Peer-to-peer Syncing and Live Editing of Shared Virtual 3D Spaces: Challenges and Opportunities

Edward Misback
Steven Tanimoto
misback@uw.edu
tanimoto@uw.edu
Paul G. Allen School of Computer Science and Engineering
Seattle, Washington

Abstract

As groundwork for a virtual live programming collaboration environment, we built a peer-to-peer network of devices designed for 2D and 3D interactions that independently host, edit, and sync the state of a virtual space in real time. Both updates driven by a Unity-based peer's game engine and updates driven at 60 Hz by a browser-based peer (running in a reactive JavaScript notebook) were observed. Our system showed significantly lower latency than a popular client-server networking service for Unity, and we observed realistic physics-based interactions for over 100 shared objects using a naïve algorithm that allows a peer to claim temporary ownership of an object's physics. We see peer-to-peer networks like this as increasingly relevant to remote and in-person collaboration on a variety of tasks including learning and programming, and identify opportunities for improvement in the tools involved in their implementation.

ACM Reference Format:
Edward Misback and Steven Tanimoto. 2021. Peer-to-peer Syncing and Live Editing of Shared Virtual 3D Spaces: Challenges and Opportunities. In *Proceedings of LIVE 2021.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Mark Weiser's 1991 article *The Computer for the 21st Century* [1] features a memorable graphic depicting an office where a variety of devices, including conventional workstations, laser printers, file servers, and tablets of various sizes all play a role in the live editing of a shared document. Weiser contrasts this scene of "embodied virtuality" with the development of virtual reality, which he says "attempts to make a world inside the computer" and "excludes … other people not wearing goggles and bodysuits."

Is there a way to get the best of both worlds? While we think virtual reality offers unprecedented opportunities for the design of live programming environments and virtual tools that are easy to open up, explore, and adjust (see for example the projects described by Kao et al. in [2]), we also believe that VR may have potential for connecting to physical spaces and 2D devices with existing programming infrastructure.

To study live collaboration between users and programmers inside and outside of a virtual space, we have set up a fast, flexible communication layer as a first step. In this paper, we discuss our work on the communication layer of a virtual collaboration space that can be modified programmatically at runtime, including observations of its performance in preliminary cross-platform tests and implementation challenges we've faced while setting communications up.

2 Opportunities and objectives

The following scenario is intended to make some of our goals for our system concrete.

2.1 Scenario: pair programming inside and outside of VR

Team members A and B are collaborating remotely in developing a VR game. Member A is using a desktop computer and is coding in a browser. Member B is wearing a VR headset and experiencing a virtual environment in which objects are arranged in space around B's avatar. The arrangement and behaviors of the virtual objects are controlled by the code that A is editing.

1. Over a voice channel, B requests that A create a cube in front of B.
2. A enters code like `CREATE(CUBE, B.FRONT)` in the browser.
3. The code is sent to B's headset and checked for errors that would prevent execution from beginning.
4. A receives feedback that the code can run.
5. The code is executed, showing a cube in front of B.
6. B draws something on the cube.

LIVE 2021, nnnnn
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

7. B asks A to move the cube to a position B is pointing to.
8. A sees B's gesture on a video feed from the virtual space.
9. A switches video to the cube's perspective and navigates to the requested position using the keyboard and mouse.

2.2 Associated goals

Support networking abstractions for live collaboration and programming. When operating in a shared space, users should not have to worry about sending their changes to other users and the space's client-to-client consistency. In the scenario above, member A should think of the CREATE operation as simply operating on the virtual space, not as operating on member A's copy of the space or member B's copy. It's up to the connection layer to make this change in a reasonable time for all clients. In a more advanced system, the space might allow a member to temporarily "fork" the space, test some updates privately, and merge those back into the main space. These abstractions provide users without networking experience with the opportunity to write networked behaviors in a controlled way.

Minimize latency. A collaboration space requires real-time communication among players across a variety of channels (data, audio, and video). Toward this end, connection latency should be minimized. The problem latency poses for remote communication is well-known to anyone who has used a voice over IP (VoIP) application: if audio latency is above 400 ms, normal conversation is difficult, and <100 ms is the industry standard for a "good" network.

When audio latency is less than 50 ms, the opportunity for musical collaboration appears, with reported improvements down to latencies of around 30 ms. [3] [4] [5]

For a 3D application involving embodied users, the data channel is especially important. As latency in the data channel decreases, users are able to respond to each other through gesture and jointly manipulate objects physically in ways that mimic fully in-person interactions. As throughput increases, motions become smoother, reducing the need for interpolation.

In the scenario above, very low latency in the video and data channels is critical for member A's ability to move the cube smoothly to the requested position through teleoperation in step (9).

Widen the variety of clients, and give clients equal access. As stated above, we'd like our space to support connections including standalone VR headsets for 3D interactions, web apps for live-editing and observing the space in 2D, logging servers, and possibly other types of clients like microcontrollers or game engine servers. In the scenario above, both A and B need low-latency access to the space. A communication layer that is truly platform-independent will give the space's design much more flexibility.

Minimize centralized server requirements. Note that this does not refer to non-essential servers that still improve the system's function when present! The main idea behind this goal is that if communications with a server fail due to instability in the system (not unheard of in the context of live programming systems or distributed systems), as long as the server is not totally trusted, there will be a sensible fallback behavior.

There are several other reasons to limit the use of "trusted" servers. Authoritative administrative servers (whether for tracking the official current state in a database server, computing state updates in a game engine server, or managing users in a session server) can be expensive in terms of server maintenance costs and application latency. They also present system usability issues: they make a system harder for non-administrative developers to use (and impossible to modify), harder for administrative developers to deploy, and mean that applications running on the system may have to understand special rules around requesting system updates and obtaining permissions in order to function. Nonetheless, we think a central signalling server is probably necessary for setting up connections in most cases.

For our given scenario, the lack of a central server means it would be easy for the code-checking behavior in step (3) to be modified (by just updating the headset client) without taking down or affecting the whole system.

3 Our system

Here we present the choices we made for our space. This section is relatively technical, but we expect the details of our implementation and design choices may help researchers who are interested in developer choices around low-latency collaborative live-editing systems and collaborative virtual environments.

3.1 Connections: WebRTC

WebRTC is a set of APIs and protocols for enabling real-time communication (RTC) between applications on internet-connected devices. As Blum et al. note in a 2021 article describing the protocol's success [6], WebRTC is an increasingly popular and increasingly standardized option for communication even beyond audio and video.

For our system's communication layer, WebRTC happens to satisfy all of the goals stated above: it is an open, cross-platform standard, and its peer-to-peer nature both minimizes latency and the need for centralized servers. Based on these properties, we view WebRTC as highly relevant to future research on live collaboration, and offer here a short guide for navigating its two most significant stumbling blocks below for those interested.

3.1.1 A brief primer on WebRTC–2 potential stumbling blocks. This section is intended as a guide for those interested in implementing a system like ours, and can be skipped by those not interested in such constructions.

Despite the goal of providing a simple API, establishing a basic WebRTC connection is an involved process.

The first challenge in a peer-to-peer network is resource discovery. WebRTC does not have anything like DNS, where resources can be looked up by a familiar name—instead, it's up to the developer to provide a way for peers to find each other and authenticate. This process is called "signaling." Our system uses a small NodeJS signaling server we wrote that runs on a public AWS instance and accepts WebSocket connections. However, we recommend using a standard, well-supported signaling server like the PeerJS signaling server[7] and keeping application concerns out of the server. We discuss this requirement further in Section 5.

WebRTC's goal of real-time communication presents a second challenge. Once they have been connected through the signaling server, peers must use the connection to negotiate a faster, more direct connection. To accomplish this, they ask a server outside the network[1] to describe what connection methods look possible for the peer[2], then trade this information with a peer (via the signaling server) to choose the best connection path.[3] Recipes for "perfect negotiation" should be used to avoid reinventing the wheel and ensure this process does not deadlock [8].

See Section 5 for further discussion of ways these challenges might be alleviated.

3.2 Unity-based peers

3.2.1 Two platforms. Based on its popularity for VR game development and its reputation as a beginner-friendly, cross-platform development tool, we chose the Unity game engine to represent our virtual space in 3D. Unity's game engine is based on Microsoft's cross-platform .NET framework; as a result, it can run both in the Unity editor on a desktop computer (PC or Mac) and on a number of platforms including Android devices.

We chose the Android-based Oculus Quest (and Quest 2) as our target VR platform despite concerns about the device's openness; the Quest currently represents 75% of the VR headset market [9] and is the only major "standalone headset." It requires no external hardware to run games, but can be connected to a computer via USB or wireless network.

As of April 2021, Unity supports WebRTC on Android devices [10]. However, the WebRTC library requires that the application be compiled with the IL2CPP scripting backend. We discuss this issue in Section 5.

[1]Called a STUN or TURN server; Google offers public STUN servers at e.g. stun:stun.l.google.com:19302, while TURN servers can be hosted privately as optional fallbacks for relaying data if it's impossible to connect directly.
[2]Called "candidates;" examples would be "connect directly to my IP via UDP" or "connect to this port of the NAT I am behind."
[3]A "session description" detailing the type of media that will be sent and how it will be output is also passed to allow optimization for e.g. video.

We used both the Unity editor and various Quest headsets for testing. Although these two platforms were running the same code, we treated them as different when testing based on significant differences in their graphical and central processing power.

3.2.2 Shared physics. Using multiple game engine clients to simulate shared objects leads peers to disagree about the positions of objects. A typical symptom of disagreement is an object "phasing" between two or more positions. While testing, we used a naïve greedy algorithm to avoid disagreements and distribute physics among the Unity peers. Our algorithm (for a peer named Alice) requires tracking the last position of each shared object, but no explicit assignment of object owners:

- If an object shifted from its last position due to the physics engine or an interaction from Alice, Alice turns its physics on (Unity kinematics and gravity) and sends a positional update to other peers.
- If an object shifted due to a positional update from another player, Alice turns off physics for the object and continues receiving updates.
- If Alice is holding an object, she responds to positional updates from other players with a counter-update maintaining its position.

This algorithm worked remarkably well for keeping objects in sync between peers. It survived tests involving piles of objects, objects handed between players, and using held objects to push other objects around. It does not guarantee consistency, though, and occasionally resulted in an object being left suspended until it was interacted with.

3.3 Browser-based peers—a short review of Observable JS as a prototyping platform

We used the Observable JS platform[11] for our browser connections. Observable is an in-browser JavaScript notebook designed by the authors of D3.js for quickly visualizing data; Lau et al. note that some of its relatively unique features include special syntax for displaying and mutating variables, an execute-on-edit behavior, and reactive cell evaluation [12]. We include a short discussion of our experience with the platform below.

3.3.1 Pros. Observable has a long list of great features for application prototyping that really helped us with our work. We are happy to report

- tweaking/adding new functions at runtime without a problem, including cells for controlling the positions of objects in VR.
- using a notebook to walk through establishing a WebRTC connection.
- adding useful visualizations of our data on the fly.

- trivially sharing our peer code among collaborators and friends for live demonstrations, some of whom updated it on the fly to see what would happen.

3.3.2 Cons. We had trouble with

- choosing between un-portable reactive cells that are inspectable vs. portable function blocks that aren't inspectable.
- determining where things went wrong, especially when features like mutables were used.
- choosing between having an IDE and having live editing features.
- bulky cells that can't be folded away altogether; this interacted poorly with the out-of-order presentation of cells, and our peer notebook grew more disorganized over time.

4 System tests and performance

4.1 Latency

The results of latency testing are summarized in Table 1.

4.1.1 Results. Values for browser one-way latency were obtained by taking 40 samples and dividing the average round-trip time by two. Headset measurements were averaged from 15 log entries at different times over an hour, with no major variations noticed.

At the latencies we measured for our system, recalling that a frame rate of 60 fps corresponds to a gap of 16.7 ms, it's possible for nearby peers to agree on the state within a single frame. It also means that a peer can receive feedback on a message within 2 frames (or 4 frames, in the case of a peer at a distance of 1000 miles). This means the teleoperation behavior we describe in step (9) of section 2.1 should be nearly seamless, especially for nearby peers.

4.1.2 Comparison with PUN. Photon Unity Networking (PUN) Object Sync performance was measured by slowing down videos we took for comparing the motion of an objects in the space to a PUN-synced mirror. When shaking the objects up and down rapidly, a PUN-mirrored object noticeably lags a few frames behind the "actual" object. The latency for WebRTC-updated objects in our system is generally less than the time between frames at 60 Hz (16.7 ms), so this effect does not occur.

4.2 Performance

4.2.1 Browser peer driving many objects. For this test, we had an array of objects in a Unity peer follow position updates sent from a browser peer. We tested both updates mapping mouse position onto position in the 3D space and updates from loops running at different frequencies. We tracked the performance of this connection by maintaining a queue of all processed update timestamps from the previous

Peer 1	Peer 2 (distance)	Latency (ms)
Browser	Browser (same room)	8
Headset	Unity editor (same room)	12
Browser	Browser (3 miles)	9
Browser	Browser (1000 miles)	32
Headset	PUN RPC (same room)	55
Headset	PUN Object Sync (same room)	*70

Table 1. Measured latency for a single message. (*Estimated from a video.) Messages were passed from Peer 1 to Peer 2. For discussion of results for PUN, see section 6.

second and reporting its length periodically. We compared the number of successfully-processed updates to the total number sent.

These tests indicated that the Quest could handle around 6000 distinct WebRTC messages per second before messages were dropped and frame rates faltered noticeably.[4] Without any optimizations, this allows for the remote operation of around 100 independent objects at 60 Hz (which we verified). In this test, we also ran 6 other browser peers that received the same updates without trouble.

4.2.2 Unity peer physics sharing. For our final test, we created piles of several hundred small cubes in VR and pushed them around to see whether the system could handle the updates resulting from the cubes bumping against each other.

This task is no problem for a single headset, but in a fully-connected network, the number of update messages that must be issued is multiplied by the number of peers. As expected, for just 3 connected peers, the frame rate did drop noticeably during this test. We can partially solve this problem by sending updates asynchronously, capping the number of updates, and allowing some peers to fall temporarily out of sync, but to preserve guarantees around consistency and to scale past more than a few users, more work is needed.

5 Implementation challenges and future plans

5.1 Simplifying WebRTC

We already described WebRTC's complexity in section 3.3.2. We think signaling and negotiation requirements are significant problems for the widespread use of WebRTC–at present, we would not expect an average developer to set up a WebRTC connection. While excluding these components helps WebRTC cover as many use cases as possible, many users do not care about that flexibility and just want a fast, direct connection, and we think this capability should be a standard.

The main barrier to this standardization is the signaling server. The signaling server only has to hold a small amount

[4]See the other performance test for details on how asynchronous tasks can be used as a partial solution for this issue.

of data linear in the number of active participants, but it does need to be trusted, reliable, and secure (since it holds peers' IP information). Several projects aiming to simplify WebRTC have popped up over time; the most popular one right now is PeerJS[13], which offers a free signaling server and a JavaScript wrapper library. Projects like this are great and we recommend using them, but they are not part of the WebRTC standard. We wonder if a higher-level API with built-in signaling and negotiation (similar to a DNS system) will be available one day.

As a step in this direction, after cleaning up the code, we plan to release a C# implementation of perfect negotiation integrated with the PeerJS server, which will allow Unity users to take advantage of the free server they provide.

5.2 Unity's slow edit-compile-test feedback loop

For the PC development build, the time from making an edit to seeing a result is average at best. On our machine, Unity requires a few seconds to recompile, followed by around 15 seconds to restart the application, and then it's often necessary to put on a headset in order to manually produce a behavior.

For a standalone Quest build, the turnaround time is very bad, on the order of several minutes. On our machine, the IL2CPP build takes around 2 minutes by itself, transferring the application to the headset (using SideQuest) takes another 30 seconds, and opening the application on the headset takes another 30 seconds or so due to putting on the headset, development applications being hidden from normal users, and the Unity engine's boot time. Fortunately, many issues can be tested using the PC development build, but in the case of feature differences between the PC and the Quest (discussed in the next section), it can take a significant amount of time spent logging and reasoning without much feedback to fix errors.

The goal of our system is to allow developers to flexibly live-edit event handlers while Unity continues to run, eliminating both of these long loops in many cases.

5.3 Unity: not completely unified?

We were surprised to find during testing that the feature support of Unity on the PC and Quest were slightly different, and had to make several adjustments to our application so that the same code could run in the Unity editor and on the headsets. The following features were handled correctly on the PC but led to errors at runtime on the Quest (our solutions included in parentheses):

1. Resolving an incomplete SSL certificate chain. (Included the full chain in the certificate.)
2. Native JSON decoding. (Switched to Json.NET, a third-party library.)
3. Property access on native dynamic objects. (Also fixed using Json.NET.)

(We believe (1) resulted from a difference in Android and PC system-level SSL support. (2) and (3) may have been a result of different .NET versions running on the PC and Quest, though this is not certain.)

6 Similar systems

Currently, the most popular free networking package for Unity is Photon PUN[14]. PUN has been used for a number of popular games, including social VR applications like VRChat. As we showed in Table 1 from Section 4, our communication layer achieves significantly lower latencies than PUN, which is server-based. PUN supports sending messages to non-Unity platforms through WebSockets. The use of PUN for free is capped at 20 concurrent users.

The goal for our system is somewhat similar to Mozilla Hubs [15], which offers virtual collaboration spaces that run in the browser using the A-Frame 3D/VR web framework. Running in the browser gives Hubs significant device-level portability. However, Hubs does not support more advanced game engines and is marketed primarily as a social VR tool, so latency is less of a focus than in our system. Additionally, live scripting is not a priority for Hubs.

Neos VR [16] is a highly customizable social "metaverse" game which is more serious about performance and offers an in-game visual programming language. Unlike our system, the game focuses on a fully virtual experience, though it can be connected to other devices via WebSockets.

Google's Stadia gaming platform delivers cloud-hosted video games as a service with WebRTC. We see this as a good sign for the development of teleoperation systems that provide feedback across a network.

7 Conclusion

We considered the performance and implementation challenges posed by a WebRTC connection across three platforms to assess its use in a small distributed system representing a live-programmable virtual world. We think this type of connection might be appropriate for researchers of live collaboration and distributed VR who want a free, fast, and flexible peer-to-peer communication layer with state-of-the-art audio/video streaming support. However, while the speed of the connection does allow more naïve algorithms for distributed physics to succeed for a limited number of players and shared objects, systems intended to support many players, serious physics simulations, or large numbers of shared objects will continue to require a mixture of dedicated game engine and networking servers and more sophisticated state update algorithms. Overall, we are optimistic that systems like ours will be increasingly common for live collaboration applications and may soon enable interesting new modes of collaboration between users inside and outside of VR, and look forward to further development of our system's live programming features now that we have some confidence in its communication layer.

References

[1] Mark Weiser. The computer for the 21st century. *ACM SIGMOBILE mobile computing and communications review*, 3(3):3–11, 1999.

[2] Dominic Kao, Christos Mousas, Alejandra J Magana, D Fox Harrell, Rabindra Ratan, Edward F Melcer, Brett Sherrick, Paul Parsons, and Dmitri A Gusev. Hack.vr: A programming game in virtual reality. *arXiv preprint arXiv:2007.04495*, 2020.

[3] Cristina Rottondi, Chris Chafe, Claudio Allocchio, and Augusto Sarti. An overview on networked music performance technologies. *IEEE Access*, 4:8823–8843, 2016.

[4] Alexander Carôt, Christian Werner, and Timo Fischinger. Towards a comprehensive cognitive analysis of delay-influenced rhythmical interaction. *Proc. ICMC*, 2009.

[5] H. Busse A. Carôt, C. Hoene and C. Kuhr. Results of the fast-music project—five contributions to the domain of distributed music. *IEEE Access*, 8:47925–47951, 2020.

[6] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. Webrtc-realtime communication for the open web platform: What was once a way to bring audio and video to the web has expanded into more use cases we could ever imagine. *Queue*, 19(1):77–93, 2021.

[7] https://github.com/peers/peerjs-server.

[8] Establishing a connection: The webrtc perfect negotiation pattern.

[9] Karn Chauhan. Oculus quest 2 cumulative sales hit record 4.6 mn, Jul 2021.

[10] https://github.com/Unity-Technologies/com.unity.webrtc.

[11] https://observablehq.com/.

[12] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11. IEEE, 2020.

[13] https://github.com/peers/peerjs.

[14] https://www.photonengine.com/en-US/PUN.

[15] https://hubs.mozilla.com/docs/system-overview.html.

[16] https://neos.com/.