Sound-box for tabletop role-playing games

Sound Computing and Sensor Technology

Project Report
MTA20436

Aalborg University
Electronics and IT

Electronics and IT
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Sound-box for tabletop role-playing games

Theme:
Sound Computing and Sensor Technology

Project Period:
Spring Semester 2020

Project Group:
MTA20436

Participant(s):
Mikkel Sang Mee Baunsgaard
Freja Bøcher Kaastrup Johansen
Andrei-Calin Mares
David Mockovsky
Magnus Kornbeck Thomsen
Oscar Bill Zhou

Supervisor(s):
Taewoong Lee
Mads Græsbøll Christensen

Copies: 1

Page Numbers: 72

Date of Completion:
November 18, 2024

Abstract:

This report describes the process of creating a sound-box meant to be used by a Game Master running a tabletop role-playing game session. The purpose of the sound-box is to enhance immersion in tabletop role-playing games by adding sound tracks, which corresponds to the players' actions. The report explains the theory of the audio processing methods used and the design areas considered when creating this prototype. Furthermore, it describes the implementation process and the evaluation, including MUSHRA listening test and User Experience Questionnaire. Finally, it describes the results gathered by the tests and discusses these findings, as well as prospects for future iterations.

Contents

Chapter 1

Introduction

An immersive atmosphere is an aspect that is desired in multiple forms of media. Additionally, in some of these domains, music is heavily relied upon to generate this immersion. However, there are still platforms such as role-playing tabletop games which have to rely on techniques such as narration, dialogue and description as the backbone of creating immersion. The purpose of this project is to try to introduce music to role-playing tabletop games through the use of a sound-box. This is done in order to make use of the techniques that create immersion, and take inspiration from mediums such as video games, where music has been used successfully for this exact purpose.

The idea was to create a physical standalone box, which would have the ability of playing and applying effects on three sound tracks. There would be two effects that could be added to the tracks: pitch changing and reverberation. In combination, this would add to the versatility of each track, while still keeping everything on a single interface.

The report will describe the background research behind the idea of the project and explain the audio processing methods used for the implementation. Furthermore, it will describe the process of designing, implementing and testing the sound-box, as well as describing how it can be further developed for an improved version of it.

Chapter 2

Problem Research

2.1 Initial Problem Statement

Even with the rapid rise of music in all domains, there are still some remaining areas with potential, which have not yet adopted sound. One such domain is tabletop games. Few tabletop games are designed around sound, and those that exist rely on an external source of music such as phone applications. Sound is already being used as a tool for setting an immersive atmosphere in video games. With this in mind, one has to ask themselves: if audio is so successful and widespread in video games, could it then also be used to enhance tabletop games? Sound could evoke a certain theme, fill in the silence, portray certain emotions, or even describe characters and locations in role-playing tabletop games. As such, adding a device that produces sound, could greatly improve the overall experience of most tabletop games.

Taking this into account, the following Initial Problem Statement (IPS) was created:

> "The immersion of tabletop games can be improved by the addition of a sound-box."

Moving ahead, further research had to be conducted to analyse the problem properly.

2.2 Using Sound In Video Games

To better understand how sound can improve tabletop games, this section will look into how audio is already used in a similar domain: video games.

Sound in video games has been present since the first arcade machines [14]. In the beginning, simple 8-bit and 16-bit melodies were used to get the attention of people passing by and attract them to play. However, as video games gained popularity and the technology evolved, so did the sound within them. With increased popularity, new music genres emerged, which were purely focused on video games [18]. Some modern video games have a "movie-like" experience, with the usage of cinematic techniques and the quality of both visuals and audio. An example of the visual quality can be seen in Figure 2.1.

Figure 2.1: *A scenery from the game Uncharted 4, showing that modern video games can be compared to movies [20].*

Since video games can have multiple outcomes based on player decisions and are not as linear as movies, the audio in games has to be dynamic and correspond with the player's actions. For example, if a player decides to enter a new location, the whole scene has to be changed. In most games this could happen at any moment, whereas in movies scene changes are scripted [14].

Audio in games can be used in many ways. A distinct sound can be played every time a certain event takes place (e.g. if a player picks up an item or levels up) to give the player feedback and associate the sound with that event. Sounds and music can be used to match the current state of the game, like playing action music when being in combat, or even guide the player and suggest what next steps to take. Ambient sound can also be tied to certain locations which can make them special and create a specific atmosphere for that location. Additionally, ambient sounds add to the realism and immersion of the location (e.g. having the sound of rustling leaves in a forest). Another very powerful tool is the use of voice, which can be included by adding dialogue or narration. This can also greatly improve the game, especially the storytelling aspect of it. Using these sounds in games evokes

strong sensory stimuli which in turn makes for a better game experience and can improve the immersion [14].

That being said, it is clear that audio plays an important role in video games. As stated by Peter Peerdman in his article:
"...audio also offers even more opportunities, for instance the ability to evoke emotional responses, immerse players, and create a sense of location within a game." [14]

2.3 Immersion In Games

The concept of immersion in the form of sound was briefly touched upon in Section 2.2. Here, this concept will be further explored by researching its importance, as well as the way immersion is created.

Firstly, immersion is considered to be one of the main reasons why video games are so appealing. It allows for people to escape the limitations of reality and carry their imaginations into a whole new world on the screen. It helps shorten *"the subjective distance between player and game environment"*, offering the sensation of inhabiting the space represented in-game [7].

Similarly, immersion is a core aspect of tabletop role-playing games (TRPGs). Just as regular board games, TRPGs are played face-to-face, but involve players "acting out" a character. The acting tends to be relaxed, meaning that costumes or exclusively speaking in-character is often not the norm. Instead, certain rules are set, and players develop characters within these rules and are responsible for what those characters do over the course of the game. The Game Master (GM) is in most cases the one providing and developing the setting of the game as well as the rules. The GM presents the players with situations, which are referred to as encounters and asks them questions such as "What do you do?". This will lead to the player(s) having to make a choice for their character(s) between multiple options [9].

Immersion can bring a group of players closer and help them engage with the narrative of the game. It is also an incentive for them to return to the game, to further explore or engage with its world. Furthermore, for some people these benefits of immersion are not solely applicable to the role-play genre, instead they like to "explore fantasy worlds in any type of game" [9].

Given the relevance of immersion when it comes to games, researchers have tried to understand immersion and how it can be created. From this research, immersion

was broken down into six categories: immersion in the activity, game, environment, narrative, character, and community [4]. These are valid for both tabletop and video games. However, a difference can be identified. While video games can rely on visuals and audio to create these types of immersion, tabletop games generally have to stick to narration and description as a method for generating immersion.

Taking everything presented previously in consideration, it can be concluded that one aspect which could be integrated into tabletop games that will help with immersion is audio. Given the similarities between the two genres of games, audio can similarly be used in tabletop games as they are already being used in video games. As explained in Section 2.2, it can be used for ambient sounds and music, for evoking a certain location, emotion and for enhancing narration, improving many aspects of immersion in the process.

2.4 State Of The Art

Knowing that sound can play an important role for players and their immersion in games, it is necessary to also explore the available solutions made for tabletop games. As such, multiple solutions made for different platforms can be mentioned. Some examples of these possible solutions will be described with a focus on potential upsides and downsides.

2.4.1 Tabletopaudio.com

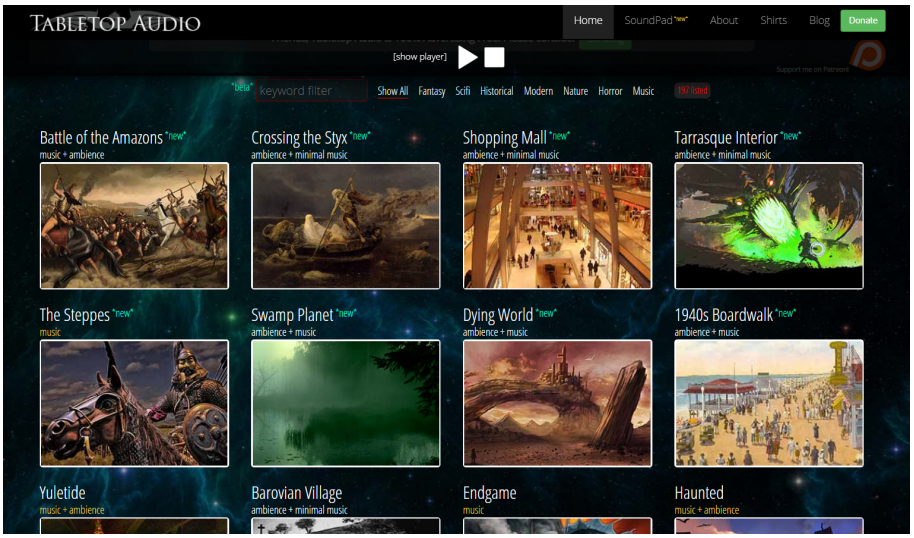


Figure 2.2: *Main page of Tabletopaudio showing thumbnails for certain settings [12].*

Tabletopaudio.com is a website, which contains content free to its users. Since it is a website, this solution can be used by anything with access to an internet connection and browser (e.g. computer, phone, tablet). On the main page, there are numerous ambient soundtracks, which are available to the user. These can be seen in Figure 2.2. These tracks can be categorised into e.g. *Fantasy*, *Scifi* or *Nature* to make it easier for the user to choose a fitting setting for the game. Each track also has a description, which states that the track is either *Music* only, *Ambiance + Music* or *Ambiance + Minimal music*. In case the user wants more control over the sounds in the environment, there is also an option to access a feature called "SoundPad".

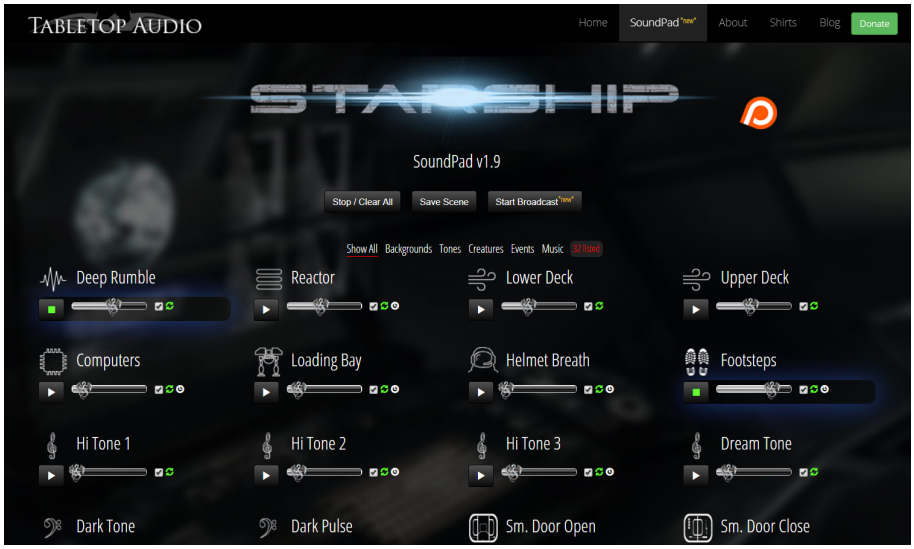


Figure 2.3: *Soundpad for the Starship setting from Tabletopaudio.com [12].*

The "soundpad" contains a variety of sound effects, ambient sounds, and background music. As with the tracks at the main page, there are numerous "soundpads" available, fitting for different occasions such as *The Tavern*, *Jungle Planet*, and *Starship*. This make this solution versatile, and is one of the strong points of this solution. The soundpad for Starship can be seen in Figure 2.3. Users can play single sounds or multiple sounds at once, as well as control the volume of the sounds, loop them or set a timer for them. However, one thing to keep in mind is that it can be troublesome to use if an internet connection is not available. Furthermore, the fact that the sounds are categorised strictly might lead to some downsides as well. While it is true that it is easy to find audio, switching to and from sounds that are found on different "soundpads" will require multiple open tabs and could add to the number of tasks a user has to do.

Similar solutions include a website called ***myNoise.net*** and a downloadable pro-

gram, for computers and phones, called ***Syrinscape***. However, myNoise only contains limited ambient sound effects and no background music and Syrinscape is an application which gives the user three free sound environments, whereas the rest requires payment.

2.4.2 Melodice.org

Melodice.org is a community based website. It allows users to search for specific board games and provides them with a list of songs, fitting for the game, found by other users.

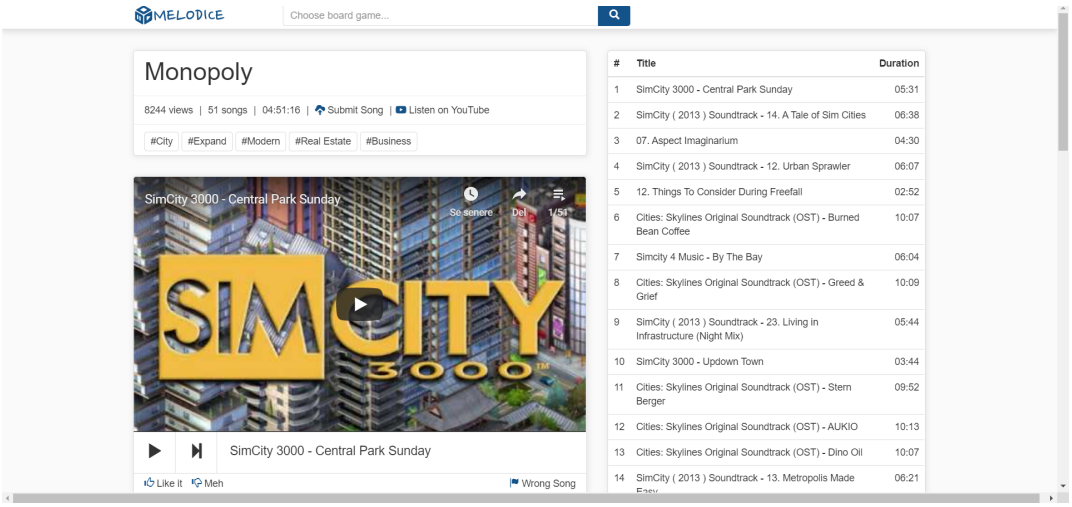


Figure 2.4: *Songs suggested by Melodice for the game Monopoly [13].*

The songs which have been added to the lists are commonly background music from games or movies of a similar theme. An example of this can be seen in Figure 2.4, where the popular board game *Monopoly* has music from *SimCity* on the list.

Should a user encounter a game which does not currently have any listed songs, they are encouraged to submit music to the list. In doing so, the user is instructed to add music which will not be distracting during the game and music which does not contain vocals.

Melodice.org is an easily accessible solution, as long as the user has an internet connection available. Contrary to Tabletopaudio.com, it does not provide the users with sounds effects, but focuses only on ambient background music. Similarly to Melodice, it was found that popular streaming services such as *Spotify* can be used for the same purpose, where users create public playlists for specific games. However, there is no restrictions about the songs which can be added, which can be

both a negative and positive feature. Without restrictions, there is a higher chance of distracting and unfitting songs being added, but also a higher amount of user freedom.

2.4.3 Novation Launchpad

Figure 2.5: *The Novation Launchpad [11].*

The ***Novation Launchpad***, which can be seen in Figure 2.5, is a musical instrument digital interface (MIDI) controller. This solution is mainly used for music creation and works very similarly to a keyboard or synthesizer. When using the launchpad, the user needs to use a digital audio workstation (DAW), such as Ableton Live, to create sounds with the launchpad. Here, the users can customize the sounds assigned to the specific buttons on the launchpad, which can be anything from sounds from the DAW library to custom sounds imported by the user. The interface of the ***Ableton Live*** workstation can be seen in Figure 2.6.

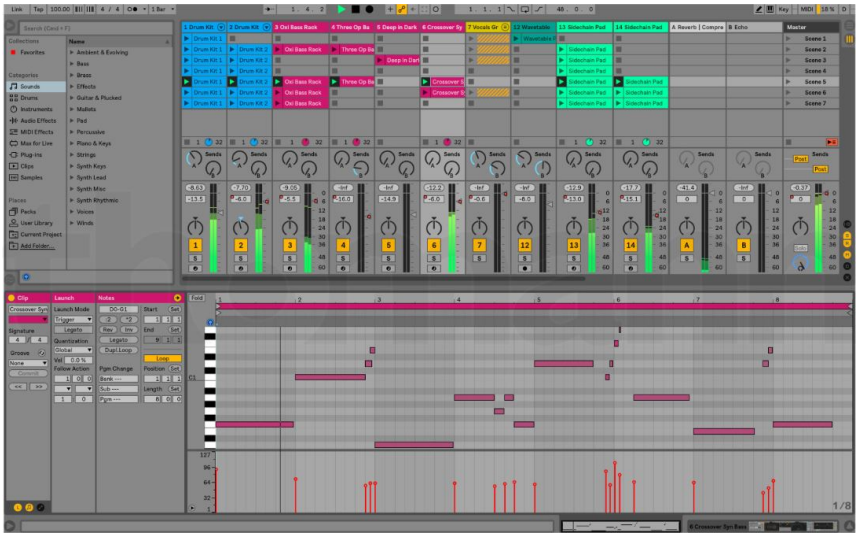


Figure 2.6: *The Ableton Live interface [10].*

The user can set specific sounds to be looped or played once. As the user can import any sounds, the launchpad can be used for more than just creating music - it could also be used as a soundboard, containing music and sound effects fitting for a tabletop game.

The launchpad requires more preparation from the users since they would have to find and import specific sounds themselves. However, it allows a great amount of creative freedom and allows the users to create a setting catered specifically to their needs.

Additionally, a consequence for using the launchpad is, that it requires a lot of expensive hardware, which may not be ideal for many players. Lastly, the interface of the launchpad has no labelling and does not have a lot of free space to allow manual labelling, which means that the user must remember which button contains which sound. The Ableton Live software may also look very confusing and unintuitive for a new user.

Chapter 3

Methods Within Audio Processing

This chapter will go through the different methods within the Audio Processing domain, which will be used for implementation purposes.
Since the purpose of this project is to create a system capable of creating sound, it is important to explain how audio works, as well as how it can be manipulated. In this chapter, audio processing techniques used to accomplish this, as well as the basics of audio, will be presented.

3.1 Sinusoids

To start with, sound signals can be described as the sum of many sinusoids in the form of the formula: $x(t) = A\cos\Omega\, t + \Phi$, with the given attributes being the *amplitudes* (A), *phases* (Φ), and *frequencies* (Ω). These will be explained further in the chapter.

A sinusoid can be explained by using theory from complex numbers. A complex number has a real and an imaginary part and there are multiple ways of representing a complex number. One of these ways, and the one most relevant for audio processing, is as a phasor. The form of a phasor is the following: $z(t) = Ae^{(\Omega\, t+\Phi)}$. As any complex number, a phasor has a real and an imaginary part, and this is where sinusoids appear. In short, a sinusoid is the real part of a phasor, and has the form: $x(t) = Real(z(t)) = A\cos(\Omega\, t + \Phi)$ [8].

3.1.1 The Frequency

The frequency which is denoted Ω and is measured in $\frac{radians}{second}$, determines the number of cycles (in radians) of the sinusoid there are per second (see Figure 3.1). This is known as the *pitch* of the sinusoid. The frequency can, however, also be written as, f, which is measured in Hertz (Hz) and can be used to calculate Ω by using the

formula: $\Omega = 2\pi f$ [8].

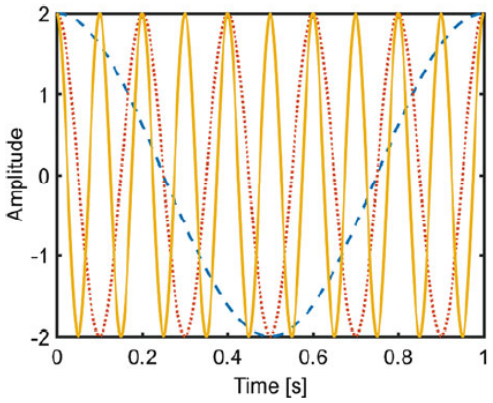


Figure 3.1: *Sinusoids of the form $x(t) = 2\cos\Omega t$ for three different frequencies. 1 Hz (dashed and blue), 5 Hz (dotted and red), and 10 Hz (solid and yellow) [8].*

Controlling Pitch

As mentioned previously, pitch is determined by the number of cycles per second, which means that by increasing or decreasing the duration (in seconds) of a sound, its pitch will be affected. By slowing down the playback speed of a track, the duration of the track is increased. A longer duration means that the new pitch will result from a division with a higher number, and so, it will be lower than the pitch of the original track. The opposite is true as well, if the playback speed is increased, the track is shorter in length (in seconds), resulting in a higher pitch.

3.1.2 The Amplitude

The *amplitude*, A, determines how loud the sinusoid is. It scales the sinusoid from -A to A instead of -1 to 1. A visual representation of different amplitudes can be seen in Figure 3.2 [8].

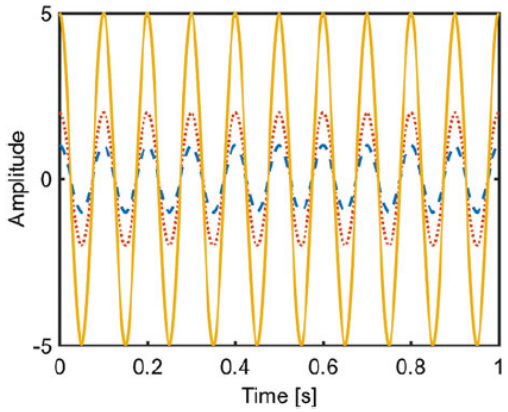


Figure 3.2: *Sinusoids with three different amplitudes. 1 (dashed and blue), 2 (dotted and red), and 5 (solid and yellow) [8].*

3.1.3 The Phase

The *phase*, Φ, is a time-shift of the sinusoid (see Figure 3.3) and can be written as: $\cos(\Omega t + \Phi) = \cos(\Omega(t - t_0))$. Where t_0 is the time-shift corresponding to a phase Φ [8].

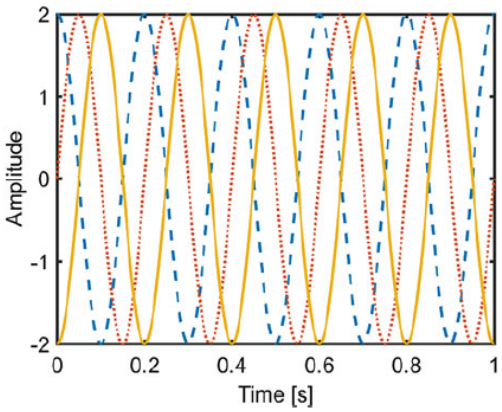


Figure 3.3: *Sinusoids of the form:* $2\cos(2\pi 10(t - t_0))$ *for three different time-shifts denoted as Time[s]. 0 seconds (dashed and blue), 0.25 seconds (dotted and red), and 0.5 seconds (solid and yellow) [8].*

3.2 Digital Audio Signals

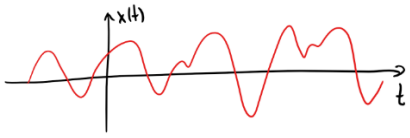
Converting analog sound signals to digital sound is the process of taking a real-world sound and converting it into something the computer can interpret. Analog sound signals are also called continuous-time signals. The problem with representing continuous-time signal on a computer is that, even at a particular time instance,

it can take on an infinite number of values.

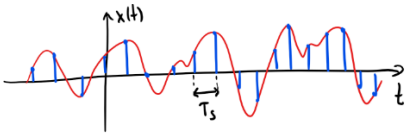
This sort of signal has a value $x(t)$ for every possible time t. Additionally, amplitude-wise, the signal value $x(t)$ can take on any value from a range of numbers. Trying to store a a continuous-time signal in a computer would take an infinite amount of memory. However, a computer can only store a finite amount of numbers, and in turn, these numbers can only take a finite number of values. The solution to this is to use sampling in order to turn that continuous-time signal into a discrete-time signal. This is called sampling and will be explained below.

3.2.1 Sampling

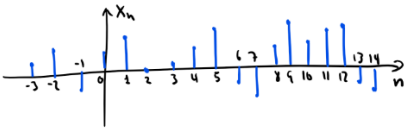
When sampling a continuous-time signal, its values are only measured every T_s seconds, turning it into just a sequence of numbers. T_s is called the sampling period, and the smaller it is, the more finely the original signal is sampled. When representing digital signals, it is more usual to use the sampling frequency (f_s) rather than the the sampling period. This is defined as: $f_s = \frac{1}{T_s}$ and represents the amount of samples obtained per second. By using sampling, as mentioned previously, a continuous-time signal is converted to a discrete-time signal. The difference is that the discrete-time signals only have a value x_n at certain times. These times can be represented as $t_n = nT_s$, where n is called the sampling index. An example of the process a continuous-time signal would go through, when converted to a discrete-time signal using sampling, can be seen in Figure 3.4.



(a) *Continuous-time signal illustrated on the x_t and t axis.*



(b) *Sampling the continuous-time signal with a sampling period of T_s, illustrated on the x_t and t axis.*



(c) *The resulting discrete-time signal, with the sampling indexes $n = ..., -3, -2, -1, 0, 1, 2, 3, ...$, illustrated on the x_t and n axis.*

Figure 3.4: *The process of converting an analog audio signal to a digital audio signal* [3].

To summarise, in programming audio can be represented as a discrete-audio signal. The sampling frequency which is used to sample the signal is also usually represented. By having access to these two values, a signal can be digitally processed and also played.

3.3 Filters

Filtering is the most basic form of audio processing and is used for many different purposes such as amplifying, passing and cutting certain audio frequencies. Additionally, filtering is also the base for creating different effects on a sound.

Figure 3.5: *An illustration of the input, x_n, being passed through a filter to obtain the output, y_n [8].*

As seen in Figure 3.5, a filter is something that takes a sequence of numbers, x_n, and converts it into another sequence of numbers, y_n. This can be referred to as a *simple filter* and comes in the form of *difference equations*, meaning that it involves *"multiplying and adding a number of delayed versions of the input and output [8]"*. A difference equation is given by: $y_n = x_n + ax_{n-1}$, with the input x_n at time n and output y_n. Furthermore, a is the filter coefficient and x_{n-1} is the input delayed with one sample.

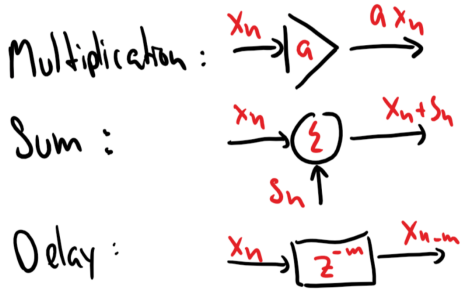


Figure 3.6: *The difference equation operators [3].*

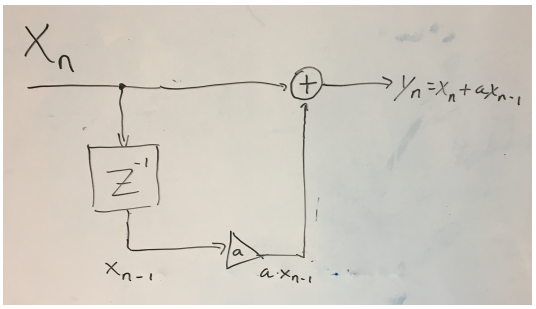


Figure 3.7: *Example of a diagram, which shows a feeedforward filter.*

All difference equations contain some difference operators. These can be seen in Figure 3.6, where each operator is represented with a unique shape. For multiplication, a represents the number that will be multiplied with the original signal. For the sum, S_n represents the number that will be added to the original signal, and for the delay, m, in the z^{-m} notation, represents the number of samples the original signal will be delayed by. An example of how these can be used is illustrated in Figure 3.7. Here, the x_n signal goes through a delay operator, which delays it by one sample, resulting in x_{n-1}. It then goes through a multiplication operator which multiplies it by a, resulting in ax_{n-1}. This result goes through a sum operator and is added together with the original signal, resulting in the output: $y_n = x_n + ax_{n-1}$.

Lastly, there are two fundamental types of filters: feedforward and feedback

filters. Feedforward is when the input x_n is delayed, possibly scaled and added (as shown before with the simple filter example in Figure 3.7). Feedback is when both the input (x_n) and the output (y_n) are used. Its difference equation being: $y_n = x_n + ay_{n-1}$ [8].

3.3.1 Comb Filter

An example of a feedback filter is the comb filter. It has the following general difference equation:

$$y_n = x_n + R^D y_{n-D}$$

As mentioned previously, x_n is the input signal and y_n is the output signal. D is the delay (in samples) and $b < 0$ is the filter coefficient which is (usually) negative. R is a compression factor, often in the size of $0 < R < 1$. An illustration of the comb filter can be seen in Figure 3.8.

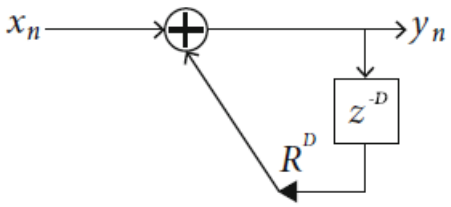


Figure 3.8: *An example of a comb filter which is a feedback filter with a delay of D samples in the feedback path [8].*

A comb filter is usually used to create a certain delay effect. Furthermore, there is also a feedforward version of the comb filter called inverse comb filter. This one can be used to achieve certain audio effects, such as echo, and has the difference equation:

$$y_n = x_n + bx_{nD}$$

3.3.2 All-pass Filter

An all-pass filter is a fractional delay filter. The purpose of the filter is to be able to change the phase without changing the magnitude of the input signal. An all-pass filter with the only filter coefficient being b, has the following equation:

$$y_n = bx_n + x_{n-1} - by_{n-1}$$

Where y_n is the output signal, bx_n is the filter coefficient applied to the input signal, x_{n-1} is the input signal with a delay of d samples, and $-by_{n-1}$ is minus the filter coefficient applied to the output signal with a delay of d samples [8].

3.3.3 Reverberation

Reverberation is a term that refers to the persistence of the sound after it is produced. Put in other words, it is the echo in a closed and small space, such as a room. To create a sound that mimics this effect, an inverse comb filter, mentioned in Section 3.3.1, is used with the following formula:

$$y_n = x_n + bx_{n-D}$$

As mentioned previously, this is a feedforward filter, which will produce an echo of x_n at D samples later, and its strength is influenced by b.

For a more complex sound, it is beneficial to involve many echoes, which can have different delays and gains. To create more of these echos it is then more favorable to use a normal comb filter. This approach to creating a reverb is called a plain reverberator.

Furthermore, there is also an all-pass reverberator. Its name comes from the fact that it is essentially a generalisation of the traditional all-pass filter, described in Section 3.3.2. The difference is that in the all-pass reverberator, the fixed delay of one sample has been replaced by the variable d, which represents the amount of delay the filter should apply:

$$y_n = bx_n + x_{n-d} - by_{n-d}$$

These two types of reverberators can be combined to create a more complex artificial reverb effect, called Schroeder's reverb [8]. This will be explained in the following section.

Schroeder's Reverb

Schroeder's reverb is a design for creating an artificial reverberation effect. It consists of four plain reverberators that are used in parallel, and two all-pass reverberators used in a series [8]. As can be seen in Figure 3.9, the outputs of all the plain reverberators are put together and then processed in succession in the all-pass reverberators.

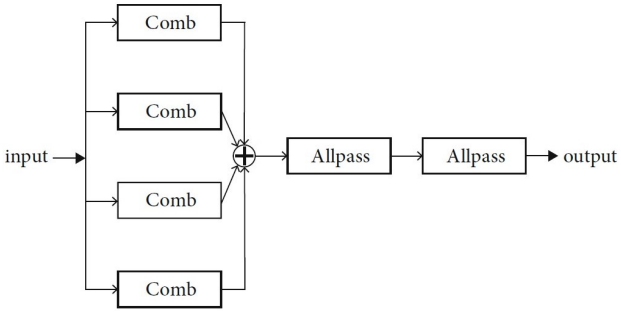


Figure 3.9: *An illustration of components and their interaction in the Schroeder's reverberation algorithm [8].*

This can be written as $H(z) = (H_1(z) + H_2(z) + H_3(z) + H_4(z))H_5(z)H_6(z)$, where $H_1 - H_4$ are the comb filters and H_5, H_6 are the all-pass filters. The formula for $H_1 - H_4$ transfer function is as follows:

$$H_k(z) = \frac{b_k}{1 - a_k z^{-D_k}}$$

Where k is the index number of the comb filter, b_k is the mixing parameter, a_k is the feedback coefficient and D_k is the delay [8]. Alternatively, a_k can be calculated based on the desired reverb time and the plain reverberator delay using the formula: $a = 10^{\frac{3D}{t_{60}fs}}$. Here, D is the delay, t_{60} represents the reverb time and f_s is the sampling frequency. The aforementioned mixing parameters are scalar numbers (between zero and one) that are multiplied with the output signal of the plain reverberators. Their purpose is to control how much each plain reverberator affects the final signal.

Furthermore, the elements in the formula have to be chosen according to certain rules. Firstly, the mix parameters have to sum to one. Secondly, the delays of the plain reverberators have to be large and mutually be prime numbers, whereas the delays of the all-pass reverberators have to be small. Finally, the coefficients of the all-pass reverberators have to not be too close to one. An example of optimal parameters can be seen in Table 3.10 below.

Parameter	$H_1(z)$	$H_2(z)$	$H_3(z)$	$H_4(z)$	$H_5(z)$	$H_6(z)$
D_k	1553	1613	1493	1153	223	443
a_k	0.85	0.85	0.75	0.75	0.7	0.7
b_k	0.30	0.25	0.25	0.20	N/A	N/A

Figure 3.10: *Parameters that should be used in Schroeder's reverb function, considering the sampling frequency is 44100Hz [8].*

Chapter 4

Final Problem Statement

As mentioned previously in Section 2.2, audio is an element that is successfully used in video games to create immersion. Furthermore, as concluded in Section 2.3, it can be used in tabletop games for the same purpose. Moreover, compared to other tabletop games, TRPGs tend to be more reliant on immersion. Thus, they could benefit more from the introduction of music. Due to this, the focus of the project will be on introducing music to TRPGs specifically, rather than tabletop games in general.

Looking at the current state of the art, described in Section 2.4, it can be concluded that people playing tabletop games have multiple ways of adding sounds to their games. However, these solutions require to have either an internet connection or specialised expensive hardware. Some of them also have a very strict categorisation of sound options which results in the opening of multiple tabs to operate, this might lead to a lot of multitasking and slow down the game overall. Furthermore, some solutions have an interface that is unintuitive at first, require too much preparation on behalf of the person running the game or are too expensive.

With this in mind, it can be said that there is a need for creating a device that would provide versatile sounds without the use of the internet, while being inexpensive. This can be achieved with a microcontroller-based device with a single intuitive interface. To summarise all these aspects, the final problem statement is:

"Ambient music and sound effects, which correspond to the players' actions, enhances immersion in TRPGs. The current solutions use either software requiring network or unintuitive and expensive hardware. How can we create a physical device which would provide sound, using a microcontroller, with the purpose of creating a cheap and intuitive alternative without the need for an internet connection?"

4.1 Target Group

Based on the information provided so far, it is apparent that the target group should be people who already play tabletop games, as the premise would be to improve their immersion by the addition of sound. More specifically, this solution would target the person in charge of running the game, the GM. In short, the target group of this project is:

Game Masters running a tabletop role-playing game.

4.2 Summarising The Concept

To summarise the goals and overall idea for the solution, it will be a physical device that would be used during a role-playing tabletop game by the Game Master. This device is supposed to act as a sound box, which has game-specific sound tracks stored on it for the user to play.

To make the sounds tracks more versatile, while still keeping a single interface, sound effects will be used. These effects would be applied to the tracks, allowing them to be used in more situations. For this purpose, two effects will be used. The effects chosen were: reverberation and a pitch modifier. They were chosen in order to be applicable to a large variety of different tracks, while still making sense for the context of the game.

4.3 Success Criteria

For the project to be considered a success, the following success criteria have to be fulfilled.

- Must have an intuitive physical interface, with a high score of perspicuity on the User Experience Questionnaire.

- The tracks must score a minimum score between 60 and 80, which is a corresponding to a grade of "good" in the MUSHRA test.

4.3.1 MoSCoW

Before creating the design, certain priorities will be set to exclude unnecessary design aspects. In order to achieve this, the prioritisation method called "MoSCoW" is used, which is an acronym for *Must have*, *Should have*, *Could have* and *Wont have*. Their meaning can be explained as *Critical requirements*, *Important requirements*, *Desirable requirements* and *Least critical requirements* in that respective order. Taking these categories into consideration the following requirements were created and categorised.

MoSCoW			
Must have	**Should have**	**Could have**	**Will not have**
A microcontroller	The ability to run without being connected to an external device and the internet	Additional sound track	Dynamic filters
Three sound tracks	Three additional tracks	Play multiple sounds at a time	-
Two static sound effects	One additional effect	USB / SD reader	-
Physical container	One soundtrack	Importing of music	-
Speaker	-	Volume control	-
The ability to play the corresponding sound upon button press every time	-	-	-
Labelling for tracks and effects	-	-	-
The ability to play sounds without any unwanted distortions	-	-	-

Chapter 5

Design

After making requirements for the proof of concept, the next step is to design it. This chapter will go through the various considerations made during the design phase of this prototype.

5.1 Initial Design

5.1.1 10 Plus 10 Method And Sketches

Some design elements arise from looking at the MoSCoW table in Section 4.3.1, which have to be included for the prototype. There is a need for physical components, which would provide a choice between at least three sound tracks. Furthermore, a physical component which would allow the choice between at least two effects on these tracks has to be included. Adding to this, a speaker and a container to cover all these features have to be taken into consideration as well.

With this in mind, to explore different designs, the "10 plus 10 method" [6] was used. This method requires that group members work individually to sketch around 10 ideas in total. These ideas are then shared and one or a few sketches are chosen as a new starting point. Next, one or more sketches are elaborated on by everyone, resulting in 10 more sketches, which are more focused in one direction. This results in a final design choice of the desired product.

5.1.2 Design Considerations

A quick breakdown of all the design choices explored by using the 10 plus 10 method will be presented in this section.

System Functionalities

The requirements in the MoSCoW, see 4.3.1, help inform what functions and components the prototype needs to have, in order to achieve its desired functionality. The system must be able to react to user input in regards to selecting a track and then play the track equivalent to the users request. This means, that it must include one or more physical user-adjustable components, which can navigate through the sound tracks, as the system contains multiple tracks. The system should, additionally, include a way to generate feedback corresponding to the sound track that is currently in use, so the user can keep an overview of what is currently happening in the system.

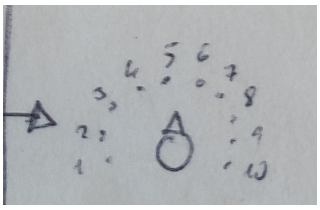
The static effects must also be adjustable by the user, in regards of applying different strengths. The individual strengths of the effects are pre-set and must remain static. As an example, this means that applying the pitch effect at a low strength should lower the pitch at the same rate for each track. Once a song is stopped, all the effects applied to it must be removed.

The user must also be able to control when audio is playing. Ideally, this should be toggled by a user input, e.g. by flipping a switch to start a selected track and doing the same to stop it again. This should, furthermore, toggle the speaker within the system, so the sound can be heard, when it is being played.

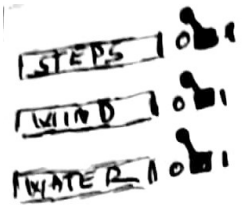
Providing An Interface For Sound Tracks

The sketches contained different ways of representing the sound tracks. For some of them, each sound track was represented as a number around a rotary control knob. This knob could be turned, pointing in the direction of the desired track, which can be seen in Figure 5.1a. Using this approach saves space, as all the options will be present on the same component. However, it may be limiting for future iterations and users might lose track of which number represents which sound track. This drawback was mitigated by the fact that the design came with a “cheat sheet" (a piece of paper on the side of the container) which would show the meaning of each number. Still, it might prove overwhelming for the user to keep track of so many different numbers.

Another approach was the use of buttons or switches, with each button being linked to a track, which can be seen in Figure 5.1b. As such, the user would be able to see the name of the sound track they are about to play, displayed as a label next to its specific button. This approach would provide a more useful understanding of where each sound can be found. However, it will require a large amount of space, if future iterations with more sound tracks are to be considered.



(a) *A sketch illustrating a control knob as the interface for choosing between sound tracks.*



(b) *A sketch illustrating buttons as the interface for sound tracks.*

Figure 5.1: *Examples of different interfaces that can be used for sound tracks.*

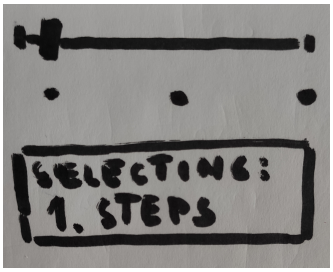
Providing Feedback For Ongoing Sound Tracks

An important aspect of the design is also how the user receives feedback about what tracks are being played. These have been touched upon briefly with the mention of labels, in the case of buttons, as well as the “cheat sheet", in the case of the knob. Another solution proposed to the problem of labelling was the use of a screen. While scrolling through the options using a physical component the name of the corresponding track would show up on the screen. The two physical components that were mostly commonly used in the sketches were the rotary knob and the control slider. The control slider is a type of potentiometer, that instead of rotating around an axis, slides from one side to the other.An example of this can be seen in Figure 5.2a. Overall, it was deemed necessary to have at least one of these two elements to allow the user to understand what sound they were about to play.

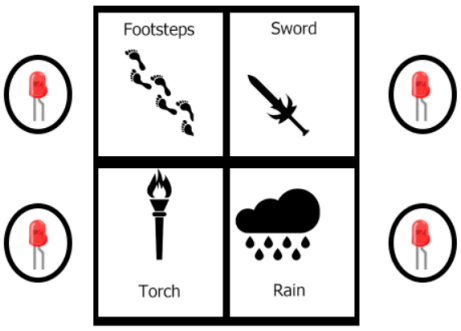
Thad being said, the idea of using small LEDs to provide visual feedback, was explored. The LEDs would light up, when the sound tracks were playing, to give the user an overview of the sounds that are currently in use. However, this would only be useful in conjuncture with buttons. An example of the LED lights and labelling used in conjuncture with each other can be seen in Figure 5.2b.

Providing An Interface For Sound Effects

Next design element considered is how the sound alterations would be provided to the user. Different components were explored in the designs, the most common across all sketches being the knob. However, there were distinctions between the designs utilising this element. Some proposed a knob for each alteration option, which would be labelled with the name of that effect (pitch or reverb), and would have intensity options ranging from Off to Low, to High. An example of this, can be seen in Figure 5.3a. This approach, while easy to use, might take up too much space if too many effects are available. However, since the number of effects would be between two and four, it was considered acceptable. Another approach was to



(a) *A sketch illustrating the combination of a slider with a screen for feedback.*

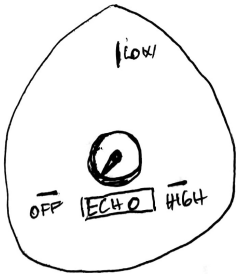


(b) *A sketch illustrating LED lights and labels as feedback.*

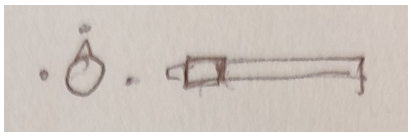
Figure 5.2: *Examples of possible feedback interfaces.*

have two knobs, one which would select the effect, and one which would select its strength. This approach is also favourable, as it would reduce the physical space these options take.

Another considered component was a slider. This would work similarly to the "two knobs" approach described earlier. However, here the second knob was replaced with a slider for modifying the strength of the alteration, as seen in Figure 5.3b.



(a) *A sketch illustrating a knob as the interface for sound alteration.*



(b) *A sketch illustrating a slider and a knob as the interface for sound alteration.*

Figure 5.3: *Examples of different interfaces that can be used for sound alterations.*

Providing A Source Of Power, Storage Space And A Speaker

To provide the basic functionality of the solution, these elements (power source, speaker, storage space) had to be taken into consideration as well. Most of the designs included an SD card slot or a USB port. The speaker was sketched as either being part of the container or connected to it by cable. Using the cable would allow the speaker to be pulled out and moved a short distance. In consideration

to power, the microcontroller would receive it from a rechargeable battery, perhaps with a port on the container for easy charging.

5.2 Final Design

Based on the 10 plus 10 method and the considerations described in Section 5.1.1, the following design was created. As seen in Figure 5.4, this design consist of a slider controlling the selection of the played sound, and an LCD screen that shows what is selected with the slider. Two knobs that control sound effects (pitch and reverb), with each controlling the strength of the respective effects. In this case, the knob can be turned from off to low or high. A play button which is used to play the selected sound track and a stop button that stops the playing sound. Finally, a speaker plays the sound track, that was selected and potentially adjusted, after the play button is pressed. The final design covers all the "must have" requirements from the presented MoSCoW from Section 4.3.1, while utilising some of the design options presented in Section 5.1.2.

Figure 5.4: *A sketch of the initial design showing a speaker on the top left, an LCD screen into the right of it, a slider in the centre below the LCD screen, two knobs on the bottom left, a play and a stop button in the bottom right corner.*

As for a physical interface, which was discussed in Section 5.1.2, the choice and placement of components for the system heavily affect how intuitive the system is. Placing the speaker at the surface with the components affects the system's affordance since it is made clear that using the components will affect sound. Additionally, a combination of sliders, knobs and buttons was chosen. These three controls have a different appearance, which would make the user associate that specific category of controls with a certain action. Assigning specific controls to

specific actions affects the mapping of the systems interface, which should make it less confusing for the user. Sliders were chosen to select tracks to play, however, the aforementioned issues in Section 5.1.2 with a rotary knob would carry over to the slider with more tracks present. This was solved by introducing an LCD screen as extrinsic feedback with labelling, which shows currently selected track, eliminating the need of the aforementioned "cheat sheet".

The slider was also chosen, since the system needs a dynamic continuous control, which would allow the user to quickly select which track they would want to play, with instant feedback on the LCD screen. There would be small marks to indicate the threshold of the different tracks so that if the sliding knob was moved outside of the threshold and into another one, the track would change. This provides the user with feedforward information of how much they would have to slide the knob in order to change track.

The functionality of rotary knobs compared to sliding knobs are almost identical, with exception of the intrinsic feedback of feel with directional movement instead of rotational, which is why the product will have rotary knobs for selecting effects to apply. These knobs would have three different settings: An off, low and high setting for each effect, as mentioned in Section 5.1.2. These settings would be indicated with small labels within a threshold similar to those for tracks. The labels for both the rotary and sliding knobs would work as signifiers for the user, letting them know which direction to turn the knobs to and where to, to get their desired output from the system. Additionally, the labels around the rotary knobs work as a sign of constraints for the user. This is since the user will only be able to apply two strengths of the effects to the sound tracks.

For playing tracks and stopping them, a binary button was chosen for each, with a universally used symbol (play as a triangle, stop as a square) as a label on them for the user to differentiate between them. Discrete control buttons were chosen, since the system would only need a binary input such as playing a song which it would automatically do in a loop when the play button is pressed. The same applies to the stop button, that would stop and reset the tracks when pressed. However, this would mean that the user would have to manually reset all of the knobs back to their off state, to be able to play an unaltered track.

5.2.1 Storyboard

To get a better understanding of how the system would be used, a storyboard is presented. The storyboard can be seen in Figure 5.5. It shows a session of a Dungeons and Dragons game, with the device presented in Section 5.2. Looking at the top row, middle column, the player decides to go to a dungeon. The GM

chooses the sound of a torch with the slider set to the left and adds reverb to the sound by turning the knob labelled "Reverb" in the clockwise direction. The screen also informs the GM which sound is being chosen. He plays the sound by pushing the "Play" button. On the next panel, the GM introduces someone walking towards the player by playing a sound of footsteps. First, they select the footsteps with the slider. To give the impression that the person is walking slowly, the pitch is set to low. The next panel, on the bottom left, works the same way, by choosing and playing a sound based on the situation. Finally, the middle panel on the bottom shows the GM stopping the sound by pressing the "Stop" button.

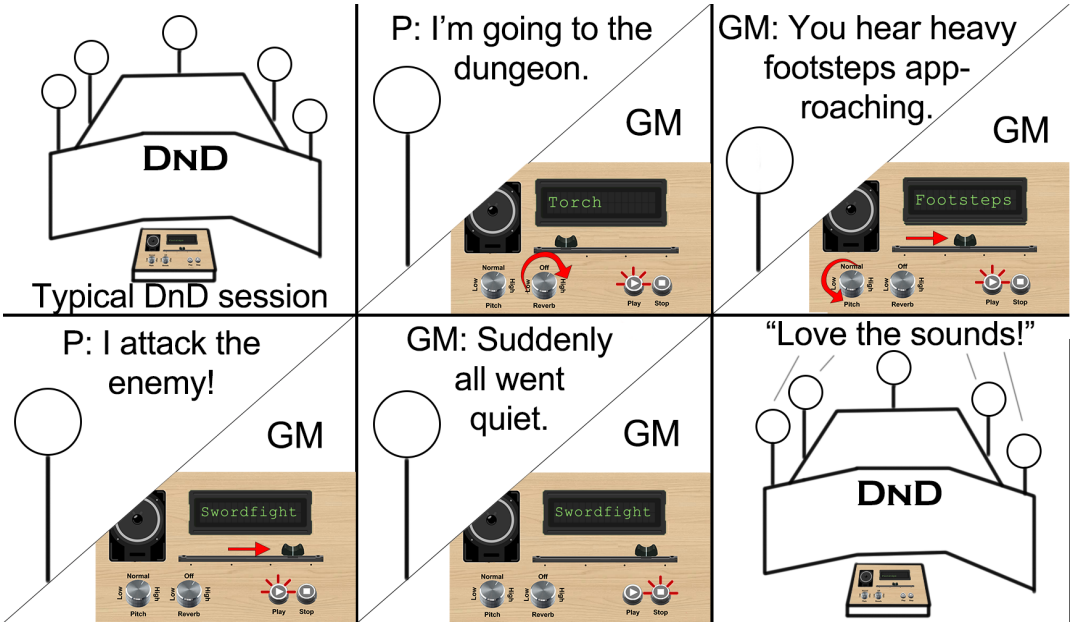


Figure 5.5: *Storyboard showing basic use of the prototype during a regular Dungeons and Dragons (DnD) session. It is shown from a perspective of the person running the game (the GM). Capital "P" represents a player of the game interacting with the GM.*

Chapter 6

Implementation

In this chapter, the implementation of the hardware and software will be covered.

For maintaining version control of the code, GitHub [5] was used, which allows multiple people to share and merge files in the same repository. The physical interface was created on an Arduino [1], and the code was written using the python programming language [21].

An Arduino is a single-board microcontroller with sets of digital and analogue pins allowing for connecting components to it, making it ideal as a physical interface for this project. The board used is a replica of the R3 Arduino Uno, which was chosen due to its simplicity and affordability. However, it will be further referred to simply as Arduino.

6.1 Delimitations

Since the Arduino itself has its limitations, both in physical and in computational aspect, some changes had to be made that ultimately ended up altering the design and functionality of the device as they were presented in Secton 5.2. These changes involve shifting all the audio processing from the Arduino to a computer. The second major difference is removing the speaker from the prototype and using the computers speaker. This makes this version of the prototype more of a remote than a standalone device that plays sound. However, for future iterations it is desired to use a more capable microcontroller or a microcomputer, that would satisfy the ideas presented in the Design chapter 5.

6.2 Arduino Circuit Schematics

The circuit for the sound-box can be illustrated by the following schematic diagrams, which shows the system components mapped with each other in relation to the Arduino. Figure 6.1 represents the circuit with the buttons and potentiometers and Figure 6.2 represents the circuit with an LCD screen. In practice, all components are connected to the Arduino, however, the figures are split up to provide a better overview.

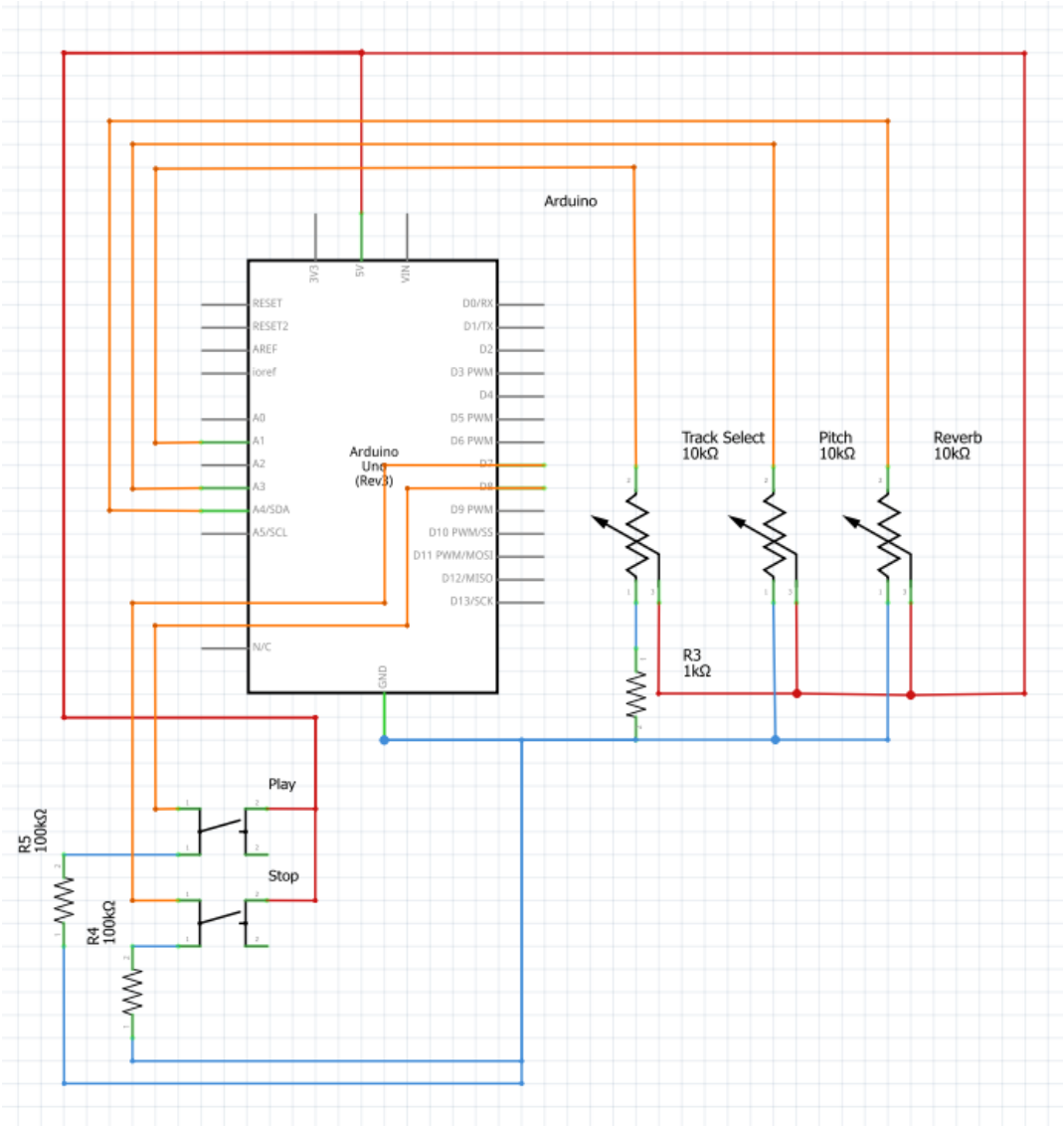


Figure 6.1: *Schematic of the interactive parts of the physical interface.*

Figure 6.1 showcases a circuit with buttons and potentiometers with colour coded wires being red (current), blue (ground) and orange (pins). The potentiometers can be seen to the far right represented as adjustable resistors with three terminals which take both the current and ground and outputs a corresponding voltage. The buttons can be seen at the bottom left as momentary switches that are normally closed.

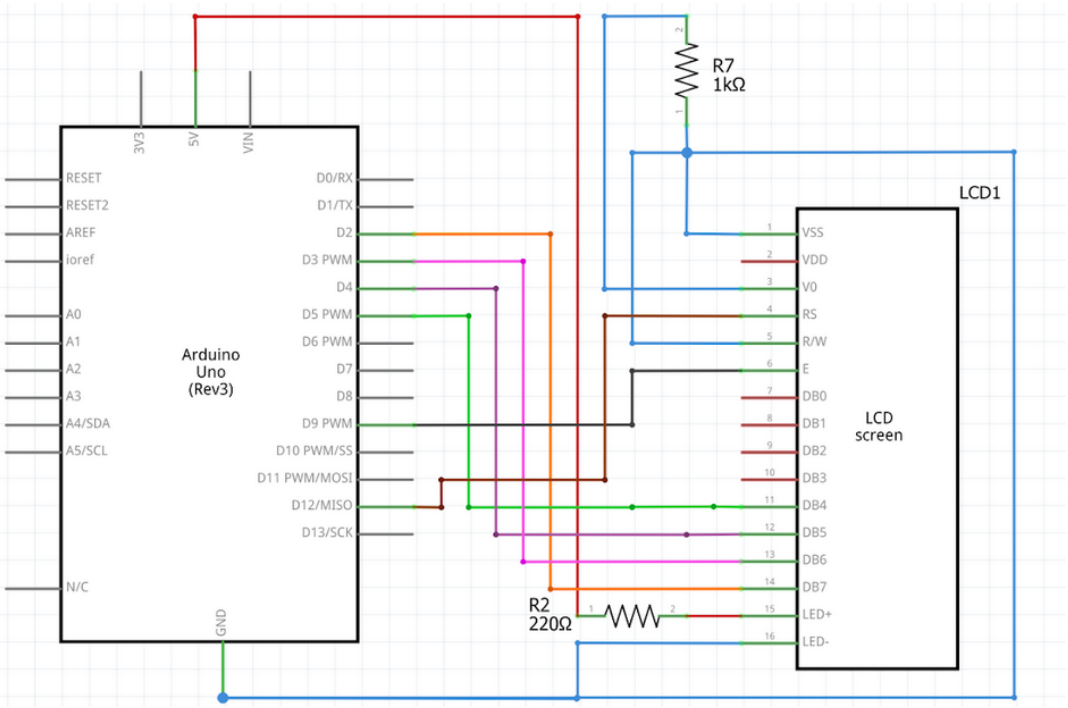


Figure 6.2: *Schematic of the connections between the Arduino Uno and the LCD screen.*

The pin wires in Figure 6.2 are represented differently from Figure 6.1. This is due to the amount of pin wires, where they are all represented by a different colour so it would be easier to differentiate between them. However, the ground is still represented by a blue wire and the current by a red. The reason for the LCD receiving multiple grounds, is that the LCD itself has to be powered through backlighting (LED+, LED-) and contrast (V0). A 220Ω resistor is connected to LED+, so that it does not receive the full voltage which might cause the LCD to short-circuit. Furthermore, a 1kΩ resistor is connected to the V0 to adjust the contrast on the screen.

6.3 Code Overview

As mentioned previously, this version of the box acts purely as an interface. For this reason, the code can be broken down into two parts, one which controls the interface, and one which controls the audio processing.

The first part handles the Arduino and is the code that constantly reads and interprets the inputs from the user and prints them in the serial monitor. This can be seen as a bridge between the computer and the Arduino, allowing the Arduino to send information to the other part of the code. Furthermore, this part also handles what the LCD screen (the one connected to the Arduino) should display. The aforementioned inputs are the values from the potentiometers and the state of the two buttons (pressed or unpressed). The potentiometer values are transformed in such a way that the input printed in the serial monitor is only an integer between 1 and 3, denoting the strength of a desired effect. When one of the buttons is pressed, a string corresponding to that button is printed in the serial monitor. The two possible strings are: "playButton" and "stopButton". In short, this part of the code is isolated from the rest, and the only way it is linked with the rest of the system is through the serial monitor. This part will be explained in its own section.

The second part of the code is the one that processes the audio. It takes the inputs that are printed in the serial monitor, and calls different functions based on those inputs. This is done to apply the effects with the chosen strength on the desired track. It then plays the modified track using the speaker connected to the computer.

6.4 Arduino Code

This code's main purpose is to get input data from the physical interface and send it to the rest of the python script, which will handle the audio processing. In addition, the code displays the appropriate song on the LCD screen.

To understand how the inputs can be read, the way the components work has to be explained first. The components described in Section 5 are connected to the analog and digital pins on the Arduino with current flowing through the circuit. This current influences what values are read from the pins. The difference between the two types of pins is that, when read, digital pin values can either be "LOW" or "HIGH", whereas analog pins have a wider range of values which it fluctuates between. For this reason, buttons are connected to digital pins (since a button can either be pressed or not pressed), whereas potentiometers are connected to analog pins.

Most of the code presented in this section has to do with reading these values from the potetiometers and buttons and interpreting them correctly. For this reason, this code can be divided into three main parts: initialising and reading the values, interpreting these values and sending the values to the serial monitor.

There is also a part of the code which controls the LCD, however, this part is separate from the rest since the LCD only communicates with the slider. It also does not influence the audio processing part of the code which comes later.

6.4.1 Initialising And Reading Input Values

Firstly, the user's input have to be read and stored in appropriate variables. This is what this part of the code does.

```
1  #include <LiquidCrystal.h>
2
3  int potPinLCD = A1;
4  int potPinVol = A5;
5  int potPinReverb = A3;
6  int potPinPitch = A4;
7  LiquidCrystal lcd(12, 9, 5, 4, 3, 2);
8  int button = 8;
9  int button2 = 7;
10 int selectedTrack;
11 int selectedReverb;
12 int selectedPitch;
13
14
15 void setup() {
16     Serial.begin(115200);
17     lcd.begin(16, 2);
18     pinMode(button, INPUT);
19     pinMode(button2, INPUT);
20     delay(1000);
21
22 }
```

Figure 6.3: *Code snippet from showing how variables are declared and initialised.*

Figure 6.3 shows the first lines of the Arduino code. Here (line 3 - 6), four variables are initialised, which represent the analogue PIN to which each potentiometer is physically connected to. In similar fashion, the variables which store the digital PIN, which the buttons are connected to are also declared (line 8 - 9). These pin variables are what will be used in order to specify which component's values should be read. Additionally, three other variables are declared here (line 10

- 12). These variables will hold the value that will be printed in the serial monitor at the end of the code, and they represent the choices of the user. As such, they will hold the selected track, reverb and pitch strength. Furthermore, the button pins have to be interpreted as inputs, and to achieve this the $pinMode(pin, mode)$ function (line 18 - 19) is used.

The LCD screen is also set up in this snippet of the code. The LiquidCrystal library[2] (line 1) is used for initializing and displaying text on an LCD. This library is instantiated by using PINs as arguments (line 7). This tells the program how the LCD is connected to the Arduino and what pins it should use to communicate with the LCD. By creating a liquid crystal object (line 7) the code is able to run class functions from the library in order to modify what is written on the LCD. The $lcd.begin(cols, rows)$ function (line 17) initialises the LCD interface and specifies the dimensions of the screen to 16 columns and 2 rows.

Lastly, the $Serial.begin(speed)$ function (line 16) specifies the data rate in bits per second (baud) for communication with the serial monitor and the $delay(ms)$ function delays the main thread with 1000 milliseconds.

```
24  void loop() {
25    int valPotLCD = 0;
26    int valPotReverb = 0;
27    int valPotPitch = 0;
28    valPotLCD = analogRead(potPinLCD);
29    valPotReverb = analogRead(potPinReverb);
30    valPotPitch = analogRead(potPinPitch);
31
```

Figure 6.4: *Line 24 - 31 in the loop function.*

Figure 6.4 shows the $loop()$ function (line 24) which loops continuously. This is where most of the functionality of the board is coded. Each physical component is represented in code by a variable holding their pin number, as has been explained before. However, for the potentiometers, it is wise to also create variables which would hold the actual value read by the Arduino. Three of these variables are initialized (line 25 - 27), one for each potentiometer. To assign the reading to them, the $analogRead(pin)$ function is used (line 28 - 30), which returns the analogue reading from the specified pin.

Overall, after these lines of code, the input values from the potentiometers have been read, and the buttons are ready to be checked as well.

6.4.2 Interpreting Input Values

After having read and stored the values from the components, the next step is to threshold these values, to transform them into a single integer that will then be sent as input to the python code that deals with audio processing. This is achieved in the same $loop()$ function that was mentioned before.

```
49  //Reverb---------------------------------------
50   if (valPotReverb > 0 && valPotReverb < 341){
51      selectedReverb = 1;
52   }
53   if (valPotReverb > 341 && valPotReverb < 682){
54      selectedReverb = 2;
55   }
56   if (valPotReverb > 682 && valPotReverb < 1023){
57      selectedReverb = 3;
58   }
```

Figure 6.5: *Reverb selection.*

```
60  //Pitch----------------------------------------
61    if (valPotPitch > 0 && valPotPitch < 341){
62      selectedPitch = 1;
63   }
64    if (valPotPitch > 341 && valPotPitch < 682){
65      selectedPitch = 2;
66   }
67    if (valPotPitch > 682 && valPotPitch < 1023){
68      selectedPitch = 3;
69   }
```

Figure 6.6: *Pitch selection.*

As mentioned previously, there is a variable for each potentiometer which holds its value. This value ranges from 0 to 1023. However, these values have to be transformed into an integer from 1 to 3. This is since, in reality, the user only has three choices when it comes to the strength of the effect, not 1024. As such, the values read from the potentiometers have to be thresholded. As seen in Figure 6.5 and 6.6 the if statements achieve this objective by mapping the read value into one of three options. Depending on the value of the input from the physical component, the variable that will be sent trough the serial monitor is modified to either 1, 2 or 3.

```
32  //LCD-----------------------------------
33    if (valPotLCD > 0 && valPotLCD < 150){
34      selectedTrack = 1;
35      lcd.setCursor(0, 1);
36      lcd.print("Torch          ");
37    }
38    if (valPotLCD > 150 && valPotLCD < 275){
39      selectedTrack = 2;
40      lcd.setCursor(0, 1);
41      lcd.print("Swords         ");
42    }
43    if (valPotLCD > 275 && valPotLCD < 1023){
44      selectedTrack = 3;
45      lcd.setCursor(0, 1);
46      lcd.print("Footsteps      ");
47    }
```

Figure 6.7: *Track selection and LCD display.*

In addition to selecting the desired pitch and reverb, the user will also have to select a track. This is handled in a similar manner to the previously shown code. However, the code controlling the display of the LCD is controlled at the same time as the track selection. In Figure 6.7, the variable for the selected track is set to either one, two or three (line 38, 43, 48) based on the value of the slider. Afterwards, it uses the function $lcd.setCursor(col, row)$ which specifies where on the LCD it should place the text (line 39, 44, 49). Lastly, the function $lcd.print(data)$ prints the variable data onto the LCD screen (line 40, 45, 50).

6.4.3 Sending The Input

At this point in the Arduino code, the input values have been read and transformed into single integer variables that can be used later. The only step left is to send these variables through the serial monitor to the rest of the code.

```
71    if (digitalRead(button) == HIGH){
72      Serial.print("selectedTrack");
73      Serial.println(selectedTrack);
74      Serial.print("selectedVol");
75      Serial.println(selectedVol);
76      Serial.print("selectedReverb");
77      Serial.println(selectedReverb);
78      Serial.print("selectedPitch");
79      Serial.println(selectedPitch);
80      Serial.print("playButton");
81      Serial.println(1);
82      delay(1000);
83      }
84
85    if (digitalRead(button2) == HIGH){
86      Serial.print("stopButton");
87      Serial.println(1);
88      delay(1000);
89    }
90
91    lcd.setCursor(0, 0);
92    lcd.print(valPotVol);
93  }
```

Figure 6.8: *Printing data to the serial monitor.*

Sending the data to serial monitor can be seen in Figure 6.8. When the user presses the play button, the value of the digital pin associated with it will be "HIGH" (this value is read using $digitalRead(pin)$). This means that the if statement in line 86 will run if the play button is pressed. If this happens, the program will send all the variables, which hold the user's inputs, to the serial monitor, as well as the "playButton" string, informing the rest of the program that the button has been pressed.

On the other hand, if the user presses the stop button (line 100) it will go down to the other if statement where the string "stopButton" is printed in the serial monitor, informing the rest of the code that music should be stopped.

To summarise, this part of the code reads the input from physical components, turns them into a single integer for potentiometers (or a string for buttons) and sends those to the rest of the code by printing them into the serial monitor. All this code is present in a single file.

6.5 Audio Processing Code

As mentioned previously, the purpose of the audio processing code is to filter a selected sound file based on the inputs provided by the Arduino through the serial monitor. It needs to be able to apply a reverb effect, change the pitch of a track, play a certain track, and stop that track from playing.

This part of the code is made out of five python files. One of them contains a function which can be called to change the pitch of a track. Another contains multiple functions which are all used together to create a function which can apply Schroeder's Reverb, as described in Section 3.3.3. The third file contains a function which will create a copy of the track selected by the user. This makes sure that the effects are not applied to the original tracks. The fourth file has two functions, one for playing the copy of the track and one for stopping the playback and deleting the copy. The final file is called "main", which makes use of all the previously mentioned files and calls them, giving the information taken from the serial monitor as input.

To present this part of the program, each file will be explained separately, one by one. Finally, the main file will be shown, which will make use of all the other functions.

6.5.1 Changing The Pitch

One of the files is called "changePitch". It contains a function with the same name, which has the purpose of changing the pitch of a track. This is done by changing the track's playback speed, as mentioned in Section 3.1.1

```
5   def pitch(inputSignal, factor):
6       if factor == 1:
7           indices = np.round(np.arange(0, len(inputSignal)-1, 0.8))
8           return inputSignal[indices.astype(int)]
9       if factor == 2:
10          return inputSignal
11      if factor == 3:
12          indices = np.round(np.arange(0, len(inputSignal)-1, 1.2))
13          return inputSignal[indices.astype(int)]
```

Figure 6.9: *The changePitch function, which purpose is to change the pitch of a sound file based on user input and return a new output signal with the applied pitch.*

As seen in Figure 6.9, the function takes two inputs - an array containing the

data from the sound file denoted as "inputSignal" and a factor, which represents the user's input. This factor can be an integer from 1 to 3, as mentioned previously. The way the function works is essentially by taking the input signal and creating a copy of it based on the "indices" array (lines 8 , 13). This array contains numbers that range from zero to the length of the "inputSignal" array, but instead of iterating one number at a time, it iterates by a number determined by "factor" (lines 7 , 12). As such, two neighbouring elements in the array will have the form: n and $n + factor$, where n is any element of the array except the last one.

The copied array will only contain those elements from the input signal, which are at the indices contained within the "indices" array. Thus, the copy can either have certain samples removed, if the number is above one or have samples added if the factor is below one, by repeating some or all existing samples. An example of increasing the pitch can be seen when the factor is 3 (line 11). Here, the iteration number is set to 1.2 (line 12), which would remove every fifth element within the array. Opposed to an example of decreasing the pitch, which can be seen when the factor is 1 (line 6), where the iteration number is 0.8 (line 7). This would result in some elements being repeated, thus adding to the length of the signal. The function then returns the new signal with the changed pitch, which was created from the input signal and the factor. If the factor is 2 (line 9), it means the user has put the knob in the state labeled as "normal", meaning that the pitch should not be changed. In this case, the input signal is returned directly as output signal, with no modifications (line 10).

6.5.2 Applying Schroeder's Reverb

Another file in the project is the "applyReverb". As stated in the name, the purpose of this file is to provide a function which can apply a reverb effect to a signal.

```
7   def combfilter(inputSignal, filterCoefficient, delay):
8
9       signalLenght = np.size(inputSignal)
10      outputSignal = np.zeros(signalLenght)
11
12      for n in np.arange(signalLenght):
13          if n < delay:
14              outputSignal[n] = inputSignal[n]
15          else:
16              outputSignal[n] = inputSignal[n]+filterCoefficient*outputSignal[n-delay]
17
18      return outputSignal
```

Figure 6.10: *The combfilter function, whose purpose is to run an input signal through a comb filter based on a certain coefficient and a delay time in samples.*

As mentioned previously in Section 3.3.3, to achieve Schroeder's Reverb, one has to use both comb filters and all-pass filters. As such, both had to be implemented.

The implementation of a comb filter can be seen in Figure 6.10. This is done in the form of a function, which takes three arguments as inputs. These arguments each represent a variable in the comb filter formula presented in Section 3.3.1: $y_n = x_n + R^D y_{n-D}$. "inputSignal" is the first argument and it is used in several functions, which will be seen later. In all of them, this is an array which holds the audio data from a file previously read. Each element of this array represents a sample of the audio. Referencing back to the mentioned formula, the "inputSignal[n]" (line 14, 16) takes the place of x_n. Moving on to the next part of the formula, the function's next input is a filter coefficient. The last input the function takes is a delay, which has to be represented in samples.

As for the code itself, an empty array of the same length as the input array is created (line 9 - 10). This will hold the output signal. To apply the formula, the program loops through the array (line 12). The first samples of the signal (the ones that are produced before the first delay should appear) are copied directly into the output signal (line 13 - 14). Then the comb filter formula is applied to the rest of the samples, producing a delay (line 16). In code, "outputSignal[n-delay]" represents y_{n-D} from the formula. Finally, the output signal is returned.

```
21  def allpassfilter(inputSignal, filterCoefficient, delay):
22
23      signalLenght = np.size(inputSignal)
24      outputSignal = np.zeros(signalLenght)
25
26      for n in np.arange(signalLenght):
27          if n < delay:
28              outputSignal[n] = inputSignal[n]
29          else:
30              outputSignal[n] = filterCoefficient * inputSignal[n] + inputSignal[n - delay] - filterCoefficient * outputSignal[n - delay]
31
32      return outputSignal
```

Figure 6.11: *The allPassFilter function, whose purpose is to run an input signal through an all-pass filter based on a certain coefficient and a delay time in samples.*

The other filter that is implemented is the all-pass filter. This can be seen in Figure 6.11. The function for this filter takes the same arguments as the comb filter function. In similar fashion, it also creates an empty array which will be used as output (line 23 - 24), and skips the samples before the first delay (line 28). Afterwards, the all-pass filter formula, which was explained in Section 3.3.2, is applied to the input signal: $y_n = bx_n + x_{n-d} - by_{n-d}$. "outputSignal[n]" represents y_n, "filterCoefficient" is b, "inputSignal[n]" is x_n, "inputSignal[n-delay]" is x_{n-d} and "outputSignal[n-delay]" is y_{n-d}. In the end, the new output signal is returned.

```
35  def reverb(inputSignal, mixingParameters, combDelays, combFilterParams, allPassDelays, allPassParams):
36      signalLenght = np.size(inputSignal)
37      outputSignal = np.zeros(signalLenght)
38
39      combFiltersNrs = np.size(combDelays)
40      for n in np.arange(combFiltersNrs):
41          outputSignal = outputSignal + mixingParameters[n] * combfilter(inputSignal, combFilterParams[n], combDelays[n])
42
43      allPassFilterNrs = np.size(allPassDelays)
44      for n in np.arange(allPassFilterNrs):
45          outputSignal = allpassfilter(outputSignal, allPassParams[n], allPassDelays[n])
46
47      return outputSignal
```

Figure 6.12: *The reverb function, whose purpose is to create a reverb effect using plain and all-pass reverberators.*

The comb and all-pass filter functions are then used together in *reverb*, which is a function that generates a reverb effect on an "inputSignal", and its code can be seen in Figure 6.12. As additional inputs, it requires four delays for the comb filters, in the form of the "combDelays" array, four comb filter parameters, in the form of the "combFilterParams" array, two delays for the all-pass filters, in the form of the "allPassDelays" array, and two all-pass filter coefficients, in the form of the "allPassParams" array. This is since, as mentioned in Section 3.3.3, Schroeder's Reverb consists of four parallel comb filters and two all pass-filters. Additionally, the function needs mixing parameters, as described at the end of Section 3.3.3, which are provided in the "mixingParameters" array.

This function creates an empty array for the output signal (line 36, 37), then it calculates the signal after it passes through the four comb filters (lines 39 - 41). Since the filters are in parallel, the input signal is run here through the *combfilter* function four times. However, each time, there is a different delay, coefficient, and mixing parameter. These results then get added to the output signal (line 41). Furthermore, this output signal is then taken and run through the all-pass filters in order to get the final result (lines 43 - 45). These filters get applied in series, which means that the second all-pass filter gets applied on the signal that was outputted by the first all-pass filter. This happens in line 45.

```
58  def applyReverb(inputSignal, sampleFreq, strength):
59      inputSignal = inputSignal/max(inputSignal)
60      mixingParams = np.array([0.3, 0.25, 0.25, 0.20])  # they need sum to 1
61      combDelays = np.array([1553, 1613, 1493, 1153])  # they need to be large and have mutually prime numbers
62      allPassDelays = np.array([223, 443])  # they need to be small
63      allPassParams = np.array([-0.7, -0.7])  # they need to not be close to 1
64
65      if strength == 1:
66          reverbTime = 0.7
67          combFilterParams = createCombFiltersParams(reverbTime, combDelays, sampleFreq)
68          return reverb(inputSignal, mixingParams, combDelays, combFilterParams, allPassDelays, allPassParams)
69      if strength == 2:
70          reverbTime = 1.2
71          combFilterParams = createCombFiltersParams(reverbTime, combDelays, sampleFreq)
72          return reverb(inputSignal, mixingParams, combDelays, combFilterParams, allPassDelays, allPassParams)
```

Figure 6.13: *The applyReverb function, whose purpose is to initialise the parameters for the reverb effect and apply it to a signal.*

As previously explained, four comb filters and two all-pass filters have to be used for the reverb effect. As such, there is need for multiple coefficients (parameters) and delays for all the different filters, hence the need for arrays. These are initialised in the *applyReverb* (lines 60 - 63), function which can be seen in Figure 6.13, and then used as inputs to call the aforementioned *reverb* function.

```
50  def createCombFiltersParams(reverbTime, combDelays, samplingFreq):
51      combeDelaysNrs = np.size(combDelays)
52      combFilterParams = np.zeros(combeDelaysNrs)
53      for n in np.arange(combeDelaysNrs):
54          combFilterParams[n] = 10**(-3*combDelays[n]/(reverbTime*samplingFreq))
55      return combFilterParams
```

Figure 6.14: *The createCombFiltersParams function, whose purpose is create the coefficients for the plain reverberators, based on reverberation time and the desired delays.*

There is, however, one exception, and that is the comb filter coefficients. These are created based on the comb filter delays by using the formula described in Section 3.3.3: $a = 10^{\frac{3D}{t_{60}fs}}$. This formula is applied through the *createCombFiltersParams* function, which can be seen in Figure 6.14. As for the delays and the all-pass coefficients, they were chosen using the rules described in Section 3.3.3.

Finally, the *applyReverb* functions returns the signal after it went through all the modifications described above.

6.5.3 Creating The Track

To be able to modify the same sound file multiple times without overlapping applied effects a function that creates a copy of a track is needed. Therefore the file

"createTrack" was created.

```
4   #Tracks are read here
5   samplingFreq1, signal1 = wave.read("torch.wav")
6   samplingFreq2, signal2 = wave.read("swords.wav")
7   samplingFreq3, signal3 = wave.read("footsteps.wav")
8
9   #Copy is created here
10  def createTrack(sliderSong):
11      if sliderSong == 1:
12          wave.write("selectedTrack.wav",samplingFreq1,signal1)
13      elif sliderSong == 2:
14          wave.write("selectedTrack.wav",samplingFreq2,signal2)
15      elif sliderSong == 3:
16          wave.write("selectedTrack.wav",samplingFreq3,signal3)
```

Figure 6.15: *The file containing the createTrack function.*

As can be seen in Figure 6.15, three tracks are read (line 7 - 9), which are the sound of a torch, swords, and footsteps. These are the tracks that the user will be able to modify and/or playback. The function *createTrack* is defined (line 10). The function takes one input, which is the selected track (either 1, 2 or 3). This input will be provided through the serial monitor and determines which track should be copied. This copy is created by writing a new file with the same sampling frequency and signal as the track selected by the user. This copy can then be modified with the selected effects while the original track is unchanged.

6.5.4 Playing And Stopping Playback

As mentioned before, one of the files in this part of the code contains functions which take care of playing and stopping a track. The implementation of both of these functions can be seen in Figure 6.16.

```
8   samplingFreq, signal = wave.read("selectedTrack.wav")
9
10  def buttonPlay():
11
12          sd.play(signal, samplingFreq)
13
14  def buttonStop():
15
16          sd.stop()
17          try:
18              os.remove("selectedTrack.wav")
19          except:
20              print("nothing to delete")
```

Figure 6.16: *The file containing the buttonPlay and buttonStop functions.*

Firstly, the track which would have been created previously by the *createTrack* function is read. The *buttonPlay* then simply plays that track, since all the effects are applied directly on it. The *buttonStop* function, as the name implies, stops the playback. Additionally, it deletes the copy of the selected track.

6.5.5 Main

As mentioned previously, "main" is the file that assembles all the aforementioned functions and executes them. It serves as the point of execution and handles getting the input from the Arduino, through the serial monitor, and running the functions accordingly.

Figure 6.17: *Setup lines in main.*

As seen in Figure 6.17 all the functions are imported from their respective modules (lines 1 - 6) which allows the program to use them later in the code. Once the imports have been completed a $Serial(port, baudrate)$ object is created (line 8) which allows python to read the bytes of a serial monitor with a specified port and baud rate. Lastly, a dictionary is created (line 9 - 11) to store the values received from the serial monitor later in the code.

```
13  def readInput(string):
14
15      if "selectedTrack" in string:
16          trackNumber = [int(i) for i in string if i.isdigit()]
17          createTrack(trackNumber[0])
18
19      elif "selectedReverb" in string:
20          reverbNumber = [int(i) for i in string if i.isdigit()]
21          variables["reverb"] = reverbNumber[0]
22
23      elif "selectedPitch" in string:
24          pitchNumber = [int(i) for i in string if i.isdigit()]
25          variables["pitch"] = pitchNumber[0]
26
27      elif "playButton" in string:
28          print(variables)
29          samplingFreq, signal = wave.read("selectedTrack.wav")
30
31          signal2 = applyReverb(signal, samplingFreq, variables.get("reverb"))
32          signal3 = pitch(signal2, variables.get("pitch"))
33
34          buttonPlay(signal3, samplingFreq)
35
36      elif "stopButton" in string:
37          buttonStop()
```

Figure 6.18: *readInput function.*

For ordering and categorizing the inputs received from the Arduino the function $readInput(string)$ was created (line 13) as seen in Figure 6.18. This function will take the inputs and store them in a dictionary. This is done on certain conditions. The Arduino sends its inputs in a package consisting of a keyword and a number, meaning that the python script has to recognize the keyword and then map the value to a variable. Firstly the program checks what track number the user selected (line 15) and creates a new track using the function $createTrack(sliderSong)$ (line 17). This will create a temporary wav-file which the filters are going to process later. Once the track has been created, the values for reverb and pitch are stored in the dictionary mentioned earlier (line 21, 25, 29). If the user presses play, the program will first load the temporary wav-file and feed it to the filters. First, the reverb filter is applied with the "reverb" value from the dictionary (line 35) as input. Afterwards, the pitch filter is applied to the filtered reverb signal with the "pitch" value from the dictionary (line 36) as input. This will end up with a fully processed track which is then played using the function $buttonPlay(signal, samplingFrequency)$. If the user at any point presses stop the program will run the function $buttonStop()$ which will stop the playback (line 41).

```
43  while True:
44
45      line = data.readline()
46      string = line.decode("utf-8")
47      readInput(string)
```

Figure 6.19: *Execute code.*

Now that the function has been defined it is time to run it. As seen in Figure 6.19 the function is being called in a while loop which essentially runs forever. First, it reads the input from the serial monitor (line 45) and since that data received is in bytes, and the function needs a string, the program needs to decode it (line 46). Once done, it then feeds the string into the function and runs (line 47).

Chapter 7

Testing and Evaluation

This chapter will cover the testing and evaluation of the prototype, which was achieved by using two tests. The first test involved a Multi Stimulus test with Hidden Reference and Anchor (MUSHRA) [17], where the focus was to get feedback on the quality of the sound produced from the prototype. The second test involved a User Experience Questionnaire (UEQ) [16], where the focus was on testing the intuitiveness of the prototype, through user's experiences.

7.1 MUSHRA Listening Test

7.1.1 Purpose

The purpose of using the MUSHRA listening test was to test the technical aspect of the product. This was done in order to find out if the program could apply effects to a sound track without reducing the quality. According to the success criteria in Section 4.3, in order to fulfill this, the sound tracks should score a minimum of "good" in this test. To get valuable and reliable results within the test it is necessary to use external listeners, who can objectively evaluate the sound tracks. Objectivity is an important aspect, as the sounds are not to be judged by personal preference, but rather by the sound quality itself. The participants will listen to tracks that are either hidden references or anchors throughout the test. Hidden references are unaltered versions of the tracks and the anchors are modified versions of the same tracks, which will ultimately help compare the quality of the effects.

7.1.2 Apparatus

Since the testing was conducted online, the apparatus was represented by the participants' own computer and headset.

7.1.3 Participants

The test was conducted on 10 participants. Five participants were randomly selected, while the remaining five were selected based on their further participation in the UEQ testing. The participants were within the age range of 20–30 years old. They were all students of Medialogy at Aalborg University, on varying semesters.

7.1.4 Procedure

Each participant was contacted by e-mail. They were asked to listen to the sound tracks with a headset of their disposal and use any music software they had available on their device. They were also provided with a link to a survey, from SurveyMonkey [19], where they could rate the quality of the sound tracks.

The participants were tested individually and were provided with a consent form, a document with written instructions and a personal ZIP-file containing a total of 16 sound tracks, which were all 10 seconds long. Five of the tracks were separated into a folder called "Test 1" and the remaining 11 were in a folder called "Test 2". The participants were tasked to start with the folder called Test 1 and rate the quality of all the tracks, by using a slider that scaled from 0–100, using SurveyMonkey. 0 on the scale would be considered "poor" and 100 would be considered "excellent". They would then repeat the same procedure for the folder called "Test 2".

Test 1 was a pilot test, containing ambient sound tracks that were not used in the prototype. They were of both good and purposely bad quality, which was done to prepare the participants for critical hearing to make the results more reliable. It served an additional purpose of providing insight into how well the participants could pick up on audio of poor quality.

Test 2 consisted of eleven sound tracks. Three of them were the unaltered sound tracks that were used in the project (footsteps, swords and a torch). These were the hidden references. The eight remaining tracks were generated by applying effects to the original tracks, meaning they were the anchors. Each combination of effects was included, and can be seen below.

1. Low reverb
2. Low reverb and low pitch
3. Low pitch
4. High reverb and high pitch

5. Low reverb and high pitch
6. High pitch
7. High reverb
8. High reverb and low pitch

The effect was randomly selected for each track and was only used one time, meaning that low pitch was e.g. only tested on "footsteps".

The participants were asked to listen to the tracks in numerical order, however, that numerical order was random for each participant. This was done to increase the reliability, which will be explained further in Section 7.1.5.

The participants would be able to identify the sound tracks and the sliders by their respective names, both within the folders and the survey itself. The participants would, additionally, be able to go back and change the rating they had previously given to a track, should they change their mind.

The test results were gathered in Microsoft Excel after each participant had completed the test and later processed in R Studio [15]. The data were afterwards ordered in a final Excel sheet and ported into a CSV file.

7.1.5 MUSHRA Results and Discussion

In this section, the results from the MUSHRA test will be explored.

Results

To analyse the data, all the scores from the hidden references had to be compared to the anchors i.e. the unaltered tracks had to be compared to the tracks after the effects were added.

To do this, the mean rating for each track was calculated, which would then be compared with the hidden references. However, in order to figure out how accurate the data would be, the confidence interval needed to be calculated. This will show an interval of where the true mean would lie. To do this, the data were split into three categories, with a category for each sound track i.e. Footsteps, Swords and Torch.

To calculate the confidence interval, the standard deviation of the data is needed. This is calculated by finding the variance of the data, which is done by using the following formula:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N} \tag{7.1}$$

Here, σ^2 is the variance, x_i is every data point (e.g. track scores), μ is the mean of all the data and N is the population size (e.g. amount of participants). Once the variance has been attained the standard deviation can be calculated by taking the square root of it. Resulting in $\sigma = \sqrt{\sigma^2}$. For future reference, this will be referred to as the $s - value$.

To calculate the confidence interval, a percentage of accuracy has to be chosen. In this case it was picked to be 95% as it is a commonly used percentage. This means that 95% of the results would be the true mean. For this interval the corresponding Z-value is 1.960. For calculating the confidence interval, the following formula has to be used:

$$mean \pm Z\frac{s}{\sqrt{N}} \tag{7.2}$$

This will return a error range which represents the 95% confidence interval. For example, track one has an error margin of: 75 ± 11.38.

To visualise the data, three barplots were made with an error margin representing the confidence interval.

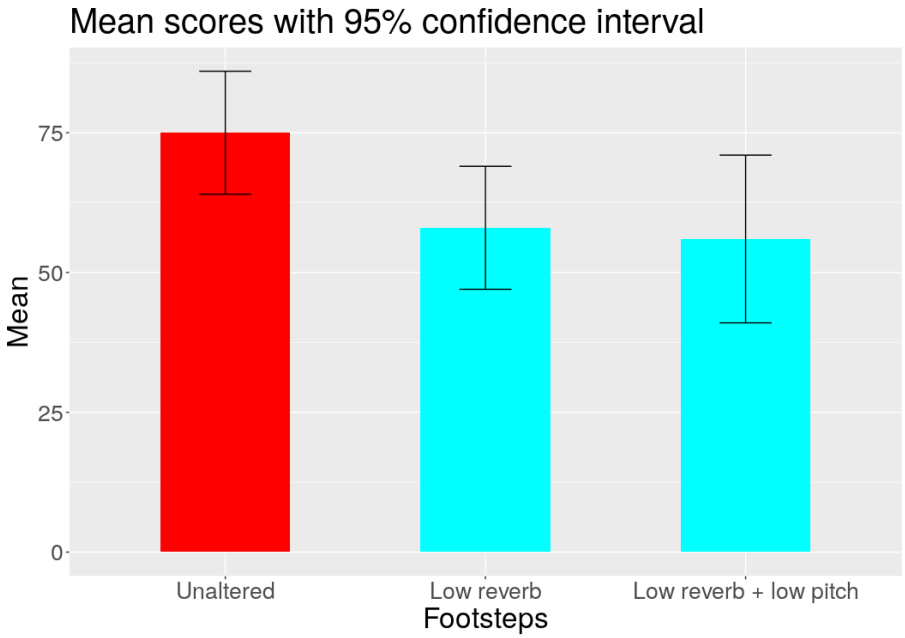


Figure 7.1: *Mean and confidence interval for each version of footsteps.*

Each bar in Figure 7.1 above represents a different sound track with specific modifications, where their height is represented as the mean. By looking at the bars, the unaltered version of footsteps scored higher than the other modified versions. This would indicate that the participants thought of it being the best track with a mean of 75, compared to the other two which scored 58 and 56. However, another factor to look at is the confidence interval which is represented by the error margin on top of the bars. Looking at the error range of each confidence interval, it can be seen that the *Unaltered* track's error margin has values distributed from 64 to 86 compared to the *Low reverb* which has values from 47 to 69 and *Low reverb + low pitch* which has values from 41 to 71. This means that even though their confidence intervals overlap, the unaltered track is still clearly the best version of footsteps with higher values within the error margin.

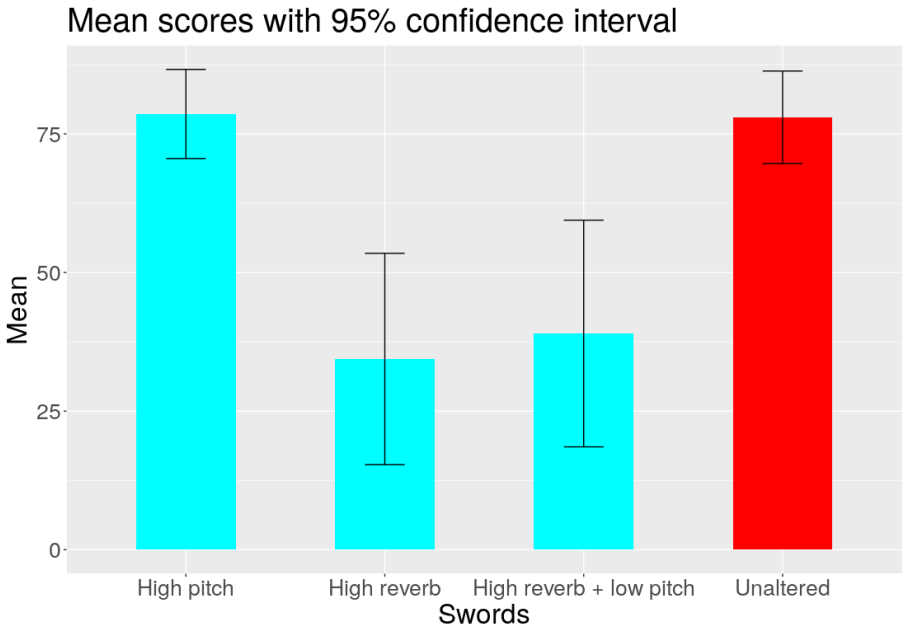


Figure 7.2: *Mean and confidence interval for each version of swords.*

In Figure 7.2, it can be seen that the unaltered and *High pitch* versions of the sword track scored relatively high. However, *High reverb* and *High reverb + low pitch* were substandard as they both scored a mean under 50. However, the error margin for the low scoring tracks is larger, which indicates a high amount of uncertainty of the real mean. Although, it would still be beneath the lowest value within the error margin of the unaltered (red) track. All in all, it can be derived that *High pitch* leaves an insignificant change in quality and *High reverb + low pitch* certainly reduces the quality of the track based on these results.

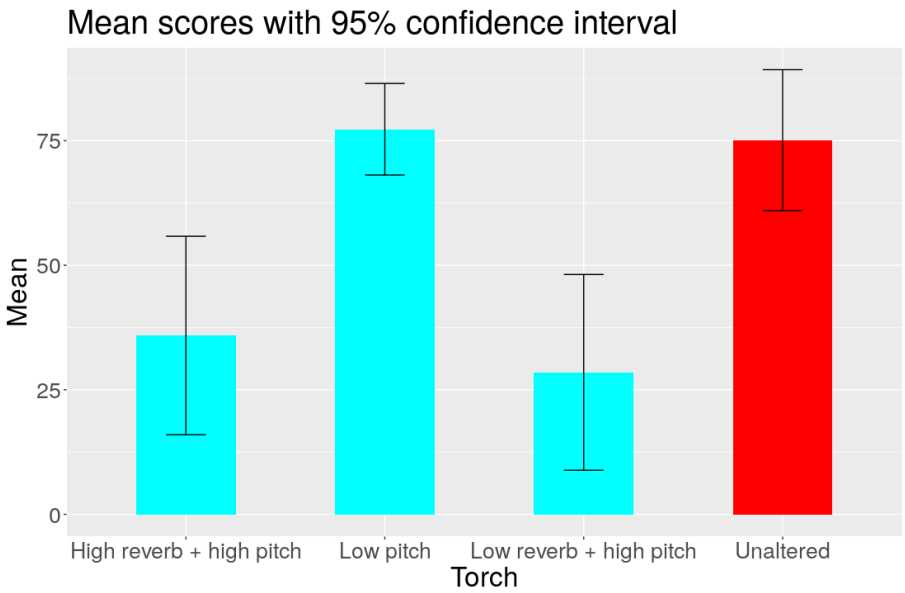


Figure 7.3: *Mean and confidence interval for each version of torch.*

In Figure 7.3 the results for the torch track can be seen. There are many parallels that can be drawn between it and the results from Figure 7.2. Once again, the effect that only alters the pitch in *Low pitch* has a similar score to the unaltered track, and they are both the highest rated tracks. The track *High reverb + high pitch* and *Low reverb + high pitch* both scored low with a large error margin.

This indicates the participants had overall varied opinions of all tracks that included reverberation. Furthermore, it might also indicate that the participants only prefer tracks with little to no effects.

Track	Mean	Low (CI 95%)	High (CI 95%)	
Unaltered	75.10	60.95	89.25	
Low pitch	77.30	68.11	86.49	Torch
High reverb + high pitch	35.90	15.98	55.82	
Low reverb + high pitch	28.50	8.86	48.14	
Unaltered	78.00	69.66	86.34	
High pitch	78.60	70.56	86.64	Swords
High reverb	34.40	15.34	53.46	
High reverb + low pitch	39.00	18.56	59.44	
Unaltered	75.00	63.62	86.00	
Low reverb	58.00	47.00	69.00	Footsteps
Low reverb + low pitch	56.00	41.00	71.00	

Figure 7.4: *Mean and 95 % confidence interval for each track.*

Looking at Figure 7.4 above, only track *torch unaltered*, *torch low pitch*, *swords unaltered*, *swords high pitch* and *footsteps unaltered* were considered successful, as they scored a mean between 60-80 and would be classified as "good", or above. However, in this case the minimum confidence interval was the standard used for comparison, which would mean that in all cases the true mean would be included. That makes the conclusion more reliable, yet less valid as some of the tracks with the potential to pass the criteria are considered failures. This means that the success criteria *"The tracks must score a minimum grade of "good" in the MUSHRA test"* was not fulfilled, since it was only five of the total 11 tracks that scored good.

To assess the normality of the data, the Shapiro test was used. From the test, a $p - value$ will be generated, that will determine a percentage of the chance of a false premise, meaning that the data would have to be rejected. If a result is below 5%, the data would not be considered significantly different from a normal distribution, making the data valid.

```
> shapiro.test(means)

        Shapiro-Wilk normality test

data:  means
W = 0.83973, p-value = 0.03132
```

Figure 7.5: *Shapiro Wilk test of mean and median values of the MUSHRA test from RStudio.*

Figure 7.5 shows the mean of all the tracks. The p-value of the mean in the Shapiro test is at 0.03132, which is approximately 3% and by the aforementioned definition of a normal distribution, means that the data had a low chance of error and pass the Shapiro test. In other words, the data is considered normally distributed.

7.2 User Experience Questionnaire

Besides the MUSHRA test, a UEQ was also conducted and will now be explained in detail.

7.2.1 Purpose

The purpose of the User Experience Questionnaire was to test the participant's emotion and attitude to the product, i.e. the user experience. That was done through a questionnaire that includes 26 items on six scales: *attractiveness, perspicuity, efficiency, dependability, stimulation,* and *novelty.* Each scale described a distinct quality of the product by evaluating the sum of scores in the items related to it [16].

Each item was in the form of a semantic differential, which meant that they were represented by two terms with an opposite meaning [16]. An example could be attractive and unattractive. In this UEQ test, it is represented on a 7-stage scale ranging from one to seven.

7.2.2 Apparatus

The testing itself involved the use of the physical prototype, which had to be on camera and streamed via the Internet for the participants to interact with. For this purpose, a phone was attached to a tripod to capture the entire device. It is important to note that attaching the phone to the tripod resulted in covering the microphone, making audio feedback incoherent and at times inaudible. To stream the camera feed, the Microsoft Teams app was used. The prototype itself was placed on the table and connected to a computer. This entire setup can be seen in Figure 7.6.

Figure 7.6: *Setup for the testing.*

7.2.3 Participants

The test was conducted on five participants, which were all specifically selected for their former experience as GM's. This would allow further insight into how useful the product would feel to a user who would actively use it.

7.2.4 Procedure

Before the testing, the participants were contacted through e-mail, which included a consent form that was to be signed and sent back before the testing, and a sched-

ule. The participants were asked to join an MS Teams call, together with two researchers, one of which was controlling and video-streaming the device while the other was giving instructions to the participants. Since the testing of the physical prototype was done over the Internet, the participants were told to interact with it by giving instructions to the researcher, who would then follow these instructions accordingly. The participants were also asked to think aloud for a better understanding of their perception of the system, as well as better navigation for the researcher in control of the device. The tasks that the participants were asked to perform were the following:

- Play footsteps sound with high reverberation filter applied
- Stop the sound
- Choose and play the sword sound with low pitch setting

These tasks were designed to test the interaction with every physical component of the device. After each task, the researchers would inform the participant about the completion of the task. This was done because the participants did not have audio feedback from the prototype during the testing, as reasoned above, in the Section 7.2.2. When all tasks were completed, the participants were asked to complete the UEQ.

7.2.5 Results and Discussion

In this section the results from the UEQ test will be discussed in detail.

Transforming The Data

In the questionnaire that was used, the answers for each item ranged from 1 to 7. These answers have been collected in a table which can be seen in Figure 7.7. However, the meaning of the number given is different depending on which end of the scale the positive and negative values are placed. There are items where 1 represents the best possible response and others where 1 represents the worst possible response. As such, to properly analyse the data, a transformation, which would uniformise the scores, was necessary.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
6	7	5	1	3	3	6	7	1	2	6	2	7	5	4	5	3	2	2	7	1	6	1	3	1	5
4	7	5	1	3	5	3	7	1	6	5	3	7	5	4	5	1	3	1	5	1	5	2	3	3	4
4	5	2	2	5	5	6	6	2	3	6	3	6	5	4	4	3	3	2	4	2	5	4	5	2	5
4	6	3	1	5	5	5	5	5	3	5	2	6	6	3	6	3	3	2	4	2	5	3	5	3	5
6	6	1	1	1	7	7	6	1	1	6	1	6	6	7	7	1	1	7	7	2	7	1	1	1	7

Figure 7.7: *A table of the collected data, with the results ranging from 1 to 7. Each column represents one item in the questionnaire (one question) and each row represents the responses from one participant.*

To achieve this, the data was transformed from a scale of 1 to a 7 to a scale of -3 to 3. Where -3 is the most negative response a user can give, and 3 is the most positive response a user can give. Because of this, the overall score for each scale could be calculated as a mean, later. The converted results can be seen in Figure 7.8.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
2	3	-1	3	1	-1	2	3	3	2	2	2	3	1	0	1	1	2	2	3	3	2	3	1	3	1
0	3	-1	3	1	1	-1	3	3	-2	1	1	3	1	0	1	3	1	3	1	3	1	2	1	1	0
0	1	2	2	-1	1	2	2	2	1	2	1	2	1	0	0	1	1	2	0	2	1	0	-1	2	1
0	2	1	3	-1	1	1	1	-1	1	1	2	2	2	-1	2	1	1	2	0	2	1	1	-1	1	1
2	2	3	3	3	3	3	2	3	3	2	3	2	2	3	3	3	3	-3	3	2	3	3	3	3	3

Figure 7.8: *A table of the collected data, after transformation. The results range from -3 to 3. Each column represents one item in the questionnaire (one question) and each row represents the responses from one participant.*

To better explain the transformation of the data, an example can be given. Firstly, when looking at participant one's response for item 2 (row 1, column 2) in Figure 7.7, we can see the answer was 7. For this particular question, the possible responses ranged from "not understandable" (which was at position 1) to "understandable" (which was at position 7), this means that the higher the number, the more positive the response, as such, an answer of 7 represents the most positive response possible. For this reason, in Figure 7.8, at the same position (row 1, column 2), the response was marked as a 3.

Secondly, looking at the response of the same participant for item 4 (row 1, column 4), the answer was a 1. If the same rules as above applied, the response would be seen as the most negative possible response. However, this particular item ranges from "easy to learn" (at position 1) to "difficult to learn" (at position 7), meaning that here, 1 represents the most positive possible response. That is why this transformation is needed, and, as can be seen in Figure 7.8 (at row 1, column 4), the response was changed from a 1 to a 3.

Results

The data gathered from the UEQ were evaluated using the analysis tool created by Dr Martin Schrepp [16].

After the data was transformed, it was evaluated based on the mean and the variance. Firstly, the mean was calculated for each item. This was done by taking the sum of all the responses for that item and dividing it by the number of responses. Secondly, using the means for each item belonging to a scale, we can calculate the mean of that scale.

After calculating the mean, variance 7.1, standard deviation and confidence interval 7.2, each item could be computed.

Looking at Figure 7.9 from the top left and horizontally to the right, the data is divided into categories. The "item" column represents each question. The next columns each give extra information about that particular item. As such, one can see the "Mean", "Variance" and standard deviation which were explained earlier. Next to them, the "No." column shows the number of responses for that item. Furthermore, the "Left" and "Right" columns express what each outer pole of that question represents. The last column, "Scale" shows which scale each item belongs to. The scales are also colour-coded in the far right of the table.

Item	Mean	Variance	Std. Dev.	No.	Left	Right	Scale	
1	0,8	1,2	1,1	5	annoying	enjoyable	Attractiveness	
2	2,2	0,7	0,8	5	not understandable	understandable	Perspicuity	
3	0,8	3,2	1,8	5	creative	dull	Novelty	
4	2,8	0,2	0,4	5	easy to learn	difficult to learn	Perspicuity	
5	0,6	2,8	1,7	5	valuable	inferior	Stimulation	
6	1,0	2,0	1,4	5	boring	exciting	Stimulation	
7	1,4	2,3	1,5	5	not interesting	interesting	Stimulation	
8	2,2	0,7	0,8	5	unpredictable	predictable	Dependability	
9	2,0	3,0	1,7	5	fast	slow	Efficiency	
10	1,0	3,5	1,9	5	inventive	conventional	Novelty	
11	1,6	0,3	0,5	5	obstructive	supportive	Dependability	
12	1,8	0,7	0,8	5	good	bad	Attractiveness	
13	2,4	0,3	0,5	5	complicated	easy	Perspicuity	
14	1,4	0,3	0,5	5	unlikable	pleasing	Attractiveness	
15	0,4	2,3	1,5	5	usual	leading edge	Novelty	
16	1,4	1,3	1,1	5	unpleasant	pleasant	Attractiveness	
17	1,8	1,2	1,1	5	secure	not secure	Dependability	
18	1,6	0,8	0,9	5	motivating	demotivating	Stimulation	
19	1,2	5,7	2,4	5	meets expectations	does not meet expectations	Dependability	
20	1,4	2,3	1,5	5	inefficient	efficient	Efficiency	
21	2,4	0,3	0,5	5	clear	confusing	Perspicuity	
22	1,6	0,8	0,9	5	impractical	practical	Efficiency	
23	1,8	1,7	1,3	5	organized	cluttered	Efficiency	
24	0,6	2,8	1,7	5	attractive	unattractive	Attractiveness	
25	2,0	1,0	1,0	5	friendly	unfriendly	Attractiveness	
26	1,2	1,2	1,1	5	conservative	innovative	Novelty	

Figure 7.9: *An overview of how the data gathered from each question asked in the user experience questionnaire is evaluated.*

Taking a look at the mean in Figure 7.9 it can be seen that every question was rated with a score from 0.4 to 2.8. This indicates that the participants were either indifferent or found the aspects mentioned about the product positive.

Even though the mean gives us some insight into the data, it does not describe the range of the given values. This can give a misleading result, in cases where the participants are split almost evenly into both sides of the spectrum. To figure

out if this is the case, the variance is used. In Figure 7.9 it can be seen that the questions have a variance in the range from 0.2 to 5.7. This indicates that there is an agreement in certain aspects of what the product achieves and disagreement in other aspects. An example of disagreement in what the product achieves can be seen looking at "item" 10. Here the variance is 3.5, which means that there is no agreement whether the product meets or does not meet expectations. To get a more specific number in term of how much the answers deviated, the standard deviation is used. Continuing with the last mentioned example being "item" 10 it can be seen in Figure 7.9 that the answers in general varied with 1.9 points, indicated by the standard deviation. This means that the participants were not in an agreement, which can also be seen in Figure 7.7 where the results from "item" 10 were 1, 2, 3, 3, 6.

Since the main focus of the UEQ in this project is to test the intuitiveness of the prototype, it is important to look at the questions denoted under the scale called perspicuity. The perspicuity scale covers question 2, 4, 13, and 21 as can be seen in Figure 7.9. Combining the data from these questions, the overall mean of perspicuity is 2.45 This means that the participants, in general, found the system intuitive.

However, in order to check how accurate this value is, the confidence interval can be used. The error margin was calculated using a 95% confidence interval. As can be seen in Figure 7.10 the perspicuity scale has a confidence interval of 0.475. This means that the true mean is more likely to be within a range of 1.975 to 2.925.

Confidence intervals (p=0.05) per scale						
Scale	Mean	Std. Dev.	N	Confidence	Confidence interval	
Attractiveness	1,333	0,858	5	0,752	0,581	2,085
Perspicuity	2,450	0,542	5	0,475	1,975	2,925
Efficiency	1,700	1,204	5	1,055	0,645	2,755
Dependability	1,700	0,597	5	0,523	1,177	2,223
Stimulation	1,150	1,055	5	0,925	0,225	2,075
Novelty	0,850	1,365	5	1,196	-0,346	2,046

Figure 7.10: *An overview of the data divided in their respective scales.*

To further analyse the data, the data is compared to a benchmark test. The benchmark test is a part of the data analysis tool, and its results are based on answers from 20190 people from 452 studies. The purpose of the test is to compare the gathered data with other data to have an idea of the quality of the created system.

In Figure 7.11 it can be seen that the UEQ questions are divided into the six dif-

ferent scales mentioned before. These scales can then be rated either bad, below average, above average, good, or excellent. The explanation of the ratings can be seen below in Table 7.1.

Defining the ratings	
Rating	Interpretation
Excellent	top 10%
Good	top 10% to 25%
Above average	top 25% to 50%
Below average	top 50% to 75%
Bad	top 75% to 100%

Table 7.1: *Explanation of the ratings given based on the benchmark test.*

The plot of all the scales, with their means and confidence intervals is illustrated in Figure 7.11. As can be seen, even when taking the lowest value in the confidence interval, the perspicuity of the system can still be considered good. This is contrast with other scales, such as novelty or efficiency which are hard to pinpoint in a certain category.

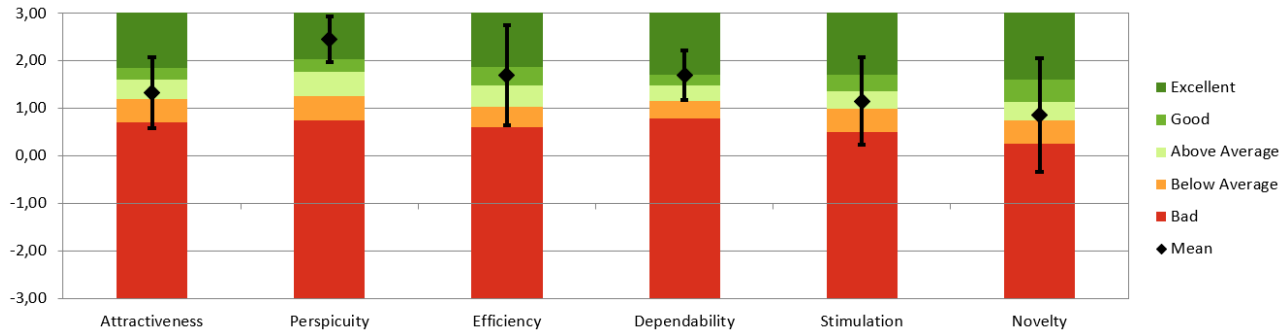


Figure 7.11: *A comparison of the collected means and their respective confidence intervals to the benchmark test, in the categories being Attractiveness, Perspicuity, Efficiency, Dependability, Stimulation, and Novelty to evaluate the quality of the created product.*

Inconsistencies

There is always the possibility that errors might occur with certain answers. To detect such mistakes, it is possible to look for inconsistencies in the data. As previously mentioned in Section 7.2, each item in the UEQ belongs to one of 6 scales, with each of the scales measuring a different aspect of the user experience. This means that if a participant has big differences between their responses to items belonging to the same scale, there is an inconsistency in the data. To check for these

inconsistencies, it is possible to look at the best and worst evaluation that each participant has provided, for each scale, and to check how much of a difference there is between them. If this difference is more than 3, there is an inconsistency in that participant's answer for that particular scale. If multiple inconsistencies can be found in the answers of one participant, then their results might be considered unreliable.

In the case of the data collected for this project, there is one inconsistency that appears in the answers of participant five. This inconsistency appears on the scale of "Dependability", which encompasses items 8, 11, 17 and 19. As can be seen in the Figure 7.8, participant five had a response of -3 to item 19 and a response of 3 to item 17, meaning the difference between the best and worst response for "Dependability" is a 6 for this participant. As per the heuristic explained above, this is an inconsistency in the data. However, since it is the only one that appears for this participant and the only one that appears for this scale, the answer of participant five is not fully unreliable. Instead, it is likely that a mistake in rating the item, or a misunderstanding of the item occurred, which lead to this error. Still, it is important to note that this inconsistency still affects the data. Specifically, the overall results for "Dependability" become slightly inaccurate, as well as the variance for item 19. The inconsistency makes the variance for item 19 become 5.7, which is, as mentioned in the results, the largest variances out of all the items. For this reason, this variance is problematic to take into consideration.

Chapter 8

Discussion

This chapter will discuss the upsides and downsides of the evaluation of the prototype, the prototype itself and possibilities for future iterations.

8.1 MUSHRA Consideration

The MUSHRA test results from Section 7.1.5 would indicate that there might be a correlation between the amount of effects and the quality rating. Thus, it is possible to make a hypothesis for a future iteration;

The MUSHRA quality rating of tracks decreases as the amount of effects applied increases.

8.2 Reliability and Validity

Multiple factors affected the reliability and validity of the test results in both positive and negative ways. As a start, the participants were from the same study as the researchers, which could introduce bias and negatively affect the validity. This could happen, since the participants may find it difficult to provide negative feedback and instead be supportive in order to please their peers and possibly get them to participate for their own project.

Another factor is the number of participants that could affect the validity. This applies for both the UEQ test, which was conducted on five participants, and the MUSHRA test, which was conducted on ten participants. While the consistency might be higher with a low amount of tests, the low amount of data might provide an inaccurate depiction of how a larger group of users would feel towards the prototype. This will overall negatively affect reliability.

The reliability is highly affected by the set-up for both tests. This is since there

was little control over the apparatuses that the participants would use during the testing. Examples of this could be the quality of the headphones that they used during the MUSHRA listening test and the stability and quality of their network, which they needed for the video meeting during the UEQ testing. All of these factors lead to a higher possibility of the participants having very different experiences.

Additionally, for the UEQ testing, it would have been ideal for the participants to physically interact with the prototype. This might have had an effect on the results, as the participants did not have tactile or audible feedback as they originally would have had in a physical testing set-up.

Furthermore, a factor which affects the reliability of the MUSHRA listening test is whether the participants managed to stay objective within their rating, rather than subjective. The participants were not found based on their listening skills, which means that the test was not conducted on listening experts, which the MUSHRA listening test recommends. As discussed in Section 7.1.5 and seen in Figure 7.4 the responses from the participants are very different for some sound tracks. This may be a result of the participants giving the sound track a poor rating, based on personal preference, which does not say a lot about the actual quality of the sound track. This makes it difficult to gather a real understanding of the results. Once again, having a higher number of participants may narrow this diversity in ratings.

8.3 Prototype And Future Iterations

The prototype in its current state could unfortunately not reach the intended form with all the features. As explained in Section 6.1, the use of an Arduino microcontroller prevented making the device a standalone system that could play sounds. This could be solved by introducing additional components to the system, such as a dedicated MP3 shield component that could handle the computational part of the system, and even the storage space that proved to be a problem. Alternatively, a different microcontroller or a microcomputer which would have more storage space and processing capabilities, could be used instead. The prototype currently lacks some intended key features and components such as a speaker, an on/off switch or a volume knob that would be necessary if it was a completely standalone system. Some components, such as the slider, could be replaced for the future iterations, to allow more sound options to be integrated into the device, as it would be impractical to put a slider on a very specific location to select the desired track. This could be achieved by either replacing it with a rotary or scrollable knob or simply with buttons to navigate through the sounds.

Another improvement could be made by adding more sound effects and even music to be a playable option. This addition might result in a desire for additional physical components on the device, therefore, a compromise would have to be made between the number of functionalities and intuitiveness/simplicity of the device.

Furthermore, an important aspect that should be worked on for the future could be the ability to play multiple sounds at the same time, as the game might require such a feature at times. This feature was originally meant to be included, however, the available components prevented this to be implemented as well.

Additionally, a possible improvement for a future iteration could be the possibility of applying filters to the sounds dynamically, as the current prototype only allows the filters to be applied before a sound is played. This feature is again tied to the capability of the microcontroller since it would be much more demanding in terms of processing power.

Finally, the physical appearance of the device could be improved by creating a wooden box made specifically to house these components. To finalise the prototype, it would also be needed to solder the cables to the components to prevent randomly losing contact with them, as it currently possesses an issue.

To summarise, the current state of the prototype does not meet the intended requirements, since it is limited by its components. However, even in this state the prototype was well received by the participants during the testing and yielded decent scores.

8.3.1 Wider Context

This project and its prototype used Dungeons and Dragons as an example for choosing the sounds and present the storyboard. However, the initial idea, as touched upon in Section 2, was to create a device that could support multiple board games. Even though it is mostly aimed at TRPGs, which could benefit the most from the addition of sounds for the purpose of immersion, having a device that produces game-specific sounds could enrich any tabletop game immersion.

Chapter 9

Conclusion

The focus of this project was to create a prototype that would be used in tabletop role-playing games to enhance the overall experience by improving immersion. This was to be achieved by creating a standalone physical prototype, which would utilise audio processing techniques, that would provide an easy to use interface for sound effects in these sort of games.

Background research was conducted, which lead to the conclusion that music and sound effects are used in other forms of media to evoke emotions, set the mood, describe locations, and that it could be used for TRPGs for the same purpose.

As a result of this, the concept design of the solution was developed: a physical music box that could store commonly used sound tracks, and also have the ability to allow for the modification of these sounds through certain audio effects, to provide more versatility. This solution was supposed to work on its own, with no need for extra devices.

With this solution in mind, a proof of concept was implemented. Because of the limitations of the hardware used, the prototype does not work on its own, but rather relies on a laptop or computer that runs python code. The physical interface consisted of an Arduino microcontroller replica and other components connected to it. The Arduino sends input to a code written in the Python programming language which takes care of the audio processing part. The prototype contains three sound tracks, which can be modified by applying reverberation effect or changing the pitch and speed. It has the capability of playing one sound effect at a time, and the effects are not applied in real-time.

The solution was tested using a UEQ test, as well as a MUSHRA test. The feedback was mostly positive, however, one of the tracks was rated poorly in the MUSHRA

(namely the sword sound effect).

The solution doesn't satisfy all the success criteria. Firstly, in this stage of development, the solution relies on an external device to work, as such, it cannot be considered standalone. Secondly, not all of the tracks were rated as "good" or above in the MUSHRA test.

To conclude, the current iteration of the solution does not meet all the requirements, but it got a positive response from users. The prototype could be iterated on further in order to fulfill the initial concept of the solution. Furthermore, in this project, the fact that this solution enhances immersion in a TRPG game has not been tested, so there are no definite conclusion about this matter.

Bibliography

[1] Arduino. *Arduino - Home*. en. URL: https://www.arduino.cc/ (visited on 05/06/2020).

[2] Arduino. *Arduino - LiquidCrystal*. en. URL: https://www.arduino.cc/en/Reference/LiquidCrystal (visited on 05/06/2020).

[3] *Audio processing course in Medialogy*. URL: https://www.moodle.aau.dk/course/view.php?id=32682 (visited on 03/10/2020).

[4] Sarah Lynne Bowman. "Immersion and Shared Imagination in Role-Playing Games". In: *Zagal, José P. and Deterding, S. (eds.), Role-Playing Game Studies: Transmedia Foundations. New York: Routledge, 379-394* (2018). URL: https://www.researchgate.net/profile/Sarah_Bowman14/publication/331758162_Immersion_and_Shared_Imagination_in_Role-Playing_Games/links/5d0275eb92851c874c643838/Immersion-and-Shared-Imagination-in-Role-Playing-Games.pdf.

[5] *Build software better, together*. en. Library Catalog: github.com. URL: https://github.com (visited on 05/06/2020).

[6] Bill Buxton. *Sketching user experiences: getting the design right and the right design*. Morgan kaufmann, 2010.

[7] Gordon Calleja. *In-Game: From Immersion to Incorporation*. en. Google-Books-ID: KIQKV_MXvBAC. MIT Press, May 2011. ISBN: 978-0-262-29454-6.

[8] Mads G Christensen. *Introduction to Audio Processing*. Springer, 2019.

[9] Jennifer Grouling Cover. *The Creation of Narrative in Tabletop Role-Playing Games*. en. Google-Books-ID: xl7P7GwME3gC. McFarland, Jan. 2014. ISBN: 978-0-7864-5617-8.

[10] Thooman GmbH. *Abletine Live Workstation*. en. Hans-Thomann-Straße, Burgebrach, Germany. URL: www.thomann.de/dk/ableton_live_10_suite.htm (visited on 02/26/2020).

[11] Thooman GmBH. *Novation Launchpad image*. en. Hans-Thomann-Straße, Burgebrach, Germany. URL: https://www.thomann.de/dk/novation_launchpad_pro.htm (visited on 02/26/2020).

[12] *Home page of tabletop audio*. URL: https://tabletopaudio.com/ (visited on 02/13/2020).

[13] *Monopoly playlist on Melodice*. URL: https://melodice.org/playlist/monopoly-1933/ (visited on 02/13/2020).

[14] Peter Peerdeman. "Sound and music in games". In: *And Haugehåtveit, O* (2006).

[15] RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, Inc. Boston, MA, 2015. URL: http://www.rstudio.com/.

[16] Martin Schrepp. "User experience questionnaire handbook". In: *All you need to know to apply the UEQ successfully in your project* (2015).

[17] B Series. "Method for the subjective assessment of intermediate quality level of audio systems". In: *International Telecommunication Union Radiocommunication Assembly* (2014).

[18] Jamie Sexton. *Music, sound and multimedia*. Edinburgh University Press, 2007.

[19] SurveyMonkey. *SurveyMonkey Inc.* da. Library Catalog: da.surveymonkey.com. San Mateo, California, USA. URL: www.surveymonkey.com (visited on 05/13/2020).

[20] *Uncharted 4 image:* URL: https://lolbua.no/video/lets-play-uncharted-4-uten-storre-spoilers/ (visited on 02/13/2020).

[21] *Welcome to Python.org*. en. Library Catalog: www.python.org. URL: https://www.python.org/ (visited on 05/06/2020).