Relationships and Multi-User Interfaces via BabylonJS and Matrix during COVID-19

Angel Sylvester and Allison Miller
University of Minnesota - Twin Cities

Abstract

With the advent of COVID-19, people have grown more isolated as social gatherings amongst loved ones and strangers have been curtailed. However, with the accessibility of VR technology, perhaps those interactions can be simulated to a more authentic degree, which offer a tantalizing alternative to interacting through existing mediums such as social media. This project proposes the usage of Matrix in conjunction with the Baylon.js rendering engine and WebXR API to introduce a virtual alternative to in-person gatherings, without the loss of the emotional gratification that is gained through such meaningful encounters. Information gathered from research on similar multi-user interfaces as well as consideration of the key factors in creating new relationships and sustaining existing ones will be used to create a more optimal environment for social gatherings.

Keywords: Multi-User Interfaces, Virtual Reality, Relationships, COVID-19

Index Terms:

1 Introduction

Multi-user applications in virtual reality can create a sense of shared space between the users, enabling them to connect with each other even if they are in different areas of the world. Other forms of communication can also achieve this goal; the use of text messages and video calls makes it easier than ever to keep in touch with family, friends, and acquaintances. However, virtual reality can provide something that most existing applications lack, which is a sense of presence. Multi-user virtual reality enables participants to interact with each other and the world almost like they are in the same room; they can talk to each other, play games with each other, and do any variety of other activities that they would normally do in person - all while connected remotely. The primary frameworks that appear promising for this purpose include Babylon.js and Matrix.

Babylon.js is a web-based 3D rendering engine that includes support for WebXR, which is a Javascript API that supports connection with virtual reality devices. Using these tools, developers are able to create virtual reality experiences that run through a web browser, which allows for more flexibility in development and in sharing the experiences with others. This flexibility, depending on the application, provides an advantage over traditional game engines such as Unity; instead of having to completely deploy the application each time a developer wants to test their application on the headset, they can simply open a web page, assuming their computer is set up properly. Using Babylon.js, the goal of this project was to explore the potential of multi-user interfaces within the sphere of virtual reality, especially in the context of creating inviting environments for interpersonal relationships in the time of COVID-19. We implemented this goal by creating a basic test bed with basic object manipulation that can be synchronized across all devices in the same environment.

2 Problem Description

Despite the relative flexibility of the existing capabilities of Babylon.js and WebXR, there are some limitations that exist. One such limitation is the lack of a multi-user virtual reality framework within the WebXR API. As multi-user experiences are an important application of virtual reality, the lack of such a framework is a fairly severe limitation for WebXR and for Babylon.js in general. To address this issue, we integrated the Matrix framework into a 3D interface created with the Babylon.js rendering engine.

Matrix[18] is an open-source project that facilitates secure, decentralized, and real-time communication. Through Matrix, the user is connected to a single home server and all home servers can communicate with each other in a chatroom-like arrangement. Our goal was to use the Matrix standard to enable a multi-user virtual reality experience using Babylon.js and WebXR. In general, this would introduce opportunities for users to interact with each other - enhancing the immersion and opportunities for relationship growth through activities that involve more than one user.

2.1 Matrix Interface Framework

The Matrix framework allows for decentralized real-time communication through user-created chat rooms. Messages sent in one of these rooms are visible to any user who has joined the room. Using this functionality, we were able to implement virtual environment synchronization between multiple users. Users first need to register for an account with Matrix; afterwards, they are able to enter the environment, sign in with their username and password to join the room we set up specifically for the shared environment, synchronize the initial environment state to the state it is in for the other users to prevent differences in the environment based on when users joined, and begin participating in the shared experience.

2.2 Multi-User Interfaces in Context

Multi-user interfaces are useful in many different contexts. One of the most important of these is connecting people who don't have the ability to come together in person; for example, connecting a father with his family after he moves away for a job, or helping friends and acquaintances who no longer live in the same area

keep in touch. Multi-user interfaces are also useful for closer connections, such as having a quick chat with group members about a project, or meeting up with a friend in a game for an hour.

The COVID-19 pandemic has isolated a great number of people who are used to regular in-person contact with others. Mandatory quarantine has forced most activities that were formerly in-person, such as class meetings for schools and general work-related activities for those with jobs, to be conducted remotely from individual homes using typical remote multi-user interfaces such as Zoom or Google Hangouts. While interfaces such as these do serve their intended purpose, they aren't ideal for facilitating personal connections; much of a conversation is expressed in body language, which is challenging to transmit over a phone call, even if participants are able to see each others' faces. A multi-user virtual reality interface could potentially address some of these shortcomings by enabling participants to interact in the same (simulated) environment despite physical separation.

3 Related Works

3.1 The Loss of Interpersonal Relationships

As the workplace and social place go virtual, the natural interpersonal dynamics of real life interaction tend to disappear. With video conferencing calls or e-mail correspondence being the primary form of communication, the level of bonding exhibited declines [15]. Further, given that the majority of communication comes from body language [9], being able to accurately depict those motions is crucial for the development of a realistic interpersonal experience with others. So based on this assumption, more implementation work in terms of body movement alignment will be a large focus for the culmination of this multi-user experience.

3.2 Prior Multi-User Interface Implementations

3.2.1 Exploration of Existing Frameworks

There exist many implementations of large scale multi-user virtual environments. The existing real-life applications of such frameworks include activities that are intended to optimize learning and interactivity through such as in an online laboratory [11], immersive language classroom experience [4], and creating narrative-based multi-user story telling environments [8]. Framework implementation methods include scalable architectures that have a server based hierarchy [2, 12, 13], distributed architectures in MMVE (Massively Multi-User Environment) networks that utilize multidimensional index querying [4], zone selection and distributed filtering architectures [5], as well as other publicly accessible tool kits/frameworks that permit ease of multi-user VR experiences such as Wonderland [12] and EasyChair [13]. The myriad of different methodologies employed in the realm of multi-user interfaces in virtual reality will offer a variety of different considerations that previous researchers also had to address, such as ensuring a seamless communication between users with appropriate environmental responses, without fear of inappropriate lag (due to improper treatment of multiple users) or excessive memory use due to hyperactive updating.

Other frameworks that have been addressed in the literature for specifically online multiplayer game synchronization include SimMud [7] , Gameservers [17] , MOPAR [3] , OPeN [14], and Colyseus [1], which are DHT (Distributed Hash Table) based architectures. Other implementations include a hybrid of different methods like Thorsten [16] and Mediator [10]. The primary focus of these frameworks includes ensuring that the performance of the game does not deteriorate as more users join the interface through the usage of decentralized game servers for enhanced flexibility. Through the perusal of these instances of multi-user environment execution, the general consensus is that the primary problem that needs to be addressed is in the realm of scalability and minimizing latency. In order to make an implementation successful, it is important to ensure that updates are made accordingly to users within a specific range, while not impeding on the quality of the virtual reality experience.

4 Project Overview

Our solution used the Matrix standard (specifically, the matrix-js-sdk, as Babylon.js runs on Typescript and Javascript) to synchronize the actions of multiple users in a virtual environment created through Babyon.js. In Matrix, users can join a room hosted on a server; any messages sent in this room are synchronized to all of the other servers participating in the room (which equates to all of the users, since each user is connected to a single server, which then participates in the room). Instead of using these messages for user communication with natural language, we used them to communicate changes in the environment state to the other users in the room; that is, every time a user changes some aspect of the virtual environment on their end (i.e. picking up an object or some similar action), a message is sent to all the other connected users describing the new state of the environment (i.e. that an object was just picked up by another user), and the environment is updated accordingly. To simulate the social aspect and to convey as much body language as possible, we implemented simple user avatars (including a head, hands, and a body) that will convey the location, orientation, and actions of users, updating as the user performs distinct actions within the environment. The environment consists of a simple testbed that allows users to move around and see each other as well as to create, manipulate, and remove objects.

4.1 Integrating Matrix into Babylon.js

4.1.1 An Overview of BabylonJS and Testbed Scene

Babylon's WebXR API enables users to initialize a session, a camera, general scene interactions, and other technical aspects helpful for creating virtual worlds in an asynchronous fashion.To initially create a scene, the program creates an engine element, which is responsible for handling calls for lower level APIs such as WebGL and audio. The engine class has a runRenderLoop function that constantly updates the scene each frame depending on the state of the environment. Additionally, scene and canvas elements (among others) are created in the constructor of the class so that the initial arrangement of environment elements are in place and are available for further updates as the engine continues to update the scene.

The basic functionalities successfully implemented in the virtual environment include simple teleportation for locomotion, randomized object creation, object texture customization, object removal, and manipulation of an object's placement in the environment through ray casting. To avoid conflicts, the objects in the environment can only be altered by one individual at a time, and the scene highlights meshes other users are currently working with in the color of that user's avatar (as all users are automatically assigned a random avatar color when they enter the environment). The test bed environment was left intentionally rudimentary to limit the possibility of decreased performance from imported meshes/environments. However, in future iterations, basic scenes such as a conference, golf course, or classroom/laboratory would be another functionality that would enhance the applicability of this project to existing social environments.

4.1.2 Synchronizing and Optimizing Environment Behaviors

The use of a chat room to pass environment status updates between users requires those messages to be in a text-based format. As such, any information communicated about the state of the environment must be reduced to a text format, and the information has to be structured in such a way that the relevant information can be quickly and easily parsed out and applied to the objects in the environment. The information also needs some sort of identifier associated with it so the program applies the update to the correct object on the connected clients. Finally, the information may need to be structured differently depending on the subject of the update and the type of change the subject went through; for example, a message notifying users that an object was removed requires significantly less information that notifying users that a mesh was created, as a mesh creation update requires all the information necessary to recreate that mesh in a user's local version of the environment, and a user position update requires different information than an object position update, as the user has three nodes with distinct positions (the head and both hands), whereas the object only has its own single position.

Since objects and users are updated frequently in a shared environment, the program required some efficient structure for storing and retrieving shared objects and users. We used a pair of maps, one for objects and one for users, mapping each object or user to a unique identifier. This identifier is the same one used to specify the subject of an update in our update messages; the map structures thus enabled us to find the correct object by looking its identifier up in the proper map.

Our initial concept for the message structure was to simply communicate any required information as a text string with identifiable delimiters, such as commas or colons, and to parse the required information using several iterations of text processing. This concept never proceeded past the planning stage, however, as we quickly realized that with the amount of information required and the differences between message types processing that text string into usable update information would be a lengthy and error-prone operation. After looking through a variety of resources on the subject, we discovered that Javascript (and, by extension, Typescript) includes built-in support for JSON objects, including stringification and parsing, which would enable us to declare fields within the object for easier recognition on the message's receiving end, simplifying both the message creation and parsing processes. The JSON structure we used included the name of the object (used as the identifier mentioned earlier to ensure the program updates the correct object), the type of the update (to help the parser determine what information might be included in the message), and the update information itself, including position, rotation, and other necessary data.

```
message = {
    id: id,
    type: type,
    mesh: serializeNew ? ("data:" + JSON.stringify(SceneSerializer.SerializeMesh(this.envObjects.get(id)!, false, false))) : ""
    info: serializeNew ? {} : {
        position: this.movementArray,
        rotation: this.rotationArray,
        scaling: this.selectedObject!.scaling.clone(),
        selected: this.selectedObject!.parent ? true : false,
        color: this.userColor
    },
```

Figure 1. Example of JSON-formatted object update message structure

Deciding what value to use for the object identifier presented its own challenges. Since objects are synchronized across multiple different users, corresponding objects in different users' environments need to have the same identifier; additionally, this identifier needs to be unique among all the other objects to ensure the correct object is updated when requested. One idea we attempted to use was the object's unique ID property, as it seemed like an obvious choice due to the property's name; however, we discovered while testing our implementation that, as might be expected, the unique ID is only unique to the user that first created the mesh, which resulted in conflicts when specifying the object to update.

Additionally, since the mesh itself cannot be transmitted through the chatroom, it must be individually recreated in each of the users' local environment copies, and recreating a mesh creates a separate unique ID for that mesh. Even with this mismatch, item storage isn't an issue, as new items would be properly stored in the map with the communicated identifier, and further updates on that object would function as expected; but since message creation uses the identifier of the mesh directly from the object instead of from the map, if that same mesh was updated in any user's environment other than that of the user that initially created it, the update would not be properly applied in the connected environments, as they would be unable to find the correct object.

Our realization of why the unique IDs were not working also led to the solution for the issue; since unique IDs are unique for each user, combining the unique ID of the mesh with the username of the user would result in a unique identifier for each object, and using this as the name of the object (and using the object's name as the unique identifier) eventually led to proper object updating.

This approach does have one key limitation, namely the issue of multiple users with the same username; however, solving this issue would require us to ensure no two users had the same username (as it is also an issue for user identification) and as such is out of scope for our project. (Ensuring two users do not have the same username would likely fall to Matrix itself to check; by the time the program has a chance to realize a duplicate exists, the offending user will have already logged in and joined the room. The easiest solution may be to simply assign the duplicate user a different username for their duration of their time in the environment.)

The final version of our message structure allows for four different types of messages: user updates, removal of users and items, item updates, and full-environment synchronization. User updates are relatively simple; user representations are handled individually on each client's end (as a user does not need a separate representation of themself), and as such the only information communicated is the position information for the controllers and the headset, along with the rotation of the headset (as it is represented as a box) and the user's assigned color. User updates are sent when the user first joins the environment and when they perform an action in the environment - if a client attempts to find the user in their map of users and cannot, the program assumes the user has newly joined, and adds them to the map, thus allowing us to use a single message type for both creation and general updates. Removal messages are even simpler, as they only include the identifier of the entity being removed; each type of object has its own disposal method, and no other information is required.

The two message types that presented the most challenges were the object updates and the overall initial environment sync. Like user updates, object updates include the creation case, when the object does not yet exist in the local environment; this case is handled easily in user updates, as the representation is static (i.e. the head is always represented as a box and the hands always with spheres), but with object updates, the representation has to match the original object, which is challenging to convey in a text format. Our original plan was to limit the objects available in the environment to meshes that could be created by using MeshBuilder, as we would be able to include the type of mesh to create and the options for the CreateX method easily in the message. This plan was successful, as the meshes were created in the remote environments properly; however, as might be expected, it was extremely limited in terms of what users could do with the objects and what kinds of objects they could create.

While researching JSON objects for the overall message structure, we discovered that it might be possible to convert the entire mesh into a JSON-formatted string for later parsing; however, performing this conversion directly is impossible due to circular references. Upon further research, we found documentation for Babylon's SceneSerializer class, which includes a method to serialize a single mesh into a JSON-formatted string without the circular reference issues that arise when attempting to directly stringify the mesh. Attempting to parse this string and assign it directly to a mesh object as the message parser does with the position vectors was unsuccessful, but further investigation discovered the corresponding SceneLoader class, which has an ImportMesh method that can be used to either import an external mesh or import a serialized mesh from a JSON string. For the most part, the objects do not need every scrap of mesh information for each update, and the update can include only basic information with which to update the mesh; however, if mesh information outside of the most basic information is updated, the mesh will need to be re-serialized, deleted, and recreated in the client environments to update properly. Serializing the mesh and importing it in the remote environments to create those local copies allowed us to handle any type of mesh instead of limiting our environment to a select set of meshes. In our testbed, this is demonstrated by the creation of a random polyhedron when the right grip button is pressed; the serialized mesh contains all the information required to faithfully recreate the exact mesh in connected environments, without the need to specify mesh types or parameters.

The environment synchronization issue combined the issue of object communication with a variety of other considerations. When users initially join the shared environment, their local copy is completely empty. To enable a viable shared environment that users can enter and leave at any time, we needed to find a way to communicate the existing state of the entire environment to the new user (as opposed to the state of a single identifiable object). We came up with two possible solutions to this challenge: having the newly joined user read through all previous update messages from the start of the shared session until their entry to manually recreate the environment step by step, and having one of the existing users consolidate a snapshot of the environment into a single message that the new user would then use to create their copy. Both approaches had challenges; with the first solution, the major challenge was figuring out how to either clear the message history after the last user left a room or delineate the start of the current session so the user would know where to start reading previous messages, and for the second solution, the major challenge was figuring out which user would end up sending that sync message to avoid syncing the new user multiple times. We decided the second solution would be the more efficient option, as there is no need to start from the beginning and go through the entire update process when the new user could simply have the entire environment available immediately. The first user to join the environment is designated as the 'admin' for that session; to figure out if a new user is the first user in the environment, the new user first alerts any other possible users that they have joined the environment (a user update message). If the user receives a 'sync' message in response, they know they are not the first user, and sync the environment appropriately; if the user never receives this update, they assume they are the first user (and thus the 'admin' responsible for environment synchronization) and update new users appropriately. This 'assume until proven otherwise' method seems like it would be susceptible to network errors, such as dropped or erroneous messages; however, no malfunctions

were detected at any point throughout project implementation. Future work would likely include figuring out a more robust method of performing this initial synchronization and ensuring that it only happens once for any given user.

After figuring out the mesh serialization method of communicating mesh status, the synchronization message structure itself was quite straightforward: the user already keeps track of everything in the environment separately from its other internal elements, and creating the message was simply a matter of including an array of properly formatted update messages instead of a single object's data. User synchronization was handled by the individual users (when each user receives the new user notification, they send their own user update message to add themselves to the new user's user list).

The message structure described above is sufficient for communicating the environment state and state updates between any amount of connected users; however, another consideration, and an extremely significant roadblock, is how often entities in the environment send updates to the other clients. Originally, we intended on updating the connected users every time an entity's information changed in real time, as this type of exact movement would result in smooth object movement and lifelike user behavior. We were prepared for the possibility of lag with the sheer volume of messages that would need to be processed with this method; however, we ended up running into a different issue altogether. The network rate limit of the chatroom (i.e. the maximum number of network requests that can be sent from a single device in a certain time span) was far stricter than we thought it would be, and any attempt to update environment entities in real time would immediately result in the network throttling the rate of the update messages until one was being sent every second or so - obviously not ideal for real-time movement and synchronization. The issue was compounded by the fact that Matrix's interface, since it is in the form of a chatroom, has quite a bit of overhead associated with each message (in the form of status updates and presence updates and such). This overhead added to the total network requests the system was attempting to send, resulting in four or five extra network requests for each update message, which only made the network rate limit seem that much lower. As there is no way to stop a message from being sent after it is sent, there was no way to stop an object from receiving update messages when the update volume exceeded the rate limit, and the environment would slow down so far as to be unusable, and the user would have to leave the environment and log back in if they still wanted to be in that environment. We attempted to implement a number of different possible solutions to the issue, and those possibilities are described here.

One possible solution we tried using to limit the update rate was only sending update messages once the object had traveled farther than a certain distance away from the position at which it had previously sent an update message. This solution was partially successful, as it cut down on the volume of update messages by a significant amount, especially if the user held the object in place for a period of time (small variations in position when keeping the trigger button pressed originally led to sending a large number of insignificant updates); however, we still ran into the same problem. A small threshold value would capture the motion more accurately but encounter the network rate limit faster, and a larger threshold value would hit the network rate limit later in the movement while also not capturing the details of the movement.

The second solution we decided on was simply updating the object's position at the start (when the user initially selects the object, to update that object as having already been selected by a user in the connected environments) and end points of movement. This managed to avoid hitting the rate limit almost completely at the expense of losing all movement details; depending on how many objects a user moved in a certain time frame, the rate limit would still occasionally come into play. Despite the visual limitations, we decided this solution was preferable to the previous one as it mostly avoids hitting the rate limit. If a user hits the rate limit, their update messages are sent extremely slowly, which results in that user's actions lagging in other environments by an unacceptable amount for what is supposed to be a real-time shared virtual environment.

After unsuccessfully attempting to figure out how to silence the other messages that are sent with each of our message events, we decided that we could communicate the details of object movement by storing the movement as an array of positions (recorded every 20 frames if the object moved further than a threshold value to avoid recording insignificant movements), including it in the message, and animating the object in the connected environments through those positions. This solution had the same message efficiency as the second solution (i.e. it only sends updates at the start and end points of movement) while also retaining the movement details and creating a continuous movement in the other users' environments. The process has a couple relatively minor limitations; beginning the animation after the object has already completed its movement results in a fair amount of delay between the sending user and the receiving users, and on occasion the rotation portion of the animation rotates the object in strange ways, but the solution avoids the rate limit issues while also providing a sense of continuous motion to the connected users. There is a bug that does occur when attempting to animate meshes that, due to their update content, need to be re-imported, such as when the texture is changed; as the animation was added later, objects that require a full recreation are not animated and simply jump to their final position. All attempts to rectify the issue resulted in either the same jumping behavior or no movement at all, and due to time constraints we decided to leave it as it is.

Another issue that we encountered while testing the environment was related to the teleportation operation when updating the placement of the user controllers. Unlike selection, for which we used Babylon's default pointer selection, we manually implemented teleportation to increase compatibility with the rest of our code. Users teleport by pushing the right thumbstick

forward and pointing to select a location, as usual, and users can adjust the direction they will face after completing the teleport by rotating the left controller. A green arrow on the ground indicates the proposed end position of the teleport, and the direction the arrow points indicates the proposed end rotation of the teleport. With both the default and custom teleportation, however, the controllers would not get updated in the same frame as the headset, resulting in a rather humorous situation in which it appeared as though users were leaving their hands behind with each teleport. The solution to the issue ended up being relatively simple; instead of sending the user update message precisely upon teleportation, the program waits to send the message until the controllers properly update their position after the camera object is moved in the environment.

4.1.3 Integrating Spatiality into Implemented Techniques

Initially, the user's avatars were simple geometric shapes to represent the body and controllers for testing purposes. Once the basic synchronization framework was finalized, more work was done to create a more realistic rendering of the users during different phases of motion. The user's avatar properties included a cube head, body, and arms as lines connecting the body to the right and left controllers. No inverse kinematics were used to realistically depict these positions, but the basic avatar components served as standard elements that could be rendered with more sophisticated meshes in future iterations of this project. The final environment is depicted in Figure 2. Depending on the angle from which the user is viewed, the lines representing the arms might not be visible (as is evident from Figure 2); from previous experience with Babylon, we believe this is a rendering issue with the Babylon lines meshes, and were unsure how to resolve the issue within the available time constraints.

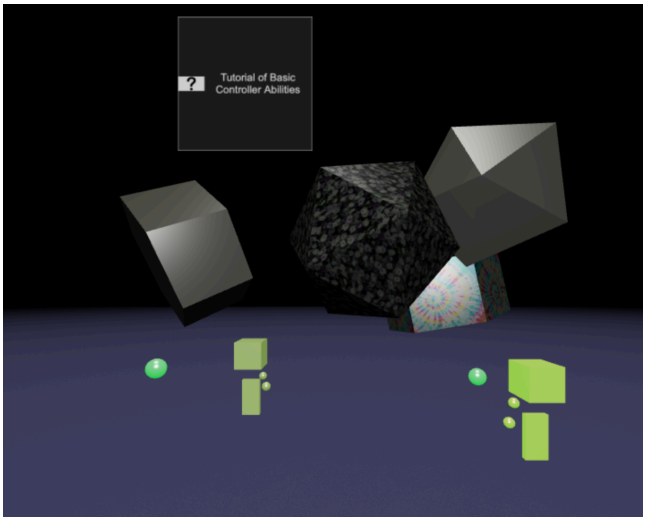

Figure 2. Example of the virtual users simultaneously attempting to interact with the environment

We added a couple more features within the environment itself to make the experience as a whole more interesting and interactive. The main interaction technique within the environment is ray casting to select and move objects around; however, upon selecting an object, two other features become available. Instead of using a typical graphical menu to display these options, we decided to use a spatial interface: the additional interaction options are represented as small control meshes located around the wrist of the hand the user is using to manipulate the object. Users may select one of these two options by dragging the appropriate control mesh off of their wrist with their opposite hand and releasing it in front of them. Users may select the red sphere to destroy the currently selected object, and they may select the blue cube to change the texture of the selected object. The menu for texture choice is also spatial; when the user releases the blue cube, a set of textured cubes appear around the release location, and the user selects the texture they want by moving their controller into the cube representing the desired texture. Five textures are currently included, but adding more would simply be a matter of adding an additional texture into the texture array, since the code is set up to work with the length of the array. A visualization of this control interface is depicted in figure 3 and a perspective view of that same user from a different perspective is shown in figure 4. This exploration of a spatially designed control interface to customize objects in an environment offers potential routes in application based projects. Depending on the context in which the framework is used in the future, programmers could adapt tools that a user might need to use in a similar fashion.

We had originally planned on including a third option to allow the user to directly change the color of the mesh, but our attempts to include this feature using Babylon's ColorPicker GUI object were unsuccessful due to the limitations of pointer selection; only one pointer can be triggered at a time, and using the other controller to select a value for the color deselected the object.

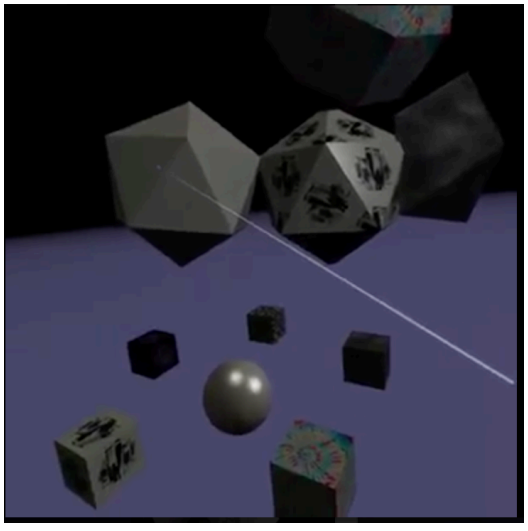

Figure 3. Example of texture interface with selected object.

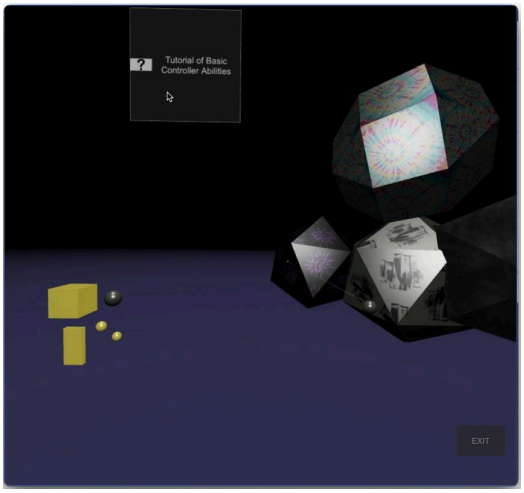

Figure 4. A different perspective of the user altering the texture of a selected object.

4.2 Evaluation of Interface and Functionality

Ultimately, we were able to successfully integrate the Matrix client-server architecture with Babylon.js to enable a shared multi-user environment. The majority of our time was spent optimizing the communication of object motion to avoid hitting the network rate limit. For the most part, the depiction of the object's motion is faithful to the rate at which the object is moving in real time, with the exception of times when the user is holding the object in the same location, as those small movements are not recorded for the animation. Because the users were not directly interacting with each other (through hand gestures or direct touching), the controllers and avatars were not updated as frequently as the manipulated objects. However, it is possible in the future to use the same framework for user controller movement when the user is hoping to engage in actions that require continuous observation of the movement. Another method could include integrating haptic sensing so that users are aware of the fact that a user is attempting to interact with their respective avatar. The distribution of such messages would follow the same format where upon intersection with some user, only that other user is able to receive some vibration indicating their presence.

A key component of the development of relationships that appear to be absent in existing modes of virtual communication includes the lack of visible body language and sensation of real human presence. In order to specifically address this deficiency, the usage of avatars is a promising start that offers users the appearance of another user's presence. Embedding some implementations of inverse kinematics to better depict the user's precise position as well as incorporating the ability of the user to send certain gestures (ie waving, pointing) could also offer a more visually genuine interaction; however, the limits of the existing network architecture place severe restrictions on the real-time inter-user interaction possibilities.

Additionally, the UI for this environment is fairly intuitive; when the user first loads the program, they are presented with a fairly typical login screen, though it does not specifically mention that it requires the user's Matrix account information. If the user does not correctly enter their login information, the interface displays an error message, and the textboxes where the user entered their information clear so that they may re-attempt the login. When the user successfully logs in, instructions for how to manipulate the environment appear, and the user can dismiss these instructions by selecting the button in the top left corner. The existing login structure is fairly straightforward for development purposes, but future potential UI enhancements could include an option to register new users (to avoid forcing users to register on the web client before joining the virtual environment) or an option to select a specific test bed environment to address different use cases for the multi-user environment.

4.3 Future Work

One major aspect of future work on this project would include generalizing the Matrix integration code into its own library for use in other projects. At the moment, the code and message structure are fairly closely tied into our specific project, and it would need to be generalized fairly extensively to enable other people to use it for their own projects. For instance, the connection made between the Matrix chatroom, revamped teleportation method to handle movement with multiple users in the same environment, and the synchronization of selected objects would not require future users to understand the details of the implementation, but would still be capable of abstracting the functionality necessary to implement these features in the environment.

Another aspect important for the facilitation of human connection is incorporating some form of communication into the environment by enabling text chat, voice chat, or both. Text communication would likely be the easiest to implement - the user is already connected to a chatroom for the environment synchronization, so sending messages and receiving text-based messages would be simple (though the interface to view and respond to these message would be less so, as visibility and usability requirements become more important - text input in virtual reality is generally clumsy and error prone, especially without hand tracking, so ideally future work would also incorporate the creation of a more efficient method of text input). The Matrix framework may include some functionality to enable voice communication; however, we would have to investigate it further, as audio and voice communication was outside the proposed scope of our project.

Finally, there is currently no acknowledgement of how the objects in the environment would behave and provide feedback to the user in the real world, such as physics interactions or haptic feedback. A far reaching goal would include being able to create mesh objects that react to a user's touch to enhance the interactivity and immersion of the environment. Users could interact with each

other more realistically if objects were more like real-world objects and not simply static meshes.

4.4 Conclusion

Through the exploration of previous implementations of multi-user interfaces in online games and application-based projects in the literature, the primary problem that this project attempted to address was the area of scalability and latency. In order to accommodate the massive amounts of updates necessary to depict movement in a multi-user environment, the update rate was decreased to only update the connected environments in response to discrete actions and events. The updates occur sequentially so with additional users, lag will be a potential problem that would need to be addressed. Further, the project explores the usage of a generalized control system to allow for enhanced customizability for users who seek to adapt this to certain situations (ie. in a classroom or conference setting). Future work includes expanding on these functionalities so that they are more context specific and more adaptable to existing frameworks by incorporating the work into a formalized library.

References

[1] A. Bharambe, J. Pang and S. Seshan, “Colyseus: A Distributed Architecture for Multiplayer Games,” in Proc. ACM/USENIX NSDI, May 2006.

[2] A. M. Burlamaqui, M. A. M. S. Oliveira, L. M. G. Goncalves, G. Lemos and J. C. De Oliveira, "A Scalable Hierarchical Architecture for Large Scale Multi-User Virtual Environments," 2006 IEEE Symposium on Virtual Environments, Human-Computer Interfaces and Measurement Systems, La Coruna, 2006, pp. 114-119, doi: 10.1109/VECIMS.2006.250803.

[3] A. Yu and S. T. Vuong, “MOPAR: a Mobile Peer-to-Peer Overlay Architecture for Interest Management of Massively Multiplayer Online Games,” Proc. NOSSDAV, Jun. 2005, pp. 99-104.

[4] B. Hariri, S. Ratti, S. Shirmohammadi and M. R. Pakravan, "A distributed latency-aware architecture for massively multi-user virtual environments," 2008 IEEE International Workshop on Haptic Audio visual Environments and Games, Ottawa, Ont., 2008, pp. 53-58, doi: 10.1109/HAVE.2008.4685298.

[5] B. Hariri, S. Shirmohammadi and M. R. Pakravan, "A distributed interest management scheme for massively multi-user virtual environments," 2008 IEEE Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, Istanbul, 2008, pp. 111-115, doi: 10.1109/VECIMS.2008.4592763.

[6] B. Hariri, S. Shirmohammadi and M. R. Pakravan, "A distributed interest management scheme for massively multi-user virtual environments," 2008 IEEE Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, Istanbul, 2008, pp. 111-115, doi: 10.1109/VECIMS.2008.4592763.

[7] B. Knutsson, H. Lu, W. Xu, B. Hopkins, “Peer-to-Peer Support for Massively Multiplayer Games,” In INFOCOM,Mar, 2004.

[8] C. Brown, G. Bhutra, M. Suhail, Q. Xu and E. D. Ragan, "Coordinating attention and cooperation in multi-user virtual reality narratives," 2017 IEEE Virtual Reality (VR), Los Angeles, CA, 2017, pp. 377-378, doi: 10.1109/VR.2017.7892334.

[9] Exelbert, R. (2020, May 6). How Virtual Platforms Impact Our Experience of Social Relationships. Psych Central. https://psychcentral.com/lib/how-virtual-platforms-impact-our-experience-of-social-relationships/.

[10] L. Fan, H. Taylor and P. Trinder, “Mediator: A Design Framework for P2P MMOGs,” in Proc. Netgames 2007, Melbourne, Australia, Sept. 2007, pp. 43-48

[11] L. Rodriguez-Gil, J. García-Zubia and P. Orduña, "An architecture for new models of online laboratories: Educative multi-user gamified hybrid laboratories based on virtual environments," 2016 13th International Conference on Remote Engineering and Virtual Instrumentation (REV), Madrid, 2016, pp. 202-203, doi: 10.1109/REV.2016.7444465.

[12] M. B. Ibanez, J. J. García, S. Galán, D. Maroto, D. Morillo and C. D. Kloos, "Multi-User 3D Virtual Environment for Spanish Learning: A Wonderland Experience," 2010 10th IEEE International Conference on Advanced Learning Technologies, Sousse, 2010, pp. 455-457, doi: 10.1109/ICALT.2010.132.

[13] Novotny, A., Gudmundsson, R., & Harris, F. (2020). A Unity Framework for Multi-User VR Experiences. https://doi.org/10.29007/r1q2

[14] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera, “Enabling Massively Multi-Player Online Gaming Applications on a P2P Architecture,” in Proc. IEEE Intl. Conf. Information and Automation, Dec. 2005, pp. 7-12.

[15] Sherman, L. E., Michikyan, M., & Greenfield, P. M. (2013, July 1). The effects of text, audio, video, and in-person communication on bonding between friends. Cyberpsychology. https://cyberpsychology.eu/article/view/4285/3330.

[16] T. Hampel, T. Bopp, and R. Hinn, “A Peer-to-Peer Architecture for Massive Multiplayer Online Games,” in Proc. Netgames, Oct. 2006.

[17] T. Limura, H. Hazeyama, Y. Kadobayashi, “Zoned Federation of Game Servers: a Peer-to-Peer Approach to Scalable Multi-player Online Games,” In Proc. SIGCOMM’04.

[18] “Discover Matrix ,” Matrix.org, 24-Dec-2020. [Online]. Available: https://matrix.org/discover/. [Accessed: 24-Dec-2020].