Senior Honours Project

University of
St Andrews

Remote Collaborative Library for Virtual Environments

Paul Graham (*170004605*)

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 14062 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. I retain the copyright in this work, and ownership of any resulting intellectual property.

Supervised by

Dr. Alan Miller

April 27, 2020

Abstract

The School of Computer Science has accumulated reconstructions in **Unreal Engine** [1] from other student projects over many years. However, these reconstructions are currently are only visible by one person at a time, not providing any collaborative experiences.

In this project, we present **Unreal Collaboration**: a toolkit for developers of these reconstructions to add support for collaboration in a simple, yet customizable way. In addition, we present **Unreal Selector**: a platform for hosting projects made with the **Unreal Collaboration** toolkit.

Using **Unreal Selector**, developers can serve their projects to many clients in a simple manner. Clients can subscribe to these projects, customise their experience and connect with others exploring the reconstruction. **Unreal Selector** also provides a chatroom using the messaging platform, Discord [2] & a variety of media showing off the reconstructions.

Contents

1 Introduction **4**

1.1 Unreal Collaboration 4
1.2 Unreal Selector 5
1.3 Aims & Objectives 6
1.4 Structure of the dissertation 6

2 Context Survey **6**

2.1 Background 7
2.2 Multiplayer Games 7
2.3 Application Distribution 8
2.3.1 Steam 8
2.3.2 Heroku 8
2.4 Justification 9

3 Software Engineering Process **10**

3.1 Process 10
3.2 Effects of the 2019-20 Coronavirus Pandemic 11
3.2.1 Effects on Questionnaire 11
3.2.2 Effects on Additional Integration 12
3.2.3 Effects on Projects 12

4 Ethics **13**

5 Design **13**

5.1 Design Decisions 13
5.2 Unreal Collaboration 15
5.2.1 Architectural Design 15
5.2.2 Multiplayer Support 16
5.2.3 Signs 16
5.2.4 Virtual Reality 17

5.2.5 Ease of use . . . 17
5.3 Unreal Selector . . . 17
5.3.1 Integration with Unreal Collaboration . . . 18
5.3.2 Stability . . . 19
5.3.3 User Interface . . . 19
5.3.4 Related Media . . . 20
5.3.5 Communication . . . 20
5.3.6 Ease of use . . . 21

6 Implementation **21**
6.1 Unreal Collaboration . . . 21
6.1.1 Core Ideas . . . 21
6.1.2 Player Controller . . . 23
6.1.3 Character . . . 24
6.1.4 Sign . . . 25
6.1.5 Virtual Reality . . . 26
6.1.6 Build Tools . . . 27
6.2 Unreal Selector . . . 27
6.2.1 Server Overview . . . 27
6.2.2 Account Management . . . 28
6.2.3 HTTP Server . . . 28
6.2.4 RESTful JSON API . . . 29
6.2.5 Administration . . . 30
6.2.6 Client Overview . . . 31
6.2.7 User Interface . . . 31
6.2.8 Opening Unreal Collaboration Projects . . . 32
6.2.9 Discord . . . 33

7 Evaluation & Critical Appraisal **34**
7.1 Testing . . . 34
7.1.1 Unreal Collaboration . . . 34
7.1.2 Unreal Selector . . . 35
7.1.3 Composition . . . 35
7.2 Evaluation of Original Objectives . . . 35
7.3 Evaluation to Steam . . . 37
7.4 Quantitative Evaluation . . . 37

8 Conclusions **39**
8.1 Summary . . . 39
8.2 Future Work . . . 39
8.2.1 Unreal Collaboration . . . 39
8.2.2 Unreal Selector . . . 39

9 Appendices **40**
9.1 Build Instructions 40

References **40**

1 Introduction

Unreal Engine [1] is a very powerful tool that can be used for game development, level design, and reconstruction among many other applications. The School of Computer Science has used Unreal Engine for cinematic reconstructions of various historical sites in the past. These reconstructions provide the viewer a guided, cinematic tour of the site as it would have been sometime in the past.

1.1 Unreal Collaboration

Unreal Collaboration aims to provide more interactivity & support for collaboration in these reconstructions. Existing projects can be augmented by including the **Unreal Collaboration** C++ and Blueprint files to provide the following with little setup:

- Support for multiple clients, all of which can interact with each other.
- Support for signs, which can be placed in the environment, edited by multiple clients, and destroyed.
- Support for room-scale virtual reality headsets.
- Integration with **Unreal Selector** to provide access to administrative & personalisation controls.

Figure 1: A showcase of **Unreal Collaboration** in Abernathy, showing multiplayer support along with a sign in the environment. We obverse user "*paul*" has a red name as they are an admin in **Unreal Selector**.

Unreal Collaboration focuses on ease of use for developers, developers can start off of with the example project provided or add it to their existing project through the use of a simple Python script. Much of the code of **Unreal Collaboration** is abstracted within C++ library files to subtract complexity from view of the developer within the editor. Some parameters & functions from the library are exposed to developers through the use of Blueprints to give them customisation options for their own projects. Developers can also dive deeper and edit the features of **Unreal Collaboration** directly, guided by the extensive comments within.

The details of how a developer can add **Unreal Collaboration** to their project is described within the build instructions for this project **Appendix 9.1**. Once **Unreal Collaboration** has been added to a project, it can be built and distributed via the **Unreal Selector** platform.

1.2 Unreal Selector

Unreal Selector is a platform for distributing **Unreal Collaboration** projects to users, it consists of two applications: a server implemented in Python with the Flask library [3], and a client implemented in Javascript using Electron [4].

Using **Unreal Selector**, an institution can host any number of **Unreal Collaboration** projects. Users can download and use the client application to login to an institutions **Unreal Selector** server, from there the client can setup their desired settings and connect to a project[1] & learn about the environment while exploring with others. **Unreal Selector** also features an auto-update system so when another version of a project is pushed, institutions do not need to worry about users connecting on prior versions.

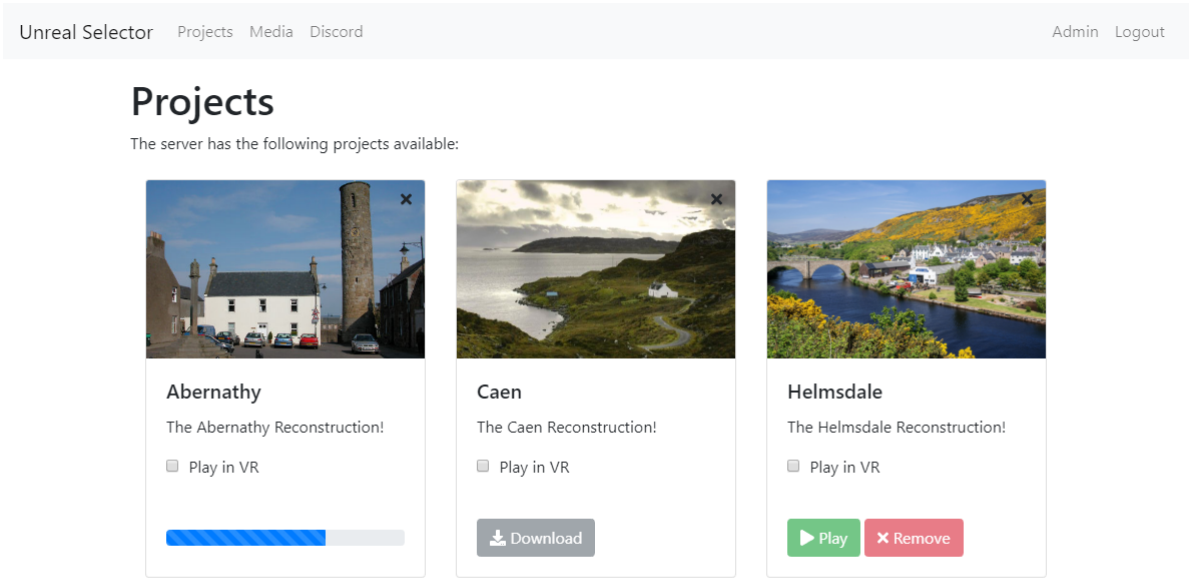

Figure 2: A showcase of **Unreal Selector** showing several reconstructions. We obverse the user is currently downloading Abernathy.

Clients can also communicate to each other, and learn more about the platform by using the embedded Discord chatroom & through communicating with the included bot user. Clients can also explore other cinematic reconstructions via YouTube & Twitch, and customise their experience within projects. **Unreal Selector** makes it really easy for institutions to setup a multiplayer experience for their clients, and for them to configure the platform to their requirements. It is also simple to upgrade the platform over time - adding, upgrading and removing projects is handled via a single page within the application.

Unreal Selector can also be used in other applications outside of reconstructed virtual environments. We see possible use within small companies wanting to distribute their Unreal Engine multiplayer games to their users & architectural firms wanting to showcase their projects made within Unreal Engine to potential clients around the world. **Unreal Selector** could also be modified to provide for more applications, including those outside of Unreal Engine.

To demonstrate the platform, we have produced a showcase video - `https://www.youtube.com/watch?v=7zl_0zTU8U4`. It shows off several features of the platform with real examples from the previous reconstructions.

[1] **Unreal Selector** handles creation and destruction of the project server, so that it won't be running unless there is at least one client connected

1.3 Aims & Objectives

To bring this project to life, a proposal was drafted outlining a simple description of the project alongside objectives for the success of the project. Each objective was given a level to indicate how key that objective was to the success of the project as a whole. As a result of the 2019-20 Coronavirus Pandemic, an objective had to be changed, this is talked more about in **Section 3.2**. The primary objectives are as follows:

- Client which can connect to the server, and allows the user to:
  - Select a virtual environment from a library
  - Explore the selected environment
  - See representations of other clients in the environment
- Server which allows client connections, which:
  - Holds the library of all explorable environments
  - Serves a client their selected environment & representation of other connected users
- Developers should be able import the application into both existing & new projects
  - It should take minimal setup for them to include working multiplayer
  - Importing into the library of explorable projects should be easy

The secondary objectives:

- The ability for clients to dynamically change the environment:
  - Add markers to explain more about a certain place
  - Transition dynamically between environments (e.g. by walking to a certain area)
- The ability for clients to interact through:
  - Text chat
  - Simple gestures
- Support for Room-Scale Virtual Reality (e.g. HTC Vive)

Finally, the tertiary objective:

- The ability for clients to add or remove objects in the environment

1.4 Structure of the dissertation

The rest of this dissertation is split into 8 sections: **Section 2** looks at similar projects implementing multiplayer functionality into different projects, **Section 3** looks at the processes implemented in the development of this project, **Section 4** investigates ethical considerations, **Section 5** looks at the overall design of both **Unreal Collaboration** & **Unreal Selector** and **Section 6** investigates how they were implemented. Finally, **Section 7** evaluates the project and **Section 8** is the conclusion.

2 Context Survey

In this section we will consider some background as to why this dissertation was undertaken. We'll begin by identifying weaknesses in previous reconstructions & identify problems as to why these weaknesses came about. We'll then look at other sources and see if there are any improvements to the reconstructions that they implement well. We'll consider each of these problems in turn, looking into possible solutions & how other sources solved some of these problems. Finally, we'll justify as to why we worked on this dissertation.

2.1 Background

History is a vital part of our development as a species. Our ability to critically evaluate our past decisions to inform future decisions allows us to be more informed when we shape our future. When learning about history, it is important to evaluate it from many different viewpoints, learning as much as we can from our past. We have a duty to perverse history to that future generations can still learn from it. Recent advancements in technology have allowed for us to preserve history through digital reconstructions, a virtual environment where history can be recreated and saved securely for the future.

The School of Computer Science has made a number of digital reconstructions, showing off important historical sites such as Helmsdale [5], Edinburgh in 1544 [6] & St. Andrews Cathedral [7]. Many of these reconstructions offer a virtual tour of the environment, showing off the important sites within, however do not give the viewer an opportunity to walk around the scene as if they were there. These reconstructions also lack the ability for collaborative exploration to occur within the experience - instead relying upon the presenter to offer this.

Many of these reconstructions are built within Unreal Engine [1], primarily because of its ease of use for this application - assets can be imported and placed easily within the world, and functionality can be programmed through the use of Blueprints, a visual programming tool. Unreal Engine was originally built for game developers, and therefore also offers the ability to control a character within the game, and the ability to support multiplayer experiences. However, these features offered by the engine are usually not built into any reconstruction offered as it takes developer time away from the main objective of developing the reconstruction, is complex to work with, and offers no easy path for distribution. Developing a tool that can solve all of these problems will allow developers to easily add multiplayer support to their projects, and increase viewer engagement with their history.

2.2 Multiplayer Games

As internet started to become widely available, more and more multiplayer games were developed. Many of these games look at history for inspiration - recreating areas of historical significance within the game for players to enjoy. The *Assassin's Creed* [8] series features many real historical sites along with fictional sites for players to explore in. For example, *Assassin's Creed: Brotherhood* features sites from 16th Century Rome, a screenshot of the game is shown in Figure 3 [9]. While video games usually offer questionable historical accuracy, they offer great immersion into the environment by building a game around it. It is interesting to investigate into how they are made to be able to develop a historically accurate, immersive environment.

Figure 3: Multiplayer gameplay footage from *Assassin's Creed: Brotherhood*.

Immersion in video games comes from the level in which the player can immerse themselves into the environment. A user passively watching a tour around a virtual environment will be less immersed than one who is surrounded by an environment they can interact with. Further technological advancements have been made allowing for even greater immersion through the use of *virtual reality* headsets, where a user is literally surrounded by the environment from all sides.

2.3 Application Distribution

After a project has been built, institutions using that project will need to also develop a way for their users to access, and run the built project. For a single interactive reconstructed environment, it is enough to allow users to download and run the built executable. However, when building for multiplayer, organisations need to consider a lot of additional factors: how to inform clients to connect to the correct server from their executable; how to deal with updates to the project over time; and most importantly, how to host the server for the application.

When hosting the server, it is important to consider how many users you expect to be connecting to the server at one time, as this informs what hardware the server will need to run on. A server serving content to at least 200 people concurrently is very different to a server serving a small group of people, who connect only for a short time. Many companies use services such as Amazon Web Services [10], or Google Cloud Platform [11] to host servers for applications both large & small - these services charge in terms of the time the server is used for - and therefore companies using these services will want the server to not run unless it is required to. It is also important to consider how problems with the server will be identified and solved as well as how the server will scale to deal with increased requests over time.

There are some existing platforms that attempt to solve some of these problems, which we'll discuss.

2.3.1 Steam

Steam [12] is a platform for game distribution. Players can buy games from the store, and then play them within their library (as shown in Figure 4), these games are then kept up to date through their service. However, the other problems discussed above are left up to the developer to solve: clients may still need to enter the address of the server manually (if there is not a built-in list of servers in the application), and developers still need to host the server manually.

Steam also offers an API for applications to connect and use some features available on the platform, such as the the ability to connect with friends, and earn digital goods while playing games. There are many other platforms similar to steam, such as Epic Games Launcher (from creators of Unreal Engine) [13] that offers distribution of games made with Unreal Engine, and Origin [14] which offers distribution of games made by EA.

2.3.2 Heroku

Heroku [15] is a platform for web applications hosted in the cloud, shown in Figure 5. Using Heroku, developers can deploy their applications to *dynos* which host the application within a *containerised* environment. While not solving the discovery problem, nor the auto-update problem. Using Heroku can save a developer a lot of time over having to deploy the server manually, allowing the developer to simply push their application to git instead of having to manually update files on the server. Heroku also allows for auto-scaling of applications, meaning a new *dyno* is made whenever one is needed thus saving cost over having to always pay for the highest capacity you expect.

Unfortunately, Heroku only supports applications running over HTTP so will not support Unreal Engine applications by default (as these use UDP for communication). There are also other companies offering similar services to Heroku, such as DigitialOcean, however these are much more complex to setup.

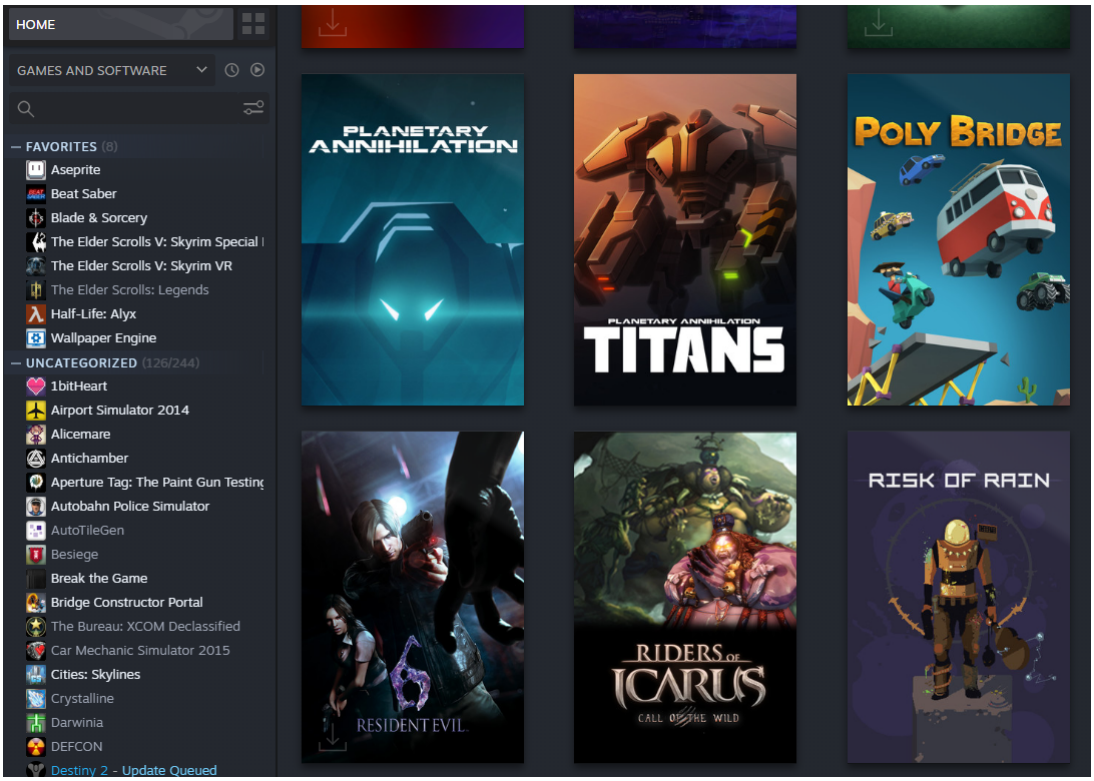

Figure 4: A users steam library, showing the games & software they own in the left sidebar.

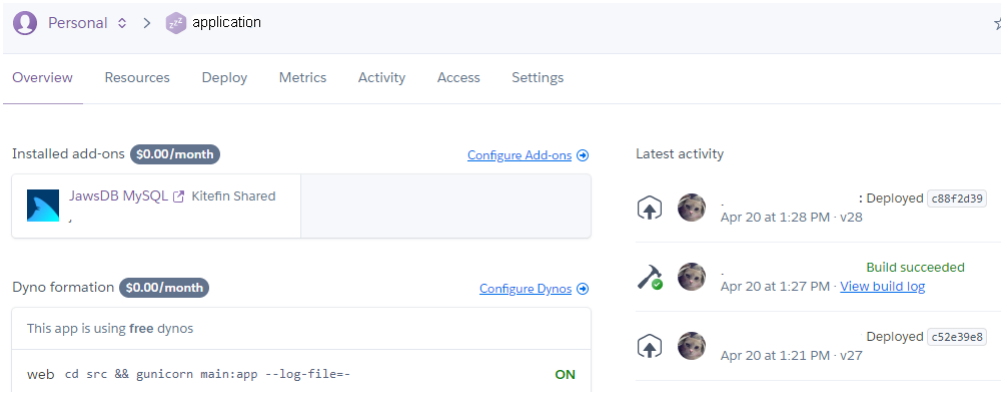

Figure 5: A photo showcasing a users view of their application on Heroku, we obverse how simple deployment is using `git`.

2.4 Justification

Working on creating a system for developers to quickly and easily make their reconstructions into interactive multiplayer environments will increase immersion for users, allowing them to learn much more about their history. It will also allow for users to collaborate while they explore the environment - for them to discuss and critique their viewpoints of history. Allowing developers to easily export these experiences to a centralised library will allow developers to support multiplayer for every environment, as well as create a digital hub for showcasing every reconstruction & additional resources centrally so that users will be able to explore and learn from many different reconstructions in one session.

3 Software Engineering Process

3.1 Process

I chose to develop the software with an adapted version of the SCRUM agile methodology [16]. With this, the **product owner** identities what the application has to feature from the perspective of a stakeholder of the product called a *User Story* - e.g. "*As a user, I'd like to select a* ***Unreal Collaboration*** *project to launch within* ***Unreal Selector***.". Each user story is given a priority as to how important is it, and how long it is estimated to take, and added to the **Product backlog**, this is an ever expanding, prioritised list of the user stories the system must fulfil. Every sprint (usually a period of a month), a number of these user stories are selected from the product backlog to work on during the sprint backlog. Every day, developers select user stories from the sprint backlog to work on, if work is done on a user story the developer can move onto working on another user story.

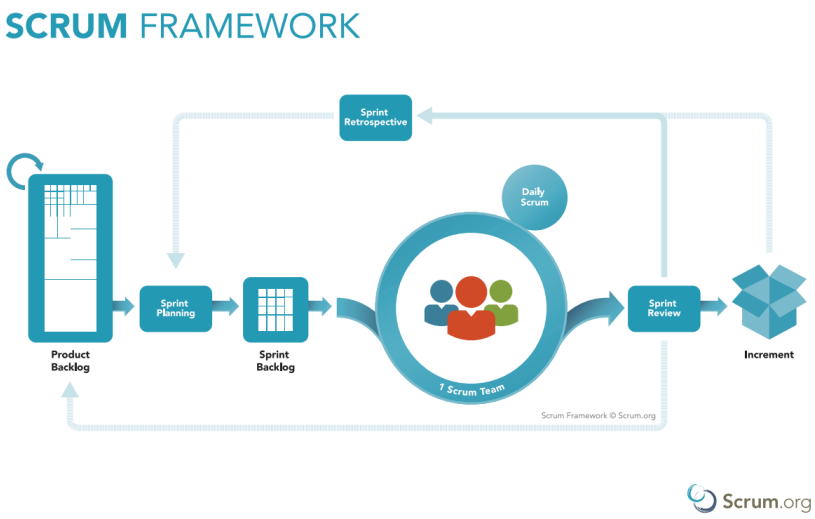

Figure 6: A diagram showing how the SCRUM methodology functions.

As I worked alone on this project, the methodology was not followed perfectly. Each week myself & my supervisor met to discuss the project, we discussed which goals we'd like to have implemented within the software in reference to an objective, e.g. "*The addition of user placeable signs so that users could take note of interesting things they've found while exploring the environment*", these goals were noted within the meeting log of what should be worked on that week.

Every week, I worked towards implementing these goals, however their scope usually expanded, e.g. "*The sign should also be editable by developers to support ease of customisation for developers*" or "*The sign, along with its text needs to replicate*". This scope creep was quite common due to the goals were fuzzy: my knowledge of how to work within the Unreal Engine platform was initially very limited & the documentation for advanced features (such as multiplayer) was lacking. Therefore, when I came to set goals they were usually too ambitious to be implemented in the same week. As my knowledge of the platform increased I became better at setting less ambitious goals, however even with this scope creep occurred due to the unknown nature of what I was going to be working with. Whenever scope creep occurred, the goal was broken down into smaller, more manageable goals that should be able to be implemented which hopefully one could be implemented within the week & the others left for another time when they should be able to be implemented.

I attempted to follow the principles of the agile manifesto [17], notably I tried to follow "*Working software over comprehensive documentation*". I couldn't have fully working software within the reconstructions themselves due to the the inability for several Unreal projects to share C++ assets & the size of the projects causing them to be time-consuming to work on and export. Instead, I had a sample project in which development

would act as the authoritative version of **Unreal Collaboration**, this code would be transferred to the reconstructions themselves when development of **Unreal Collaboration** was done. Additionally, **Unreal Selector** couldn't always have a fully working library of projects at all time, instead several sample projects were added to the library (including a build of the sample project) so that *all* features of the library could always be tested.

Overall, this process worked well for this project, I was able to follow principles of the agile manifesto & attempt to follow the SCRUM methodology. I was able to respond to change within the project quickly, sometimes having the change fully implemented on the same day it was suggested, and I was also able to always keep a fully working version of both parts of the system at all time. However, I would try to follow the follow the SCRUM methodology more closely next time, as this would allow me to track how often scope creep occurred, and adjust my estimations based on this knowledge. It would also allow me to more easily deal with the additional goals scope creep created, instead of vaguely guessing as to whats the next small thing I could implement without considering a proper prioritised list, I could implement this list into my development and use it to help keep focus on working toward goals.

3.2 Effects of the 2019-20 Coronavirus Pandemic

The 2019-20 Coronavirus Pandemic greatly affected this project, however we were able to adapt to change quickly. The following section will investigate into what ways we adpated to the changes caused by the pandemic.

3.2.1 Effects on Questionnaire

Originally, the third primary objective was as follows:

- A questionnaire which evaluates the performance of the software:
  - The system should be intuitive to use
  - Clients focused around collaboration & learning about the environment

This objective was changed to the current third primary objective due to the pandemic. Our tentative plans for the questionnaires would have been to evaluate how intuitive developers thought **Unreal Selector** was with the use case of setting up the platform & adding a simple project, and evaluate how clients thought of the focus of the platform through the use case of a short exploration setup in a virtual environment. We were unable to move these questionnaires online due to the extremely large file size of Unreal Engine projects & the limited internet connection within our current residence.

As the pandemic caused us to enter a national lockdown before these plans were fully fleshed out, we decided to evaluate the ease of use of the software on a small scale ourselves by importing it into an exiting projects. After importing, and changing the gamemode as stated in the build instructions (**Appendix 9.1**). There were some issues with due to the modular skeletal meshes (Figure 7) which this project used for characters, after a fix for this was implemented, other Blueprints were edited to look realistic. The project was then built for Windows & imported into **Unreal Selector**. Overall, this took about a day due to mostly to the dynamic skeletal mesh this project required. However, if we take away the time we spent debugging the skeletal mesh, the project was made to fully support multiplayer in just a few hours.

Figure 7: Replication of Skeletal meshes in Abernathy

Additionally, we decided to change the original objective to the current third primary objective. This new objective supported the ease of use we want the system to possess for developers, as this has been one important factor in the implementation of the project. If the system was not easy for developers to use, the overall quality of the reconstructions created would decrease as developers time is taken away from working on the actual reconstruction to focus on integrating **Unreal Collaboration**, which we didn't want.

3.2.2 Effects on Additional Integration

Since this project uses a lot of reconstructions from the group that runs the CINE service within the School, we were asked to include CINE support into **Unreal Selector** so that CINE users could login to the service without having to create an account. As a result of the pandemic, the CINE endpoint **Unreal Selector** was going to use was unable to be implemented. We have therefore commented out the CINE button on the frontend of the application, but have left the connection code intact to show how integration would have worked in the backend.

3.2.3 Effects on Projects

As a demonstration of the project, we were planning to add **Unreal Collaboration** support to a number of reconstructions and deploy these projects to a Linux server so the School could show off these reconstructions using **Unreal Selector**. Due to the extremely large file size of the projects, the projects were kept only on a computer within the School, where development on these projects would occur. As a result of the pandemic, these projects were unable to be transferred to our current residence, and so **Unreal Collaboration** support cannot be added to them. Fortunately, two of these projects were transferred before the pandemic, we have added **Unreal Collaboration** support to both of these projects and are able to showcase them in the showcase video.

4 Ethics

The project developed a system for connecting several virtual environments together, and therefore had no ethical concerns as we didn't work with any human or animal subjects. However, we prepared to give a questionnaire to evaluate the ease of use of the the project - this use case is covered by the School's software artifact ethical approval application.

5 Design

5.1 Design Decisions

After deciding on objectives for the problem, we started to look into designing solutions to meet those objectives. It is important to first consider the scope of the library, that is whether the library is **static** or **dynamic**. While a static library would be able to support all of these objectives, reconstruction developers would not be able to easily add new reconstructions to the library, having to create and distribute an entirely new library to clients. This would take time out of their development of reconstructions with less benefit each additional project. Therefore we have decided to build a dynamic library to give developers an much easier time when deploying new projects.

The objectives describe a client-server model where the clients can choose one of the available environments to connect to on the server and explore it. From the objectives we outlined two different ways this could be implemented:

- The clients would login to a virtual hub, with a number of portals to environments they can explore with others within that application (like in many RPGs [18] & Figure 8). Developers would be able to add worlds, and change up portals dynamically, perhaps even at runtime.
- A separate launcher application which would have a number of disconnected worlds which the clients could explore with others. Developers would be able to add more worlds by simply adding them to the list of files the launcher application retrieves from.

Figure 8: An artist rendering of a room of portals leading to different worlds from one hub world.

These implementations vary in their **connectivity**, the first allows the player to always stay inside of the Unreal Engine, while the second has players jumping between applications when connecting & disconnecting from worlds. Considering the dynamic nature of the library, for the first implementation to be possible developers will need to be able to update the server with new worlds, while this could be done at runtime it would be incredibly complex to implement within Unreal Engine. The very large size of projects within Unreal Engine means that clients will need to download all assets from all worlds for each iteration of the server. While other techniques are possible, such as streaming needed assets to clients when they are needed, these would be very complex to implement within Unreal Engine & would better suit a boutique engine. Remote clients may not be interested in, and may not be able to download very large files every time they want to explore a world, taking away the immersive experience we want them to enjoy. Therefore, for portals within Unreal Engine it is better implement the second option, which we have decided upon.

This implementation decision also lends itself to distribution rather well. As worlds for the player to connect to are stored on the server, the client now only needs to download the assets they need to connect to the world they want to explore. This download can be done anywhere that can connect to the server hosting the downloads, even a website. However, web distribution brings up the problem of how clients will use the downloaded files to connect to the server. After building an Unreal Engine project & learning about the structure of the built project as well as the command line arguments that could be given to it, we learned that one launches the project from an application deep in the file hierarchy & one connects to the server via a command line argument. These are not user friendly and would need to be automated with one of the following:

- A script placed inside the download the client would run to open the application & connect to the server automatically.
- An application which would download the application into a specific place. It would then automate the running of the application from the place it was saved in.

These approaches are very similar in many ways, however the first approach dominates in its ability to be standalone, and only require a web browser for clients to view the library, from there clients would download the project and the script would connect them to the world. However, this implementation also has a major drawback in its support for **updates**. Clients would need to either re-download the project from the website, or the script placed inside the product would need to check for updates for itself & the project. As we didn't want to implement a self-updating script that had to be placed into each project, we decided to implement the second approach and focus on stability on the underlying application so it wouldn't need frequent updates.

Finally, we need to consider how the servers for the projects themselves will be **hosted** & **controlled** as we require them to replicate data from client $\rightarrow$ client. For hosting, we assume that institutions who will use our platform will host it on their own web server they use for their website - as this is the cheapest option for them. We consider a model shown in Figure 9 where most clients are connecting to a the web server to see the web site, and fewer are connecting to **Unreal Selector** to download projects. We therefore want to minimise our resources on their server so that their main web server will get the majority of the resources, instead of constantly using up resources for the project servers which won't be used all of the time. We have therefore decided to control the project servers via a main controlling server which will allow them to run when a player wants to connect.

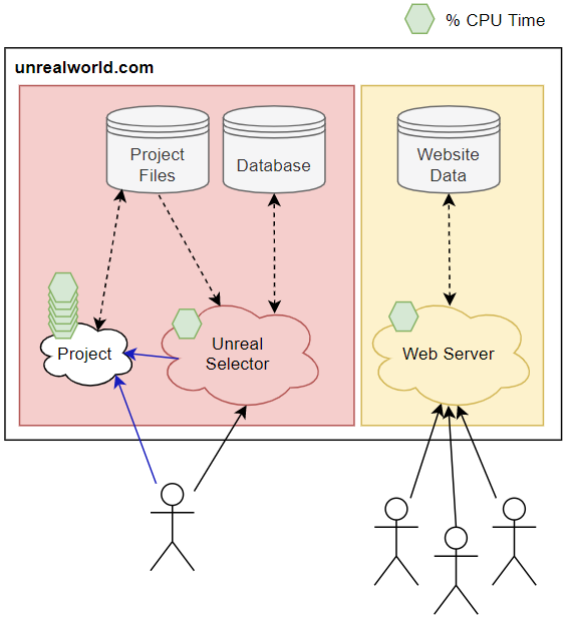

Figure 9: We expect many more people to be connecting to the web server rather than the multiplayer reconstructions, but they take up much more CPU Time.

5.2 Unreal Collaboration

Unreal Collaboration is designed to be an easy to use library for adding multiplayer support to projects in Unreal Engine. It was decided to develop the library in C++ as we were much more familiar with it compared to Blueprints. Developing in C++ also means that future developers who develop entirely within Blueprints will not be exposed to the complexity of networking within Unreal Engine. It was decided that **Unreal Collaboration** would support the following features, which we will discuss in more detail:

- Multiplayer support
- Signs to denote important places around the environment
- Room-scale virtual reality for added immersion
- Ease of use

5.2.1 Architectural Design

Unreal Collaboration was designed to be an addition to any game in Unreal Engine. For multiplayer support, several structures are overridden with custom implementations - such as the player. Developers however, can extend these implementations to implement their own functionality. The following diagram (Figure 10) shows a basic overview of how **Unreal Collaboration** interacts with Unreal Engine:

From Figure 10 we can see the player controller is the heart of **Unreal Collaboration**, this component translates player input to in-game actions. The player controller can spawn signs and change control from the player character over to a sign, and from a sign back to the player character. The appearance and properties of components such as the player character & the sign are controlled via Blueprints which allow for developers to create instances of objects and change their properties within the editor. Additionally, these components expose functions to Blueprint for easier development within the editor.

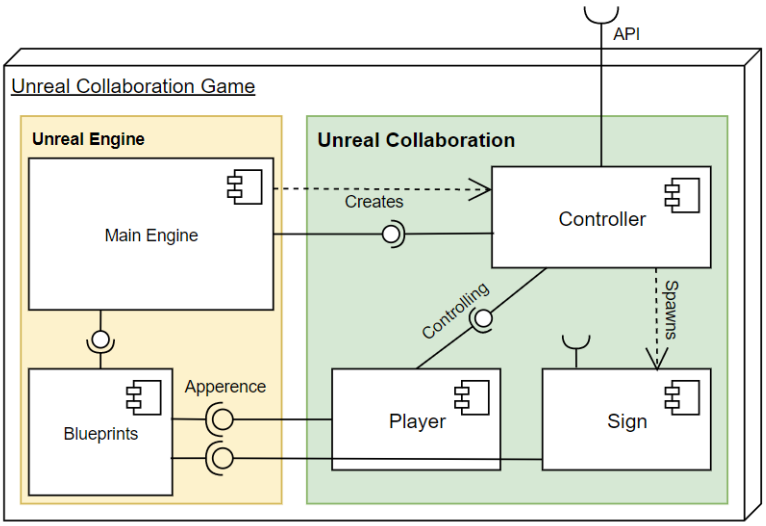

Figure 10: A basic overview of how **Unreal Collaboration** interacts with Unreal Engine. The controller is currently controlling a player. We obverse the API connection **Unreal Collaboration** users.

The controller can also call certain endpoints within the **Unreal Selector** API to allow for information to be retrieved from the **Unreal Selector** server, where the **Unreal Collaboration** server is controlled. This API can be used by the controller to, for example, retrieve information about the player from **Unreal Selector**, such as their username and render it within the world, or allow Blueprint developers to use this within their project.

5.2.2 Multiplayer Support

Multiplayer support within Unreal Engine can take one of two forms: a *listen* server, where one client also acts as the server, and holds the authoritative state of the game; or a *dedicated* server, where a server exists outside the set of clients and holds the authoritative state of the game. We decided to use a dedicated server for **Unreal Collaboration**, as having the server outside the set of clients would allow us to control the server independenly, changing it whenever we'd like to.

With that in mind, we also wanted the clients to be able to see, identify & chat to each other within the game as this would allow for a good collaborative experience. We wanted clients to be able to move freely throughout the world, and explore it from *any* angle. Finally, we wanted clients to see each other as realistic characters within the world - developers would be able to choose what styles a user could appear as so they appear realistic to the world they are exploring within, and users would be able to customise this experience.

5.2.3 Signs

We wanted users to be able to place down signs in the environment to take notes and point out interesting things they find while they explore. These signs were to be transient - only lasting for however long the *current session* (that being the time a group of users is on the server for) lasts for, editable - so that users could go back and add additional information and even delete the sign if necessary, and customisable - so developers could edit the feeling of the sign to make it fit within the environment, and players can hide them if they want to.

Figure 11 shows that signs are spawned via the player controller, as shown in **Section 5.2.1**. The player controller will choose a place close to the player to spawn the sign at, and delegate control over to the new sign, releasing the player character. This will also cause the HUD to change to support the editing of text upon the sign. The player can also choose to either stop editing the sign, or delete it, in which case control would be given back to the player character that was released earlier.

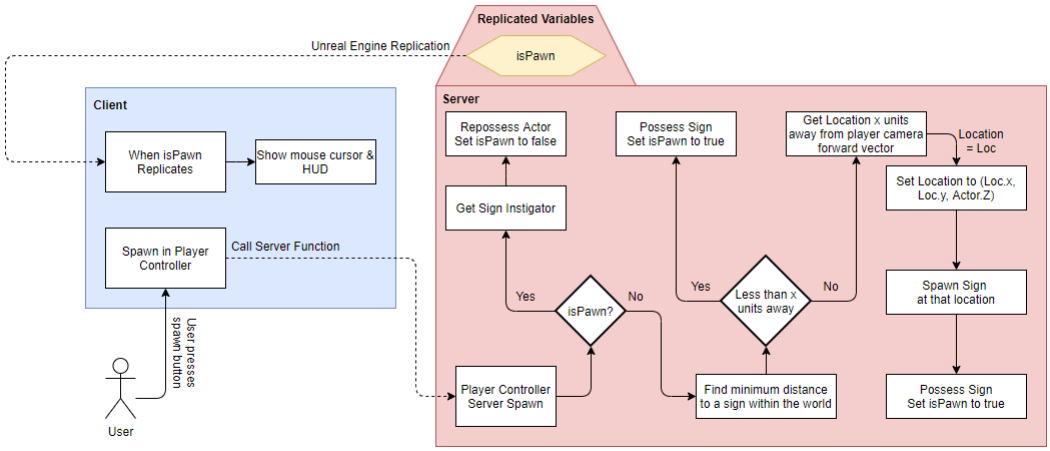

Figure 11: A diagram showing the interaction between the server & client when a sign is spawned.

5.2.4 Virtual Reality

For increased immersion, we also want to allow users to join with a room-scale virtual reality setup (such as HTC Vive). When in this mode, we would like it to feel like the player was within the virtual environment. For movement therefore, we decided upon allowing the player to move either by moving about within their virtual reality setup, or by launching themselves toward a specific location. This method was choose over teleportation as it offers the player the ability to explore in mid-air. While in virtual reality, we also decided to not allow users to create, edit, nor delete signs as this could distract from the immersive environment the player is in (although other developers could change this).

5.2.5 Ease of use

One of the most important features of **Unreal Collaboration** is ease of use for developers. We want developers to be able to add **Unreal Collaboration** to their project without hassle, and not need to edit any of the code from **Unreal Collaboration** to get it working with their project. Additionally, we want developers to be able to configure the project to their needs - customising on top of the library to provide whatever their project requires.

In terms of networking, we do not want developers to have to develop an application capable of storing account information for **Unreal Collaboration** to work out when it is imported. Instead, we want the library to support **Unreal Selector** as an expandable platform developers can deploy to provide this information to **Unreal Collaboration**.

5.3 Unreal Selector

Unreal Selector is designed to be the library for **Unreal Collaboration** projects, allowing users to join and explore any reconstructed environment, learn more about them from related media, and communicate with each other. Since **Unreal Collaboration** interfaces directly with **Unreal Selector**, we expect developers to set both of these services up together. With this in mind, it was decided that **Unreal Selector** would support the following:

- Ability to control **Unreal Collaboration** dedicated servers
- Ability to launch **Unreal Collaboration** applications, and connect automatically with the corresponding **Unreal Collaboration** server

- Library of related media from online
- Ability for users to communicate to each other within the application
- Ease of use - adding, upgrading & removing projects should be simple for developers & navigation should be easy for users

5.3.1 Integration with Unreal Collaboration

The main goal of **Unreal Selector** is to manage projects developed with **Unreal Collaboration**. To achieve this, there are a few components that need to be considered: how will we control the dedicated servers built which include **Unreal Collaboration**? How will be connect users automatically to those servers? How will we ensure the server & clients are always compatible with each other?

To start to answer these questions, we decided to built **Unreal Selector** as two applications who would communicate with each other. The first application would be a server, which institutions would deploy to a server they host. This server application is to be written in Python, using the Flask library [3], and mainly serves to control **Unreal Collaboration** dedicated servers & serve files to the connecting client, a diagram of this is shown in Figure 12

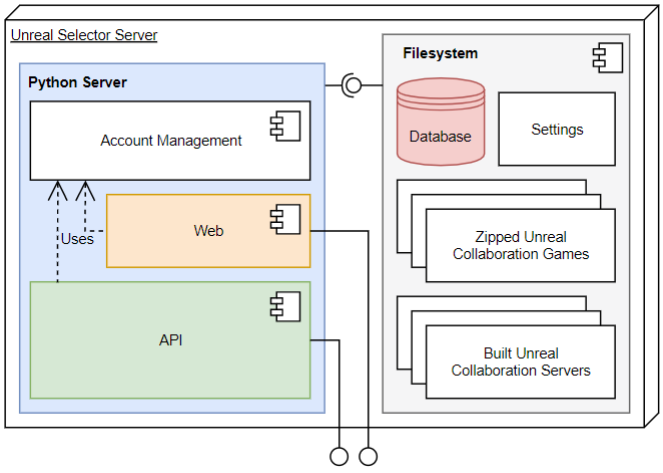

Figure 12: A basic overview of the **Unreal Selector** server application. We obverse the connections **Unreal Selector** provide both for the web & API connections.

The second application would be a client, which could be downloaded from anywhere, allowing users to connect to their desired institutions server. This application is to be written in Javascript, using the electron library [4]. The use of electron allows us to work as if this application was inside a browser, but with more control such as direct access to the users file system, a diagram of this is shown in Figure 13.

The client would function by presenting a login page to users when they first connect, asking them to enter login or signup to an intuitions server. After the player has successfully signed in, the server serves content to the client over HTTP that allows them to explore that servers available projects (on the index page), related media (in the media page), and communicate with other players (in the discord page).

Unreal Selector acts as a subscribe system, which keeps the client up to date on whatever projects they've subscribed to. Whenever a client requests for a project, the client will ask for the current version of the project from the server. If the client doesn't already have the project, the latest **Unreal Collaboration** client would be downloaded from the server via the API. If the client has an earlier version, it will be invalidated and the client will be requested to download the latest copy of the project.

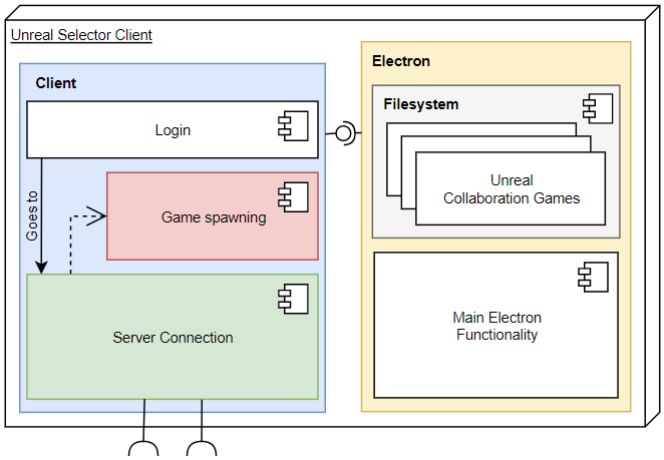

Figure 13: A basic overview of the **Unreal Selector** client application. We obverse the connections **Unreal Selector** needs to connect fully to the **Unreal Selector** server.

Whenever a project is finished download, it is automatically extracted & executed with flags to point toward the **Unreal Collaboration** server. While the **Unreal Collaboration** client application starts to execute, the **Unreal Collaboration** server would be launched on the server, thus allowing the client to connect & explore.

The server will also control other functionality within the client, such as controlling the **Unreal Collaboration** servers to shut off when there is are no users connected to them. This is designed to function around an idle timer, which will tick down from when the server is first started. **Unreal Collaboration** client applications are programmed to find the time this idle timer takes to tick down, and to call the reset function on the server to keep the server alive for users to explore on it. Whenever users stop exploring, this idle timer runs out and the server is killed.

This two application approach to **Unreal Selector** allows institutions to customise their server to match their projects, but still be compatible with any version of the **Unreal Selector** client - still allowing projects to be downloaded & executed automatically. Institutions can still easily change the client applications to be customised for a unique experience and remove compatibility with normal **Unreal Selector** servers, but will now have to offer a way to download the customised client application.

5.3.2 Stability

We decided to use the application pathway for its ability to be stable over a long period of time. As we are implementing with electron, we can delegate NodeJS to only be implemented for very core functionality (such as downloading projects, and allowing clients to connect to the server), while allowing the rest of the functionality (including calling core functionality) can be handled with Javascript inside of chromium. With this unique setup, instead of having to frequently update the client to receive new features, we can consider the clients code to be stable, and implement new functionality by only using the server & client-side Javascript.

5.3.3 User Interface

Unreal Selector requires a core user interface to allow players to connect to a server in the first place, from there the server can take over how the user interface is displayed. As this user interface is core functionality, we want it to be stable & understandable for all. As we are using electron running on top of chromium, we can stability render any HTML document we want to.

We also want the user interface to be understandable for all, therefore we have chosen to use a vanilla implementation of bootstrap [19] as this allows for responsiveness of elements on the page, so they can fit to any window size they are rendered on. We have also chosen for the server to be designed identically to match

the default client. This same approach should be taken to pages which the server sends to the client, their function should be easily recognisable to all users. We have decided to go with a top navbar so that the user knows where they are at all times, and can navigate to any other main category page within one click, as shown in Figure 14. From each page, users can be taken deeper for additional information, but they should easily be able to escape back to where they came from with one click.

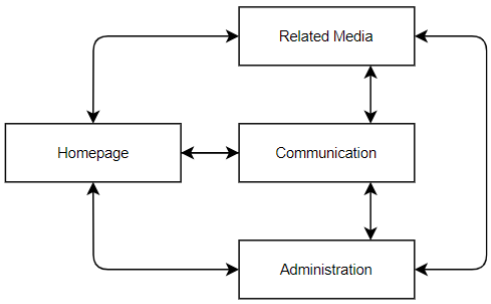

Figure 14: A simple diagram showing the ability for the navbar to take the user to anywhere else within the application with a single click.

The state of the application is also shown within the UI via a single progress bar that changes colour underneath every project so that the user always knows what the application is doing. The user can visually see when an project is downloading content (when the progress bar is blue), unzipping the downloaded **Unreal Collaboration** client files (when the progress bar is green), encountered an error (when the progress bar is red), or is waiting for something to happen (when the progress bar is grey). Helpful error messages are also provided to clients in case anything breaks so they know what to do to resolve the error.

5.3.4 Related Media

One important feature of **Unreal Selector** is the ability for institutions to show off media related to the projects they have available on **Unreal Selector**, such as cinematic videos and streams of others exploring. We decided to include streams as this allows for collaboration to occur outside of Unreal Engine, and for viewers of the stream to be guided through a tour of their history by a presenter, and even give them the opportunity to join the presenter in Unreal Engine and learn more about their history.

We decided on supporting two media platforms: YouTube for videos & Twitch for streaming. Institutions can place a specific YouTube playlist to be shown within **Unreal Selector**, and can also specify a specific title & category for streams to link to within Twitch. We have also created a stream guide for new streamers within **Unreal Selector**, so that they can also lead others to the exploration of their history.

5.3.5 Communication

We decided to handle communication via the messaging service Discord [2]. Discord allows for users to setup servers where users can chat to each other both with text (over text chat), and with their voice. For functionality within **Unreal Selector**, we decided to use TitanEmbeds [20] which allows for users to use text channels from within a browser context, such as electron. For voice communication, we rely on users downloading, or using the full Discord application where they can communicate with others inside the virtual world by using their voice. Institutions can create their own Discord server, and can customise **Unreal Selector** by inserting both a TitanEmbeds link & a regular invite, or use the offical Discord server provided by default.

Our official discord server also features a Discord bot user for server automation tasks - such as directing users to links where they can download **Unreal Selector**, or links to official **Unreal Selector** servers they can join. The bot also offers server moderation options, such as timing out badly behaving users. As we are able

to control the embedded frame discord appears within, we are also able to give developers support for custom functionality within the embedded frame, such as going to internal links, or opening modals.

5.3.6 Ease of use

Similar to **Unreal Collaboration**, **Unreal Selector** is built to be a easy to use platform for both developers & users. In terms of ease of use for developers, dependencies for the python application are handled with the `pipenv` package, which provides an easy to use abstraction with dependency management built for python virtual environments. A single JSON file defines all of the customisation options for the server itself, along with this file the `README` file is dense in information about these options as well as every other part of the server so that developers can easily understand how the server operates, and how to deploy it on top of their current setup.

Additionally, **Unreal Selector** is designed for developers to be able to add their own functionality to the server easily, while only having to edit one file to add in this functionality. This feature is designed mostly for developers who want to integrate custom authentication sources, or who want to add in new API endpoints for an external service to interact with.

6 Implementation

In this section we will investigate how each component of this project was implemented. We will look into the core functions of the two applications, as well as how each application communicates with the other within their own sections.

6.1 Unreal Collaboration

This section details the implementation of **Unreal Collaboration**, here we will start by looking at some core ideas behind implementing multiplayer in Unreal Engine, before going on to look at how these core ideas are used to create the rest of the library.

6.1.1 Core Ideas

To begin, we look further into how Unreal Engine handles networking. The basic principle behind networking in Unreal Engine is that of ***replication***, where the authoritative versions of variables or assets on the server are automatically *replicated* to clients, as shown in Figure 15. Clients may also call functions on the server to update this authoritative state to others. This basic principle reduces complexity for developers as they no longer need to consider low level networking architecture such as network protocols like TCP vs UDP, nor do they need to consider serialisation of network data. Developer complexity is also reduced as they just just need to work with a single code base, containing both the client & server code and therefore do not need to update another code base whenever a change occurs.

Using replication does not take away all of the complexity as developers still need to consider minimising network usage to allow the application to be usable for all clients. For this, Unreal offers the option to choose what exactly is replicated from server $\rightarrow$ client (e.g. a player mesh which is the same for all players will not need to be replicated). Unreal also offers more customisation options for replication, such as the option of a function being called when a variable has been replicated from the server.

Before we look into how these methods of replication are used within **Unreal Collaboration**, it is important to first look at what is naturally replicated. Within Unreal Engine, there are two important structures which we will discuss - the **game mode**, and the **player controller**. The **game mode** exists only on the server and serves to control how players enter, pause and leave the game (we will look into this into further detail on game modes in **Section 6.1.5**). For each player who joins the game, a **player controller** is created on the server and is replicated to that players client only, this structure controls how the player inputs translate to actions within the world. The player controller delegates player input actions to pawns (an actor which is controlled

Figure 15: Replication of several variables & assets in the Helmsdale project.

by a controller, i.e. the physical representation of a player within the world) which the player can possess, and these pawns can choose how to handle this player input, for example to move the player forward or to shoot a bullet from a gun the player holds.

Within Unreal Engine, every client is given a **role** in the network. These roles are described as an enum so their values can be compared - servers are given `ROLE_Authority` (value of 3), while clients are given `ROLE_None` (value of 0). Since Unreal Engine acts on a single code base, comparing what is returned from `GetLocalRole()` to `ROLE_Authority` can limit code to only be executed on the server or on the client.

On top of using comparison to perform actions for a specific role, a special `UFUNCTION()` macro can be attached to the function prototype in the header. This `UFUNCTION()` can define a function to be executed only on the server (the full definition is `UFUNCTION(Server, Reliable, WithValidation)`, as shown with the `ServerSpawn()` & `ServerDelete()` functions in Figure 16. When clients call the decorated function, it will only be executed upon the server if the *validation* function returns true. The *validation* function is a special function that exists to stop malicious actors exploiting functionality within the application, for example, if a player was to ask the server to spawn a bullet the player will need ammo, without ammo the player shouldn't be able to shoot the bullet.

Finally, the client can react to when a variable or asset is replicated from the server using the `UPROPERTY(ReplicatedUsing = OnRep_MyFunc)` macro on a variable, as shown in the `isPawn` variable in Figure 16. This function is useful for applying client specific properties to variables that have been replicated, such as visibility.

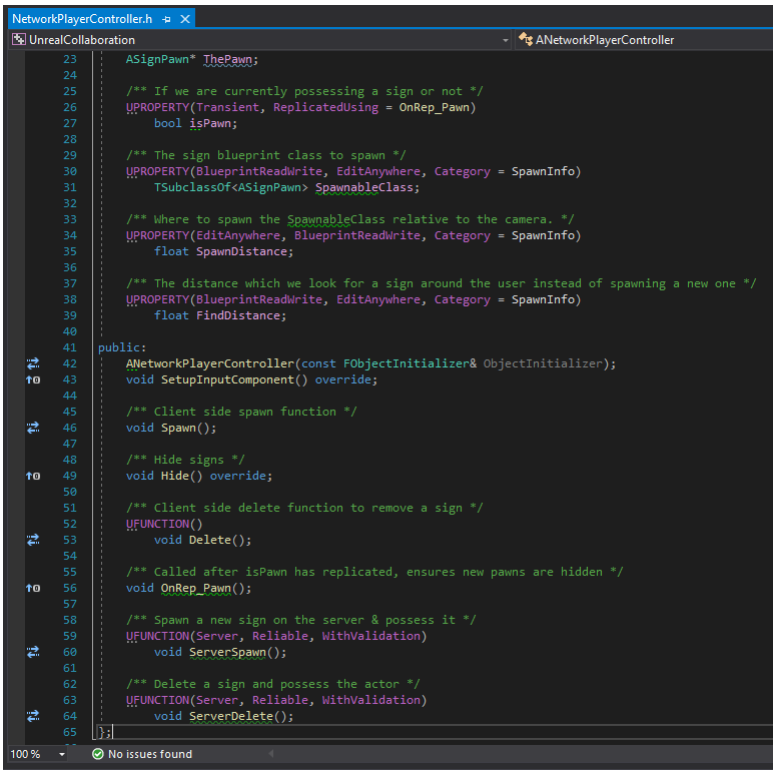

Figure 16: The `UFUNCTION()` & `UPROPERTY()` macros in `NetworkPlayerController.h`

6.1.2 Player Controller

The player controller is the root of control within Unreal Engine, controlling how player inputs are mapped to in game actions. Within **Unreal Collaboration**, the player controller is used to control two things:

- Controlling the switch between player character & sign pawn
- Controlling how the **Unreal Collaboration** server stays alive by resetting the timer within **Unreal Selector**

The player controller controls the switch between player character & sign pawn by a calling a server-side function whenever the player tries to spawn a pawn. The player controller will first search the world for nearby unpossessed signs, and attempt to set the *instigator* of the pawn if one is found before possessing it. If one is not found, a new pawn is spawned a number of units away from where the camera is looking, the *instigator* on this pawn is set to be the player character, and it is possessed. However, if the player is already a pawn then we possess the pawns *instigator* (which will be the player controller who possessed the pawn) and continue.

Communication with **Unreal Selector** is handled through a HTTP Service that allows us to serialise structs to JSON to send to **Unreal Selector** (as shown in Figure 17). The service also handles deserialising JSON response back to structs using a generic function which can be used within code. To communicate with **Unreal Selector**, the **Unreal Collaboration** clients need to know the hostname of the **Unreal Selector** server, as well as the session generated from **Unreal Selector** which authenticates the user. This information is provided from a JSON file **Unreal Selector** places in the **Unreal Collaboration** client directory before it launches the application, this JSON file can be deserialised into a struct which we can use within code.

```
[2020-04-27 14:59:46,844] INFO in connect: /game_info: (127.0.0.1) Echoed project Helmsdale version (3)
127.0.0.1 - - [27/Apr/2020 14:59:46] "POST /api/project/info HTTP/1.1" 200 -
[2020-04-27 14:59:46,867] INFO in connect: /game_keepalive: (127.0.0.1) Started up Helmsdale(7777) with pid 20200
127.0.0.1 - - [27/Apr/2020 14:59:46] "POST /api/project/keepalive HTTP/1.1" 200 -
[2020-04-27 14:59:54,392] INFO in connect: /time: (127.0.0.1) Echoed server check time 60
127.0.0.1 - - [27/Apr/2020 14:59:54] "POST /api/project/time HTTP/1.1" 200 -
[2020-04-27 14:59:54,392] INFO in connect: /account_info: (127.0.0.1) Gave info to user user - (2, user)!
127.0.0.1 - - [27/Apr/2020 14:59:54] "POST /api/account/info HTTP/1.1" 200 -
127.0.0.1 - - [27/Apr/2020 15:00:06] "POST /api/project/keepalive HTTP/1.1" 404 -
[2020-04-27 15:01:06,644] INFO in connect: /game_keepalive: (127.0.0.1) Updated server Helmsdale
127.0.0.1 - - [27/Apr/2020 15:01:06] "POST /api/project/keepalive HTTP/1.1" 200 -
```

Figure 17: Log of a user opening the Helmsdale project with a current idle check time of 1 minute. We obverse the **Unreal Collaboration** client got rate limited from resetting the timer directly after loading into the game.

This data from **Unreal Selector** is used to reset the internal idle timer & keep the server alive, as well as query player information to use in game. When multiple clients are connected to the server, it is important to ensure that clients reset the idle timer for the server at different times to ensure the server isn't flooded with requests, we can do this easily by starting the timer which resets the **Unreal Selector** idle timer with a random offset.

6.1.3 Character

The actual player character being controlled by the player controller can also call **Unreal Selector** to query their information and change their properties from this information, such as the players username. A client can query **Unreal Selector** for this information at anytime, when this response has completed successfully, the response is used to set replicated variables on the server & call a Blueprint exposed function which can use the newly updated variables (as shown in Figure 18). One of these replicated variables is the player username, which is used to set the players username text actor which spawns on top of the players head.

BaseNetworkCharacter.cpp

UnrealCollaboration → ABaseNetworkCharacter

```
103 }
104
105 // Client-side change function
106 void ABaseNetworkCharacter::Change(FInfoStruct_Response response) {
107     if (GetLocalRole() == ROLE_AutonomousProxy) {
108         // Set variables on client
109         username = response.username;
110         info = response.info;
111         rank = response.rank;
112         isAdmin = response.isAdmin;
113
114         // Allow the client to call blueprints if given
115         if(ClientUponInfoChangedEventCalled) UponInfoChanged();
116
117         // Let Blueprint know, then update the server
118         ServerChange(response);
119     }
120 }
121
122 // Server side change function
123 void ABaseNetworkCharacter::ServerChange_Implementation(FInfoStruct_Response response) {
124     // Set variables on server
125     username = response.username;
126     info = response.info;
127     rank = response.rank;
128     isAdmin = response.isAdmin;
129
130     // Update the controller
131     ABaseNetworkPlayerController* controller = Cast<ABaseNetworkPlayerController>(GetController());
132     if (controller) controller->isAdmin = isAdmin;
133
134     // Call info changed for that user
135     UponInfoChanged();
136
137     // Update our name
138     TextActor->text = FText::FromString(username);
139 }
140
141 // Validation functions
142 bool ABaseNetworkCharacter::ServerChange_Validate(FInfoStruct_Response response) {
143     return true;
144 }
```

Figure 18: Code from `BaseNetworkChrarcter.cpp`

These text actors are originally spawned on the server when a character enters the world, and replicated to all other clients. The actors text is updated on the server as a result of a chain of callback functions resulting from the call to the **Unreal Selector** API, causing the text to update on all other clients. We also obverse the call to `UponInfoChanged()`, which can trigger events within Blueprints, causing more actions to occur.

While the text actor itself is replicated, its properties are not by default - such as the text its rendering, its movement, etc. These properties are replicated using a custom text render actor class that takes advantage of server sync functions (Figure 19). When the server sets the text of the text actor, its properties are not automatically updated. To update these properties we set them to replicate (`text` & `textColor`), and use server sync functions for the `OnRep_[x]` functions to be called whenever one of these variables is replicated. Finally, we can use the newly updated variable to update properties of the actor.

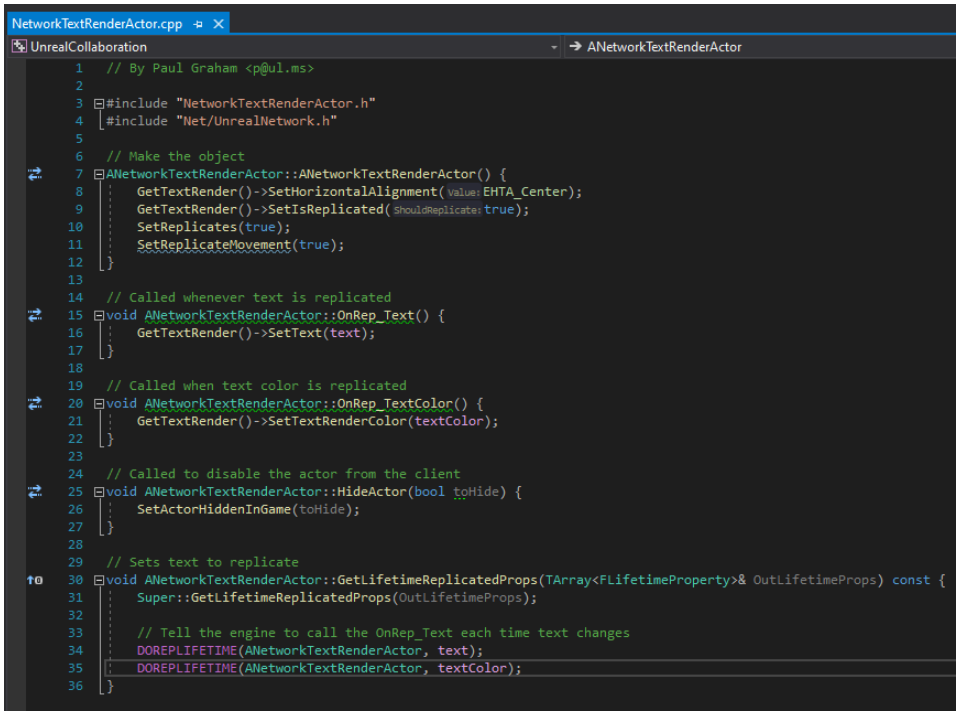

Figure 19: Code from `NetworkTextRenderActor.cpp`

Another important property of the player character is that it is allowed to move *anywhere* within the world, and view the world from any angle. This is handled by allowing the player character to always jump (even if they are no longer on the ground) and to move around in the air in a way that feels nice to use - we found by trail and error that player movement felt nice to use when `Air Control` was set to 1, and `Gravity` was set to 0.35 (although developers can change this via Blueprint).

6.1.4 Sign

The sign is controlled through UI elements that allow the player to enter in text to the sign, confirm the edit & delete the sign (as shown in Figure 20). When the player controller successfully replicates if the player is currently a pawn from the server to the client, the visibility of these UI elements is toggled to share this same state. The UI elements are spawned from a UI Widget, and controlled by the HUD manager which allows functions in both the controller (to delete the sign from the world and repossess the player character) & sign pawn to be called. Whenever a button in the UI is called, the function inside the HUD Manager is called, this can do some computation before the text on the actual pawn is updated, such as checking it is under some maximum length requirement.

The actual sign pawn is a replicated actor which holds a text actor (to draw the text on the sign), and can be hidden. The ability to be hide every sign is controlled by the player controller, which loops over every sign

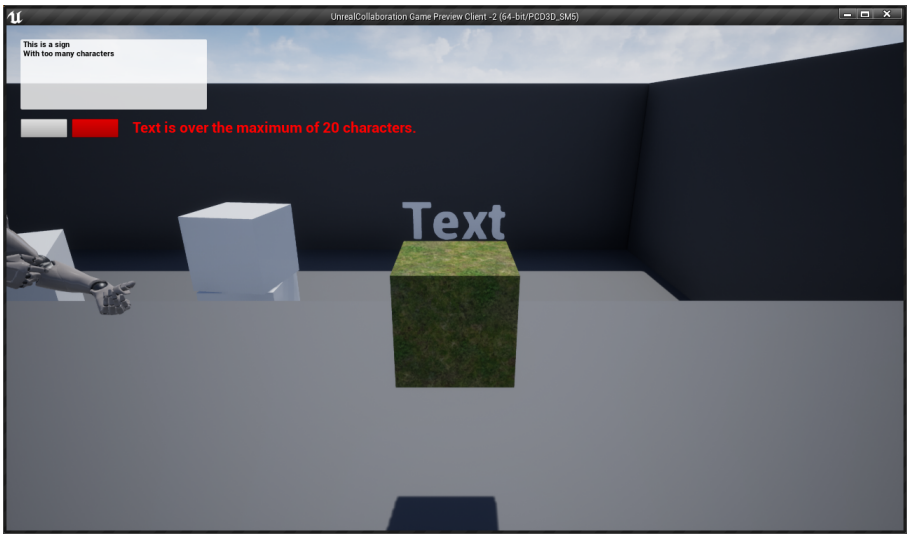

Figure 20: The sign pawn inside of the testing project, the sign UI elements are overlaid. We obverse the player has went over the character limit of 20 characters.

and calls sets them to be hidden. When a new sign is created while signs are hidden, we use the replication of the text actor on that sign to hide it locally for the player it was just replicated to. As with the username actor for the player character, the pawn & attached text actor can also rotate to always face the player.

6.1.5 Virtual Reality

To implement Virtual Reality, we first need a way for the server to assign player controllers to players based upon if the player is within Virtual Reality or not. This is implemented through a custom **game mode**. As we discussed in **Section 6.1.1**, this structure only exists on the server & is responsible for how players enter, pause & exit the game. Within the game mode base class implemented within Unreal, there are variables which control the default class which will be used to set options such as the default player controller, pawn, HUD, etc.

To differentiate between players in VR and regular players, **Unreal Selector** will add an extra command line option to the **Unreal Collaboration** clients arguments - `VRMode`. This will be set to one when we are launching in VR and will not be present otherwise. Within the game mode we can override a function that contains these options from the client to decide what default player controller & default pawn classes we will use.

The VR Player Controller & VR Character are implemented differently from the regular player controller & player character we discussed previously. The VR Player controller strips out sign spawn & possession functions from the regular player controller since we cannot access UI elements easily while in VR. The VR Character functions similarly to the regular player character with the difference of implementing max speeds on all axes (so that players won't become sick while moving around) & implements rendering of the room-scale VR motion controllers by attaching them to a scene component at runtime, shown in Figure 21.

Movement of the VR Character is handled through the VR Player controller - the player can move around both my simply moving around in their room scale setup & by launching themselves in the direction of their right motion controller. The later is handled by calculating the vector to launch the player in, and then calling a function on the server to actually launch the player in that direction. The player can also turn around in VR both by turning in their room scale setup, and by changing the yaw of the player controller via their motion controllers.

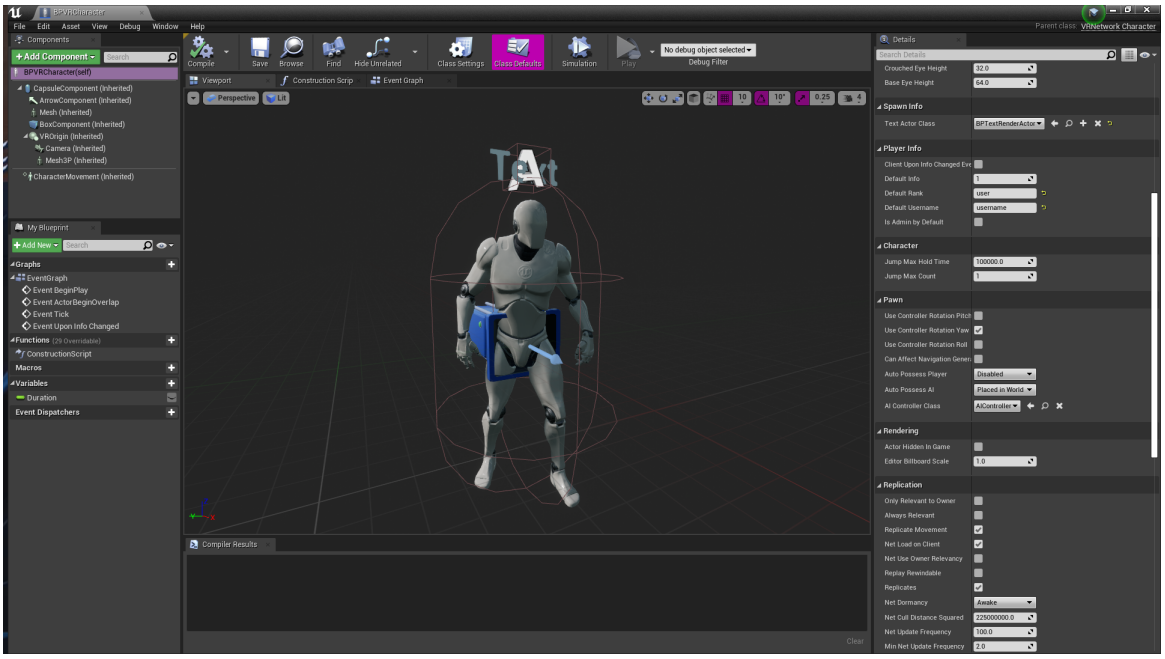

Figure 21: The structure of the VR Player within Unreal Editor. We obverse the motion controllers aren't present as these are spawned dynamically.

6.1.6 Build Tools

Unreal Collaboration also comes with a Python build tool to help adding the project to any existing Unreal C++ project. This build tool will download the latest copy of **Unreal Collaboration** and copy it into the target projects directory, replacing references to `UnrealCollaboration` with the target projects name. It finishes by adjusting the editor settings (stored within `Config/*.ini`) to allow **Unreal Collaboration** to be added, and to bind default inputs.

6.2 Unreal Selector

This section will focus on the implementation of **Unreal Selector**, we will start by looking at implementation of the **Unreal Selector** server and will move onto looking at the **Unreal Selector** client.

6.2.1 Server Overview

The server is built around three main components, which we will discuss in further sections:

- An account management system, which interfaces with the database to authenticate users to use the other components
- A standard HTTP server which serves HTML webpages, and static files to clients
- A RESTful JSON API which allows the **Unreal Collaboration** client application to interface with the database of user accounts & projects stored on the **Unreal Selector** server, as well as the actual server itself

6.2.2 Account Management

Within **Unreal Selector**, user accounts are used to map a user to a number of properties - username, player information, server rank, etc. User accounts are used throughout the RESTful API to allow us to ensure the user is always mapped to these properties when using the application. User registration & login forms are presented locally within the client application, with an additional form for the server the user would like to connect to. When submitted, these requests are served by the RESTful API on the server to either create the users account (they will have to now login with the newly created account) or to attempt to log the user into their account.

When login is successful, we generate a `session` token for the user which they can use to access all other endpoints as a substitute for their credentials, the user will also store this `session` token as a cookie. All endpoints use a single class to check this `session` token before allowing processing to continue, is the `session` token is ever invalid then any standard web endpoint will return a page which logs the user out of their account & all API endpoints will return a simple error string with the response code 401.

User accounts have different privileges depending upon the rank of the user, in **Unreal Selector** there are three types of user, with different privileges, as shown below:

- Normal users
- Project admins - Able to lock the server, and administer projects by adding, updating, or removing projects
- Server admins - Has all the rights of a project admin, but with the extra ability to change the rank or delete other users (who are not themselves server admins)

The server also supports the addition of custom developer functionality via only changing one file. Within this build of the project, integration with the School's CINE service [21] is supported (but not actually implemented due to the 2019-20 Coronavirus Pandemic - see **Section 3.2**). This would have worked by sending a request to CINE with their CINE username & password, along with a shared secret authenticating the server for integration with CINE. CINE would have responded with a status code of 200 if the user CINE details were correct, we could then make the user their **Unreal Selector** account, or log them in if it already existed.

6.2.3 HTTP Server

The HTTP Server will respond with HTML documents or static files upon requests so that the client can render out the main interface for the application. This component is also authenticated by checking the `session` cookie when a request is made, a logout page is returned if it was incorrect.

Flask also offers support for Jinja [22], a tool which allows one to insert code directly into HTML documents to get processed on the server before they are sent to the client. With this tool, the server can render a document by calling Jinja with the document to render, and a dictionary of variables, Jinja can then process this dictionary inside the document. Jinja has a lot of support for Processing within the document, being able to perform calculations, work with lists, process loops, extend base documents, etc. We can see how Jinja can prepare documents differently based upon variables given to it from the server in Figure 22.

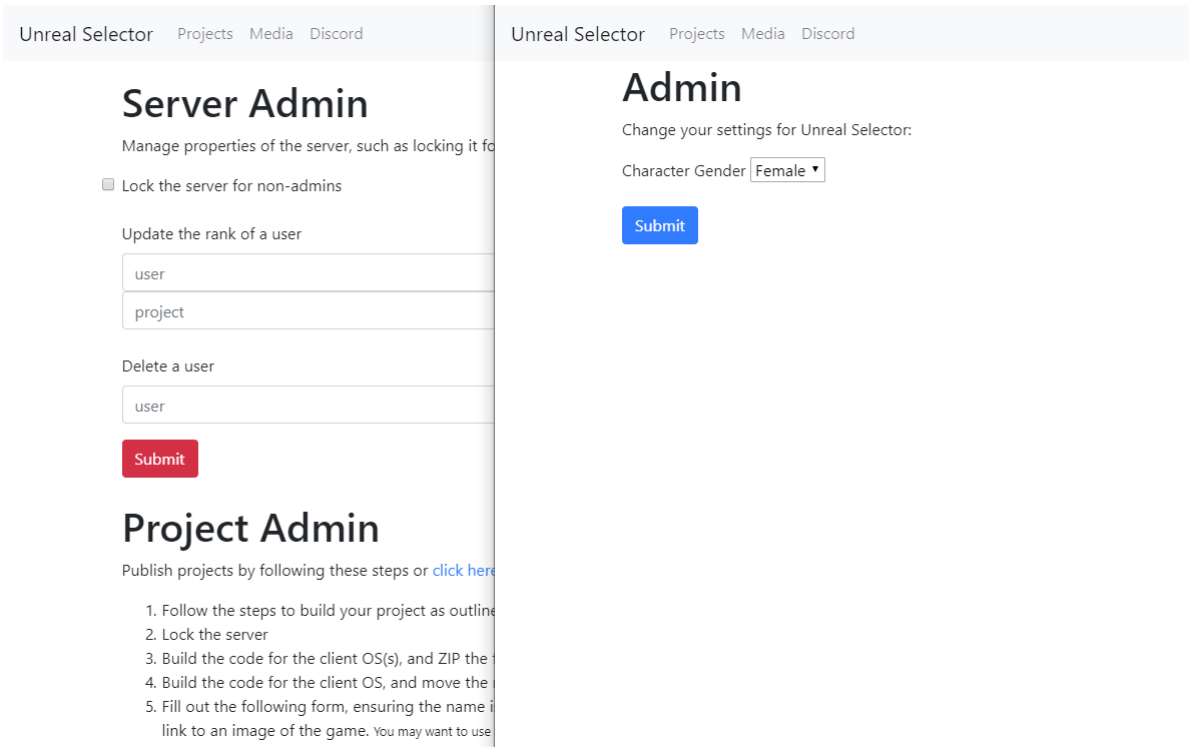

Figure 22: We obverse the difference between two users viewing the same page, the admin user (*Left*) & the non-admin user (*Right*).

6.2.4 RESTful JSON API

The RESTful JSON API makes up the heart of the **Unreal Selector** server. It allows both the **Unreal Selector** client, and Unreal to communicate with the server & the database. A HTTP request can be sent to the API with the requests body being a JSON object, containing serialised data to be sent to the server, the server will respond with a HTTP response, the status code of which indicates the outcome of the request (i.e. 200 = Successful, 401 = Not authenticated, etc.), and the body of which is another JSON object the requester can use for their own purpose. An example request is shown in Figure 23.

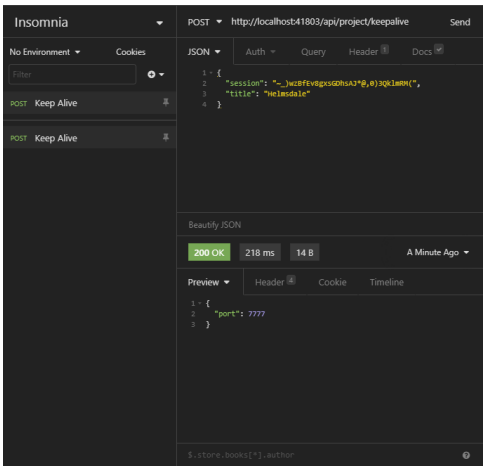

Figure 23: Sending a POST request to reset the idle timer for the Helmsdale project, we obverse authentication using the `session` key. The server has reset the timer, and responds with the port the server is open on.

Most of these endpoints interface with the embedded SQLite database stored on the server, issuing SQL instructions to store & retrieve data from it. As data is coming from the user, it could be potentially malicious, attempting to perform exploits such as SQL injection attacks. To deal with these problems, the instruction is not simply prepared by inserting user data into the instruction, it is only inserted after the data has been properly cleaned and can no longer pose a threat.

Other attacks rely on clients executing malicious Javascript from dynamically loaded pages. Since the only source of user input that appears in dynamically loaded pages is the project cards, these other attacks cannot be executed without first having a privileged account therefore we only do limited cleaning of user input to prevent these types of attacks.

Finally, we protect attempt to protect against malicious users issuing too many requests to the server using a custom fork of flask-limiter [23]. The limiter works on IP Addresses of clients, in memory, to limit the number of requests a user can perform in an amount of time, if this is ever above the limit then the request will be blocked and the user will be returned a response with status code 429 - Too Many Requests. Interestingly, returning 429 to a download within electron does not result in the error callback function being called (we do not know if this is a problem with chrome or electron), therefore flask-limiter was extended to support custom response codes so that we can limit downloads within our electron client.

There also exist endpoints that interface with other things on the server, other than the database, such as launching & controlling **Unreal Collaboration** servers with the idle timer approach. Whenever a server is launched or updated, an entry is updated in a dictionary of the servers name, the time the idle timer expires at, and the process ID of the **Unreal Collaboration** server process. A background scheduler checks this dictionary every user-defined time unit, and will terminate any process who's idle timer has expired, thus stopping the server.

6.2.5 Administration

An important component of the RESTful JSON API is project administration - adding, updating & removing projects from view of the client over time. To update or remove a project will remove compatibility for clients who already have that project, so we do want want clients to be able to access their projects through the **Unreal Selector** client while changes to projects are being made. To solve this, we introduce a server-wide lock, this lock will only apply to non-admins and will return a HTTP response containing an error message and a 403 Forbidden response code to API requests (shown in Figure 24), or return the logout page to other endpoints. This lock ensures that clients will not be able to start incompatible servers, or download incompatible clients while they are being updated. It also allows for admins to ensure the server is setup property before any client is allowed to sign up and use their server.

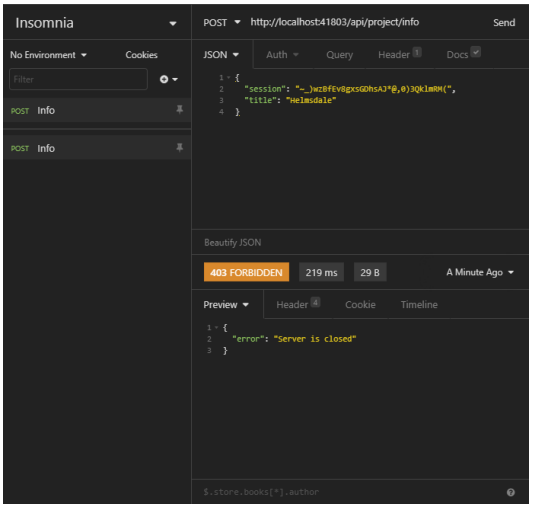

Figure 24: An example of how the server responds to non-admin users when the server is locked.

After the project has been successfully updated or removed, **Unreal Collaboration** clients will not allow clients to play incompatible projects, and will require the client to update projects to their new version before they can be played.

6.2.6 Client Overview

The client is built to serve three main functions, which we will discuss in further sections:

- Provide a user interface for interaction with the **Unreal Selector** server
- Open **Unreal Collaboration** projects for users
- Provide a way for clients to communicate via Discord

6.2.7 User Interface

Electron [4] allows for the implementation of cross-platform desktop applications through Chromium & NodeJS. An electron application is essentially a very lightweight chrome browser, with added functionality through NodeJS, and therefore designing a user interface for an electron application is identical for designing one for the web. **Unreal Selector** uses Bootstrap [19] for allowing elements within the page to be responsive, fitting to any display size & improving the default page styling. An example of a page within **Unreal Selector** is shown in Figure 25.

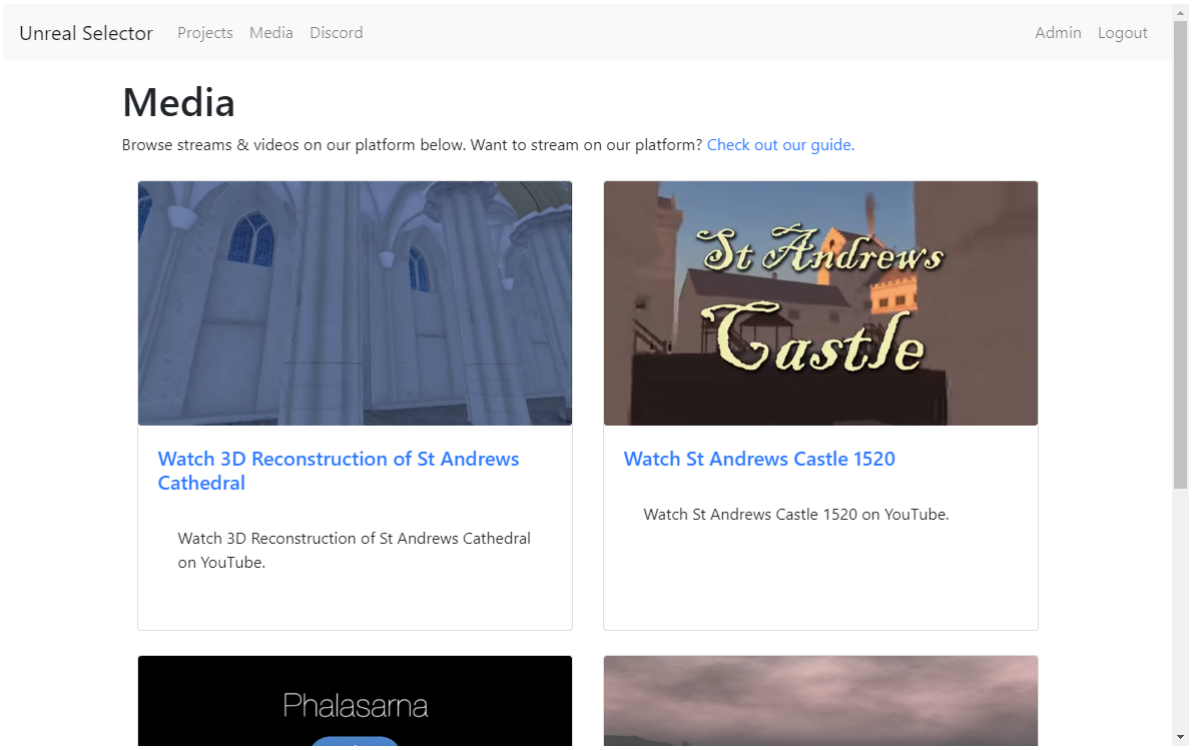

Figure 25: A screenshot of the media related media within the selector. We obverse no twitch streams are currently live.

The default user interface is designed to be accessible to all users & easy to support with Jinja which automates the creation of dynamic elements on the page. Most of these automated elements are placed within cards to create a boundry between the card & other elements of the page. Modals are created to give the user additional information, or highlight important actions within the page - such as ensuring an admin *really* wants to remove a project & will follow additional steps to fully remove it from the library.

6.2.8 Opening Unreal Collaboration Projects

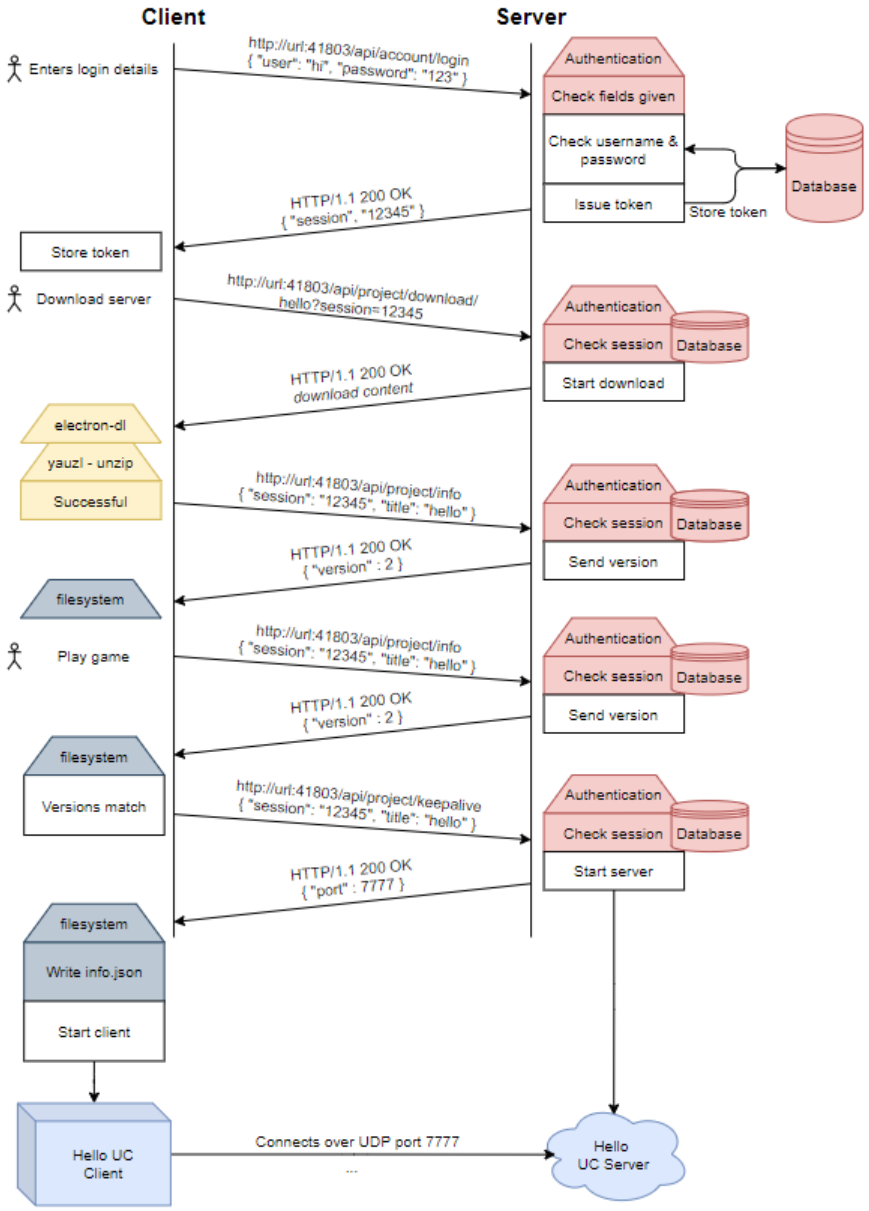

Figure 26: A diagram of communication between an **Unreal Selector** client & server to download & play a project called "*hello*". Note that further interaction when the **Unreal Collaboration** client begins is not shown, all stick figure actions have been issued by the user interacting with the UI.

The following section refers to Figure 26. When a client wants to play a **Unreal Collaboration** project, we first need to obtain the built client for the project from the server. These are obtained from a special endpoint on the JSON API where the `session` token is sent via the URL instead of via a cookie to avoid the client having to send a JSON body with the request (which the download library, electron-dl [24] doesn't support). While a download is ongoing, we prevent the user from starting any other downloads until the current one is completed.

The download then begins via a custom implementation of electron-dl, this library provides a better interface to download files than electron supports natively. Our implementation of electron-dl adds a callback function whenever the download is interrupted (this will happen if there is an error within the server, or the client is

rate limited from making too many requests - see Figure 27 for an example of what these errors look like on the client). Once the download of the zipped client files has completed successfully, we begin extracting the contents of the zip file into its own folder, and removing the unnecessary zipped client files. Any errors during this process will be caught successfully & an informative error will be presented to the user.

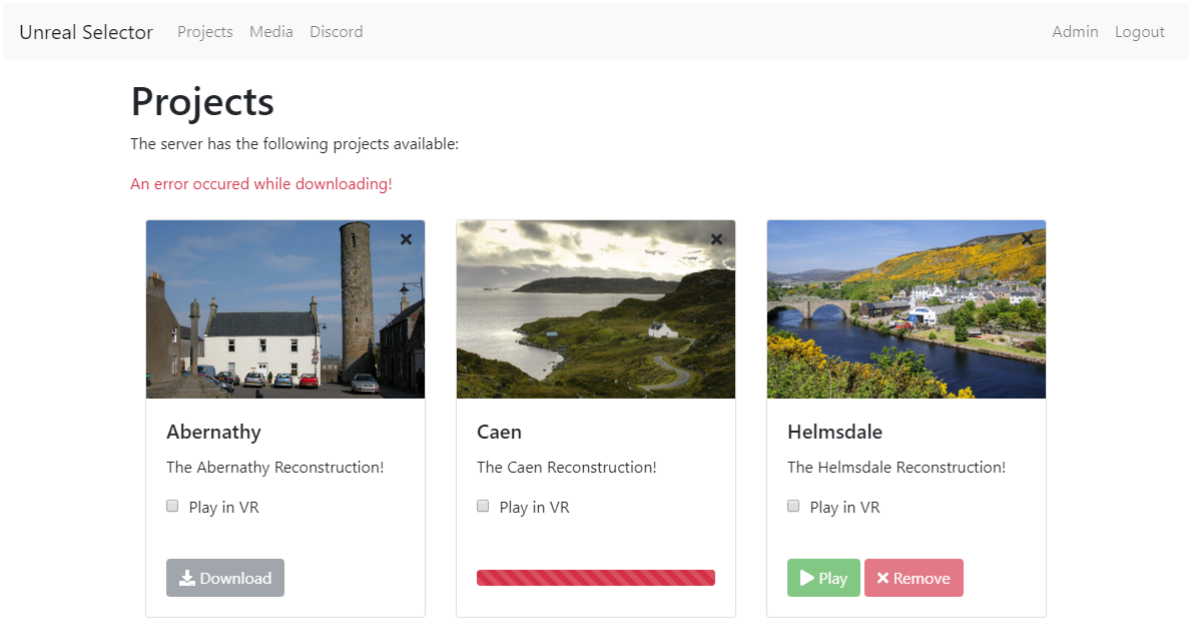

Figure 27: A download error occurs as it cannot find the Caen reconstruction.

Once the download is complete, we create a file which holds the current version of the downloaded project (got through the JSON API) to use when checking for updates. The buttons on the frontend will add in a new remove button & change the download button to a play button. When play is pressed, we begin by checking if there is a newer version available, if there is then we delete the current game files, and ask the client to redownload. If the version matches the latest version available, we begin by requesting the server to start up, or reset the idle timer for, the **Unreal Collaboration** server for us to connect to. The server will return 200 and the port number assigned for the client to connect using if successful.

Finally, we have to get the IP Address of the server from the hostname to tell the **Unreal Collaboration** client where to connect to, we can get this by using the DNS resolver built into electron. We then write some information to an `info.json` file that the **Unreal Collaboration** client can read which contains the URL of the **Unreal Selector** server, the port the **Unreal Collaboration** server is listening on, the users current `session` token & the name of the **Unreal Collaboration** project. We can then launch the **Unreal Collaboration** client with command line arguments telling it the server to connect to, whether or not to launch in VR & the port of the server to connect to. The user can then enjoy exploring with others connected to that **Unreal Collaboration** server.

6.2.9 Discord

Discord is an important part to the **Unreal Selector** system as it allows for clients to communicate with each other while they are exploring within the projects. The embedded discord is implemented through TitanEmbeds [20], which implements almost all Discord features apart from support for voice (which users will need to download the full version of Discord to access), and is shown in Figure 28. Included with the default Discord server is **Unreal Selector Bot** which allows for automation of some server functionality.

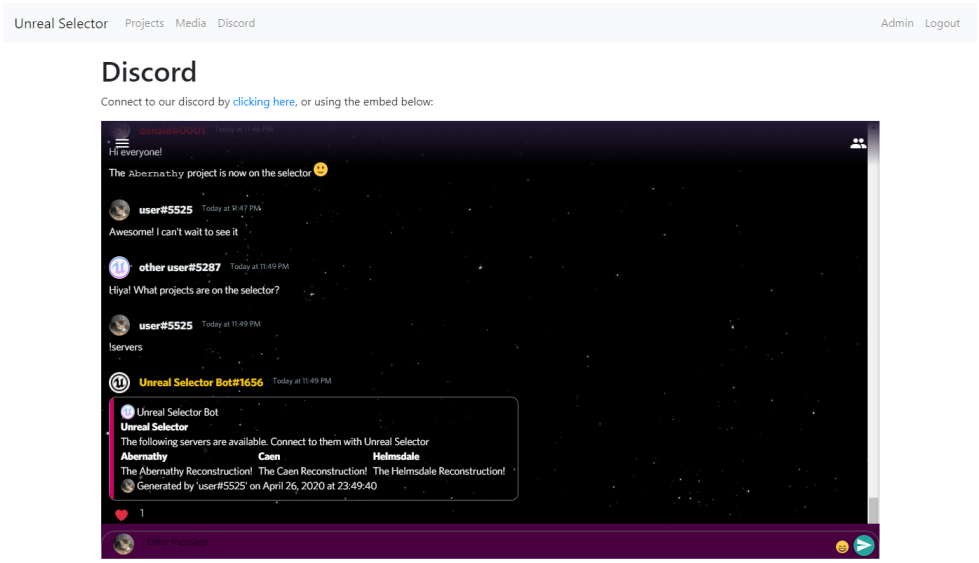

Figure 28: Several users talking in the embedded discord. A user prompts the bot to show the available **Unreal Selector** servers by using `!servers`.

Unreal Selector Bot is implemented in Python, using the discord.py library [25]. It allows for users both in the official Discord application & **Unreal Selector** to issue commands which the bot can respond to, such as the `!download` command to allow official Discord users to download **Unreal Selector** & `!servers` which allows users to get a list of official **Unreal Selector** servers they can connect to.

Additional functionality can be added to the embedded Discord implementation through the use of NodeJS within Electron, we use this to ensure that links within the embed open in within the users default browser instead of inside the **Unreal Selector** window. Since we can get the link the user wants to communicate to, developers could also add links to functionality within **Unreal Selector** itself if required.

7 Evaluation & Critical Appraisal

7.1 Testing

Frequent testing occurred after changes to either component of the system. Overall testing can be broken down into testing **Unreal Collaboration** alone, testing **Unreal Selector** alone and testing their combination.

7.1.1 Unreal Collaboration

Unreal Collaboration could be tested entirely within the editor by unchecking the `Use Single Process` option under advanced play settings - this would create a separate server process, and a number of client processes which would connect to the sever, allowing us to test features within **Unreal Collaboration** in the context they would be used within **Unreal Selector**. Additionally, testing could be done with the `Use Single Process` option checked to support breakpoints in server functions, although sometimes this lead to behaviour that was not going to occur when it was used within **Unreal Selector**. Finally, it was sometimes useful to analyse what exactly was happening within the server (when evaluating if properties were being replicated), this was achieved by changing the networking mode to the **listen server** mode.

When virtual reality development started within **Unreal Collaboration**, the testing setup was much more complex. The full room-scale virtual reality setup was switched on, with the controllers set in front of the headset so that each controller was distinguishable. Virtual reality development in multiplayer was hard as a

problem within SteamVR [26] caused SteamVR (and so the editor) to crash when entering VR with the server running in a different process for the first time (This may be due to a non-blocking mutex within the application, but I cannot confirm this). A workaround was found by launching VR with the integrated server process first, before switching back to the separate server. More on this problem is given in the build instructions **Appendix 9.1**.

Additionally, testing within VR was rather physically draining as the headset was sore to wear for a long time, it was also sore to repeatedly put the headset on. It was also mentally draining, as some problems with the implementation caused the camera to not rotate about a point, and therefore cause feelings of nausea. Because of these problems, testing within VR was minimised.

7.1.2 Unreal Selector

Testing **Unreal Selector** alone was also relatively easy as most of it runs entirely separate from **Unreal Collaboration**. The server could be setup with assets which could trigger all functions of the client, e.g. error handling on the client, testing how server controls the **Unreal Collaboration** server process. The **Unreal Selector** client could be tested by using the integrated inspect & debug tools within chromium (a component of electron). The **Unreal Selector** server itself could be tested in multiple ways:

- By inspection: using either simple print statements, or breakpoints
- By request: the API could be tested by sending it prepared JSON using a REST client (such as Insomnia [27])
- By the client: the client could also send requests to the server, and render responses - this was most useful for investigating what variables were being transferred to the client using Jinja [22]

7.1.3 Composition

Finally, testing them in composition was also relatively easy as the implementation of how data is transferred could be tested independently: **Unreal Selector** writes a file to somewhere a **Unreal Collaboration** client can read, and uses this file to call endpoints within the **Unreal Selector** server. This file could be created by hand and placed within the directory structure of the current Unreal project and tested on the **Unreal Collaboration** server alone. Communication between the **Unreal Collaboration** & **Unreal Selector** is handled by the **Unreal Selector** API which could also be tested independently.

Building of projects was postponed until the very end of the project due to them taking a very long time to complete building both within the editor, and within the IDE every time a build was requested (this lengthy procedure had to be repeated for builds targeting different operating systems). When the final build was made, they were added to the library of projects within our local copy of **Unreal Selector** for us to explore.

7.2 Evaluation of Original Objectives

When beginning this project, we identified several objectives which outlined the success of the project, each with a level indicating it's importance to the success of the project as a whole. The following section evaluates the final project compares to these objectives. The primary goals were as follows:

- Client which can connect to the server, and allows the user to:
  - Select a virtual environment from a library
  - Explore the selected environment
  - See representations of other clients in the environment

This objective was fully met, with users being able to connect to a **Unreal Selector** server and can join any **Unreal Collaboration** project listed in that servers library. The user is able to fully explore the environment by being able to freely fly around the world, thus are able to view the world from *any* angle. Finally, the user can see other clients in the environment as realistic characters within the world. Additionally, users are able to identify other others in the world via their username displayed above their head.

- Server which allows client connections, which:
  - Holds the library of all explorable environments
  - Serves a client their selected environment & representation of other connected users

This objective was fully met, the user is allowed to connect to **Unreal Selector** servers and view list of all **Unreal Collaboration** projects available for the user to connect to. Admins are able to add, upgrade & remove projects over time to change the library of projects over time. When the user wants to explore a world, the user downloads the **Unreal Collaboration** client, which holds the representation of the explorable world. The user also starts an **Unreal Collaboration** dedicated server which will replicate representations of other clients over the network.

- Developers should be able import the application into both existing & new projects
  - It should take minimal setup for them to include working multiplayer
  - Importing into the library of explorable projects should be easy

This objective was fully met, developers may use the **Unreal Collaboration** build tools to import Unreal Collaboration into their exiting projects, or create a new project starting off with the simple example project we provide. Importing the built **Unreal Collaboration** projects into **Unreal Selector** is also simple, with the only operations being file manipulation & zipping. A document is also provided to help developers build & use both **Unreal Collaboration** & **Unreal Selector**, this is given in **Appendix 9.1**. The secondary objectives were defined as follows:

- The ability for clients to dynamically change the environment:
  - Add markers to explain more about a certain place
  - Transition dynamically between environments (e.g. by walking to a certain area)

This objective was partially met, with the client being able to add signs in the environment to explain more about structures within the virtual environment. These signs are also replicated so other clients can also see & interact with these signs. Transitioning between environments is not supported within the **Unreal Collaboration** client itself, but instead players can transition between environments using the **Unreal Selector** client.

- The ability for clients to interact through:
  - Text chat
  - Simple gestures

This objective was partially met, with users being able to communicate with each other through the embedded Discord text chat in **Unreal Selector**. Simple gestures are not supported within **Unreal Collaboration**, instead users can interact with each other via the Discord text chat & Discord voice chat, which the user can access from within **Unreal Selector**.

- Support for Room-Scale Virtual Reality (e.g. HTC Vive)

This objective is fully met, the user is able to connect using their room-scale virtual reality setup, and have the same freedom of movement as in normal mode. This addition of signs, Discord messaging allows the project to provide users with additional support for collaboration between users. The addition of virtual reality support allows users to achieve even greater immersion while exploring virtual environments, hopefully increasing how much information they learn while exploring the environment. Finally, we outlined a tertiary objective:

- The ability for clients to add or remove objects in the environment

This objective is partially met, while the **Unreal Collaboration** clients cannot add arbitrary objects to the environment while exploring it, they can add informative signs to the environment.

Overall, all objectives were met, showing the project as an easy way to add support for collaboration into any Unreal Engine reconstruction - allowing clients to explore & learn about the immersive virtual environment with others.

7.3 Evaluation to Steam

Steam [12] (and others [1], [14]) is much like **Unreal Selector** in its core approach, to download & auto update applications the user downloads as we dicussed in **Section 2**. From this core functionality, Steam does not support the deployment of multiple servers the user can choose from to pick a game to play, but however has a global server registering all games users can play. While this approach is desirable in some circumstances, we don't think it's the best for a library of historic reconstructions as our multiple server approach allows institutions to tailor user experience toward their own reconstructions, and therefore increase engagement.

Additionally, Steam has a great auto-update system, both updating itself & subscribed games automatically without using too much bandwidth. Our implementation within **Unreal Selector** lacks a lot of features this system has, such as automatically queuing updates & not having too redownload the whole application every time, however it is much more complex to implement this better system.

7.4 Quantitative Evaluation

To evaluate the platform quantitatively, we considered an example setup with a single project, Helmsdale. It was deployed on a Windows 10 Computer, running an Intel i7-9700K with 32GB Memory & a NVIDIA GTX 970. The server was allowed to run, and 5 clients were connected to it, memory & CPU usage were then obtained from Task Manager to get an estimate of the requirements of running these reconstructions, as shown in Figure 29, the highlighted record is the server.

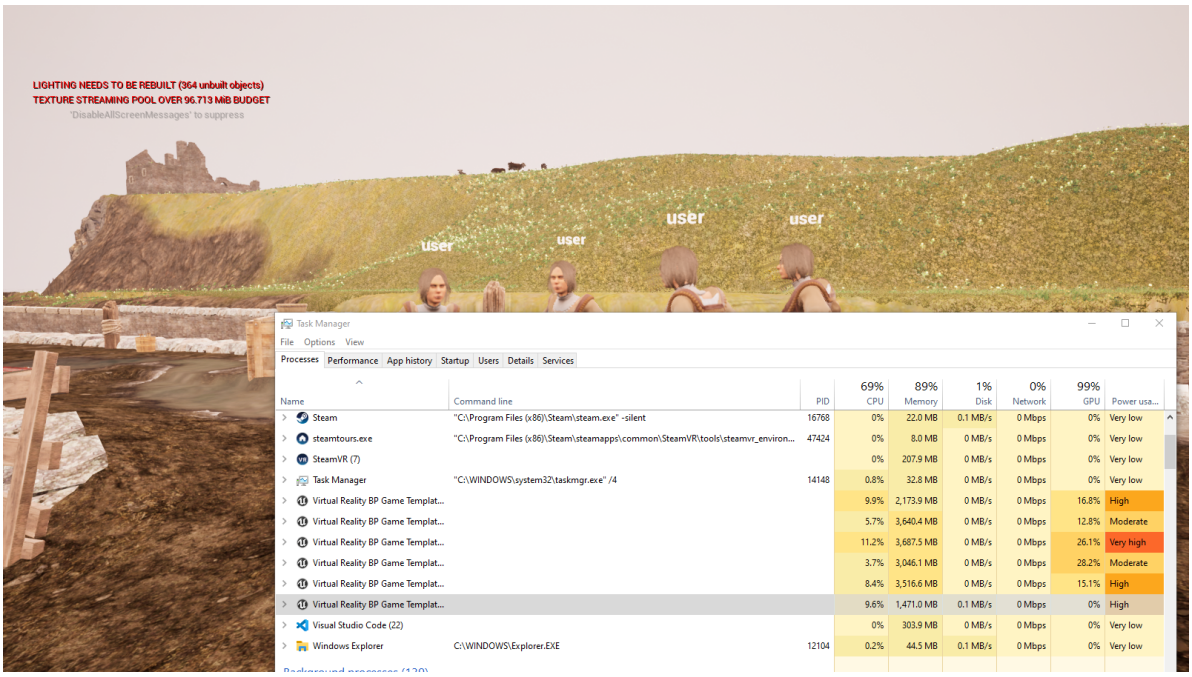

Figure 29: Readings from Task Manager readings of CPU, Memory and GPU Usage with 5 clients

We obverse the server running at about 10% CPU Usage, using 1.5GB memory. Each client has a similar breakdown, however using twice the amount of memory & about a fifth of the GPU each due to rendering assets. We also obverse the VSCode record below the server, this is running the whole **Unreal Selector** server for clients to connect to. This means to build a server capable of running several projects at once, an institution would need about `2GB` memory minimum, with an increase of `1.5GB` per additional project (however this would vary due to the size of the reconstructions). However, the server may also need to be equipped with a powerful CPU to handle running an additional copy of Unreal Engine in the background.

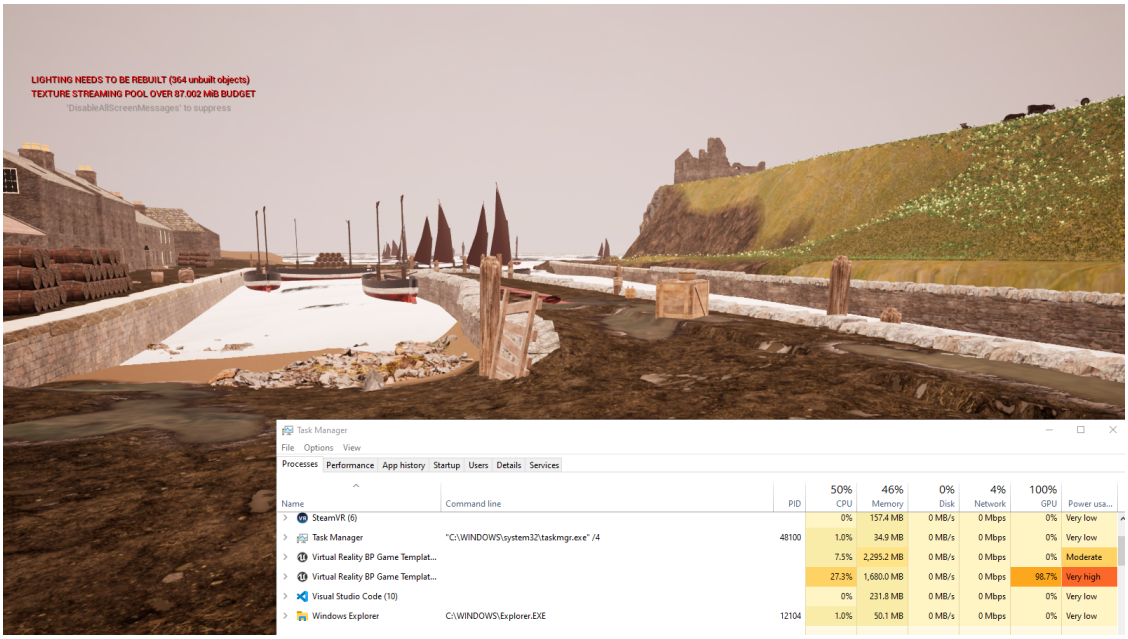

Figure 30: Readings from Task Manager readings of CPU, Memory and GPU Usage with a single client

It seems that clients are GPU bound within these built projects, with the GPU also going to 100% even when only one client is connected to the server (see Figure 30). This may be due to lack of culling with the renderer, and may be able to be solved from adjusting settings within the Unreal Editor itself? Overall, even with 5 clients connected at once, the experience was still playable on this machine, with each client being able to move around and explore.

8 Conclusions

8.1 Summary

In summary, we have introduced **Unreal Collaboration**: a toolkit for reconstruction developers to easily add support for collaborative exploration, and additional immersion with support for room-scale virtual reality. We have also introduced **Unreal Selector**: an extendable & upgradable platform for institutions to host **Unreal Collaboration** projects, and for users to view and explore within the library of **Unreal Collaboration** projects, explore media related to them and communicate with each other.

8.2 Future Work

8.2.1 Unreal Collaboration

We'd want to improve the second secondary objective by furthering support for communication within **Unreal Collaboration** itself (instead of relying on **Unreal Selector**). Unreal Engine supports voice chat with several different implementations (such as using the Steam online subsystem [28], or implementing it by interfacing directly with the codec [29]). Doing this will allow users to chat with each other without relying on Discord to provide this feature.

We'd also like focus on attempting to implement a streaming system for the first secondary objective by allowing users to transition dynamically between worlds without without having to use **Unreal Selector**. Unreal Engine supports dynamic travel between worlds on the same server [30] by having the client disconnect & reconnect to the same server, asking for a different world to be loaded. However, we would need to look into how to do this without the client having these worlds in advance. Additionally, we could also look into using this idea within projects with multiple worlds for clients to explore the same world at different points in time, for example.

8.2.2 Unreal Selector

Within **Unreal Selector**, we'd like to improve customizability for clients to allow them to have achieve greater immersion when exploring environments. Currently, clients are only able to change one setting about their character & this applies globally to all characters. We'd like to improve this by allowing developers to have multiple settings per-project so that developers can allow the client a fine-grained level of customizability in all projects. This would be implemented as an expansion to the current JSON based info system for clients information, but the database would be modified to support this change.

Additionally, we'd like to improve how updates are handled through **Unreal Selector** to improve the ease of use for clients. Currently, updates are handled by deleting the prior version of the project & asking clients to redownload the full version of the project to explore. For clients, it would be much better if clients could only download the differences between the files they have within their current version & the newer version on the server. This could be achieved by having the client send a manifest of what they currently have to the server (sending a compressed form of the output of `ls -lar` in the projects directory). The server would calculate the difference between the current project and the manifest the client provided, sending the client all files that are different.

9 Appendices

9.1 Build Instructions

This document is attached under `Build Instructions.pdf`.

References

[1] "Unreal engine - the most powerful real-time 3d creation platform." [Online]. Available: https://www.unrealengine.com/en-US/

[2] "Discord - free voice and text chat for gamers¡." [Online]. Available: https://discordapp.com/

[3] Pallets, "Flask - the python micro framework for building web applications." Apr 2020. [Online]. Available: https://github.com/pallets/flask/

[4] "Electron - build cross-platform desktop apps with javascript, html, and css." [Online]. Available: https://www.electronjs.org/

[5] A. Fabola, S. Kennedy, A. Miller, I. Oliver, J. Mccaffery, C. Cassidy, J. Clements, and A. Vermehren, "A virtual museum installation for time travel," *Communications in Computer and Information Science Immersive Learning Research Network*, p. 255–270, Jun 2017.

[6] E. Rhodes, A. Miller, C. Davies, I. Oliver, and S. Kennedy, "Viewing the past: Virtual time binoculars and the edinburgh 1544 reconstruction," *Communications in Computer and Information Science Immersive Learning Research Network*, p. 117–128, Jun 2019.

[7] S. Kennedy, R. Fawcett, A. Miller, L. Dow, R. Sweetman, A. Field, A. Campbell, I. Oliver, J. McCaffery, C. Allison, and et al., "Exploring canons & cathedrals with open virtual worlds: The recreation of st andrews cathedral, st andrews day, 1318," *2013 Digital Heritage International Congress (DigitalHeritage)*, Oct 2013.

[8] "Assassin's creed." [Online]. Available: https://www.ubisoft.com/en-gb/game/assassins-creed/

[9] "Assassin's creed brotherhood multiplayer gameplay #11." [Online]. Available: https://www.youtube.com/watch?v=RpY_hTdsESg

[10] "Aws case studies." [Online]. Available: https://aws.amazon.com/solutions/case-studies/

[11] "Customers — google cloud." [Online]. Available: https://cloud.google.com/customers

[12] "Steam store." [Online]. Available: https://store.steampowered.com/

[13] "Epic games store." [Online]. Available: https://www.epicgames.com/store/en-US/

[14] "Origin store." [Online]. Available: https://www.origin.com/gbr/en-us/store

[15] "Heroku." [Online]. Available: https://www.heroku.com/

[16] "What is scrum?" [Online]. Available: https://www.scrum.org/resources/what-is-scrum

[17] K. B. et al., "Agile manifesto," https://agilemanifesto.org/principles.html, 2001.

[18] Dan, "What is a hub world?" [Online]. Available: http://howtomakeanrpg.com/a/what-is-a-hub-world.html

[19] M. Otto and J. Thornton, "Bootstrap - the most popular html, css, and js library in the world." [Online]. Available: https://getbootstrap.com/

[20] "Titan embeds for discord." [Online]. Available: https://titanembeds.com/

[21] “Cine.” [Online]. Available: https://cineg.org/

[22] Pallets, “Jinja - a very fast and expressive template engine.” Apr 2020. [Online]. Available: https://github.com/pallets/jinja

[23] alisaifee, “Flask limiter - rate limiting extension for flask applications.” Mar 2020. [Online]. Available: https://github.com/alisaifee/flask-limiter

[24] sindresorhus, “Electron-dl - simplified file downloads for your electron app.” Jan 2020. [Online]. Available: https://github.com/sindresorhus/electron-dl

[25] Rapptz, “Discord.py - an api wrapper for discord written in python.” Apr 2020. [Online]. Available: https://github.com/Rapptz/discord.py

[26] Beryllium, “Game crashes when running in dx12 (directx12) mode from ue4 with steamvr,” Jul 2019. [Online]. Available: https://steamcommunity.com/app/250820/discussions/7/1640917832748103274/

[27] “Insomnia rest client.” [Online]. Available: https://insomnia.rest/

[28] “The easy way to add voice chat into your multiplayer unreal engine 4 game,” Apr 2020. [Online]. Available: https://couchlearn.com/the-easy-way-to-add-voice-chat-into-your-multiplayer-unreal-engine-4-game

[29] “How can i make in game voice communication?” [Online]. Available: https://answers.unrealengine.com/questions/49442/trying-to-make-in-game-voice-communication.html

[30] “Travelling in multiplayer — unreal engine multiplayer.” [Online]. Available: https://docs.unrealengine.com/en-US/Gameplay/Networking/Travelling/index.html