Multi-Actor VR Framework

Final Report

Team: May 17-04
Client/Advisor: Nir Keren

Team Members:
Nicholas Boos - Co-Webmaster
Andrew Buchta - Communications Lead
Marcus Eidahl - Co-Design Lead
John Heiling - Co-Webmaster
Thomas Kiss - Team Lead
Robert Slezak - Co-Design Lead

Logo Design: Robert Slezak

Email: may1704@iastate.edu
Website: may1704.sd.ece.iastate.edu

Table of Contents

Project Design

Project Need and Goal

The HTC Vive is one of the leading head mounted devices in the virtual reality (VR) industry, however currently it is only designed for single actor. The problem we faced was to develop a platform to facilitate multi-actor virtual reality environments. This platform will primarily be used for training simulations which would be impossible or too dangerous to simulate in the real world. Our solution to this problem was to use Unity and its built-in networking to allow multiple users to interact with each other and make decisions using a decision matrix in the same virtual reality environment. While in the environment, the user's actions and decisions are recorded so they can then be replayed after the simulation for further review.

Functional Requirements

I. Network Framework

The network framework needs to be a stable and reliable network connection with as few disconnects as possible. For optimal user experience, all users animations and movements needs to be smoothly synced between all users in the virtual reality environment (VRE). Finally, voice chat amongst the users must sound clear.

II. Decision Matrices

The decision matrices must be available to the users whenever they want them in the VRE.

Non-functional Requirements

I. Network Framework

When using voice chat, each user should have the ability to choose which other users they wish to communicate with. Interactable objects should be synchronized for all users meaning that objects should be in the same position for all users and that all users can interact with the objects. This is in order to simulate that they are in the same place. Users need to be able to connects with each other worldwide to run simulations regardless of location.

II. Decision Matrices

An authoring system for users to create decision matrices outside of the simulation so they can later be imported into the simulation. The choices of each decision matrix need to be randomly shuffled for each simulation to avoid user bias. All interactions and decisions with the matrices are logged so researchers can analyze the results.

III. Replay System

The ability to replay all users' actions so key decisions are able to be further evaluated. In addition, there should be the ability to change between different speeds of watching the replay.

Resource Requirements

There were both hardware and software resources that were required to complete this project. Multiple VR headsets were required to develop and test the multi actor requirements. We used HTC Vives as the VR hardware for this project. For software, we used the Unity game engine and the SteamVR library to allow for our project to function with various other VR hardware. This includes the Oculus Rift which we were also able to test with.

Risk Identification & Mitigation

To mitigate the potential risks we could encounter throughout our project's life cycle we held weekly team meetings and weekly demos to our client. During our team meetings we discussed what we planned to do for the upcoming week and any possible problems that could arise. This helped us to identify and plan for potential risks before they happened. The weekly demos helped us to mitigate the risks because we were able to get feedback from our client on what we produced each week. This feedback was crucial to find and correct problems quickly.

Functional Decomposition

This project was divided into four major sections. Networking, VR mechanics, decision analysis, and after action review. Networking involved synchronizing the movements of the various actors across the internet and transmitting voice communications between actors. The VR mechanics were developing systems that made moving and interacting with objects in VR simple and intuitive. This included schemes for movement, interacting with ui, and interacting with objects in the environment. Decision analysis was done by creating something called a decision matrix where an actor could choose options out of a grid of given options in real-time during a simulation. Their decisions are then recorded for analysis by researchers afterwards. After action review was split into recording the actions of the actors during the simulation and the ability to replay a scene afterwards for review by researchers or trainers.

User Interface Design

Network Menu

The network menu is designed so that the UI surrounds the VR player, allowing him or her to see all options he or she can configure when looking his or her surroundings. The UI gives the VR player the ability to select their gender, create VRE room, and join existing VRE rooms. Each configuration option uses a different UI to match the number of potential choices offered, ask for input from the VR player, or return a list of items the VR player can choose.

I. Gender Selection

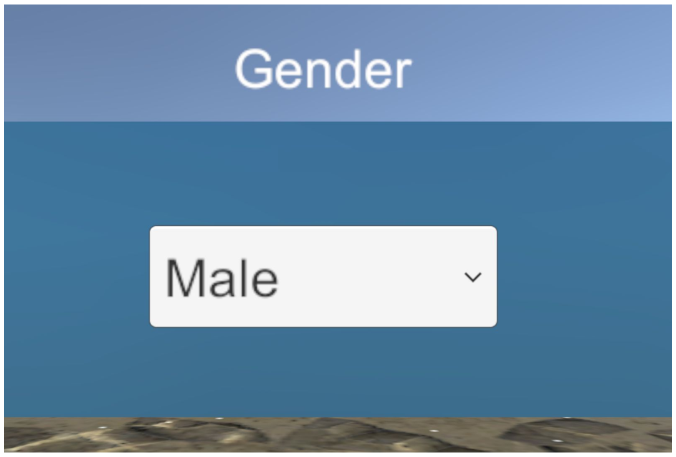

Fig. 1 - UI of the Gender Selection Option

The gender selection option consists of the title labeled “Gender” and a drop-down menu button. The title tells the VR player that he or she is looking at the option to select genders of their choice. The drop-down menu button was used since there are only two choices the VR player can select (male or female) and will only display the selected choice of gender, making sure that they will enter the VRE with that gender selected. When selecting genders, the two choices of genders will pop up and a checkmark will appear next to one of the choices, which indicates which gender is currently selected. The avatars representing the gender will be loaded when they create or join a VRE room.

II. Creating VRE Room

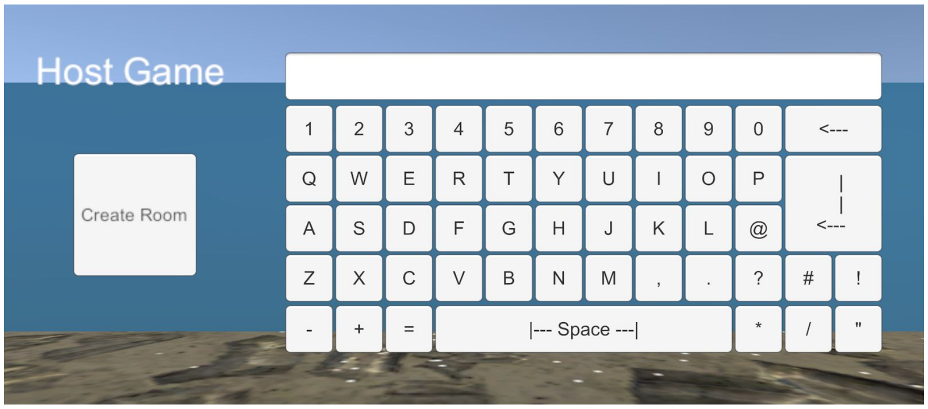

Fig. 2 - UI of the VRE Room Creation Option

The VRE room creation option consists of the title labeled “Host Game”, a room name input box, a QWERTY-style keyboard, a “Create Room” button. The “Host Game” title indicates to the VR player that he or she is looking at the VRE room creation option. The room name input box is placed above the keyboard to show the VR player that the input is manipulated by the keyboard. The keyboard is arranged similar to a QWERTY keyboard since it is a standard keyboard and the VR player will be most likely familiar with the layout. The “Create Room” button is self-explanatory; the VRE room will be created when the VR player clicks the button assuming that the room name in the room name input box is not blank and will be inside the newly created room.

III. Joining VRE room

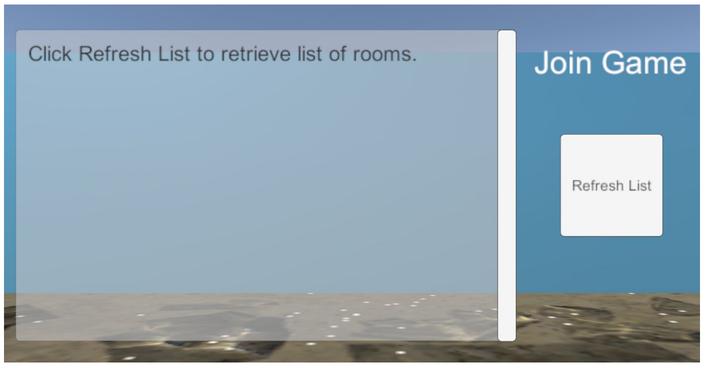

Fig. 3 - UI of the Join VRE Room Option

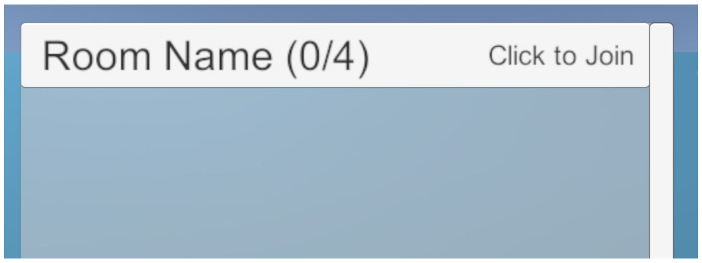

Fig. 4 - UI Element representing a VRE room that can be joined.

The Join VRE room options consists of the title labeled “Join Game”, a scroll list containing all rooms created, and a “Refresh List” button. The “Join Game” title indicates to the VR player that he or she is viewing the Join VRE room option. The “Refresh List” button tells the VR player that the scroll list will be updated when clicked. Initially, the scroll list will tell the VR player to click on the “Refresh List” button so that it can retrieve a list of rooms and display the existing VRE rooms as shown in Fig. 4 with the room name (“Room Name”) chosen by the VR player, the number of VR players out of the maximum number of VR players in the given room (“0/4” - *Note: This UI Element only appears when there is at least one VR player in the VRE room.*) and the text “Click to Join”, telling the VR player to click on it if the VR player desires to join the room.

Decision Matrix

The decision matrix system needed to be very user friendly for non-technical users. This required careful UI design to minimize questions within the matrix authoring environment and also during use in the live simulations. This consideration lead our decision to make a very simple graphical interface that had a large amount of back-end automation invisible to the user. This resulted in an easy and painless experience for the user with the only cost being long development time.

I. Authoring System

The Authoring system consists of a series of menus that allow the user to choose the name and dimensions of the matrix. Then, an empty matrix is presented to the user. The user can then aim at and select column and row titles to edit them, as well as individual square-shaped canvases to choose what informational content is presented upon a user clicking on that canvas. Once the user selects a canvas, they can either type text content with a virtual keyboard or select audio and video files from a dropdown to convert the text section into a video player or allow audio to play instead of text or video being presented. Text instructions are also displayed to the side of the system to help the user along the process.

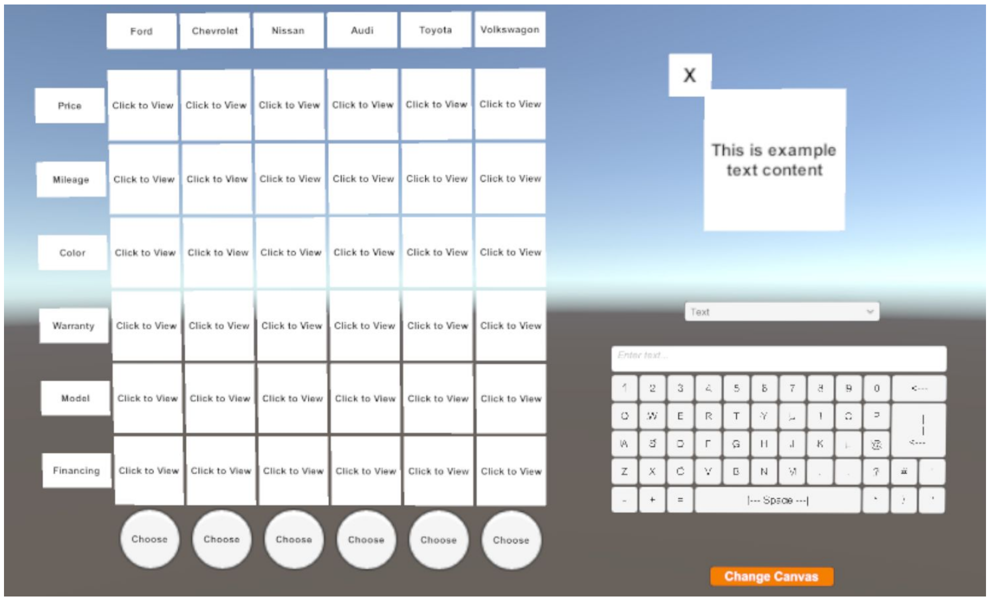

Fig. 5 - Authoring matrix UI, text instructions not pictured

II. Live Matrix

Upon a matrix being loaded, the matrix appears almost exactly the same as when it was being authored. The first main difference is there are no authoring tools (keyboard, dropdowns, save buttons). The second difference is the content is displayed directly in front of the matrix instead of off to the side. This matrix floats about 2 meters in front of the user and follows them as they move. It also has a semi-transparent appearance to remove the possibility of motion sickness while moving (not pictured).

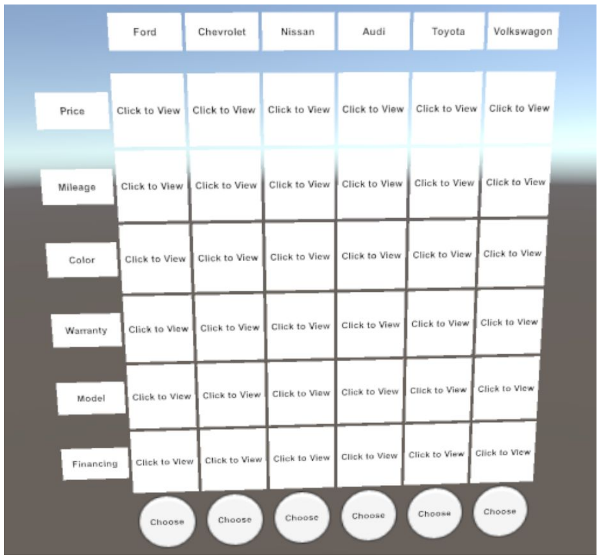

Fig. 6 - Live Matrix

Implementation

Networking

Networking is the main backbone for voice communication and for bringing VR players worldwide into a single VRE room and allowing them to interact with objects or make decisions together. The networking component is built upon UNET (Unity Networking) that is built into the Unity 3D Game Engine and is made up of several elements that help VR players create and join VRE rooms, establish voice communication, allow objects to be interacted by all VR players, and create avatars to represent VR players.

I. VR Network Manager

The VR Network Manager is a modified version of UNET's Network Manager that assist in initializing rooms for VR players to join inside a specified VRE scene. UNET's default Network Manager didn't fulfill our requirements for initializing voice communication and spawning different VR avatars that represented their gender, but UNET allows us to override the default functions with custom functionality to fulfill those requirements.. This allowed us to modify the Network Manager so that it can be tailored to VR. To meet the requirements, we modified the function that adds VR players to the VRE room so that it can load male or female avatars depending on their selection in the Gender Selection UI in the network menu. We also modified functions that run and stop the server and client to also control the voice communication in order to start the voice chat when the VR players join the VRE room and stop when the VR player leaves.

II. Matchmaking

Matchmaking is a method that allows VR players worldwide to create and join VRE rooms in remote locations to interact with each other without knowing technical details such as IP addresses and port numbers as the technical details are handled by servers. It is also an extension of our VR Network Manager since the VR Network Manager handles all data transfer locally. Matchmaking handles all VRE room creations, voice data transmissions, and object and VR player synchronizations on Master Servers provided by Unity. Matchmaking functions were created per Unity documentation and are attached to the VR Network Manager to enable matchmaking service and use the VR Network Manager functionality to create rooms in a remote location.

III. Avatars

Avatars are 3D model character representations of VR Players where their body position and hand states are manipulated by VR players' controllers. We have two avatars, each representing male and female avatars for VR players. To create the avatars, we used four separate 3D models to represent the player head of each gender and left and right hands to represent pair of HTC Vive controllers. Then we attached network identities, built-in UNET scripts that give objects network IDs for manipulation over the network, to the male and female heads and to the hands. Left and right hands contain a grabbing animation, which is useful to show that the VR player is grabbing an object or to show the current state of their hand. To manipulate the animation state of teach hand, we created a script that manipulated the animation states based on how far the HTC Vive controller triggers are pushed in. To spawn the VR players' avatars, we created a script that sent the server a command to spawn both the avatar head and the hands.

IV. Linear Interpolation and Smooth Movements

When an object transitions to new position (e.g. avatars moving inside the VRE room), transition often looks "choppy" or otherwise unnatural, which breaks the immersion for a more realistic experience. To make the objects transition from different positions look more natural, we created a script and attached them to objects to use linear interpolation, which predicts where the object is going until the new position coordinates are received from the server. This method makes the position transitions look more natural and enhances the immersion in the VRE.

V. Object Client Authority

By default, the server spawns all objects found in the VRE and gives the hosting VR player authority to manipulate the objects. Client VR players, however, are not given the authority to manipulate objects, which violates our requirements for having multiple VR players interact with each other and make decisions together. To fulfill this requirement, we created a script that gives all current and future client VR players the authority to manipulate objects when grab using an event listener provided by VRTK Unity asset and attached them to any object we want to synchronize over the network.

Decision Matrices

Within a virtual reality research or training simulation, situations often arise where the user needs to be presented with a decision to make. Whether it be how a firefighter wants to enter a burning building, or simply a customer choosing a car, the user will often have questions about each choice before making a decision. A viable method of solving this issue is through decision matrices, which are a grid of information available to a user before they make a decision. In the grid, each column represents a choice, while each row is a category of information about the choices. For example, if a customer is choosing a car, column titles would be: "Car 1", "Car 2", etc, while row titles could be: "Color", "Make", "Model", etc. If the user wants to learn more about Car 2's color, they would point at and select the square in the "Car 2" column and "Color" row. It this point there would either be text, audio, or a video content presented. After the user has learned what they wanted to about the decision, they can choose. All actions of the user within the matrix are recorded with timestamps for research or training purposes.

I. Authoring System

Implementing this functionality into a virtual reality environment was a complicated task. First, administrators with no assumed technical skills needed to be able to design a matrix to be put into a simulation. The system needed to be as user-friendly as possible, making creation of a matrix feel easy without any background technical knowledge. Our solution was an authoring environment where the user first picks the name and dimensions of the matrix. Then an empty decision matrix is built and presented to the user. The user can then click on column and row titles to edit their names and also click on each square in the matrix to edit the content presented upon a user selecting that square. Upon clicking a content square, the user can either type text with a virtual keyboard, or select audio or video files from dropdowns. Upon selecting a file, the system automatically converts the text field into a video player or completely removes the field and makes an audio file play instead of text/video being displayed. Once the administrator has edited each title and content square, they can then save the matrix. The system then parses the matrix and saves the details of the matrix into an XML file.

II. Live System

The second half of the task was allowing loading of these authored matrices into a real simulation and tracking interactions with them. First, functionality was developed to allow loading of a matrix when referenced by name. The system then rebuilds the matrix, makes it non-editable, and attaches a tracker. The decision matrix follows the player as they walk, and can be displayed and dismissed as needed. All viewing of content and the choice made is tracked to an xml file, which can then be viewed or parsed later by an external system.

After Action Review

The implementation for this is relatively straightforward. By relying on the fact that the scene itself is static, meaning that from one run to the next, everything that isn't either the player, or something the player can interact with, will always be in the same place. This works greatly to our benefit, as trying to record an entire scene is extremely costly, and will almost certainly result in poor performance of the scene. Our solution was to simply record all the necessary components to reenact exactly what happened in any given scene, and since the scene was static, that meant all we needed was the player and intractable objects. To be able to properly replay these objects, two attributes were recorded: position and rotation. Position is simply the object's location in space, and is described by a Vector3 (X, Y, Z) coordinate set. Being that this is three-dimensional space, we also need to know where the objects are facing. This is the rotation attribute. It is described by a Quaternion (X, Y, Z, W) coordinate set.

Once we knew what details we needed, how to record them was the next step. We decided the best approach was to write these attributes for each object being recorded, to a text file once every given period of time. With the given period of time being specified prior to runtime, the higher the time, the less accurate the recording with better performance, and the lower the time, the more accurate the recording with worse performance. A script that did all of this was attached to any object that was to be recorded. Then to replay, another script was attached to any object that was recorded, except the player character, which in the case of VR, got a little complicated.

Whenever a scene makes use of a SteamVR headset, that headset is spawned into the scene no matter what, and it immediately takes the place of the player character. So what we did in the case of the player, was simply created a separate game object with the replay script for the player attached to this. Another script was also needed to indicate to the other objects in the scene that this new object was taking the place of the player, and to ignore the VR headset, which still spawns and is used to view the scene as a third party. This script also held the master boolean that indicates if the scene was to be recorded or replayed.

Testing Process

Our testing process followed the general guideline displayed below. In short, the developers built each feature in an isolated environment. We live tested as we developed and had weekly demonstrations to our client to gather feedback. Once a feature was finalized and confirmed by the client, we merged that feature into a master environment to integration test. Finally, we demonstrated the integrated features to the client for validation.

General Testing Outline

1. Developer creates empty Virtual Reality (VR) environment
2. Import sample Unity assets (Basic objects, players, cameras)
3. Live smoke testing phase*
4. Test edge cases
5. Feature validation testing phase**
6. Integrate into “Master” environment
7. Integration testing phase***
8. Final Validation testing phase**

Live Smoke Testing Phase

1. Developer works at a VR workstation
2. Build outline of intended behavior for feature
3. Attach empty scripts and components to sample assets
4. Work on script and compile
5. Enter VR, test functionality
6. Repeat 4-5 until base functionality is in place

Validation testing phase

1. Demonstrate functionality to advisor/client
2. Answer questions
3. Ask for preferred adjustments/improvements
4. Implement adjustments

Integration testing phase

1. Import feature assets into master environment
2. Fix file/asset conflicts
3. Build activation functionality (ex: Button on controller)
4. Test in VR environment alongside other features

Testing Results

Through this process we were able to develop with close supervision of our client to ensure we delivered exactly what he requested. Careful live testing allowed continuous progress on each feature and integration testing made sure we resolved conflicts as soon as they arrived. This resulted in a very successful and efficient development process that minimized the amount of re-work that needed to be done.

Appendix I: Operation Manual

Multi-Actor Scene Setup

Configuration

Replay

To enable replay in the scene, attach the record movement and movement playback scripts to any objects to be recorded. In the editor set the desired interval (typically 0.1), as well as the type of object, i.e. if its a box, the string would be “Box.” In the case of the player character, only attach the record movement script and set the attributes in the editor accordingly. To replay the player character, a gameobject that two scripts will be attached to. One is the initialize replay player script, which will set the player to be the player in the scene, and not the VR headset. In this script is also the master boolean that indicates if the scene is to be recorded or replayed. Uncheck the box to record, check it to replay. Also attached to this object should be the movement playback script for the player character, as it will take the place of the VR headset during replay. Set the attributes in the editor accordingly.

Voice Chat

To implement voice chat you will first have to import the VoiceChat Unity Asset. Once the asset is imported attach the following scripts to an empty game object in your VRE scene.

- VoiceChatRecorder.cs
- VoiceChatSettings.cs
- StartRecording.cs

For voice chat to work correctly you must be using our custom Network Manager script and also make sure your computer's microphone and speakers settings are set to use the HTC Vive microphone and headset output.

Using Custom Camera Rig

If you want to use our VR avatars, but have a custom Camera Rig made specifically to the VRE scene, you can attach it to the VR avatars. To attach them, find the VR avatars, “male2” and “female2” under Multi-ActorNetworking\MAIN_PROJECT\NetworkedVRAssets\PlayerObjects. Click on both avatars and drag your custom Camera Rig to “Vr Camera Rig” under “VR Player Controller”.

Using Player Spawn Points

Player Spawn Points are required for VR players to spawn in your VRE. To get VR players to spawn in your VRE scene, open your VRE scene, create an empty GameObject in the scene and attach the Player Spawn Point script to it. Your VR player should be able to spawn and interact inside the VRE. You may position of the Player Spawn Point GameObject if you want to have the VR players spawn at a different area.

Adding VRE Scene to Build Setting and Setting it as Online Scene

To use your VRE scene as the online scene for VR players to join, your VRE scene must be part of the build settings, or the VR Network Manager will refuse to set it as the online scene. To add your VRE scene to the build settings, open your VRE scene and then click on File > Build Settings. Then click on Add Scene and make sure that it is checked. Now, open the NetworkMenu scene in Multi-ActorNetworking\MAIN_PROJECT\Scenes. In the Hierarchy, click on “Network Manager” and drag your VRE scene to the “Online Scene” in the VR Network Manager section. It is now ready to be connected.

Create/Join Room

You are now ready to interact within your VRE. If you are running the project in the Unity Editor, open the NetworkMenu scene in Multi-ActorNetworking\MAIN_PROJECT\Scenes and run it. If you are using a build, simply open the executable. Before you create or join a VRE room, make sure you selected your gender of your choice by pressing down on the trackpad on either HTC Vive controller and pressing down on the trigger while pointing to the dropdown box. Use the same method to select your gender.

If you are planning to host the VRE room, press the trackpad down on either of the HTC Vive controllers and press down on the trigger while pointing at one of the keys on the keyboard to enter a room name. You don’t need to hold down the trackpad. It will display in the input box above the keyboard. Once you finished entering the room name, point your HTC Vive controller at the “Create Room” button and press down on trigger. It will create the room and spawn you inside the VRE.

If you are planning to join an existing VRE room, turn to the right and while the pointer is on the “Refresh List” button, press down on the trigger. It will refresh and retrieve a list of rooms you can join. Point your HTC Vive controller to the room you want to join and press down on the trigger. It will start the connection to the server and spawn you into the VRE with other VR players.

Scene Replay

In order to enable replay in a scene, three scripts are needed. One is for recording movement (RecordMovement.cs), one for movement playback (MovementPlayback.cs), and the other is to initialize a new object to act as the player (ChangePlayerPrefab.cs). Before the initial running of the scene, a game object that will act as the replay player, will need to be created, and the script that will initialize it needs to be added, as this also holds the master boolean that will indicate if the scene is be recorded or replayed. Also needed to be attached to this object is the replay script for the player.

Once this is completed, any object that is not the player character, that is to be recorded and replayed, must have the record movement script and the movement playback scripts attached to them. In the case of the player character, only the record movement script will need to be added, as the object to be replayed has already been handled.

In the editor, for both the record movement and movement playback scripts, the interval of time for which to record the position and rotation of the objects must be set (typically 0.1). Also needed to be set, is the type of object that is being recorded, i.e. if its a box, the string "Box" will be entered. This is for saving purposes.

If this is the first time in the scene, be sure to uncheck the playback boolean in the script on the replay player game object to indicate that the scene should be recorded. In order to replay the scene, it must first be stopped, and then this same boolean should now be checked to indicate that this time, we instead want to view the replay. Once the play button is then pressed, the replay player will be spawned and start moving according to the recorded movement of the original VR player. The other objects recorded will also move accordingly.

Decision Matrix

Decision matrices are separated into two systems. The authoring system and live behavior. The authoring system is it's own scene with it's file structure already set up. Prior to authoring a matrix, video and audio files need to be put into their respective folders within the "Resources" directory. This needs to be done through the Unity Editor because Unity parses these files to make them usable assets within Unity. Then, copy the Resources folder into the "AuthoringSystem data" folder that is the same directory as the AuthoringSystem.exe file.

Once the files are prepared, launch the AuthoringSystem.exe to create a matrix following the displayed instructions. Once the matrix is finalized, click the "Save" button in the bottom right of the authoring display. This will create an XML file in the Resources/DM XML folder in the data folder previously mentioned. This is the save file of the decision matrix.

In order to load this matrix into a simulation, the "Resources" folder needs to be copied into the data folder of the chosen simulation. Then, the function LoadDM() needs to be called with the matrix name as the first argument and player camera gameobject as the second argument. This needs to be done via technical means by the creator of the simulation scene. This will spawn the decision matrix and attach it to the player whose camera was given as an argument. The player can then interact with the matrix and all actions will be recorded to an xml file called (Matrix Name) (timeandday).xml in the XML folder in the data directory. This file can be parsed by an external system to gather the data.

Appendix II: Design Changes

Game Engine

Early on in the design process we had to make a decision about what base software we wanted to start with to interface with the VR hardware. We looked into various solutions and eventually the decision came down to using Unreal game engine or Unity game engine as they seemed to have the best support for the VR hardware. The Unreal engine would have had us writing C++ code while Unity supported C#. We ultimately went with Unity game engine because our group decided we were more comfortable building in C# and our client was able to direct us towards some graduate students who had some previous experience using VR with Unity who we could ask more technical questions to.

P2V

Originally our client wanted position to velocity (P2V) movement inside the scene, but it was later scrapped due to not being fit for the current scene (International Space Station) we were working with.

Full Body Avatars

We considered using full body avatars to represent VR players rather than using a head and two hands. However, using full body avatars requires predicting how the VR players are positioned with their arms and legs, which may not be accurate, and creating full body animations when they are moving or grabbing. Seeing other VR players in unnatural positions breaks the immersion in the VRE and may generate a risk of “uncanny valley.” Furthermore, there wasn’t enough time to implement full body avatars due to its complexity. This idea was scrapped in favor of using avatar heads and hands, which is simpler and has been done before.

Networking

When we started the networking component of this project, we actually started with the Network Essentials Unity Asset from a different developer. It was also built on UNET. It came with scripts for controlling and spawning VR player avatars and had their custom Network Manager. However, nearly all of the assets included in Network Essentials was discarded as it was no longer meeting our needs and we made our own scripts, which gave us more control of what we wanted to accomplish.

We also considered using a third party Unity Networking asset, Photon. Photon is an fully-featured asset designed to manage all of the networking on the Photon Cloud servers rather than the Unity Master Servers. Photon also uses dedicated servers for creating rooms for players to join and manages the networking for all clients unlike UNET where it uses server-client connections and the hosting VR player manages the networking and authority. Photon also gives developers the ability to host their projects on their own servers or create a server backend for Photon. We decided to go with UNET since it was already built into Unity and the Network Essentials asset used UNET.

Decision Matrix Saving

At first we used the Unity Prefab functionality to save and load decision matrices. Prefabs are structures that Unity uses to simplify object creation by allowing a developer to save a configured object and make copies of it whenever needed. This made saving and loading decision matrices easy, we could simply create a prefab using the configured matrix. However, following implementation and within integration testing. We discovered that Unity does not support creation of prefab objects when the project is launched from an executable file instead of the Unity Editor. Since launching from an executable is mandatory for ease of use by end-users, we had to transition to manually record the contents of a matrix to an XML file. This required development of a saving and loading system to make the behavior feel identical to using Unity prefabs.

Appendix III: Other Considerations

Bandwidth Limitations

Bandwidth is defined as the rate of data transfer in seconds. In the networking world, you want to keep the bandwidth as small as possible to prevent overloading the servers. However, in VR, bandwidth usage is greater than traditional video games. In VR, the servers have to keep track of the avatar heads and hands unlike in traditional video games where only the full body avatar is track. In other words, more objects in VR are synchronized over the network than in traditional video games, meaning more bandwidth is used. Later in development, we discovered that the Unity Master Servers have a bandwidth limit of 4 kb/s, which is very small for us especially in the development stage. When the bandwidth limit is exceed, the offending VR player gets kicked out of the VRE, which shortens the interaction time with other VR players and objects. To prevent these disconnections, we have to optimize our networking solution to stay under the bandwidth limit.

C6

Something of relevance to our project, was that it already exists in some ways for use in the C6, the Virtual Reality Applications Center's six sided virtual reality room. It has however, a few flaws that the Vive addresses well.

It is difficult for more than one person to be in the C6 at any given time, while possible, it is only able to render a scene from the perspective of one person, it will seem somewhat blurry for all other people in there. The main goal of our project was to make it so more than one person can be in a virtual reality environment at any given time. While this means multiple HMDs will be needed, it is ultimately a better experience for those involved.

Another thing the Vive addresses well, is portability. The C6, while incredible is completely stationary. The Vive, on the other hand, while a computer that supports VR is needed, is relative to the C6, very portable.

The last main thing is cost of operation. The average cost to run the C6 for an hour is roughly $1000. Of course this is not always the case, this takes into consideration the necessity to replace parts, which can be quite expensive. The Vive costs only what is needed to power it, aside from the initial cost of purchase plus a VR ready computer. While somewhat expensive to acquire, over time, it easily costs less than the C6.

All of this is not to say that the C6 is completely obsolete, just that with the arrival of the Vive and its competitors has made virtual reality much more accessible and easy to use than the likes of the C6.