UNIVERSITÀ DI TRENTO

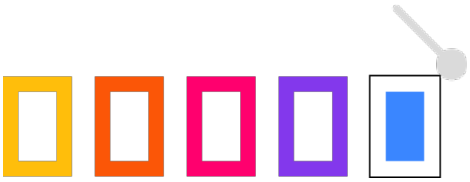
Department of Information Engineering And Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Rythmus

an XR sandbox for the musical metaverse



Supervisor
Prof. Luca Turchet

Co-supervisor
Dr. Alberto Boem

Candidate
Ovidiu Costin Andrioaia

Academic Year 2024/2025

Acknowledgments

First and foremost, I would like to thank Dr. Alberto Boem for his patience and guidance in helping me draft and finalize this thesis, which would not have been possible without his expertise, and for the time spent debugging and testing the application during my internship at the University of Trento. I would also like to thank Prof. Luca Turchet for introducing me to the field of HCI through his course and for giving me this project, as it has allowed me to explore topics I am deeply passionate about.

I would like to thank my girlfriend Anna, for believing in me and for the immense amount of love and support she's given me in the last few year, during long days spent studying for exams and writing this thesis. I am grateful for her presence every day, and it would have been exponentially more difficult to accomplish this goal without her.

I am also grateful to my family, which has supported me emotionally and financially during my university career, and to my friends in Verona, for accompaying me during this journey and for helping me unwind when I came home during the weekends. Special thanks also to my friends in Trento, which I've met through latin dance courses and through the university's gaming community. Lastly, I'd also like to mention my cat, Pizu, for allowing me to use him as a pillow whenever I was particularly stressed, and for being such a nice and well behaved cat.

Contents

Abstract

This thesis presents the design and implementation of Rythmus, a Social Musical XR application developed to facilitate low-latency collaborative music-making for remote users. It addresses the significant disparity in latency requirements between Networked Music Performance (NMP) and existing Social XR platforms, as well as hardware and software limitations of current XR devices for high-quality musical interaction over the internet.

Rythmus combines immersive virtual environments with low-latency audio to support musical creativity and group interaction. The system leverages Ubiq, an open-source Unity library, for avatar management, voice chat, and synchronization. It also integrates the Elk Live service for lossless audio streaming and sound synthesis. Designed following Virtual Reality Musical Instrument (VRMI) principles, Rythmus emphasizes responsive interaction through natural and imaginative musical tools, and aims to foster a strong sense of social presence for remote collaboration.

Chapter 1

Introduction

1.1 The Musical Metaverse

In recent years, digital technologies have changed how music is made, performed, and experienced. One of the emerging ideas in this space is the Musical Metaverse (MM), such as virtual worlds designed specifically for musical activities [19]. This idea combines Extended Reality (XR) and Networked Music Performance (NMP) technologies to create immersive, shared environments where people can interact and make music together, even from different locations. These environments are also known as Collaborative Virtual Environments (CVEs) or Social eXtended Reality (Social XR) [8]. A key feature of social musical XR is creating a sense of connection and presence among participants, which is especially important for collaborative music-making.

1.2 Core Technologies

The musical metaverse is made possible by combining three main technologies:

- **Extended Reality (XR)**: XR refers to a group of technologies used to integrate real and virtual environments to various degrees. These technolgies are VR (Virtual Reality), AR (Augmented Reality), and MR (Mixed Reality). In order to better differentiate them, a Reality-Virtuality Continuum was proposed by Milgram and Kishino [14]. On the virtual end of the spectrum, VR seeks to replace the real environment by creating a computer-generated one which is experienced through sensory stimuli and which can respond to an user's actions [12]. On the other end, AR seeks to only add minor cues to the already existing world, ideally without the user being able to distinguish between what is real and what isn't. In the context of Rythmus, VR allows users to interact through the use of virtual instruments, and with remote other users connected by providing avatars that the users can control with the use of Head-Mounted Displays (HMDs) and associated input peripherals.
- **Networked Music Performance (NMP)**: NMP technologies allow musicians in different locations to play together in real time using an internet connection. These systems require extremely low latency, ideally under 30 milliseconds, to keep the synchronization between musicians [15].
- **Virtual Reality Musical Instruments (VRMIs)**: Based on the definition by Serafin et al. [18] VRMIs are musical instruments that include a simulated 3D visual component rendered through an HMD or stereoscopic immersive system for visualization. They offer new ways to interact musically that aren't possible in the physical world, and offer interactivity, spatial awareness, and sound generation in real-time. While most current VRMIs focus on single-user systems [20], they have been explored in shared and collaborative environments, where users can play together, by combing XR and NMP technologies [1].

1.3 Challenges and Opportunities

The musical metaverse has huge potential. It opens up new ways for musicians to perform, create, and teach. It can connect people across the world and create performances and interactions that aren't possible in the real world. XR technologies let artists rethink how music works by offering new forms of expression and interaction (Fig. 1.1). However, building real-time, collaborative music experiences in XR poses many challenges. As showed by Boem et al., there are several issues that currently limit the possibility of deploying effective instances of the musical metaverse [2].

Figure 1.1: Participant while using the Rythmus application during a live test

The biggest challenge is latency, delays in the system that make it hard for musicians to stay in sync. Musicians generally need latencies under 30 ms, but current XR platforms like VRChat or Rec Room often have delays over 100 ms [7, 9], which is too high to be compatible with NMPs. Latency issues come from both local delays (e.g., motion-to-photon and mouth-to-ear latency) and network delays, which also grow as more users join the system. These problems affect audio, visuals, and the sense of timing needed for group performance.

There are also hardware challenges. Current XR devices (especially standalone ones, such as Meta Quest 3 or Apple Vision Pro) are mostly designed to maximize visual rendering, and not high-quality sound for both input and output. Headsets can be heavy, expensive, and difficult to use, and lack the precision needed for expressive performance. Developers often need to work around these limits to make musical XR systems work well.

Game engines like Unity and Unreal Engine are commonly used to build XR experiences, but don't provide strong audio tools in their standard configuration. Developers must add extra software or develop custom plugins to support advanced audio features, especially for real-time audio synthesis and analysis.

Ultimately, combining existing XR technologies for shared and collaborative interactions with NMPs such as JackTrip [5] or Elk Live is complex and not completely straightforward. Moreover, testing and developing these systems is also time-consuming and resource-heavy, as they often require multiple users and devices for proper evaluation. To address these challenges, particularly the difficulty of achieving effective integration between XR and NMP technologies, Rythmus presents a proof of concept for a unified system designed to bridge the gap between NMP and social musical XR platforms.

1.4 Thesis Goals and Structure

This thesis explores the design process and implementation of Rythmus, a social musical XR application that enables remote users to collaborate in real time. Rythmus combines low-latency audio with immersive virtual environments to support musical creativity and group interaction.

Rythmus was built using the Unity game engine, chosen for its ease of use and comprehensive XR HMD integration. The system also uses Ubiq [10], an open-source Unity library, to handle networking features like synchronization messaging used to keep the game state consistent among peers, avatars and proximity-based voice chat. Furthermore, it uses the Elk Bridge and Elk Live [1] service (Fig. 1.2) for fast audio streaming and sound synthesis. Rythmus follows design principles from VRMIs [18], with focus on reimagining and expanding existing musical tools for VR (Fig. 1.3), and delivering social presence.

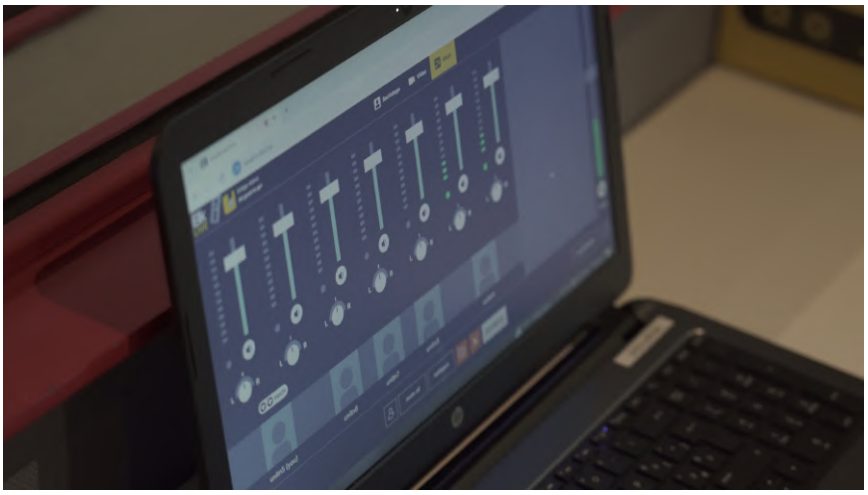


Figure 1.2: Elk Live interface during a live session

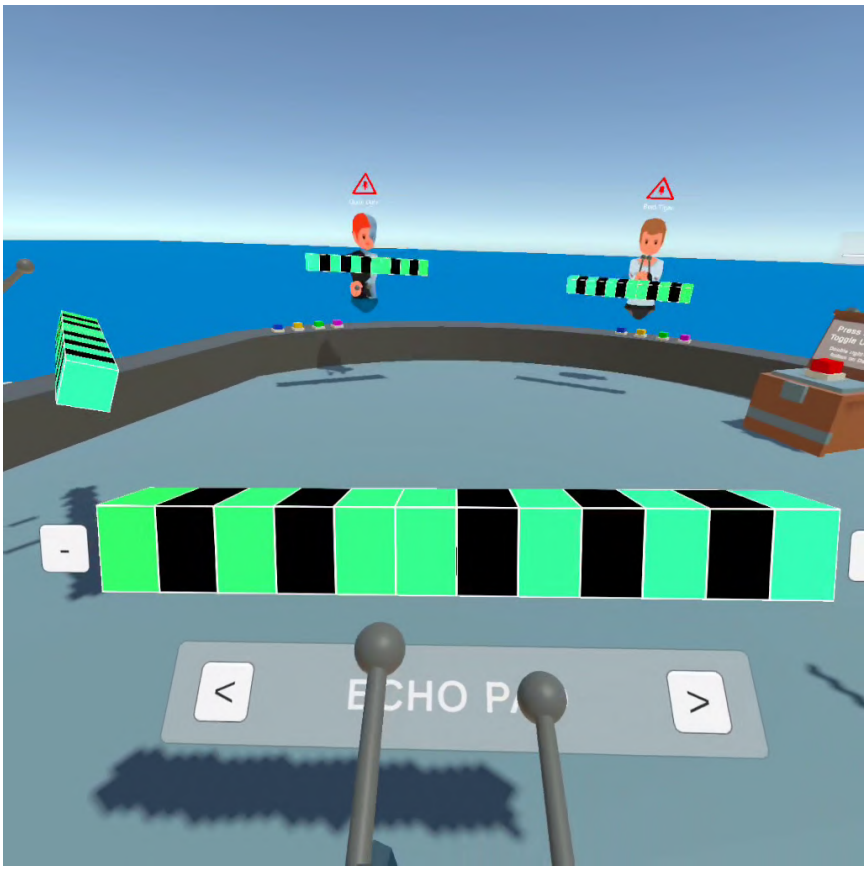


Figure 1.3: An user's point of view during a Rythmus live session

[1] Elk Live: `https://elk.live/`

The structure of this thesis is as follows:

- **Chapter 2: Related Work** - Reviews key literature and technologies behind the musical metaverse and its challenges.
- **Chapter 3: Design and Implementation** - Describes how Rythmus was built, including technical decisions and system architecture.
- **Chapter 4 : Test Results and Analysis** - Presents the results of user testing, system performance, and feedback on the collaborative music experience.

Chapter 2

Related work

2.1 The Musical Metaverse

Recent years have seen growing interest in the convergence of immersive technologies and music, forming what is now referred to as the Musical Metaverse (MM) [19]. The musical metaverse is an emerging area that combines NMP and social XR[1]. In this view, good music performance in the MM depends not only on sound quality but also on how real the virtual presence feels and how well users can coordinate over the network.

NMP, the first component of the MM, enables musicians in different geographical locations to play together in real time over wired or wireless networks [15], and are integral in mitigating latency in musical applications, and providing high quality audio streaming.

One of the most important aspects of music is that it is a social activity [17], whether through performing, improvising, or learning, music happens best when shared. It is imperative therefore that musical XR experiences be designed with a social component in mind, if we are to enable users to fully leverage their functionalities. Collaborative Virtual Environments (CVEs) [8] or social XR [9], which represents the second component of the MM, are immersive environments designed to support collaborative interaction between users in distributed VR landscapes. These experiences should ultimately enable distant musicians and audiences to interact and collaborate in shared, multisensory virtual or augmented environments.

2.2 Issues and Challenges

As outlined by Boem et al. [2], although promising, the MM is still in its early stages, with both hardware and software technologies not yet fully mature: XR devices like HMDs are limited by their weight, cost and steep learning curve. Furthermore, not having been developed with musical usage in mind, the associated peripherals through which musicians interact with VRMIs are deemed not accurate enough, negatively impacting the expressivity of musicians. Lastly, from a software standpoint, XR HMDs focus mostly on gaming or general multimedia applications, leaving musicians and developers to work around limitations in input, audio output, and system responsiveness, due to the lack of adequate integration of audio technologies.

Similarly, game engines commonly used to build XR experiences such as Unity and Unreal Engine have their own trade-offs. Such tools are often chosen for their ease of use and strong XR development tools, but often offer insufficient built-in tools for audio synthesis and analysis.

This makes developing reliable and high-quality musical systems in XR challenging, especially when aiming for real-time collaboration and performance. Achieving this kind of real-time interaction requires

dealing with complex problems like network latency, jitter, and audio synchronization, issues that current consumer technologies still struggle to solve.

Furthemore, most current Musical XR applications focus on single users [20, 18], which limits the chance to explore collaboration and social interaction in immersive environments. While CVEs have been studied in areas like entertainment, education, and workplace training, there has been little research into their use for creative and collaborative music-making [1].

In the Musical Metaverse, latency plays an important role in creating responsive musical experiences by enabling musicians to coordinate and maintain rythm. It is therefore a primary concern to address when developing hardware or software for NMPs, as research has shown that even when musicians perform together in the same physical space, such as a rehearsal room or concert hall, they can only tolerate up to 25–30 ms of latency, which corresponds to a spatial distance of roughly 8 to 10 meters between players [15], with even slightly higher latency values significantly disrupting rhythm, coordination, and overall musical flow.

Jamming with virtual instruments is particularly demanding in terms of latency, as it requires tight synchronization between audio playback and the XR interactions that drive it to maintain a coherent and responsive musical experience. Herein lies the problem, as current social XR implementations are incompatible with the low latencies demanded by NMP requirements, as even the best performers among popular social XR platforms achieve average end-to-end latencies of 100ms or higher [7]. While mismatches between audio and visual cues can be tolerated up to 250 ms in non-musical contexts, this cannot be said for NMPs, where maintaining tighter sync is critical to preserving the sense of shared timing and presence. The consequences of loose synchronization can be observed in an attempt to use video-conferencing tools in the context of NMPs: as the video stream in video-conferencing tools typically suffers higher latency than dedicated audio systems, musicians ended up disregarding the video entirely to maintain rhythm. In contrast, VR offers the possibility to deliver synchronized, spatially accurate visual representations of performers, making it a more suitable platform for immersive collaboration.

The integration of NMP systems into social XR platforms is therefore essential to address these issues. However, doing so is far from trivial: integrating existing NMP technologies, such as JackTrip [5] or Elk Live [1], with current commercial XR systems is quite challenging due to a lack of comprehensive hardware and software infrastructures that can simultaneously handle high-quality audio signal streaming and control data transmission. Most of existing MM applications (e.g., [13, 3, 11, 16]) have relied on local area networks to reduce latency in co-located experiences, but supporting remote interaction adds significant challenges, particularly for consistent networking performance and gesture capture.

This technical complexity is further underlined by concerns expressed by developers and artists actively working in the field. Interviews conducted with practitioners and developers consistently identified network latency and jitter as major barriers to enabling meaningful, real-time musical interaction with XR technologies over distance [1]. These issues hinder expressive performance and reduce the sense of presence and connection that is crucial to successful remote collaboration. As such, innovative solutions need to be developed to address these issues, in order achieve the technological requirements needed by the musical metaverse.

Other identified issues include difficulties in deploying and testing multi-user systems (which require multiple users to be available simultaneously), challenges with networking infrastructure, and the lack of hardware and software capable of simultaneously handling sound control data and real-time audio streaming.

Due to the issues of latency and audio quality, the difficult integration between Social XR and NMP systems, as well as the technical complexity in developing and deploying such applications, current research on multi-user XR systems is still limited, with most studies focusing on only two simultaneously connected

[1] Elk Live: `https://elk.live/`

users (e.g. [13], [4]). Rythmus has been tested with up to four users, demonstrating its capability to support larger collaborative groups, although further validation with more participants is needed.

2.2.1 Outlook on the MM and possible solutions

Despite these challenges, there is growing interest in the MM from both artists and researchers [1]. Many see it as a space to explore new forms of composition and performance that are not possible in the physical world. Others are motivated to create MM systems as research platforms, helping to better understand how musical collaboration works in immersive environments.

Rythmus, developed in Unity, provides its own solution to the problems described in the previous sections by coupling commercial HMDs such as the Meta Quest 3 with the Elk Bridge and Elk Live service, which addresses the inadequate audio processing capabilities of these HMDs without having to integrate potentially resource-intensive NMP software into the application. Furthermore this also confronts the issue of latency, as the Elk Bridge device was developed specifically to provide the low-latency requirements needed by NMPs.

2.3 Virtual Reality Musical Instruments

In computer music, the term Virtual Musical Instruments (VMIs) has been used for many years to describe software-based versions of real instruments or new digital tools for musical expression. These systems mainly focus on how the instrument sounds, often using methods like physical modeling synthesis. In contrast, VRMIs [18] add a visual component by using immersive displays like HMDs. This means that VRMIs combine both sound and visuals in an interactive 3D environment. Unlike earlier VMIs, VRMIs focus on visually immersive experiences, and are part of the broader musical XR domain.

VRMIs are developed to facilitate experiences that cannot be encountered in the real world, by enabling new musical interactions that extend beyond those offered by conventional instruments. They can provide levels of abstraction, immersion, and imagination not possible with traditional musical interfaces. For instance, VRMIs can allow users to change instrument dimensions while playing, or to create musical sounds based on virtual objects that defy physical laws.

Nine design principles have been proposed for VRMIs:

1. Design for Feedback and Mapping: focus on sound, visual, touch, and proprioception in tandem, considering mappings between these modalities. This includes ensuring 3D sound matches visual object location and motion. The lack of tactile feedback, for instance, can diminish performance speed and accuracy.
2. Reduce Latency: all interactions should be smooth with minimum delay, particularly crucial for both sound and visuals. This is vital for timely and synchronized audiovisual feedback.
3. Prevent Cybersickness: minimize symptoms like disorientation and nausea, often caused by sensory conflicts. This requires efficient tracking, high frame rates, and appropriate mapping between virtual and real movements.
4. Make Use of Existing Skills: leverage familiar musical techniques and extend instrument possibilities rather than simply copying existing instruments. The goal is to discover interfaces uniquely suited for the VR medium.
5. Consider Both Natural and "Magical" Interaction: incorporate interactions that conform to real-world constraints (natural) and those that defy them (magical), such as playing instruments beyond normal reach or with non-isomorphic control.
6. Consider Display Ergonomics: address the potential strain and discomfort from wearing HMDs and managing wires, as these devices can be bulky and restrictive.

7. Create a Sense of Presence: foster the sensation of "being there" in the virtual environment, influenced by technological immersion and the illusion of body ownership.
8. Represent the Player's Body: track and map the user's real body to a virtual representation, which is crucial for visual feedback and the sensation of virtual body ownership. The level of realism in this representation should be appropriate for the interaction precision offered by the system.
9. Make the Experience Social: design for shared social experiences, as music is inherently collaborative. This includes collaborative performances and virtual concerts for an audience. While HMDs can be occlusive, other sensory channels like auditory and tactile feedback can facilitate social interaction.

Various VRMIs have been developed. Some examples include:

- Virtual Membrane, Xylophone, and Air Guitar: These instruments explore physics-based models and the ability to dynamically alter parameters beyond physical constraints. Latency was noted as a major problem above 60 ms, and the lack of tactile feedback diminished performance accuracy
- Crosscale: Uses Oculus Rift and Razer Hydra for playing notes as coloured spheres in 3D space, mimicking a keyboard. Latency was not observed, but interaction speed was limited by arm movement replacing nuanced finger control
- ChromaChord: Combines Oculus Rift with Leap Motion for hand tracking to interact with a three-panel visual interface. Fast movements could result in slight latency and cybersickness
- Wedge: A customizable environment for composition and performance, using Leap Motion and Oculus Rift. Occlusion and vibrotactile feedback were missing, potentially causing cybersickness.

While VMIs have a long history and many examples exist in the literature, VRMIs are still quite new and not widely explored. Most VMIs focus on a single function, like controlling one sound effect or one type of synthesis, while VRMIs often use visual interfaces that can handle multiple processes at once [18]. However, VRMIs also don't yet have a long tradition of use or a shared musical repertoire. For them to grow and succeed, users need to develop new practices around performance and composition.[20].

Lastly, although the principles outlined by Serafin et al. provide a valuable framework for the design of VRMIs, they are primarily focused on single-user experiences. To date, relatively limited research has explored the design and implementation of social VRMI systems, and there remains a notable lack of design guidelines for creating VRMIs in the context of the musical metaverse.

Chapter 3

Design and Implementation

3.1 Overview

The application's implementation has to achieve several goals. As such, various objects and functionalities are provided, such as:

- A space for users to interact through their avatars and proximity voice chat. This is achieved through Ubiq's built-in systems.
- A way for players to utilize the capabilities of the Elk Bridge, a networked device which offers, among many features, an integrated synthesizer. This is achieved through a VRMI called the "piano scale".
- An interface to dynamically spawn application-related objects, to allow a variable number of players to utilize the application. This is achieved through the "Spawner" scripts and UI elements.
- A messaging layer, to keep the game state consistent across peers connected to the same session. This is achieved through discrete messaging through networked objects.

To execute this implementation, Rythmus uses various technologies, which work in tandem with one another. The main modules of the application are listed in the section below and illustrated in Fig. 3.1.

1. **Rythmus**, the main VR application built for the Meta Quest series of HMDs, which utilizes the following components:
    - **XR Interaction Toolkit**, a framework for building XR applications, which handles the logic behind XR interactions and user input.
    - **Ubiq Client**, a Unity-based networking library that implements the main functionality and UI interfaces for multiuser interaction within the application.
    - **OSC Transmitter**, which enables virtual instruments to send messages using the Open Sound Control (OSC) protocol [21] to the Elk Bridge to produce audio and music.
2. **Elk Bridge**, a dedicated hardware device for remote low-latency audio collaboration controllable via protocols such as OSC and responsible for producing the audio output of the instruments inside the Rythmus application.
3. **Ubiq Server**, a Node-based rendezvous server that allows users to create and join rooms before transitioning to P2P communication.
4. **Elk Live**, a web application that provides a way for users to establish a direct connection to their respective Elk Bridges and a user interface to control device parameters, like output volume.

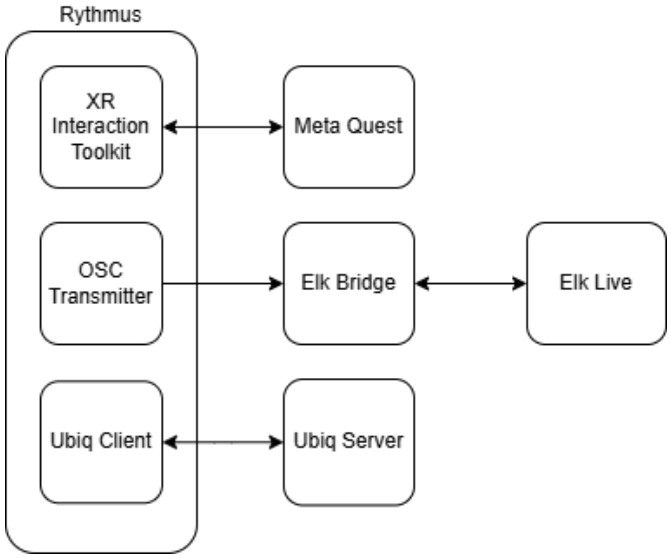


Figure 3.1: Application modules and their interactions

The design of the application's environment and interactable objects has been done in accordance to the VRMI principles described by Serafin et al. [18]. In particular, the application tries to achieve a compromise between Principle 4, which suggests creating familiar VRMIs in order to leverage a musician's existing skill with an instrument, with Principle 5, which recommends creating "magical" interactions, by using the digital medium to ignore real-life constraints such as gravity or physics. These two principles were used when designing the piano scale which, while adopting a familiar xylophone-like design, uses movements impossible to achieve in reality for one of its features, mainly vibrato. Furthermore, none of the objects featured in the applications are influenced by gravity, allowing users to place them wherever they please.

Principle 7 and 8 were also adopted. The former advocates creating a sense of presence by using movement naturally done by users to achieve the same actions when inside the application (for example, moving one's head to better look at something). The latter suggests simulating the user's body to better increase presence, through the sensation of natural body ownership. Both of these principles are embodied by Rythmus' avatar system which, while simple in its design, enables the user to control a virtual avatar with relatively natural movements, to move and communicate inside the application's environment.

Lastly, the core focus of the experience has been recognized in Principle 9, which suggests to make the experience social. As music is achieved through cooperation, a solo experience would contradict the core principles of such activity. The application achieves this by utilizing Ubiq's functionalities to network users together in a cohesive experience, accessible from any location.

3.2 Ubiq

Ubiq [1] is an open-source Unity networking library developed by the Virtual Environments and Computer Graphics (VECG) group at University College London (UCL) [10]. It is designed to support the development of social virtual reality experiences, especially in research, teaching, and prototyping contexts.

Ubiq provides essential networking functionalities such as message passing, room management, rendezvous, object spawning, logging, and voice chat. It has been used extensively in the Rythmus application. In addition, Ubiq includes XR components that support user interaction, such as a UI panel for room and avatar management and an XR interaction setup.

[1] Ubiq: `https://github.com/UCL-VR/ubiq`

These features are built around the NetworkScene, the core component responsible for managing message passing. All service managers, including SpawnManager and LogManager, interact with this central module.

3.2.1 Messaging

Ubiq's messaging system does not follow a typical client-server architecture. Instead, it uses discrete messages exchanged between Unity components that implement specific networked behaviors. These components register with the Ubiq NetworkScene and receive a Network ID derived from their name, position in the hierarchy, and the other networked components attached to the same GameObject.

This deterministic approach means that two objects with identical names and positions in the hierarchy will generate the same ID and can communicate by sending and receiving messages. The NetworkScene automatically routes incoming messages to the correct component and triggers their respective ProcessMessage callbacks.

The Rythmus application uses this mechanism extensively to synchronize the positions and states of various objects.

Objects and Prefabs can be spawned through the SpawnManager, which supports two distinct spawning modes:

- **Peer Scope**: This mode uses a local-first approach, where the object is created locally, even if the user has not joined a room, and is automatically destroyed when the user leaves the room.
- **Room Scope**: This mode allows object spawning only after the user has joined a room, and the object persists even after the user disconnects.

The Rythmus application has made use of both spawning modes, though the latest version exclusively uses Room Scope spawning to simplify implementation.

3.2.2 Logging

Ubiq provides a centralized logging system via the LogManager, enabling components to write and collect JSON-formatted logs efficiently. When a component logs data, the entry is first buffered in memory. Once a peer initiates log collection, it is designated as the log collector. This peer not only records its own logs to disk but also receives and stores the buffered logs from all other peers.

Logs are categorized into one of four labels: Application, Experiment, Debug, or Info, with each category directed to a separate output file. This system allows experimenters and developers to collect, organize, and analyze logs either programmatically or manually, using the embedded UI in the Unity component.

In Rythmus, this functionality was used to log user interactions, environment events, latency measurements, and debug information.

3.2.3 Avatars

In Ubiq, avatars serve as the virtual embodiment of users in the environment. Any Prefab that includes an Avatar component at its root can function as an avatar and is, by default, driven by the XR Interaction Toolkit's XR Origin. The AvatarManager manages avatars during each user's session and leverages the SpawnManager to instantiate a user's avatar at application launch. It also enables users to customize or switch avatars at runtime.

Users interact with each other through VoIP-based proximity chat. In Rythmus, the default stylized Ubiq avatar has been extended with a raycast feature originating from the head, allowing the system to track and log when one user looks at another. Runtime avatar switching has also enabled the implementation of Spectator Mode, a feature allowing the experiment supervisor to become invisible during sessions.

3.2.4 Rendezvous and Rooms

As part of its peer-to-peer architecture, Ubiq includes a rendezvous server that coordinates peer discovery via a room system. A room consists of a list of peers who can exchange messages directly. When a user interacts with the RoomClient to create a room, the server generates a room and returns a three-digit code that others can use to join. Once joined, users are added to the peer list and communicate with one another directly in peer-to-peer mode. This means that after joining a room, message exchange occurs directly between clients, without routing through the server.

The rendezvous server is provided as a Node.js application within Ubiq's GitHub repository, including both the source code and a ready-to-run executable. It can be run locally for development purposes. Alternatively, developers can use the Nexus, a public instance of the rendezvous server maintained by the VECG group at UCL.

3.2.5 XR Components

XR Interaction Setup

Ubiq's XR interaction setup is an extended version of Unity's XR Origin component, based on the XR Interaction Toolkit. It supports core interaction features such as continuous movement, snap-turning, teleportation, UI panel interaction, and object manipulation.

To increase accessibility, the system also includes a Desktop Controller, which mimics XR interactions. It allows users to interact with the application using a mouse and keyboard, in addition to standard XR headset controls.

Social Menu

A social menu is provided to facilitate interaction with Ubiq's avatar and room systems. This menu is an XR-enabled UI panel that can be moved and resized using standard XR controls. Through it, users can create or join rooms, set their display name, and customize their avatar's appearance.

3.3 Elk Bridge and Elk Live Setup

To deliver a musical XR experience with low-latency audio, Rythmus relies on the Elk Bridge and the Elk Live system.

The Elk Bridge is a specialized hardware audio interface developed by Elk, designed to support real-time, low-latency music collaboration over the internet. It works in tandem with Elk Live, a proprietary, browser-based software platform that enables users to connect and play together seamlessly.

Figure 3.2: the Elk Bridge

The Elk Bridge features multiple input and output ports for connectivity with various devices. It also supports SSH access to its underlying Linux-based operating system and can be controlled via the OSC protocol. Additionally, parameters, tracks, and presets can be visualized and modified using a dedicated client application called Sushi-gui. Interaction between Rythmus and the Elk Bridge is organized as follows:

- **The Elk Bridge**, connected to the internet via an Ethernet port and to an output device (such as headphones or speakers) via a 3.5mm or 6.35mm audio jack.
- **The Rythmus application**, which on a computer or Meta Quest headset and is configured with the Elk Bridge's IP address, enabling communication with the Bridge via the OSC protocol.
- **The Elk Live application**, used to create networked sessions between users of the Elk Bridge. It also allows partial control of the Bridge, audio mixing, and an optional video chat.
- **The Sushi-gui application**, a graphical interface that communicates with Sushi, Elk's track-based audio/MIDI engine. It is used to discover the Bridge's capabilities and to monitor the successful execution of the `remote_system` script.
- **The `remote_system` script**, sets up the VST synth used by participants when interacting with instruments inside the application. It also establishes an audio routing system that allows users to hear audio generated by themselves and others.

OSC Messages Sushi Reacts To

By default, Sushi listens on port 24024 for the following OSC commands:

Path	TypeTag	Arguments
/parameter/plugin_name/parameter_name	f	parameter value
/parameter/plugin_name/property_name	s	property value
/bypass/plugin_name	i	bypass state (1 = bypassed, 0 = enabled)
/keyboard_event/track_name	siif	event type (“note_on”, “note_off”, “aftertouch”), channel, note index, norm. value
/keyboard_event/track_name	sif	event type (“modulation”, “pitch_bend”, “aftertouch”), channel, norm. value
/program/plugin_name	i	program id
/engine/set_tempo	f	tempo in beats per minute
/engine/set_time_signature	ii	time signature numerator, time signature denominator
/engine/set_playing_mode	s	“playing” or “stopped”
/engine/set_sync_mode	s	“internal”, “ableton_link” or “midi”
/engine/set_timing_statistics_enabled	i	1 = enabled, 0 = disabled
/engine/reset_timing_statistics	s(s)	reset target (“all”, “track”, “processor”), track name/processor name

Figure 3.3: OSC commands supported by Sushi and their parameters

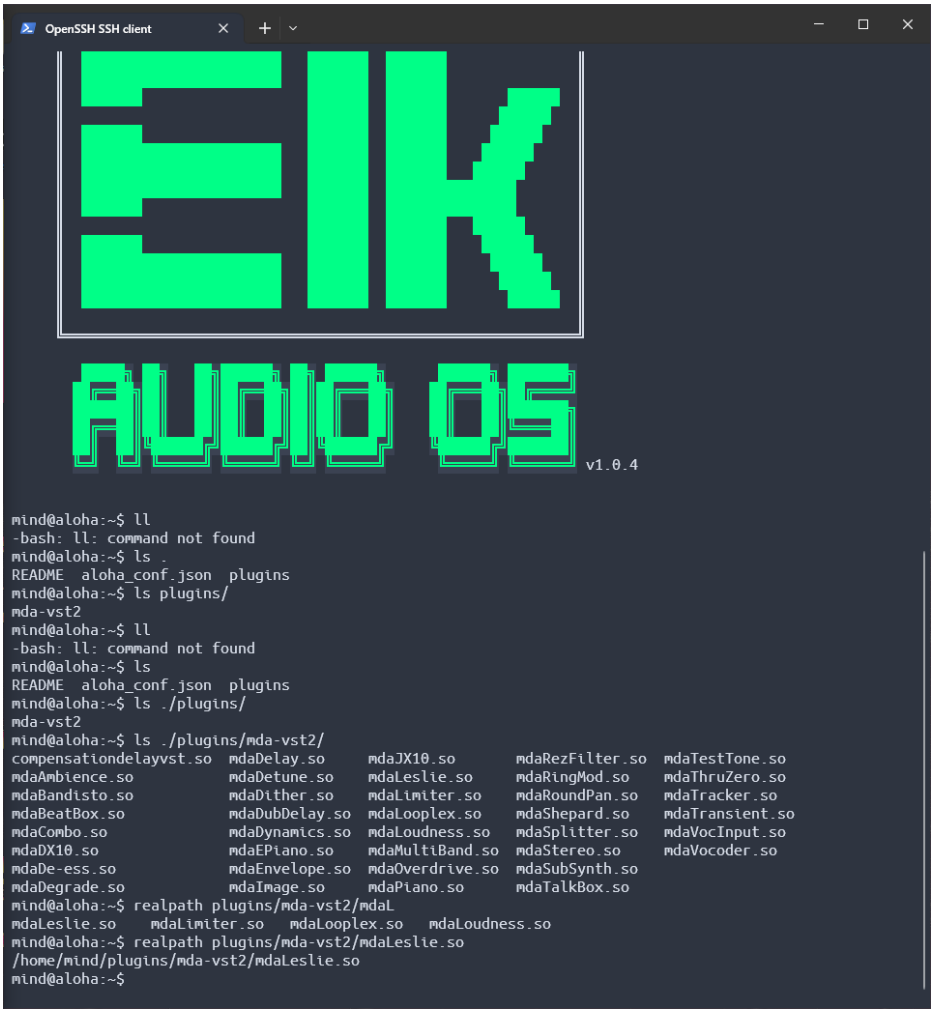


Figure 3.4: plugins available on the Elk Bridge, viewed through a remote terminal connected via the SSH protocol

The `remote_system` script

The `remote_system` Python script interfaces with Elk Audio OS's Sushi engine. It automates the setup of tracks and plugins on the remote Sushi instance running on the Elk Bridge device. The script is executed from the command line, requiring the Elk Bridge's IP address and a .proto file that defines the structure of messages and remote procedure calls.

Upon execution, the script connects to the remote Sushi instance, creates an audio track labeled "own" and sets up a return plugin named "return_own". Return plugins act as output stages: they receive audio forwarded from VST instruments and route it to the appropriate output channels, allowing users to hear the generated audio.

```
own_id = sushi.audio_graph.get_track_id('own')
muter_own_id = sushi.audio_graph.get_processor_id('muter_own')
sushi.audio_graph.create_processor_on_track('return_own', 'sushi.testing.return', None,
↪  PluginType.INTERNAL, own_id, muter_own_id, False)
```

Next, the script creates a new source track and instantiates the MDA JX10 plugin on it.

```
sushi.audio_graph.create_track('x_source', 2)
source_id = sushi.audio_graph.get_track_id('x_source')
sushi.audio_graph.create_processor_on_track('x_mda_jx10', None,
↪  '/home/mind/plugins/mda-vst2/mdaJX10.so', PluginType.VST2X, source_id, None, True)
```

Finally, the plugin's output is routed to the return_own track via a send plugin.

```
sushi.audio_graph.create_processor_on_track('x_cimil_track_send_own', 'sushi.testing.send',
↪  None, PluginType.INTERNAL, source_id, None, True)
sendId = sushi.audio_graph.get_processor_id('x_cimil_track_send_own')
destId = sushi.parameters.get_property_id(sendId, 'destination_name')
sushi.parameters.set_property_value(sendId, destId, "return_own")
```

The user now has a VST plugin that they can control via OSC messages and monitor through their chosen audio output device, typically a pair of headphones.

A similar mechanism is used to exchange audio with other participants. When a remote user connects to the Elk Bridge via Elk Live, they appear in Sushi as a "buddy" track. For each buddy track, the script adds a return plugin, allowing the local user to hear that remote participant. Additionally, the script creates a set of send plugins on the x_source track, one for each buddy, which forwards the user's audio to the other participants.

Each user must run this script after joining an Elk Live session. Once executed, the setup is complete, and users are ready to interact using the Rythmus application.

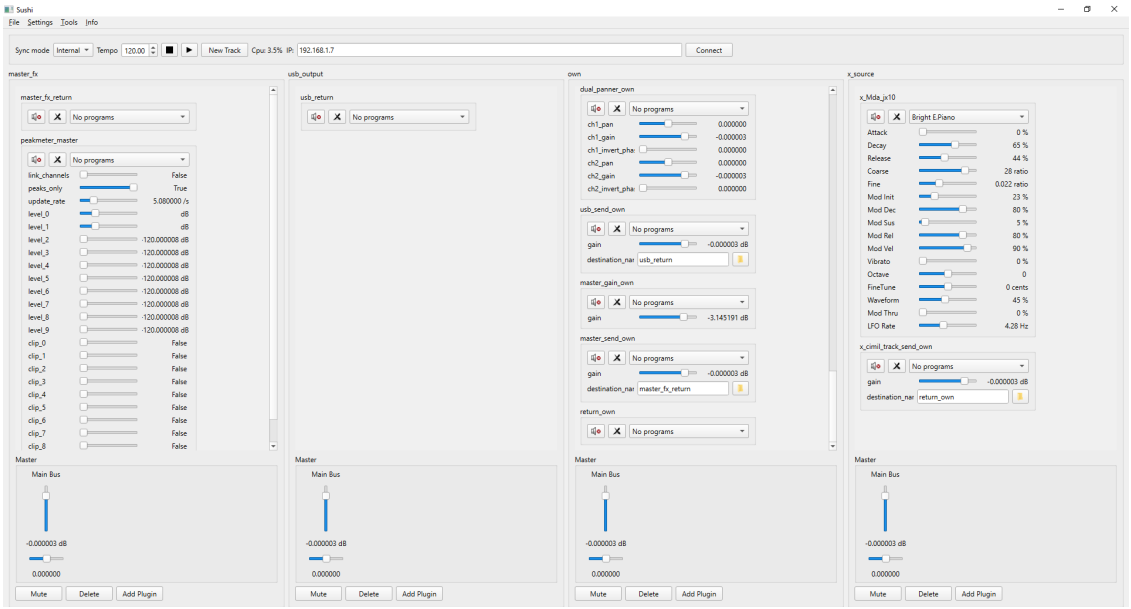


Figure 3.5: The sushi-gui interface. Notice the presence of the x_source track added by the `remote_system` script

3.4 Rythmus

Rythmus is a Unity application built for Meta Quest headsets. It allows one or more users to join a session, communicate, and interact with each other. Furthermore, users can play a virtual instrument that utilizes Elk Bridge, a low latency transmission system with built-in synthesizers. The following sections will explain the application's features, architecture, and codebase.

3.4.1 Interactable Objects

Drumstick

The drumstick is the simplest object in Rythmus, allowing players to interact with music blocks and play music. Visually, it consists of a cylinder with a sphere attached. The sphere acts as a trigger that detects collisions with other objects, mainly music blocks, to activate them.

This object is a kinematic XR Grab Interactable, like all interactable objects in the application. This means it can be grabbed directly or from a distance using the user's ray interactor. Being kinematic, it does not respond to physics forces such as gravity or mass, and follows the player's hand movements instantly. This behavior simplifies position synchronization, which is managed by the SyncedTransform component attached to the object.

Finally, an AttachTransform is positioned on the handle to ensure the drumstick aligns correctly with the user's hand, just like a real drumstick.

Figure 3.6: Drumstick

Music Block

The Music Block is the first iteration of the blocks that will serve as the foundation for the Piano Scale instrument. Although it was removed from the final version of the application, it remains useful for gradually introducing the Piano Scale and Instrument Blocks, which are discussed later in this document.

Unlike the Instrument Block, this block does not utilize Elk Bridge. Instead, it plays one of eight piano notes stored locally as MP3 files via Unity's built-in Audio Source component. The object is spawned through a UI panel, which also allows the user to select the note for that block. The selected note is synchronized across all hosts by sending a message to all remote instances of the block.

When struck by a drumstick, the block's color changes and the selected audio file plays. No explicit network message is sent when the block is hit. Instead, because the block's position is synchronized using the NetworkedTransform component, each remote instance of the drumstick will eventually collide with the remote block, triggering the audio playback for all users. Due to network latency, a variable delay is to be expected before this occurs.

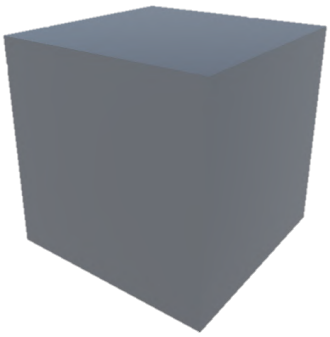


Figure 3.7: An idle Music Block

Instrument Block

The Instrument Block is the building block of the piano scale. Compared to the simpler Music Block, it is more sophisticated in both appearance and behavior. It communicates with Elk Bridge and provides richer visual feedback during user interaction.

Functionally, the block plays a single MIDI note when struck with a drumstick. Users can then engage vibrato, a rapid, oscillating pitch effect, by shaking their drumstick within the block. The intensity of the vibrato is determined by how quickly the drumstick is moved.

Visually, each block is assigned a unique color based on its note. This is achieved by dividing the hue range of the HSV color space into 131 segments and using the first 128, one for each MIDI note. When arranged as a scale, the blocks form a smooth color gradient.

During interaction, the block slightly changes its color when struck. If vibrato is applied, the block emits a white glow along its edges. The thickness of this glow corresponds to the intensity of the vibrato movement.

Lastly, when an object enters the block, it is outlined with a white glow to enhance visibility within the block's interior.

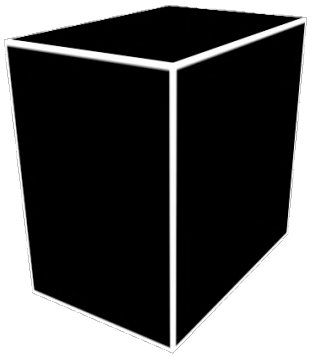


Figure 3.8: An idle Instrument Block in its default, uncolored, configuration

Piano Scale

The Piano Scale consists of twelve Instrument Blocks, each uniquely colored to represent a note. It also includes two UI elements: the Octave Selector, which changes the active octave (and updates the blocks' colors accordingly), and the Preset Selector, which lets the user choose a sound preset from the Elk Bridge.

The Piano Scale communicates with the Elk Bridge via an OSC Transmitter. When an Instrument Block is struck, it sends its note and vibrato values to the Piano Scale, which then forwards them to the Elk Bridge through the OSC Transmitter to trigger audio playback.

Figure 3.9: A piano scale, in its default state after spawning

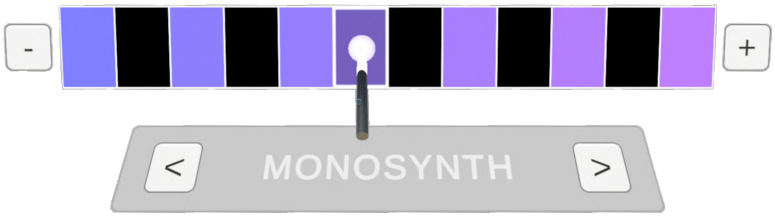


Figure 3.10: A drumstick playing a block on the piano scale. Notice the change in octave and preset.

Instrument-related panels and buttons

To allow users to dynamically spawn and control the drumstick and piano scale, two UI panels were created: the Instrument Spawner panel and the Instrument Gain panel.

The Instrument Spawner panel enables users to spawn either a drumstick or a piano scale. There is no limit to the number of drumsticks a user can create, but only one piano scale is allowed per user. If the

"Spawn Piano" button is pressed more than once, it simply repositions the existing piano in front of the panel instead of spawning a new one.

All instruments are instantiated using Room Scope spawning, which requires the user to be connected to the rendezvous server and to have joined a room. This ensures persistence and synchronization across peers in the session.

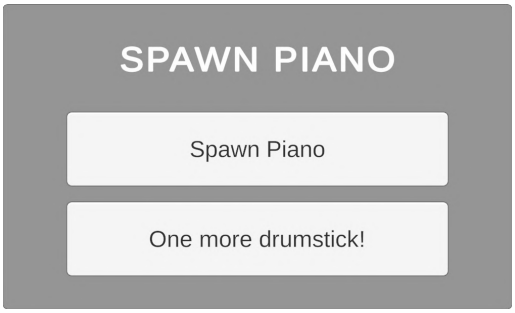


Figure 3.11: The instrument spawner panel

The Instrument Gain panel enables users to adjust the gain of their instruments using a slider. The selected value is transmitted to the Elk Bridge via the panel's OSC Transmitter reference.

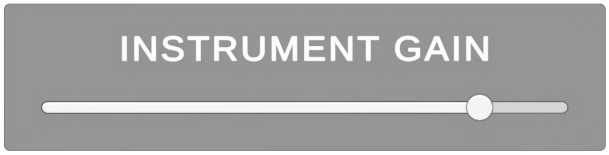


Figure 3.12: The gain controller panel

To prevent interference with the experience, both panels can be hidden while using the application. Visibility is toggled via a red virtual button: XR users can press the button by using their virtual avatar's hand, while Desktop users can click it directly using their mouse.

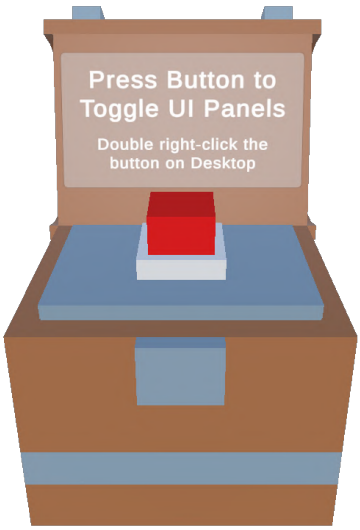


Figure 3.13: The toggle panels button

Social Menu

The social menu is a UI panel through which the user interacts with Ubiq's core features. In this version, the panel has been modified to include a Spectate button, which allows a user, typically an experimenter, to become invisible in order to avoid interfering with an ongoing experiment.

Figure 3.14: The social menu

3.4.2 Components and Behaviors

The objects described above rely on a combination of Unity components and custom scripts to function as intended. The following section outlines the key implementation details that define their behavior.

SyncedTransform

The SyncedTransform is a fundamental component that synchronizes an object's position and rotation across all peers.

On each frame, the object checks whether it has moved by comparing its current position to the previously stored lastPosition. If movement is detected, the velocity is calculated, and a message is sent to the SyncedTransform components on the object's remote instances to update their transforms accordingly.

```
private void Update()
{
    // Check if object has moved.
    if (lastPosition != transform.position)
    {
        Velocity = Vector3.Distance(transform.position, lastPosition) / Time.deltaTime;
        lastPosition = transform.position;

        context.SendJson(new Message()
        {
            Position = transform.position,
            Rotation = transform.rotation,
            Velocity = Velocity
        });
    }
}
```

The remote instances update their current and last recorded positions upon receiving the message via the ProcessMessage callback.

```
public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
{
    Message content = message.FromJson<Message>();
    transform.position = content.Position;
    transform.rotation = content.Rotation;
    Velocity = content.Velocity;

    lastPosition = transform.position;
}
```

Although velocity is not strictly part of the transform, it is also synchronized to reduce inconsistencies caused by packet loss, especially when a user performs vibrato inside a block. This ensures that the glowing white borders of the block remain visually consistent across all peers.

Drumstick

This component, used by the object of the same name, tracks which user is holding a given drumstick. This is necessary to write logs of interactions with blocks (Section 3.4.5). Although collisions between drumsticks and blocks occur on all peers, since the position is networked, only the user holding the drumstick should log the action of playing a block.

To achieve this, the component registers two callbacks on the selectEntered and selectExited events of the drumstick's XRGrabInteractable component, which correspond to "grabbed" and "released" events, respectively.

```
grabInteractable = GetComponent<XRGrabInteractable>();
grabInteractable.selectEntered.AddListener(OnSelectEntered);
grabInteractable.selectExited.AddListener(OnSelectExited);
```

When the drumstick is grabbed, the peer's ID is retrieved from their NetworkScene and stored locally, then broadcast to other peers. When released, the same process occurs, except the ID is set to null instead.

```
private void OnSelectEntered(SelectEnterEventArgs args)
{
    grabbedBy = NetworkScene.Find(this).Id;
    SendUpdateMessage();
}

public void SendUpdateMessage()
{
    context.SendJson(new Message()
    {
        GrabbedBy = this.grabbedBy
    });
}
```

DrumstickSpawner

To support a variable number of users, and thus a variable number of interactable objects, the application dynamically spawns these objects on demand using a class of components known as "spawners." The DrumstickSpawner is the first implementation of this class and is triggered by the "One more drumstick!" button in the Instrument Spawner panel (Section 3.4.1).

Upon application startup, the component locates Ubiq's NetworkSpawnManager within the scene graph. This manager handles local spawning for each peer and registers the object as networked via the NetworkContext component, enabling message exchange through Ubiq's messaging layer.

The DrumstickSpawner registers its OnSpawned method as a callback for the OnSpawned event, which the NetworkSpawnManager fires upon successful spawning.

If logging is enabled via the inspector, the component optionally creates a LogEmitter object to log drumstick spawn events.

```
void Start()
{
    manager = scene.GetComponentInChildren<NetworkSpawnManager>();
    manager.OnSpawned.AddListener(OnSpawned);

    if (logging)
    {
        logEmitter = new ExperimentLogEmitter(this);
    }
}
```

When a user presses the "One more drumstick!" button on the Instrument Spawner panel, the Spawn function is called. If the NetworkSpawnManager is defined, the object is spawned with Room Scope.

```
public void Spawn(int index)
{
    if (!manager) return;
    manager.SpawnWithRoomScope(manager.catalogue.prefabs[index]);

    if (logging)
    {
        logEmitter.Log("Drumstick Spawned");
    }
}
```

Once the object has been spawned, the OnSpawned function is called. This function executes different logic depending on whether the spawned drumstick is local or remote.

If the item is local, the Start function of the SyncedTransform component is executed, and the drumstick's position is set to a predetermined "spawn point". This synchronizes the object's transform across peers. Next, the Drumstick component is initialized, and its grabbedBy attribute is set to the peer that spawned it. This addresses an edge case: since drumsticks and piano scales share the same spawn position, spawning them sequentially could cause the drumstick to unintentionally strike a block even if no user is holding it.

If the item has been spawned by a remote peer, the OnSpawned function has already run locally on the spawning peer (this is guaranteed, as the local peer triggers the function immediately, while remote peers experience latency). Therefore, the drumstick's position on remote peers will be updated via network messages, and remote peers must wait for these updates instead of sending their own. To prevent the remote peer from sending an unnecessary initial position update, the SyncLastPosition function sets lastPosition to the current position. By default, lastPosition is uninitialized, which would otherwise cause the OnUpdate function (Section 3.4.2) to send a position update immediately after spawning.

```
private void OnSpawned(GameObject go, IRoom room, IPeer peer, NetworkSpawnOrigin origin)
{
    if (go.CompareTag("Drumstick") && origin == NetworkSpawnOrigin.Local)
    {

        SyncedTransform st = go.GetComponent<SyncedTransform>();
        st.Start();
        spawnPoint.GetPositionAndRotation(out var pos, out var rot);
```

```
        st.transform.SetPositionAndRotation(pos, rot);

        Drumstick d = go.GetComponent<Drumstick>();
        d.Start();
        d.grabbedBy = NetworkScene.Find(this).Id;
        d.SendUpdateMessage();
    } else if (go.CompareTag("Drumstick") && origin == NetworkSpawnOrigin.Remote)
    {
        SyncedTransform st = go.GetComponent<SyncedTransform>();
        st.Start();
        st.SyncLastPosition();
    }
}
```

MusicBlock

The MusicBlock component, used by the object of the same name, defines the basic behavior of music blocks, a simpler variant of the InstrumentBlock with limited functionality.

The MusicBlock is characterized by three main elements: an idle color, an active color, and a soundIndex. The first two determine the block's visual appearance when struck, while the soundIndex refers to an entry in a list of possible sounds. When struck by a drumstick, the block plays the corresponding sound through its AudioSource component and changes its color accordingly.

```
private void OnTriggerEnter(UnityEngine.Collider other)
{
    if (other.gameObject.transform.CompareTag(interactableTag))
    {
        GetComponent<AudioSource>().PlayOneShot(soundList.sounds[soundIndex]);
        GetComponent<Renderer>().material.color = pressedColor;
    }
}
```

When the drumstick is removed from the block, the color is reset to its idle variant.

```
private void OnTriggerExit(UnityEngine.Collider other)
{
    if (other.gameObject.transform.CompareTag(interactableTag))
        GetComponent<Renderer>().material.color = idleColor;
}
```

There are additional, finer mechanisms related to the MusicBlock, including its own Spawner component; however, these will not be discussed here, as MusicBlocks were primarily used for prototyping and are not included in the final release of the application.

InstrumentBlock

This component, used by the object of the same name, defines the functionality of the blocks that compose the Piano Scale. Like the MusicBlock, its code primarily serves two purposes: playing sounds and providing visual feedback.

As part of an instrument, the InstrumentBlock holds several important references to its parent components. Most notably, it references the BlockGroup component, which drives the Piano Scale and sends note data to the OSC Transmitter, and the OctaveSelector, which determines the octave and thereby affects which note the block plays. Additionally, the component maintains a reference to the object's MeshRenderer to apply visual effects.

```
[SerializeField] private BlockGroup blockGroup;
[SerializeField] private OctaveSelector octaveSelector;
private MeshRenderer meshRenderer;
```

In addition to external references, the InstrumentBlock component includes several parameters that influence its behavior and visual feedback. The zeroVelocity and oneVelocity parameters map a range of input velocities to a normalized float value between zero and one, which defines the vibrato intensity. Velocity values below zeroVelocity do not trigger vibrato, while values above oneVelocity have no additional effect, as the vibrato intensity is already at its maximum.

Related to these are the zeroEdgeThreshold and oneEdgeThreshold parameters, which set the minimum and maximum thickness of the glowing borders around the block.

Finally, the rateOfChange parameter smooths transitions in vibrato intensity. Without this smoothing, even slight changes in velocity would cause immediate fluctuations in vibrato, making it difficult for users to maintain a consistent vibrato effect.

```
[SerializeField] private float rateOfChange;
[SerializeField] private float zeroVelocity;
[SerializeField] private float oneVelocity;
[SerializeField] private float zeroEdgeThreshold;
[SerializeField] private float oneEdgeThreshold;
```

Regarding sound reproduction, the InstrumentBlock holds two key variables. The first, sequenceNumber, defines the block's position on the scale, with zero representing the leftmost block and eleven the rightmost. The second variable, note, is a derived value calculated from the sequenceNumber combined with the current octave, which is determined by the OctaveSelector.

```
public int note {get; private set;}
[SerializeField] private int sequenceNumber;
```

Three variables are defined to reference the object's material and its properties, which are controlled by the IntersectionGlow shader (see Section 3.4.4).

```
private static readonly int BaseColor = Shader.PropertyToID("_Base_Color");
private static readonly int EdgeThreshold = Shader.PropertyToID("_Edge_Threshold");
[SerializeField] private Material cubeMaterial;
```

Lastly, two variables store the calculated baseColor and pressedColor of the block, based on its note, while a boolean tracks whether the block is currently being triggered. This boolean controls whether the coroutine that animates the block's edges continues or stops.

```
private Color baseColor;
private Color pressedColor;
private bool pressed = false;
```

Let's now explore how these variables are used. When spawned, the InstrumentBlock sets its note value and registers to receive updates whenever the piano's octave changes.

```
private void Start()
{
    meshRenderer = GetComponent<MeshRenderer>();

    SetNote();
    octaveSelector.OnOctaveChange.AddListener(SetNote);
}
```

The note is calculated based on the current octave and the block's sequenceNumber.

```
private void SetNote()
{
```

```
    note = octaveSelector.octave * 11 + sequenceNumber;
    CalculateColor();
    ApplyColor(baseColor, zeroEdgeThreshold);
}
```

When the note, and thus the octave, is first set or changed, the colors of the InstrumentBlock must also be recalculated and applied. The first function executed is CalculateColor, which sets the baseColor and pressedColor variables based on the note and its position in the piano scale, as illustrated in Fig. 3.10. If the block's base color is light, the pressed color is a darker shade; if the base color is dark, the pressed color becomes lighter.

```
private void CalculateColor()
{
    var noteColorOffset = 1f / (12f * 11f);

    var h = noteColorOffset * note;
    var v = 1f;
    var vPressed = 0.75f;

    if (sequenceNumber <= 4 && sequenceNumber % 2 == 1 ||
        sequenceNumber > 4 && sequenceNumber % 2 == 0)
    {
        v = 0f;
        vPressed = 0.25f;
    }

    baseColor = Color.HSVToRGB(h, 0.5f, v);
    pressedColor = Color.HSVToRGB(h, 0.5f, vPressed);
}
```

Once the base and pressed colors are calculated, the ApplyColor function sets and applies these colors and the border thickness to the material. Note that in the Start function, the applied color is initially the base color, and the border thickness is set to the zeroThreshold, its minimum value.

```
private void ApplyColor(Color color, float edgeThreshold)
{
    var propertyBlock = new MaterialPropertyBlock();
    propertyBlock.SetColor(BaseColor, color);
    propertyBlock.SetFloat(EdgeThreshold, edgeThreshold);
    meshRenderer.SetPropertyBlock(propertyBlock);
}
```

With this, the base state of the InstrumentBlock is set and ready for interaction. When a user spawns a drumstick and begins interacting with the block, the OnTriggerEnter function is triggered, starting a coroutine that recalculates the block's border thickness each frame. After the coroutine starts, the note is sent to the BlockGroup, the component that manages the piano scale, which then forwards the note to the Elk Bridge for sound playback.

```
private void OnTriggerEnter(UnityEngine.Collider other)
{
    if (other.gameObject.transform.CompareTag(interactableTag))
    {
        pressed = true;
        StartCoroutine(SmoothVibrato(other.gameObject));
        blockGroup.SendNote(true, note);
    }
}
```

The SmoothVibrato coroutine controls the intensity of the vibrato effect, which is visually represented

by the glowing edges of the block. The goal is for the user to modulate this intensity by shaking the drumstick inside the block: the faster the shaking, the higher the vibrato value.

To achieve this, the velocity of the drumstick must be mapped to the vibrato intensity. However, mapping velocity directly to vibrato would cause rapid fluctuations, making it difficult for users to maintain a steady vibrato. Instead, the block maintains a smoothed velocity value that gradually increases towards its maximum when the drumstick moves quickly, and smoothly decreases towards its minimum when the drumstick slows down or stops.

This smoothing is implemented through a rate-limited algorithm that limits how fast the vibrato value can change. It either adjusts the vibrato at a bounded rate towards the current velocity or, if close enough, snaps directly to that velocity, resulting in a fluid and controllable vibrato effect.

Let:

- $v_{\text{mapped}} = \frac{v - v_0}{v_1 - v_0}$ be the velocity, mapped from $[v_0, v_1]$ to $[0, 1]$
- $\Delta v = v_{\text{mapped}} - v_{\text{smooth}}$
- $r =$ rate of change
- $\Delta t =$ time passed since last frame

The update rule for the smoothed velocity is:

$$v_{\text{smooth}}^{(t+1)} = \begin{cases} \text{clamp}\left(v_{\text{smooth}}^{(t)} + r \cdot \Delta t \cdot \text{sign}(\Delta v),\ 0,\ 1\right) & \text{if } |\Delta v| > r \cdot \Delta t \\ \text{clamp}\left(v_{\text{smooth}}^{(t)} + \Delta v,\ 0,\ 1\right) & \text{otherwise} \end{cases}$$

Where clamp ensures that the result stays within the $[0, 1]$ range. The full code that achieves this effect is shown below.

```
private IEnumerator SmoothVibrato(GameObject drumstick)
{
    var drumstickTransform = drumstick.GetComponentInParent<SyncedTransform>();
    var smoothedVelocity = 0f;

    while (pressed)
    {
        var mappedVelocity = Mathf.InverseLerp(zeroVelocity, oneVelocity,
        ↪  drumstickTransform.Velocity);
        var deltaVelocity = mappedVelocity - smoothedVelocity;

        if (Math.Abs(deltaVelocity) > rateOfChange * Time.deltaTime)
        {
            var nextVelocity = smoothedVelocity + rateOfChange * Time.deltaTime *
            ↪  Math.Sign(deltaVelocity);
            smoothedVelocity = Math.Clamp(nextVelocity, 0f, 1f);
        }
        else
        {
            smoothedVelocity = Math.Clamp(smoothedVelocity + deltaVelocity, 0f, 1f);
        }

        var edgeThreshold = Mathf.Lerp(zeroEdgeThreshold, oneEdgeThreshold, smoothedVelocity);
        blockGroup.SetVibrato(smoothedVelocity);
        ApplyColor(pressedColor, edgeThreshold);

        yield return null;
```

```
    }
}
```

BlockGroup

The BlockGroup component implements the functionality needed to streamline communication between the OSC Transmitter and its sub-components: the twelve InstrumentBlocks, the OctaveSwitcher and PresetSelector.

To accomplish this goal, three functions are provided: SendNote, SetVibrato and SetPreset. Their implementation is very similar: each packages a message with different parameters and sends it to the Elk Bridge through the OSC Transmitter, as seen in the following SendNote implementation.

```
public void SendNote(bool play, int note)
{
    if (spectatorMode)
        return;

    var message = new OSCMessage("/keyboard_event/x_source");
    message.AddValue(OSCValue.String(play ? "note_on" : "note_off"));
    message.AddValue(OSCValue.Int(0));
    message.AddValue(OSCValue.Int(note));
    message.AddValue(OSCValue.Float(1.0f));

    transmitter.Send(message);
}
```

The component also features a function to disable transmission of notes and interaction with the UI elements of the OctaveSwitcher and PresetSelector. This is done in order to prevent users from interacting with piano scales different from their own. The function is used by the InstrumentSpawner component when spawning remote Piano Scale objects. The function also disables block interaction logging, if enabled.

```
public void Disable()
{
    spectatorMode = true;

    foreach (BlockInteractionLogger blockLogger in
    ↪  GetComponentsInChildren<BlockInteractionLogger>())
    {
        blockLogger.enabled = false;
    }

    OctaveSelector octaveSelector = GetComponentInChildren<OctaveSelector>();
    octaveSelector.GetComponent<MenuAdapterXRI>().enabled = false;
    octaveSelector.transform.GetChild(0).gameObject.SetActive(false);

    PresetSelector presetSelector = GetComponentInChildren<PresetSelector>();
    presetSelector.GetComponent<MenuAdapterXRI>().enabled = false;
    presetSelector.transform.GetChild(0).gameObject.SetActive(false);
}
```

OctaveSwitcher

The OctaveSwitcher component allows users to change the octave of their piano scale. It is attached to the object of the same name, visually identifiable by the two arrows on either side of the instrument (Fig. 3.9).

Octave changes are handled via Unity Events. When a user presses an arrow, the NextOctave or PreviousOctave function is invoked, updating the octave index. This triggers the OnOctaveChange event, subscribed to by all InstrumentBlock components, prompting them to update accordingly. The component also sends a message to remote peers to ensure the octave and visual state are synchronized across all clients.

```
public void NextOctave()
{
    if (octave >= 10) return;

    octave++;
    context.SendJson(new Message() { Octave = octave });
    OnOctaveChange.Invoke();

    if(logging)
        Log();
}
```

When a remote instance receives the message, it updates its octave value and triggers the OnOctaveChange event. This ensures the visual state of all InstrumentBlock components is synchronized across peers.

```
public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
{
    octave = message.FromJson<Message>().Octave;
    OnOctaveChange.Invoke();
}
```

PresetSelector

The PresetSelector allows users to choose from one of four presets on the Elk Bridge. It is attached to the panel of the same name and is visually identifiable as the UI element located below the Piano Scale (Fig. 3.10).

When a user presses one of the navigation buttons beside the panel, the NextPreset or PreviousPreset function is invoked, updating the current preset index. This index is then used to access the corresponding preset from a list containing preset names and their associated Elk Bridge IDs.

```
public void NextPreset()
{
    presetIndex = (presetIndex + 1) % presets.Count();
    ApplyPreset();

    if (logging)
        Log();
    }
}
```

The ApplyPreset function updates the text displayed on the UI panel and sets the selected preset on the Elk Bridge. It does so by calling the SetPreset function, which sends a message containing the preset ID via the OSC Transmitter.

```
private void ApplyPreset()
{
    panelText.text = presets.Get(presetIndex).Item1;
    blockGroup.SetPreset(presets.Get(presetIndex).Item2);
}
```

3.4.3 Environment

Careful attention was given to creating a calming and immersive environment in which users can interact with the application. To this end, the Waves scene was designed, featuring a tall tower at the end of a long pier.

Figure 3.15: The Waves scene, as seen from the front

Users spawn at the outer edge of the pier, where they are greeted by the Social Menu. From here, they can create or join rooms, customize their display name and avatar, and activate Spectator Mode to observe without participating.

At the top of the tower, accessible via a circular staircase, users will find the interface panels for spawning instruments and drumsticks, as well as for adjusting instrument gain. This area serves as a private stage, and its size has been iteratively refined to comfortably accommodate varying numbers of players.

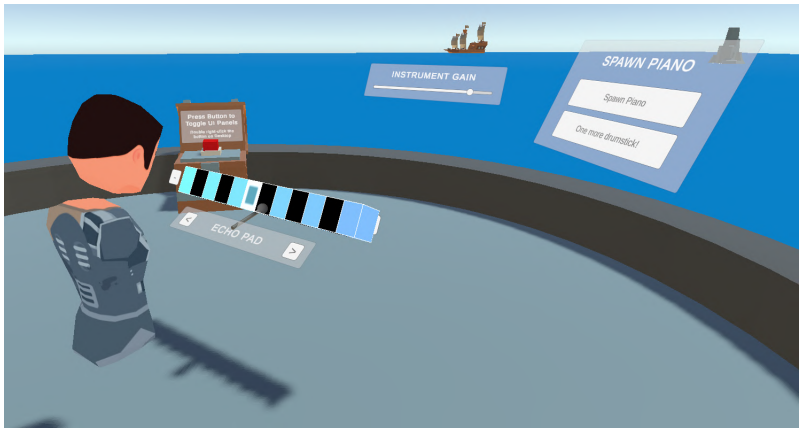


Figure 3.16: An user interacting with a piano scale on top of the tower

The water surface uses a custom material that animates in a sinusoidal pattern and generates a foam-like effect when objects collide with it. Although the water appears close to the player, it is non-interactable. To prevent accidental falls, invisible colliders have been strategically placed around the scene.

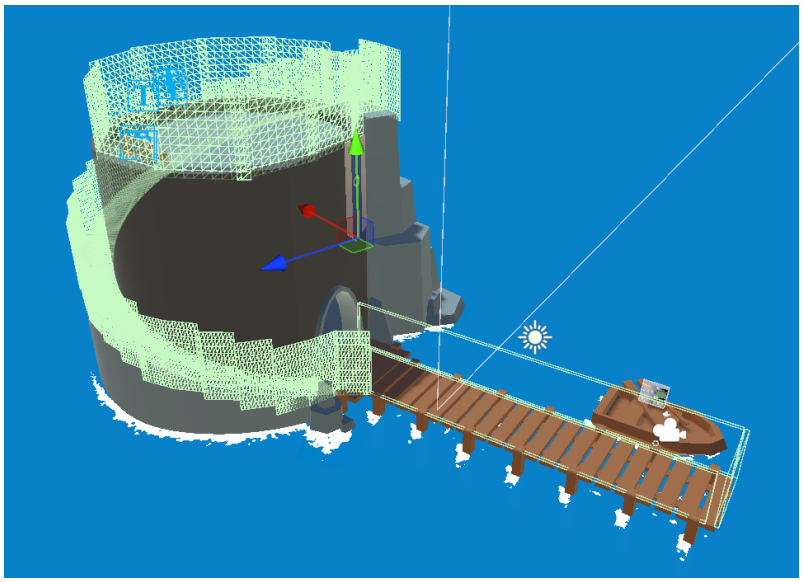


Figure 3.17: A view of the scene's barriers, which limit movement to intended areas of the scene

The circular staircase wrapping around the tower includes an invisible ramp, allowing users to ascend smoothly without abrupt snapping at each step. Additionally, all walkable surfaces are assigned to the "Teleport" layer mask. This enables the use of the Teleport Interactor via the avatar's XR Origin, improving accessibility, especially for users who may experience motion sickness with continuous movement [18].

3.4.4 Shaders

The Rythmus application employs custom shaders in two main areas:

- **Environment effects** to enhance the visual fidelity of waves as they interact with the tower, rocks, and pier.
- **Instrument feedback**, specifically in the piano scale, to provide responsive visual cues during user interaction.

The following section focuses on the DepthIntersection shader, which provides a foundation to two key effects:

- **The IntersectionFoam shader**, responsible for animating waves with a sinusoidal motion and generating foam at points of intersection with objects.
- **The IntersectionGlow shader**, which produces a glowing outline around instrument blocks during vibrato and when intersected by a drumstick.

We then continue by examining the implementation of the IntersectionGlow shader, the simpler of the two.

DepthIntersection

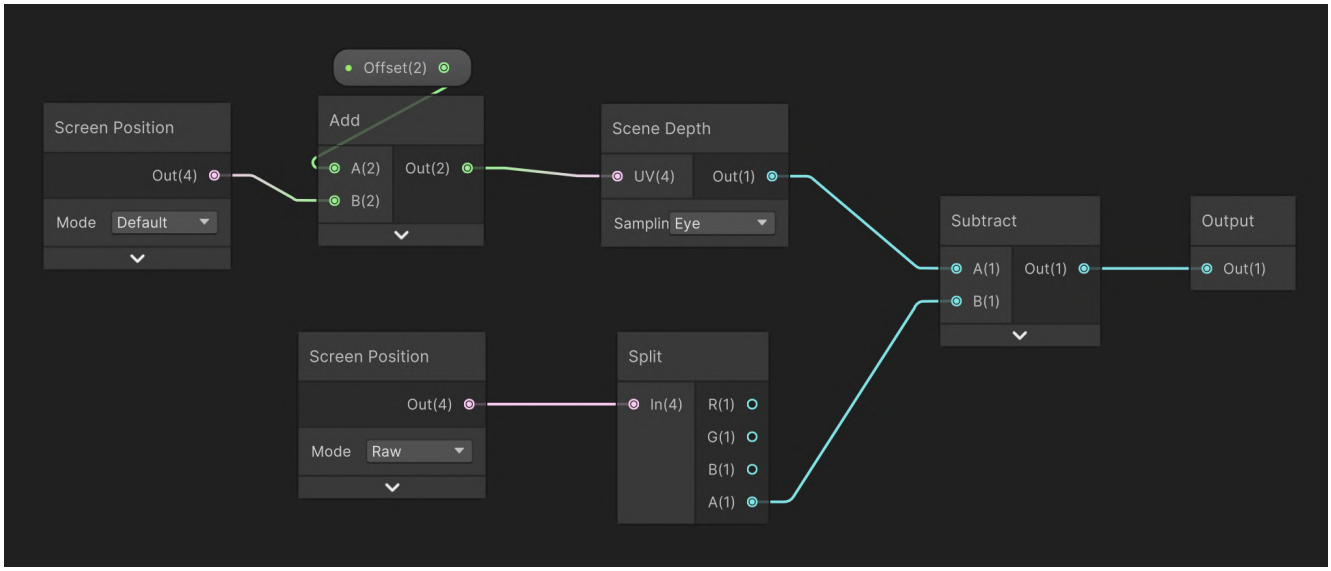


Figure 3.18: The DepthIntersection shader graph

This sub-shader is utilized by both the IntersectionGlow and IntersectionFoam shaders to produce their visual effects. It leverages the depth buffer to calculate the distance between a transparent object's surface and an opaque object located behind it, thereby enabling the creation of outline effects.

A key element in this process is the Scene Depth node, which retrieves the depth of the current pixel relative to the camera. By default, this depth is returned as a non-linear value between 0 and 1, where values closer to 0 represent objects nearer to the camera, and 1 corresponds to the farthest depth defined by the camera's clip space. When the node is set to Eye mode, it instead returns depth in world units. This linear representation is particularly useful for measuring spatial distances in world units directly within the scene.

We use the Scene Depth node to retrieve the distance to the opaque object behind the transparent surface. This works as expected because transparent objects are not written to the depth buffer, and therefore do not interfere with the depth values of opaque geometry behind them.

To determine the distance from the camera to the transparent surface itself, we use the Screen Position node. When set to Raw mode, this node outputs clip space coordinates, despite its name. From this output, we extract the fourth component (w), which represents the distance from the camera to the currently rendered vertex of the transparent object. By subtracting the w value from the depth obtained from the Scene Depth node, we compute the intersection depth between the transparent and opaque surfaces.

An optional offset can be added to the screen position input of the Scene Depth node to enhance the foam visual effect, allowing the foam to "drift" slightly away from the object. The IntersectionFoam shader uses this offset to improve visual fidelity, while the IntersectionGlow shader sets the offset to its default, zero vector.

IntersectionGlow

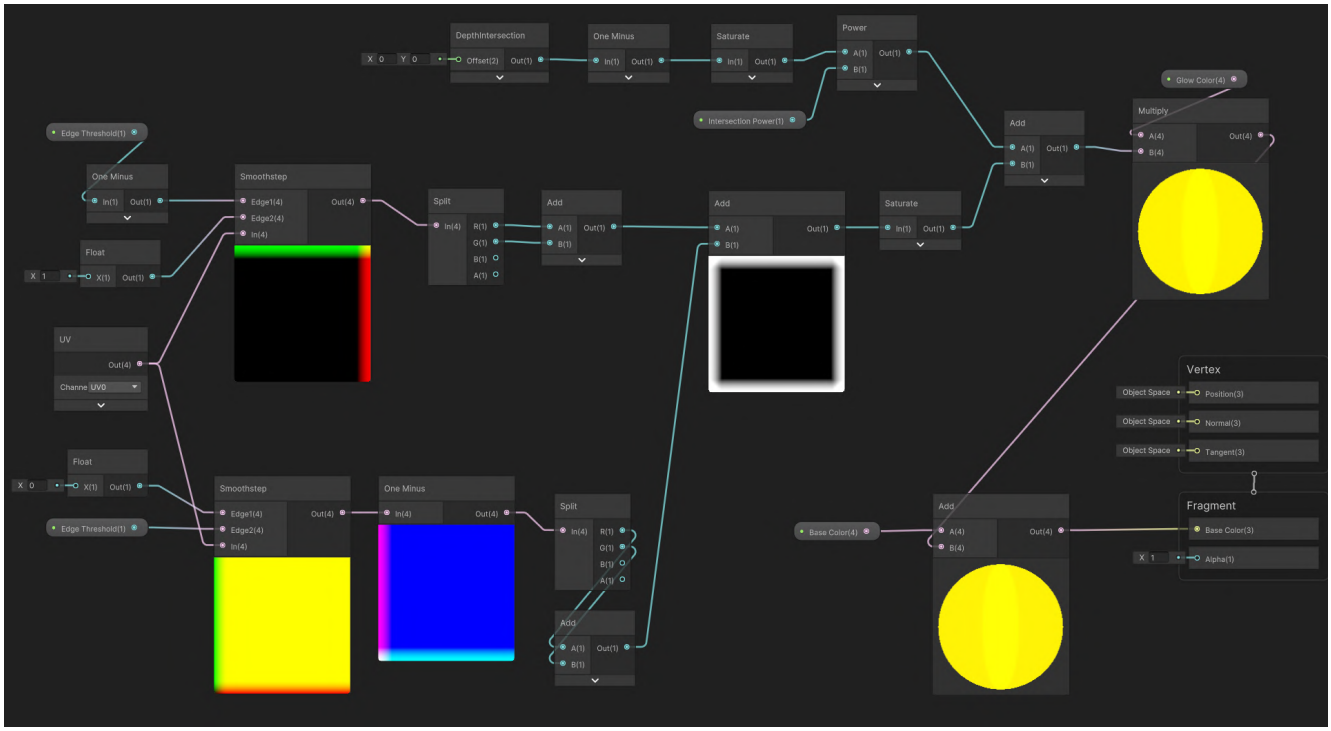


Figure 3.19: The IntersectionGlow shader graph

We begin at the top of the graph shown in Fig. 3.19 to explain the outline effect visible when drumsticks intersect instrument blocks. The output of the DepthIntersection node ranges from $[0, +\infty)$, where 0 indicates complete overlap between the opaque and transparent surfaces. The distance increases as the gap grows, theoretically without bound, but practically limited by the camera's clipping plane.

To convert this raw distance into a normalized float in the range $[0, 1]$, where 1 represents maximum intersection strength, we use a One Minus node to invert the value. A Saturate node is then applied to clamp the output within the valid range, ensuring negative values are set to 0 and values above 1 are capped. To control how quickly the outline fades based on distance, a Power node is introduced with an adjustable exponent.

For the glowing edge effect, the graph uses UV coordinates, which map positions across the surface of the mesh. On a cube, each face is a flat plane, where the bottom-left corner corresponds to $(0, 0)$ and the top-right to $(1, 1)$. To create a square border, we divide the outline into two sections: bottom and left, and top and right.

Focusing first on the bottom and left edges (visible in the lower section of Fig. 3.19), we isolate the $[0, 0.1]$ range of the UV space using a Smoothstep node. This produces a smooth transition within that domain. We then apply a One Minus node to invert the gradient, so that pixels near the edge receive a value closer to 1, fading to 0 away from the edge. The UV output is also split to discard the z and w components, since planar UV coordinates require only the x and y values.

A symmetrical process is performed for the top and right edges. These two gradient values are then summed. To avoid values exceeding 1 (especially in corners where contributions overlap), the combined result is passed through a final Saturate node.

In the final step, this border effect is combined with the depth-based intersection result. The shader then applies colors to the outline and intersected areas and sets a base color for the cube's surface.

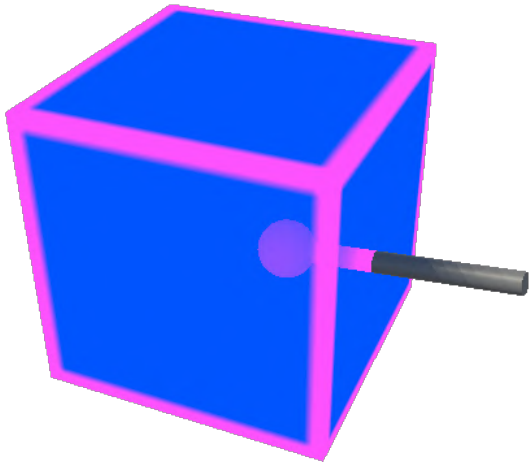


Figure 3.20: The IntersectionGlow shader in action

3.4.5 Logging

To extract valuable insights from experiments, it is essential to log meaningful interaction data during application use. These logs support the analysis of user behavior and assist in identifying potential issues in the application's design or implementation.

Logging is primarily handled through dedicated "logger" components, although it is not limited to them. The main logging components include:

- **BlockInteractionLogger**: logs "Block Entered" and "Block Exited" events. This component is attached to each instrument block object.
- **MainCameraLogEmitter**: periodically logs the user's position and viewing direction as three-dimensional vectors. It also detects when a user is looking at another avatar using a raycast emitted from the avatar's head. This component is attached to the user's head object.
- **LatencyLogger**: uses Ubiq's built-in LatencyMeter to periodically log round-trip latency between peers. When a latency measurement is recorded, a callback within the LatencyLogger writes the result to the log. It is attached to Ubiq's Network Scene object.

To better understand how these components log user actions, we can examine the Update function of the MainCameraLogEmitter. This function runs once per frame and checks whether a raycast, originating from the user's avatar head, intersects with the collider on another user's avatar head.

If this is the first frame in which the raycast detects another user, the function records the peer's network ID, sets an internal boolean flag to true, and logs the beginning of the interaction (i.e., that the user has started looking at someone).

When the user looks away, the raycast will no longer hit the collider. However, because the internal flag remains set to true, the function detects that the user has just stopped looking at the other peer. It then logs the end of the interaction and resets the flag.

```
private void Update()
{
    if (Physics.Raycast(transform.position, transform.forward, out var hit, raycastDistance)
    ↪  &&
        hit.collider.gameObject.CompareTag(headTag))
    {
```

```
        if (lookingAt) return;

        lookingAt = true;
        lookingAtPeerId =
        ↪  hit.collider.GetComponentInParent<Avatar>().Peer.networkId.ToString();
        logEmitter.Log("Looking At", new LookingAtEvent("Started", lookingAtPeerId));

    }
    else if (lookingAt)
    {
        logEmitter.Log("Looking At", new LookingAtEvent("Stopped", lookingAtPeerId));
        lookingAt = false;
    }
}
```

Chapter 4

Test Results and Analysis

4.1 Participants and Setup

A user test was carried out at the SAE Institute in Milan, involving 12 participants divided in three groups, in order to evaluate the system's capabilities and gather feedback on the usability of the virtual instrument provided by Rythmus, the piano scale.

Before starting the test session each participant was instructed to occupy a specific position: three of the four participants were placed within 5 meters of each other in different rooms, while Peer 3 was located on the floor below. With regards to the setup, an Elk Bridge and a laptop connected to Elk Live was prepared for each of the participants plus one for the experimenter, in order to check the session's progress.

Each Elk Bridge was connected to the internet via Ethernet to the building's main network, and connected to the Elk Live to communicate with the other bridges, whereas the Meta Quest headsets running the Rythmus application relied on a local Wi-Fi connection to connect to a local server hosted on the experimenter's laptop, which enabled them to utilize the Rythmus application in the same networked session.

In order to hear the audio produced by the bridges, each participant was instructed to wear a pair of headphones connected to the Elk Bridge. These were worn over the HMDs integrated speakers, allowing the participants to also hear their fellow group members through the application's proximity voice chat.

4.2 Procedure

Each group took part in a separate session, which started with a brief introduction of the system's functionalities by an experimenter. Participants were then accompanied to a different room, and assisted by the experimenter on wearing the HMD and headphones. After a 15 minute guided session to introduce the participants to the system, each engaged in a 30-minute guided improvisation session.

4.3 System Performance Metrics

The results were gathered through the logs produced by Rythmus, which utilized Ubiq's built-in logging layer. Such logs which include latency between peers and interactions with game objects.

Fig. 4.1 displays Ubiq's round-trip latency, which measures the time in milliseconds it takes for a message to reach another peer and back. This measurement is used to determine the system's responsiveness in updating the state of the users and the environment they interact in. This difference in connection type is

especially evident when comparing Ubiq's round-trip latency measurements with those of the Elk Bridges, as shown in Fig. 4.3 and Fig. 4.1.

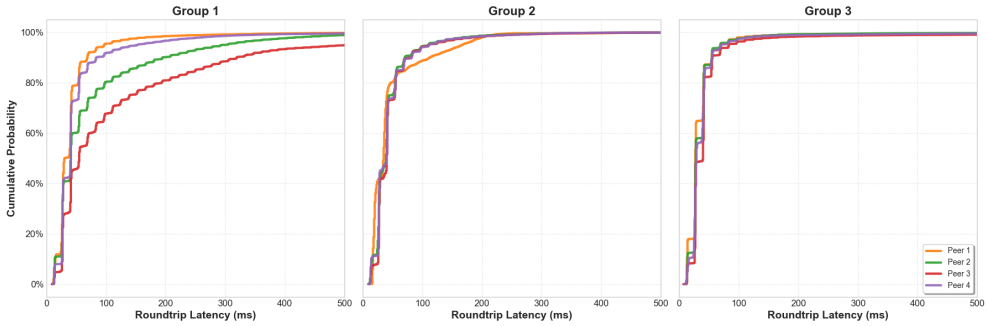


Figure 4.1: A cumulative distribution function of Ubiq's round-trip latency measurements for each peer

Fig. 4.2 measures the amount of time, expressed in milliseconds, elapsed from the last frame computed by the application. This measures the application's responsiveness for an individual user and represents a rough evaluation of the computational stress bearing on the HMD's systems. For reference, the Meta Quest's target of 72 Frames Per Second (FPS) is approximately 13.8ms when expressed in frame time.

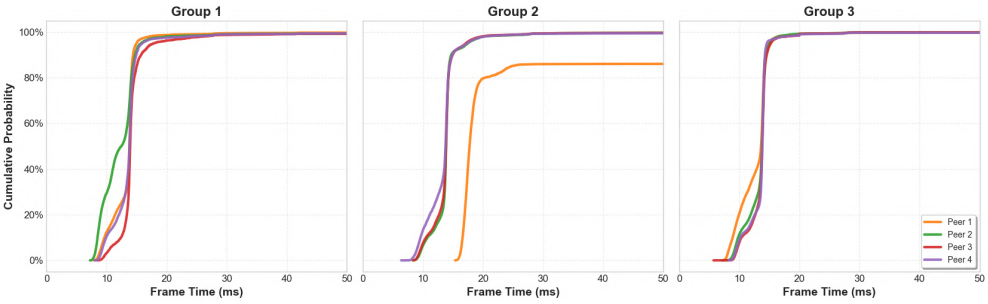


Figure 4.2: A cumulative distribution function of Ubiq's frame time measurements for each peer

Fig. 4.3 and Fig. 4.4 display the Elk Bridge's latency and packet error respectively. Elk's latency measurements, unlike Ubiq's, are not round-trip, meaning that when comparing the two, Ubiq's latency must be halved. The packet error measurement represents the percentage of packets that failed to arrive to a receiving peer.

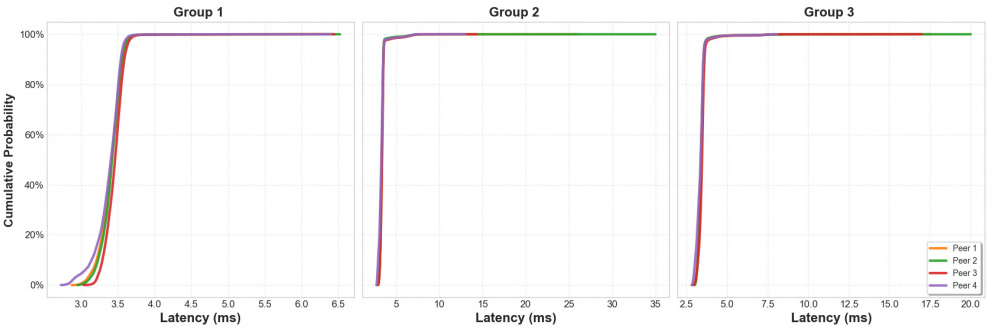


Figure 4.3: A cumulative distribution function of Elk Live's latency measurements for each peer

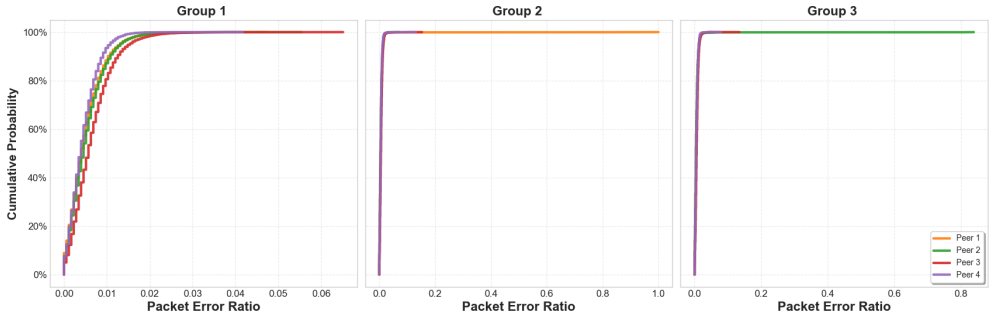


Figure 4.4: A cumulative distribution function of Elk Live's packet error ratio measurements for each peer

Overall, Ubiq reported mean round-trip latencies of 76.63 ms for Group 1, 46.54 ms for Group 2, and 45.53 ms for Group 3. When looking at the Coefficient of Variation (CV) of the three groups, we can notice that Group 3 showed relatively high instability (2.97) when compared to Group 1 (1.515) and Group 2(0.976). Ubiq's round-trip latency and CV results might be linked to the environment conditions of the experiment, as factors such as walls and acoustic insulation can introduce variability in latency when evaluating Wi-Fi connectivity.

In contrast, the Elk Bridge's measurements were consistently lower, averaging between 3.39 and 3.41 ms across all three groups. CV calcualtions for Elk's latency confirms its low latency capabilities, with a CV of 0.043 for Group 1, 0.131 for Group 2 and 0.105 for Group 3.

The packet error rate has also remained stable, at approximately 0.54%-0.58%, with a CV of 1.51 for Group 1, 1.30 for Group 2 and 1.22 for Group 3.

Regarding rendering performance, frame rates generally met the Meta Quest's 72 FPS target. Group 2 was the exception, with a significantly lower average of 46.88 FPS, largely due to anomalous behavior from Peer 1's XR headset. Group 1 and Group 3 averaged 68.85 FPS and 72.46 FPS, respectively. The CV is generally comparable, with a measurement of 1.363 for Group 1, 1.648 for Grupo 2 and 1.058 for Group 3.

4.4 Creative Support Index

Lastly, the Creativity Support Index [6] (7-point Lickert scale) was used for evaluating the effectiveness of the piano scale in supporting the creative process of the test's participants. Mean CSI across all groups was 52.22%, with a Standard Deviation (SD) of 4.54. The CSI values are 48.1% for Group 1, 56.9% for Group 2 and 51.7% for Group 3. These scores, which aggregate around the midpoint, show that the piano scale was deemed to offer some support for the user's creative process, but not extensively.

This score indicates that the piano's design might not have been sufficiently refined, or too simplistic. The selection of presets offered could have also been improved, as the instrument was mainly conceptualized as a proof-of-concept, and thus offered only four presets picked mainly for testing the Elk Bridge's capabilities. Despite these results, it's important to bear in mind that the participants had no prior interaction with XR HMDs. As such, the steep difficulty curve associated with these devices might have negatively interfered in the usability of the piano scale, therefore impacting its CSI evaluation. Therefore, testing the application with users more experienced with XR systems might lead to different results.

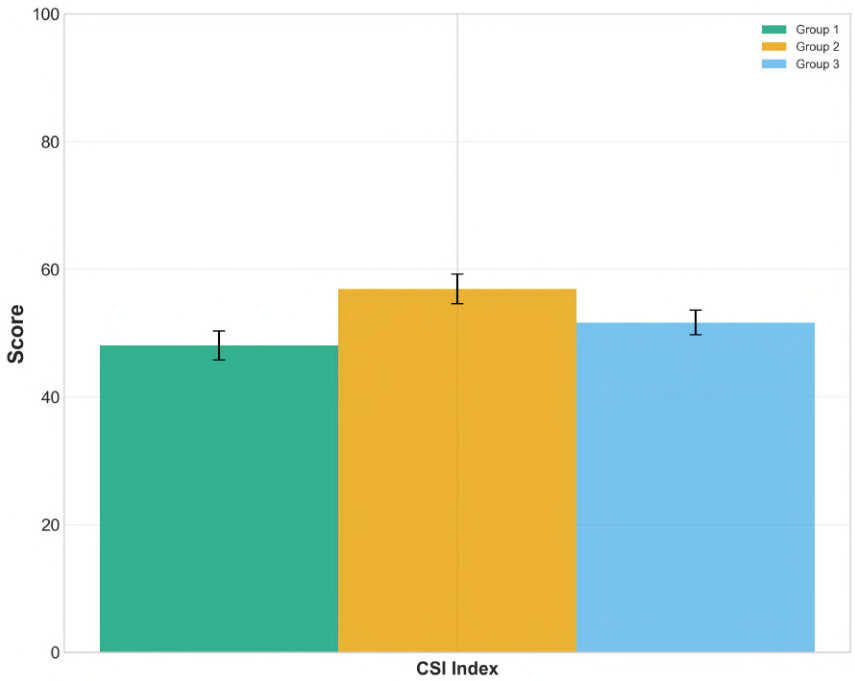


Figure 4.5: A vertical barplot representing the CSI evaluation for each group

Chapter 5

Conclusion and Future Work

Rythmus was developed to address the challenge of enabling low-latency collaborative music-making in Social XR environments by integrating Social XR systems with Networked Music Performance (NMP) technologies. This integration sought to solve two primary technical obstacles:

1. The significant disparity in latency requirements between NMP and existing Social XR platforms.
2. The limited support for real-time, networked audio synthesis in widely used game engines, such as Unity.

While Rythmus offers a functional solution to these challenges, its implementation is not without limitations. The following sections outline key lessons learned during development and identify areas where further improvement is possible.

5.1 Latency Performance Evaluation

As shown in Chapter 4, Test Results and Analysis, Rythmus successfully bridges the latency gap between Elk Bridge, used for Rythmus' NMP layer, and Ubiq's peer-to-peer Social XR networking system, at least within a local network environment. These results demonstrate the feasibility of achieving responsive, real-time interaction when conditions are controlled.

However, the system's performance under wide-area network conditions remains untested. Given that Ubiq operates on a peer-to-peer architecture, selected for its simplicity and low deployment overhead, it is likely that performance will degrade in remote or bandwidth-limited settings. In particular, last-mile connectivity issues may hinder the ability to maintain the low latency necessary for synchronous musical collaboration, especially for users in geographically isolated areas.

A more traditional server-based architecture, featuring regional server replicas distributed across multiple geographic locations, may offer greater potential for scaling the low-latency performance observed during testing. Furthermore, integrating a networking framework optimized specifically for low-latency communication could further enhance system responsiveness and overall performance.

5.2 Elk System Integration

Although the current integration with Elk Live functions as a proof of concept, several aspects of the workflow present challenges for practical use and require further development.

5.2.1 Setup complexity

The setup process is complex and involves multiple manual steps. Users must first connect the Elk Bridge to the internet via Ethernet and initiate a session through the Elk Live platform using a dedicated computer. They must then run a Python script, `remote_system`, referencing the Bridge's IP address to enable transmission of its internal audio channels to audio output devices and other users. Subsequently, this IP address must be entered into the Rythmus application, which must then be compiled and deployed to a Meta Quest HMD.

Due to the absence of automated device discovery within Rythmus, this process must be repeated manually each time the Bridge's IP changes. Once the application is installed on the headset, users must connect headphones directly to the Elk Bridge and wear them over the HMDs integrated speakers, since audio output is handled externally and not processed through the application itself. Finally, the user must launch Rythmus, create a room using Ubiq's interface, and either share their room through the public server browser or send a private access code to allow others to join.

This multi-step process requires a relatively high level of technical knowledge, limiting accessibility for users without such a background. Streamlining this setup process and automating key steps, such as device discovery and audio routing, are essential for improving usability and expanding the reach of the platform.

5.2.2 Audio spatialization

As discussed in the previous section, users are currently required to wear headphones connected directly to the Elk Bridge, positioned over the HMD's integrated speakers. This workaround results from the lack of proper audio integration between Ubiq and the Elk Bridge. One consequence of this separation is the absence of spatialized audio for Rythmus' virtual instruments. Because the Elk Bridge functions as an external audio module relative to the XR environment, the application does not apply spatialization to the instrument audio. Instead, spatialization is limited to XR-native elements, such as proximity-based voice chat. To enhance the overall audio fidelity and immersive quality of the application, it is essential to integrate these two audio sources into a unified and spatially coherent audio pipeline.

5.3 System Capabilities and Functionality

5.3.1 Instrument variety

The current instrument variety in Rythmus is highly limited, with only a single instrument, the piano scale, available in the current implementation. Although the instrument is modular by design, the available preset selection is restricted to just four options, despite the Elk Bridge supporting a wider range of presets. Expanding this selection to include all available presets would significantly enhance the application's expressive potential. Additionally, while drumstick velocity is currently used within the instrument block to control vibrato intensity, this parameter is not utilized when striking the block externally. Incorporating velocity sensitivity in this context could further improve the expressiveness and realism of note articulation.

5.3.2 Object persistence

To simplify development, the current implementation does not include object permanence. State updates are transmitted only as discrete messages to currently connected peers, and no game state is saved or restored. As a result, if a new user joins the session after significant changes have occurred, such as selecting a different preset on the piano scale, they will see the object in its default state rather than its updated configuration. To improve the usability and consistency of the application, it is important to implement object persistence, ensuring that all participants share a synchronized and accurate view of the virtual environment regardless of when they connect.

5.4 Final remarks

Despite its limitations, the development of Rythmus proved to be a highly educational experience, largely due to the multidisciplinary nature of the project. The system integrates a range of diverse fields, including environment design, real-time rendering, game development, Human-Computer Interaction (HCI), networking, and audio synthesis. This combination offered valuable insights into the challenges and opportunities involved in building complex, interactive applications.

Bibliography

[1] Alberto Boem, Matteo Tomasetti, and Luca Turchet. "Harmonizing the Musical Metaverse: unveiling needs, tools, and challenges from experts' point of view". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Ed. by S M Astrid Bin and Courtney N. Reed. Utrecht, Netherlands, Sept. 2024, pp. 206–214. DOI: 10.5281/zenodo.13904834. URL: http://nime.org/proceedings/2024/nime2024_33.pdf.

[2] Alberto Boem, Matteo Tomasetti, and Luca Turchet. "Issues and Challenges of Audio Technologies for the Musical Metaverse". In: *J. Audio Eng. Soc* 73.3 (2025), pp. 94–11.

[3] Alberto Boem and Luca Turchet. "Musical Metaverse Playgrounds: exploring the design of shared virtual sonic experiences on web browsers". In: *2023 4th International Symposium on the Internet of Sounds*. IEEE. 2023, pp. 1–9.

[4] Alberto Boem et al. ""It Takes Two" - Shared and Collaborative Virtual Musical Instruments in the Musical Metaverse". In: *2024 IEEE 5th International Symposium on the Internet of Sounds (IS2)*. 2024, pp. 1–10. DOI: 10.1109/IS262782.2024.10704079.

[5] Juan-Pablo Cáceres and Chris Chafe. "JackTrip: Under the Hood of an Engine for Network Audio". In: *Journal of New Music Research* 39 (Sept. 2010). DOI: 10.1080/09298215.2010.481361.

[6] Erin A. Carroll and Celine Latulipe. "The creativity support index". In: *CHI '09 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '09. Boston, MA, USA: Association for Computing Machinery, 2009, pp. 4009–4014. ISBN: 9781605582474. DOI: 10.1145/1520340.1520609. URL: https://doi.org/10.1145/1520340.1520609.

[7] Ruizhi Cheng et al. "Are we ready for metaverse? a measurement study of social virtual reality platforms". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. IMC '22. Nice, France: Association for Computing Machinery, 2022, pp. 504–518. ISBN: 9781450392594. DOI: 10.1145/3517745.3561417. URL: https://doi.org/10.1145/3517745.3561417.

[8] E F Churchill and D Snowdon. "Collaborative virtual environments: An introductory review of issues and systems". In: *Virtual Reality* 3.1 (Mar. 1998), pp. 3–15.

[9] Carlos Cortés, Pablo Pérez, and Narciso García. "Understanding Latency and QoE in Social XR". In: *IEEE Consumer Electronics Magazine* 13.3 (2024), pp. 61–72. DOI: 10.1109/MCE.2023.3338130.

[10] Sebastian J Friston et al. "Ubiq: A System to Build Flexible Social Virtual Reality Experiences". In: *Proceedings of the 27th ACM Symposium on Virtual Reality Software and Technology*. VRST '21. Osaka, Japan: Association for Computing Machinery, 2021. ISBN: 9781450390927. DOI: 10.1145/3489849.3489871. URL: https://doi.org/10.1145/3489849.3489871.

[11] Rob Hamilton. "Collaborative and competitive futures for virtual reality music and sound". In: *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE. 2019, pp. 1510–1512.

[12] Jason Jerald. *The VR book: Human-centered design for virtual reality*. Morgan & Claypool, 2015.

[13] Liang Men and Nick Bryan-Kinns. "LeMo: Exploring Virtual Space for Collaborative Creativity". In: *Proceedings of the 2019 Conference on Creativity and Cognition*. C&C '19. San Diego, CA, USA: Association for Computing Machinery, 2019, pp. 71–82. ISBN: 9781450359177. DOI: 10.1145/3325480.3325495. URL: https://doi.org/10.1145/3325480.3325495.

[14] Paul Milgram and Fumio Kishino. "A Taxonomy of Mixed Reality Visual Displays". In: *IEICE Trans. Information Systems* vol. E77-D, no. 12 (Dec. 1994), pp. 1321–1329.

[15] Cristina Rottondi et al. "An Overview on Networked Music Performance Technologies". In: *IEEE Access* 4 (2016), pp. 8823–8843. DOI: 10.1109/ACCESS.2016.2628440.

[16] Ruben Schlagowski et al. "Wish you were here: Mental and physiological effects of remote music collaboration in mixed reality". In: *Proceedings of the 2023 CHI conference on human factors in computing systems*. 2023, pp. 1–16.

[17] Alfred Schütz. "Making Music Together: A Study in Social Relationship". In: *Social Research* 18.1 (1951), pp. 76–97. ISSN: 0037783X.

[18] Stefania Serafin et al. "Virtual Reality Musical Instruments: State of the Art, Design Principles, and Future Directions". In: *Computer Music Journal* 40.3 (2016), pp. 22–40. ISSN: 01489267, 15315169. URL: https://www.jstor.org/stable/26777007 (visited on 06/26/2025).

[19] Luca Turchet. "Musical Metaverse: vision, opportunities, and challenges". In: *Personal and Ubiquitous Computing* 27.5 (Jan. 2023), pp. 1811–1827. URL: https://doi.org/10.1007/s00779-023-01708-1.

[20] Luca Turchet, Rob Hamilton, and Anil Çamci. "Music in Extended Realities". In: *IEEE Access* 9 (2021), pp. 15810–15832. DOI: 10.1109/ACCESS.2021.3052931.

[21] Matthew Wright. "Open Sound Control: an enabling technology for musical networking". In: *Organised Sound* 10.3 (2005), pp. 193–200.