CSOUND VS. CHUCK: SOUND GENERATION FOR XR MULTI-AGENT AUDIO SYSTEMS IN THE META QUEST 3 USING THE UNITY GAME ENGINE

Pedro LUCAS (pedroplu@ifi.uio.no) [1], **Stefano FASCIANI** (stefano.fasciani@imv.uio.no) [2], and **Kyrre GLETTE** (kyrrehg@ifi.uio.no) [3]

[1, 3] *RITMO, Department of Informatics*, **University of Oslo**, Oslo, Norway
[2] *Department of Musicology*, **University of Oslo**, Oslo, Norway

ABSTRACT

Extended Reality (XR) technologies, particularly headsets like the *Meta Quest 3*, are revolutionizing the field of immersive sound and music applications by offering new depths of user experience. As such, the *Unity* game engine emerges as a preferred platform for building such auditory environments. As part of its capabilities, *Unity* allows the programming of sound generation through a low-level digital signal processing API, which requires specialized knowledge and significant effort for development. However, wrappers that integrate *Unity* with programming languages for sound synthesis can facilitate the implementation of this task. In this work, we focus on applications for the *Meta Quest 3* involving multiple spatialized audio sources; such applications can be framed as XR multi-agent audio systems. We consider two wrappers, *CsoundUnity* and *Chunity*, featuring *Csound* and *ChucK* programming languages. We test and analyze these wrappers in a minimal XR application, varying the number of audio sources to measure the performance of both tools in two device environments: the development machine and the *Meta Quest 3*. We found that *CsoundUnity* performs better in the headset, but *Chunity* performs better in the development machine. We discuss the advantages, limitations, and computational issues found on both wrappers, as well as the criteria for choosing them to develop XR multi-agent audio applications in *Unity*.

1. INTRODUCTION

Currently, *Extended Reality (XR)* technologies are emerging for different aspects of society as part of a novel paradigm. In the field of sound and music, XR technologies offer new possibilities for musicians and audiences, in terms of audio-visual and interactive experiences [1].

The development of XR applications requires platforms capable of covering a range of features to achieve such immersive experiences. Today, two platforms are broadly used to develop these systems: *Unreal* [1] and *Unity* [2]. Both are game engines widely used in the industry. Nevertheless, although they offer an extensive set of tools to develop full XR experiences, their audio capabilities are still insufficient for specific XR sound and music systems [2]. In the context of sound synthesis and analysis, *Unreal* provides a DSP toolset called *MetaSounds* [3]. On the other hand, *Unity* lacks a similar built-in module and requires third-party add-ons. Despite this, Unity is still preferred due to its high compatibility with the XR ecosystem [2].

An option to achieve sound generation in *Unity* is using audio programming languages interfaced through OSC messages [3], which allows the use of widely known languages such as *Max 8*, *Pure Data*, *Faust*, *Csound*, *ChucK*, or others that support OSC connections. However, it implies playing the sound output outside the Unity application, and if we want to have built-in sound generation within the same system, we need to use wrappers that allow direct interfacing with the engine.

In this work, we focus on XR applications developed in Unity for the recently released *Meta Quest 3* XR headset. As the *Meta Quest 3* is based on the Android operative system, the only wrappers that are currently supported for this OS are *CsoundUnity* and *Chunity* interfacing *Csound* and *ChucK* respectively.

Our research aims to investigate whether both wrappers function correctly in Unity when we have multiple spatialized instances of audio sources, which are playing patches developed in these programming languages for an XR application in the *Meta Quest 3*. This scenario often occurs in XR multi-agent audio systems, when several audio sources are present in the same environment. Our study also examines how these wrappers are integrated into the game engine. Furthermore, we analyze their performance in terms of scalability, i.e., their ability to handle an increasing or decreasing number of audio sources, and identify any potential issues in terms of resource utilization.

An additional contribution to this paper is a public GitHub repository [4] that can be used as a base for the development of XR audio systems, and where further testing can be applied.

This paper is organized as follows: Section 2 refers to multi-agent audio systems and the specific technologies that we are using in this work, section 3 describes the

[1] https://www.unrealengine.com
[2] https://unity.com/

[3] https://docs.unrealengine.com/5.0/en-US/metasounds-in-unreal-engine/
[4] https://github.com/pedro-lucas-bravo/sound_engines

methodology followed in this study, section 4 presents the results and discussion regarding the integration of these wrappers in Unity and the experiments performed over them, section 5 lists recommendations based on the obtained results, and finally in section 6 we provide conclusions and future work.

2. XR MULTI-AGENT AUDIO SYSTEMS

Multi-agent systems are systems that comprise multiple elements called agents. An agent is a computer system that operates within a specific environment and can act autonomously to achieve its objectives [4]. In interactive situations, we can also include humans as additional agents who interact with these artificial agents. On the other hand, systems where sound and music characteristics are emphasized in virtual or physical-virtual immersive environments can be framed in the field of *Music in Extended Realities (Musical XR)*, which is a multidisciplinary area of research focused on XR technologies and experiences with diverse types of projects such as virtual musical instruments, immersive concert experiences, generative musical systems and gamified musical environments [1].

The combination of these two approaches can lead to systems where multiple entities can participate autonomously in sound and music activities or include external interaction of human users as for human-swarm interactive music systems [5] where collective behaviours emerge from the multiple interactions among artificial and living agents. Works as in [6] and [7] can be considered XR multi-agent audio systems that require the integration of several technologies to achieve immersiveness. In some cases, complex integrations are needed due to certain limitations; for instance, in [7], where Unity is used, the sound is generated externally in *Max 8* and therefore requires additional effort for system communication. However, generating sound inside the application would reduce the complexity of communication, and allow us to use the built-in capabilities of the XR devices for spatial audio to provide the sensation of location for the multiple audio sources in an XR scene.

This paper contributes by providing criteria for including sound generation in the latest XR technologies, which we describe in the rest of this section. The focus of this paper is on the programming languages suitable for developing XR multi-agent audio systems using these technologies.

2.1 The *Meta Quest 3*

Meta Quest 3[5] is an XR (VR + MR) headset (shown in Fig. 1) released by *Meta* in October 2023. Its technical specifications are: *Quest system software*, a 64-bit operative system based on Android source code; processor Qualcomm Snapdragon XR2 Gen 2; 8 GB LPDDR5 RAM memory, graphics processor Adreno 740, and 128 GB or 512 GB storage (depending on the version).

Additionally, the device provides spatial audio capabilities through its 2 built-in speakers, which, together with the rest of its features, allows for building immersive sound and music systems contained in one piece of hardware.

[5] `https://www.meta.com/quest/quest-3/`

Figure 1: *Meta Quest 3* XR headset.

Our objective is to utilize this device also for sound generation, and here we compare different options for its audio programming through Unity.

2.2 The *Unity* Game Engine

Unity[2] is a general-purpose game engine widely used for building real-time systems across several platforms. For our work, we consider the Android platform since the *Meta Quest 3* works under that OS.

In terms of sound generation, Unity offers access to the audio buffer that runs in the audio thread through the `OnAudioFilterRead`[6] function, where sound data is processed independently from other parts of an application in development. In that sense, it is possible to implement complex audio processing or synthesis using the `C#` programming language directly or building custom plug-ins in `C` or `C++`[7]. Such low-level programming can require considerable development effort compared to other high-level programming environments that are designed for interactive sound and media.

For that reason, integrating Unity with a programming language for procedural audio would accelerate the development process and allow a faster exploration of sonic possibilities within the game engine.

2.3 *Csound* and *ChucK* for Unity: *CsoundUnity* and *Chunity*

Csound and *ChucK* are programming languages designed for digital sound synthesis and processing. *Csound* operates as a compiler; hence, the code must be ready beforehand to be compiled and then run to produce sound [8]. *ChucK* is an on-the-fly programming language, which

[6] `https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnAudioFilterRead.html`

[7] `https://docs.unity3d.com/Manual/AudioMixerNativeAudioPlugin.html`

[8] `https://docs.unity3d.com/Manual/upm-ui.html`

[9] `https://assetstore.unity.com/`

Feature	*CsoundUnity*	*Chunity*
Unity project set up	It uses the *Unity Package Manager* [8] and does not become part of the main project files.	It is downloaded from the *Unity Asset Store* [9] and becomes part of the main project files.
Instantiation	A Unity `GameObject` with an `AudioSource` component attached needs a script called `CsoundUnity` in order to have an instance of *Csound* running. Several instances can be created in the same scene. The destruction of an instance is automatic when the `GameObject` is destroyed.	A Unity `GameObject` with an `AudioSource` component attached needs a script called `ChuckSubInstance`. Additionally, the scene needs to have one `GameObject` with the scripts `TheChuck` and `ChuckMainInstance`, then several instances of `ChuckSubInstance` can be used. The destruction of a `ChuckSubInstance` is explicit when its `GameObject` is destroyed.
Patch Integration	It supports attaching files to the `CsoundUnity` component as well as direct *Csound* code in Unity `C#` scripts.	Direct *ChucK* code can be used through Unity `C#` scripts. Files are supported, but they need to be explicitly managed using the *Chunity* API.
Parameters Access	*Csound* control-rate variables and score events can be manage through `C#` scripts in Unity. Additionally, it is possible to use editor features with graphical controls in the `CsoundUnity` component. It is not possible to communicate from *Csound* to Unity.	*ChucK* control-rate variables, as well as signal and broadcast events, can be managed through `C#` scripts in Unity. It is possible to communicate from *ChucK* to Unity using these types of events.
Language support	Both wrappers support all function generators (opcodes and UGens) of recent releases of *Csound* and *ChucK*; however, it is not ensured that every function generator will work correctly when using on different platforms.	

Table 1: Features for *CsoundUnity* and *Chunity* wrappers.

means that, unlike *Csound*, users can write code while the program is running. It uses a unique time-based concurrent programming model [9].

Both programming languages are interfaced for Unity through *CsoundUnity* [10] and *Chunity* [11] [10], respectively. These wrappers implement the `OnAudioFilterRead` function mentioned previously to achieve sound generation in Unity.

The necessary common features for these wrappers relevant for fulfilling sound generation in Unity are described in Table 1. Additional features can be explored in their corresponding websites[10] [11].

In section 4.1, we return to these features and discuss additional considerations after integrating these wrappers in an XR Unity application.

3. METHODOLOGY

We initially searched for suitable wrappers that support sound generation in Unity for Android applications, as the *Meta Quest 3* headset is based on that OS. We found that *Csound* and *ChucK* are the current options that offer such functionality through the wrappers *CsoundUnity* and *Chunity*, respectively. Both programming languages and their wrappers are actively maintained.

A Unity project for the Android platform was set up and configured for the *Meta Quest 3* device. Both wrappers were added to this project. Then, we implemented a simple scene in *Pass-through* mode (i.e. mixed reality or physical-virtual environment) as shown in Fig. 2, consisting of coloured spheres rotating around a cube that can be moved by the user using any hand. Every sphere has an audio source attached, which is an instance of a continuous sound built through one of the wrappers. A sphere can be spawned in a random location around the perimeter of the cube and adopt a random colour. With this simple scene, we wanted to assess that spatial audio works for both wrappers by hearing the output as we moved in the scene. A video showing this scene in action was published [12]. This video was captured using a streaming service for *Meta Quest 3* [13] with the limitation of having monaural audio.

We used the scene to perform a series of experiments when the number of sources was varied to collect performance measurements. These experiments were executed based on the following scenarios:

1. **Scalability:** In this scenario, we start the scene without sound sources, and then we add one by one. With every increase, we take several measurements of the digital signal processing (DSP) time that Unity spends on generating the sound in its audio buffer. Additionally, we measure the memory that Unity uses exclusively for audio. Both parameters, *DSP time* and *audio memory*, are frequently sampled dur-

[10] `https://github.com/rorywalsh/CsoundUnity`
[11] `https://chuck.cs.princeton.edu/chunity/`
[12] `https://youtu.be/OtpDR9MnnbI`
[13] `https://www.oculus.com/casting`

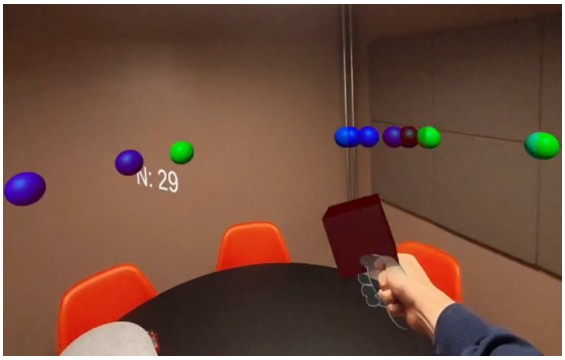


Figure 2: Screenshot for the mixed reality application taken in the *Meta Quest 3*. The audio sources are represented by coloured spheres orbiting around an interactive cube that can be moved using any user's hand. In this scene, 29 objects were instantiated, but some are not visible due to the field of view limit.

ing a period of time before adding a new source. While this experiment is running, we listen to the output to identify the number of sources that produce audio artifacts (e.g. glitches), indicating that sound processing is taking longer than allowed for real-time.

2. **Resource Leak Detection:** For this case, we start again without any sources, then we take several measurements for *DSP time* and *audio memory* in a determined period; next we add one audio source that will be playing for some time, and then we remove this source and take measurements again without any sources. We repeat this trial several times. The goal is to identify whether the DSP time and the audio memory are the same as we increase the number of trials. The increase in any of both parameters would indicate resource leak problems.

To collect these measurements, we implemented a custom script using the `ProfilerRecorder`[14] API from Unity to access the *audio memory*. For the *DSP time*, we measured the execution time for the code inside the `OnAudioFilterRead` function.

We applied both experiment scenarios to the following two devices:

1. **The Development Machine:** This is the environment where the XR application is developed. The experiments are performed over the application running in the Unity editor. The machine used was a Windows 11, 64-bit, powered by a 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz, with RAM 16 GB, and a laptop graphics processor NVIDIA GeForce RTX 3050 Ti.

2. **The *Meta Quest 3*:** A XR headset that supports virtual and mixed reality applications. The device used was the version with 128 GB of storage. Its specifications are described in section 2.1.

[14] `https://docs.unity3d.com/ScriptReference/Unity.Profiling.ProfilerRecorder.html`

For the first scenario, we considered implementing two patches: a *sine oscillator*, and a *simple synthesizer* having the signal chain `saw_oscillator` → `low_pass_filter` → `reverb` → `amplitude_envelope` → `gain`. The second scenario used only the sine oscillator patch.

An audio source that uses the sine oscillator plays continuously, and a random frequency is assigned when it is instantiated in the scene. For the simple synthesizer, the envelope plays one second every 2 seconds, and parameters for the components in the chain are changed randomly on every play. Implementation details can be explored in our public repository[4] .

For all the experiments, we set the *DSP buffer size* in Unity as `Best Performance`, meaning a buffer size of 1024 samples. We used the default audio sample rate in Unity (48000 Hz), thus the buffering latency of real-time audio is 21.33 ms. The specific settings for every experiment, as well as results and interpretations, are presented in section 4.2.

4. RESULTS AND DISCUSSION

4.1 Integration with Unity

After the testing performed, we can confirm that both wrappers are operational on the *Meta Quest 3* for the development of XR audio applications. However, there are differences in the development process that might influence the decision to use one or both of them. We discuss some of these differences as follows:

- **Learning the languages:** *Csound* and *ChucK* use very different syntax. While *ChucK* utilizes a more traditional syntax similar to programming languages like `Java`, `JavaScript`, and `C#`, making it relatively easy to learn, *Csound* has a unique syntax that may require some effort to learn, especially for those who are inexperienced. However, the effort is worth it if we want to take advantage of the many function generators offered by *Csound*, which might not be available in *ChucK*.

- **Implementing the patches:** The complexity depends on what a developer wants to achieve and what the languages offer. For instance, a simple sine wave could be relatively straightforward in both languages; however, other implementations, such as an amplitude envelope, can differ depending on the language paradigm. In our case, we implemented such an envelope for a simple synthesizer. Differences can be identified in the Unity scripts for the synthesizer behaviour[15] [16] . Additionally, *Csound* offers more built-in function generators than *ChucK*; nevertheless, *ChucK* allows to create custom plug-

[15] `https://github.com/pedro-lucas-bravo/sound_engines/blob/main/sound_engines_unity_project/Assets/Scripts/CsoundSimpleSynthe.cs`

[16] `https://github.com/pedro-lucas-bravo/sound_engines/blob/main/sound_engines_unity_project/Assets/Scripts/ChuckSimpleSynthe.cs`

ins and expand the possibilities for signal processing [17].

- **Setting up a Unity project:** In terms of project organization and file optimization, *CsoundUnity* has an advantage over *Chunity*, since it can be included in a Unity project through the *Unity Package Manager*[8] allowing its access locally in the development machine and saving space for a version control system (e.g. git) since the package is downloaded every time the project is cloned. As *Chunity* files lie within the project itself, they must be tracked by the version control system.

- **Developing multi-agent audio systems:** Instantiation and destruction of audio sources occur in a multi-agent audio system and are carried out according to the application's needs. We would expect to create an audio agent with a particular behaviour that is independent of others and not implement additional considerations to ensure that independence. In the case of *CsoundUnity*, this independence is ensured for every new instance created; however, *Chunity* needs an extra effort to achieve it; that is, variables in a *ChucK* script must be different for every instance, thus if we are using the same patch, we have to rename the variables to not interfere among sources, which requires additional code to automatize this process. Due to this issue and in comparison to *Chunity*, *CsoundUnity* facilitates the development of a multi-agent audio system.

- **The system in action:** Perceptually, there is no difference between the output generated by the wrappers, which means that any of them can be used for the intended purpose. Moreover, there were no crashes or artifacts observed when a reasonable number of sources were used in the experiments. However, there can be differences in performance, as explained in the next section.

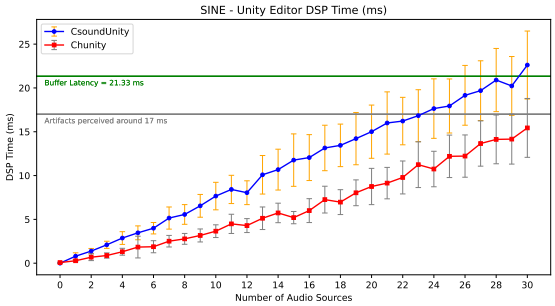
Furthermore, it is possible to include in one project both wrappers at the same time, which increases the options for sound generation in a *Meta Quest 3* application.

4.2 Computational Performance

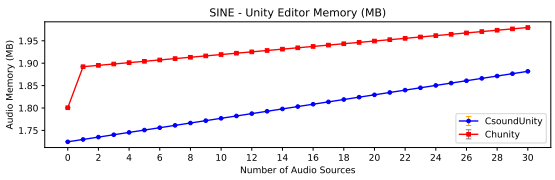
We accomplished the experiment scenarios specified in section 3 on the simple scene from the mixed reality application mentioned in the same section. We initially used the *Unity Profiler* [18] to look for noticeable performance issues like frame-rate drops or important memory leaks. We did not find such issues at that stage for both wrappers. Then, we executed the experiment scenarios with the following results.

4.2.1 Scalability

For the scalability scenario, we added up to 30 audio sources for the *sine oscillator* and the *simple synthesizer* patch. For both patches we started from zero sources, then we took measurements every 0.2 seconds until having 30 samples; next, we rested for 1 second before adding a new audio source, and so on. We performed this procedure for the Unity Editor (development machine) and the *Meta Quest 3*. Fig. 3 shows the results for the Unity Editor and Fig. 4 for the *Meta Quest 3* regarding the DSP time and the audio memory for the sine oscillator patch. The charts depict the average value and standard deviation for each number of audio sources. Note that every chart illustrates the buffer latency (21.33 ms), although we experimentally realized that glitches were perceived around 17 ms since Unity needs to use additional computational time for internal processes.



(a)



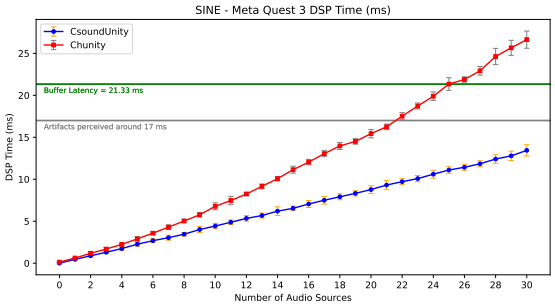
(b)

Figure 3: Results for the *Sine Oscillator* patch tested in the Unity Editor (development machine) (a) DSP Time. (b) Audio Memory.

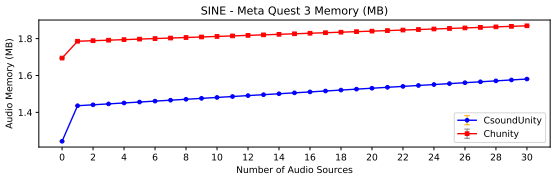
The sine oscillator patch: Note that, for the Unity Editor in Fig. 3, *Chunity* invests less DSP time than *CsoundUnity*; however, *Chunity* consumes more audio memory, although this amount is relatively low (less than 2 MB) for the overall system memory. Considering the threshold for artifacts perception of 17 ms, we found that glitches start to appear for *CsoundUnity* when there are around 25 sources, and for *Chunity* when we have 37. Although these results may vary according to the development machine, the important aspect is that ***Chunity* performs better than *CsoundUnity* in the Unity Editor under the Windows implementation of *ChucK* and *Csound***. Memory usage can be negligible in this case.

For the *Meta Quest 3* in Fig. 4, the results are different from the Unity Editor since we have less variability in the data, which is expected because the editor deals with additional processes. Moreover, the most important difference is that *CsoundUnity* takes less DSP time than *Chunity*, which is the opposite tendency regarding the Unity Editor. For the audio memory, we still have a negligible result under 2 MB. Therefore ***CsoundUnity* performs better than *Chunity* in the *Meta Quest 3* under the Android imple-**

[17] https://chuck.cs.princeton.edu/extend/
[18] https://docs.unity3d.com/Manual/Profiler.html



(a)



(b)

Figure 4: Results for the *Sine Oscillator* patch tested in the *Meta Quest 3*. (a) DSP Time. (b) Audio Memory.

mentation of *ChucK* and *Csound*. We start to perceive glitches for *CsoundUnity* when we have 39 sources, and for *Chunity* when there are 22.

The simple synthesizer patch: A more realistic example was tested considering the simple synthesizer mentioned in section 3. We show the results for the Unity Editor and the *Meta Quest 3* in Fig. 5 and Fig. 6, respectively. In this case, for both devices, the audio memory usage is similar to the sine oscillator patch and thus is not necessary to show these results; however, we can notice important differences in the DSP time. These results support the previous statements regarding which wrapper is better according to the device, although we notice that *Chunity* takes less DSP time for this patch than the sine oscillator in such a way that its performance is similar in the *Meta Quest 3* until 18 sources in Fig. 6. We speculate that *Chunity* performs better in this case due to how *ChucK* manages the amplitude envelope in comparison to *Csound* and due to the resting period between the envelope execution since it is played for one second every 2 seconds on each audio source. Nevertheless, we cannot ensure a fair comparison for this patch since we do not know details about the internal implementation of the function generators for each programming language; hence, results might change if using different generators (e.g. using a `moogladder` low pass filter instead of a `lowres` in *Csound*).

4.2.2 Resource Leak Detection

We performed the second scenario described in section 3 to detect resource leaks in DSP time and audio memory for both devices using the sine oscillator patch. We perform 100 trials for each device. Fig. 7 and Fig. 8 show the results for the Unity Editor and the *Meta Quest 3*, respectively. Ideally, we would expect zero DSP time process and a constant memory that is not invested in the audio generation after several adding-removing sources actions; however, we detect the following issues:

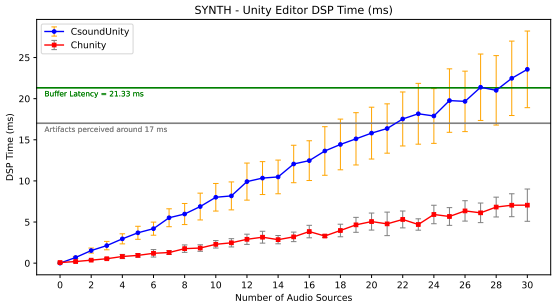


Figure 5: DSP time results for the *Simple Synthesizer* patch tested in the Unity Editor (development machine).

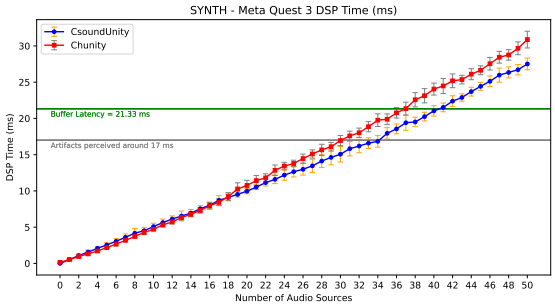
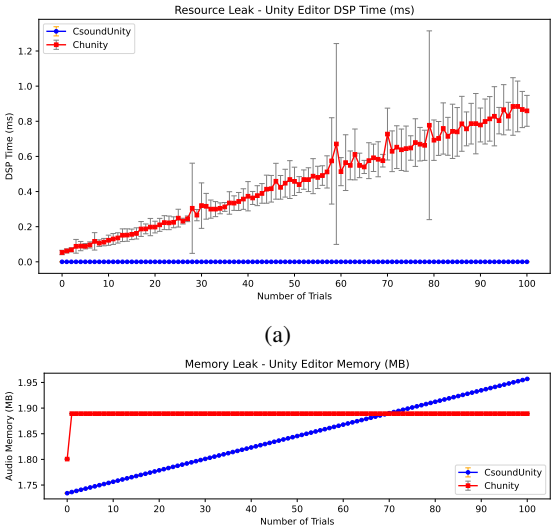


Figure 6: DSP time results for the *Simple Synthesizer* patch tested in the *Meta Quest 3*.

Resource Leak for DSP Time: In both devices, we verify that *CsoundUnity* unity is not processing anything during the several trials in both devices; that is, it portrays the expected behaviour. Nevertheless, *Chunity* starts with a small DSP time even though we have not started any adding-removing process, and then this time increases linearly as we progress in the trials for the two platforms. After the 100 trials, *Chunity* wasted close to 1 ms in resources for the Unity Editor, as shown in Fig. 7a, while for the *Meta Quest 3* in Fig. 8a, this value is close to 3 ms, although the variability is lower than in the editor. We hypothesize that the reason behind this issue is that *Chunity* would manage a centralized execution of *ChucK* scripts since every variable must be different per instance, and when we stop or remove an audio source, the code is still running even though the sound generation is not happening anymore. Moreover, *Chunity* needs the `TheChuck` and `ChuckMainInstance` components as pre-requisites to have multiple audio sources in one scene, which may cause the initial processing time when there are no sources yet.

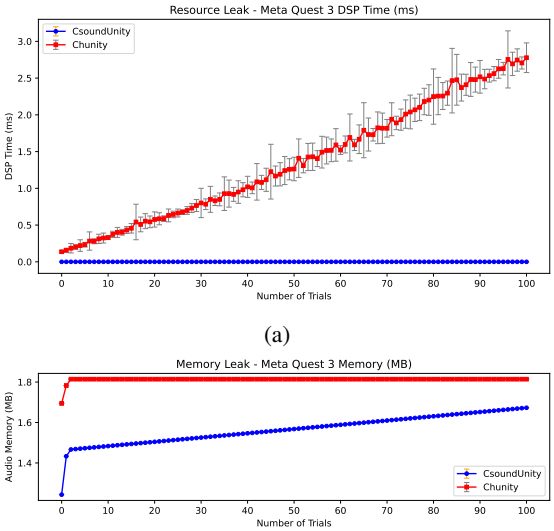
Memory Leak: In the case of audio memory, we identify an initial increment for *Chunity* in both devices as shown in Fig. 7b and Fig. 8b. For the Unity Editor, *Chunity* reaches a value close to 1.9 MB that is constant along the trials, while for the *Meta Quest 3*, this value is slightly higher than 1.8 MB. In that sense, *Chunity* fulfils the expectation of constant memory. However, *CsoundUnity* denotes a linear increment of memory that overcomes *Chunity* in the editor as shown in Fig. 7b; that is, it passes 1.9 MB around the trial 70. For the *Meta Quest 3*, the audio memory for *CsoundUnity* is also increasing but, after the 100 trials, is still below the 1.8 MB reached by *Chu-*



(a)

(b)

Figure 7: Resource Leak Detection in the Unity Editor (development machine) (a) DSP Time. (b) Audio Memory.



(a)

(b)

Figure 8: Resource Leak Detection in the *Meta Quest 3*. (a) DSP Time. (b) Audio Memory.

nity. We think that, although *CsoundUnity* is not processing audio anymore, the implementation is not freeing all the memory and, therefore, leading to this leak. While, for *Chunity*, although it is still processing and spending unnecessary DSP time, the audio memory does not increase since the audio generation is stopped, although it could be using the main memory to keep track of the hanging scripts while adding and removing sources.

5. RECOMMENDATIONS

Based on the results and their interpretations provided in the previous section, we list the following recommendations for developing multi-agent audio systems in the *Meta Quest 3*.

- **Develop for the target platform:** For our case, the target platform is the *Meta Quest 3*. If choosing a wrapper does not depend on factors other than performance, we can go for *CsoundUnity* according to our results; however, other considerations may influence this decision depending on the system, time restrictions, and the developer's skills. The important aspect is that the Unity Editor should not be used as an accurate reference for how an XR system performs.
- **Detection of issues through profiling tools:** For the *Meta Quest 3* we can use the *Unity Profiler*[18] and the *Meta Quest Developer Hub*[19] as debugging tools for quick identification of performance issues. Custom profiling tools can be implemented as in this work. Use these options to improve the application's performance if critical issues arise.
- **Estimate the maximum number of agents:** As the results show, both wrappers present resource leak issues. A way to work around this problem, and as a good practice in real-time systems, is to use *object pools* and just active or deactivate objects. It can be useful not only for audio but for other components when complete agents are pooled.
- **Identify additional audio processes:** An XR system may use additional audio objects that increase the DSP time in Unity (e.g. mixer, plugins, etc). Being aware of these additional processes, which can be quantified through profiling, is important to avoid overloading the audio buffer.
- **Think in multi-platform if needed :** Unity is a multi-platform game engine, so the same application may be potentially built for different devices and operative systems. As we have shown in the results, performance can vary depending on the platform (Windows in the editor and Android in the *Meta Quest 3*). If the development requires targeting several platforms, we need to choose the wrapper that best fits all of them, and still, the ideal case is to focus on one target platform, as pointed out before.
- **Optimization beyond Unity :***Csound* and *ChucK* have particular programming paradigms and their own ways of optimization. An example is the close performance for both in the simple synthesizer experiment, as shown previously in the results. Use the power of any of those languages as much as possible to achieve highly optimized sound generation that helps to reach your performance goals for an XR application in Unity.
- **Explore choices according to the system :** We might encounter various challenges while developing a system. Depending on the particular requirements, we may need to change our decision regarding which engine to use. For instance, if you are

[19] `https://developer.oculus.com/meta-quest-developer-hub/`

more fluent in *ChucK* or *Csound*, and performance is not critical to a certain extent, then you can use any or even both languages. However, you should consider the earlier recommendations and balance your choices based on your skills.

6. CONCLUSIONS

We explored *Csound* and *ChucK* as mediums for sound generation in Unity, focused on multi-agent audio systems running in the *Meta Quest 3* through the wrappers *CsoundUnity* and *Chunity*. We developed a simple mixed reality scene to test both wrappers and discussed their integration into Unity. The same scene was used for several experiments to evaluate their computational performance regarding DSP time and audio memory. We confirmed that both wrappers are effective with spatialization in the *Meta Quest 3*.

In terms of integration with Unity, *CsoundUnity* ensures independence for audio sources that implement the same behaviour, while *Chunity* needs explicit identification of every source, which requires an extra development effort. Additionally, both wrappers can be used for the same application.

In terms of computational performance, results regarding the DSP time tell us that *Chunity* performs better than *CsoundUnity* in the Unity Editor for a Windows machine, but *Csound* is more efficient in the *Meta Quest 3* (based on Android). When using any of the wrappers, the audio memory is relatively low (less than 2 MB) and thus negligible to have a considerable impact. Despite these results, patches might be more or less efficient depending on how they are implemented and how efficient the function generators are in every language, which might influence in the results.

We discovered resource leak issues for both wrappers. *Chunity* consumes DSP time when several adding-removing actions happen, while *CsoundUnity* does not free all memory under the same conditions and *Chunity* does; however, *CsoundUnity* does not consume DSP Time. An object pool can be used to mitigate these issues.

Based on these results, we suggested the recommendations listed in the previous section that can be used to select the most suitable wrapper (or both) for an XR application.

For future work, we plan to assess these wrappers for other platforms supported by Unity, such as general Android devices, iOS devices, PCs (Mac, Linux, Windows), and web browsers. In addition, we aim to enable an XR application to switch between the default spatial audio system for the *Meta Quest 3* and an external multi-speaker system. To achieve this, we need to investigate how to use *Csound* and *ChucK* patches in external sound engines that support ambisonic capabilities.

Acknowledgments

This work was partially supported by the Research Council of Norway through its Centres of Excellence scheme, project number 262762.

7. REFERENCES

[1] L. Turchet, R. Hamilton, and A. Camci, "Music in Extended Realities," *IEEE Access*, vol. 9, pp. 15 810–15 832, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9328440/

[2] L. Turchet, "Musical Metaverse: vision, opportunities, and challenges," *Personal and Ubiquitous Computing*, vol. 27, no. 5, pp. 1811–1827, Oct. 2023. [Online]. Available: https://link.springer.com/10.1007/s00779-023-01708-1

[3] D. Johnson, D. Damian, and G. Tzanetakis, "OSC-XR: A Toolkit for Extended Reality Immersive Music Interfaces," in *16th Sound and Music Computing Conference*, Málaga, Spain, May 2019, iSBN: 9788409085187 Publisher: Zenodo. [Online]. Available: https://zenodo.org/record/3249318

[4] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.

[5] P. Lucas and K. Glette, "Human-Swarm Interactive Music Systems: Design, Algorithms, Technologies, and Evaluation," *Proceedings of the 16th International Symposium on Computer Music Multidisciplinary Research*, Nov. 2023, publisher: Zenodo. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.10113080

[6] A. Bargum, O. I. Kristjansson, S. Rostami Mosen, and S. Serafin, "Spatial Audio Mixing in Virtual Reality," in *19th Sound and Music Computing Conference*, Saint-Étienne, France, Jun. 2022, publisher: Zenodo. [Online]. Available: https://zenodo.org/record/6572821

[7] P. P. Lucas and S. Fasciani, "A Human-Agents Music Performance System in an Extended Reality Environment," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, M. Ortiz and A. Marquez-Borbon, Eds., May 2023, pp. 10–20, place: Mexico City, Mexico. [Online]. Available: http://nime.org/proceedings/2023/nime2023_2.pdf

[8] V. Lazzarini, O. Brandtsegg, J. ffitch, J. Heintz, I. McCurdy, and S. Yi, *Csound: A Sound and Music Computing System*, 1st ed. Cham: Springer International Publishing : Imprint: Springer, 2016.

[9] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, vol. 39, no. 4, pp. 10–29, Dec. 2015. [Online]. Available: https://direct.mit.edu/comj/article/39/4/10/106778/ChucK-A-Strongly-Timed-Computer-Music-Language

[10] J. Atherton and G. Wang, "Chunity: Integrated Audiovisual Programming in Unity," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, T. M. Luke Dahl, Douglas Bowman, Ed. Blacksburg, Virginia, USA: Virginia Tech, Jun. 2018, pp. 102–107, iSSN: 2220-4806. [Online]. Available: http://www.nime.org/proceedings/2018/nime2018_paper0024.pdf