Sonic Playground: Immersive Audio Design Project

Pranavv Jothinathan, Suhang Liu, Skyler Sun, Yifei Liu

December 12, 2025

Introduction

Sonic Playground is an interactive virtual reality exhibition designed to explore how sound behaves in physical space through direct user interaction. The project adopts a museum-style layout consisting of three interconnected exhibits: Resonance, Cave Acoustics, and Material-Dependent Sound. Each exhibit focuses on a distinct acoustic principle and allows users to engage with sound as a physical and spatial phenomenon rather than passive background audio. The experience is designed to be exploratory, educational, and accessible, enabling users to move freely between exhibits while interacting with sound-producing objects in a controlled virtual environment.

The audience experiences sound like something that can be created, manipulated, and shaped by their actions. In the first exhibit, users record their own voice and use it to excite resonance in a virtual wine glass, observing vibration and breakage as audible and visual feedback. In the second exhibit, users walk from an open outdoor environment into an enclosed cave, where changes in reverberation, echo, and reflection become immediately perceptible through footsteps, object collisions, and environmental sounds. In the third exhibit, users interact with balls made of different materials, rolling and dropping them to hear how material properties affect both motion and sound. Across all exhibits, sound responds dynamically to user input, movement, and spatial context, reinforcing the connection between physical action and auditory outcome.

The project is exciting because it transforms abstract acoustic concepts into embodied, experiential learning. Rather than explaining resonance, reverberation, or material acoustics theoretically, the project allows users to directly cause and observe these phenomena. Breaking a glass with one's own voice, hearing exaggerated echoes when entering a cave, or comparing the sound of metal and wood on the same surface creates moments of surprise and engagement. The simplicity of interaction ensures that users can focus on listening and experimentation, while the enhanced audio effects amplify perceptual differences to make the underlying principles clear and memorable.

Interaction in *Sonic Playground* is addressed through intuitive object selection and manipulation techniques designed for VR. Users can physically grab and release objects, such as balls or blocks, using XR-based grab interactions, allowing natural hand-driven manipulation. Poke-based buttons are used to trigger actions including recording sound, resetting objects, and switching between exhibits, providing reliable and accessible control through direct touch. In some cases, ray-based interactions are used to adjust parameters such as amplitude or volume from a distance. These interaction methods ensure that users can easily and deliberately influence sound-producing objects, reinforcing the cause-and-effect relationship between interaction and audio response.

Physics plays a central role in shaping both interaction and sound behaviour. In the resonance exhibit, the system analyses the frequency content of the user's recorded voice and compares it to the natural frequency of the virtual wine glass. When resonance conditions are met, the glass vibrates, and excessive energy leads to structural failure, simulating a real-world physical limit. In the cave exhibit, environmental geometry and surface materials determine how sound reflects and decays, demonstrating how irregular rock surfaces scatter sound energy to create complex echoes. In the material-based exhibit, balls made of wood, plastic, and metal are assigned different masses, damping values, and physics materials, causing them to move and collide differently. These physical differences directly influence the resulting impact and rolling sounds, ensuring that audio behaviour emerges from simulated physical properties rather than scripted playback.

Locomotion and navigation are designed to support immersion while maintaining user comfort. Users move through the exhibition space using a combination of walking and teleportation, allowing both continuous exploration and quick transitions between exhibits. Audio feedback is integrated into navigation through footstep sounds and teleportation cues, which provide auditory confirmation of movement and help maintain spatial awareness. This approach ensures that navigation itself contributes to the overall acoustic experience rather than breaking immersion.

Audio spatialisation is applied consistently to all sound-producing objects to allow users to perceive direction, distance, and relative position. Unity's 3D spatial sound system is combined with the Steam Audio plugin to deliver HRTF-based binaural rendering, distance-based attenuation, occlusion, and air absorption. As users move around the environment, sounds change in intensity and timbre depending on their position and the presence of obstacles. This enables listeners to localise sound sources naturally and understand the spatial layout of the environment through listening alone.

A convincing acoustic environment is achieved by ensuring that sound behaviour is tightly coupled to physical space and material context. Steam Audio Geometry components define walls, floors, and large structures, allowing sound to reflect, scatter, and decay realistically. Probe baking is used to precompute sound propagation paths, improving consistency and performance across the environment. In the cave exhibit, where real-time reverberation alone was insufficient to clearly convey enclosure, pre-processed echo effects are combined with Steam Audio's reverb and reflections to exaggerate perceptual differences between open and enclosed spaces. This hybrid approach ensures that users can clearly perceive how spatial scale and material properties shape sound, fulfilling the project's goal of demonstrating sound operation in physical space.

Background Research

Resonance in Unity and Audio Feature extraction

Unfortunately, there are currently no mature projects available that demonstrate how to achieve resonance effects in Unity. However, there is extensive literature on human voice audio analysis. Exhibit 1 should primarily draw on methods for converting voice signals from the time domain to the frequency domain and extracting meaningful information. [8] [10]

Cave sound design

Acoustic measurements of real caves show that acoustic parameters such as reverberation time (T20) and early decay time (EDT) inside caves vary significantly with spatial volume and geometry and have a longer reverberation tail compared to open/semi-enclosed spaces.[1] This makes the Cave environment a very typical and frequently used scenario in immersive sound design. For example, Minecraft has an excellent cave sound design.

The game enhances the sense of space by switching between different ambient sound layers between the open world and enclosed spaces. Comparing the sound effects inside and outside the cave, we can see that footsteps inside the cave have an echo. Furthermore, when players strike rocks inside the cave, the sound is more humid than outdoors.[2] This clear acoustic contrast effectively conveys the change in spatial scale and the degree of enclosure to the player, and it inspired my sound transition design in XR projects.

Figure 1.Screenshot showing an area in Minecraft.

Logic：The sound of smashing a wall with a hammer change depending on your location.

Sound interaction design

To improve the interaction, active player participation can be addressed with adaptive audio—that is, audio that responds to changes in game states. For instance, Planet Xerilia uses a collision-based system to create an acoustic environment in which water drips down from a ceiling. The dropping water excites the environment's reverberation, aiming to evoke the impression of a cavernous, wet room. The timer's instantiation frequency can be set by the sound designer, providing control over the density of the resulting texture. [3] This not only suggested that the sound of dripping water was a great audio element for caves but also inspired me to use collision-triggered sound effects and timers to control the duration of the sound effects.

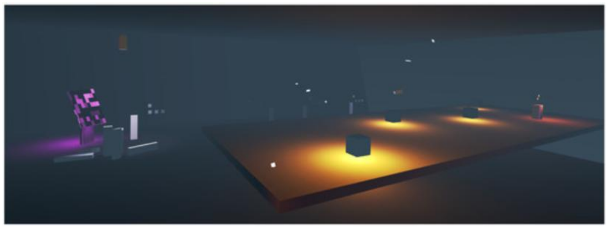


Figure2. Screenshot showing an area in Planet Xerilia. Logic：The collision-based system is implemented in the droplets, here visible as randomly rotated white small squares, which fall from the area’s ceiling.

In general, physics is widely regarded as one of the most intellectually demanding STEM disciplines, requiring not only strong analytical skills but also the ability to conceptualize abstract phenomena. Understanding the principles of physics involves integrating mathematical reasoning with experimental observation. A marble-run environment provides an intuitive, hands-on way for users to perceive the physical characteristics of different materials such as wood, plastic, metal, and concrete. As the marbles roll, collide, and come to rest on surfaces made of these materials, users can clearly observe variations in friction, bounciness, and sound responses. The systematic review by [13] reports that virtual reality learning environments, especially when combined with gamification, tend to be more motivating, engaging, and interactive than traditional environments, and they enhance knowledge gaining through realistic and interactive experiences.

A similar project to the marble run

[14] Bringing on a project which is thematically similar to the exhibit03 is where five distinct planetary environments collectively contain 25 levels of progressively increasing difficulty, guiding players through foundational to more complex concepts. Each environment is to introduce and reinforce specific physical characteristics like the Mars environment introduced external forces that influence a ball’s movement in its run. This would help users understand the physical characteristics of the planets. This project and the Sonic Playground, both try to explain characteristics of physics by using the abilities of Virtual Reality. The project was developed using C# as the scripting language and the Unity game engine and uses Meta Quest devices to deliver a Virtual Reality co-learning experience, in which they have been successful. Similar to this, the exhibit03 of Sonic Playground also uses C# as the scripting language and Unity game engine with Meta Quest devices to deliver the Virtual Reality experience.

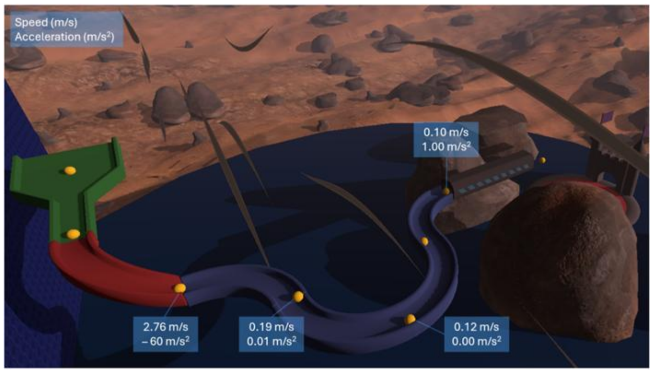


Figure 3. Screenshot of the marble run used in a multi-player Virtual reality game for Physics Co-Learning

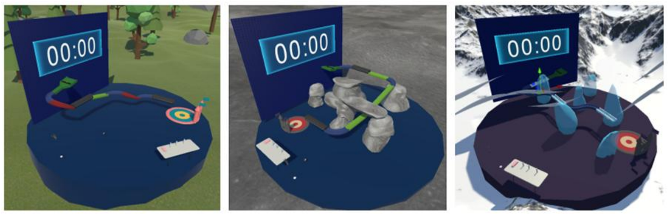


Figure 4. Three level from the Earth, Moon and Ice planet used in a multi-player Virtual reality game for Physics Co-Learning

Interactions in the Experience:

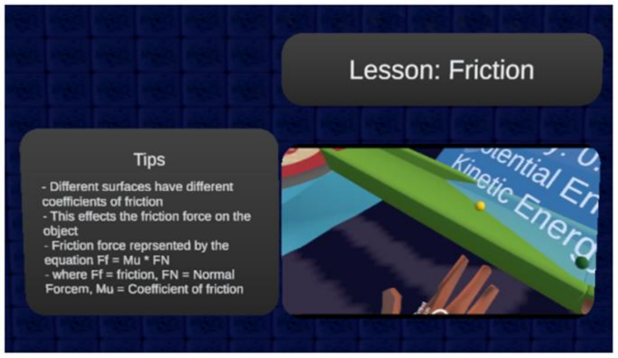
[14] Use Hand interaction as their main way to interact with gameObjects in the experience, with grabbing and teleportation being two of four hand gestures used. Interaction through teleportation is used as one of the main navigations. A similar interaction is used in the exhibit03, where interaction through grabbing is implemented to let the users pick the ball up to play with it by throwing, rolling and dropping them, enabling the users to learn by interaction in a gamified way.

[14] uses a palm menu for the users to release the marbles, to reset the marbles to their start positions. In the exhibit03, not a menu but a series of buttons are placed for the users to release the balls on to the marble run either separately or at the same time and to reset the balls to their original start positions.

Guidance in the Experience:

[14] uses text boxes as a way to provide learning materials to the users between the experience helping the knowledge gain. Similarly, in the exhibit03, text boxes are used to

guide the users on how the users can interact with the gameObjects inside the experience and which are all the gameObjects the users can interact with.



Tips

- Different surfaces have different coefficients of friction
- This effects the friction force on the object
- Friction force reprsented by the equation Ff = Mu * FN
- where Ff = friction, FN = Normal Forcem, Mu = Coefficient of friction

Figure 5. Text boxes used to guide users in a multi-player Virtual reality game for Physics Co-Learning

Design & Implementation

Project Kickoff

The initial distribution of tasks and brainstorming sessions were guided by Professor Pete Bennett. Ultimately, our group agreed to pursue a direction that minimizes project risk. Although each member contributed ideas, much of the concept selection was constrained by the project brief itself; our final idea was essentially determined by the themes assigned to us.

To avoid scope creep and unrealistic ambitions, we repeatedly consulted with Pete, Chris, and other faculty members to verify that our suggested ideas were feasible within the available time. Ambitious concepts such as a battlefield simulation or a full spaceship interior were excluded due to heavy visual and audio workload. Instead, we converged on a museum–style interactive experience focused on acoustic phenomena.

The three selected installations of resonance, acoustic reflections in a cave, and material–dependent impact sounds were all achievable in Unity and aligned well with material covered in laboratory sessions. To keep the workflow manageable, each exhibit was assigned to a different team member. While this helped everyone specialize, it also required constant communication to track dependencies and integration. We chose this division of labor because all members were multi-skilled but lacked confidence in certain technical areas, making specialization the most efficient choice.

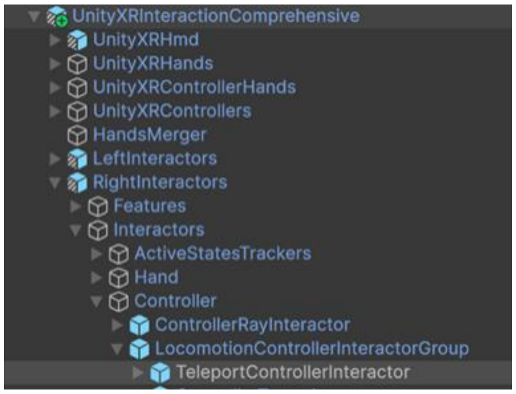
General developments:

Teleport Audio

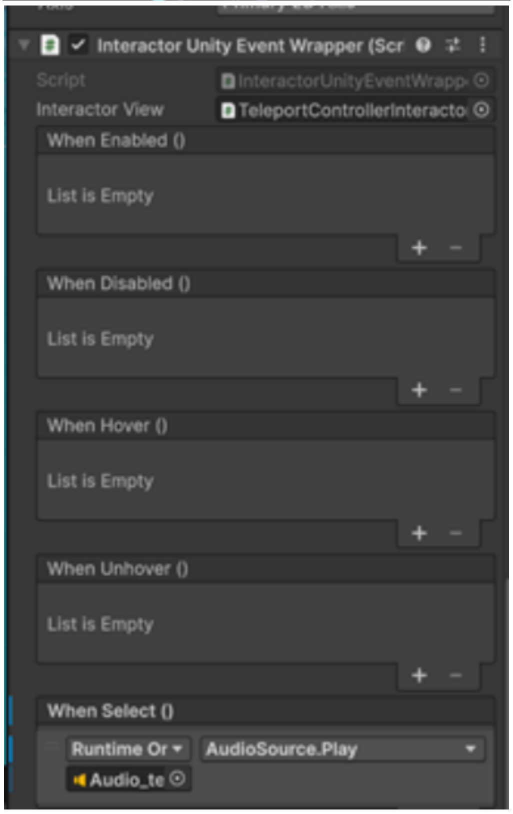
Implementing teleport audio requires understanding the distinction between interactors and interactables in Unity's XR Interaction Toolkit. The process itself is straightforward: locate the TeleportControllerInteractor on the XR camera and add a Unity event to the interactor. Conceptually, an interactor is the object that initiates an interaction, while an interactable is the object that receives the interaction. When an object such as the XR camera already contains an interactor component, events can be added directly to the "When Select" event.

If an object does not contain an interactor, one must be created manually. This situation occurs, for example, when attempting to add events to a grab interaction. Initially, no interactor component is visible, so it becomes necessary to examine the grab interaction scripts provided by the XR toolkit. After reviewing Grabbable.cs, the challenge becomes identifying how to trigger the corresponding events. Although the event exists within PointerEvent, calling the available functions produces no effect. Many functions referenced in older Unity documentation are deprecated in newer versions, which complicates the process.

A breakthrough occurs when examining Teleport.cs while attempting to add teleport-related events. During this process, the locomotion-performed callback becomes visible. Adding the grab event to this structure reveals a useful function named grabbable.whenPerformed. Once this function is used, the script begins functioning correctly.



```
▼ UnityXRInteractionComprehensive
  ▶ UnityXRHmd
  ▶ UnityXRHands
  ▶ UnityXRControllerHands
  ▶ UnityXRControllers
    HandsMerger
  ▶ LeftInteractors
  ▼ RightInteractors
    ▶ Features
    ▼ Interactors
      ▶ ActiveStatesTrackers
      ▶ Hand
      ▼ Controller
        ▶ ControllerRayInteractor
        ▼ LocomotionControllerInteractorGroup
          ▶ TeleportControllerInteractor
```



Interactor Unity Event Wrapper (Scr

Script	InteractorUnityEventWrapp
Interactor View	TeleportControllerInteracto

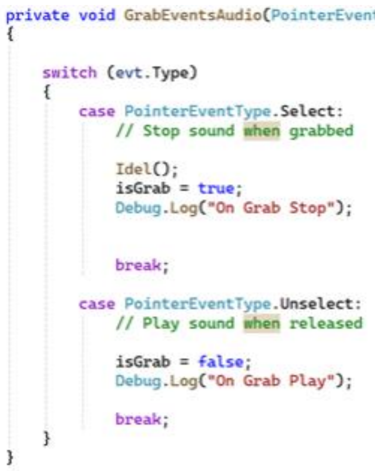
When Enabled ()
List is Empty

When Disabled ()
List is Empty

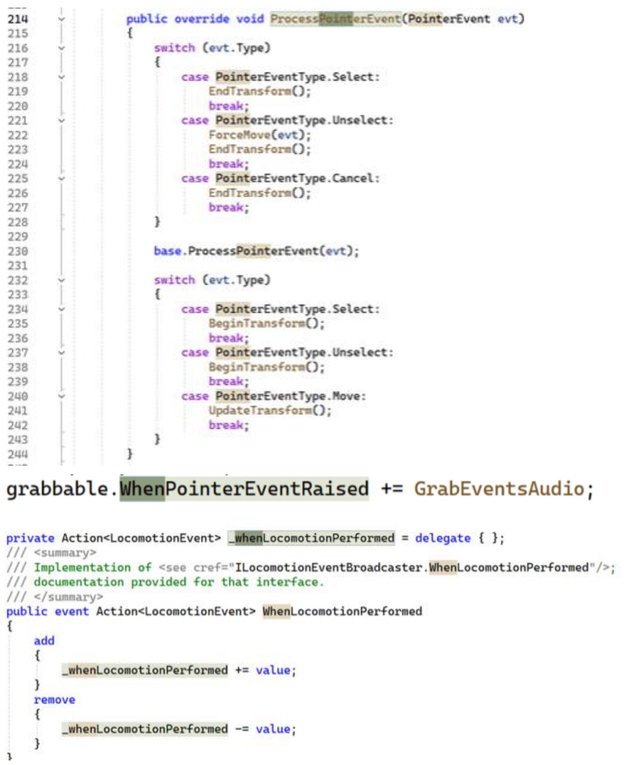
When Hover ()
List is Empty

When Unhover ()
List is Empty

When Select ()
Runtime Or | AudioSource.Play
Audio_te



```
private void GrabEventsAudio(PointerEvent evt)
{

    switch (evt.Type)
    {
        case PointerEventType.Select:
            // Stop sound when grabbed

            Idel();
            isGrab = true;
            Debug.Log("On Grab Stop");


            break;

        case PointerEventType.Unselect:
            // Play sound when released

            isGrab = false;
            Debug.Log("On Grab Play");

            break;
    }
}
```



```
        public override void ProcessPointerEvent(PointerEvent evt)
        {
            switch (evt.Type)
            {
                case PointerEventType.Select:
                    EndTransform();
                    break;
                case PointerEventType.Unselect:
                    ForceMove(evt);
                    EndTransform();
                    break;
                case PointerEventType.Cancel:
                    EndTransform();
                    break;
            }

            base.ProcessPointerEvent(evt);

            switch (evt.Type)
            {
                case PointerEventType.Select:
                    BeginTransform();
                    break;
                case PointerEventType.Unselect:
                    BeginTransform();
                    break;
                case PointerEventType.Move:
                    UpdateTransform();
                    break;
            }
        }
```

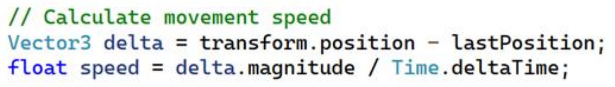
```
grabbable.WhenPointerEventRaised += GrabEventsAudio;
```

```
private Action<LocomotionEvent> _whenLocomotionPerformed = delegate { };
/// <summary>
/// Implementation of <see cref="ILocomotionEventBroadcaster.WhenLocomotionPerformed"/>;
/// documentation provided for that interface.
/// </summary>
public event Action<LocomotionEvent> WhenLocomotionPerformed
{
    add
    {
        _whenLocomotionPerformed += value;
    }
    remove
    {
        _whenLocomotionPerformed -= value;
    }
}
```

Figure 6. Teleport Audio code & componet

Walk audio

Walk audio is implemented by calculating the player's movement speed. This requires understanding Unity's built-in object state system, particularly the Transform component, along with deltaTime, which provides the time elapsed between frames.



```
// Calculate movement speed
Vector3 delta = transform.position - lastPosition;
float speed = delta.magnitude / Time.deltaTime;
```

Figure 7. walk speed detect code

The input of the voice

Unity provides example scripts for recording and playing back audio. Although the logic is simple—recording into an AudioClip and then playing it—the AudioSource fails to store the clip correctly in practice, resulting in an empty AudioClip at the start of recording. Because of this limitation, an alternative approach is required: saving the recorded audio to a file and then analyzing it through script.

Saving microphone input as a file is not straightforward. A widely referenced solution is found in the online resource “Unity3D: script to save an AudioClip as a .wav file.” The useful portion of the code lies between lines 108 and 186.

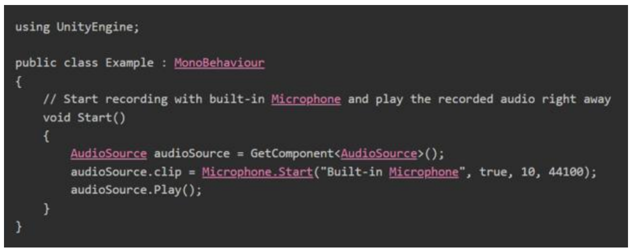
On Android devices, the file path must use Application.persistentDataPath rather than Application.dataPath, as the latter works only on Windows during testing.

The WAV file is constructed manually. First, the RIFF header is written, including the RIFF identifier and file size (36 + samples.Length × 2). Next, the WAVE format identifier is added. The format sub-block follows, containing the format space identifier, sub-block size (16), audio format (1 for PCM), number of channels (1 for mono), sample rate, byte rate × 2, block alignment (2), and bit depth (16). Finally, the data sub-block is written, including the data identifier and data size (samples.Length × 2).

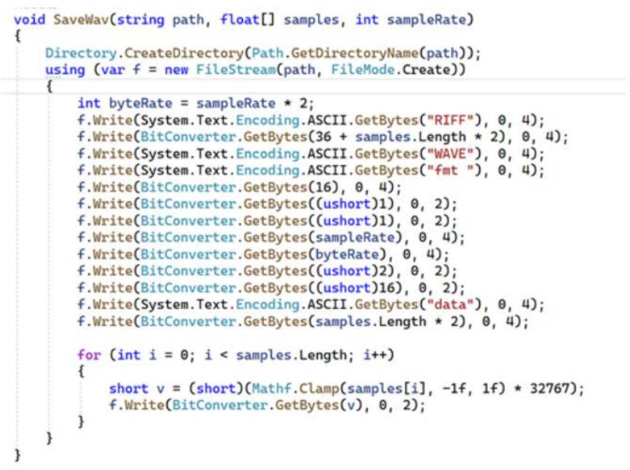
Each audio sample is processed in three steps:

1. Amplitude limiting using Mathf.Clamp to restrict values to the range −1 to +1.
2. Quantization to the 16-bit integer range (−32768 to +32767), scaled by 32767.
3. Immediate writing of the resulting 2-byte value to the file.

After saving the audio as a WAV file and confirming the file path, playback works correctly on Windows. However, testing on a VR headset initially fails, leading to the need for proper debugging tools.



```
using UnityEngine;

public class Example : MonoBehaviour
{
    // Start recording with built-in Microphone and play the recorded audio right away
    void Start()
    {
        AudioSource audioSource = GetComponent<AudioSource>();
        audioSource.clip = Microphone.Start("Built-in Microphone", true, 10, 44100);
        audioSource.Play();
    }
}
```



```
void SaveWav(string path, float[] samples, int sampleRate)
{
    Directory.CreateDirectory(Path.GetDirectoryName(path));
    using (var f = new FileStream(path, FileMode.Create))
    {
        int byteRate = sampleRate * 2;
        f.Write(System.Text.Encoding.ASCII.GetBytes("RIFF"), 0, 4);
        f.Write(BitConverter.GetBytes(36 + samples.Length * 2), 0, 4);
        f.Write(System.Text.Encoding.ASCII.GetBytes("WAVE"), 0, 4);
        f.Write(System.Text.Encoding.ASCII.GetBytes("fmt "), 0, 4);
        f.Write(BitConverter.GetBytes(16), 0, 4);
        f.Write(BitConverter.GetBytes((ushort)1), 0, 2);
        f.Write(BitConverter.GetBytes((ushort)1), 0, 2);
        f.Write(BitConverter.GetBytes(sampleRate), 0, 4);
        f.Write(BitConverter.GetBytes(byteRate), 0, 4);
        f.Write(BitConverter.GetBytes((ushort)2), 0, 2);
        f.Write(BitConverter.GetBytes((ushort)16), 0, 2);
        f.Write(System.Text.Encoding.ASCII.GetBytes("data"), 0, 4);
        f.Write(BitConverter.GetBytes(samples.Length * 2), 0, 4);

        for (int i = 0; i < samples.Length; i++)
        {
            short v = (short)(Mathf.Clamp(samples[i], -1f, 1f) * 32767);
            f.Write(BitConverter.GetBytes(v), 0, 2);
        }
    }
}
```

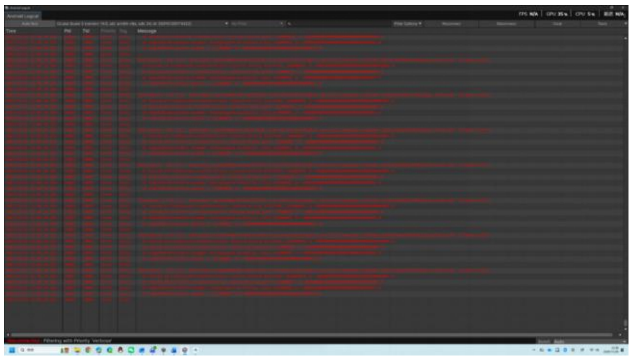


Figure 8. Audio save code & debug log

The method to test in headset

1. Android Logcat
Installing the Logcat package in Unity allows real-time viewing of output logs while the headset is running. This method provides the most effective debugging feedback.

2. Meta Developer Hub
This method proves unreliable due to software-related errors that prevent log searching. Online sources indicate that the issue originates from the tool itself.

3. Viewing the Scene in the Headset During Play Mode
By selecting Oculus as the Play Mode OpenXR Runtime and connecting the headset to the PC, the Unity scene can be viewed directly inside the headset during Play Mode. This approach allows immediate observation of scene changes without building and deploying the project.

Technical implementation of the three exhibits

1. Exhibit 1: Resonance

1.1 Initial Requirements Analysis

The goal of this exhibit was to recreate the physical principle of acoustic resonance while permitting a moderate degree of user interaction. In acoustics, resonance occurs when a vibrating source transfers energy to an object sharing the same natural frequency. Users have limited ways to directly influence the resonating object (a wine glass). Instead, interaction focuses on the sound source: users may reposition the amplifier or adjust its volume. Since audio originates from the user's voice, shaking and breaking animations serve as visual reactions to sound rather than components of sound design.
During early development, it became clear that this exhibit offered limited room for creative audio design. After extensive discussion, we explored artificially amplifying the 440 Hz resonance frequency within the user's recorded voice. However, technical limitations and perceptual issues ultimately caused us to abandon this idea.

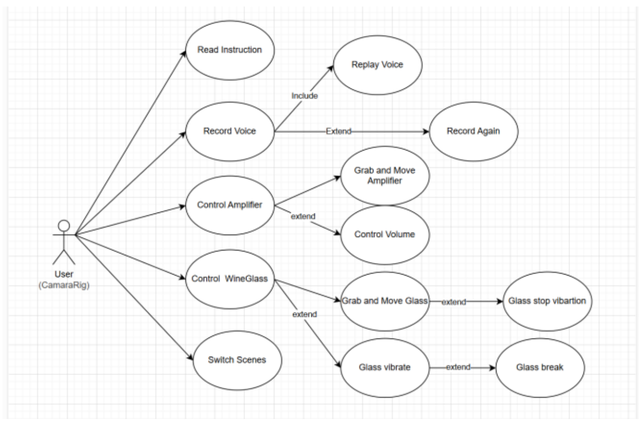
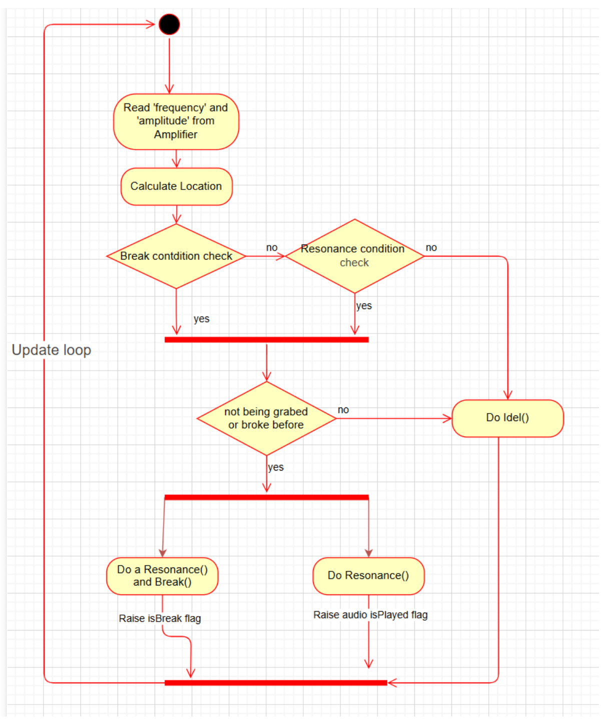


Figure 9. Use Case Diagram for Resonance Exhibit

1.2 Prototype Phase Design

The resonance prototype was co-designed by Suhang and Yifei. Suhang implemented the visual and animation systems, while Yifei focused on capturing and storing microphone input.
The visual component was completed within a week. However, introducing physics caused major issues: frame-by-frame movement of the glass led to unstable rigidbody interactions, including tunneling through colliders or explosive motion under continuous collision detection. These attempts remain in the commented section of resonance_vibration.cs. The shaking effect was eventually replaced by an Unity's Animator state machine. This method was stable but occasionally desynchronized, with animations lagging by one frame which is an issue that remained unresolved.
Meanwhile, Yifei completed the recording and saving pipeline, creating a functional prototype. To increase the sound-design depth, we began experimenting with STFT-based frequency enhancement. [10]


Read 'frequency' and 'amplitude' from Amplifier
Calculate Location
Break contdition check
no
Resonance condition check
no
yes
yes
Update loop
not being grabed or broke before
no
Do Idel()
yes
Do a Resonance() and Break()
Do Resonance()
Raise isBreak flag
Raise audio isPlayed flag
Figure 10. Activity Diagram for Resonance Prototype

Logic: If the detected frequency is valid and amplitude is dangerous or distance becomes unsafe, the glass vibrates and breaks immediately. Otherwise, it vibrates under resonance conditions or stays idle.

1.3 First Iteration: Frequency Analysis and Enhancement

During the first week of lectures, we learned Fourier's idea that sound can be expressed as a sum of sinusoidal components. In 1965, Cooley and Tukey published An Algorithm for the Machine Calculation of Complex Fourier Series, introducing the FFT (Fast Fourier Transform) [9]. This dramatically reduced the computational cost of spectral analysis and enabled real-time audio processing.
Our strategy was therefore straightforward: [10]

1. Convert the user's voice to the frequency domain.
2. Amplify the 440 Hz bin.
3. Reconstruct the enhanced audio using inverse FFT.

This allowed any vocal input to achieve resonance and helped demonstrate what reinforced resonance frequencies sound like.
However, the output resembled the original voice with a synthetic sinusoid layered on top. This occurred because we simply multiplied a single bin without smoothing adjacent frequencies. Multi-filter approaches might have worked, but were too time-consuming. The unused code remains in Voicetuning.cs.

1.4 Second Iteration
During this phase, the audio pipeline was reintegrated with the resonance system, and much time was spent debugging. One major issue was microphone sampling rate. The Meta Quest 3 records at 44,100 Hz, while the laptop used for testing recorded at 48,000 Hz. This mismatch caused significant phase shifts, breaking the alignment of the analysis window. At first, I mistakenly assumed the FFT itself was incorrect due to unfamiliarity with C#.
I then switched to the Goertzel algorithm, which measures the energy of a specific frequency. [11] [12] It produced no useful results until the sampling-rate mismatch was fixed. After correction, Goertzel worked but was still unsuitable for real-time use.
Although Goertzel is normally ideal for DTMF tone detection [Reference], my implementation analyzed only the 440 Hz component and did so once per frame on very small audio snippets. As a result, normalized energy values were extremely low. Worse, Quest 3's microphone appeared to apply strong noise suppression: synthetic 440 Hz tones produced worse results than human voices, with output sounding quiet and fragmented.
Thus, the Goertzel approach was abandoned. The code remains in AudioEnhencer.cs.

1.5 Third Iteration
Switching to Unity's built-in GetSpectrumData() immediately solved all earlier issues. In this method, it no longer matters how many harmonics or partials exist in the user's voice. Resonance is detected as long as the 440 Hz frequency bin contains enough energy to surpass a threshold. The method is simple and effective.

Originally, the system was supposed to dynamically compute both the frequency and amplitude used by the resonance logic. However, amplitude proved too unstable due to inconsistent microphone output, so it was replaced with a user-adjustable fixed value. This is why users cannot increase amplitude merely by shouting louder during recording.

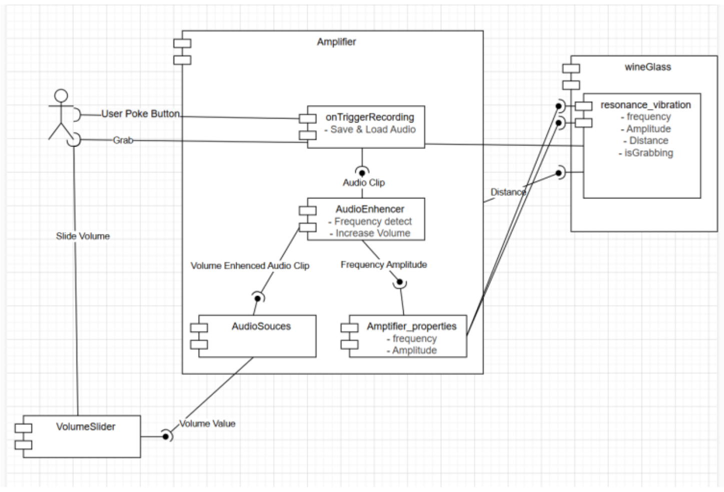


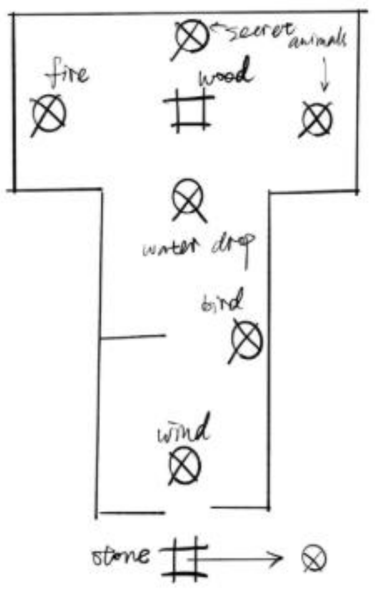
Figure 11. Final Component Diagram for the Resonance System

2. Exhibit 2: Cave

2.1 Ambient soundscape interaction design

2.1.1 Audio source selection and distance-based attenuation

To create an immersive audio environment in exhibit2, we selected some typical cave sounds, such as wind, dripping water, and bird sounds as ambient sound effects. The sound sources are placed at a specific location, and by using 3D Spatial Blend and setting Volume Rolloff, combined with Steam Audio occlusion calculations, distance attenuation and filtering of the sound are achieved.



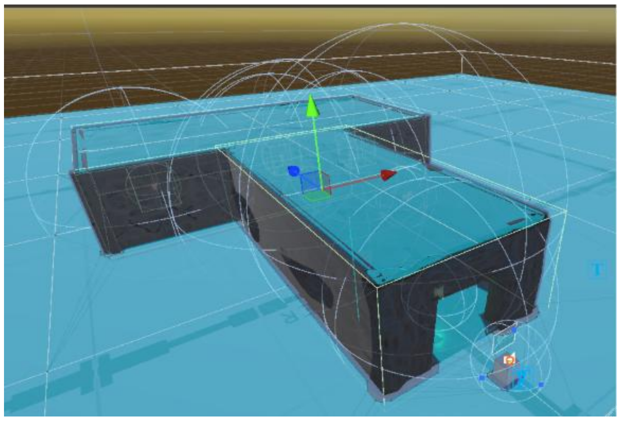


Figure 12. Distribution of sound locations

Taking the wind sound at the entrance as an example, we made some adjustments to its sound decay range. We believe the sound of wind at the cave entrance could serve as an auditory cue [4], allowing the audience to experience the sense of space through the cave entrance. Initially, we attempted to add Linear or Square decay scripts. However, we found custom scripts did not coordinate well with other acoustic effects from Steam Audio, such as occlusion and reflection. As a result, we used Steam Audio's built-in Logarithmic model to adjust the decay radius and the meta simulator to test the relationship between sound intensity and location.

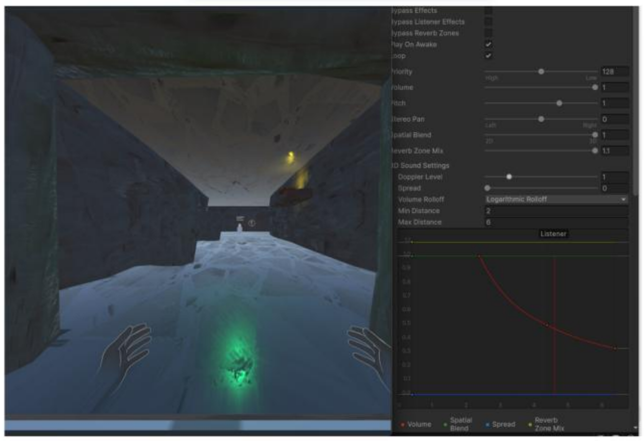
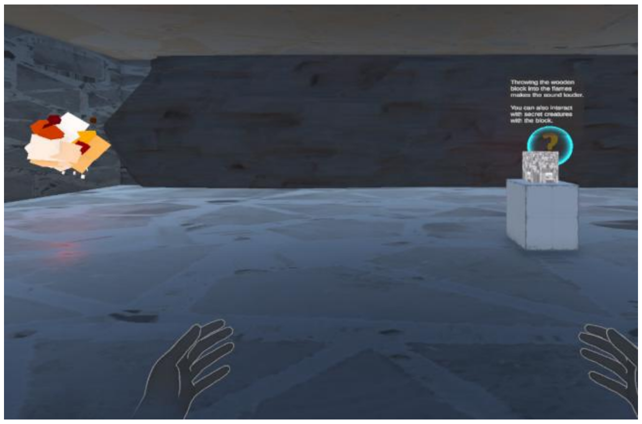

ypass Effects
ypass Listener Effects
ypass Reverb Zones
lay On Awake ✓
oop ✓
riority 128
olume 1
itch 1
tereo Pan 0
patial Blend 1
everb Zone Mix 1.1
D Sound Settings
Doppler Level 1
Spread 0
Volume Rolloff Logarithmic Rolloff
Min Distance 2
Max Distance 6

Figure 13. The relationship between sound intensity and location

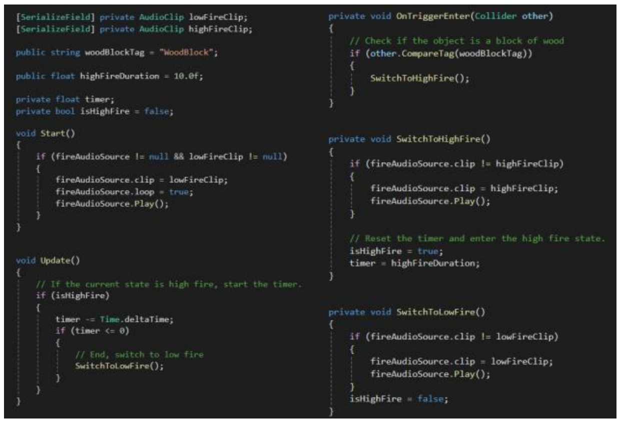
2.1.2 Sound interaction design

On top of static ambient sounds, we have also incorporated some dynamic feedback mechanisms to make the sound interaction more engaging. In the collision-based system, forces push objects to move and, eventually, collide with other objects, with the collision resulting in sound change.[3] We used a wood block to interact with sound sources including burning fire and secret animals. The Fire Controller script listens for block collisions and, upon a collision, switches the audio source to a louder High FireClip, which continues for 10 seconds before automatically reverting to normal. At the same time, audiences could also throw the wood block onto the question marks in the scene to hear the echoing animal calls.


Throwing the wooden block into the flames makes the sound louder.

You can also interact with secret creatures with the block.

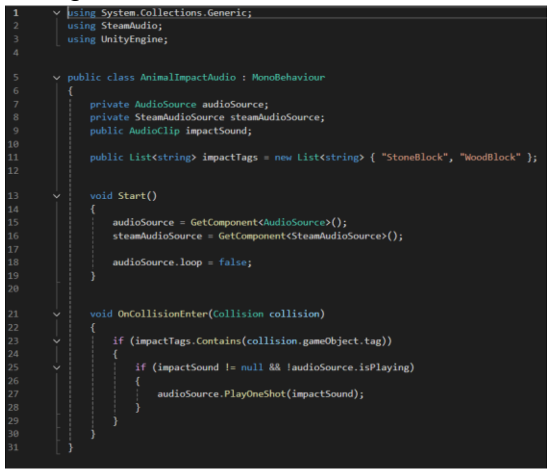
Figure 14. Wooden blocks interact with sound sources



```
[SerializeField] private AudioClip lowFireClip;
[SerializeField] private AudioClip highFireClip;

public string woodBlockTag = "WoodBlock";

public float highFireDuration = 10.0f;

private float timer;
private bool isHighFire = false;

void Start()
{
    if (fireAudioSource != null && lowFireClip != null)
    {
        fireAudioSource.clip = lowFireClip;
        fireAudioSource.loop = true;
        fireAudioSource.Play();
    }
}


void Update()
{
    // If the current state is high fire, start the timer.
    if (isHighFire)
    {
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            // End, switch to low fire
            SwitchToLowFire();
        }
    }
}
```

```
private void OnTriggerEnter(Collider other)
{
    // Check if the object is a block of wood
    if (other.CompareTag(woodBlockTag))
    {
        SwitchToHighFire();
    }
}


private void SwitchToHighFire()
{
    if (fireAudioSource.clip != highFireClip)
    {
        fireAudioSource.clip = highFireClip;
        fireAudioSource.Play();
    }

    // Reset the timer and enter the high fire state.
    isHighFire = true;
    timer = highFireDuration;
}


private void SwitchToLowFire()
{
    if (fireAudioSource.clip != lowFireClip)
    {
        fireAudioSource.clip = lowFireClip;
        fireAudioSource.Play();
    }
    isHighFire = false;
}
```

Figure 15. Flame sound enhancement code



```
using System.Collections.Generic;
using SteamAudio;
using UnityEngine;


public class AnimalImpactAudio : MonoBehaviour
{
    private AudioSource audioSource;
    private SteamAudioSource steamAudioSource;
    public AudioClip impactSound;

    public List<string> impactTags = new List<string> { "StoneBlock", "WoodBlock" };


    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        steamAudioSource = GetComponent<SteamAudioSource>();

        audioSource.loop = false;
    }


    void OnCollisionEnter(Collision collision)
    {
        if (impactTags.Contains(collision.gameObject.tag))
        {
            if (impactSound != null && !audioSource.isPlaying)
            {
                audioSource.PlayOneShot(impactSound);
            }
        }
    }
}
```

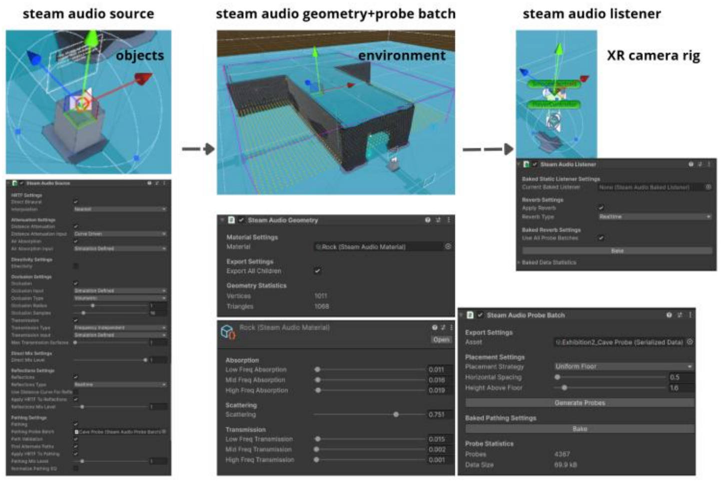
Figure 16. Animal sound effect trigger code

2.2 A hybrid approach to achieve cave sound effects

In addition to the ambient sound design mentioned above, we made the audience distinguish the acoustic environment inside and outside the cave by throwing stones and listening to the sounds. The sound effects inside the cave would be strong reverb and echo/delay, while the sound effects outside the cave would be clean and concise.

2.2.1 Steam audio plugin

To achieve that, we used Steam audio plugin as the primary acoustic engine. We bound the Steam Audio Source to the sound-emitting virtual objects and ensure reflections and pathing settings stay active. Bound to all the walls of the cave, the Steam Audio Geometry component was used to define the geometric structures in the scene. To make the effect more obvious, we attempted to maximize the sound scatterings and absorption rate of the materials inside and outside the cave. Combined with Steam Audio Geometry, the Probe Batch helped to calculate and bake the sound propagation paths for the entire space. We tried different horizonal spacing to test the sound effects. In this way, the listener component attached to the camera rig could receive all audio streams processed by Steam Audio [5].



Steam Audio Source
HRTF Settings
Direct Binaural ✓
Interpolation Nearest
Attenuation Settings
Distance Attenuation ✓
Distance Attenuation Input Curve Driven
Air Absorption ✓
Air Absorption Input Simulation Defined
Directivity Settings
Directivity
Occlusion Settings
Occlusion ✓
Occlusion Input Simulation Defined
Occlusion Type Volumetric
Occlusion Radius 1
Occlusion Samples 16
Transmission ✓
Transmission Type Frequency Independent
Transmission Input Simulation Defined
Max Transmission Surfaces 1
Direct Mix Settings
Direct Mix Level 1
Reflections Settings
Reflections ✓
Reflections Type Realtime
Use Distance Curve For Refle
Apply HRTF To Reflections ✓
Reflections Mix Level 1
Pathing Settings
Pathing ✓
Pathing Probe Batch Cave Probe (Steam Audio Probe Batch)
Path Validation ✓
Find Alternate Paths ✓
Apply HRTF To Pathing ✓
Pathing Mix Level 1
Normalize Pathing EQ

Steam Audio Geometry
Material Settings
Material Rock (Steam Audio Material)
Export Settings
Export All Children ✓
Geometry Statistics
Vertices 1011
Triangles 1068

Rock (Steam Audio Material)
Absorption
Low Freq Absorption 0.011
Mid Freq Absorption 0.018
High Freq Absorption 0.019
Scattering
Scattering 0.751
Transmission
Low Freq Transmission 0.015
Mid Freq Transmission 0.002
High Freq Transmission 0.001

Steam Audio Listener
Baked Static Listener Settings
Current Baked Listener None (Steam Audio Baked Listener)
Reverb Settings
Apply Reverb ✓
Reverb Type Realtime
Baked Reverb Settings
Use All Probe Batches ✓
Bake
Baked Data Statistics

Steam Audio Probe Batch
Export Settings
Asset Exhibition2_Cave Probe (Serialized Data)
Placement Settings
Placement Strategy Uniform Floor
Horizontal Spacing 0.5
Height Above Floor 1.6
Generate Probes
Baked Pathing Settings
Bake
Probe Statistics
Probes 4367
Data Size 69.9 kB

Figure 17. Steam audio reverb generation diagram

While Steam Audio works well in the simple geometric environment like exhibit 3, it has some limitations when dealing with complex environments like exhibit 2. Although steam audio provides a clear reverb following such steps mentioned above, we found its calculated delay/echo effect insufficient to achieve the desired sound. Even when combined with Unity's Audio Reverb Zone, the echo effect we needed remained unremarkable.

2.2.2 Steam Audio reverb combined with pre-processed echo

To solve this problem, we used discussions in the online developer community as a reference. One frequently mentioned approach is to separate 2D reverbs from 3D spatialized dry audio for better control. [6] As a result, the team decided to adopt a hybrid approach combining external preprocessing with custom script switching. Firstly, we preprocessed the original dry audio source in Audacity software. By precisely adding the desired multiple delay/echo effects, we generated an "echo" audio file specifically for the cave interior. We chose to use the Delay effect instead of the Echo effect in Audacity because the Delay effect allows for more precise control over the number of echoes and the interval between them. [7] We set a Delay Time of 0.500 seconds and the Number of Echoes of 3 to ensure that the echoes are distinguishable and repeat multiple times. Finally, we employed a custom script to switch the acoustic zones. When the audience enters the cave, the script triggers our pre-processed sound source, which, combined with the Reverb calculated by Steam Audio, ultimately creates a convincing cave acoustic environment.

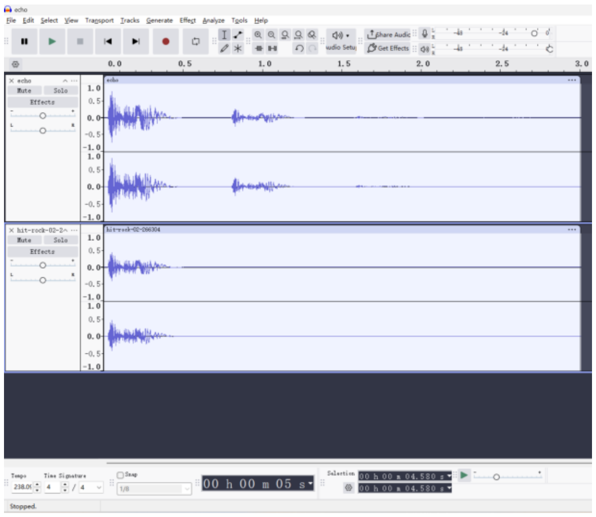
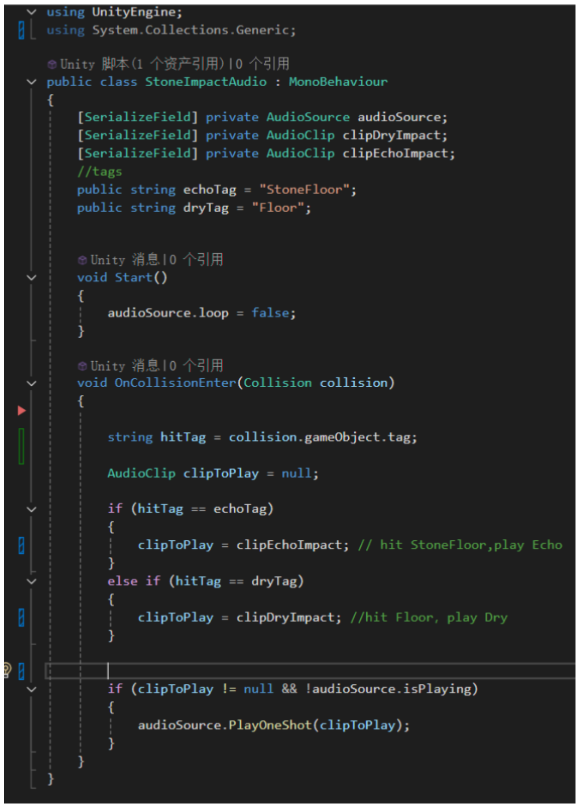


Figure 18. Contrast between echo and dry sound



```
using UnityEngine;
using System.Collections.Generic;

Unity 脚本(1 个资产引用)|0 个引用
public class StoneImpactAudio : MonoBehaviour
{
    [SerializeField] private AudioSource audioSource;
    [SerializeField] private AudioClip clipDryImpact;
    [SerializeField] private AudioClip clipEchoImpact;
    //tags
    public string echoTag = "StoneFloor";
    public string dryTag = "Floor";


    Unity 消息|0 个引用
    void Start()
    {
        audioSource.loop = false;
    }

    Unity 消息|0 个引用
    void OnCollisionEnter(Collision collision)
    {

        string hitTag = collision.gameObject.tag;

        AudioClip clipToPlay = null;

        if (hitTag == echoTag)
        {
            clipToPlay = clipEchoImpact; // hit StoneFloor,play Echo
        }
        else if (hitTag == dryTag)
        {
            clipToPlay = clipDryImpact; //hit Floor, play Dry
        }


        if (clipToPlay != null && !audioSource.isPlaying)
        {
            audioSource.PlayOneShot(clipToPlay);
        }
    }
}
```

Figure 19. Echo switching script

3. Exhibit 3: Rolling Balls

3.1 User Interface and Scene Setup

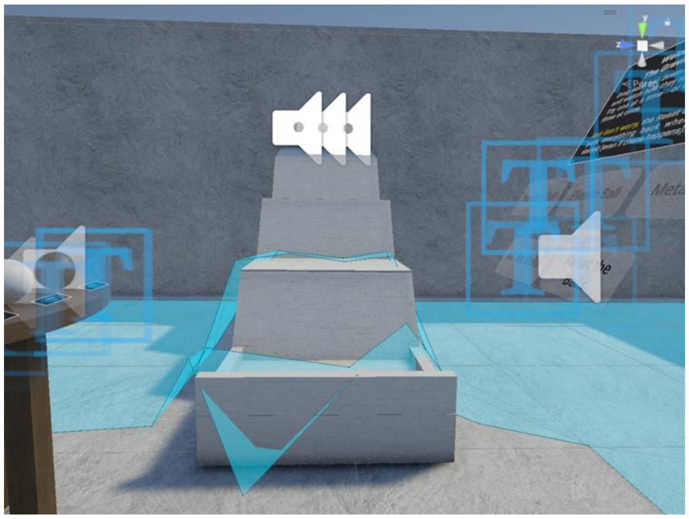
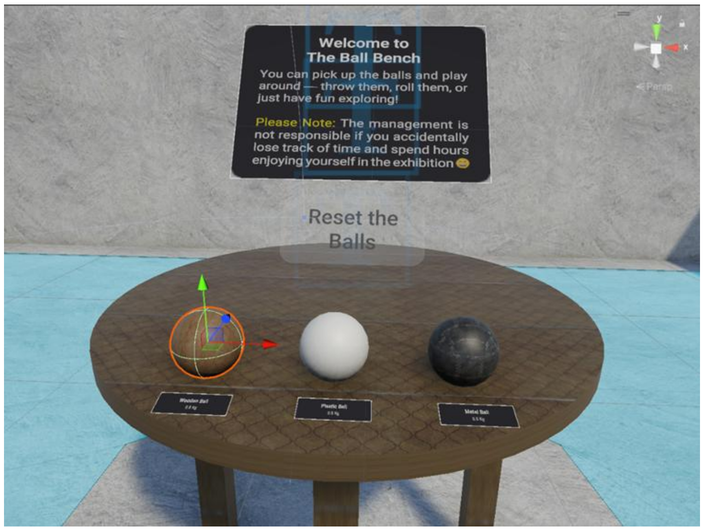


Figure 20. The marble run in the exhibit 03 with wood, plastic and metal balls



Welcome to The Ball Bench

You can pick up the balls and play around — throw them, roll them, or just have fun exploring!

Please Note: The management is not responsible if you accidentally lose track of time and spend hours enjoying yourself in the exhibition 😊

Figure 21. The table in exhibit 03 with wood, plastic and metal balls

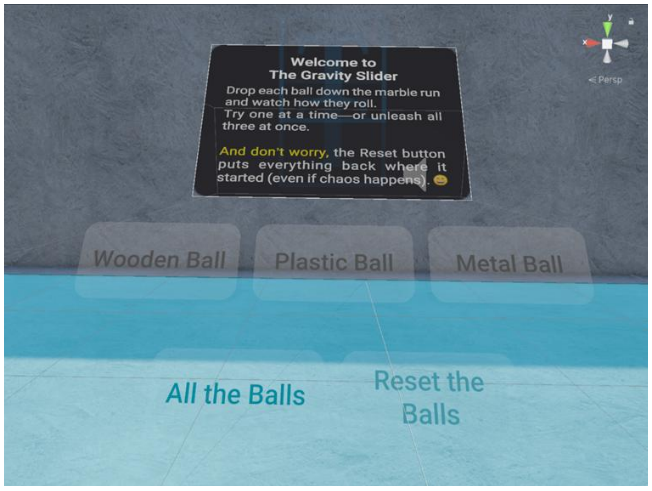
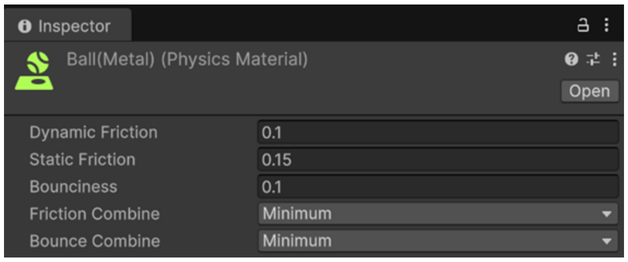

Welcome to
The Gravity Slider
Drop each ball down the marble run and watch how they roll.
Try one at a time—or unleash all three at once.
And don't worry, the Reset button puts everything back where it started (even if chaos happens).
Wooden Ball
Plastic Ball
Metal Ball
All the Balls
Reset the Balls

Figure 22. The buttons and text box in the exhibit 03 for the users to interact with, which is associated with the ball's behaviour in the marble run

By using a combination of physics-based objects and intuitive User-Interaction controls, the scene was developed to provide an immersive experience based around Interactive Sound Objects – demonstrating the ability to create convincing sonic environments. Just straight to the user's view when they land on the scene is the marble run named as 'The Gravity Slider', with three balls of wood, plastic and metal properties. The marble run was made using cubes combined, and the whole of marble run as a group was assigned a physics material component to make the balls interact similar to the real-world interaction. Adjacent to the marble run, there is a textbox, guiding the users to which and how they can interact with other gameObjects in the scene. Along with the textbox, there are five buttons available in two rows. The first row contains three buttons, each button for each ball. Clicking the button releases the respective ball down the marble run. The second row with two buttons, one to release all three balls down the marble run at the same time and the other button to reset the balls to their original position, which is on top of the marble run. Adjacent to the marble run, and opposite to these buttons is 'The Ball Bench'. The Ball Bench is a table with three balls of wood, plastic, and metal materials. The balls are attached with text boxes saying what ball they are, and what are their masses helping the users know about the balls, making the interactions understandable by relating it with real-world interactions. Similar to the previous textbox, there is another text box placed above the table, to help the users know what and how they can interact with other gameObjects. The users can interact, play, and spend time with the balls by grabbing them from the table to learn about the physical characteristics of the materials. There is a button placed between the table and the textbox to reset the positions of the balls to their respective original positions. The scene also contains two buttons for users to teleport to the Exhibit 01 scene and to the Exhibit 02 scene, facilitating smooth navigation across the virtual exhibition space. Together, these UI components are carefully arranged to create a seamless and engaging experience, combining intuitive control, clear guidance, and interactive physics exploration, allowing users to fully engage with the objects, understand material properties, and enjoy a dynamic and playful environment without confusion or disruption. All the buttons in the scene were the Hover Buttons, used from the Meta Interaction SDK's sample scenes. The Hover Buttons have Poke Interactable functions inside them as children. [19] Poke interactions let you interact with surfaces via direct touch using hands, controller driven hands, and controllers.

3.2 Properties of the ball

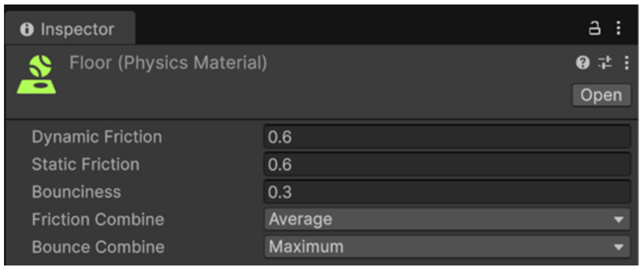
3.2.1 Physics materials



Ball(Metal) (Physics Material)

Dynamic Friction	0.1
Static Friction	0.15
Bounciness	0.1
Friction Combine	Minimum
Bounce Combine	Minimum

Figure 23&24. Top: Physics Material values used for the balls
Bottom: Physics Material values used for the floor, walls and the marble run



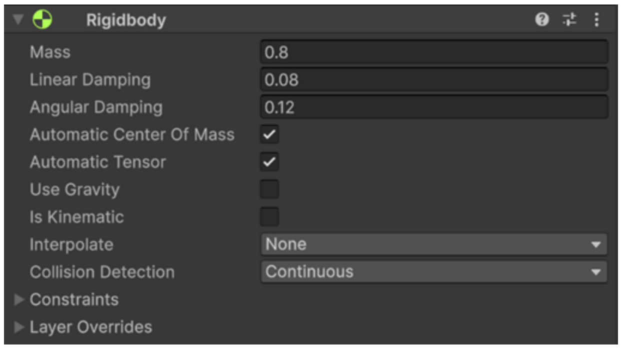
Floor (Physics Material)

Dynamic Friction	0.6
Static Friction	0.6
Bounciness	0.3
Friction Combine	Average
Bounce Combine	Maximum

[15] helped to learn what is the 'Physics material' component in Unity, what is it used for, how to implement

it, how it impacts a gameObject's interaction with other gameObjects. The balls were assigned physics materials respective to their surface materials (wood, plastic, metal). Additional to the balls, the 'Physics material' component was also assigned to the floor, walls, and the marble run in the process to replicate the ball's interaction with other gameObjects similar to that of a ball's interaction with floor, and wall in real-world.

3.2.2 Rigidbody properties



Rigidbody

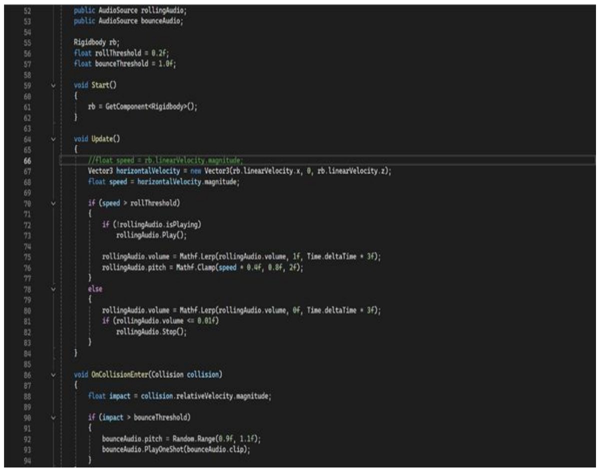
Mass	0.8
Linear Damping	0.08
Angular Damping	0.12
Automatic Center Of Mass	✓
Automatic Tensor	✓
Use Gravity	
Is Kinematic	
Interpolate	None
Collision Detection	Continuous
Constraints	
Layer Overrides	

Figure 25. An image of the Rigidbody settings used on the balls to simulate real-world behaviour

In the view of improving the interaction between the balls and ither gameObjects simulating realistic physical behaviour, the mass, linear damping, and angular damping values were assigned according to the different surface materials of the balls. The mass determines how heavy it is to lift, throw, and fall on the ground, which directly affects how strongly it reacts to interactions. The linear damping decides how quickly the ball slows down when it moves, which is to avoid unnatural spinning. The angular damping is useful to gradually reduce the ball's continuous spinning and rotation instead of spinning infinitely.

3.3 Scripting

3.3.1 Interaction with balls in the table

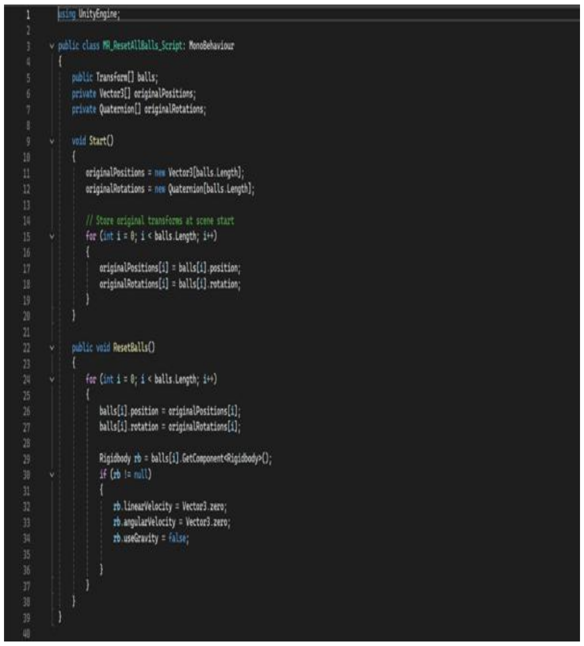


```
public AudioSource rollingAudio;
public AudioSource bounceAudio;

Rigidbody rb;
float rollThreshold = 0.2f;
float bounceThreshold = 1.0f;

void Start()
{
    rb = GetComponent<Rigidbody>();
}

void Update()
{
    //float speed = rb.linearVelocity.magnitude;
    Vector3 horizontalVelocity = new Vector3(rb.linearVelocity.x, 0, rb.linearVelocity.z);
    float speed = horizontalVelocity.magnitude;

    if (speed > rollThreshold)
    {
        if (!rollingAudio.isPlaying)
            rollingAudio.Play();

        rollingAudio.volume = Mathf.Lerp(rollingAudio.volume, 1f, Time.deltaTime * 3f);
        rollingAudio.pitch = Mathf.Clamp(speed * 0.4f, 0.8f, 2f);
    }
    else
    {
        rollingAudio.volume = Mathf.Lerp(rollingAudio.volume, 0f, Time.deltaTime * 3f);
        if (rollingAudio.volume <= 0.01f)
            rollingAudio.Stop();
    }
}

void OnCollisionEnter(Collision collision)
{
    float impact = collision.relativeVelocity.magnitude;

    if (impact > bounceThreshold)
    {
        bounceAudio.pitch = Random.Range(0.9f, 1.1f);
        bounceAudio.PlayOneShot(bounceAudio.clip);
    }
```

Figure 26. A screenshot of the script used for user's interaction with the balls on the table

Two audio sources, one for ball dropping and the other for rolling audio were initialized. Along with that, threshold values for comparing the ball's speed to decide if the audio has to be played or not, were initialized. Here, the rolling threshold was set at '0.2f' and drop threshold was set at '1.0f'. In the Start() method, the rigidbody component of the ball is grabbed. In the Update() method, the horizontal velocity of the ball is calculated by using 'Vector3' with the (x, y, z) parameters being (velocity in the X direction, 0, velocity in the Z direction), "(rb.linearVelocity.x, 0, rb.linearVelocity.z)". The speed of the ball is calculated by calculating the magnitude of horizontal velocity. The speed is compared to the rolling threshold, and if the speed exceeds the threshold, the rolling audio plays and gradually fades in using Mathf.Lerp and adjusts its pitch according to the speed, making the sound realistically increase or decrease as the ball rolls faster or slower. If the speed drops below the threshold, the audio smoothly fades out and stops once nearly silent. The OnCollisionEnter() method detects impacts and checks if the collision force exceeds the bounceThreshold. When it does, a bounce sound is played using PlayOneShot, with a slight random pitch variation to avoid repetitive audio and make each impact feel more natural. The script provides dynamic and responsive audio behaviour that enhances realism for rolling and bouncing balls in the scene.

3.3.2 Resetting balls



```csharp
using UnityEngine;

public class MR_ResetAllBalls_Script: MonoBehaviour
{
    public Transform[] balls;
    private Vector3[] originalPositions;
    private Quaternion[] originalRotations;

    void Start()
    {
        originalPositions = new Vector3[balls.Length];
        originalRotations = new Quaternion[balls.Length];

        // Store original transforms at scene start
        for (int i = 0; i < balls.Length; i++)
        {
            originalPositions[i] = balls[i].position;
            originalRotations[i] = balls[i].rotation;
        }
    }

    public void ResetBalls()
    {
        for (int i = 0; i < balls.Length; i++)
        {
            balls[i].position = originalPositions[i];
            balls[i].rotation = originalRotations[i];

            Rigidbody rb = balls[i].GetComponent<Rigidbody>();
            if (rb != null)
            {
                rb.linearVelocity = Vector3.zero;
                rb.angularVelocity = Vector3.zero;
                rb.useGravity = false;

            }
        }
    }
}
```

Figure 27. A screenshot of the script used for resetting the balls back to their original positions

Using three arrays to store the number of balls, the original positions of the balls, the original rotation of the balls. The 'Transform' component is used to store the position, rotation, scale of the ball in the scene. The 'Vector3' component was used to store the position of the balls at the start of the scene. The 'Quaternion' component is used to store the rotation of the balls at the start of the scene. The Start() method, runs a 'for' loop to store the original position and rotation of the ball at the start of the scene. When the Reset button is pressed, the Reset method is called, which changes the current position and rotation of the balls to the original position and rotation of the balls which was stored at the start of the scene. Then the ball's rigidbody component is grabbed and the 'linearvelocity and angularvelocity' is made zero so that the ball does not move due to it's leftover momentum. Here the gravity of the ball is set at 'true'.

For resetting the balls on the marble run, the same methodology is followed, with one change, where the gravity of the ball is set to 'false', so that the ball does not fall as soon as the balls are placed on their respective original positions.

3.3.3 Dropping the balls from the marble run

To drop the balls from the marble run, the rigidbody component of the balls were grabbed and the gravity of the ball was made 'true'.

3.4 Audio and Steam Audio Integration

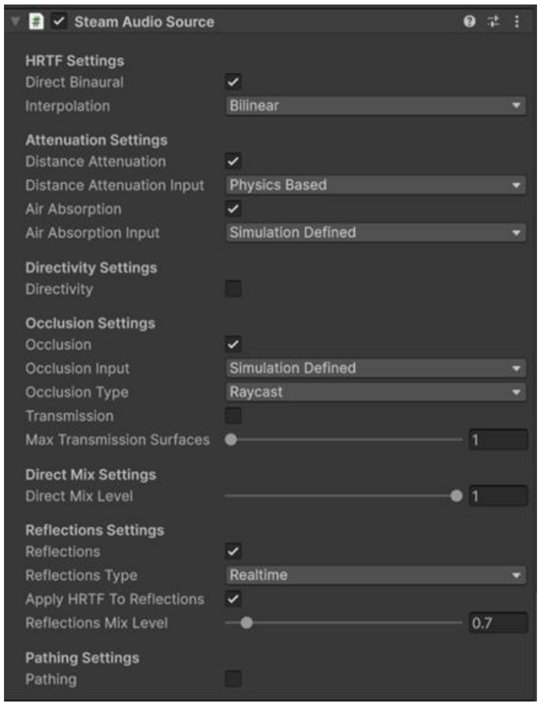
3.4.1 Audio Sources

The websites[16] [17] were helpful in searching and picking audio sources to use them in the scene for the dropping and rolling audios of wood, plastic and metal balls.

3.4.2 Audio Editing

The audio files that were downloaded had to be edited by trimming them by closely listening to the audio so that it feels natural when a ball collides with a gameObject and when a ball rolls. The audio files also had to be edited by inserting fade in and fade out according to the purpose of the audio and the ball's material in the process of making the experience more immersive and real-world. [18] was helpful for this work.

3.4.3 Steam Audio

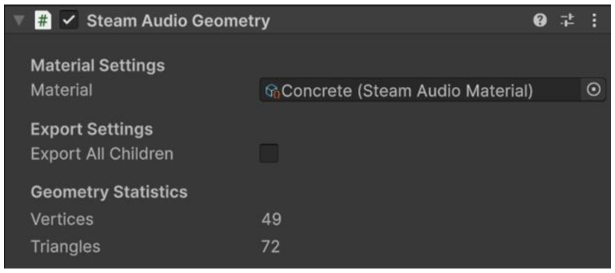


Steam Audio Source ✓

HRTF Settings	
Direct Binaural	✓
Interpolation	Bilinear
Attenuation Settings	
Distance Attenuation	✓
Distance Attenuation Input	Physics Based
Air Absorption	✓
Air Absorption Input	Simulation Defined
Directivity Settings	
Directivity	
Occlusion Settings	
Occlusion	✓
Occlusion Input	Simulation Defined
Occlusion Type	Raycast
Transmission	
Max Transmission Surfaces	1
Direct Mix Settings	
Direct Mix Level	1
Reflections Settings	
Reflections	✓
Reflections Type	Realtime
Apply HRTF To Reflections	✓
Reflections Mix Level	0.7
Pathing Settings	
Pathing	

Figure 28. The Steam Audio Source settings that were used on the balls

The Steam Audio was integrated into this scene and the project to make the audio more realistic and physically accurate 3D sound, making the audio and the experience immersive rather than just simple audios. This helps to play sound like how it behaves in a physical space rather than just playing them just like that. Rather than playing the audio equally to both the ears, the Steam Audio calculates how audio bounces, reflects, gets blocked, and loses energy depending on the environment's walls, floor, slope, and the materials attached to them in the track to simulate how sound behaves in a real physical space. Steam Audio was integrated into the project by enabling spatialized audio on all ball-related sound sources, activating HRTF-based binaural rendering for accurate 3D directional sound, and turning on real-time reflections to allow rolling and bouncing audio to interact naturally with surrounding surfaces. To simulate realistic sound fall-off with distance, to bring in muffling behind obstacles, for high frequency loss over distance, the Distance attenuation, occlusion, and air-absorption settings were enabled in the Steam Audio Source.


Steam Audio Geometry

Material Settings
Material: Concrete (Steam Audio Material)

Export Settings
Export All Children: ☐

Geometry Statistics
Vertices: 49
Triangles: 72

Figure 29. The Steam Audio Geometry settings that were used on the walls, floor and the marble run

With appropriate changes, the floor, marble run, table, and the walls were assigned Steam Audio Geometry where the material was given as Concrete, so that the reflection and absorptions behaved like how the materials of the balls will behave with concrete in the real-world. To improve the accuracy and performance, the scene's acoustics were baked. All these changes in the Steam Audio were made to make the audio sources in the scene feel immersive and simulate real-world natural sounding when a user experiences the scene.

3.5 Texturing

[20] The gameObjects in the scene were textured using high-quality materials sourced from the Unity Asset Store, allowing each gameObject in the scene to have realistic surface appearances. These textures enhanced the visuals of the scene by accurately representing materials like wood, metal, plastic, and concrete, making the environment more immersive and visually good.

3.6 Development Cycle

3.6.1 Cycle 01: VR Environment and Navigation

The first development cycle focused on creating the VR environment setup by creating the floor, walls, marble run, balls, and table. These gameObjects were also assigned their respective physics materials such as concrete, wood, plastic, and metal. Each ball's rigidbody component was edited to make sure it collides and reacts in a way similar to real-world interaction. The UnityXRCameraRig was integrated to enable smooth tracking. Smooth and teleport navigation were implemented to move around the environment. The XR Grab Interaction was integrated to introduce a grabbing function to enable the user to pick the ball and interact with it.

3.6.2 Cycle 02: Audio components and Scripting

The second development cycle focused on audio components and scripts. Finding good audio files, trimming and editing, and importing them into Unity. Steam Audio was integrated to make the audio spatial make the experience immersive. The scripts were written for controlling the ball's behaviour and to interact with the balls with ease, via buttons to release them and reset them to their original positions. This cycle was to make sure that the environment becomes immersive acoustically.

3.6.3 Cycle 03: User Interface and Texturing

The third development cycle focused on improving the visual quality of the environment by adding buttons to release and reset the balls. Text boxes were added to guide users on interaction with gameObjects in the scene. By using the Unity Asset Store, textures were downloaded, imported into Unity, and the gameObjects were assigned respective textures, giving a realistic appearance. This cycle was to make sure that the quality of the environment in terms of visuals are good enough to enable immersiveness for the users.

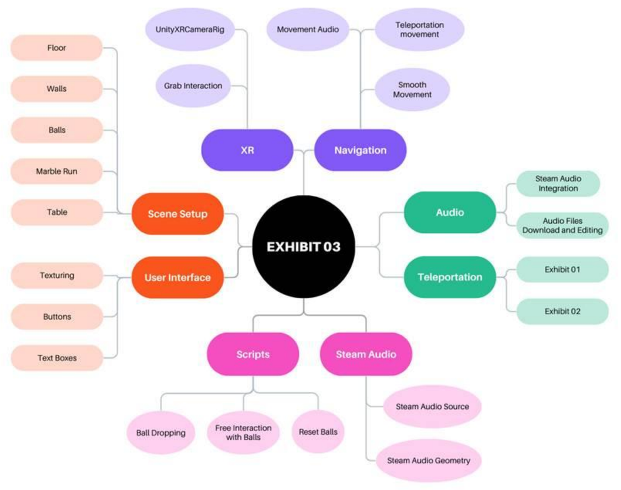


Figure 30. The development cycle used for the development of exhibit 03

Critical Evaluation

Exhibit 1: Resonance

The exhibit largely achieved its intended goal, but several limitations remain. Peer reviews and presentation demonstrations showed that the resonance interaction generated strong excitement and novelty—users enjoyed the experience of breaking a virtual glass using their own voice.

However, frustration was equally strong when the resonance failed to trigger. Several users, including professors and team members, struggled to activate the system. While reducing the threshold could improve accessibility, it cannot simply be lowered without consequence. If the threshold becomes too low, any sound could trigger the effect, undermining the scientific credibility of the physical phenomenon being represented. Therefore, the most crucial challenge is to find a balanced threshold that preserves both scientific accuracy and user enjoyment.

Among the technical issues that contributed to negative experiences, the Unity Animator was a major source of unresponsiveness. When users adjusted their volume too quickly or moved the amplifier toward the glass at high speed, the system occasionally failed to play the shattering animation in time. In some cases, the breaking sound effect disappeared entirely and only returned after a full reset. These issues occurred intermittently and remain difficult to resolve.

Another issue is that the physical simulation of the glass is imprecise: collision detection is not fully accurate, and the glass does not break upon hitting the ground. In audio processing, the system is still unable to generate a smooth, frequency-enhanced signal that blends harmonically with the user's voice, which remains a notable shortcoming.

Exhibit 2: Cave

Our audiences can identify the difference in reverb time and echo density between the cave exterior and interior, which successfully provides an immediate and clear environmental distinction. It aligns the primary goal of the scene.

However, the current implementation lacks the necessary difference in echoes for different spatial zones within the cave. The similarity of the acoustic feedback between the entrance, intermediate breaks, and the deepest parts of the cave fails to accurately convey the changes in cavern geometry and volume. To improve the experience, we could implement multiple, distinct Acoustic Presets linked to different parts of the cave geometry.

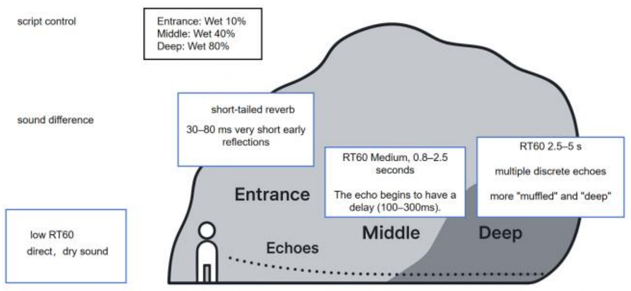


Figure 31. Cave sound field diagram

Furthermore, while 3D attenuation was used for the wind and water dripping sounds at the cave entrance, the fade-in/fade-out effects lacked the necessary smooth transitions. We can use scripted transitions based on Audio Mixer snapshots or custom curves.

Exhibit 3: Rolling Balls

The users were able to identify and know the differences between different materials with the help of interaction between different balls of wood, plastic, and metal with the walls, floor, and marble run which is of concrete. From a user-experience perspective, the inclusion of clear visual guidance through buttons and text panels supports accessibility and reduces user effort, particularly helping users understand what the scene has and how the users can interact with the gameObjects. However, the experience can be influenced by factors such as controller tracking accuracy, UI interaction sensitivity, and hence the need for more careful calibration of physics and audio parameters to avoid unintended behavior.

Conclusion

Sonic Playground demonstrates how virtual reality can be used to communicate acoustic principles through direct interaction and spatial audio design. By combining three exhibits focused on resonance, environmental acoustics, and material-dependent sound, the project enables users to experience how sound responds to frequency, space, material, and physical interaction rather than learning these concepts theoretically.

Throughout development, the team gained practical experience with Unity XR interaction, physics-based object behaviour, and spatial audio implementation using Steam Audio. The project highlighted the importance of balancing technical accuracy with clarity and accessibility, particularly when translating real-world acoustic phenomena into interactive VR experiences.

While the project successfully delivers clear and engaging audio demonstrations, limitations remain in areas such as resonance responsiveness, acoustic variation within complex environments, and interaction robustness. With additional development time, the project could be improved through refined audio–physics synchronisation, smoother environmental transitions, and more advanced acoustic modeling. Overall, Sonic Playground provides a strong foundation for immersive, audio-driven learning experiences and demonstrates the potential of VR as a medium for acoustic exploration.

Reference

[1] Zhao, W., Li, Q., Wang, Y. et al. Acoustics of karst tourist caves: a case study in Guizhou Province, China. Sci Rep 15, 42067 (2025). https://doi.org/10.1038/s41598-025-26251-2

[2] Mojang Studios. 2011. Minecraft. Microsoft Game Studios. https://www.minecraft.net/

[3] Constantin Popp and Damian T. Murphy. 2022. Creating Audio Object-Focused Acoustic Environments for Room-Scale Virtual Reality. Applied Sciences 12, 14 (2022), 7306. https://doi.org/10.3390/app12147306

[4] N. Binetti, L. Wu, S. Chen, E. Kruijff, S. Julier, and D. P. Brumby. 2021. Using visual and auditory cues to locate out-of-view objects in head-mounted augmented reality. Displays 69 (2021), 102032. https://doi-org.bris.idm.oclc.org/10.1016/j.displa.2021.102032

[5] Valve Software. 2025. Steam Audio Unity Integration Documentation. Accessed 11 Dec 2025. https://valvesoftware.github.io/steam-audio/doc/unity/guide.html

[6] Reddit user u/later_oscillator. 2024. Comment on separating 2D reverbs from 3D spatialized dry audio for better control. Posted in r/GameAudio. Accessed 12 Dec 2025. https://www.reddit.com/r/GameAudio/comments/1hbbhgo/how_valid_is_bakedin_reverb_for_game_audio_assets/

[7] Audacity Team. 2024. Echo and Delay Effects Documentation. Accessed 11 Dec 2025. https://manual.audacityteam.org/

[8] PyTorch Team. *Audio Feature Extractions — Torchaudio 2.8.0 Documentation*. PyTorch (2025). Available at: https://docs.pytorch.org/audio/2.8/tutorials/audio_feature_extractions_tutorial.html

[9] Wikipedia Contributors. *Cooley–Tukey FFT Algorithm*. Wikipedia (2025). Available at: https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm

[10] Jake Newman, Audio Processing, Lecture presented at University of East Anglia (2024).

[11] StackOverflow Contributors. Detect a specific frequency/tone from raw wave-data. StackOverflow (2011). Available at: https://stackoverflow.com/questions/4808893/detect-a-specific-frequency-tone-from-raw-wave-data

[12] Wikipedia Contributors. Goertzel Algorithm. Wikipedia (2025). Available at: https://en.wikipedia.org/wiki/Goertzel_algorithm

[13] Lampropoulos, G., Kinshuk Virtual reality and gamification in education: a systematic review. *Education Tech Research Dev* **72**, 1691–1785 (2024). https://doi.org/10.1007/s11423-024-10351-3.

[14] Ranc, W., Nguyen, T., Yu, L., Zhang, Y., Kim, M., Huang, H., Yu, L.F. and Contributors, E., Multi-Player VR Marble Run Game for Physics Co-Learning. Available at: https://www.researchgate.net/profile/Liuchuan-Yu/publication/397738178_Multi-Player_VR_Marble_Run_Game_for_Physics_Co-Learning/links/691f473a19b35058639bb060/Multi-Player-VR-Marble-Run-Game-for-Physics-Co-Learning.pdf

[15] Overgaard, J. (2018). *Rapid Unity Tutorials #1: Physics Materials*. [online] Medium. Available at: https://medium.com/sun-dog-studios/rapid-unity-tutorials-1-physics-materials-68758351fd8a [Accessed 13 Dec. 2025].

[16] Pixabay (2010). *Pixabay*. [online] Pixabay.com. Available at: https://pixabay.com/.

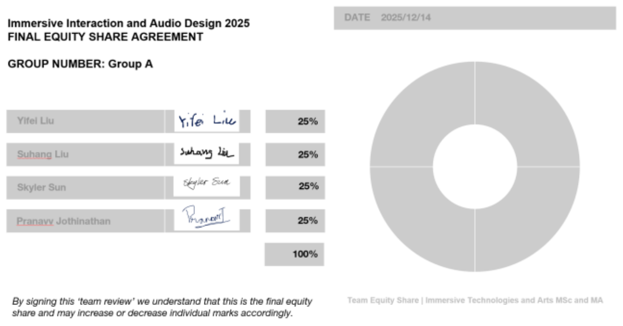
[17] Freesound (2012). *Freesound*. [online] Freesound.org. Available at: https://freesound.org/.

[18] Main Website. (2020). *Online Mp3 Cutter - Audio Trimmer*. [online] Available at: https://audiotrimmer.com/

[19] Meta.com. (2024). *Meta Developers*. [online] Available at: https://developers.meta.com/horizon/documentation/unity/unity-isdk-poke-interaction/

[20] Unity (2000). *Unity Asset Store - The Best Assets for Game Making*. [online] @UnityAssetStore. Available at: https://assetstore.unity.com/.

Final equity share agreement



Immersive Interaction and Audio Design 2025
FINAL EQUITY SHARE AGREEMENT

DATE 2025/12/14

GROUP NUMBER: Group A

Yifei Liu	Yifei Liu	25%
Suhang Liu	Suhang Liu	25%
Skyler Sun	Skyler Sun	25%
Pranavv Jothinathan	[illegible]	25%
		100%

By signing this 'team review' we understand that this is the final equity share and may increase or decrease individual marks accordingly.

Team Equity Share | Immersive Technologies and Arts MSc and MA