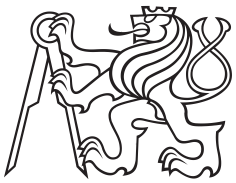
Bachelor Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

A Platform for Virtual Reality Applications

Jakub Hlusička

Supervisor: Ing. David Sedláček, Ph.D.
January 2020

Acknowledgements

I would like to express my gratitude to my supervisor Ing. David Sedláček, Ph.D. for the opportunity to work on a topic I find genuine interest in, and for his useful advice.

I would also like to thank my family for the support they provided me with during my studies and the completion of this thesis.

I am very thankful for being in a privileged enough position to be able to work on personally motivated projects, such as the one covered in this thesis. One of the reasons why I chose this topic for my thesis, is that I believe that free communication is the basis of a free society, and I hope it will benefit those who were not as fortunate as me.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Prague, 13. 5. 2020 _____

Abstract

The World Wide Web has revolutionized the way humanity shares and accesses information. Despite its success, the World Wide Web has numerous deeply rooted shortcomings. I investigate these shortcomings and come up with solutions to creating a platform based on consumer VR technology. This part of the work focuses on the development of the base rendering and virtualization engine for applications utilizing the platform.

Keywords: virtual reality, real time rendering, rasterization, communication, virtualization, glTF, WebAssembly, Rust

Supervisor: Ing. David Sedláček, Ph.D. Department of Computer Graphics and Interaction

Abstrakt

Vynález webu (World Wide Web) vyvolal převrat ve způsobu, jakým lidstvo sdílí informace. Přes jeho úspěch obsahuje web mnoho hluboce zakořeněných nedostatků. V této práci zkoumám tyto nedostatky a vymýšlím řešení pro vytvoření platformy založené na technologiích VR. Tato část práce je soustředěna na vývoj základového vykreslovacího a virtualizačního enginu pro aplikace této platformy.

Klíčová slova: virtuální realita, vykreslování v reálném čase, rasterizace, komunikace, virtualizace, glTF, WebAssembly, Rust

Překlad názvu: Platforma pro aplikace virtuální reality

Contents

Figures

Chapter 1

Introduction and Motivation

Virtual Reality (VR) has been an active area of research since the 20th century. Arguably, one of the most immersive devices related to Virtual Reality, is the head-mounted display (HMD). This device is used to display stereoscopic video to the user, significantly improving the immersion in Virtual Reality.

With recent developments, VR devices, such as the HMD, are becoming more affordable to the general public than ever before. However, various vendors of VR devices have engaged in tactics such as making software exclusive to their platform in order to gain market advantage, despite there being no fundamental limitations to making such software available on a wider range of functionally equivalent devices.

I envision an application platform able to bridge the gap between VR devices of various vendors, one inspired by the ubiquitous web browser. In this chapter, I cover the motivations behind the project and I distill those motivations into goals for the implementation of my solution.

1.1 Centralized Services

1.1.1 Encryption

In the news, there have been many articles about well known companies getting breached by hackers. Despite these recurrent occurrences, users still place their trust into these companies, as many of these companies dominate their targeted market sectors. There are simply no worthy competitors to some of these companies, and even if there were, they would suffer from the exact same shortcomings, which lead to their IT infrastructure getting breached. The main problem is, that there is a single point of failure. If a company gets hacked, the hacker may get access to the private data of all of the company's users. This is caused by the fact, that the services these companies run are built on centralized architectures.

Take a company, that provides services to their users, in a personalized way. Facebook[1], for example, one of the largest, most popular social media

[1] `https://facebook.com/`

platforms. In order for a company to be able to deliver their services, it needs to be able to store information about its users. For Facebook, that might be the list of friends a user has, so that the user may use Facebook to exchange messages with their friends. The users essentially share valuable information with the company, and in turn, they entrust the company to handle the data carefully. However, what happens, if the company's services get breached by hackers? The data we entrust to these companies gets stolen and very likely abused by parties we do not trust.

The obvious solution to personal data getting stolen is to make sure it is encrypted. How can we encrypt the users' data, so that it cannot be stolen? We cannot entrust the company to encrypt the personal data for us, because they would have to manage the encryption key pair themselves, which means, that the encryption key could itself be stolen, and the personal data could be decrypted by whoever stole it.

The key (no pun intended) lies in end-to-end encryption (E2EE). At no point in time does the company get access to the encryption key pair. Whenever the users' personal data needs to be stored, the data is encrypted before being sent to the company's servers. When the data needs to be accessed by the user, it is sent back to the user in its encrypted form, to be decrypted by the user. The key pair never leaves the user's device and the company (nor the attackers) never get access to the unencrypted personal data.

However, in most cases, as it is with the most popular services such as those provided by Facebook, there exists a conflict of interest: data is very valuable from the company's perspective. It is often used to make executive decisions on the future direction of the company, thus having overwhelming impact on the company's future success and earnings. The fact, that the companies need to keep innovating their services is indisputable, otherwise they would get steamrolled by the competition. However it would be very naive to think, that new technologies are perfectly secure and not prone to breaches.

If we assume, that these companies do not encrypt their users' data in an effective way (by using E2EE, for example), it is impossible for these companies to ensure the security of the data, while keeping up the innovation. These companies end up in a perpetual chase after security, which is fundamentally never going to be sufficient, all just to keep the access to the data provided by their users.

1.1.2 Availability

It is worth mentioning, that there do exist centralized services, which use effective encryption. Examples include the messaging applications Signal[2] and Keybase[3]. The success of these companies proves, that there do exist monetization schemes compatible with effective personal data encryption.

[2] `https://signal.org/`
[3] `https://keybase.io/`

Nevertheless, there is still a significant drawback to relying on these services. The users of these services have to assume, that the service will remain available in the timespan they plan on using the service. If something were to happen to the service, the users' personal data would become inaccessible, possibly forever. The users still end up relying on the fact, that the centralized service does not delete the data the user stored there.

If the company the user relies on goes bankrupt, unless the user has made local backups, they will never be able to access their data anymore.

1.1.3 Decentralization as a Solution

Assuming the goal is to provide services to the user, we can take existing centralized services and redesign them in a way, which removes all single points of failure, effectively making them decentralized. However, due to the nature of decentralized services, it may be very difficult to come up with a working monetization scheme. This may not be a problem, if such decentralized service is created for non-profit purposes. Nonetheless, even non-profit efforts require funding, making the development of decentralized services difficult.

1.2 The Lack of a Universal Platform for VR Applications

VR headsets and peripherals have finally matured enough to be affordable by the general public and serve primarily as entertainment media. However, I believe that VR devices have the untapped potential to be used by the general public not just as a source of entertainment, but as devices for communication, in the broad sense of the word.

Let me present the idea of a universal platform; such a platform would work in the following way. The user would turn on their VR device along with the peripherals, put on the head-mounted display (HMD), launch the platform application (a kind of VR application browser), then choose which VR applications to launch. These applications would manifest themselves in the shared virtual space the user finds themselves in. The user would be free to interact with these coexisting applications, and by interacting with them, other applications could be downloaded, which the user could seamlessly launch and use. The user would be able to make use of multiple applications simultaneously. This platform would provide an uninterrupted, immersive experience, as the user would not have to take off their VR headset to download new applications or switch between them.

1.2.1 Web Browsers

The World Wide Web, in its current state, does not harness the potential of VR devices in a seamless way. While it is true, that it is possible to browse

some web content with a VR device, such content must have been purposefully created with the intent to be viewed using a VR device, while the overwhelming majority of web content remains unsuitable to be viewed via this method. While there are efforts to integrate VR support into web browsers, the content that has traditionally been distributed via web browsers differs fundamentally from the content, which would be viewed using VR devices. New APIs are being added to web browsers, adding onto their complexity, which is very high already. Moreover, developers hoping to create VR content viewable in the web browser are encumbered with the requirement of non-VR boilerplate for compatibility reasons. A VR developer should not be required to have the skills of a web developer. Nevertheless, it is undeniable, how much momentum the World Wide Web currently has, and that efforts to get VR content to the web browser will continue.

1.2.2 Metaverses

There exist many environments, some more popular than others, which aim to aid communication through VR devices. Those, which focus on the social aspect, have been dubbed as *Metaverses*. The two, arguably among the most popular and therefore relevant, are VRChat[4] and NeosVR[5]. Both are being developed in the Unity[6] engine.

NeosVR makes a successful attempt at creating a metaverse with a built-in visual scripting language. Built on the Unity game engine, it makes use of a set of sophisticated rendering techniques, which make it appealing for developers. While the visual scripting language makes scripting easily accessible, as users can develop without having to take off the HMD, this approach has many drawbacks. The developer is required to learn the visual scripting language, which is unique to NeosVR. Creating a new programming or scripting language can become very demanding, as optimization requirements rise with the development of the platform; after all, programming languages are a very active area of research, so no wonder it would be difficult to get right. Moreover, dealing with code reuse becomes problematic, as novel mechanisms for package management, suitable for the development environment of a visual scripting language, need to be invented. While, without any doubt, well crafted metaverses can become successful communication platforms, their potential is limited by how much the scripting language is integrated with the platform and therefore how much control the developer gains by using the scripting language.

1.2.3 Virtualization as a Solution

Let us take a different approach to creating a platform for VR applications. We are not interested in creating a game, in which the user can use a scripting language to influence the world around them, after all. What we are truly

[4] `https://vrchat.com/`
[5] `https://neosvr.com/`
[6] `https://unity.com/`

interested in, is creating an environment, which itself is built by its users; a platform, which only provides the protocols and the APIs — in other words, the glue, which makes these applications interplay. With this idea in mind, it becomes clear, that we cannot cherry-pick functionalities to provide to the application developers. It is, after all, our goal, to provide the developers with as much freedom of expression as possible, to let them create any kind of VR application, without restrictions on which capabilities of the platform become accessible through the scripting language.

I believe, that the way this can be achieved, is by building a platform with the utmost focus on the VR applications themselves. The experience provided to the user should be provided first and foremost by the applications running on the platform, not the platform itself. By shifting the source of the functionality from the platform to the applications, the platform developers can focus on providing the most basic low-level APIs to the application developers. The application developers, in turn, get the freedom to develop complex systems using these low-level APIs (instead of already being handed the limited pre-built systems of the engine), which can be adjusted as they wish, to suit their needs.

The idea seems simple enough. The platform provides low-level APIs, and applications are made, which make use of those APIs. Nonetheless, there is a concern to be had about the users' security. How can we ensure, that the applications do not act in a malicious way toward the users? The applications could be made by anyone, after all. Even by people with the malicious intent of stealing the user's personal data stored on their computer, for example. This pitfall can be avoided by the use of virtualization, which is a fancy word for making sure, that whatever code we run, cannot interact with the rest of the operating system, unless we grant it permission to do so. This is a very powerful idea, which has been used by web browsers to display interactive content to the user, without compromising the user's security. Not only is virtualization a great tool for ensuring security, it can often be implemented in a cross-platform way, meaning an application can run in a virtual environment on any operating system, without the need to recompile the application for that particular operating system. These properties are ideal for our platform.

However, it should be noted, that virtualizing the applications is not enough to ensure the safety of the users. Careful attention must be put towards ensuring the safety of the low-level API provided by our platform, so that it cannot be abused in a malicious way.

1.3 Introduction Summary

During the introduction, we set our goal to develop a novel communication platform, addressing issues of the current, most widely used communication media. We also explored the idea of a platform, which would provide primary support for virtual reality devices. We have come to the conclusion, that it would be desirable, for such platform, to have the following properties.

1. The user should be protected from getting their sensitive personal information stolen.
2. The platform should encourage developers to create decentralized applications, rather than centralized applications, so as to improve security and availability of the applications.
3. The platform should be primarily built for VR devices.
4. The platform should support and encourage multitasking.
5. The platform should provide functionality for seamless plug-and-play installation of applications.
6. The platform should provide low-level APIs to the applications, so as to empower application developers to iterate on complex systems they build on top of the APIs.
7. It should be possible to write the applications in a well known programming language, so as to take advantage of the developments of the language, such as the optimization capabilities of its compiler.
8. The platform should execute the applications in a virtual environment.

Chapter 2
Implementation

In this chapter, I justify the chosen approach to fulfilling the goals specified in the introductory chapter.

2.1 Platform Design

Our platform should be able to facilitate rendering capabilities to the applications. These capabilities should ideally be as flexible as possible, so that application developers can harness their functionality to create complex systems. If possible, they should not impose biased restrictions upon the developers. Nevertheless, providing a safe graphics API and ensuring its safe implementation is no easy task. For this reason, I decided to simplify the first iteration of the implementation of our platform. This simplification lies in the decision not to expose the rendering API directly, but instead to expose an API to load models in a certain format and render the loaded models in the scene.

The decision to simplify rendering by only allowing applications to render models reveals a nice property: the platform gains access to the mesh of every object in the scene. This fact makes it possible to provide applications with a unified method to ray trace the scene for rendered geometry.

In this work, I do not focus on the networking part of the project. Our main task is going to be creating the base rendering engine, a way to execute applications in a virtual environment, and providing those applications with APIs to communicate with the platform.

In the end, we will focus on creating a sample application for our platform. The sample application will display an interface, which will allow us to switch between displayed models provided by the sample application.

2.2 Used Technologies

2.2.1 Primary Programming Language

The programming language Rust has been chosen to provide as little execution overhead as possible. It is a language, which compiles to machine code,

and requires no runtime or garbage collection. It can integrate well with other programming languages, as it provides a foreign function interface for C. Rust has a rich type system and an ownership model, which provides guarantees about memory-safety and thread-safety. It provides unparalleled tooling in the world of systems programming, which makes package management very simple. Rust is a modern language, which strives to be as fast as C/C++, while avoiding many of their pitfalls.

> *[Rust] is designed to support concurrency and parallelism in building applications and libraries that take full advantage of modern hardware. Rust's static type system is safe and expressive and provides strong guarantees about isolation, concurrency, and memory safety.* [6]

Languages such as C, C++, Rust and Zig would all be great candidates for our project. However, I think, for our use case, Rust's features outshine those of its competitors. Additionally, before starting to work on the project, I already had experience with Rust, so I did not have to learn the language from the ground up, which sped up the development process. I think learning Rust has been a great investment, which has paid off during the development of this project.

2.2.2 Graphics API

The graphics API Vulkan[1] has been chosen, for this project. Vulkan is a very attractive graphics API, because of its cross-platform availability and because it is designed with the intention of imposing as low performance overhead as possible. It is a modern graphics API, in structure similar to DirectX 12, yet unlike DirectX 12, it is available on more platforms, including Windows, Linux, Android, and, despite the lack of official support on macOS, it can even be used there, thanks to the open source MoltenVK[2] compatibility layer. Like Vulkan, the widely used graphics API OpenGL[3] is also supported by many different platforms. However, despite OpenGL being, in terms of usability, comparatively simpler than Vulkan, this simplicity affects the performance of the API in a negative way.

It is worth mentioning, that there is an experimental graphics API in the works, which could be very useful for our project. The API is currently called WebGPU[4] and is supposed to become the web standard for accelerated graphics and compute, maintained by W3C. Because of its purpose to be used on the World Wide Web, it is built with security guarantees in mind, which would be ideal for our project. In Vulkan, the developer must ensure the API is used correctly; incorrect usage of the API results in undefined behavior. Because of this, any APIs making use of Vulkan exposed by our

[1] `https://www.khronos.org/vulkan/`
[2] `https://github.com/KhronosGroup/MoltenVK`
[3] `https://www.khronos.org/opengl/`
[4] `https://webgpu.io`

platform to the VR applications must be implemented with sanity checks in mind, otherwise the security of the users could be put at risk. The usage of WebGPU instead of Vulkan would simplify the aspect of ensuring the API is interacted with correctly. Unfortunately, at the time of writing, the WebGPU specification has not yet been finalized. It remains an attractive option for future consideration.

2.2.3 API for VR Devices

There are many different APIs for communication with VR devices. Many of them are proprietary, such as those made by Oculus and Microsoft. Those APIs only support devices produced by their respective companies. If we were to support as wide range of VR devices as possible, we would need to choose such an API, which allows us to do that.

The name of the OpenVR API might suggest, that it would support VR devices from multiple vendors. The reality is, that while it is, in fact, an API any vendor may implement for their devices, the support is limited, with no sign of vendors adding support for new devices. OpenVR was developed by Valve and does provide support for the HTC Vive, and somewhat surprisingly, the devices Oculus DK1, Oculus DK2 and Oculus CV1 are supposedly also supported.

There have been efforts by the OSVR[5] developers to unify VR devices under a single API, but progress on the OSVR SDK seems to have become stagnant, without any progress by its original developers.

Finally, in July 29th, 2019, a new specification was announced by the Khronos Group, the organization behind OpenGL and Vulkan. This API, called OpenXR[6], strives to be "a royalty-free, open standard that provides high-performance access to Augmented Reality (AR) and Virtual Reality (VR)—collectively known as XR—platforms and devices"[7]. Currently, the API is implemented by two runtimes. There is a runtime by Microsoft[8] for Windows Mixed Reality headsets, and an open source runtime by Collabora[9], called Monado[10], with support for OpenHMD[11]-compatible devices.

OpenXR is an API, which may be implemented both in a closed source, as well as in an open source manner. This makes it especially interesting for developers, because vendors of new VR devices may choose to implement a proprietary runtime with support for the newly released device, and software making use of OpenXR should work correctly with the new device, without having to wait for a community-made open source runtime. Being maintained by the Khronos Group, with their legacy of support for widely used APIs, it is safe to say, that the longevity of the API will be great.

[5] http://www.osvr.org/
[6] https://www.khronos.org/openxr/
[7] https://www.khronos.org/openxr/
[8] https://aka.ms/openxr
[9] https://www.collabora.com/
[10] https://monado.dev/
[11] http://www.openhmd.net/

Additionally, Monado, the open source OpenXR runtime by Collabora, provides support for the HTC Vive, which I currently own, meaning I can prototype using this API and my VR device. For this reason, I chose to use OpenXR as the API to interface with VR devices.

As a side note, it is worth mentioning, that the production of the HTC Vive seems to have been discontinued, despite it being, in my opinion, a very cost effective option for many users.

2.2.4 Model Format

There are many formats for 3D model transmission to choose from. A format, which would let artists encode surfaces with physically-based properties, would be preferred, with the assumption that the content we want to display is supposed to represent the physical reality. If we consider, that the artists might want to represent real-life objects within the applications, it only makes sense to choose a format, that allows for such objects to be approximately represented. One such format for 3D models is the GL Transmission Format[12], or glTF for short. It is another standard developed by the Khronos Group, with its second major version released in 2017. Being a royalty-free specification with support for physically-based rendering makes it a great candidate for our use case. The format is well documented with a publicly available specification[3]. Furthermore, it is supported by a wide range of tools, and has found its use as a common model transmission format for the World Wide Web. For these reasons, I have made the decision to use glTF in the implementation.

2.2.5 Virtualization Technology

There are various kinds of virtualization to consider. Because we are mainly interested in sandboxing application logic, it is not necessary to maintain entire virtual operating systems for our applications. It is instead possible to choose a language designed to be run within a virtual machine. JavaScript, the Java bytecode and Lua are all examples of languages, which are commonly executed in virtual machines made specifically for each language. Recent developments in this space have lead to the creation of WebAssembly[13].

> *[WebAssembly] offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web.* [2]

WebAssembly is one of the compilation targets of LLVM[14], which is a compiler infrastructure used by many modern programming languages, including

[12] `https://www.khronos.org/gltf/`
[13] `https://webassembly.org/`
[14] `https://llvm.org/`

Rust. This property lets us use Rust as a programming language not only for the platform itself, but also for the applications running on top of the platform. However, application developers are not forced to use the Rust programming language to develop their applications in; they can use any language targeting WebAssembly, such as C, C++, Go, Java, Python, and many other. Furthermore, in the Rust ecosystem, there already exist several open source virtual machine implementations to execute WebAssembly. With WebAssembly being such a promising piece of technology, I could not pass on the chance of using it within the project.

Another reason why WebAssembly is well suited for our case, is the possibility of embedding additional non-executable resources within WebAssembly binaries. In Rust, this can be done, for example, using the `include_bytes!` macro, which embeds the file at the specified path as a byte array within the source code. This allows us to distribute various kinds of resources alongside the application logic, including 3D models, scripts, or multimedia files.

2.3 Project Structure

All git repositories related to the work can be found on the page of the metaview-org GitHub organization[15] I made. There is also a website[16] I maintain, with articles about updates related to the project.

The project is structured into several sub-projects. Rust projects maintained using the Cargo package manager for Rust are called *crates*.

- `metaview`: The platform for VR applications.
- `ammolite`: The Vulkan-based rendering engine with glTF support, used by `metaview` directly.
- `ammolite-math`: A linear algebra mathematics library, located in the `ammolite` git repository, currently used by all crates for most geometric computations.
- `mlib`: Common utilities for developing `metaview` applications in Rust.
- `example-mapp`: An example `metaview` application.
- `example-mapp-2`: Another example `metaview` application.

2.4 Platform Lifecycle

The initialization phase of the lifecycle of `metaview` consists of the following events. `ammolite` is initialized with OpenXR. A single window and a single HMD are used. Next, the Specs[17] entity-component system (ECS)

[15] `https://github.com/metaview-org/`
[16] `https://metaview.link/`
[17] `https://github.com/amethyst/specs`

is initialized. The ECS is used to maintain the scene hierarchy. Finally, the `wasmtime`[18] WebAssembly runtime with the provided `metaview` application (`mapp`) is loaded. Which `mapp` to execute is determined by the first command line argument when running the `metaview` binary:

```
> metaview [PATH_TO_MAPP.wasm]
```

When the initialization is finished, the main part of the lifecycle takes place, where the loaded `mapp` is initialized and executed.

2.5 The `metaview` API

WebAssembly supports function imports and exports. This functionality lets the host invoke a specific function, that is exported by the WebAssembly module. Function imports make it possible for the WebAssembly module to invoke functions provided by the host. At the time of writing, while the functionality for function exports was already implemented, function imports were not. Therefore, I decided to resort to using a command-based API architecture.

WebAssembly supports only a handful of types with exported functions. Those include `i32` (32-bit integer), `i64` (64-bit integer), `f32` (32-bit floating point number) and `f64` (64-bit floating point number). Using just these types would quickly become tedious. The low applicability of these types has spurred the creation of the WebAssembly Interface Types proposal[19]. An experimental implementation of the proposal is provided by the `wasmtime-interface-types` crate[20]. This implementation currently provides additional support for integers with distinct signedness, and, most importantly, dynamically allocated strings. The ability to use UTF-8 strings as parameter types and return types of exported functions makes it possible for arbitrary types to be transferred, by the use serialization. Serde[21], a well known and widely used serialization and deserialization crate, can be employed for this task.

However, since we want to avoid having the `mapp` developers do the serialization and deserialization themselves, as that would contribute to the boilerplate, we can provide a macro to generate the exported functions. These generated exports perform the argument deserialization, they pass the deserialized arguments to a function implemented by the `mapp` developer, they serialize the return value, which is then returned as a string. This is one of the use-cases of the `mapp` procedural macro provided by `mlib`.

As an example, take a hypothetical `exported_function`, function exported by the `mapp` and invoked by the `metaview` platform. This function takes a single argument `I` and returns a single value `O`. A visualization of the steps required to invoke the function is shown in 2.1.

[18] `https://github.com/bytecodealliance/wasmtime`
[19] `https://github.com/WebAssembly/interface-types`
[20] `https://crates.io/crates/wasmtime-interface-types`
[21] `https://serde.rs/`

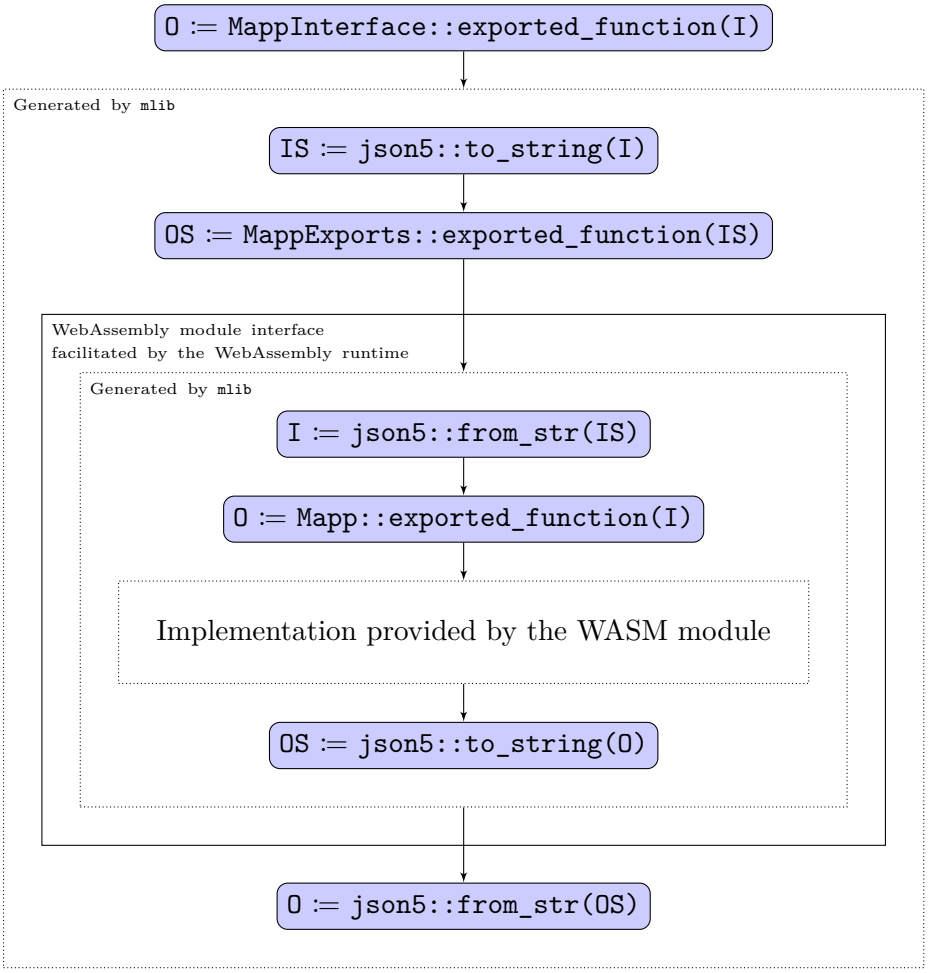


Figure 2.1: A visualization of how invocations of functions exported by the WebAssembly module are performed. The string conversions necessary to transfer arbitrary types across the WebAssembly module interface are handled by code generated by the `mlib` library.

2.5.1 Required Module Exports

In order for a WebAssembly module to be a fully qualified `mapp`, it must implement the following functions. These functions are implemented by the `mapp` developer and are invoked within the exports generated using the `mapp` macro.

```
fn update(&mut self, elapsed: std::time::Duration);
```

The `update` function is called in every iteration of the render loop. The argument `elapsed` is the duration since the initialization.

```
fn send_command(&mut self) -> Option<mlib::Command>;
fn receive_command_response(
    &mut self,
```

```
    response: mlib::CommandResponse,
);
```

The functions `send_command` and `receive_command_response` are used for communication between the host and the `mapp` via commands. `mapp`s schedule commands in a queue during the invocation of `update`. When the execution of `update` is finished, the host polls for the scheduled commands by calling `send_command`, and executes the commands accordingly, invoking `receive_command_response` with an appropriate response after each command. The types `mlib::Command` and `mlib::CommandResponse` are described in 2.5.2.

```
fn receive_event(&mut self, event: mlib::Event);
```

This function is invoked by the host whenever a window-related, device-related or OpenXR-related event has occurred. It allows the application to respond to events such as the mouse buttons being pressed, or the state of the OpenXR session being changed. Every invocation of this function is followed up by invocations of `send_command`, so as to allow the application to apply desired changes immediately in response to `receive_event` being invoked.

```
fn flush_io(&mut self) -> mlib::IO;
```

The function `flush_io` is used to transfer the content of `mapp` buffers to be printed to the `stdout` and `stderr` by the host. The buffers should be emptied as a result of the invocation. This function is especially useful when debugging `mapp`s. It is worth clarifying, that WebAssembly modules, being executed in a sandboxed environment, do not have access to the standard IO by default.

2.5.2 Command Types

The types used in the command-based API are described by the `mlib` crate. The commands and responses are generated by the macro `command_kinds!`:

```
macro_rules! command_kinds {
    {$(
        $name:ident
        $({ $($request_fields:tt)* })?
        $(-> { $($response_fields:tt)* })?
    ),*$(,)?} => { /* Macro body omitted. */ }
}
```

The macro generates variants for the `enum`s `CommandKind` and `CommandResponseKind`. The syntax of this particular macro consists of:

1. the name of the command, followed by

2. fields of the command (request) variant surrounded by curly brackets (or nothing for no fields), followed by

3. the arrow symbol `->` and fields of the response variant surrounded by curly brackets (or nothing for no fields).

The following commands are generated.

```
ModelCreate {
    data: Base64ByteSlice,
} -> {
    model: Model,
},
```

Requests a model to be loaded from the provided byte slice. The byte slice takes form of a Base64 string to speed up command serialization and deserialization described in 2.5. As a response, the platform provides a reference to the loaded model. References are encoded as indices of the respective resources.

```
EntityRootGet -> {
    root_entity: Entity,
},
```

Requests the *scene root entity*, which is unique to every `mapp` instance. In order for entities to be visible in the scene, they must be descendents of the *scene root entity*, in terms of the scene graph hierarchy. The response contains a reference to the *scene root entity*.

```
EntityCreate -> {
    entity: Entity,
},
```

Requests an entity to be created. The response contains a reference to the created entity.

```
EntityParentSet {
    entity: Entity,
    parent_entity: Option<Entity>,
} -> {
    previous_parent_entity: Option<Entity>,
},
```

Requests the `entity`'s parent to be set to `parent_entity`. The response contains the previous parent entity assigned to the `entity`, or `None`, if none was assigned.

```
EntityModelSet {
    entity: Entity,
    model: Option<Model>,
} -> {
    previous_model: Option<Model>,
},
```

Requests the `entity`'s model to be set to `model`. The response contains the previous model assigned to the `entity`, or `None`, if none was assigned.

```
EntityTransformSet {
    entity: Entity,
    transform: Option<::ammolite_math::Mat4>,
} -> {
    previous_transform: Option<::ammolite_math::Mat4>,
},
```

Requests the `entity`'s transformation matrix to be set to `transform`. The response contains the previous transform assigned to the `entity`, or `None`, if none was assigned.

```
GetViewOrientation -> {
    views_per_medium: Vec<Option<Vec<View>>>,
},
```

Requests information about the views. These contain the view transforms for each resulting framebuffer; that is, the view matrices used for rendering the scene to the windows, as well as view matrices used for each eye of HMDs.

```
RayTrace {
    origin: ::ammolite_math::Vec3,
    direction: ::ammolite_math::Vec3,
} -> {
    closest_intersection: Option<Intersection>,
}
```

Requests a ray to be cast into the scene from the origin `origin`, with direction `direction`. The response contains the closest intersection with any rendered object within the scene. The intersection contains the point of intersection, the distance of that point from the origin, and a reference to the intersected entity.

2.5.3 Potential Changes Induced by the Addition of Function Imports

When the support for function imports is added to `wasmtime`, command types will be transformed into imported functions, which will be possible to invoke directly from the application source code. The command fields will

take form of the function parameters and the response fields will take form of individual `struct`s (or `()` if no fields are specified) to be used as the return type. This change will result in significant simplification of the API and lowering of cognitive load when implementing `metaview` applications.

Chapter 3

Sample metaview Applications

Two example mapps, example-mapp and example-mapp-2, were created in order to demonstrate the capabilities of the metaview platform.

3.1 An Example mapp #1, example-mapp

When loaded into the metaview platform, the application displays an interface to list through displayed models packaged with the application. The application is interacted with via two buttons, which may be triggered by directing the HMD at them for a brief while.

3.1.1 The Lifecycle of example-mapp

When the application is loaded, several commands are scheduled for execution. First, the *scene root model* is requested using the EntityRootGet command, then, all required models, which are embedded in the WebAssembly module, are loaded via the ModelCreate command. Finally, commands to create the entities are scheduled. One for each button (*Previous* and *Next*), one for the ray intersection indicator (which is shown as a red sphere), and as many as specified (by default, 3) for the currently displayed model.

The update function uses the elapsed parameter to compute the transformation matrices for each entity used to display the current model, which are then updated using the EntityTransformSet command. The function also queries the view orientations using GetViewOrientation on every invocation.

The receive_command_response function takes track of the current state of the initialization, and processes the responses accordingly. During the response to GetViewOrientation, the forward vector of the HMD is calculated as an average of the direction of both views. Using this forward vector, the RayTrace command is scheduled. The RayTrace command is used to keep track of how long the user has been looking at which button entity, and whether the button should be triggered.

The triggering of a button results in the entities used to display the current model being assigned the next (or previous) model using the EntityModelSet command.

Figure 3.1: A screenshot of the `metaview` window running the `example-mapp` application. A helmet model can be seen, along with a pair of buttons, and a small red sphere indicating the ray intersection point.

3.2 An Example mapp #2, example-mapp-2

The second example application, `example-mapp-2`, builds on top of the structure of the `example-mapp`, but differs in functionality. The goal of this application is to demonstrate the ability of the platform to handle more complex interactivity requirements, which is demonstrated by allowing the user to rearrange a set of objects in the scene. It additionally introduces mouse handling to control whether the user is picking up an object, holding onto it, dropping it, or whether they are performing none of those actions. Several additional models are included in the scene for world-space position reference.

Figure 3.2: A screenshot of the HMD screen while `metaview` is executing the `example-mapp` application. A helmet model can be seen, along with a pair of buttons, and a small red sphere indicating the ray intersection point, this time from two points of view.

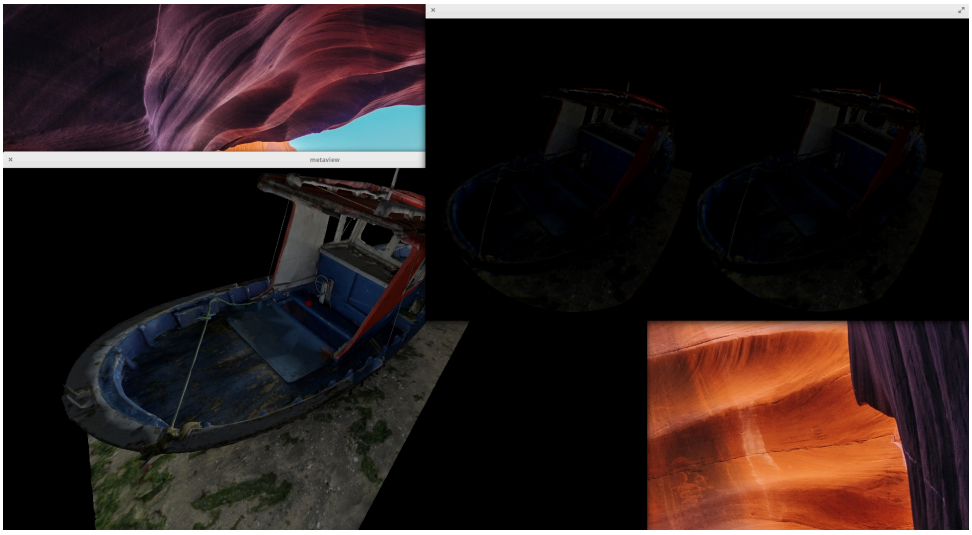


Figure 3.3: A side-by-side view of both the main `metaview` window and a debug screen displaying picture sent to the HMD. Notice the barrel distortion applied by OpenXR. This image was taken before the addition of buttons to `example-mapp`.

Battle Damaged Sci-fi Helmet - PBR by theblueturtle_, published under a Creative Commons Attribution-NonCommercial license. Link.

Battle Damaged Sci-fi Helmet - PBR by theblueturtle_, published under a Creative Commons Attribution-NonCommercial license. Link.

Figure 3.4: A screenshot of the `metaview` window running the `example-mapp-2` application. The helmet model being grabbed in the first screenshot, then being released in the second screenshot, with its position and orientation changed in relation to the world, but unchanged in relation to the camera.

Chapter 4

Performance Evaluation

In order to benchmark the performance of the implementation, two different tests were performed on the `example-mapp-2` application. The tests were constructed to measure the impact of using the abovementioned approach to host a `mapp` using a WebAssembly module (labeled as "WASM") as opposed to an approach where the `mapp` would be compiled with `metaview` and where no command serialization and deserialization would have to be performed (labeled as "Native").

In order to perform the evaluation, an abstraction had to be created so that both "Native" and "WASM" applications could be interfaced with the same way. This abstraction has been added to the `mlib` library, which generates appropriate bindings for both approaches.

The hardware specifications of the machine this evaluation was performed on are shown in figure 4.1.

4.1 Application Load Time Test

During the development of the example `metaview` applications, I have noticed a significant impact on application load times, proportional to the size of the models embedded in those applications. Undeniably, the cause of this issue was the fact, that since every command sent across the `metaview` interface has to be serialized, even the glTF models (embedded in the functionally equivalent binary format *.glb*) had to be first serialized into UTF-8, then sent across, and finally, deserialized again, into its original binary form. An inspection of the serialized form of the `ModelCreate` command showed, that the type `Vec<u8>` (the Rust type for an array-backed list of unsigned bytes) was being serialized as a JSON array of individual bytes in their decimal form. That is reasonable behaviour one would expect from a serialization library, but it was not ideal for our use-case. I managed to lower the size of this command type by encoding the binary data using the Base64 encoding scheme, which yields a UTF-8 string, that can be used directly in the serialized version of the command. This change significantly improved the load times of the applications, but measurements still show a large difference in load times.

This test measured the time it takes to initialize the application module

CPU Model	AMD Ryzen 9 3900X
CPU Core Count	12
CPU Thread Count	24
Max Single-Core CPU Clock	4.6 GHz
CPU Base Clock	3.8 GHz
CPU L1 Cache	768 KiB
CPU L2 Cache	6 MiB
CPU L3 Cache	64 MiB
GPU Model	NVIDIA GeForce GTX 1070
GPU Core Count	1920
GPU Boost Clock	1638 MHz
GPU Base Clock	1506 MHz
RAM Architecture	Dual Channel
RAM Memory Type	DDR4
RAM Configuration	2x 8 GB
RAM Clock	3.6 GHz
RAM Latency	16-16-16-36-2N
Operating System	elementary OS 5.1 (Ubuntu 18.04), Linux 4.15.0

Figure 4.1: Machine Configuration

(either loading the WebAssembly module in the "WASM" case or simply instantiating the application in the "Native" case) and process all commands sent by the application during the first invocation of the 'update' function. The results are shown in 4.2.

4.2 Application Frame Time and Framerate

This test was constructed to measure the difference in the time it takes to `update` the application and render it to the head-mounted display and the window. The results are shown in 4.3.

4.3 Conclusion of Performance Evaluation

As can be seen from the results, the performance impact of the current implementation is significant enough to become a drawback when considering the `metaview` platform. However, I am convinced that most of the performance regression is caused by the serialization and deserialization steps when communicating with the applications across the WebAssembly interface. This approach was suitable enough for the prototyping of the platform and the applications, but would be insufficient in real-world applications, where high-framerates and short load times are desirable.

Fortunately, this drawback could be mitigated by making use of WebAssem-

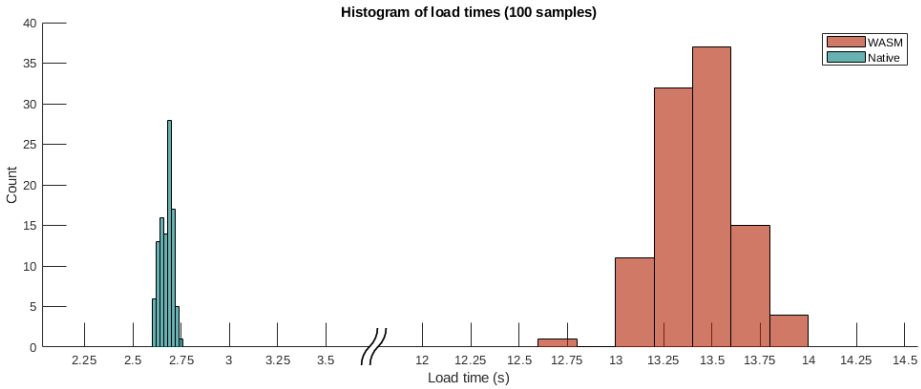
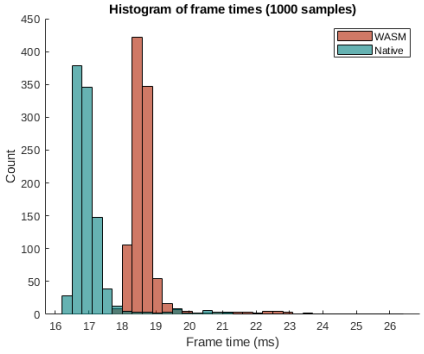
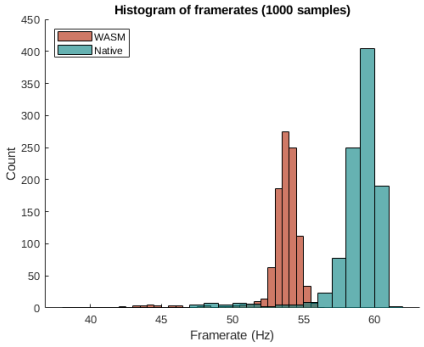


Figure 4.2: A histogram of measured load times. 100 samples each.
Load time statistics (s):
"WASM" (red): median of 13.42, mean of 13.42, standard deviation of 0.21.
"Native" (cyan): median of 2.68, mean of 2.67, standard deviation of 0.03.



(a) : Frame time



(b) : Framerate (inverse of frame time)

Figure 4.3: A histogram of measured frame times. 1000 samples each.
Frame time statistics (ms):
"WASM" (red): median of 18.58, mean of 18.71, standard deviation of 0.75.
"Native" (cyan): median of 16.87, mean of 17.05, standard deviation of 0.83.

bly Interface Types from the WebAssembly Interface Types proposal[1], once it is finalized and made available through a WebAssembly runtime. This should make it possible to transfer data across the interface without the necessity to serialize it into UTF-8, thereby reducing the time spent on serialization and deserialization of commands.

[1] `https://github.com/WebAssembly/interface-types`

Chapter 5

Usage Guide

This chapter documents the way the `metaview` platform is compiled and run on a machine with the configuration in figure 4.1 and the HTC Vive headset.

5.1 Acquiring Dependencies

Install the official, proprietary NVIDIA drivers. I used version 440.59.

The following libraries are required to compile and install Monado:

```
sudo apt install libeigen3-dev libv4l-dev libglew-dev \
    libusb-1.0-0-dev libhidapi-dev libxcb1-dev \
    libxcb-randr0-dev
```

Compile and install OpenHMD by following the installation instructions[1].

Follow the build instructions[2] to install glslang.

Follow the installation instructions for Monado[3]. It may be possible to use another OpenXR runtime, but during the time of writing, Monado is the only runtime with Vulkan support.

In order for Monado to be able to interact with the HTC Vive headset, the following dependencies must be installed via the package manager:

```
sudo apt install steam-devices libudev-dev
```

Following these steps, a system restart may be required.

5.2 Compiling `metaview` and related projects

Install the Rust toolchain via the official installer *rustup*[4].

Install *wasm-pack* via Rust's package manager *cargo*:

```
cargo install wasm-pack --version 0.8.1
```

[1] `https://github.com/OpenHMD/OpenHMD#compiling-and-installing`
[2] `https://github.com/KhronosGroup/glslang#building`
[3] `https://gitlab.freedesktop.org/monado/monado#getting-started`
[4] `https://rustup.rs/`

Clone the following projects into their respective directories:

```
git clone --branch bachelor-thesis \
    https://github.com/metaview-org/metaview
git clone --branch bachelor-thesis \
    https://github.com/metaview-org/example-mapp
git clone --branch bachelor-thesis \
    https://github.com/metaview-org/example-mapp-2
```

In order to compile example `mapp`s, resources must be added to the source directories. These resources are included alongside the digital version of the thesis.

To compile a `mapp`, run the following command from within the repository:

```
WASM_INTERFACE_TYPES=1 wasm-pack build
```

This generates a WebAssembly binary file in the `pkg` subdirectory.

To compile `metaview`, navigate to the `metaview` repository and execute the following:

```
cargo build --release
```

This generates a native `metaview_bin` binary in the `target/release` subdirectory.

5.3 Running `mapp`s

Compiled `metaview` binary and `mapp` binaries may be found in the *Releases* page of each respective GitHub repository. An installed OpenXR runtime is necessary to run `metaview`.

To run a compiled `mapp`, execute the following:

```
XR_RUNTIME_JSON=$PATH_TO_MONADO/build/openxr_monado-dev.json \
    ./metaview_bin $PATH_TO_WASM
```

where `$PATH_TO_MONADO` is the path to the Monado repository and `$PATH_TO_WASM` is the path to the compiled `mapp`. Multiple `mapp`s may be specified.

Chapter 6

Conclusion

In this work, I have explored the reasons for the creation of a new browser-like application platform. I have focused on implementing a subset of the platform, under the name `metaview`. I have justified the usage of chosen technologies for the creation of the platform. The implementation consisted of a glTF rendering engine, built on top of Vulkan, with support for VR devices via the OpenXR API. Furthermore, the execution of applications (`mapp`s) was virtualized using a WebAssembly runtime and an API for communication between the applications and the host platform was developed.

Finally, two example applications, `example-mapp` and `example-mapp-2`, were developed, to demonstrate the capabilities of `metaview`, and the performance impact of the virtualization approach was measured on the `example-mapp-2` application.

6.1 Notable Issues Encountered During Development

This section contains issues, that I have encountered during the development of the `metaview` platform, in no particular order.

6.1.1 View Matrix Confusion

Getting the transformation chain right, so that the shaders, used to render the scene, would work correctly, was no easy task, and was very difficult to debug. Additionally, HMDs require off-axis perspective projection matrices, which resulted in more complexity. Thankfully, I could use a derivation[1] of the off-axis perspective projection matrix by Song Ho Ahn.

6.1.2 Compiling Rust to WebAssembly

While the WebAssembly ecosystem within Rust is very active, it is fair to say, that many tools in this space lack maturity.

First, as previously mentioned in 2.5, is the lack of function imports in the `wasmtime` runtime.

Second, there currently seems to be a problem with compiling Rust projects with large embedded files (such as glTF models) into WebAssembly. The compilation seems to run out of memory, if a file too large is embedded within the module. I believe, that this is due to a bug in the LLVM compiler.

6.1.3 Lack of OpenXR Runtimes

Currently, only two OpenXR runtimes exist. One of them, Monado, was very useful to me during the development of `metaview`, but it is far from feature-complete. Its support for the HTC Vive, a VR device I own, is currently limited to 3DOF (3 degrees of freedom) tracking, with experimental 6DOF tracking based on libsurvive[1], and does not support the HTC Vive controllers at all. Hence, why I had to resort to interactions by pointing the HMD at entities, as described in 3.1.1. Additionally, I believe there to be some bugs in the Monado runtime, like incorrect color space handling, as can be seen in fig. 3.3.

Despite these shortcomings, I would like to express my gratitude to the developers of Monado, who were very willing to offer a helping hand. I feel very lucky that the existence of Monado allowed me to pursue this effort.

I was hoping an official OpenXR runtime by HTC would be released during the development of `metaview`. However, it seems that the production of the HTC Vive has been discontinued, and I am now unsure, whether such a runtime will ever be released, and if it is, whether support for the HTC Vive will be included.

6.1.4 Vulkano

The `ammolite` rendering engine was built using the Vulkano[2] Rust library, which aims to provide a safe abstraction over Vulkan. However, during the development of `ammolite`, it became evident, that many adjustments to the library were necessary[3]. As `ammolite` grew, the practicality of Vulkano became questionable, and so did its performance. I was told by a colleague about his ongoing work on rewriting the Vulkano library, as Vulkayes[4], which I believe would be a good idea to eventually switch to.

6.2 Future work

6.2.1 A Lower-Level Graphics API

The idea of using a lower-level graphics API is explored in appendix A. The usage of such a lower-level API will have a significant impact on the structure of the entire platform, which should not be neglected. In order to create a

[1] `https://github.com/cnlohr/libsurvive`
[2] `https://github.com/vulkano-rs/vulkano`
[3] `https://github.com/Limeth/vulkano`
[4] `https://github.com/TheEdward162/Vulkayes`

truly flexible platform, such an API is going to be required. I think the best course of action, currently, would be to use the Vulkan-based glTF rendering engine, until WebGPU becomes more mature. Once the usability of WebGPU becomes better, I believe it would be a great candidate for our project, as discussed in 2.2.2.

6.2.2 Error Handling

The command-based API explored in 2.5 currently assumes, that no errors occurr during the execution of `mapp`s and during the fulfillment of commands received from `mapp`s. Better error handling should be employed.

6.2.3 Permission System

With multiple `mapp`s running at the same time, a need for some kind of inter-`mapp` communication protocol might emerge. The platform should assume application sandboxing by default, so some kind of permission system for communication between applications should be developed. It should be investigated as to how resources should be shared between `mapp`s.

6.2.4 Networking

This work has not focused on the implementation of one of the most important properties of the platform, and that is decentralized networking, as discussed in 1.1.1. I intend to consider exploring this part of the platform in my master thesis, as it is a very extensive topic.

Bibliography

[1] Song Ho Ahn. *OpenGL Projection Matrix*. URL: http://www.songho.ca/opengl/gl_projectionmatrix.html (visited on 01/05/2020).

[2] Andreas Haas et al. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: https://doi.org/10.1145/3140587.3062363.

[3] The Khronos Group Inc. *The GL Transmission Format 2.0 specification*. URL: https://github.com/KhronosGroup/glTF/tree/master/specification/2.0 (visited on 04/24/2020).

[4] Jason Jerald. *The VR Book: Human-Centered Design for Virtual Reality*. Association for Computing Machinery and Morgan & Claypool, 2015. ISBN: 9781970001129.

[5] Steven M. LaValle. *Virtual Reality*. Cambridge University Press, 2019.

[6] Nicholas D. Matsakis and Felix S. Klock. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 103–104. ISBN: 9781450332170. DOI: 10.1145/2663171.2663188. URL: https://doi.org/10.1145/2663171.2663188.

Appendix A

Discussing the Requirements of a Lower-Level Graphics API

In this appendix, I would like to explore the implications of using a safe graphics API, such as WebGPU, to expose lower-level graphics functionality to `mapp`s. This would allow `mapp`s to render graphics in a cutomizable way, without the limitations of glTF. This discussion serves as a reference for future work.

Let us strive for an implementation, where every `mapp` defines their own rendering pipelines with custom shaders. Common uniform variables, such as the view and projection matrices, should be provided to those shaders automatically by the `metaview` platform, so as to prevent implementation discrepancies between `mapp`s. Using a single framebuffer for all of the custom pipelines would make it possible to render geometry with correct depth occlusion, as long as the geometry is either fully opaque or fully transparent. However, a problem arises when also considering translucent geometry, which would most certainly also be required.

A.1 Translucency

Rendering translucent geometry correctly is a fundamental problem in rasterization based rendering. In order to get an accurate result, fragments need to be rendered and blended from the farthest to the closest. However, draw calls are typically not scheduled in this order, because ordering them would be resource demanding. Additionally, in our case, where draw calls would be scheduled by `mapp`s, we would have no control over the order of these draw calls. It would be possible analyze the geometry passed to the draw calls, break it up into sorted pieces, and reschedule it. This would however come at a great cost of computational resources, if this analysis were to be done every frame.

A more feasible, yet opinionated approach, would be to make use of order-independent transparency (OIT), which is a set of techniques, one of which I used in the implementation of the glTF rendering engine `ammolite`. OIT attempts to approximate the result one would get by rendering ordered geometry, without actually ordering it. The techniques do not yield perfect

results, but they come with great improvements to computational requirements, with often indistinguishable results from those one would get from order-dependent techniques.

A.2 Visual Cohesiveness

Perhaps an OIT technique could be applied globally to all fragment shaders used for rendering translucent geometry. This would ensure, that the result would look visually cohesive, assuming that the usage of multiple different blending techniques would be easy to spot. The advantage of this approach would be the ease of implementation by the `mapp` developers. They would only have to create a fragment shader with a function that returns the fragment RGBA color and depth. This shader (in the SPIR-V format) would then be automatically patched by `metaview`, overriding the shader's entry point with a function implementing the OIT technique-specific blending.

Nonetheless, I cannot help but feel that using this approach would end up limiting `mapp` developers too much. We have basically replaced the problem of requiring a specific model transmission format by the requirement of a specific blending implementation.

I think that a reasonable solution would be to make `mapp`s be rendered in a user-defined order. The rendered frames of each `mapp` could then be blended together into a single frame using the user-defined order. If two subsequent layers opted to use the platform-provided OIT technique as described above, by default, they would be rendered into a single frame, sharing the depth buffer for opaque geometry and buffers required for the OIT technique. However, it would make it possible to let applications opt-out of the platform-provided OIT technique to implement custom blending within the application's layer, which could be an acceptable compromise. Moreover, this layer-based solution would allow the user to enforce a specific order of rendering on the `mapp`s, which could be useful to prioritize access to certain, more important, applications. An example is shown in figure A.1.

A.3 Input Handling and Interaction

Input handling, like rendering, should be as customizable as possible, to satisfy the interaction requirements of arbitrary geometry rendered by `mapp`s. While I was implementing the ray casting used to detect intersections with glTF models, it became clear to me, that the usage of an acceleration structure is necessary for an efficient ray casting implementation. However, consider the scenario, where many `mapp`s are running, while each contains a single object that the user can interact with. Since each `mapp` would implement its own acceleration structure, acceleration structures of all currently displayed `mapp`s would have to be traversed.

In order to allow interaction prioritization based on the layered structure as shown in section A.2, the interactions would have to be managed by the

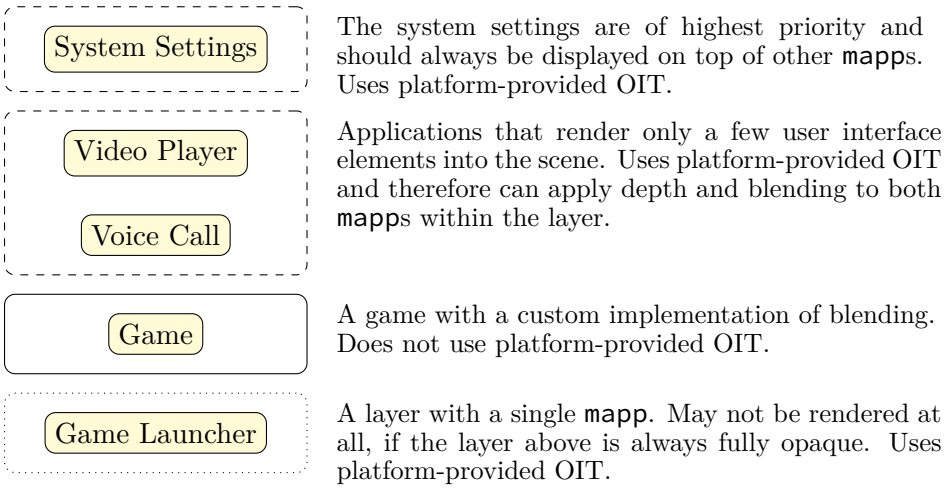


Figure A.1: An example of how mapps (yellow) could be composed using layers (surrounding boxes).

platform. First, an intersection point would be requested from all currently running mapps using a common ray. Then, the mapp with the closest intersection within the uppermost layer which contained an intersection, would actually be notified of the interaction happening.

I think this could be a reasonable approach, and if needed, it would be possible to introduce a global acceleration structure accessible by all mapps. A common acceleration structure for mesh-based geometry could even make it possible to implement a shared physics engine. I do not think these implementations should be provided by the platform, as they would inherently be biased. Maybe a better solution would be to implement these global acceleration structures and shared physics engines as mapps themselves. There could be a single physics engine mapp, that the other mapps would register their geometry in.

This proposal raises many other questions, such as how the communication between mapps would be facilitated by the platform, how versioning of such mapps would work, etc. More work needs to be done to research suitable solutions for the creation of a truly universal platform for VR applications.

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hlusička Jakub** Personal ID number: **474372**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

A Platform for Virtual Reality Applications

Bachelor's thesis title in Czech:

Platforma pro aplikace virtuální reality

Guidelines:

Design and implement the foundation of a platform for browsing virtual reality (VR) applications on immersive devices based on the Vulkan graphics API.
Devise a way to develop applications for the platform and choose suitable formats for the distribution of such applications. Make it possible to distribute content along with the application, such as 3D models, scripts, and necessary multimedia files. Measure the performance impact of the final implementation.
Demonstrate the functionality of the solution by creating minimally two sample applications and running it on the platform.

Bibliography / sources:

[1] Jason Jerald. The VR Book: Human-Centered Design for Virtual Reality. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA 2015.
[2] Steven M. LaValle. Virtual Reality. Cambridge University Press 2016.
[3] Khronos Group. Vulkan API. Online: https://www.khronos.org/vulkan/ , 2018.

Name and workplace of bachelor's thesis supervisor:

Ing. David Sedláček, Ph.D., Department of Computer Graphics and Interaction, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. David Sedláček, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature