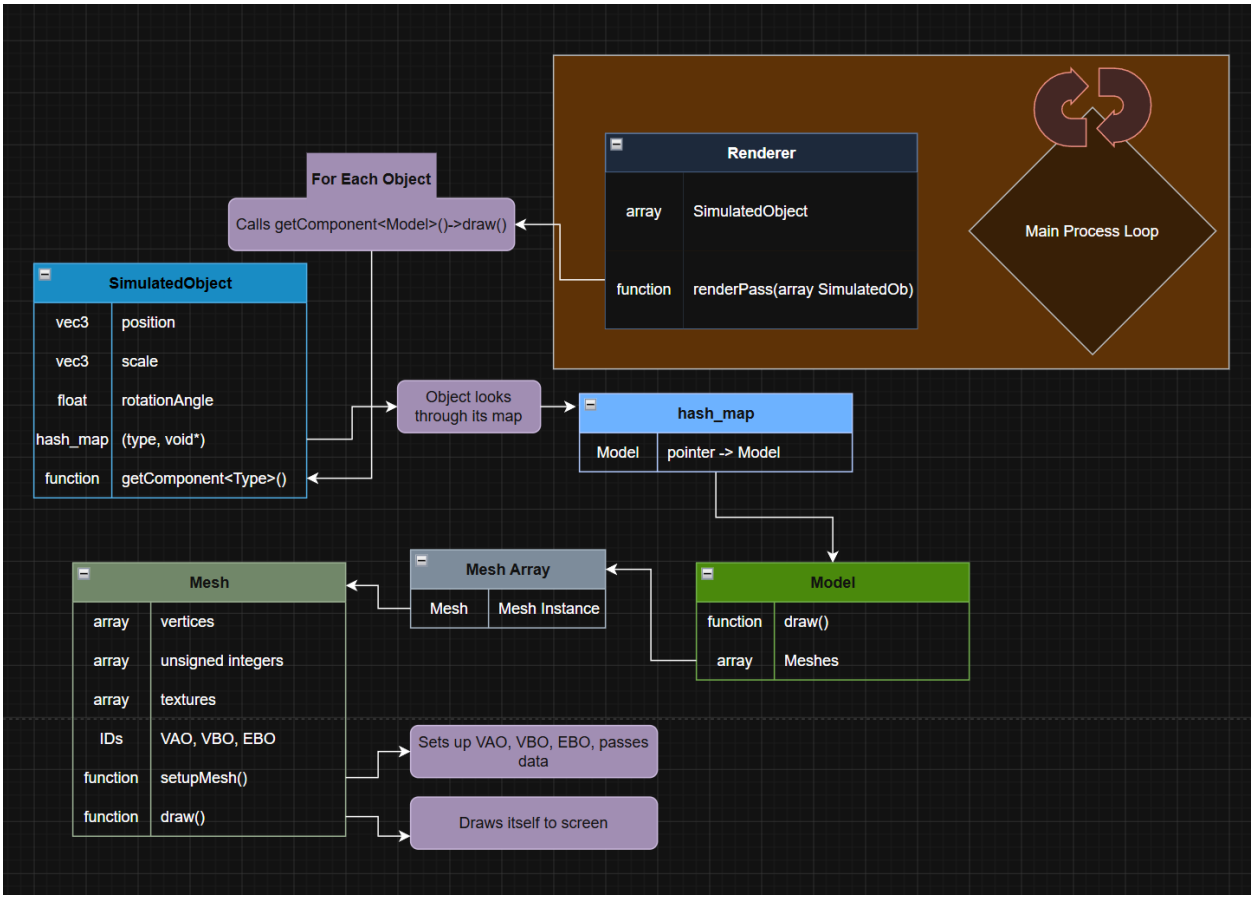

Figure 34, a highly simplified graph of the rendering process using SimulatedObjects.

Now that there is object representation in the program, the meshes aren't just floating in a void, but instead are bound to objects. The rendrer in the update loop, on each loop iteration, calls renderPass. The renderPass function performs complicated logic, but most importantly for this section, it calls each SimulatedObject's getComponent<Model> method. This method looks through the map of types and finds a model. The model is then told to use its draw() method, which looks through the array of meshes and calls their draw methods. This happens in every update loop iteration and is made simpler by having an array of SimulatedObjects. This SimulatedObject concept is very scalable, because any type can be thrown into that hash_map. So, the object could have a light source as well, which would call its own methods.

This can lead to undefined behavior, however, considering that a void pointer is used, but that's up to the programmer's discretion right now.

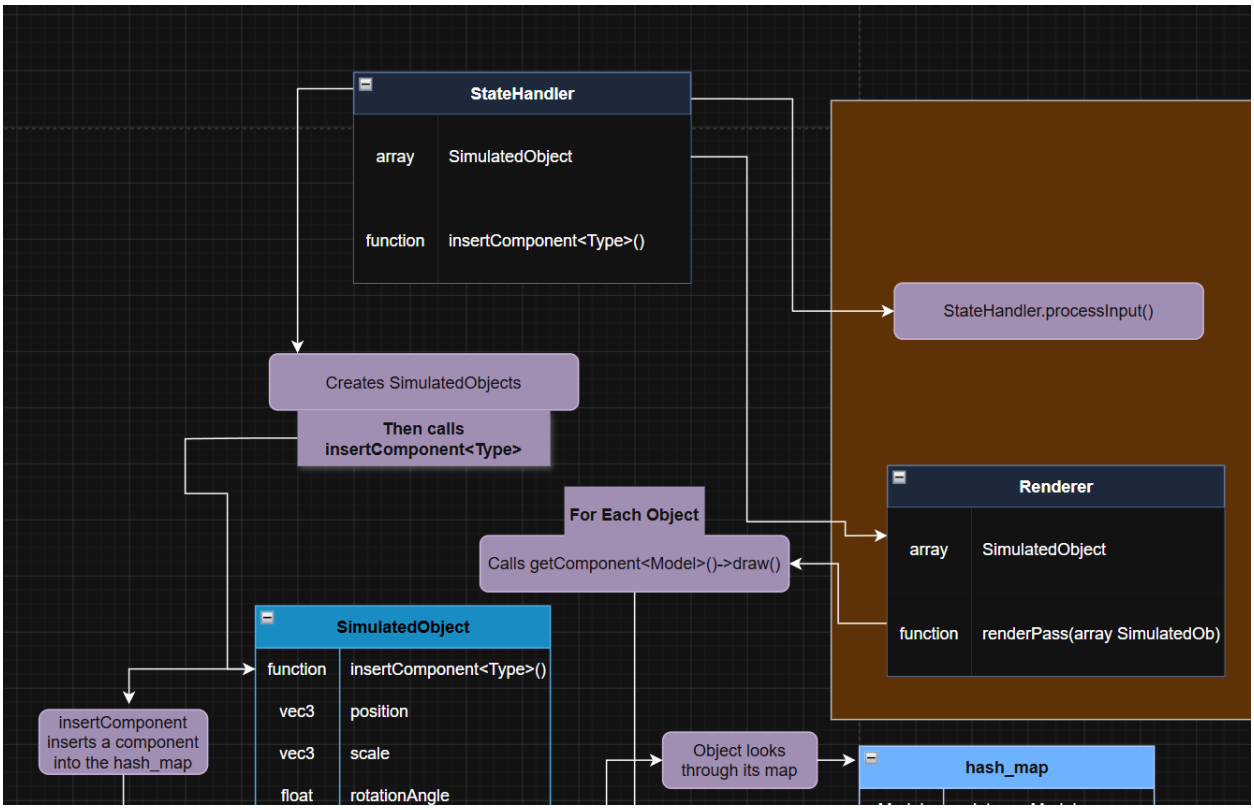

Figure 35, a highly simplified graph showing how the stateHandler's role

How does the SimulatedObject obtain components? The StateHandler, which as the name suggests, contains information about the current state, such as the objects in the scene, the light sources, the window size; all types of things that should be saved to a file. This StateHandler lives outside of the main process loop, but some of its functions, like the processInput, live inside the main loop. When any SimulatedObject is created, the StateHandler also immediately calls the insertComponent function of said object and gives it whatever component the user specified.

Camera + Camera Control

The Camera is an object that has its own local space, just as any object does. It uses the forward vector of its local space to determine where it's pointed, and this is where the camera view will be generated.

```
class Camera {
private:
    glm::vec3 cameraPos;
    glm::vec3 cameraTarget;
    glm::vec3 cameraRight;
    glm::vec3 cameraUp;
    glm::vec3 cameraDirection;
    glm::vec3 cameraFront;
    glm::mat4 cameraView;

    float yaw;
    float pitch;

public:
```

Figure 36, the camera class. The vectors store its local space

The vectors need to be stored in the camera class, because many complex calculations must be done for each update loop iteration for the view to work correctly. The GLM library is used to perform calculations on these vectors. The camera requires a camera position and a camera target for these calculations.

```
// Calculating new direction vector based on the angles of the yaw and pitch obtained by mouse movement
glm::vec3 newDirection;
newDirection.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
newDirection.y = sin(glm::radians(pitch));
newDirection.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

updateCameraState(getCameraPos(), newDirection);
```

Figure 37, camera look direction calculation based on mouse position

The details of these calculations are a little out of scope for this thesis; however, it's important to note that the mouse position handled by StateHandler has an influence on the view direction. The updateCameraState function makes sure that the camera's vectors are updated every time a change occurs in the camera's position.

```
mainCamera.setCameraView(glm::lookAt(mainCamera.getCameraPos(),
    mainCamera.getCameraPos() + mainCamera.getCameraDirection(),
    mainCamera.getCameraUp()));

if (stateHandler.cameraToggle) {
    mainCamera.cameraRotation(stateHandler.mouseData.relMouseX, stateHandler.mouseData.relMouseY);
}
```

Figure 38, in the update loop, the cameraView is set to the camera's look target, the position + direction, the camera's up vector

The camera's rotation function only activates if the middle mouse button is held down. It also takes in the mouse's x and y positions relative to the window. All of these calculations in the end update the camera view matrix, which is then passed to the vertex shader to get multiplied by the model and projection matrices, and then it influences each vertex.

GUI (Graphical User Interface)

Now that entities are represented using SimulatedObjects, displaying their properties on the screen is important for clarity. Creating a GUI from scratch is incredibly time consuming and can be very complex, so for this project the decision was made to use an external library for this task. This library is called ImGui, and it comes pre-packaged with code examples and debug tools.

```
int main(int argc, char* argv[]) {
    //Initializing state handler and state
    StateHandler stateHandler;

    SDL_Window* window = stateHandler.getWindow();
    SDL_GLContext context = stateHandler.getContext();

    Gui gui(stateHandler.main_scale, window, context);
```

Figure 39, the gui is instantiated at the beginning of the main process, along the statehandler and sdl window

```
Gui::Gui(float mainScale, SDL_Window* window, SDL_GLContext context) {
    IMGUI_CHECKVERSION();
    ImGui::CreateContext();
    io = &ImGui::GetIO(); (void)io;
    style = &ImGui::GetStyle();
    io->ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;     // Enable Keyboard Controls
    io->ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad;      // Enable Gamepad Controls
```

```
// Setup scaling
style->ScaleAllSizes(mainScale);
style->FontScaleDpi = mainScale;

// Setup Platform/Renderer backends
ImGui_ImplSDL3_InitForOpenGL(window, context);
ImGui_ImplOpenGL3_Init("#version 450 core");
```

Figure 40, ImGUI creates its own context, hooks into the input/output, and creates a style. It then implements the OpenGL SDL backend

```
while (running) {
    while (SDL_PollEvent(&event)) {
        //Listening for imGui events here
        ImGui_ImplSDL3_ProcessEvent(&event);
        // ---------------------------------
        switch (event.type) {
        case SDL_EVENT_QUIT:
            running = false;
            break;
```

Figure 41, ImGUI hooks into event listening in the main update loop

```
//passing IDs
stateHandler.selectedObjectId = renderer.clickedObjectId;
glBindVertexArray(0);
ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
SDL_GL_SwapWindow(window);
```

Figure 42, ImGUI hooks into the main update loop and draws itself using OpenGL

ImGui is given the SDL window and SDL_GL context, and the main scale of the program. It then listens to event calls and draws itself in the main update loop. In this project, the Gui class that houses ImGui has a method called renderUi. This method gives ImGui all the instructions required to render the UI.

```
//Rendering pass and main update loop
renderer.modelUniform(lightingShader, model);
renderer.viewUniform(lightingShader, mainCamera);
renderer.projectionUniform(lightingShader, projection);
gui.renderUi(stateHandler, lightingShader);
renderer.renderPass(myTexture, pickingTexture, lightingShader, stateHandler.getSimulatedObjects(), stateHandler);
```

Figure 43, note the renderUi function

The renderUi function currently handles all UI instructions, so it is a very large snippet of code. The most important fact to remember about this function is that it takes in a copy of the stateHandler, because it requires, among other things, the currently selected SimulatedObject id. This object selection process will be covered in the next section.

```
ImGui::Begin("Object Information");

ImGui::Checkbox("Demo Window", &show_demo_window);
ImGui::Checkbox("Object Spawning Window", &show_another_window);

ImGui::Text("Light Position");
ImGui::PushID(1);
```

Figure 44, snippet of code inside the renderUi function. Note the ImGui functions

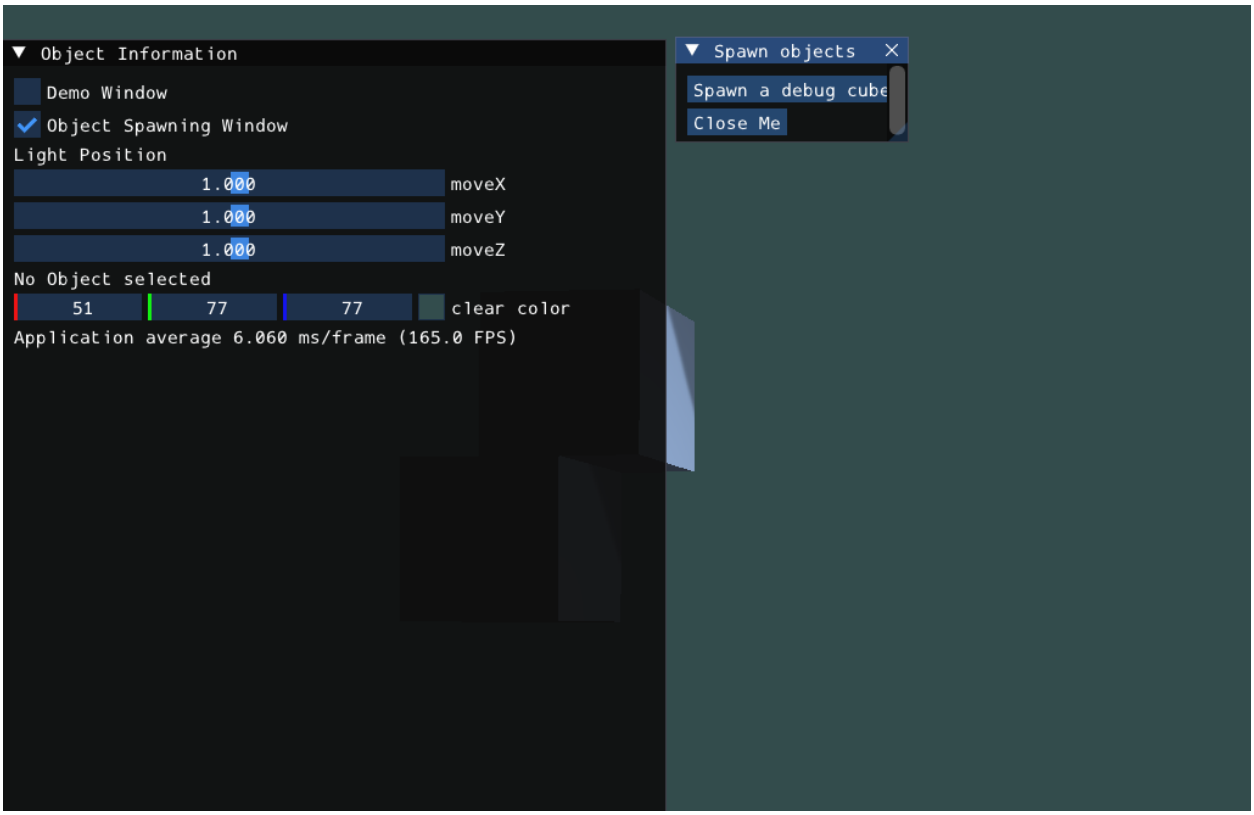

Figure 45, imGui UI

Object Selection and Importing

The first concept to cover for this implementation is object selection. Objects in this project should be clickable, so that the UI can display their properties when the user clicks on them. The methodology to establish this property of rendered objects is quite complicated, so the explanation here will be simplified for brevity.

```
//Picking texture solution credited to OGLDEV's tutorials https://www.youtube.com/watch?v=71G-PVpaVk8&t=31s
class Picking {
private:

    GLuint pickingTexture = 0;
    GLuint depthTexture = 0;
public:
    GLuint bufferObject = 0;
    Picking(unsigned int windowWidth, unsigned int windowHeight);

    struct PixelInfo {
        unsigned int objectID = 0;
        unsigned int drawID = 0;
        unsigned int primitiveID = 0;

        void objectDebug() {
            printf("Object id: %d, draw id: %d, primitive id %d", objectID, drawID, primitiveID);
        }
    };
```

Figure 46, Picking texture class

The technique that will be used for object selection is called object picking. The core of this technique is the generation of what is called a “picking” texture. This is a color texture, which means that each pixel in the texture is assigned an RGB value, which is similar to how the

final output texture works for regular rendering. The difference here is that the actual values reference object and mesh data. R stores the objectID, G stores the drawID, and B stores the primitive (triangle) ID. This means that in this texture, each pixel stores important object information about the object it's representing.

```
// Frame buffer object generated
glGenFramebuffers(1, &bufferObject);
glBindFramebuffer(GL_FRAMEBUFFER, bufferObject);

// Texture object generated for the framebuffer
glGenTextures(1, &pickingTexture);
glBindTexture(GL_TEXTURE_2D, pickingTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32UI, windowWidth, windowHeight, 0, GL_RGB_INTEGER, GL_UNSIGNED_INT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, pickingTexture, 0);
```

Figure 47, generating picking texture, and a new framebuffer to output it to

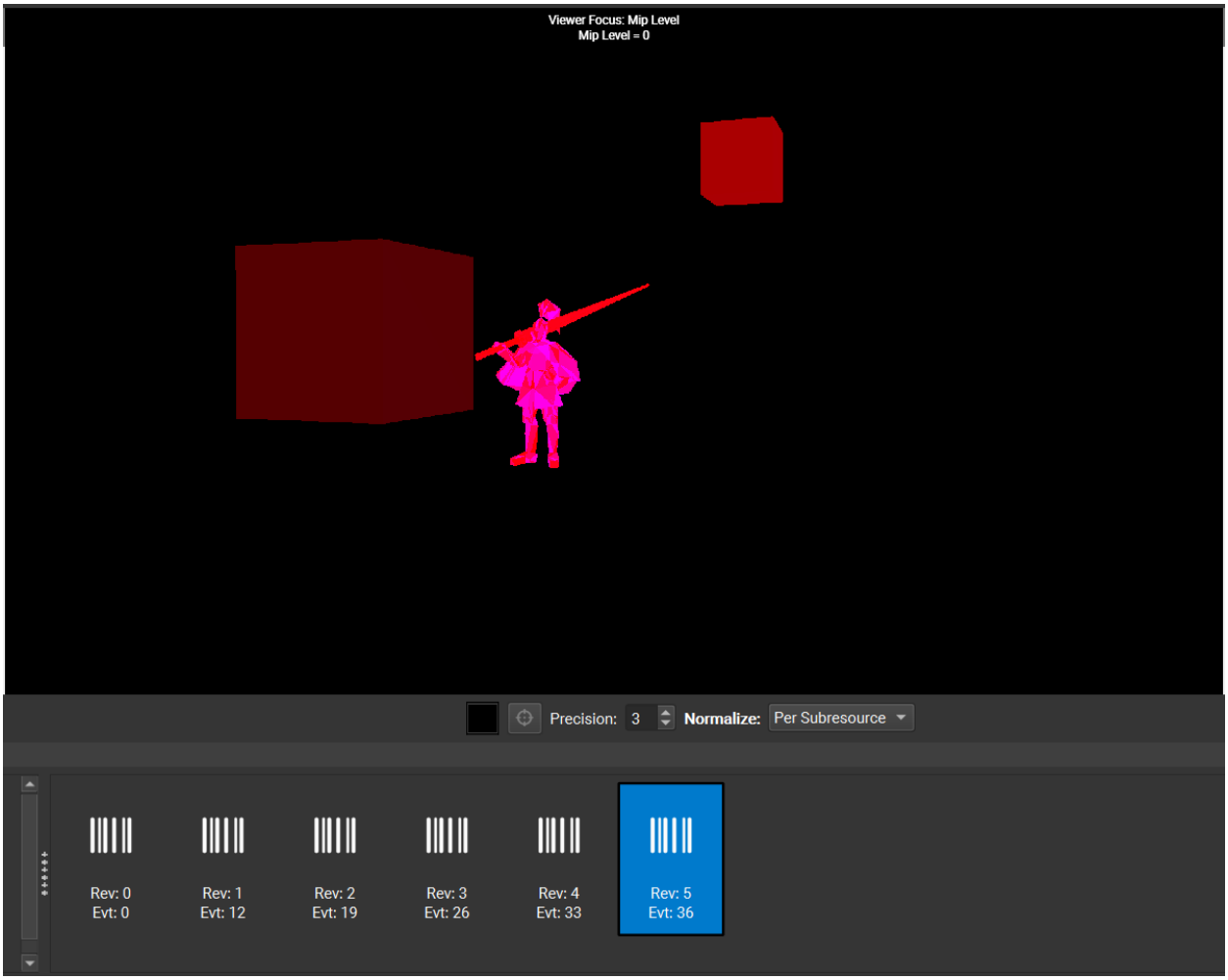

Figure 48, picking texture rendered to its framebuffer. Note how each object is a different shade of red.

Usually, the color buffer becomes part of the frame buffer than is rendered to screen. In this case, a new framebuffer is created which takes in the picking texture and the depth texture. This new framebuffer is not displayed on the screen, but it's used for the next step.

```
Picking::PixelInfo Picking::readPixel(unsigned int x, unsigned int y) {
    glBindFramebuffer(GL_READ_FRAMEBUFFER, bufferObject);
    glReadBuffer(GL_COLOR_ATTACHMENT0);
    //Return pixel color
    PixelInfo pixel;
    glReadPixels(x, y, 1, 1, GL_RGB_INTEGER, GL_UNSIGNED_INT, &pixel);

    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_READ_FRAMEBUFFER, 0);

    return pixel;
}
```

Figure 49, function for reading pixel data

The picking class readPixel function uses this buffer to read pixels and save them to the pixel data. This returns a PixelInfo object that can be read to determine the object's ID on that pixel.

```
void Renderer::pickingPass(Picking& pickingTexture, std::vector<SimulatedObject> simulatedObjects, Shader shader) {
    pickingTexture.enableWriting();
    //Clearing framebuffer to 0 so that the background doesn't crash the picking logic
    glClearColor(0, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindVertexArray(VAO);
    for (unsigned int i = 0; i < simulatedObjects.size(); i++)
    {
        //Giving the fragment shader the object ID for each object
        glm::mat4 model = glm::mat4(1.0f);
        model = glm::translate(model, simulatedObjects[i].position);
        model = glm::scale(model, simulatedObjects[i].scale);
        float angle = i * SDL_GetTicks() / 100;
```

Figure 50, pickingPass function called every update loop iteration by the renderer

```
        int objectIndexLocation = glGetUniformLocation(shader.ID, "gObjectIndex");
        glUniform1ui(objectIndexLocation, i + 1);
        // Passing the view projection model matrix to the picking shader
        if (simulatedObjects[i].getComponent<Model>()) {
            simulatedObjects[i].getComponent<Model>()->draw(shader);
        }
    }
    pickingTexture.disableWriting();
```

Figure 51, pickingPass function called every update loop iteration by the renderer

In the update loop, the picking pass enables the writing to the picking frame buffer, clears it, binds vertex array and then each simulated object also draws their models to this new frame buffer. Essentially, all rendered meshes must be rendered to the regular framebuffer, and the picking framebuffer, for this technique to work.

```
if (stateHandler.mouseData.leftClickPressed && stateHandler.mouseData.isHoveringOverUI == false) {
    Picking::PixelInfo pixel = pickingTexture.readPixel(stateHandler.mouseData.mouseX, stateHandler.getResolution().y - stateHandler.mouseData.mouseY - 1);
    /*pixel.objectDebug();*/
    if (pixel.objectID != 0 && objectSelected == false) {
        clickedObjectId = pixel.objectID - 1;
        assert(clickedObjectId < simulatedObjects.size());

    }
    if (pixel.objectID <= 0) {
        std::cout << objectSelected << std::endl;
        clickedObjectId = -1;
    }
}
for (unsigned int i = 0; i < simulatedObjects.size(); i++)
```

Figure 52, reading pixel data and saving the clickedObjectId so that it can be sent to the GUI (credit to OGLDEV for the readPixel algorithm https://www.youtube.com/watch?v=71G-PVpaVk8&t=31s)

Finally, in the render loop, the pixel data of the picking texture is read and passed to the clickedObjectId, which is then stored to the stateHandler so it can be used in the GUI.

```
std::cout << stateHandler.getSimulatedObjects().size() << std::endl;
float xPosNew = stateHandler.getSimulatedObjects()[objectToDisplay].position.x;
float yPosNew = stateHandler.getSimulatedObjects()[objectToDisplay].position.y;
float zPosNew = stateHandler.getSimulatedObjects()[objectToDisplay].position.z;
float& xPosActual = stateHandler.getSimulatedObjects()[objectToDisplay].position.x;
float& yPosActual = stateHandler.getSimulatedObjects()[objectToDisplay].position.y;
float& zPosActual = stateHandler.getSimulatedObjects()[objectToDisplay].position.z;
std::unordered_map<std::type_index, void*> objectComponents = *stateHandler.getSimulatedObjects()[objectToDisplay].getComponents();

ImGui::SliderFloat("moveX", &xPosNew, -20.0f, 20.0f);
ImGui::SliderFloat("moveY", &yPosNew, -20.0f, 20.0f);
ImGui::SliderFloat("moveZ", &zPosNew, -20.0f, 20.0f);
```

Figure 53, the GUI gets the exact object that's selected by querying the stateHandler's selectedObjectID

Lighting

The lighting is added in the fragment shader using the diffuse lighting method, which grabs the light source's position vector and the mesh's normal vector, finds the difference in angle, and brightens the fragment based on that angle.

```
    int objectColorLocation = glGetUniformLocation(shader.ID, "objectColor");
    glUniform3fv(objectColorLocation, 1, &glm::vec3(1.0f, 0.5f, 0.31f)[0]);

    int lightColorLocation = glGetUniformLocation(shader.ID, "lightColor");
    glUniform3fv(lightColorLocation, 1, &glm::vec3(1.0f, 1.0f, 1.0f)[0]);

    int lightPositionLocation = glGetUniformLocation(shader.ID, "lightPosition");
    glUniform3f(lightPositionLocation, stateHandler.light.position.x, stateHandler.light.position.y, stateHandler.light.position.z);
```

Figure 54, data related to diffuse lighting passed to the lighting shader

```
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    vec3 normal = normalize(Normal);
    vec3 lightDirection = normalize(lightPosition - FragPos);

    float difference = max(dot(normal, lightDirection), 0.0);
    vec3 diffuse = difference * lightColor;

    vec3 result = (ambient + diffuse) * objectColor;
    FragColor = texture(ourTexture, TexCoord) * vec4(result, 1.0);
}
```

Figure 55, shader lighting calculation

The light direction is a normalized difference between the light position vector and the fragment position vector. The dot product between the the face's normal and light direction is then multiplied by the light color, and then the final object color is multiplied by that. The color output then includes the texture and the lighting.

The object importing is handled by the Assimp library. The model files are scanned and parsed into the project's mesh vertices. The textures are also imported and put into the project's mesh texture struct.

Testing and Evaluation

Debugging

Debugging a program that makes use of asynchronous communication between the CPU and GPU is difficult, as there are multiple points of failure that can occur independant of each other, which creates a confusion as to where the core of the bug lies. Visual Studio has a good, robust debugging environment in its local Windows debugger mode. There is a window that specifically lists diagnostic tools, and the general performance of the software.

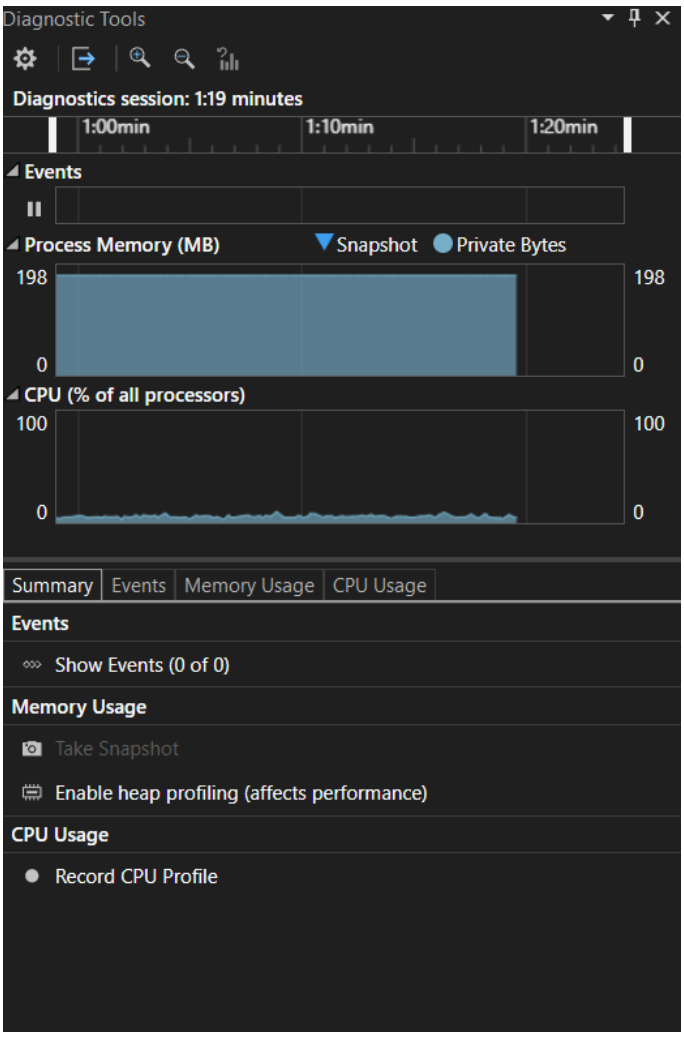

Figure 56, diagnostic tools. Note the process memory readout and CPU %.

This tool will also list the call stack and a scoped variable readout if a breakpoint is inserted into the code.

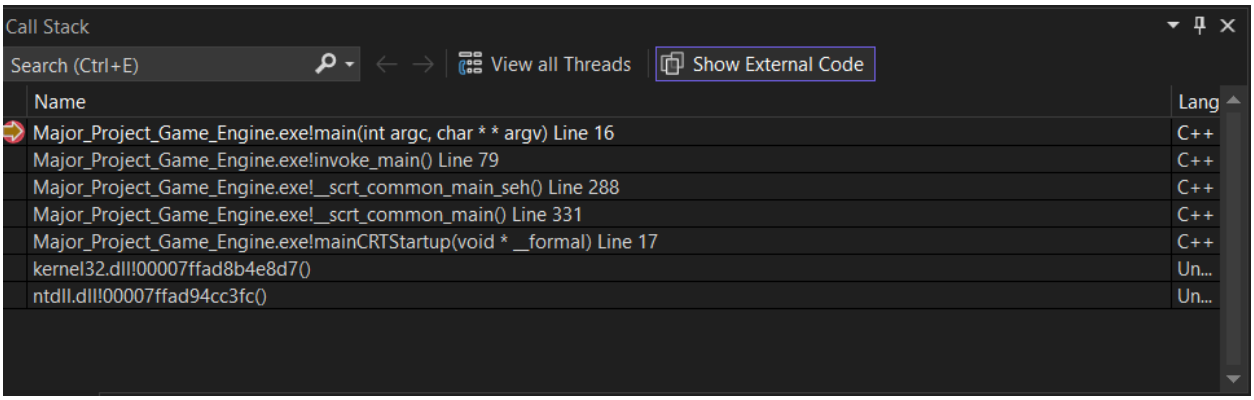

Figure 57, call stack, displaying the calls that lead to the break in the code.

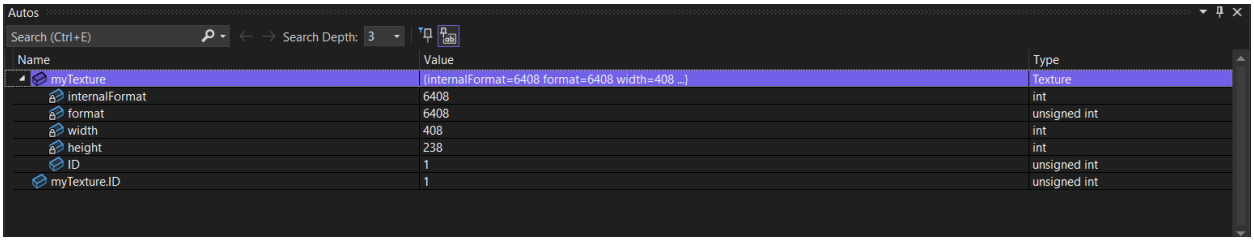

Figure 58, variable readout

These tools, combined with printing out to the console, handle the CPU side of debugging. Using this, it's more manageable to get to the bottom of typos, coding bugs and so on. The C++ compiler will also complain in the output console when there is a syntax error or logic error. If the error isn't caught immediately in the IDE, the compiler will stop code compiling if the program runs while a compiler error is present. It will also show any linking errors. All the above-mentioned errors and bugs usually leave an error code, which can be searched on online forums such as Stack Overflow.

The more complex type of debugging is GPU-side debugging. The project simply couldn't move forward if GPU code errors were not visualized or broken down into more manageable pieces. To do this, the development environment uses Nvidia Nsight debugging tools. The most useful tool for the scope of this project is the Nvidia Nsight Graphics Frame Debugger. This debugger hooks into the local application, and then the user can pause execution and analyze what happened in a frame of rendering. With this, it's possible to step through GPU API calls and visualize the rendering step by step. The frame debugger includes a trace of all the events (GPU calls), event details, API inspector, and an output visualization for rendering. It also contains a window which can be scrubbed through to capture each detail of rendering.

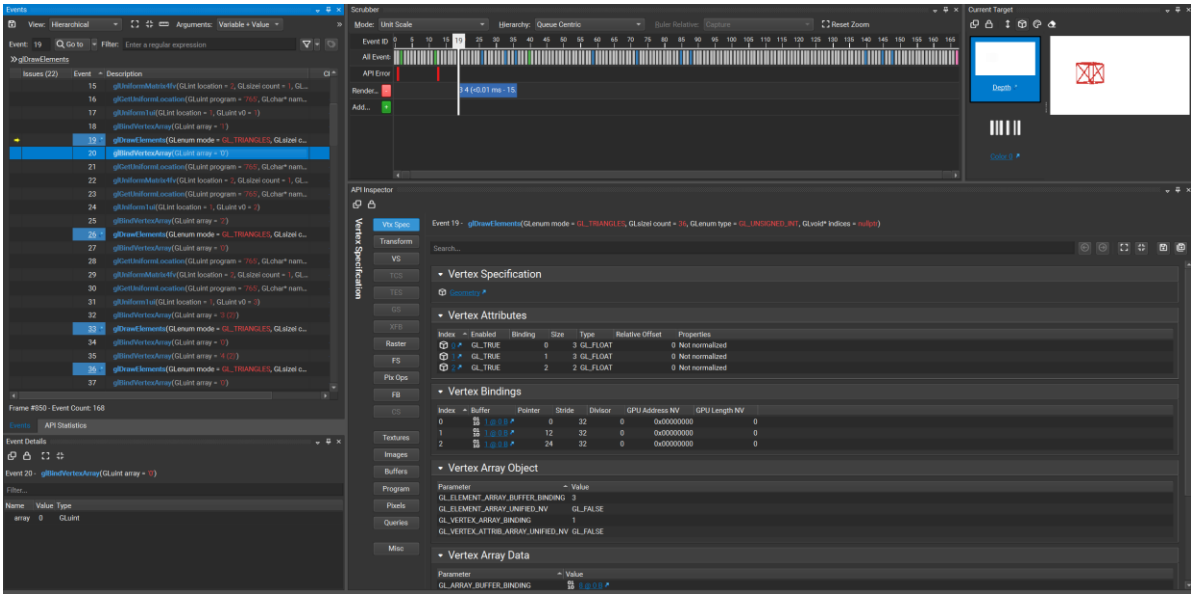

Figure 59, entire Graphics Frame Debugger window.

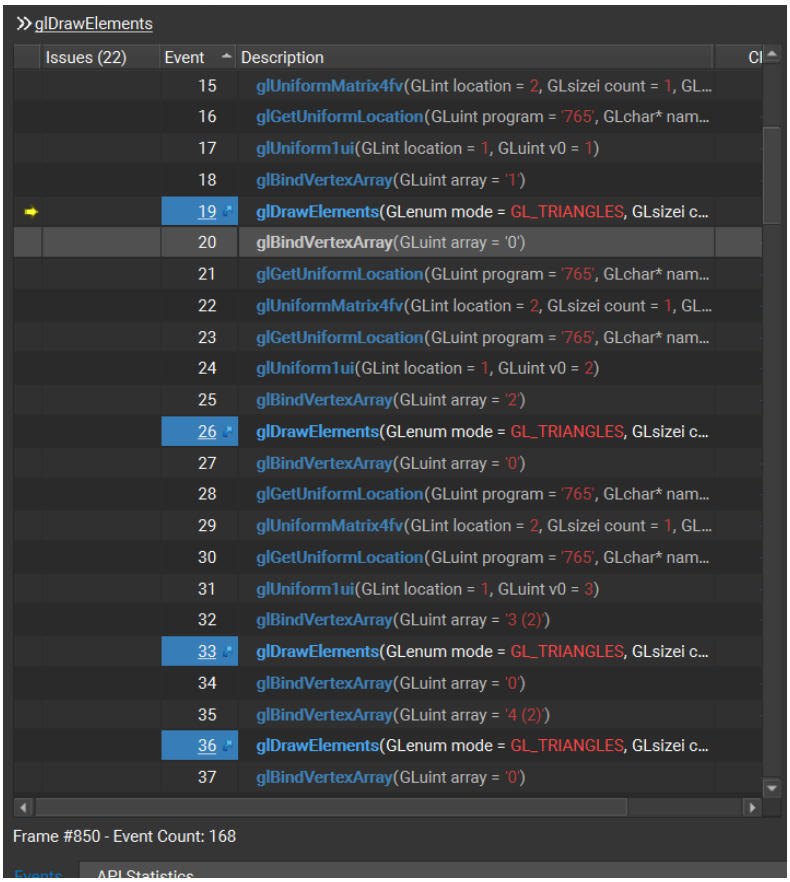

Figure 60, event tracker, note the OpenGL API calls

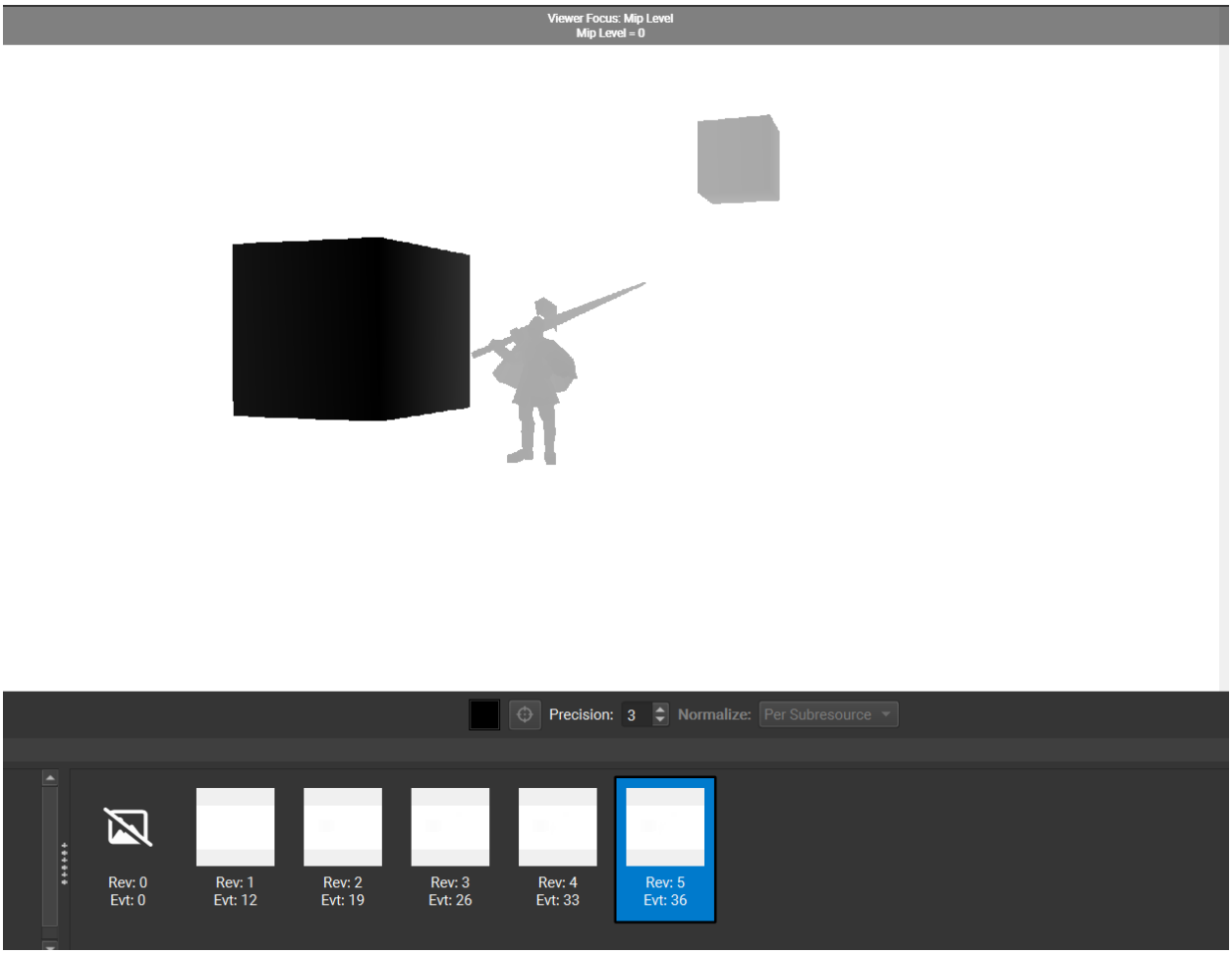

Figure 61, frames showing the rendering of the depth buffer

As can be seen from the above figure, the depth buffer texture stores how far away each object is. This is an important debugging tool, as I can see if objects' depths aren't being properly reflected in the texture. This could indicate a core issue with the depth buffer shader and how the texture is drawn. This did happen in the process of creating the object picking feature; the objects were not correctly drawn in the depth buffer, while being drawn perfectly in the regular rendering output. This caused a strange bug where none of the picking code was working even though everything looked fine on the surface. This is the bug that necessitated the installation of GPU debugging software, as it was simply invisible to the CPU side, and could not be visualized intuitively. After using Nvidia Graphics Frame Debugger to render the depth buffer in isolation, it was found that the objects' vertices were misplaced. The bug was easily found and remedied. This couldn't have been fixed without proper GPU debugging.

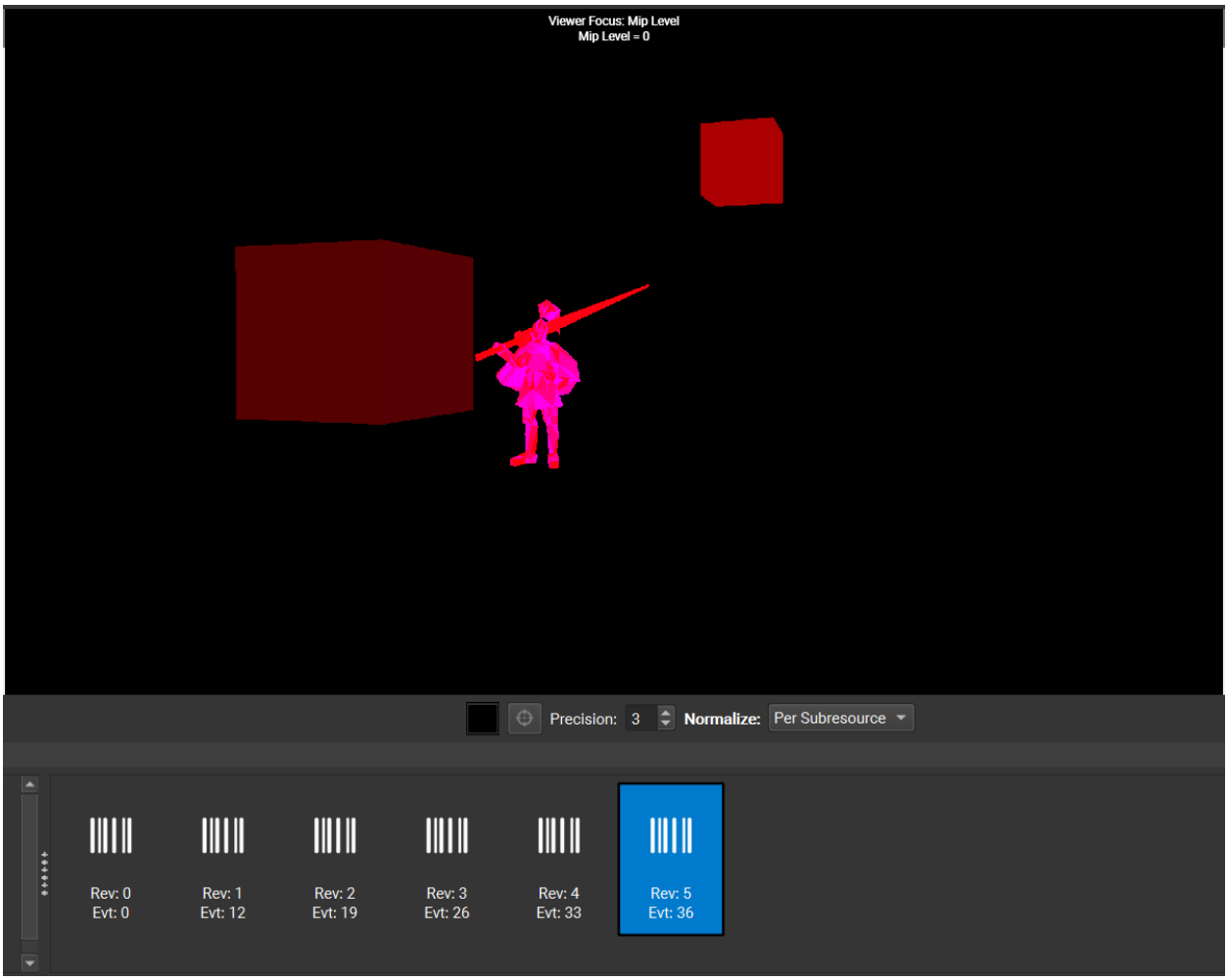

Figure 62, frames showing the progression of the picking texture color buffer, with R being object ID and B being object primitive ID

The object picking feature required rendering to a color buffer, with R being the ObjectID and B being the primitive ID on that object. This could also be visualized in the frame debugger, which allowed for quick confirmation of code validity, since the errors are very visual in this readout. If, for example, the background had an incorrect ObjectID, it would appear red or white, but it appears black here because it's rendering the correct ID. Note how the actual objects in the scene look red or purple.

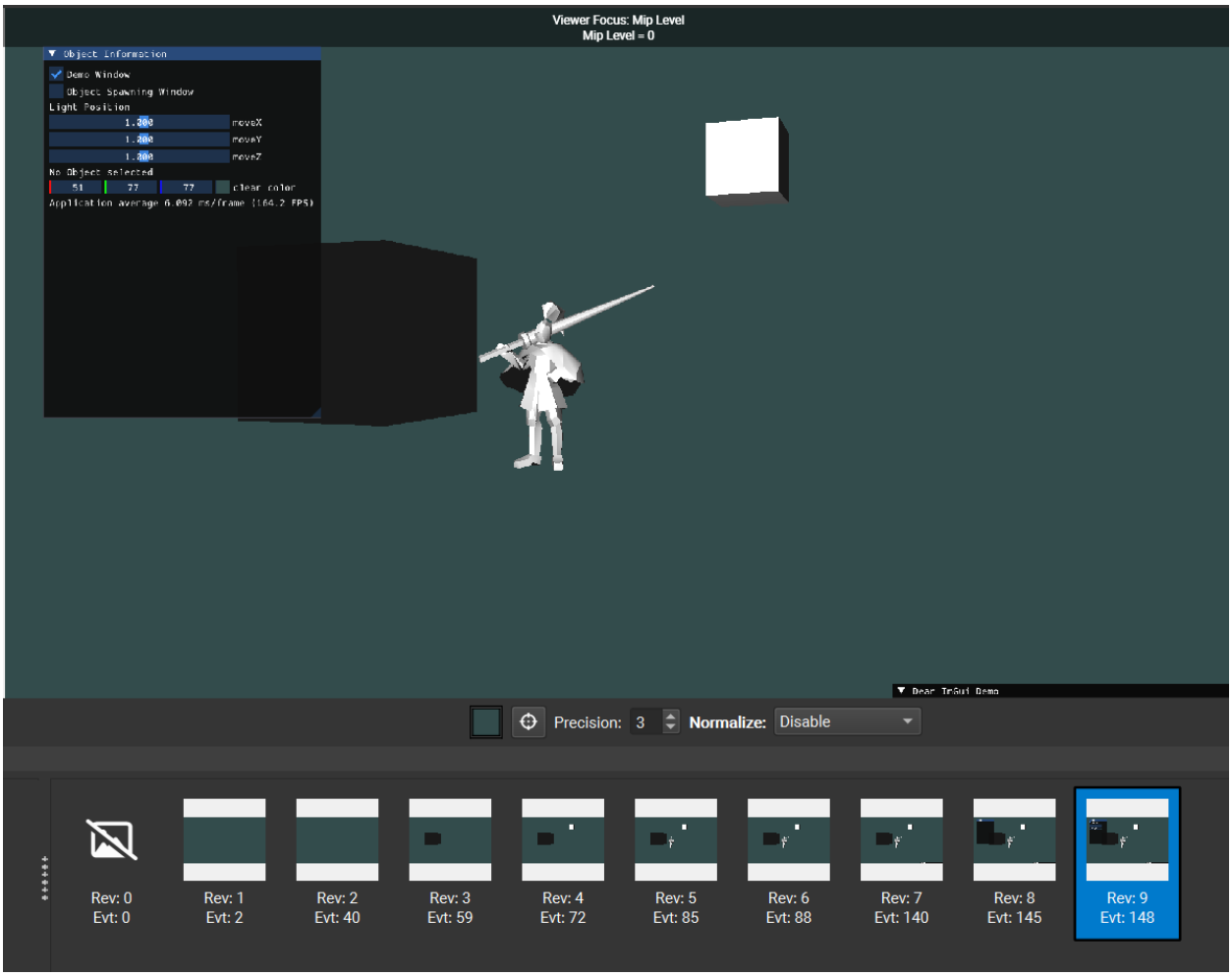

Figure 63, main frame buffer that will be displayed to the screen. Note the individual steps taken in the frames below the main image.

Performance Testing

Performance Testing the 3D engine, just like debugging, was done in two steps; the CPU performance test, and the GPU performance test. Testing the performance on the CPU involved using Visual Studio's debugging tools to show the memory and CPU usage. Generally, the data that was relevant to CPU performance was how the memory acted during prolonged usage of the program. If the memory usage keeps climbing through regular use, that would indicate a memory leak. An important metric for this project in particular is that adding more buffers to the rendering pipeline increased the memory usage drastically.

Another avenue for CPU testing was the debug cube spawning. If the project can handle hundreds or thousands of debug cubes without dropping frames, it would be safe to say the optimization is working. Importing complex objects would also test the program's efficiency, since there would be many primitives on screen.

The GPU side of performance testing was performed using the frame debugger and the GPU trace tool. The GPU trace tool could capture frames and display how long certain actions took to execute in the timeline view. The summary view shows the important facts about the performance of the GPU, such as VRAM throughput.

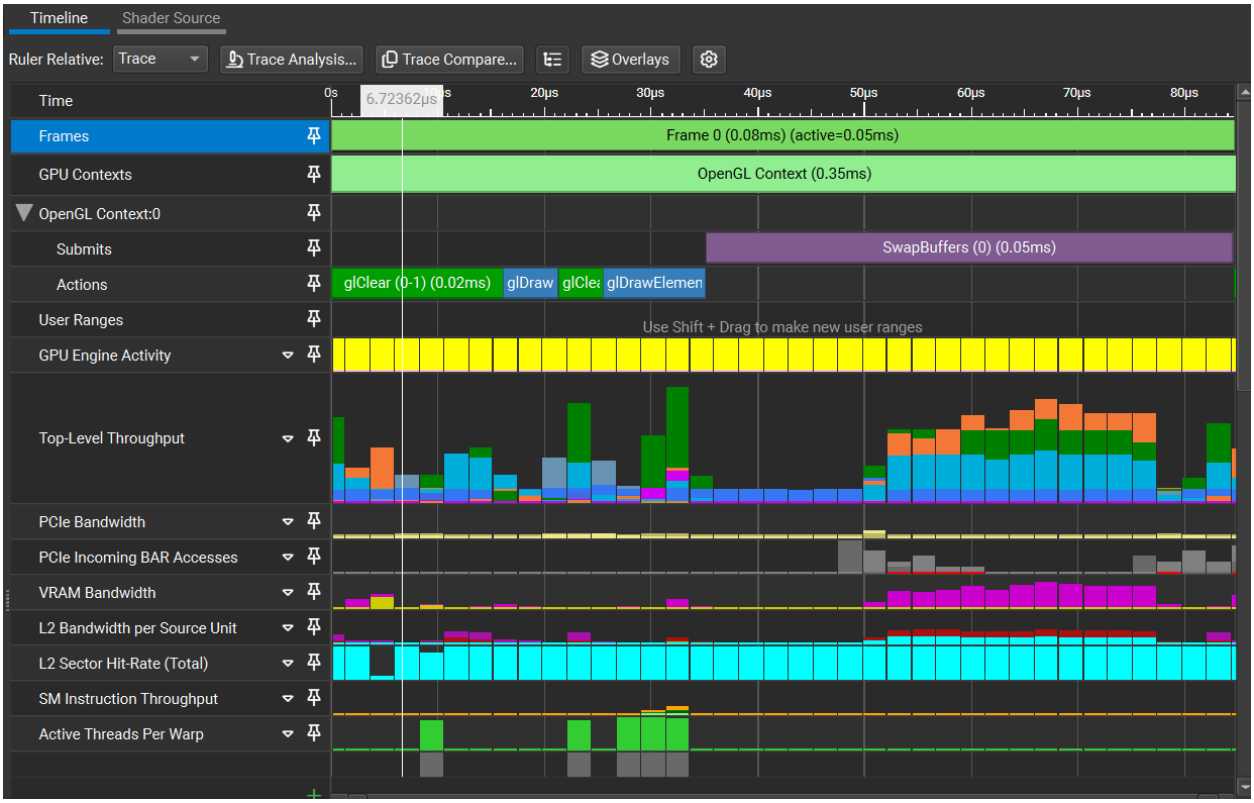

Figure 64, timeline view that displays important actions executed on the GPU during the frame

The timeline view can display which actions took the most time to resolve, or where the program is bottlenecked.

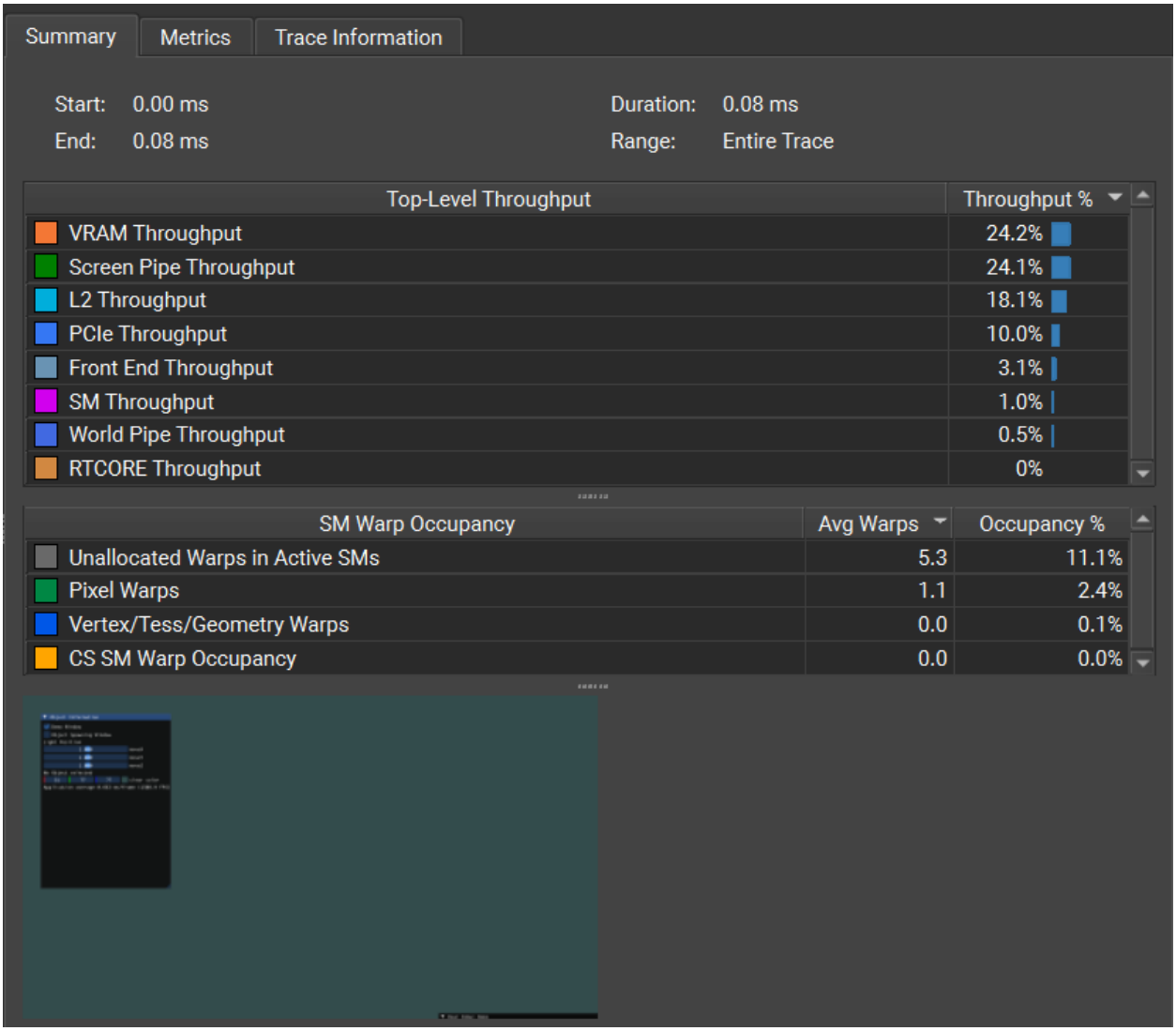

Figure 65, summary of performance during the captured frame.

The summary window is important for performance testing, as bottlenecks can be clearly seen. The project is using 24% VRAM throughput right now, which is low; however, a 100% VRAM throughput would indicate a bottleneck.

Test Plan and Test Report

This section contains two segments; the overall user testing plan, and the user test reports. The project had 3 testers, with each tester using their own personal computer to run the 3D engine. These testers also had the GPU trace tracker installed on their system, so that the session could be evalutated after the testing has ended.

The test plan is detailed below, and the testing results are described after the plan.

Test Plan

1. Testers and Duration
   - 1 participant per test
   - 3 minutes allocated

2. Test Setup
   - One facilitator for greeting, instruction, notetaking and test overseeing
   - Quick introduction to test context (Controls, 3D engine description etc.)
   - Facilitator runs debug tools

3. Test Procedure
   - Welcome and briefing
   - Participant is asked to do attempt a series of tasks (move the camera, spawn a cube, move the cube, move the light source, change background color, close and open gui windows)
   - Debrief and informal questions
     - What felt good to use?
     - Was the experience intuitive?
     - What felt awkward to use?
     - What would you want to see in the next build?

4. Data Capture
   - Facilitator captures key reaction
   - Facilitator takes a screenshot of the debug screen

5. Test Outcome
   - Main positives and negatives
   - Data analysis

- Improvement suggestions
- Short summary for next build

Test Report 1

1. **Key Reactions:**

- Participant was disoriented at the start, since the camera points at nothing.
- Participant wanted to increase camera speed
- Participant thought the object selection was intuitive but wanted to be able to move objects using the mouse and not the UI

2. **Positives and Negatives based on informal questions**

Positives

- The camera controls were intuitive
- The lighting looks good
- The tasks were easy to complete

Negatives

- No variable camera speed
- Camera speed tied to framerate
- Objects spawn randomly instead of in a set area

3. **Session Performance Snippet**

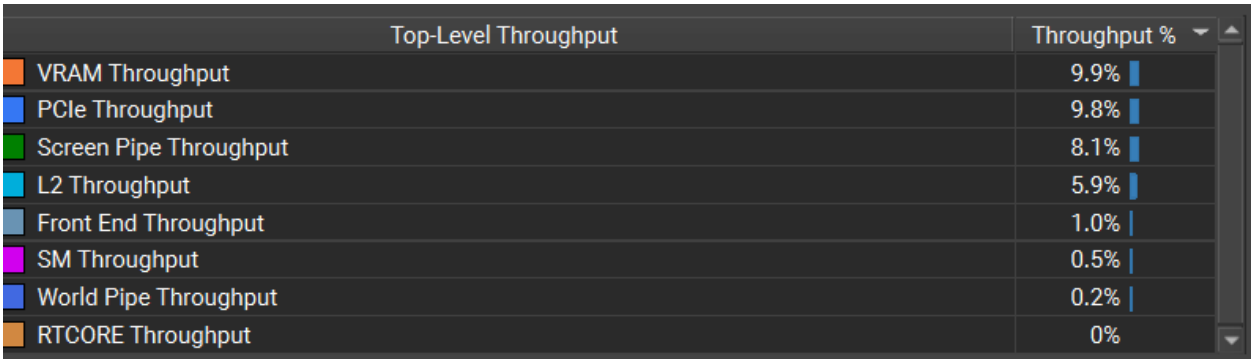

Figure 66, 30 millisecond performance snippet, taken at the end of the test

Test Report 2

1. **Key Reactions:**

- Participant had trouble moving the objects using the UI sliders
- Participant wanted to scale objects
- Participant had trouble finding the light source

2. **Positives and Negatives based on informal questions**

Positives

- Good camera controls
- Participant was able to build a small house using the debug cubes

Negatives

- No object scaling
- Light source hard to find

3. **Session Performance Snippet**

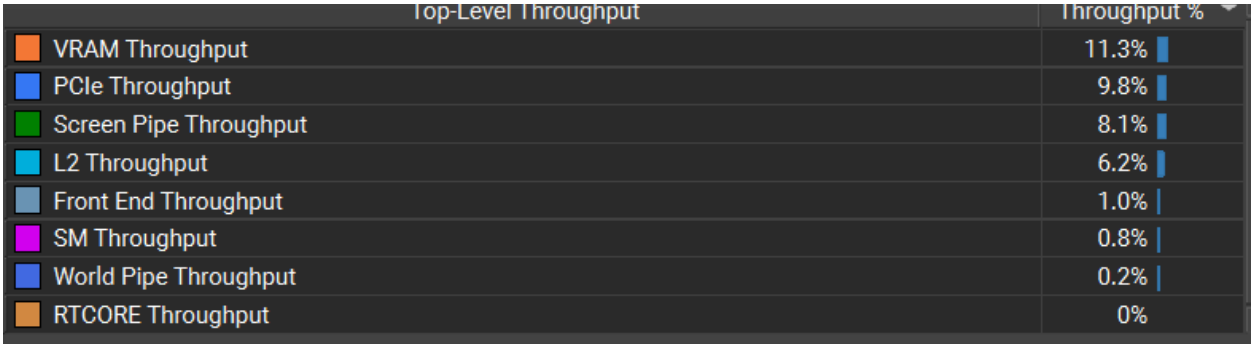

Figure 67, 30 millisecond performance snippet, taken at the end of the test

Test Report 3

1. Key Reactions

- Participant liked being able to move the light source around
- Participant thought the UI was too small

2. Positives and Negatives based on informal questions

Positives

- Being able to change background color
- Moving and resizing the UI is a good feature

Negatives

- No object scaling
- No object rotating
- Wanted to be able to import objects at runtime

3. Session Performance Snippet

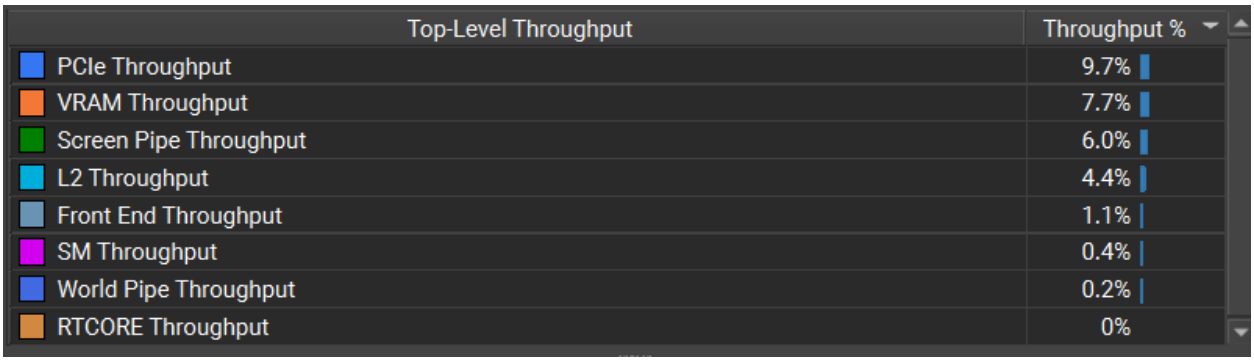

Figure 68, 30 millisecond performance snippet, taken at the end of the test

Final System Evaluation

The 3D engine can import custom objects, has a GUI, has lighting, has object movement, has object spawning, and has custom shader support. The final system, when compared to the original scope of the project (that being a simple game engine) is obviously not up to par, but the 3D engine created instead is a robust, functional piece of 3D software that is ready to receive further updates and functionality.

All of the objectives have been met except for the final objective, that being Comprehensive Unit and Feature testing.

Project Management

Methodology

The methodology used for this project is the Agile Scrum methodology. The advantage of this methodology is that the workload is separated into manageable segments called “sprints”. Each of these segments can have one goal, or a specific set of goals that all build towards the final project (Drumond, 2024).

The project was split into 4 phases, with each of the first 3 phases having their own sprint. These sprints were then created on the GitHub repository as milestones, and the sub-goals were created as issues assigned to each sprint. The 4th and final phase was reserved for any outstanding coding issues and thesis writing in the final month of the project.

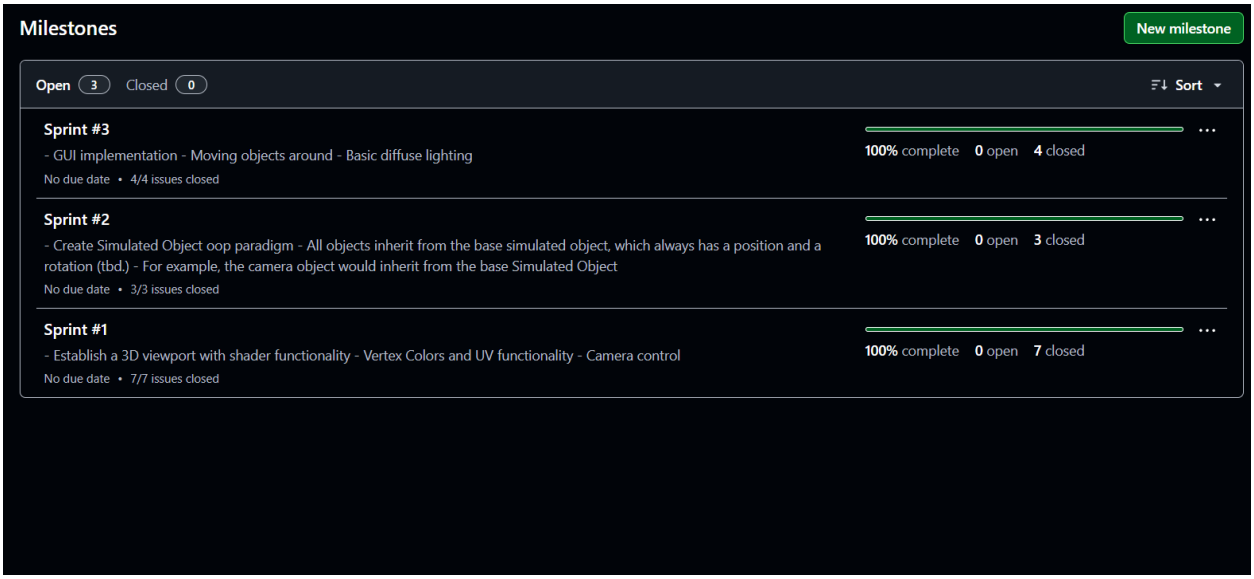

Figure 69, milestone sprint setup

Originally, these sprints were planned to include goals required to create a 3D game engine, however the first sprint (which was focused fully on 3D rendering) was so difficult that the planned sprints were re-organized to only include goals required for a 3D model viewer.

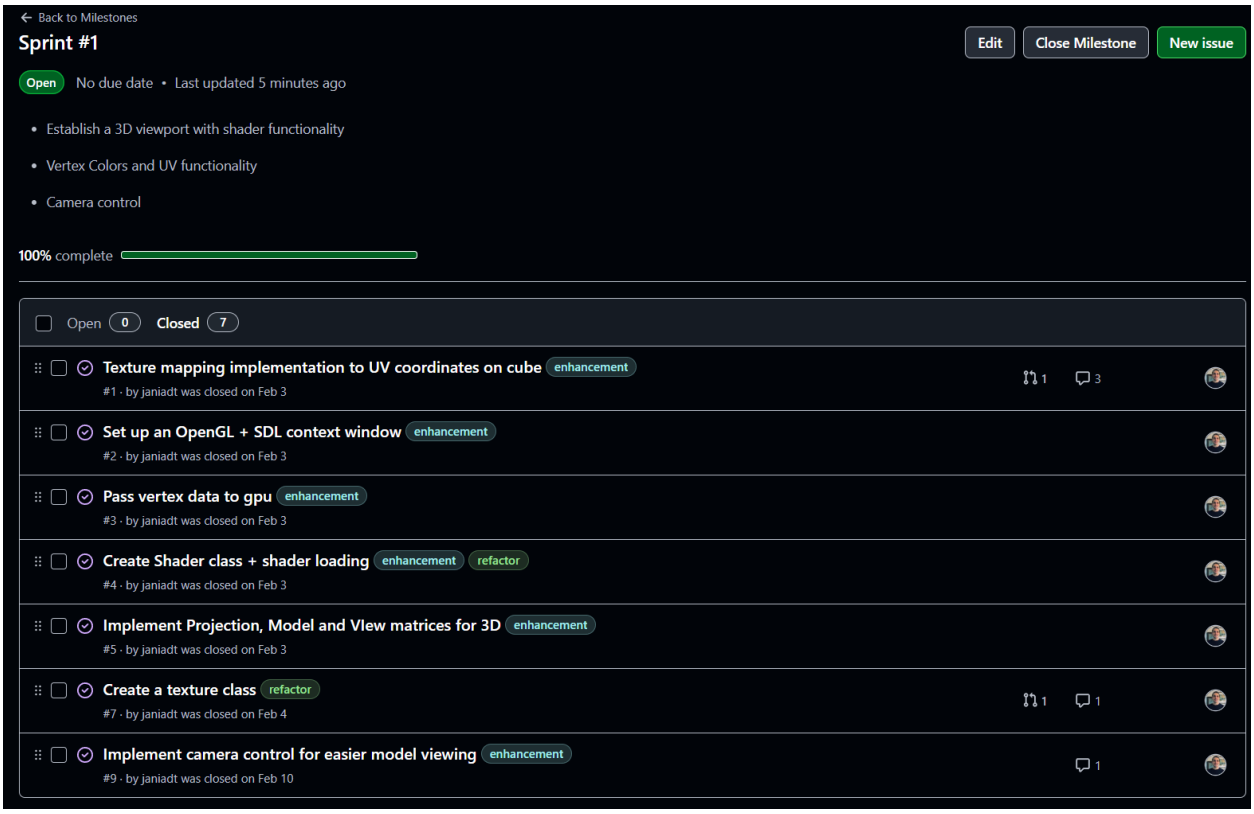

Figure 70, sprint issue setup

The above figure displays the sprint layout on GitHub. Note the sprint goals and the closed issues, with their issue type and description. Each of these issues had their own goals and descriptions, and some of them were even documented using comments.

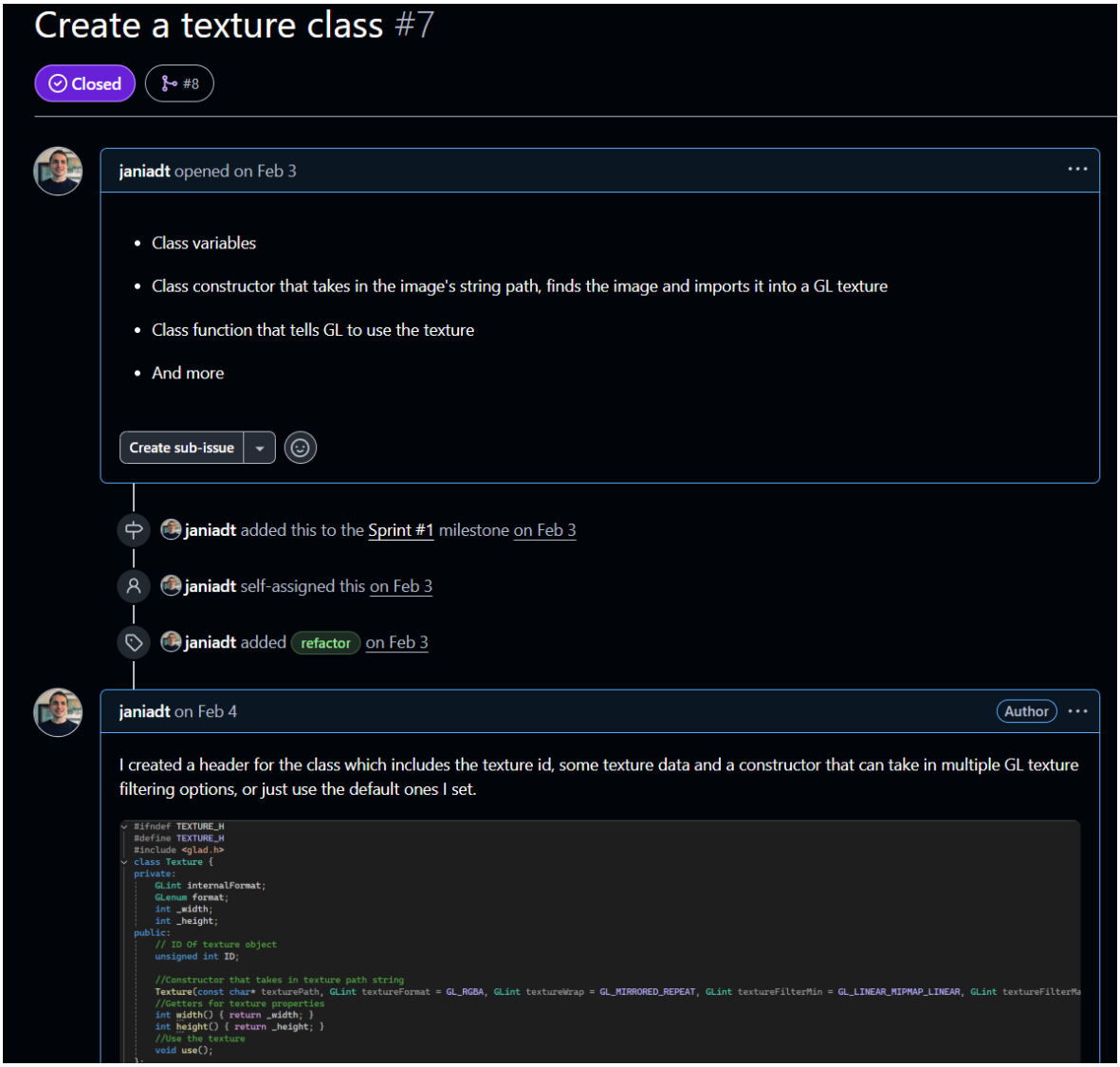

Figure 71, issue commit setup + feature branch

The final thing to mention is how GitHub feature branches were implemented to work directly with the issue system. A feature branch is a branch created from the main branch, which is used for further development of the project. This feature branch can have multiple commits and is usually merged into the main project once the feature has been implemented. Every new piece of functionality was created on a feature branch, which allowed issues to be closed automatically, as issues can be closed directly from a branch. This allowed a system where an issue would have a corresponding branch, and vice versa, which created an efficient project management system. These branches can be retroactively viewed and even pulled, so the project is never in jeopardy.

Phase Description

Sprint 1

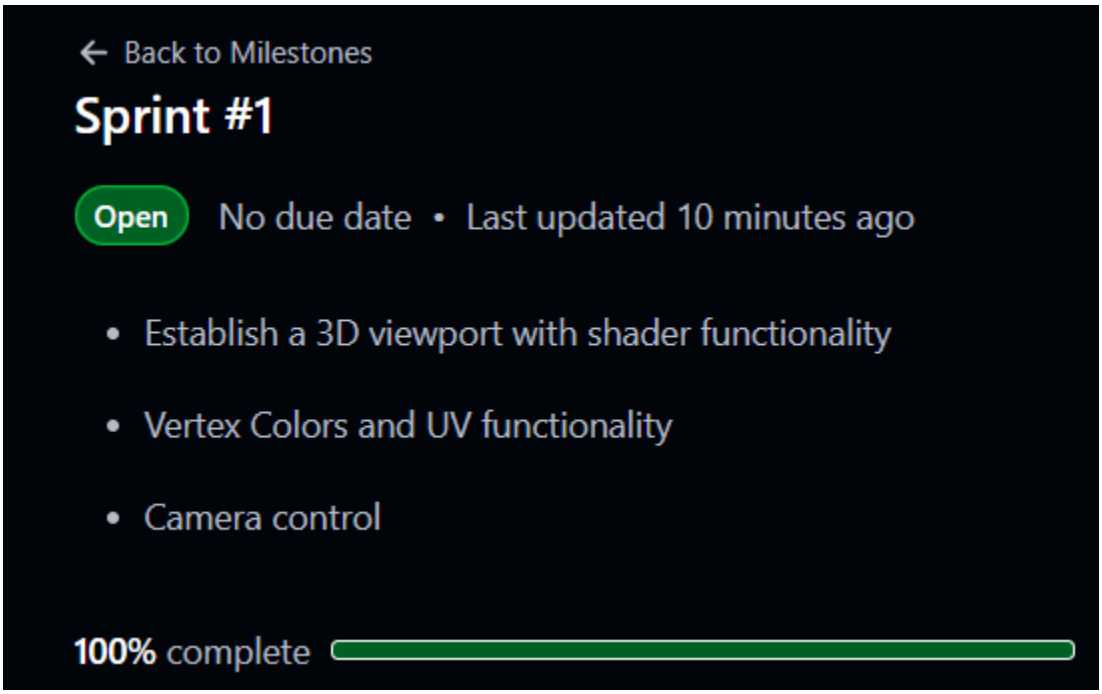

Figure 72, sprint 1

Sprint 1 was focused on establishing a 3D viewport/shader, vertex colors, uv and camera control. The base code for a lot of the functionality of the project was written at this time, so it was important to take care and write code with reusability in mind. This sprint was highly complicated, and it involved most of the steps required for 3D rendering basics. This meant that a lot of learning and research was required to develop these features. This is also where the focus of the project became clear, and the sights were set on developing a 3D model viewer.

The final outcomes, in this case, matched the initial goals. All the goals were met, and somewhat exceeeded, as this sprint also included texture implementation.

Sprint 2

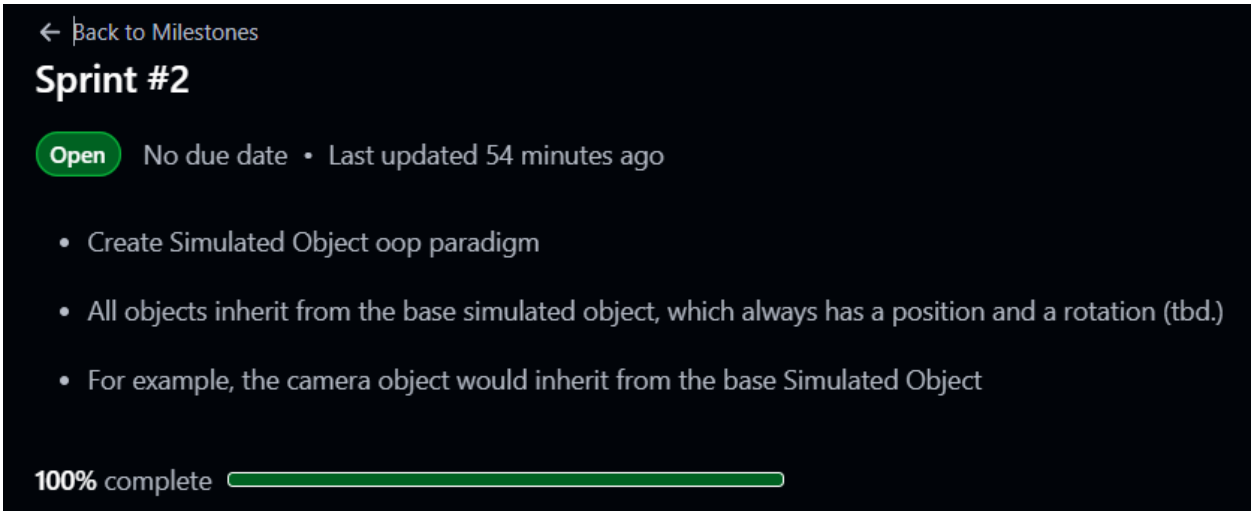

Figure 73, sprint 2

Sprint 2 was dedicated to implementing the OOP object system into the engine. This also included many code refactors to better fit this new paradigm. This was a very time-consuming sprint, as rewriting OpenGL code can be very complicated, considering the debugging overhead.

The initial goals of the sprint were reached in time, with some effort. The extra goal was to refactor the camera to use mouse + keyboard movement. Before this, the camera simply rotated around the model automatically. Another extra goal included adding the mesh component to the simulated objects, which would hold vertex array data.

Most of the time in this phase was spent on creating a way for objects to have multiple components of different types, so that each object can be given different kinds of functionality based on their components.

Sprint 3

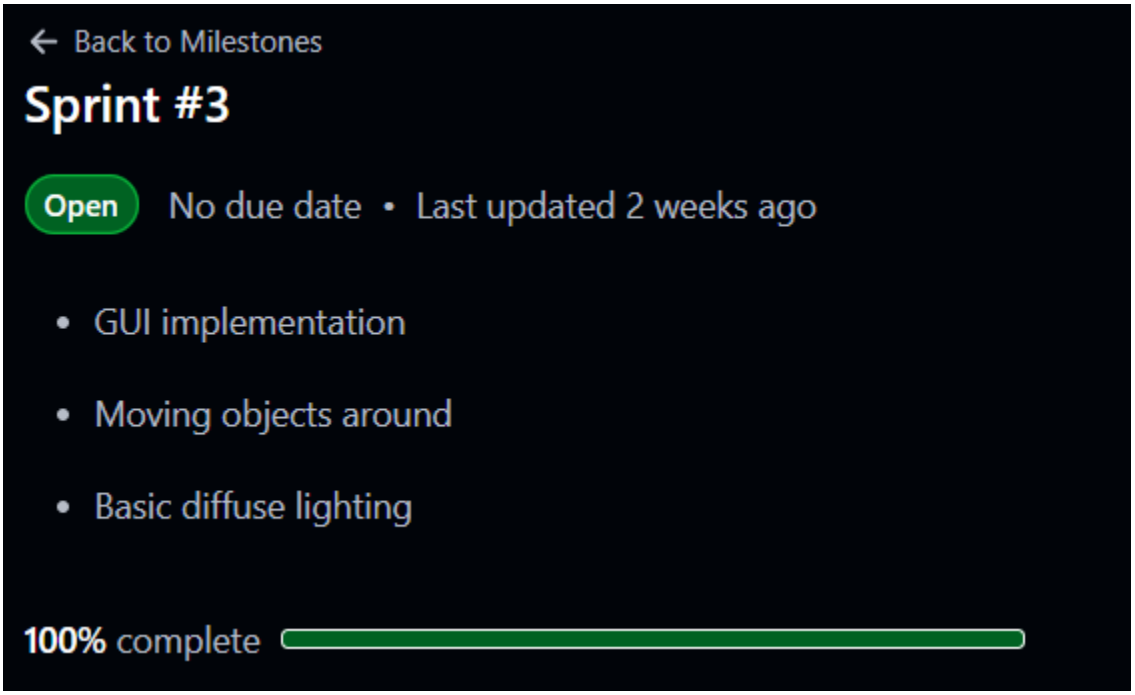

Figure 74, sprint 3

Sprint 3 was dedicated to implementing the GUI using ImGUI, adding object movement and basic lighting. This sprint took the least amount of time as getting object movement working was quite easy, as well as having the GUI taken care of by an external library. The diffuse lighting was quite easy to implement as well.

All of the goals were met for this sprint, and no extra goals were completed.

Phase (sprint) 4

Phase 4 was dedicated to the time between sprint 3 and project deadline. This phase includes mostly thesis writing and bugfixing + adding small functionality. The reason this sprint doesn't have a set list of goals is because the project now requires many different tweaks, so it's mostly free-for-all at this stage.

Conclusion and Future Work

Summary of Findings + Reflection

The findings of this project can be summarized in a few key points;

1. 3D software development is extremely difficult, time consuming and intensive, increasing exponentially with time invested.
2. 3D engines developed with OpenGL as the graphical API use shaders to achieve all the graphical effects on screen, from rendering to post-processing
3. The OOP paradigm works well with node(object)-based 3D software
4. SDL3 + OpenGL is a very effective and feature-rich combination when developing 3D software

The project's initial aim was to create a simple 3D game engine. This, however, proved to be quite out of scope for a single-person project limited to ~5 months, so the focus changed to researching 3D software development in general, with a heavy focus on OpenGL. This, while regrettable, was a good choice, because just the 3D rendering part of the game engine took a tremendous amount of learning, revision and trial & error. This ended up being a good thing, however, because diving deep into the intricacies of 3D rendering was a priceless learning experience, one that I will surely be grateful for in the future.

Personally, I started this project knowing little to nothing about the techniques, theories and workflows involved in creating 3D software, or even standalone software in general. I learned what a context is, what a window is, how to use external libraries with C++ just to name a few of the most basic concepts. The intricacies of shader coding, rendering pipelines, GPU communication, rasterization, and frame debugging have sparked a new interest in software development for me, and I would like to pursue this area of research further.

To talk a bit about development methodology, I must admit my focus was waning by the end of the project, as the rapidly increasing complexity had taken a toll on my motivation. This ended up reflecting in the development stages, as the later sprints are less organized and sparser with updates and documentation. I would've liked to have spent more time working on object importing and object textures, for example. I think this does show in the final product, as it is not nearly as feature rich compared to other 3D software.

For future development, I would like to add robust model importing with the GUI, a 3D grid and 3D widgets for object movement, full material support with lighting, skyboxes, and bring the project more in line with other 3D software.

Overall, I would classify this project as a success, and technical research as a valuable replication of already established techniques and paradigms.

References

De Vries, J. (2014). *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL*. learnopengl.com. Retrieved December 5, 2025, from https://learnopengl.com

Shreiner, D., Sellers, G., Kessenich, J., & Licea-Kane, B. (2013). *OpenGL programming guide: The official guide to learning OpenGL, version 4.3.* Pearson Education.

Khronos Group. (n.d.). *OpenGL - the industry standard for high performance graphics*. Copyright (C) 2000 - 2025 Khronos Group. https://www.opengl.org/

Gregory, J. (2018). *Game Engine Architecture*. https://doi.org/10.1201/9781315106946

OpenGL Wiki. (n.d.). https://wikis.khronos.org/opengl

Lantinga, S. (1998). *GitHub - libsdl-org/SDL: Simple DirectMedia Layer*. GitHub. https://github.com/libsdl-org/SDL

Unity Technologies. (2023, November 10). *Unity Manual*. Unity Technologies. https://docs.unity3d.com/Manual/index.html

Drumond, C. (2024). *What is scrum and how to get started*. Atlassian. https://www.atlassian.com/agile/scrum

Appendix

Github Repository for documentation: https://github.com/janiadt/college_major_project

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student : Jan Pantic N00222591

Signed

Failure to complete and submit this form may lead to an investigation into your work.