Rostislav Goncharov

# DEVELOPING TOOLS FOR AUDIO VISUALIZATION AND DATA SONIFICATION IN UNREAL ENGINE 5

Bachelor's thesis

Bachelor of Culture and Arts

Game Design

2023

South-Eastern Finland
University of Applied Sciences

XAMK

South-Eastern Finland
University of Applied Sciences

| | |
|---|---|
| Degree title | Bachelor of Culture and Arts |
| Author(s) | Rostislav Goncharov |
| Thesis title | Developing Tools for Audio Visualization and Data Sonification in Unreal Engine 5 |
| Commissioned by | Dark Amber Softworks / Eleven Productions |
| Year | 2023 |
| Pages | 48 pages |
| Supervisor(s) | Marko Siitonen |

## ABSTRACT

This production-based thesis was aimed at exploring the possibilities of implementing audio visualization and/or data sonification in video games using Unreal Engine 5.

In the first part of this thesis, an overview of previous research in the field was provided. Functions of game audio were identified along with purposes of audio visualization and data sonification in video games. Practical examples were provided of how the techniques have been used in several popular video games of the past decade. Additionally, basics of audio analysis and synthesis were discussed in that part.

In the second part of the thesis, a practical project in Unreal Engine 5 was documented. The project was commissioned by Dark Amber Softworks and produced over the course of the work. The techniques discussed in the theoretical part were implemented in five prototype scenes and intended for use as development tools in current and future Dark Amber Softworks projects. Possible use cases for each prototype were discussed and further development ideas were suggested. Feedback from focus group testing within the company was also discussed in that part.

The research indicated that audio visualization and data sonification could be an important part of the gameplay and showcased how the discussed techniques could be implemented in an Unreal Engine 5 project.

**Keywords**: Unreal Engine 5, audio visualization, data sonification, sound design, game audio

**CONTENTS**

## ABBREVIATIONS

| | |
|---|---|
| FFT | Fast Fourier Transformation |
| DAW | Digital Audio Workstation |
| VST | Virtual Studio Technology |
| LFO | Low-Frequency Oscillator |
| GPU | Graphics Processing Unit |
| UI | User Interface |
| AR | Augmented Reality |
| VR | Virtual Reality |

# 1 INTRODUCTION

Over the past few decades, the video game industry has shown explosive growth, while video games themselves have become one of the most popular multimedia formats in existence. Video games are one of the main driving forces behind the further development of certain technologies (such as GPUs for example), while game engines and game development techniques often find use in non-gaming fields such as cinematography, architecture or medical training.

From a technical standpoint, however, video games are a manifestation of the same process as any other software product: data manipulation. This thesis focuses on two closely connected ways of audio-related data manipulation: audio visualization and data sonification. The purpose of this work is to showcase the way those processes have been used in video games, explore the most common techniques for audio visualization and data sonification in modern game engines, and create a project in Unreal Engine 5 which would provide a number of reusable templates for implementing the aforementioned techniques in video games.

The first part of this thesis focuses on existing material (mainly articles, books and conference talks) which provides a theoretical basis for the project. It describes the scientific foundations for audio visualization and data sonification and explores various implementations of the two processes in popular video games.

The second part of this work describes some of the audio visualization and data sonification techniques available in Unreal Engine and Unity. It also documents a practical implementation of said techniques in an Unreal Engine 5 project which allows other developers to use them in their own games.

The primary goal of this thesis is to assess the importance of audio visualization and data sonification in video games, as well as to create a set of tools for implementing those processes in video games. Parts of the resulting project will be implemented in future Dark Amber Softworks products.

Dark Amber Softworks is an independent software development and media production studio based in Kotka, Finland. The majority of the company’s work to date has been performed in Unity engine and has a strong audiovisual focus. This thesis project has allowed the company to branch out into Unreal Engine 5 while simultaneously developing new audio-related ideas for further implementation.

# 2 RESEARCH OUTLINE

## 2.1 Research aims and questions

The aim of this research is to produce an Unreal Engine 5 project which would contain a series of prototypes with a focus on audio visualization and data sonification. The project will later be used in future games by Dark Amber Softworks and might also be made available to other game developers. Thus, the primary question for this research can be formulated as follows: how can various audio visualization and data sonification techniques be implemented in Unreal Engine 5?

To reach the primary goal of the research, the following secondary questions must be considered as well:

- What audio visualization and data sonification techniques exist?
- Are any of the techniques supported in popular game development engines such as Unity and Unreal Engine?
- Are there any downsides (either technical or design-related) to using any of the techniques in a game?

## 2.2 Research methods

In *The Video Game Theory Reader 2*, Perron & Wolf (2009, 2) state that “video game systems and games themselves are the starting points of theories” while also emphasizing that video game studies tend to evolve slower than the video game industry. Additionally, they point out the lack of a rigid academic approach to video game research due to the volatile nature of the field (Perron & Wolf

2009, 5-6). Thus, researchers have a considerable amount of freedom in methodology choices when studying video games and related concepts.

Seeing as the purpose of this thesis is to build a set of tools for further use within Dark Amber Softworks and possibly to make it available to other developers as well, it is reasonable to use gameplay analysis and focus group testing as the two key research methods for this work.

Gameplay analysis focuses on specific audio visualization and/or data sonification techniques in a selection of games. The way those techniques affect the gameplay in each case is explained, and possible risks arising from using the technique are assessed. This research method constitutes a direct way to analyze video games from the player's perspective, which is important for the outcomes of the practical part of this thesis. Additionally, gameplay analysis highlights the author's subjective perception of the discussed phenomena and provides inspiration for the practical project.

Focus group testing was performed by offering the rest of the Dark Amber Softworks team to test the tools created as part of this thesis. Team members were asked for feedback regarding their experiences and possible suggestions for further development. The feedback is discussed in the final part of the thesis. This method seems appropriate because of the subjective nature of the phenomena in question (audiovisual perception and workflow convenience). Collecting and analyzing the focus group's opinions is a way to get a deeper, more objective understanding of the impact the discussed phenomena might have on individuals, which is not possible when using quantitative research methods (Boncz 2015, 23).

Aside from the two methods mentioned above, this work analyzes multiple relevant articles, blogs and other publications along with the official documentation for Unity and Unreal Engine. All sources used in this thesis are included in the list of references.

## 2.3 Key concepts

This paper views audio visualization, data sonification and any related concepts within the context of game design. This means that player experience is taken into account in terms of both gameplay and accessibility. Any audio visualization or data sonification techniques implemented in the practical project are supposed to enhance player experience from an aesthetical and/or functional standpoint. Figure 1 represents the key concepts for this research.

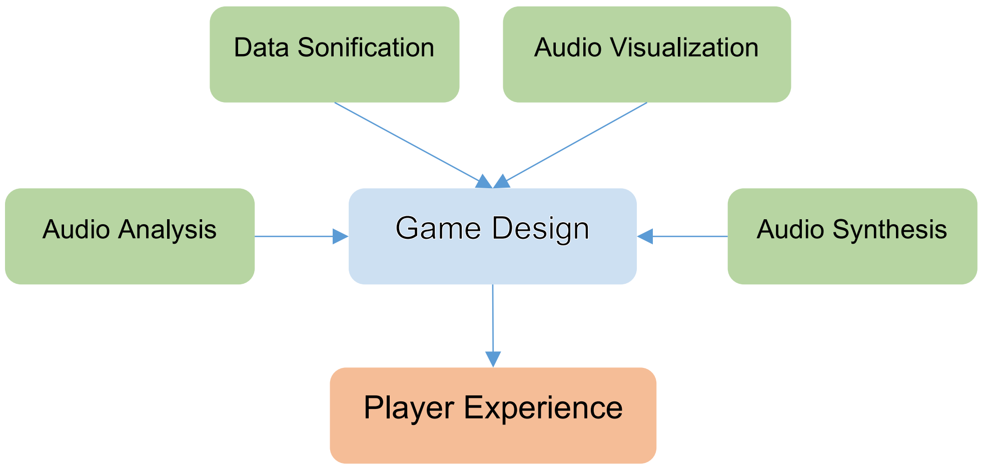


Figure 1. Key research concepts

While audio visualization and data sonification in the context of video games is still a relatively new area of studies, a vast amount of research has been done on those concepts in general, as well as on related concepts such as sound synthesis. Notable previous works in the field which provide basis for this thesis include The Sonification Handbook by Hermann et al. (2011) and On the Functional Aspects of Computer Game Audio by Jørgensen (2006). Additionally, the official documentation for Unity engine and Unreal Engine 5 provides the knowledge base for practical implementation of the concepts discussed in this thesis. The list of references contains all sources which have been used in this thesis.

# 3 THEORETICAL BASIS

## 3.1 Basic functions of audio in video games

Before starting to explore audio visualization and data sonification in video games, it is important to understand the basic functionality of game audio. Juul (2005, 1) suggests viewing games as a set of real rules and events which are set in a fictional world. This means that the role of sound in video games includes a wide range of functions which support this “real / fictional” duality.

According to Jørgensen (2006), five different functions can be defined for in-game audio: those include action-oriented, atmospheric, orienting, control-related and identifying functions. A brief summary of each function type is provided below.

Action-oriented functions imply using auditory icons and earcons (both of which are explained in more detail in section 4.3.2 of this thesis). In other words, any audio that provides auditory feedback to in-game events and/or urges the player to perform certain actions has an action-oriented function. Examples include sounds of weapons hitting or missing enemies, enemy attack alerts, adaptive soundtracks, and any other auditory icons or earcons which have a connection with in-game events. (Jørgensen 2006, 2.)

Atmospheric functions imply using audio for emotional engagement, immersion, and emphasis. While this can be seen as a mostly aesthetical element, atmospheric audio can also influence the player’s behaviour to a certain extent. For example, a stereotypical horror game soundtrack is supposed to convey the feelings of fear and anxiety to the player, which may influence their emotional state and thus affect the way they play the game (the player may start exploring the game world with extra caution, avoid enemy encounters, and perform other similar actions). Additionally, atmospheric functions are tightly connected to other function types: for instance, an adaptive soundtrack which goes up in intensity when enemies are nearby can both inform the player of immediate danger and set the mood for combat. (Jørgensen 2006, 3.)

Orienting functions have a somewhat similar role to that of action-oriented ones: both function types convey information about the actions and events in the game world to the player. The distinction is that orienting functions are focused on informing the player about the relative location of in-game characters, items and/or events. This function type implies that game audio can be used to extend the player's visual perception by presenting them with data regarding events which are not immediately visible. An obvious example of such a function would be hearing distant gunshots or enemy shouts: while the enemies themselves are not yet within the player's line of sight, the loudness and position of their sounds in the stereo field informs the player about the approximate distance and direction towards the enemies. (Jørgensen 2006, 3-4.)

Control-related functions are in many ways similar to orienting ones, but they focus specifically on the player's ability to control certain parts of the game world. For instance, in a real-time strategy game the player may receive an alert that their base is under attack while they are at a completely different point on the map. Such a notification guides the player to a certain location within the game world and urges them to use their ability to influence the game state. (Jørgensen 2006, 4.)

Finally, identifying functions are once again bound to auditory icons and/or earcons, except that the sound is bound to various identifying characteristics of objects, characters or events within the game, rather than to specific player actions. Different audio responses from units with different ranks and purposes in a strategy game can be an example of this type of functions. (Jørgensen 2006, 4.)

Aside from the five function types, another important characteristic of game audio is whether it is diegetic or non-diegetic. The Greek word "diegesis" can be translated as "narration"; in a video game, diegesis would apply to the game world and everything it includes and/or implies. According to Berndt (Grimshaw 2011, 62), diegetic and non-diegetic audio differs in whether its source exists within the diegesis. A simplified way to define this distinction would be that diegetic audio comes directly from within the game world and can be perceived by in-game characters, while non-diegetic sounds originate from within the game and can only be heard by the player. This thesis does not aim to explore more complex, “fourth-wall-breaking” cases of audio implementation in games, so the simplified definition should suffice for its purposes. The project discussed in the practical part of this thesis features both diegetic and non-diegetic audio.

A good example of game audio performing multiple functions is the way sound is implemented in Hellblade: Senua’s Sacrifice (2017). One of the core elements of the game’s design is a number of voices which the protagonist hears inside her head almost constantly. The game has no visual UI; instead, all the relevant information is delivered to the player by the voices, which behave differently depending on the situation: for example, they may get agitated when the protagonist’s health drops, provide hints regarding the objectives, or reveal pieces of in-game lore. Additionally, each voice seems to have their own personality. A point can be made that the voices perform all five of the aforementioned functions in a fully diegetic manner.

It is, however, worth considering accessibility when designing audio for a game. While Hellblade: Senua’s Sacrifice (2017) is a great example of game audio having a critical role in the gameplay, its accessibility is quite low: it is possible to complete the game based on visuals alone (and subtitles can be turned on for the voices), but a player who is unable to hear the audio would still miss a substantial part of the game experience. Thus, it is important to identify the desired balance between general accessibility and the desired prominence of in-game audio for any game project.

While the practical project does not have a specific goal to highlight any particular function(s) described in this chapter, it is safe to assume that at least action-oriented, atmospheric and/or orienting functionality can be easily identified in its parts.

## 3.2 Basics of sound analysis and synthesis

To have a better understanding of the core principles behind the practical project described later in this thesis, it is beneficial to possess basic knowledge of acoustics. This chapter takes a closer look at audio waveforms, basics of sound synthesis, and the Fast Fourier Transformation (FFT) algorithm for audio spectrum analysis.

Fundamentally, any sound we hear is produced by a vibrating body: for example, a guitar string or a speaker cone. This vibration creates a chain reaction of vibrating air molecules which eventually reaches our ears and can be represented as a waveform. (Comeau 2018.)

A waveform can be defined as a graph which represents the way the vibration's amplitude changes over time. The shape of this graph depends on the harmonic content of the sound: in other words, it is influenced both by the fundamental pitch of the sound and its overtones (if any). The fundamental pitch is the basic pure tone which the sound possesses; if the sound has no overtones, the waveform looks like a perfect sine wave. However, most sounds we hear are much more complex than that: aside from the fundamental frequency, they contain a number of harmonic and/or non-harmonic overtones. A musically meaningful sound has a clear fundamental tone which is accompanied by a series of higher-frequency overtones, which usually have lower amplitude. Harmonic overtones are directly related to the fundamental frequency as they are exact multiples of it. Non-harmonic overtones, on the other hand, have no such relation to the fundamental; a sound containing much non-harmonic content can be perceived as noisy and/or dissonant. (White 1994.)

This relationship between a sound's fundamental tone and any overtones it might have is the basis of various sound synthesis techniques. While a thorough exploration of sound synthesis is beyond the scope of this work, it is beneficial to have a basic understanding of one of its oldest forms: additive synthesis.

Additive synthesis is a technique based on summing multiple waveforms to achieve a different one. While technically any kind of waveform can be used in additive synthesis, sine waves are used most commonly, as their lack of harmonics yields the most predictable and controllable results. By summing a number of sine waves, each with its own frequency, amplitude and phase, much more complex waveforms can be achieved. This summation principle is the reason why the Fast Fourier Transformation algorithm is a popular choice for sound analysis, as it essentially deconstructs the sound into a series of sine waves. (Mantione 2017.)

The Fast Fourier Transformation (FFT) algorithm is an audio measurement method which samples the analyzed audio over a period of time, splits it into separate spectral components, and provides frequency information about the signal as a result. Each spectral component is a sinusoidal oscillation at a distinct frequency with its own amplitude and phase characteristics. (Fast Fourier Transformation FFT – Basics n.d.)

The output of the FFT algorithm can be represented by a spectrogram which displays the amplitude or “loudness” of various frequencies present in the signal over time (What is a Spectrogram? n.d.). Figure 2 displays a spectrogram of a harmonically rich noise sample processed through a band-pass filter, which has been obtained from Renoise DAW. A band-pass filter reduces the signal’s harmonic content both above and below a specified frequency; in this case, the spectrogram shows that the frequencies approximately between 1.2kHz and 2kHz are the most prominent in the signal, while ranges below 200Hz and above 10kHz have little to no content at all.

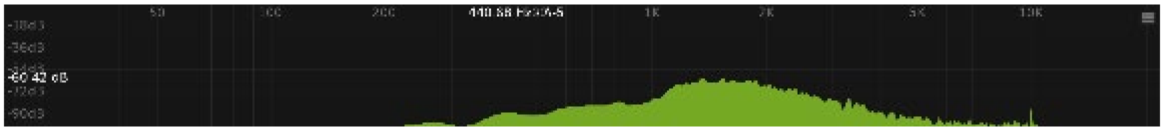


Figure 2. Screenshot of a spectrogram in Renoise DAW

## 3.3 Audio visualization and data sonification in video games

This section focuses on the definitions and possible use cases of audio visualization and data sonification in video games, as well as several important accompanying concepts.

### 3.3.1 Audio visualization

According to Yingfang Zhang et al. (2018, 2), audio visualization can be defined as “an objective interpretation and judgement for music representation, and an approach of representation used to understand, analyze and compare representation capacity and internal structure of music”. The definition fits the purposes of this thesis, but it should be noted that the principles and techniques utilized in the practical project are not necessarily limited to music and are applicable to any type of in-game audio.

Speaking of music visualization based on mathematical properties, Li & Li (2020) define three distinct levels of music information: underlying (which includes the sound’s waveform and frequency spectrum as well as other mathematical and physical properties), mid-level (which concerns musical properties such as melody, rhythm or harmony), and high-level (which describes the overall structure and perception of the music). This thesis assumes that game audio can be viewed in a similar manner without being limited to music specifically: for example, a voiceover track can be analyzed and subsequently visualized on all three levels. The practical part of this work mostly uses elements of low-level audio information analysis to visualize in-game sounds.

### 3.3.2 Data sonification

In contrast to audio visualization which represents audio data in a visual way, data sonification is an act of representing non-audio data through sound. This type of data manipulation is widely spread in the modern world: sounds such as EKG machine blips and smartphone notification chimes are examples of data sonification. (Geere 2020.)

Hermann et al. (2011) suggest that there are at least 5 distinct techniques for data sonification: audification, auditory icons, earcons, parameter mapping sonification, and model-based sonification. While all those techniques can potentially be used in video games, the three that seem immediately usable are auditory icons, earcons, and parameter mapping sonification.

Auditory icons and earcons are two very similar techniques of assigning acoustic markers to certain events. The key distinction between those two approaches is that earcons bind an abstract sound to an event (for example, a sequence of tones which plays upon startup of a computer's operating system), while auditory icons are based on an existing relationship between the sound and the underlying data. (Hermann et al. 2011, 325 – 361).

Auditory icons are used in video games as a natural link to various objects, events and actions existing in the game environment. For example, gunshots, footsteps and enemy shouts are auditory icons which can be commonly found in first-person shooter games. By providing a naturally recognizable sonic representation of objects and events, auditory icons provide the player with an intuitive non-visual way of gathering information about the game world. (Ng & Nesbitt 2013.)

Earcons, on the other hand, are abstract tones which are used to create auditory messages. Their independence from context allows them to be used with any interaction happening in the game while often providing a more precise message compared to auditory icons. However, this lack of context also means that the player has no immediate intuitive association with the sound and needs to explicitly learn the connection while playing the game. UI sounds, warning signals and adaptive background music can all be considered earcons. (Ng & Nesbitt 2013.)

Parameter mapping sonification is an approach which associates data values with auditory parameters for visualization purposes. A simple example would be mapping the temperature of water in a tea kettle to the pitch of a continuous sound signal: the higher the temperature gets, the more the signal's pitch goes up, and vice versa. However, this approach allows for infinitely more complex data sonification systems due to enormous range of mapping decisions available. (Hermann et al. 2011, 363.)

Because of the vast number of possibilities regarding parameter mappings and interpretations, parameter mapping sonification can be challenging to implement. Deliberate choices need to be made when designing such systems due to the fact that mappings can be continuous (such as sound volume or duration) or discrete (such as choice of instrument). Some parameters can even fit both categories: for instance, when data parameters are continuously mapped to audio pitch, the latter is often quantized to a musical scale in order to achieve more aesthetically pleasing results. (Geere 2020.)

Despite its inherent complexity, parameter mapping sonification can be a powerful data sonification tool in video games due to its flexibility. It is particularly interesting within the scope of this work as a method for creating generative and adaptive soundtracks, which will be explored in the practical project.

### 3.3.3 Gameplay analysis: the Dark Souls series (2011, 2014, 2016)

An example of auditory icons and earcons working together is the Dark Souls series (2011, 2014, 2016). In each of those games, combat has multiple auditory icons such as weapon hits, footsteps or screams; those are diegetic sounds which naturally originate from relevant actions such as swinging a sword or running. At the same time, prominent earcons are present as well: for example, distinct sounds can be heard whenever a character gets stunned, a successful parry happens, the player is killed, or an enemy dies and grants the player some in-game currency. Those earcons do not seem to necessarily originate from within the game world, and they are more abstract in nature than the auditory icons. However, the connection between the earcon and the event it represents is established at the early stages of the game, and the player quickly becomes able to distinguish important combat events based on game audio alone (this consecutively mitigates the lack of instinctive connection between the earcon and the linked event, which is an inherent downside of using earcons). Moreover, those earcons have become so iconic that they are consistent throughout all of the Dark Souls series (2011, 2014, 2016), as well as some other games from the same developer such as Elden Ring (2022). This way, a player who has experienced one of those games will instantly recognize the earcons when starting another game from the series.

# 4 PRACTICAL IMPLEMENTATION

This chapter describes the implementation of various audio visualization and data sonification techniques in a series of prototypes which has been created for Dark Amber Softworks / Eleven Productions.

First, the chapter describes the tools and instruments (both hardware and software) which were used in the development process. Next, two Unity prototypes are briefly reviewed and explained. Finally, the chapter provides a detailed description of the development process for a project in Unreal Engine 5 which can provide game developers with a number of reusable tools and prototypes to be used in their own games.

## 4.1 List of tools and instruments

Game engines:

- Unity 2021.3.3f1
- Unreal Engine 5.1.1

DAWs:

- Reaper
- Renoise

Instruments:

- Dreadbox Erebus v3 analog synthesizer
- Neutral Labs MEG waveshaper Eurorack module
- Electric guitar
- Electric bass
- Ugritone KVLT II VST

## 4.2 Unity Engine Prototypes

The prototypes described in this section were done in Unity and might be ported to Unreal Engine later. Their inclusion in this thesis is warranted for three reasons. Firstly, they illustrate some of the audio visualization and data sonification concepts described in the previous chapters (such as the usage of FFT for audio visualization purposes). Secondly, most of the techniques used in the scenes can be implemented in a similar manner in Unreal Engine. Finally, the Unity prototypes served as inspiration for creating the Unreal Engine project.

### 4.2.1 Music visualizer using FFT in C#

This scene visualizes the audio coming from the audio source by utilizing the available audio spectrum-related functions in C#. The scene consists of an audio source, a main camera, and multiple stationary point lights. In addition, moving light sources are spawned throughout the scene's lifetime. Figure 3 shows the default layout of point lights in the scene.

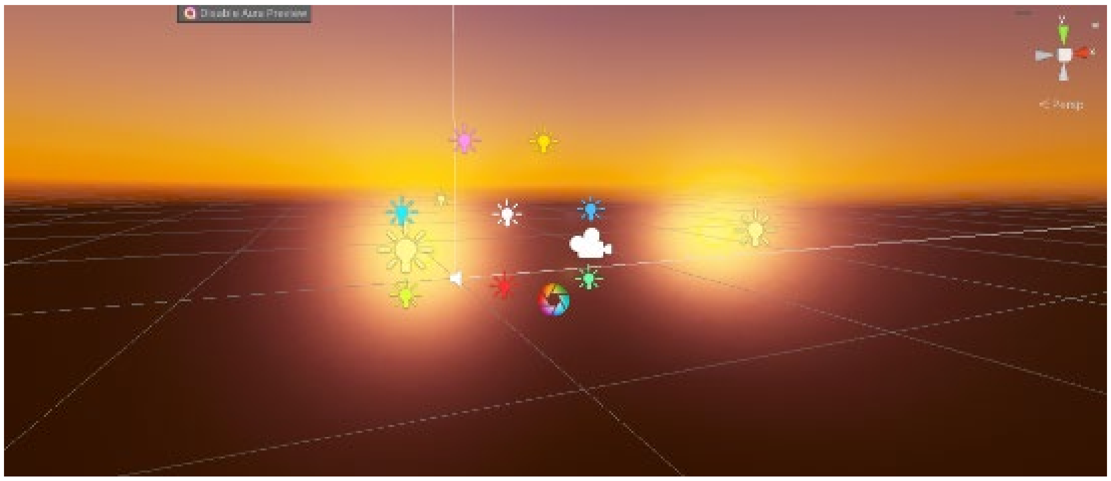


Figure 3. Default light positioning in the Audio Visualizer project

The audio is split into 8 frequency bands, then the volume of each band is normalized (i.e. its value is converted into a float between 0 and 1). The average volume of all normalized bands is calculated as well. The normalized band volumes and the normalized average volume are then used to control various visual parameters: the colour and/or intensity of separate light sources, the rate at which new moving lights are spawned, and the post-processing effects on the main camera. Figure 4 shows a screenshot of the ongoing audio visualization with a noticeable lens distortion effect.

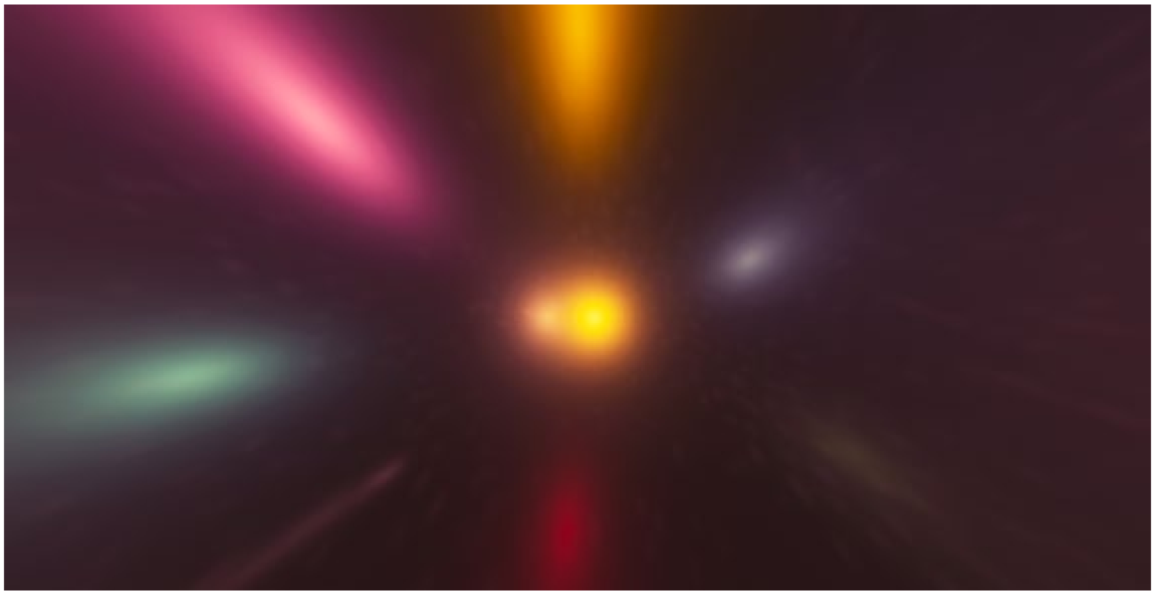


Figure 4. Lens distortion effect in the Audio Visualizer project

The key element that makes this scene work is the GetSpectrumData() function which is a member of the AudioSource and AudioListener classes in Unity. This function provides a block of the currently playing audio source’s spectrum data and fills a samples array with its values. It utilizes the Fast Fourier Transformation algorithm and allows the user to select the FFT window for optimized results. (Unity Technologies 2023.)

Splitting the samples array into frequency bands and normalizing each band is done manually. At the time of writing, Unity does not seem to provide dedicated functions for doing that.

The techniques used in this audio visualizer can be applied to game scenes which do not necessarily require high visualization precision and have a mostly aesthetic purpose. A good use case would be an animated background and/or other atmospheric elements which react to an audio loop.

### 4.2.2 Cutscene using manual data sonification

This Unity prototype uses audio visualization and data sonification in a more narrative-focused way, combining the spectrum analysis capabilities of the AudioSource and AudioListener classes with manual setup of triggers for audio and visuals. The cutscene aims at setting the mood through a combination of synced audiovisual content, as well as to highlight the importance of a particular location while maintaining a mysterious vibe.

While using spectrum analysis for audio visualization can yield great results, in certain situations it may be quicker and easier to achieve the desired effect by manually setting up the audiovisual events. Non-interactive cutscenes are one such example, as the developer usually wants them to display the exact same behaviour every time they happen. Using a spectrum analysis algorithm could be an overly complicated approach to setting up something that essentially never needs to change; additionally, precise fine-tuning of a given effect can often be easier to perform by changing the relevant values manually.

For this reason, the cutscene has several audio triggers placed around it with a custom script attached (AudioTrigger.cs). The audio trigger script offers a number of customization options: audio clip selection, audio volume, fade-in switch, fade-in duration, and fade-in curve. Combined with two separate scripts for one-shot and looping sound sources, this allows the developer to fine-tune the way in which a specific audio file is triggered at a specific point of time. Additionally, one of the trigger objects has another script attached to it which handles a skybox change. Figure 5 displays the position of the audio triggers along the train track.

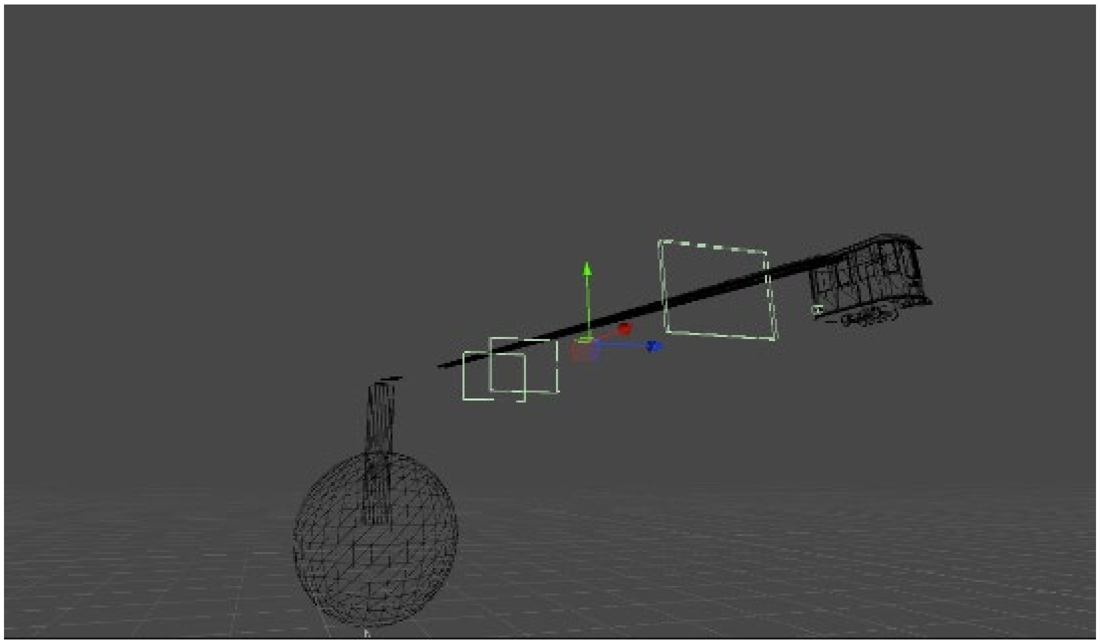


Figure 5. Positions of audio triggers in the scene

The tower at the far end of the scene is supposed to be the focal point of the whole arrangement. It has a point light and an audio source attached to it, as well as a script (TowerPulse.cs) which handles the synchronized changes in audio volume and light intensity. The script allows the developer to specify the rates at which audio volume and light intensity should increase every frame, as well as the maximum volume at which both values are reset to 0, after which the cycle continues. This essentially creates a ramp-shaped LFO. Figure 6 shows what the tower lighting looks like while its intensity is high.

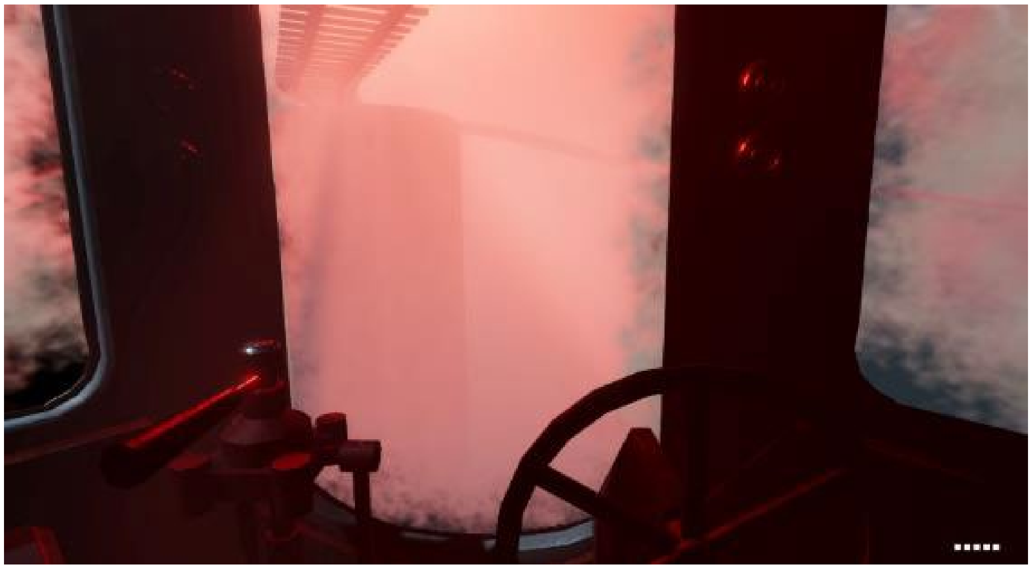


Figure 6. A close view of the tower

The way the tower is represented in the scene can be viewed in terms of both audio visualization and data sonification. Technically, the most important value in the TowerPulse.cs script is the maximum volume, as it determines the volume at which a new cycle of the waveform begins; from this standpoint, the audio is the primary element which also gets visualized by the intensity of the light source. On the other hand, the player can faintly hear the low-pitched drone emitted by the tower right before he or she even sees the red light, and the tower itself is only revealed when the player arrives a little closer to it. By this logic, the fact that the player is approaching a potentially important location is conveyed through audio among other things, allowing it to be viewed as an instance of data sonification as well. Additionally, this scene opens up the possibility to use the tower's sound as an earcon further in the game, as it allows the player to establish the mental connection between the sound and the importance of the location.

Finally, the perceived importance of the tower is highlighted by the simple audio visualizer in the bottom left corner of the screen. It operates on the exact same principles as the audio visualizer described in chapter 4.2 and provides a visual representation of the loudness of specific frequency bands in the tower's audio. This is the only element in the scene which requires no manual setup outside of selecting the frequency bands for each of its segments, as the audio visualizer's purpose here is purely aesthetical and requires no extra precision.

Overall, this prototype demonstrates the way in which audiovisual content can be synchronized manually in order to craft a particular experience which is supposed to look and sound the same every time. While the same (or at least similar) results could have been obtained through output and spectrum analysis algorithms, manual placement of trigger objects has proved to be an efficient way of achieving the desired outcome in this case.

## 4.3 Unreal Engine 5 Project

The primary goal of doing the thesis project in Unreal Engine 5 was to create a series of audio-focused prototypes which could be easily integrated into present and future projects by Dark Amber Softworks. Each prototype’s idea was based on team discussions to ensure maximum relevance to the company’s needs. This section describes the five scenes which the project consists of.

All prototypes were created using Blueprints in Unreal Engine 5. C++ implementation is planned for the future. Unless specifically noted, all the audio in the scenes is generated in real time using MetaSounds within the engine.

### 4.3.1 Audio visualization: submix envelope follower

This prototype focuses on audio visualization and can be used for puzzles/obstacles which have to be resolved through audio manipulation. Real-time envelope following analysis is used to visualize the audio in the scene. To understand the operation of this prototype, it is necessary to be familiar with submixes in Unreal Engine 5, as well as with the concept of envelope following.

The official documentation for Unreal Engine 5.1 defines a submix as a “DSP (digital signal processing) graph that is always running, even when no audio is being sent” (Epic Games, Inc. n.d.). A simple way to think about submixes is that they allow the developer to group multiple sound sources together and apply DSP effects to them; aside from a few differences, submixes are very similar to audio buses (which are also present in Unreal Engine). Submixes in Unreal Engine 5 support real-time analysis through envelope following or spectral analysis (Epic Games, Inc. n.d.). This prototype uses envelope following analysis for audio visualization.

An envelope follower is an algorithm which analyzes the amplitude of an audio signal and outputs a control signal based on the sound’s dynamics. The control signal is usually smoothed out by having its own attack and decay parameters, which control how fast the envelope follower reacts to each change in the signal’s amplitude (Ferguson 2020). Envelope followers should not be confused with envelope generators, which are utilized in other parts of this project. Unreal Engine 5 offers multiple ways of using envelope followers; the submix envelope following approach is used in this prototype.

The basic setup for the scene involves two platforms which are interconnected by a bridge. The player starts on one of the platforms and needs to get to the other one; however, the bridge segments are displaced because their position is affected by the sound coming from a source on the start platform. The player needs to touch the sound source (represented by the glowing white cylinder) to fade out the sound, which will cause the bridge segments to float back into place. Figure 7 shows the basic layout of the scene. Figure 8 illustrates one of the possible ways the bridge may look like while the sound is playing.

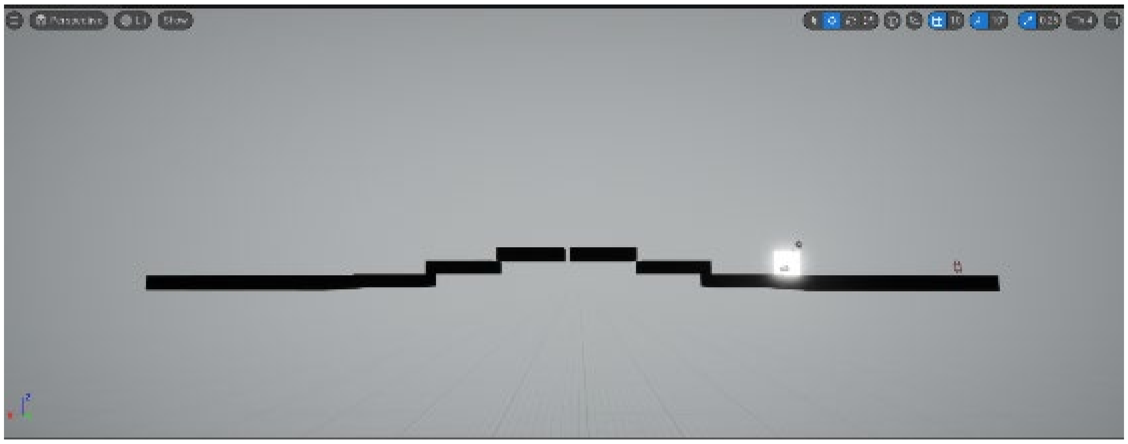


Figure 7. Basic layout of the scene

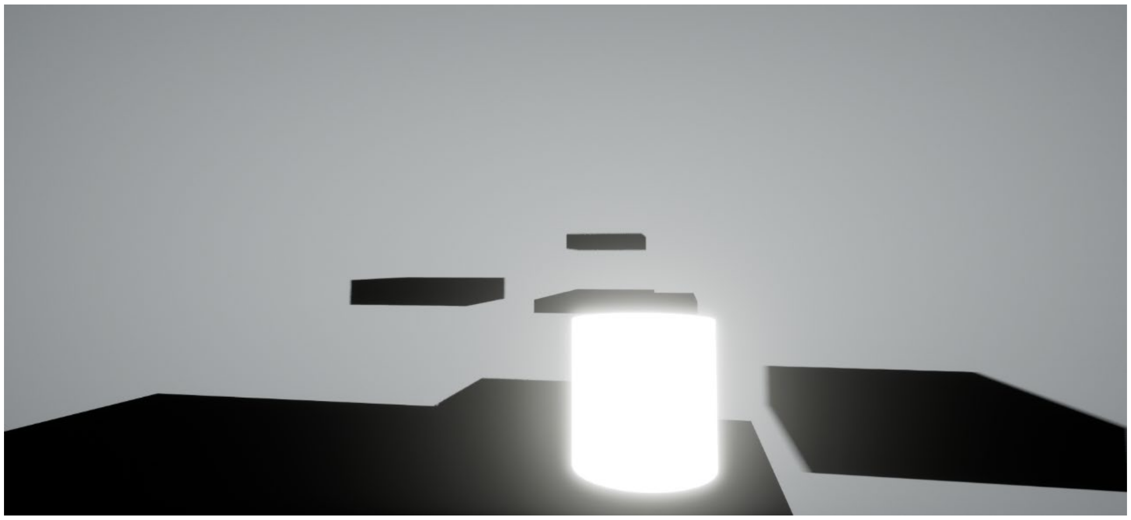


Figure 8. Bridge segment positions affected by audio

There are two distinct looping sounds in the scene: a noisy sound (which affects the position of the bridge segments) and a calmer ambient soundtrack. Each of the two sounds is sent to its own submix; the ambient soundtrack submix has a sidechain compressor (represented in Unreal Engine 5 by a submix effect of the SubmixEffectDynamicsProcessorPreset type) which is controlled by the noise submix. Sidechain compression automatically reduces the amplitude of the affected signal when the amplitude of another specified audio signal increases. The compressor is set to a low threshold and high ratio, meaning that the ambient soundtrack is playing at low volume while the noise is still audible but fades into full volume once the noise has been turned down.

The bridge segments have their own blueprint which facilitates envelope following. This is done by getting a reference to the submix which the noise goes to, then calling an Add Envelope Follower Delegate node and specifying that submix as the target at the start of the scene. This node will fire an event every time the envelope follower value changes, providing an array of envelope values (one per channel) each time. In this prototype, the array has a length of 2 because the project uses stereo sound; only one of the channels is used for analysis. After the envelope follower delegate has been added, it is also necessary to call the Start Envelope Following node targeting the same submix.

Each time the envelope follower event is fired, the bridge segment blueprint calls one or several of its move functions (for the X, Y and/or Z axis). The move functions take a float input for the amount of movement which is calculated from the envelope follower's output via the Map Range Clamped node; the maximum and minimum move distances are set at random within a specified range of floats. The range of envelope follower values is mapped to the move distance range in a way that low envelope follower values result in negative values (or positive ones if the maximum move distance happens to be negative). This results in the segments moving back and forth with the sound, rather than just moving away without changing direction.

Finally, the bridge segment blueprint has exposed Boolean inputs for X-, Y- and Z-axis movement. This way, the level designer can specify the direction of the movement while the rest of the parameters are either randomized (maximum and minimum move distances) or controlled by the envelope follower values (the actual movement amount).

To turn off the noise, the player needs to collide with the cylinder, which will trigger a volume fade-out over 5 seconds. As the signal fades out and envelope follower values approach zero, the bridge segments will smoothly float back into place, allowing the player to make their way to the other platform. It is important to note that the submix to which the noise is sent should have Auto Disable turned off, otherwise the envelope following will stop when the noise fades out and not let the segments fully reach the appropriate position.

This prototype is a simple but effective way of utilizing audio visualization as a game mechanic rather than a purely aesthetical element. The idea can be further developed by including several more sound sources (which may or may not affect the position of the bridge segments) and allowing the player to increase or decrease the volume of each sound. This way, the player will have to interact with multiple sound sources and observe the effect of those interactions, which can provide a foundation for more complex puzzles or obstacles.

### 4.3.2 Data sonification: proximity-based diegetic sound

This scene focuses on creating a diegetic sound source which grows louder as the player approaches it. There are multiple ways of achieving that in Unreal Engine 5: for example, by making a script which would explicitly control the volume of the sound, or by using the UE.Attenuation interface in the MetaSound source. However, this scene uses the simplest implementation which does not require any code at all: it sets up attenuation parameters of audio components in the sound source blueprint.

The scene consists of a corridor which is sealed at both ends. One of the ends is the starting point for the player, while the sound source is placed at the other end. The sound source consists of a point light (which is one of the few light sources in the scene), a Niagara particle system using Vortex Force to make the particles spin around, and two audio components (ProximitySoundFar and

ProximitySoundClose). Each of the audio components has its own MetaSound source, as well as its own attenuation settings. Figure 9 shows a close-up view of the sound source in the scene.

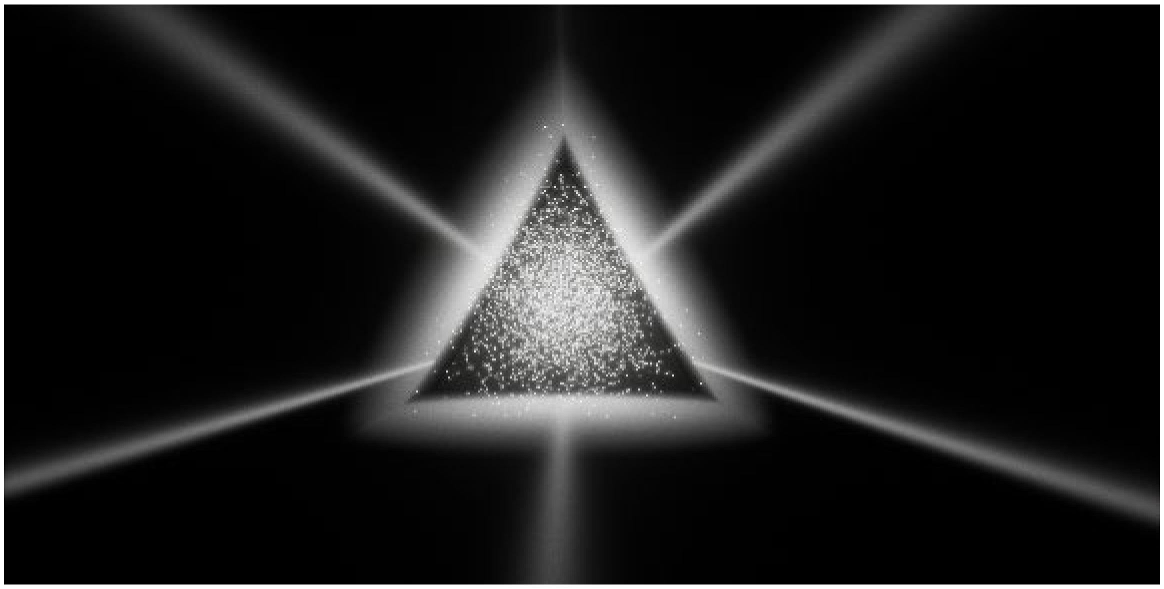
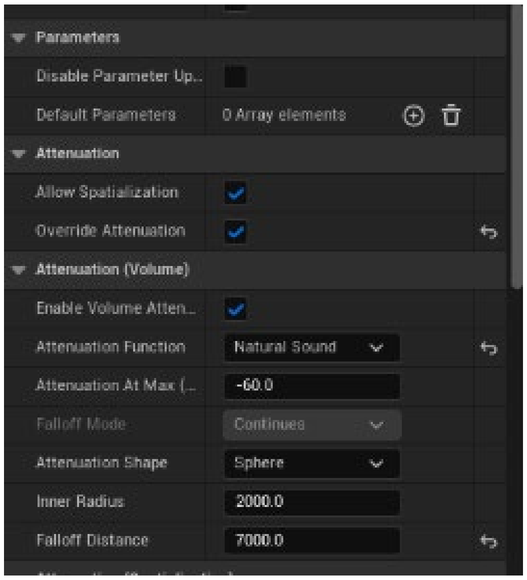


Figure 9. A close-up view of the sound source object

The audio components are set up in a way that one of them starts fading in almost immediately when the player starts moving towards the sound source, while the other is only audible when the player gets close to it. This is achieved by setting up their attenuation settings individually. Both sources have Allow Spatialization and Override Attenuation set to true in the blueprint editor; this allows the developer to access specific attenuation settings. The important section for this scene is Attenuation (Volume), which exposes parameters for the distance and character of the attenuation process. For both audio components, Attenuation Function is set to Natural Sound to simulate natural sound falloff behaviour, while Attenuation at Max is set to -60 (meaning that the sound source is playing at a volume of -60dB when the listener is beyond falloff distance). Attenuation Shape is set to Sphere for both components.

The only difference between the two components is in the Inner Radius and Falloff Distance settings. Inner Radius defines the distance between the listener and the sound source at which attenuation begins; in other words, as long as the listener is within this distance, the sound is playing at full volume. The Falloff

Distance parameter determines the distance over which the sound will attenuate: the further the listener moves away from the source, the quieter the sound will become until eventually decreasing to the minimum level (specified by the Attenuation at Max parameter here) upon reaching the Falloff Distance value. Figure 10 shows the settings for the audio component which fades in when the player is close to the sound source.



| | |
|---|---|
| **Parameters** | |
| Disable Parameter Up... | ☐ |
| Default Parameters | 0 Array elements |
| **Attenuation** | |
| Allow Spatialization | ✓ |
| Override Attenuation | ✓ |
| **Attenuation (Volume)** | |
| Enable Volume Atten... | ✓ |
| Attenuation Function | Natural Sound |
| Attenuation At Max (... | -60.0 |
| Falloff Mode | Continues |
| Attenuation Shape | Sphere |
| Inner Radius | 2000.0 |
| Falloff Distance | 7000.0 |

Figure 10. Attenuation settings for one of the audio components

Figure 11 represents the top-down view of the Falloff Distance and Inner Radius settings for both audio components. Starting from the largest, the spheres are ordered as follows: ProximitySoundFar Falloff Distance, ProximitySoundClose Falloff Distance, ProximitySoundFar Inner Radius, ProximitySoundClose Inner Radius.

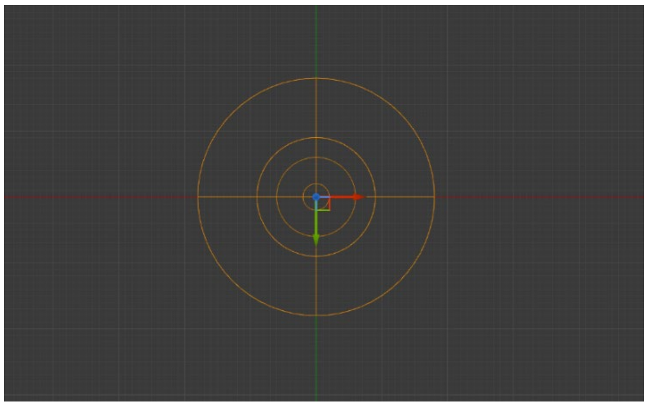


Figure 11. Visual representation of the falloff distance and inner radius for both audio components

With this parameter arrangement, the player first starts hearing a buzzing sound slowly fading in while moving along the corridor towards the sound source. About halfway through, another sound starts fading in – a more tonal but still noisy loop. Soon after that, the buzzing sound reaches its full volume, while the tonal loop keeps fading in. Eventually the latter reaches its own full volume when the player gets very close to the sound source.

To make the scene more visually appealing and highlight the introduction of the tonal loop, a post-processing volume is placed around the sound source with its bounds roughly corresponding to the tonal loop's falloff distance. It adds bloom and film grain effects to the camera around the time when the sound becomes audible and removes them if the player gets further away from the sound source.

Additionally, the particles in the sound source blueprint react to the frequency spectrum of the sound in the scene: the higher the volume, the faster some of them move (depending on the sound's frequency content). This is achieved by setting up a Niagara Module Script (not to be confused with the Niagara System which sets up the visual representation and overall behaviour of the particles). Among other things, Niagara Module Scripts allow developers to use real-time spectrum analysis to control various parameters of a Niagara System. In this case, velocity of the particles is controlled by the frequency spectrum of the master submix, which all the audio in the scene flows into. The frequency band which each individual particle reacts to is determined by the particle's normalized age, which is its actual age mapped to a range of 0 to 1 (where 1 represents its maximum age determined by the Niagara System settings). The result is that the particles are moving quite slowly when the listener is far away and the sound is playing at a low volume. As the player gets closer, the sounds start fading in, causing the particles to move faster and in a slightly more chaotic way. While this has a purely aesthetical purpose (just like the use of the post-processing volume), it can also slightly improve accessibility by introducing visual changes to the scene when the player is close to the sound source.

This simple setup results in a fully diegetic sound source which requires no code or blueprint nodes to operate. The flipside of this simplicity is its limited usability: while this approach can work very well for highlighting a single specific object in the scene (such as a portal to the next level), it is not particularly usable if the scene has multiple objects which need to be sonified with the exact same sounds simultaneously. The reason is that the sound source has the audio components attached to itself, meaning that two or more identical sound sources in the scene will result in two or more identical sets of sounds potentially playing at the same time, which is not always desirable. Additionally, this approach implies that the sound source is fully spatialized (meaning that the sound's position in the stereo field depends on the relative angle at which the listener is rotated). Unless these effects are intended by the developer, a better approach to the sonification of multiple similar objects would be to have a non-diegetic sound (or set of sounds) which reacts to the events happening in the scene. The implementation of such an approach is explained in the next section of this thesis.

### 4.3.3 Data sonification: proximity-based non-diegetic audio

As detailed in chapter 4.3.2 of this thesis, attaching a sound source directly to an object in the scene may not always be the optimal solution as it may cause various side effects (such as several identical pieces of background music playing diegetically at the same time, which may lead to volume jumps and phasing issues). While that approach may work well for the sonification of a single unique object in the scene, developers might often want to use non-diegetic sound effects and/or background music as a functional element of the game – for instance, as an alert that one or more enemies are near. In this case, instead of sonifying each enemy individually, it makes more sense to focus on whether or not the player is within dangerous distance from any number of enemies. In other words, the developer can essentially set up an audio cue for a Boolean value (true/false), which in this scene answers the question of whether or not there is at least one enemy within a specified distance from the player.

The scene consists of a large area with four enemies moving back and forth across it. The player starts near the edge of the area and can freely move around, observing the audio effects of getting closer to the enemies or further away from them. To make things more interesting, there are three different sounds (EnemyFar, EnemyMedium and EnemyClose) which get layered on top of each other depending on how close the distance is between the player and the enemies. Additionally, there is one ambient noise track which is playing continuously but fades out while enemies are near.

There are multiple ways to set up a non-diegetic sound which would still react to events happening in the scene. A rather common solution would be to use the singleton pattern, where a single (and often persistent) object is exposed to all other objects in the scene. By default, Unreal Engine 5 provides developers with several objects of this kind including the Game Mode blueprint, which can be edited or replaced to implement the needed functionality. This scene uses a customized Game Mode blueprint to set up the non-diegetic audio.

It is worth noting that there is much debate among developers regarding the use of the singleton pattern. Documenting the pros and cons of the pattern is beyond the scope of this work, but using the Game Mode blueprint appears to be one of the viable solutions for implementing the functionality discussed in this chapter.

The three sounds used for alerting the player about enemy presence are attached to the Game Mode blueprint as audio components. Auto Activate is set to false for all three to prevent them from playing automatically at the start of the scene.

The scene uses a specific player pawn which is different from other scenes. This pawn has three collision spheres of different sizes attached to it. The spheres are set to generate overlap events when colliding with enemies. At the start of the game, the player pawn gets a reference to the active Game Mode blueprint (which is exposed to every object in the scene by default), then binds overlap events to each sphere. The overlap events ensure that whenever an overlap occurs between an enemy and a sphere, an appropriate sound source attached to the Game Mode blueprint starts playing and fades in (using the Play and Fade

In nodes; a status check is also implemented before that to make sure that the sound source is not already playing). Whenever an enemy stops overlapping with a sphere, the sphere checks whether it is overlapping with any other enemies; if it finds none, the audio is faded out and stopped (using the Fade Out and Stop Delayed nodes with matching time values).

Figure 12 shows the top-down view of the player pawn with three collision spheres attached. Each sphere is generating overlap events which affect a particular sound source in the Game Mode blueprint (EnemyFar, EnemyMedium and EnemyClose respectively, starting with the largest sphere).

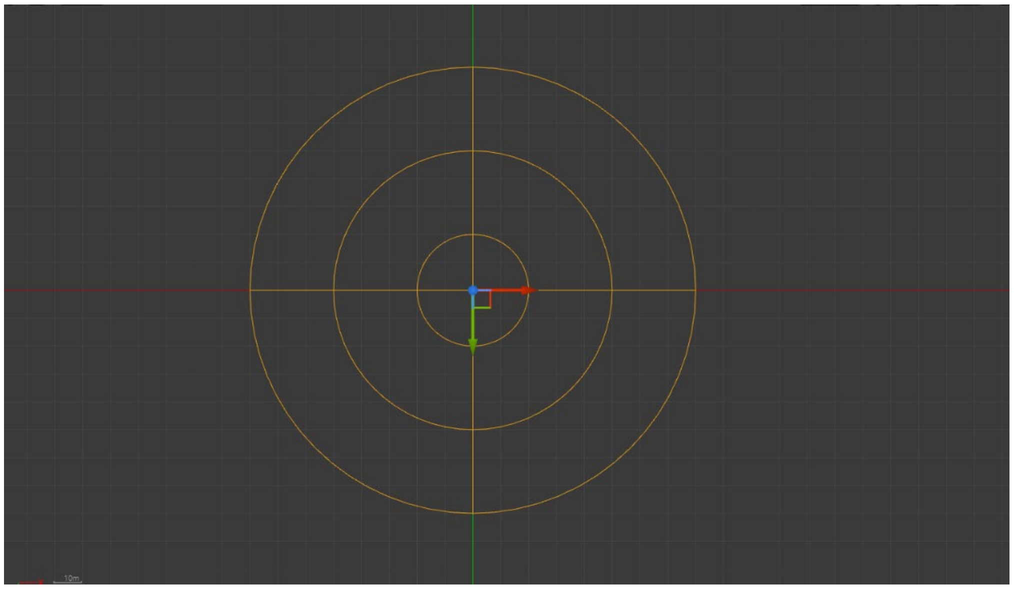


Figure 12. Collision spheres attached to the player pawn

Each of the sound sources attached to the Game Mode blueprint has its own submix (technically the scene would work the same if all of them were assigned to a single submix for enemy sounds, but having individual submixes allows for setting up more complex effects in the future). The ambient noise track is also assigned to its own submix which has a sidechain compressor attached to it. The submix for the EnemyFar sound is assigned as the key source for the compressor; in other words, the ambient noise track fades out whenever at least one enemy is within the largest collision sphere attached to the player pawn, and it fades back in when the player gets sufficiently far away from the enemies.

This arrangement results in a non-diegetic sound which is always centered in the stereo field and consistently reacts to the proximity of enemies regardless of their actual count (meaning that it does not get louder with each additional enemy overlapping one of the collision spheres). This way, the audio can be used as a general alert for the player while also retaining its atmospheric function.

Two issues have been identified while testing this setup. Firstly, the enemy sound may not start playing if at least one enemy starts the game within the radius of any collision sphere attached to the player pawn. This is likely caused by the fact that in this case the overlap event happens before the audio effects are bound to the spheres. If the scene is set up so that enemies might begin the game close to the player, a different system might be needed. Otherwise, the easiest way to overcome the issue is to simply move the player's starting position further away from the enemies.

The second issue is that sounds may sometimes fail to play if the enemy is rapidly entering and exiting the bounds of one of the player pawn's collision spheres. This is caused by the Fade Out and Stop Delayed nodes in combination with the status check which prevents an already playing sound source from retriggering. While this is mostly a non-issue with the current movement speed and fadeout time settings in the scene, it may be worth implementing more variables and status checks if the player and/or enemies are supposed to be moving rapidly in the scene.

Additionally, it should be noted that the current setup of the Game Mode blueprint (which has the three sound sources attached to it directly in the blueprint editor) works well if the alert sounds are supposed to be consistent throughout the game. Otherwise, it might be beneficial to attach them dynamically at the start of the level (or at any other appropriate point).

### 4.3.4 Data sonification: procedural audio with parameter mapping

This scene explores the usage of MetaSounds' synthesizer capabilities for data sonification through procedurally generated audio. To achieve that, a complex synthesizer voice with multiple parameter inputs has been constructed in MetaSounds and used as a sound source in the scene.

Procedural audio generation implies that the developer provides a set of rules and boundaries for the process but has no way to influence the outcome directly. This matches up well with the parameter mapping approach to data sonification, which (as discussed earlier in this thesis) generates an audio representation of the sonified object or event based on its properties. This scene combines the two concepts by allowing the player to hear the results of different values being sent to the synthesizer's inputs.

The scene consists of a number of planets, each having four parameters (Population Density, Atmosphere Density, Water Coverage and Rings Density) with different values. Those values are used to control the main sound source in the scene, which is a complex synthesizer voice with external inputs for the following parameters:

- Saw wave oscillator frequency
- Saw wave oscillator amplitude envelope attack
- Saw wave oscillator amplitude envelope decay
- Sine wave oscillator frequency
- Sine wave oscillator amplitude envelope attack
- Sine wave oscillator amplitude envelope decay
- Noise oscillator amplitude envelope attack
- Noise oscillator amplitude envelope attack
- Mixer gain: saw wave oscillator
- Mixer gain: sine wave oscillator
- Mixer gain: noise oscillator
- Filter cutoff frequency
- Bit depth
- Sample rate

Each of the inputs goes through a Map Range (Float) node before reaching its destination. That node allows transforming a value from a specified range to an appropriate value of a different range (for example, 0.0 – 1.0 for mixer gain inputs). The Map Range node is clamped to prevent the output value from going beyond the specified range boundaries. This way, the system allows the developer to provide virtually any float value to the input and still get somewhat predictable results. The input range is set to 0.0 – 100.0 for each input but can easily be changed to accommodate for different parameters.

The scene starts with the planets laid out in front of the player; this time, the player pawn cannot move but allows the player to click on objects with the mouse. An ambient noise soundtrack is playing – this is another MetaSound source which uses a different synthesizer voice and has no parameter inputs (meaning that its sound characteristics do not depend on any other objects in the scene).

Whenever the player clicks one of the planets, a UI widget appears on the screen, listing all the parameter values for the selected planet. Additionally, a Niagara particle system is spawned at the planet's location to help visualize the selection. Figure 13 shows an example of an active widget.

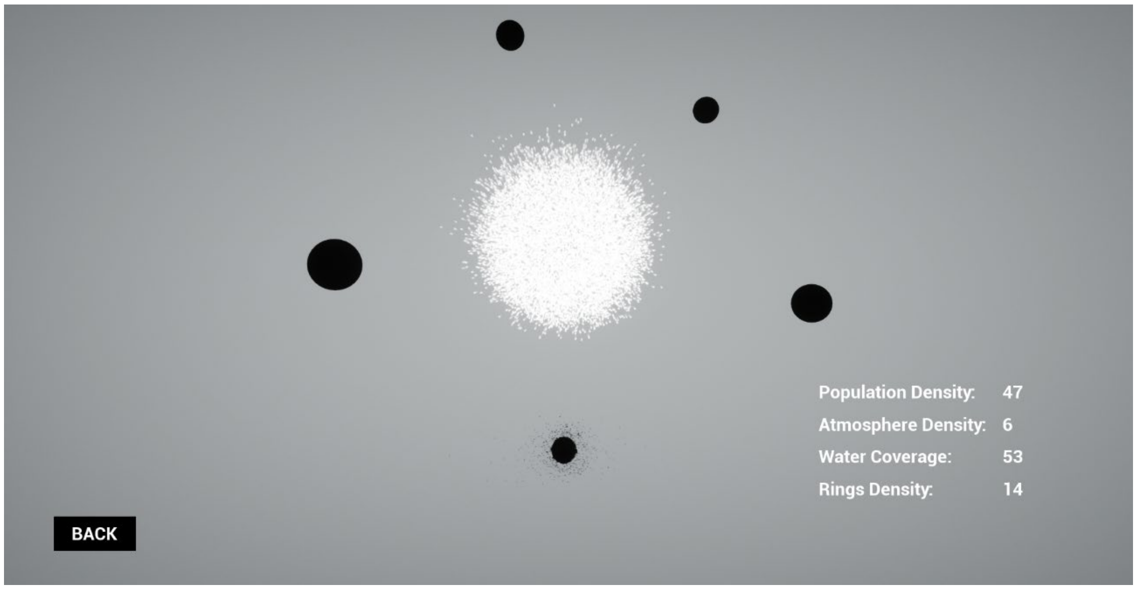

Population Density: 47
Atmosphere Density: 6
Water Coverage: 53
Rings Density: 14

Figure 13. UI widget layout

As soon as the widget appears, the MetaSound source used for data sonification starts playing while the ambient noise track fades out (this is once again achieved by assigning the sounds to two different submixes and using the sidechain compressor effect on the ambient noise submix). This time, the sound that the player hears is determined by the planet parameters, which are listed in the bottom right corner of the screen.

When the Back button is clicked, the widget disappears along with the particle system. The sonification track stops playing, prompting the ambient noise track to fade back in. After that, the player can repeat the process by clicking another planet and hearing a different sound (assuming that the planet’s parameters have been assigned different values).

The way the scene works is that each planet is an instance of the same blueprint which has four public variables of type float (the four parameters which are used as inputs for the synthesizer). The synthesizer that is used for data sonification is also a separate blueprint, which allows the Game Mode blueprint to have a variable of that type. That variable is referenced both by the planet blueprint and the UI widget to start and stop the audio; the planet blueprint also uses it to send the values of its parameters to the synthesizer’s inputs. Spawning the UI widget and the Niagara particle system is also handled by the planet itself via an On Clicked event, while the logic for removing the widget and the particles resides in the UI widget blueprint.

As previously noted in this thesis, parameter mapping can be a challenging way to implement data sonification because of its inherent complexity and unpredictability. Indeed, the sound source in this scene is biased towards noisy textures rather than conventional musical results, which may or may not fit a particular game. It should also be noted that each mixer input (for the saw wave, sine wave and noise oscillators) is mapped to have the full range of 0.0 to 1.0; this means that a planet with all parameters set to 0 or just above that will produce little to no sound. This behaviour is intentional, just like the overall character of the sound in the scene. However, it is possible to achieve more musical results by adjusting the ranges to which input ranges are mapped, as well as by using MIDI note quantization and/or trigger sequences to introduce conventional rhythm and melody into the sound.

### 4.3.5 Data sonification: adaptive loop-based background music

The fifth and final prototype scene in the project focuses on creating a loop-based adaptive soundtrack where all the loops are synchronized to each other. While this scene uses MetaSounds just like all the other ones, it is the only case in this project where externally recorded audio is used as actual sound sources.

In MetaSounds, playing an imported audio file is done via the Wave Player node. Among other things, the node allows looping the sound and has Play and Stop trigger inputs, which can be triggered externally. This functionality alone is enough to set up a simple looping soundtrack where a single audio file would be looped throughout the level (with optional play/stop triggers placed in the game world). Additionally, it allows the developer to play multiple looping audio files which may or may not be in sync. Simultaneously playing multiple audio loops which are triggered at different points in time and/or have different lengths is a valid generative music technique; depending on the audio content, musical results can be reliably achieved by using this approach.

Figure 14 shows the Wave Player node in MetaSounds. Inputs are placed on the left side of the node while outputs are on the right.

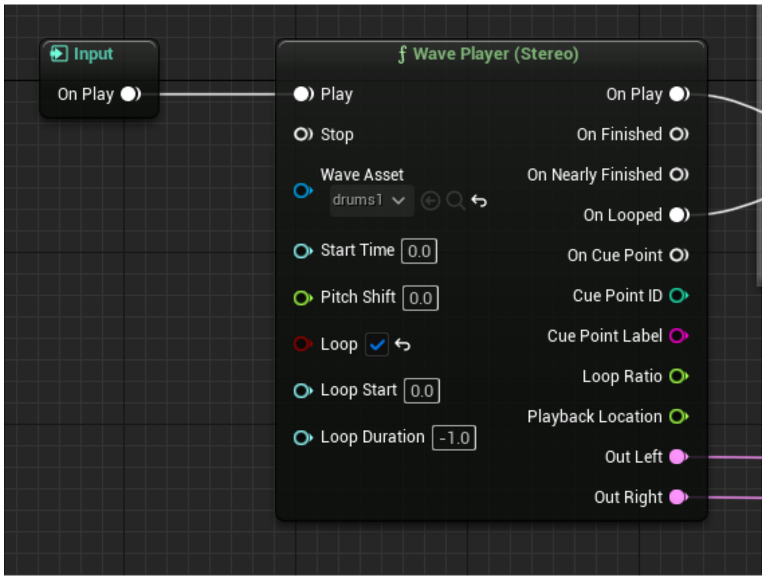


Figure 14. Wave Player node in MetaSounds (Unreal Engine 5)

However, a more complex setup is required if the soundtrack consists of multiple looping parts which are supposed to be synchronized to each other. In this case, simply triggering the loops at various points during the gameplay is not enough: unless the player manages to do that with perfect timing, the loops will be playing out of sync and the result may be vastly different from the composer’s intention. MetaSounds provides developers with tools for trigger manipulation, which can be helpful in setting up an adaptive soundtrack with multiple looping pieces. This chapter details one of the possible solutions which utilizes those tools.

The scene consists of a sealed corridor with four platforms as its floor, each platform adding a new loop to the soundtrack. The player starts at one end of the corridor and can freely move in any direction within the enclosed area.

The soundtrack consists of four separate loops: drums, bass, and two guitar parts. The loops are supposed to be added on top of each other and are not supposed to be stopped at any point. Each loop is a separate MetaSound source which uses a Wave Player node with Loop set to true; however, the only sound source that is actually played is the master loop (which is drums in this case). The master loop’s MetaSound graph references all the other loops, sending them an On Looped trigger: in other words, it attempts to trigger every other loop in the scene each time it has completed an iteration. The master loop’s output is then mixed with all the other loops’ outputs via the Stereo Mixer node.

Each non-master loop is set up identically: it uses a Wave Player node with Loop set to true, a Trigger Control node, and a Trigger Delay node. The latter two nodes are needed to ensure that the loop is only triggered once during the whole playthrough; otherwise, if the non-master loop is longer than the master, it would be retriggered with every iteration of the master loop and never play to the end. The Trigger Control node acts as a gate which can be either open or closed at any given point in time. When the gate is closed, it does not let triggers through and produces no output. In the graph, it is set to start in the open state; this means that the first On Play trigger the loop receives will go through the Trigger Control node to the Wave Player node and start the playback. The On Play trigger also goes to the Trigger Delay node, which in turn sends a trigger to close the gate after 0.1 seconds (this value can be anything that is sufficiently short to make sure that the Trigger Control node is closed by the time the next On Play trigger arrives). With this setup, the first On Play trigger goes through and starts the loop, and the Trigger Control node gets closed shortly after, ensuring that the loop is never retriggered again. Figure 15 shows the layout of any non-master loop graph in this scene.

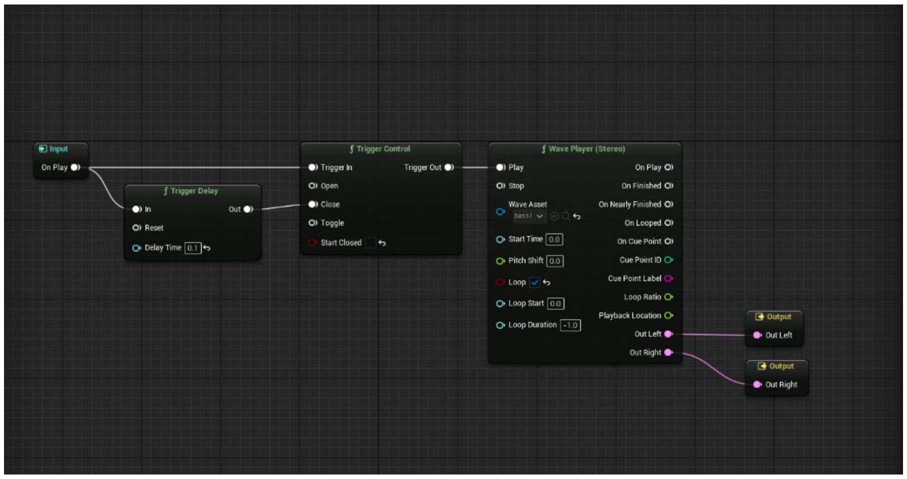


Figure 15. Non-master loop graph layout

The master loop graph also uses Trigger Control nodes to send triggers to the other loops. Without those, every other loop would start playing simultaneously as soon as the master loop has finished its first iteration. The Trigger Control nodes in the master graph are set to start closed, and custom triggers are used to open the gate. Each of the custom triggers is bound to its own trigger box placed at the start of the next platform. This way, the further the player gets through the level, the more triggers get through the Trigger Control nodes, and the more loops are added to the master loop.

Finally, the master loop utilizes the Trigger Counter node to keep the playback in sync. The node is required if the master loop is shorter than the other loops; in this scene, the master loop is intentionally set to be 4 times shorter than the other loops to showcase this functionality. The Trigger Counter node's purpose in this case is to keep track of the number of triggers it has received (which corresponds to the number of times the master loop has played through) and to send out a trigger every time the Reset Count is reached. Since the master loop is 4 times shorter than the other loops, Reset Count is set to 4. However, only sending an On Looped trigger to the Trigger Counter node would not be correct, as the trigger appears to be sent out at the start of the next iteration; this means that in order to reach the Reset Count of 4, the master loop would need to play 5 times (one initial playthrough + 4 iterations which actually yield the On Looped triggers). Thus, the On Play trigger also needs to be sent to the Trigger Counter node so that the initial playthrough is included into the count. This can be done via a Trigger Any node, which outputs a trigger when it receives any of the specified input triggers. Figure 16 shows the full layout of the master loop graph.

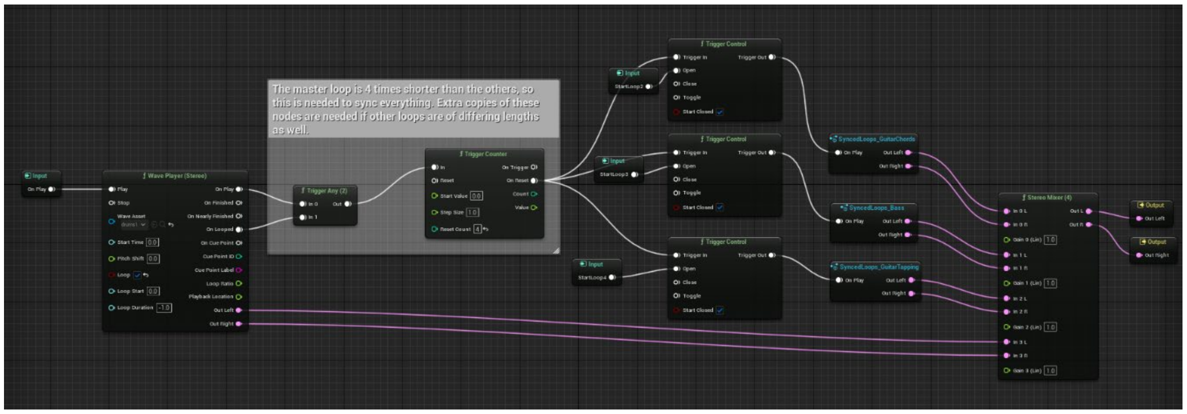

The master loop is 4 times shorter than the others, so this is needed to sync everything. Extra copies of these nodes are needed if other loops are of differing lengths as well.

Figure 16. Master loop graph layout

The master loop is set up and triggered in the Level Blueprint as soon as the scene starts. After that, each of its custom triggers used to open the Trigger Control nodes for other loops is assigned to an overlap event with one of the trigger boxes placed throughout the level. Executing a MetaSounds trigger is done in blueprints via the Execute Trigger Parameter node (it is important to make sure that the name of the trigger is spelled exactly the same as in the MetaSounds graph).

Figure 17 showcases the Level Blueprint for this scene. Note the Do Once nodes in the overlap events: while they are technically not required given the Trigger Control setup within the master loop's MetaSounds graph itself, it is still beneficial to use them to make sure that the trigger can only happen once even if the MetaSounds graph is changed. Additionally, they are chained in such a way that triggering the next loop is only possible after having triggered the previous one; with the current scene layout this is not needed, but it can be useful for more complex levels.

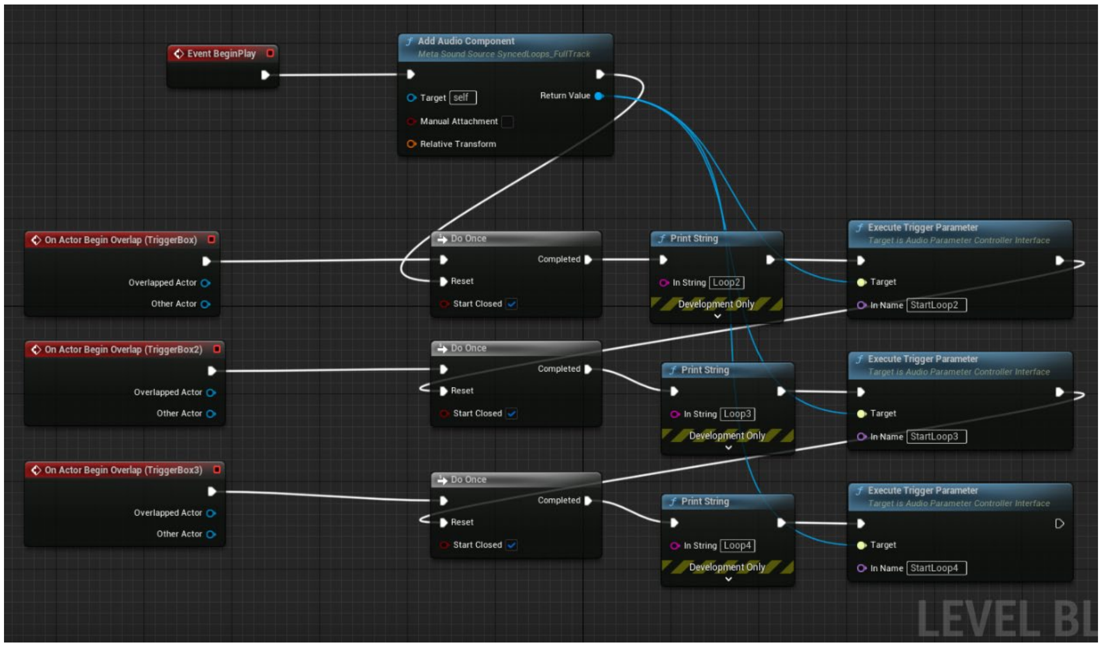


Figure 17. Level blueprint for the scene

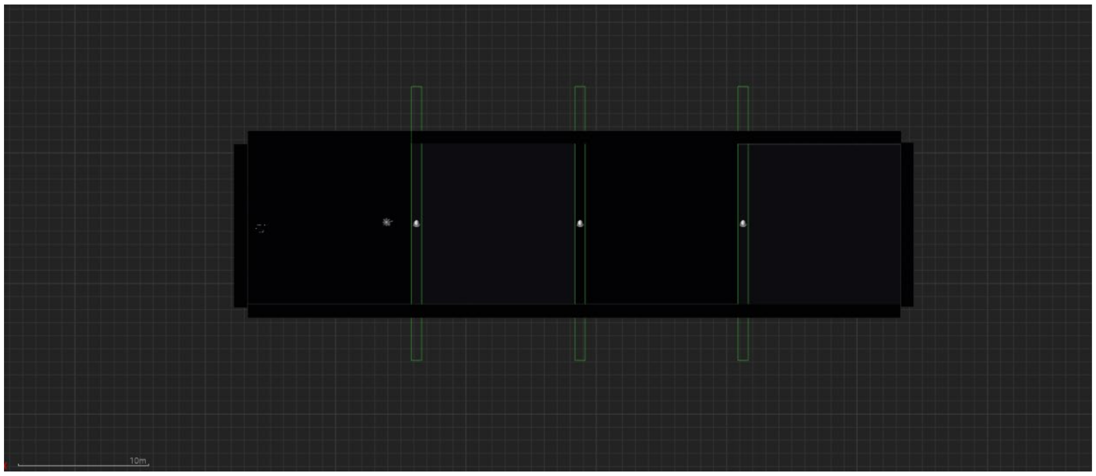


Figure 18. Trigger box layout in top-down view

Figure 18 shows the layout of the three trigger boxes (drawn with a green outline) in a top-down view of the level. The resulting setup allows the developer to create an adaptive loop-based soundtrack which builds up as the player progresses through the level. Currently, loops can only be triggered once and cannot be stopped regardless of the player’s location in the level. To expand the functionality, additional trigger boxes can be placed in the level (or the existing ones can be reused) to stop individual loops.

The easiest way to prepare audio loops for such a setup would be to make sure that they are all the same length, which would allow the developer to get rid of the Trigger Counter node in the master loop graph. However, as shown in this prototype, it is possible to use shorter master loops with the help of the Trigger Any and Trigger Counter nodes (as long as the length of the other loops is divisible by the master loop length with a remainder of 0). Additional Trigger Counter nodes may be needed if more length variation is introduced among the loops; depending on the length ratios between individual loops, it may or may not be convenient to set it up in the engine, so it is recommended to try and make all the loops the same length in the DAW.

## 4.4 Focus group testing

As part of the research, the Unreal Engine 5 project was tested by the rest of the Dark Amber Softworks team as a focus group. Each team member has a different technical background and skill set, meaning that the testing conditions varied from one team member to another. Thus, the focus group testing was conducted in a free format where each participant was asked for their general impressions, opinions and suggestions regarding the project.

The overall feedback for the prototypes presented in this thesis project was very positive, highlighting the variety of audio-related features explored across the scenes as well as their usefulness for the company’s ongoing and future projects. The modular nature of the prototype was also noted, as it allows for relatively easy implementation of the audio features in new and existing projects alike. Finally, the fact that the project was built in Unreal Engine 5 was well-received, as it has allowed the company to branch out into that environment while continuing the Unity productions as well.

The feedback also included several suggestions for further development of the project. First of all, it would be beneficial to have a uniform UI for each existing scene (and any scenes that might be added to the project in the future). While the current prototypes allow enough parameter customization to be used as development tools and satisfy the company’s needs, they require a good understanding of the concepts and goals relevant to each scene. Developing a user interface which would allow easy control over those parameters would make the project more accessible to users of any experience level while also improving the testing process for feature implementations. A well-developed UI could turn the current collection of prototypes into a fully fledged audio framework which could later be shared with third-party developers (for example, through Unreal Marketplace).

The second suggestion concerns testing the prototypes in VR. As Dark Amber Softworks frequently works on AR/VR projects in Unity engine, it would be useful to add Unreal Engine VR to that while maintaining the focus on the engine’s audio features explored in this thesis project.

Finally, the feedback expressed encouragement to continue developing the project and exploring the audio-related possibilities facilitated by Unreal Engine 5. It has been noted that tools focusing on a single feature implementation are particularly useful, as they can be later used for building more complex scenes in a modular fashion.

# 5 CONCLUSION

In this thesis, the theory behind audio visualization and data sonification in video games has been explored, and a number of techniques has been successfully implemented in the practical project. This work has proved that audio visualization and data sonification can play a significant part in a game – not only as an aesthetical element but also as a functional aspect of the gameplay. In the practical part of this thesis, five different scenes have been created in Unreal Engine 5, each focusing on different aspects and techniques of audio visualization and/or data sonification. The advantages and possible drawbacks of using the techniques have been discussed, and changes have been proposed to accommodate for different use cases. Thus, the thesis has provided a concrete practical example of how various audio visualization and data sonification techniques could be implemented in Unreal Engine 5, which was the primary question for this research.

The research has also provided answers to the following secondary questions:

- What audio visualization and data sonification techniques exist?
- Are any of the techniques supported in popular game development engines such as Unity and Unreal Engine?
- Are there any downsides (either technical or design-related) to using any of the techniques in a game?

While it would have been physically impossible to explore every relevant technique due to the vast nature of the study field, this thesis has highlighted several important techniques which can be directly applied to video games (for example, adaptive background music or parameter mapping for data visualization), while also referencing multiple notable works in the field.

Some of the researched techniques were then implemented both in Unity and Unreal Engine 5 (with the focus being on the latter). While it can be confidently stated that both engines allow for rather complex audio-related implementations, a subjective outtake would be that Unreal Engine 5 provides more audio solutions by default, while Unity commonly expects the developer to create custom implementations and/or use third-party solutions to achieve the result. With that said, all the features implemented in the Unreal Engine 5 project could be replicated in Unity engine as well.

Regarding the possible downsides of using the aforementioned techniques in a game, it is important to consider accessibility concerns during the development. Introducing audio visualization and/or data sonification features may or may not affect the player's perception of the game; if the gameplay strongly depends on audio-related features, the game can become unplayable to individuals who are unable to hear the sound. While it might be impossible to make a game (or any other multimedia product) which would be universally accessible to all target audience groups, it is beneficial to keep accessibility concerns in mind when designing the game, as they can have an effect both on design and marketing decisions for the product.

In conclusion, it can be said that this thesis has fulfilled its goals and provided answers to all of its research questions. Development of the practical project will be continued in the future and the results will be used in Dark Amber Softworks products. Additionally, this thesis might provide a foundation for further research of audio visualization and data sonification in video games.

## REFERENCES

Bandai Namco Entertainment. 2011. Dark Souls. Video game. Tokyo: Bandai Namco Entertainment.

Bandai Namco Entertainment. 2014. Dark Souls II. Video game. Tokyo: Bandai Namco Entertainment.

Bandai Namco Entertainment. 2016. Dark Souls III. Video game. Tokyo: Bandai Namco Entertainment.

Bandai Namco Entertainment. 2022. Elden Ring. Video game. Tokyo: Bandai Namco Entertainment.

Boncz, I. 2015. Introduction to Research Methodology. Pécs: University of Pécs.

Comeau, J. 2018. Let's Learn About Waveforms. The Pudding. Web page. Available at: https://pudding.cool/2018/02/waveforms/ [Accessed 4 March 2023].

Epic Games, Inc. n.d. Unreal Engine 5.1 Documentation: Submixes Overview. Web Page. Available at: https://docs.unrealengine.com/5.1/en-US/overview-of-submixes-in-unreal-engine/ [Accessed 4 April 2023].

Fast Fourier Transformation FFT – Basics. n.d. NTi Audio. Web page. Available at: https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft [Accessed 4 March 2023].

Ferguson, K. 2020. Audio Reactive Programming: Envelope Followers. Blog. 21 November 2020. Available at: https://kferg.dev/posts/2020/audio-reactive-programming-envelope-followers/ [Accessed 4 April 2023].

Geere, D. & Quick, M. 2020. Telling Stories With Data & Music. Blog. 8 December 2020. Available at: https://medium.com/nightingale/telling-stories-with-data-music-f60ac0b5f1be [Accessed 7 March 2023]

Grimshaw, M. (ed.) 2011. Game Sound Technology and Player Interaction: Concepts and Developments. USA: IGI Global.

Hermann, T., Hunt, A. & Neuhoff, J.G. (eds.) 2011. The Sonification Handbook. Berlin: Logos Verlag Berlin GmbH.

Jørgensen, K. 2006. On the Functional Aspects of Computer Game Audio. Research paper. Available at: https://bora.uib.no/bora-xmlui/bitstream/handle/1956/6734/paper-KJorgensen.pdf?sequence=1 [Accessed 11 March 2023].

Juul, J. 2011. Half-Real: Video Games Between Real Rules and Fictional Worlds. Cambridge: MIT Press.

Li, W. & Li, J. 2020. Research on Music Visualization Based on Graphic Images and Mathematical Statistics. *IEEE Access,* 8, 100654. E-journal. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9104977 [Accessed 12 March 2023].

Mantione, P. 2017. The Basics of Additive Synthesis. Pro Audio Files. Web page. Available at: https://theproaudiofiles.com/what-is-additive-synthesis/ [Accessed 5 March 2023].

Ng, P. & Nesbitt, K.V. 2013. Informative Sound Design in Video Games. Conference paper. Available at: https://novaprd.newcastle.edu.au/vital/access/services/Download/uon:13441/ATTACHMENT03?view=true [Accessed 7 March 2023].

Ninja Theory. 2017. Hellblade: Senua's Sacrifice. Video game. Cambridge: Ninja Theory.

Perron, B. & Wolf, M.J.P. 2009. The Video Game Theory Reader 2. New York: Routledge.

What is a Spectrogram? n.d. PNSN. Web page. Available at: https://pnsn.org/spectrograms/what-is-a-spectrogram [Accessed 13 March 2023].

Unity Technologies. 2023. Scripting API: AudioSource.GetSpectrumData. Web page. Available at: https://docs.unity3d.com/ScriptReference/AudioSource.GetSpectrumData.html [Accessed 12 March 2023].

White, P. 1994. Sound Synthesis: Part 1. *Sound On Sound.* E-magazine. Available at: https://www.soundonsound.com/techniques/sound-synthesis-part-1 [Accessed 5 March 2023].

Zhang, Y., Pan, Y. & Zhou, J. 2018. *Journal of Physics: Conf. Series,* 1098. E-journal. Available at: https://iopscience.iop.org/article/10.1088/1742-6596/1098/1/012003/pdf [Accessed 12 March 2023].