



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

MASTER THESIS IN ICT FOR INTERNET AND MULTIMEDIA

Interactive sound simulation in Unity: the case study of an ancient clay rattle

MASTER CANDIDATE

Tahsin Can Yilmaz

Student ID 2080613

SUPERVISOR

Prof. Antonio Rodà

University of Padova

CO-SUPERVISOR

Giulio Pitteri

University of Padova

ACADEMIC YEAR
2024/2025

*To my family
for never making me feel alone, not even for a single day. I get my strength from you.*

*To my friends
during my master's program. We overcame all the difficulties together, shared meals,
and laughed together.*

Thank you all for being in my life.

Abstract

The preservation of cultural heritage in museums often renders historical artifacts as static objects for visual inspection only. This is particularly problematic for musical instruments, whose primary cultural function—the creation of sound through physical interaction—is lost. This thesis addresses the challenge of this "interaction gap" by presenting the design and implementation of a complete, low-cost system for the creation of a playable and expressive digital reconstruction of a Roman percussion instrument.

The methodology integrates a custom-built hardware interface with a real-time software simulation. The hardware component consists of an ESP32 microcontroller and an MPU-6500 Inertial Measurement Unit (IMU), which captures the user's gestural performance data. This data is transmitted wirelessly to a host computer using the Bluetooth Low Energy (BLE) protocol.

The software component, developed in the Unity engine, is responsible for the audio-visual experience. It receives the incoming BLE data stream and uses it for two simultaneous purposes: to drive the real-time rotation of a 3D model of the instrument, providing immediate visual feedback, and to feed that rotation information to a custom procedural audio engine.

The most important part of this work is the sound generation. The system does not play pre-recorded sound files. It creates the sound in real time with a procedural audio engine. This engine is a C# script that runs a mathematical model of the instrument's physics. The user's movements, like the speed and sharpness of the rotation, are the direct inputs for this model. This method creates a strong connection between the user's action and the sound. The result is a dynamic and expressive digital instrument. This project is a successful proof-of-concept. It shows a practical method for creating low-cost, interactive exhibits that can make cultural heritage more interesting for people.

Sommario

La conservazione del patrimonio culturale nei musei rende spesso i reperti storici oggetti statici per la sola ispezione visiva. Questo è particolarmente problematico per gli strumenti musicali, la cui funzione culturale primaria—la creazione di suono attraverso l'interazione fisica—viene persa. Questa tesi affronta la sfida di questo "vuoto di interazione" presentando la progettazione e l'implementazione di un sistema completo e a basso costo per la creazione di una ricostruzione digitale suonabile ed espressiva di uno strumento a percussione romano.

La metodologia integra un'interfaccia hardware su misura con una simulazione software in tempo reale. La componente hardware consiste in un microcontrollore ESP32 e un'Unità di Misura Inerziale (IMU) MPU-6500, che cattura i dati gestuali dell'utente. Questi dati vengono trasmessi in modalità wireless a un computer host utilizzando il protocollo Bluetooth Low Energy (BLE).

La componente software, sviluppata nel motore Unity, è responsabile dell'esperienza audiovisiva. Riceve il flusso di dati BLE in ingresso e lo utilizza per due scopi simultanei: guidare la rotazione in tempo reale di un modello 3D dello strumento, fornendo un feedback visivo immediato, e fornire energia a un motore audio procedurale personalizzato.

La parte più importante di questo lavoro è la generazione del suono. Il sistema non riproduce file audio pre-registrati. Crea invece il suono in tempo reale con un motore audio procedurale. Questo motore è uno script C# che esegue un modello matematico della fisica dello strumento. I movimenti dell'utente, come la velocità e la rapidità della rotazione, sono gli input diretti per questo modello. Questo metodo crea una forte connessione tra l'azione dell'utente e il suono. Il risultato è uno strumento digitale dinamico ed espressivo. Questo progetto è un proof-of-concept di successo. Mostra un metodo pratico per creare exhibit interattivi a basso costo che possono rendere il patrimonio culturale più interessante per le persone.

Contents

List of Figures	xi
List of Tables	xiii
List of Algorithms	xv
List of Acronyms	xvii
1 Introduction	1
1.1 Problem Statement	1
1.2 Technological Developments and Opportunities	2
1.3 Research Goal and Approach	2
1.4 Contribution of This Thesis	3
1.5 Thesis Outline	3
2 Background	5
2.1 Digital Reconstruction in Cultural Heritage	5
2.2 Digital Pathways to Ancient Acoustics	5
2.3 From Static Models to Interactive Experiences	6
2.4 Acoustic Context: Environments and Soundscapes	7
3 System Architecture	9
3.1 Overview	9
3.2 Hardware Components	10
3.2.1 Microcontroller: ESP32-WROOM-32	10
3.2.2 Sensor: MPU-6500 IMU	12
3.2.3 Wireless Communication: Bluetooth Low Energy (BLE)	14
3.3 Software Platform: Unity	14

CONTENTS

4	Software Development and Implementation	17
4.1	Firmware Implementation	17
4.1.1	Libraries and Global Variables	18
4.1.2	Setup Function	19
4.1.3	The Main Loop: Reading, Filtering, and Sending Data . . .	20
4.2	Software Design	21
4.2.1	BLE Communication in Unity	22
4.2.2	3D Model Visualization and Control	27
4.2.3	Procedural Audio Synthesis	28
5	Results and Evaluation	37
5.1	System Functionality Test	37
5.2	Audio Engine Evaluation	38
5.3	System Capabilities and Potential	39
6	Conclusions	41
6.1	Summary of Work	41
6.2	Main Contributions	42
6.3	Limitations	43
6.3.1	Hardware Prototype	43
6.3.2	The Sound is an Approximation	43
6.3.3	Interaction Based Only on Rotation	43
6.3.4	The Visual Model is a Placeholder	44
6.3.5	Lack of User Evaluation	44
6.4	Future Works	44
6.4.1	Build a Controller	45
6.4.2	Make the Sound Better	45
6.4.3	Add More Shaking Gestures	45
6.4.4	Create a 3D Model of the Instrument	46
6.4.5	Experiment with Museum Visitors	46
	References	47
	Acknowledgments	51

List of Figures

3.1	The project hardware setup on a breadboard, featuring the ESP32-WROOM-32 development board and the MPU-6500 IMU sensor. .	11
4.1	The Unity Editor interface for this project. On the left is the scene view. In the center is the object hierarchy. On the right, the Inspector window shows the public parameters for the BLE Reader and Shaker Synth scripts.	22
4.2	The final render of the 3D instrument model in the Unity scene. The Standard Shader provides realistic lighting and a smooth surface.	28
4.3	The public parameters of the ShakerSynth.cs script as they appear in the Unity Inspector. These values can be changed in real-time to adjust the sound.	31

List of Tables

3.1	ESP32 Configuration Parameters	12
3.2	MPU-6500 Sensor Configuration and Parameters	13

List of Algorithms

1	Pseudocode for Firmware Libraries and Global Definitions	18
2	Pseudocode for the setup() function.	19
3	Pseudocode for the main loop() function.	20
4	Pseudocode for BLEReader class variables.	24
5	Pseudocode for Start and Update methods.	25
6	Pseudocode for the core BleLoop() logic.	26
7	Pseudocode for ShakerSynth public parameters and initialization.	30
8	Pseudocode for the Drive() method.	32
9	Pseudocode for the OnAudioFilterRead() audio generation.	33
10	Pseudocode for the RecalcFilter() method.	34
11	Pseudocode for the rolling sound generation.	35
12	Pseudocode for managing audio effects.	36

List of Acronyms

BLE Bluetooth Low Energy

DLL Dynamic Link Library

DSP Digital Signal Processing

GAP Generic Access Profile

GATT Generic Attribute Profile

I2C Inter-Integrated Circuit

IDE Integrated Development Environment

IMU Inertial Measurement Unit

PCB Printed Circuit Board

UUID Universally Unique Identifier

WinRT Windows Runtime



Introduction

1.1 PROBLEM STATEMENT

Digitally preserving a historical musical instrument as a 3D model captures its visual form, but misses its most important quality: its sound. According to foundational texts on heritage, such as the UNESCO convention, the ‘intangible’ aspects of an object—like the techniques used to play it—are as crucial as the physical object itself [1]. A static model can be looked at, but it cannot be played. This approach preserves the object as a silent artifact, fundamentally failing to represent its function as a tool for making music through physical interaction.

This limitation represents a gap in digital reconstruction methods. This is also a problem discussed in museum studies. Digital tools are seen as a way to ‘bridge the gap’ and create more social and inclusive exhibits [2]. The common solution is to use pre-recorded sound samples, where an interaction triggers a specific audio file. This approach is not interactive in a meaningful way. The sound does not respond dynamically to how the user handles the virtual instrument; a fast shake might sound the same as a slow one. The direct, real-time feedback loop between a player’s gesture and the resulting sound—the core of a musical performance—is broken. This thesis addresses this gap by building a system that restores this link. It uses a physical controller and real-time sound synthesis to recreate the interactive experience of playing the instrument. Recent work also shows that sound can support access and social inclusion in museum settings [3], [4].

1.2. TECHNOLOGICAL DEVELOPMENTS AND OPPORTUNITIES

This project presents a system for the interactive digital reconstruction of a ancient percussion instrument. It has two main parts. The first part is a custom hardware controller. It uses an Inertial Measurement Unit (IMU) to track the user's hand movements. The second part is a software application. It takes the movement data and uses it to generate sound in real-time. This is called procedural audio synthesis.

1.2 TECHNOLOGICAL DEVELOPMENTS AND OPPORTUNITIES

Modern technology can fix this problem. It is possible to make 3D models of old objects. These models can be interactive. The Unity game engine is a good tool for this. It has a physics system and an audio system.

The project also uses cheap electronics. An ESP32 microcontroller reads data from an IMU sensor. This sensor measures how the user moves the device. The ESP32 sends this movement data to a computer. It uses Bluetooth Low Energy (BLE) for the wireless connection. The computer runs the Unity application, and a script gets the data. This hardware setup works like a physical controller for the digital object. When a person moves the controller, the 3D model on the screen moves and makes sound at the same time. This creates a direct connection between the user and the instrument. In museum practice, sound is also curated content, not just output [5].

1.3 RESEARCH GOAL AND APPROACH

The goal of this thesis is to build an interactive system that simulates a ancient ellipsoidal percussion instrument. The system is made in Unity and has two parts. The first part is a physical device with an IMU sensor. This device sends movement data over BLE. The second part is the Unity application. It gets the data and uses it to control a 3D model of the instrument.

How the sound is made is the most important part of this project. The system does not use pre-recorded sound files. Instead, it uses procedural audio synthesis. This means the sound is generated in real time from a physics model. The user's gestures have direct control over this sound model. Fast shaking creates sharp, high-frequency sounds. Slow tilting creates soft, low-frequency rolling sounds. This method makes the interaction feel much more real and

dynamic than using a simple recording.

1.4 CONTRIBUTION OF THIS THESIS

This thesis builds on work from several fields. It uses the methods of digital reconstruction from Sun et al. [6] and Avanzini et al. [7]. It is also inspired by the move to create more interactive museum exhibits, as discussed by Roberts et al. [8].

The main contribution of this work is the specific way it integrates these ideas. Previous research has focused on two separate goals. One goal is to create accurate, but static, digital copies of instruments. The other goal is to create general interactive exhibits. This project connects those two goals.

The novelty is in the complete system architecture. This project combines three key parts.

1. A physical, handheld interface using an IMU sensor and BLE to capture expressive gestures.
2. A real-time 3D visualization in Unity that gives immediate visual feedback.
3. A procedural audio synthesis engine that generates sound dynamically, based on the physics of the user's movements, instead of just playing static audio files.

Putting these three parts together for the purpose of historical instrument simulation is the unique contribution of this thesis. It moves beyond a simple reconstruction. It creates a full, playable digital instrument that users can experience in a physical way. You can find the project's source code in this link¹.

1.5 THESIS OUTLINE

The rest of this thesis is split into chapters.

- **Chapter 2 - State of the Art:** Looks at related projects in digital museums and procedural audio. This shows what work other people have done on the topic.
- **Chapter 3 - Background:** This chapter explains the key technologies. The hardware section describes the ESP32 and MPU-6500 sensor. The software section talks about Unity and BLE.

¹<https://github.com/SanCruzo/MusicProject>

1.5. THESIS OUTLINE

- **Chapter 4 - Design and Implementation:** Shows how the project was built. This section gives details about the hardware and the code on the ESP32. The Unity software is also explained, including the BLE connection, the 3D model, and the sound engine.
- **Chapter 5 - Results and Evaluation:** This part will talk about the final project. It will look at how well the system works. It will also discuss if the user experience is good.
- **Chapter 6 - Conclusion and Future Work:** The last chapter summarizes the project. It talks about what was achieved. It also gives ideas for future work and how to make the system better.



Background

2.1 DIGITAL RECONSTRUCTION IN CULTURAL HERITAGE

Lately, the cultural heritage field has been using more and more digital technology. These new tools are a strong alternative to the old ways of preserving and showing historical artifacts. For a long time, if an ancient instrument was broken, the only way to understand it was for an expert to build a physical copy. But that process is slow, costs a lot, and it's tough to update if you discover something new [6].

Digital methods like 3D scanning and virtual modeling are a lot more flexible. As Sun et al. (2020) pointed out, when an artifact is in bad shape, it's hard to "test and evaluate different possibilities" with a physical copy. Virtual models solve this problem easily. This change to digital tools has given researchers new ways to study old objects and show them to a global audience.

2.2 DIGITAL PATHWAYS TO ANCIENT ACOUSTICS

Recreating a musical instrument involves more than just its physical shape. The main purpose of an instrument, after all, is to create sound. Therefore, a successful digital reconstruction must also think about its acoustic properties. This field of study, sometimes called "virtual archaeoacoustics," tries to digitally simulate the sounds of instruments that have been silent for centuries [7]. A key example of this approach is the work on a Roman brass instrument by Sun et

2.3. FROM STATIC MODELS TO INTERACTIVE EXPERIENCES

al. (2020). Their study provides a clear roadmap for this kind of work. They started by using a structured light system to create high-resolution 3D scans of the instrument's broken pieces. After this, they used a special algorithm to virtually repair the digital models and put them back together. The final and most important step was the sound simulation. Using a physically-based synthesis approach, they were able to create sounds that were directly connected to the geometry of the final, reconstructed model [6]. This research shows a solid method for taking a damaged artifact and bringing its potential sound back to life digitally. This methodology is not limited to just one type of instrument. Similar approaches have been used for other ancient instruments, like the successful reconstruction of an ancient Egyptian pan flute, which also combined 3D modeling with acoustic simulation to understand how it might have sounded [7]. Another good example is the work on the ancient Greek Lyre of Hermes. In that project, researchers used 3D laser scanning and audio analysis to build a modern reconstruction of the instrument [9]. These studies show that there is a reliable process for recreating the acoustics of historical instruments, representing the current state of the art for these kinds of non-interactive digital reconstructions.

2.3 FROM STATIC MODELS TO INTERACTIVE EXPERIENCES

These digital reconstructions are an amazing step forward. It is possible to see the instrument in 3D and hear a simulation of its sound. But there is a big piece of the puzzle missing: a user cannot actually play it. The feeling of holding the instrument and creating the sound is completely lost. Playing a musical instrument is a physical skill. It connects the body's movement directly to the sound that is produced. The existing research is great for answering the question "what did it sound like?", but it does not help with "how did it feel to play?".

This limitation is often called the "interaction gap." Fortunately, modern technology is making it possible to close this gap. Small and affordable sensors, like Inertial Measurement Units (IMUs), can detect movement. Wireless technologies like Bluetooth Low Energy (BLE) can then send that movement data to a computer. A real-time engine like Unity can use this data to power an interactive simulation. This technological combination is the key to connecting a person's

physical actions to a digital object in real time.

This idea is not just a fantasy. It fits into a larger trend where museums are trying to create more fun and engaging exhibits. For example, a study by Roberts et al. (2015) looked at how motion sensors could get visitors more involved with museum displays [8]. They found that letting people use their hands and bodies to explore digital things made the experience much better. This aligns with recent research on 'participation patterns' in museums, which shows how carefully designed interactive exhibits can guide a visitor's journey and create deeper engagement [10]. The same principle can be applied to musical instruments. Instead of just looking at a reconstruction, it becomes possible for people to actually perform with it. Moreover, recent work highlights how sound itself can structure social space within exhibitions, strengthening multimodal engagement [3].

2.4 ACOUSTIC CONTEXT: ENVIRONMENTS AND SOUNDSCAPES

Digital reconstructions often focus on the instrument itself, but the acoustic context shapes what is heard and how it is perceived. Two strands are relevant. First, room and environmental acoustics can be simulated through auralization: measuring or modeling room impulse responses and rendering them for listeners. This provides spatial cues (early reflections, reverberation) that affect clarity, timbre, and the sense of presence [11].

Second, the soundscape perspective treats listening as situated within a broader social and ecological context. Beyond physical metrics, it considers how people perceive, interpret, and share sonic environments. This lens is useful for museum exhibits where sound is part of a multimodal, social experience. Integrating a simple auralization stage with a soundscape-aware design can make interactive reconstructions feel more believable and informative [12].

In practice, a basic pipeline can cover both needs. A dry signal from the instrument model is sent through a room response that matches the target space. For a museum gallery, a short reverberation with clear early reflections helps speech-like clarity and keeps the sound anchored to the display. For a larger hall or a virtual courtyard, a longer decay creates distance and a stronger sense of place. Even a simple convolution with a measured or synthetic impulse response is enough to change how the same gesture is perceived by the visitor

2.4. ACOUSTIC CONTEXT: ENVIRONMENTS AND SOUNDSCAPES

[11].

The social side matters as much as the physics. Exhibitions are shared settings. Visitors stand together, talk, and react to what they hear. Sound can guide attention, mark interaction zones, and invite participation without adding visual clutter. A calm ambient bed can make the space feel continuous, while short, localised sonic cues can signal that an object is responsive. These choices affect comfort, dwell time, and how people move through the gallery. Thinking in soundscape terms keeps the design focused on the human experience rather than only on acoustic measurements [12]. Recent experimental work also shows that immersive setups can be used to evaluate visitor soundscape preferences in museum contexts [13].

For historical instruments, context also carries meaning. A ancient object placed in a bare, anechoic sound will feel detached. A modest, historically plausible room response helps the visitor connect the instrument to a lived space. The goal is not hyper-realism. It is to provide enough spatial and contextual cues so that the interaction feels natural, readable, and respectful of the setting. This balance is achievable with lightweight methods that work well alongside real-time procedural audio.

3

System Architecture

3.1 OVERVIEW

The system is designed as a real-time interactive loop. The main idea is to connect a person's physical movements directly to a digital simulation. The system has two main parts: the physical hardware device and the software application. The communication between them happens wirelessly using Bluetooth Low Energy.

The process starts with the user. The user holds and moves the physical device. Inside the device, an MPU-6500 IMU sensor measures this movement. It specifically measures the angular velocity on three axes.

The ESP32 microcontroller¹ is the brain of the hardware. It is connected to the MPU-6500² with the I2C protocol. The ESP32 continuously reads the latest sensor data. It then formats this data into a simple text string. This string is then sent out wirelessly using the ESP32's built-in BLE radio.

The software part runs on a computer. It is an application built in the Unity engine. The software is always listening for BLE data from the hardware. When new data arrives, a C# script parses the text string to get the numbers for the three rotation axes.

This data is then used in two ways at the same time. First, it is used to rotate

¹<http://espressif.com>

²<https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6500/>

3.2. HARDWARE COMPONENTS

the 3D model of the instrument on the screen. This gives the user immediate visual feedback. Second, the data is sent to the procedural audio engine. This engine uses the movement data as an energy input to generate the instrument's sound in real time.

The result is a closed-loop system. The user's action creates a sound and a visual change. This sensory feedback then guides the user's next movement. This creates an expressive and interactive experience. This chapter covers the main tech used for both parts, and explains why each component was chosen.

3.2 HARDWARE COMPONENTS

3.2.1 MICROCONTROLLER: ESP32-WROOM-32

The 'brain' of the hardware is an ESP32-WROOM-32 module (Figure 3.1). It's a small, single-chip computer made for specific jobs. The ESP32 is a powerful and popular chip with a dual-core 240 MHz processor. This is more than enough power to read the sensor data and handle the Bluetooth connection at the same time without any lag.

The technical specifications of the ESP32 make it a very good choice for real-time projects. The chip has a dual-core Xtensa LX6 microprocessor. This is a big advantage. One core can be dedicated to managing the complex wireless communication tasks. The other core can run the main application code, which reads the sensor data. This separation is important. It helps make sure that the timing is reliable. The sensor can be read at a steady rate, and the BLE data can be sent without delays or interruptions. The chip also has many built-in hardware interfaces. For this project, the I2C interface is used. It provides a simple and standard way to connect to the MPU-6500 sensor.

The main reason this chip was chosen is its built-in wireless. It has a radio that supports Bluetooth Low Energy (BLE), which is perfect for this project. BLE is good at sending small, steady streams of data, like the orientation data from the sensor, without using much power. The ESP32's job here is to be a bridge: it grabs the sensor data through the I2C protocol and broadcasts it over BLE so the Unity application can pick it up. Table 3.1 shows the basic setup.

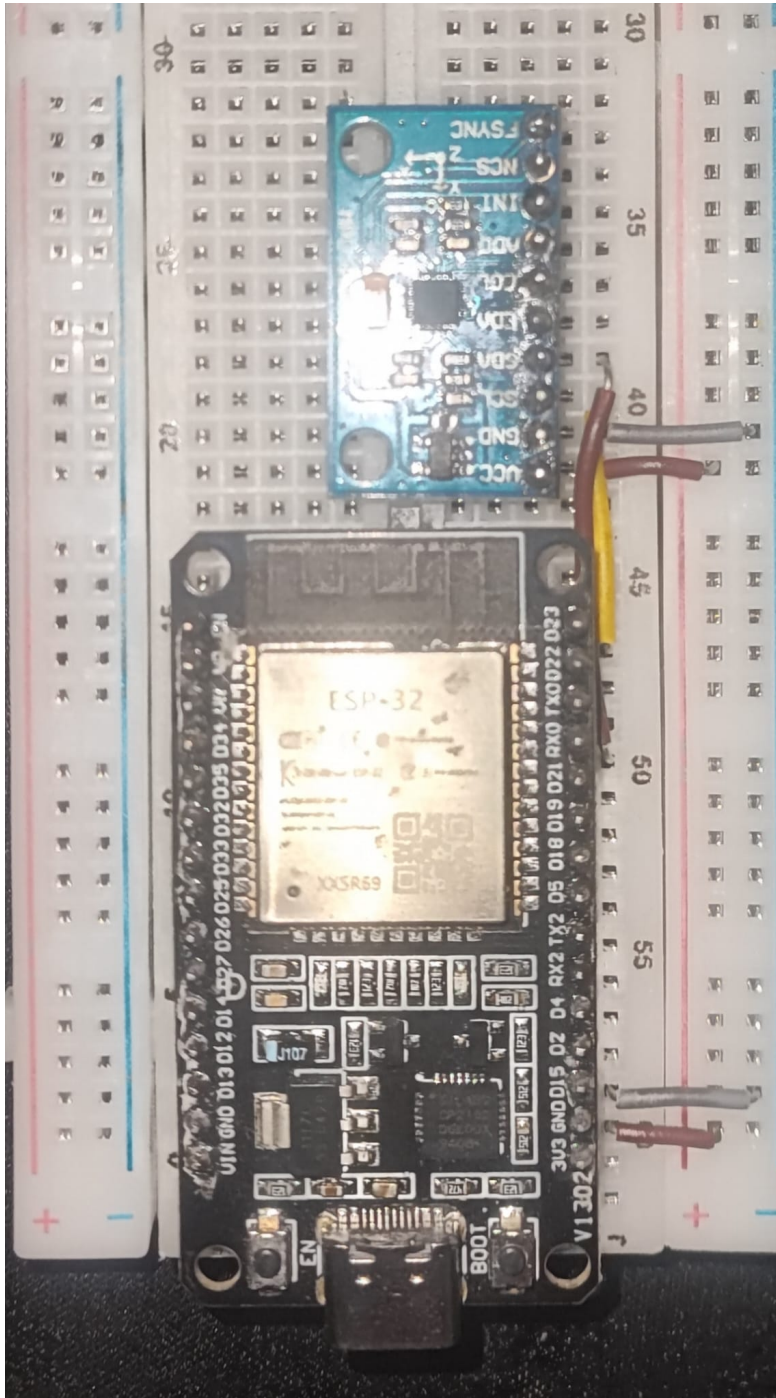


Figure 3.1: The project hardware setup on a breadboard, featuring the ESP32-WROOM-32 development board and the MPU-6500 IMU sensor.

3.2. HARDWARE COMPONENTS

Table 3.1: ESP32 Configuration Parameters

Parameter	Value
Module	ESP32-WROOM-32
Communication	Bluetooth Low Energy (BLE)
I2C Pins	SDA: GPIO21, SCL: GPIO22
Power Source	5V USB

3.2.2 SENSOR: MPU-6500 IMU

To get the player's movements into the computer, the project uses an MPU-6500 Inertial Measurement Unit (IMU). This small chip measures motion by combining a 3-axis accelerometer and a 3-axis gyroscope. The accelerometer tracks linear movement (like shaking), while the gyroscope tracks rotation (like tilting).

An IMU works by measuring two different types of motion. The accelerometer measures linear acceleration. This is the rate of change in velocity in a straight line. It is good for detecting quick, sharp movements, like a shake or a tap. The gyroscope measures angular velocity. This is the rate of rotation around an axis. It is good for detecting tilting, turning, and spinning motions. For this project, the gyroscope data is the most important. The main interaction with the instrument is rotational, so the '(x, y, z)' values from the gyroscope are the primary input for the system.

A key feature of the MPU-6500 is that it combines the 3-axis gyroscope and 3-axis accelerometer on the same piece of silicon. This is an advantage because it makes the sensor very small and means the two sensors are perfectly aligned. The sensitivity of the sensor is also programmable. The gyroscope's full-scale range can be set to ± 250 , ± 500 , ± 1000 , or ± 2000 degrees per second. The accelerometer has similar settings. For this project, the default range of ± 250 degrees per second for the gyroscope was sensitive enough to capture the fine movements needed to control the instrument.

The ESP32 reads the raw data from the MPU-6500 to understand how the instrument is being moved in real-time. This data is then sent wirelessly to Unity, which uses it to control the virtual instrument. The MPU-6500 is a practical choice because it is small, low-power, and connects easily with the ESP32 via the I2C protocol. Table 3.2 shows the specific configuration used.

In applications that need full orientation (quaternions) rather than just rota-

tion speed, lightweight fusion filters such as the gradient-descent method are commonly used; this approach is known to run well on embedded targets [14]. In this project, only gyroscope rates are required for the interaction, so fusion was not applied.

Table 3.2: MPU-6500 Sensor Configuration and Parameters

Parameter	Value
Sensor	MPU-6500
Data Outputs	Raw Accelerometer (x,y,z), Raw Gyroscope (x,y,z)
I2C Address	0x68 (default)
Update Interval	100 ms (10 Hz sampling)
Gyroscope Range	± 250 degrees/sec
Accelerometer Range	$\pm 2g$

3.2.3 WIRELESS COMMUNICATION: BLUETOOTH LOW ENERGY (BLE)

Bluetooth Low Energy (BLE) connects the hardware to the computer. It is a type of Bluetooth made for low-power uses. It is good for sending small amounts of data all the time without using much battery. Wi-Fi was not needed for this project. Normal Bluetooth uses more power.

The BLE standard organizes the connection using a defined structure. This structure includes two main parts: the Generic Access Profile (GAP) and the Generic Attribute Profile (GATT). GAP is responsible for controlling the connection. It defines roles for the devices. In this project, the ESP32 has the **Peripheral** role. This means it is the device that has the data and advertises itself. The computer running Unity has the **Central** role. This means it is the device that scans for peripherals and starts the connection.

GATT is responsible for how the data is structured. It uses a client-server model. The device that has the data is the GATT Server. In this project, that is the ESP32. The device that wants the data is the GATT Client. That is the computer. The data on the server is organized into **Services** and **Characteristics**. A service is a group of related data points. A characteristic is a single data point. Each service and characteristic has a unique number called a UUID.

So, in this project, the ESP32 acts as a GAP Peripheral and a GATT Server. The Unity application is a GAP Central and a GATT Client. The C# script in Unity scans for the ESP32. When it finds it, it connects. Then it looks for the specific service and characteristic using their UUIDs. Finally, the script subscribes to the characteristic. This tells the ESP32 to start sending data automatically. The data is sent as a simple text string with commas, like "1.23,4.56,7.89". This is easy for C# to read. BLE fits this project because it sends small, periodic packets with very low power and supports notify-based streams over GATT, which matches a 10 Hz IMU data rate and battery-friendly operation [15].

3.3 SOFTWARE PLATFORM: UNITY

The software part of this project was built in Unity. People know it as a game engine, but its tools for real-time 3D graphics and physics made it a good choice for this project.

So, why not just connect a speaker to the ESP32 and play a sound when it shakes? This approach would miss the main point of the project. The goal

isn't just to make a sound, but to create an interaction that is visual and feels physically real.

A hardware-only setup has two big problems. First, there are no visuals. A key part of this work is to digitally show the instrument and let the user see how their movements affect a 3D model. Without a graphics engine like Unity, the user would just be shaking a circuit board. Second, a complex audio simulation requires significant processing power that the ESP32 simply does not have. The characteristic sound of the ancient Roman instrument was generated by a multitude of small internal elements colliding with each other and the instrument's walls. Recreating this acoustic complexity in a believable way requires a sophisticated approach to sound synthesis.

In real-time interactive engines, both procedural synthesis and efficient sound propagation are established practices for games and VR [16]. In Unity specifically, dedicated libraries demonstrate viable workflows for sample-accurate, thread-safe real-time synthesis [17]. Integrated audiovisual toolchains also exist, enabling algorithmic audio directly within Unity [18]. Unity has also been used for instrument modeling that links interaction and real-time sound [19].

Initially, one might consider using Unity's built-in physics engine to simulate each of these internal particles as a separate physical object. In this scenario, a C# script would monitor the collision events generated by the physics engine, using data like impact velocity to trigger corresponding sounds. However, simulating hundreds or even thousands of individual rigid-body collisions in real-time is an extremely computationally expensive task. This method would lead to high CPU usage, an unstable frame rate, and, most critically, potential audio latency. Such latency would break the crucial sense of immediacy between the user's physical gesture and the resulting sound, defeating the project's core goal of a seamless interactive experience.

For this reason, a better solution was developed. It uses a procedural audio model. This model takes the movement data from the sensor. It uses this data as the "energy" for the sound. The C# script uses this energy to drive a DSP chain in real-time. This creates the sound of many small impacts using filtered noise. It sounds real, but it does not use a lot of CPU power. This method is common for creating believable sound in interactive virtual environments [20]. This approach keeps the performance smooth and gives a lot of control over the sound.

4

Software Development and Implementation

This chapter explains the technical details of how the interactive system was designed and put together. The whole project can be understood as two separate but connected areas. The first is the physical hardware component, which has the job of capturing the user's physical movements in real-time. The second is the software simulation, which is responsible for creating the complete audio-visual experience on the computer. The following sections will walk through the implementation details for both of these parts, and will also explain some of the key design choices that were made during the development process.

4.1 FIRMWARE IMPLEMENTATION

The ESP32 was programmed in C++ with the Arduino IDE. This is a common way to program these boards. The IDE is simple and there are many libraries available from the community. The firmware code has a few main jobs. It needs to include the right libraries for the hardware. It has to get the sensor and Bluetooth radio ready. It must also create a BLE server and then loop to send the sensor data to the computer.

4.1. FIRMWARE IMPLEMENTATION

4.1.1 LIBRARIES AND GLOBAL VARIABLES

Algorithm 1 Pseudocode for Firmware Libraries and Global Definitions

Library Usage: I2C, MPU-6500 Sensor, Bluetooth LE.

CONSTANTS:

SERVICE_UUID

CHARACTERISTIC_UUID

GLOBAL VARIABLES:

sensor_object

ble_characteristic

is_connected ← **FALSE**

EVENT HANDLERS:

function ONBLECONNECT

is_connected ← **TRUE**

end function

function ONBLEDISCONNECT

is_connected ← **FALSE**

end function

The first part of the code is for libraries. `Wire` is for I2C. `Adafruit_MPU6050` is for the sensor. The BLE libraries are for Bluetooth. The code also defines the unique UUIDs for the BLE service and characteristic. A global variable for the characteristic is made. This lets both the setup and loop functions use it.

4.1.2 SETUP FUNCTION

Algorithm 2 Pseudocode for the setup() function.

```

function SETUP
  INITIALIZE(Serial Monitor)

  if Sensor_initialization_fails then
    PRINT ERROR MESSAGE(to Serial Monitor)
    HALT(program execution)
  end if

  INITIALIZE BLE(with device name "ESP32_BLE_IMU")
  CREATE(BLE Server)
  ASSIGN(connection_handlers to Server)

  CREATE(BLE Service with SERVICE_UUID)
  CREATE(BLE Characteristic for the Service with CHARACTERIS-
  TIC_UUID)
  SET CHARACTERISTIC PROPERTIES(to NOTIFY)

  START(Service)
  START(BLE Advertising)
end function

```

The 'setup()' function runs just one time. This happens right when the ESP32 is powered on. Its main job is to get all the hardware and software parts ready for the main loop. First, it starts the serial monitor. This is a simple but useful tool for debugging. It lets you see status messages on the computer. Next, it tries to initialize the MPU-6500 sensor. If the sensor can't be found on the I2C bus, the code stops running. This is a fail-safe, because without a sensor, there is no data to send. The most important part is the BLE setup. The code gives the BLE device a name. It makes a server. Then it makes the service and characteristic with our UUIDs. The characteristic is set to allow notifications. This is key for real-time data. Finally, the code starts BLE advertising. This makes the ESP32 visible. The Unity app can then find and connect to it.

4.1.3 THE MAIN LOOP: READING, FILTERING, AND SENDING DATA

Algorithm 3 Pseudocode for the main loop() function.

```

function LOOP
  if is_connected_flag ← TRUE then
    raw_gyro_data ← READ(sensor_object)

    calibrated_data ← raw_gyro_data − calibration_of fsets
    filtered_data ← APPLY(dead_zone_filter TO calibrated_data)

    data_string ← FORMAT(filtered_data AS "x,y,z")

    SET(ble_characteristic_value TO data_string)
    SEND(notification_to_client)

    PAUSE(100 milliseconds)
  end if
end function

```

The ‘loop()’ function is the firmware’s core. After ‘setup()’ finishes, this function runs over and over. Its job is to read the sensor, apply calibration and filtering, and send the final data if a device is connected.

The loop continues indefinitely; it only stops on reset or power loss. A BLE disconnection does not stop the loop—data acquisition continues and notifications are skipped until reconnection. If sensor initialization fails, the firmware halts earlier in ‘setup()’.

The process begins by checking the ‘deviceConnected’ flag. If no device is connected, the code does nothing, saving processing time. If a device is connected, it calls ‘mpu.getEvent()’ to get the latest data from the MPU-6500. This fills a struct with acceleration, gyroscope, and temperature values.

Next, calibration and filtering are applied. To ensure the sensor reads zero when it’s not moving, pre-calculated offset values (*gx_offset*, etc.) are subtracted from the raw gyroscope readings. Then, a "dead zone" filter is applied. If a reading is very small (within a threshold of +/- 20), it is set to zero. This stops tiny, random sensor noise from causing unwanted movement in the simulation.

The three filtered gyroscope values are then formatted into a string using ‘sprintf’. This function is efficient and creates a comma-separated string like “-0.25,1.52,0.98”. This string is placed into the BLE characteristic with

'pCharacteristic->setValue()', and 'pCharacteristic->notify()' sends it to the computer.

Finally, 'delay(100)' pauses the loop for 100 milliseconds. This sets the data rate to about 10 Hz, which is fast enough for smooth motion without overwhelming the BLE connection.

4.2 SOFTWARE DESIGN

The software part of the project was built using the Unity game engine. Unity was chosen because it is good for making real-time 3D applications. It has a strong physics engine, it can create high-quality graphics, and it can be programmed with C#. This made it a good choice for this project, which needs to do real-time physics simulation, 3D rendering, and audio synthesis all at the same time.

The software has a component-based design. This is a standard way of working in Unity. The application is built from different 'GameObjects', and each 'GameObject' holds a collection of components, or scripts. Each script is responsible for a single, specific job. For example, one script handles BLE communication. Another script controls the 3D model's rotation. A third script simulates the physics of the instrument's beads.

This approach is a practical application of the Component design pattern. It keeps the code organized and easier to manage. Instead of having one large, complex program, the system is broken down into smaller, independent parts that communicate with each other. This makes it simpler to develop, test, and debug individual features without affecting the rest of the system. It also improves code reusability, as the same component can be used on different 'GameObjects'. Using small, independent parts is a standard way to build complex software and speed up development, a practice supported by technical literature [21], [22]. The main jobs of the software are:

- Get sensor data from the hardware over BLE.
- Use this data to move the 3D instrument on the screen.
- Simulate the physics of the instrument's internal parts.
- Play sounds based on the physics.

4.2. SOFTWARE DESIGN

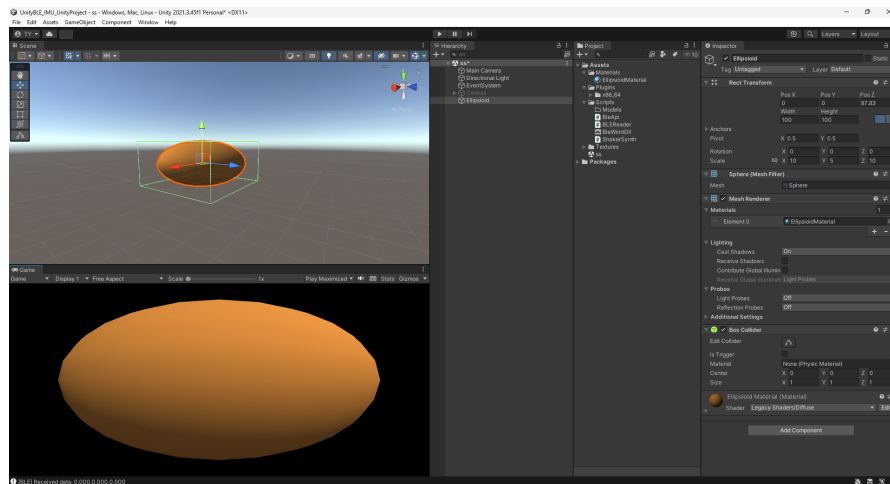


Figure 4.1: The Unity Editor interface for this project. On the left is the scene view. In the center is the object hierarchy. On the right, the Inspector window shows the public parameters for the BLE Reader and Shaker Synth scripts.

4.2.1 BLE COMMUNICATION IN UNITY

A C# script gets data from the ESP32. Unity on Windows does not have good native BLE support. A library is needed. The reason is a mismatch. Unity uses the .NET Framework. Windows BLE is part of the Windows Runtime (WinRT). They cannot communicate directly with each other.

This project uses 'BleWinrtDll'¹. This library is a bridge. It is written in C++. It can talk to the native WinRT APIs. It wraps the Windows functions. It puts them in a simple DLL file. This DLL is put in the Unity project. C# scripts can then call the functions. This gives access to BLE features.

The main script, 'BLEReader.cs', manages the entire connection process. When the application starts, the script initializes the library and begins scanning for devices. It looks for a device with the name "ESP32_BLE_IMU", as defined in the firmware. As soon as it finds the correct device, it stops scanning and establishes a connection.

After connecting, the script must find the correct characteristic to receive data. This is a two-step search that uses the UUIDs from the ESP32 code. The script first finds the correct service, and then searches within that service to locate the characteristic.

Once the characteristic is found, the script subscribes to it. This subscription

¹<https://github.com/adabru/BleWinrtDll>

acts as a request, telling the ESP32 to automatically send an update whenever the data changes. This method creates the real-time data stream, which is more efficient than constantly asking for new data.

It is important to handle disconnections. The first version of the code did not do this well. If the ESP32 was unplugged, the app would get stuck. The user had to restart it. The new 'BleLoop' method is better. It has one main loop. This loop first tries to connect. If it connects, it listens for data. It also uses a timer. If it gets no data for two seconds, it assumes the device is disconnected. It then goes back to the beginning of the loop and starts scanning again. This makes the system better. The user can unplug the device and plug it back in. The app will reconnect by itself.

DATA HANDLING AND PARSING

The ESP32 sends a notification. The C# script gets a byte array. This array must be converted. It is turned into a string. This is the text from the firmware. For example: "-0.25,1.52,0.98".

Then, the script splits this string. The 'String.Split(',')' method is used. It breaks the string at the comma. This makes an array of three new strings. Each string is one gyro value.

Finally, the strings are converted to 'float' numbers. 'float.Parse()' does this. The three numbers go into a 'Vector3'. A 'Vector3' is a Unity data type for 3D vectors. This final 'Vector3' stores the rotation data. It is public. Other scripts can access it.

IMPLEMENTATION OF BLEADER.CS

The following C# code is the 'BLEReader' component. This script is attached to a 'GameObject'. It handles all BLE communication. The code is shown in smaller parts. Each part is explained.

Algorithm 4 Pseudocode for BLEReader class variables.

PUBLIC CONFIGURATION:

▸ These values must match the firmware settings.

targetDeviceName ← "ESP32_BLE_IMU"

targetServiceUUID ← "unique_service_address"

targetCharacteristicUUID ← "unique_characteristic_address"

INTERNAL STATE:

connectedDeviceId ← ""

cube_object

gyro_data

shaker_synth_object

THREAD MANAGEMENT:

ble_thread

is_running ← TRUE

Class Variables The script starts with several variables.

- 'targetDeviceName', 'targetServiceUUID', 'targetCharacteristicUUID': These three strings are important. They must match the firmware. They help the script find the right device and characteristic. They are 'public'. This means they can be changed in the Unity editor.
- 'connectedDeviceId': A private string. It stores the device ID after connection.
- 'cube': A public 'GameObject'. This is the 3D model in the scene. Sensor data will rotate it. It can be assigned in the Unity editor.
- 'gyroData': A private 'Vector3'. The latest sensor data is stored here after parsing.
- 'shaker': A private reference to the 'ShakerSynth' script. This script makes the audio. '[SerializeField]' lets it be assigned in the editor.
- 'bleThread' and 'running': These manage the background thread. 'bleThread' is the thread itself. 'running' is a flag to stop the thread.

Algorithm 5 Pseudocode for Start and Update methods.

```

function START
  if shaker_synth_object is NULL then
    shaker_synth_object ← GET(audio_component)
  end if

  ble_thread ← CREATE_THREAD(withBleLoopfunction)
  START(ble_thread)
end function

function UPDATE
  rotation_change ← gyro_data × frame_time_delta
  ROTATE(cube_object by rotation_change)

  if shaker_synth_object is NOT NULL then
    CALL(shaker_synth_object.Drive with gyro_data)
  end if
end function

```

Start() Method This is a Unity method. It is called once. Its main job is to start the BLE process. It creates a new ‘Thread’. This thread runs the ‘BleLoop’ method. This is an important design choice. All BLE work is on a separate thread. This stops the main Unity thread from freezing. If the work was on the main thread, the app would stutter.

Update() Method This is another Unity method. It is called every frame. It has two jobs. It takes the ‘gyroData’. It uses it to rotate the ‘cube’ ‘GameObject’. Multiplying by ‘Time.deltaTime’ makes the rotation smooth. It is not dependent on the frame rate. It also passes ‘gyroData’ to the ‘shaker’ component. It calls the ‘Drive()’ method. This is how the audio engine gets data. It uses the data to make sound.

Algorithm 6 Pseudocode for the core BleLoop() logic.

```

function BLELOOP
  while application is running do
    if NOT connected then
      SCAN(for BLE devices)
      CONNECT(to the target device)
      SUBSCRIBE(to data notifications)
    else ▷ is connected
      POLL(for gyroscope data packet)
      if data is received then
        RESET(connection_timeout_timer)
        PARSE(data into gyro_data vector)
      else
        if connection_timeout_timer > 2 seconds then
          RESET(connection state to disconnected)
        end if
      end if
    end if
    PAUSE
  end while
end function

```

BleLoop() Method - Connection Phase The 'BleLoop()' method runs on a different thread. This is important so the main app does not freeze. The whole code is a big 'while' loop. First, it checks if the device is connected. If the 'connectedDeviceId' string is empty, it means we are not connected, so it tries to connect.

To start, it calls 'BleApi.StartDeviceScan()'. This begins looking for BLE devices. Then, a smaller 'while' loop starts. This loop is for finding our specific device. It calls 'BleApi.PollDevice()' over and over. It checks the name of each device it finds. It is looking for "ESP32_BLE_IMU".

The code finds the correct device. It saves the ID. Then it calls 'BleApi.StopDeviceScan()'. The scan is now finished. The next job is to find the correct service and characteristic. The code searches for them with their UUIDs. When it finds the characteristic, it calls 'BleApi.SubscribeCharacteristic()'. This is a very important step. It tells the ESP32 to start sending data. The code also calls 'Thread.Sleep(100)'. This makes the loop pause for 100 milliseconds. This stops the CPU from working at 100%.

BleLoop() Method - Data and Timeout Phase If the device is already connected, the loop does a different job. It reads the incoming data and also watches the connection to see if it is still alive. For this, it uses a 'Stopwatch' object, which is just a timer. The timer is started, and the code enters another sub-loop. Inside, it calls 'BleApi.PollData()' to check for a new message from the ESP32.

When a message with data arrives, it means the connection is working. The 'Stopwatch' timer is reset back to zero. The code then processes the data. The data comes as a byte array. It is converted to a string, which looks something like "-10.5,25.1,0.8". The code splits this string by the commas. It uses 'float.TryParse()' to change the text into numbers. This function is safe. It will not crash the app if the data is broken. The final numbers are put into the 'gyroData' vector.

If 'PollData()' does not find any new data, the 'Stopwatch' timer just keeps counting. If it counts up to two seconds (2000 milliseconds), the code decides the connection is lost. To handle this, it clears the 'connectedDeviceId' and sets 'gyroData' back to zero. This makes the main 'while' loop go back to the beginning and start the connection phase all over again.

OnApplicationQuit() Method The 'OnApplicationQuit()' method is the last important part. Unity calls this method when the application is closing. The code here makes sure the app shuts down correctly. First, it sets the 'running' variable to 'false'. This tells the 'BleLoop' to stop running. Next, it calls 'BleApi.Quit()'. This function closes the BLE connection properly. The last step is 'bleThread.Join()'. This makes the application wait for the background thread to finish completely. This is important so the app does not close with an error.

4.2.2 3D MODEL VISUALIZATION AND CONTROL

The next part of the software is the 3D visualization. This is the graphical part of the instrument. The user sees it on the screen. The main goal is simple. The 3D model's rotation must match the user's hand movements.

This is done in the 'Update()' method of the 'BLEReaders.cs' script. The 'gyroData' 'Vector3' has the latest rotation speed data. This data rotates the 'cube' 'GameObject'. In Unity, every object in a scene is a 'GameObject'. Every 'GameObject' has a 'Transform' component. The 'Transform' stores the object's

4.2. SOFTWARE DESIGN

position, rotation, and scale. The code changes the 'Transform's rotation every frame.

The line `'cube.transform.Rotate(deltaRotation, Space.Self)'` performs the rotation. 'Space.Self' means the rotation is relative to the object's own axes. This feels more natural.

The rotation value is multiplied by `'Time.deltaTime'`. This is important. 'Time.deltaTime' is the time since the last frame. Using it makes the rotation smooth. It also makes the rotation speed independent of the computer's performance. The model rotates at the same speed on a fast computer and on a slow one.

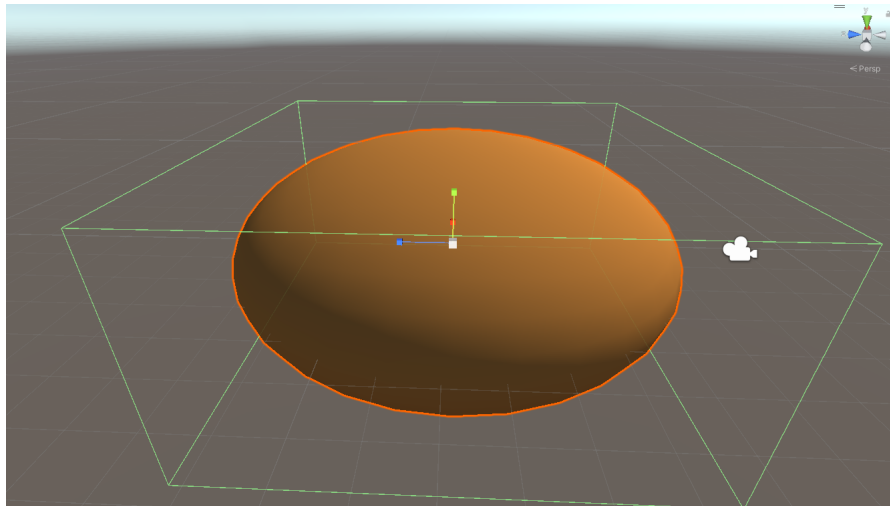


Figure 4.2: The final render of the 3D instrument model in the Unity scene. The Standard Shader provides realistic lighting and a smooth surface.

4.2.3 PROCEDURAL AUDIO SYNTHESIS

The final part of the software is the sound generation. For a percussion instrument like this, a simple solution would be to record a '.wav' file of a shaker sound. Then, the script could play this sound file when the user's movement is fast enough. This approach is easy. But it is not very realistic or expressive. The sound would be the same every time. It would not react to small changes in motion.

This project uses a better method. It is called procedural audio synthesis. The sound is not pre-recorded. It is generated in real-time by a mathematical model. This model simulates the physics of the small beads inside the instrument hitting

the walls. The 'gyroData' is used as the input energy for this simulation. This makes the sound very dynamic. It responds to every detail of the user's gestures. This approach is consistent with real-time procedural audio practices in Unity [17]. Industry practice in game audio also aligns with this approach through engine workflows and middleware [23].

The C# script for this is 'ShakerSynth.cs'. It is a component that can be attached to any 'GameObject'. It works by using Unity's 'OnAudioFilterRead()' method. This is a special function. It gives a script direct access to the audio output buffer. This means the script can write its own audio samples directly to the sound card. The script generates filtered noise. The properties of this noise (like volume and frequency) are controlled by the 'energy' calculated from the gyroscope data. This creates the sound of many small beads colliding.

SHAKERSYNTH.CS

The 'ShakerSynth.cs' script is the core of the procedural audio engine. It does not use any pre-recorded samples. Instead, it generates the shaker sound in real-time based on the gyroscope data. It uses a resonant band-pass filter to shape white noise, which is a common technique in audio synthesis. Classic modulation-based methods such as FM are foundational references for real-time synthesis design [24]. Likewise, granular approaches are well established in interactive sound design and can inform texture layers when needed [25].

Algorithm 7 Pseudocode for ShakerSynth public parameters and initialization.

```
1: PUBLIC SOUND PARAMETERS:  
2: level  
3: centerHz  
4: resonance  
5: energyDecay  
6: startThreshold  
7: inputGain  
  
8: function AWAKE  
9:   GET(the attached AudioSource component.)  
  
10:  CONFIGURE(AudioSource to play a continuous silent clip.)  
11:  START(playing.)  
  
12:  INITIALIZE(the random seed for noise generation.)  
13:  CALCULATE(initial filter coefficients based on public parameters.)  
14: end function
```

Public Parameters The script has several public variables. These are shown in the Unity Inspector. This makes it easy to change the sound without changing the code.

- 'level': The main volume.
- 'centerHz' and 'resonance': These control a filter. The filter is the main part of the sound. It makes noise sound like beads. 'centerHz' is the filter's frequency. 'resonance' is its sharpness.
- 'energyDecay': Controls how fast the sound fades. A high value is a long fade.
- 'startThreshold' and 'inputGain': These control how the synth uses movement. 'inputGain' scales the gyro data. 'startThreshold' is a gate. Movement below it makes no sound.

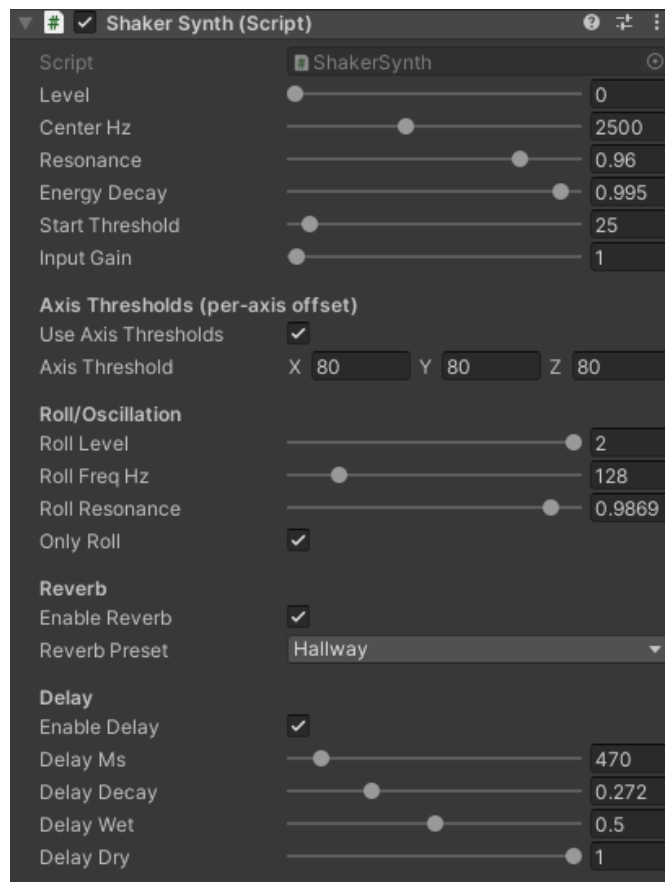


Figure 4.3: The public parameters of the ShakerSynth.cs script as they appear in the Unity Inspector. These values can be changed in real-time to adjust the sound.

Awake() Method This method runs once. It sets up the 'AudioSource'. 'OnAudioFilterRead()' needs an active 'AudioSource'. The code makes a silent audio clip. It plays it in a loop. This is a trick. It makes Unity call 'OnAudioFilterRead()' all the time. It also starts a random number generator. It calculates filter values.

Algorithm 8 Pseudocode for the Drive() method.

```

function DRIVE(gyro_input)
    movement_magnitude ← CALCULATE(magnitude from scaled_input,)
                                applying axis thresholds if enabled

    filtered_movement ← MAX(0, movement_magnitude -
    startThreshold)

    jerk ← (filtered_movement - previous_movement) / frame_time_delta
    previous_movement ← filtered_movement

    if filtered_movement > 0 then
        energy ← energy + (filtered_movement × constant_A) +
                    (ABS(jerk) × constant_B)
    end if

    energy ← CLAMP(energy between 0 and 1)
end function

```

Drive() Method This is the main input method. 'BLEReader' calls it every frame. It gets the 'gyroData'. The method turns this rotation data into 'energy'. This 'energy' value controls the sound.

It first scales the gyro data with 'inputGain'. Then it checks the 'useAxisThresholds' option. If this is on, the code uses a special filter. It checks each axis (x, y, z) alone, and removes small movements from each one. This gives more control. If the option is off, the code does a simple thing. It just gets the total speed of the rotation.

Next, it applies the global 'startThreshold'. This is a final check on the total movement. It makes sure no sound is made if the overall movement is very small.

The code also calculates "jerk". This is how fast the rotation speed changes. The final 'energy' is a mix of two things. It uses the normal rotation speed. It also adds a small amount of this jerk value. So, the sound gets louder with fast movement. It also gets a small boost with very sudden, sharp movements. This makes the sound more lively. The 'energy' is always kept between 0 and 1 with 'Mathf.Clamp01'.

Algorithm 9 Pseudocode for the `OnAudioFilterRead()` audio generation.

```

function ONAUDIOFILTERREAD(output_buffer, num_channels)
  current_energy  $\leftarrow$  energy

  for each sample_frame in output_buffer do
    current_energy  $\leftarrow$  current_energy  $\times$  energyDecay

    noise_sample  $\leftarrow$  GENERATE(randomnumber between - 1 and 1)
    scaled_noise  $\leftarrow$  noise_sample  $\times$  current_energy  $\times$  level

    filtered_sample  $\leftarrow$  APPLY(band-pass filter TO scaled_noise)

    for each channel from 1 to num_channels do
      output_buffer[sample_frame + channel] += filtered_sample
    end for
  end for

  energy  $\leftarrow$  CLAMP(current_energy between 0 and 1)
end function

```

OnAudioFilterRead() Method The sound is made in the ‘OnAudioFilterRead’ method. This is a special Unity function. It runs on the audio thread, not the main game thread. This is important for audio. It stops the sound from stuttering. The method gives the script direct access to an audio buffer. This is an array of numbers that is sent to the sound card.

The sound generation has a few steps. The process runs for every sample in the audio buffer.

First, the sound needs a source. The script makes a random number. A stream of random numbers is called **white noise**. It sounds like a hiss. This noise is the raw material for the sound. The energy from the user’s movement controls the volume of this noise. No movement means no energy, so there is no sound.

Second, the white noise must be shaped. It does not sound like a shaker. The code uses a **digital filter** for this. It is a resonant band-pass filter. The filter acts like an EQ. It cuts the very low and very high frequencies. Only a narrow band of sound can pass through. This is what creates the metallic, bead-like sound. The `centerHz` and `resonance` variables control this filter’s shape.

The final sample is a single float number. This number is written into the data array. The code loops through these steps until the buffer is full. Unity

4.2. SOFTWARE DESIGN

then plays this buffer. This process repeats hundreds of times per second. This makes a continuous sound that changes instantly with the user's movement.

Algorithm 10 Pseudocode for the `RecalcFilter()` method.

```
function RECALCFILTER
  PI ← 3.14159...

  angular_freq ← 2 × PI × centerHz / sample_rate
   $\alpha$  ← SINE(angular_freq) / (2 × resonance)

  b0_raw ←  $\alpha$ 
  a0_raw ← 1 +  $\alpha$ 
  a1_raw ← -2 × COSINE(angular_freq)
  a2_raw ← 1 -  $\alpha$ 

  a0 ← b0_raw / a0_raw
  b1 ← a1_raw / a0_raw
  b2 ← a2_raw / a0_raw
end function
```

Filter Coefficient Calculation The filter needs a few special numbers to work. These numbers are called coefficients. They are prepared by the '`RecalcFilter()`' method. This method runs when the script starts. It also runs if the user changes a filter setting in the Unity editor.

The job of this method is to do math. It takes the easy-to-use variables, like '`centerHz`'. It turns them into the hard mathematical coefficients, like '`a0`', '`b1`', and '`b2`'. The filter needs these numbers. This is a common idea in DSP. The hard math is kept separate from the audio loop. This makes the program fast. The math is only done once when a setting changes. The '`OnAudioFilterRead()`' method can then use these numbers to make sound very quickly.

ADVANCED AUDIO FEATURES

The shaker sound is not just filtered noise. The script adds more layers. These layers make the sound better. It adds a rolling sound for the beads. It also adds echo and reverb effects.

Algorithm 11 Pseudocode for the rolling sound generation.

```

impacts_per_second ← MAP(energy TO a range from 2 to 45)

impact_probability ← impacts_per_second/sample_rate

random_value ← GENERATE(random number between 0 and 1)
if random_value < impact_probability then
    impulse ← (random_value × 2 - 1) × current_energy
else
    impulse ← 0
end if

roll_sound ← APPLY(roll_filter TO impulse)

final_sample ← (main_friction_sound) + (roll_sound × rollLevel)

```

Simulating Rolling Beads A simple shaker sound is not enough. When a real instrument is moved slowly, the beads inside do not just hiss. They also roll and make soft, individual tapping sounds. The script simulates this with a second sound generator.

This sound uses a second filter. It is called a ****resonator****. A resonator makes a sound at a specific musical pitch. It is different from the first filter, which makes a noisy sound. This resonator creates a low, rumbling tone. This is the rolling sound.

The rolling sound is not always on. It is triggered by random impulses. The code generates a random number for every audio sample. If the number is very small, it creates a small 'click' or impulse. This impulse is then sent into the resonator. The resonator rings for a short time, making the rolling sound. The chance of an impulse happening depends on the 'energy' variable. More energy means more impulses per second. This feels very natural. Fast movements create a lot of noise. Slow, gentle movements create a few soft, rolling sounds.

Algorithm 12 Pseudocode for managing audio effects.

```

function SETUPREVERB(enabled, preset)
    filter ← GETORADDCOMPONENT("AudioReverbFilter")
    filter.enabled ← enabled

    if enabled then
        filter.preset ← preset
    end if
end function

function SETUPDELAY(enabled, delay, decay)
    filter ← GETORADDCOMPONENT("AudioEchoFilter")
    filter.enabled ← enabled

    if enabled then
        filter.delay ← delay
        filter.decay_ratio ← decay
    end if
end function

```

Reverb and Delay Effects The sound synthesis creates a very "dry" signal. It has no feeling of space. To fix this, the script can add two standard audio effects: reverb and delay. These are common effects. Reverb simulates the sound of a room. Delay creates echoes.

The script does not make these effects itself. It uses built-in Unity components. They are 'AudioReverbFilter' and 'AudioEchoFilter'. The script's job is to add these components to the 'GameObject'. It also changes their settings. There are 'public' variables in the script. They let the user turn the effects on or off. They also let the user change the settings, like 'delayMs' or 'reverbPreset'. This is all done in the 'SetupReverb()' and 'SetupDelay()' methods. These methods are called when the game starts. They are also called in the editor when a value is changed. This allows for real-time control of the effects.

5

Results and Evaluation

This chapter evaluates the performance of the final system. As this project is a proof-of-concept, a formal user study was not within the scope of the work. This evaluation, therefore, focuses on the technical performance and functional correctness of the prototype. The key evaluation criteria are: the reliability of the hardware-software connection, the real-time performance of the audio engine, and the responsiveness of the overall interactive experience.

5.1 SYSTEM FUNCTIONALITY TEST

The first test checked if the whole system works together. The hardware controller was connected to the Unity application to see if they could communicate. The test was a success.

Starting the Unity application made it scan for BLE devices. It found "ESP32_BLE_IMU" and connected automatically. After connection, the script located the correct service and characteristic. The firmware was doing its job, reading the sensor and sending the data. This data was seen in the Unity console, and the C# script parsed it into a 'Vector3' correctly. The test showed that the entire data path, from the physical sensor to the software, was working.

The visual part of the test was also good. The 3D model's rotation on the screen matched the controller's rotation exactly. The movement was immediate and smooth, with no delay.

An important test was for the automatic reconnection feature. The ESP32

5.2. AUDIO ENGINE EVALUATION

controller was unplugged from power while the system was running. In Unity, the 3D model and the sound stopped after a two-second timeout. The application then started to scan for the device again. The ESP32 was plugged back in. The application found it and reconnected automatically. The system returned to normal operation. No manual restart was needed. This confirms the system is robust.

A free-running timing test of the firmware loop (delay removed) reported an average work time of about 1.21 ms and an average loop period of about 1.23 ms (810 Hz). In exhibit mode the loop is period-limited to 20 Hz for BLE and power efficiency.

5.2 AUDIO ENGINE EVALUATION

The sound generation is the most important part of the project. Several tests were done on the 'ShakerSynth.cs' audio engine to check its quality.

The first test was for performance. The script was able to generate the audio in real time with no problems. The sound was clean and continuous, with no stuttering or clicking. This is because the audio code is efficient and runs on a separate thread from the main game logic, which is a good design.

The second test was for expressiveness. The controller was moved in many different ways to see how the sound would change. Fast rotations made the 'energy' variable in the script go up, which made the sound loud and bright. Slow rotations kept the 'energy' low, which made the sound quiet. The "jerk" calculation was also tested. A sudden, sharp movement created a clear attack sound. This shows that the audio engine responds to the character of the movement, not just the speed.

The third test was for flexibility. The public parameters in the 'ShakerSynth.cs' script were changed in the Unity Inspector while the program was running. Changing the 'centerHz' and 'resonance' values changed the timbre of the sound immediately. It was possible to make the sound deep and wooden, or sharp and metallic. This confirms that the engine is a flexible tool. It can be used to design many different shaker sounds.

5.3 SYSTEM CAPABILITIES AND POTENTIAL

The prototype's most important quality is its responsiveness. There is no noticeable delay between moving the physical controller and seeing the 3D model react and hearing the sound change. This instant feedback is critical. It creates a believable connection between the user's actions and the virtual instrument, giving the feeling of direct, physical control.

The system's main achievement is turning a static 3D model into something that can be played. Instead of just looking at a silent object or hearing a recorded sound, a user can interact with it. The sound is generated directly from the user's movements. This provides a more direct and hands-on understanding of the instrument.

There are practical uses for this technology. A museum could build an exhibit based on this system. This would let people interact with an instrument that is normally too old or fragile to touch. The project also works as a good starting point for more advanced work. For example, the sound synthesis could be made more complex, or other sensor data like the accelerometer could be used to create different effects. Prior work also shows that carefully designed sound can measurably shape visitor engagement and perceived quality in museum settings [26]. Case studies also document how curated sound is integrated in real exhibitions to convey historical context [27].



Conclusions

6.1 SUMMARY OF WORK

This thesis presented a project to solve a problem in the field of digital heritage. Past research often created digital copies of ancient instruments that were static. You could look at them, but you could not play them. The goal of this project was different. The aim was to create a fully interactive and playable digital instrument, based on an ancient percussion instrument. The main contribution was the design and implementation of a complete system that closes this gap between a static museum object and a live, expressive tool.

The system is made of two main parts. The first is the physical hardware. A simple, handheld controller was built using common, off-the-shelf components. It has an ESP32-WROOM-32 microcontroller and an MPU-6500 IMU sensor. These parts were chosen because they are low-cost and have good support in the Arduino community. The firmware, written in C++, reads the user's hand movements in real time. It then sends the gyroscope data to a computer using Bluetooth Low Energy (BLE).

The software part uses the Unity game engine. It is programmed in C#. The code is split into components. Each component has a job. The 'BLEReader.cs' script handles the BLE communication. It finds the device, connects, and reads the data. It can also reconnect automatically if the signal is lost. The script sends the gyro data to two places. It sends it to the 3D model to make it rotate. It also sends it to the 'ShakerSynth.cs' script to make sound.

6.2. MAIN CONTRIBUTIONS

The audio is a key part of the project. The system does not play recorded sound files. It generates all sounds in real time with a physics simulation. The 'ShakerSynth.cs' script uses filtered noise for the main "shaken" sound. It also uses resonators to make the softer sound of beads rolling. The user's movements directly control this sound. Both the speed and the suddenness of the gestures change the audio. This makes the instrument feel responsive. The project was a successful test. It showed that a physical gesture can be connected to a virtual object's sight and sound. It turns a static digital artifact into a playable, hands-on experience.

6.2 MAIN CONTRIBUTIONS

The main contribution of this thesis is not just one new part, but the way all the parts are put together. The project builds a complete system from end to end. Other projects have focused on just one piece, like only 3D modeling or only audio synthesis. This work is different. It connects historical reconstruction, modern sensor hardware, wireless communication, and real-time graphics and sound into one single, working experience.

The project acts as a blueprint that others can follow. It shows a clear path for how to take a static digital model and turn it into a physical, playable object. The design is also very practical because it is low-cost. It uses an ESP32 board and the free version of the Unity engine. This is important. It means that museums or university researchers can build similar interactive exhibits without needing a large budget or special, expensive equipment.

The Unity software is another key contribution. The Bluetooth communication system was built to be strong and reliable. Wireless connections in public places can often fail. The 'BLEReaders.cs' script was written to handle this problem. It runs on a separate thread so it does not freeze the application. It also has a timeout and an automatic reconnection loop. If the connection is lost, the system will keep trying to connect again by itself. No person needs to restart it. This makes the exhibit much more stable and useful for the public.

Finally, the use of procedural audio is a big step forward. Most interactive systems just play a pre-recorded sound when something happens. This feels repetitive. This project generates its sound in real time from a mathematical model. The sound is a simulation of the physics inside the instrument. The

user's gestures, including both speed and sharpness, directly control the sound. This creates a much stronger feeling of connection between action and sound. It makes the digital instrument feel more like a real, expressive object and less like a simple computer program.

6.3 LIMITATIONS

This project is a proof-of-concept. It shows that the main idea works. But there are many ways it could be better.

6.3.1 HARDWARE PROTOTYPE

A major limitation is the physical controller. The current hardware is built on a breadboard with jumper wires. This is fine for a lab environment where things can be fixed easily. But it is not a finished product. The wires can come loose and the components are exposed. It is too fragile to be given to the public in a museum. It serves its purpose as a functional prototype for development, but it is not a robust, final design.

6.3.2 THE SOUND IS AN APPROXIMATION

The sound generation method has its limits. The audio is not a recording from a real instrument. Instead, it is made in real time by a computer program. This program is a simple model that imitates the physics of the shaker. This method is what makes the instrument feel interactive. But the model is just an approximation. It does not include every small detail of a real object. For example, the sound is convincing, but it lacks the very complex textures you would hear from a real instrument. To simulate all the tiny interactions between the objects inside and the container walls perfectly would need a supercomputer. This is not possible for a system that has to work in real time.

6.3.3 INTERACTION BASED ONLY ON ROTATION

The way the user plays the instrument is also not complete. The project's scope was focused on rotational gestures, so the system only uses gyroscope data from the IMU sensor. This means it understands rotation, but nothing else.

6.4. FUTURE WORKS

Real percussion instruments like this are played with many different gestures, including sharp hits, gentle shakes, and linear movements. The decision was made not to include accelerometer data in this version of the project to keep the scope manageable. This is a significant limitation because it means the range of expression for the user is small. A future version that includes accelerometer data would allow for a much more nuanced and realistic performance.

6.3.4 THE VISUAL MODEL IS A PLACEHOLDER

The 3D model in this project is functional, but it is not a detailed recreation. Its main job is to give the user visual feedback for their rotations. It is a simple shape, not a historically accurate reproduction. A real digital heritage exhibit would need a much more faithful model. This would require using techniques like 3D laser scanning or photogrammetry on the actual artifact. That process would capture the exact shape, size, and even the surface texture. The current visual model was not intended to provide this level of detail or authenticity.

6.3.5 LACK OF USER EVALUATION

Finally, the project was never tested by its target audience. All the testing was done by the developer. This means we only know that the system works on a technical level. We do not have any information about the user experience. A formal study would be necessary to see how people in a museum setting would use the device. Would they understand it? Would they find it interesting? Is it an effective educational tool? Without this kind of evaluation, we cannot be sure if the project truly succeeds in its goal of making digital heritage more engaging for the public.

6.4 FUTURE WORKS

This project showed that the idea works. Now, to make it ready for a museum, there are several jobs to do. These next steps all come from the limitations we found in the project.

6.4.1 BUILD A CONTROLLER

The breadboard is not for the public. It is too easy to break. A custom PCB must be designed. This is job number one. The PCB will be strong. Connections will not get loose. A battery and charging circuit will be on the PCB. No more USB cable. The finished PCB will go inside a 3D printed case. The case will be a copy of the real historical instrument. It must be strong enough that people can drop it and it will not break. We can use a special filament with wood powder to make it feel better to hold.

6.4.2 MAKE THE SOUND BETTER

Although the project is currently good, the sounds are still being generated by the computer. To make it sound like a real instrument, there are two jobs to do. The first job is to improve the physics model. The current calculation provides a good foundation. A better model needs more complex math. It must calculate the real physics. This includes the container's shape and size. It also includes the properties of the objects inside. This is a big job and it will take a lot of research. It also needs a fast computer. The second job is a different idea. We can use machine learning. With this method, we do not need to write the physics rules. We just need to get a lot of audio recordings of a real instrument. We train a program with these recordings. The program learns by itself how to connect the movement of the controller to the correct sound.

6.4.3 ADD MORE SHAKING GESTURES

The instrument today only uses gyroscope data. It only knows about rotation. This is a big limit. The next software version must use the accelerometer data too. First, the ESP32 code must be changed. It needs to read the accelerometer and send the x, y, and z values over BLE. Second, the Unity code must be changed. The 'BLEReader.cs' script needs to parse these new values. Then it can send them to the 'ShakerSynth.cs' script. With this new data, we can program new sounds. A hard shake on the y-axis can make a loud sound. A soft shake on the x-axis can make a quiet sound. This gives the user more control.

6.4. FUTURE WORKS

6.4.4 CREATE A 3D MODEL OF THE INSTRUMENT

The 3D model is a simple placeholder. For a museum, it is not good enough. We must create a real model. We should use a 3D scanner. A 3D scanner measures the object and creates a perfect digital copy. It captures the exact size, shape, and all the details of the surface. This new model will be imported into Unity. It will replace the placeholder shape. When a user plays the instrument, they will see an authentic model on the screen. This makes the experience much better.

6.4.5 EXPERIMENT WITH MUSEUM VISITORS

The project is not validated until real people test it. An engineer can say it works, but that is not enough. We must take the final instrument to a museum. We set up an interactive installation and let people use it. Then we gather the data from them by answering them some questions. Do they understand how to use it without help? How long do they play with it? Do they look like they are having fun? We can also ask them directly. Was it easy? Was it interesting? Did it make you think about the history? This is the most direct way to obtain a qualitative measure of the installation's behavior and the user's preferences about it.

References

- [1] United Nations Educational, Scientific and Cultural Organization (UNESCO), *Convention for the safeguarding of the intangible cultural heritage*, 2003. [Online]. Available: <https://ich.unesco.org/en/convention>.
- [2] G. Ergin, "Bridging the gap: Using digital interactives for social museums," *yedi Sanat, Tasarım ve Bilim Dergisi*, 2024. DOI: 10.17484/yedi.1494586.
- [3] A. Cortez, "Sound as a producer of social spaces in museum exhibitions," *Curator: The Museum Journal*, vol. 66, no. 2, pp. 317–328, 2023. DOI: 10.1111/cura.12547. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cura.12547>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cura.12547>.
- [4] Y. Zhang and R. Trocchianesi, "Perspectives of sound: Promoting social inclusion under the principle of "access for all" in museums," *Diid*, vol. 1, Feb. 2024. DOI: 10.30682/diiddsi23t2q.
- [5] D. Rich, "Acoustics on display: Collecting and curating sound at the science museum," *Science Museum Group Journal*, vol. 7, Apr. 2017. DOI: 10.15180/170706.
- [6] Z. Sun, A. Rodà, E. Faresin, E. Whiting, and G. Salemi, "3d virtual reconstruction and sound simulation of an ancient roman brass musical instrument," in *Culture and Computing. HCII 2020. Lecture Notes in Computer Science*, vol 12215. Springer, Springer, 2020, pp. 267–280.
- [7] F. Avanzini et al., "Archaeology and virtual acoustics: A pan flute from ancient Egypt," in *Proceedings of the 12th International Conference on Sound and Music Computing (SMC)*, 2015, pp. 31–36.

REFERENCES

- [8] J. Roberts, L. Lyons, F. Cafaro, and R. Eydt, "Harnessing Motion-Sensing Technologies to Engage Visitors with Digital Data," in *Museums and the Web 2015*, 2015. [Online]. Available: <https://mw2015.museumsandtheweb.com/paper/harnessing-motion-sensing-technologies-to-engage-visitors-with-digital-data/>.
- [9] N. Koumartzis, D. Tzetis, P. Kyratsis, and R. Kotsakis, "A new music instrument from ancient times: Modern reconstruction of the greek lyre of hermes using 3d laser scanning, advanced computer aided design and audio analysis," *Journal of New Music Research*, vol. 44, no. 4, pp. 324–346, 2015.
- [10] D. Mast, J. Broekens, S. I. de Vries, and F. J. Verbeek, "Participation patterns of interactive playful museum exhibits: Evaluating the participant journey map through situated observations," in *Proceedings of the 2023 ACM Designing Interactive Systems Conference*, 2023, pp. 1861–1885. doi: 10.1145/3563657.3595985.
- [11] M. Vorlaender and J. Summers, "Auralization: Fundamentals of acoustics, modelling, simulation, algorithms, and acoustic virtual reality," *The Journal of the Acoustical Society of America*, vol. 123, p. 4028, Jul. 2008. doi: 10.1121/1.2908264.
- [12] A. Farina, *Soundscape Ecology: Principles, Patterns, Methods and Applications*. Nov. 2013, pp. 1–315, ISBN: 978-94-007-7373-8. doi: 10.1007/978-94-007-7374-5.
- [13] M. Jonas Bem, S. Chabot, V. Brooks, and J. Braasch, "Enhancing museum experiences: Using immersive environments to evaluate soundscape preferences," *The Journal of the Acoustical Society of America*, vol. 157, pp. 1097–1108, Feb. 2025. doi: 10.1121/10.0035832.
- [14] S. Madgwick, A. Harrison, and R. Vaidyanathan, "Estimation of imu and marg orientation using a gradient descent algorithm," *IEEE ... International Conference on Rehabilitation Robotics : [proceedings]*, vol. 2011, p. 5975346, Jun. 2011. doi: 10.1109/ICORR.2011.5975346.
- [15] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Sensors*, vol. 12, no. 9, pp. 11734–11753, 2012. doi: 10.3390/s120911734.

- [16] N. Raghuvanshi, C. Lauterbach, A. Chandak, D. Manocha, and M. Lin, "Real-time sound synthesis and propagation for games," *Communications of the ACM*, vol. 50, pp. 66–73, Jul. 2007. doi: 10.1145/1272516.1272541.
- [17] E. Dorigatti and S. Pearse, "Designing a library for generative audio in unity," in *Proceedings of the 26th International Conference on Digital Audio Effects (DAFx-23)*, Copenhagen, Denmark, 2023. [Online]. Available: https://www.dafx.de/paper-archive/2023/DAFx23_paper_71.pdf.
- [18] J. Atherton and G. Wang, "Chunity: Integrated audiovisual programming in unity," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, Blacksburg, Virginia, USA, 2018, pp. 102–107. doi: 10.5281/zenodo.1302695.
- [19] F. Fontana, R. Paisa, R. Ranon, and S. Serafin, "Multisensory plucked instrument modeling in unity3d: From keytar to accurate string prototyping," *Applied Sciences*, vol. 10, p. 1452, Feb. 2020. doi: 10.3390/app10041452.
- [20] F. Avanzini, "Procedural modeling of interactive sound sources in virtual reality," in Oct. 2022, pp. 49–76, ISBN: 978-3-031-04020-7. doi: 10.1007/978-3-031-04021-4_2.
- [21] A. A. Yunanto et al., "Implementation of design patterns on unity components to increase reusability and game speed development," in *2023 International Electronics Symposium (IES)*, Aug. 2023, pp. 366–373. doi: 10.1109/IES59143.2023.10242419.
- [22] N. Bucher, "Introducing design patterns and best practices in unity," in *ACMSE '17: Proceedings of the 2017 ACM Southeast Conference*, Apr. 2017, pp. 243–247. doi: 10.1145/3077286.3077322.
- [23] D. Gibson and M. Cryne, "Game sound: Working practices, technologies, challenges and emerging opportunities," in *SAMIS Conference, SAMIS Conference*, July 2022, Jul. 2022.
- [24] J. M. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the Audio Engineering Society*, vol. 21, no. 7, pp. 526–534, 1973.
- [25] C. Roads, "Introduction to granular synthesis," *Computer Music Journal*, vol. 12, no. 2, pp. 11–13, 1988.

REFERENCES

- [26] M. Jonas Bem, "Effects of sounds on the visitors' experience in museums," Ph.D. dissertation, Rensselaer Polytechnic Institute, Aug. 2023.
- [27] K. Hjortkjær, "The sound of the past: Sound in the exhibition at the danish museum mosede fort, denmark 1914–18," *Curator: The Museum Journal*, vol. 62, pp. 453–460, Jul. 2019. doi: 10.1111/cura.12326.

Acknowledgments

I would first like to thank my supervisor, Professor Antonio Rodà. His guidance and support were very important for this project. He always had time for my questions and gave me the freedom to explore my own ideas.

I also want to thanks to Giulio Pitteri. His technical help and advice were very valuable, especially in the early stages of the project.

I am grateful to all my friends and colleagues in the ICT for Internet and Multimedia program. Our discussions and the time we spent together made my master's journey much better.

Finally, my family—words cannot express my gratitude to them. Their support and encouragement were essential. I could not have completed this thesis without them.