MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kotov Igor**	Personal ID number:	**473276**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Branch of study:	**Human-Computer Interaction**		

II. Master’s thesis details

Master’s thesis title in English:

Non-linear Playback of Collection of Audio Samples for Video Games

Master’s thesis title in Czech:

Nelineární přehrávání v rámci množiny zvukových vzorků v kontextu videoher

Guidelines:

Audio effects in video games are generally either “one-shots”, i.e. one-off triggered audio effects which are played back whenever a finite and time-limited action happens (e.g. a footstep, a gutshot, etc.), or “sustained sounds”, which are played as long as an action is in progress and its acoustic activity should be illustrated to the player (e.g. running car engine, howling wind, rain).
Investigate the state of art of implementation of sustained sounds in video games. Investigate current technologies used in this context (e.g. FMOD Studio) and how they enable the video game developers to parametrize the playback of those sounds in real time, so that they can make the sound reflect immediate changes in the video game. Investigate various methods of audio synthesis, especially wavetable (PCM-based) synthesis and granular synthesis, which enable that the sounds in the video games are generated using existing recordings (samples).
Design and implement a system which would enable a non-linear playback of those recordings. The system may automatically identify suitable loop points and cross-fade points between different samples. Implement an API to this system so that it may be used in the context of video games (FMOD Studio, Unity, or other platforms). Test the implemented system by (1) running a perception test with at least 5 participants, and by (2) implementing a very simple video game illustrating the use of such a mechanism. The exact extent and scope is to be discussed with the supervisor of the thesis.

Bibliography / sources:

Karen Collins (2013) Playing with Sound. MIT Press

Name and workplace of master’s thesis supervisor:

doc. Ing. Adam Sporka, Ph.D., Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **15.02.2019** Deadline for master's thesis submission: ___________

Assignment valid until: **20.09.2020**

doc. Ing. Adam Sporka, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Ing. Pavel Ripka, CSc. Dean’s signature

CZECH TECHNICAL UNIVERSITY
IN PRAGUE

FACULTY OF ELECTRICAL
ENGINEERING

Department of Computer Graphics and Interaction

Open Informatics

Diploma thesis

Non-linear Playback of Collection of Audio Samples for Video Games

Author: Bc. Igor Kotov
Supervisor: doc. Ing. Adam Sporka, Ph.D.
Prague, 2019

Declaration

I hereby declare that I have completed this thesis independently and that I have used only the sources (literature and webpages) listed in the enclosed bibliography.

Prague, 31. May 2019

....
signature

Acknowledgements

I would like to express my deepest gratitude to my supervisor, doc. Ing. Adam Sporka, Ph.D., for his kind and willing guidance over the course of research and writing of this thesis. I would also like to thank all participants of the perception testing for their time and interest.

Bc. Igor Kotov

Název práce:	**Nelineární přehrávání rámci množiny zvukových vzorků v kontextu videoher**
Autor:	Bc. Igor Kotov
Obor:	Otevřená informatika
Druh práce:	Diplomová práce
Vedoucí práce:	doc. Ing. Adam Sporka, Ph.D. Fakulta elektrotechnická ČVUT

Abstrakt: Tato diplomová práce se zabývá návrhem, implementací a testováním algoritmu pro zpracování digitálního signálu, který by mohl detekovat všechny možné body přechodu v hudebním souboru s pevným rytmem tak, aby jeho nelineární přehrávání na základě těchto přechodů bylo bezproblémové a přirozené. Tento algoritmus by mohl být použit v různých oblastech elektronické zábavy, jmenovitě v adaptivním zvukovém designu pro videohry.
Nejprve je uveden přehled současných populárních technik pro realizaci interaktivního zvuku s případy použití vzorku. Diskuse se pak zaměřuje zejména na pojem nelineárního přehrávání zvukových souborů přes přechod a dotýká se několika metod zvukové syntézy. Poté je poskytnuta nezbytná podkladová práce, analýza výzev, které představují úkoly programování a navrhovaná řešení. Dále je podrobně popsána implementace algoritmu a podpůrná technologická rozhodnutí. Využití nového přístupu je pak demonstrováno na jednoduché GUI aplikaci. Testování vnímání je prováděno, jeho metoda a výsledky jsou diskutovány. Následně je ukázková ukázka použití algoritmu pro účely procedurálního zvuku demonstrována na demo hře, která se opírá o problémově přizpůsobené API. Dále jsou navržena perspektivní vylepšení uvedené techniky.

Klíčová slova: pracování digitálních signálů; procedurální audio; adaptivní hudba; Unity; zvukové srovnání

Title: **Non-linear Playback of Collection of Audio Samples for Video Games**

Author: Bc. Igor Kotov

Abstract: This master thesis is concerned with designing, implementing, and testing a digital signal processing algorithm that could detect all possible transition points within a rigid-beat music file so that its non-linear playback based on those transitions is seamless and natural. That algorithm could be employed in various fields of electronic entertainment, namely, in adaptive sound design for videogames. First, an overview of currently popular techniques for realizing interactive audio is given with sample use cases. Then, the discussion focuses particularly on the notion of non-linear audio file playback via transitioning and touches several methods of sound synthesis. After that, the necessary background of the work is provided, the analysis of challenges that the programming task poses as well as of suggested workarounds is done. Next, the algorithm's implementation and the supporting technological choices are thoroughly described. Then, the usage of the new approach is demonstrated on a simple GUI application. Perception testing is carried out, its method and results are discussed. Afterwards, sample usage of the algorithm for the purposes of procedural audio is demonstrated on a demo game that relies on a problem-tailored API. Finally, prospective enhancements of the shown technique are proposed.

Key words: digital signal processing; procedural audio; adaptive music; Unity; sound comparison

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the modern electronic entertainment industry, the role of audio becomes more and more difficult to overestimate. Historically, the potential of sound was known to affect humans: provoke their emotions, shape their thoughts, touch their feelings – sometimes, without one even noticing the mental change. Nowadays, sound designers in the field around the world try to make use of that perceptional phenomenon to make virtual environments more immersive, more exciting, more alive for players and viewers, introducing audio not just as a supportive – but instead, as an experiential dimension of a composition. The spreading notion of employing music and sound effects to work together with graphics to boost the artistic impact has since been spoken of as interactive audio (sometimes also "procedural", "adaptive", or "generative" audio). Generally, that term intends bringing either, some, or all of the following features into a product:

- Making the characteristics of the perceived sound dependent on the current content of (events in) the scene (or making them appear as such to the listener);
- Providing the mechanism such that the sounds of repeated "one-time" actions (e.g. footsteps) and steady continuous events (e.g. mountain river stream) are perceived as non-repetitive and natural;
- Allowing user to affect composition's audio by sending different (combinations of) trigger signals to it.

Procedural audio systems are made to deliver an exceptional diversity of sound – but, instead of playing back loads of raw source files repeatedly, they aim to construct new, lifelike sound patterns and sequences by processing a relatively small amount of existing data in elaborate ways. The concept of generating music based on previously programmed rules is quite old, however – the very first computer games were already relying on sound generator chips like Texas Instruments SN76489, General Instrument AY-3-8910 or Yamaha OPL3. These offered basic synthesis with the help of oscillators, filters, ADSR-envelopes (attack-decay-sustain-release envelopes) and modulators [1]. Today, the market presents plenty of software tools for high-level programming of adaptive music and sound effects (SFX). The most popular IDEs (integrated development environments) up to date, – FMOD Studio, Wwise and Elias Studio, – do not only have all the aforementioned fundamental components but encapsulate mechanisms that help significantly increase programming possibilities (see chapter 2).

This master thesis considers the most important and, currently, the least flexible of those mechanisms – the concept of transitions. The goal of the work is to build, implement and evaluate a new algorithm that could help expand the capabilities of a context-aware transitioning during interactive music design. The algorithm should locate frequencywise similar moments within a rigid-beat musical audio file and output data containing locations of acceptable samples (grains of playback) which can be interpreted as transition points or loop region border points in higher-level software. To achieve the goal, the following tasks should be completed:

- State-of-the-art techniques for realizing interactive audio in electronic entertainment products (and transitioning in particular) should be investigated, their scopes of use, advantages and limitations should be reviewed;
- The notion of non-linear file playback as one of the techniques used in procedural audio design should be introduced;
- Necessary background for practical work should be researched, analysis should be conducted of the challenges that the algorithm programming task poses and of possible ways of facing those challenges;
- The algorithm's implementation and the supporting technological choices should be described;
- The performance of the new approach should be demonstrated on a simple application with graphical user interface (GUI);
- Perception testing should be carried out in order to evaluate the correctness of the algorithm, the steps and the results of testing have to be discussed;
- Sample practical usage of the algorithm for the needs of procedural audio should be shown in a simple demo game relying on a custom problem-tailored API (application programming interface);
- The prospective enhancements of the shown technique should be be suggested.

Chapter 2

Survey

Currently, the adaptive audio behaviour in videogames and other entertainment software is mostly set up and controlled via designated middleware. The three most advanced and most relevant solutions today are FMOD Studio, Wwise and Elias Studio. In the following two sections, the features that these IDEs offer to help bring the interactivity into the sound system of a game are illustrated at the example of FMOD Studio[1]. In the third section of the chapter, the non-linear approach to playback through transitions is addressed in particular. In the fourth section, an overview of three popular techniques of audio synthesis as small-scale non-linear playback methods is given. In the last section, recapitulation is done.

2.1 FMOD Studio fundamentals

In order to understand how FMOD Studio serves the purpose of designing interactive game audio environments by providing its own problem-tailored tools, it is necessary to briefly look at the basic components of the software. Along with its analogues on the market, FMOD Studio functionality revolves around the same core principles as that of most digital audio workstations.

The main workspace in the IDE is arranged in a set of so-called "tracks" – the independent layers of sound that can be populated with audio clips. The sound perceived at every immediate moment is made up by combinations of processed inputs from all the tracks. The position at which audio is played is dictated by the playhead, which, in its turn, can either be linked to the timeline or to the values of in-game variables (see section 2.3). Figure 2.1 shows sample editor window of FMOD Studio.

In order to give sound designers control over separate tracks that make up the output signal, the mixer feature is introduced. It allows to calibrate volume levels between audio stems, fine-tune their panning, and apply various effects to them (more on effects later in this section). Moreover, the mixer supports grouping of multiple tracks which makes it easier to share common settings between a number of layers. Finally, users are presented a possibility to establish custom signal routing between tracks and track groups at any place of signal processing pipeline.

[1] All three aforementioned middleware products have different user interface layouts and workflow mechanics, however the concepts they rely upon are the same. FMOD Studio was chosen for an overview as the one being mostly used among major game development studios [2].

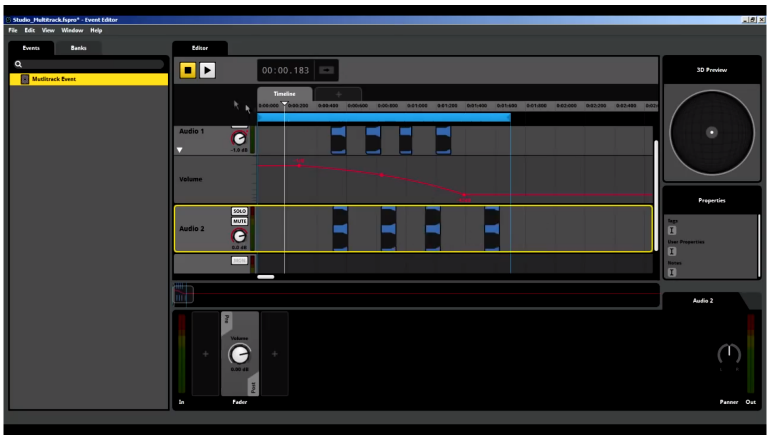


Figure 2.1: The editor window of FMOD Studio. Two tracks are created, each populated with four audio clips.

Layering and mixing of multiple complex signals and audio tracks assists to build up soundscapes of different intensities, characters and moods.

Additionally, FMOD Studio comes with a variety of stock effect plugins that can be inserted into signal routing chains to modify the input in a certain way before letting it pass further. Just like in DAWs (digital audio workstations), those plugins help increase the diversity of produced sounds and timbres. The effects include equalizing, compression, spatial saturation, 3D panning, pitch correction, phasing, distortion, LFO (low frequency oscillation) modulation and others[2]. Chains of multiple effects can be also assigned to a special type of track – a return channel. That gives one the ability to manipulate the clean and the processed version of signal separately. An example of mixer window is depicted in figure 2.2.

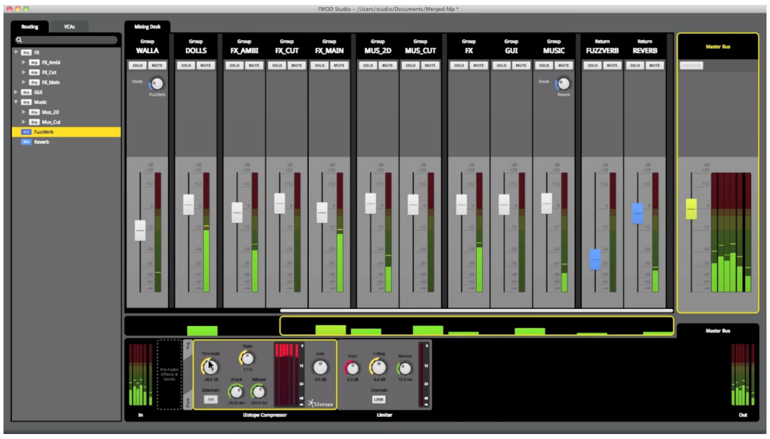


Figure 2.2: Mixer window of FMOD Studio. Ten groups of tracks and two return channels are created. The groups "WALLA" and "MUSIC" send part of their signal to the return tracks with effects set up. Outputs from all groups and returns are routed to the main output – the master bus. The master bus has two effects chained – compressor and limiter.

[2]Neither FMOD Studio, Wwise nor Elias Studio support the integration of third-party VST-plugins, however, Wwise has paid proprietary plugins available for download.

2.2 FMOD Studio features for interactive audio design

With multitracking, routing, mixing and plugin-tweaking possibilities forming its foundation, FMOD Studio delivers specialized features for dynamic audio-shaping that set it apart from regular DAWs. Firstly, almost every parameter of an audio clip, track, track group or effect can be automated or/and modulated. Automation is used to predefine how certain property should change depending on playhead position. This is done by drawing value automation curves for corresponding properties. A good example of turning to this technique is a military shooter game, where a low-pass filter can be placed on the SFX bus of the mix with its frequency property linked to the level of player's health. That way, the more player's health level drops, the less high frequencies are audible to the gamer, which reflects the natural partial loss of consciousness and hearing that a solider would experience in a real-life situation. Another example of automating track volume is shown in figure 2.3.

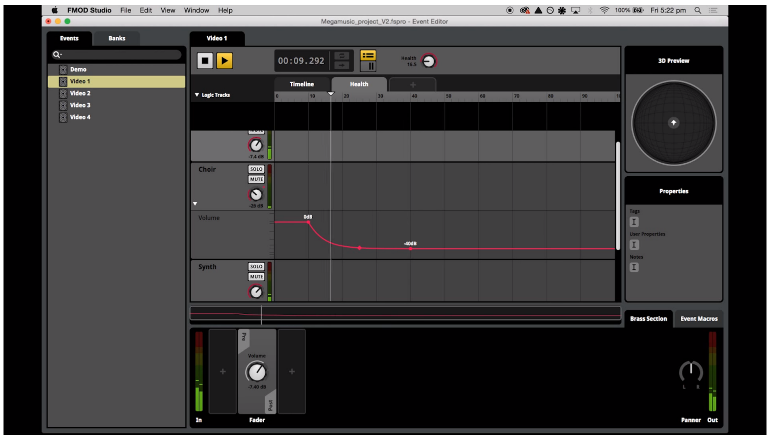


Figure 2.3: Example of volume automation within FMOD Studio. The curve drawn defines the volume that the "Choir" track will be played on depending of the value of parameter "Health".

Modulation can be used to specify how the parameter should change over the course of playback of the corresponding audio file. FMOD Studio features three types of modulators: random (make a property value fluctuate unpredictably within specified range), AHDSR (apply an attack-hold-decay-sustain-release envelope to the associated parameter) and sidechain modulators (link the modulating behaviour to signals received from other sources of the mix). The most primitive use case for modulation is humanizing repetitive one-shot sounds like footsteps or gun shots – here, the pitch and the volume of audio file are slightly randomized so that the sound is realistically slightly different each time it is played back. More sophisticated setups can have multiple types of modulators chained and accompanied by a number of automation curves. Figure 2.4 illustrates an example of using modulation.

Another advantageous tool that can be found in FMOD Studio and similar IDEs are multi-sound containers. The name speaks for itself – those structures are special types of audio clips that can contain more than one sound file. As such clip is being repeatedly triggered, it can either play its contents in order or pick one file randomly each time. This feature is commonly used together with pitch and volume modulation when working with one-shot SFX to avoid monotonousness in the tone.

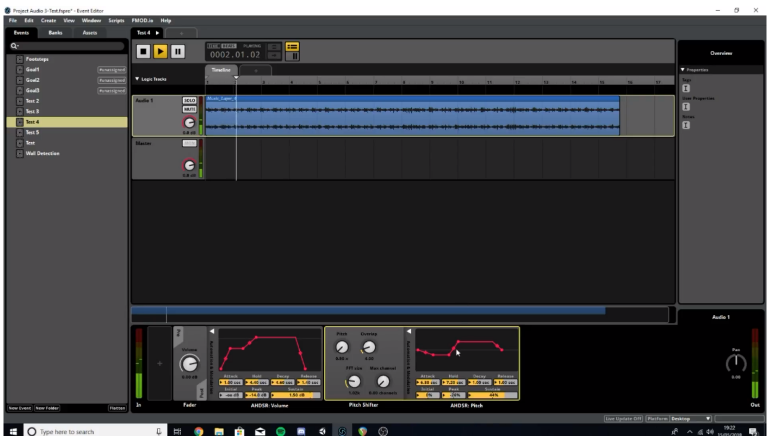


Figure 2.4: Modulating track parameters within FMOD Studio. The volume and the pitch of an audio clip are modulated according to corresponding AHDSR-envelopes preset for each of the two properties.

The technique finds another popular application in background game music, when each container corresponds to a separate instrument in the arrangement and hosts its own set of homogeneous, yet mutually different sounds. That variability in timbers achieved by this move makes the whole composition appear more organic and more lively to the listener. All the functionality covered in this section so far unleashes much more of its potential when employed together with the system of playback transitions. This concept is closely investigated in the following section.

2.3 Playback transitioning

Transitioning is a powerful instrument that is arguably the foremost within contemporary interactive music design paradigm. Having been evolving over time within adaptive audio middleware, the technique bases on an extremely basic principle. A transition is understood as a jump of a playhead from and to a certain point on a timeline. Provided that the timeline has audio-clips and sound containers placed across it and possibly distributed between multiple tracks, transferring from one moment of playback to another makes the whole composition unfold in a different, sometimes unpredictable manner. As opposed to the normal way of reproducing music, where audio samples are processed subsequently from start to end, transitioning in adaptive music is referred to as a non-linear playback method.

One of the key goals of modern IDEs like FMOD Studio is to let its users construct transition systems that would organize generating of dynamic sound patterns based on advanced logic, but at the same time would be invisible to the consumer in their operation. In other words, ideally, the player should never be able to tell if there even was a transition – what is heard should be thought of as a complex, life-like soundscape that transforms along with the everchanging game world. At every second of time, game audio is made up from a set of diverse timbres and voices that suggest certain emotional context to players and make them feel the specific way. In that sense, each subjectively different state of game audio falls under the term ”sound theme” (or ”sound scene”). The ”seamlessness” that developers try to get to can be looked into from two contrasting perspectives and is defined depending on whether the motivation for transition is staying within the same sound scene or transferring between different ones.

2.3.1 Transitioning within sound themes

One type of challenge that designers face is keeping the playback going in conditions of somewhat stable audio environment, i.e. when no drastic changes are supposed to be made to the mood of the music arrangement or to the character of the sound effects. When it comes to game music, the most obvious, though extensively used approach is to define so-called loop regions[3] – these are audio clips[4] (or parts of those clips) that are being iteratively played back in a linear fashion the way that the transition is always made from the last sample of a region to the first sample of the same region. If during its repetition a loop perfectly extends itself (i.e. it is hardly possible to determine when the playhead jumps by ear), then such region is spoken of as "a seamless loop" (or, alternatively, "perfect loop" or "synchronized loop").

Usually, seamless loops are composed/recorded and prepared even before the phase of interactive sound design. As those loops should later form a cohesive music piece, sound producers, relying on their personal artistic taste, have to make sure that tempos, keys, time signatures and durations of those segments match or correlate with one another musically. Of course, composers should also have comprehensive knowledge of what game they are writing for and what kind of tune would fit a particular setting. All audio loops are then imported into adaptive music IDEs with time signature and tempo settings adjusted in accordance to what they were on the production step. That way the arrangement parts align with the project's timeline grid and can be easily put inside of loop regions.

For one-off and sustained SFX, offset randomization comes in handy as another method of within-scene transitioning. It picks a different start sample for an audio clip every time it is triggered. This trick can be complemented by turning on the looping, so that once the last sample of the clip is played, the playhead jumps to a random position inside a predefined region. An example of such strategy would be adding sounds to a number of identical game objects that all have to be audible at the same time, e.g. burning torches. With randomization, it is possible to get by a single audio file for all of the object instances: because the playback starts at a different point of the clip each time, the outputs from multiple sources do not blend into one but create a pleasant, volumetric effect[5] (see figure 2.5).

[3]Looping regions are sometimes used for sustained sound effects too – for instance, an audio file of a creaking windmill could be set up to be replayed over and over. Of course, other techniques, such as randomized modulation, should be used to guarantee realistic audio experience in that case.

[4]What is stated about audio clips here and later in this section applies to multi-sound containers as well.

[5]Proper 3D-panning of the sound should be set up with the linkage to the position of player's character to achieve realistic panoramic soundscape.

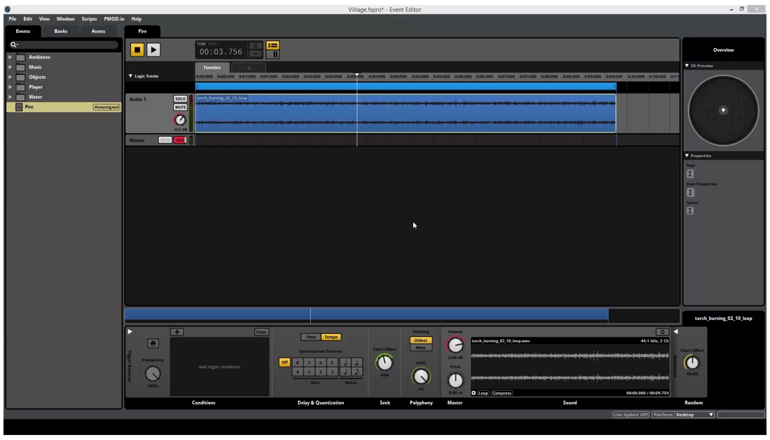


Figure 2.5: Example of start offset randomization of the clip. The offset is set to 45% with a modulation of 50%, which means that every time the sound is triggered, it will start from a random sample laying in between 20% and 70% of clip duration.

2.3.2 Transitioning between sound themes

The huge part of procedural audio transitioning that is performed in games today is motivated by the need to move from one sound scene to another in order to reflect more or less significant changes that occur in the virtual world.

Let us see how between-theme transitioning is handled in FMOD Studio. First of all, theme label is put at a certain position of the timeline to identify the moment of time where the playback of a particular sound scene should start. After that, a transition marker has to be added with the jump destination selected. With that done, the playhead will navigate to a corresponding label (i.e. to the start of a designated sound scene) upon reaching the marker. Instead of single labels, loop regions could be chosen as destinations for transition markers. Obviously, that requires those regions to be defined beforehand.

In praxis, it is unlikely that sound designers would want the transitions to happen every time a transition marker is met. During gameplay, it is often unforeseeable how long a certain sound scene will last (that is to say, only within-theme transfers will be taking place) and when it will shift to some other scene. For that reason, the game has to somehow communicate its current state to the sound system so that it knows when to allow a transfer.

This issue is resolved by enabling the adaptive music middleware to access values of determinative variables from the source code of the game or/and synchronized game engine (e.g. Unity, Unreal). The transition markers are now being provided with a list of conditions, with each condition defining acceptable values for observed variables. At the moment where the playhead reaches such marker, the conditions are checked and, depending on whether they are met or not, the playhead skips to a new position or continues moving unaffected.

In addition to that, FMOD Studio features the support of transition regions. While the playhead moves inside this type of region, the prescribed conditions for transition are being continuously tested each frame. A jump to a set destination is made immediately if all requirements for variable values appear fulfilled. That tool is especially helpful when those values are expected to change several times during the playback of an average-length clip, enabling audio system to become more responsive, instantly adapting to occurring in-game events.

A sample setup demonstrating the use of labels, markers, loop and transition regions is shown in figure 2.6.

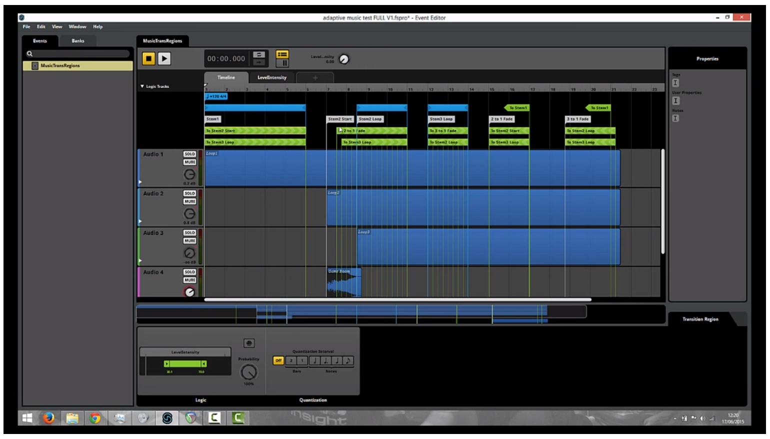
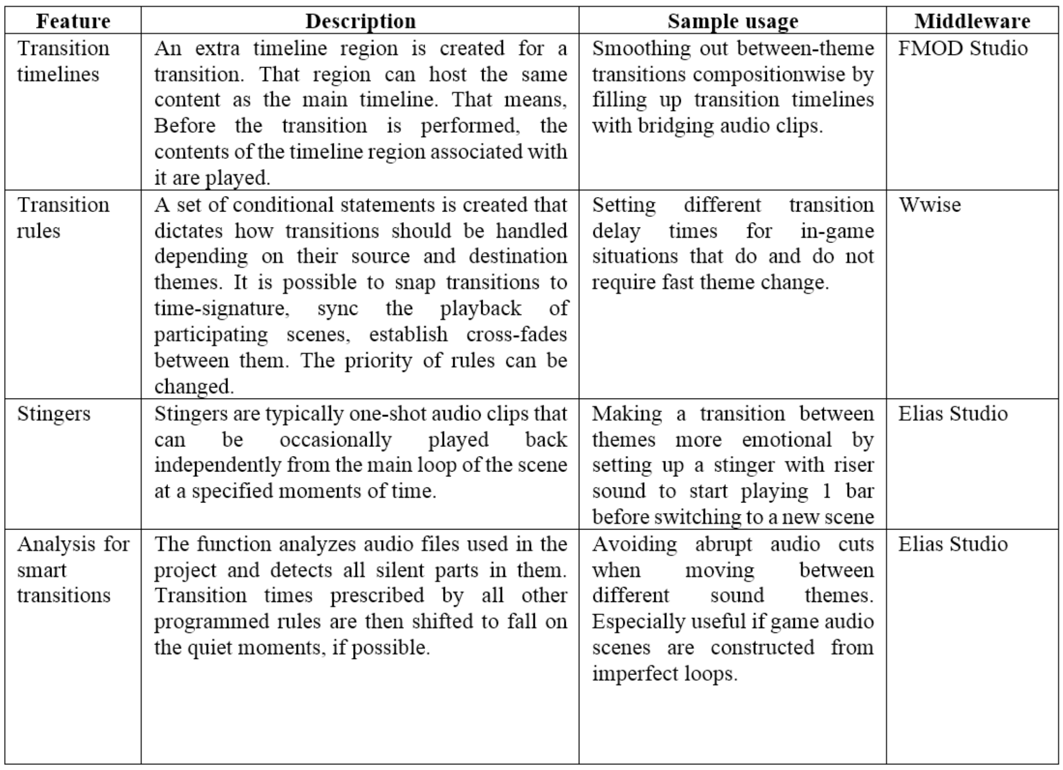


Figure 2.6: Example of using FMOD Studio tools for transitions within and between sound themes. The first row under the timeline ruler contains a time-signature-and-tempo marker. The second row hosts three loop regions and two transition markers. Six labels denoting the starting points for different sound scenes are placed in the third row. Each of the last two rows has five transition regions for instant playback transfer. The panel ”Logic” in the bottom displays the passing condition for transferring from ”To Stem2 Start” region – variable ”LevelIntensity” is expected to have a value in between 30.1 and 70.

The scope of existing techniques for non-linear playback can be further widened by the possibilities that each piece of adaptive audio middleware offers exclusively. An overview of some of this extra functionality is given in table 2.1.



Feature	Description	Sample usage	Middleware
Transition timelines	An extra timeline region is created for a transition. That region can host the same content as the main timeline. That means, Before the transition is performed, the contents of the timeline region associated with it are played.	Smoothing out between-theme transitions compositionwise by filling up transition timelines with bridging audio clips.	FMOD Studio
Transition rules	A set of conditional statements is created that dictates how transitions should be handled depending on their source and destination themes. It is possible to snap transitions to time-signature, sync the playback of participating scenes, establish cross-fades between them. The priority of rules can be changed.	Setting different transition delay times for in-game situations that do and do not require fast theme change.	Wwise
Stingers	Stingers are typically one-shot audio clips that can be occasionally played back independently from the main loop of the scene at a specified moments of time.	Making a transition between themes more emotional by setting up a stinger with riser sound to start playing 1 bar before switching to a new scene	Elias Studio
Analysis for smart transitions	The function analyzes audio files used in the project and detects all silent parts in them. Transition times prescribed by all other programmed rules are then shifted to fall on the quiet moments, if possible.	Avoiding abrupt audio cuts when moving between different sound themes. Especially useful if game audio scenes are constructed from imperfect loops.	Elias Studio

Table 2.1: Some of the middleware-specific features used for transitioning in adaptive audio

2.4 Audio synthesis

Loosely speaking, the notion of transitions can be seen as a higher-level or a bigger-scale version of sound synthesis, the term which describes generating new audio (electronic) signals in designated computer software from somewhat basic starting components provided. The overview of three types of synthesis that most resemble non-linear sample playback via transitioning is given in this section.

2.4.1 Wavetable synthesis

The idea of wavetable audio generation is grounded on the principle of looping within a given set of one-cycle waveforms [3]. The waveforms that form this set can be the four basic ones (sine, square, triangle and sawtooth waves, see figure 2.7) or of more advanced shapes morphed form those four primitives. When one or more waves are put in the set (named "wavetable"), the playback is composed from periodic reproduction of wavetable sonic contents such that the following parameters can be automated:

- The wave that is currently being played;
- The rules of switching between different table positions;
- AHDSR-envelopes of participating waves and behaviour of filters;
- LFO modulation and phasing settings.

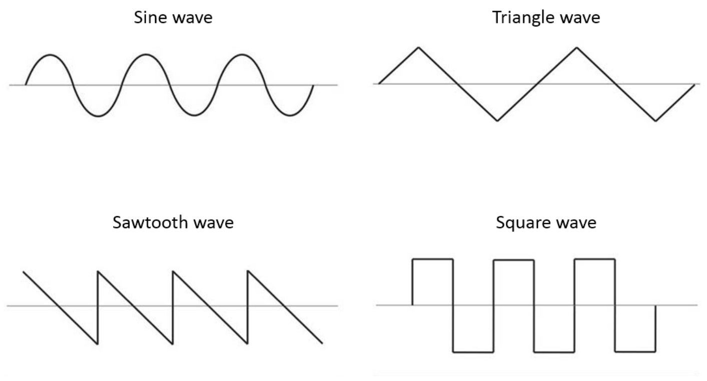


Figure 2.7: Basic waveforms used for wavetable synthesis.

The endless configuration variability that comes with all present controls allows for creation of very complex and rich sounds and timbres. Nowadays, software tools for wavetable synthesis mainly exist in the form of VST-plugins, each of those offers a unique set of instruments to regulate wavetable synthesis, independently modifying and randomizing the parameters of tables' contents over time. The most widely known plugins are Massive by Native Instruments and Serum by Xfer Records. Their user interfaces that demonstrate featured functionality are shown in figure 2.8.

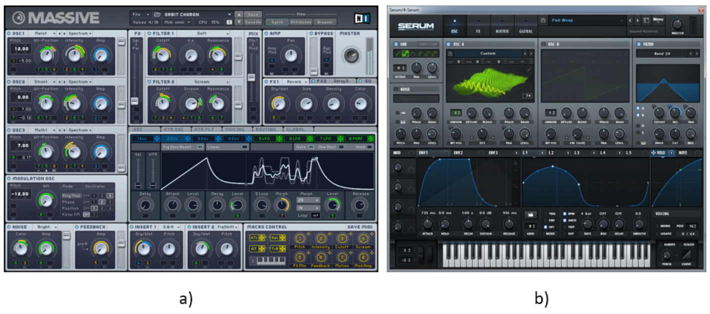


Figure 2.8: Popular VST-plugins for wavetable synthesis: a) – Massive, b) – Serum

2.4.2 Sample-based synthesis

Sample-based audio synthesis is very similar to wavetable generation. The major difference is that instead of being constructed using basic waveform oscillators, new sounds are made by processing previously created, usually short files (not more than few seconds long) called samples (not to be confused with voltage values that form signal waves in audio sampling) [4]. Those files can be layered on top of each other and manipulated in a lot of ways, such as:

- Defining the static or dynamic fashion in which active playback samples are chosen;
- Specifying the content of samples that is played in time-frequency domain;
- Processing samples with distortion, flanger, reverb, delay and other effect plugins;
- Assigning AHDSR-envelopes to sounds;
- Establishing signal sends in between samples;
- Randomizing the parameters of controls mentioned above.

A good example of a sample-based synthesizer is Iris 2 VST-tool by iZotope. Its interface window is depicted in figure 2.9. In fact, this approach is so famous that most of the contemporary DAWs (like Pro Tools, Ableton Live, Studio One 3) already come with stock extensions intended for the reviewed type of synthesis. Those plugins are often referred to as samplers.

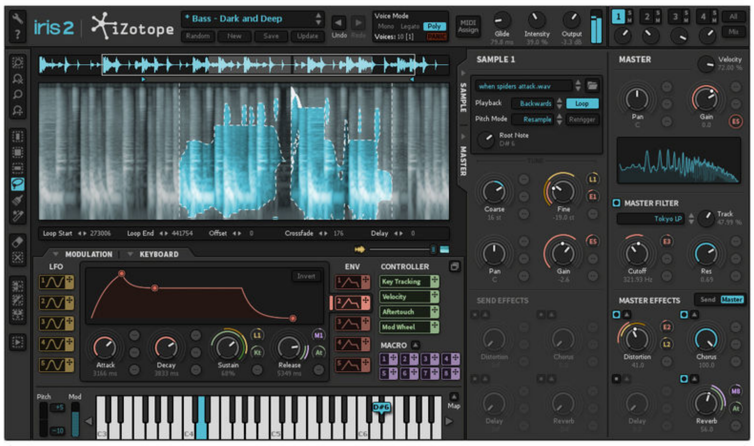


Figure 2.9: iZotope Iris 2 sample-based synthesizer

2.4.3 Granular synthesis

Just like sound generation based on prerecorded samples, granular synthesis operates over custom user audio files. However, the feature that makes this method special is that it works on a microsound scale. The source sample is initially divided into little parts – grains – of length of about 1 to 50 milliseconds [5]. From that point on, the resulting sound perceived during sample playback is defined by:

- The order and the speed in which the grains are played;
- Sizes and positioning of grains;
- Envelope and modulation grain settings;
- Added processing effects and their scope of reach (whether the whole sample or individual grains are affected).

This list can be continued with hundreds of controls that modern granular synthesizers have. Employing those mechanisms allows to both obtain timbers that finely alter the precepted characteristics of raw samples or transform them into something totally unlike. The interface layout of one of the popular solutions today – Quanta by Audio Damage – is shown in figure 2.10.

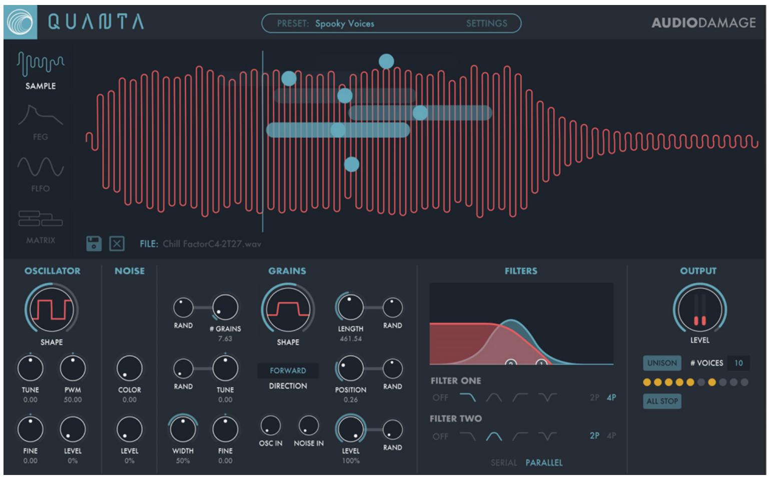
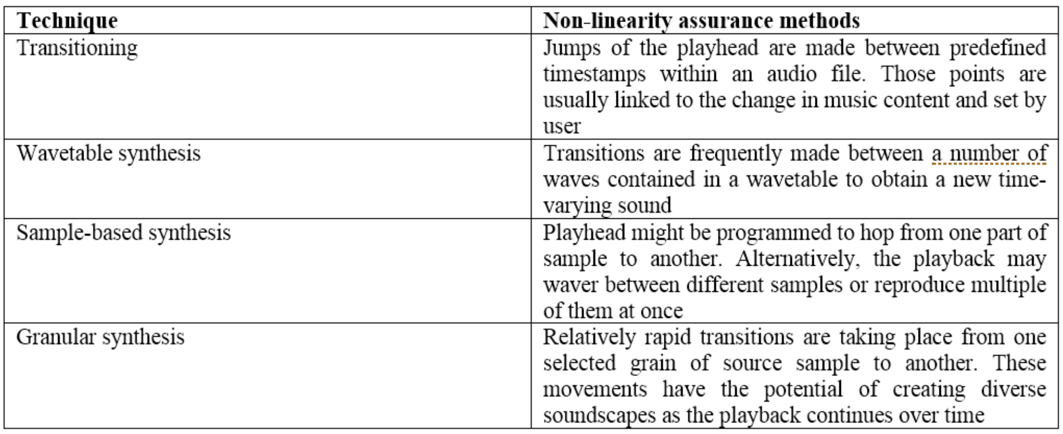


Figure 2.10: Quanta granular synthesizer

2.4.4 Audio synthesis for playback non-linearity

The three reviewed types of audio signal synthesis are closely connected to the concept of transitioning – as a matter of fact, all these techniques help to produce brand new audio sequences by interacting with basic sound entities. A brief summary of how considered synthesis types and usage of transitions bring non-linearity into playback is given in table 2.2.



Technique	**Non-linearity assurance methods**
Transitioning	Jumps of the playhead are made between predefined timestamps within an audio file. Those points are usually linked to the change in music content and set by user
Wavetable synthesis	Transitions are frequently made between a number of waves contained in a wavetable to obtain a new time-varying sound
Sample-based synthesis	Playhead might be programmed to hop from one part of sample to another. Alternatively, the playback may waver between different samples or reproduce multiple of them at once
Granular synthesis	Relatively rapid transitions are taking place from one selected grain of source sample to another. These movements have the potential of creating diverse soundscapes as the playback continues over time

Table 2.2: Non-linearity of playback in transitioning and audio synthesis

2.5 Recapitulation

Thus far, the methods of transitioning within and between sound themes and signal synthesis described in this section form a solid foundation for creative design decisions, notably when combined with each other and with other instruments and components. However, speaking of transitions, the one big issue that has not still been addressed is that all major IDEs for interactive audio provide means for context-aware transitioning, but do not ensure it by default. That means, for a sample generative audio system to operate in a natural, flexible and dynamic way, a lot of preliminary work has to be manually done inside DAWs and the middleware itself by music/sound producers and designers. These efforts include, but are not limited to:

- Producing seamless loops of audio for music arrangements;
- Setting up the procedural sound engine to conform to arrangement settings of time signature and tempo;
- Distributing audio clips between tracks and sound themes, aligning them properly with the bars and beat of the timeline ruler;
- Placing labels for sound theme regions;
- Defining the borders for loop regions and transition regions;
- Creating transition markers.

An attempt to make up for this deficiency is made in this work. The idea is to design and implement a system that could eliminate the need to assign transition points by hand. It should provide a method to automatically locate key samples inside of an input audio file, that could be later interpreted as transfer markers (or otherwise) in higher-level software.

The solution this master thesis centers upon is partially inspired by the "Smart transition analysis" feature that was introduced by the developers of Elias Studio (see table 2.1). While this algorithm solely aims to suggest transitions at least audible moments of the playback, the new approach being proposed attempts to detect moments that are similar perception- and frequencywise. That helps to increase the number of appropriate playhead jump locations and make the technique more robust to clips of diverse musical content (when Smart analysis in Elias Studio becomes useless when a processed audio clip has no or too few moments of silence).

Chapter 3

Analysis

The mechanism of transitions is crucial for state-of-the-art generative game audio design and is embedded into most of popular middleware titles. The investigation of capabilities that FMOD Studio, Wwise and Elias Studio have in that aspect shows that existing instruments for non-linear playback give one flexible enough control over the sound environment. This is achieved similarly in all the products of the triad: previously composed audio clips are packed in groups to form themes (or scenes) – complex soundscapes of a distinct mood and feel – when played together. Although implementation details differ, each of the IDEs under attention supports its own tools to help designers label different themes on the timeline, so that the system knows where they sit and how to get to a certain scene when it becomes necessary. The transitions are then handled in correspondence to preprogrammed logic, which is also highly adjustable due to presence of various controls and parameters and ability to link and/or randomize the behaviour of most of them.

The modern concept of between- and within-scene transitioning relies heavily on the positioning of border markers that constrain acceptable playback regions. Regardless of how labeling is realized in particular software, the common flaw persists – all the setting has to be done manually by a responsible person. This circumstance slows down the workflow and, when interactive audio system is sophisticated and has a lot of sound themes, may even make the creative process tedious and hold back the whole game development cycle.

This diploma thesis is concerned with compensating for this drawback by designing, implementing, and testing a digital signal processing (DSP) algorithm that could help automate the task of determining the placement of transition markers. This algorithm should take an input music file and provide means for locating the most preferred points in it that could be used as source and destination positions for both within-theme and between-theme transitioning.

Any kind of non-linear music playback is expected to maintain natural, musical flow, contain no abrupt audio cuts and cues that could reveal the artificiality of perceived sound. Taking that into account, setting up the desired required functionality comes down to discovering pairs of samples that would act as shortcuts to similarly sounding moments in an input file in the first place. Once such pair is registered, it may be denoted as either suitable for transitions between different parts ("themes" at a local scale) of the file, or appropriate for transfers inside a specific part.

3.1 Essentials of audio sampling

Before laying out possible technical approaches to the solution, it is important to understand the essence of how modern digital music is made and consumed by humans. When a sound recording is taken, the power of analog signal received from real world is sampled, i.e. measured with a defined frequency. Today's most common sampling rate used in audio CD music production is 44100 Hz [6]. The choice of such a number is based on the Nyquist-Shannon Theorem, which states that "every continuous-time signal can be sampled and perfectly reconstructed from its samples if the waveform is sampled over twice as fast as its highest frequency component" [7]. That means, since the highest limit of human's hearing range is around 20 kHz, it is necessary to capture the analog signal at least 40000 times per second to reach sufficient fidelity. A slightly more high-resolution standard of 44.1 kHz has gained its popularity due to historical reasons.

Capturing incoming signal that often allows analog-to-digital converters to accurately picture the geometry of sound waves within the full audible spectrum and store that audio data in a machine-readable format (see figure 3.1). When the recording is played back, the sound waves are reconstructed based on the sample data. Because human's hearing system is itself analog, we accordingly digest music and sounds in a continuous manner. We do not think of or discriminate between individual samples in the file – instead, we absorb what we hear by forming patterns, sequences and making out remarkable and somewhat persistent tones. A single sample in isolation contains literally no valuable musical information for a listener and is distinguished as a click (if heard at all). That means that picking out individual grains from the playback and comparing them between each other makes no sense.

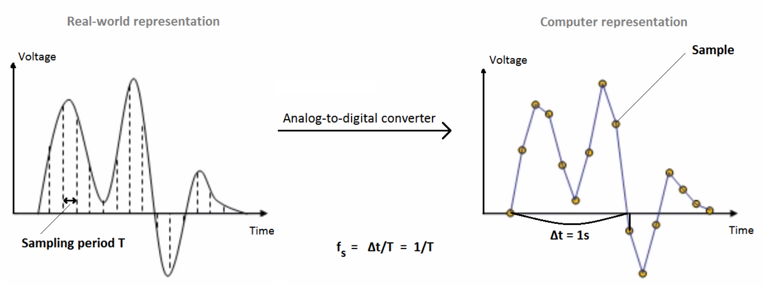


Figure 3.1: Sampling of audio signal. On the left side of the figure the incoming analog signal is shown, that is caused by air pressure sound waves hitting the sensitive surface of a recording device (technical details omitted here). Sampling period T shows the time difference between two subsequent voltage measurements. After going through an analog-to-digital converter, the shape of a sound wave is stored as a series of those snapshots (samples). Sample frequency f_s is then defined as $f_s = 1/T$. The greater sampling rate is, the better computer representation corresponds to the original signal.

On the contrary, a series of subsequently played back samples can be interpreted as a discretized sound wave reproduced with a certain degree of truthfulness and does hold meaningful data that can be processed. However, this data is quantitative, that is, it only has the voltage values changing over time. The analysis that is required for the desired functionality of target algorithm is interested in qualitative information, which is, in case of music and sound, frequency spectrum information. That means, there has to be a way to move between temporal and frequential domains.

3.2 Fourier Transforms

The modern theory of digital signal processing makes extensive use of a technique called Fourier Transform. It aims to project an analog waveform from time-voltage into frequency-magnitude coordinate space. To do so, it decomposes input signal into a sum of sinusoidal basis functions. Each of these periodical functions accounts for a certain frequency within the spectrum of processed sound (see figure 3.2) [8].

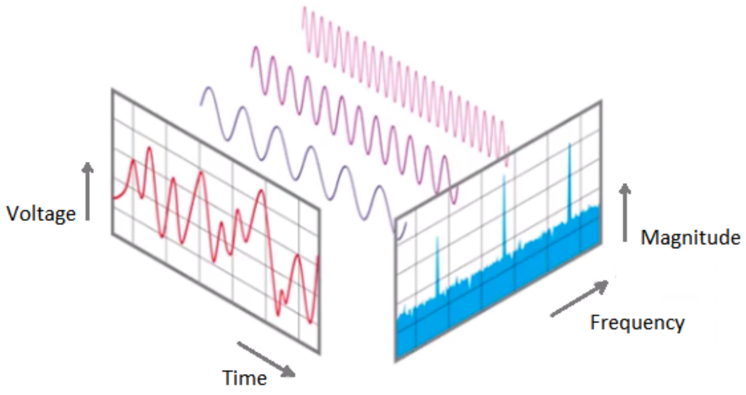


Figure 3.2: The function of Fourier Transform. A continuous soundwave is separated into sinusoid-type functions which are later associated with particular frequencies of a sound range.

When Fourier Transform is performed over a finite collection of equally-spaced measurements, such as over samples of a digital signal, it is referred to as Discrete Fourier Transform (DFT). Operating over a chain of samples, DFT is able to determine the frequencies that make up the total signal, as well as degrees of their contributions to it. While temporal dimension exists in the input data and plays an important role in the conversion, it is disregarded as the result frequency characteristic is formed. The spectrum appears to represent averaged tonal content over the duration of a sample sequence that was provided to the algorithm. It is worth pointing out that DFT is typically executed over a number of playback snapshots less than the sampling rate, which means the timespan of a processed chunk of audio is relatively small and can well be neglected. In practice, the size n of chunk is often chosen to be a power of two (e.g. 1024, 2048, 4096 samples), because this allows to carry out a special algorithmic approach of DFT which goes after the name of Fast Fourier Transform (FFT). That method is widely preferred due to being extremely fast and computationally cheap (with the complexity of $O\left(n \cdot logn\right)$) and hence can be easily run on virtually any relevant computer hardware [9].

It is important to realize, though, that no DFT solution is capable of providing volume information of every single frequency (there is simply an infinite number of them inside the spectrum). In other words, discrete time-to-frequency domain translation assumes granularity on both sides. The way FFT deals with this is by dividing the whole observed range (which is defined as half of the sampling rate) into equal frequency intervals – so-called "bins". Given N samples on input, FFT produces

$$n_b = \frac{N}{2} \tag{3.1}$$

values, each of which represents average magnitude for the corresponding bin. For instance, doing Fast Fourier Transform over a 1024-sample chunk taken from a recording that has been made with a sample rate ω_s of 48000 kHz will let one obtain

$$n_b = \frac{1024}{2} = 512 \tag{3.2}$$

spectrum values, each keeping the relative volume information of

$$\omega_b = \frac{\omega_s \cdot 0.5}{n_b} = \frac{48000 \cdot 0.5}{512} = 46.875Hz \tag{3.3}$$

of sound inside one of 512 bins. The total range represented in the output is from 0 Hz to 24000 Hz – the sampling rate in the formula is divided by 2 for that very reason.

With sample rate held constant, the increasing number of waveform measurements participating in FFT improves output resolution, as bins become narrower. At the same time, this causes the algorithm to average the produced spectrum image over a longer time period, which means a risk of missing important subtle changes in tone arises. At this case, the choice of input segments' length has to be fairly justified by the specification of a DSP task.

Let us now get back to the programming purpose of this work. The fact that applying Discrete Time Transform to short-time sample sequences provides a spectral fingerprint of somewhat momentary interpretation prompts that the smallest units of a sound file that are made subject to mutual comparison should be segments ("groups", "chunks", "fragments") of (subsequent) samples. Those chunks should be small enough to assure locality of output spectrum data.

3.3 Spectrum similarity metrics

Another aspect of the researched problem that can not be overlooked is defining the distance metric between two spectrum outputs. When an audio clip is being broken down into a collection of chunks, each separate segment can be seen as a distinct item with a number of attributes two times smaller than the number of samples used to form a chunk – those values are clearly frequency bin averages. There come three apparent ways of taking advantage of this data to design a measure for mutual spectrum arrays' similarity:

- Leave all the attributes intact and perform element-by-element comparisons between chunks, when the final metric might be the average, sum or the absolute maximum of all computed differences;

- Attempt to somehow exclusively encode the sequence of bin values as a whole, i.e. reduce the number of attributes to one;

- Derive a new, smaller set of attributes from the existing set according to a reasonable logic, so that the second collection of properties still captures most of the significant frequency information about the sound. After that, perform bitwise comparison of updated attributes.

3.3.1 Bin-by-bin comparison

The naivest fashion in which spectra could be set against each other requires no preliminary steps. Having two equal-length[1] bin arrays as input, the algorithm could individually find absolute (or squared) Euclidean distances for each pair of magnitude values that represent same frequency range. That is, the first bin of the first spectrum is compared to the first bin of the second spectrum, then the common array index is incremented by one and the process repeats until every pair of bins has its own distance score. As a final step, those scores could be added up[2] to obtain a single similarity value.

The major drawback of this strategy is its computational complexity. Obtaining similarity scores for all n bins among two spectrum arrays takes up time of $O(n)$. That is decently fast on its own, but the state of the matter is worsened by the following two circumstances:

- Since the fragments that the source audio file is being partitioned into are relatively tiny in size compared to the file duration, their number will be huge. Therefore, the processing algorithm would have to deal with loads of possible combinations of chunks to compare. For instance, a sixty-second-long music file ($t = 60$) with a sample rate of $omega_s =$ 44100 Hz split into non-overlapping segments of length $N = 1024$ will provide a total of

$$N_s = \lfloor \frac{\omega_s \cdot t}{N} \rfloor = \lfloor \frac{44100 \cdot 60}{1024} \rfloor = 512 \tag{3.4}$$

of those segments to process and

$$N_c = \frac{N_s \cdot (N_s - 1)}{2} = \frac{2583 \cdot (2583 - 1)}{2} = 3334653 \tag{3.5}$$

of comparisons to make [10]. Taking into consideration a number of bin values received from each chunk, the overall time complexity jumps to unsatisfactory

$$O\left(\frac{N}{2} \cdot N \cdot \frac{N-1}{2}\right) \to O(N^3) \tag{3.6}$$

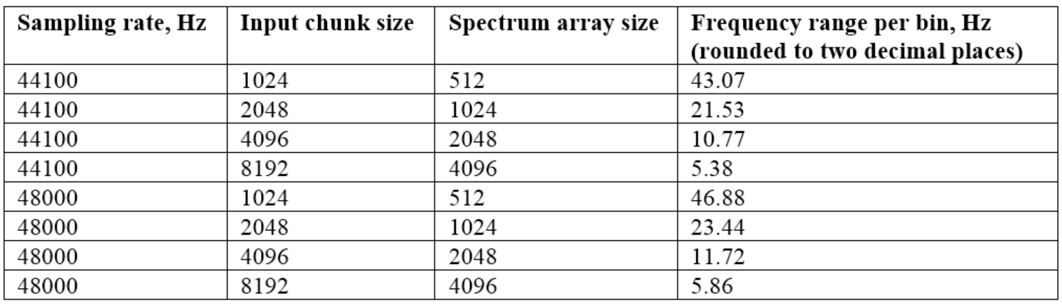
[1]Lengths are expected to match so that the values of the bins across both spectra can be grouped into pairs that relate to the same frequency content.

[2]Picking the maximum of individual bin difference scores as an ultimate similarity measure is going to cause misleading results. For example, two almost equal signals with just a single difference in a narrow frequency band could be identified as totally different due to the greatest distance value taking prevalence and not being compensated by close-to-zero ones. Calculating the arithmetic mean to normalize sum of scores over their quantity is redundant because the compared spectra are of same size.

- When putting sequences of audio samples into groups, we would typically want to do that in a way that neighboring groups share some of the voltage values, or, in other words, overlap in N_o samples. That helps to significantly increase the coverage of all sound wave shapes that can be observed during playback. Theoretically, the most robust move would be to make two subsequent segments differ only in one sample ($N_o = N - 1$). However, this tactic will cause too many chunks to be formed – the number will be almost the same as the total number of samples in a file. This degree of precision, luckily, is not needed for the purposes of current type of sound analysis. An overlap of 25%, 33% or 50% can well be chosen. When the chunks are formed with intersection, their total number N'_s increases substantially so that

$$\frac{N'_s}{N_s} \approx \frac{N}{N - N_o} \tag{3.7}$$

- In order to achieve finer granularity in frequency bin representation of spectra, bigger sizes of chunks could be chosen. Table 3.1 contains the information about some of the possible bin sizes.



Sampling rate, Hz	**Input chunk size**	**Spectrum array size**	**Frequency range per bin, Hz (rounded to two decimal places)**
44100	1024	512	43.07
44100	2048	1024	21.53
44100	4096	2048	10.77
44100	8192	4096	5.38
48000	1024	512	46.88
48000	2048	1024	23.44
48000	4096	2048	11.72
48000	8192	4096	5.86

Table 3.1: Possible sizes of frequency bins of a spectrum data obtained from FFT.

  When increasing chunk size, time and memory requirements of an algorithm based on element-by-element comparison would grow cubically, as it has been deduced.

The mentioned factors indicate that straight-up juxtaposing of spectrum values is inefficient.

3.3.2 Single-attribute encoding

The alternative approach to measuring similarities of spectra is reminiscent of one of the techniques in graph theory, where some graphs[3] can be represented via a unique string of zeros and ones called "certificate" – that string explains the structure of graph (i.e. how nodes are interconnected via edges) and can be "unwrapped" back into the corresponding network-like diagram [11]. The character-by-character comparison of such hashes is then done to determine whether the graphs are isomorphic 3.3. For the current problem, however, that kind of ciphering appears infeasible

[3]The solution is commonly implemented for trees but is sometimes modified to fit more complex types of graphs.

because all values inside spectral output are independent and do not fall into any hierarchy or other relationship that could be captured in a single entity.

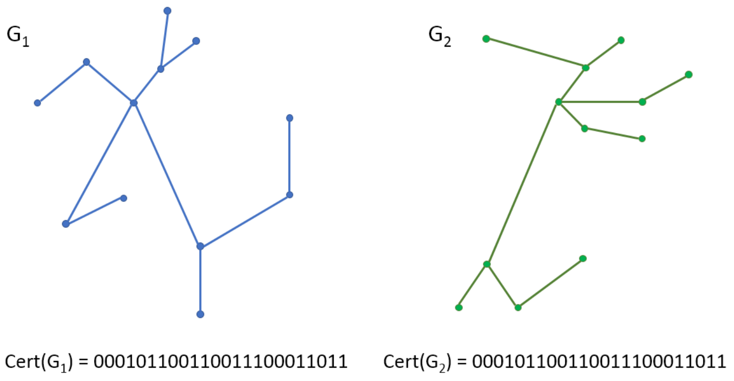

G1
G2
Cert(G1) = 0001011001100111000110111
Cert(G2) = 0001011001100111000110111

Figure 3.3: Using certificates for fingerprint encoding of trees. Even though graphs G_1 and G_2 look different, their certificates $Cert(G_1)$ and $Cert(G_2)$ completely match, indicating that the graphs can be considered as isomorphic (structurewise "equal").

The inspiration for one more tactic of single-attribute encoding roots in polynomial interpolation. Given a finite set of points in a 2D space, this method attempts to generate a polynomial function of lowest possible degree that passes through all the points and literally serves as a guideline to help restore the positions of missing data samples (see figure 3.4) [12] [13].

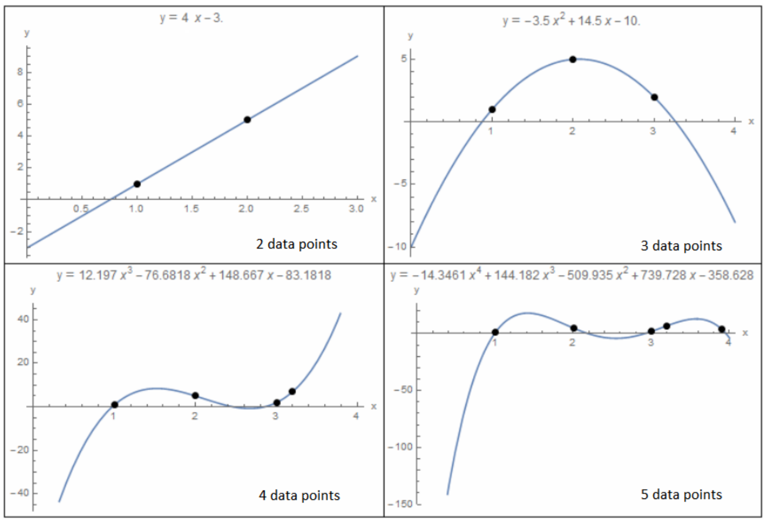


Figure 3.4: Polynomial interpolation over various sets of data points. The initial points are depicted in black, with the corresponding polynomials passing though them are shown with blue lines. The mathematical representations of derived polylines are given as functions $y = f(x)$

If we think of a set of frequency bin magnitude values of FFT output as a sequence of x-values of an unknown function, we might be able to perform polynomial interpolation over them to construct an equation that would correspond to a polyline fitted to include each source data point. Comparing two spectrums, we could then regard those functions as their signature attributes. We could come up with certain test frequency values that were not present in the source spectrum array, put them into two compared functions as abscissas and compare the resulting y-values. Again, we could accumulate acquired distance scores into a final value.

Unfortunately, this does not seem like a viable plan either. Even if the chunks of audio clip samples are fed to FFT by groups of 1024 and are transformed to 512 spectrum values[4], that would mean that target functions would have to be polynomials of degree 511. Calculating such function is a time-consuming task – one of the fastest implementations, Lagrange's interpolation, has a complexity of $O\left(n \cdot logn\right)$ [14]. Subsampling, i.e. using smaller quantities of points will lead to partial loss of frequency information and, furthermore, fitted polyline will deviate even more from the original spectrum shape. On top of that, the need to compute y-values for trial data would make the comparison cycle take even longer to execute. Decreasing the number of test values lessens the reliability of algorithm's outcome. Finally, the entire comparison code loop might have to be repeated a large number of times due to the duration of the source audio file (and, consequently, many combinations of chunks to count similarity degrees of).

The disadvantages of reviewed bin-by-bin and single-attribute comparison methods bring about the necessity of inventing another metric, that would both be easy to compute and use for the programming purpose.

3.3.3 Compressed-attribute entity

To compensate for the weaknesses and take advantage of the strengths of two previously discussed approaches to the comparison of spectral characteristics of sound fragments, a combination of these strategies can be employed. The idea is, using a foundation of frequency magnitude values, a special signature entity, a so-called "image", should be complied for each processed audio segment. The image should:

- On one hand, be compressed from indicative attributes and encapsulate all relevant spectrum data inside itself, so that the frequential snapshot of audio signal is represented with a high enough level of detail;
- On the other hand, be designed in a way that optimizes calculations of distance scores by providing the algorithm with means to minimize elementwise comparisons.

To satisfy the listed requisites, balance has to be found between the size of frequency data involved in building a spectral fingerprint and the significance of each stored value. By significance we mean how useful a certain data point is in helping to reflect audible differences between to chunks of audio. Surely, this is a subjective criterion and therefore has to have common-sense reasoning behind its definition.

It is well-known that human's hearing range spans from about 20 Hz to around 20 kHz. This interval is usually broken into seven frequency bands (see figure 3.5).

[4]Setting chunk size below that number (i.e. 512 samples or less) would make little sense as in that case each of 256 spectrum values will represent a bin of approximately 86 Hz (at a 44.1 kHz sample rate), which is too low of a resolution for music analysis – the difference is frequency between neighboring musical notes can be as faint as 9 Hz (see subsection 4.2.3).

Most of harmonic and non-harmonic sounds that we hear in everyday life are unevenly distributed across these bands. The sound energy that is transmitted with music and other types of sustained audio is generally concentrated within bass, low-midrange and midrange bands. As we move further to the right along the frequency axis, the signal would gradually become less prominent on average (see figure 3.6).

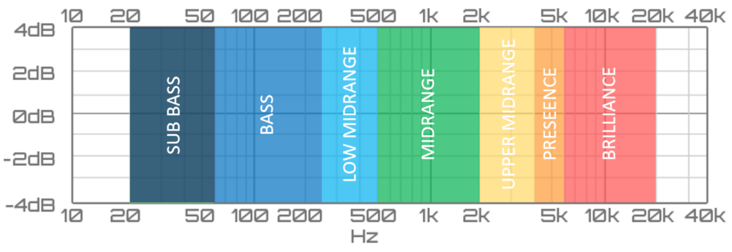


Figure 3.5: Seven frequency bands of human's hearing range.

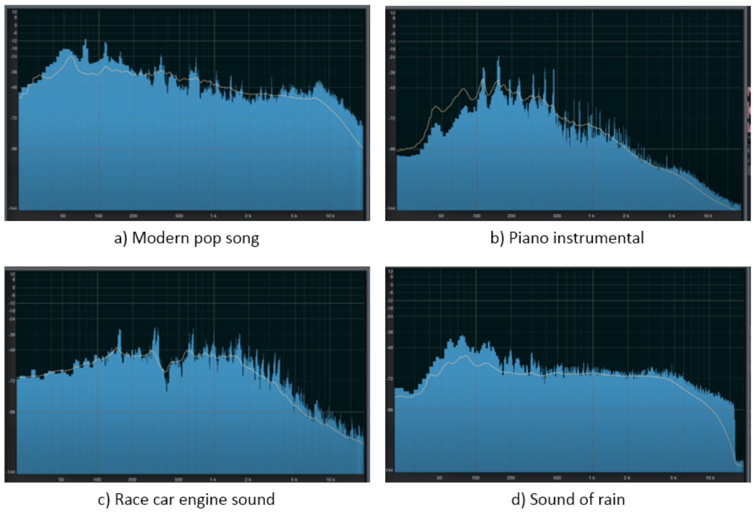


Figure 3.6: Spectral snapshots of some musical (a, b) and non-musical (c, d) sound files. The most of acoustic content is generally found in the region between 50 to 2000 Hz.

This fact leads to a conclusion that frequency bin values of an FFT spectrum output are of varying importance to the construction of spectral fingerprint. Specifically, that hints the following solution: only those bins that correspond to frequencies within most utilized range should be incorporated into the target entity intact. Those spectrum values that represent less noticeable frequencies can be combined with each other in groups of ascending size (as importance of a connected spectrum band goes down). Most of sub-bass-range and brilliance-range values of FFT output can simply be disregarded. This neglect is justified because adding such data to the segment characteristic will not allow to explain much extra difference between chunks – the compared segments will likely have almost matching, small magnitudes of border spectrum frequencies.

The technique described above makes it possible to considerably reduce the number of attributes that make up the identifying image of an audio chunk, while still capturing valuable tonal information about it.

The comparison of such images could be done in an elementwise fashion. After all pair bin distances are calculated and added up into a decisive score, there should exist a threshold value that would determine whether two segments are viewed as "similar enough" to be included into the set of possible transition points or not. This criterion should demonstrate persistent behaviour, i.e. no matter which content and volume input files might have, roughly same percentage of comparison pairs should be left out as not having passed the required limit. To achieve this, the minimum passing value for chunk alikeness can be defined as some small percentage of average magnitude of the entire audio file. This kind of normalization evens out the unwanted effects.

However, the circumstances that were reviewed in subsection 3.3.1 still apply to the "compressed-attribute comparison" approach. Processing of large amounts of sample fragments is quite memory- and time-demanding. For that reason, it seems preferable to impose a rule that would wisely reduce the number of comparisons. One of applicable techniques here is snapping future transition points to the rhymical grid of the song file. This strategy is addressed in the next section.

3.4 Beat-restricted processing of sample chunks

As it was discovered in the chapter 2 of this thesis, today's most relevant middleware for generative music mainly relies on rhythm-based transitioning. The jumps of the playhead happen at the moments when it reaches the start (or the end) of primary time signature measures – a beat or a bar. That convention was made to correlate with of the way that the songs and other pieces are consumed by humans. The natural habit of our brain to systematize the information we perceive subliminally causes us to detect beat patterns, and, speaking of transitions, to more likely expect the soundscape change at the beginning of a grid measure than anywhen else.

When applied to the current programming problem, the beat-constrained transitioning will improve the solution in two ways:

- The probability of unexpectedly sudden, unpleasantly sounding transitions will be decreased, because no pair of similar segments will be labeled as source and destination if the chunks are taken from randomly located timestamps;
- The algorithm's performance will be elevated, because the initial number of chunks to do the frequency analysis of and compare will be reduced dramatically – all samples that lay far from key timeline grid markings will be ignored when forming fragments.

To lower false positive rate[5] of transfer point selection even more, the processing engine might also require that each tuple of audio chunks that is involved in comparison contains segments that both relate to the same beat of the bar. That would ensure that, at an example of a 4/4-signature-song, if a hop is made from the end of beat four, the playhead always jumps to the start of beat one and no other.

[5]By false positive we mean two samples that were selected as acceptable source and destination for the transition, but empirically turned out to not provide smooth transfer.

Surely, to link transition moments to the rhythm of the played recording, the mechanism should have the information about where different signature markings are in the time domain of that song. We refer to a structure that stores such data as "beatmap". The creation of a beatmap and its application to the choice of segments is discussed in the chapter 4.

3.5 Findings

The analysis of the challenges that the algorithmic task of this thesis sets forth has shown that:

- The input file should be processed in subsequent, overlapping groups of audio samples of power of two, not individual samples. That would permit the use of Fast Fourier Transform, which helps obtain a frequential characteristic of a small portion of signal – a binned spectrum;
- An expressive compressed representation ("image") of the spectral array has to be created for each processed chunk. The image should be calculated using significant bin values taken from the FFT output. The number of attributes in a complete image should be considerably less than the number of spectrum bins to keep performance plausible;
- The ultimate similarity score of a pair of segments should be defined as a sum of Euclidean distances between corresponding values of the two images;
- The program should only form a list of acceptable transition points out of segments whose alikeness value is above a certain normalized threshold. That limit could be defined as some small fraction of an average loudness of the song (or of other akin metric);
- All allowed transitions should occur snapped to the rhythm grid of a music file. That should provide more organic and seamless, beat-preserving playhead transfers and save up computational and temporal efforts (out of many chunks the file is divided into, only a little number of relevant ones are selected).

A decision was made to use these ideas as the ground for the follow-up practical work. Firstly, that contained the development of GUI application, which demonstrates the operation of the transition points' detection algorithm on its own (chapters 4 and 5). Secondly, it included the creation of a simple 2D-game based on a problem-tailored API to showcase one example of how suggested technique could be applied to serve the purposes of adaptive game audio in custom projects (chapter 6).

Chapter 4

Algorithm implementation

Modern software solutions for adaptive game sound design are built upon the notion of non-linear playback of audio samples via transitioning. Along with the creative freedom that those IDEs provide for developers, they do not provide means for analyzing the structure and the content of project files. Hence, a lot of workspace configuration has to be done by hand for transitions to be smooth and rhythmical. To tackle this problem, a new algorithm is proposed in this master thesis that performs automatic detection of possible points for tempo-linked transfer and saves this data for future use. In this chapter, the technical choices assisting the implementation are explained first. After that, the details of the actual realization are discussed. Finally, an overview of a simple application that visualizes algorithm's functionality is given.

4.1 Technical Choices

The choice of software technologies and other tools used for the implementation of target algorithm is motivated by the programming objectives. As the main working environment, Unity was decided on. Unity is arguably the most popular 2D/3D/VR/AR game engine that a lot of indie developers and bigger studios turn to in their projects attracted by the broad range of capabilities it offers and a relatively steep learning curve it has [15] [16]. For the purposes of current implementation, the following features of Unity are important:

- The graphical components of a scene can easily be assigned scripts that would control the properties of on-screen objects. That comes in handy when creating a graphical user interface for the demo application;
- The IDE supports direct reading of audio clips and has built-in functions to retrieve sample data from sound files;
- All three most-favoured procedural music middleware titles – FMOD Studio, Wwise and Elias Studio – have officially supported plugins for integration with Unity. That may become useful in future, if the necessity emerges to quickly pass the information about transfer points from the developed application to the adaptive sound engine;

The primary scripting API (application programming interface) of Unity is written in C#, so the creation of custom source code is also done in that programming language. For working on scripts, any text editor fits, but to take advantage from workflow-boosting features like syntax highlighting, automatic code formatting and predictive typing, one has to look for specialized applications. The choice here was Visual Studio – the leading multi-language IDE in the field [17].

Even though Unity has native functions for accessing sample data of an audio clip, it is only able to get hold of the actual frequency content in real time [18]. The proposed solution entails preprocessing, so third-party had to be looked into. After investigating a number of libraries for magnitude-driven FFT analysis, a library named "DSPLib" was included into the project. The benefits of this framework remarkable for the scope of implementation task are [19]:

- The library is distributed under free MIT License;
- The package is lightweight and contains the basic methods for performing Fourier Transform and FFT, at the same time offering advanced flexibility in their parametrizing;
- DSPLib comes equipped with service functions that find average and root mean square values of spectra. These operations could be used to find values that are set as segment similarity threshold during comparison procedures.

When exectuing Fourier transforms, the library is working with complex numbers. The supportive package for that data type cannot be found inside of Unity, so it was downloaded from the official Microsoft GitHub repository [20] and configured to compile in the game engine.

The last piece of software that was occasionally accessed over the course of the application testing and debugging is the Studio One 3 digital audio workstation. This choice is of researcher's personal preference.

4.2 The algorithm

The final implementation of the target algorithm generally conforms to the guidelines laid down in section 3.5. Only several small modifications have been made to its parts as the source code was being written. All of the steps of the algorithms' pipeline are presented in the subsections below.

4.2.1 Calculating the beatmap

First of all, to correctly locate playback moments at which rhythm-based transitioning can take place, the algorithm first has to know where beats and bars start in a song, i.e. match different signature divisions with their time-domain positions. Correctly determining such parameters as tempo, number of beats per bar and number of beats per minute (BPM) during preprocessing is spoken of as Beat Detection. This problem is, in fact, considered to be currently unsolved[1]. For the purposes of

[1]Beat is a highly subjective perceptional characteristic of a sound and can not even be properly defined to computer systems. No unified model for detecting beat by preprocessing frequency

current programming task, it was decided the mentioned values will be taken from user as input. The choice of time signatures will be limited to two most popular ones today: 3/4 and 4/4.

When a song is fed to the algorithm, it might not start immediately at zero seconds, but a little later, instead. Alternatively, the beat might not kick in straightaway. Having key rhythmic parameters of the song, it is possible to calculate such an shift via frequency analysis with a consequent search of transients[2], but for the sake of keeping source code as simple and concise as possible, the offset value is obtained from the user as well.

Once provided with values for tempo, BPM and time signature, the algorithm fills up the beatmap, which in memory is represented as an array of timestamp values, where each corresponds to a starting point of a subsequent beat. This information is used in the next step when grouping series of samples into chunks.

4.2.2 Forming audio segments

In the current implementation, chunk size of 4096 is chosen. This allows to get a spectrum of length 2048 and, with a standard sample rate of 44.1 kHz, a binning resolution of 10.77 Hz per value (see table 3.1). This range is small enough to capture most of the differences in magnitudes of frequencies relating to some of the closest neighbouring musical notes (see next subsection).

The segments are formed starting from the first sample of an audio file. They overlap with a fraction of 25% – that means that the first fragment is collected from samples 1 to 4096, the second one contains samples 1025 to 5120, etc. If at the very end of the file there are not enough samples to compose a chunk, they are disregarded.

However, not all assembled groups enter the comparison. Here the previously created beatmap comes into play. The algorithm finds the time-domain borders of each chunk and determines whether any of the rhythm-linked timestamps lay within this interval. Only those segments that fall close to the moments where beats and bars start are sent to FFT analysis, where their voltage values are converted to spectra. What is more, each qualified chunk is assigned a number, which signifies the within-bar beat it belongs to. This is used at later stages to establish beat-preserving transfers.

4.2.3 Creating segment images

As it was defined in subsection 3.3.3, an image of a segment is a set of indicative attributes that is derived from the spectrum array and that captures most of the significant frequency information about the chunk. In the implementation, an image of length 64 is constructed from 73 key bins of each FFT output in the following way:

analysis currently exists, because in different music compositions the beat is sonically different, i.e. created using different instruments, patterns and tricks. Employing machine learning to develop and train such a model is a hard challenge as well, because the behaviour and the appearance of beat within the song is unpredictable. However, this topic is beyond the research scope of this master thesis.

[2]In acoustics and audio, a transient is a high-amplitude, short-duration sound at the beginning of a waveform that usually occurs in musical sounds in the beginning of a new time measure [21]

- The first characteristic 62 values are copied from bins as-is;
- The prelast image value is calculated as an average of the next three key spectrum values;
- The last image value is received as an average of eight last selected bins.

The indices for taken spectrum values are picked so that the represented ranges cover the frequencies of one or two of musical notes. In fact, the technique captures all notes from D3 to D7 individually. As FFT provides poorer resolution in lower ranges, the investigated bins are distributed unevenly across the whole audible span – most of the elected measurements sit very densely in the bass, lower-midrange and midrange bands, getting sparser towards the upper-midrange and presence regions (see figure 4.1). The two values at the end of image are bin averages due to those bins being located in brilliance band of human's hearing range – there is no need for a great degree of precision there. The information about selected spectrum values and their contributions to a segment image can be found in appendix D.

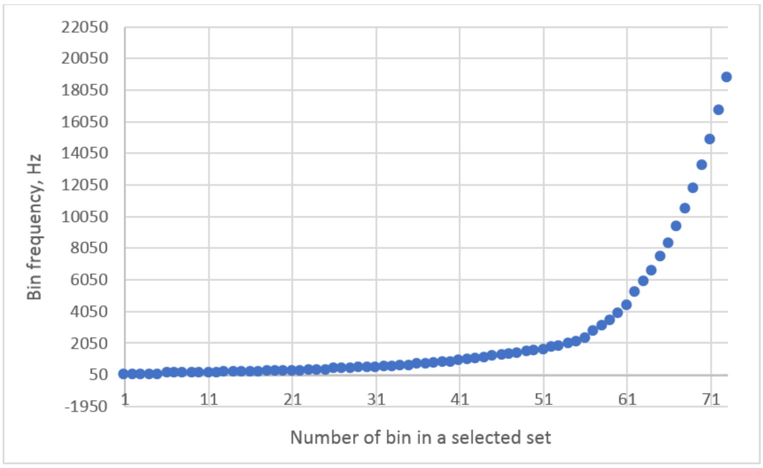


Figure 4.1: The distribution of frequency bins involved in image compilation across the spectrum.

4.2.4 Comparing segments

After images for participating chunks are calculated, all possible pairwise combinations of segments are subject to comparison.

The difference metric between two fragments is found by computing element-by-element Euclidian distances between the same-index elements of two segment image arrays and adding up those distances. To classify chunks as similar or unlike, their comparison score is set against a threshold T_s. It is defined as

$$T_s = RMS \cdot f_{RMS} \tag{4.1}$$

where RMS is a root mean square magnitude value of all spectra received by FFT on the segment formation step, and f_{RMS} is a fraction from zero to one.

If the dissimilarity score of a certain couple of fragments appears to be below T_s, the pair is thought to have met the requirements of likeness and proceeds to the alignment step, otherwise the tuple is left out of the further operation.

The greater the value of f_{RMS} is, the easier it is for segments to fit under the threshold and be considered appropriate for transitioning, and vice versa. Making the fraction too big would significantly increase false positive rate. On the other hand, setting f_{RMS} overly small can decrease the number of qualified segments remarkably (which will result in a more time spent waiting for an initiated transition) or bring it down to absolute zero. It was practically discovered that the best outcome is achieved when giving that multiplier a value between 0.01 and 0.3. For such range, the algorithm typically eliminates 94% to 99% of all possible transitions, but the rest bit is enough for more than satisfactory performance. The effects that different choices of f_{RMS} has on transfer points detection are proven in chapter 5.

4.2.5 Precise segment alignment

The alignment of transition segments is an extra step that was not originally planned for implementation but came out as profitable during the practical work. The core idea of this move is to refine a pair of source and destination transfer points to correspond to the exact same locations within the duration of a bar. The thing is, the size of chunks that are compiled at the start of the algorithm is not at all related to a beatmap of the song. Therefore, is very unlikely for the first sample of each observed chunk to match the exact signature measure on the timeline. The segments do belong to certain beats – but it does not entail they perfectly line up with them.

When a couple of similar fragments enters the step, it is first computed by how many samples and in which direction the source segment deviates from its ideal prescribed-by-beat placement. Then, the destination chunk is aligned in relation to its closest beat the way that the shift becomes the same among fragments' starting points.

Let us think of an example of two segments passed to the alignment stage: the first one is linked to beat 3 of bar 1 and begins 1100 samples before that beat, and the other one lays 100 samples before beat 3 bar 9. That tiny desynchronization might seem negligible, however, when trying to attempt to transition between the chunk starting points that are left intact, one would be disappointed, as human's ear would be extremely sensitive to such rhythmical inconsistency and the transfer itself would be perceived as abrupt and unnatural. Whereas if the timing of second segment is adjusted so that its leftmost value is also 1100 samples early relative to beat 3 bar 9, then the transition would sound much more pleasing.

4.2.6 Transitioning

As soon as akin segments are aligned, their two leftmost values become the source and the destination for a viable transition and are put in a separate data structure – we will denote is as "transfer table". It is completely up to the user how and where to take advantage of the algorithm's final output (the discussed realization of the programming task provides means to save obtained data into a Comma-Separated

Values file[3]). Below, one possible usage is suggested that can be experimented with in the test application.

Let us assume the audio clip is playing, and at that moment the user comes up with a certain position they want the playhead to jump to. That serves as a command to the program to access the transfer table and search for an appropriate pair of samples to jump between. In the implementation, the determination of suitable pair is done as follows:

- The location of the playhead at the time of initiated transition is captured;
- The algorithm scans source samples in the transfer table and maximally picks the predefined number of those (in our implementation it is equal to 50) that are in front of the playhead and is closest to it. If no entry points are found between the current playback marker and the end of the audio file, it is assumed that no transitioning path is feasible;
- If a source sample is found, a list of all possible hops from that sample is examined to find all of those that would appear before the predefined destination point and, again, would be the least distant from it. If at least one such exit point is spotted, the algorithm saves found entry and destination samples to the data structure of possible transfer options;
- The list of possible options is scanned to obtain N_w value for each of the selected pairs $< s_{entry}, s_{exit} >$. This value shows how many samples will have to be played ("waited for") before the playhead gets from the current playhead position to the required destination with a transition that the pair specifies. It is defined as

  $$N_w = (s_{entry} - s_{curr}) + (s_{dest} - s_{exit}) \tag{4.2}$$

  Where s_{entry} and s_{exit} are numbers of transition's entry and exit points, s_{curr} is current playhead position, and s_{dest} is the destination sample;
- The pair with a minimum value of N_w is passed to the audio player as the preferred transition. In its turn, the player moves the playhead to the second sample of a tuple once the playback reaches the value of the first sample of the pair.

In the test application, both forward and backward transitions are supported. The placement of transition samples calculated by the coroutine above is shown via application GUI.

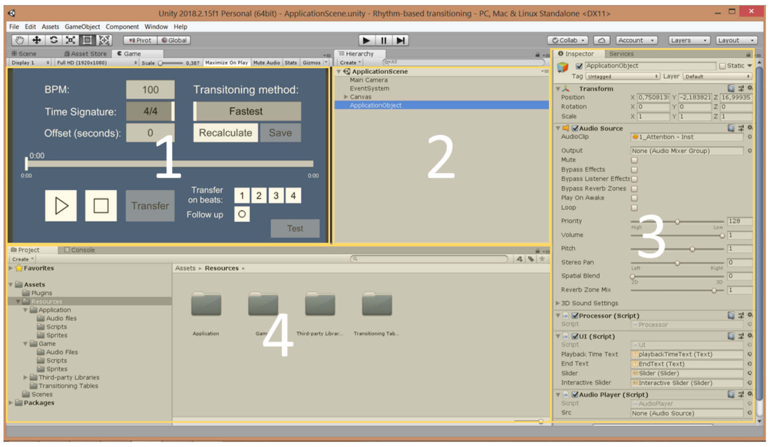
4.3 The application

This section covers the aspects of how the programming solution is encapsulated into a sample GUI application within Unity game engine.

[3]Each pair of entry-exit samples for transitioning is placed on a separate line and the values are split by commas.

4.3.1 Unity workspace

The window of the application project open in Unity with its main components labeled by numbers is shown in figure 4.2.


Unity 2018.2.15f1 Personal (64bit) - ApplicationScene.unity - Rhythm-based transitioning - PC, Mac & Linux Standalone <DX11>

BPM: 100
Time Signature: 4/4
Offset (seconds): 0
Transitoning method: Fastest

ApplicationScene
Main Camera
EventSystem
Canvas
ApplicationObject

ApplicationObject
Tag Untagged
Layer Default
Transform
Position X 0,750813 Y -2,183821 Z 16,99935
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Audio Source
AudioClip 1_Attention - Inst
Output None (Audio Mixer Group)
Priority 128
Volume 1
Pitch 1
Stereo Pan 0
Spatial Blend 0
Reverb Zone Mix 1
Processor (Script)
Script Processor
UI (Script)
Script UI
Playback Time Text playbackTimeText (Text)
End Text EndText (Text)
Slider Slider (Slider)
Interactive Slider Interactive Slider (Slider)
Audio Player (Script)
Script AudioPlayer
Src None (Audio Source)

Figure 4.2: Solution project window in Unity: 1 – GUI window, 2 – Object Hierarchy, 3 – Properties tab, 4 – Project explorer / console tab

The graphical user interface window (figure 4.2, label 1) is the main area where the behaviour of application is visualized. All UI elements are described in detail in subsection 4.3.3.

The hierarchy tab (figure 4.2, label 2) shows the list of all (groups of) objects that are created in the scene. The current implementation has four objects:

- Main Camera – a service component that provides the background for the user interface layout;
- Canvas – a parent entity for all interactive and static UI objects that are visible in the application;
- Event System – a compulsory helping object for the canvas;
- ApplicationObject – the object that ensures the main functionality of the application. All source code scripts are attached to and executed on this entity. That way, it acts as a bridge between the raw algorithm and its actual realization.

Each of listed objects has a list of properties, that can be seen and edited using Properties tab (figure 4.2, label 3). That workspace area is also used to provide an input music file to the processing pipeline.

The File Explorer tab allows to navigate through the hierarchy of project files and drag-and-drop necessary ones into objects' property fields. At the same window, the console can be accessed where compilation/runtime errors, warnings and debugging messages appear (figure 4.2, label 4).

4.3.2 Application files

As displayed in figure 4.2, the project assets folder consists of three main directories. Plugins folder has only one Numerics package file inside that is needed to support computations involving complex numbers. In the Scenes catalogue, there are two files – they store the vital information about the application scene and the demo game scene (see section ref6), such as default object layout and initial visualization preferences. The directory Resources contains three subfolders. The first two contain custom-created or imported user files that support the functionality of the developed application and the demo game. The third one is named Transitioning Tables. The CSV files compiled by Processor to save the information about transfer point locations for future use are put inside that subfolder.

Inside the Application catalogue, the following contents are packed:

- The core of current realization – C# scripts reside in the Scripts subfolder. In total, three source code files are linked to the ApplicationObject. The first one defines class Processor, which is responsible for executing the proposed algorithm and providing the required output – a set of appropriate transition points. This information and several other calculated parameters are handled in AudioPlayer class to control the playback, both casually and in a non-linear manner. The UI script is in duty for establishing mutual connection between the data model of the solution and its visual representation. On one hand, this class adjusts the application appearance in correspondence to changes in in-program variables and audio playback. On the other hand, it provides means to manage user interactions with GUI such as tweaking of settings by sending response commands to Processor and AudioPlayer;
- Some of the custom icons and images used to build user interface are placed in the Sprites subfolder;
- The Audio Files subdirectory host various music files that were used over the course of implementing the solution for debugging, testing and that can be used for demonstration purposes;
- The CSV files compiled by Processor to save the information about transfer point locations for future use are put into the subfolder.

The brief description of all programming methods utilized in the application within Unity scripts is given in the appendix E.1. The source code itself is in the Project folder on the CD provided with a hard copy of this master thesis. The contents of the CD are listed in appendix B.

4.3.3 Application GUI

The graphical user interface of the application serves multiple purposes:

- Helps control the linear playback of an audio clip and observe current playhead position;
- Gets BPM, time signature and beat start offset values from user's input that the main algorithm needs to fill up the beatmap;

- Presents the user with an opportunity to specify desired transfer destination, see the calculated entry and exit points, trigger and cancel transitions;
- Allows the user to modify default non-linear transferring behaviour by manipulating advanced controls.

All elements of GUI are shown in figure 4.3 and thoroughly described in table 4.1 according to how they are numbered in the figure.

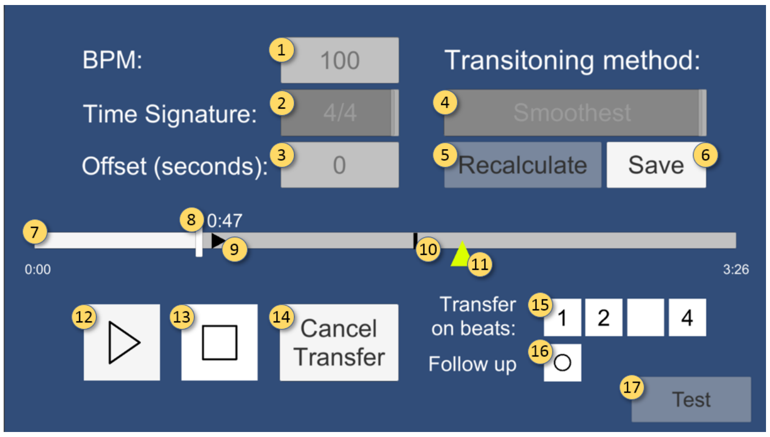


Figure 4.3: Elements of application GUI: 1 – BPM field, 2 –Time signature slider, 3 – Beat offset field, 4 – Transitioning method slider, 5 – Transitions' calculation button, 6 – Transition data export button, 7 – Playback bar, 8 – Playhead, 9 – Transition entry point marker, 10 – Transition exit point marker, 11 – Transition destination marker, 12 – Play/pause button, 13 – Stop button, 14 – Transfer trigger/cancel button, 15 – Transitioning beats' toggles, 16 – Follow-up mode toggle, 17 – Perception test mode button.

Element label number	Name and type	Possible values/modes	Description	Remarks
1	BPM Field	Any decimal number, 5 characters max	Takes user's input as the number of beats per minute for the song	The value cannot be changed if calculation of transition points has been done at least once
2	Time signature slider	3/4 or 4/4	Sets song's time signature to one of the two supported options	
3	Beat offset field	Any decimal number, 5 characters max	Takes user's input as the number of seconds that the first bar of the first beat is shifted for from the start of audio file	
4	Transitioning method slider	"Smoothest"/"Balanced"/ "Fastest"	Determines the value of f_{RMS} that forms a qualifying threshold for segment similarity score	The slider is not interactable during (paused) playback
5	Transitions' calculation button	Enabled/disabled	If pressed the first time, runs functions that create the beatmap, select segments for comparison, and compute the transitioning table. Otherwise, only transitioning table is recalculated	
6	Transition data export button	Enabled/disabled	Exports the most recently calculated transitioning table into a CSV file.	Interactable only if the transitioning table has been computed at least once
7	Playback bar	–	A graphical representation of a song in a time domain. Hosts the playhead and transition markers. Displays the duration of the audio file next to its right bottom corner	–
8	Playhead	Any point of the playback bar	Shows the playback progress. Leftmost position corresponds to the start of the audio file	–
9	Transition entry point marker	Any point of the playback bar	Signifies the point upon reaching which the playhead will jump to the exit point marker	Only shown if a transfer has been triggered, disappears once the transition is performed or cancelled.
10	Transition exit point marker	Any point of the playback bar	Signifies the point to which the playhead will transfer upon reaching the entry point marker	
11	Transition destination marker	Any point of the playback bar	Specifies the position to which the playhead should get non-linearly. Set by the user by clicking at a corresponding place of a playback bar.	–
12	Play/pause button	Play/ Pause	Starts or pauses linear playback of audio file samples	–
13	Stop button	–	Stops the linear playback of samples (the playhead is returned to its leftmost position)	Cancels any ongoing transitions and removes transition markers
14	– Transfer trigger/cancel button	Transfer/ Cancel Transfer, Enabled/disabled	Initiates the search of appropriate entry and exit points for a transition and draws the transition markers if the transition is found. If the transition is in progress, cancels the transfer and removes all the markers.	Can only be interacted with if the transitioning table has been computed at least once and the destination marker is set
15	Transitioning beats' toggles	On/Off, Enabled/Disabled (toggle 4)	Restrict the AudioPlayer class to only attempt to perform transitioning on specific beats.	If the time signature is set to 3/4, the fourth toggle becomes inactive.
16	Follow-up mode toggle	On/Off	If this setting is on, then the first playhead jump immediately triggers the search of another transition if the exit point of the initial transfer appears to be further than 1 second away from the destination marker. This continues until either the destination is reached, or no subsequent hops are found. If the setting is off, follow-up transitions are prohibited	–
17	Perception test mode button	On/Off	Enables a special playback mode used in perception testing, where the transitions to random audio file locations are detected at performed (see section 5.1)	Can only be interacted with if the transitioning table has been computed at least once

Table 4.1: Description of application's GUI elements

The user manual explaining the setup needed for interacting with the application is given in appendix section C.1.

Chapter 5

Testing

One of the most important concerns of this master thesis is design and implementation of the algorithm that would serve as a foundation for automated context-aware transitioning utilized in adaptive music systems for games. In order to evaluate the proposed programming solution for correctness and robustness, testing must be done. The choice of testing methodology can be motivated by one of the initial principles of procedurally generated audio – the flow of musical composition should remain steady, smooth and natural. In other words, none of non-linear playback jumps should not be noticeable to the listener. Those performance characteristics are highly subjective and therefore evaluation may show contrasting results when made by different people. Taking this circumstance into account, it looks reasonable to set the focus of the testing on the perceptional aspect.

The perception tests are carried out in order to accomplish the following goals:

- Assess the accuracy (”invisibility”) of transitioning provided by the suggested technique;
- Evaluate the robustness of the proposed method on different inputs and outline the recommended scope of use;
- Detect if the algorithm is subject to anomalistic behaviour and, if yes, determine the factors that cause such exceptions;
- Inspire prospective improvements to the realized technique.

5.1 Perception testing

5.1.1 GUI application’s testing mode

To perform the perceptional evaluation, a special testing mode was introduced to the GUI application. It is aided by extra functions added to the target application.

When testing mode is enabled, the following happens:

- The playback of an input file is launched from the beginning;

- The algorithm is forced to randomly set the desired destination and perform a follow-up transition sequence[1]. If it happens that there is no viable pair of transfer points, another generation attempt is made after a short delay and the search is repeated;
- Once a relatively big number of jump sequences are complete (for the purposes of current testing, a value of 10 was chosen), the playback is stopped;
- The total number of transfers made is saved for future reference.

That way, each of the musical compositions being constructed by dedicated methods in the application comes out as a random compilation of parts of a source clip. If we would suppose that the proposed transfer point detection algorithm was perfect, it would mean that audible sound created during each test trial would be perceived as a new cohesive composition with no abrupt cuts or other unpleasantly sounding artifacts. The examination of the algorithm's quality comes down to determining to which extent this assumption holds in praxis.

5.1.2 Procedure

For each testing session, 18 trials were made. During each of them, the implementation organized non-linear playback via transitioning inside one of six sonically different music files across the three different settings of the segment similarity threshold T_s (defined in subsection 4.2.4).

To decrease researcher bias, the solution was tested on five participants (aged 19 to 25). Each of them was suggested to listen to each of the 18 generated playback sequences played in a random order and, at the end of each audition, to state how many times they noticed that "something was wrong with the playback". The task was formulated that informally because it neither restricts the participants to lock their attention on a narrow set of characteristics of presented audio nor overwhelms them with any kind of sophisticated criteria. The only thing that matters for the evaluation is how smooth the playback is and how "invisible" the transitions are to listener's ears.

The number of perceived transitions was set against the actual number of transfer occurrences (this value was written down for each trial run at the end).

5.1.3 Results

The results of testing along with the information about the original music files used for test recordings are shown in table 5.1.

[1]Follow-up transitioning entails a series of non-stop playhead transfers. In the developed application, enabling this setting allows to reduce the time needed to get to the selected timeline destination.

№	**Test sequence source file description**	**BPM**	**Time-signature**	**Perceived / actual transitions during a test trial**														
				Loose threshold (f_{RMS} = 0.3)					**Moderate threshold (f_{RMS} = 0.125)**					**Strict threshold (f_{RMS} = 0.01)**				
				Participant number														
				1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
1	Bass-driven pop song instrumental	100	4/4	8/17	9/17	8/16	10/16	8/17	6/15	6/16	5/15	7/15	5/14	4/15	5/15	4/14	6/15	4/16
2	Brass-band upbeat theme	134	4/4	3/10	2/10	4/11	4/10	5/10	2/8	2/9	1/7	3/8	2/9	–	–	–	–	–
3	Grand piano ballad	90	4/4	10/21	9/20	10/20	13/21	11/20	8/22	9/22	10/20	11/21	9/21	4/15	5/16	3/14	6/16	3/14
4	Classical crossover score	124	3/4	9/21	9/21	9/19	9/19	9/20	8/15	7/16	7/16	7/16	7/17	4/15	5/15	7/15	5/14	6/17
5	Cheerful minimalistic piece	82.5	4/4	5/10	4/11	5/11	6/10	5/10	1/3	0/3	0/4	1/3	1/4	–	–	–	–	–
6	Fast action orchestral music	135	4/4	1/8	2/7	2/7	2/6	1/7	1/5	1/5	0/5	2/4	1/5	–	–	–	–	–

Table 5.1: Perception test results

Generally, the algorithm has shown a good level of transitioning accuracy. As it is seen in the table, the percentage of perceived transitions generally tends to decrease as the value of threshold for transition selection gets lower. On the other hand, the number of occurring transitions decreases as well. In our testing set, songs 2, 5 and 6 turned out to be the ones for which the least number of viable transitions was discovered by the implemented technique. On the strictest f_{RMS} setting, no transitions were found at all over the course of corresponding trials for those recordings. One possible explanation for this behaviour is that those pieces where the songs that were not produced professionally but were made by amateurs that may have overlooked the procedures of record mastering and normalization. That circumstance caused the comparison technique based on root mean square value (RMS) to detect considerably smaller amounts of similar segments.

Surprisingly, the implementation performed best on file 1, which is an instrumental of a dense pop mix. On average, three transfers out of four were completed absolutely unnoticed by the participant. After auditioning test sequences made up from files 3 and 4, some of the listeners mentioned that the only wrong thing they heard during playback were short clicks, and that ruined the experience. This might have happened because those scores involve piano and strings as instruments, and hence contain many segments where a sustained sound is playing. In that case, a little mismatch in a soundwave dispositions (which is natural for live string instruments) may have come out as unpleasant playback artefacts. One way to remove the clicks or make them less prominent would be to perform transitions with crossfades. Instead of doing sharp jumps between the moments of audio within the same clip, the game audio engine could be set up to move the playhead between two identical music files. To make a transfer, the playback of currently active file is gradually zeroed out, while at the same time the volume of the second clip is increased. At the next transition, the second clip softly blends back to the first and so on.

Another pleasing finding is that the algorithm does not discriminate between music of different tempos and two of the supported time signatures – its operation is not affected by these settings. This means that the beatmap calculation is being done correctly and the integration of support of various time signatures is just a matter of small code modifications.

The conducted user testing has inspired two possible improvements to the algorithm that would help further improve the seamlessness of transitioning. First, when the chunks are compared, it is not only their spectra similarity that may be inspected but the ratio of overall signal magnitudes as well. This measure would eliminate the cases when the transition is perceived as a rapid change of playback volume. Moreover, a value of acceptable magnitude difference could be made dependent on the frequency that it is measured at. That way, for example, midrange and upper-midrange bands may be assigned less strict thresholds, as each piece of music is commonly more stable in bass range and more diverse in higher frequencies. That would allow the playhead to jump between different music themes revolving around similar low-frequency content.

Another prospective enhancement would be to ensure locating possible transition positions by comparing segment groups and not individual chunks as the current approach prescribes. For instance, segments 1, 2, 3 could be set against segments 12, 13, 14 when the similarity score is computed for chunks 1 and 12, 2 and 13, 3 and 14 separately and then accumulated into a single metric of alikeness. This would enable the comparison algorithm to analyze audio files at a bigger scale, which might make the detection of similar complex, time-stretched sounds possible. It appears that the more segments will used to form a group, the more the algorithm will deviate from being a local-scale search tool to being a pattern-matching instrument. Of course, balance should be found between the size of comparison groups and the computational load, as this advanced technique will entail the multiplied number of required image comparison operations. Spectral similarity threshold should will have be adjusted as well to conform to new selection scheme.

The last suggestion regards the aspect of how T_s is defined. The testing has shown that for some audio files very few transitions fit under the necessary value. That circumstance prompts that a better solution should somehow normalize T_s over the set of several sonic parameters of each individual song. Those properties may as well be computed for different frequency bands separately. Alternatively, their values may be found locally for each of relatively big parts that the source file can be subdivided to, so that each segment comparison relies on the similarity threshold that corresponds to the area where compared chunks reside.

5.2 Preprocessing speed testing

Even though execution speed is not the primary criterion for evaluating the quality of the developed algorithm, this type of performance testing was still conducted to get the outline notion of how fast the implementation is able to accomplish the programming task. That aspect can not be ignored because the faster the solution is, the shorter time will be required to initialize it within into game audio engine, and, consequently, to launch the game. It is obvious that faster loading times provide higher user satisfaction.

For the testing, the execution times of main preprocessing methods inside Processor application class were measured. As variable parameters, input audio file and f_{RMS} value were picked. Five measurements were made for each testing combination. The average results obtained are shown in Table 5.2. Beatmap calculation times are not given as those appeared to be almost instant for each of the trial cases.

№	**File length, s**	**Average preprocessing step time, s**					**Average total preprocessing time, s**		
		Segment FFT analysis and creation of images	**Comparison and filtering of segments**	**Aligning similar segments**					
				High f_{RMS}	**Moderate f_{RMS}**	**Low f_{RMS}**	**High f_{RMS}**	**Moderate f_{RMS}**	**Low f_{RMS}**
1	206	6.902	1.381	0.006	0.004	0.003	8.289	8.287	8.286
2	189	8.500	2.077	0.003	0.003	0.003	10.58	10.58	10.58
3	218	6.711	1.287	0.005	0.005	0.003	8.003	8.003	8.001
4	190	7.903	2.436	0.011	0.008	0.003	10.35	10.347	10.342
5	145	3.981	0.465	0.002	0.002	0.002	4.448	4.448	4.448
6	124	5.611	0.910	0.002	0.002	0.001	6.523	6.523	6.522

Table 5.2: Application preprocessing steps' execution times and transfer calculation times measured for of six test audio files

Overall, the algorithm has demonstrated impressive performance – for current implementation, it takes no more than 10 seconds on average to create a transitioning table for a three-and-a-half-minute long audio file. A certain linear correlation between input file length and waiting time can be drawn – the execution went almost two times faster when done on shorter files 5 and 6. Also, the testing has revealed that the choice of f_{RMS} has no significant effect on computation time – the segment alignment technique handles various quantities of chunks equally good. The FFT and chunk image compilation expectedly appeared to be the most resource-demanding tasks out of all observed ones. This leads to a conclusion that the preprocessing time is mostly dependent on the total number of chunks that are formed according to beatmap, chunk size and overlap settings at the start of the algorithm's operation.

Chapter 6

Demo game and API

The implementation of the proposed transfer point detection algorithm was visualized and tested via a GUI application. This allowed to conduct perceptional testing (the smoothness of calculated transitions) and computational performance evaluation. However, in praxis, the algorithm will never be run on its own – instead, it supposed that it will be used to aid the creation of an interactive music system for games.

To showcase how the solution can be used to serve purposes of procedural audio design, a designated application user interface was created inside Unity engine. The idea of API is to provide the user with an easy way to take advantage of the algorithm within any custom project by passing it several input parameters and obtaining certain output that can be used in game. That way, all computationally and logically heavy operations take part behind the scenes of the project, on the API side.

As a foundation for this API, the core audio processing functions from the Processor class of the implementation package were taken. They were complemented by several public methods that could accept song information variables from the user and return appropriate transition point locations when the need for a transfer emerges during gameplay. After that, a very simple 2D-game was created that relies on the API to interactively change background music depending on the state of the game.

6.1 Game description

The game features a playable character, an enemy character and a target flag. All three objects are placed on a flat ground. The objective of the character has to kill the enemy and reach the target flag. The player can move the character left and right with arrow keyboard keys. The character kills an enemy by coming close to it (see figure 6.1).

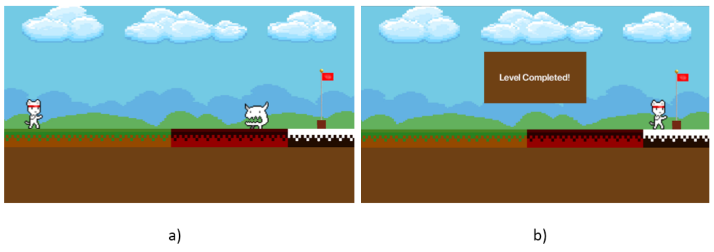


Figure 6.1: Demo game scene: a) – At the start of the game, b) – At the end of the game

6.2 Game files

Inside the Game subdirectory of Resourses folder of the Unity project are three catalogs. The first, Audio Files, contains a single file "Igor Kotov - Demo Game Music.mp3", which is a one-minute musical score arranged exclusively for the game by the author of this thesis. This song is used as a source audio clip that helps build a primitive API-based interactive audio system for the game. Catalogue Sprites has the raster assets used to visualize the objects of game environment. All gameplay-controlling scripts and the interface script reside in Scripts folder.

It should be noted that all 2D-sprites used to build the game were either taken from publicly available asset packs distributed under Creative Commons Attribution Unported 3.0 licence or made individually by the author of this thesis [22] [23].

The foundation for the demo game is formed by the following classes:

- API – the application programming interface class. Handles all the programming logic related to audio processing and the search of transition samples;
- EventHandler – monitors properties of game objects to determine if the game state has changed and communicates those changes to the MusicPlayer class;
- MovementManager – performs the movement of the game character by taking keyboard key inputs from the player;
- MusicPlayer – simulates game's audio system controller. Contains user calls of public API methods to add information about music themes, request the calculation of transitioning tables and the search for appropriate entry and exit transition points when directed to do so by the EventHandler class. Performs playback transitioning relying on data returned by the API class.

The brief description of all programming methods utilized in the demo game within Unity scripts is given in the appendix section E.2.

6.3 Game-API interaction

To use the implementation's algorithm through API, the programmer first has to pass this data to the API via calculateTransitioningTables method:

- The audio clip;
- The BPM value for the song;
- The number of beats per bar (only 3/4 and 4/4 scales are supported currently; if a value of 3 is passed, the time signature is thought to be 3/4, if a value of 4 is passed, it is assumed to be 4/4);
- The beat offset value;
- The desired threshold value T_s. It is suggested that T_s lays in between 0.01 and 0.3. The lower the threshold, the more transitioning points will be found in an audio file, but the higher false positive rate can be, and vice versa.

Once API receives all those fields, it computes the transitioning table and stores it for future use. The user is then required to add timestamp borders for music themes within the audio file on the game side (i.e. from the MusicPlayer class). Those markers are used in two ways:

- To define transition destination when asking API for a pair of preferred transfer points;
- To maintain the playback by looping over theme segments when no scene changes are taking place.

Since this setup is done, the music player is now capable of reacting to game state changes by turning to the interface class to get transfer points and non-linearly move the playhead based on the obtained positions so that current in-game audio corresponds to a new music theme. The user can claim transition samples by calling the getTransitionSamples API method, providing current playback time and desired destination time as input arguments. The game-API interaction is further explained in the next section, where interactive audio system is being set up and tested for the demo game.

6.4 Demo game run

As soon as the demo game enters the runtime mode, the music player submits the soundtrack file, its tempo-related properties and the user-set threshold value for segment comparison via calculateTransitioningTables method. The transitioning table then computed by the API. After that, three music themes are added. Those correspond to possible game states and are called "exploration", "danger" and "victory". The game states have the same names.

During gameplay, current state is set by the following rules:

- At the very start of the game, the state is set to "exploration";
- If the player's character is close to the enemy, the state is "danger";
- If there are no enemies nearby player's character and he has not yet reached the target flag, the state is "exploration";

- If the character makes his way to the flag, the mode is set to "victory" and the game ends.

For debugging purposes and to provide a better visualization of adaptive audio at work, the ground surface in game is colored in green and brown, black and red or black and white, which is done to visualize how EventHandler class is expected to switch game states according to occurring events:

- If the player is in the green and brown zone (i.e. far from the enemy), the game should always stay in exploration mode;
- If the player is touching black and red ground, danger mode should be triggered if the enemy is not yet killed;
- If the player is inside black and white region, victory mode should be activated and the game should end.

When it so happens that during gameplay the game state changes, the musicPlayer class calls getTransitionSamples method of API. As input arguments, current playback position is provided along with the timestamp that corresponds to the start time of the music scene that the transition has to be made to. That way, the API will search for the fastest way to get from the old music theme to the beginning of a different theme that agrees with the newly set game state. Once the interface class returns selected entry and exit points, music player begins to keep track of the playback and performs the transition when prescribed.

The event-driven adjustability of created game's audio system proves that the transfer point detection algorithm designed and implemented in this work can be put to use to establish simple procedural audio behaviour within custom user projects in the field of electronic entertainment.

The user manual explaining the setup needed for running the demo game is given in appendix section C.2.

Chapter 7

Conclusion

The research conducted over the course of work on this diploma thesis revolved around the concept of non-linear playback of audio samples via transitioning, a method which is currently the foundation of modern paradigm of adaptive music design for electronic entertainment.

First, an overview of relevant procedural audio techniques was made to introduce the domain under interest. The discussion touched upon the features that popular designated middleware titles such as FMOD Studio offer to aid generative sound design, and the notion of transitioning was investigated in particular. In addition to this, a brief insight into audio synthesis was provided – it was mentioned as another tool that aims to challenge the traditional start-to-end playback to create brand new audio patterns and timbres. The drawbacks of current transitioning mechanisms were listed that motivated the main goal of this thesis, which was to suggest, implement and test an algorithm that would assist automatic context-aware playhead transfers within music files by locating appropriate entry and exit positions for smooth and seamless transitions.

After that, the necessary background for follow-up work was explored by taking a look into the basics of sound sampling and processing. This helped to brainstorm the matter and come up with the approach to solving the formulated problem.

Then, the developed algorithm, which entailed determining the degrees of frequential similarities between small groups of source file samples, was proposed, and all its steps were thoroughly described. The technical choices for algorithm's implementation were justified – and Unity game engine was picked as the main execution environment.

One part of practical realization was an application with graphical user interface which illustrated the functionality of the suggested method in various modes. The GUI provided a framework for controlling the playback, scheduling and canceling playback transitions, as well as restricting transfers to certain positions within the song's rhythmic grid and affecting the number and the preciseness of viable jumps depending on the selected value of sample segment similarity threshold.

The correctness of the algorithm's operation was proven by perception testing, where participants were suggested to spot transitions while listening to on-the-fly composed compositions built by moving the playhead around test music files according to precalculated entry-exit pairs of transfer points. The implementation has demonstrated a good level of non-linear playback invisibility, organizing most transfers in an unnoticeable manner. A number of possible future improvements to preprocessing procedure, that were inspired my user testing, were put forward. Additionally, the algorithm's execution speed was measured for a number of combinations of variable performance parameters. The results turned out to be more than satisfactory, with an average of ten seconds of time needed to compute the transitioning scheme for a typical three-minute long music piece.

To showcase how realized technique could be applied in practice to serve the purpose of creating adaptive audio systems within custom projects, the core algorithm functionality was encapsulated into an API which usage was demonstrated on an example of a simple developed Unity game. During the gameplay, the changing properties of in-game objects can cause game engine to switch between different game states. In response to that, game's music system is capable of making calls to API methods to quickly adapt the sound environment to the new state by transitioning. The application programming interface performs all necessary calculations behind the scenes and returns a pair of entry and exit points that can be used to non-linearly move to a new music theme when needed.

Taking into account all aforementioned achievements, it can be claimed that the goal of this master thesis has been accomplished.

7.1 Future plans

The implementation of transfer point detection algorithm described in this work can be further improved in the following ways:

- The smoothness of calculated transitions be rectified by elaborating on the possible modifications laid down in subsection 5.1.3;
- The speed of preprocessing routine can be increased by optimizing the source code both logically and programmatically (i.e. making sure that appropriate class hierarchies are build and the right data types and structures are chosen);
- The scope of algorithm's practical usage can be expanded by using the source code of the created API to create standalone and integrated software components such as VST-plugins, graphical and command-line applications. In the long run, the foundation devised in this thesis may be involved in a new IDE for adaptive sound design.

Bibliography

[1] Procedural Audio for Video Games: Are we there yet? https://www.gdcvault.com/play/1012645/Procedural-Audio-for-Video-Games. [online 20.3.2019].

[2] FMOD Games. https://www.fmod.com/games. [online 20.3.2019].

[3] R. Bristow-Johnson. Wavetable Synthesis 101, A Fundamental Perspective. Audio Engineering Society (AES), 1996.

[4] J. Strawn. Digital Audio Signal Processing: An Anthology. A-R Editions, 1985.

[5] C. Roads. Microsound. The MIT Press, 2001.

[6] Digital Audio Basics: Sample Rate and Bit Depth. https://www.presonus.com/learn/technical-articles/sample-rate-and-bit-depth. [online 1.4.2019].

[7] Nyquist Sampling Theorem. http://musicweb.ucsd.edu/~trsmyth/digitalAudio171/Nyquist_Sampling_Theorem.html. [online 2.4.2019].

[8] Fourier Transformation and Its Mathematics. https://towardsdatascience.com/fourier-transformation-and-its-mathematics-fff54a6f6659. [online 5.4.2019].

[9] Sparse Fast Fourier Transform. https://groups.csail.mit.edu/netmit/sFFT/. [online 5.4.2019].

[10] Richard A. Brualdi. Introductory Combinatorics (5th ed.). Pearson Prentice Hall, 2010.

[11] D.L. Kreher, D.R. Stinson. Combinatorial Algorithms: Generation, Enumeration and Search. CRC press LTC, 1998.

[12] D. Mayers. An Introduction to Numerical Analysis. Cambridge University Press, 2003.

[13] Introduction to Polynomial Interpolation. https://sameradeeb-new.srv.ualberta.ca/introduction-to-numerical-analysis/polynomial-interpolation/. [online 17.4.2019].

[14] The complexity of evaluating interpolation polynomials. https://www.sciencedirect.com/science/article/pii/0304397585900787?via%3Dihub. [online 18.4.2019].

[15] I researched the market share of game engines on Steam. `https://www.reddit.com/r/gamedev/comments/8s20qp/i_researched_the_market_share_of_game_engines_on/`. [online 27.4.2019].

[16] DeepMind partners with gaming company for AI research. `https://www.dailydot.com/debug/unity-deempind-ai/`. [online 29.4.2019].

[17] The most popular IDEs? Visual Studio and Eclipse. `https://www.infoworld.com/article/3217008/the-most-popular-ides-visual-studio-and-eclipse.html`. [online 2.5.2019].

[18] Algorithmic Beat Mapping in Unity: Preprocessed Audio Analysis. `https://medium.com/giant-scam/algorithmic-beat-mapping-in-unity-preprocessed-audio-analysis-d41c339c135a`. [online 24.3.2019].

[19] DSPLib - FFT / DFT Fourier Transform Library for .NET 4. `https://www.codeproject.com/Articles/1107480/DSPLib-FFT-DFT-Fourier-Transform-Library-for-NET-6`. [online 13.4.2019].

[20] Microsoft/referencesource/System.Numerics/System/Numerics/Complex.cs. `https://github.com/Microsoft/referencesource/blob/master/System.Numerics/System/Numerics/Complex.cs`. [online 13.4.2019].

[21] P. Stepanishen. Handbook of Acoustics J. Malcolm. John Wiley & Sons, Inc., 1998.

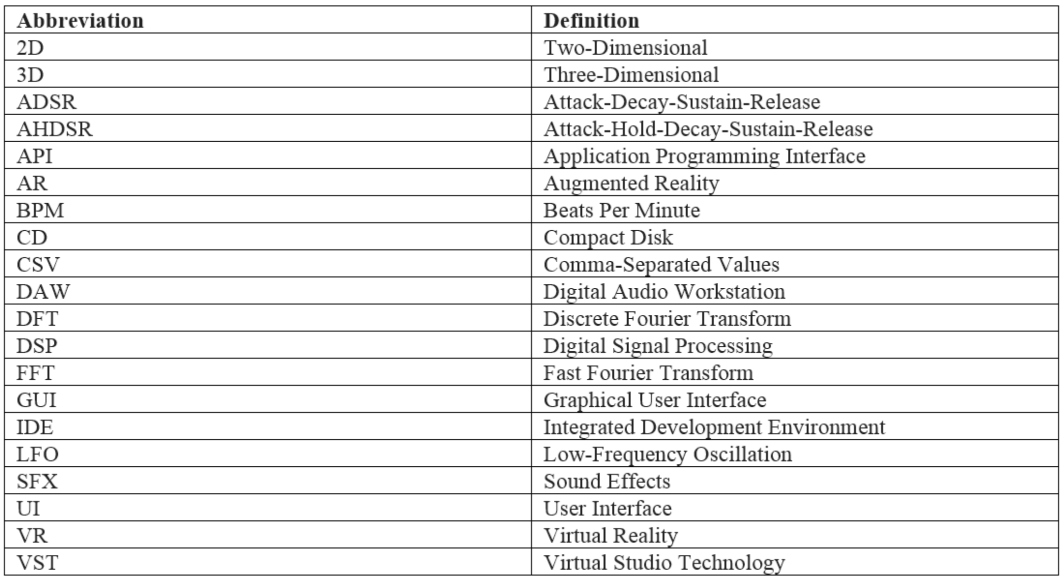
[22] Cat Fighter Sprite Sheet. `https://opengameart.org/content/cat-fighter-sprite-sheet`. [online 29.5.2019].

[23] Cute Monster Sprite Sheet. `https://opengameart.org/content/cute-monster-sprite-sheet`. [online 29.5.2019].

Appendix A

Abbreviations

The abbreviations used in this work and their definitions are listed in table A.1.



Abbreviation	Definition
2D	Two-Dimensional
3D	Three-Dimensional
ADSR	Attack-Decay-Sustain-Release
AHDSR	Attack-Hold-Decay-Sustain-Release
API	Application Programming Interface
AR	Augmented Reality
BPM	Beats Per Minute
CD	Compact Disk
CSV	Comma-Separated Values
DAW	Digital Audio Workstation
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
GUI	Graphical User Interface
IDE	Integrated Development Environment
LFO	Low-Frequency Oscillation
SFX	Sound Effects
UI	User Interface
VR	Virtual Reality
VST	Virtual Studio Technology

Table A.1: Used abbreviations and their definitions

Appendix B

CD Contents

The CD provided with the hard copy of this master thesis contains the following items:

- thesis.pdf – The thesis research paper file;
- Project – a folder of the source Unity Project. Has all code scripts, images, audio clips and other files that were used during practical part of this work. The description of the most important catalogues and files is given in subsection 4.3.2 and in section 6.2;
- demoGameRun.mp4 – a video file that demonstrates the work of the implementation algorithm for the needs of adaptive audio within a sample game setting.
- demoGameRunComments.txt – a text file that describes the demoGameRun video. All comments are complemented with the assosiated timestamps, which explain what happens in a certain moment of video playback.

Appendix C

User manual

In this appendix, the list of instructions are given that could be followed to launch and interact with the created GUI application and the demo game.

To access the project flies and scenes, Unity game engine of version 2018.2.15f1 or newer has to be installed on the host PC. Then the subfolder "Rhythm-based transitioning" of folder "Project" located on the attached CD should be opened in Unity environment. This could be done either by pressing "Open" button on Unity start screen or going to "File" → "Open Project" from the environment's top toolbar.

C.1 Interacting with the GUI application

To run the GUI application, the user has to do the following:

1. Navigate to "Assets/Scenes" folder using the Project explorer tab (located in the bottom-left corner by default);

2. Double-click on ApplicationScene file to open the application scene;

3. Click on the ApplicationObject inside the object hierarchy panel (located on the top center by default) to display its properties in the Properties tab (to the right of the hierarchy tab);

4. Go to "Assets/Resources/Application/Audio files", drag and drop one of the selected sound files into the AudioClip field of Audio Source component in the Properties Inspector. Custom files could be put into the Audioclip field as well, but they have to be to the "Audio files" directory first;

5. Press Scene Play/Stop button (centered on the very top of Unity workspace). The interactions with interface can now be made. The detailed description of GUI elements is given in table 4.1;

6. The run of the application can be stopped by pressing the Scene Play/Stop button again.

C.2 Playing the demo game

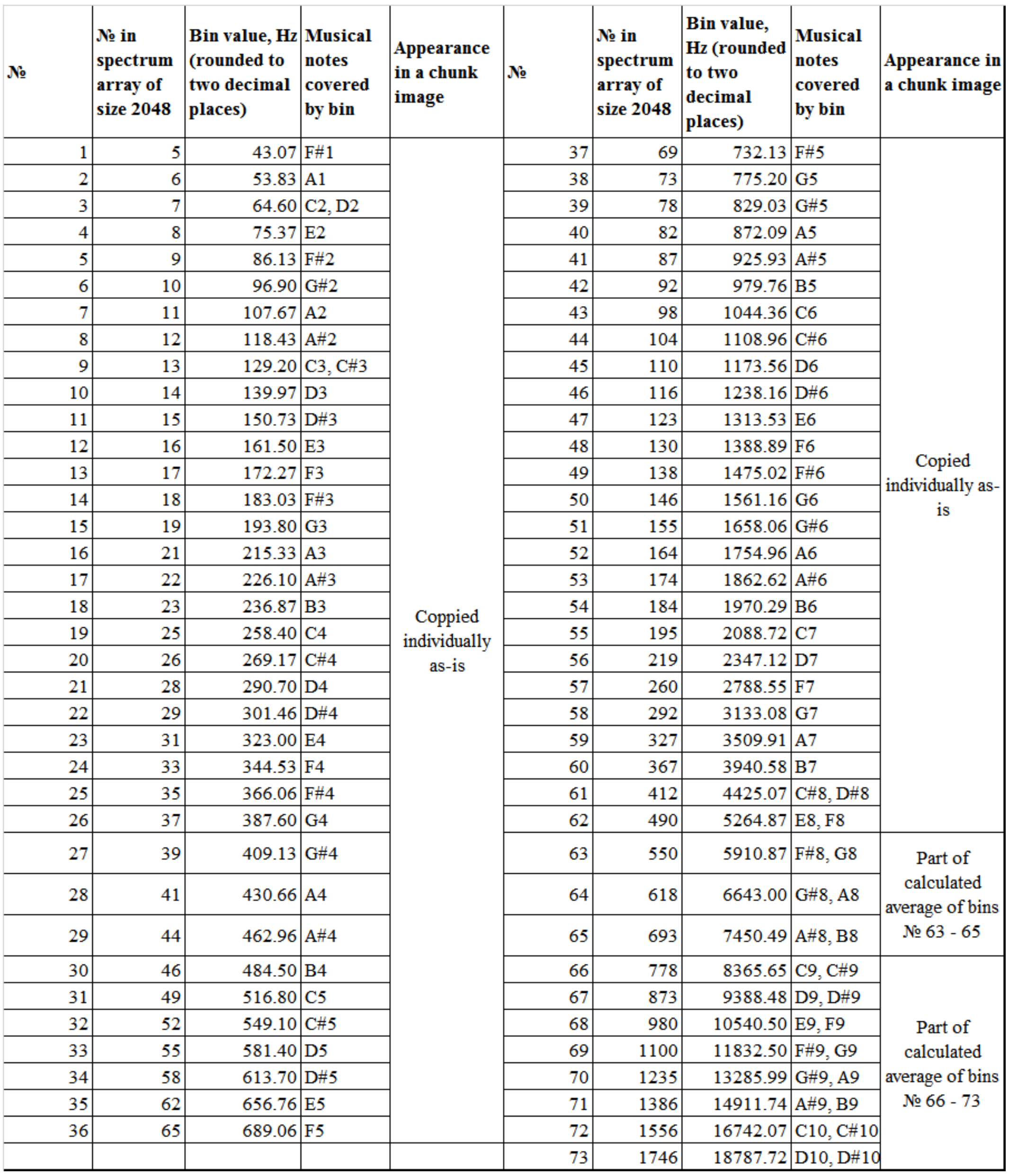
To run the demo game, the user has to go through the following steps:

1. Navigate to "Assets/Scenes" folder using the Project explorer tab;
2. Double-click on GameScene file to open the game scene;
3. Press Scene Play/Stop button. The game is now started and the music within it is played. The user can move the charachter via left and right arrow keys on the keyboard and hear how game music changes in correspondence to the game events. The way it is done is broken down in section 6.4;
4. The run of the game can be stopped by pressing the Scene Play/Stop button again. When the game is completed, it is required to restart it to get back to gameplay.

Appendix D

Segment image contents

Before the comparison of sample segments is done in the implementation, the images of chunks are formed. Table D.1 shows which samples of FFT spectrum output are taken to construct a segment image and how they are used.



№	№ in spectrum array of size 2048	Bin value, Hz (rounded to two decimal places)	Musical notes covered by bin	Appearance in a chunk image	№	№ in spectrum array of size 2048	Bin value, Hz (rounded to two decimal places)	Musical notes covered by bin	Appearance in a chunk image
1	5	43.07	F#1	Coppied individually as-is	37	69	732.13	F#5	Copied individually as-is
2	6	53.83	A1		38	73	775.20	G5	
3	7	64.60	C2, D2		39	78	829.03	G#5	
4	8	75.37	E2		40	82	872.09	A5	
5	9	86.13	F#2		41	87	925.93	A#5	
6	10	96.90	G#2		42	92	979.76	B5	
7	11	107.67	A2		43	98	1044.36	C6	
8	12	118.43	A#2		44	104	1108.96	C#6	
9	13	129.20	C3, C#3		45	110	1173.56	D6	
10	14	139.97	D3		46	116	1238.16	D#6	
11	15	150.73	D#3		47	123	1313.53	E6	
12	16	161.50	E3		48	130	1388.89	F6	
13	17	172.27	F3		49	138	1475.02	F#6	
14	18	183.03	F#3		50	146	1561.16	G6	
15	19	193.80	G3		51	155	1658.06	G#6	
16	21	215.33	A3		52	164	1754.96	A6	
17	22	226.10	A#3		53	174	1862.62	A#6	
18	23	236.87	B3		54	184	1970.29	B6	
19	25	258.40	C4		55	195	2088.72	C7	
20	26	269.17	C#4		56	219	2347.12	D7	
21	28	290.70	D4		57	260	2788.55	F7	
22	29	301.46	D#4		58	292	3133.08	G7	
23	31	323.00	E4		59	327	3509.91	A7	
24	33	344.53	F4		60	367	3940.58	B7	
25	35	366.06	F#4		61	412	4425.07	C#8, D#8	
26	37	387.60	G4		62	490	5264.87	E8, F8	
27	39	409.13	G#4		63	550	5910.87	F#8, G8	Part of calculated average of bins № 63 - 65
28	41	430.66	A4		64	618	6643.00	G#8, A8	
29	44	462.96	A#4		65	693	7450.49	A#8, B8	
30	46	484.50	B4		66	778	8365.65	C9, C#9	Part of calculated average of bins № 66 - 73
31	49	516.80	C5		67	873	9388.48	D9, D#9	
32	52	549.10	C#5		68	980	10540.50	E9, F9	
33	55	581.40	D5		69	1100	11832.50	F#9, G9	
34	58	613.70	D#5		70	1235	13285.99	G#9, A9	
35	62	656.76	E5		71	1386	14911.74	A#9, B9	
36	65	689.06	F5		72	1556	16742.07	C10, C#10	
					73	1746	18787.72	D10, D#10	

Table D.1: Some of the middleware-specific features used for transitioning in adaptive audio

Appendix E

Description of methods

In this appendix, a brief description of all created programming methods within Unity scripts is given, both for the test application and the demo game.

E.1 Methods of GUI application classes

E.1.1 Processor class methods

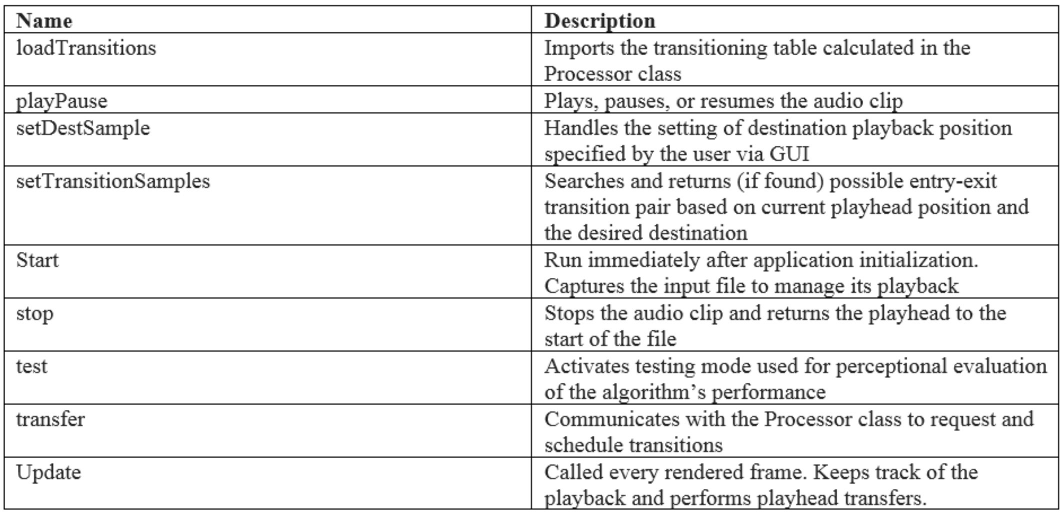
The description of methods defined for class Processor of GUI application is given in table E.1.

Name	**Description**
alignDestinationSegments	Aligns the selected transition entry-exit sample pairs to the same position within a bar
calculateBeatMap	Calculates the beatmap based on input tempo parameters
clearData	Clears the transitioning table
compareBeatSegments	Compares all discovered segments and fills up the transitioning table with the pairs that have acceptable similarity score. Returns a number of found pairs.
createSegmentImage	Compiles and returns image of a segment from its spectrum data
FFT	Performs FFT over a chunk of samples and returns the corresponding spectral array.
fillSegmentImage	Fills up data structures for different-beat chunks with images of those chunks
prepareSegmentData	Groups samples into chunks according to beatmap and passes them to fillSegmentImage function
readAudioData	Reads the input file and returns an array of all its sample values
RecaclulatePaths	Launches the whole processing cycle of the algorithm from beatmap calculation to finding acceptable transitions
segmentNumberOfSample	Returns the number of segment to which sample of input number belongs
setThresholdsForMethod	Sets the f_{RMS} value depending on the transitioning method chosen by user
Start	Run immediately after application initialization. Triggers the function readAudioData
startSampleOfSegment	Returns the number of the first sample in a given segment
timeOfSample	Returns the value in seconds which shows to which playback timestamp the given sample corresponds
updateLastCalculationMethod	Updates the last calculation method. Used to guarantee that when transitioning table is saved into file, the correct postfix is added to its name
writeTransitionsToFile	Writes the transitioning table into a CSV file and places this file into the project's folder.

Table E.1: Methods of application class Processor

E.1.2 AudioPlayer class methods

The description of methods defined for class AudioPlayer of GUI application is given in table E.2.

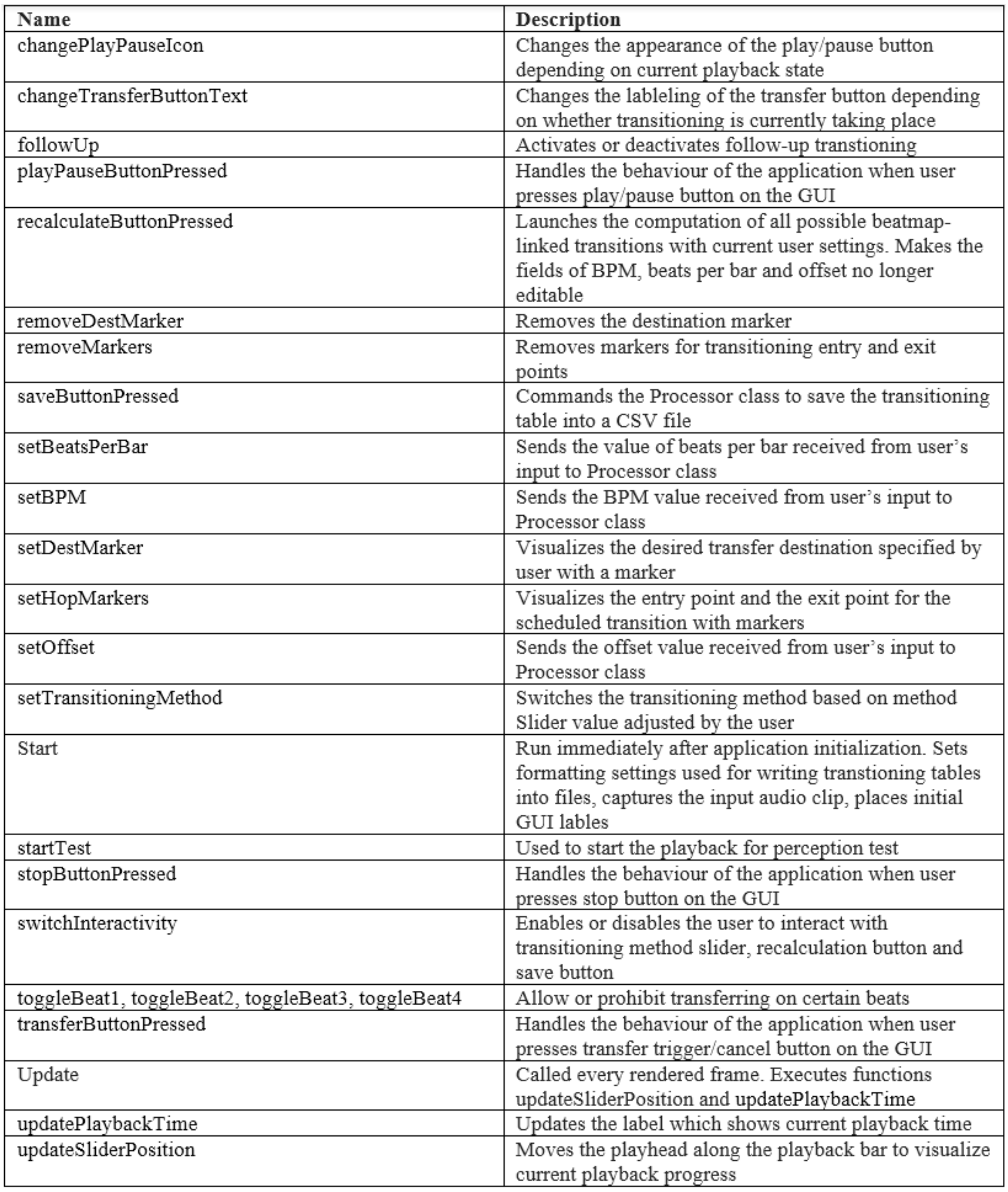


Name	Description
loadTransitions	Imports the transitioning table calculated in the Processor class
playPause	Plays, pauses, or resumes the audio clip
setDestSample	Handles the setting of destination playback position specified by the user via GUI
setTransitionSamples	Searches and returns (if found) possible entry-exit transition pair based on current playhead position and the desired destination
Start	Run immediately after application initialization. Captures the input file to manage its playback
stop	Stops the audio clip and returns the playhead to the start of the file
test	Activates testing mode used for perceptional evaluation of the algorithm's performance
transfer	Communicates with the Processor class to request and schedule transitions
Update	Called every rendered frame. Keeps track of the playback and performs playhead transfers.

Table E.2: Methods of application class AudioPlayer

E.1.3 UI class methods

The description of methods defined for the user interface class of GUI application is given in table E.3.



Name	Description
changePlayPauseIcon	Changes the appearance of the play/pause button depending on current playback state
changeTransferButtonText	Changes the lableling of the transfer button depending on whether transitioning is currently taking place
followUp	Activates or deactivates follow-up transtioning
playPauseButtonPressed	Handles the behaviour of the application when user presses play/pause button on the GUI
recalculateButtonPressed	Launches the computation of all possible beatmap-linked transitions with current user settings. Makes the fields of BPM, beats per bar and offset no longer editable
removeDestMarker	Removes the destination marker
removeMarkers	Removes markers for transitioning entry and exit points
saveButtonPressed	Commands the Processor class to save the transitioning table into a CSV file
setBeatsPerBar	Sends the value of beats per bar received from user's input to Processor class
setBPM	Sends the BPM value received from user's input to Processor class
setDestMarker	Visualizes the desired transfer destination specified by user with a marker
setHopMarkers	Visualizes the entry point and the exit point for the scheduled transition with markers
setOffset	Sends the offset value received from user's input to Processor class
setTransitioningMethod	Switches the transitioning method based on method Slider value adjusted by the user
Start	Run immediately after application initialization. Sets formatting settings used for writing transtioning tables into files, captures the input audio clip, places initial GUI lables
startTest	Used to start the playback for perception test
stopButtonPressed	Handles the behaviour of the application when user presses stop button on the GUI
switchInteractivity	Enables or disables the user to interact with transitioning method slider, recalculation button and save button
toggleBeat1, toggleBeat2, toggleBeat3, toggleBeat4	Allow or prohibit transferring on certain beats
transferButtonPressed	Handles the behaviour of the application when user presses transfer trigger/cancel button on the GUI
Update	Called every rendered frame. Executes functions updateSliderPosition and updatePlaybackTime
updatePlaybackTime	Updates the label which shows current playback time
updateSliderPosition	Moves the playhead along the playback bar to visualize current playback progress

Table E.3: Methods of application UI class

E.2 Methods of demo game classes

E.2.1 API class methods

The description of methods defined for API class used in the demo game is given in table E.4.

Name	**Description**
alignDestinationSegments	Same functionality as in Processor class of the GUI application
calculateBeatMap	
calculateTransitioningTables	Takes the audio clip, BPM, beats per bar and offset values and the f_{RMS} value as the function is called from the MusicPlayer class. Processes this data to calculate and store the transitioning table
compareBeatSegments	Same functionality as in Processor class of the GUI application
createSegmentImage	
FFT	
fillSegmentImage	
getTransitionSamples	Same functionality as function setTransitionSamples in Processor class of the GUI application. Called from the MusicPlayer class with user-provided values of current playback position and the desired destination
prepareSegmentData	Same functionality as in Processor class of the GUI application
readAudioData	
sampleAtTime	Returns a number of audio sample that corresponds to the input playback time
segmentNumberOfSample	Same functionality as in Processor class of the GUI application
startSampleOfSegment	
timeOfSample	
writeTransitionsToFile	

Table E.4: API class methods

E.2.2 MusicPlayer class methods

The description of methods defined for MusicPlayer class of the demo game is given in table E.5.

Name	**Description**
addMusicScene	Stores border playback positions for a specified music theme in an inner data structure. These values are then used to control looping and transitioning
Start	Run immediately after application initialization. Adds border timestamps for game music scenes. Calls the API function calculateTransitioningTables with necessary parameters
Update	Called every rendered frame. Requests transition points from the API class upon game state change and moves the playhead accordingly

Table E.5: Methods of MusicPlayer class

E.2.3 EventListener class methods

The description of methods defined for EventListener class of the demo game is given in table E.6.

Name	**Description**
Start	Run immediately after application initialization. Sets initial parameters of variables that keep track of game music themes
Update	Called every rendered frame. Monitors mutual positions and properties of game objects and changes game state variable accordingly.

Table E.6: Methods of EventListener class

E.2.4 MovementManager class methods

The description of methods defined for MovementManager class of the demo game is given in table E.7.

Name	**Description**
Start	Run immediately after application initialization. Sets the initial position of game character.
Update	Called every rendered frame. Converts input from user pressing keyboard keys to the movement of the game character

Table E.7: Methods of MovementManager class