EPiC Series in Computing
Volume 88, 2022, Pages 61–70
Proceedings of 31st International Conference on Software Engineering and Data Engineering

MuVR: A Multiuser Virtual Reality Framework for Unity

Joshua Dahl, Erik Marsh, Christopher Lewis, and Frederick C. Harris, Jr.

Department of Computer Science and Engineering
University of Nevada, Reno
joshuadahl@nevada.unr.edu, erik.i.marsh@gmail.com,
christopher_le1@nevada.unr.edu, fred.harris@cse.unr.edu

Abstract

Due to the rapidly evolving nature of the Virtual Reality field, many frameworks for multiuser interaction have become outdated, with few (if any) designed to support mixed virtual and non-virtual interactions. We have developed a framework that lays an extensible and forward-looking foundation for mixed interactions based upon a novel method of ensuring that inputs, visuals, and networking can all communicate without needing to understand the others' internals. We tested this framework in the development of several applications and proved that it can easily be adapted to support application requirements it was not originally designed for.

1 Introduction

Currently, there are very few multiuser Virtual Reality (VR) frameworks available in the literature. Likewise, much of the formalized work on interactions between multiple VR and non-VR users remains in its infancy. Novotny et al. [17] is a notable exception to the first issue, providing a framework for multiuser VR development. However, the VR field is rapidly evolving and many previous works have become obsolete as newer standards and methodologies emerge. MuVR [11] serves as a total overhaul of the framework designed by Novotny et al. with major emphasis put on supporting the OpenXR [4] standard and adjusting several parts of the framework to be easily extendable, with an eye toward future support for non-VR users as well. Towards this aim, we have developed a novel method of ensuring that inputs, visuals, and networking can all communicate without needing to understand each other thus allowing each of the before-mentioned systems to be replaced without disrupting the others.

The rest of this paper is structured as follows: Section 2 briefly reviews some of the existing literature and then dives into library choices that we made, with particular emphasis placed upon comparisons to existing VR and networking frameworks for Unity [20]. Section 3 details the design and implementation decisions of the overhaul. Section 4 details two applications we implemented with MuVR, showing how easily the framework can be extended; and finally Section 5 wraps up the paper with conclusions and plans for future work.

2 Background

There are very few multiuser VR frameworks available in the literature. This fact exists in stark contrast to the fact that many simulations, educational experiences, and entertainment experiences are now being developed for VR [6, 14, 18]. Thus facilities to ease this development would be beneficial.

Unity is a commercial game engine with extendable scripting in the C# language. It utilizes the Object-Oriented Composition paradigm [13] that was common in game engine architecture from the last decade, where component classes implement various types of encapsulated behavior that can then be attached to container objects. This extensible model is interfaced with using a visual environment editor with the ability to visually change properties and create references to other objects in the environment thus using code to walk the engine's object hierarchy to find references is uncommon. Since MuVR is tightly coupled to the Unity ecosystem, migrating it to another game engine would prove to be nontrivial.

Most proprietary libraries for Unity are distributed as precompiled shared objects. Unity supports a wide variety of platforms, many of which are not supported by the shared objects provided by most proprietary libraries. Thus we have striven to avoid such resources as much as possible, instead utilizing free and open source alternatives that we can compile to any platform ourselves.

2.1 VR Frameworks

The main criteria used for choosing between the many available VR frameworks was the number of platforms they support. Additionally, we only considered frameworks that were open source or included with Unity. Historically, the two leading VR frameworks used with Unity were SteamVR and the Oculus SDK. The standard SteamVR implements, OpenVR, has been deprecated in favor of OpenXR [4], and the Oculus SDK only supports hardware created by Meta. While both platforms provide a range of useful utilities such as an extensive interactables library (providing buttons, levers, etc...) and positioning appropriate controllers to indicate the location of your hands, neither SteamVR nor the OculusSDK are ideal considering the current rapid proliferation of VR devices.

OpenXR is an open standard from the Khronos group (the same group that maintains the OpenGL and Vulkan standards) that acts as an abstraction layer between applications and a large selection of popular XR[1] devices. However, accessing functionality inherent to a single device or manufacturer using OpenXR is difficult.

Unity provides the XR Interaction Toolkit [3] (XRIT), a toolkit that provides a platform-agnostic framework for implementing interactions in VR, however, the framework does not provide as large of a selection of precreated interactables as its competitors. When combined OpenXR and XRIT provide a widely supported device-agnostic method of interacting with VR environments. Thus the combination of OpenXR and XRIT was chosen as the foundation for MuVR. Table 1 summarizes the differences between the discussed VR frameworks.

2.2 Networking Frameworks

While the choice of VR framework was fairly straightforward given our goals and SteamVR's recent deprecation, choosing a networking solution proved more difficult. Several Unity networking frameworks require that the code for clients and servers be written in different projects;

[1]eXtended Reality (XR) is an umbrella term used to describe both Virtual and Augmented Reality devices.

Table 1: VR Framework Comparison Summary

Framework	Supported Platforms	Interactables Library	Hand Presence	Deprecated
Open XR + XRIT	HTC, Valve, Oculus Microsoft, and Varjo	Basic*	Basic†	No
Steam VR	HTC and Valve	Yes	Yes	Yes
Oculus SDK	Oculus	Yes	Yes	No

* Includes support for basic grabbable objects that follow the user's hand.

† Includes support for representing the location of the user's hand, however it may not properly represent the type of controller the user is using.

all of these frameworks were rejected since providing a unified framework with a fragmented codebase produces extra complexity that we decided it would be best to avoid.

The first framework we considered was Photon's Fusion [2] framework. This was the framework that sparked the platform support requirement, since difficulties were encountered when using their provided DLLs. Additionally, Fusion's documentation is lacking, making development with the framework a frustrating experience. Alternatively, Fusion is one of the two considered frameworks (along with Novotny et al.) to provide a matchmaking system for players to discover each other without requiring an IP address. All of these factors paired with the existence of a price tag on these services led us to search for other options.

Fish-Networking [9], Mirror [21], and Unity's Netcode for GameObjects [1] (NCGO) were all considered next. All of these frameworks are open source and have similar designs, with minor differences in usability between the three, but nothing major enough to strongly influence a decision. Mirror supports a purely peer-to-peer-based architecture[2] while Fish-Networking and NCGO support a client-server architecture[2] where the server can either be dedicated or hosted on one of the clients.

After identifying the several candidate frameworks, a performance benchmark was performed; the results of which along with several other comparison details are explained in Table 2. The performance benchmarks were conducted utilizing the methodology outlined in Fish-Networking's documentation [8] except: the tick rate for every framework was set to 60 ticks per second, the server was run from within Unity's editor, and thirty separate client executables were launched, all on a single machine[3]. Thirty clients were chosen since more would result in GPU throttling. Bandwidth information was captured using Wireshark's [10] Protocol Hierarchy statistics, filtered to only scan relevant ports, that captured the number of bytes transferred which were then divided by the timestamp of the last packet scanned to find the average bandwidth. Since data was captured on a single machine, the bandwidth statistics represent both sent and received data. All data was captured over a period of five minutes.

Once the benchmarks had been performed, Fish-Networking proved to perform better than its competition, with similar framerates and significantly reduced bandwidth overhead; and

[2]In a peer-to-peer architecture like Mirror, data is sent from every connected user to every other connected user, without the aid of a central authority. Alternatively, in a client-server architecture, users send their data to a central server which is then responsible for either rejecting it or forwarding it to the other users.

[3]The machine used to run the benchmarks is custom built with an Intel i7-12700k, EVGA GeForce RTX 3090 with 24GB of dedicated RAM, 32GB 2133MHz Corsair RAM, and a Samsung 980 Pro NVME SSD, running Unity 2021.3.5f1 set to build executables with the IL2CPP backend. All of the code utilized for these benchmarks can be found in MuVR's Git repository spread across several branches whose names all start with "benchmark/".

Table 2: Networking library comparison over several performance metrics.

Framework	Est. Max CCU*	AVG FPS	Band-width	Cost	Supp. Arch.	Match-making	Voice
Fish-Networking	500+	60.19	0.94 MB/s	O.S.	Hosted/ Dedicated	No	No
Mirror	200+†	60.23	2.15 MB/s‡	O.S.	P2P	No	No
Netcode for Game-Objects	Not Published	59.97	3.42 MB/s‡	O.S	Hosted/ Dedicated	No	Yes§
Photon Fusion (Shared Topology)	2000	25.08	1.82 MB/s	Per User	Hosted/ Dedicated/ P2P	Yes	Yes¶

* CCU is an acronym for ConCurrent Users.

† Old stress test demos have shown Mirror supporting 480 concurrent users, however this has not been tested in practice.

‡ Due to how Mirror and Netcode for GameObjects calculate their tick rate, these numbers are not based on exactly 60 ticks per second (we found it was between 55 and 60) whereas the other frameworks are.

§ Provided by separate subscription priced Vivox package.

¶ Provided by separate subscription priced Photon Voice package.

thus reduced bandwidth utilization indicating that more information can be exchanged before network infrastructure becomes overloaded. The benchmark utilized appears to be designed to fairly showcase Fish-Networking's performance superiority with a minimal amount of bias; that being said, we acknowledge that there is an unlikely potential of biasing in the results that we missed. With all features considered, including several of Fish-Networking's nicer usability features, Fish-Networking became the clear choice to base MuVR upon.

3 Design & Implementation

3.1 Ownership

In most applications, when networking provides a latency spike, it is an annoying irritant that is usually ignored, however, in VR even a minor latency spike is substantially more noticeable. The increased immersion makes many people more sensitive to issues with the simulation, and a momentary lack of movement due to a latency spike increases Transport Delay as described by Stoner et al. [19], which could easily invoke a bout of simulator sickness. To account for this discrepancy, all physics simulations and other interactions in MuVR are performed client authoritatively[4]. This client authoritative structure poses an important question: For any given object, who should simulate it?

Fish-Networking, and every other considered networking framework, supports the concept of object ownership. One particular user owns the object, and thus is responsible for simulating its behavior. However, these implementations typically only allow for ownership to be transferred between users upon object creation or some other manually invoked event. MuVR elaborates upon this feature by allowing ownership to be assigned to certain volumes of space and automatically transferred upon interaction.

Ownership management in MuVR is orchestrated by a Unity component appropriately named OwnershipManager. Since ownership management is facilitated by a component, it is an opt-in feature, thus certain objects (notably the UserAvatars discussed in the next section)

[4]Performed on the local client's machine, with the results relayed to other clients through the server. In contrast to a server authoritative model where all of the simulations are performed on the server and then propagated to the clients.

can simply belong to a single user without any possibility of transfer. The OwnershipManager has two main responsibilities: first, it is assumed that if someone is actively interacting with an object, they own it and are responsible for its simulation. Second, we facilitate a simplified version of Kawano and Yonekura's Allocated Topographical Zone [15] (AtoZ) algorithm that we call Ownership Volumes.

Early implementations of Ownership Volumes were afflicted with an interesting bug where the small amount of jitter present when a physics simulation changes owners would cause an object to re-enter the volume it just departed, which often resulted in a jittery loop of repeated ownership transfers that could last for up to several seconds. We solved this issue by adding a short window of time lasting 10 ticks, approximately 78 milliseconds, after an ownership transfer occurs during which another ownership transfer can not occur.

Ownership Volumes, unlike AtoZ's regions which dynamically morph based on users' positions, are predefined fixed regions of space. This allows for more fine-grained control over exactly where ownership transfers will occur. Ownership Volumes have several methods of deciding who owns them. The two primary methods, oldest user and newest user, rely on collisions to detect when users enter or exit the volume. Additional methods are provided where ownership is assigned to the objects creator and ownership is not managed by the component but instead managed manually, similar to how ownership is managed by default in Fish-Networking.

3.2 Clear Separation of Inputs, Visuals, and Networking

One of MuVR's major priorities is laying a foundation for integrating mixed VR and non-VR users into the same shared environment. For this to be possible there needs to be a separation between the visuals displayed to users, which are then synchronized over the network, and the inputs driving those visuals. To facilitate this, we have developed a novel slot-based system where pose data (three-dimensional positions and rotations) can be stored in named slots. Originally, this Pose-Slot System featured 12 slots, head and pelvis, along with left and right shoulders, elbows, wrists, knees, and ankles that should be capable of representing the majority of human poses (without regard to finger positions). We then discovered that for some applications some of these points are unnecessary and several additional points would be useful, thus we generalized to a system where a variable number of named slots can be associated with pose data.

These pose-slots act as a unified layer of glue code where various input methods can store pose data and then a single visual representation can use well known skeleton, Inverse Kinematic, or straightforward pose copying techniques to position the visuals [7, 16]. In MuVR's current design the visuals are responsible for synchronizing their state across the network, thus the Network Layer need not have any awareness of the Input Layer.

3.2.1 UserAvatar

The UserAvatar serves two major purposes. First, it tracks the object's current owner and if the local user is also the current owner it creates appropriate input controls for them. Whenever the ownership of an object changes, we reperform this check, always ensuring that only the current owner has input control. This system is designed to support multiple types of input depending on the medium of the user.

Second, it acts as the storage location for pose-slots. The Pose-Slot System is implemented using a dictionary mapping strings to referable pose data. Behind the scenes, links to this dictionary are converted to direct references to the associated pose data so that no runtime performance is lost using this system.

Arbitrary property storage can easily be added through inheritance with convenient access to an event function that is called after input is spawned, which is useful if there is any additional non-generalizable glue code that needs to be executed. Combined with the utilities provided by SyncPoses, the UserAvatar provides a powerful, generalized, and extendable input storage utility.

3.2.2 SyncPose

SyncPose is also a Unity component that is used to load or store data from or to the UserAvatar, possibly with an offset. SyncPoses attached to objects in the Input Layer read the location of objects with local input control and then store them in one of the UserAvatar's pose-slots. Similarly, SyncPoses attached to objects in the Visual Layer load the location of objects from the UserAvatar and apply that location to objects in the visual representation. To facilitate this, SyncPoses take a reference to a prefab version of the UserAvatar they are synchronizing with and provide a custom graphical interface to make selecting pose-slots simple.

Additionally, SyncPoses support locking each axis of the position and each Euler axis of the rotation so that they are not copied. This provides the capability of synchronizing from the position of one object and then copying one of the rotational axis of another object into the same pose-slot.

3.2.3 Network Synchronization

After input data has been transferred to the Visual Layer, positional information must be propagated to other clients on the network. Fish-Networking provides a NetworkTransform component that synchronizes position, rotation, and scale across the network; however, for objects not based upon the UserAvatar model, Fish-Networking does not provide any client authoritative method of synchronizing physics properties.

Thus, we implemented a NetworkRigidbody component that synchronizes the physics properties (velocity, angular velocity, gravity, and drag) utilized by Unity's physics simulation system. Due to the Client Authoritative nature of MuVR, these properties are only necessary when an ownership transfer occurs; however, we discovered that the delay encountered when synchronizing these properties during such an event, paired with the nondeterministic nature of Unity's physics system would produce unpredictable changes in the object's trajectory after an ownership transfer.

Likewise, we discovered that utilizing an unreliable method of delivery produced similar unpredictable changes. Behind the scenes, Fish-Networking uses an unreliable UDP-based C# transport that provides its own optional reliability layer with a design very similar to TCP, but without dedicated congestion control. Thus, we use Fish-Networking's reliability mode to synchronize velocity and angular velocity to all clients every tick. While other less frequently changed properties are reliably synchronized whenever a change is detected.

The UserAvatar-based design at MuVR's core has many moving parts, however, most of them are not unique. Figure 1, provides a visual summary of how these components interact. The figure's color-coded lines clearly illustrate how each layer morphs its input data into a form the next layer can understand without needing to have any awareness of the original form.

3.3 Voice

The final aspect of MuVR's design worth mentioning is its voice communication implementation. We wanted a voice implementation that was simple and did not rely on any costly

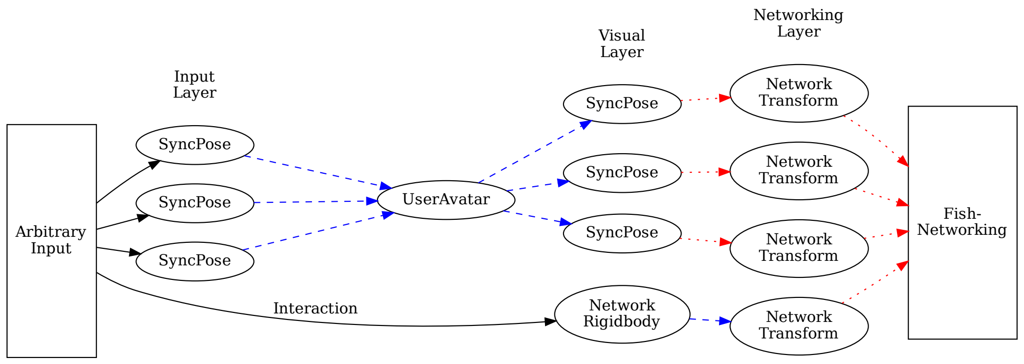

Figure 1: A visual overview of the components used to link inputs, visuals, and networking. Inputs (depicted in solid black) are applied to various objects in the environment. SyncPoses then transfer modified position information (depicted in dashed blue) from the Input Layer to the Visual Layer via the UserAvatar. Finally, global position data (depicted in dotted red) is sent through the network and used to set the position of this user as seen by other users. Simultaneously, interactions adjust properties of the physics simulation which are propagated to the rest of the network via a NetworkRigidbody and NetworkTransform.

thirdparty subscription services. UniVoice [5] met all of these requirements, however, its non-Unity idiomaticity and entirely separate networking stack were less than desirable.

Before we can discuss the adjustments we made to UniVoice, a basic understanding of its architecture is required. UniVoice is based on four main classes: an IChatroomNetwork that is responsible for transporting voice data, an IAudioSource that is responsible for acquiring voice data, an IAudioOutputFactory that is responsible for creating an audio output for every relevant peer, and finally a ChatroomAgent that manages the previous three. The ChatroomAgent supports separating users into rooms, limiting the pool of other relevant users to only the users within the same room.

The first change we made was implementing a Fish-Networking-based ChatroomNetwork. It utilizes Fish-Networking's remote procedure calls to transfer data between users, invoking subscribable events at either end so that other interested objects can listen to the data. Furthermore, a dictionary is kept in sync between all of the connected users that maps room names to lists of connections currently within those rooms. Behind the scenes, UniVoice ensures that audio is only sent to users within the same room as the sender.

In terms of improving Unity idiomaticity, we implemented a component that allows easy selection of audio input sources from within the Unity editor. The Fish-Networking-based VoiceChatroom is implemented as a component that can be easily referenced using Unity's graphical editor and provides several extra convenience functions that set up an agent using the VoiceChatroom; all referenceable using Unity's editor.

4 Applications

Once we had developed our basic implementation, we needed to test it in an actual application. We began by implementing a Ping-Pong like game with only a subset of the full game's rules, the source code for this application is available in MuVR's GitHub repository [11]. We then integrated our framework into an existing project, performed as a joint partnership with the University of Nevada, Reno's Mining & Metallurgical Engineering department, that needed multi-user functionality.

4.1 Mining Applications

The addition of multiuser functionality allowed us to collaborate with mining researchers on a training simulation. This simulation is designed to teach mining truck drivers how to utilize a new proximity warning system to avoid collisions with equipment and personnel they can not see. This integration greatly challenged our original pose-slot implementation.

Instead of only needing to translate input for a human body, we also had to synchronize several properties associated with a mining dump truck whose Visual and Input Layers are depicted in Figure 2. This caused us to realize the inadequacies of our original fixed pose-slot mapping and led to its replacement with the current dynamic system.

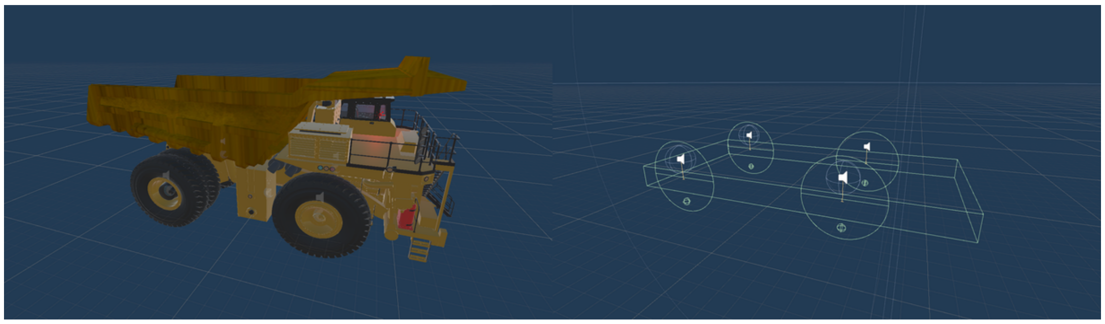

Figure 2: The mining dump truck's Visual Layer (left) and Input Layer (right).

In addition to extra poses that now needed to be tracked, we also had a car controller that needed direct access to wheel meshes for both physics and animation purposes. Similarly, several audio and haptic feedback utilities needed to be synchronized in both the Input and Visual Layers. Instead of rewriting all of this functionality ourselves, we used the UserAvatar's flexible extension ability to link these properties for us. Figure 3 lists all of the code necessary to facilitate this additional linkage.

5 Future Work and Conclusion

Looking to the future several aspects of MuVR could be improved. The most notable one being how the physics simulation handles ownership transfers. While we have been able to greatly reduce the jitter this entails, we have not been able to eliminate it. Perhaps a Predictive Behavioural Model [12] could be used to ease this issue.

Furthermore, while a foundation has been laid for work on mixed VR and non-VR interactions, we have not even scratched the surface of the work that must be done in this field. Factors

```
using UnityEngine;
using UnityStandardAssets.Vehicles.Car;

public class TruckControlLinker : InputControlLinker {
    public CarController car;
    public Telemetry telemetry; // Haptic feedback
}

public class TruckAvatar : UserAvatar {
    public GameObject[] wheelMeshes;
    public Alarm alarm;
    public NetworkCarAudio carAudio;

    protected override void OnInputSpawned(GameObject input) {
        var linker = GetComponentInChildren<TruckControlLinker>();
        linker.car.m_WheelMeshes = wheelMeshes;
        linker.car.Start();

        carAudio.carController = linker.car;
        alarm.telemetry = linker.telemetry;
    }
}
```

Figure 3: The code used to link additional properties of the Mining Dump Truck. Every public attribute of these classes is set using Unity's visual editor.

such as compelling interactions in both VR and non-VR, and appropriate body presence for both mediums still need to be developed.

In conclusion, Novotny et al. created a useful framework for multiuser VR experiences. We have followed their example and expanded upon the foundation they laid. MuVR has been designed to be simple and extendable. These claims have been tested and proved by using the framework for the development of several applications, one having requirements the framework was not originally designed to support. We hope this framework will prove to be a great boon to other developers and the VR field as a whole.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under grant numbers OIA-1301726, OIA-2019609, and OIA-2148788 Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] About netcode for gameobjects. https://docs-multiplayer.unity3d.com/netcode/current/about, Last Accessed (6/29/2022).

[2] Setting the benchmark for multiplayer games. https://www.photonengine.com/en-US/Fusion, Last Accessed (7/5/2022).

[3] XR interaction toolkit: 2.0.2. https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/index.html, Last Accessed (6/29/2022).

[4] OpenXR - high-performance access to AR and VR -collectively known as XR- platforms and devices, Dec 2016. https://www.khronos.org/openxr/, Last Accessed (6/29/2022).

[5] Vatsal Ambastha. univoice: Voice chat/VoIP solution for unity. P2P implementation included. https://github.com/adrenak/univoice, Last Accessed (6/24/2022).

[6] Kurt Andersen, Simone José Gaab, Javad Sattarvand, and Frederick C Harris. METS VR: Mining evacuation training simulator in virtual reality for underground mines. In *17th International Conference on Information Technology–New Generations (ITNG 2020)*, pages 325–332. Springer, 2020. DOI: 10.1007/978-3-030-43020-7_43.

[7] Andreas Aristidou, Joan Lasenby, Yiorgos Chrysanthou, and Ariel Shamir. Inverse kinematics techniques in computer graphics: A survey. In *Computer graphics forum*, volume 37, pages 35–58. Wiley Online Library, 2018. DOI: 10.1111/cgf.13310.

[8] Benjamin Berwick. Benchmark setup. https://fish-networking.gitbook.io/docs/manual/general/performance/benchmark-setup, Last Accessed (7/1/2022).

[9] Benjamin Berwick. Fish-net, documentation. https://fish-networking.gitbook.io/docs/, Last Accessed (6/29/2022).

[10] Gerald Combs et al. Wireshark · go deep. https://www.wireshark.org/, Last Accessed (7/11/2022).

[11] Joshua Dahl, Erik Marsh, Christopher Lewis, and Frederick C. Harris. GitHub: MuVR- a multiuser virtual reality framework for unity. https://github.com/hpcvis/MuVR/tree/MuVR-AMultiuserVirtualRealityFrameworkforUnity, Last Accessed (8/1/2022).

[12] Chen Gao, Haifeng Shen, and M. Ali Babar. Concealing jitter in multi-player online games through predictive behaviour modeling. In *2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 62–67, 2016. DOI: 10.1109/CSCWD.2016.7565964.

[13] Toni Härkönen. Advantages and implementation of Entity-Component-Systems. pages 2–5, 2019. https://urn.fi/URN:NBN:fi:tty-201905231735, Last Accessed (7/7/2022).

[14] Dorota Kamińska, Tomasz Sapiński, Sławomir Wiak, Toomas Tikk, Rain Eric Haamer, Egils Avots, Ahmed Helmi, Cagri Ozcinar, and Gholamreza Anbarjafari. Virtual reality and its applications in education: Survey. *Information*, 10(10):318, 2019. DOI: 10.3390/info10100318.

[15] Yoshihiro Kawano and Tatsuhiro Yonekura. On a serverless networked virtual ball game for multiplayer. In *2005 International Conference on Cyberworlds (CW'05)*, pages 270–278, 2005. DOI: 10.1109/CW.2005.68.

[16] John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172, 2000. DOI: 10.1145/344779.344862.

[17] Alexander Novotny, Rowan Gudmundsson, and Frederick C. Harris. A unity framework for multi-player VR applications. *International Journal of Computers and Their Applications*, 27(3):115–121, Sep 2020. DOI: 10.29007/r1q2.

[18] Anastasia Rychkova, Alexey Korotkikh, Andrey Mironov, Artem Smolin, Nadezhda Maksimenko, and Mikhail Kurushkin. Orbital battleship: A multiplayer guessing game in immersive virtual reality, 2020. DOI: 10.1021/acs.jchemed.0c00866.

[19] Heather A Stoner, Donald L Fisher, and Michael Mollenhauer. Simulator and scenario factors influencing simulator sickness. *Handbook of driving simulation for engineering, medicine, and psychology*, pages 220–243, 2011.

[20] Unity Technologies. https://unity.com/, Last Accessed (6/29/2022).

[21] vis2k. Mirror networking, documentation. https://mirror-networking.gitbook.io/docs/, Last Accessed (6/29/2022).