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

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 užívateľmi 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
July 29, 2024

Acknowledgements

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

Contents

List of Figures

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 helps developers create captivating environments, fun rhythmic gameplay, and more.

The final plugin is designed for the Unity Game Engine, a powerful, versatile cross-platform game engine 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, valuable information about music in Chapter 2.1, methods for retrieving it in Chapter 2.2, and its application in the plugin will be discussed. The author's proposal for the final product will be examined in Chapter 3, along with an exploration of how different components communicate and how the analysis functions. Given the importance of testing, Chapter 4 will cover the creation of scenes to demonstrate the plugin's potential and provide an analysis of participants' responses to assess the plugin's performance.

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.

The following sections explore the necessary theoretical foundations for this project. Initially, fundamental concepts in music theory, such as beat, pitch, rhythm, and tempo, are reviewed to determine which parameters should be extracted from the music. Afterward, the methods used to gather data from the audio signal are discussed, with a specific focus on the application of the Fast Fourier Transform (FFT) for data retrieval. Other topics mentioned include procedural generation and its applications in game development, as well as the Unity game engine and its audio integration.

2.1 Fundamentals Concepts

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

Different genres often utilize specific time signatures to create their unique rhythmic feels. For example, a waltz typically uses a $\frac{3}{4}$ time signature, giving it a distinct 'one-two-three' rhythm. In contrast, many rock and pop songs use $\frac{4}{4}$, also known as 'common time,' which provides a steady and familiar beat. Understanding the song's rhythm can help during beat estimation or beat generation. For example, to create a heavy rhythm game, a metronome-like effect can be achieved with the knowledge of time signature. This approach emphasizes the first beat using audio-visual cues to keep the player in the flow.

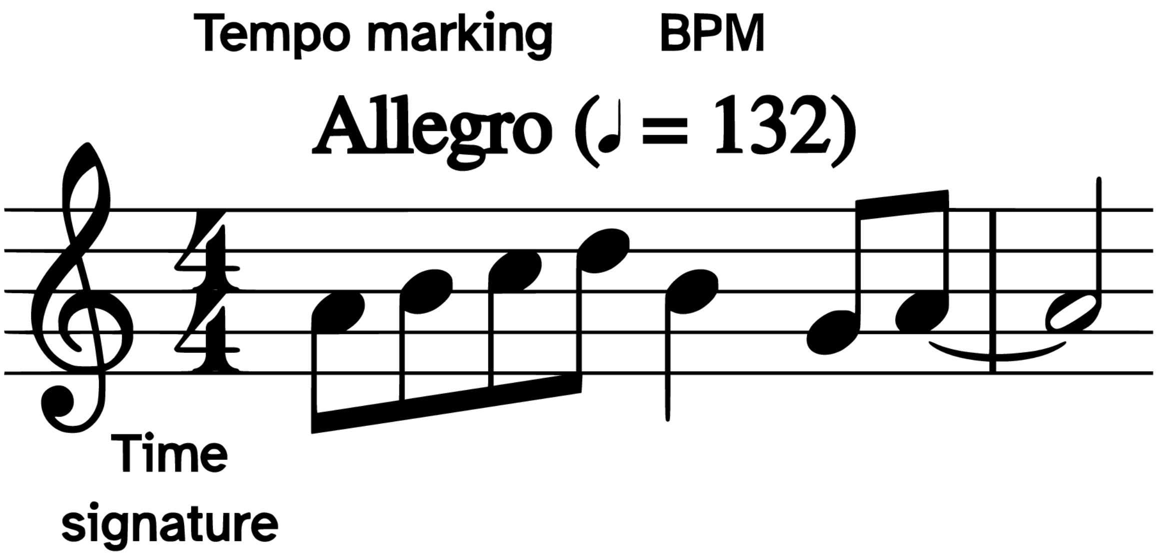
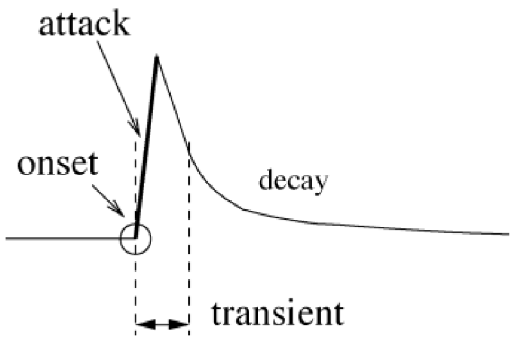


Figure 2.1: Musical notation displaying different components of music. [14]

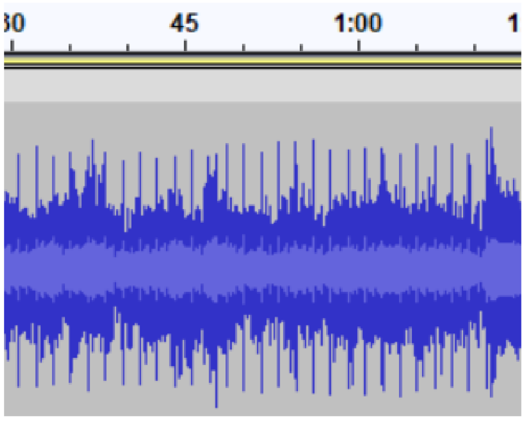
2.1.3 Onset

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. The onsets can be seen in Fig. 2.2b as general amplitude spikes or in Fig. 2.2c as specific frequency spikes.

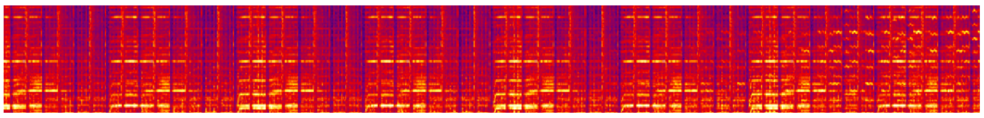
Fig. 2.2a, displays the different parts of the note. Distinguishing between these is crucial because different applications have different needs. Transient represents a short interval during which the signal rapidly evolves in some complex or relatively unpredictable way. As was said, the onset marks 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: Onsets displayed in different domains

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, each with distinct characteristics. 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 to calculate a lower octave of a note, its frequency is divided by 2; for a higher octave, it is multiplied by 2. This knowledge can be utilized to quickly determine which note has been played. For example, knowing that note C_1 has a frequency of 32.703 Hz allows for the calculation of any other octave of this note, providing an approximate location for identifying 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 human bodies receive sound (physiology of sound) and how human 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. The sounds are very diverse. The main elements contributing to this diversity are intensity, pitch, and tone. The pitch was already mentioned in the previous Section 2.1.4.

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 they 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.

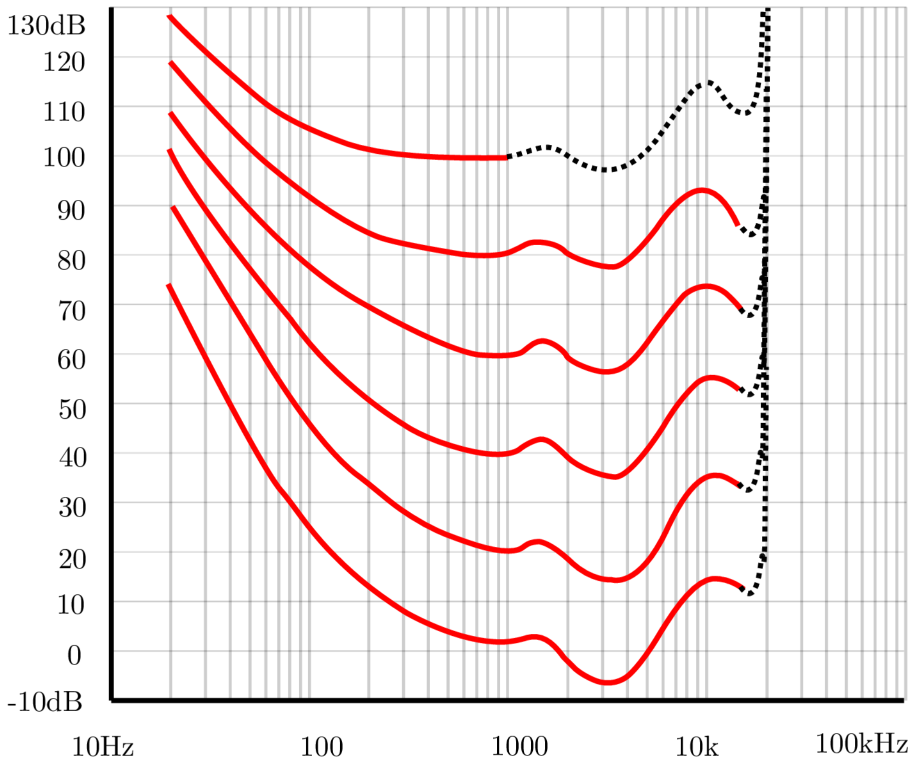


Figure 2.3: The equal loudness contours showing human perception of different frequencies.[1]

Tone quality is influenced by the combination of different frequencies, which gives the sound its unique characteristics. Even though the same pitch is being played, the sound is different when two different instruments are played, 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. Humans 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.

[1]Image used from Wikipedia `https://en.wikipedia.org/wiki/Equal-loudness_contour`

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 can be 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.

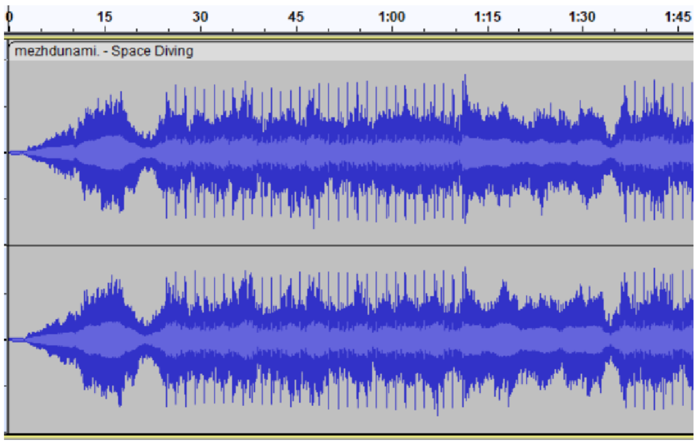


Figure 2.4: Visualization of an audio waveform in the time domain from AudaCity. In stereo, the x-axis is time, and the y-axis is amplitude.

Signal processing techniques are needed 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 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 as can be seen in Fig. 2.5.

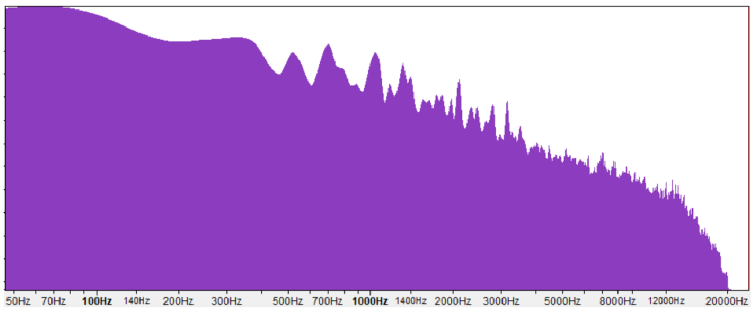


Figure 2.5: Frequency spectrum from AudaCity. The x-axis is frequency, and the y-axis is magnitude.

By analyzing the frequency domain, dominant frequencies can be detected 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.

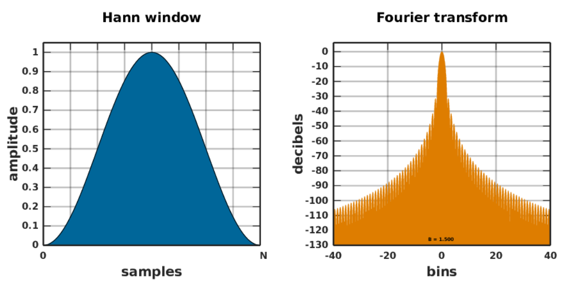
2.2.4 Fast Fourier Transform

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.

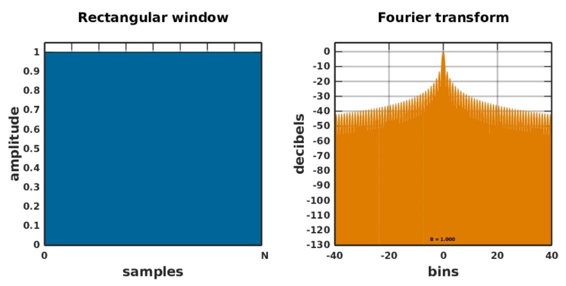
2.2.5 Windowing Techniques

After decomposing the audio signal into its frequency components using the Fast Fourier Transform (FFT), a common problem encountered is 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, this challenge can be addressed by employing windowing techniques to improve the accuracy of the 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) Rectangular window function

Figure 2.6: Examples of windowing functions. The plots display the signal's time-domain amplitude variation (blue) and its frequency-domain power distribution (orange).[2]

Fig. 2.6 shows the Hanning window, which was chosen 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 in Fig. 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 in Fig. 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]Images used from Wikipedia `https://en.wikipedia.org/wiki/Window_function`

2.2.6 Onset Detection

Onset detection detects musical events in an audio signal. As was already mentioned in the Subsection 2.1.3, 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 among different methods. 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 approach may be sufficient for beat estimation, but for more information, a more complex algorithm is needed.

When the data is converted to the frequency domain, algorithms can be used that can capture subtle changes in spectral characteristics. These characteristics may not be apparent in the time domain. Sometimes, a spike in amplitude may occur without an actual onset; this can be mitigated when analyzing the signal's frequency content. As shown in Fig. 2.7, hi-hat onsets (the grey rectangle) and kick onsets (the green rectangle) can be detected by analyzing two different frequency ranges. This would not be possible by analyzing the time domain. When the whole spectrum is analyzed, vertical lines corresponding to the waveform spikes are obtained.

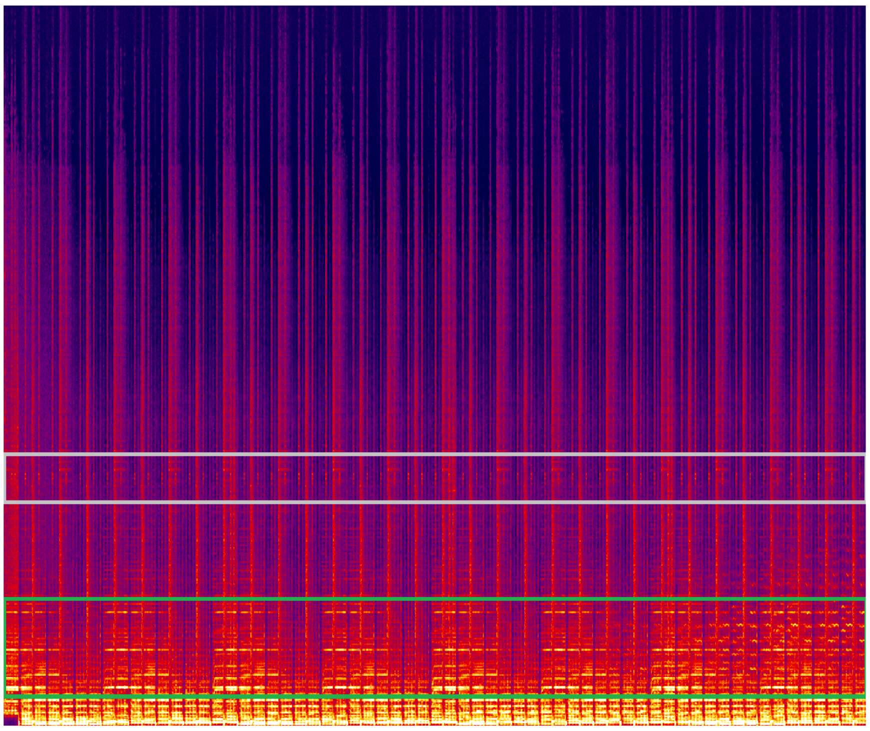


Figure 2.7: Spectrogram of a section of the song. The x-axis represents time, the y-axis represents frequency (20-20000Hz), and the color represents the magnitude of that frequency in that time.

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 depends on both sophisticated algorithms and 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 certainly is a challenging task.
- 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 a good framework for playing and manipulating audio data. The following subsection describes the components and functionalities relevant to the objectives of the thesis.

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 is not applicable to the objectives of the thesis.

The Audioclip [2], on the other hand, represents an audio asset that AudioSource can play. It represents the music with all the necessary data for analysis.

Unity seamlessly converts audio files such as .mp3 or .wav to AudioClip format, meaning that users do not need to worry about different formats.

2.4.2 AudioSource.GetSpectrumData

This method computes the audio signal's frequency spectrum using the FFT algorithm [2]. 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 [2] 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, for preprocessing the audio, an alternative method is needed.

2.4.4 AudioClip.GetData

GetData [2], 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 that tempo and melodies are recognized as interesting features of a song, and they can be identified using FFT and Onset Detection. Different data retrieval options in Unity, such as AudioSource.GetSpectrumData or AudioClip.GetData were mentioned. The 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, the song's other useful information cannot be forgotten, 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 describing 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. The schematic overview of the proposal can be seen in Fig. 3.1. 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**.

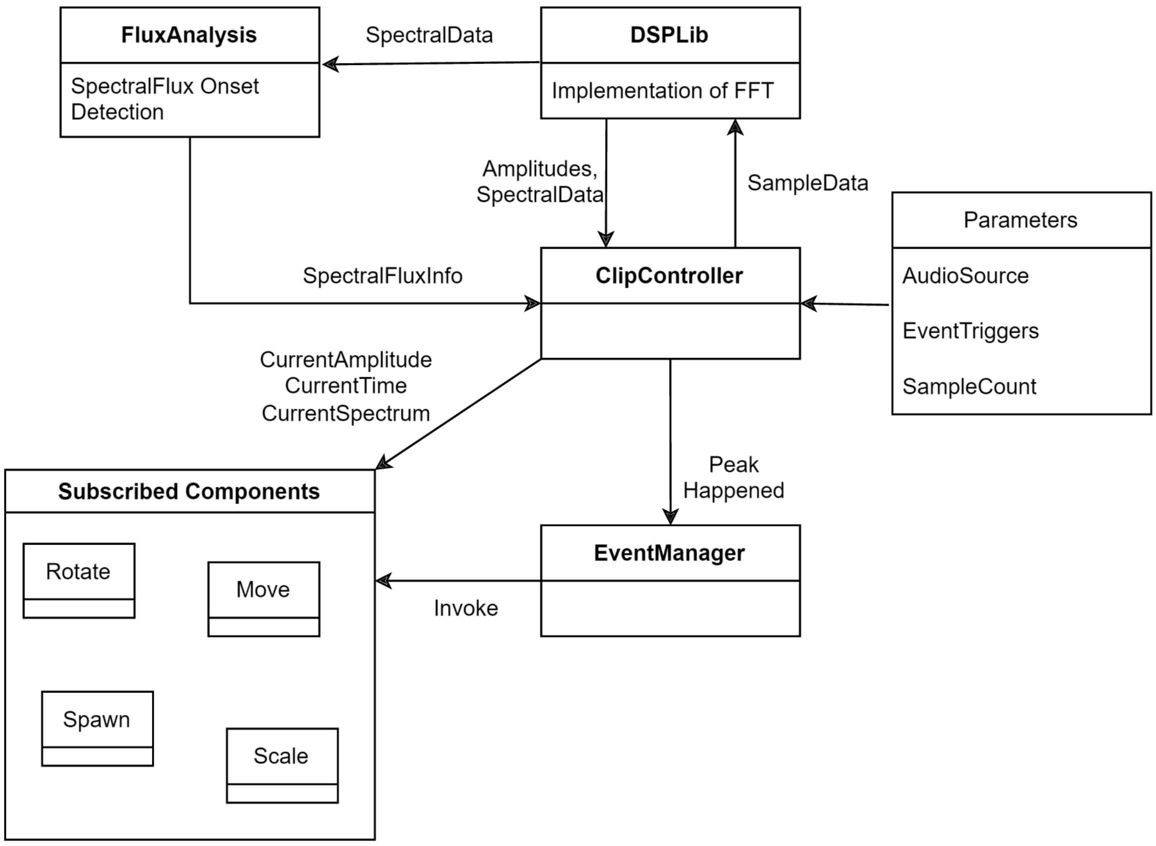


Figure 3.1: Architecture of the proposed plugin. Different objects, their functionality, and how they communicate with each other.

3.2 Analysis

The first step is 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 the 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

Structures are in place to store the analyzed data, approaching the core of the analysis; however, the FFT must not be forgotten. Since Unity does not provide its own FFT implementation, it is necessary to either create a custom solution or find one that is implemented. The library mentioned in the selected article, DSPLib [7], will be utilized for this project.

To perform FFT, necessary clip data, such as length and number of samples, must be gathered. Additionally, the sample count has to be chosen to determine the size of the frame that is going to be analyzed. The sample count has to be a power of 2. Usually, it is 512 or 1024, which is sufficient. A large sample count means a finer frequency resolution; however, this also means it will take longer to compute, which may not be desirable. While 1024 samples produced excellent results and the analysis time was satisfactory, user preferences may differ, and serializing these modifiable variables is good practice.

Handling Channels in Audio Processing

One important detail to think about is the number of channels. Songs can be either stereo, meaning that they have left and right channels, or mono which has only one channel. For the plugin to accept any type of song, it is necessary to calculate both scenarios. The GetData function returns the raw sample data of the song. In the case of a stereo song, there will be twice as much data as in a mono song for each channel.

$$DataSize = TotalSempleLength \times NumberofChannels$$

Before running the FFT library, the channel data must be combined. In the case of stereo audio, the FFT could be applied separately to the right and left channels. However, this approach offers limited advantages when the number of channels in the audio is uncertain. Combining the samples is straightforward. The GetData function always returns samples in this order: $[L_0, R_0, L_1, R_1, \ldots]$

Calculating the average of these samples generates a combined array on which the FFT can be performed. Let S be the multi-channel input array, C the number of channels, and F the final combined array. Then the calculation would look like:

$$P[j] = \frac{1}{C} \sum_{i=0}^{C} S[j \times C + i]$$

Where j is the index of the combined array F, i is the index within the current set of channels, and the expression $S[j \times C + i]$ accesses the samples in the original multi-channel array S.

Fast Fourier Transform and Simple Audio Analysis

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 as follows:

$$iterations = \frac{LengthOfSamples}{SampleCount}$$

Algorithm 1: ITERATION THROUGH CLIP SAMPLES

```
Input: $(clipSamples)$
1:  $GetCurrentClipSamples$
2:  $ApplyFFTWindow$
3:  $PerformFFTandConverttoMagnitude$
4:  $CalculateAmplitude$
5:  $CalculateCentralSpectroid$
6:  $SaveSpectrumData$
7:  $CalculateCurrentSongTime$
8:  for $trigger$ in $triggers$ do
9:      $SendDatatoSpectralFluxAnalyzer$
10: end for
```

Algorithm 1 displays steps computed in each iteration. The first step is to copy **sampleCount** amount of samples from the ClipSamples, which will be worked with. As mentioned in SubSection 2.2.5 windows are used to mitigate the effects of spectral leakage.

Applying the FFT window involves several key steps using the DSPLib [7]. First is the calculation of the coefficients for the desired window type, which in this thesis is the Hanning window. The next step is to scale these coefficients to match the window size. Finally, the computation of the scale factor is crucial for preserving the amplitude of the signal after windowing and performing the FFT.

Performing the FFT and converting to magnitudes involves these steps: First, the FFT is executed on the scaled spectrum window, transforming the time-domain signal into the frequency domain and producing complex numbers. These complex numbers represent both the amplitude and phase information of the frequencies present in the signal. Next, converting the FFT output to magnitudes, which extracts the amplitude information from the complex numbers, is necessary. This step is crucial because it provides a clear representation of the signal's frequency content, allowing for further analysis of the spectral properties of the audio. The final step is to scale the magnitudes using the scale factor to ensure the amplitude is preserved correctly after the transformation. This spectrum is saved as **scaledFFTSpectrum**.

The samples are now in the frequency domain, meaning some of the information can already be gathered. This plugin keeps track of the average amplitude, which is the mean of the magnitudes in the **scaledFFTSpectrum**. Spectral centroid, a measure of the center of mass of the power spectrum of a signal, has a more complex calculation than average amplitude.

Following equations shows the calculation of spectral centroid:

$$SpectralCentroid = \frac{\sum_{k=1}^{N}(f_k \times X(k))}{\sum_{k=1}^{N} X(k)}$$

Where N is total size of the **scaledFFTSpectrum**, f_k is the frequency corresponding to the $k-th$ item of spectrum, calculated as:

$$f_k = k \times \frac{SampleRate}{SampleCount}$$

and X_k is sum of amplitudes of the **scaledFFTSpectrum**. The numerator represents the weighted sum of all frequency contributions. The denominator is the total amplitude, ensuring that the centroid is normalized. This approach gives a weighted average frequency, reflecting the energy distribution across the spectrum.

The whole **scaledFFTSpectrum** is also saved, in case the user would like to utilize it. After that, the spectrum data is sent to the Spectral Flux Analyzer.

Keeping Track of Time

As could be seen in the Algorithm 1 the song's current time is also saved. This is because it allows simple retrieval of corresponding data by calculating the index based on the song time and accessing that index in the data list. The calculation of the precise time of the song goes as follows:

$$TimeDurationPerSample = \frac{1}{SampleRate}$$

$$TotalTimeElapsed = TimeDurationPerSample \times SampleIndex$$

$$TotalTimeDuration = TimeDurationPerSample \times SampleCount$$

Where $SampleRate$ is the number of samples captured per second in a digital audio recording. It is usually expressed in Hertz (Hz), where 1 Hz equals 1 sample per second and it can be obtained in Unity using *AudioClip.frequency*, and $SampleIndex$ is the current iteration index.

„To calculate the corresponding index for the actual time of the song, follow these steps:“:

$$LengthPerSample = \frac{TotalNumberOfSamples}{TotalLengthOfAudioClip}$$

$$SampleIndex = \left\lfloor \frac{CurrentTime}{LengthtPerSample} \right\rfloor$$

3.2.3 Onset Detection using Spectral Flux

The Spectral Flux Analyzer is an instance with variables that track data for each event. The main function AnalyzeSpectrum is called in parallel from Clip Analysis, as explained in Subsection 3.2.2, after the FFT analysis for each event trigger. Onset detection has already been discussed, and the next topic will be spectral flux.

Spectral Flux Calculation

Spectral flux or spectral difference measures the difference in magnitude between consecutive frames. 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, GetData was used to preprocess the song. This approach may introduce a loading time before a game level but results in a minimal delay between the song and data. Let $P(t)$ and $P(t-1)$ represent the power spectra of the current and previous frames, and let $SF(t)$ be the spectral flux.

$$SF(t) = \sum_f \max(0, P(t,f), P(t-1,f))$$

where f represents frequency bins in the power spectrum, which are the trigger ranges defined by the user.

Thresholding and Dynamic Thresholding

Next, a threshold must be calculated, which determines if the peak happened. A peak is identified when the spectral flux exceeds the threshold. While a static threshold could be used, it would not be precise, and it is not worth it to save some computer power for it. Therefore, implementing a dynamic threshold is necessary. Firstly the average of the N spectral flux values is calculated. This project calculates using 50 values.

$$AverageFlux = \frac{1}{N}\sum_{i=1}^{N} SF(t-i)$$

Afterward, a threshold multiplier, which defines the sensitivity of the threshold is multiplied by the Average Flux, resulting in threshold T. This threshold multiplier is a parameter of event triggers created by the event customization feature 3.2.1. That means the user can test what result works for them.

Peak detection

To detect the peak, it is important to calculate the pruned spectral flux. This is the difference between the spectral flux and the threshold, representing whether the spectral flux is greater than the threshold:

$$SF_{pruned}(t) = \max(0, SF(t) - T)$$

Peak is detected when $SF_{pruned}(t)$ exceeds both $SF_{pruned}(t+1)$ and $SF_{pruned}(t-1)$.

3.2.4 Beat Detection

The peaks have been detected, and using them to a rough estimation of the song's tempo can be made. By expanding the base algorithm on specific ranges, the chances rise, however, it still is not reliable. 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, however, users may use songs of different genres and qualities. That's why the beat generation was added, to ignore the song's quality and poor mixing. Users can obtain the tempo in BPM from third-party applications, specify it in the main script as a parameter and the beat generation will create events accordingly.

3.3 Functionality

The necessary data has been extracted from the audio clip at this stage. Users can now access this data through the main component called ClipController, which requires an Event Triggers scriptable object and an AudioSource to analyze a clip. In order to inspire users and showcase the potential, additional functionality needs to be implemented. This will include a custom event system, beat generation, and various other components.

3.3.1 Event System

The unity event system is powerful and very useful, but after introducing custom events from Subsection 3.2.1 to this plugin, a custom event manager was needed with the event system more catered to the needs of the plugin. 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 whether 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.

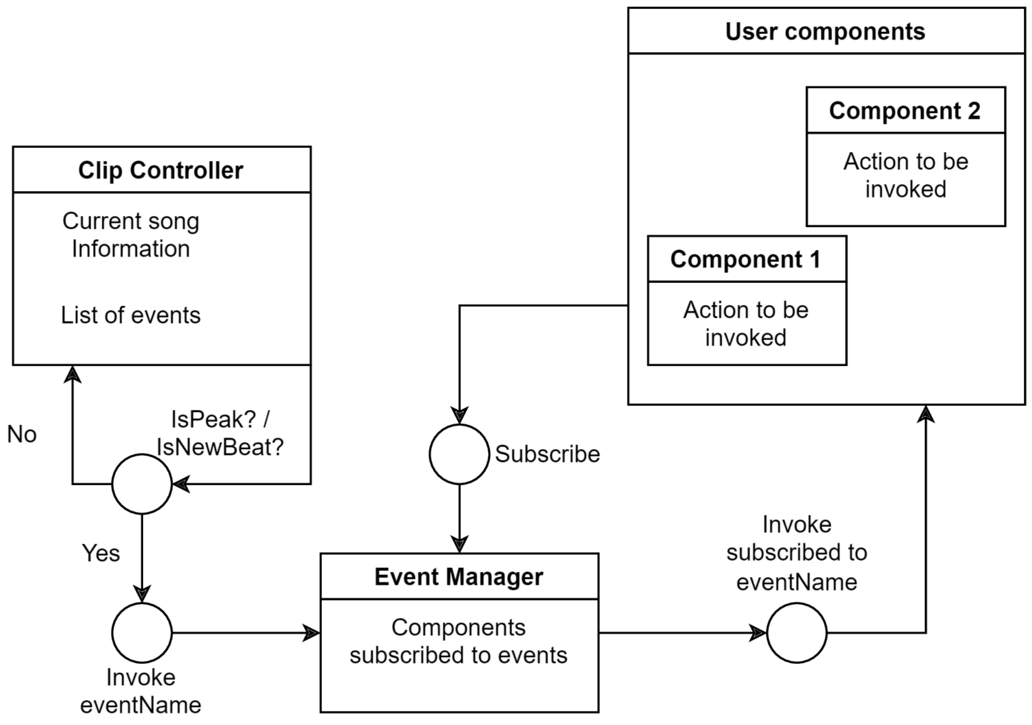


Figure 3.2: Simplified diagram of the event system displaying communication between components and Event Manager, and Clip Controller.

3.3.2 Beat Generation

As previously mentioned in Subsection 3.2.4, beat detection is a challenge. Beat generation, on the other hand, is pure math. To calculate the interval between individual beats, the song's BPM is a necessary parameter. This can be obtained using third-party software.

$$\frac{60}{BPM} = interval$$

For example, if the song has a tempo of 120 BPM, it means that there will be 0.5 seconds between each beat. Now, the beat event has to be invoked. To do that, the program has to keep track of time; a different functionality is used then in the analysis. Synchronizing object manipulation to beat is very delicate, which is why high accuracy is important. 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 [2]. Afterward, the difference between the start of the track and the current DSP time determines if a beat happened.

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. Specification of time signatures was added to give the user even more flexibility and possibilities. As Subsection 2.1.2 mentions, time signature specifies how many beats are in a bar. This beat generation is straightforward 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, a more complex system called a cinematic camera system was also created. 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 some of them are described below.

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 users 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 the event triggers.

Chapter 4

Testing

Testing is very important in all kinds of projects, especially the creation of plugins. The results obtained will determine the potential of the plugin. The focus will be on the overall user experience and synchronization with the music. Focusing more on the outcome of the plugin than the usage reveals if the plugin has any potential at all. Test questions were formulated 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 and space videos 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 is reminiscent 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 Analysis Results

To better visualize when the analysis detects peaks, the results saved as JSON files, generated by DataWriter 3.5.1 were utilized and then plotted using the Python library Matplotlib.

In Fig.4.1, the analysis shows a melody that resembles stars. The melody was mainly detected between 65 and 115 seconds, corresponding to the spectrogram in the same timeframe within the frequency range of 1046.50 Hz to 2093.00 Hz. This indicates that the melody detection was accurate. However, as shown in Fig.4.1 and the video in Appendix A, the stars occasionally appear too close together, even when the melody is not playing. This clustering is visible around the 100-second mark in the plot. To address this, increasing the threshold multiplier or detecting each note individually might improve accuracy.

The peaks detected before the melody starts or ends are problematic, as they were not intended. To mitigate this issue, the stars object was activated just before the expected start of the melody.

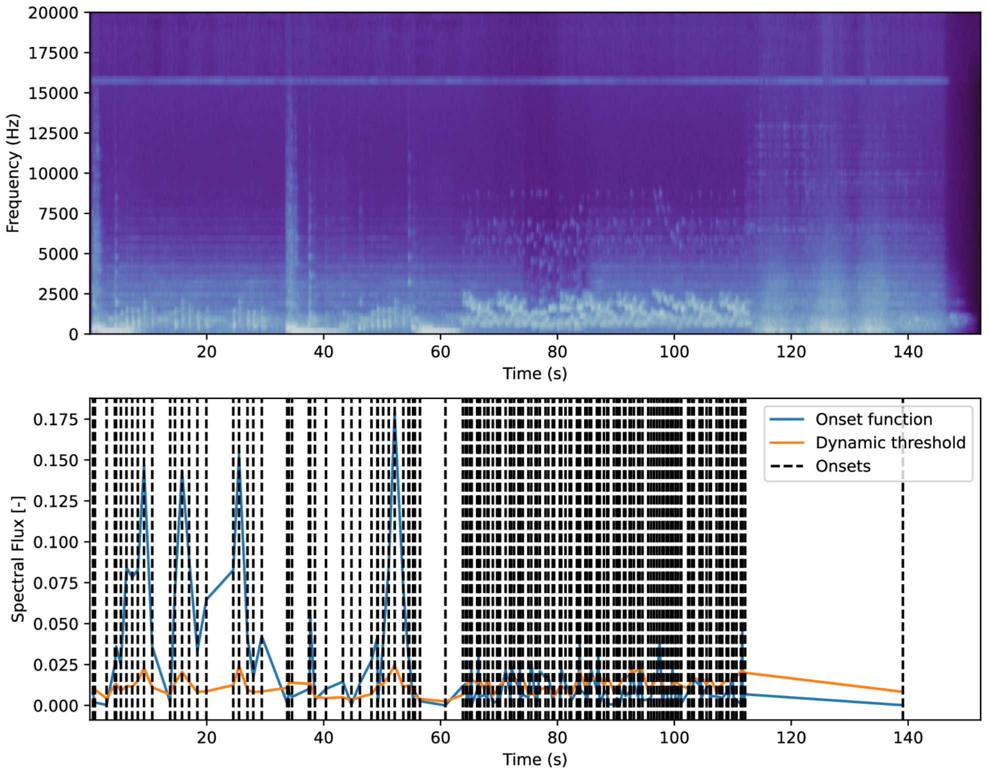


Figure 4.1: Figure displaying spectrogram (top plot) of *The Baudelaire Orphans by* ***Thomas Newman*** and flux values with the dynamic threshold and onsets of the melody event trigger (bottom plot).

4.1.2 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 then the procedural sound-based elements were added 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, the unity fog system was used to create a feeling of mystery, the light was changed to lighter blue to mimic the moon, and the scene was finished.

Take a look at Fig. 4.2a to see the whole scene without the fog that would block the view.

4.1.3 Scene Functionality

For functionality showcase, the custom-made checkpoint camera movement, mentioned in Subsection 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. 4.2b, 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, a pleasant melody reminiscent of stars was utilized. Two particle systems were created, one representing the light of a star and the second adding flare. Together, they created a pretty star-like effect.

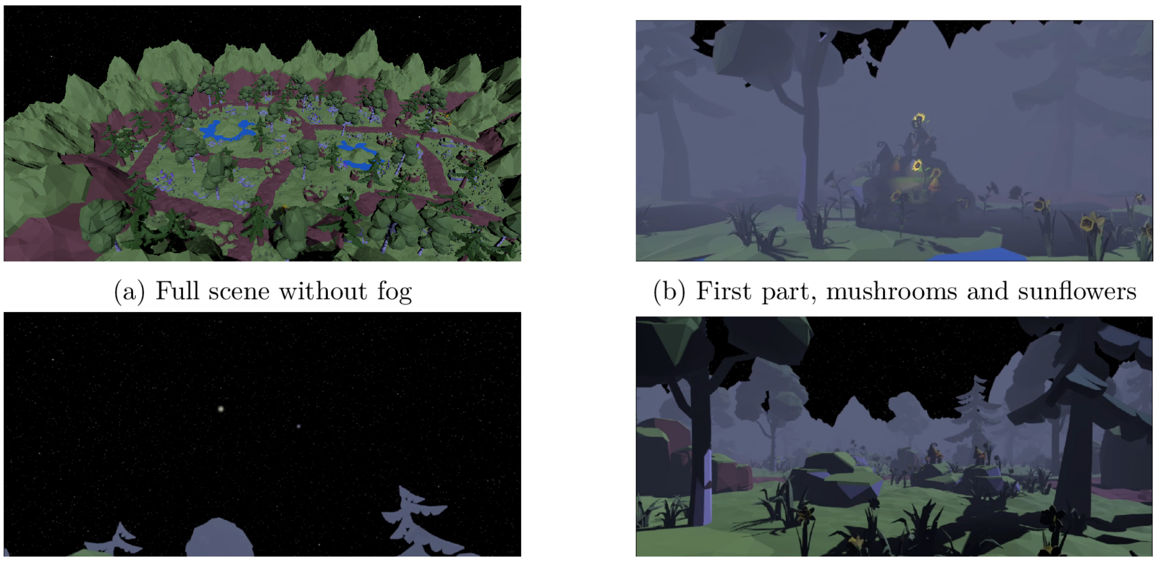
To synchronize it with the piece, a script was created that randomizes the position of the particle systems and emits one particle with a dynamic size of the current amplitude in the camera direction.

The decision was made to include the lower arpeggio, which accompanied the melody. The particle systems were duplicated, assigned a different event trigger, and given a distinct color for improved visibility, resulting in an appealing star dance effect.

The second part can be seen in Fig. 4.2c.

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. 4.2d


(a) Full scene without fog

(b) First part, mushrooms and sunflowers

(c) Second part, starry sky

(d) Third part, fading fog

Figure 4.2: Different parts of the forest scene video and important functionality.

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 Analysis Results

The spectrogram was also created using the Python library Matplotlib and results were generated by DataWriter 3.5.1.

In Fig. 4.3, there are no distinct onsets shown. This is due to the chosen song having a tempo of 160 BPM (approximately 2.5 beats per second) combined with some imperfections in the analysis, resulting in the plot being dominated by continuous lines.

The bottom plot illustrates the Bass range 2.1.4, in the frequency range of 60–250 Hz. Onsets can still be identified by examining the Dynamic Threshold line and the Onset Function values. An onset occurs when the Onset Function value exceeds the Dynamic Threshold line. However, except for some quieter sections, onsets were being detected almost continuously. This problem required the development of the Beat Generation.

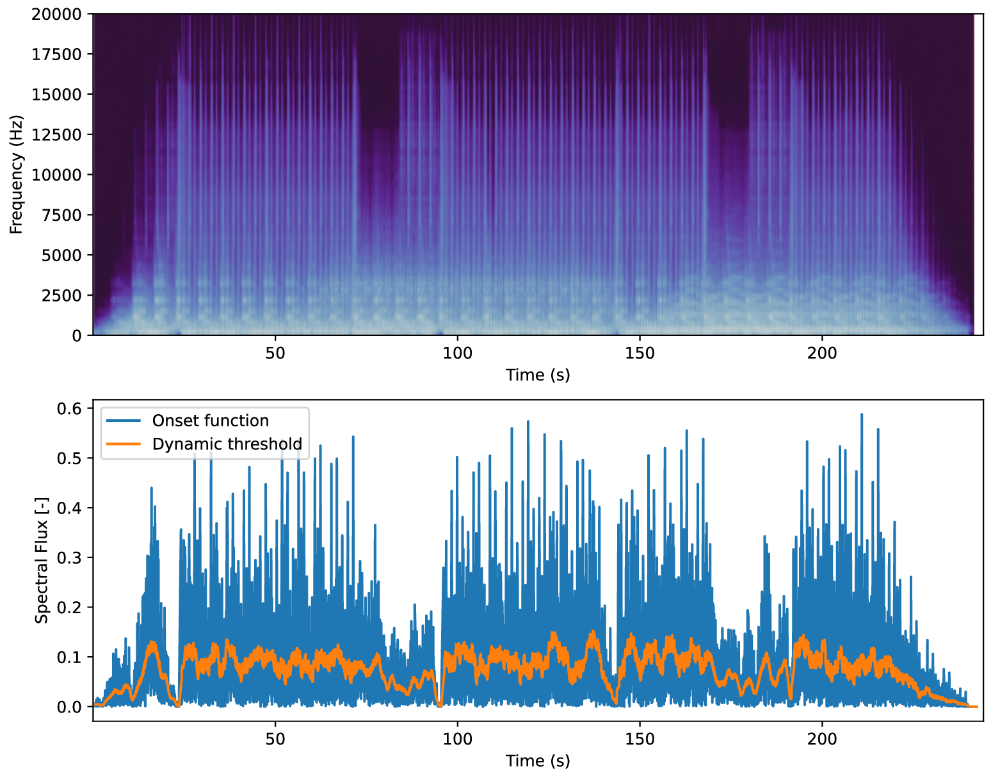


Figure 4.3: Figure displaying spectrogram of Space Diving by **mezhdunami.** (top plot) and flux values with the dynamic threshold of the SubBass range(bottom plot).

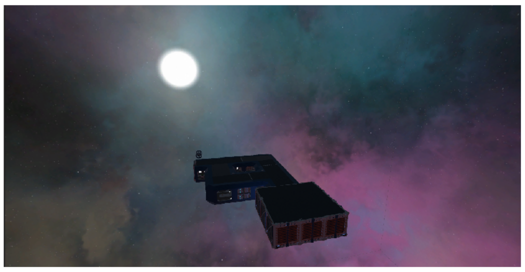
4.2.2 Scene Creation

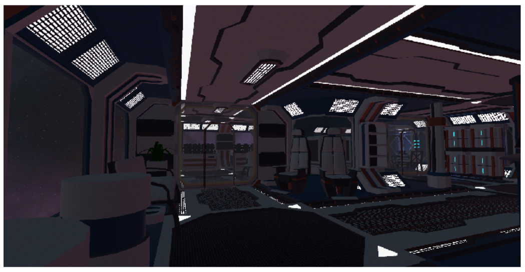
As with the Forest Scene 4.1.2, all the assets used in this scene are free from the asset store. The asset contained a space station scene that was already prepared and just needed the plugin's functionality to be complete. A player movement script was created with a rotating camera for video recording. Additionally, a space skybox was implemented, seen in Fig. 4.4a, using the same third-party tool, 3DTool, that was utilized in the forest scene to position the space station within the cosmos. This scene lacks the post-processing touch that the first scene had, but more importantly, it has to showcase the beat synchronization, which is visible. See Fig. 4.4 to see the space station.

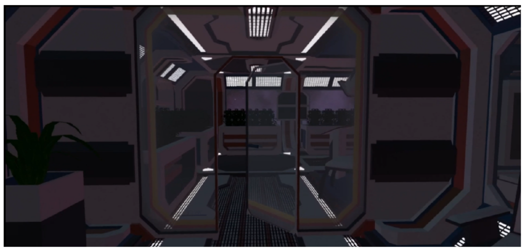
4.2.3 Scene Functionality

Here, all the beat components mentioned in the Subsection 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

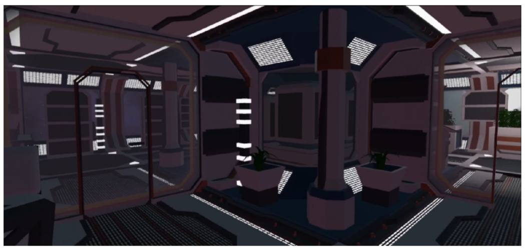

(d) Another angle of the space station

Figure 4.4: Different figures of the space scene video.

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. 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.

At the start of this chapter, it was mentioned that 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 questionnaire.

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 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, 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 questionnaire.

Chapter 5

Conclusion

Music is closely tied to daily lives, accompanying people almost everywhere: in the car, in the supermarket, while watching movies or series, and even while playing games. Utilizing the natural perception of music in different kinds of mediums can create an immersive environment that enhances emotional feelings and perception of the music being played.

This paper explored fundamental music concepts, as discussed in Section 2.1, including tempo, beat, onset, and frequency. It explains how humans perceive sounds and volume in SubSection 2.1.5. Section 2.2 covered audio signal processing providing an overview of the time domain and frequency domain, and discussed the Fourier Transform and its application, including the Fast Fourier Transform. Section 2.4 highlighted the benefits of procedural generation and examined Unity's handling of audio clips and potential data retrieval methods.

The proposal for a plugin that analyzes background music playing in a Unity scene was then examined. The steps of clip analysis mentioned in Subsection 3.2.2 and onset detection using spectral flux in Subsection 3.2.3 were explored, revealing the challenges of beat detection. Data management and event customization solutions were introduced, featuring a custom event system designed to work with any created event. Afterward, a solution for rhythmic elements in games using simple beat generation was found. Additionally, the various components offered by the plugin were discussed, including object transform modification to the rhythm of a song and dynamic fog controlled by the song's current amplitude. A custom cinematic camera script that allows users to plan camera movement with the song timeline and rotate the camera towards a selected object was also mentioned, along with some quality-of-life features in the plugin.

Testing was conducted after the creation of all components, and the results were analyzed. The creation of two testing scenes, their motivation, and the differing reactions they received were discussed. The forest scene, showcasing the dynamic camera and environment, received more positive feedback compared to the space scene, which displayed rhythmic patterns prone to inaccuracy, as detailed in Section 4.3. The plugin's potential was evident even before testing, but the results confirmed that it was progressing in the right direction.

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 waiting time needed to 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 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 the 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.

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 package on the disk *Procedural_Sound_based_Elements.unitypackage* contains all the necessary files for it to work. The project runs on Unity version 2021.3.24f1. Simply import the package into Unity, and download the dependencies listed in *README.md*. To explore the prepared scenes seen in this thesis navigate to **ProceduralSoundBasedElements** folder and import the project into Unity.

```
/
├── README.me ......... Markdown file with information about the package installation.
├── Procedural_Sound_based_Elements.unitypackage . Unity package containing the
│   library
├── Procedural_Sound_based_Elements_in_Games.pdf ............... Thesis pdf file
├── Procedural_Sound_based_Elements_in_Games_poster.pdf ........ Thesis poster
├── 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 with the scenes for testing
    └── 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
```