BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

PROCEDURAL SOUND-BASED ELEMENTS IN GAMES

PROCEDURÁLNÍ HERNÍ PRVKY NA ZÁKLADĚ ZVUKŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PATRIK JEŠKO**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ POLÁŠEK**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156350

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Ješko Patrik**
Programme:	Information Technology
Title:	**Procedural Sound-Based Elements in Games**
Category:	Computer Graphics
Academic year:	2023/24

Assignment:

1. Survey the current state of sound analysis and procedural generation in games.
2. Design a library for sound-based procedural generation.
3. Implement the library by means of your choice.
4. Create a demonstration of the library's functionality.
5. Evaluate your library and perform a user study.
6. Present your results using a poster and a short video.

Literature:

- Koster, Raph. Theory of fun for game design. O'Reilly Media, Inc., 2013.
- Schell, Jesse. The Art of Game Design: A book of lenses. CRC press, 2008.
- Yao, Richard et al. Oculus VR Best Practices Guide. Online, 2014.
- Leap Motion, VR Best Practices Guidelines. Online, 2015
- Unity Learn. Unity, https://learn.unity.com/.
- Further sources according to the supervisor.

Requirements for the semestral defence:
Goals 1, 2 and a basic demonstration of the library's functionality

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Polášek Tomáš, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	9.11.2023

Abstract

This thesis explores music-driven object manipulation in the Unity game engine, which offers a versatile toolset for artists and creators looking to integrate dynamic elements based on background music. This project implements Fast Fourier Transform using an external library, Onset Detection, beat generation, and related functionalities within Unity. Consumer testing and experimentation demonstrate the potential of the implementation and functionality of the plugin. By creating this proof of concept, the intention is to inspire further innovation in this area and leverage music as a creative tool in not only game design but other media as well.

Abstrakt

Táto bakalárska práca sa zaoberá manipuláciou objektov v hernom prostredí Unity na báze hudby. V tomto projekte sa pozrieme na Rýchlu Fourierovu Transformáciu, detekciu nástupov, generáciu dôb a vhodnú funkcionalitu v Unity. Testovanie s uživateľmi a experimentácia, demonstrujú potenciál navrhnutej implementácie a funkcionalite pluginu. Zámer tohto projektu je inšpirovať inováciu v tejto sfére a využiť hudbu ako kreatívny nástroj do nie len hier ale aj ostatných digitálnych medii.

Keywords

Fast Fourier Transform, Beat Generation, Unity Game Engine, Onset Detection, Dynamic environment, Sound analysis

Kľúčové slová

Rýchla Fourierova Transformácia, Generovanie dôb, Herné prostredie Unity, Detekcia nástupov, Dynamické prostredie, Analýza zvukou

Reference

JEŠKO, Patrik. *Procedural Sound-based Elements in Games.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Polášek

Rozšírený abstrakt

Cieľom tejto bakalárskej práce je zoznámiť sa s problematikou analýzy hudby a vytvoriť softvérový modul (ďalej len "plugin"), ktorý poskytne vývojárom v hernom prostredí Unity analýzu hudby v pozadí a funkcionalitu na manipulovanie objektov nachádzajúcich sa v scéne.

Zaoberáme sa prvkami hudby, ktoré sú vhodné na vizualizáciu, ako napríklad tempo, melódia či hlasitosť. Pre správny návrh pluginu je potrebné pochopiť analýze zvukových signálov, ich spracovaniu, ako celková analýza prebieha a aké matematické funkcie sa na ňu používajú. Spolu sa pozrieme na časovú a frekvenčnú doménu, rýchlu Fourierovu Transformáciu a detekciu nástupov. Taktiež je treba rozobrať rôzne metódy na získavanie dát v Unity, ako napríklad AudioSource.GetSpectrumData alebo AudioClip.GetData.

Plugin sa skladá z viacerých častí; externá časť pre výpočet rýchlej Fourierovej Transformácie, detekcia nástupov na rôznych frekvenčných rozsahoch, manažér udalostí (event manager) a rôzne komponenty poskytujúce funkcionalitu pre užívateľa. Užívateľ si vie zadať frekvenčné rozsahy, ktoré chce analyzovať. Plugin poskytuje rôzne informácie analyzovanej hudby, ku ktorým má užívateľ prístup, ako napríklad priemernú amplitúdu počas hudby, sprektrálne informácie, a či v určitom rozsahu v stanovenom čase nastal nástup. Tieto informácie sú poskytované hlavným komponentom ClipController alebo poslané ako udalosť manažérom udalostí komponentom, ktoré danú udalosť odoberajú. Poskytovaná funkcionalita pluginu na manipulovanie objektov v scéne zahrňuje: pohyb, rotácia a modifikovanie veľkosti objektu na základe udalostí a priemernej amplitúdy, cinematickú kameru, ktorá sa pohybuje po kontrolných miestach na základe času hudby a mnoho ďalších.

Kvalita pluginu bola testovaná dvanástimi účastníkmi, ktorí sa pozreli na videá ukazujúce funkcionalitu pluginu. Na základe ich odozvy vieme zhodnotiť, že plugin má potenciál či už v rytmickej hre ale aj v scénach založených na hudbe.

Touto bakalárskou prácou chceme inšpirovať ďalších ľudí na experimentovanie s hudbou nie len v hrách ale aj ostatných digitálnych médiách.

Procedural Sound-based Elements in Games

Declaration

I declare that I was working on this thesis by myself, with the help of Ing. Tomáš Polášek, and I referenced every source and publication I used.

.....
Patrik Ješko
May 6, 2024

Acknowledgements

I would like to thank Ing. Tomáš Polášek for his patience and useful tips during the whole project.

Contents

1 Introduction **3**

2 Theory **4**
2.1 Fundamentals Concepts 4
2.2 Audio Signal Processing 8
2.3 Procedural Generation in Games 13
2.4 Audio in Unity 13

3 Plugin Proposal **15**
3.1 Plugin Architecture 15
3.2 Analysis 16
3.3 Functionality 21
3.4 Components 23
3.5 Quality of Life 23

4 Testing **25**
4.1 Forest Scene 25
4.2 Space Scene 27
4.3 Results 28

5 Conclusion **30**

Bibliography **32**

A Disk Structure **34**

List of Figures

2.1 Note notation [14] . . . 5
2.2 Figures showing onsets . . . 6
2.3 The equal loudness contours. [15] . . . 8
2.4 Visualization of an audio waveform in the time domain from AudaCity. In stereo, the x-axis is time, and the y-axis is amplitude. . . . 9
2.5 Frequency spectrum from AudaCity. The x-axis is frequency, and the y-axis is magnitude. . . . 9
2.6 Examples of windowing functions [16] . . . 11
2.7 Spectrogram of a section of the song. The x-axis represents time, the y-axis represents frequency (20-20000Hz), and the color represents the magnitude of that frequency in that time. . . . 12

3.1 Architecture of the proposed plugin . . . 16
3.2 Simplified diagram of the event system. . . . 22

4.1 Figures of the forest scene . . . 26
4.2 Figures of the space scene . . . 27

Chapter 1

Introduction

Music is an important part of video games, helping to create the right mood, reinforce emotions, and tell stories. It can also help to create immersion, capture the player's attention, and notify them about certain events. Music is a powerful tool used not only in games but also in movies, TV series, and other forms of media. Sounds and music can also be used as a memorization tool, helping players recognize certain events by sound.

Rhythm games are a well-established genre that relies entirely on music and the natural human reaction to rhythm. They use the elements of music to create captivating gameplay. Precision, accuracy, and flow are essential in these games, which is why they are often hand-crafted to match the song's rhythm perfectly. Some games even use all the elements of a song to create procedural levels.

Using the environment to display music can have many benefits. It can provide players with more information, such as helping them keep track of the beat.

Creating a dynamic environment that responds to music is not a new concept. However, it is usually done manually. This thesis serves as a proof of concept for procedural elements based on music in the Unity game engine. It creates a plugin that can help developers create captivating environments, fun rhythmic gameplay, and much more.

The final plugin is designed for the Unity Game Engine, a powerful, versatile cross-platform game engine that is widely used to develop video games, simulations, and interactive experiences. Unity's flexibility and scalability allow developers to prototype quickly, iterate efficiently, and deploy their projects across multiple platforms with ease. As a result, Unity has become one of the leading game development engines in the industry, powering thousands of games and experiences worldwide. It was an excellent choice for this plugin due to its popularity among game developers and its large community that uses assets from the Unity asset store.

In the upcoming chapters, we will discuss valuable information about music 2.1, how to retrieve it 2.2, and how to use it for our plugin. We will examine the author's proposal for the final product 3 and explore how different components communicate and how the analysis works. Testing is crucial, so we will cover the creation of scenes that demonstrate the plugin's potential 4. Afterward, we will analyze participants' responses to determine how the plugin performs.

Chapter 2

Theory

Analyzing music using computers presents unique challenges. Unlike humans, computers cannot perceive music's rhythm, pitch, and mood. While these elements come naturally to us, computers require various calculations and algorithms to process the complexity of musical signals.

In the following sections, we will dive into the necessary theory for this project to gain a good understanding of the fundamental concepts in music theory, such as beat, pitch, rhythm, and tempo, and decide which ones we want to gather from the music, then we will talk about how the data is gathered from the signal. We will examine how the Fast Fourier Transform (FFT) is utilized to retrieve the data. We will look into procedural generation and how it is used in games, and finally, we will look into the Unity game engine and how it works with audio.

2.1 Fundamentals Concepts

Understanding fundamental music concepts is helpful for effectively analyzing the gathered data from audio signals. In this section, we will discuss some of the key concepts relevant to this project, such as tempo, beat, and frequency. Rather than delving into the traditional music perspective, we will focus on more relevant numerical representations and implications.

2.1.1 Tempo and Beat

Tempo refers to the speed or pace at which music is played, measured in beats per minute (BPM) [12]. It dictates the overall rhythm and energy of a piece. The beat is a regular, repeating pulse that underlies a musical pattern. Humans naturally synchronize movement to the beat, which is invaluable for our game because it enhances the user experience and immersion.

In Western classical music, the tempo is either written in BPM for more precise indication or left to the conductor to specify the tempo by just describing it in Italian words. For example, *Grave* describes a very slow song, *Moderato* describes a moderate tempo, and *Prestissimo* a very, very fast tempo [12]. You can see the tempo marking in both forms in Fig 2.1.

2.1.2 Time Signature

In Western music, a time signature also referred to as a meter signature, notation specifies the number of note values of a particular type within each measure (bar). In music notation, it appears as two stacked numerals, as seen in Fig. 2.1. Time signatures are categorized as simple (grouping note values in pairs like $\frac{2}{4}$, $\frac{3}{4}$, $\frac{4}{4}$) or compound (grouping in threes like $\frac{6}{8}$, $\frac{9}{8}$, $\frac{12}{8}$), with less-common ones representing complex, mixed, additive, or irrational meters [13].

The upper numeral in a time signature represents the number of note values per bar, while the lower numeral indicates the type of note being counted. Understanding the song's rhythm can help during beat estimation or beat generation. For example, if we create a heavy rhythm game, we can create a metronome-like effect with the knowledge of time signature, meaning that we can emphasize the first beat.

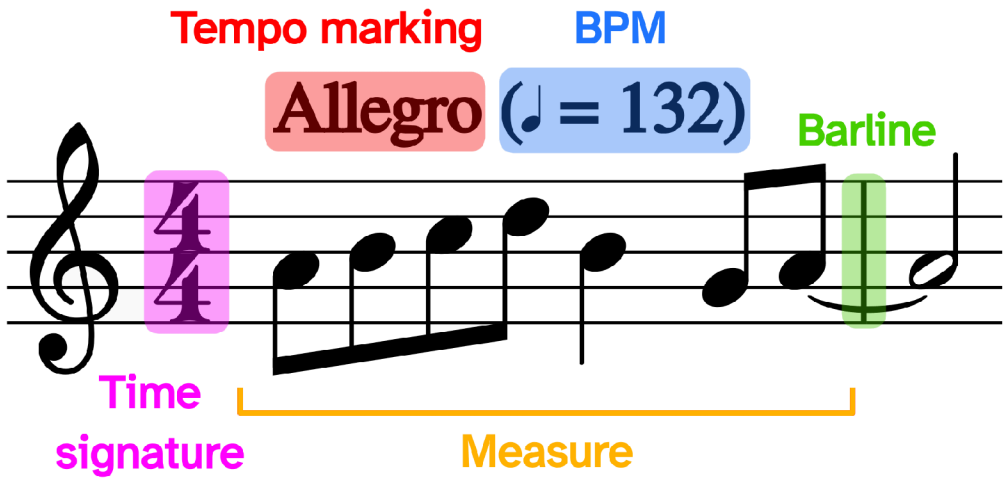
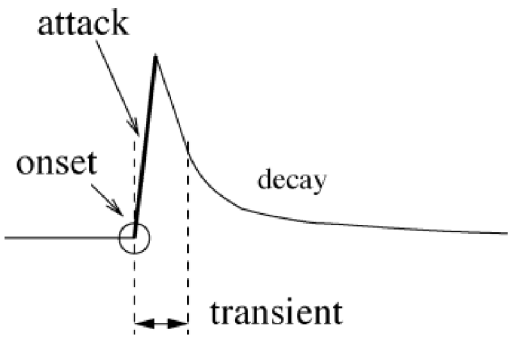


Figure 2.1: Note notation [14]

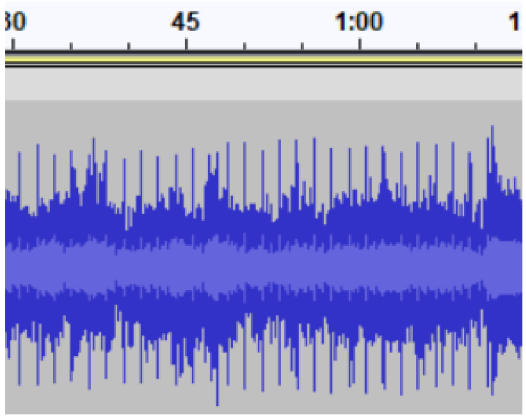
2.1.3 Onset

The onset is the beginning of a musical note or sound, characterized by the rise in amplitude from zero to an initial peak. It is important for beat estimation and rhythm analysis, serving as a reference point for identifying rhythmic patterns within the music. We can see the onsets in Fig. 2.2b as general amplitude spikes or in Fig. 2.2c as specific frequency spikes.

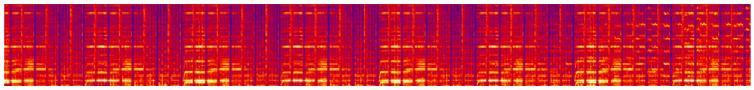
In Fig. 2.2a, we can see the different parts of the note. Distinguishing between these is crucial because different applications have different needs. Transient represents a short interval during which the signal rapidly evolves in some complex or relatively unpredictable way. As we said, the onset marks the start of the note. More precisely, it marks the beginning of the transient or the earliest point at which the transient can be reliably detected. Attack is the time interval during which the amplitude rises. Additionally, the release of sustained sounds can also be perceived as a transient period [3].



(a) Different parts of a single note in ideal case [3]



(b) Possible onsets seen in waveform as vertical spikes



(c) Possible onsets seen in spectrogram shown with brighter color

Figure 2.2: Figures showing onsets

2.1.4 Frequency and Pitch

Frequency represents the speed of vibration in a sound wave, measured in Hertz (Hz), determining its pitch or perceived musical tone. Detecting frequency variations is essential for identifying melodies and harmonies within the music. Pitch refers to the position of a single sound within the complete range of sounds.

The frequency of a sound wave determines the number of vibrations that occur per unit of time. As a result, sounds with higher frequencies are perceived as having a higher pitch and a more distinct, sharper quality. In other words, the higher the frequency, the higher the pitch of the sound.

Unlike some animals, human ears detect frequencies between 20 and 20,000 Hz (assuming optimal conditions) [10]; anything beyond (ultrasounds or below (infrasounds) this range is imperceptible to us. These audible frequencies are further divided into smaller ranges. Different frequency ranges correspond to distinct tonal qualities and characteristics [1]:

- **Sub Bass (20 - 60 Hz):** Felt more than heard, contributes to the overall richness and depth of the sound.
- **Bass (60 - 250 Hz):** Determines the thickness or thinness of the sound, providing the foundational notes of rhythm.
- **Low Midrange (250 - 500 Hz):** Contains low-order harmonics and contributes to the bass presence in the mix.
- **Midrange (500 - 2000 Hz):** Determines the prominence of an instrument in the mix, influencing its perceived clarity and definition.

- **Upper Midrange (2000 - 4000 Hz):** Emphasizes the attack on percussive and rhythmic instruments, adding presence and impact.
- **Presence (4000 - 6000 Hz):** Enhances the clarity and definition of sound, often adjusted using treble controls in home stereos.
- **Brilliance (6000 - 20000 Hz):** Contains harmonics that contribute to the overall brightness and shimmer of the sound.

An important note to remember is the relationship between frequency and pitch. Each octave represents a doubling of frequency. This means that if we want to calculate a lower octave of a note, we divide its frequency by 2; if we want a higher frequency, we multiply it by 2. We can utilize this knowledge to determine what note has been played quickly. For example, if we know that note C_1 has a frequency of 32.703 Hz, we can calculate any other octave of this note and know approximately where we can look for that particular note.

2.1.5 Psychoacoustics

Psychoacoustics researches how humans perceive sound. It is an important field that helps aid in the development of communication. It combines how our bodies receive sound (physiology of sound) and how our brains interpret it (psychology of sound). These disciplines provide an understanding of people's different reactions to sounds [10]. These insights are important because sound is essential in many fields, such as communications devices, music and film production, and even the game industry. Sounds are very diverse. The main elements contributing to this diversity are intensity, pitch, and tone. We already talked about the pitch in the previous chapter.

Intensity is represented by amplitude, which is a measure of energy. It is measured in decibels, and it determines the loudness of sound [8]. Human ears are more sensitive to higher frequencies, which means that we may perceive them as louder, though the intensity is independent of human perception. This can be seen in Fig. 2.3. Equal loudness contours illustrate how the human ear perceives sound at different frequencies. The figure shows that the ear is most sensitive to frequencies in the range from 1 to 5 kHz. Each curve corresponds to a 10 dB increase using the 1 kHz tone as a reference point.

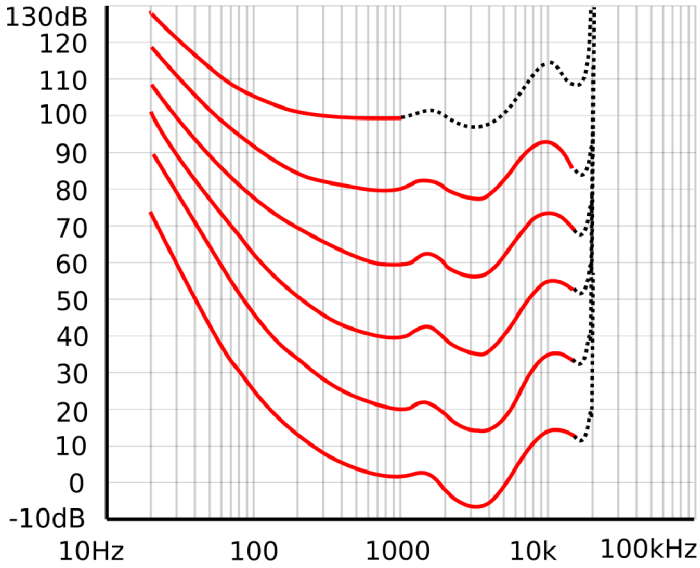


Figure 2.3: The equal loudness contours. [15]

Tone quality is influenced by the combination of different frequencies, which gives the sound its unique characteristics. Even though we play the same pitch, the sound is different when we play two different instruments, for example, guitar and saxophone. When a source vibrates, it vibrates with multiple frequencies at once. The quality of sound is influenced by a mixture of various frequencies of sound waves. We mainly hear the main pitch called fundamental, which is the lowest from the mixture. Higher frequencies are called overtones, and those vibrating in whole-number multiples are called harmonics [8]. This is why the guitar and saxophone sound different; the material from which the instrument is created, the playing technique, and the generation of the sound wave (blowing in saxophone or strumming a guitar string) all affect these frequencies and change the tone.

Both music and noise are types of sounds. Usually, people consider music as pleasant, and noise as unpleasant. However, this definition can be subjective because someone practicing the violin could sound terrible. There are three properties that the sound must have: to be musical to classify sounds. First, it must have an identifiable pitch. Second, it must have a good-quality tone that sounds pleasing. Third, it must have a repeating pattern or rhythm to be music. On the other hand, noise has no identifiable pitch, no pleasing tone, and no steady rhythm [8].

This study is used extensively in the game industry, from understanding how humans utilize sound to create a realistic environment to using specific sounds with different indicators (for example, roughness, sharpness, and loudness) to inform the player about danger or interesting areas.

2.2 Audio Signal Processing

Audio signal processing is a subfield of signal processing involving electronic manipulation of audio signals. Audio signals are representations of sound that are in digital format, a series of binary numbers. This digitization process, known as sampling, captures discrete snapshots of the sound wave's amplitude at regular intervals.

2.2.1 Domains

Audio signals are commonly visualized as waveforms in the time domain, as seen in Fig. 2.4, illustrating the variations of amplitude over time [5]. While this representation offers a basic understanding of the time-related characteristics, such as changes in volume, potential peaks, or intensity, it provides limited insight into its underlying spectral characteristics.

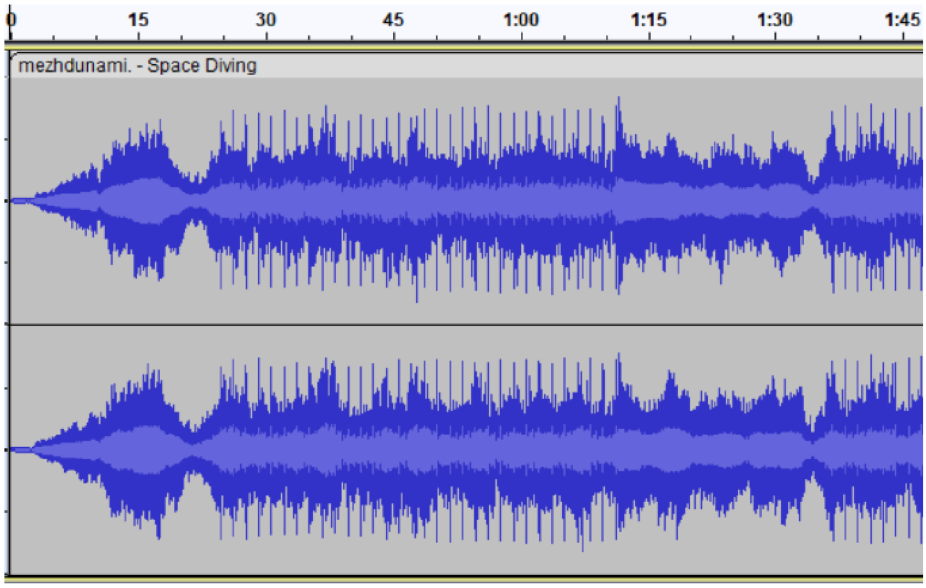


Figure 2.4: Visualization of an audio waveform in the time domain from AudaCity. In stereo, the x-axis is time, and the y-axis is amplitude.

We need to use signal processing techniques to transform the signal into a frequency domain to extract more information from the audio data [5]. The frequency domain reveals the signal's spectral composition, which provides us with valuable insights into its frequency components and distribution. Within this domain, the spectrum represents the magnitudes of frequencies present in the signal at a specific point or range of time, providing a detailed view of its frequency content 2.5.

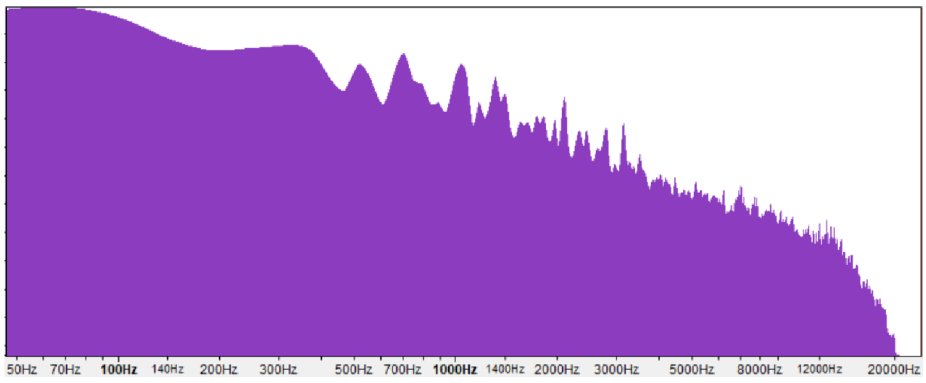


Figure 2.5: Frequency spectrum from AudaCity. The x-axis is frequency, and the y-axis is magnitude.

Analyzing the frequency domain, we can detect dominant frequencies at a given time, providing information about the musical notes or percussive hits present in the audio signal.

2.2.2 Fourier Transform

The Fourier Transform (FT) [4] is an essential mathematical tool used to analyze frequency domain signals. It breaks down a continuous-time signal into its constituent frequencies and provides insights into the signal's frequency content. Mathematically, the continuous Fourier Transform $S(f)$ of a continuous-time signal $x(t)$ is given by the following equation:

$$S(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt$$

This equation represents the integral over all time of the product of the signal $x(t)$ and a complex exponential function $e^{-j2\pi ft}$, where f represents frequency in Hertz. The FT provides a continuous representation of the signal's frequency spectrum, making it suitable for analyzing continuous-time signals.

2.2.3 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) [4] is a mathematical technique used to analyze the frequency content of finite sequences of samples in discrete-time signals. It is the discrete counterpart of the FT and provides a discrete representation of the signal's frequency content. The DFT is computed by finding the Fourier coefficients of the sequence of samples. Mathematically, the DFT $X(k)$ of a discrete-time signal $x(n)$ can be represented as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}$$

Here, N represents the number of samples in the sequence, $x(n)$ represents the discrete signal samples, and k represents the frequency index. The DFT transforms the discrete signal from the time domain to the frequency domain, enabling analysis of discrete-time signals in terms of their frequency components.

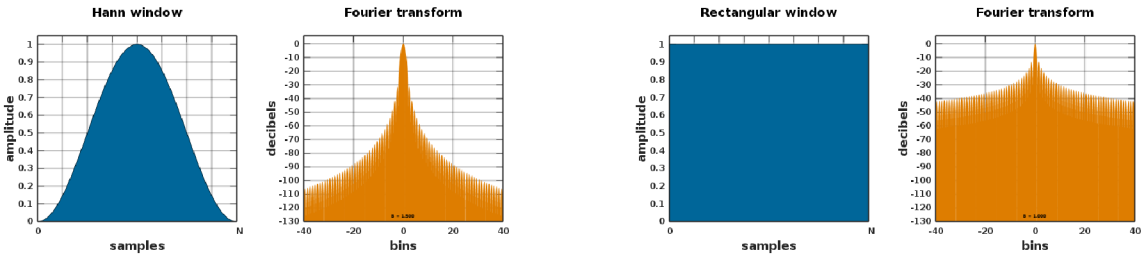
2.2.4 Fast Fourier Transform

The Fast Fourier Transform (FFT) [4] is a powerful algorithm that is used to calculate the DFT (Discrete Fourier Transform). It reduces the computational complexity compared to the direct computation of the DFT, making it practical for real-world applications. The FFT algorithm is based on the divide-and-conquer principle, breaking down the DFT computation into smaller sub-problems. This algorithm enables the rapid computation of the frequency spectrum of a signal, making it useful for efficient signal-processing tasks such as filtering, spectral analysis, and modulation. Due to its speed and efficiency, the FFT algorithm is widely used in various fields, including digital signal processing, communications, and audio processing, to compute the frequency content of signals.

2.2.5 Windowing Techniques

After decomposing the audio signal into its frequency components using the Fast Fourier Transform (FFT), we encounter a common problem: spectral leakage. Spectral leakage means energy at one frequency „leaks“ into adjacent frequency bins, resulting in inaccurate and distorted frequency analysis. Shorter segments of audio signals are more prone to cause this phenomenon.

Fortunately, we do not have to fight this ourselves, and we can employ windowing techniques to improve the accuracy of our analysis. Windowing means segmenting the audio signal into shorter overlapping frames, called windows, each of which is multiplied by a window function. Various window functions are available, each with a different outcome and suitable for different use cases [11]. For example, Rectangular window or Hanning, Hamming, and Blackman windows. The Hanning window was chosen for this project because it worked well during testing.



(a) Hann (Hanning) window function

(b) Rectangle window function

Figure 2.6: Examples of windowing functions [16]

Fig. 2.6 shows the Hanning window, which we chose for this project, and the Rectangular or box window, which is one of the simplest ones. The figures show how the functions affect the signal.

The Rectangular window 2.6b segments the signal into a box-like shape with equal-sized frames. The amplitude within each segment results in sudden transitions at the edges of the window; therefore, it is less effective at mitigating spectral leakage compared to other window functions [11].

The Hanning window 2.6a has the shape of a raised cosine. In comparison to the Rectangular window, the Hanning window gradually decreases the amplitude towards the edges of each windowed segment. This creates a smooth transition and reduces spectral leakage, leading to more precise frequency analysis. As a result, sudden changes are minimized, and the accuracy of the analysis is improved [11].

2.2.6 Onset Detection

The last but not least term we need to explain is onset detection. Onset detection detects musical events in an audio signal. As we already know, an onset is the beginning of a musical note or sound. The algorithm analyzes the amplitude envelope of the audio signal's spectral characteristics to detect these events. This technique is often used with a thresholding technique to distinguish peaks from background noise or sustained sounds [3].

Using onset detection can be found in beat estimation, speech recognition, or, in general, sound event detection. In the context of music analysis, onset detection is useful for rhythm analysis, tempo estimation, and identifying musical structure. (Melodies and more)

Onset detection algorithms vary from one to another. The simplest algorithms are based on the amplitude of the signal alone, comparing the amplitudes directly without converting the data to the frequency domain. For example, the onsets can be seen clearly for percussive elements when looking at Fig. 2.4 of the waveform. This could be enough for beat estimation, but we want to detect more than that.

When we convert the data to the frequency domain, we can use algorithms that can capture subtle changes in spectral characteristics that may not be apparent in the time domain. Sometimes, we can get a spike in amplitude, but no onset happens; we can prevent this when analyzing the signal's frequency content. As we can see in Fig. 2.7, we can detect hi-hat onsets (the grey rectangle) and kick onsets (the green rectangle) by analyzing two different frequency ranges. This would not be possible by analyzing the time domain. If we analyze the whole spectrum, we will get the vertical lines matching the waveform spikes.

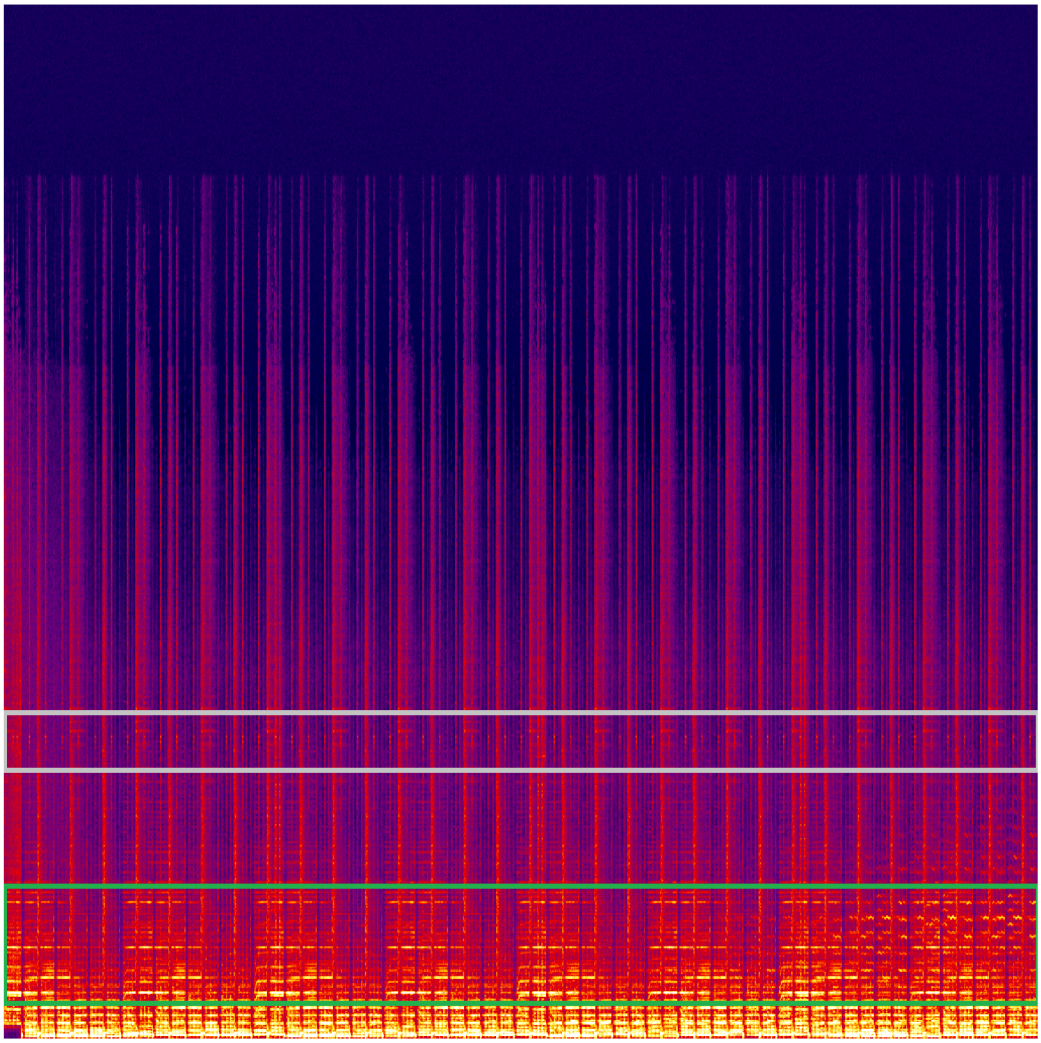


Figure 2.7: Spectrogram of a section of the song. The x-axis represents time, the y-axis represents frequency (20-20000Hz), and the color represents the magnitude of that frequency in that time.

2.3 Procedural Generation in Games

Procedural Generation is a game development technique that automates the creation of game elements using algorithms instead of manual input. This method generates content dynamically during runtime, allowing developers to create vast and diverse game worlds rather than relying on pre-made assets [6].

The success of procedural generation relies on the reliable production of random numbers [6]. These random numbers are the building blocks for generating diverse and unpredictable content. Additionally, the ability to reproduce the same sequence of random numbers through a consistent starting seed and algorithm ensures consistency in generated content across different gameplay sessions.
Procedural generation offers several advantages for game developers, some of them are:

- Infinite Variety - Randomly generating provides infinite content possibilities, increasing game replayability, a core concept in most game genres.
- Saves Time - Generating levels by script is much faster than creating them manually. It can be a tough nut to crack, though.
- Adaptability to the Player - Procedurally generated content can be adapted to the player's skill level and experience by incorporating the „difficulty“ variable, hence creating a more personalized gaming experience.
- Exploring New - Procedural generation encourages players to explore, making the game engaging and fun.

Procedural generation can be used in multiple aspects of the game:

- Procedural Level Generation - Useful for sandbox and roguelike games. It creates a unique experience each time the player progresses through the game.
- Procedural Generation of Enemies and NPCs - Generating different kinds of enemies can create interesting gameplay and challenges for the player.
- Procedural Item and Loot Generation - Useful for creating random rewards for the player, making the playthrough more interesting by incorporating randomness.

Procedural generation is a powerful tool for game developers to automatically generate levels, landscapes, and other game elements, enabling the creation of virtually limitless content. It conveys a sense of never-ending content within limited resources and is a great solution for mitigating production costs and storage and distribution limitations. Procedural generation empowers developers to create immersive gaming experiences with rich and ever-changing environments, enhancing player engagement and replayability.

2.4 Audio in Unity

Given the decision to work in the Unity engine, it is crucial to understand its audio capabilities. Audio is managed through the AudioSource and AudioClip components, which provide us with a good framework for playing and manipulating audio data. Below, we will check out the components and other functionalities that are interesting to our objective.

2.4.1 AudioSource and AudioClip

The AudioSource [2] component serves as a controller for playing audio clips (AudioClip component) and offers parameters to adjust playback settings such as volume, pitch, and spatial blend. While it is useful for modifying the audio clip itself, it isn't useful for our purpose.

The Audioclip [**?**], on the other hand, represents an audio asset that AudioSource can play. It represents the music with all the necessary data we need to analyze.

Unity seamlessly converts audio files such as .mp3 or .wav to AudioClip format, so we don't have to worry about that.

2.4.2 AudioSource.GetSpectrumData

This method computes the audio signal's frequency spectrum using the FFT algorithm [**?**]. Developers can extract spectral features by analyzing the frequency components returned. It is mostly used for tasks such as audio visualization, frequency-based effects processing, and onset detection.

2.4.3 AudioSource.GetOutputData

GetOutputData [**?**] retrieves raw waveform data directly from the AudioSource, allowing for real-time audio signal analysis. It returns raw audio samples that developers can use to perform signal processing tasks such as visualization of waveforms and manipulation of audio effects.

Both GetSpectrumData and GetOutputData provide real-time data chunks. However, if we want to preprocess the audio, we need alternative methods.

2.4.4 AudioClip.GetData

GetData [**?**], on the other hand, is an AudioClip method that returns sample data for the entire song at once. This feature allows developers to preprocess the song, providing flexibility to apply various algorithms as needed.

Chapter 3

Plugin Proposal

Now we know that tempo and melodies are interesting features of the song, we know that we can find them using FFT and Onset Detection. We mentioned data retrieval options in Unity, such as AudioSource.GetSpectrumData or AudioClip.GetData. Our implementation of the onset detection is inspired by the algorithms in [9]. The article clearly explains the implementation of onset detection and offers both preprocessing and real-time implementation.

Onset detection is a great tool for estimating tempo and detecting melodies in a song, so its placement at the center of this plugin is perfect.

Of course, we cannot forget the song's other useful information, like the amplitude, which can be used in dynamic light modulation or atmospheric effects. The spectral centroid is another interesting piece of information. It indicates where the center of mass of the spectrum is located, and it is connected to the impression of a sound's brightness.

3.1 Plugin Architecture

Before we move on to the implementation stage, it is helpful to understand the problem and consider the data flow and communication between parts. The plugin's main component will be ClipControler, which needs to be attached to an object with AudioSource. See Fig. 3.1 for the schematic overview of the proposal. The data flow goes as follows:

- **Step 1 ClipController** retrieves clip data and sends them to **DSPLib** for FFT analysis.
- **Step 2 DSPLib** performs analysis on a chunk and returns average amplitude and spectral data back to ClipController, simultaneously sending the spectral data to FluxAnalysis.
- **Step 3 FluxAnalysis** performs OnsetDetection and returns SpectralFluxInfo into ClipController.
- **Step 4** After all the data is preprocessed, **ClipController** checks for peaks in the current time and updates the public values like average amplitude and spectrum.
- **Step 5 Components** can retrieve the data directly from **ClipController** or by subscribing to the event and waiting for invocation from the **EventManager**.

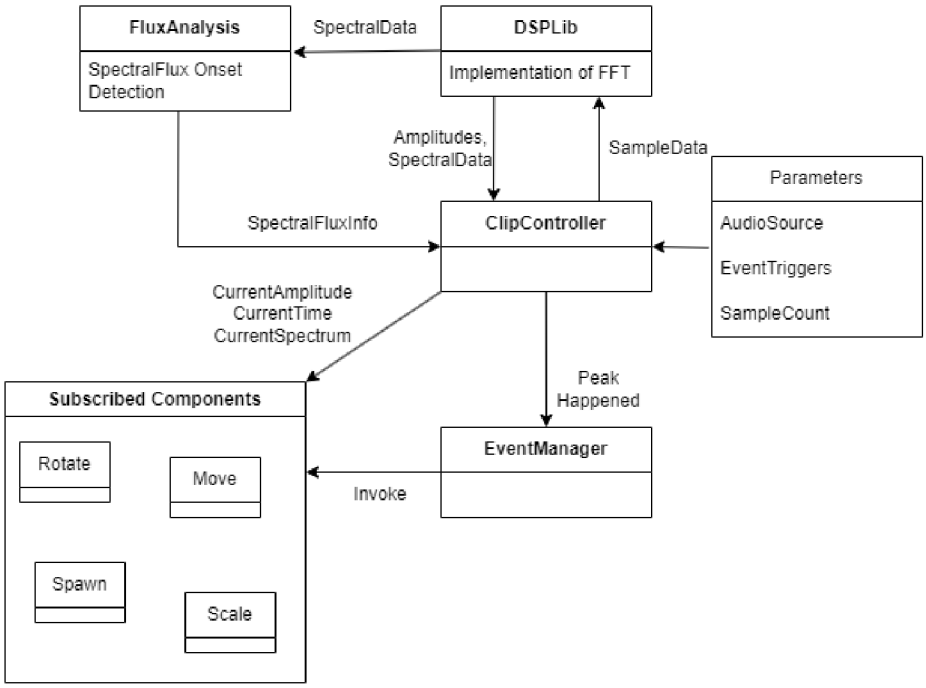


Figure 3.1: Architecture of the proposed plugin

3.2 Analysis

To start, we need to gather the data. As previously discussed, Unity affords multiple choices for data retrieval: real-time analysis via GetSpectrumData and GetOutputData or preprocessing using GetData. While real-time analysis offers immediacy, the preprocessing grants enhanced flexibility and the ability to anticipate future song dynamics.

3.2.1 Data Management

In this project's infrastructure are two key data structures: SpectralFluxInfo, which keeps data from our spectral flux analysis, and Parsed Clip, a class made for managing multiple information about the clip attributes. A custom DataWriter class has been developed to serialize data into JSON format to ensure data persistence across runtimes. This allows users to run the same song multiple times with just one analysis.

Event Customization

Furthermore, the plugin contains a versatile event customization feature. Users have the opportunity to specify frequency ranges and thresholds and assign unique identifiers to each event. This functionality allows for adaptability and empowers users to experiment and modify analysis to their specific requirements.

Event triggers are saved as scriptable objects so the data persists. A custom editor was developed to manage this scriptable object. In addition to creating event triggers, the user can edit them and does not have to worry about naming two different events with the same name because checks are implemented to prevent that from happening.

To help experiment and not leave the user guessing the frequencies, there is a generated grid with musical notes from the note C_0 at 16.35 Hz up to B_7 at 3951.36 Hz. This ensures that a developer with the knowledge of the note range of a melody in a song will have quick access to that frequency, and if he does not have the knowledge, this note grid makes it less complicated to try different things out.

3.2.2 Clip Analysis

We have structures to keep our analyzed data in; we can almost reach the center of our analysis, but we cannot forget about the FFT. Because Unity does not have its implementation of FFT available, we need to either create our own or find a solution. In this project, we will use the same library that the chosen article mentioned, called DSPLib [7].

To perform FFT, we need necessary clip data, such as length and number of samples. We also need to choose a sample count, which is the size of the frame we are going to analyze. The sample count has to be a power of 2. Usually, it is 512 or 1024, which is sufficient. The more samples we want to have in the frame, the finer the frequency resolution we get; of course, that means it will take longer to compute, and we may not want that. 1024 samples had great results, and the analysis time was acceptable, but the user may want something different, and it is good practice to serialize these modifiable variables. Algorithm 1 shows the pseudo-code of the ProcessNewClip function, which gathers data from the clip and calls another function that runs the analysis.

Algorithm 1: ProcessNewClip

Input: AudioCLip newClip
Output: ParsedClip parsedClip

1: $ClearVariablesBeforeNewSong()$
2: $numOfChanels = newClip.channels$
3: $totalSamples = newClip.samples$
4: $clipLength = newClip.length$
5: $multiChannelSamples = newfloat[totalSamplesLength * numberOfChanels]$
6: $newClip.GetData(multiChannelSamples, 0)$
7: $awaitParseClipData(multiChannelSamples)$
8: $parsedClip = newParsedClip()$
9: **return** $parsedClip$

One thing we need to think about is the number of channels. Is the song stereo or mono? We don't know what song we will analyze in the future, so we need to plan accordingly. The GetData function returns raw sample data of the song, but for each channel, a stereo song will have twice as much data as a mono song. Simply creating an array of size **totalSempleLength * numberOfChanels** will give us enough room to get all the data, as seen in algorithm 1.

Before we run the FFT library, we need to combine the channel data. In the case of stereo, we could run the FFT on the right and left channels separately, but it does not have many advantages if we don't know 100% that the song will be stereo. Combining the samples is simple; the array we got from the GetData function always returns samples in the left, right, left, and right order, so we know that index 0 and 1 are the same time-wise, but one is left and the other right. Simply calculating the average of those samples will give us a combined array on which we can now perform the FFT. Algorithm 2 shows the combination of the channels.

Algorithm 2: CombineChannels

Input: float[] clipSamples
Output: float[] combinedSamples

```
1:  preProcesedSamples = newfloat[totalSamplesLength]
2:  numProcessed = 0
3:  combinedChannelAverage = 0.0f
4:  for i = 0 to multiChannelSamples.Length do
5:      combinedChannelAverage+ = multiChannelSamples[i]
6:      if (i + 1) % numOfChanels is 0 then
7:          preProcesedSamples[numProcessed] =
              combinedChannelAverage / numOfChanels
8:          numProcessed + +
9:          combinedChannelAverage = 0.0f
10:     end if
11: end for
12: return preProcesedSamples
```

The FFT analysis is executed iteratively across the entire audio clip, with each iteration processing a window with a specified size of sample count. The number of iterations is calculated by dividing the length of **combined samples** by **sample count**. We utilize the library for each iteration to convert the samples into a frequency domain.

This allows us to save the average amplitude, spectral centroid, and the whole spectrum for future use. After that, we send the spectrum data to our Spectral Flux Analyzer. For a better understanding, please look at the algorithm 3 of the ParseClipData.

Algorithm 3: ParseClipData

Input: (*clipSamples*)

```
1:  preProcessedSamples = CombineChannels(clipSamples)
2:  iterations = preProcesedSamples.Length/sampleCount
3:  fourier = new FFT()
4:  sampleWindow = newdouble[sampleCount]
5:  for i = 0 to iterations do
6:      sampleWindow =
          CopyNextSamplesIntoArray(prePRocessedSamples)
7:      ApplyFFTWindow()
8:      ExecuteFFT()
9:      SaveAverageAmplitude()
10:     SaveSpectralCentroid()
11:     SaveSpectrumData()
12:     currentSongTime = getTimeFromIndex(i) * sampleCount
13:     for trigger in triggers do
14:         AnalyzeSpectrum(scaledFFTSpectrum, currentSongTime)
15:     end for
16: end for
```

As you can see from the algorithm 3, we keep track of the song's current time. It is crucial to determine our position within the song, as this allows us to easily retrieve the corresponding data by calculating the index based on the song time and accessing that index in the list of data.

The time calculation from the index is based on the audio clip's sample rate, which represents the number of samples recorded per second. Dividing one by the sample rate gives us time duration per sample, and multiplying it by the index of the iteration returns the total time elapsed up to that sample. Multiplying it by the sample count provides the total time duration covered by all the samples processed up to that index.

The reverse calculation goes as follows: First, the length per sample is computed by dividing the total length of the audio clip by the total number of samples, which gives the duration of each sample. Then, it divides the current time by the length per sample to determine how many samples have elapsed up to that time. The result is the corresponding sample index at the given time, rounded down to the nearest integer index.

The logic behind this function is based on the audio clip's sample rate, which represents the number of samples recorded per second. To calculate the time value corresponding to a sample index, the function divides the index by the sample rate (1.0f / sampleRate), resulting in the time duration per sample. Multiplying this by the index gives the total time elapsed up to that sample

3.2.3 Onset Detection using Spectral Flux

The Spectral Flux Analyzer is one instance with variables keeping track of data for each event. The main function AnalyzeSpectrum is called parallelly from Clip Analysis 3.2.2 after the FFT analysis, for each event trigger there is. Now, we are at the core of the plugin. We already know what onset detection is. Next, we will talk about spectral flux.

Spectral flux or spectral difference is the difference between one frame's power spectrum and its previous one's power spectrum. The power spectrum represents the magnitude of the spectrum. Here came the deciding point for choosing the gathering method. The analysis would function well with real-time data collection but might lag slightly behind the actual song. To mitigate this, we decided to use GetData and preprocess the song. This approach could potentially make a loading screen before a game level but have almost no delay regarding the song and data. The analysis keeps track of 2 spectrums, current and previous.

After we calculate the spectral flux, we rectify it; there is no point in keeping track of negative numbers, which would mean nothing important. Next, we need to calculate the threshold, which will tell us if the peak happened. If the spectral flux is bigger than the threshold, we have a peak.

We could use a static threshold, but it would not be precise, and it is not worth it to save some computer power for it. So, how do we make a dynamic threshold? It is quite simple. First, we must decide how many spectral fluxes we want to compare. This project has 50 because, after some testing, it had great results. So, we calculate an average of the 50 spectral fluxes. We need to start the calculation of at least half of the window size into the spectrums so we can calculate the dynamic threshold. Then, we need a threshold multiplier. It is just a parameter that says how sensitive it should be.

Then, we calculate the pruned spectral flux, which is just the spectral flux minus the threshold. Then, we check for a peak. If the current pruned spectral flux is higher than the future one and the previous one, we have a peak.

Till now, it was just following the article [9] with some changes to the data types to suit mine infrastructure. Now, we need to extend this algorithm for the ranges we talked about previously. It is as simple as limiting the calculation of rectified spectral flux. By simply calculating the spectral flux in a given range, we look for changes in that area. Check the pseudo-code of the algorithm 4 for a better understanding of the flow; for simplicity, the pseudo-code is just the core onset detection using spectral flux.

Algorithm 4: AnalyzeSpectrum

Input: $(newSpectrum, time)$

```
1:  GetNewSpectrum(newSpectrum, eventName)
2:  currentInfo.time = time
3:  currentInfo.spectralFlux = CalculateRectifiedSpectralFlux()
4:  spectralFluxSamples.Add(currentInfo)
5:  if spectralFluxSamples.Count >= thresholdWindowSize then
6:      spectralFluxSamples[currentIndex].threshold =
          CalculateDynamicThreshold()
7:      spectralFluxSamples[currentIndex].prunedSpectralFlux =
          GetPrunedSpectralFlux()
8:      if isPeak(currentIndex − 1) then
9:          spectralFluxSamples[currentIndex − 1].isPeak = true
10:     end if
11:     currentIndex + +
12: end if
```

3.2.4 Beat Detection

Now, if we have the peaks, we should be able to know the tempo of a song. Well, yes and no. By expanding the base algorithm on specific ranges, we have higher chances, but it still is a swing and a miss. Beat detection is very tricky and comes with a lot of variables. Some of them are mixing of the song, genre, and noise. The ideal song would have clear percussion elements, such as kick and snare, that would be seen in the spectrum, but we don't know what song users are going to use. That's why the beat generation was added, to battle the uncertainty of the song quality and poor mixing. If the user knows at least the tempo of the song (which he can get from third-party applications).

3.3 Functionality

We retrieved all interesting data from the audio clip; technically, that should be all. Now, users can access the data from the main component named ClipController, which needs an Event Triggers scriptable object and AudioSource to run the analysis on a clip. However, we are going to create some functionality for the users, such as a custom event system, beat generation, and some components.

3.3.1 Event System

The unity event system is powerful and very useful, but after introducing custom events 3.2.1 to this plugin, a custom event manager was needed with the event system more catered to our needs. So, an Event Manager is created automatically and instantiated by the Clip Controller. It is singleton to ensure the event system won't get messy. The event system consists of a dictionary where the key is the event name, and the value is a list of actions with the SpectralFLuxInfo parameter. This helps quick access to all event subscribers. The Event Manager has a custom Subscribe and Unsubscribe function to keep track of subscribers. Other than that, it works just like a basic event system. Clip Controller checks if an event trigger occurs, and if yes, it calls the Invoke method of Event Manager, which then invokes all Actions of subscribers to that specific event. See Fig 3.2 for a better understanding.

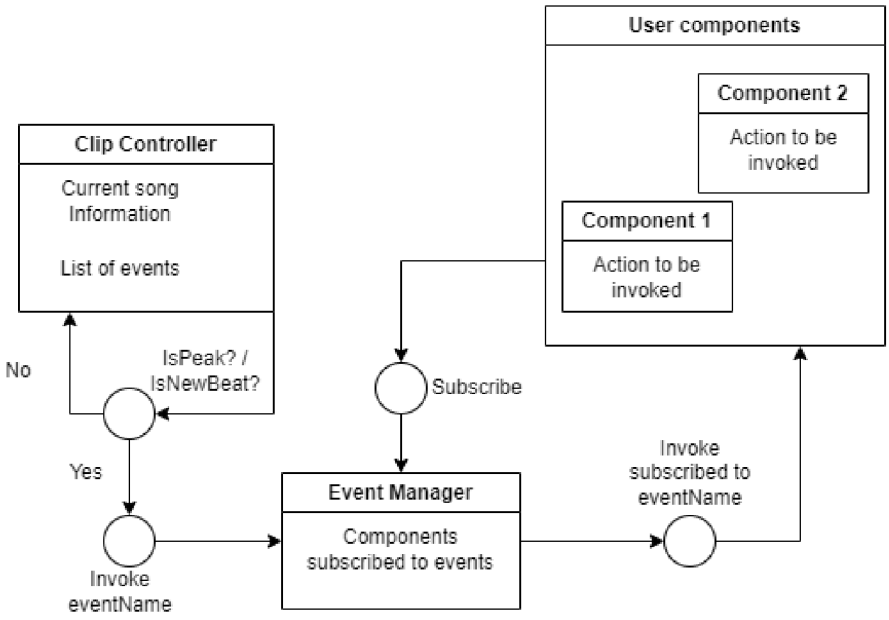


Figure 3.2: Simplified diagram of the event system.

3.3.2 Beat Generation

As previously mentioned, beat detection is tricky. Beat generation, on the other hand, is pure math. We need to get the song's BPM so that we can calculate the interval between individual beats. $\frac{60}{BPM} = interval$

For example, if the song has a tempo of 120 BPM, it means that there will be 0.5 seconds between each beat. Now, we just need to invoke the beat event. To that, we need to keep track of time; we are going to use separate functionality that our analysis uses. Synchronizing object manipulation to beat is very delicate, which is why we need accuracy as high as possible. Using AudioSettings.dspTime, which returns the current time of the audio system, is much more precise than simply using AudioSource.time because it is based on the actual number of samples the audio system processes [**?**]. Then, we just keep track of the start of the audio and always calculate the difference between the current DSP time and the start time.

To broaden the beat events, an enumerate of different times of beat is created; the user can now specify on what beat to listen to the event. For example, the user can choose to trigger every beat, just the second beat, even beats, or odd beats. We introduce time signatures to give the user even more flexibility and possibilities. A s we know from 2.1.2 time signature specifies how many beats are in a bar. This beat generation is very simple and expects only $\frac{4}{4}$ or $\frac{3}{4}$ time signatures, which are the most common ones. This allows the user to create accents, for example, on the first beat of a bar, or create an environment that is more responsive to the song.

3.4 Components

Now, game developers and other users have access to all data and systems, and it is up to them to create what they want. Components were developed for their smoother creative process and quicker prototyping of different scenes. Some are more complex than others, all based on the information retrieved from the song.

3.4.1 Beat-based Components

The beat-based components are simple, and only the main properties of objects, such as scale, positions, and rotation, are modified. These components are simple and just keep going back and forth between specified values in specified beat intervals. Other components are modifying the intensity of a light or enabling and disabling an object.

One of the more complex components is checkpoint movement. It works by creating a list of transforms that act as checkpoints through which the object moves. The user can specify how many beats it takes the object to reach a checkpoint so that it moves more quickly or slowly.

3.4.2 Amplitude Based Components

These components are not based on events; they are based only on the amplitude values of the song. It can create dynamic light or fog. Exactly that was created. A component for dynamic light is created to work with spot and point types of light. It changes light's intensity and area with the changing amplitude. The dynamic fog component works by changing the density of a Unity fog created by their renderer settings; when the music is busier and has a higher amplitude, there is less fog in the scene.

Last but not least is scaling with amplitude, which scales the object by the amplitude in the song currently.

3.4.3 Cinematic Camera

Other than basic, general components, which can be used quickly, we also created a more complex system called a cinematic camera system. It is a custom-made movement script through checkpoints for the camera, designed to synchronize closely with the song's progression. Each checkpoint is defined by position and corresponding timestamp, indicating a specific moment in the song timeline when the camera should reach the checkpoint. Wait time, rotation time, and rotation target were added to give the user more freedom. It creates even more possibilities and mimics an animation clip but with easier song synchronization.

3.5 Quality of Life

During the development and testing, functionality for solely quality of workflow and convenience was created. This functionality is still relevant for future developers who would work with the plugin; that's why we are going to know a little bit about them.

3.5.1 DataWriter

DataWriter is one of the classes that provide the functionality to serialize the ParsedClip into JSON format and save it locally, which provides quicker experimenting with things not related to the analysis, such as beat events. The ClipController has a flag to run the analysis and overwrite the saved file for easy workflow; if we do not want to re-analyze the song, the DataWriter will read it from a file and send it to ClipController to continue the flow of the game.

In case the developer wants to examine the data inside the ParsedClip for a deeper understanding, there is a function to print multiple files with different data for each frequency range; this is because the one file is so large that it is impossible to read anything except the clip's name from it.

3.5.2 Testing Environment

It was essential to see what was happening in the song during the tweaking process. That's why a testing playground was created to let users see the spectrum data of the song and see in which frequency ranges something is happening. This visualization helps to create more precise frequency ranges for our event triggers.

Chapter 4

Testing

Testing is very important in this kind of project; based on the results, we can say whether the plugin has potential or not. We will focus on overall feeling and synchronization with music. The test questions were created using Google Forms.

Two unity scenes were created, each showcasing a different functionality of the plugin. Videos showcasing the important functionality were created and added to the questionnaire. The videos could not be uploaded to YouTube due to copyright reasons. Therefore, readers interested in viewing the videos can watch the forest video and space video by watching them directly from the disk; the structure is described in appendix A.

4.1 Forest Scene

The forest scene was based on a piece from the movie *A Series Of Unfortunate Events* called *The Baudelaire Orphans* created by **Thomas Newman**. This song was chosen because of a pretty melody midway through that reminded us of stars. The piece also has crescendos, which are interesting to visualize. It is an orchestral piece without any percussion elements, which made it a bad choice for a rhythm showcase but great for a more cinematic, artistic experience.

4.1.1 Scene Creation

All the assets are free from the asset store. The decision to use a low-poly forest was made because it worked well for this music piece. The asset comes with the whole scene prepared with rocks, trees, mushrooms, sunflowers, and more, which was not altered in any way. This scene was used and we added procedural sound-based elements to it.

A skybox of the starry night sky created by a third-party tool named 3DTool was added to the scene, which greatly helped to set the right mood. Then, we used the unity fog system to create a feeling of mystery, changed the light to lighter blue to mimic the moon, and the scene was born.

Take a look at Fig. 4.1a to see the whole scene without the fog that would block the view.

4.1.2 Scene Functionality

For functionality showcase, the custom-made checkpoint camera movement 3.4.3 was utilized to move through the scene dynamically. This guarantees that the video captures everything the plugin has to offer. Three checkpoints were created, each focusing on a different functionality.

First Part

The first checkpoint was set to look at a bunch of rocks with mushrooms and sunflowers, seen in Fig. 4.1b, showcasing the functionality of amplitude scaling and light modification.

Second Part

The second checkpoint focused on stars to show what can be done with the Onset detection. Here, we utilized the pretty melody that reminded us of stars. We created two particle systems, one representing the light of a star and the second adding flare. Together, they created a pretty star-like effect.

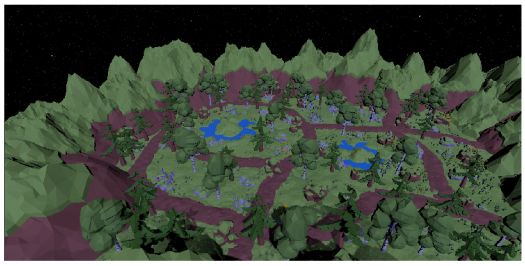
To synchronize it with the piece, a script was created that randomizes the position of the particle systems and emits one particle with a dynamic size of the current amplitude in the camera direction.

Because during the melody, there is a lower arpeggio accompanying the main melody, the decision to show it too was made. We duplicated the particle systems, gave it a different event trigger and color so it is seen better, and a pretty little star dance was done.

The second part can be seen in Fig. 4.1c.

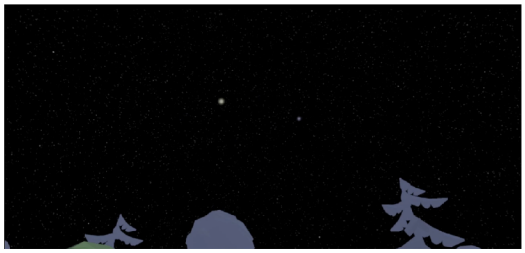
Third Part

The last checkpoint was looking into the forest, seeing clearer when the crescendo happened and seeing how the fog was coming back after it, almost as if the instruments were creating wind blowing the fog away. It can be seen in Fig. 4.1d

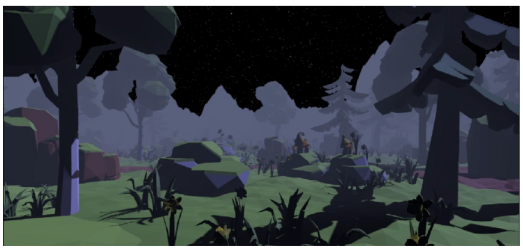


(a) Full scene without fog

(b) First part, mushrooms and sunflowers



(c) Second part, starry sky



(d) Third part, fading fog

Figure 4.1: Figures of the forest scene

4.2 Space Scene

This scene was made to show the beat generation and potential of the plugin to be used in rhythm games. For a song, *Space Diving* by an artist named **mezhdunami.**, which falls into the synth-wave electro genre, was perfect. The song features distinct, percussive elements, evoking images of a game-level performance perfectly suited to this type of music.

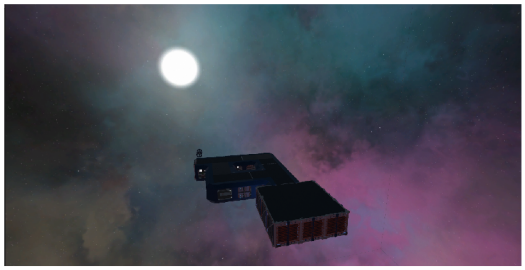
4.2.1 Scene Creation

Again, free assets from the asset store were used. The asset contained a space station scene that was already prepared and just needed our functionality to be complete. We created a player movement script with a rotating camera for the video recording. Then, we added a space skybox, seen in Fig. 4.2a, using the same third-party tool, 3DTool, as in the forest scene to set the space station into space. This scene lacks the post-processing touch that the first scene had, but more importantly, it has to showcase the beat synchronization, which is clearly visible. See Fig. 4.2 to see the space station.

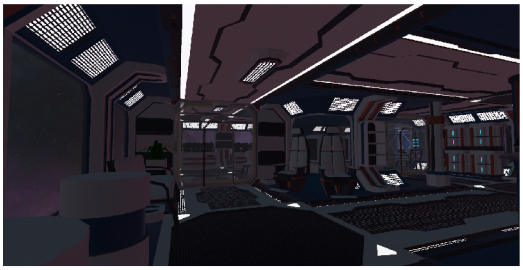
4.2.2 Scene Functionality

Here, all the beat components mentioned in the section 3.4.1 are utilized. Doors that open by rotating, moving from side to side, or disappear completely can be seen in the scene. Checkpoint movement is also utilized on a floating object in space to mimic orbit. The light component was added to showcase not only the gameplay possibilities of the components but also to set the mood. The lights mimic the flickering of an abandoned space station.

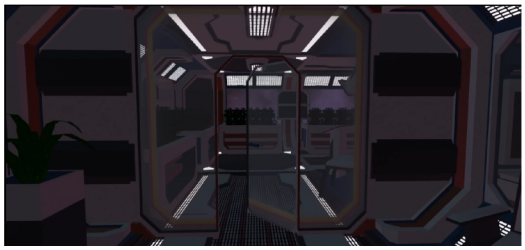
All of these elements were intended to contribute to an engaging level filled with potential puzzles and narratives. Whether it was a success or not is not up to us.



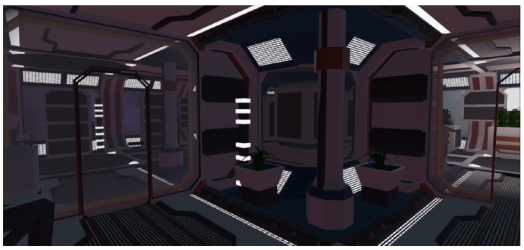
(a) Full scene of the space station



(b) Starting position in the space station



(c) Doors in rotating motion



(d) Another angle of the space station

Figure 4.2: Figures of the space scene

Please refer to Appendix A for picture location for better resolution.

4.3 Results

The respondents were in the 20-25 age range and varied in profession. Totally there were 12 participants. Notably, a quarter of them were not students. Furthermore, only 16.6% of the participants reported having experience with the rhythm game genre, while 41.6% indicated no prior gaming experience whatsoever. Interestingly, one respondent works in the game industry. Additionally, a significant majority of 66.6% had experience with music other than listening, whether playing an instrument or studying music theory.

As we mentioned at the start of this chapter, the main focus is on overall feeling and synchronization with music. The results show two videos with completely different feelings.

4.3.1 Forest Scene

The forest scene had been given a score of 8.8 out of 10 in total. That is very pleasant. The majority of participants, 66.7%, said they did not see any problem with the scene. The other users had multiple different comments, but the most common comment was about the stars in the second part of the video.

Even though there were small problems, it seems that almost every participant could imagine this functionality being utilized in some kind of digital media, be it an advertisement, a video clip, or a trailer. The other participants answered „maybe“, which indicates that the deciding factor would be the execution. Take a look at the questions and results in the Tab. 4.1.

In summary, the scene garnered a very positive response, indicating that the plugin is on the right path.

Questions	**Have you noticed any problems with the audio visualization shown in the video?**	**Would this functionality be good in trailers, video clips, advertisements, or other kinds of digital media?**
Yes	33.3%	83.3%
No	66.7%	-
Maybe	-	16.7%

Table 4.1: Response percentages from questionare

4.3.2 Space Scene

On the other hand, the space scene had a different story. With a total score of 7.5 out of 10 in total, it was not such a success. Even though the majority of 58.3% still answered that they did not see any problem with the synchronization, it was clearly seen that participants who have more experience with rhythm games voted that they had indeed seen problems. The deciding factor was the door movement. The combination of different beat triggers and styles of opening the door created a very confusing environment to wrap our heads around beat synchronization, so it was hard to see if the doors were really on beat. Other than that, there were slight issues with the scene, but these were unrelated to the beat synchronization, so they are irrelevant.

Even though this scene shows bigger problems, after the question if they think that the plugin could be used to create rhythm games, only one participant said „No“; others said „Maybe“ and „Yes“, which again indicates that it will be based on the execution and care for the game. See the experimental results in the Tab. 4.2.

In summary, although this scene did not receive as positive a response as the forest scene, it still got good comments and tips on what to look at and tweak more. Again, the potential stays, but it may need more work.

Questions	**Have you noticed any synchronization or any other problems?**	**Would this kind of beat synchronization precision be enough for a game?**
Yes	41.7%	58.3%
No	58.3%	8.3%
Maybe	-	33.3%

Table 4.2: Response percentages from questionare

Chapter 5

Conclusion

Music is closely tied to us, accompanying us almost everywhere: in the car, in the supermarket, when we watch movies or series, and even when we play games. Utilizing our natural perception of music in different kinds of mediums can create an immersive environment that will enhance our feelings and perception of the music we are listening to.

In this paper, we looked into some of the fundamental concepts 2.1 of music, such as tempo, beat, onset, and frequency. We learned about psychoacoustic and volume. We dived into audio signal processing 2.2 and familiarized ourselves with the time domain and frequency domain; we talked about the Fourier Transform and its use and mentioned the Fast Fourier Transform. We mentioned the benefits of procedural generation, and then we talked about Unity and how it handles audio clips 2.4, and we researched possible data retrieval methods.

After that, we looked at the proposal for a plugin that analyses music playing in a Unity scene's background. We explored the steps of our clip analysis 3.2.2 and the onset detection using spectral flux 3.2.3, discovering the challenges behind beat detection. We familiarized ourselves with our data management and event customization solution, with a custom event system tailored to work with any custom event created. We found a solution for rhythmic elements in games using simple beat generation. Additionally, we discussed all the different components the plugin offers, such as object Transform modification to the rhythm of a song or dynamic fog controlled by the current amplitude of the song. We cannot forget about the custom cinematic camera script that lets users plan the movement of a camera with the song timeline, also giving them the possibility of rotating the camera to an object of their choice, and we mentioned some of the quality of life features in the plugin.

Testing was necessary when everything was created, and we looked into the results. We discussed the creation of the two testing scenes, their motivation, and how they received very different reactions. The forest scene showcasing the dynamic camera and environment received a more positive rating than the space scene showing rhythmic patterns, which were much more prone to inaccuracy. The plugin's potential was there even before testing, but the testing just assured us that it was going the right way.

A potential improvement of the plugin could be running half of the preprocess functionality and then running it while the game runs. This would minimize the time we must wait for the analysis to finish. Adding a list of AudioClips, which would be processed before playing, would help if the scene had multiple songs.

The analysis has ways to improve, and adding machine learning would provide us with even more functionality, like mood detection or genre estimation. This would be a great tool for procedurally generating levels with colors and assets matching the song.

Another direction for the plugin could be into a more cinematic sphere, with custom checkpoint creation functionality that would make the checkpoint system easier to use and custom time stamps into custom-made clip objects.

The beat estimation analysis could be improved to make the beat generation obsolete, which would eliminate another concern from developers about finding the tempo of the song and opening up possibilities for procedural levels with rhythmic elements.

This project has multiple possible ways to evolve, and the way it will continue is just up to us developers interested in music and digital media working closely together with it.

Bibliography

[1] *Mixing Techniques - Audio Spectrum* online. Teach Me Audio, april 2020. Available at: https://www.teachmeaudio.com/mixing/techniques/audio-spectrum. [cit. 2024-04-28].

[2] *Unity Documentation scripting API* online. Unity Technologies, may 2024. Available at: https://docs.unity3d.com/ScriptReference/index.html. [cit. 2024-05-01].

[3] Bello, J.; Daudet, L.; Abdallah, S.; Duxbury, C.; Davies, M. et al. A tutorial on onset detection in music signals. *IEEE Transactions on Speech and Audio Processing*, 2005, vol. 13, no. 5.

[4] Brigham, E. O. and Morrow, R. E. The fast Fourier transform. *IEEE Spectrum*, 1967, vol. 4, no. 12.

[5] Constantinescu, C. and Brad, R. An Overview on Sound Features in Time and Frequency Domain. *International Journal of Advanced Statistics and IT&C for Economics and Life Sciences*, december 2023, vol. 13.

[6] Cox, G. *Procedural Generation of Computer Game Maps* online. Baeldung, march 2024. Available at: https://www.baeldung.com/cs/gameplay-maps-procedural-generation. [cit. 2024-05-1].

[7] Hageman, S. *DSPLib - FFT / DFT Fourier Transform Library for .NET 4* online. CodeProject, june 2016. Available at: https://www.codeproject.com/Articles/1107480/DSPLib-FFT-DFT-Fourier-Transform-Library-for-NET-6. [cit. 2024-04-28].

[8] Iowa State University Center for Nondestructive Evaluation. *Components of Sound* online. Available at: https://www.nde-ed.org/Physics/Sound/components.xhtml. [cit. 2024-05-04].

[9] Jesse. *Algorithmic Beat Mapping in Unity* online. Medium, february 2018. Available at: https://medium.com/@jesse_87798/d4c2c25d2f27. [cit. 2024-04-20].

[10] Minard, A. *DPsychoacoustics: Understanding the Listening Experience* online. ANSYS, july 2023. Available at: https://www.ansys.com/blog/understanding-psychoacoustics. [cit. 2024-04-29].

[11] Prabhu, K. M. M. *Window Functions and Their Applications in Signal Processing.* Taylor & Francis, 2014.

[12] Preis, J. *What is Tempo in Music?* online. Hoffman Academy. Available at: https://www.hoffmanacademy.com/blog/what-is-tempo-in-music/. [cit. 2024-04-29].

[13] The Editors of Encyclopaedia Britannica. *Time signature* online. March 2024. Available at: https://www.britannica.com/art/time-signature. [cit. 2024-05-2].

[14] Wet, R. de. *Music Duration Calculator* https://www.omnicalculator.com/other/music-duration. Accessed on May 02, 2024.

[15] Wikipedia contributors. *Equal-loudness contour* online. 14. march 2024. Available at: https://en.wikipedia.org/wiki/Equal-loudness_contour.

[16] Wikipedia contributors. *Window function* online. April 2024. Available at: https://en.wikipedia.org/wiki/Window_function. [cit. 2024-05-2].

Appendix A

Disk Structure

This appendix describes the structure of the disk. The Multimedia folder contains the videos used in testing and pictures used in this thesis. The latex folder contains the PDF and the source files for this thesis.

In case the reader wants to try the plugin on their machine, the Unity folder contains all the necessary files for it to work. The project runs on Unity version 2021.3.24f1. Simply import the project into Unity and explore the prepared scenes seen in this thesis.

```
/
├── Multimedia/ ..... Folder with multimedia
│   ├── Videos/ ..... Videos used in the testing questionnaire
│   └── Screenshots/ ..... Screenshots of the game
├── Latex/ ..... Folder with latex scripts and pictures
└── ProceduralSoundBasedElements/ ..... Unity project folder
    └── Assets/ ..... Folder with the assets in the Unity project
        ├── Pure Poly/ ..... Asset pack used in ForestScene
        ├── Scenes/ ..... Testing scenes
        ├── Sci-Fi Styled Modular Pack/ ..... Asset pack used in SpaceScene
        ├── ScriptableObjects/ ..... Scriptable objects used in the testing scenes
        ├── Scripts/ ..... Scripts of the plugin
        ├── SkyBox/ ..... PNGs for the skybox
        └── TestingMusic/ ..... Music for testing the plugin
```