Optimizing workflow for virtual environments using Vulkan Model Viewer and Exporter

The University of Roehampton
Department of Arts and Digital Industries
London, United Kingdom

Author
Mr. Zakariya Oulhadj

Supervisors
Dr. Charles Clarke
Alex Collins

Submitted in partial fulfillment of the requirements for the degree of
Bachelors of Science in Computer Science

April 23, 2023

I would like to dedicate this report to my family, friends and lecturers who have supported me throughout my degree

Abstract

Computer graphics is a rapidly growing field that is vital in many industries that rely on digital graphics including scientific research, simulations, education and training, entertainment and more. The flexibility of this field and the increase in computing resources is what makes it so powerful and provides real-world benefits in ways not previously observed before the use of graphics software.

This report presents my final year project an application built from the ground up using the latest technologies for 3D graphics rendering. Its goal is to be easy to use, and performant and include a collection of tools for graphics manipulation that users can take advantage of when designing and building virtual environments.

Declaration

I hereby certify that this report constitutes my work, that where the language of others is used, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions, or writings of others. I declare that this report describes the original work that has not been previously presented for the award of any other degree or institution.

April 23, 2023

Date

z.oulhadj

Signature

Contents

1 Introduction

Computer graphics is a vast area within the field of Computer Science. Since its conception in the early 1960s, it has been used for a wide range of purposes including gaming, film, scientific research, education, architecture, engineering, medicine and more recently within vehicles. This is a testament to how versatile this field has become and the major benefits that computer graphics provide to the real world.

Although widely used, the development of virtual graphics and environments commonly poses a steep learning curve due to the technical details. Many different rendering applications already exist that aim to provide a platform to easily create digital graphics such as Unreal Engine[1], Unity[2] and RenderMan[3] just to name a few. These applications are known as "3D creation tools" that provide a plethora of features and functionality giving the users the freedom to create anything they need. However, these tools as a result of a large number of features inadvertently introduce several issues such as increased complexity and significant hardware requirements.

To address these issues, I introduce my final year project, Vulkan Model Viewer and Exporter (VMVE). A standalone application developed in the domain of Computer Graphics that provides an easy-to-use, efficient platform for 3D graphics rendering and 3D asset safety. VMVE will provide a specific subset of features that are designed from the ground up for rapid graphics prototyping which include constructing virtual environments, simulating the effect of light on 3D geometry and securing digital assets through the use of encryption.

1.1 Overview

This report presents the development of the final-year project and is structured such that it outlines each aspect of the development in chronological order. There are a total of four key sections in this report:

Design Focuses on the concept of the project, the design of the requirements obtained during the requirements-gathering stage followed by the design of the application including the user interface and internal engine architecture.

Implementation Will take the proposed designs and consist of the development of the application as well as presenting the technical implementation details/decisions.

Evaluation comes after the implementation and the main goal of this section is to ensure that the application has met the aims and objectives outlined in the projects.

Future Work is the final key section that aims to outline the aims and specific features that will need to be implemented going forward in regards to the project.

1.2 Requirements gathering

Successfully designing, implementing and evaluating a project relies on a set of aims and objectives to be fully defined. The first step toward obtaining these requirements relied on a "Requirements gathering" stage. The purpose of this was to understand who the stakeholders are and their specific needs.

1.3 Aims and Objectives

VMVE has three main aims and consists of several objectives that outline the concrete steps required to achieve those aforementioned aims.

Platform for virtual environment authoring The first goal of VMVE is to provide an "easy-to-use" platform for creating virtual environments. This means that the application should not have a steep learning curve and users with little to no prior experience with 3D rendering software and the underlying technology can use the application to achieve their intended goals in the domain of computer graphics.

The project will be built to run as a desktop application that renders a user interface editor as well as a virtual environment. The User Interface (UI) must be designed intuitively and reduce the amount of friction for a user. Furthermore, the language used throughout the user interface must reduce if not eliminate jargon. A simple example could be to use the word "Move" instead of "Translate" when referring to objects in the virtual environment and as such, would make the application more accessible.

Lower hardware requirements The second aim of the application is to be as efficient as possible with the goal of lowering the hardware requirements. The application's efficiency and aims vary based on the component in question that include the Central Processing Unit (CPU), Graphics Processing Unit (GPU), memory and storage. From

the ground up VMVE will strive to be as efficient as possible by using the latest technologies available.

Provide secure 3D asset encryption The last of the three main goals is for the application to include its custom model format that makes use of encryption to safely secure a user's digital assets. The motivation for this functionality formed as a result of the increase in theft regarding digital assets. A recent example relates to the recent data leak that affected Rockstar Studios and their upcoming game (Grand Theft Auto 6)[4] that saw assets, source code and documentation released to the public.

VMVE would be able to provide a platform to encrypt these assets and only unlock them using a special key that only the user will know. This ensures that in the event of digital assets begin obtained by third parties, they cannot be used. This requires several key elements to work in tandem to achieve this aim.

The raw model must be able to be encrypted using different types of encryption algorithms. In addition to this, VMVE must include a parser that can read the internal structure of the encrypted file format and successfully load the model into the application.

1.4 Considerations

1.4.1 Stakeholders

Two main stakeholders will be directly affected by the project and subsequently VMVE. These are the final-year project supervisors and the end users. Each stakeholder needs a different set of requirements to be successfully achieved.

The supervisors (Dr. Charles Clarke and Alex Collins) oversee the progress of the project. Meetings are to be held regularly and aim to inform all parties of the progress being made. This ensures that the requirements set out are being met.

The users on the other hand are individuals that will use VMVE once it has been developed. The aims and objectives discussed earlier directly affect these users.

1.4.2 Legal

Legal considerations are important when distributing an application indented to be used in production. The project and subsequently the application must abide by local laws and regulations.

VMVE is to be licensed under the MIT license[5]. This ensures that the application can be used without restriction allowing for it to be copied, modified as well as distributed.

Additionally, security must be taken into account when developing the application. As mentioned earlier, VMVE will include the ability to secure critical assets including 3D models. This will be achieved using encryption by making use of a secret key and initialization vector.

When developing the system, it is, therefore, important to ask questions such as "How will this data be kept secure?" and "Does the application store or send any private data?".

Throughout the development of this project, the application will not contain any networking functionality. This ensures that all data remains local to the application and the user's device.

Given the domain in which the application is intended to be used, such as handling third-party 3D assets by importing and exporting geometry data. VMVE mustn't infringe upon the copyright of those who own the models. One such example includes not modifying the geometry data unless specified by the user. Since this will be a new application, a lot of the intricacies behind Intellectual Property (IP) need to be further defined and taken into account based on user feedback. VMVE will not modify any of the data being imported and only change the visual appearance based on several factors including rendering settings, lighting, translation, rotation and scaling.

1.4.3 Social

VMVE must also take into account social considerations for those using the application to provide a consistent user experience.

Accessibility is a vital aspect that needs to be addressed throughout the development to ensure the application is useable for those with disabilities.

1.4.4 Ethical

One of the main ethical considerations that must be taken into account is the computation that the application will perform internally. The technical implementation details are not accessible to users as the source code is not open source. Therefore, individuals do not know exactly what the software is doing during runtime.

To ensure that VMVE remains ethical, the application will not perform any additional computations that are not strictly required for 3D rendering. Furthermore, the application will not store or send any data as this is not required.

Deceptive design also known as "Dark patterns" refers to designs/tricks that manipulate users into performing certain actions without their informed consent. VMVE will make use of a user interface, however, during the design and implementation stages extra effort will be spent on ensuring that no such design pattern is introduced. This will greatly benefit users and thus, increase the overall user experience.

2 Technology Review

This project made use of various technologies at different stages throughout the development process and was a key aspect in helping achieve the final goal of developing a 3D model viewer and exporter.

Technologies are categorized into two areas based on the impact they have on the project such as direct and indirect influence. A technology that has a direct impact means that it assisted in some way in the implementation of the project. Whereas, indirect impacts are technologies that are used in some areas not directly responsible for the outcome of the project such as project management.

2.1 Tools

2.1.1 Project Management

A Version Control System (VCS) is a tool used for backing up and/or collaborating with developers on a project. The use of a VCS was an obvious choice as this would provide a platform on which the project source code could be hosted. This gives me the peace of mind of knowing that if for some reason a local copy of the project is lost or corrupted then another copy is safely hosted on the servers managed by the VCS.

Another key feature of a VCS is project management. These types of systems provide various tools that greatly benefit developers. One such feature is known as a "commit" which records any changes made to a particular repository at that moment in time. Developers use commits to view changes that occur at each stage but also, provide means of reverting to previous versions of particular sections, files or even an entire repository. Due to the length and complexity of this project, tools such as this provided by a VCS are invaluable throughout the development process.

The version control system that was chosen was Git [6]. This is the most widely-used, free and open-source VCS. Git can be used in several ways such as by installing Git onto a server manually or by using existing platforms that are built on top of Git. A few examples include GitHub, GitLab and Bitbucket. By far the most popular option is GitHub which is the platform that I am most familiar with and therefore, GitHub will be the platform of choice for this project.

The VMVE project is hosted on GitHub as a private repository `https:`

//github.com/ZOulhadj/vmve/

GitHub as well as providing hosting for the repository also provides different features related to task management. One such feature is known as GitHub issues. This will be used as the task tracker in which "posts" are created that would track outstanding tasks including the priority, current progress and the expected deadline. Figure 1 shows a preview of the Kanban board. It consists of three main columns used to categorize an issue including backlog, in progress and done.

When an idea for a new feature is formed or a bug within the application is discovered a GitHub issue is created and moved to the "Backlog" board. This is used to document a particular task that needs to be worked on within a desired timeframe.

Then once a specific issue is ready to be worked on/resolved, it's moved to the "In Progress" board. As the issue is being addressed, key points of discussion are added as comments to the issue for documentation purposes and to be referred back to.

Lastly, once the task has been completed, the GitHub commit that includes the specific fix is referenced in the issue and is then finally marked as complete by being moved to the "Done" board. This subsequently marks a GitHub issue as closed.

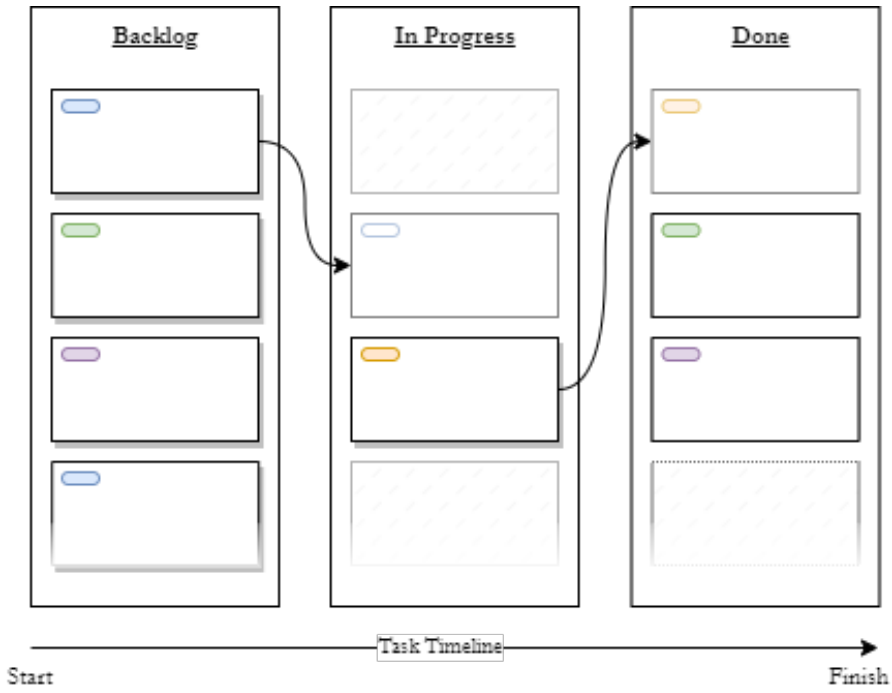

Figure 1: Kanban design

2.1.2 Microsoft Visual Studio 2022

Developing a program that runs directly on the underlying operating system requires a compiler. This is a program that parses source code and generates assembly instructions that the CPU will be able to understand and therefore, execute. Microsoft Visual Compiler (MSVC) also known as CL will be the compiler of choice. This is a compiler that comes bundled with the Microsoft Visual Studio Integrated Development Environment (IDE).

The IDE also provides debugging functionality that will be used extensively to fix crashes, bugs and generally ensure that the application runs as expected[7].

Some additional tools will be used that will further improve the ease of development. Visual Studio Assist X[8] is one such tool. This is a Visual Studio extension that provides many useful features that the base IDE does not provide such as reliable symbol renaming, file symbol outlining, quick

file searching and much more.

2.1.3 RenderDoc

As well as using software for debugging on the CPU, there also exists software that allows for debugging the GPU. More specifically, these tools allow for analyzing per-frame data as well as detailed frame synchronization metrics.

RenderDoc developed by Baldur Karlsson can be used to inspect different rendering Application Programmable Interfaces (APIs) including individual frames and the entire state[9]. This ensures that you can debug specific GPU related bugs such as model data, textures, rendering commands etc.

2.1.4 AMD Radeon GPU Profiler

Similarly to RenderDoc, AMD has its own GPU profiling tool called "AMD Radeon GPU Profiler"[10]. This tool is only available for AMD-supported GPUs which is ideal as the hardware going to be used throughout development meets this requirement. The tool analyzes per-frame data which subsequently provides insightful performance metrics.

2.1.5 Programming Language

The application needs certain requirements to be met regarding performance. For example, the application must use a high-performing programming language, allows for low-level memory access and can be compiled into an executable.

The language chosen for this project was C++ 23. There were two main reasons why this specific programming language was chosen. Firstly, it's the language that I have the most experience with which will significantly help during the implementation stages of the project. Secondly, the C++ Standard Template Library (STL) was one of the key reasons why C++ will be used. It provides many prebuilt data structures, containers and algorithms including "std::vector", "std::string" and "std::find" to name a few. Furthermore, it saves time as I would not have to implement a custom solution in the limited time frame. Other language features that solidified by choice include, function overloading, templates, compile-time expressions, direct memory access and generally faster performance compared to higher-level languages such as Python and Javascript.

2.1.6 Rendering API

A rendering API is a set of instructions that acts as an intermediary and allows for an application to make use of a systems GPU. There are several different rendering APIs that exist including OpenGL, Vulkan, Direct3D 11 and Direct3D 12. Appendix section 12.1 goes into great detail explaining the history of rendering APIs and the differences between them.

OpenGL and Direct3D 11 are previous generation APIs whereas Vulkan and Direct3D 12 are using the latest technologies. Each of these has its benefits and disadvantages. The image below shows a table that compares each of these with specific types of functionality.

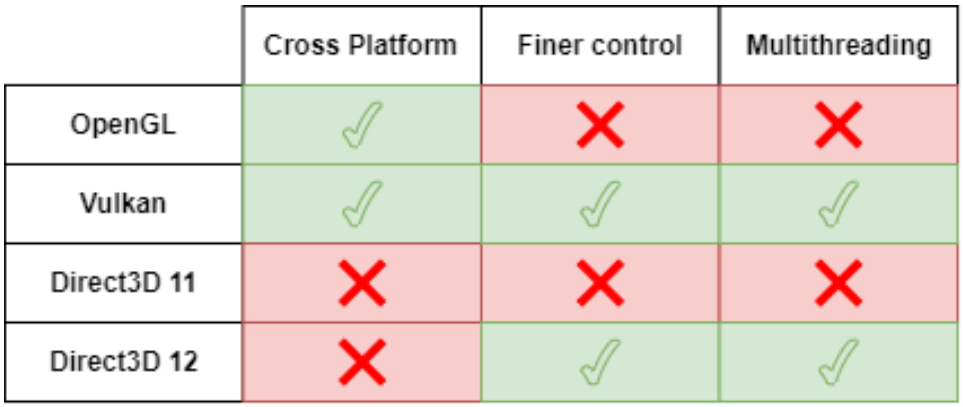

Figure 2: Rendering API comparison

Figure 2 shows that Vulkan has the most amount functionality including the ability to run on different operating systems, finer rendering control and support for multithreading.

In addition to Vulkan inhibiting the most functionality, there are numerous other reasons why this API is the most suitable for VMVE. Firstly, Vulkan is extremely verbose as it exposes the technicalities related to the GPU and requires the application to implement the basic functionality as opposed to the driver. This reduces the complexity of the display driver and can potentially increase performance as it no longer needs to make assumptions about the application and the commands being issued as the entire state of the rendering pipeline is predefined. Using Vulkan also implicitly helps understand the complex inner workings of graphics development and working

with the GPU.

2.2 Libraries

The application will make use of different external libraries to speed up development due to the short timeframe.

2.2.1 User Interface

Users will need a way of interacting with VMVE and the 3D environment. This will be achieved through the use of a user interface in which the user can directly manipulate the application. As mentioned earlier, VMVE is an application that uses the GPU for rendering and therefore, the UI will have to interact with the GPU. To reduce the development time of this particular aspect of the application, the decision was made to make use of a preexisting library.

The library of choice was Dear ImGui [11]. This is an immediate-mode user interface library that provides an API to render UI elements.

2.2.2 Encryption

VMVE will include its own model file format. This is a custom format that will be encrypted as standard. Implementing encryption is a very complex area that can be considered an entire project on its own. Instead, the project will make use of a well-known encryption library known as Crypto++ [12] and the application will use version 8.7. This is a C++ library that provides many algorithms including AES, Diffie-Hellman, GCM, RSA and much more.

2.2.3 Serialisation

In addition to encrypting the model data, the custom file format will need to store additional information such as application version and encryption type. Therefore, VMVE must be able to serialize the data when saving to disk as well as deserialize when loading a ".vmve" model. To achieve this, the cereal C++ library will be used [13]. This library allows for fast and efficient data serialization into different formats such as compact binary encodings, XML or JSON.

2.2.4 Additional libraries

The other libraries used in VMVE can be seen in the list below.

assimp Model loading

glm Mathematics

stb Image loading

vma Vulkan memory allocator

volk Vulkan meta-loader

3 Design

The next stage of the project was designed and occurred between milestones two and three.

Having discussed the different technologies being used in VMVE, they must now be evaluated and incorporated into the design of VMVE and its various subsystems.

3.1 Branding

The branding i.e the look of the application was an important factor during the design as it will be presented to users and can directly influence the user experience. Furthermore, being recognizable results in an increase in brand recognition that may also increase user traffic.

3.1.1 Project Name

VMVE stands for "Vulkan Model Viewer and Exporter". The name is made up of two parts. The first is "Vulkan" the other is "Model Viewer and Exporter". The rendering API used to render 3D digital assets is Vulkan and hence the first part of the name. The second part is because the application will include functionality such as importing and exporting models into a custom file format as mentioned in section 2.2.2. The combination of these two allows a user to have a basic understanding of what the application does without having used it before.

3.1.2 Logo

There are two versions of the VMVE logo that was designed and intended to be used in different scenarios. The first is the complete logo as seen below in figure 3 which is referred to as the "large logo". It contains the icon and the name of the application to the right of that.

Figure 3: VMVE Large Logo

This is to be used on the VMVE website as well as documentation pages providing consistent branding across different mediums.

The second version is designed to be minimal and therefore, only consists of the icon. This is intended to be mainly used throughout the application and in locations where there is a minimal amount of space such as the Windows taskbar.

Figure 4: VMVE Small Logo

3.2 VCS Architecture

As the only developer for this final-year project, the design and architecture of the VCS will remain as simple as possible while still providing the core requirements. Figure 5 shows the proposed VCS architecture and includes two branches named “main” and “develop”. Develop branch will be the primary branch used throughout the implementation stages. In other words, when changes occur throughout the source code, all commits will be published to the development branch.

On the other hand, "Main" will be used as a stable branch that is only updated for each official release. This would occur for each project milestone such that for "Sprint 1" a pull request will be made from develop to main and this will be tagged as v0.0.1 and likewise "Sprint 2" will be tagged as v0.0.2.

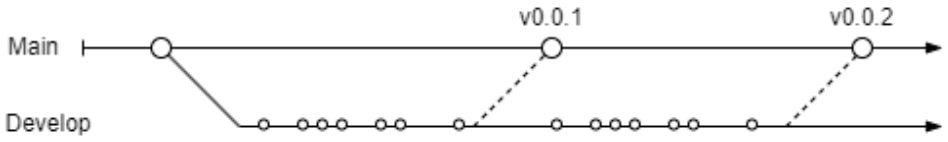

Figure 5: Git branch design

This architecture ensures that as the sole developer of the project, changes can be made consistently while ensuring that users are provided stable versions of the application.

3.3 Source code

From the beginning, the projects source code and overall architecture were to adhere to the C++ Core Guidelines[14] to ensure that the project follows best practices.

A significant amount of consideration was spent planning out a suitable project-wide programming style. In regards to naming, section NL.10 of the core guidelines recommend using the snake case style as it follows the standard libraries naming convention. Since the project has no existing code base and therefore, no existing convention to follow, the project will make use of the underscore style for types, functions and variables.

Additionally, section NL.17 states that the use of the "K&R" indentation style[15] should be used as it preserves vertical space whilst maintaining readability. In other words, reducing vertical line height for code blocks such as "if, else, while and for" allowing for more lines of code to be visible at any given point whilst allowing for a more distinct separation of structures and functions.

The combination of these two specific conventions regarding source code style can be seen in figure 6.

```
1  struct foo
2  {
3      int a;
4  };
5
6  void bar(int a)
7  {
8      if (a) {
9          printf("This is a example.\n");
10     } else {
11         printf("This is another example.\n");
12     }
13 }
```

Figure 6: Source code convention

VMVE will also be developed in a functional style and by using “structs“ instead of an object-oriented one with “classes”. This is primarily related to the ease of use in terms of development since data can be more easily accessed and manipulated. Furthermore, this will eliminate the need for getter and setter functions that needlessly take up source code space and is redundant when member variables are accessible.

3.4 Project Architecture

VMVE will be a combination of two projects. The “Engine” project also known as the core of VMVE will contain the fundamental implementation details. This includes the Window, Renderer, UI and Audio. This project will be distributed as a library file (.lib) that other projects can import for specific use cases. The “VMVE” project will include the “Engine” project by importing the library (.lib file). A high-level overview of the project architecture can be seen in figure 7. Furthermore, by making this distinction between the core and VMVE, it will allow for the same sets of tools and technologies to be used in the future for other similar projects.

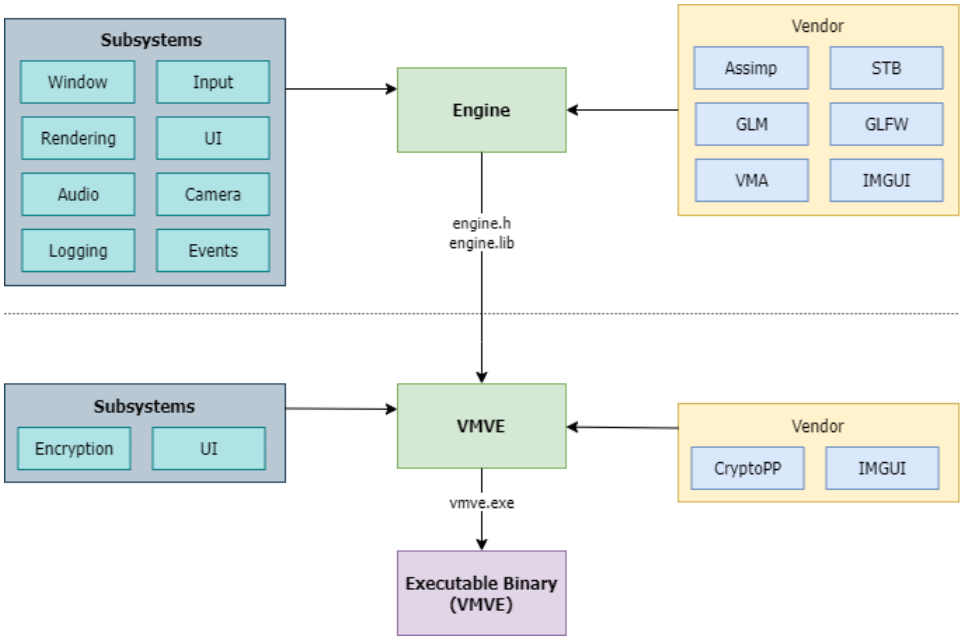

Figure 7: Project Architecture

3.4.1 Platform Architecture

The application will be developed to take advantage of the 64-bit platform. This would allow for the application to go beyond the 4GB limit posed by 32bit programs. In addition to this, 64-bit provides better performance and security as a result of the larger address space.

3.4.2 Renderer Architecture

As mentioned in section 2.1.6. Vulkan will be the rendering API of choice for VMVE. The API is extremely verbose giving the programmer the flexibility to control every aspect of the GPU. When interacting with Vulkan throughout the engine a certain degree of encapsulation is necessary to reduce the amount of effort required to implement functionality as a result of Vulkans verbose API.

Therefore, the renderer must be designed so that flexibility is maintained whilst simplifying the API. Figure 8 shows the proposed renderer architecture which makes distinctive boundaries and encapsulates key sub-systems.

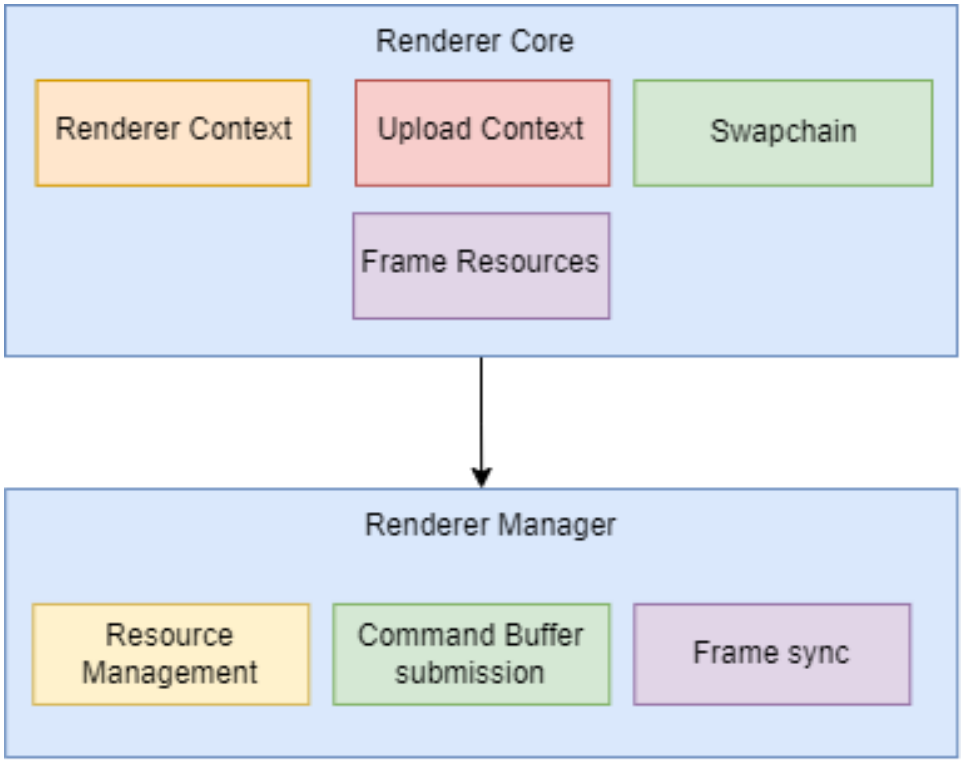

Figure 8: Renderer Architecture

3.5 User Interface

Designing the user interface was the next step as part of the design stage of the project. Figure 9 shows the initial user interface wireframe that includes four main elements titled “Global Controls”, “Logs”, “Model Controls” and “Main Viewport”. Each of these UI elements is located in their respective windows which are designed in such a way that common controls are grouped and located appropriately if not within the same panel.

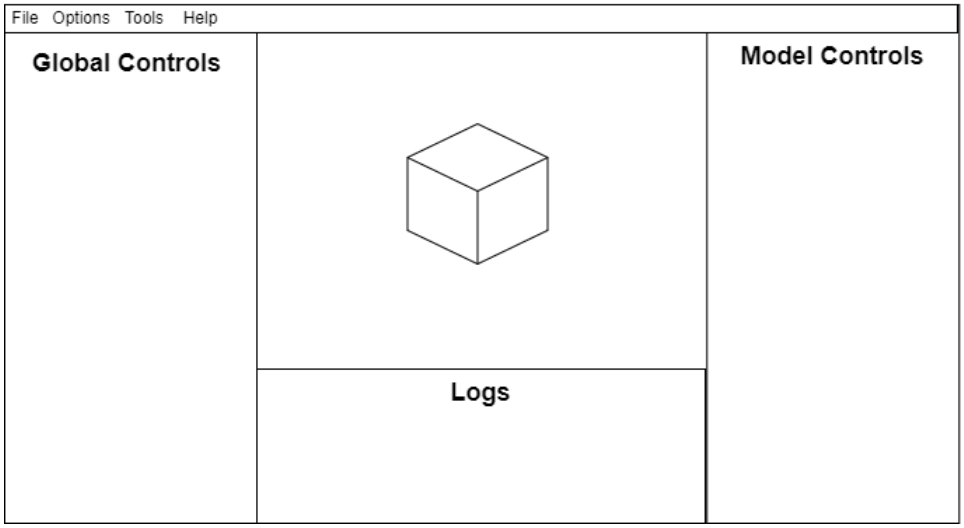

Figure 9: UI Design

The purpose of the user interface is to provide an easy-to-use platform for interacting with VMVE and more specifically the main viewport located in the center of the application.

3.5.1 Main Viewport

The main viewport is located in the center of the window and is the main feature of the user interface. The viewport displays the virtual environment and

3.5.2 Global Controls

3.5.3 Model Controls

Each model loaded into the application and currently highlighted is shown model-specific information within the model controls panel. This panel will include various

3.5.4 Logs

The logs panel is designed to contain all internal messages that the application prints out. These messages will then be displayed within the logs

panel. Each log message will be categorized as either log, warning or error. Depending on which log type the message is, it will be shown in a different color such as white, orange or red respectively.

This is designed so that the user will have a clear understanding of the internal state of the application.

3.6 Custom file format and encryption

3.6.1 File format

VMVE will include a custom file format that allows for model data to be encrypted for security purposes. Figure 10 shows the proposed internal structure of the file. The internal structure is split into two main sections. The header contains key pieces of information that help the application determine how and if the main data should be decrypted.

Version indicates which version of VMVE was used to encrypt the file. At the moment, this is to be purely informational however, in future versions, this can be used for reporting compatibility issues and also, allowing for encrypted files to be updated to newer file layouts.

Mode is the encryption algorithm used (e.g AES, DH) and is what will allow the application to know which decryption algorithm needs to be used.

Encrypted Keys is used for validating the input key and initialization vector. If these match then the application will then decrypt and load the data.

The second item in the VMVE header is the encryption mode. This allows for VMVE to know which algorithm must be used to decrypt the data.

The remaining section of the file consists of the encrypted data which includes the raw model data.

Figure 10: VMVE File internal structure

3.6.2 Encryption and Decryption

When dealing with encryption there are two primary stages, encrypting and decrypting. The term "encryption" means taking some data and running it through an algorithm making the data unreadable. "Decryption" is the opposite of that which takes the unreadable data and turns it back to the original data.

Designing such a system requires careful consideration and extra care to ensure that the data being managed is not negatively affected such as information being lost when converting from one stage to another.

The primary encryption algorithm that will be used is AES and the key length can be two different values (128 or 256 bits) that the user will choose based on the level of security they want for their assets.

Figure 11 shows a flowchart describing the general process of how a model will be loaded into VMVE.

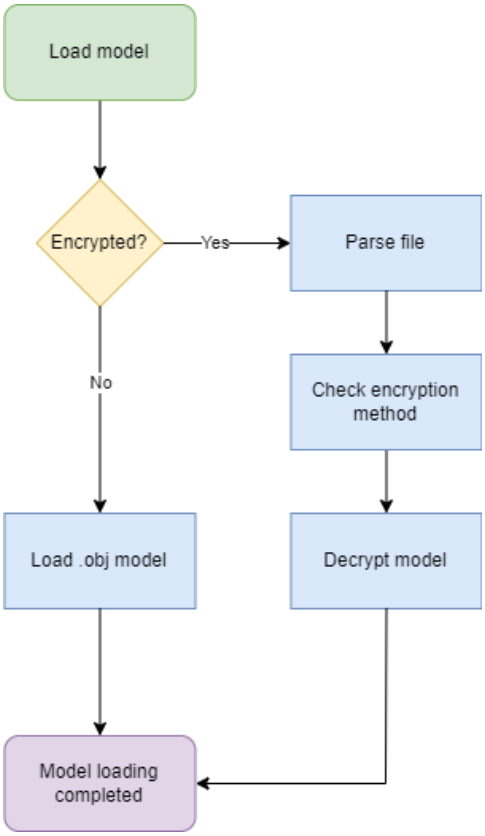

Figure 11: Load model flowchart

Likewise encrypting a model will consist of three main stages which are loading, encrypting and exporting as show in figure 12.

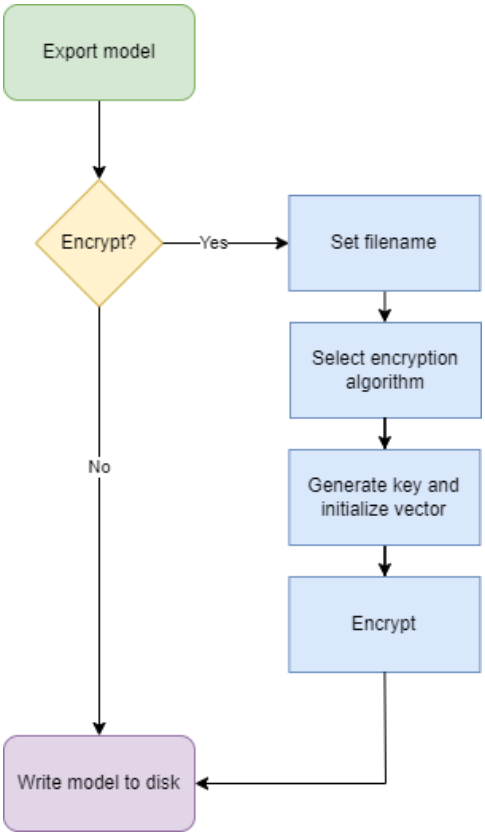

Figure 12: Export model flowchart

4 Implementation

This section of the report presents the technical implementation details of VMVE. This section is presented in order of initialization i.e starting at the core of the application and discussing each subsequent system.

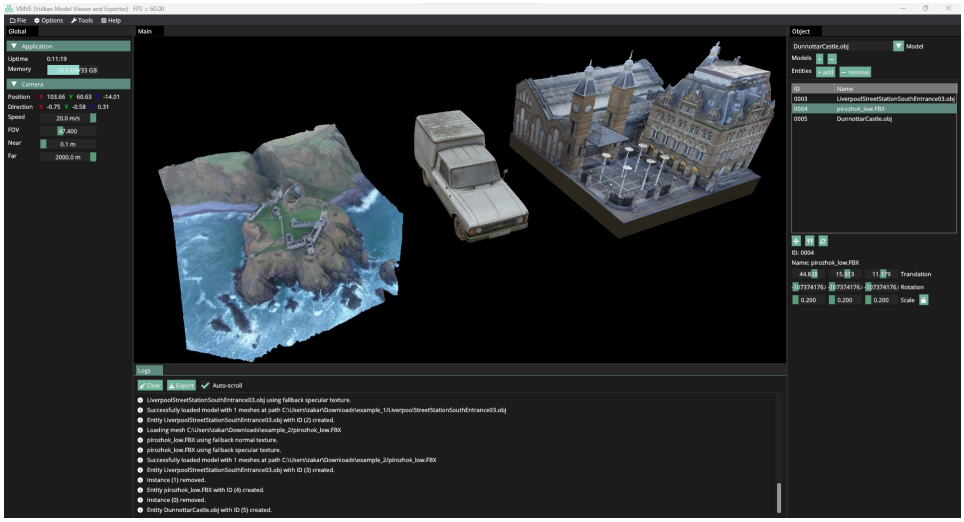

Figure 13: UI implementation

4.1 Overview

Figure 14 shows the general overview of the application and the various stages that occur during initialization, runtime and shutting down.

```
1
2  int main()
3  {
4    // begin application initialization
5    create_window(width, height, name);
6    create_renderer();
7    create_audio();
8    create_ui();
9
10   // set default configuration options
11   create_camera();
12
13   // begin application rendering
14   while (running)
15   {
16     update_renderer();
17
18     render_geometry();
19     render_ui();
20
21     update_window();
22   }
23
24   // shutdown application
25   destroy_ui();
26   destroy_audio();
27   destroy_renderer();
28   destroy_window();
29
30   return 0;
31 }
```

Figure 14: Implementation overview pseudo code

4.2 Window System

The first system within VMVE that gets initialized is the window. This system is responsible for creating a desktop window based on a series of configuration options specified at the start of the application. These options include the window width, height and name. Internally, VMVE uses the lightweight GLFW library to handle window creation. The purpose of this library is to provide an API which is cross-platform and allows applications to easily create windows on different operating systems.

Under the hood, on Windows, GLFW uses the Win32 API provided by

Microsoft.

In addition to the window creation, various function callbacks are created which allows VMVE to handle specific events such as window resizing, input, cursor position etc.

4.3 Rendering System

The implementation of the rendering system was one of the key areas worked on throughout the project. This system is responsible for communicating with the GPU and rendering objects onto the screen. As discussed in the 2, Vulkan was the renderer API of choice.

4.3.1 Renderer Context

The first step in creating the renderer is initializing an object called "vk_context". This object is created within the application that groups Vulkan handles that are responsible for initializing resources. In other words, the "vk_context" structure is used to allocate GPU resources. The handles within the context structure can be seen in the code snippet below and a diagram can be seen in figure 8.

```
struct vk_context
{
  VkInstance          // Initializes Vulkan library
  VkPhysicalDevice    // Handle to physical hardware
  VkDevice            // Logical handle to physical hardware
  VkQueue             // Graphics queue
  VkQueue             // Presentation queue
  VmaAllocator        // VMA memory allocator
};
```

The initialization of the renderer context can be broken down into several stages. The first of these states acquires the function pointers into the Vulkan loader which contains the implementation details for the GPU driver. This is achieved by statically linking a "stub" library file into the executable. However, as discussed in a blog post by Arseny Kapoulkine titled *"Reducing Vulkan® API call overhead"* [16], he finds that this method has some overhead due to how function calls are dispatched. To reduce this overhead, a meta-loader called Volk is used which contains pointers to the real implementation and is dynamically loaded at runtime.

The next step is acquiring a GPU with a specific set of requirements that can be used for rendering. This step involves querying all available GPUs

their names, hardware capabilities as well as if it supports basic screen rendering.

4.3.2 Swapchain

A swapchain is a series of images also known as framebuffers used by Vulkan that allocates a region of memory that stores frame data i.e pixel colors. As the name suggests, a "swapchain" is a chain of these images. The most common number of images is two or three which is called "double-buffering" and "triple-buffering" respectively. The renderer within VMVE is configured to use two images, one for the front buffer that is shown on a monitor and a back buffer that is used for rendering.

In addition to buffers, a swapchain is also responsible for handling vertical refresh rate also known as vsync. This is the rate at which new images are rendered and then presented on a user's screen. There are two main types of vsync modes used within Vulkan which are `VK_PRESENT_MODE_FIFO_KHR` and `VK_PRESENT_MODE_IMMEDIATE_KHR`. The "fifo" mode indicates to the swapchain and subsequently, the presentation engine that images are to wait for the monitor's refresh rate before being present whereas "immediate" simply renders images as fast as possible.

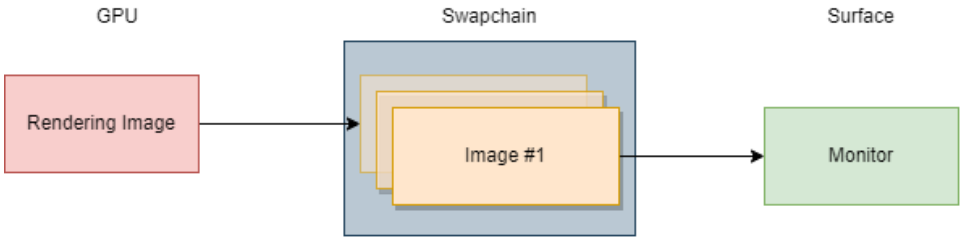

Figure 15: Swapchain

VMVE uses the "fifo" presentation mode as it conserves hardware resources as new images will not be rendered needlessly. Also, this mode ensures that no screen tearing occurs between refresh rates. Figure 17 shows a detailed image regarding the presentation engine and frame synchronization.

4.3.3 Frame synchronization

Once the swapchain along with its framebuffers has been fully initialized, the renderer can now start creating the necessary resources required for frame synchronization. Vulkan has two concepts that relate to synchronization which are VkFence and VkSemaphore.

```
1     struct vk_frame
2     {
3         VkFence submit_fence;
4
5         VkSemaphore image_ready;
6         VkSemaphore image_complete;
7     };
8
```

Figure 16: Frame sync structure

The Vulkan 1.3.246 specification states that "`VkFence` is a synchronization primitive that can be used between queue submissions and the host" [7.3 Fences]. In other words, a fence can be signaled on the CPU to indicate in this case, when a new image from the presentation engine has been acquired via the `vkAcquireNextImageKHR` function call.

Similarly, "`VkSemaphore` is a synchronization primitive that can be used between queue operations" [7.4 Semaphores]. For the application, this means that rendering operations can occur simultaneously for multiple frames while ensuring that frames "in-flight" are not interrupted.

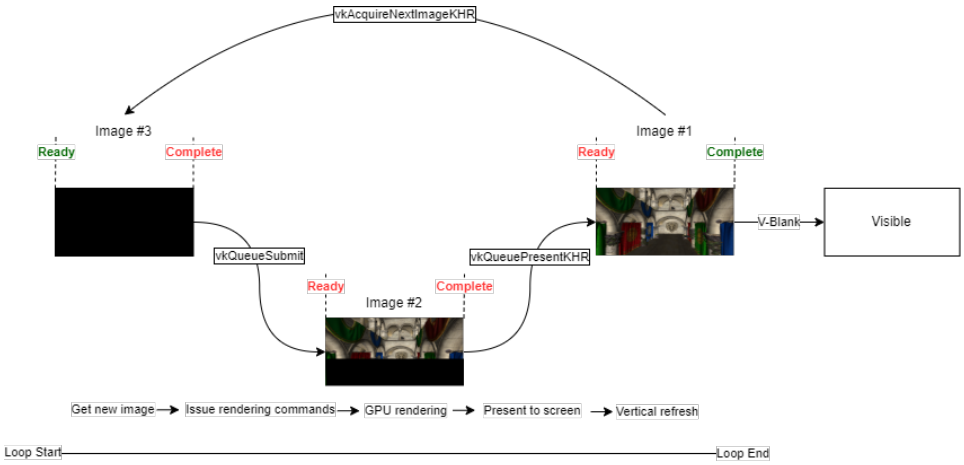

Figure 17: Frame Synchronization

4.3.4 Deferred Rendering Pipeline

The next rendering feature that was implemented was a deferred rendering pipeline. During rendering, there are two main passes. The first pass renders the raw geometry data and performs MVP translation (section 4.3.9). For each rendered frame a total of five framebuffers are filled with different types of information including colors, world positions, normals, depth and specular.

Then the second pass reads the data from the five framebuffers and performs the lighting calculations. This is a common technique in rendering applications that significantly improve performance when large numbers of light sources are present compared to forward rendering. This is because deferred rendering defers all lighting calculations to the final pass and only needs to perform those calculations once. However, for forward rendering, this needs to occur for each object in the scene resulting in poorer performance.

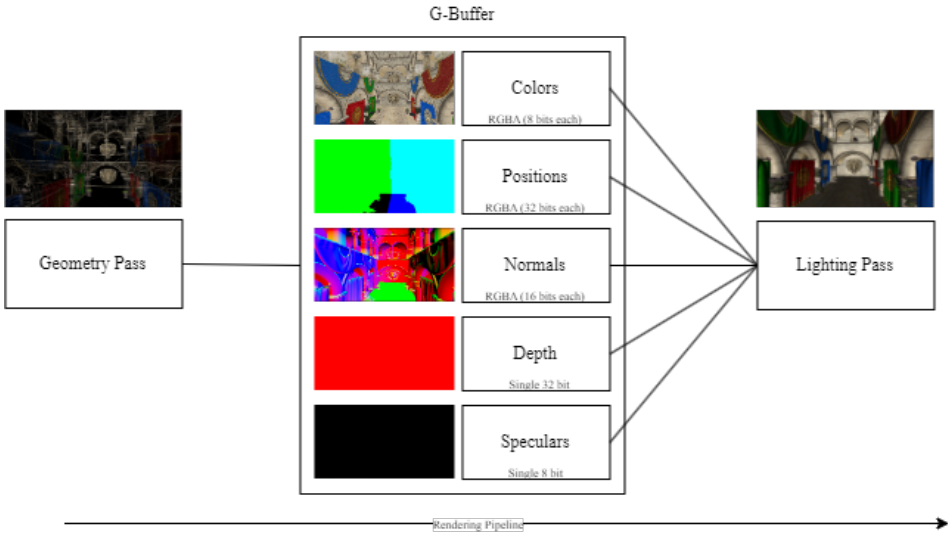

Figure 18: G-Buffer Pipeline

4.3.5 Dynamic Uniform Buffers

Uniform buffers are regions of memory located in Video Random Access Memory (VRAM) on the GPU. This memory can be accessed by shaders when creating a frame. These buffers can contain any sort of data including camera information, light properties, object transformations etc.

VMVE makes use of two dynamic uniform buffers one for the camera projection and the other for scene information such as lights. A dynamic buffer is a buffer that is large enough to store per-frame data and each frame of data can be accessed using an offset into the buffer and the current frame index. Figure 19 shows the internal structure of a dynamic uniform buffer that stores three sub-regions of memory in a triple-buffered frame setup.

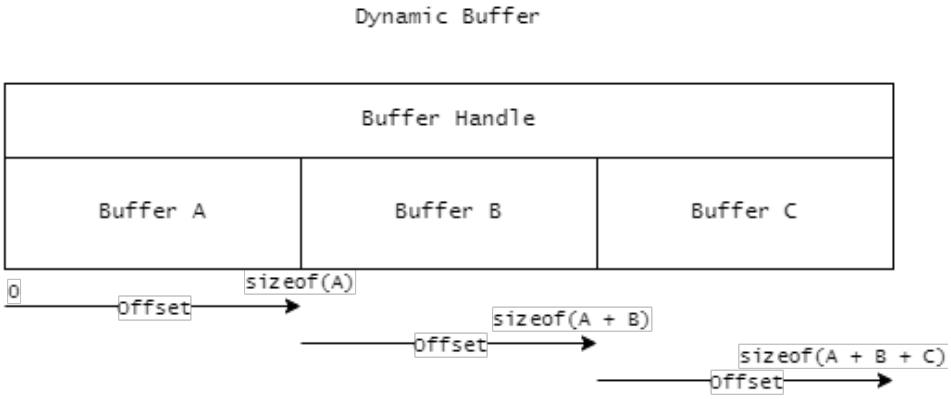

Figure 19: Dynamic Uniform Buffer

The use of dynamic uniform buffers provides several benefits compared to using a unique buffer handle for each frame.

4.3.6 Texture Mipmapping

VMVE makes use of a rendering technique known as "Texture mipmapping". This improves the visual fidelity by reducing if not eliminating Moire patterns which are visual artifacts that appear when identical patterns/textures are transformed on the screen[17].

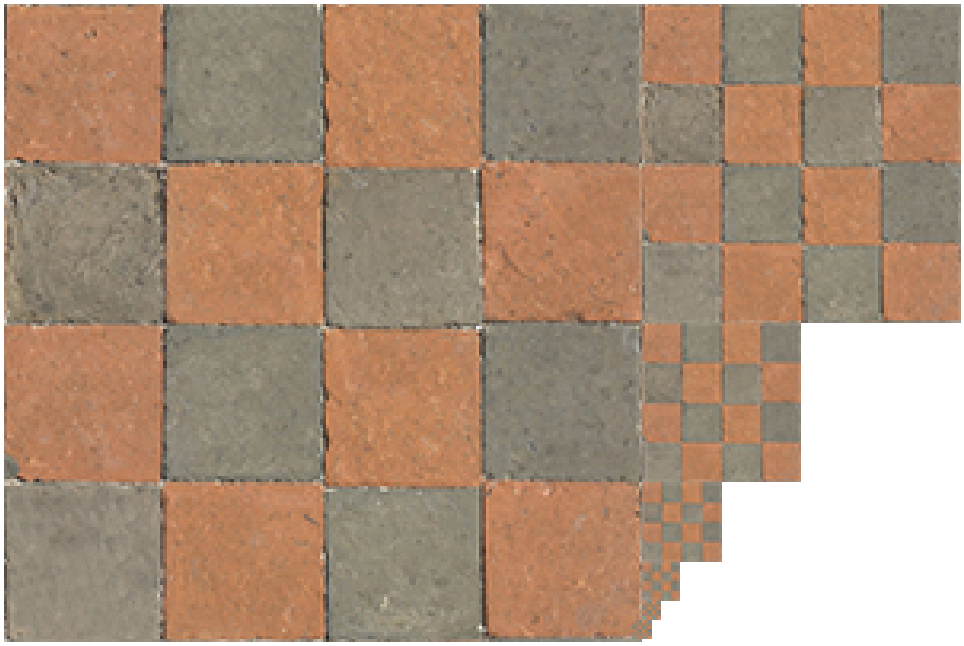

Figure 20: Texture mipmapping
[18]

Using Vulkan, this is implemented by taking the originally loaded texture and continuously halving it on both the width and height so that the subsequent texture is a quarter of the previous size. This is performed using the following equation provided by "Sascha Willems".

$$m = \lfloor \log_2(\max(w, h)) \rfloor + 1 \tag{1}$$

where m is the number of mipmap levels, w and h are the width and height of the source texture respectively. For each texture mipmap created the size of the texture becomes exponentially smaller.

In the source code, it looks like this:

```
1     const auto mip_levels = static_cast<uint32_t>(std::floor(
    std::log2(std::max(width, height)))) + 1;
2
```

Figure 21: Mipmapping equation

4.3.7 Models

In the context of the rendering system, a "model" is a structure that represents a 3D geometry object. This could be as simple as a cube or as complex as an entire scene. The data structure contains several key pieces of information such as geometry data and textures.

Complex models are not singular pieces of geometry. Instead, artists combine multiple smaller objects to form the final model. These smaller parts are known as "Meshes" within the application. Each mesh has its own vertex and index buffers stored on GPU that describe to Vulkan how the piece of geometry is to be interpreted.

Figure 22 shows the internal structure of an example model which has been loaded into VMVE.

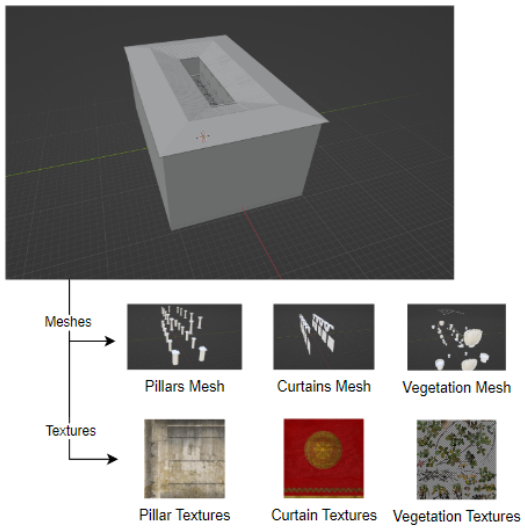

Figure 22: Model Structure

When a user loads a model, the model is appended to a continuous array (`std::vector<Model> models;`) which is iterated over each frame in order to render all the models on the screen.

4.3.8 Entities

An entity represents an instance of the model in the virtual environment. There can be many different entities using the same model. An example of this can be seen in figure 23 which shows four entities using the same backpack model.

This is achieved by storing a model index that indicates which model the entity refers to within the contiguous array of models. Secondly, an entity contains a transformation matrix that positions the entity in the world. A more in-depth explanation of the model matrix can be found in section 4.3.9.

Figure 23: Multiple entities in virtual environment

4.3.9 Camera

3D geometry data must be transformed through a series of mathematical calculations that will take vertex points from a 3D "world" and then convert them onto a 2D image for it to then be displayed on a monitor. This transformation is known as perspective projection [19] and is accomplished by computing several intermediate coordinate spaces as seen in figure 29.

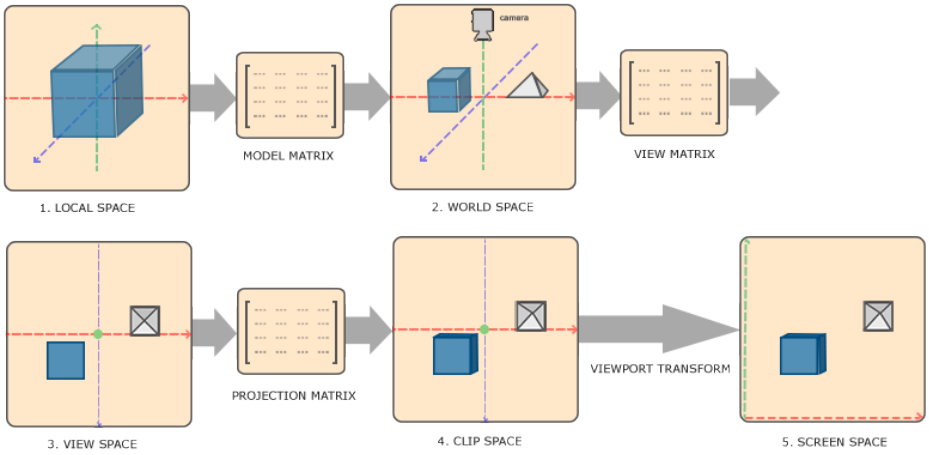

Figure 24: MVP Transformation [20]

Geometric data is first created within a local coordinate space. This "space" is positions relative to the object. In other words, the model data imported by VMVE will be in this local coordinate system often created by a 3D modeling program such as Blender or 3DSMax.

The next step is to convert these local coordinates to "world" space. This is the actual virtual environment that an object will live in. The transformation to this space is done by constructing a 4x4 matrix and performing a combination of translation, rotation and scaling. A pseudo-code example can be seen in 25.

```
1     mat4 model = mat4(1.0f);        // Identity matrix
2     model = translate(position);    // Move object
3     model = rotate(radians, axis); // Rotate object
4     model = scale(scale);           // Scale object
5
```

Figure 25: Model matrix construction

The next step is transforming the world space positions to view space or also known as camera space. This is important as how the virtual scene is displayed depends on the properties of the camera. In the context of graphics rendering the concept of a "camera" does not exist and is only

really an illusion. In reality, all points/vertex positions in the world space are transformed relative to the "camera". For example, if the position of the camera along the z-axis increases i.e. we move forward then all objects in the world will be moved towards us. A nice visualization of this is shown in figure 26 courtesy of a blog post by Jordan Santell https://jsantell.com/model-view-projection/.

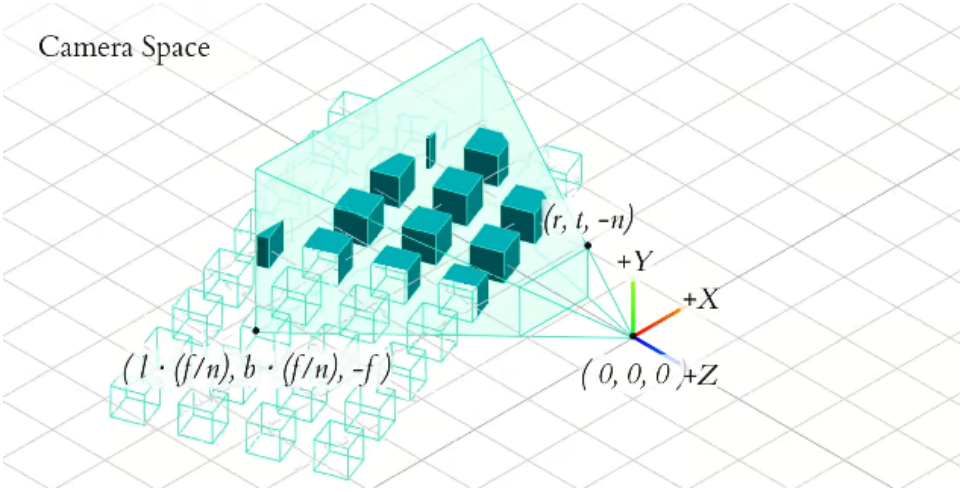

Figure 26: Camera/view projection [21]

VMVE makes use of a quaternion to perform the view projection. This is an alternative to Euler angles and has several benefits including, preventing gimbal lock and easier interpolation between orientations.

```
1     mat4 view = lookAt(camera_position, view_direction,
      camera_up);
2
```

Figure 27: View matrix construction

The final step in calculating the MVP matrix is the projection which converts points from local space to clip space. This involves projecting the points to either a perspective or orthographic projection.

```
1     mat4 proj = perspective(fovy, aspect, near, far);
2
```

Figure 28: Projection matrix construction

With the fully constructed mvp matrix, this can now be multiplied with each vertex position and it will display onto the screen as expected.

$$Output = MVP \times Vertex \tag{2}$$

Figure 29: MVP projection

A more detailed code example can be seen in figure 57 in the appendix.

4.3.10 Lighting

The implementation of lighting was the final rendering system that was implemented. Lighting in Computer Graphics refers to how the virtual environment i.e. the colors of the resulting pixels are manipulated based on certain properties such as light, surface direction, occlusion, emissive materials and more. The use of lighting significantly improvises the visual fidelity of the graphics and increases the realism.

The lighting model that was implemented is called Blinn-Phong which derives from the original Phong model was developed by Bui Tuong Phong in a paper titled *"Illumination for Computer Generated Pictures"* published in 1975[22]. It describes three main elements including ambient, diffuse and specular lighting that works together to produce the final result.

Ambient Ambient is a constant value that is used instead of global illumination as it requires more computational resources and more complex algorithms.

$$A = G + P \tag{3}$$

Diffuse The next step is implementing diffuse lighting. This takes into account the light direction L and the normal N for a surface at a particular pixel. The formula below calculates the intensity at which light reflects off of an object based on the angle off a particular surface.

The dot product of the light direction and surface normal returns a value between the ranges of -1 to 1 depending on how parallel the directions are. If this value is equal to or less than 0 then it indicates that the light is behind the surface and thus, is clamped as a result of the max function which returns the largest value for the given two parameters.

$$\mathrm{I} = \max(\vec{\mathrm{L}} \cdot \vec{\mathrm{N}}, 0) \tag{4}$$

$$\tag{5}$$

Having calculated the intensity value we can now simply multiply it with the objects' surface color as shown in the equation below.

$$\mathrm{D} = \mathrm{I} \times \mathrm{C} \tag{6}$$

Figure 30 demonstrates a simple example of diffuse lighting and how light interacts with an object based on the various properties discussed.

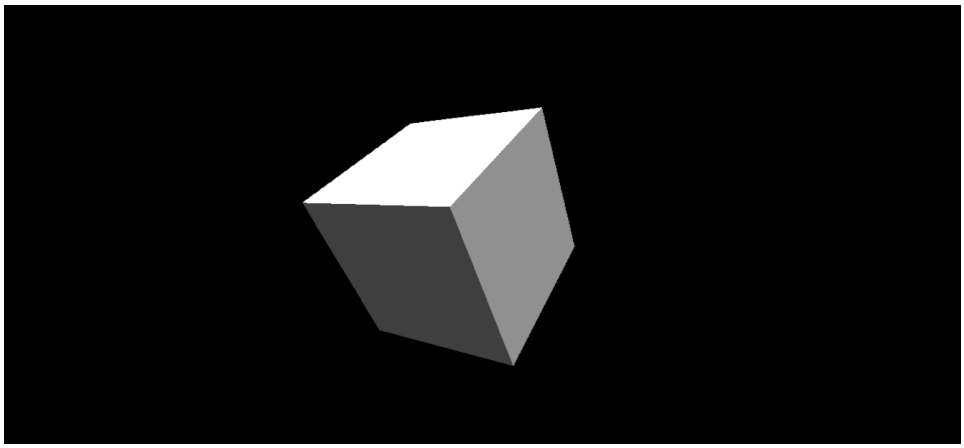

Figure 30: Diffuse Lighting

Specular The final step in the Blinn-Phong lighting model is specular. This effect adds highlights to certain parts of an object and is where the the intensity of light is strongest.

4.4 User Interface

The implementation of the user interface was the next major system. This is the front end and how the users will be able to interact with the application. The user interface is built on top of the Vulkan renderer.

4.4.1 Controls

The term "Controls" refers to a combination of keys also known as "Shortcuts" that are used to perform specific actions within the application. Various built-in controls aim to increase productivity for the user by reducing the amount of clicking and menu interaction for repeating tasks.

Controls are split into two categories which are Global and Viewport. Global controls are controls that can be activated at any point. Viewport controls, however, can only be activated whilst using the viewport i.e the camera.

This distinction for controls between different modes allows for greater context and better usability.

4.4.2 Fonts and Icons

The font used for the user interface is Open Sans which provides a simple and easy to read font.

VMVE also uses icons throughout the user interface and is an important aspect in conveying key information such as the task being performed or for additional information. The icon font used is provided by Font Awesome[23].

Typically, data for fonts and icons are stored in a font file which ends using extensions such as ".ttf" or ".otf". However, one of VMVEs goals is to be distributed as a single executable file. Therefore, we cannot depend on external font files. Instead, data for fonts and icons are stored directly in the application in a continuous array of bytes and are encoded in base85. A small example of what this looks like can be seen in figure 31. The full version of this is over 3000 lines long due to the vast amount of data stored within the font.

```
1    // 206 out of 201650 characters are shown
2
3    const char open_sans_regular[201650 + 1] =
4    "7])#######2Pc7('/###I),##d-LhLjKI##4%1S:'*]n8)K.v5*8_c)iZ
    ;99=$$$$c(m]4pKdp/(RdL<snZo'oI,hLNDnx4Uu/>8Q7oo^eFb3hB4JYc'
    Tx-3l_wgd2Tf._r+&sAqV,-G"":F8LD=5,n]A&aA+<gXG-<iobW&>$>QJ8Z
    .W$jg0Fv-o^(^JJnf4T"
5
```

Figure 31: Base85 encoded font

4.4.3 Menu Bar

Figure 32: Menu Bar

4.4.4 Load Model Window

The load model window is accessed through `File > Load Model` or by pressing the `Ctrl + L` shortcut. This window allows the user to load a 3D model at runtime by traversing the file system and selecting a model to load.

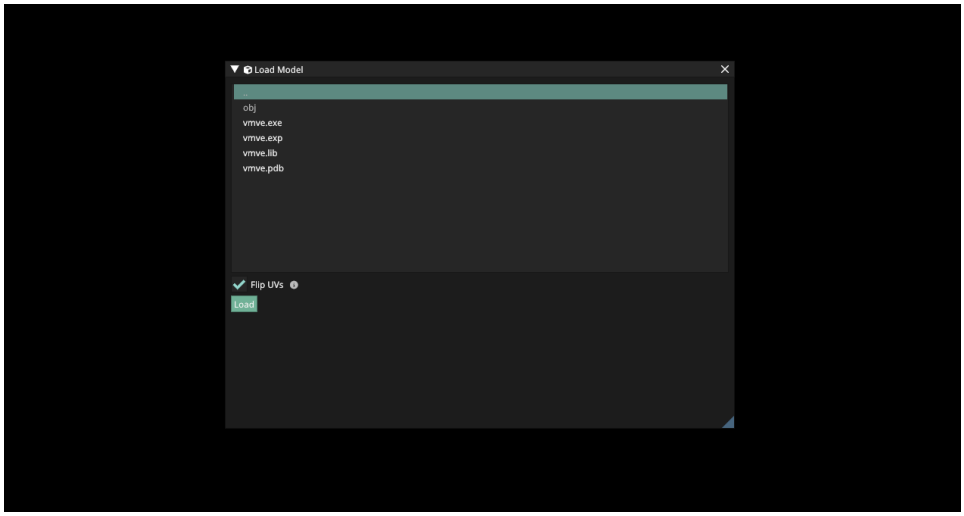

Figure 33: Load Model Window

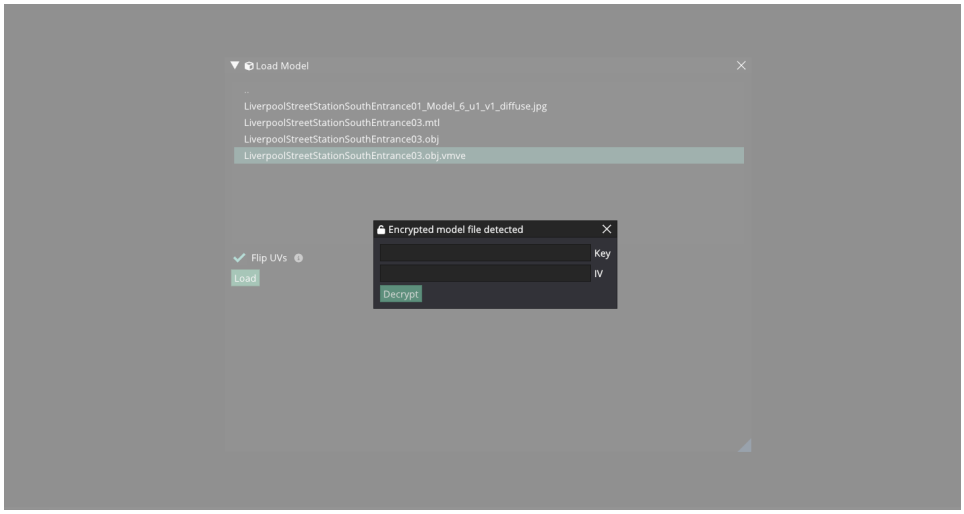

Figure 34: Load Model Encrypted Window

4.4.5 Export Model Window

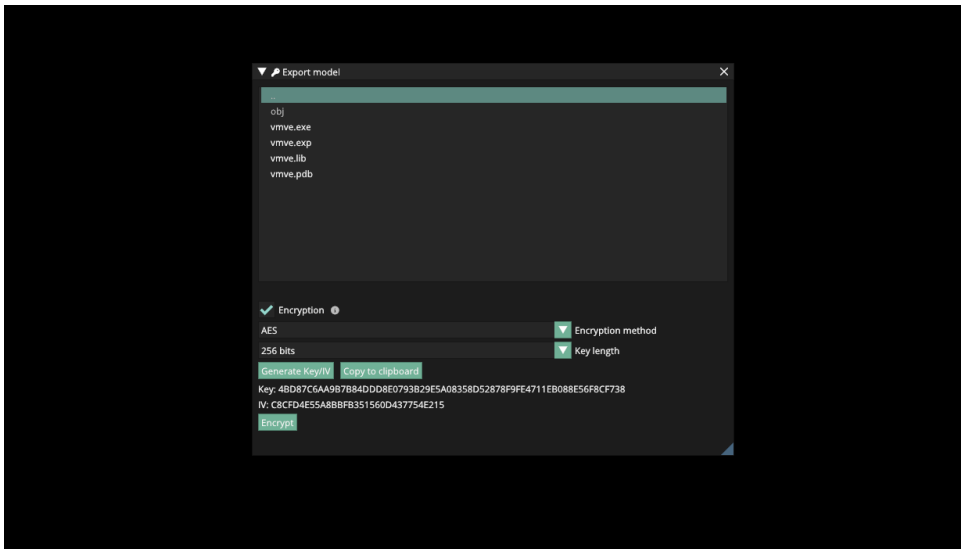

Figure 35: Export Model Window

4.4.6 Settings Window

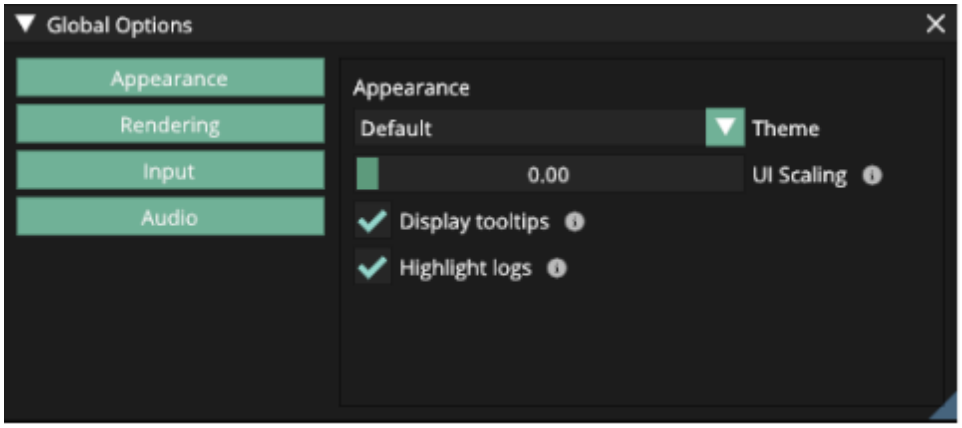

Figure 36: Settings Window

4.4.7 Object gizmo

An additional feature that is part of the user interface is the gizmo. This is a visualization of one of three operations is the main method of interaction that a user has with an object and is used to specify the exact location and orientation within the virtual environment. The operations are translation (moving), rotation and scaling and can be seen in figure 37 respectively.

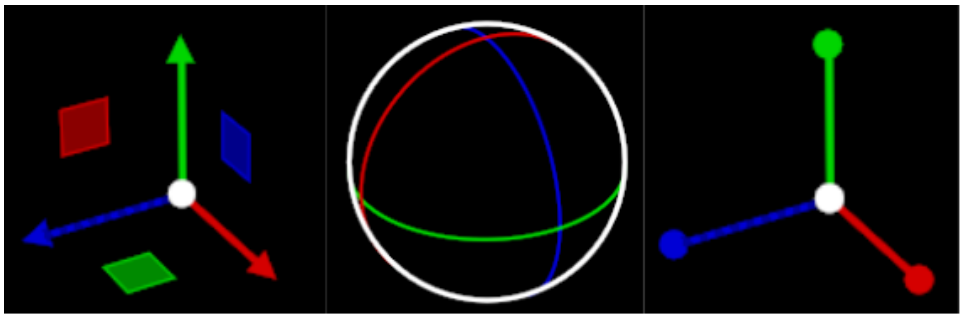

Figure 37: Gizmo operations

This functionality is implemented by taking the matrix transformation of an object and feeding that information to ImGuizmo[24]. It will then perform the necessary mathematical calculations to convert from the objects world space to screen space. In other words, no matter where in the virtual environment the object is located, it will always be shown relative to the users screen. A small code snippet can be seen below 38 where M represents the model in the MVP projection.

```
1     const auto& operation = static_cast<ImGuizmo::OPERATION>(
    guizmo_operation);
2     const auto& mode = ImGuizmo::MODE::WORLD;
3     ImGuizmo::SetDrawlist();
4     ImGuizmo::SetRect(x, y, viewport_x, viewport_y);
5     ImGuizmo::Manipulate(view, proj, operation, mode, m);
6
```

Figure 38: ImGuizmo world to screen space

4.5 Logging system

Due to the relative complexity of the underlying application and number of moving components a logging system was developed to provide a clear understanding of the current state of VMVE.

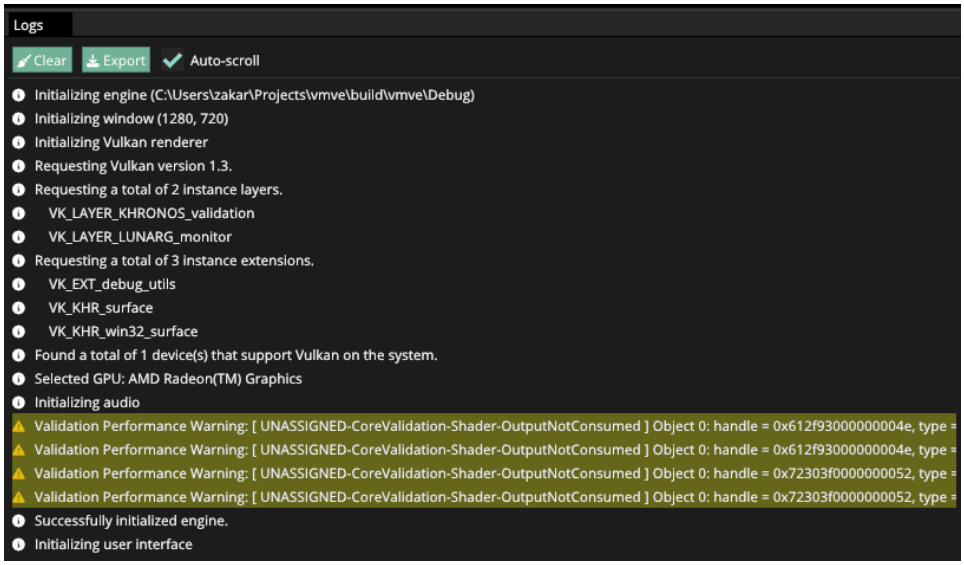

Figure 39: Logging window

Messages are stored in a fixed-size array that is preallocated. This ensures that there are no potential allocations occurring each time "print_log" is called. By default VMVE allows for a maximum of 10,000 log messages before the buffer becomes full. In the event that the buffer does become full, the first log message i.e the oldest one is overwritten when a new log message is created 40.

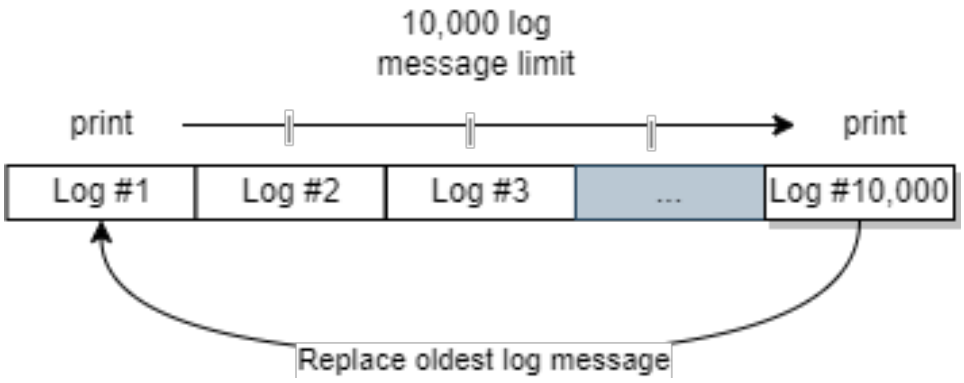

Figure 40: Logging FIFO system

4.6 Encryption and custom model format

The simplicity of the Crypto++ library results in a relatively straightforward function, shown in Figure 41 which shows how a model is encrypted using the AES algorithm.

```
1     std::string encrypt_aes(const std::string& text,
      encryption_keys& keys, int key_size)
2     {
3         std::string encrypted_text;
4
5         // Set key and IV to encryption mode
6         CryptoPP::CBC_Mode<CryptoPP::AES>::Encryption
      encryption;
7         encryption.SetKeyWithIV(
8             reinterpret_cast<const CryptoPP::byte*>(keys.key.
      data()),
9             key_size,
10            reinterpret_cast<const CryptoPP::byte*>(keys.iv.
      data())
11        );
12
13        // Encrypt text
14        CryptoPP::StringSource s(text, true, new CryptoPP::
      StreamTransformationFilter(encryption, new CryptoPP::
      StringSink(encrypted_text)));
15
16        return encrypted_text;
17    }
18
```

Figure 41: AES Encryption

4.6.1 Decryption validation

When a model is being decrypted the system must check and ensure that the key and iv being used match those originally used to encrypt it. Without this check in place, the system will not know what the "correct" key and iv are.

As outlined during the design stage, the file header contains the encrypted key and iv. When the user enters their specific key and iv as input parameters into the decryption dialog box as shown in figure 34 in section 4.4.4, VMVE will first decrypt the header section and check if the input values match. If successful, the application will then proceed to decrypt the data section and then load the file. If however, this check fails, then the application will throw an error and prevent the file from begin decrypted.

4.7 Distribution

- Release mode (optimized) - Windows only currently

4.7.1 Versions

VMVE follows the major, minor and patch system of versioning and is used as follows [Major].[Minor].[Patch]. The use of versions is important in differentiating between VMVE releases. Each new version of the application comes with new features as well as bug fixes that aim to improve the software over time. Due to the rapid development of new features and functionality, backward compatibility cannot be guaranteed. The current official release of VMVE is v0.0.4.

4.7.2 Website

Alongside the application, a website was created as a platform for easily distributing the application. It includes a features section that showcases and gives users a preview of the application through the use of images and videos. Futhermore, a download link is provided giving users an easy way of obtaining the executable without needing to understand the technicalities of GitHub as it is designed with programmers in mind.

In regards to downloads, there are two types. The first is an active development version which is regularly updated and acts as a beta release for versions that have not yet been officially released. The second is current and past VMVE versions going all the back to the first official release (v0.0.1).

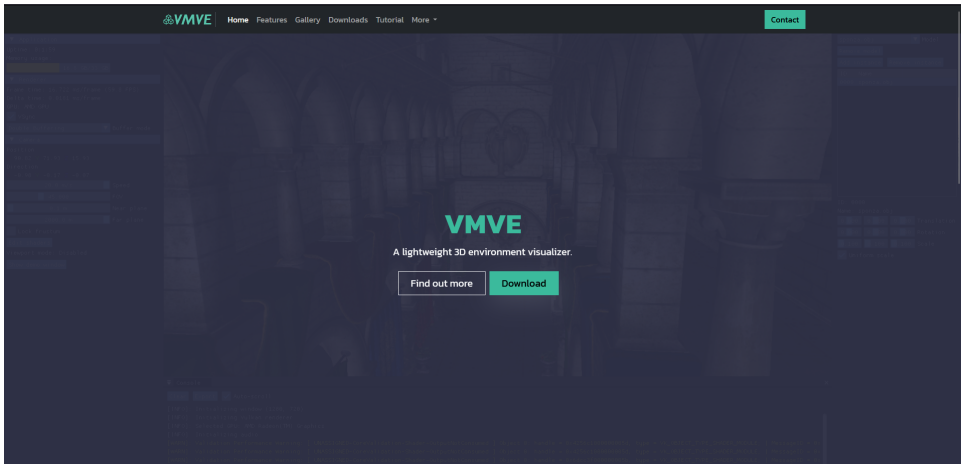

Figure 42: VMVE website

4.7.3 Example models

The VMVE website provides an additional download link to six different example models. These are provided by third parties (with appropriate credits) and are corrected to ensure they can be loaded into VMVE without any issues. The purpose of these example models is to allow users to easily test the application without needing to search the internet or create their own.

4.7.4 Documentation

In addition to a website, a documentation website was also created curtiсy of Read the Docs (https://readthedocs.org). The purpose of this site is to provide additional information about VMVE and also, provide an in-depth tutorial that aims to help users understand how to use the application.

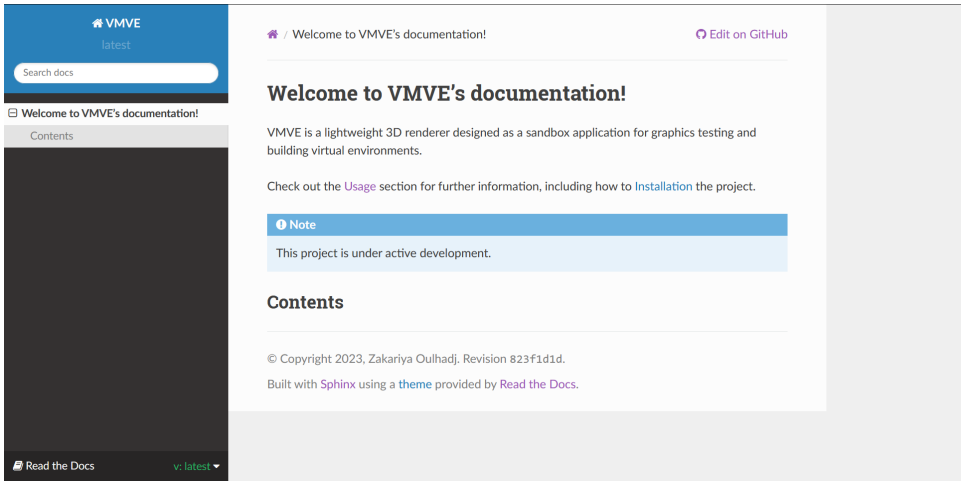

Figure 43: VMVE documentation

5 Evaluation

The completion of the project transitions into the evaluation stage where the project is evaluated against the original goals and requirements in order to judge if those requirements have been met. When evaluating the project there are two distinct categories. The first is self-evaluation and the other is user feedback.

5.1 Distribution

In hindsight, the most challenging aspect of the entire project was by far, distribution and more specifically ensuring that VMVE was able to run on all supported systems as expected. Throughout the applications development, there were points where VMVE would work on the development machine, however, throw an exception error on other systems. These inconsistencies paired with the lack of reproducibility made these issues quite difficult to debug. The majority of these inconsistencies between different systems occurred during the initialization stages. This would include creating the window, initializing the renderer, initializing the UI, creating the audio system etc.

Some examples of these inconsistencies include the application crashing when resizing the window, crashing if audio is disabled on a system as well as frame stuttering.

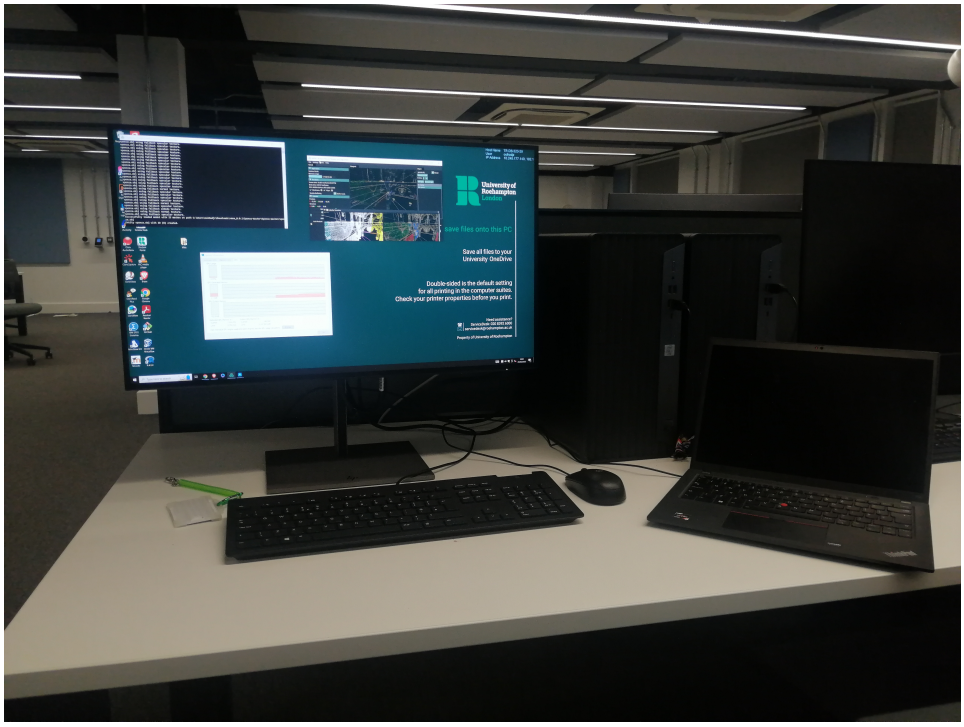

Figure 44: Working on distribution

During this part of development, the decision was made to design the logging system so that all messages in the buffer would be stored and written to disk before exiting. This allowed for unexpected crashes to be better understood as the `vmve_crash.txt` file could be analyzed.

5.2 Time Management

Given the complexity of this project, it was vital to effectively manage time during development and to ensure that all of the requirements of the project could be met on time. Failure to correctly manage this workload would result in key goals not being achieved and/or features not being implemented.

As mentioned in section 2.1.1, GitHub was chosen and has been a highly effective tool throughout development regarding management and distributing tasks. Over 50 different GitHub issues have been resolved such as small user

interface issues, implementation bugs and major structural changes.

- How effective was the use of project management tools during development.
- Key stages of the project - Initial core systems/graphics development stage
- System redesign - Encryption development stage

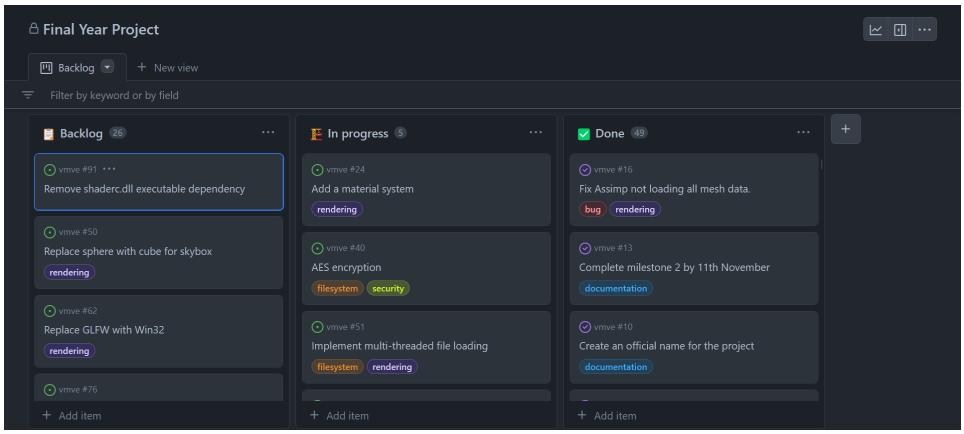

Figure 45: GitHub Issues Kanban

VMVE currently has 32 outstanding GitHub issues that have yet to be resolved. These issues are not considered critical and do not hinder the project in terms of meeting its original requirements. However, they are notable and should be resolved soon.

5.2.1 Hardware

These tests were performed on the main development machine used throughout the project which was a Thinkpad T14s Gen 3. The following list shown in figure 1 shows the laptop's specifications.

Component	Name
CPU	AMD Ryzen 7 Pro 6850U
GPU	Integrated
RAM	32GB DDR6
SSD	1TB [1ex]

Table 1: Development Machine

5.3 Performance

Measuring the performance of VMVE is another key aspect of evaluation that ensures the original goals and requirements have been met. Two main aspects can be measured which are compilation and runtime performance.

Compilation refers to the process of building the application in an offline setting. The speed at which the project is compiled directly affects the developer and subsequently the development of the project.

Runtime refers to the performance of the application while it is running and its effects on the end users.

5.3.1 Compilation Performance

A full clean build of VMVE takes an average of 53.260 seconds using Microsoft Visual Studio and the CL compiler. This includes the compilation of the source code and precompiled headers. As the codebase for VMVE grew in both size and complexity, ensuring that compilation times were reasonable was a vital aspect that had to be taken into consideration.

The main technique used for reducing compilation times was making use of a Precompiled Header (PCH). This is a header file (pch.h) that includes various header files such as those from the standard library as well as external dependencies that are not indented to change often. This is because a change to any of the included header files will require a full precompiled header rebuild. The compiler compiles this file once and is reused across compilations. This significantly reduces the amount of work the compiler needs to perform since it does not have to recompile the same sets of files needlessly each time.

This technique is done by including the “pch.h” header file at the top of each translation unit (.cpp file) and before any other header file. This ensures that the necessary types can be found for both header files and their corresponding source files.

Col1	Debug Full	Debug Partial	Release Full	Release Partial
1	6	87837	787	123
2	7	78	5415	123
3	545	778	7507	123
4	545	18744	7560	123
5	88	788	6344	123

Table 2: VMVE compilation performance

5.3.2 Runtime Performance

CPU timing is primarily measured by calculating the difference in time at two specific sections of the source code. Using C++ this can be easily achieved by using the "chrono" library that allows for the duration to be calculated in different time units such as nanoseconds, milliseconds, seconds and more. The timing will be measured using milliseconds as that is the typical timescale for applications such as VMVE.

Using the aforementioned "chrono" library, the startup time for VMVE averages around 300ms in the distribution version of the application which is calculated by taking away the time at the start of the initialization from the end of the initialization.

The Microsoft Visual Studio IDE allows us to further analyze this by taking advantage of the built-in profiler which can be accessed by going to `Debug > Performance Profiler`. Figure 46 shows the "flame graph" that takes a snapshot of the internal state of the application at runtime.

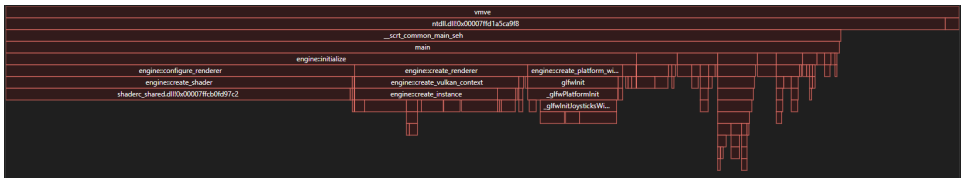

Figure 46: VMVE startup profiling

What this shows us is that a large chunk of the initialization is taken up by the `engine::configure_renderer` function and more specifically the `engine::create_shader` function which is responsible for compiling the application

shaders.

This can be taken a step further by expanding the hot code path which is what Visual Studio refers to as slow code. Figure 47 shows the exact function that takes around 36% of the total initialization time.

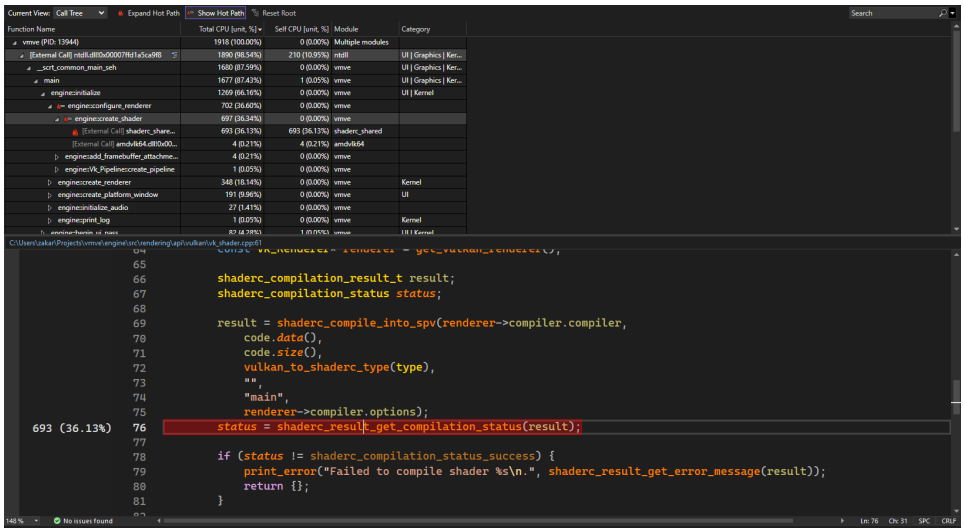

Figure 47: Profiling slow code path

The question one might ask is, “How can we improve this startup time?”. Currently, VMVE stores the shader source code as simple strings within a header file called `shader.h`. The main advantage of this is that shaders can be combined with the executable rather than be in separate files.

The downside to this, however, is that the shader source is not in a precompiled binary format, meaning that each time VMVE runs, it must recompile the shader files.

In the future, this will need to be addressed and the system redesigned in such a way that allows for shader files to be precompiled and bundled with the executable at the same time eliminating the need to compile shader files and as a result improving the startup time.

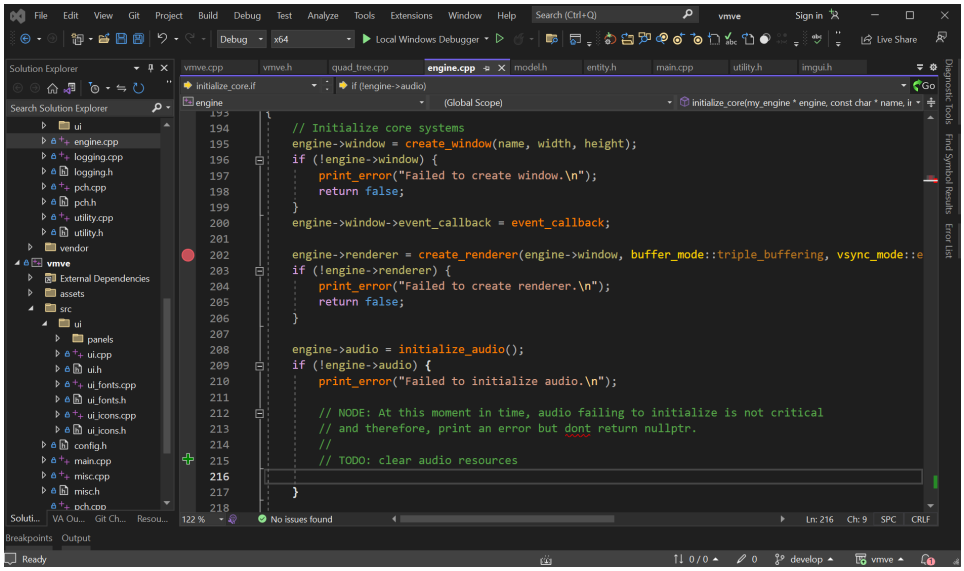

Figure 48: Microsoft Visual Studio

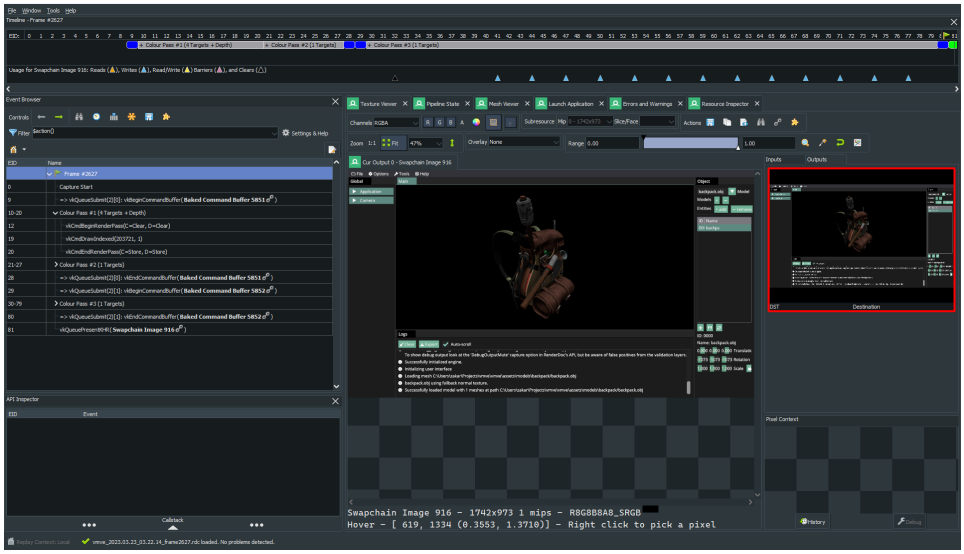

Figure 49: RenderDoc

Metric	Debug	Release	Col3
Initialization	6	87837	787
Update Loop	7	78	5415
Shutdown	545	778	7507

Table 3: VMVE runtime performance

5.4 User Feedback

Having conducted the evaluation, it is equally important to obtain feedback from the stakeholders such as users. The users that tested the application were predominantly other students. The process of obtaining this feedback was separated into two stages.

The first was to measure how intuitive the user interface is which was achieved by giving a user a set of instructions as seen in figure 50 [file]. A score would be given for each task based on how easily the user was able to complete it.

User	Availability	Keys	Load a model	Add an instance	Enter the viewport	Enter instance edit mode	View surface normals	Encrypt a model	Load encrypted model	Change theme	Export logs to file
Tom	Yes	Completed									
Adam	Yes	Required assistance									
Bogdan	Yes	Not completed									
Taylor	Yes										
Danny	Yes										
Trevor	Yes										
Savergery	Yes										

Figure 50: User feedback instructions

The second stage would obtain direct feedback from the users by presenting a series of questions that would assess VMVEs usability, performance and overall user experience. This data was recorded using Google Forms and further analyzed to produce different sets of visualizations.

6 Related Work

Computer graphics is a large field - Others who have done the same. - How good is my work compared others given the time constraints

7 Reflection

One of the major downsides that this project suffers from is the projects time constraints. Given the relative complexity of the project many of the features implemented in VMVE are quite rudimentary and serves as a basic prototype that showcases the underlying technology and the potential future work that can be undertaken.

- Learn a lot in terms of solving problems - Patience Could have planned better by designing systems before implementing them.

7.1 Choice of rendering API

This project was built on top of the Vulkan which as previously discussed is a low-level GPU API. One of the major downsides to having chosen Vulkan as the API of choice for this project was the significant amount of time had to be spent implementing the various rendering features due to how verbose and technical the API is. The use of the API is only really beneficial if a developer wants to make full use of the additional performance through the use of multithreading, multiple command buffers and very specific memory allocation requirements.

If given the choice, I would rewrite the renderer, using a far simpler API such as DirectX11 or OpenGL as they provide same functionality but with significantly less work required. This would allow me to focus and spend more time implementing the many other features required in such an application.

7.2 Development logs

At key stages of the project's development, development logs i.e. videos were recorded showcasing the state of development at that specific point in time. All videos were uploaded and are hosted on the Zakariya Oulhadj YouTube channel https://www.youtube.com/@ZOulhadj

8 Future Work

Going forward there are various features that are currently being evaluated as potentially implemented in the near future. These features aim to greatly increase VMVEs usability and provide a whole host of new features.

8.1 Rendering

8.1.1 Multiple rendering APIs

VMVE currently only supports one rendering API which is Vulkan. As discussed in the technology review section, Vulkan is officially supported on Windows, Linux and macOS (through MoltenVK). However, to support additional operating systems as well as hardware that does not support Vulkan, more rendering APIs should be supported including DirectX12 and previous generation APIs such as OpenGL and DirectX11 for increased compatibility.

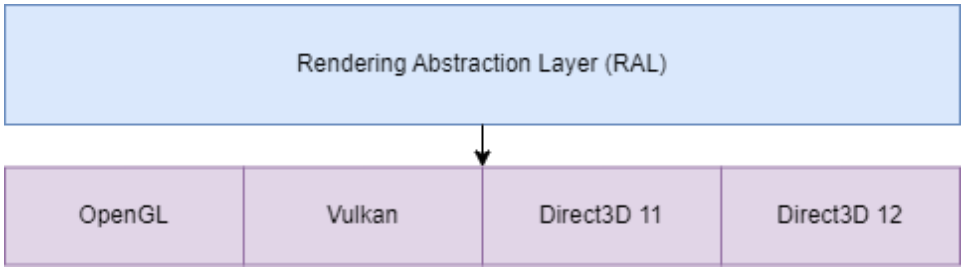

Figure 51: Rendering API Abstraction

8.1.2 Frustum Culling

Currently, VMVE sends all object data to the GPU to be processed and rendered. The GPU then subsequently traverses each vertex to figure out if it needs to be "discarded". This process is part of the graphics pipeline and occurs for each vertex. As the complexity of both the scene and the objects themselves increases, this starts to become a GPU intensive task and results in increased GPU usage and lower performance.

To solve this, frustum culling must be implemented which is a rendering optimization in which objects not visible from the "cameras point of view" are discarded completely and not sent to the GPU entirely. The term "frustum"

refers to the camera projection frustum which can be seen in figure 52 and "Culling" simply means discarding.

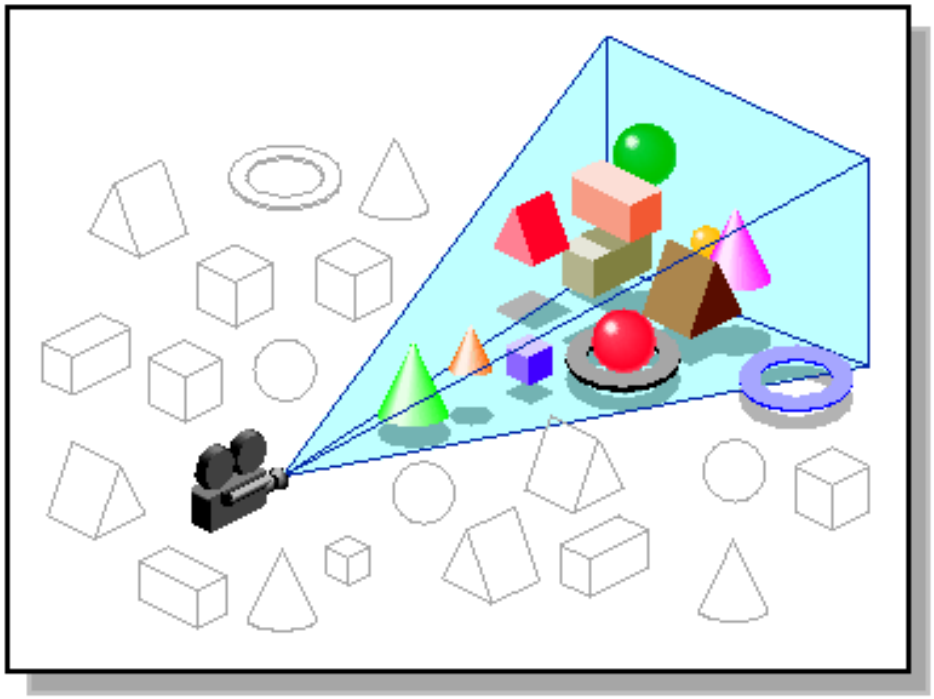

Figure 52: Frustum Culling[25]

This technique significantly improves performance as each object in the world is contained within a "bounding box" which is often a cube or a sphere 53. Then for each object, a check is performed against the bounded box instead of the actual vertices. This allow for in the best case a single check or at its worst 8 checks per object. This is far better than needing to perform thousands of checks for all the vertices of an object.

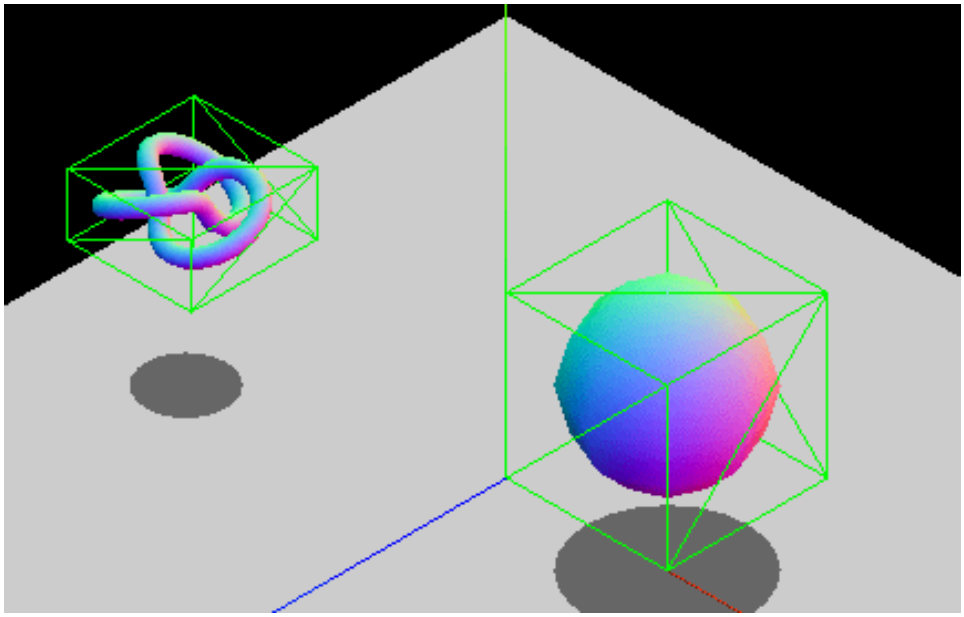

Figure 53: Bounding Boxes[26]

8.1.3 Spatial Acceleration Structures

VMVE only supports loading basic models however, in the future many other features need to be implemented. One such feature is large scale terrain as seen in figure 54.

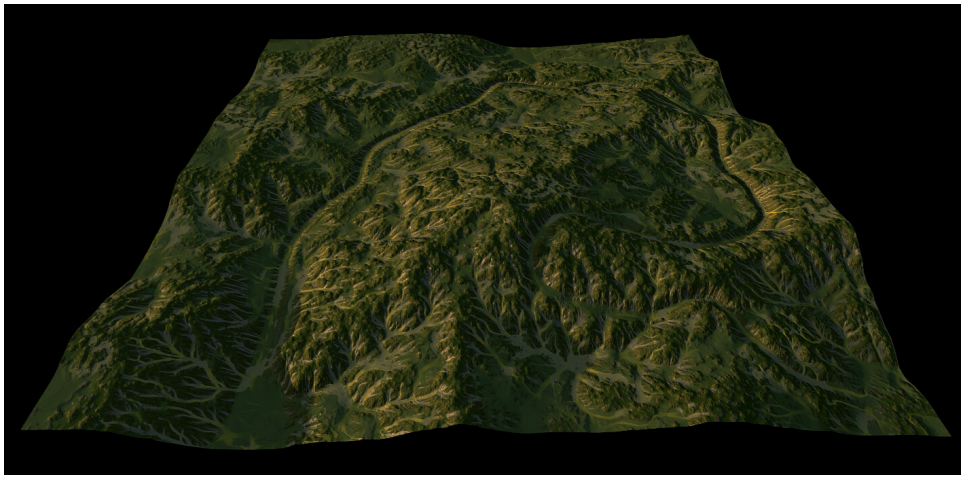

Figure 54: Large scale terrain

Currently, VMVE would struggle to render such objects due to their complexity in terms of the number of vertices required. For example, an area of 20x20km with a resolution of 1 meter would require 400 million vertices. Each vertex, if packed efficiently could be 32 bytes each that includes positions (12 bytes), normals (12 bytes) and texture coordinates (8 bytes). In terms of memory, this would require 12.8GB of data just for the terrain.

A common solution to this, is implementing a type of spatial acceleration structure also known as Level of Detail (LOD). These structures are designed as the name suggests, to increase the speed for algorithms in the spatial domain including images and environments.

A quadtree is a type of spatial acceleration structure that stores a hierarchical collection of nodes that each represent a 2D area (and an octree in the case of 3D). Figure 55 visualizes a quadtree that shows the resolution of each tile and how there are fewer tiles the further away from the camera they are.

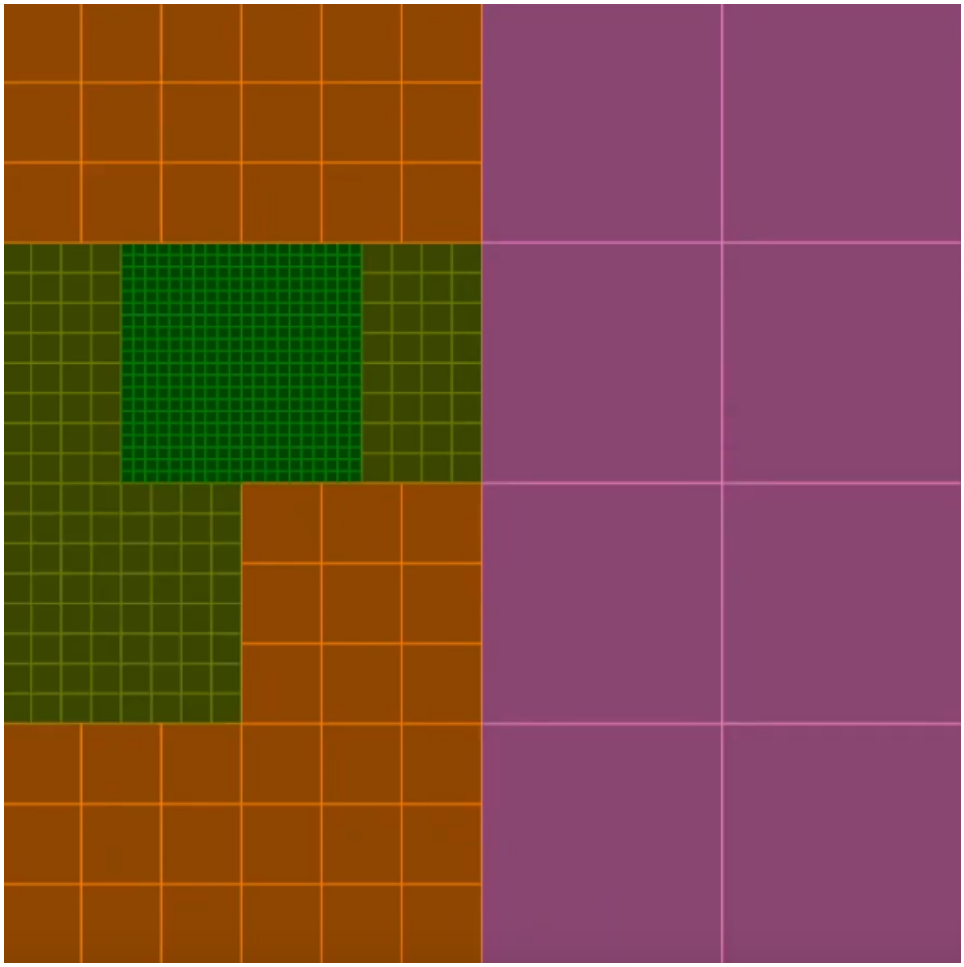

Figure 55: Quad Tree visualization

The benefit of using this technique is that the number of vertices is significantly reduced saving on performance and memory usage.

8.1.4 glTF Support

Currently, the engine only really supports the VMVE and OBJ model formats for importing. The main issue with the OBJ file format is its size since the data is stored in a text format which subsequently increases the assets file size. glTF [27] is a new model format by The Khronos Group that

has many useful features such as compression, binary representation, scene hierarchy and more.

8.2 Cross Platform

In contrast to the project's initial proposal which was the support multiple Operating Systems (OSs), Windows is the only OS that VMVE officially supports due to the time constraints for this project.

Going beyond the final deadline, VMVE should add cross-platform support which would greatly benefit the application as it would increase flexibility in terms of which OSs its able to run on and subsequently, increase the potential user base. Implementing such functionality would require significant changes to the underlying architecture of the application such as implementing opaque types for structures that have different implementations depending on the OS being used. Additionally, a different build system would have to be used that is cross-platform such as CMake[28] or Premake[29].

8.3 Networking Support

VMVE is a virtual environment editor and as such can be significantly improved by adding support for networking. The idea is that there would be a server running which would keep track of the virtual world and multiple clients via VMVE would be able to connect and interact with the environment simultaneously.

This would allow for greater efficiency and speed up various tasks. For example, multiple users could work together to construct a virtual environment in preparation for scientific research which would otherwise have taken much longer had only one user worked on it.

8.4 File Format/Encryption

VMVE supports both the AES encryption standard by default. The application should also include additional encryption algorithms such as Diffie-Helm and RC6. This would provide users with greater flexibility in terms of how their assets are encrypted as well as varying levels of encryption strength/methods based on the user's encryption requirements.

Another feature that will need to be added to the application is related to the custom model format and the ability to store and encrypt other types of data. Currently, the encrypted data stores just the vertices and their

corresponding information. The downside to this is that assets such as textures are still required to be located alongside the ".vmve" file since the format does not support textures.

In the future, VMVE should be able to store textures so that users can be left with a single asset file that includes all the information, similar to how the glTF model format supports the ".glb" binary file. This would significantly increase the user's experience since assets files would become easier to manage.

8.4.1 Runtime memory protection

One of the major flaws present is the lack of security that prevents a user from maliciously altering the memory of the application during runtime so that encrypted models can be accessed without the need for a valid key or iv.

This can be achieved using applications such as Cheat Engine which provides users with a set of tools for analyzing memory addresses, offsets and values that can also be manipulated [30].

This will need to be addressed in future versions of VMVE so that digital assets remain secure for all users.

8.5 Version Control

In regards to the VCS architecture, a potential change that can be done is to add a third branch called "Beta". The purpose of this branch would be to release relatively stable yet unfinished features to users. This would allow users to use new features and test them before an official version is released. Overall, this would reduce the number of bugs in final versions and be beneficial to users who are keen on using recently developed features. Figure 56 shows how the VCS architecture could look like with the addition of a "Beta" branch.

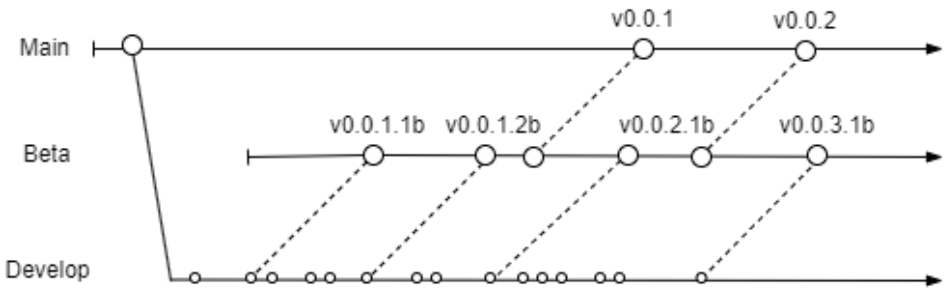

Figure 56: Potential future git branch design

8.6 Reduce library dependencies

VMVE makes use of several libraries which allows specific functionality to be added quickly. This is ideal given the project's time constraints and requirements. However, going forward the aim will be to reduce the number of external libraries being used.

This will provide various benefits such as less reliance on external code, faster compilation times as only functionality that is required will be implemented and generally finer control of the code that VMVE is built on top of.

8.7 Accessibility

Another aspect that needs to be addressed soon is improved accessibility. The application includes some features that address this such as icons, a tutorial, documentation as well as additional information. However, some of the features that VMVE does not include is taking into account color blindness and text-to-speech. The reason for this is that these are more complex topics and required additional time and research to ensure a proper implementation is released.

Another task that must be considered is localization. VMVE only supports the English language and has no way of adapting to other languages. Therefore, the internal structure of the application must be redesigned to allow for other languages to be added and thus, improving accessibility.

9 Conclusion

VMVE has been designed to be a platform in which users are provided easy to use 3D graphics tools without needing to know the complex details that come with that. This allows users to worry less about technicalities and more about their specific needs and requirements when using VMVE.

This has been a long journey that has presented me with various challenges throughout the development of VMVE.

10 Documents

Milestone 1 Milestone 2 Milestone 3 Meetings

Glossary

CL The Microsoft C compiler and linker. 52

Vulkan Low-level GPU application programmable interface. 10, 13, 17, 27, 28, 32, 33, 58, 73

Acronyms

API Application Programmable Interface. 9, 10, 11, 13, 17, 58, 73

CPU Central Processing Unit. 2, 8, 9, 28, 53

GPU Graphics Processing Unit. 2, 9, 10, 11, 26, 30, 33, 58, 59, 73

IDE Integrated Development Environment. 8, 53

IP Intellectual Property. 4

LOD Level of Detail. 62

MSVC Microsoft Visual Compiler. 8

OS Operating System. 64

PCH Precompiled Header. 52

STL C++ Standard Template Library. 9

UI User Interface. 2

VCS Version Control System. 6, 14, 65

VMVE Vulkan Model Viewer and Exporter. 1, 2, 3, 4, 5, 6, 10, 11, 12, 13, 14, 16, 17, 19, 20, 21, 24, 25, 26, 27, 30, 31, 33, 35, 36, 39, 44, 46, 47, 48, 49, 51, 52, 53, 54, 56, 58, 59, 61, 62, 63, 64, 65, 66, 67, 73

VRAM Video Random Access Memory. 30

11 References

[1] "Unreal engine," https://www.unrealengine.com/en-US/.

[2] "Unity," https://unity.com/.

[3] "Renderman," https://renderman.pixar.com/.

[4] Keza MacDonald, Keith Stuart and Alex Hern, https://www.theguardian.com/games/2022/sep/19/grand-theft-auto-6-leak-who-hacked-rockstar-and-what-was-stolen, 9 2022.

[5] "The mit license," https://opensource.org/license/mit/.

[6] "Git version control system," https://git-scm.com/.

[7] "Microsoft visual studio 2022," https://visualstudio.microsoft.com/.

[8] "Visual studio assist x," https://www.wholetomato.com/.

[9] "Renderdoc," https://renderdoc.org/.

[10] "Amd radeon gpu profiler," https://gpuopen.com/rgp/.

[11] "Imgui," https://github.com/ocornut/imgui/.

[12] "Crypto++ library," https://www.cryptopp.com/.

[13] "cereal - a c++11 library for serialization," https://uscilab.github.io/cereal/.

[14] "C++ core guidelines," https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[15] "K and r indentation style."

[16] "Vulkan loader," https://gpuopen.com/learn/reducing-vulkan-api-call-overhead/.

[17] Wikipedia contributors, "Moire pattern — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Moir%C3%A9_pattern, 2023, [Online; accessed 21-April-2023].

[18] "Texture mipmapping," https://learnopengl.com/Getting-started/Textures.

[19] Wikipedia contributors, "3d projection — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=3D_projection&oldid=1148285432, 2023, [Online; accessed 6-April-2023].

[20] "Coordinate systems," https://learnopengl.com/Getting-started/Coordinate-Systems.

[21] "Model view projection," https://jsantell.com/model-view-projection/.

[22] Bui Tuong Phong, "Illumination for Computer Generated Pictures," 1975, [Online; accessed 12-December-2022].

[23] "Font awesome," https://fontawesome.com/.

[24] "Imguizmo," https://github.com/CedricGuillemet/ImGuizmo.

[25] "High-level strategic tools for fast rendering," https://techpubs.jurassic.nl/manuals/nt/developer/Optimizer_PG/sgi_html/ch05.html.

[26] "Mdn web docs, 3d collision detection," https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection.

[27] "gltf runtime 3d asset delivery," https://www.khronos.org/gltf/.

[28] "Cmake, open source and cross platform build system," https://cmake.org/.

[29] "Premake - powerfully simple build configuration," https://premake.github.io/.

[30] "Cheat engine," https://www.cheatengine.org/.

[31] AMD, "RDNA architecture - The all new Radeon gaming architecture powering "Navi"," 2019, [Online; accessed 23-January-2023].

[32] Henry C Lin and Andrew Burnes, "How Ada advances the science of graphics with DLSS 3." 2022, [Online; accessed 2-May-2023].

12 Appendices

12.1 Rendering API history

One of the core aspects of VMVE is making use of the underlying hardware and more specifically the GPU (Graphics Processing Unit) which will be primarily used for rendering. Taking advantage of the GPUs hardware capabilities requires low-level access to the hardware and is not as straightforward as one would hope. To understand why, we must first understand how Graphics Processing Units function.

There are different types of GPUs such as dedicated or onboard, different architectures including AMDs RDNA[31] or NVIDIAs ADA[32] as well as various capabilities that differ between hardware vendors. An application attempting to target GPUs would need to consider this including having access to the GPU drivers (Low-level software that allows a specific piece of hardware to function). Oftentimes, drivers are considered trade secrets that companies do not want freely available. Given the complexity and variations of modern GPUs this is simply not feasible.

Companies from across different industries solved this issue by creating an open, non-profit consortium in the early 2000s called The Khronos Group. This organization develops, publishes and maintains standards for different areas but most notably for 3D Graphics and Computation. Companies follow these standards when developing software allowing for interoperability across hardware. As of 2023, The Khronos Group actively maintains 16 different standards. Out of all those standards, there are two which are the most suitable for VMVE, OpenGL and Vulkan.

OpenGL and Vulkan are two types of rendering APIs that are designed based on The Khronos Group specifications. When attempting to provide graphics support for a particular GPU, hardware vendors follow the OpenGL and/or Vulkan specifications when implementing their drivers. For applications, a API is provided allowing for direct control of the GPU.

VMVE could support both OpenGL and Vulkan however, due to the vast amount of work required to accomplish this as well as the project's time constraints this is simply not feasible. Instead, both need to be evaluated to choose one.

12.2 Camera transformation code

```
1     // Construct M (model)
2     glm::mat4 m = glm::mat4(1.0f);
3     m = glm::translate(m, world_position);
4     m = glm::rotate(m, radians, rotation_axis);
5     m = glm::scale(m, scale_amount);
6
7     // Construct V (view)
8     glm::mat4 v = glm::lookAt(camera_position, camera_direction
      , camera_up);
9
10    // Construct P (projection)
11    glm::mat4 p = glm::perspective(field_of_view, aspect_ratio,
       near, far);
12
13    // Construct MVP
14    glm::mat4 mvp = p * v * m; // Order is important
15
16
```

Figure 57: Complete MVP example