UCL

Inter-networking Heterogeneous 3D Applications using the Hyperverse Transfer Protocol (HVTP)

by

Jaspreet Singh Dhanjan

supervised by

Professor Anthony Steed

A thesis submitted as part requirement for the MSc Degree in Computer Graphics, Vision and Imaging at University College London.

September 2020

Declaration of Authorship

I, Jaspreet Singh Dhanjan, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Acknowledgements

Many individuals have influenced this thesis who I would like to mention. First and foremost, I would like to thank my project supervisor, Professor Anthony Steed, for his outstanding support, guidance and patience throughout this project. Any credit for this project's success goes to him. It has been a thoroughly enjoyable experience to work with him over the past year.

Thank you to Kevin Coutinho from The Windsor Fellowship for his mentorship, in addition to DeepMind for their generous scholarship which allowed me to fund my dream Master's degree.

I would like to acknowledge, with a deep sense of gratitude, my father Ranjit Singh Dhanjan and mother Baljeet Kaur Dhanjan, who have inspired me to push forward in my studies. I would not have been able to experience this part of my life if it wasn't for your guidance and support - thank you. Lastly, thank you to Jasmin Kaur Dhillon for your tremendous help over the past year.

Abstract

The distribution of 3D information between native heterogeneous applications has been a challenging problem to solve. Today, developers are uncomfortable with the costly and restrictive "walled-garden" distribution ecosystems. More importantly, there is no real protocol and convention for real-time model sharing between applications. We propose a new approach: the Hyperverse Transfer Protocol (HVTP). This is an application-level networking protocol experiment, which aims to allow clients to view and share changes made to the collaborative virtual environment (CVE) using a shared scene-graph. This project uses glTF as the common interchangeable scene-graph format to describe the CVE. Moreover, since glTF is a widely supported format, we investigate the opportunities to create a platform-agnostic graphical distributed system. In one example, we investigate the sharing of graphics between a Unity and Python client. Furthermore, we investigate message-based optimisation techniques through the creation of three prototypes in our protocol. We compare the bandwidth and latency of these prototypes and suggest improvements for future iterations of HVTP.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A phenomenon that has yet to be observed in computer graphics is the cross-pollination of 3D information between immersive heterogeneous applications. This work is concerned with creating a persistent shared environment, that allows users to view, modify and edit 3D content in real-time. The requirement for such a system stems from several events, such as the rapid growth of consumer virtual reality (VR) hardware, improvements in internet bandwidth and the ever-increasing opportunities for remote collaboration. Today, the current incompatibilities of the platforms and available tools make it difficult to share existing 3D information through a common interface.

This is a hurdle that is not entirely idiosyncratic in software engineering. Towards the end of the twentieth century, the distribution of text-based information was standardised in HTTP. Little did we know at the time, the proliferation of this protocol ultimately led to a wealth of academic, economic and social change.

However, there are added dimensions of complexity when considering the distribution of 3D information on a network. This information could be textures, meshes and complex lighting equations. These components come together to define an object within a virtual environment (VE). These objects may even have their own behaviours defined by some language. Furthermore, each game engine or rendering framework defines these objects within their own implementation of the scene-graph - we refer to an application that uses a unique scene-graph implementation as a heterogeneous application.

1.1 Approach

The scene-graph is a tool that is all-pervasive in game engines. At the very least, a scene-graph is an undirected graph data structure used to model environments. Typically, each leaf may represent some geometry and the interior nodes represent the transformations relative to the parent node. A rendering pipeline within the engine may traverse this graph and perform draw calls to the underlying graphics card. Figure 1.1 summarises this. The scene-graph shields the developer from the underlying display devices, graphics processors and in some cases the rendering library.

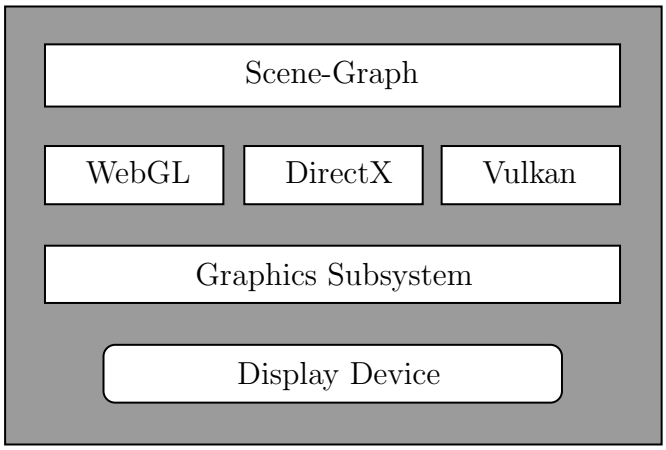


Figure 1.1: The scene-graph programming model adapted from [46]. The scene-graph communicates draw calls to the underlying graphics application programming interface (API), which is in turn passed to the graphics accelerator. The display device is purposefully vague, it can be a monitor or a head-mounted display.

This report aims to provide a solution to this problem using an independent scene-graph format as the vehicle for communications between heterogeneous applications. Hence, we present the Hyperverse Transfer Protocol (HVTP). HVTP aims to be a low-latency and interoperable transmissions standard for disparate rendering technologies. By using glTF as the interchangeable file format, we can demonstrate that graphical applications can be written in different languages and executed in different runtimes to render the same collaborative virtual environment (CVE).

We refer to the shared environment as the Hyperverse. Like most distributed systems, reliable TCP packets are used to communicate the change of state between the different HVTP client applications. In this project, we develop three separate

HVTP clients written in Unity, Python and JavaScript technologies. HVTP server applications are written in Java and Python.

The protocol itself is implemented in a series of three prototypes, with each protocol theoretically being faster than the last. Our first prototype consists of transmitting the glTF file upon each modification of the Hyperverse. We realise that this could be inefficient and introduce a second message type that consists of an XOR diff of the file with gzip compression. In our benchmarks, we discovered that this was just as inefficient as the first message type. We develop a third message type that explicitly defines that change of pose for a transformed object in the scene-graph, which vastly improves performance.

1.2 Structure

This thesis is organised as follows:

Chapter 2 takes a look at a few CVE systems throughout history and the technologies behind them. In the following section, we look at ways to represent 3D graphics within a file and how this too has vastly changed in recent decades. Lastly, we briefly look at the impact of latency in CVEs and how it can be measured.

Chapter 3 assimilates this background information. It is here where we begin to construct our ideas of how our system could be built and how it should function. We justify our decisions using the information we previously gathered.

Chapter 4 formalises everything we have discussed into a specification to not only assist us during the development of the project but to help visualise a working understanding of the system for the reader.

Chapter 5 is a commentary of the implementation of our protocol design into several working prototypes. We discuss our pitfalls and our low-level decisions that were not previously covered. This chapter is highly recommended for readers with an active interest in the inner-workings of our system.

Chapter 6 puts our prototypes to the test and extensively breaks down the components of the system. We investigate code correctness, protocol bandwidth and the end-to-end processing time of our clients.

Chapter 7 discusses the results of our tests. We dig deeper into our system

to uncover previously unknown behavior and how it could be improved in a more formalised draft of our specification.

Chapter 8 provides our concluding thoughts and discusses any opportunities for further work for our proposed system.

Chapter 2

Background

2.1 An Overview of Collaborative Virtual Environments

The terms collaborative virtual environment (CVE) and networked virtual environment (NVE) is considerably broad. They typically refer to a distributed system, where two or more users can inhabit and interact with one another within a shared 3D environment. Most CVEs will differentiate users by an avatar, and some may even support voice communications or video streaming. A diagram illustrating the nature of these interactions within a CVE can be seen in Figure 2.1. This section will investigate the different methods, protocols and systems behind distributing virtual environments (VEs) to users.

One of the first attempts at creating an NVE was SIMNET in the mid-1980s. SIMNET was a networked simulator built for military vehicle training exercises [30]. Although it was an expensive project, it allowed the US government to save on big-budget live helicopter and tank-based training exercises. SIMNET paved the way for standalone simulation hardware to interface with one another. This ultimately led to the development of the Distributed Interactive Simulation (DIS) [41]. DIS was an expansion of SIMNET and allowed for more complex distributed simulations. Vehicle simulators are quite complex and some may be built with a different hardware or software architectures. Thus, DIS had to be interoperative between these simulators to ensure support. Communications were achieved using

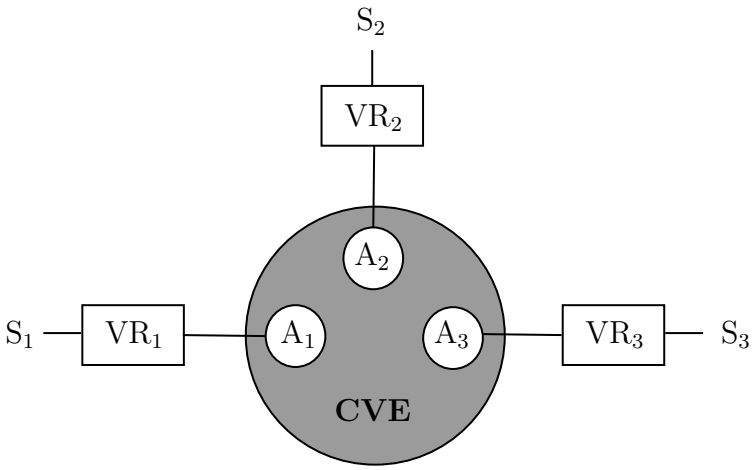


Figure 2.1: General structure of relationships in a CVE. Interactions and relationships occur between three Subjects S_n via Avatars A_n. Diagram retrieved from Presence in Shared Virtual Environments and Virtual Togetherness. [12]

.

Protocol Data Unit (PDU) messages [5]. Simulators communicated with one another in a peer-to-peer manner using these PDUs. PDU messages described the change of state of entities, such as the position of a tank. Or they could describe an event such as an explosion within the shared environment. One of the key disadvantages of this method was that if a simulator was to join late, it could not query another peer on the network. Instead, it had to wait until events or state was retransmitted by another simulator [15].

In the decade after, there were more contributions made to the research field in general. But perhaps one of the most significant was DIVE (Distributed Interactive Virtual Environment), developed at the Swedish Institute of Computer Science [16]. This is not a simulation-based protocol, but a full multi-user virtual environment (VE). As much of DIVE is built from the ground up, there is a considerable amount of published research around the implementation of DIVE. One of the novel characteristics of this system, which this report explores further, is its management of the scene-graph. The scene-graph is an entire subject we will discuss in more detail in the next section, but for now, think of it as the representation of the VE. All DIVE entities in the shared "world" interact through a distributed database. This is composed of the entity class hierarchy and very much reminiscent of the graph-based data structure we mentioned earlier. DIVE also introduces the idea of

"partial, active database replication", where segments of the world can be replicated for clients with particular interest in them. In theory, this should reduce latency for this peer-to-peer networking architecture in comparison to traditional database systems. This is because in traditional storage systems all processes must agree on a state before committing updates.

It was also around this time where graphics processing power in consumer PCs had significantly improved. Individuals now had direct access to rasterised graphics. They could participate in networked games, such as Doom, Quake and Everquest. Zeleznik et al. recognised the opportunities for interoperability between heterogeneous graphical applications and developed Scene-Graph-As-Bus (SGAB) [51] in 2000. This is significantly different than what we have discussed so far. Since many of these graphical applications had their unique implementation of a scene-graph, one could design a common interface to link them together. This is achieved through the installation of callbacks. If something had changed within the scene, a callback was triggered. The local scene-graph was then converted to a common format and added to the network bus. Other clients would receive these changes and convert this common format back into the desired local scene-graph format. The result of SGAB was a transfer of 3D content between heterogeneous client systems that had no general awareness of one another.

Fast-forwarding another decade later, further improvements had been made to consumer-level graphics. Developer tools such as OpenGL had matured significantly and graphics cards could support even more memory for buffered data and higher-resolution textures. However, with these increases in resolution comes increases in size for 3D file formats. Hnidek recognised the trade-offs between fast communications and reliability in Network Protocols for Applications of Shared Virtual Reality [24]. His proposal was The Verse protocol, developed at the Technical University of Liberec. Before we review this protocol, the reader must understand the basics of two Transport Layer protocols in the OSI model: TCP and UDP. TCP (Transmission Control Protocol) [32] is a connection-orientated protocol that is considered a reliable form of transport. It aims to achieve this reliability through inbuilt retransmission of dropped or garbled packets and acknowledges all successfully received packets. However, a UDP (User Datagram Protocol) [33] is a connectionless com-

munication model and unreliable. If data is sent using UDP, there is no guarantee that it will reach the destination. Because of this unreliability, UDP has the added benefit of being low-overhead and much faster. This makes UDP optimal for some streaming applications such as live video because it would be redundant to retransmit video frames once the time has passed. In an NVE, this situation is quite different. We aim to be as low-latency as possible whilst concurrently ensuring that every client reliably receives the correct information to update their scenes for a consistent VE. The Verse protocol aims to solve this by using fast UDP transportation and building a custom resend mechanism into the transmission process. For every UDP packet successfully received an acknowledgement (Ack) packet is sent back to the server. For any missing packets within the sequence, a not acknowledged packet (Nack) is sent. Therefore, Verse can be interpreted as a hybrid of TCP and UDP: faster than regular TCP and more reliable than regular UDP.

This new era of computer graphics also ushered in new abilities to deliver and render 3D information through a web browser. To clarify, this has happened for decades and the next section will discuss the early work of scene-graph on the web. However, a recent phenomenon is WebXR (formerly known as WebVR), which is an attempt to expose a low-level JavaScript interface to a client for immersive 3D experiences in Virtual Reality, Mixed Reality or Augmented Reality [49]. It can be used in conjunction with popular graphics libraries such as three.js [44], which uses the WebGL standard by the Khronos Group [48]. Developers can benefit from such a diverse set of libraries to create synergy. One such example of a VR framework is A-Frame [1], an open-source framework built on top of three.js, where users can use HTML to create a WebXR scene. However, developers who decide to create an online WebXR experience must include one of these JavaScript libraries. This poses a security vulnerability. If exploits are found within self-hosted libraries, then webpages can inherit these exploits. Although library vendors may be quick to release a patch, these libraries have to be updated by the webpage author to mitigate any risk of exploits. In reality, this is not what happens. In a study carried out by Northeastern University, 37% of a sample size of 133K websites used an outdated library with a "known vulnerability" [28].

Since JavaScript is a single-threaded language, some have pondered if it can

achieve the level of performance required for high-fidelity immersive systems. One suggestion to improve JavaScript performance is allowing the web developer to take advantage of hardware acceleration on the host environment. WebAssembly (Wasm) is an open standard defined to allow for this, which aims to execute binary-code at native speeds [47]. A 2018 report by McGill University's Sable Research Group noted that WebAssembly "showed impressive gains over their (Firefox's) JavaScript engines" and that it neared the performance of native C in the numerical benchmark set [6]. However, much like other low-level languages, Wasm also contains memory safety vulnerabilities. Although Wasm runs in an "isolated sandbox", it is not considered memory-safe and it has neither spatial safety, temporal safety or pointer integrity [9].

Game engines such as Unity and Unreal Engine 4 have done well to tackle the trade-offs between simple scripting and the requirement for more complex code. In Unreal Engine 4, developers have the option to use Blueprints, a visual scripting system. The alternative to this is C++, which gives more fine-grained control to the developer. In Unity, C# acts as a multi-purpose tool for both scripting and programming. Scripting is more than sufficient for most developers who want to perform simple tasks such as object transformations. Developers can also use C# for more than just scripting and build-out much larger functionality. For developers who require access to native code, a plugin interface is provided in Unity. Built underneath this elegant abstraction is the many lines of low-level systems code managed by Microsoft's .NET assemblies. These characteristics of abstraction are not a coincidence, as Steed suggests in Chapter 4 of Defining Interaction within Immersive Virtual Environments, that this in some way provides some support to the developer through inference [40]. This is particularly the case with visual programming languages such as Blueprints.

Both game engines are also able to target compilation for a variety of systems and architectures. This can be anything from mobile devices to games consoles. The problem which then arises from multiple compilation targets is how can content be shared unilaterally between applications for different architectures? In the web model, this paradigm is easily implemented using a Hypertext Reference (href) in the author's Hypertext Markup Language (HTML) document. There have been

attempts to solve this and the most important to mention is Yther, by Steed [39]. This is a proposal for a content sharing system in Unity, which has become more than relevant five years later as Unity has almost become the go-to VR development platform. Yther's specific goal is to provide a content sharing standard within Unity via the generation of an executable. This is quite different from the CVEs we have looked at so far, which are concerned with the distribution of graphical content and not runtime code. Finding safe ways of distributing high-level code at runtime is a difficult problem within computer science. It is not quite within the aims of this project but it is still necessary for an NVE or CVE as it is this code that will fundamentally describe the behaviour of the objects within the scene-graph. This is something that DIVE supports, as Steed mentions DIVE's support for TCL code snippets.

So far, we have given a brief overview of CVE systems that we feel are relevant to this project. We began by looking at simulation technologies and how each simulator was very different from one another but still was able to communicate through a common protocol. We have then looked at research projects - such as DIVE and SGAB - which took into consideration how the scene-graph was managed. Finally, we have looked at some state-of-the-art technologies which are being used today in real-time computer graphics software, such as WebXR and Unity. However, all of these examples are simply a glimpse into the tremendous amount of research that has been invested into the fields of CVEs. For a more robust overview, the reader can be redirected to the following sources: [41, 15, 4].

2.2 The Scene-Graph

To capture a VE, there must be a process of abstracting away all complexity and detail by attempting to represent objects and the relationships between them [38]. The scene-graph allows us to achieve this within a file. An early attempt at representing 3D graphics in a file format was the Virtual Reality Modeling Language (VRML) during the 90s [43]. VRML also aimed to bring a platform-independent VR capabilities to the web [35] as developers were encouraged to embed VRML within HTML documents. Objects within the VRML scene could also be manip-

ulated by JavaScript or even Java code. This was later determined as a security risk. In 2001, VRML was replaced by Extensible 3D (X3D) [14]. Much like other XML languages such as HTML, an X3D scene can be simply represented by a tree of nodes. Today, X3D is still very much around and an X3D v4 draft specification was presented at SIGGRAPH 2019 [50]. However, X3D is not as widely discussed or adopted today. Instead, developers opt for the WebGL approach, which almost gives the web developer direct access to the graphics card.

Fortunately, there have been vast improvements made into 3D file formats in recent years. There are a plethora to choose from and each may serve a different purpose. Pixar's Universal Scene Description (USD) [25] is a highly scalable format intended for rapid previewing of 3D geometry and shading. Autodesk's FBX [2] is another favoured by modellers and could be considered as an extension to Wavefront's OBJ format. However, the most interesting format to consider is the GL Transmission Format (glTF) 2.0 by the Khronos Group [20].

Since its initial introduction in October 2015, glTF has received wide-support from the computer graphics community. A glTF 2.0 asset is represented by a JSON-formatted file which contains the scene description. The scene can describe many basic elements of a 3D environment, such as cameras, animations, textures and the node hierarchy. To describe the geometry, the scene description points to a binary file via a URI which contains the buffer-based data, such as vertices. The key advantage of this is that large vertex data sets can be compressed and since the buffered data is already in an OpenGL layout, it can be forwarded directly to the graphics card. Textures are also supported and these too can be referenced in the scene and point to an image file via a URI.

Binary glTF (GLB) is an extension to glTF, where all the glTF components such as the JSON file, binary files and image files are merged into a single binary blob. The layout of this little-endian format can be seen in Figure 2.2. The fixed-length header provides information relating to the file itself. Chunk 0 is our aforementioned scene description in ASCII representation. Chunk 1 is our binary buffer of vertices. The chunk type differentiates these bytes, it may be 0x4E4F534A, which is the ASCII representation of the string "JSON", or 0x004E4942, which is the ASCII representation of the string "BIN".

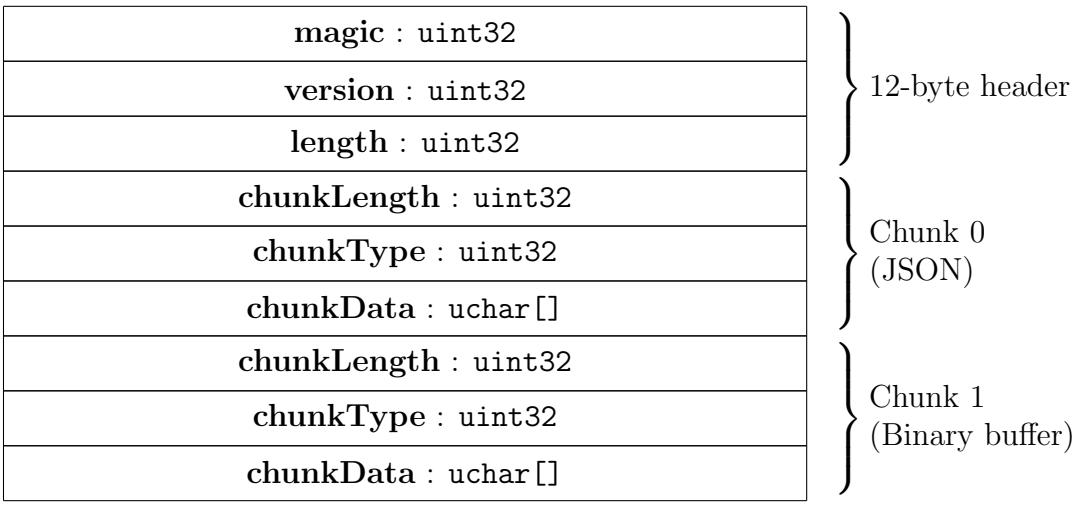


Figure 2.2: GLB layout.

However, some do not consider the glTF specification as a complete scene-graph. There is currently no way of including lights within the scene description without the use of an extension. Extensions are supported in glTF and the Khronos Group have developed several of their own, such as the KHR Lights or the Specular-Glossiness PBR extension. Such an extensible format is welcomed as there is potential for developers to create custom descriptors for future clients in our system. These descriptors could describe multi-pass rendering techniques or perhaps a bounding volume hierarchy for client renderers using ray-tracing.

Researchers have understood the significance of this format and much work has been conducted around the transport of glTF formats. Scully and Friston et al. describe a method of streaming glTF models from a database to an X3DOM client [37]. Schilling et al. also describes a method of streaming CityGML models using glTF and Cesium.js [36]. Beyond academic research, there are a plethora of open-source projects designed for glTF. The most critical to the success of this project is the range of glTF/GLB exporters and loaders available. [45, 21, 23] are just some of the examples for the Unity engine and there is support for frameworks such as Python [7] and Unreal Engine 4 [22].

2.3 Latency in Collaborative Virtual Environments

We have briefly mentioned performance in our discussion of CVEs, but this is something that we must investigate rigorously to create a useful CVE of our own. A seminal paper by Park and Kenyon demonstrates the importance of low network latency within CVEs [31]. Through experiments, they discovered that high-latency systems significantly impact user coordination within cooperative tasks. Poor network performance also caused a difference of view within the shared workspace. Individuals were unable to make reliable predictions from the poor visual feedback leading to an increase of time taken for the completion of a task.

More recently, Khalid et al. extensively classifies some of the reasons of why latency might occur in a CVE [27]. This may be due to congestion delay, queuing delay, propagation delay, processing and serialisation delay and compression. These are all stages that exist in any normal networked application. Thus, any developed software must take into account the significance of data generation, preparation and transmission from a client or server system.

A more quantifiable paper by Jay et al. investigates the delay of haptic and visual feedback within a CVE setting [26]. In this study, participants were able to detect latency above 50ms. However, results also showed that latency less than 100ms did not slow down the movements of the user to adapt to the latency. This may be attributed to the fact that the latency is not sufficient enough to cause the breakdown of the perception of immediate causality [3].

Latency doesn't necessarily have to be introduced over a network. It can also exist locally between the computer and the head-mounted display (HMD). Before we approach this, the reader must understand how a frame is rendered in VR. This is quite different than rendering directly to a screen in a typical computer graphics application. The first step in a series of many is to capture the coordinates from the HMD tracker and pass this information to the graphics pre-processing stage. The graphics pre-processor determines what part of the scene-graph should be drawn. These graphics are then drawn to the back buffer. Finally, the display is refreshed and the front buffer and back buffers are swapped. Stereo HMDs perform these processes twice: one for each eye [19]. HMD refresh rates vary. A typical entry-level consumer headset could have a refresh rate of anywhere from 60 to 72Hz and higher-

end models can go up to 120Hz. Game engines have since implemented techniques to minimise the time spent in the rendering pipeline. In Unity, the VR SDK predicts the head transforms twice: once to draw the current frame and one to simulate the next frame.

This local latency can also be measured. Friston and Steed analyse two methods of calculating latency using signal processing techniques in [17]. Specifically, Di Luca's method of calculating latency (introduced in [8]) uses two photodiodes: one attached to the tracker and one attached to the screen. These signals are captured by a soundcard and then processed using a fast Fourier transform (FFT). The shift in phase can give an accurate description of latency.

For our NVE proposal, the latency between receiving, rendering and sending 3D information must form some success criteria. Otherwise, such a system would not encourage collaboration between users. Furthermore, the bandwidth constraints must also contribute towards the success criteria, similarly to Verse protocol. This project should evaluate the size of the information that is sent and received and how it can be improved.

In the next chapter, we continue this discussion of the literature and begin to think about the implementation of our system.

Chapter 3

Analysis

From the research we have gathered in the previous chapter, we can begin to get an understanding of how we can develop such a project. We have extensively looked into the glTF format. If we can capture the state of a 3D scene, with high precision, then we can begin to contemplate how we might distribute such a state. This section will derive the beginnings of such a distribution protocol.

3.1 Networking Considerations

We argue that the principle of an interoperable shared scene-graph, much like the approach used in SGAB, still stands twenty years later. Furthermore, we believe that using a state-of-the-art format such as glTF as the interchangeable format for our distributed system could open up new opportunities for collaborative graphics in real-time. We intend to achieve this through building a transport mechanism and leveraging the pre-existing community-made glTF importers and exporters.

Earlier, we briefly touched upon the role of HTTP in computer graphics. However, something we have not discussed is defining APIs over HTTP. In the software development industry, representational state transfer (REST) over HTTP is a highly favoured way of deploying web services. In [10], Doboš and Steed outline a 3D version control framework which defines an interface over REST to a persistent MongoDB store, this would ultimately be called 3DRepo[1]. 3DRepo would later publish much more work in and around the area of 3D assets over REST, such as 3DRepo4Unity

[1] `https://3drepo.com/`

[18], a paper by Friston et al. that describes the loading of assets from a 3DRepo repository to a Unity application.

3DRepo4Unity provides a fairly comprehensive outline of how a Unity system can load assets from a REST endpoint. An interesting point made in this research by Friston is that although Unity supports dynamic modifications of its scene-graph, it uses a single-threaded programming model. For large and unorganised 3DRepo models, this is a challenge as there is the potential for thousands of HTTP calls to be made to load in a single model. To avoid sequentially downloading all of these models, the developers built a separate .NET library outside of the Unity runtime to handle the parallel loading of external assets using a multi-threaded system. These assets were passed to Unity using a callback at each frame. There are many similarities between this research and SGAB, such as the concept of a 'Portable Scene Graph', which is the transportation format to describe the 3DRepo model. The ISceneGraphTranslator then translates this portable format into Unity's own scene-graph format.

Although REST over HTTP offers an elegant abstraction layer, it does add some overhead. Furthermore, we believe the request-response paradigm is not entirely useful when constructing a CVE. In a CVE, there is no real convention to hyperlinking environments yet. Instead, we still may be able to borrow some characteristics of RESTful systems to avoid reinventing the wheel. We can construct our shared scene-graph interface using the same type of bidirectional sockets that power HTTP. Sockets are a low-level form of communications found in most programming languages. Typically, information is serialised to bytes and passed to the socket programming interface. The underlying operating system then takes up the responsibility to pass this information to the destination. There are several variants of the socket interface, but the most relevant is a streaming socket. This is a connection-orientated socket using reliable TCP communications.

Although we have discussed the negative impact on networked performance using TCP, we can leverage TCP's reliability to make a much more simpler and streamline protocol and focus on optimising the protocol at a later stage. We may even consider utilising techniques mentioned in the Verse Protocol. However, this is not within the scope of this project just yet.

As we communicate over a network with other distinct machines, we must then consider the concept of network byte order. Different CPUs have different conventions on the order in which bytes in structures are stored and processed [41]. For example, Intel CPUs are typically little-endian as the most significant bit is stored on the rightmost side. However, transmitting data over the network will convert this value to a big-endian representation, where the leftmost side is the most significant bit. Virtualised languages such as C# can often detect this. The IP Address namespace has built-in routines to handle this issue explicitly. To maintain interoperability, this networking protocol should adopt a big-endian-only convention for types and it should be entirely up to the client to choose their preferred endianness.

As of yet, we have not discussed the system architecture. Networked architectures can often be divided into two categories: variations of peer-to-peer versus client-server. In the beginning, networked games such as Doom used a peer-to-peer architecture [41]. In this model, each machine is treated as equal and carried out the same game logic using the same inputs. Because of the nature of this lock-step architecture, each player perceived the same reality. However, this led to the same problems as DIS. There was a lack of persistence - all players had to start the game together and late joiners couldn't poll existing machines. Today, a client-server model is much more common. Initially pioneered by Quake, the client-server model took away much of the responsibilities from the client. It was the server that had the authoritative control over the gameplay decisions. This was enhanced in Quake 2, where some of the simulation and prediction logic moved to the client to reduce network bandwidth use [41]. For example, the initial trajectory of an object was sent from the server to the client and it was the responsibility of the client to make rudimentary predictions based on these trajectories.

Since HVTP is an experiment, the architecture is not an important concern as of yet. In this iteration of the project, a simple client/server model may be sufficient enough to demonstrate functionality. Peer-to-peer architectures generally have better performance over client-server architecture [41]. Thus, there may be an opportunity to investigate this in the future.

3.2 Optimisations

Much like how Quake 2 reduced bandwidth by allowing the client to make predictions, we can also introduce ways to alleviate bandwidth usage in our protocol.

We have described in detail how we can represent a 3D scene as a state in glTF. However, after we have transmitted this initial scene to all of the clients, there is no real need to retransmit the same information again if nothing has changed. Similarly, if only minor modifications have been made to the shared environment, it is critical to only propagate the changes in the file and not the entire file.

Furthermore, a common action within CVEs is the animation or transformation of objects within the scene. This may be an avatar moving or an object changing rotation. For these particular use cases, it may not be necessary to send any scene-graph information over the network. Instead, a much more optimised way would be to send explicitly-defined transformation values such as rotation or translation. This almost mimics what we have described in Quake.

3.3 Languages and Interfaces

Distributed software systems often utilise more than one language or architecture. In computer graphics, programming languages such as C or C++ are ubiquitous as the application must interface with the graphics card. However, game developers do not necessarily need access to such low-level controls in their programs. As we explained earlier in Chapter 2, game engines such as Unity and Unreal Engine 4 provide several layers of programming abstractions to the developer.

Our proposal should also carry the same characteristics, but not in the way the reader might think. For example, we have decided to communicate information from client to server, or vice-versa, using sockets. Our low-level socket communications should not be visible in the final API. Instead we should provide a set of well-documented abstractions built upon this. Developers should pass over their types and the libraries developed in the project should take up the responsibility of serialising and communicating these types over the network to the server. We have described one small example, but it should give a clear illustration of how such a programming interface should be presented.

C# and Java are two examples of managed runtimes that have succeed in the software development industry. The feature of these languages that this report wishes to memorialise most is the nature of their well-documented and extensible libraries. Over the years, C# and Java have iterated over their APIs to produce developer-friendly tools, but have not replaced or removed existing code [42]. This allows for backwards-compatibility, which is highly desirable within monolithic codebases. Therefore, it is critical that the developer tools presented in this initial prototype be extensible and clear.

3.4 Summary

Distributed systems are a difficult branch of software engineering. Although VR and computer graphics present almost-limitless opportunities, it was important to set realistic project expectations. To summarise, the goals of this project are to:

- Develop a fast and robust protocol that allows us to make changes to a shared scene using the glTF 2.0 standard.
- Demonstrate the interoperability of such a protocol using different client languages and frameworks.
- Allow users to engage in a collaborative experience, which is built using the protocol and clients.
- Evaluate the performance of such a protocol.

Chapter 4

Design and Specification

This chapter outlines our proposal for our new communications standard: the Hyperverse Transfer Protocol (HVTP). The following sections describe the methods, behaviours and convention for transferring changes that have been made to the Hyperverse.

4.1 Terminology

There are various terms we use to describe the exchange of information within this protocol. It is highly recommended that the reader is familiar with these terms to understand the specification.

ASCII. The ASCII (American Standard Code for information interchange) character set is defined in the ARPA-Internal Protocol Handbook. It is used to encode any characters to bytes. Multi-byte representations such as UTF-16 are not supported.

Client. The client is responsible for the acquisition and and transfer of information between the client. Clients are simultaneously responsible for the drawing of the glTF information passed from the server. An example of a client may be written in Unity - see next chapter.

Hyperverse. The shared virtual environment which users inhabit.

Local Hyperverse. The shared virtual environment that exists at a client machine but has not yet been propagated to the server.

Packet. The terms packet and message are used interchangeably in this report. A packet is the transportation vehicle for information that is passed between server and client or vice-versa.

Server. The server is responsible for the persistent storage of the glTF file. Upon start-up, the server loads the glTF file and stores this on the heap. As clients connect, the server is responsible for the transmission of the glTF file. See the next section to see how this file is transmitted.

User. A user be may referred to any entity interacting with the Hyperverse via a client.

4.2 System Architecture

With the above definitions in mind, the following architecture (shown in Figure 4.1) provides a snapshot of the components within the distributed system.

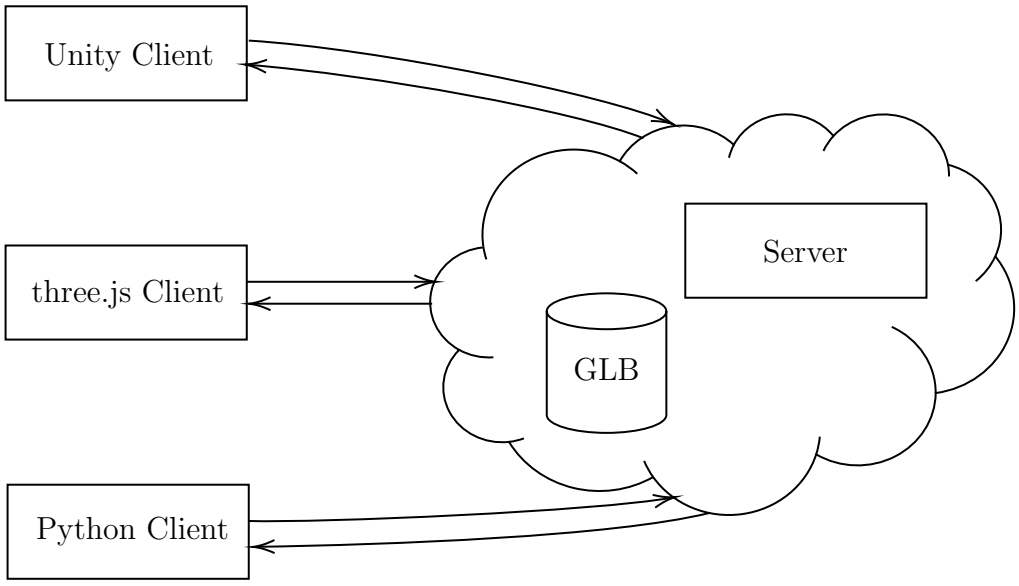


Figure 4.1: Message passing between clients and the server within the system.

In the model described in Figure 4.1, three separate clients running on remote machines transfer and receive packets to and from the server. In this example, the server is running on a cloud service, where the glTF binary file is also located.

4.3 Packet Structure

Messages that are passed between the clients and the server are designed to be as simple as possible. Since this is a bidirectional protocol, it is up to both the server and the client to efficiently and effectively communicate state.

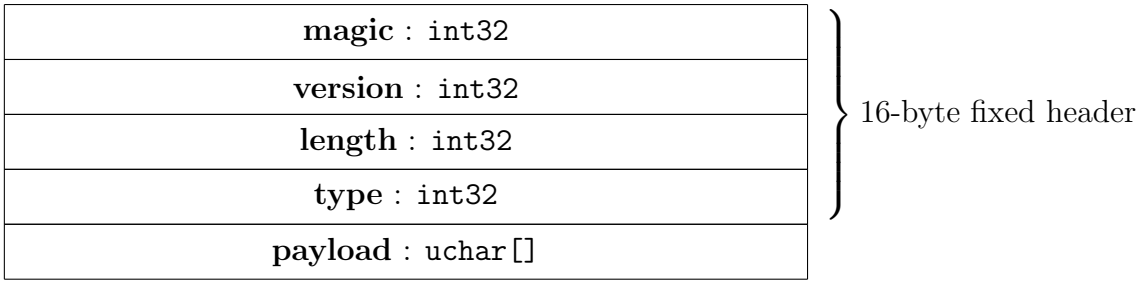


Figure 4.2: HVTP packet layout.

Figure 4.2 describes the packet structure for any message sent using HVTP. Any messages not using this structure will be discarded.

Magic. Describes the binary file. A constant value of 0x48565450, which is the ASCII representation of "HVTP".

Version. Describes the version of the protocol. For this proposal, it is a constant value of 0x1.

Length. Indicates the logical size of the payload.

Type. Describes the type of payload. It is used to signal to the client or server application how the payload should be processed.

Payload. The scene-graph information we would like to transmit - see Table 4.1.

The reader may notice that the fields within the packet do not contain unsigned types. This is not consistent with the glTF format which does include unsigned types in the header. This is not a mistake. HVTP is designed to be as interoperable as possible and an unsigned type such as uint32 may not be supported in the application language. Java is an example of a language that does not support unsigned types.

4.4 Payload Types

Figure 4.1 gives a brief description of the types of payload that can be included in a packet.

Type	ASCII	Payload Description
INIT	0x494e4954	The entire scene as GLB, see Figure 2.2
UPDT	0x55504454	A byte diff of the new changes
TRNS	0x5452534c	Transformation made to a specified object

Table 4.1: Payload description per packet type.

INIT. Once the client connects to the server successfully, it is the responsibility of the server to send the entire GLB file to the client in the payload. This type of message is referred to as an INIT type.

UPDT. As we mentioned earlier, we want to be as efficient as possible when describing changes that have been made to the scene-graph. If a client has modified the scene, such as introduce new geometry or textures, then it is this client's responsibility to calculate the diff of the new GLB file against the original. The diff is then transferred to the server under the UPDT message type and it is then propagated to all other clients in the Hyperverse. The clients and server each merge the diffs into their own copies of the scene-graph.

A diff is calculated as a bitwise XOR operation of the original GLB file against the new GLB file. The result of the XOR operator can be interpreted as the difference of bits across the two GLB files. Theoretically, since the number of changed bits will be less than most of the total bits, we will have more zeros than ones. This can be referred to as a sparse binary string and it can be compressed. The entire UPDT process should be faster than solely using the INIT packet type.

TRNS. Lastly, there is a requirement for this project to support basic transformations for individual objects within the scene. In our specification, we shall define four floating-point values for each of the three transformations we support: rotation, translation and scale. To avoid gimbal lock, we describe any rotations made to the node as a normalised four-dimensional quaternion - see Figure 4.3. To describe scaling and translations, we use a homogeneous coordinate system such that there

are three floating-point values and a constant value of one - see Figures 4.4 and 4.5.

$$\mathbf{q} = \cos\frac{\theta}{2} + (u_x i + u_y j + u_z k)\sin\frac{\theta}{2}$$

$$\mathbf{p'} = \mathbf{qpq}^{-1}$$

Figure 4.3: The quaternion, **q**, can be calculated by a Euler axis unit vector **u** and an angle θ. The final position of the rotated point **p'** is found by the conjugation of **p** by **q**.

$$S(\mathbf{v}) = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.4: Scale matrix, S, by a vector $\mathbf{v} = (v_x, v_y, v_z)$.

$$T(\mathbf{w}) = \begin{bmatrix} 1 & 0 & 0 & w_x \\ 0 & 1 & 0 & w_y \\ 0 & 0 & 1 & w_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.5: Translation matrix, T, by a vector $\mathbf{w} = (w_x, w_y, w_z)$.

This message type must also include a reference to the object within the scene we are transforming. In glTF 2.0, the node hierarchy is not represented as a scene-graph, but a disjoint union of strict trees. Since no node can be a direct descendant of more than one other node, we cannot apply a standard traversal algorithm and to indicate where the node is located in the tree. Fortunately, the glTF specification allows for an optional node "name" field. When the scene is exported to bytes, we can label each node name using a universally unique identifier (UUID).

As we are only transmitting twelve floating-point values and a sequence of chars for each transformed node, we should be improving performance significantly over the UPDT and INIT packet types. The client does not need to modify the scene-graph bytes for this process and instead, we can rely on the built-in vector maths libraries within the game engine. The byte layout for this can be seen in Figure 4.6.

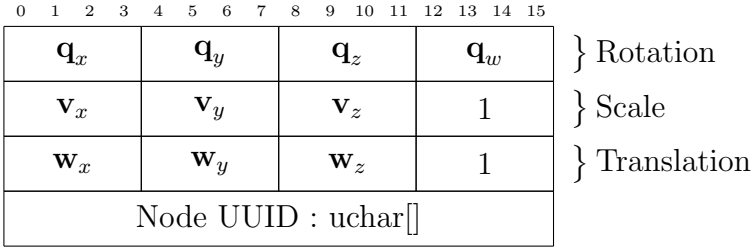


Figure 4.6: HVTP TRNS payload layout in bytes.

4.5 Rate Limiting

The streaming of packets from client to server only occurs once a change has been detected in the client's local Hyperverse. If the user makes a change to the scene, it is detected by a separate process and the relevant packet is created.

One strategy to detected changes in the environment is to perform a depth-first traversal of the client's scene-graph and calculate the global coordinates of the object. If the object is hashable and the generated hash is not based on the object's local transformation, then it can be referenced in a dictionary against its global coordinates. We poll the dictionary at the rate limit to check if the cached coordinates in the dictionary match the recently-calculated coordinates. In some game engines, this process is not necessary and an interface could be provided for the developer.

Since we can only detect changes at the rate limit, we can only send a maximum of one packet per detection. This constrains our bandwidth but ensures that the server is not flooded with packets whilst simultaneously ensuring that our client performance is not impacted. There is a trade-off between the rate limit and the user latency of our CVE. Lower rates (one-quarter of a second) seem to be the most optimum configuration as changes made to the Hyperverse are reasonably in real-time - this is further explored in Chapter 6.

4.6 Protocol Example

We have seen the variety of message types that this project will support. This section will put this into practice and demonstrate the protocol in a simple demo interaction between a server and two clients. One client is written in the Unity game engine and the other is written in Python using an OpenGL renderer.

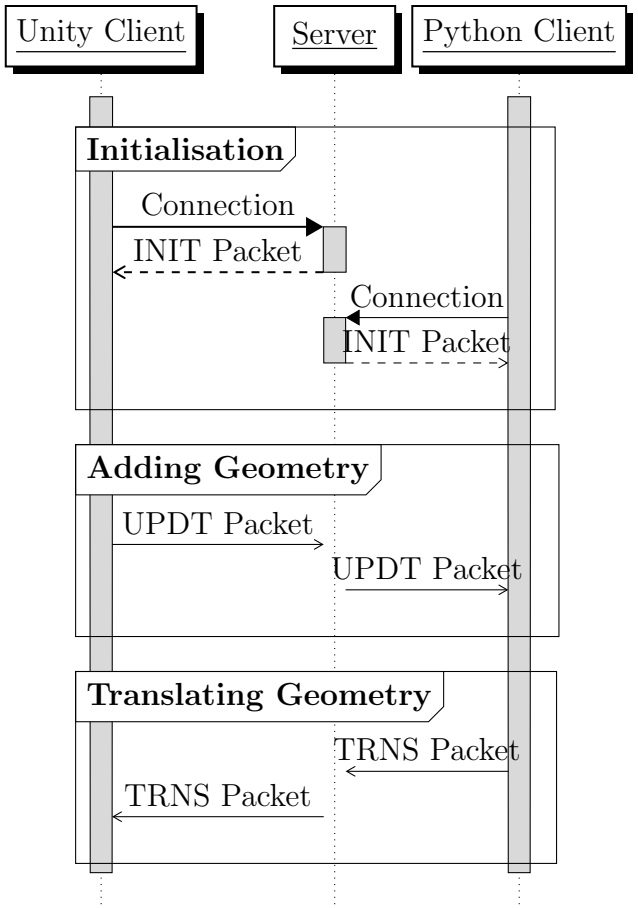


Figure 4.7: Simple sequence diagram interaction between two HVTP Clients and a HVTP Server.

Consider Figure 4.7: assume at time $t = 0$ the server starts up. At $t = 1$, the events inside the **Initialisation** block occur: the Unity and Python clients startup and connect to the server and received the INIT packet containing the entire scene-graph. This is streamed over a TCP connection. At $t = 1$, the events inside the **Adding Geometry** block occur: the Unity client adds a textured cube into the scene. This information is transmitted using the UPDT packet to the server and it is propagated to the Python client. Once both server and remaining clients receive the diff inside the UPDT packet, it is their responsibility to apply and overwrite

their local copies of the GLB scene-graph. Finally, at $t = 3$, the events inside the **Translating Geometry** block occur: the Python client translates the box by an unspecified amount **w**. This change is detected and stored within the TRNS packet and passed to the server where it will be propagated to the Unity client.

Chapter 5

Implementation

All chapters until now have been concerned with research, design and theory. In this chapter, we will focus on the project implementation. We have segmented the next several sections based on the high-level architecture of the system.

5.1 Server

We began by writing the server application. From the previous chapter, we understood that there was no real requirement for the server to be as fast or as efficient as possible. Remember, it is not the server's responsibility to draw the scene-graph or simulate the physics engine, but to receive, process and pass on messages by other clients. Understanding this meant that we were allowed to venture outside of the typical subset of languages for game engines and utilise a programming language more suited for simple message passing tasks. Two languages were considered and tested: Java and Python. Both are used extensively in the software industry due to their portability and support.

The server has its copy of the scene-graph. Any changes that a client sends to the server will also modify the server's scene-graph in addition to the other client's scene-graphs. The main purpose of this is to have a persistent and reliable store of the environment for late joiners, or in the case that the server must restart.

As the server receives messages from clients, it adds them to its internal queue. A separate process handles the processing of messages within the queue, this is referred to as the Packet Queue Processor. Figure 5.1 shows how messages are

aggregated into the queue for processing. The queue is thread-safe and the internal data structure uses a linked list, which typically has a higher throughput than array-based queue.

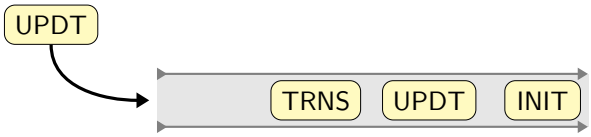


Figure 5.1: The server internal queue. Messages are received by different client processes and added to the thread-safe queue. The Packet Queue Processor polls the queue for any messages and performs the designated action within the message upon the scene-graph.

Once these critical changes are processed, the new scene-graph is recalculated and stored to disk. The server then re-transmits the changes to all of its connected clients. Any clients that receive this new scene-graph is then responsible for redrawing the scene to represent this new information.

5.2 Client

Before work began on implementing the protocol within the clients, we spent a few weeks of exploration to survey the plethora of rendering libraries and game engines that existed. Many were tested, such as Unity, Python, Unreal Engine 4 and three.js. Ultimately, we decided that the most timely way of proceeding was to pick two distinct client frameworks and focus on the protocol implementation. Ultimately, we chose a Unity and Python client application to form the test for our protocol.

The Python client utilises a popular open-source rendering framework called Pyrender [29]. We have not yet discussed Pyrender in our research. It is a lightweight rendering library that supports the glTF specification and used extensively in machine learning applications. Paired with this is Trimesh, another popular loader for Python that supports the loading of glTF and GLB assets over a binary stream [7].

By selecting Unity and Python clients, we can demonstrate that 3D graphics can be shared between two very different languages, runtimes and rendering APIs. We then spent a few days developing the base functionality of the client, such as

the routines to receive a message, pass it to the renderer and draw the scene-graph. After this, we were ready to plug-in the protocol specification that we outlined earlier to communication with our server.

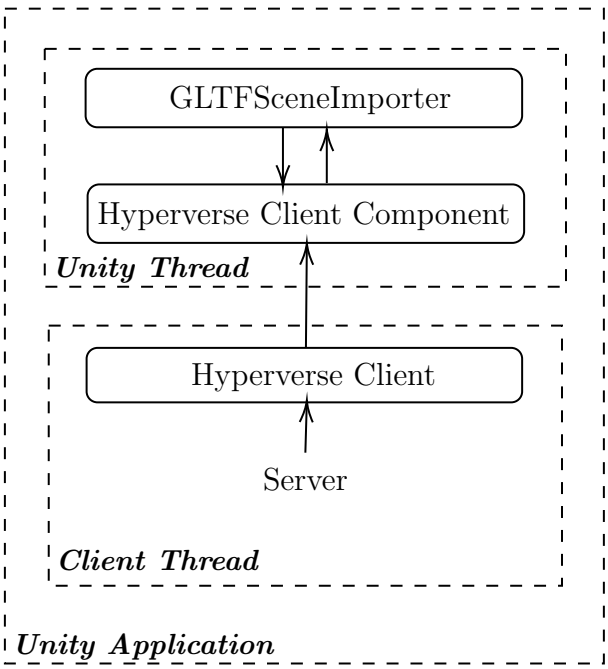


Figure 5.2: A simplified data flow diagram of the Unity client.

Figure 5.2 represents the flow of information between the various components of the Unity client application. The Hyperverse Client first receives packets from the server. In the Hyperverse Client, the raw bytes are deserialsied to a Packet object and a callback delegate to the main Unity thread is called via the Unity Main Thread Dispatcher. The packet now resides in the Hyperverse Client Component, a Unity Component class that is responsible for converting the payload into the Unity scene-graph as per the packet type.

The GLTF Scene Importer is an external class from the UnityGLTF library [45]. As the callback is ran, the importer runs as a coroutine.

The Hyperverse Client is bidirectional and can be used to communicate messages back from Unity to the server. Instead, the Hyperverse Client Component uses the GLTF Scene Exporter to convert the Unity scene-graph to a GLB byte array. This array is then serialised with the relevant header and sent via the Hyperverse Client.

5.3 Protocol Prototype A: INIT Packet Support

The first development cycle aimed at building two clients that communicated solely using the INIT packet variant. In Figure 5.3, we demonstrate that both Unity and Python clients can successfully connect to the HVTP Java server and gain the initial scene information required to start a CVE. Furthermore, the figure above also demonstrates the sharing of graphics between clients. In this particular demo, the Unity user hovers over the cube primitive and the cube texture is changed to a well-known university's logo. This interaction is reflected in the Python client, as seen on the right.

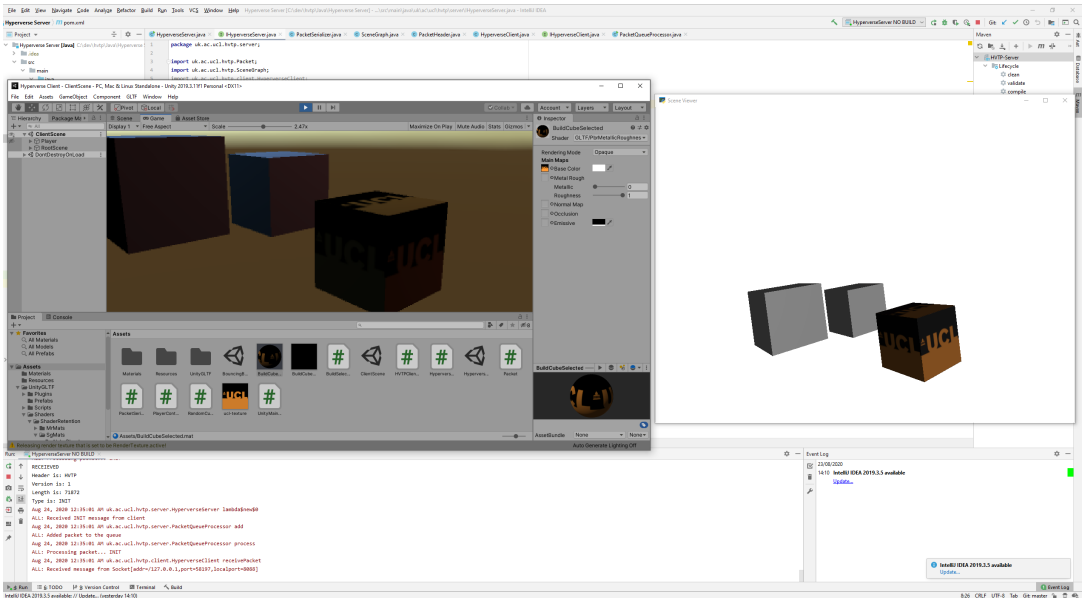


Figure 5.3: Background: HVTP Java server. Foreground: Prototype A of Unity client (left) and Python client (right).

The sharing of textures isn't as simple as it initially seems. This texture must be defined within the GLB payload. In order for us to export this texture, we had to rely on the glTF PBR Metallic Roughness shader extension provided in the UnityGLTF library.

5.4 Protocol Prototype B: UPDT Packet Support

We have seen a "brute-force" way to share graphics, but passing the entire scene-graph at every interval is not ideal. Diffs may introduce efficiency into our protocol.

This development cycle focused on solely distributing changes in the scene-graph after client initialisation. During the implementation of this packet type, we realised that the accuracy of the diff was most important. If a single byte was misaligned, then the glTF parsers found in either of the clients could fail and the client could disconnect from the server. After the binary diff was calculated, the result was compressed using gzip. gzip is based on the DEFLATE algorithm, a lossless data compression algorithm that uses a combination of LZSS and Huffman coding. It is supported in C#, Java and Python and is used extensively in web protocols.

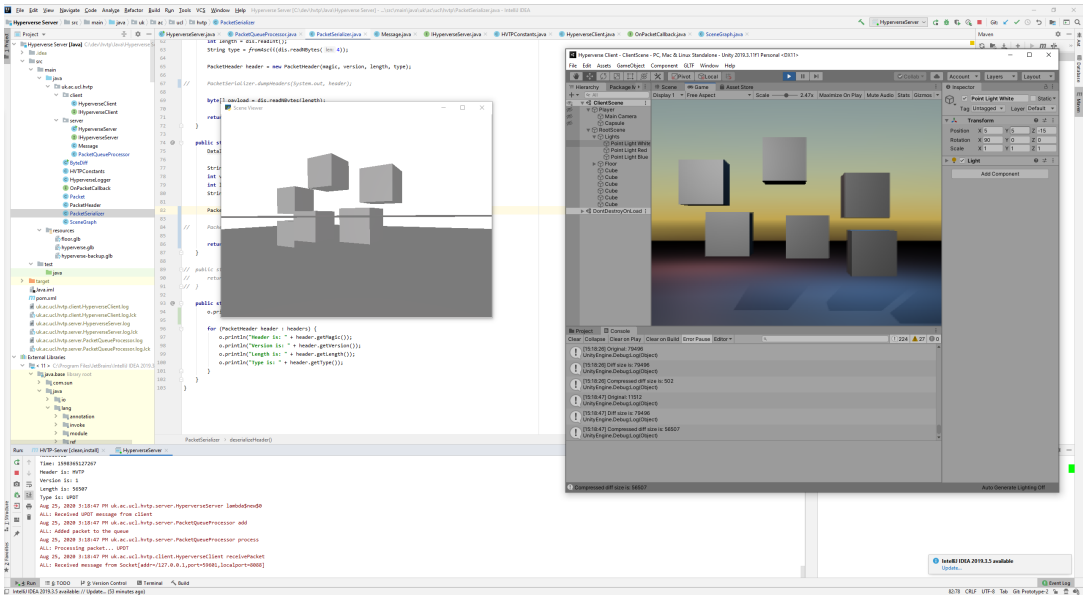


Figure 5.4: Background: HVTP Java server. Foreground: Prototype B of Unity client (left) and Python client (right) before insertion of the new cube.

In Figures 5.4 and 5.5, we see a similar scene to before. However, there is a plane representing the floor and various point lights - we refer to this scene as the "Basic Floor" scene. The two clients exist within the CVE at opposite ends to one another. In the frame before, we see the six cubes that were placed previously. We then use the Unity client to interact with the environment by adding another cube closer to the Unity client camera. In the Python client, this new highlighted cube can be seen from afar behind the existing cubes. We have slightly modified the lighting of the scene here, which is not defined in the scene-graph but in the game engines. In this case, we have boosted ambient intensity by around a factor of ten. In the case of the Python client, there is a small issue with the renderer where the ground plane has been duplicated at a higher altitude. The reasons for this are unknown.

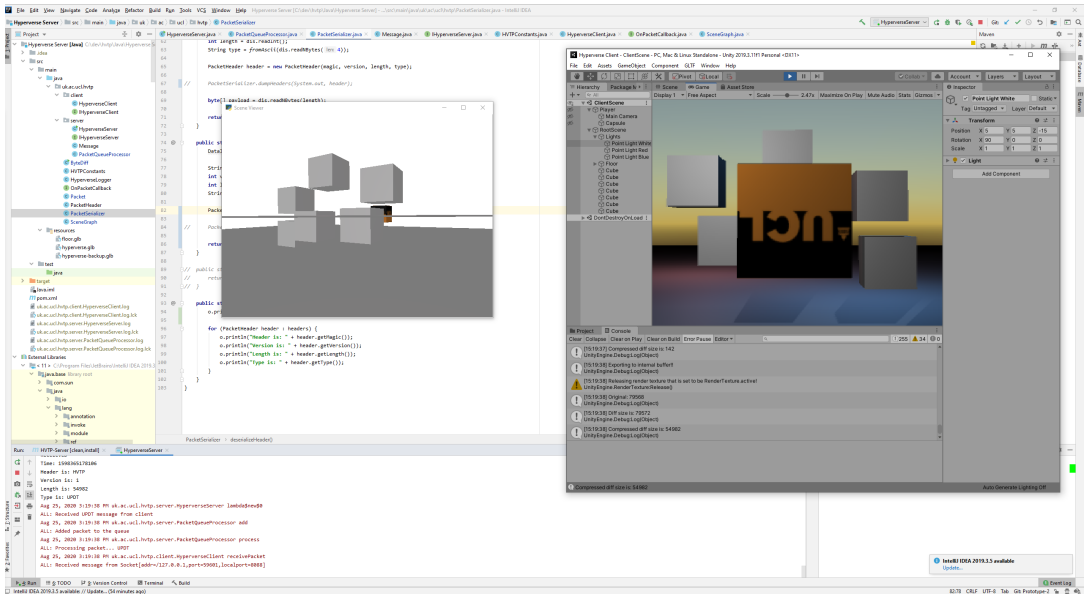


Figure 5.5: Background: HVTP Java server. Foreground: Prototype B of Unity client (left) and Python client (right) after insertion of the new cube.

In this particular example, we can see in the console that our packet size has reduced from 79568 to 54982 bytes, which is over a 30% reduction in size from the INIT packet. These results appear promising, but it is not enough to observe one or two log messages. In the next chapter, we extensively test this new packet type.

5.5 Protocol Prototype C: TRNS Packet Support

For this development cycle, we aimed to take a step-back from the GLB file. From the early stages of profiling, we understood that a significant amount of client-side CPU time was spent in deserialising, re-parsing and re-forwarding bytes from the heap to the graphics card. Simply sending and processing the change of pose of a particular node within the scene-graph was more efficient.

For this, our payload had to be more structured. It could no longer be an entire binary blob, but segmented floating-point values to represent the required transformations (see Figure 4.6).

In this prototype, each client has an internal dictionary, where the key represents the UUID for each node within the GLB file and the value is a reference to the geometry of the node within the client scene-graph. As clients received TRNS packets, they performed a look-up of the node using the UUID and applied the

relevant translations.

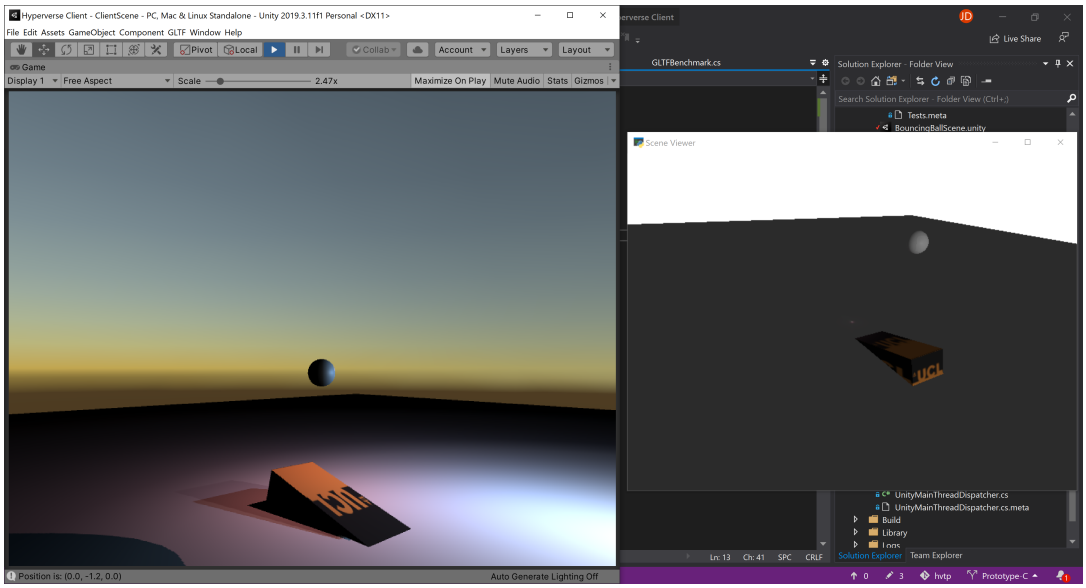


Figure 5.6: Prototype C of Unity client (left) and Python client (right).

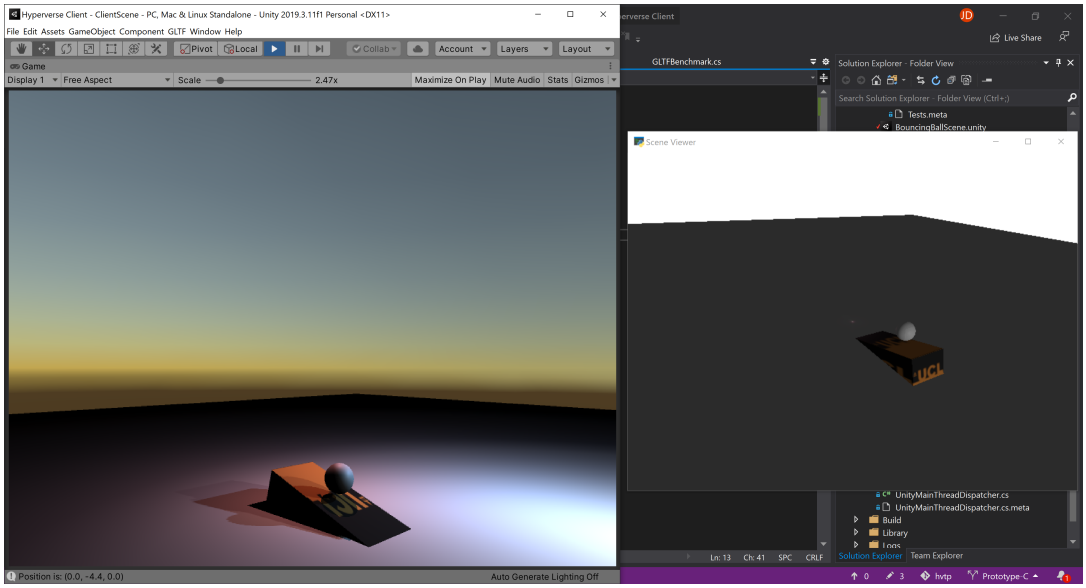


Figure 5.7: Prototype C of Unity client (left) and Python client (right).

The Figures 5.6 and 5.7 demonstrate a ball falling from the sky. In Unity, this ball has a Rigidbody physics collider attached with a mass of 0.5 units. As it falls, the change of coordinates is detected and sent using the TRNS packet to the Python client. As the ball hits the ramp and rolls away, this is also detected and shown in the Python client. From the perspective of the Python client, this object has all of the behaviours and characteristics of something that inherently exists in Unity, it is

just rendered using a completely different application.

Something we did not consider up to now is the interoperability of coordinate systems. Since Unity uses a left-handed coordinate system and glTF uses a right-handed coordinate system, the orientations of objects were flipped after transmission. This was simply rectified by negating the x, y and z components and swapping the y and z.

5.6 Protocol Prototype A2: WebSocket Support

In our initial research, we extensively discussed the wide-support of 3D graphics on the web. We understood that this is not to be overlooked. Within a relatively short period of time, we implemented support for a WebSocket extension within the server in addition to a client written in the three.js JavaScript framework. This client solely supported the INIT packet type.

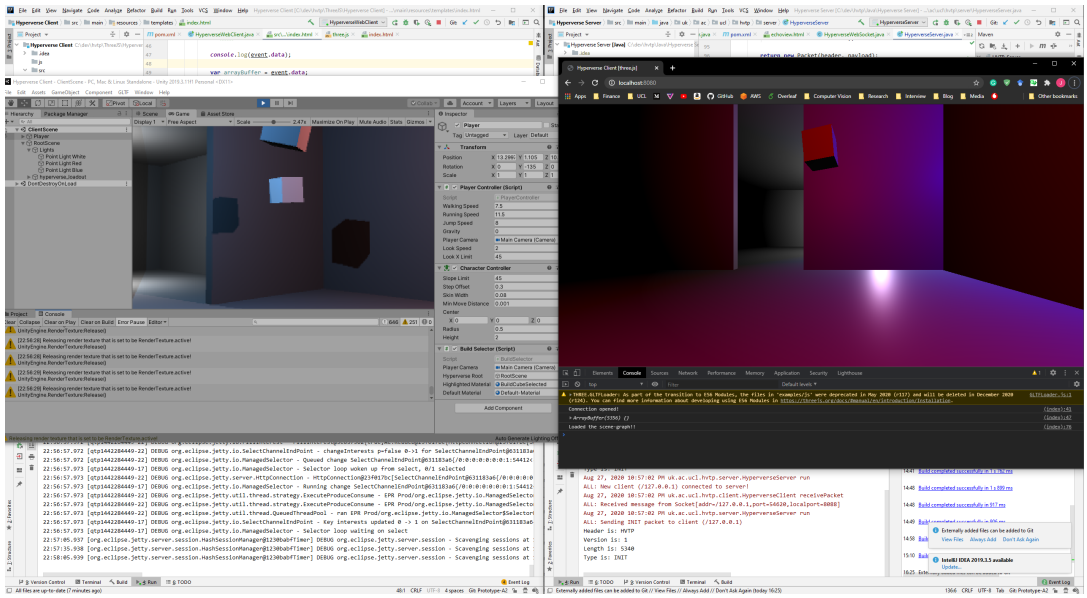


Figure 5.8: Prototype A2 of Unity client (left) and three.js client (right) using the "Cube Rooms" environment.

Figure 5.8 depicts this transfer of 3D information between web browsers and native applications. We attempted to recreate the same lighting environment in the browser as seen in Unity, by copying over the same point light parameters, but this did not produce the same effect. Since three.js is supported on a variety of browsers, there is an opportunity to render the scene-graph on multiple operating systems and

architectures.

In this prototype, we also introduced a more visually appealing scene referred to as "Cube Rooms". This was a simple environment composed of two rooms. The rooms were occluded by wall with a passage. One room contained several randomly-generated cubes, each with a random rotation and position. The Unity client was able to interact with these cubes by moving them between rooms. This environment can be seen in Figure 5.8.

The development of Protocol Prototype A2 was entirely experimental. We wanted to see how well our messages would work with the byte reader API on an entirely different runtime. If there was additional time, we could have investigated opportunities to speed up this web client through the use of the Streams API, Wasm and use of UPDT and TRNS message types.

Chapter 6

Testing and Results

From our development, we produced three prototypes, with each prototype being theoretically faster than the last. In this section, we attempt to verify if our assumptions hold by comparing the relative performance of these artefacts. We first look at the bandwidth of performing two actions within the Hyperverse: the insertion of objects and the transformation of objects between client and server. After this, we look at the end-to-end speed of our CVE by measuring the time, in frames, of an object being transformed from one heterogeneous client to another.

6.1 Bandwidth Comparison of Prototypes A and B: Object Insertion

In this testing strategy, we compared Prototypes A and B to measure the bandwidth of inserting textured geometry into the scene. Since the TRNS packet did not support the delivery of buffer-based objects, it was not necessary to include the packet type in this investigation.

Figure 6.1 shows our setup: we started up the Unity client and gave the controls to an independent actor. Upon startup, the server transmits an INIT packet to the client to transmit the initial GLB scene information - this was ignored in this investigation. The demo environment was the "Basic Floor" scene from the previous chapter. In our Unity application, the insert key was mapped to a script which inserted a textured cube (shown in Figure 5.4) into the scene. The client would pick up the changes in the scene and transmit this information back to the server.

Figure 6.1: An independent actor performs a stress test on the Unity client.

Prototype A used the INIT packet to do this, whereas Prototype B used the UPDT packet. Our goal was to measure whether Prototype B's UPDT packet was more efficient in conveying the change of geometry in the scene than Prototype A's brute-force INIT packet.

Protocol	**Payload Size (bytes)**			
Prototype	**Mean**	**Compressed Mean**	**Minimum**	**Maximum**
A	76056.08 ± 25584.06	-	12032	95776
B	89078.76 ± 5310.90	51827.43 ± 18316.79	1961	68296

Table 6.1: Mean, compressed mean, minimum and maximum payload size of each prototype for object insertions. Averages include standard deviation. Values rounded to two decimal places. Minimum and maximum values for Prototype B are compressed values.

We let the independent actor explore the scene and insert geometry wherever they pleased. The actor also had access to the Python client, where they saw the changes distributed. The server application was slightly modified to include a logging system which recorded all incoming packets to disk. The logger recorded the size of the packet's payload (in bytes), and the time of arrival (in milliseconds). For Prototype B, both uncompressed and compressed sizes were recorded. Afterwards,

the log file was parsed to a spreadsheet and analysed. Table 6.1 provides a summary of the mean values from our results from all two-hundred messages per prototype.

To summarise, our independent variable was the packet type (INIT or UPDT). Our dependent variable was the size of packets over time. Our constant variables was a client rate limit of 4 packets/second (0.25 seconds per packet), a fixed-header size of 16 bytes and the same GLB INIT packet to kick-start each client.

Prototype A. Our mean payload for Prototype A is approximately 76 kilobytes. This is excluding our fixed-size header which is always set at a constant value of 16 bytes. For this mean value, our standard deviation is approximately 26 kilobytes. This indicates a high variation between sizes of payload over time. Figure 6.2 shows a gradual increase of size over time. However, at certain points of time during our experiment, the size of the payload drastically fell. The smallest payload size in the series is 12032 bytes.

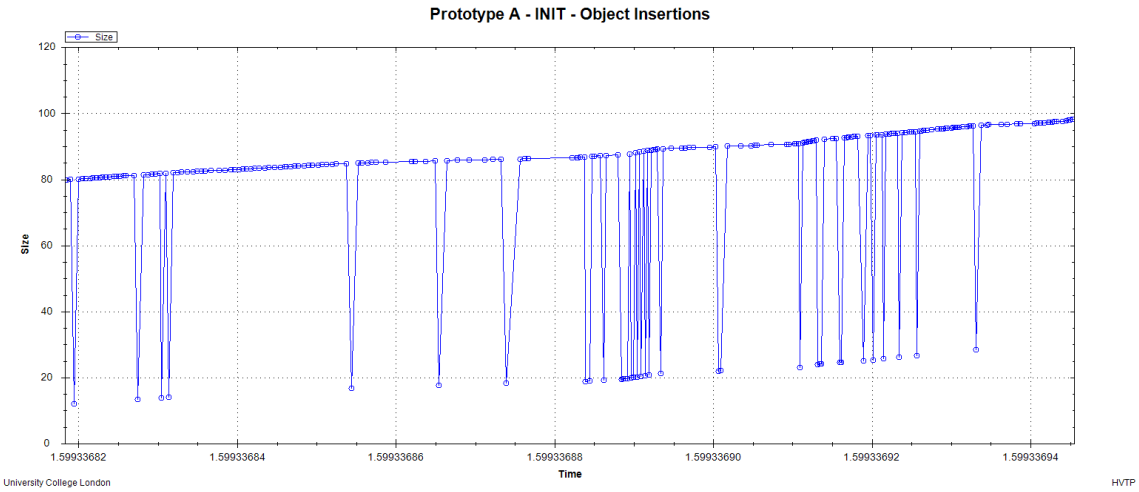


Figure 6.2: Change in payload size (bytes) over time (milliseconds) for Prototype A - object insertions.

The gradual increase in size makes sense. As the actor adds more cubes into the scene, the GLB file must represent all of this additional geometry in the scene description and the buffered data. However, the occasional drops in size over time is fairly regular throughout this experiment. The reasons for this are currently unknown.

Using this mean value, we can come up with a rate of bandwidth using the rate limit. Since the rate limit is a constant of 0.25 seconds per packet, we estimate that the bandwidth for object insertions using Prototype A is 304224 bytes/second or

304 kilobytes/second.

Prototype B. Our mean compressed payload for Prototype B is approximately 52 kilobytes. This is a decrease in over 30%. We also made sure to track the uncompressed mean size of the payload. Since the payload is not the GLB, but a diff of the new GLB against the server's GLB file, its size should be slightly greater than Prototype A's mean payload size. This is true, the mean for our uncompressed size is 89 kilobytes with a standard deviation of around 5 kilobytes. Since our standard deviation is much lower, it indicates a more uniform distribution of sizes. This decrease in variance is attributed to our method of calculating diffs. The XOR operator is applied to both the old GLB and the new GLB file. However, if the new file size is smaller or larger than the old file size, this must be indicated in the diff as a removal or addition of additional bits. Therefore, the size of the GLB diff is always the maximum size between the new and old GLB file sizes.

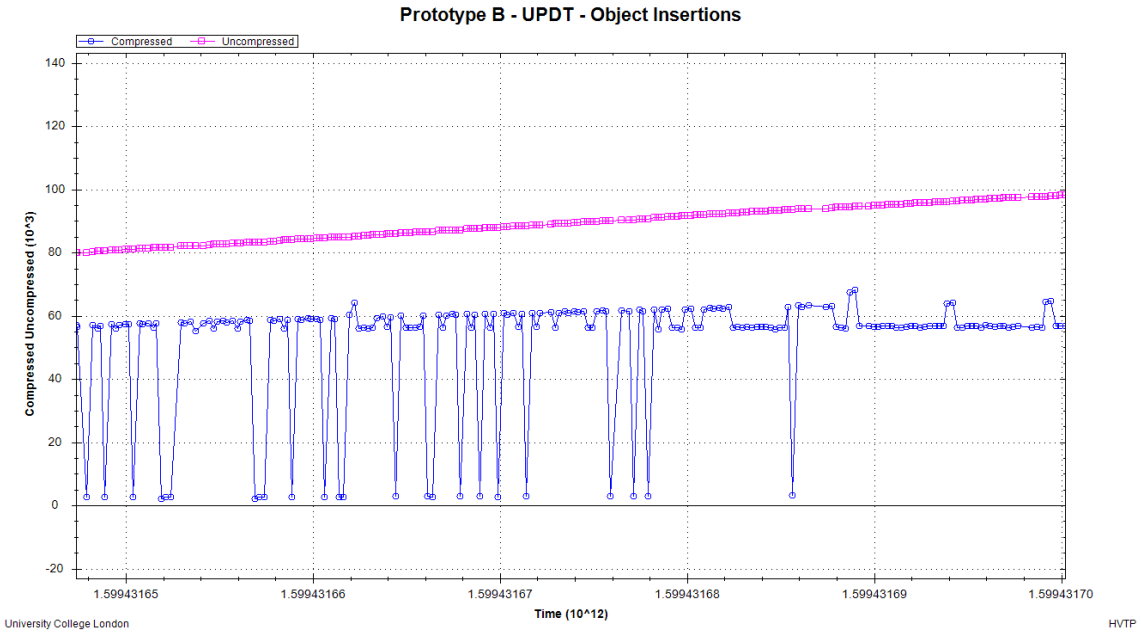


Figure 6.3: Change in compressed and uncompressed payload sizes (bytes) over time (milliseconds) for Prototype B - object insertions.

Figure 6.3 shows a comparison of compressed and uncompressed sizes across time for object insertions. We can see that uncompressed sizes are much less varied due to the method of calculating diffs. Strangely, the random drops in sizes from Prototype A is also still present in the compressed size. The size of our diffs do not contain these high levels of change, which means the variation must be attributed

to the gzip compression algorithm.

Using the compressed mean, we can generate a bandwidth estimation for Prototype B of approximately 207319 bytes/second or 207 kilobytes/second. This is around a third less bandwidth usage than Prototype A's.

6.2 Bandwidth Comparison of Prototypes A, B and C: Object Transformations

Protocol Prototype	**Payload Size (bytes)**			
	Mean	**Compressed Mean**	**Minimum**	**Maximum**
A	54874.50 ± 32536.3	-	11116	78968
B	79033.58 ± 138.61	30471.99 ± 27282.32	76	59473
C	84 ± 0	-	84	84

Table 6.2: Mean, compressed mean, minimum and maximum payload size of each prototype for object transformations. Averages include standard deviation. Values rounded to two decimal places. Minimum and maximum values for Prototype B are compressed values.

This testing setup is similar to the last. Instead of measuring the size of inserting geometry into the scene, we focused on the transformation of objects. Using the mouse pointer, the actor was instructed to spend the allotted time in each prototype transforming the pose of existing geometry. Packets from each of the three prototypes were recorded by the server and analysed after in a spreadsheet. Table 6.2 provides a summary of the mean values from our investigation from all two-hundred messages per prototype.

Prototype A. Our mean payload size for Prototype A is almost 55 kilobytes. This is less than our prior experiment in the previous chapter, which is to be expected since there should be no changes in size for the scene description and buffer-based data. However, the standard deviation is higher at around 32.5 kilobytes. This change in variation is evident in Figure 6.4.

The change in payload size almost appears to oscillate between two the minimum and the maximum values. The lower range is close to around 11 kilobytes and the

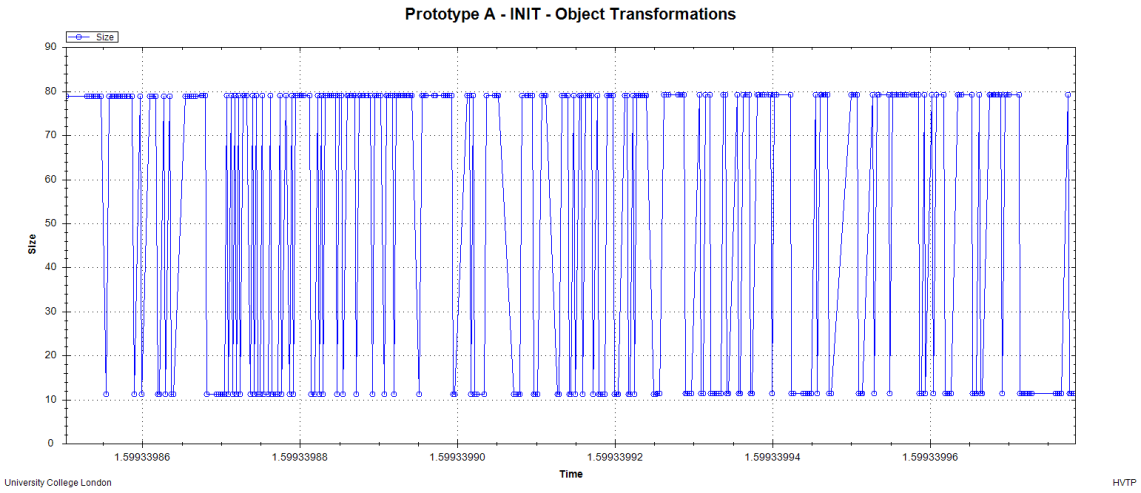


Figure 6.4: Change in payload size (bytes) over time (milliseconds) for Prototype A - object transformations.

upper range is close to 79 kilobytes.

For Prototype A, our mean estimated bandwidth use is 219498 bytes/second or around 219 kilobytes/second.

Prototype B. Our mean payload for Prototype B is approximately 30 kilobytes for compressed payloads. For object transformation, this is a change of around 45% in comparison to Prototype A. More impressively, there were several packets with a size of under 84 bytes. This means the diff of the GLB file is less than Prototype C's. Some more investigation is required to understand why this is. It is possible that the diff did not pick up any changes and instead it was a compressed file containing null values.

Figure 6.5 shows the results of compressed and uncompressed payloads for object transformations. Again, large variations of compressed sizes can be observed whereas uncompressed diffs had little variation, with a standard deviation of 139 bytes.

For Prototype B, our mean compressed estimated bandwidth use is down to 121888 bytes/second, or around 122 kilobytes/second. This is a decrease of over 44%.

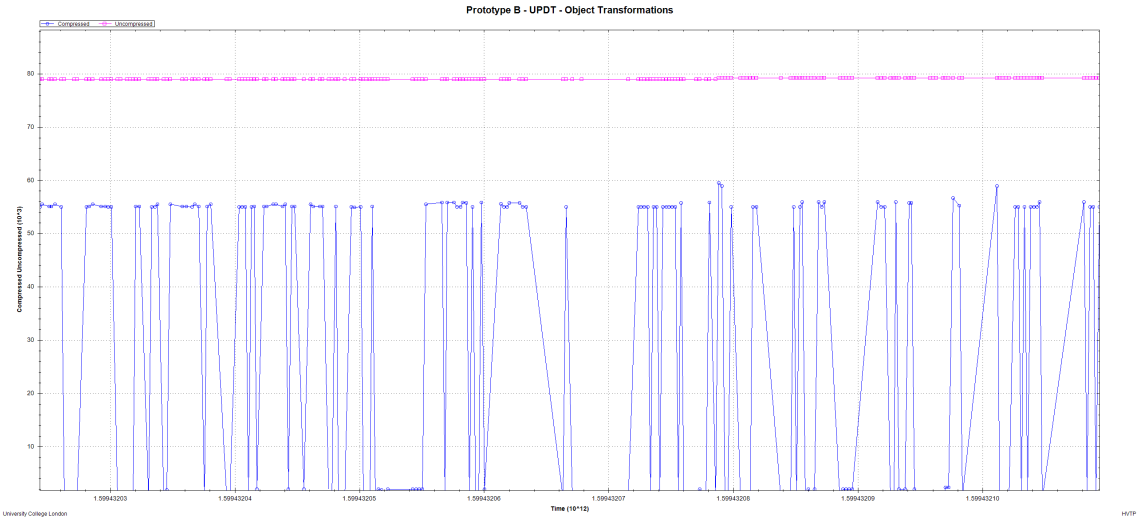


Figure 6.5: Change in compressed and uncompressed payload sizes (bytes) over time (milliseconds) for Prototype B - object transformations.

Prototype C. Our mean payload for Prototype C is exactly 84 bytes. This is expected as we are not relying on the GLTF Scene Exporter and instead using a fixed packet format. To recap, there are 16 bytes allocated for the rotation quaternion, 32 bytes are allocated for the homogeneous scale and translation vectors and a final 36 bytes is allocated for the ASCII representation of the UUID (32 hex chars and 4 hyphen chars). This is significantly cheaper than anything we have tested so far. Thus, our final bandwidth rate for this packet is 336 bytes/second or 0.34 kilobytes/second.

6.3 End-to-End Testing using Frame Counting

So far, we have only considered the size of the payloads. In this test, we test the end-to-end communications of our system. If an individual was to transform an object within the shared virtual environment, how long would it take, in frames, for these changes to be reflected in the other client?

This test set out to answer this question. In Figure 6.3 we have the time in frames taken for a cube inserted into the CVE to propagate to the Python client. We used an iPhone Xs to record the screen at 240 frames per second.

In our results, we found that Prototype B took the longest to show the changes. This may be because although we save time on transportation between clients, there

Protocol Prototype	Unity Frame Index	Python Frame Index
A	0	186
B	0	241
C	0	96

Table 6.3: Frame delay after transforming a cube in the Hyperverse via the Unity client.

are many steps to prepare the data. Prototype B first must get the new GLB bytes and compare them to the old GLB file. Next, it must apply a gzip compress and then send the data. This process is then reversed in the Python client, where the data is decompressed, deserialised and the changes are calculated.

Prototype A's INIT packet was second. The process of preparing and extracting information from this message is fairly straightforward.

Lastly, since we are not directly managing the bytes of any scene-based representation in Prototype C, we save hundreds of precious frames. There is very little to transport and there is very little to do with the TRNS packet. On the client side, we are simply performing a hashmap lookup and setting the new transformation variables. Since there are only ten variable to set, this is not a problem for the Python client.

Chapter 7

Discussion

In Chapter 6 we outlined our testing strategy and gave a summary of our results. From these results, we can begin to discuss what elements of our prototypes to carry forward towards a finalised draft of our specification and what we can improve upon. We look back to the criteria that we initially set out in Chapter 3 and see how fit for purpose our protocol is.

Action	**Prototype**	**Bandwidth (kilobytes/second)**
Object insertions	A	304
	B	207
Object transformation	A	219
	B	122
	C	0.34

Table 7.1: Summary of bandwidth usage from Chapter 6 with a rate limit of 4 packets/second.

Table 7.1 provides a summary of our bandwidth usage from each prototype and from each action. The result are promising and we see a consistent decrease of memory usage through our optimised prototypes. However, as we implemented these packet types, we understood that there was much more scope for improvement and future packet types that we could not fit into our original plans.

7.1 Prototypes

In our experiments, we set a relative baseline size of approximately 30 kilobytes for the INIT message in Prototype A. Although there is room for improvement, we believe the INIT packet would be a part of a final specification for our protocol. It is currently the only way to deliver the entire scene. The INIT packet has the added benefit of overwriting the current state of the client. This functionality fundamentally acts as an atomic operation and can be useful for certain scenarios to reset state, such as upon start-up.

We decided to keep the initial GLB scene file a constant in our experiments. However, we must consider the viability of much larger GLB files, with many hundreds or thousands of assets. Could a packet designed in such a specific way work for these kinds of environments? If so, what would be the frame delay? There is much more experimentation yet to be done. After discussing this with Steed, he suggested the plausibility of splitting up the GLB file into separate binary components and streaming them individually. We could theoretically introduce networked parallelism into our system. Furthermore, there could be an opportunity for a tile-based loading system. Priority could be given to nodes in the scene-graph. Objects that are closest to the camera could be streamed in well before objects in the distance are. These are simply a few of the optimisations that could be made.

In Prototype B, we introduced the UPDT packet which was a diff of the new changes a client had made to the scene-graph. Although the compressed size decreased bandwidth usage, the uncompressed size was greater than the INIT packet. We initially believed that applying the XOR diff method would compress at a much better size than applying a compression to just the GLB file. We drew this assumption knowing that GLB files are reasonably homogeneous, with a similar header, data, and scene description byte layout. The XOR of two very similar binary strings would be a sparse binary string, with many zeros. Our initial thoughts were that this sparse binary string would compress to around 50 to 60% of the original size. However, this was not true, and we saw compression rates of over 30% for object insertions.

In the next iteration of this prototype, we recommend introducing chunk-based diffs. This is where we break up diffs based on the chunk within the binary glTF

payload. A chunk can represent the scene description or the buffered data. We can insert information about the chunk we want to modify by modifying the packet headers. This would be a much more elegant optimisation. In the CVE, either the vertices of a mesh are changed or the scene description is changed. Thus, diffing the entire file would not make sense. These new diffs could also benefit from a compression library that is optimised for meshes or textures. Draco [11] is a mesh compression library by Google that is used in glTF for this very purpose. We may even consider writing our own compression library optimised for binary glTF files since we know the schema.

In Prototype C, not only is 84 bytes a much more compact representation of the pose of objects within our scene, but it is also much easier to process by the client. The UUID value can be used to look up the relevant node within the scene-graph using a dictionary data structure in constant time. Furthermore, something that is ubiquitous in the client's scene-graph is the ability to change the transformation matrices of the parent node. This led us to understand that the future of this protocol must take a much deeper consideration for the format we are transporting. This means a "one size fits all" approach should not be used, such as running an XOR diff over the entire payload. However, the TRNS packet is not a new concept. As we have discussed, multiplayer games have been using similar strategies to transfer the location of data for decades.

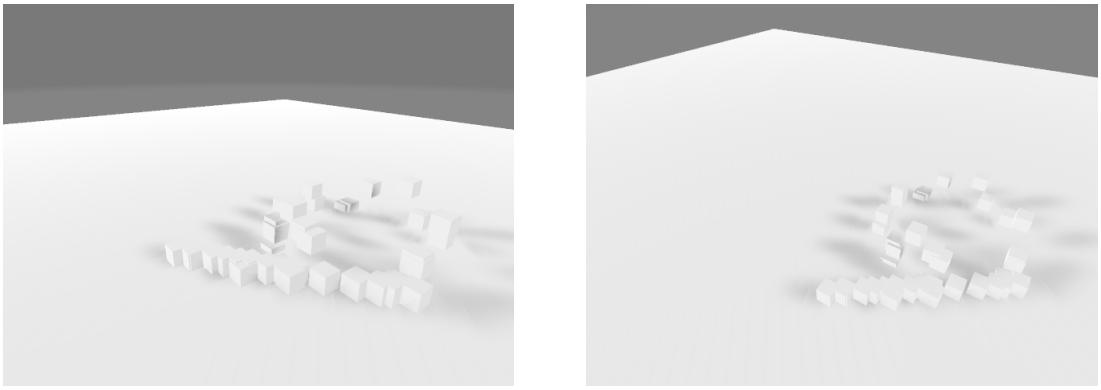
We saw a vast improvement in bandwidth for object translations in Prototype B vs Prototype C. In Prototype B, when an object was translated, we had to rely on the GLTF Scene Exporter and GLTF Scene Importer. By bypassing this serialiser, we improved end-to-end frame performance in addition to reducing memory. After speaking with Steed, we feel that the same idea could be brought to modifying existing geometry. Although in the project we did not yet look at live editing of existing geometry, we know that it is theoretically possible by using diffs via the UPDT packet. We know this because in Chapter 5 - Prototype B, we demonstrated the transfer of textured data by hovering over the cube in Unity and seeing this in the Python client.

In the future, a new packet type could be proposed that allows the developer to index into buffered data of the GLB file and modify the index or vertex array

buffers in real time. The packet would be similar to the TRNS packet, where only the modified indices or vertices are passed. When the client receives this data, we could bypass the GLB serialiser and directly modify the geometry using a concept in Unity known as Procedural Mesh Geometry (PMG). PMG allows us access to an object's mesh geometry at runtime [34]. We could use the same technique to reference an object by using a UUID.

7.2 Bottlenecks

As we have described, there are inconsistencies when looking at the size of the output of the scene using the GLTF Scene Exporter (see Figure 6.2). We sought to find out why by saving any outlying packets (less than 20 kilobytes) to the server's disk. We gathered packets of a median size and compared them to these outliers using an independent 3D GLB viewer: Windows 3D Viewer. Figure 7.1 shows a comparison of the outlier and the median payload.



(a) GLB scene of 15 kilobytes (b) GLB scene of 82 kilobytes

Figure 7.1: The two images represent two scenes that have been exported using the Unity GLTF Scene Exporter by the Khronos Group [45]. The left scene is unusually small at 15 kilobytes. The right scene is fairly regular at 82 kilobytes. Both of these scenes are rendered using Windows 3D Viewer.

The smaller (a) scene was detected as an unusually-sized scene (approximately 15 kilobytes) by the server and dumped to the disk. After this, the server found the next normally-sized (b) scene (approximately 82 kilobytes) and also dumped this to disk. From Figure 7.1, we can see that there are no real visual differences whatsoever.

These scenes are not the same, so we cannot rule out a non-deterministic exporter. However, it is fairly unusual to have such high-variations in size for scenes that very much look the same. Because of this, we suggest that future iterations of this protocol should use a custom exporter. Doing so would simultaneously mitigate the risks of inconsistent export sizes whilst allowing the GLB file to have a deeper integration into our protocol.

Some changes could also be made to rate limiting to improve client-server performance. After speaking with Steed, he suggested that the robustness of the networking protocol could be improved by introducing a networking concept known as Levels of Networked Self-Awareness [13]. Steed went on to suggest that perhaps the rate limit should not be fixed and rather become dependent on factors such as the speed of the networked connection, the size of the payload, and so forth. This would be a welcome addition to our existing architecture since we are dealing with such diverse rendering clients and potentially large packets. We may even experiment with the idea of Networked Self-Awareness into a Verse-inspired (UDP) variant of our protocol.

Finally, something we have not considered is concurrency in the Hyperverse. Our implementation of the protocol does not consider the possibility of multiple individuals editing or transforming the same objects within the scene. This, of course, is a crucial safety measure in collaboration over a network. Existing collaborative systems have tackled this problem differently. In version control systems (VCS) such as Git, the concept of conflicts and merges exist. In graphical applications, DIVE utilised partial database replication. 3DRepo utilised a similar method as Git by introducing explicit and implicit conflicts. A future draft of HVTP must take into consideration how this would work and more research into this is required.

7.3 Implications of Shared Scene-Graphs

What is perhaps the most fascinating concept in this project are the visual implications of scene-graph-based communications. From our prototypes, we see the difference of quality between Unity and Python rendering techniques. Although these scenes contain the same geometry, the same pose for each geometry, the same

textures, etc. they do not look or feel quite alike. The Unity rendering system provides a plethora of high-level rendering techniques. These are rendering parameters that are defined outside of the GLB file, such as soft shadows or dynamic point lights. The questions that arise from an inconsistently rendered VE is, what effect does this have on each user in the CVE?

This relates back to Chapter 2, where we mentioned that the current glTF 2.0 specification does not represent an entire scene-graph. If our transport format doesn't support a particular rendering parameter, it is then up to the game engine. Thus, unspecified parameters will lead to visual inconsistencies between heterogeneous applications. Much more work is required on the effects on user-level immersion from these inconsistencies.

These implications also spread to the non-visual elements of our system too. The physics engine is an important part of most VR applications and graphical applications in general. Since most game engines use a custom physics engine, the behaviours of the objects within the CVE may appear unusual to some who are using a different game engine to render the CVE. For instance, in Chapter 5 - Prototype C, we demonstrated a ball falling on to the ramp and rolling down. The ball falling and colliding against the ramp was calculated by the Unity physics engine, these calculations were represented by the behaviours of the ball bouncing and rolling due to gravity and other parameters such as planar restitution and friction. Unfortunately, our Python client did not have a physics engine. However, if we chose a client that did, such as Unreal Engine 4, then there may be inconsistent behaviours in the CVE. In our protocol, if a user interacts with an object, it is that user's physics engine that must simulate the corresponding calculations for the resulting behaviours of the object. If the user dropped a ball onto the ramp using an Unreal physics engine, with different parameters for gravity, friction and restitution, then these differences may be noticed by the users in the CVE. Some extensive user studies may be required to fully know the impact of this on the end-user.

An added benefit of these shared graphically-represented behaviours is that the sharing of code might not be necessary. The Python client does not need to execute any code when it renders the ball rolling down the ramp. Other CVE implementations we have discussed may have taken the approach of distributing the behaviour

of the ball in some compiled language and executing it on the Python client's host machine. Again, there are many downsides to this, the main one being the security risks of running unchecked code.

Chapter 8

Conclusions and Future Work

We first began this project by looking at the decades-long history of collaborative virtual environments. As networking and graphics technology improved over time, so did the quality of these CVEs. As we continued, we discovered the representation of 3D information as a scene-graph and how it has excelled in the last decade. In the chapters following this, we amalgamated these constructs and started to develop a proposal of our own CVE - the Hyperverse.

We knew from our background study that performance was a priority, so we devised three protocol prototypes to compare and contrast performance at runtime. These are not three separate prototypes, but three required forms of communication within the Hyperverse. The INIT packet intends to deliver the initial 3D information from the server to the client upon startup. The UPDT packet intends to deliver changes that have been made to the buffers or scene description. Lastly, the TRNS packet delivers changes made to the pose of objects within the scene.

Individually, these packet types are not unique. However, as we intertwine these protocols together, we start to see something more interesting. We can build a scene in a renowned game engine such as Unity and distribute it to completely foreign systems, such as web-based client using three.js and a Python application using an OpenGL renderer developed by the community. Not only this, but we can modify the properties of the scene at runtime and see how this changes for other clients. We can apply textures to primitives; we can alter the pose of a node within the scene-graph; we can introduce new geometry. All of what we have mentioned is achieved without ever tampering with the underlying graphics API, but by utilising

state-of-the-art formats such as glTF 2.0.

What we have outlined is by no means a complete specification. It is neither a perfectly-implemented product, but rather a prototype. We have found there is much more room for improvement. We have discussed how existing packet types such as UPDT can be significantly improved by taking a closer look at the glTF specification. There is much more exploration for even more packet types, to describe fine-grained changes in the scene. Ultimately, it is the long-term goal of this project to further develop these changes and perhaps bundle HVTP into existing game engines, which will allow almost anyone to construct a Hyperverse of their own and share it over a network.

We are confident that the popularity of game engines will only increase in the next decade. Film, television, architectural, automotive and manufacturing industries are just a handful of the emerging markets for game engines. Thus, the Metaverse of the future can't and won't be built entirely in a single language or framework. Instead, it might leverage a diverse set of technologies that all communicate using the same vehicle: the scene-graph. Finally, we believe that an open system which allows anyone, agnostic to any renderer, will dissuade any sole organisation from claiming the Metaverse as their own.

Appendix A

List of Acronyms

AR Augmented Reality

CVE Collaborative Virtual Environment

glTF Graphics Library Transmission Format

HMD Head Mounted Display

HTTP Hypertext Transfer Protocol

HTML Hypertext Markup Language

HVTP Hyperverse Transfer Protocol

MR Mixed Reality

NVE Networked Virtual Environment

VE Virtual Environment

VR Virtual Reality

Bibliography

[1] *A-Frame*. https://aframe.io/. [Accessed: 2020-04-11].

[2] *Autodesk FBX SDK Programmer's Guide*. http://docs.autodesk.com/FBX/2014/ENU/FBX-SDK-Documentation/index.html. [Accessed: 2020-07-05].

[3] Stuart K Card. *The psychology of human-computer interaction*. Crc Press, 2018.

[4] Elizabeth Churchill and David Snowdon. "Collaborative virtual environments: An introductory review of issues and systems". In: *Virtual Reality* 3 (Mar. 1998), pp. 3–15. DOI: 10.1007/BF01409793.

[5] DIS Steering Committee et al. "IEEE standard for distributed interactive simulation-application protocols". In: *IEEE Standard* 1278 (1998), pp. 1–52.

[6] Erick Lavoie David Herrera Hanfeng Chen and Laurie Hendren. "WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices". In: *McGill University School of Computer Science Sable Research Group* (2018), p. 22.

[7] Dawson-Haggerty et al. *trimesh*. Version 3.2.0. URL: https://trimsh.org/.

[8] Massimiliano Di Luca. "New method to measure end-to-end delay of virtual reality". In: *Presence: Teleoperators and Virtual Environments* 19.6 (2010), pp. 569–584.

[9] Craig Disselkoen et al. "Position Paper: Progressive Memory Safety for WebAssembly". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337171. URL: https://doi.org/10.1145/3337167.3337171.

[10] Jozef Doboš and Anthony Steed. "3D revision control framework". In: *Proceedings of the 17th International Conference on 3D Web Technology*. 2012, pp. 121–129.

[11] *Draco 3D Data Mesh Compression*. `https://google.github.io/draco/`. [Accessed: 2020-09-09].

[12] Nat Durlach and Mel Slater. "Presence in shared virtual environments and virtual togetherness". In: *Presence: Teleoperators & Virtual Environments* 9.2 (2000), pp. 214–217.

[13] Lukas Esterle and John NA Brown. "Levels of networked self-awareness". In: *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2018, pp. 237–238.

[14] *Extensible 3D (X3D) ISO/IEC 19775-1:2008*. `https://www.web3d.org/documents/specifications/19775-1/V3.2/Part01/Architecture.html`. [Accessed: 2020-07-28].

[15] Emmanuel Frécon. "A Survey of CVE Technologies and Systems". In: *SICS Research Report* (2004).

[16] Emmanuel Frécon and Mårten Stenius. "DIVE: A scaleable network architecture for distributed virtual environments". In: *Distributed Systems Engineering* 5.3 (1998), p. 91.

[17] Sebastian Friston and Anthony Steed. "Measuring latency in virtual environments". In: *IEEE transactions on visualization and computer graphics* 20.4 (2014), pp. 616–625.

[18] Sebastian Friston et al. "3DRepo4Unity: dynamic loading of version controlled 3D assets into the unity game engine". In: *Proceedings of the 22nd International Conference on 3D Web Technology*. 2017, pp. 1–9.

[19] Stuart Gilson and Andrew Glennerster. "High fidelity immersive virtual reality". In: *Virtual reality—human computer interaction* (2012).

[20] *glTF 2.0 Specification*. `https://github.com/KhronosGroup/glTF/tree/master/specification/2.0`. [Accessed: 2020-07-05].

[21] *gltFast by atteneder - GitHub*. https://github.com/atteneder/glTFast. [Accessed: 2020-08-10].

[22] *glTFForUE4 by code4game - GitHub*. https://github.com/code4game/glTFForUE4. [Accessed: 2020-08-10].

[23] *GLTFUtility by Siccity - GitHub*. https://github.com/Siccity/GLTFUtility. [Accessed: 2020-08-10].

[24] Jiřié Hnıédek. "Network protocols for applications of shared virtual reality". In: (2011).

[25] *Introduction to Universal Scene Description*. https://graphics.pixar.com/usd/docs/index.html. [Accessed: 2020-07-05].

[26] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. "Modeling the Effects of Delayed Haptic and Visual Feedback in a Collaborative Virtual Environment". In: *ACM Trans. Comput.-Hum. Interact.* 14.2 (Aug. 2007), 8–es. ISSN: 1073-0516. DOI: 10.1145/1275511.1275514. URL: https://doi.org/10.1145/1275511.1275514.

[27] Shah Khalid et al. "Optimal latency in collaborative virtual environment to increase user performance: A survey". In: *International Journal of Computer Applications* 142.3 (2016), pp. 35–47.

[28] Tobias Lauinger, Abdelberi Chaabane, and Christo Wilson. "Thou Shalt Not Depend on Me". In: *Queue* 16.1 (Feb. 2018), pp. 62–82. ISSN: 1542-7730. DOI: 10.1145/3194653.3205288. URL: https://doi.org/10.1145/3194653.3205288.

[29] Matthew Matl et al. *PyRender*. https://github.com/mmatl/pyrender. [Accessed: 2020-08-09].

[30] David L Neyland. *Virtual combat: A guide to distributed interactive simulation*. Stackpole Books, 1997.

[31] Kyoung Shin Park and Robert V Kenyon. "Effects of network characteristics on human performance in a collaborative virtual environment". In: *Proceedings IEEE Virtual Reality (Cat. No. 99CB36316)*. IEEE. 1999, pp. 104–111.

[32] Jon Postel et al. "Transmission control protocol". In: (1981).

[33] Jon Postel et al. "User datagram protocol". In: (1980).

[34] *Procedural Mesh Geometry*. `https://docs.unity3d.com/Manual/GeneratingMeshGeometr` `html`. [Accessed: 2020-09-09].

[35] David Raggett. "Extending WWW to support platform independent virtual reality". In: *Proc. Internet Society/European Networking*. 1995, p. 242.

[36] Arne Schilling, Jannes Bolling, and Claus Nagel. "Using glTF for streaming CityGML 3D city models". In: *Proceedings of the 21st International Conference on Web3D Technology*. 2016, pp. 109–116.

[37] Timothy Scully et al. "glTF Streaming from 3D Repo to X3DOM". In: *Proceedings of the 21st International Conference on Web3D Technology*. 2016, pp. 7–15.

[38] Mel Slater, Anthony Steed, and Yiorgos Chrysanthou. *Computer graphics and virtual environments: from realism to real-time*. Pearson Education, 2002.

[39] A. Steed. "Yther: A Proposal and Initial Prototype of a Virtual Reality Content Sharing System". In: *Proceedings of the 25th International Conference on Artificial Reality and Telexistence and 20th Eurographics Symposium on Virtual Environments*. ICAT - EGVE '15. Kyoto, Japan: Eurographics Association, 2015, pp. 151–158. ISBN: 9783905674842.

[40] Anthony James Steed. "Defining Interaction within Immersive Virtual Environments". PhD thesis. 2013.

[41] Anthony Steed and Manuel Fradinho Oliveira. *Networked graphics: building networked games and virtual environments*. Elsevier, 2009.

[42] Tim Sweeney. "Foundational principles & technologies for the metaverse". In: *ACM SIGGRAPH 2019 Talks*. 2019, pp. 1–1.

[43] *The Virtual Reality Modeling Language International Standard ISO/IEC 14772-1:1997*. `https://tecfa.unige.ch/guides/vrml/vrml97/spec/`. [Accessed: 2020-07-28].

[44] *three.js*. `https://threejs.org/`. [Accessed: 2020-03-05].

[45] *UnityGLTF by KhronosGroup - GitHub*. `https://github.com/KhronosGroup/UnityGLTF`. [Accessed: 2020-08-10].

[46] Aaron E Walsh. "Understanding scene graphs". In: *Dr. Dobb's Journal* 27.7 (2002), p. 17.

[47] *WebAssembly*. `https://webassembly.org/`. [Accessed: 2020-03-21].

[48] *WebGL Overview*. `https://www.khronos.org/webgl/`. [Accessed: 2020-03-05].

[49] *WebXR Device API*. `https://www.w3.org/TR/webxr/`. [Accessed: 2020-03-05].

[50] *X3Dv4 Draft Specification available to public now!* `https://www.web3d.org/news-story/x3dv4-draft-specification-available-public-now`. [Accessed: 2020-08-22].

[51] Bob Zeleznik et al. "Scene-Graph-As-Bus: Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications". In: *In Proceedings of Eurographics 2000*. 2000, pp. 200–.