uMuVR: A Multiuser Virtual Reality and Body Presence Framework for Unity

Joshua Dahl*, Erik Marsh*, Christopher Lewis*, and Frederick C. Harris Jr.*
University of Nevada, Reno, Nevada, USA

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Due to the rapidly evolving nature of the Virtual Reality field, many frameworks for multiuser interaction have become outdated, with few, if any, designed to support mixed virtual and non-virtual interactions.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 the development of mixed interactions based upon a novel method of ensuring that inputs, visuals, and networking can all communicate without needing to understand the others' internals. This framework also provides utilities for representing user avatars in a physicalized manner while supporting a range of different input methods. We tested this framework in the development of several applications and show that it can easily be adapted to support application requirements it was not originally designed for.

Key Words: Graphics, virtual reality, body presence, multiplayer, networking, neural networks, physics simulation, boneworks

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, while very few people are investigating ways of portraying a user's entire body in VR without the need for additional trackers. Novotny et al. [35] 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. uMuVR [12] (pronounced You-Mover) serves as a total overhaul of the framework designed by Novotny et al. with major emphasis put on supporting the OpenXR [44] standard as well as the newly emerging UltimateXR [50] framework and reworking the core concepts behind Novotny et al. 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. We have also developed some innovative techniques for representing users entire bodies (including their feet) in virtual reality in a physically interactable fashion. The literature lacks not only in facilities for supporting multiuser VR interactions, but also in predicting the position of their feet and legs without explicit sensor data.

The rest of this paper is structured as follows: Section 2 reviews some of the existing literature and details several of the frameworks we are using. Section 3 then dives into library choices that we made, with particular emphasis placed upon comparisons to existing networking and compression libraries. Section 4 details the design and implementation decisions of the networking aspects of uMuVR. Section 5 explores our novel foot prediction algorithm and physicalized interaction system as well as several of the more traditional techniques we used to portray the users' upper bodies. Section 6 details two applications we implemented with uMuVR, showing how easily the framework can be extended; and finally Section 7 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 [3], educational experiences [24], and entertainment experiences [40] are now being developed for VR. Thus facilities to ease this development would be beneficial.

Alternatively, much work has been done in the field of Virtual Body Presence. Currently, this work is based heavily upon Skeleton based Forward Kinematics and Inverse Kinematics (IK) techniques; Lewis et al. [30] and Aristidou et al. [5] respectively provide overviews of these two foundational topics.

*Department of Computer Science and Engineering. Emails: joshuadahl@nevada.unr.edu, erik.i.marsh@gmail.com, christopher_le1@nevada.unr.edu, fred.harris@cse.unr.edu

Currently, the literature suggests that being in a virtual environment where we can perceive the full bodies of others to lead us becoming more accepting of a complete body of our own [29]. Although there is a bit of contention to this theory [20], the literature generally seems to agree on a statement similar to the one above. However, it has also been theorized that in action-packed experiences (or types likely to invoke a "flow state") we are prone to focus more on the enemy or obstacle presented and less on ourselves [31].

Counter-arguments aside, little work has been done on analyzing the effects self interactions have on body presence, despite the fact that work has been put into non-contact-based interactions; work which seems to indicate that the presence of a full emotive body enhances facial expressiveness [28]. However, for all of these, either the legs were ignored [20, 31, 28] or tracked using additional sensors [29] using methods similar to the ones described by Caserman et al. [6]. The literature generally seems to indicate that more tracking leads to better acceptance, and yet the entire lower body is either ignored or handled with sensor arrays the average consumer may not have.

As far as the present authors are aware, there has been minimal, if any, work done by academia on predicting points on the body such as the feet and legs without the need for additional trackers the average consumer may not have. That being said, preliminary work is being done in this area outside of academia [9, 43], but the field as a whole is still in its infancy.

While leg and foot prediction may be in its infancy, physicalized interaction has largely been driven by the video game studio STRESS LEVEL ZERO [43] and their games BONEWORKS and BONELABS. While in academia work has been done on physicalized hand interaction by a chain of authors starting with Nasime and Kim [33] and (currently) ending with Delrieu et al. [13]. Additionally, libraries providing support for a system similar to the one proposed by Nasime and Kim are available for Unity [18], which additionally provide proprietary extensions to the whole body. However, there is not currently a unified framework tying all of these technologies together.

2.1 Unity

Unity [47] is a commercial game engine with extendable scripting in the C# language. It utilizes the Object-Oriented Composition paradigm [19] 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 uMuVR 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 this classification of libraries. This is of extra significance due to the recent trend towards standalone VR headsets, many of which run the Android [36] operating system which provides a very different set of facilities when compared to a standard desktop environment. 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.

2.2 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, named OpenVR, has been deprecated in favor of OpenXR [44], 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 eXtended Reality (XR) devices (an umbrella term used to describe both Virtual and Augmented Reality devices). However, accessing functionality inherent to a single device or manufacturer using OpenXR is difficult.

Unity provides the XR Interaction Toolkit [48] (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.

Late into the development of our original paper VRMADA released an experimental competitor to the OpenXR and XRIT stack they call UltimateXR [50]. UltimateXR itself acts as a middleware between most of the major VR device manufacturers and provides a fallback mechanism to use OpenXR if none of the platforms it has support for are available. This method was chosen since the OpenXR standard does not provide support for several common VR features, the most notable being haptic hand feedback. Additionally, UltimateXR provides not only hand presence utilities (and one of the most feature rich hand presence facilities available no less) but an IK based solution for estimating the position of a user's entire upper body. Finally, as a cherry on top, the framework also provides a utility which will blank the user's screen if their head happens to go inside of a solid object.

The combination of OpenXR and XRIT was chosen as

Table 1: VR framework comparison summary

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

\+ Includes support for not only hand presence, but pose estimation for the entire upper body.
* 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.

the foundation for uMuVR originally. However, UltimateXR provides an enticing set of features but it is experimental and could be abandoned at any point by its developer. Thus we decided to provide support for both frameworks. Table 1 summarizes the differences between the discussed VR frameworks.

3 Benchmarks

While the choice of VR framework was fairly straightforward given our goals and SteamVR's recent deprecation, choosing a networking solution proved more difficult, many of the frameworks aim to be approximately equivalent with the primary distinguishing factor being their performance. There also exists a multitude of high throughput compression libraries that all serve the same role (namely making data smaller) with different throughput-to-compression ratios.

3.1 Networking Frameworks

We began by examining several different Unity networking frameworks. Several of these frameworks require that the code for clients and servers be written in different projects; 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 [14] 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 [16], Mirror [49], and Unity's Netcode for GameObjects [45] (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 while Fish-Networking and NCGO support a client-server architecture where the server can either be dedicated or hosted on one of the clients. 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.

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 [15] 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[1]. All of the code utilized for these benchmarks can be found in uMuVR's Git repository [11] spread across several branches whose names all start with "benchmark/". Thirty clients were chosen since more would result in GPU throttling. Bandwidth information was captured using Wireshark's [8] 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 and averaged over a period of five minutes.

Once the benchmarks had been performed, Fish-Networking proved to perform better than its competition, with similar frame rates and significantly reduced bandwidth overhead; and thus reduced bandwidth utilization indicating that more information can be exchanged before network infrastructure becomes overloaded. This leads to a direct increase in the

[1]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.

Table 2: Networking framework comparison over several performance metrics

Framework	Estimated Max Concurrent Users	Average FPS	Average Bandwidth	Cost	Supported Architectures	Matchmaking	Voice
Fish-Networking	500+	60.19	0.94 MB/s	Open Source	Hosted/ Dedicated	No	No
Mirror	200+†	60.23	2.15 MB/s‡	Open Source	P2P	No	No
Netcode for Game-Objects	Not Published	59.97	3.42 MB/s‡	Open Source	Hosted/ Dedicated	No	Yes§
Photon Fusion (Shared Topology)	2000	25.08	1.82 MB/s	Per User	Hosted/ Dedicated/ P2P	Yes	Yes¶

† 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.

number of users that can be connected at once or the amount of network synchronized objects that can be in a scene while still utilizing the same amount of bandwidth. 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 factors considered, including several of Fish-Networking's nicer usability features, Fish-Networking became the clear choice to base uMuVR upon.

3.2 Compression Libraries

Data compression is a relatively hot field with many competing algorithms. It is worth noting that each considered library is a C# port of the original (usually C) algorithm. We began by considering the LZ4 [27] compression algorithm. This implementation along with the LZF [37] implementation provides an interface that allows the same buffer to be reused, reducing the burden on the C# Garbage Collector. LZF also has the advantage of being implemented as a single file and providing a tweakable compression ratio, we tested both the default high compression ratio, a faster compression ratio more suited to our needs, and a version of the implementation [22] tuned for Unity which we quickly discarded after we discovered its poor performance relative to the more general implementation.

Additionally Snappy [1], Zstandard [41], and Bontli [51] where all considered. Neither the Snappy nor Bontli implementations support buffer reuse and the Zstandard implementation requires build processors that prevent it from being used within Unity. Additionally, Bontli is a slower algorithm, thus we primarily included it as a reference for high-end compression ratios.

Since we are going to be compressing vocal audio from users' microphones, we performed this benchmark on a similar sample of audio. All of the code utilized for these benchmarks can be found in FishyVoice's Git repository [10] spread across several branches whose names all start with "benchmark/". Our voice networking library delivers segments of audio with a size of approximately 1600 bytes, thus we take similarly sized chunks of audio (55 chunks per second from our clip resulting in samples 1603 bytes in size). Since the sample clip is about 5 seconds long we collect 1375 looping samples representing five passes over the clip. For each sample, we determine how long it takes to serialize and then deserialize the packet (extra data is randomized using a fixed seed so that all algorithms are given the same data) then once these values are averaged we subtract the average time taken when no compression is used to find the extra time taken. We also record the size of both the compressed and uncompressed packet which we use to calculate the compression ratio. We discard the first sample taken from this analysis since there is a noticeable increase in time needed to initialize the serialization system, the time this extra initialization takes averaged over five startups along with the rest of the data mentioned above is presented in Table 3. All results were collected on the same machine used in the networking benchmarks, but Unity 2021.3.15f was utilized and the results were written to a CSV file by the benchmark itself.

LZF with an HLOG of 11 has the best trade off of performance to compression ratio. Slightly higher compression ratios can be achieved by the LZF algorithm at the cost of significantly more performance degradation. Thus the LZF implementation with this set of parameters was chosen as our audio compression library.

4 Networking Design and Implementation

4.1 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

Table 3: Compression library comparison over several performance metrics

Library	Extra Compression Time	Extra Decompression Time	Average Compression Ratio	Average Initialization Time
Uncompressed	$0\mu s$	$0\mu s$	1	$13480.4\mu s$
LZ4	$29.0859\mu s$	$10.1048\mu s$	3.2446	$12488.2\mu s$
Snappy	$73.9032\mu s$	$30.5655\mu s$	3.7622	$5402.0\mu s$
LZF (hlog10)	$24.75\mu s$	$14.33\mu s$	3.7877	$3355.0\mu s$
LZF (hlog11)	$21.42\mu s$	$12.18\mu s$	3.8535	$3463.8\mu s$
LZF (hlog13)	$24.89\mu s$	$12.72\mu s$	3.8536	$2794.0\mu s$
LZF (hlog16)	$54.10\mu s$	$14.01\mu s$	3.8571	$2954.6\mu s$
Bontli	$469.51\mu s$	$102.81\mu s$	7.7192	$23269.0\mu s$

as described by Stoner et al. [42], which could easily invoke a bout of simulator sickness. To account for this discrepancy, all physics simulations and other interactions in uMuVR are performed client authoritatively (Meaning they are performed on the local client's machine, with the results relayed to other clients through the server. This model exists in contrast to a server authoritative model where all of the simulations are performed on the server and then propagated to the clients). This client authoritative structure poses an important question: For any given object, who should simulate it?

Fish-Networking, and every other considered networking framework, support 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. uMuVR elaborates upon this feature by allowing ownership to be assigned to certain volumes of space and automatically transferred upon interaction.

4.1.1 Ownership Management. Ownership management in uMuVR 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) 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 [25] (AtoZ) algorithm that we call `OwnershipVolumes`.

4.1.2 Ownership Volumes. `OwnershipVolumes`, 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. `OwnershipVolumes` 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.

Early implementations of `OwnershipVolumes` 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 ten ticks, approximately 78 milliseconds, after an ownership transfer occurs during which another ownership transfer can not occur.

4.2 Clear Separation of Inputs, Visuals, and Networking

One of uMuVR'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 as defined by Hummel and Atkinson [23]. Various input methods can store pose data and then a single visual representation can use either straightforward pose copying techniques or more complex techniques which are elaborated on in Section 5 to position the visuals (although there is nothing preventing a developer from creating their own additional techniques). In uMuVR'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.

4.2.1 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.

4.2.2 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 (Unity's word for a prepackaged and reusable set of entities and with components preattached) version of the `UserAvatar` they are synchronizing with and provide a custom graphical interface as shown in Figure 1 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.

4.2.3 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 uMuVR, 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. Fish-Networking is built upon LiteNetLib [38], an unreliable UDP-based [34] C# transport that provides its own optional reliability layer with a design very similar to TCP [34] but without dedicated congestion control. Thus, we use LiteNetLib'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.

4.2.4 Post Processing. We discovered through some of our testings with Neural Network driven inputs, that it might be useful to provide developers a method of processing poses stored in a `UserAvatar` after the fact. For some of our early tests, we made use of a `PostProcessingUserAvatar` which runs a First Order Exponential Averaging Low Pass Filter, the notation for which is detailed in Equation 1, over the Neural

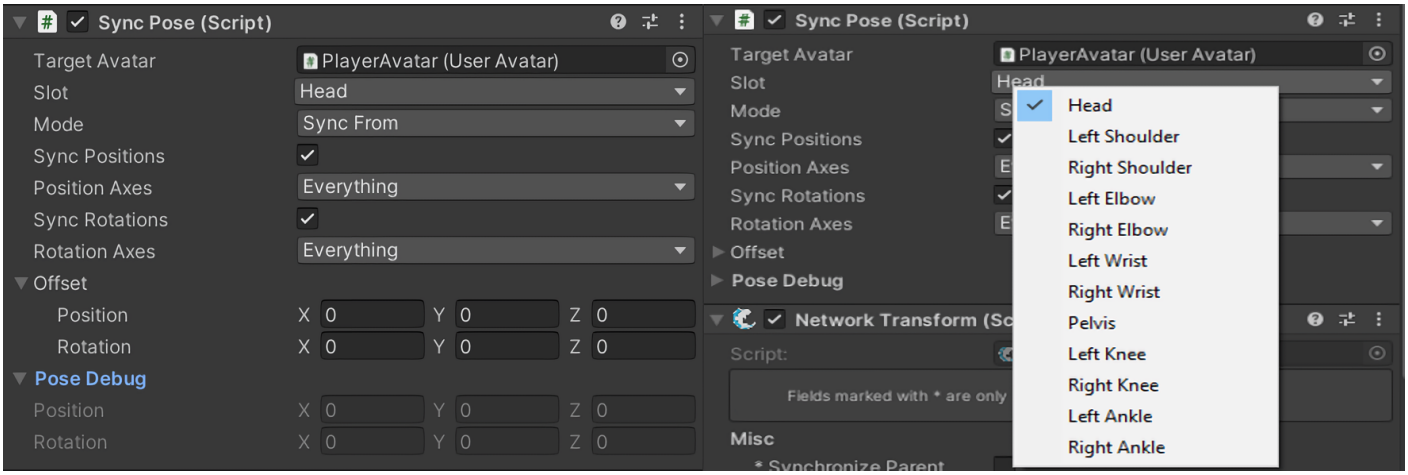


Figure 1: The interface provided for `SyncPoses` (left) and customized dropdown that makes selecting pose-slots simple (right)

Network generated poses to smooth some jittery behavior.

$$y(n) = a * x(n) + (1-a) * y(n-1) \quad (1)$$

To facilitate this we provide a separate `PostProcessingUserAvatar` type that provides a separate set of pose slots which represent the unprocessed pose data; the names of which we synchronize with the developer-defined names. `SyncPoses` are aware of this separation and will properly store and load information to and from the proper slots. Every frame we iterate over every slot, and for each slot we call a developer definable processing function that performs whatever arbitrary calculation they would like on the old pose data given the new pose data. For cases where there is no interdependence on the states of other poses, we provide an option to run these iterations using Unity's C# Job System [46]. This allows the processing to occur on background threads.

The `UserAvatar`-based design at uMuVR's core has many moving parts, however, most of them are not unique. Figure 2, 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. It is worth noting that the Pose-Slot System is a lossy approximation, thus certain rare and extreme poses are not representable using this system.

4.3 Voice

The final aspect of uMuVR's networking design worth mentioning is its voice communication implementation. We wanted a voice implementation that was simple and did not rely on any costly third-party subscription services. UniVoice [2] 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 `IChatroomNetwork` we call `VoiceNetwork`. It utilizes Fish-Networking's remote

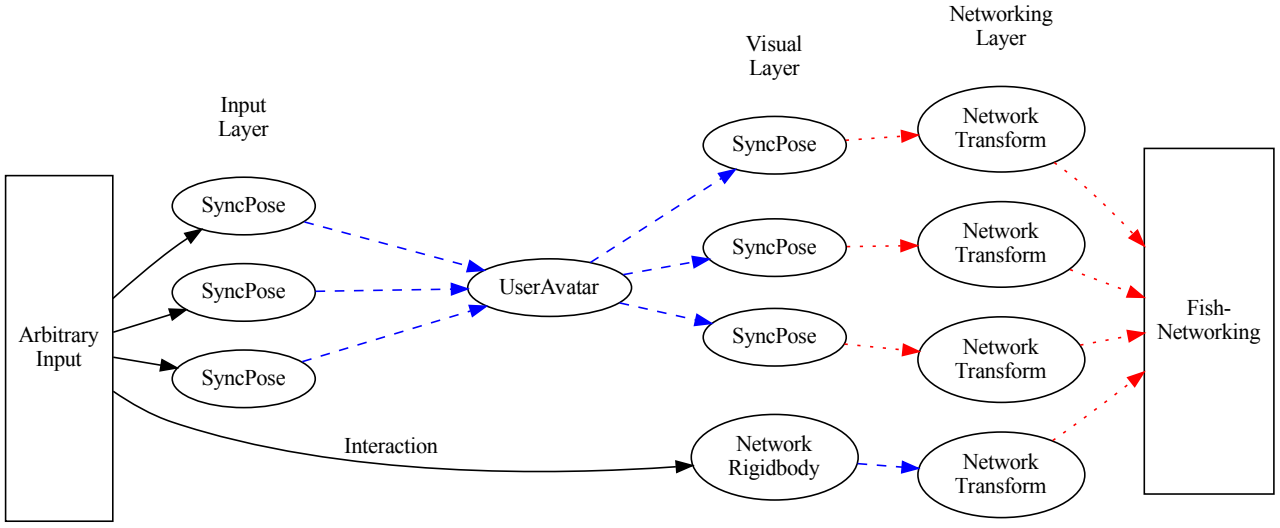


Figure 2: A visual overview of the components used to link inputs, visuals, and networking. Inputs (depicted in solid black) are applied to various objects in the environment. `SyncPoses` then transfer modified position information (depicted in dashed blue) from the Input Layer to the Visual Layer via 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`

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. Additionally, the `VoiceNetwork` 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 which utilizes the `VoiceNetwork`; all referenceable using Unity's editor.

Additionally, we added two new features on top of the UniVoice stack: automatic positional audio and disableable voice compression. The audio packets which we transfer through Fish-Networking include an additional variable that encodes the position of the speaker. A `PlayerAudioPositionReference` component is used to determine where the user should be positioned from the perspective of other users. Finally, a `PositionalAudioOutput` factory is provided which creates a specialized audio source for each user. The default `VoiceNetwork` can detect the presence or absence of positional audio components and will automatically adapt as needed.

We compress audio using the LZF [37] compression library. A discussion of why this library was chosen can be found in Section 3.2. Our audio packets have a custom Fish-Networking serializer and deserializer pair which convert the packet into a byte array which is then passed through LZF. A C# define is provided so that from the Unity script control page you can easily disable or enable compression.

5 Body Presence Design and Implementation

uMuVR's current body presence facilities can be separated into two categories: the half which lives in the Input Layer and the half which lives in the Visual Layer. Everything, including the physics simulation, has been designed in such a fashion that it can simply be disabled on remote clients who will rely on data coming from the network to position remote avatars in their scenes.

5.1 Input Layer

The input layer half has a very simple goal, namely, calculate the position and rotation of all 12 original pose-slots of the user avatar. Actually, at present the Body Presence system is using 42 pose-slots (the 12 original and a slot for every finger joint on both hands). We hope to simplify this in the future but that is a story for another paper.

5.1.1 UltimateXR. Everything from the hips up is simple from uMuVR's perspective. UltimateXR [50] provides Cyclic Coordinate Descent IK [26] based utilities for estimating elbow, spine, hip, and neck positions based on the position of the user's controllers and head. Unfortunately, we do not currently have a solution for accurately predicting shoulder positions, thus if shrugs are desired they will need to be externally tracked.

Additionally, UltimateXR provides a system for defining grab points and associated poses on interactable objects. This system is used to define the position of every finger joint. We can thus simply extract the relevant points from their implementation and pass the hip position off to a Phase-Functioned Neural Network controller.

5.1.2 PFNN. Phase-Functioned Neural Networks [21] (PFNN) provide a system for cheaply and procedurally generating the next pose of a walking cycle based on a constantly progressing phase variable. This system is capable of creating animations for convincingly traversing complex environments; give or take the fact that sometimes the feet do not find themselves correctly planted on the ground, a flaw which can be corrected with a secondary foot placement pass which we shall discuss in Section 5.1.3.

The PFNN network takes as input a target direction, the relative speed it should approach its target, and the pose last frame. It then generates a new pose. Unfortunately, this process is very sensitive to deviations from the information it was trained on. For instance, the publicly available weights utilized by Holden et al. [21] assume that poses will be requested at a constant rate of 60 times per second. Thus we have a controller sitting over the model which only requests a new pose if a 60th of a second has elapsed since the last pose request.

Additionally, to keep the legs under the body, our controller will define directions and velocities in such a way that the legs will be positioned under the hips when everything is said and done. However, care needs to be taken with the network's target position, once it has "reached" its target position it will stop rotating to match the user, but if it never "reaches" its target the simulation will destabilize. Thus our controller goes through a rather complicated set of procedures to dynamically adjust the threshold at which the network is considered to "reach" its target based on the hip's velocity and the difference in angle between the hips and feet.

This whole process also fails if the experience being developed uses a teleportation-based locomotion system. The legs will be left behind as the upper body teleports and then wander over to reattach themselves. While we are sure certain experiences could use this as a hilarious gimmick, for the average case this behavior is undesirable. Thus the controller also detects when the upper and lower body's positions have diverged too much and will reset the leg simulation to match the upper body in this case.

Much like the shoulders, this simulation can only predict "normal" movements such as standing, walking, and crouching. It will encounter issues with more exotic motions such as laying down or performing a back flip. External trackers will still be needed if these sorts of motions will be commonly necessary for the experience.

5.1.3 Slope Aware Foot Placement. Once PFNN has generated a leg and foot placement for us, we then need to ensure that the feet rest squarely on the ground. We begin by projecting the toes straight down onto the ground. Foot placement is then based on the height of the ankle above the toes which we consider a configurable constant, named Δ. We then fall into two cases based on the height of the ankle above the ground minus Δ, which we call H. If H is greater than Δ the ground is steep and we don't modify the foot placement, if H is less than or equal to Δ we move the ankle straight down to a point Δ above the ground. These two cases are represented graphically in Figure 3. This procedure mimics human behavior where we stand on our toes on steep inclines and our heels on more gradual inclines.

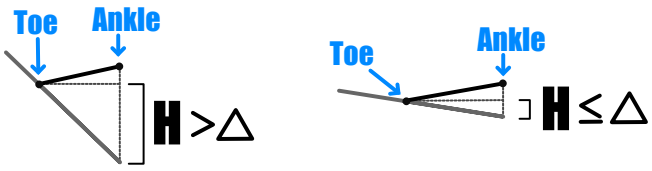


Figure 3: A representation of the Ankle and Toe points, and how they would relate to the ground in the Steep (left) and Not Steep (right) cases

Behind the scenes, all of the points from UltimateXR and PFNN are combined into a hidden `UserAvatar`. The foot placement algorithm depends on this abstraction in uMuVR's design to simplify the foot placement problem to just sliding points. We calculate how much we had to move the ankle down, and gradually spread decreases in height through the rest of the points to compensate. When combined with the IK system we run in the visual layer, this results in very convincing movements.

Additionally, the phase variable from PFNN is used as a weight factor to describe how strongly the foot placement should be applied. In parts of the phase that correspond to a raised foot in the walk cycle, we completely ignore the foot placement, while in phases where the foot should be planted, we apply it with a weight of 100% and blend between the two as necessary.

5.2 Visual Layer

The half of the system which lives in the Visual Layer is inspired from two primary sources: PhysIK [7] and HPTK [18]. PhysIK is a system for expressing IK like animations using a physics simulation. This allows for the "animations" to interact with themselves, for instance an arm can not be pushed through its body. While HPTK is a Unity library providing physics based hand interaction, instead of needing to press a button on your controller to pick up a box, it allows the user to simply use their hands to pick up the box.

HPTK's influence is simpler than that of PhysIK. They provide a proxy hand which represents the input data received from the user's controllers. Similarly we present a proxy hand to the user in the Input Layer, thus it only exists for them.

While a physics based system seems to work wonderfully for the upper body, we encountered numerous issues integrating such a system into the avatar's feet and legs. Thus the lower body is animated using the more traditional FABRIK [4] IK algorithm which is used to determine the rotations necessary for the feet, knees, and hips to all be properly positioned.

We originally were using FABRIK based IK for the arms and spine, as well as the legs. But perusing alternative implementations during development showed some of the interesting benefits of using a physics simulation; for instance, a hand properly interacting with the opposite out-stretched arm. A behavior which a properly tuned physics simulation simply provides that would be a tremendous amount of effort to emulate using traditional IK techniques. Thus we began implementing a version of PhysIK that would work in Unity. Due to the nature of Unity's physics system, this means that almost everything from PhysIK, except the ideas, was abandoned.

More specifically two main physics "constraints" have been adapted. A constraint which pulls a joint towards a location and a constraint which rotates a joint to match a rotation which, when both visualized, leave the model looking a lot like a puppet on strings as depicted in Figure 4.

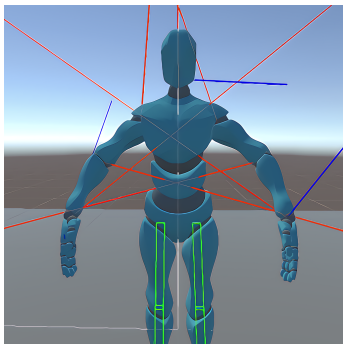


Figure 4: A visualization of the forces (red) and torques (blue) acting on an example avatar

A puppet on a string is an excellent explanation of how the location constraint works. It simply applies a force towards the targeted point which grows stronger the further away from the target the joint is; almost as if it is pulled by an invisible string. Figure 5 lists the code, executed every physics tick, to implement this functionality.

Unfortunately, there is not a nice analogy for how the rotation-matching constraint functions. It simply calculates the quaternion needed to convert the current world space rotation into the target world space rotation and then converts that quaternion to the angular velocity form that Unity expects when applying torques. There is a slight wrinkle; when applied to the center of mass, this rotation looks odd, so instead, the center of

```
// Every physics tick...
protected void FixedUpdate() {
    // We are applying a force from our
    //  current position towards the
    //  target
    var force = (target.position
        - transform.position)
        * springConstant;
    // Cap the force
    force = force.normalized
        * Mathf.Min(
            force.magnitude,
            maxForce
        );
    // Apply the force
    rigidbody.AddForce(force);
}
```

Figure 5: The code for the constraint which pulls a joint towards a target

mass has to be moved to the origin of the joint's local coordinate space since Unity always applies torques relative to the center of mass. The code for this constraint is listed in Figure 6.

6 Applications

Once we 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. We then integrated our framework into an existing project [32], performed as a joint partnership with the University of Nevada, Reno's Mining & Metallurgical Engineering Department, that needed multiuser functionality.

6.1 Ping-Pong

The main goal of the Ping-Pong application was to stress test the physics and ownership transfer aspects of uMuVR. The framework allowed this application to be implemented almost entirely without additional code; only two simple scripts: one to manage scoring and balls and another to position the scoreboard; combined only accounting for approximately 100 lines of code, where required.

Ping-Pong provides the quintessential application of `OwnershipVolumes`: a volume is positioned on either side of the net and each player owns half of the table. Figure 7 depicts this configuration. Additionally, the fast-paced nature of Ping-Pong illuminated many of the issues inherent in the behavior of the physics simulation surrounding an ownership transfer, providing us a test-bed for experimenting to reduce the jitter.

```
// Every physics tick...
protected void FixedUpdate() {
    // Reset the center of mass so that
    //  torque is properly applied
    rb.centerOfMass = Vector3.zero;
    rb.inertiaTensor = Vector3.one;

    // Calculate how much we need to
    //  rotate to match the object
    var diff = offset
        * target.rotation.Diff(
            transform.rotation
        );
    diff.ToAngleAxis(
        out var angle,
        out var axis
    );
    // Apply a torque to make this change
    //  occur
    rigidbody.maxAngularVelocity
        = springConstant;
    rigidbody.AddTorque(
        axis * (angle * Mathf.Deg2Rad)
            * springConstant,
        ForceMode.VelocityChange
    );
}
```

Figure 6: The code for the constraint which rotates a joint to match another joint's rotation. The center of mass needs to be reset so that torque is applied to the object's origin and not its center of mass

6.2 Mining Application

The addition of multiuser functionality allowed us to collaborate with Mining researchers on a training simulation [32]. 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 8. This caused us to realize the inadequacies of our original fixed pose-slot mapping and led to its replacement with the current dynamic system.

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

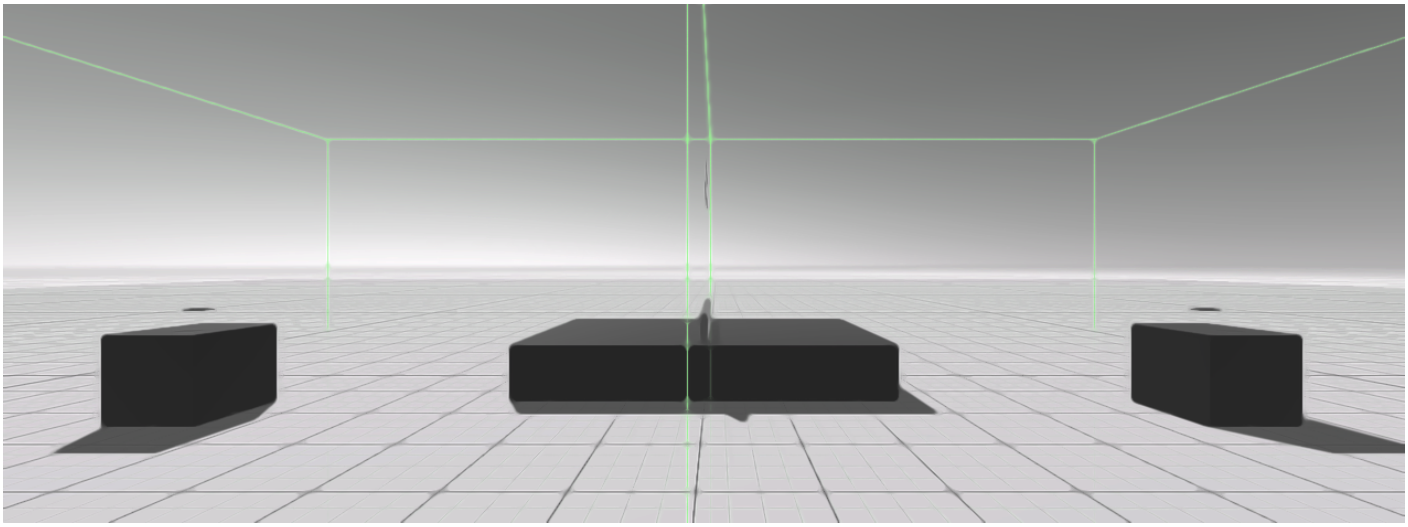


Figure 7: The Ping-Pong table with its two `OwnershipVolumes` outlined in green

of rewriting all of this functionality ourselves, we used the `UserAvatar`'s flexible extension ability to link these properties for us. Figure 9 lists all of the code necessary to facilitate this additional linkage.

7 Future Work and Conclusion

Looking to the future several aspects of uMuVR 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 Behavioral Model [17] 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 such as compelling interactions in both VR and non-VR still need to be developed.

Currently, the physics simulations in the body presence system takes a lot of manual tweaking to achieve acceptable results. Ways to automate or at least partially automate this process would be of great utility. Additionally, interactions between the simulated hands and other objects in the scene are inconsistent. Research has been done on this problem [13] and thus implementing at least part of these solutions would be beneficial. On the topic of hands, a lot of slots within the current `UserAvatars` are being devoted to storing the poses of every finger joint. It seems likely that it should be possible to simplify this representation to simply the knuckle's rotation and a single value representing how open or closed the rest of the finger is.

The publicized weights for PFNN lead to an avatar that tends to shuffle its feet, which can be undesirable when the user is standing still. Training weights optimized for the movements users tend to make in VR or maybe upgrading to newer work done on the same problem could prove beneficial. In a similar vein, the ad hoc tensor implementation that we are

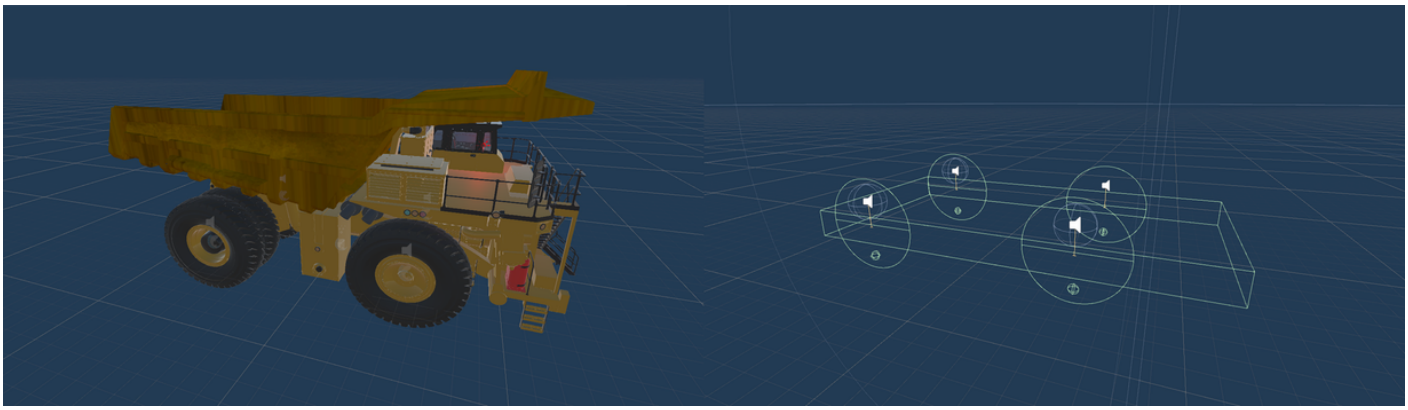


Figure 8: The mining dump truck's visual layer (left) and input layer (right)

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

public class TruckControlLinker
  : InputControlLinker {
    public CarController car;

    // Haptic feedback
    public Telemetry telemetry;
}

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 9: 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. Note that this example is based upon a slightly older version of uMuVR which required a separate `InputControlLinker` component. This requirement has since been removed

currently using for PFNN is not hardware accelerated. If we wish to use additional neural networks for tasks such as voice transcription [39], unifying our tensor library in a manner that supports hardware acceleration would be desirable.

Finally, none of this implementation matters if either users or developers do not find the framework appealing. Thus several user studies examining various aspects of the framework's implementations and development usability will need to be conducted.

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. uMuVR 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. It additionally provides a unified framework for providing rich body presence facilities for users to experience. 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-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] aloneguid (Ivan G). "GitHub - aloneguid/IronSnappy: .NET Managed Port of Google Snappy". `https://github.com/aloneguid/IronSnappy`, Last Accessed (1/11/2023).

[2] Vatsal Ambastha. "UniVoice: Voice chat/VoIP Solution for Unity. P2P Implementation Included". `https://github.com/adrenak/univoice`, Last Accessed (1/11/23).

[3] Kurt Andersen, Simone José Gaab, Javad Sattarvand, and Frederick C Harris. "METS VR: Mining Evacuation Training Simulator in Virtual Reality for Underground Mines". In Shahram Latifi, editor, "17th International Conference on Information Technology–New Generations (ITNG 2020)", Springer International Publishing. pp. 325-332, 2020. doi:10.1007/978-3-030-43020-7_43.

[4] Andreas Aristidou and Joan Lasenby. "FABRIK: A Fast, Iterative Solver for the Inverse Kinematics Problem". *Graphical Models*. 73(5):243-260, 2011. doi:10.1016/j.gmod.2011.05.003.

[5] Andreas Aristidou, Joan Lasenby, Yiorgos Chrysanthou, and Ariel Shamir. "Inverse Kinematics Techniques in Computer Graphics: A Survey". *Computer graphics forum*. 37(6):35-58, September 2018. doi:10.1111/cgf.13310.

[6] Polona Caserman, Philipp Achenbach, and Stefan Göbel. "Analysis of Inverse Kinematics Solutions for Full-Body Reconstruction in Virtual Reality". In "2019 IEEE 7th International Conference on Serious Games and Applications for Health (SeGAH)", pp. 1-8, 2019. doi:10.1109/SeGAH.2019.8882429.

[7] Frederick Choi. "PhysIK: Physics Based Inverse Kinematics for Character Posing and Animation". [online]. `https://www.cs.rpi.edu/~cutler/`

classes/advancedgraphics/S19/final_projects/fred.pdf, Last Accessed (1/11/2023), 2019.

[8] Gerald Combs. "Wireshark · Go Deep." https://www.wireshark.org/, Last Accessed (1/11/2023).

[9] John Coumerilh. "Standable: Full Body Estimation". https://www.standablevr.com/projects-8, Last Accessed (1/11/2023).

[10] Joshua Dahl. "Compression Benchmarks at Benchmark/Base". [online]. https://github.com/hpcvis/FishyVoice/tree/benchmark, Last Accessed (1/11/2023).

[11] Joshua Dahl. "Networking Benchmarks at Benchmark/Base". [online]. https://github.com/hpcvis/MuVR/tree/benchmark/base, Last Accessed (1/11/2023).

[12] Joshua Dahl, Erik Marsh, Christopher Lewis, and Frederick C. Harris. "GitHub: uMuVR-A Multiuser Virtual Reality and Body Presence Framework for Unity". [online]. https://github.com/hpcvis/MuVR/tree/uMuVR-AMultiuserVirtualRealityFrameworkforUnity Last Accessed (1/11/2023).

[13] Thibauld Delrieu, Vincent Weistroffer, and J. P. Gazeau. "Precise and realistic grasping and manipulation in Virtual Reality without force feedback". In "2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)", pp. 266-274, 2020. doi:10.1109/VR46266.2020.00046.

[14] Exit Games Inc. "Setting the Benchmark for Multiplayer Games. — Photon Engine". [online]. https://www.photonengine.com/en-US/Fusion, Last Accessed (1/11/2023).

[15] FirstGearGames. "Benchmark Setup". https://fish-networking.gitbook.io/docs/manual/general/performance/benchmark-setup, Last Accessed (1/11/23).

[16] FirstGearGames. "Introduction - Fish-Net: Networking Evolved". https://fish-networking.gitbook.io/docs/, Last Accessed (1/11/23).

[17] 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)", pp. 62-67, 2016. doi:10.1109/CSCWD.2016.7565964.

[18] Jorge Juan González. "FinalIK - HPTK". [online]. https://jorge-jgnz94.gitbook.io/hptk/integrations/finalik-1, Last Accessed (1/11/2023).

[19] Toni Härkönen. "Advantages and Implementation of Entity-Component-Systems". [online]. Bachelor of Science Thesis https://urn.fi/URN:NBN:fi:tty-201905231735, https://trepo.tuni.fi//handle/123456789/27593, Last Accessed (1/11/2023). April 2019.

[20] Fernanda Herrera, Soo Youn Oh, and Jeremy N. Bailenson. "Effect of Behavioral Realism on Social Interactions Inside Collaborative Virtual Environments". *Presence*. 27(2):163-182, 2020. doi:10.1162/pres_a_00324.

[21] Daniel Holden, Taku Komura, and Jun Saito. "Phase-Functioned Neural Networks for Character Control". *ACM Trans. Graph.* 36(4) article 42, July 2017, doi:10.1145/3072959.3073663. 2017.

[22] HouraiTeahouse. "Houraiteahouse/LZF: Simple C# LZF Compression Library Which Attempts to Minimize Memory Allocations". [online]. https://github.com/HouraiTeahouse/LZF, Last Accessed (1/11/2023).

[23] Oliver Hummel and Colin Atkinson. "The Managed Adapter Pattern: Facilitating Glue Code Generation for Component Reuse". In Stephen H. Edwards and Gregory Kulczycki, editors, "Formal Foundations of Reuse and Domain Engineering", Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 211-224, 2009.

[24] 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.

[25] Yoshihiro Kawano and Tatsuhiro Yonekura. "On a Serverless Networked Virtual Ball Game for Multi-Player". In "2005 International Conference on Cyberworlds (CW'05)", pp. 270-278, 2005. doi:10.1109/CW.2005.68.

[26] Ben Kenwright. "Inverse Kinematics – Cyclic Coordinate Descent (CCD)". *Journal of Graphics Tools*. 16(4):177-217, 2012. doi:10.1080/2165347X.2013.823362.

[27] Milosz Krajewski. "GitHub - MiloszKrajewski/K4os.Compression.LZ4: LZ4/LH4HC Compression for .NET Standard 1.6/2.0 (Formerly Known as LZ4NET)". [online]. https://github.com/MiloszKrajewski/K4os.Compression.LZ4, Last Accessed (1/11/2023).

[28] Catherine Oh Kruzic, David Kruzic, Fernanda Herrera, and Bailenson Jeremy. "Facial Expressions Contribute More than Body Movements to Conversational Outcomes in Avatar-Mediated Virtual Environments". *Scientific Reports (Nature Publisher Group)*. 10(1):1-23, 2020. doi:10.1038/s41598-020-76672-4.

[29] Marc Erich Latoschik, Daniel Roth, Dominik Gall, Jascha Achenbach, Thomas Waltemate, and Mario Botsch. "The Effect of Avatar Realism in Immersive Social Virtual Realities". In "Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology", Association for Computing Machinery, New York, NY, USA, VRST '17. doi:10.1145/3139131.3139156. 2017.

[30] 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", pp. 165-172, 2000. doi:10.1145/344779.344862.

[31] Jean-Luc Lugrin, Maximilian Ertl, Philipp Krop, Richard Klüpfel, Sebastian Stierstorfer, Bianka Weisz, Maximilian Rück, Johann Schmitt, Nina Schmidt, and Marc Erich Latoschik. "Any "Body" There? Avatar Visibility Effects in a Virtual Reality Game". In "2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)", pp. 17-24, 2018. doi:10.1109/VR.2018.8446229.

[32] Erik Marsh, Joshua Dahl, Alireza Kamran Pishhesari, Javad Sattarvand, and Frederick C. Harris. "A Virtual Reality Mining Training Simulator for Proximity Detection". In "20th International Conference on Information Technology: New Generations (ITNG 2023)", Springer International Publishing, Advances in Intelligent Systems and Computing. To Appear. 2023.

[33] Kiran Nasim and Young J. Kim. "Physics-Based Interactive Virtual Grasping". In "Proceedings of HCI Korea", Hanbit Media, Inc., Seoul, KOR, HCIK '16. pp. 114-120, 2016. doi:10.17210/hcik.2016.01.114.

[34] Network Working Group, Internet Engineering Task Force. "RFC1122: Requirements for Internet Hosts-Communication Layers". [online]. `https://www.rfc-editor.org/rfc/rfc1122`, Last Accessed (1/11/2023). 1989.

[35] 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, September 2020.

[36] Open Handset Alliance. "Android Secure & Reliable Mobile Operating System". `https://www.android.com/`, Last Accessed (1/11/23).

[37] Chase Pettit. "GitHub - Chaser324/LZF: Simple C# LZF Compression Library which Attempts to Minimize Memory Allocations". [online]. `https://github.com/Chaser324/LZF`, Last Accessed (1/11/2023).

[38] Ruslan Pyrch. "GitHub - RevenantX/LiteNetLib: Lite Reliable UDP library for Mono and .NET". [online]. `https://github.com/RevenantX/LiteNetLib`, Last Accessed (1/11/2023).

[39] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. "Robust Speech Recognition via Large-Scale Weak Supervision". doi:10.48550/ARXIV.2212.04356. 2022.

[40] Anastasia Rychkova, Alexey Korotkikh, Andrey Mironov, Artem Smolin, Nadezhda Maksimenko, and Mikhail Kurushkin. "Orbital Battleship: A Multiplayer Guessing Game in Immersive Virtual Reality". *Journal of Chemical Education*. 97(11):4184-4188, 2020. doi:10.1021/acs.jchemed.0c00866.

[41] Oleg Stepanischev. "GitHub - oleg-st/ZstdSharp: Port of ZSTD Compression Library to C#". [online]. `https://github.com/oleg-st/ZstdSharp`, Last Accessed (1/11/2023).

[42] Heather A Stoner, Donald L Fisher, and Michael Mollenhauer. "Simulator and Scenario Factors Influencing Simulator Sickness". In Donald L. Fisher, Matthew Rizzo, Jeffrey Caird, and John D. Lee, editors, "Handbook of Driving Simulation for Engineering, Medicine, and Psychology", CRC Press/Taylor & Francis, Boca Raton FL, pp. 220–243. 2011.

[43] Stress Level Zero. "Stress Level Zero". [online]. `https://www.stresslevelzero.com/`, Last Accessed (1/11/2023).

[44] The Khronos Group Inc. "OpenXR Overview - The Khronos Group Inc". [online]. `https://www.khronos.org/openxr/`, Last Accessed (1/11/2023).

[45] Unity Technologies. "About Netcode for GameObjects". `https://docs-multiplayer.unity3d.com/netcode/current/about`, Last Accessed (1/11/2023).

[46] Unity Technologies. "C# Job System". `https://docs.unity3d.com/Manual/JobSystem.html`, Last Accessed (1/11/2023).

[47] Unity Technologies. "Unity Real-Time Development Platform — 3D, 2D VR &; AR Engine". [online]. `https://unity.com/`, Last Accessed (1/11/2023).

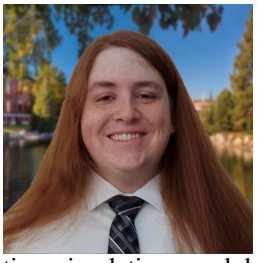
[48] Unity Technologies. "XR Interaction Toolkit: 2.0.4". [online]. `https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/index.html`, Last Accessed (1/11/2023).

[49] vis2k. "Mirror Networking, Documentation". [online]. `https://mirror-networking.gitbook.io/docs/`, Last Accessed (1/11/2023).

[50] VRMADA. "UltimateXR: The XR Framework and Toolkit". [online]. `https://www.ultimatexr.io/`, Last Accessed (1/11/2023).

[51] XieJJ99, James Hopper, and AzureGem. "GitHub - XieJJ99/brotli.net: The .Net Implementation for the Brotli Algorithm". [online]. `https://github.com/XieJJ99/brotli.net`, Last Accessed (1/11/2023).

Joshua Dahl is currently a student at the University of Nevada, Reno. He is pursuing a BS with a major in Computer Science and Engineering and a minor in Mathematics. When he graduates he is planning on pursuing a Ph.D. in Computer Science where he hopes to continue to make contributions to both the fields of computer graphics and programming languages.



Erik Marsh is currently pursuing a MS in Computer Science at the University of Nevada, Reno. He received a BS in Computer Science and Engineering with a minor in Mathematics from the University of Nevada, Reno in 2022. His research interests include input devices, real-time simulations, and data visualization.

Christopher Lewis received his BS and MS degrees in Computer Science and Engineering from the University of Nevada, Reno in 2020 and 2022 respectively. During this time he specialized in Virtual Reality as a researcher in the High Performance Computation and Visualization Lab. Since graduating, he has went on to work as an engineer for Moth + Flame, a company making high quality VR training experiences in both hard and soft skills. He has plans to return to academia for a PhD in a few years.

Frederick C. Harris, Jr. received his BS and MS degrees in Mathematics and Educational Administration from Bob Jones University, Greenville, SC, USA in 1986 and 1988 respectively. He then went on and received his MS and Ph.D. degrees in Computer Science from Clemson University, Clemson, SC, USA in 1991 and 1994 respectively.

He is currently the Associate Dean for Research in the College of Engineering, a Foundation Professor in the Department of Computer Science and Engineering, and the Director of the High Performance Computation and Visualization Lab at the University of Nevada, Reno. Since joining UNR, he has worked on research projects funded by federal agencies (NSF, NASA, DARPA, ONR, DoD) as well as industry. He is also the Nevada State EPSCoR Director and the Project Director for Nevada NSF EPSCoR. He has published more than 300 peer-reviewed journal and conference papers along with several book chapters and has edited or co-edited 14 books. He has had 14 PhD students and 81 MS Thesis students finish under his supervision. His research interests are in parallel computation, simulation, computer graphics, and virtual reality. He is also a Senior Member of the ACM, and a Senior Member of the International Society for Computers and their Applications (ISCA).