barelyMusician:
An Adaptive Music Engine For Interactive Systems

by

Alper Gungormusler, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

September 2014

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Alper Gungormusler

September 1, 2014

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Alper Gungormusler

September 1, 2014

Acknowledgments

First and foremost, I would like to thank my supervisors Dr. Mads Haahr and Natasa Paterson-Paulberg for their guidance and support in the completion of this project.

I would like to specially thank Altug Guner, Talha & Tarik Kaya (Kayabros) for the ridiculous amount of playtesting they had to suffer throughout the process.

Finally, I would like to thank my family and friends for all their encouragement and support.

Alper Gungormusler

University of Dublin, Trinity College
September 2014

barelyMusician: An Adaptive Music Engine For Interactive Systems

Alper Gungormusler
University of Dublin, Trinity College, 2014

Supervisors: Mads Haahr, Natasa Paterson-Paulberg

Aural feedback plays a crucial part in the field of interactive entertainment when delivering the desired experience to the audience particularly in video games. It is, however, not yet fully explored in the industry, specifically in terms of interactivity of musical elements. Therefore, an adaptive music engine, *barelyMusician*, is proposed in this dissertation in order to address this potential need. *barelyMusician* is a comprehensive music composition tool which is capable of real-time musical piece generation and transformation in an interactive manner, providing a bridge between the low-level properties of a musical sound and the high-level abstractions of a musical composition which are significant to the user. The engine features a fully-functional software framework alongside a graphical user interface to enable an intuitive interaction for the end-user.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Aural feedback plays a crucial part in the field of interactive entertainment when delivering the desired experience to the audience particularly in video games. It is, however, not yet fully explored in the industry, specifically in terms of interactivity of musical elements. Therefore, an extensive approach for development of an adaptive music composition engine is proposed in this dissertation research project in order to address this potential need.

The motivation behind the choice of this topic is presented in greater detail in Section 1.1. An overview of the main objectives of the proposed approach is provided in Section 1.2. The organization of the rest of this report is presented in Section 1.3.

1.1 Motivation

There has been a tremendous developmental curve with regards to the technological aspects in the interactive entertainment field in the last two decades – specifically in the video game industry. While the main focus was mostly towards the visual aspects of the experience, with the maturation of graphics technologies in the field, developers recently tend to focus more on the other areas—such as game AI—in order to take one step further in the field.

That being said, one of the most important aspects which dramatically affects the immersion—hence, the quality of the experience—in the field is the use of audio elements [1]. Until recently, the term *interactive*, was often discarded in such applications for game audio, especially in terms of musical elements, even in high-budget mainstream products [2]. The main reason why is arguably due to the misinterpretation of the problem which led the industry to treat the musical pieces as film scores. Consequently, even though many rich and sophisticated scores have been composed, they still more or less lack the main feature required; interactivity. Having said that, unlike movies, video games as interactive applications rarely follow a linear path, and more importantly they offer a great range of repeatability which makes dynamism—hence, adaptability—a must to have in terms of the audio content, as it is in other parts of such applications. While a few examples could be found in the industry which try to achieve such adaptability to a certain extent (see Section 2.2), the issue has not been solved yet to an acceptable level particularly considering the practical aspects.

Fortunately, there is recently an increasing interest and demand in the field in order to achieve adaptive and responsive aural feedback [3]. Nevertheless, while significant progress has been made for interactive sound synthesis in academic research as well as in practical applications in the industry [4, 5], achieving adaptive music compositions in real-time still remains an unsolved problem. More specifically, existing technologies in the field, labeled as dynamic, generally rely on *the vertical approach* [6] – layering and branching the composition into segments. Thus, the resulting output can be varied in an automated manner by certain event triggering approaches. The idea is, however, still based on pre-recorded audio loops which do not allow sufficient dynamic adaptability and flexibility after the offline creation process [7]. Moreover, such approaches require considerable amount of resources and dependencies mainly due to the manual authoring of the outcome.

As a matter of fact, there have been several attempts in the industry to take this approach one step further by introducing certain generative algorithms in order to produce the musical elements procedurally. However, those remained as specific solutions exclusive only for those specific projects. Having considered this, there is no generic approach, i.e. all purpose solution with such capabilities in the field.

1.2 Objectives

To address the issues and the potential need stated in the previous section, this research project proposes a practical approach for development of a middleware tool, *an adaptive music engine*, which is capable of autonomously generating and manipulating musical structures in real-time to be used by not only developers but also designers. The proposed approach treats the problem in an interactive manner, providing a bridge between the low-level properties of a musical sound and the high-level abstractions of a musical composition which are meant to be significant to the end-user. Moreover, it features a fully-functional software framework alongside a graphical user interface to enable an intuitive interaction.

Even though the main target is chosen to be the video games industry, the approach is also applicable to any other areas in the field of interactive digital entertainment if desired.

1.3 Dissertation Roadmap

State-of-the-art in the field is presented in Chapter 2 introducing some essential background research and relevant work in academia as well as the industry in order to develop an understanding and an analysis of the stated problem. It is followed by the methodology of the proposed approach and the implementation phase in full detail in Chapter 3 and 4 respectively. Two demonstrative applications are presented and described in Chapter 5 as proof-of-concept experiments. Evaluation and further discussion on the obtained outcome are made in Chapter 6 in order to quantify the main contribution of the proposed work to the field of research. Finally, the report is concluded by Chapter 7 alongside with a number of possible improvements to be made in the future.

Chapter 2

State of the art

Prior to the explanation of the proposed approach, it is necessary to provide some essential knowledge and terminology which are required to assess the musical background related to the field of research. Such information is presented in Section 2.1. Furthermore, a number of relevant works selected from the previous research articles and industrial applications are examined in Section 2.2 to obtain a better understanding of the state-of-the-art in the field.

2.1 Background

The project is built on a multidisciplinary research field which involves theoretical aspects of musical sound and composition combined with practical aspects of algorithmic techniques and technologies in the field of computer science. Therefore, both parts should be examined delicately in order to achieve a sufficient level of understanding in the topic.

2.1.1 Tonal Music Theory

Music as an art form has limitless possibilities of combination and permutation of different soundscapes together to create a compositional piece. Considering the common music theory, however, creating musical compositions is a highly structured and a rather rule-based process particularly when observing the Western music in the 18th

and 19th centuries, or even more recently in such genres as pop or jazz music [8]. Those type of compositions are generally defined as a a part of tonal music. That being said, tonality in music could be described as a system of specifically arranged musical patterns—sequence of melodies and chords—in a hierarchical way to attract the listener's perception with a stable and a somewhat pleasing manner [9].

While it overlooks the artistic means of creation, achieving tonality in a musical piece is arguably straightforward, if one could treat the musical properties in a proper way. Thus, it is fairly possible to produce a musically recognizable sequence of soundscapes just by following certain rules in the right order. It is expected that this approach does not give the most fascinating and/or inspiring outcome for a composer, nonetheless, the result will be an adequate musical piece that sounds *good* to a typical Western listener [10].

As a structural approach, such properties could be made by certain algorithmic techniques used in tonal Western music to automate the entire behaviour providing an artificially intelligent system. More detail in this topic is presented in Section 2.1.3.

Five Components of Tonality

Proceeding further, Tymozcko [11] argues that there are five fundamental features for a musical piece, regardless of its genre or time, which addresses tonality. These five components could be briefly described as below.

Conjunct melodic motion refers to the notes in the musical patterns to have vertically short distances to their neighbours respectively. In other words, it is unlikely to encounter rather big steps in terms of the pitch[1] of two consequent notes—such as more than an octave—in a tonal musical piece. These melodic rules have their origins in species counterpoint of Renaissance music.

Acoustic consonance is preferable to create the harmonic patterns in a tonal musical piece. Consonant harmonies, by definition, sound more stable—hence, pleasant—to the listener than dissonant harmonies.

[1]The term *pitch* is commonly used to refer the audio frequency of a certain note in musical notation.

Harmonic consistency should be established for stability. More specifically, harmonic patterns in a tonal musical piece tend to be structurally similar regardless of what type they are formed of.

Limited macroharmony refers to a tendency of relatively reducing the total collection of musical notes in a certain time period—such as in a single section—in the composition. In that sense, a typical tonal musical piece does generally not contain more than eight notes[2] in a musical phrase.

Centricity is another important component in tonal music. It refers to a selection of a certain musical note, named as *tonic*, for the piece to be centered around. Tonic tends to be heard more frequently than the other notes in a tonal musical piece, which could also be seen as *home* of the musical motion that the music is likely to return and rest.

Musical Form

While tonality could be achieved by using above components at a micro level, the overall structure of the composition should also be considered in macro level in order to provide an interesting and a meaningful outcome as a complete compositional piece. Having considered this, a musical composition could be thought of an organization in different levels, similar to a language form in linguistics. Sectional form is commonly used in music theory to arrange and describe a music composition in that manner. In sectional form, the musical piece is made of a sequence of *sections*, which are analogous to *paragraphs* in linguistics. Sections are often notated by single letters such as A and B to make them easy to read. Unlike paragraphs, sections might occur more than once in a musical piece. In fact, it generally is a not only preferred but also essential way to stress the message of the composition, i.e. the goal of the song. In popular music, sections are traditionally referred under specific names (rather than single letters) such as *verse*, *chorus* and *bridge*.

[2]The number eight does not appear of a coincidence, in fact that is the exact number of notes in one octave in a diatonic scale such as major and harmonic minor scales which are arguably the most common scale choices in tonal music.

Each section in a piece consists of a certain number of *bars*—also known as *measures*—to build up the song structure. Likewise, each bar is made of a certain number of *beats* as punctual points, which are traditionally notated with time signatures on the musical staff. For instance, the most common form, 4/4, claims that each bar in the piece has four beats (numerator) which are all quarter notes (denominator).

To sum up, the stated features and components could be used together to produce musically interesting compositional pieces either manually or an automated manner. On the other hand, the question remains whether the outcome is aesthetically acceptable or not as a final product.

2.1.2 Adaptiveness and Virtual Environments

Adaptive music in interactive systems can be briefly defined as the dynamic changes in the background music that are triggered by the pre-specified events in the environment. Adaptiveness, in broad sense, could be achieved by two main approaches. The traditional approach which most applications currently use involves layering and re-ordering pre-recorded musical pieces with respect to the relevant inputs and events in the application. While each compositional piece itself remains intact, different combinations (both horizontally and vertically) between those pieces illustrate the effect of adaptability to a certain extent. One major problem of this approach is repetition. In other words, even though the musical pieces are perfectly arranged according to every single event in the environment, the outcome will always be the same for the observer when the same events are triggered. Thus, the scene will become conspicuous to the observer after a certain time period. Moreover, it is fairly easy to exploit the approach when no inputs are received and/or events are triggered. In that case, a certain musical piece will loop forever waiting for a change in the environment, which makes the observers of the system become somewhat foreign to the environment resulting in a subtle break in immersion.

The second approach involves a more sophisticated way of dealing with the problem by introducing the concept of *generative music* – also known as *algorithmic music*

in some contexts [12]. While this approach represents some similarities to the former, it provides a highly grained and a fully dynamic structure. Instead of composing and recording the musical pieces (or snippets) beforehand, the whole music generation phase is done in an automated fashion (either offline or real-time). This can be achieved via different methodologies such as rule-based, stochastic and artificial intelligence techniques. While rule-based and artificial intelligence techniques (particularly such involving machine learning) rely on existent theoretical and empirical knowledge in music theory, e.g. scientific findings in previous works or well-known compositional pieces in the field, stochastic techniques provide a more radical way of generating the relevant features by taking advantage of randomness – more precisely pseudo-random number generators.

Having said that, they more or less are all capable of creating a unique output in a desired way to various degrees. As a matter of fact, it is not only feasible but also a popular trend to combine those techniques in order to generate an even more sophisticated outcome, since such techniques often are mutually independent. For instance, it is fairly common to combine stochastic methods with other generative algorithms to *humanize* the musical patterns. On the other hand, there is a major drawback in this approach that, as a creative process and a subjective context, it is difficult and a tedious work to keep the automated (generated) music pleasing and interesting, since the process involves both the technical and the theoretical aspects of music. In light of this, it still remains doubtful if it is even possible to achieve such convincing results without the aid of *human touch* considering the artistic means of musical composition as an art form.

2.1.3 A Brief History of Algorithmic Composition

Despite the controversial circumstances mentioned in the previous section, generative music and algorithmic composition have always been a strong candidate as an alternative artistic approach for several centuries, for not only researchers but also composers themselves – even before the arrival of personal computers [13].

One of the earliest examples which would be considered as algorithmic was proposed by famous composer Johann Sebastian Bach by the Baroque period during the final years of his life in his works *Musical Offering* and notably *The Art of Fugue* [14]. Bach studied and documented a procedural way of producing fugal and canonic musical composition in the latter work introducing the concept of counterpoint. The counterpoint is made of a leading melody, called *the cantus firmus*, which is followed by another voice that is delayed by a certain time period which creates the canonic form. The idea is to compose the generic fugue at the start and iterate the piece by using musical transformation techniques such as transposition, inversion and augmentation on this initial fugue to create variations afterwards to build the musical piece.

Another notable approach was proposed in the Classical period by Wolfgang Amadeus Mozart, in his work *Musikalisches Würfelspiel*—also known as the dice music—in which Mozart composed several musical phrases that would fit together when connected, and used a six-sided dice to determine the ordering of those pieces to be played, i.e. using different permutations of the phrases [15].

During the Romantic period, *serialism* was introduced as the main focus of the algorithmic composition. The trend in music was shifted to chromaticism, in which the composers made use of all the twelve notes in the equally-tampered scale rather than limiting themselves to traditional diatonic scales as in earlier works. Iannis Xenakis played a crucial role in that era, particularly by his interest of building a bridge between mathematics and music in his works. Furthermore, he introduced his music as *stochastic* to describe his use of probabilistic methods on certain musical parameters when composing the musical pieces.

Karlheinz Stockhausen took the approach one step further by applying serialism not only to note pitches but also the musical rhythm, timbre and so on. In both works, the music tends to be perceived as more atonal by the strong influence of chromaticism, hence the serial composition approach [16].

Computers finally started to take part in algorithmic composition when Leonard Isaacson and Lejaren Hiller introduced their brand new software *ISAAC Suite* in 1957.

It was officially the first program which algorithmically generates a musical piece [15]. Xenakis and another controversial composer John Cage also began to make use of the aid of computer systems to assist their works. For instance, Xenakis developed a system for creating integer-sequence generator—called *sieves*—in which the calculations were mainly done by computers due to the complexity of the process, in order to determine certain musical parameters such as pitch scales and time intervals in his later compositions [17].

Recently, there are several techniques and technologies available to generate music compositions algorihtmically. Markov models, generative grammars and genetic algorithms are only a few of examples for such procedural approaches. There also exist a number of software solutions, such as *PureData* [18] and *SuperCollider* [19], which ease the way out of development by providing certain low-level architectures to be used in the sound and music generation process.

2.2 Related Work

Adaptive music has been introduced in the field a long ago including some of the successful examples in major video games such as *Monkey Island 2: LeChuck's Revenge* by *LucasArts* [20]. However, it somewhat failed to evolve to a certain extent due to—arguably—two major reasons [21]. Firstly, as mentioned in Chapter 1, there was a low interest by both developers and consumers in the area of research until recently, due to the great focus on the technology behind the visual aspects such as graphics and physics systems. Secondly, despite their significant power, improvements and enhancements done for the quality of digitized audio data in computers actually resulted in a negative impact on the development, particularly when the industry has shifted from basic MIDI representations to streamed—lossless—audio data. While the quality of the output has dramatically increased, they made it somewhat infeasible to modify and manipulate the huge amount of raw data in real time with existing computational technologies. Hence, it was not a desirable choice (nor a priority), in general, to spend the limited resources to sophisticate audio elements in an application early on.

On the other hand, there has been notable research done in the recent past which is worth mentioning [22]. These works can be coarsely divided into three groups according to their area of focus, namely, offline music generation, real-time music generation and affective (smooth) music transformation. Some of the relevant seminal works are briefly reviewed in the following sections respectively.

2.2.1 Offline Music Generation

Even though offline music generation is out of our main focus, there are several remarkable works done in the field, which could serve as a guidance to the current research project.

WolframTones [23] is an excellent example, which not only contributed to the field in terms of its powerful infrastructure, but also presents an oversight of what the practical usage of such an application might evolve to. The application offers an online (web) interface that is implemented in Mathematica, which lets you generate multi-instrument musical pieces from scratch using MIDI notes. It mainly uses cellular automata for the generation of the musical snippet. The application is arguably interactive, as it lets you tweak certain parameters—such as tempo and style of the piece—before the generation phase. The outcome is solely score based, thus, it does not take into account the performance aspects of the musical piece. In other words, the generated score is played as it is during the playback. Likewise, the playback is done by generic MIDI instruments without affecting the low level parameters of the generated sound. Moreover, it features a limited flexibility of musical form, in which the compositions are approximately 30 seconds long. Therefore, higher level abstractions of a full musical piece such as verse, chorus and bridge structures are somewhat absent in the resulting composition. Nonetheless, WolframTones project can be fairly acclaimed as a benchmark in the field, considering the fact that the application has now successfully generated almost as much pieces as the entire iTunes database in less than a decade according to the statistical logs gathered [24].

2.2.2 Real-Time Music Generation

There is a strong tendency of referencing the relationship between musical features and perceived emotions of the listener in recent research studies for real-time generative music systems.

AMEE™ [7, 25] is a real-time music generation system which features an adaptive methodology in accordance with the desired emotional characteristics of a musical piece. The system is able to expressively generate musical compositions in real-time with respect to the selected properties—i.e. the perceived moods—in an interactive manner. It furthermore features an application programming interface written in Java for external usage. The music generation algorithm takes into account both score composition and performance. In fact, the authors try to establish an analogy to the real-world scenarios while structuring the system. For instance, the implementation of the framework includes actual performer structures to represent the artificial musicians in the system. However, one major drawback of the approach is the lack of smooth transitions between the different selections of musical pieces, which is said to be left as future work. While the proposed version offers an interactive way to modify the musical parameters in real-time, the change in the output only occurs after a time period, more specifically after the currently generated musical block is complete during the playback. Additionally, its quality of audio is currently limited to MIDI files, which makes the system somewhat inadequate for the usage in practical applications. Besides, even if the MIDI information is somehow converted to rich audio samples in real-time, it lacks low-level soundscapes manipulation due to the absence of essential information mentioned earlier in the chapter.

MUSIDO [26] is another example of automated music generation system which is based on a musical database that is constructed by preceding well-known musical pieces such as sample composition excerpts from Western classical music. It uses pre-determined source data to generate new composition in real-time using certain combination approaches. It extracts score snippets from the database and tries to process them in a meaningful way to produce new musical scores. The middleware implementation uses J# and Java together to achieve greater portability as a framework.

Similar to AMEE^{TM}, it manipulates the scores using MIDI representations. However, the main focus of the project is to be able to represent and extract the valid (critically acclaimed) musical pieces properly from its database, rather than focusing on the music generation phase.

MAgentA [27] treats the problem with an artificial intelligence approach, relying on an agent-like architecture for the whole music generation process. The music generator agent attempts to sense the current emotional state of the environment in real-time, and acts in accordance to its reasoning module. The music generation is done using multiple algorithmic composition methodologies with respect to the current state of the system. The prototype is said to work as a complimentary module of a virtual environment application called *FantasyA*. Unfortunately, there is no further/recent information available about the project to be discussed more.

The game Spore^{TM}(by *Maxis*) is an interesting example, in which most of the audio content is generated in real-time using mainly two dimensional cellular automata [28]. The concept is inspired from the unique yet sophisticatedly interesting patterns which could be found in J. H. Conway's *Game of Life* experiments [29]. While significant amount of the musical parameters are pre-defined, hence static, in their procedural music generation system, the resulting compositions could arguably stated as much more compelling than the other works reviewed in this section. The main reason is because of their choice to use real audio samples in order to offer a compatible audio output that a typical video game player in the industry is used to. On the other hand, the system is specifically designed for this particular game only. Therefore, generalization of the architecture does not seem to be possible for later applications. As a mater of fact, the system has officially never been used since then – apart from the sequels of the game.

2.2.3 Affective Music Transformation

To take the level of interactivity one step further, musical parameters could be examined and treated in a specific way to transform the musical pieces dramatically

in real-time. That being said, in contrast to the previous examples, Livingstone et al. [30, 31] and Friberg et al. [32] rather focus on the affective transformation of music in real-time in order to manage smooth transitions between the different pieces in an effective way. They successfully transform the musical piece in real-time affectively by using certain mapping methodologies between the perceived moods and musical parameters. Particularly, the former applies a rule-based fitting algorithm to the existent piece in order to *fix* the musical properties (mainly the pitches of notes, hence the harmony), to a desired pre-determined format. The format is determined according to the values of two high-level parameters, namely *valence* and *arousal* to be adjusted by the user. In return, they both rely on either pre-existing audio tracks or pre-generated musical patterns to produce the output. Thus, they lack the ability to provide unique music generation in real-time.

Eladhari et al. [33] proposes a similar approach using *Microsoft*'s *DirectSound* to achieve higher quality audio for the playback. The project in fact offers a limited music generation process in which the system selects a certain variation of musical patterns on the fly to adapt the current mood of the environment. However, it does not provide a generative approach to produce new patterns either offline or online. Having considered this, the idea is more or less similar to the traditional layering approach for adaptive music, except the final output is produced in real-time.

The most recent work could be found in the area of research is *AUD.js* by Adam et al. [34]. The project is developed in Javascript and focuses on smooth adaptation of the musical pieces in real-time. Based on Livingstone's approach, the system features two high-level parameters, namely *energy* and *stress*, to be defined by the user and is capable of making use of those values immediately in the generation process. It features a small-scale library of musical patterns to be selected on the fly. After the pattern selection, the output is generated by relatively primitive audio synthesis methods such as creating sine waveforms with the specified pitch and volume. Hence, it somewhat fails to produce sufficiently complex soundscapes as an output. Moreover, as in the previous examples, the system is not capable of generating the composition in real-time, which restricts its potential for wider usage.

In conclusion, the recent works tend to have a trade-off between the music generation and affective transformation capabilities. Moreover, the final output is often not as compelling as the other examples using the traditional approaches in the industry. In fact, they rarely focus on providing more sophisticated methodologies in terms of audio production, or even to support multiple instruments in their systems. In fairness, as academic works, those issues are usually claimed as a future work to be augmented later on. Nonetheless, there exist no example in the field that seems to proceed further in terms of such limitations.

Chapter 3

Design

A hybrid approach was proposed for the design of this research project which combines real-time music generation and affective music transformation capabilities into a single yet powerful framework in order to take the state-of-the-art in the field one step further. The proposed approach, the engine, focuses on the practical aspects of the composition and transformation techniques and technologies. Hence, the design goal was to manage a generic way to develop a comprehensive framework in that sense, which abstracts the low level musical properties and structures from the user in order to enable an intuitive solution for the dynamic usage of generative music in external systems without the need of any advanced knowledge in the theory of musical sound. In addition, the system was designed to provide an interactive way to modify the high-level properties of the music composition process in run time simply by triggering the corresponding functions respectively through an application programming interface and/or a graphical user interface.

The main architecture and the system workflow are presented in Section 3.1. Music generation process of the engine is described in Section 3.2. It is followed by an overview of the transformation methodology for the generated musical pieces in Section 3.3. The actual audio output generation phase is presented in Section 3.4 to provide a better understanding of the underlying architecture of the engine. Finally, this chapter is concluded by Section 3.5, provided the limitations and the restrictions to be considered for the implementation.

3.1 Main Architecture

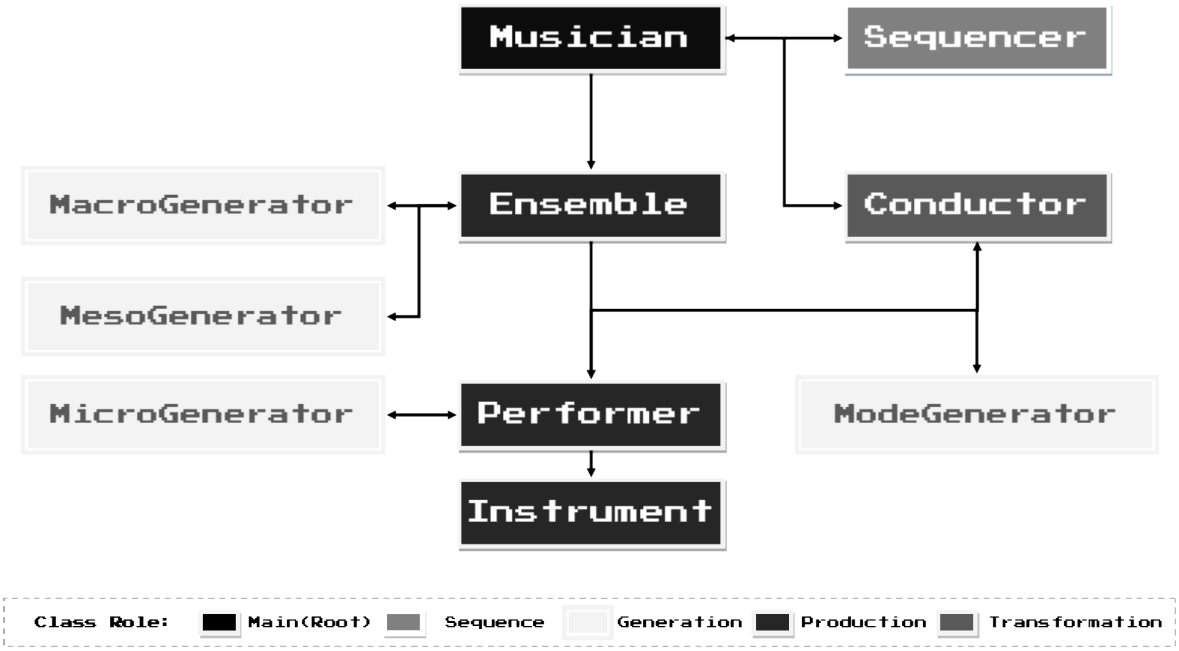

Figure 3.1: Main architecture dependencies diagram.

The main architecture of the system is vaguely based on Hoeberechts et al.'s pipeline architecture for real-time music production [35]. The components of the architecture is designed in a way somewhat analogous to a typical real-life musical performance in order to divide the workload in a meaningful manner with respect to not only the end-user but also the development phase itself. As seen in Figure 3.1, *Musician* is the main component of the architecture which is responsible for the management and the communication between the other components in the engine. *Sequencer* serves as the main clock of the engine, i.e. it sends relevant signals to *Musician* whenever an audio event occurs.

Audio events are designed hierarchically in terms of common musical form as described in Section 2.1.1. That being said, *Sequencer* follows a quantized approach in

which it keeps track of highly grained *audio pulses*[1] to occur in audio sampling rate. Counting the pulses by a phasor [36], it sequence the beats, bars and sections respectively. More detail about this process is given in the implementation stage in Chapter 4.

In each audio event, *Musician* passes that information to the *Ensemble* component – i.e. to the orchestra. Ensemble generates the song structure using its generators with respect to the current state of the sequencer. After the macro-level generation, it passes the relevant information to all its *Performer*s for them to generate the actual sequences of musical notes, such as melodic patterns and chords. Each performer produces the next sequence for the relevant bar using their generators respectively to be played by *Instrument*s. However, they initially use abstract structures of notes—only meta information—rather than producing the concrete notes, so that, the notes could be modified accordingly when necessary in order to achieve such adaptability before they get played.

That is where *Conductor* component comes into place. Conductor takes the generated note sequence in each beat and transforms the notes according to the musical parameters adjusted by the user interactively. After the transformation, the meta information gets converted to actual notes and are written into the musical score by the performer. Finally, each instrument plays all the notes in its performer's score, i.e. generates the final output which is audible to the user.

3.2 Hierarchical Music Composition

Generators in the engine jointly compose the musical pieces in three levels of abstraction, as seen in Figure 3.2. Firstly, *MacroGenerator* generates the musical form in macro level, i.e. it creates the main song structure which is made of a sequence of sections. Whenever the sequencer arrives in a new section, *MesoGenerator* generates the harmonic progression, i.e. a sequence of bars for that section. Unlike in the pipeline architecture, those two levels are managed solely by the ensemble, hence, the generations are unified for all the performers. Thus, every performer in the ensemble

[1] An audio pulse refers to the lowest level of quantization for sequencing the elements in the audio thread. It could be seen as the smallest significant interval for an audio element to be placed.

obeys the same song structure when composing their individual scores. That being said, when a new bar arrives, each *MicroGenerator* of each performer generates a meta note sequence for that bar. The meta information of a note consists of relative pitch index, relative offset (from the beginning of the bar), duration and volume of that note.

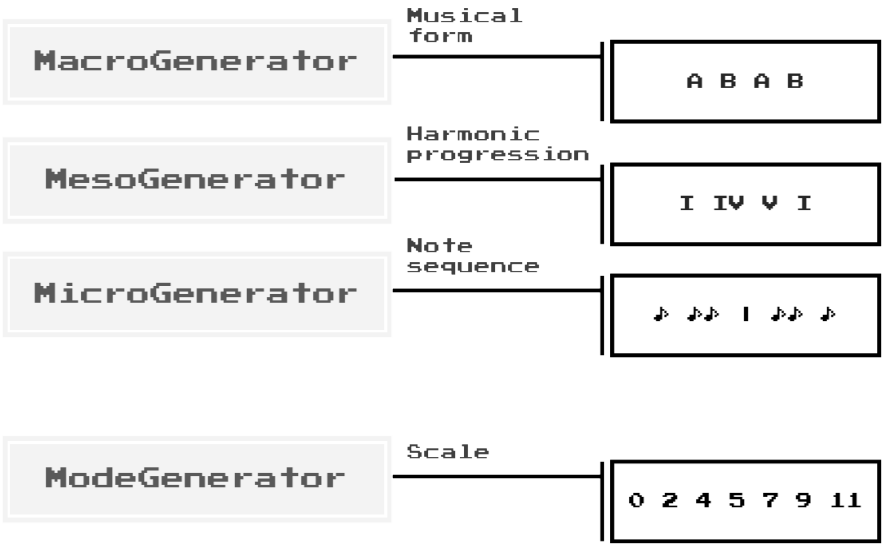

Figure 3.2: Generators structural diagram (example iterations are given on the right).

ModeGenerator generates the musical scale for the current state of the engine to be used when filling the musical score by the actual notes. The scale could be thought as the pitch quantization of the notes to be played. It is an array of ordered indices that represents the distance from the key note of the piece. In that sense, relative pitch index in a meta note refers to the note in that specific index in the array of scale.

For the generation phase in all levels, there are number of approaches and algorithms [37] which could be used in the field such as stochastic systems, machine learning, cellular automata and state-based architectures – as discussed in Section 2.1. The idea, is that this particular architecture lets the user decide what to use to accomplish the desired task. Therefore, all the generators are designed as abstracted base structures to be derived from whichever the necessity is. This approach also promotes flexibility and encourages the user to combine different techniques together, or use them interchangeably in the system even in run time.

3.3 Rule-based Note Transformation

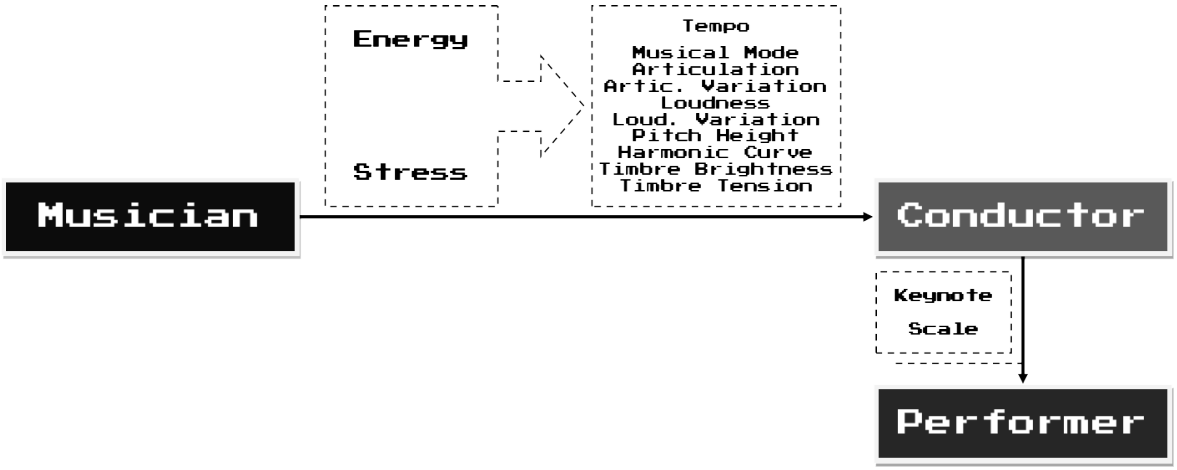

Figure 3.3: Note transformation diagram.

In terms of adaptability, the *Conductor* component is used in order to achieve affective smooth transformations when required. The interactivity between the user and the interface are done by *Musician* using two high-level parameters, namely *energy* and *stress*[2], which are meant to state *the current mood* of the system. The user is able to modify those parameters anytime to change the mood of the musical piece accordingly. The parameters are directly mapped to selected musical properties which are managed in conductor, as seen in Figure 3.3. So that, change of values in the high-level parameters by the user interaction results in an immediate response in low-level musical parameters in the engine. This mechanism is achieved by using those parameters effectively during the note transformation phase. Since the transformation process is done in each sequenced beat, the outcome is capable of reacting even to a minor change immediately in an adaptive manner.

The musical properties stated above have been chosen with respect to Livingstone et al.'s extensive research and analysis on emotion mapping [30]. While the selected

[2]The terminology was adopted from the AUD.js [34] project.

parameters strictly follows their results, not all the findings have been used in this project in order to keep the design feasible enough for the implementation. Below are brief descriptions of the selected parameters.

Tempo multiplier basically determines how fast the song will be played – in terms of beats per minute (BPM). It is directly proportional to the energy of the song, i.e. the tempo tends to be faster when the energy is higher.

Musical mode, or musical scale, determines the quantization of pitches of the notes in the piece. Different selections dramatically affects the listener in different ways. Generally, the scale is related to the stress of the song. For instance harmonic minor scale tends to be preferred more when the stress value is high.

Articulation in music theory is defined by the length of the note that gets played. This multiplier is inversely proportional to the energy, as the average length of a note tends to be shorter when the energy of the song is higher. As an example, *staccato* notes are commonly used to compose rather joyful musical pieces.

Articulation variation refers to the alteration of length of the notes in a certain pattern. Scientifically speaking, it could be seen as the deviation of articulation values over time. This property is directly proportional to the energy of the song.

Loudness multiplier determines how loud a note in the song will be played. The term could interchangeably used by the *volume* of that particular note. It has a direct relation to the energy of the song. For example, *piano* notes are often used to in music compositions to express a tender feel.

Loudness variation similarly refers to the alteration of the loudness of notes over time. It is directly proportional to both energy and stress levels of the song. For instance, when a musical piece expresses the emotion of *panic* or anger, the volume of the voices in the piece tends to fluctuate more often.

Pitch height refers to the overall pitch interval of the notes in the piece. It is used specifically for octave shifts to be applied to the notes. The parameter gets af-

fected by both energy and stress of the song. For instance, low pitched notes tend to be more suitable to achieve a moody—depressive—feel on the composition.

Harmonic curve could be defined as the direction of a melody in the composition over time. More specifically, it determines the pitches of consequent notes and the relation to each other. Likewise, it is proportional to both energy and stress of the song. When the curve is inversed, the mood of the song tends to feel more *down* and upset.

Timbre properties of the instruments plays a crucial role, yet they mostly are more or less overlooked in such systems. *Timbre* could be defined as the characteristic or the tone of the voice that an instrument produces. In this particular approach, the brightness and the tension of the timbre are taken in consideration which are both directly related to energy and stress levels of the song. For example, the timbre of an instrument tends to be hoarser in a sad song.

Note onset, also known as *attack* in sound synthesis community, is additionally used to enhance the effect on the timbre. It refers to the time interval between the start point of the playing note and the point when the note reaches its maximum volume. It is inversely proportional to the energy of the song. For instance, the value of note onset tends to reach near zero to give rather exciting expressions in a musical piece.

3.4 Audio Output Generation

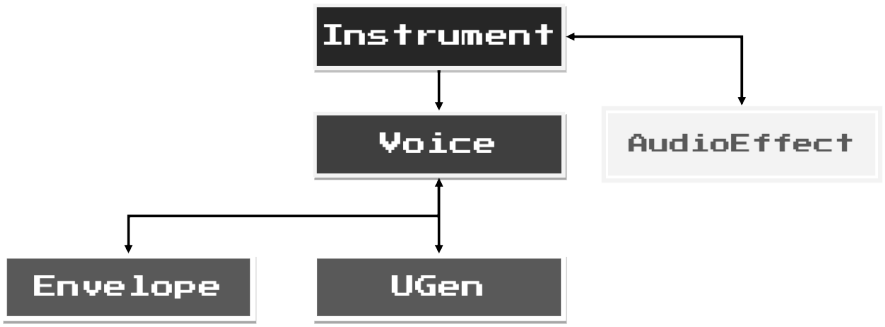

Figure 3.4: Instrument dependencies diagram.

Real-time sound synthesis and sampling techniques are chosen to be used to generate the audible output—rather than relying on more primitive technologies such as MIDI (as discussed in Section 2.2)—in order to achieve a sufficiently acceptable quality of sound as would be seen in other practical applications in the industry. More specifically, the desired audio data is generated procedurally from scratch or loaded from a pre-existing sound bank—sample by sample—by the instruments in the composition phase.

As seen in Figure 3.4, each instrument has certain number of *voice*s that generate the actual output. Voice components are capable of creating the sound by their *unit generators* either using sound synthesis techniques or using pre-existing sound samples as their audio data. The sound is shaped by the *Envelope* component when it gets played by the instrument. Moreover, after the creation process, the output can further be manipulated using *AudioEffect* components. Added audio effects, such as filtering and reverberation, are applied to each voice in the instrument to produce the final output values. Technical details of these structures are presented in the implementation phase in Chapter 4.

3.5 Limitations

Certain simplifications had be undertaken due to the limitations (time and resource) and the broadness of the topic, in order to maintain the research project feasible as a practical approach.

Firstly, sections that are obtained during the macro level generation process is restricted to certain enumeration types. The motivation behind this choice was to simplify and generalize the production for both developers and the end-users. More specifically, common song structure terminology is used for labeling the sections, namely *intro*, *verse*, *pre-chorus*, *chorus*, *bridge* and *outro* sections, which are then converted into single letters by their first letters accordingly. Still, the architecture is left open to such customization as adding more section types or bypassing the idea altogether if desired.

Even though it is presented as a feature, preserving the generator and instrument structures abstract is a way to reduce the work load in the implementation phase by providing only a proof-of-concept examples to be extended in the future. Nevertheless, this design choice enables not only an extensible but also a modular way to develop a customized, hence more diverse, applications by prospective users.

Another major simplification that was made as a design choice is to restrict the music transformation process to micro level – for notes only. In other words, musical form is built regardless of the musical properties selected by the user during run time to prevent the potential issues that might occur by such complexity. On the other hand, it could be possible to further manipulate the musical piece in a more sophisticated way by introducing macro level musical parameters such as the progression of sections and harmonies according to the selected mood.

Moreover, it is decided to keep the key note of the composition intact throughout the piece without any key modulations. The main reason why is the difficulty of preserving the consonance of a particular pattern, especially when the transformations take place. In fact, even when the scale remains the same, unintended dissonance is likely to occur in the piece when the key note changes. Thus, the harmonic progressions are limited to change only the musical mode of the piece without changing the key note of the scale. That being said, the key note still could be changed manually anytime during play if intended by the user.

All in all, the final design of the engine is capable of offering rather a generic solution for various needs and is effortlessly integrable to any external interactive systems and applications.

Chapter 4

Implementation

The engine has been implemented in C# making use of full capabilities of the language in order to provide a sufficient functionality in terms of the needs of the potential end-user. An application programming interface (API) has been developed, namely *BarelyAPI*, that enables a flexible use of the presented features to be integrated in any third party system easily. Moreover, a graphical user interface was included as a part of the framework, so that, it allows an intuitive interaction for various users with different technical backgrounds. Having said that, the system could be used not only by developers but also audio designers or even music composers themselves. The entire code of the project is open-source and publicly available on `http://github.com/anokta/barelyMusician`.

To get started, a brief overview on the available technologies that were suitable for implementing such a system is presented in Section 4.1 in order to reason the motivation behind the selection of the programming language and the environment used for the implementation of the engine. It is followed by the specific details of each component in the framework one by one—beyond the observation done in Chapter 3—in Section 4.2 to cover the technical means and challenges encountered in the implementation stage. Lastly, general properties and features of the API are examined in detail in Section 4.3 including an additional review of the graphical user interface and some supplementary features that broadens the purpose of the approach practically.

4.1 Technology Overview

Various software frameworks and libraries have been observed for the implementation phase from high level sophisticated programming environments such as PureData and SuperCollider to low level *plain* audio libraries such as OpenAL [38] and PortAudio [39]. As a matter of fact, they have been studied extensively by developing sample applications featuring basic functionality such as audio processing and sequencing to analyze and compare the capabilities of the environments. Having considered the flexibility, integrability and portability, C++ was initially chosen as the programming language for the implementation, based on PortAudio library for the communication via the audio driver, combined with Maximilian [40] audio synthesis library to wrap up the primitive functionality which is required for sampling and sequencing. Additionally, openFrameworks [41] toolkit was considered to be used for the demonstration purposes.

On the other hand, the main purpose for this research project is to provide a low-level dependency-free library to be used somewhat as a plugin structure in third party applications. After the research study and the design phase, due to the scope and complexity of this particular approach, Unity3D [42] game engine instead has been chosen as a starting point to prototype the framework. While, Unity3D does not offer any specific audio features with respect to the focus of the project, it makes a perfect environment to quickly prototype the desired features and test the expected behaviour of the system before going further in the process. As a matter of fact, after the prototyping stage, the final decision has been made to shift the development of the framework as a whole in Unity3D, as it was powerful enough to illustrate all the capabilities of the architecture in an efficient way.

One major advantage of using Unity3D is its powerful community and wide range of developers and users. Thus, it was much easier to spot and suspect potential flaws in the design as well as the implementation, so that, the final product could serve in more practical aspects in a user-friendly manner. Moreover, it is frankly beneficial to see the prospective interest in the framework beforehand.

One other motivation behind this choice is the multi-platform support. More specifically, Unity3D enables the final product to be run in different platforms such as Windows, Mac & Linux without any major effort in the implementation in terms of compatibility. In fact, the framework could even be used as it is in mobile systems such as in Android, iOS & Windows Phone platforms. Particularly, it enhances the potential practical use by delivering the approach to a huge community who had not experienced such capabilities before.

All the same, the final product is aimed to be as independent as possible from any other third party libraries to be able to provide a somewhat robust and flexible interface. That being said, the proposed design and implementation of the approach could easily be established in another environment and/or ported if desired respectively.

4.2 Main Components

The implementation of the components in the engine strictly follow the design of the main architecture as discussed in the previous chapter. All the components in the framework have been implemented from scratch including the algorithms required for the generation phase. The choice was made to manage such flexibility of manipulation of each part in the system sufficiently in every little detail. Having said, it was a challenging yet beneficial experience to work on such an inclusive system with more than fifty classes with thousands of lines of code.

Main components are described in detail below one by one in a bottom-up fashion. Technical details are as well presented in order to provide a deep understanding of the specifications and the infrastructure of the engine.

4.2.1 Sequencer

Sequencer component has been implemented in a way to serve as the main clock of the engine to sequence the musical pieces in order. During run time, it updates and

stores the current state of the system. In order to manage this functionality, an event-triggering system has been developed with a callback mechanism. More specifically, *Sequencer* class has a counter, namely its *phasor*, which gets incremented in the audio thread once per each sample.

How does the audio thread work?

At that stage, it is worth to examine how the audio thread works in greater detail. Basically, in order to produce the audio output digitally, a system must implement a callback mechanism that communicates with the audio driver and fills the audio data buffer sample by sample [43]. Similar to a graphics unit, the audio output is produced by an array of floating point values—that are normalized to [-1, 1] interval—which are referred as audio *samples*, analogous to filling the color values per each pixel on the screen. Those values represent the physical position offset of the speaker cone resulting in a continuous vibration of the cone that generates the output of audio signals (waves) which are audible in the real world.

In this particular approach, as a digital setup, the array (audio buffer) expectedly has finite number of values (samples). The number of samples that are used to generate the output per second is usually notated as the *sample rate* or *sampling rate* of that audio system [44]. The sample rate is often measured in Hertz (Hz) as in the frequency of sound, such as the typical 44100Hz sample rate. Even though the approach seems to be trivial enough, the idea is not feasible to be used in real-time systems. In other words, as the sampling rate suggests, the audio driver must be fed by exact number of samples in each second. For instance, in order to sustain the audio output in 44100Hz rate, the values must be processed and passed into the driver 44100 times in a second, for every single second. It means that the computation must be done precisely in less than a milisecond in each time. As one can expect, that is not a practical solution even for the state-of-the-art computers in the industry when processing gets even slightly complicated. Beside that, there is an unacceptable communication overhead between the system and the driver.

To resolve this issue, raw audio data is passed to the audio driver as buffers, i.e. certain number of samples are processed and bundled together before sending them to the driver. As a result, the overhead is minimized and the whole process becomes feasible while it does not affect the outcome, since the rate is still beyond recognizable to perception of the human ear. The choice of *buffer size* may vary according to the application, it is a trade-off between the latency (performance) and the quality of the output. The typical length for an audio buffer is between 512 and 2048. For instance, for the buffer size 1024 with the same sampling rate, the callback interval would be approximately 50 miliseconds which frankly allows complex calculations and operations practical such as filtering and so on.

All in all, by using such a counter as mentioned in the beginning, it is possible to measure the elapsed time period precisely which is crucial for the audio output generation. That being said, *Sequencer* manages this information in a hierarchical way to create the pulses, beats, bars and sections respectively. Relevant calculations are done with respect to the selected tempo and time signature. For example, if the tempo is 120 beats per minute (BPM) and the time signature is 4/4, then the sequencer sends beat signals in each 60/120/(4/4)=0.5 seconds and bar signals in each 0.5*4=2 seconds.

4.2.2 Instrument

Similar to *Sequencer* component, instruments also make use of the low-level audio thread in order to generate the final audible output. The component has been implemented as an abstract class with generic functionality that a typical real-world instrument would have. Every instrument has a certain number of voices and a master gain parameter to adjust the volume of the output. The instrument is played by triggering *note on* and *note off* methods respectively.

As mentioned in the previous chapter, the audio buffer is filled by the voices of the instrument. The generation of that data is done by summing up the outputs of the unit generator process of each voice. Having said that, there are mainly two types of unit generators, *oscillator* and *sampler*. *Oscillator* synthesizes the sound by us-

ing mathematical representations of typical audio signals such as in the form of sine, cosine, saw, square, triangle and white noise waves. For instance, in the most basic case, the desired audio data is generated by simply computing the trigonometric sine function with respect to the current state (time interval of the audio thread). The result is then written to the audio buffer which indeed gives the sine wave sound in an audible form. *Sampler*, on the other hand, loads the desired pre-recorded array of samples—traditionally known as an *audio sample*—and write that data directly to the audio buffer in a similar fashion.

In order to support *polyphony*, i.e. to manage multiple voices simultaneously, a channel based registration model has been implemented [45]. When a new note is queued to be played in the instrument, it looks up the list of free voices and registers that note by allocating the first available voice for play. If none of the voices are free, the new note *steals* the voice channel with the oldest registered note. Linked list data structure is used in the process for efficiency.

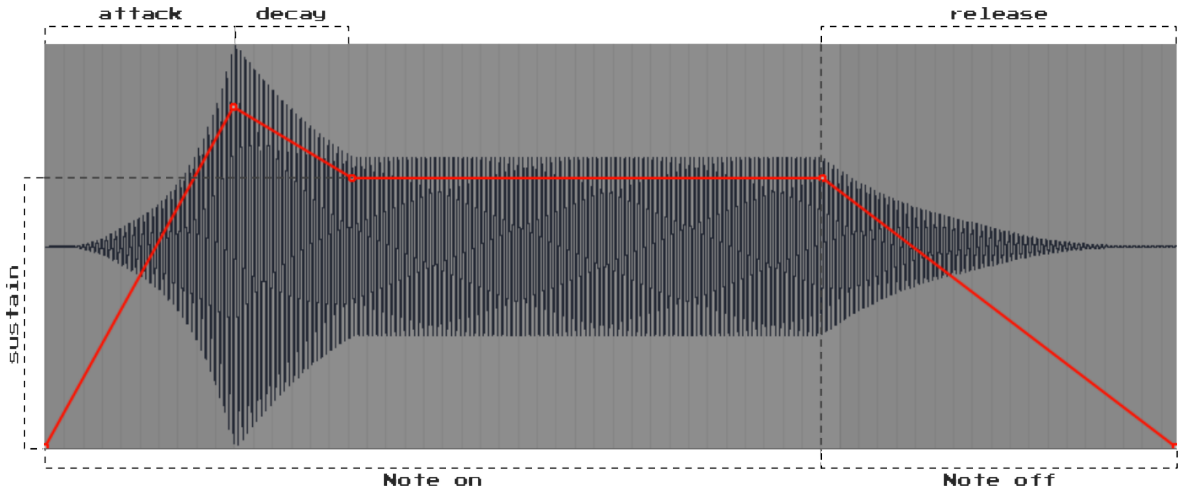

Figure 4.1: Sound envelope example (horizontal axis: time, vertical axis: amplitude).

Envelope of the instrument is used to give a specific shape to the resulting audio wave and could be used to smoothen the signal in real-time. The structure of the envelope has been implemented with respect to the traditional *sound envelope* architecture of a typical digital synthesizer [46], providing *attack*, *decay*, *sustain* and *release* values.

The sound envelope basically offers an amplitude interpolation for the generated sound. The idea is illustrated by an example in Figure 4.1.

Optionally, audio effects are applied to the generated audio samples. *AudioEffect* component has been implemented as well as an abstract class to offer full flexibility in terms of the manipulation of soundscapes. That being said, all the parameters mentioned above are modifiable in real-time in order to support the means of adaptability even in low-level, literally sample by sample.

4.2.3 Generators

There are four types of music generators that have been implemented in the engine to manage the composition process in a hierarchical way, namely *MacroGenerator*, *MesoGenerator*, *MicroGenerator* and *ModeGenerator*. Each component has been implemented as an abstract class with certain generation functions to be overwritten in any way the end-user intends to.

At top level, *MacroGenerator* is responsible for generating the musical form, i.e. the sequence of sections of the composition. *generateSequence* function is used to generate the sequence which is then stored as a string of characters - that represent the sections in order. As mentioned above, it is possible to override this function either by a manual sequence of choice or in a procedural way using certain generation algorithms. *GetSection* function is available for fetching—and generating if not available beforehand—the label of a certain section during run time. The song duration—the number of sections to be generated for the sequence—could be determined in real-time, so that, the user might specifically select how long the song should be. Moreover, the generated sequence is loopable if desired, hence, it enables the user to generate a certain musical piece and use it repeatedly in the application straightforwardly.

MesoGenerator creates the harmonic progression for the given section. *generateProgression* function is used to generate a desired number of integers which represent the harmonic pattern in order. Likewise, *GetHarmonic* function is available to get the

value of specific index (bar) of a specific section in run time. All the generated progressions are stored in the component for potential later use.

At the bottom, *MicroGenerator* is used to generate a unique note sequence for the given bar. *generateLine* function takes the section label and the harmonic information of that bar and creates the notes accordingly. The generated sequence might consist of melodic patterns and/or chords. Notes are stored as meta information – using *NoteMeta* class, so that they could be transformed easily in the process when needed by the conductor. Generated lines as well are stored in a list, thus, it is even possible to use previous iterations to generate the next pieces. As a result, it allows a cohesive functionality for the creation of rather sophisticated music compositions.

ModeGenerator plays a slightly different role than the others. This type is responsible for providing the musical scale to be used in the composition. More precisely, the scale is determined by the current mood of the system using *GenerateScale* function. The implementation includes some of the most popular scales and modes such as major, natural minor, harmonic minor scales and ionian, dorian, phrygian, lydian, mixolydian, aeloian, locrian modes. For instance, the major scale could be easily mapped to a happy mood by overriding that function accordingly. That being said, it is still possible to implement more experimental mapping methods even using different type of scales such as the pentatonic scale. When the scale is generated, it is stored as an array of note indices of one octave (array length might differ according to the scale choice). They are then used as offsets from the key note to produce the final notes. It is worth mentioning that, the indices might be made of floating points to produce unique pitches beyond the traditional equally-tampered scale of twelve notes. Having said that, unlike simpler musical score generation approaches such as in MIDI case, the generation process of the proposed engine offers great means of experimentation if desired by the user.

4.2.4 Ensemble

Ensemble component jointly serves as the musical orchestra that generates and performs the generated pieces in real-time. The actual performance is made possible by

the performers in the ensemble using their instruments respectively. Ensemble has been implemented as the management class of all the performers to be used. It is possible to add, remove and edit performers in run time when needed. The functionality also includes muting (bypassing) specific performers in real-time to add more flexibility to the user. In each relevant audio event that is passed by the main class, it iterates through all the performers and updates their states accordingly. *MacroGenerator* and *MesoGenerator* components are stored in the ensemble and managed through each new section respectively.

When the next bar signal is received, each performer is responsible to fill the bar in its score using its *MicroGenerator*. While those generated lines are stored inside the generator with respect to the section and harmonic information, the performer stores the whole musical score – from beginning to end, so that, it is possible for the user to jump to a specific part of the piece in real-time by changing the current state of the sequencer if needed. The musical score is stored as actual *Note* structures which are produced after the transformation of meta notes by the conductor in each beat. Therefore, the approach allows the user to manipulate the piece in a highly-grained fashion to achieve smooth transformations anytime during the playback.

4.2.5 Conductor

As described in the previous sections, *Conductor* is responsible for the transformation of the generated notes in real-time. It additionally stores the key note—the fundamental note index—of the musical piece that is used to achieve such functionality in terms of *centricity* principle (see Section 2.1.1).

The transformation process is done with the aid of certain musical parameters which are stored inside the class. Different mapping strategies have been applied to achieve an adaptive feel in terms of affectiveness of the output, as seen in Table 4.1. The mapping is done in two steps; first the normalized value of the relevant musical property is computed using the energy and stress values, then the value is scaled according to the specified interval of that property. For instance, to get the final

tempo value for the playback, the tempo multiplier is calculated using the energy value. Let's say if the energy value is 1.0, then the tempo multiplier will be computed by 0.85+(1.0*1.0)*0.3=1.15. Then, the outcome is multiplied by the initial tempo, say 120 BPM, resulting in 1.15*120=138.

Musical property	Interval	ΔEnergy	ΔStress
Tempo multiplier	[0.85, 1.15]	1.0	-
Musical mode	[0.0, 1.0]	-	1.0
Articulation multiplier	[0.25, 2.0]	-1.0	-
Articulation variation	[0.0, 0.15]	1.0	-
Loudness multiplier	[0.4, 1.0]	1.0	-
Loudness variation	[0.0, 0.25]	0.5	0.5
Pitch height	[-2, 1]	0.25	-0.75
Harmonic curve	[-1, 1]	-0.75	-0.25
Timbre brightness	[0.0, 1.0]	0.6	-0.4
Timbre tension	[0.0, 1.0]	0.8	0.2
Note onset	[0.25, 4.0]	0.5	0.5

Table 4.1: Mapping between the musical properties and the mood.

For the note transformation; pitch index, relative offset, duration and loudness of the note are computed separately using the meta information of that note and the outcome is returned to the owner (performer) of the note to be added to the score and then played by its instrument. *ModeGenerator* component in the conductor is actively used in this process in order to determine the final pitch index of the note properly. More specifically, the initial pitch index is first converted with respect to the current scale, and it is added to the key note index to produce the final pitch index of the note. That being said, the conversion between the pitch indices and actual pitches (frequencies) are calculated by the mathematical formula with respect to the theory of musical sound [47], in which the neutral pitch index—zero—is the *middle A* note (*A4*), that corresponds to 440Hz in frequency in common musical knowledge.

4.2.6 Musician

Musician component serves as the main class of the engine, enabling the communication between the other components and the management of the system in general as described in Section 3.1. That being said, the implementation of the class is the only one inheriting from Unity3D's *MonoBehaviour* base class in order to be able to add an instance of that object as a component to the game object hierarchy in the game engine to enable direct interaction for the user.

Musician class stores all the necessary information for setting the preferences of the music composition system in real-time. *initialTempo* parameter is used to determine the tempo of the musical piece to be generated. Song duration could be set in a similar fashion by using *songDuration* parameter using floating point values in minutes. Number of bars per sections and beats per bars could be adjusted using the class variables. This adjustment is used by the engine in order to determine the time signature of the composition. Having said that, benefiting from the flexible sequencing architecture, *irregular* time signatures are furthermore supported. In other words, it is possible to obtain less common rhythmic structures, such as 5/4 and 7/8, to achieve more interesting—and sounding arguably non-traditional—results. The key note of the song is determined by *rootNote* parameter using *NoteIndex* enumerator. Additionally, the master volume of the output could be set anytime using *masterVolume* parameter in the class.

Mood	Energy	Stress
Exciting	1.0	0.0
Happy	0.5	0.0
Tender	0.0	0.0
Neutral	0.5	0.5
Depressed	0.0	1.0
Sad	0.25	0.75
Angry	1.0	1.0

Table 4.2: High-level abstractions and the corresponding mood values.

Energy and stress parameters are also stored in the component. Whenever a change

occurs in those values, relevant music parameters (see Table 4.1) are updated accordingly. The implementation provides *SetMood* function to adjust these values smoothly by providing a *smoothness* parameter. More specifically, if smoothness is greater than zero, the adjustment will occur using exponential interpolation between the initial and target values resulting in a pleasant transition. Moreover, high-level abstractions are introduced in that manner by overloading the function, providing certain emotions such as *happy* and *sad* which are meant to be significant to the end-user, to achieve a more intuitive way of user interaction. While limited study has been done using several resources in the area [48], selections of those emotions are rather empirical, hence, for ease of use only. The corresponding energy and stress values for the selected emotions can be seen in Table 4.2. Nonetheless, the user can modify and/or extend the functionality easily by introducing new emotions to the system. As mentioned, *SetMood* function alongside allows smooth transitions by passing one of those emotions as a parameter.

4.3 API Properties

The implementation of the project has been done in a generic way by avoiding any platform-specific structures in order to keep the core functionally portable across different platforms and environments. The main goal of the framework is to provide a modular and an extensible approach which enables a powerful yet customizable engine for any type of users to be used as a personal tool in their projects. To enhance the level of interactivity, a custom editor is included to the engine in addition to the programming interface. Moreover, additional features such as keyboard controller and real-time audio recorder are provided inside the framework, which are briefly reviewed below.

4.3.1 User Interface

A custom editor has been implemented for the engine to enhance the user interaction using Unity3D editor extension tools [49] as can be seen in Figure 4.2. It was somewhat

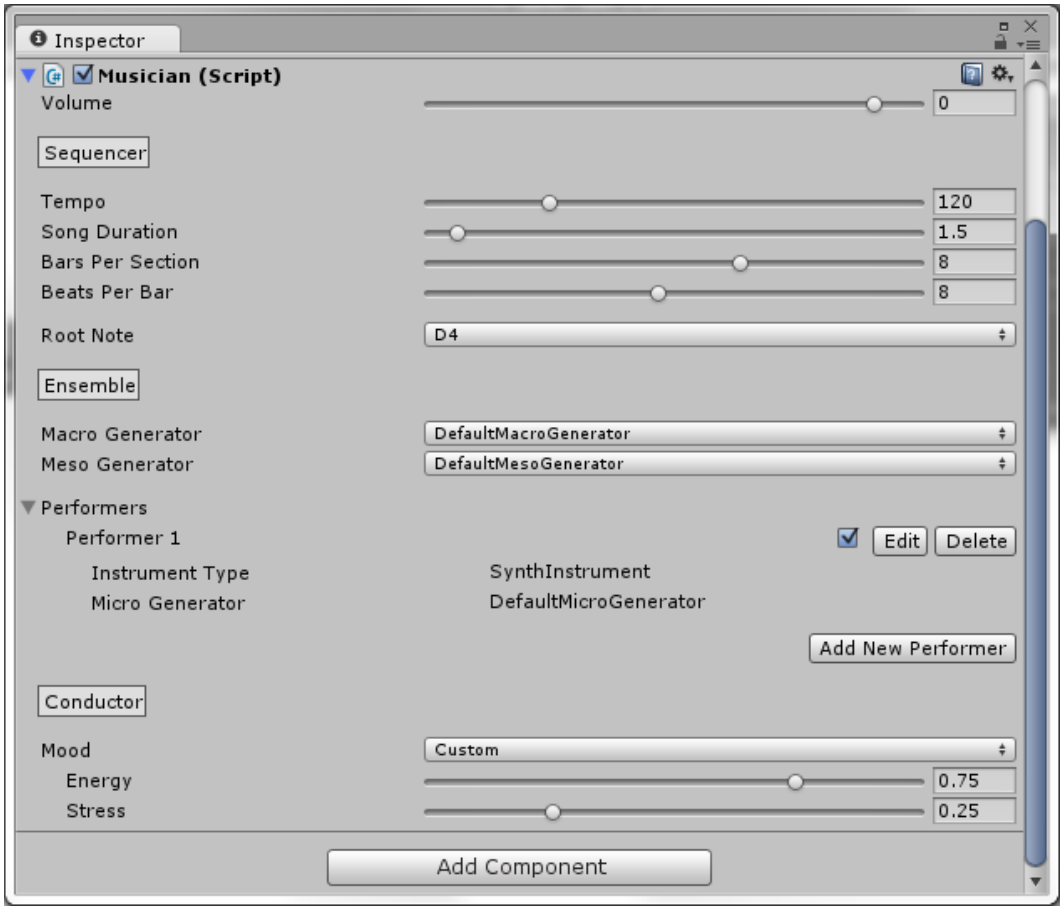

Figure 4.2: Sample screenshot of the custom editor for the main component.

a challenging attempt considering how the game engine works in terms of graphical user interface (GUI) content. Unity3D serializes and deserializes everything GUI related in use whenever the scene is played, stopped, saved or loaded. In other words, all the information that is visible to the user must be serializable for them to work properly in the scene. To achieve that, all the classes which store crucial parameters—including *Musician* and *Instrument* components—had to be modified accordingly in order to implement serializable behaviour accordingly.

In light of the foregoing, *factory method pattern* was put in to the system, namely with *GeneratorFactry* and *InstrumentFactory*, so that selected presets in the editor could be instantiated in a generic way. Type names of those presets are saved as re-

sources in the engine to make them directly accessible to the user whenever a new preset is added. The names then are used in the mentioned factory classes to instantiate an instance of the selected presets respectively.

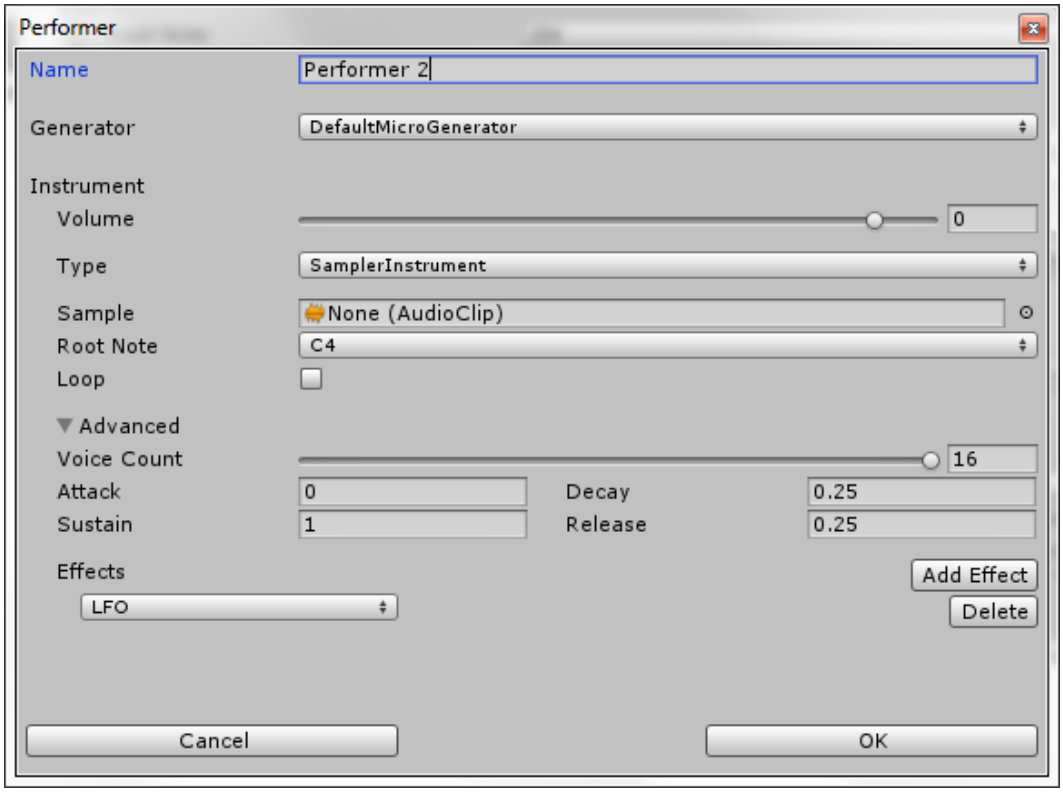

Figure 4.3: Sample screenshot of the custom editor for performer creation.

To manage the parameters in a more user-friendly manner, sliders and popup selections were introduced in the editor that restrict the user from choosing invalid values for such parameters. Overall, all the necessary settings which are crucial for music generation process are visible and modifiable by the user either in editor mode or during run time, for an easy and efficient use without having to be caught up in the code.

In the meantime, an additional editor—in the form of a popup window—has been implemented to provide a more detailed interface for performer creation (see Figure 4.3). A performer could be added straightforwardly using this editor even in real-time. The editor displays all the necessary information such as the name, the generator and

the instrument type (including its properties) of the performer, in an editable fashion. That being said, the user can edit or delete previously created performers by using the same window.

4.3.2 Supplementary Features

Apart from the features described in this section, certain supplementary functionality has been implemented for the programming interface to extend its capabilities further.

Keyboard Controller

A simple keyboard controller has been implemented that functions similarly to the ones that are used in digital audio workstations. More precisely, certain keyboard keys were mapped to resemble the piano keyset of an octave, providing an additional key pair for changing the octave when needed. Any instrument can be chosen to be played by the keyboard controller in real-time. The main motivation behind the implementation of this feature was to *playtest* the instruments beforehand to see if they sound as intended for use in the ensemble. Nevertheless, it is fairly possible to use the controller in the same key of the music composition and actually play along with the performance of the engine during the playback. In fact, in that manner, this feature might enable new ways of interaction for the user inside such an interactive system.

Recorder

Recording the output of the engine is offered anytime, real-time, which as well could be used to pre-compose pieces with desired moods, to be applied as background music beyond interactive systems such as film scores for movies[1].

[1] An example music composition that was recorded in real-time using the engine (without any further processing) is presented available online on `http://soundcloud.com/anokta/barelyhappy`.

Audio Event Listeners

Last but not least, the audio events, which are reviewed in Section 3.1 and Section 4.2.1 respectively, are not only used internally but also could be accessed externally by using the programming interface. That being said, custom event listener functions can be easily registered to the sequencer to get relevant signals on each pulse, beat, bar and section respectively.

A congruency between visual and audio elements in a virtual environment is very important for perception and emotional involvement. Having considered this, even though the functionality is provided as an extra feature, its capabilities are powerful enough for the engine to be used as the main motivation. The reason why is that most of the recent interactive applications seem to suffer synchronization issues when they were to try to combine musical elements and visual elements together. The difficulty is caused by the intolerance in latency for audio elements, thus, the synchronization method has to be perfectly accurate in order to achieve a compelling—or even an acceptable—outcome. Fortunately, with the low-level approach of the engine, it is feasible to implement such features by using precisely synced callback functions. To elaborate the idea further by means of game development, the engine could be used for not only game music but also developing *music games* themselves.

Chapter 5

Demonstration

Two demonstrative applications have been developed in order to experiment and evaluate the potential practical capabilities of the implemented engine[1]. As a proof-of-concept, the former features a typical demo scene with a graphical user interface, and the latter features a simple game prototype using the engine solely for the audio content. In addition, a number of presets with diverse methodologies have been implemented and in fact have been used in mentioned applications in order to illustrate the modularity and the extensibility of the engine.

5.1 Sample Presets

As discussed in the implementation stage in Chapter 4, generators, instruments and audio effects are implemented as abstract classes to provide flexibility for the engine. The idea resulted in the notion of *presets*, which could be implemented in a straightforward way by deriving from those classes. In order to illustrate the idea, example preset implementations have been done for different needs in the environment. Moreover, to proceed the approach further, certain generation algorithms were studied and implemented as a part of the core structure of the engine not only for demonstration but also for potential later use.

[1] A sample footage of the demo scene could be accessed through `http://youtu.be/Qs3yDVtqt9s`.

As an initial step, the implementations were done for the simplest cases such as by providing static pre-defined generations to serve as the default behaviour. In other words, concrete classes for the most basic scenarios have been implemented for the end-user to try out the engine without worrying about extending the system in more complex scenes. It was also an important step to test and debug the implementation beforehand for the demonstration stage.

For the generation of musical form, generative grammar techniques has been chosen to be examined. More specifically, L-systems and context-free grammars were studied in order to model a procedural algorithm that could be used for the generation of section sequences [50]. As a result, a generic rule-based implementation has been done, namely *ContextFreeGrammar*, which is capable of generation of a string of sections with any size. It provides a list of rules—the ruleset—that can be specified in real-time and a starting point to iterate the sequence to a desired length of non terminating symbols, as illustrated in Table 5.1.

Ruleset
Start → Intro Body Outro
Intro → A | A B
Body → B B C | B C | Body D Body
Outro → C E | E | Intro E

Terminal symbols
A B C D E

Sample iteration
Start
Intro Body Outro
A **Body D Body** Intro E
A **B B C** D **B C** A B **E**

Table 5.1: Sample ruleset and an arbitrary iteration of the generative grammar algorithm.

For the harmonic progression generation, Markov chain model has mainly be chosen. Likewise, a generic n-th order Markov chain algorithm, namely *MarkovChain*, was implemented in order to illustrate the behaviour. In order to provide somewhat compelling results in terms of the chord progression schemes, further study has been done to obtain relevant statistical data from several sources [51].

For generating the musical patterns, several methods have been studied to illustrate

diverse possibilities the engine could offer. Firstly, due to its popularity in the field of algorithmic composition, cellular automata algorithms have been analyzed. An example program was implemented modeling a 2D cellular automaton to resemble Conway's Game of Life. While the cell rows in the automaton were used to determine the timing of the notes to occur and get played, the cell columns were used to determine the relative pitch index of that particular note. An arbitrary iteration of that process is illustrated in Figure 5.1. For instance, at the beginning of that bar, say if the key note is C and the scale is major, C E G notes will be played simultaneously, resulting in the C major chord for that beat.

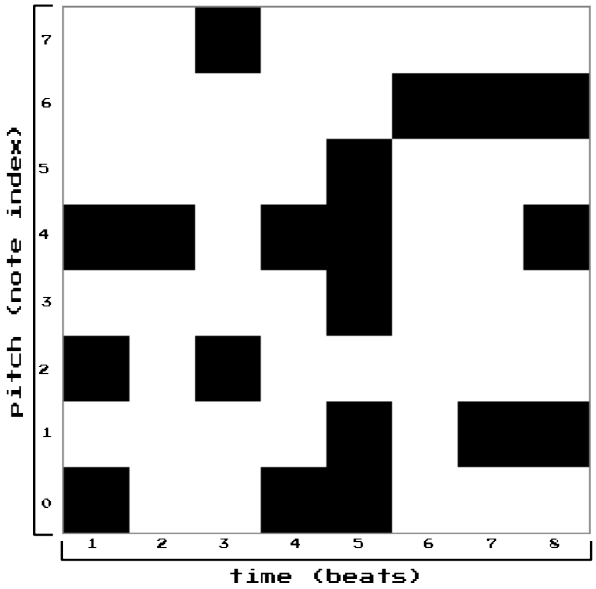

Figure 5.1: Sample iteration and note mapping of a 2D cellular automaton.

While two dimensional automata were able to produce interesting progressions to a certain extent, the outcome in general was felt too *random*, which arguably failed to achieve promising results in terms of generation of convincing musical patterns. To overcome this issue, a simplified version with a single dimensional automaton has been implemented. In this case, the cells are only used to determine the placement of the notes rhythmically, i.e. they are used in terms of timing of the notes. In fact, an experimental generator has been implemented with a hybrid approach, combining cellular automata algorithm and Markov chain algorithm together to achieve more compelling

results. While the cells of the automaton are used for timing as described, the selection of note pitches are determined by the Markov process. The resulting implementation not only gave a more compelling outcome but also illustrates the flexibility of the usage of the stated architecture.

Apart from those, around ten more generators have been implemented with various functionality and algorithm choices. One interesting generator among those was the one which is used for percussion patterns generation. The approach of this particular generator slightly differs from the rest, as it does not take into account of musical notes, rather focusing on the rhythmic patterns. Stochastic binary subdivision algorithm was used to achieve that capability, using certain probabilistic methods to determine where to place the percussive elements [50]. While the generator is intended to be used by percussion instruments, it is worth noting that the process frankly produces interesting outcome alongside for melodic instruments.

Three types of instrument presets have been implemented to be used in the demonstration. *SynthInstrument* and *SamplerInstrument* were implemented in a similar manner, using *MelodicInstrument* base class. They offer a typical digital instrument functionality, providing an envelope and polyphonic voice capabilities. While *SynthInstrument* makes use of sound synthesis implementations (*Oscillator*), *SamplerInstrument* uses samples (*Sampler*) to produce its output. In addition to those, *PercussiveInstrument* has been implemented to be able to play percussion patterns such as the drums as mentioned above. It takes certain number of samples to be used, such as the kick drum, the snare drum and the hi-hat cymbal samples, and maps them to specific notes to be played without any pitch modulation. The approach is more or less similar to the way it is used in any typical digital audio workstation.

Lastly, two basic types of audio effects have been implemented to illustrate the practical use. *Distortion* is a simple distortion audio effect which amplifies and clamps the incoming audio signals to produce the *distorted* feel of the outcome. The initial level of distortion is provided as a parameter, hence, can be modified by the user in real-time. Similarly, *LFO* audio effect uses an oscillator to modulate the amplitude of the incoming audio signal in the specified frequency. It could be used to create a

dynamic output which would commonly be found on modular synthesizers.

5.2 Proof-of-concept Applications

Demonstrative applications were developed not only to illustrate the proposed approach onstage but also to introduce the concept to the community beforehand. They furthermore served as a basis to validate the practical means of potential use during the evaluation stage. Having said, the purpose of the applications are only to show proof-of-concept examples to be extended further in a later stage.

5.2.1 Demo Scene

Figure 5.2: Sample screenshot of the demo scene.

The demo scene, namely *barelyDemo*[2], serves as the main demonstrative application that displays the general functionality of the engine. A simple graphical user interface is provided for the user interaction. Certain settings are pre-specified in the application such as the number of sections, bars and the performers in the ensemble (see Table 5.2) which were to demonstrate a typical music composition that could be found in traditional Western music . The musical piece gets generated by the generative grammar approach in macro level and the Markov chain approach in meso level. There are seven performers with different roles to present a rather rich output in terms of quality of music. Performers include some essential roles as the lead melody, the bass, the chords, the drums and the backing strings.

Parameter	Value
Volume	0 db
Initial tempo	140
Song duration	2 min.
Bars per section	8
Beats per bar	4 (4/4)
Key note	Middle A (A4)

Table 5.2: Demo scene settings.

The user can play, pause or stop the song anytime as he pleases by clicking the corresponding buttons (see Figure 5.2). In each play, the engine generates a new piece to be played and looped continuously. The user can interact with the generated music during play by either clicking the high-level mood abstractions or adjusting the energy and stress values manually using the sliders. Changes between the mood abstractions are done by interpolation between the values to further smoothen the outcome.

During the playback, a minimalistic animation in the background is played in sync with the beats of the song, in order to briefly show the audio event listening feature. It is also possible to *play along* with the song by using the mapped keyboard keys to play a sample synthesizer instrument adjusted in the same key note of the musical piece.

[2] *barelyDemo* is publicly available online on `https://dl.dropboxusercontent.com/u/19989225/IET/barelyMusician/barelyDemo/barelyDemo.html`.

Lastly, record button could be used anytime to record and save a certain part of the song in real-time.

5.2.2 Game Prototype

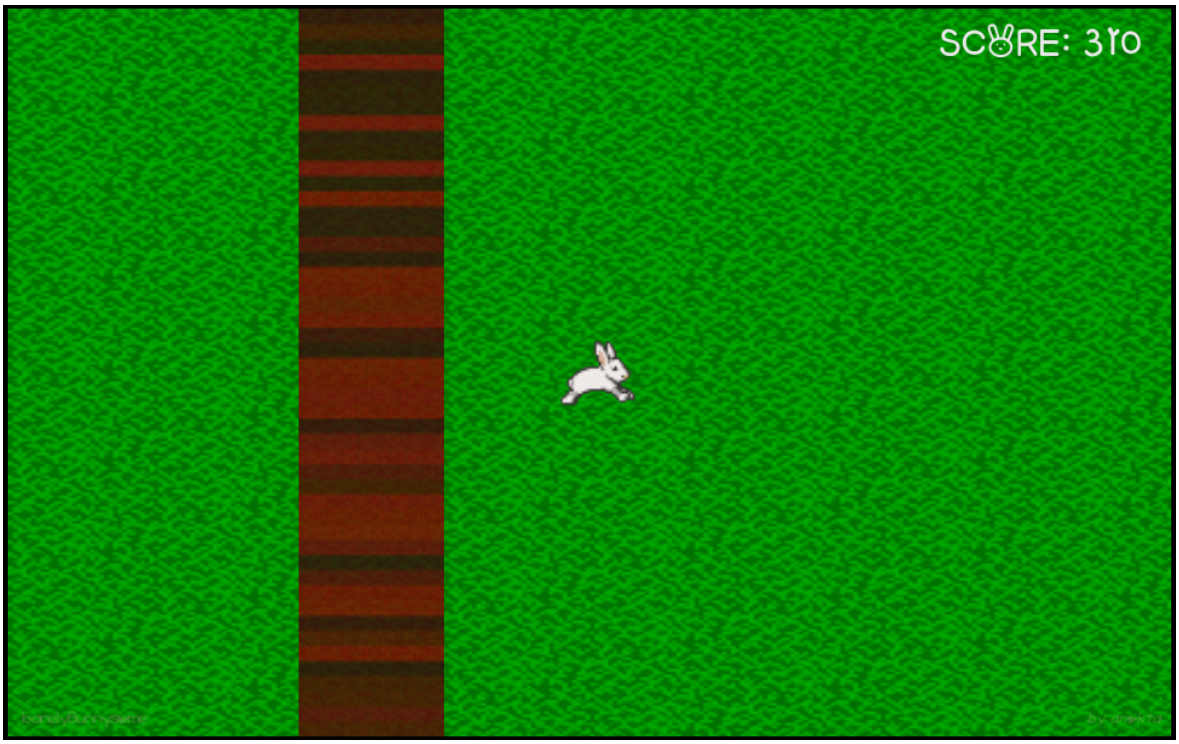

Figure 5.3: Sample screenshot of the game prototype.

The game prototype, namely *barelyBunnysome*[3], has been developed upon a fairly simple idea to illustrate how the engine would work in an external application. This attempt was an important step to evaluate the power of the programming interface and editing tools alongside the efficiency and the quality of the final product.

Game mechanics were kept as simple as possible, so that, the results could be deduced clearly without any distractions. That being said, it is a classic endless high-score

[3] *barelyBunnysome* is publicly available online on `https://dl.dropboxusercontent.com/u/19989225/IET/barelyMusician/barelyBunnysome/barelyBunnysome.html`.

game where the goal is to survive as long as possible without touching the hazardous red zones on the screen (see Figure 5.3). The red zones move in various patterns that are precisely synced with the beats of the song.

Parameter	Value
Volume	-6 db
Initial tempo	152
Song duration	1 min.
Bars per section	2
Beats per bar	8 (8/4)
Key note	Middle C (C4)

Table 5.3: Game prototype settings.

Game audio is generated solely by the engine on the fly. The background music implements similar parameter settings to the demo scene (see Table 5.3). The application begins with *tender* mood displaying the main menu. When the game starts the mood shifts to *happy*, which rises the energy level. After that, in each new section, the energy gets higher by a certain amount until it reaches *exciting* mood. Initial tempo starts to get increased after that point to further enhance the pace of the game. Both the protagonist (*bunny*) and the hazardous zones (*killzone*) move perfectly in sync with the tempo of the song. Thus, the more exciting the music gets, the harder the game becomes. When the protagonist dies, the music dramatically shifts to *angry* mood to emphasize that the game is over. After restarting the game, the piece shifts back to *happy* mood smoothly.

Additionally, there are couple of performers in the ensemble that serve as somewhat sound effects, triggered by the user input in their generation process. More precisely, the movement of the bunny and the killzones are supported by additional instruments that fade in whenever they move, which are coherent to the rest of the musical piece. As a result, the final outcome is in fact produced jointly by the game mechanics and the user interaction, which arguably offers a unique experience to the player even in this simplest case.

Chapter 6

Evaluation

An informal survey has been prepared based on the demonstrative applications in order to experiment and gather some feedback from a few dozen of people with varied musical backgrounds. The survey was planned to address two main focuses, which are presented below combined with some personal thoughts. Further discussion is done in Section 6.3, particularly in terms of potential practical usage of the system, to conclude the chapter. The survey questions are provided in the appendices for more information (see Appendix A).

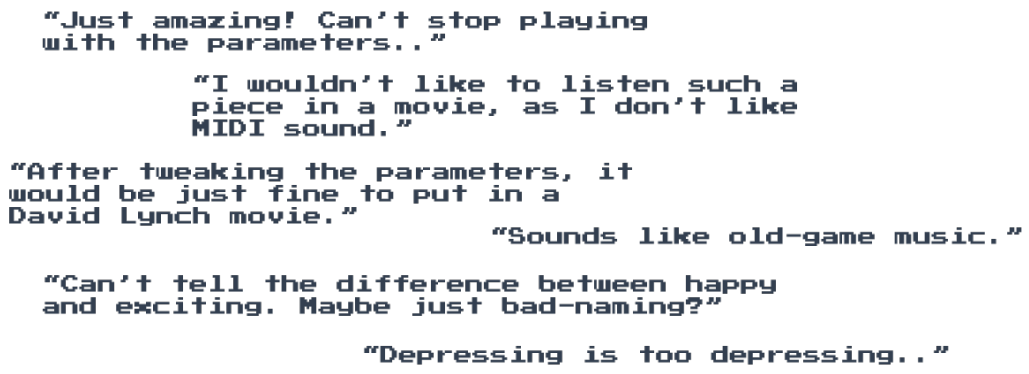

Figure 6.1: Some interesting comments from the participants.

6.1 Level of Interactivity

First focus of the evaluation step was to determine how the user interaction affects the participants. The main goal was to measure the level of interactivity and affectiveness that the user experiences. In light of the foregoing, almost all the participants responded with positive feedback on the change of the characteristics of the songs with regards to the modifications they made during the playback. Over 95% of the participants stated that the changes were clearly recognizable. The comments were concrete enough to validate that they were immediately able to perceive the difference between different settings of mood selections during play. Particularly, they reacted positively to the smoothness of transition between particular values even when adjusted manually.

On the other hand, some participants were not able to differentiate certain mood settings from each other, especially such in the case between *tender*, *happy* and *exciting*. Few responded that they were too similar to each other. Moreover, while such moods as *depressive* were spotted easily by the participants, some commented on such as *angry* as feeling rather *upset*. In that case, it is worth to study further on the abstractions of moods, particularly renaming and readjusting values might be an appropriate attempt to resolve such issues. Having considered that, certain emotions are known to be more difficult to perceive by the listener[30], hence, further experimentation with different scenarios—e.g. with visuals and a narrative support—might be beneficial in the future to obtain more precise results. Nonetheless, the main focus of the stated approach does not take into account of those abstractions specifically, hence, in terms of the energy and stress levels, it can be said that the resulting work is able to accomplish the purpose sufficiently.

6.2 Quality of Generated Audio

Participants were asked to evaluate the generated musical pieces in comparison to traditional music compositions which could be found on other interactive entertainment systems, including film scores and game audio. There were mixed reactions in this topic, presenting both positive and negative feedback. In order to obtain a more clear

evaluation, the question was prepared to address the quality of the audio elements and the quality of the composition individually.

In terms of quality of the audio elements, i.e. the instruments, arguably due to the motivation of subject of the approach, there was a high tendency towards typical game music elements to resemble the sound of the instruments in the piece. This could be counted as an achievement, since it completely coincides the main purpose of the project. However, it also means that, in this current stage, the outcome somewhat fails to offer such sophisticated audio elements which would be found in the mainstream music industry and/or high-budget entertainment systems. In fact, that fallacy gained strength when one of the participants commented on the produced sound stating he would not be pleased to hear such a song as a film score due to his disinterest in MIDI music (see Figure 6.1). This particular feedback was intriguing enough for the need of precisely recovering the output at a later stage, since none of the MIDI virtual instruments—or even the structure itself—do take part in the output generation process at all.

For quality of music, the gathered feedback was more likely to be positive and encouraging to proceed further. Some participants were, in fact, surprised to hear that the musical pieces they listened to were generated autonomously in real-time. This particular type of response was a clear accomplishment for the engine, considering the fact that every single thing related to the audio content is generated from scratch in low-level on the fly. Therefore, when combined with other comments from the participants, it can be said that the resulting musical composition is no worse than an average song that would be composed by a typical Western composer. It further validates that the outcome is sufficiently acceptable in that manner for a typical Western listener.

While it is *acceptable*, some participants (with broader musical backgrounds) commented on the song as being *monotonous* over time. This issue was arguably caused by overfitting certain musical parameters for the generation phase in the demonstrative application. The choice was made to be able to show a traditional piece that was as *normal* as possible for a typical listener. Therefore, a slightly modified version of the demonstrative application—with irregular time signatures and patterns—was alongside

presented to those participants to further evaluate the discussion. As a result, they mostly reacted positively to the new version, while some stating that they were quite fascinated about the dramatic change, hence the diversity, the engine is capable to offer.

6.3 General Discussion

First of all, the development stage of demonstrative applications was a crucial step in the project, as the whole process made it possible to spot the flaws and missing parts of the system clearly in terms of practical aspects before the evaluation phase. As a matter of fact, particularly during the development of the game prototype, numerous features were added to the programming interface such as user input driven generators to provide a broader interaction and flexibility to the prospective end-user. That being said, the current state of the engine is arguably fully-functional and ready-to-use in any third party interactive system easily.

While the demonstrations were made as a proof-of-concept showcase, they enabled to gather considerable amount of constructive feedback which were obtained in the evaluation phase to be consulted in future work. Especially, in terms of quality of sound of the outcome is considered to be improved in a later stage to provide more compelling results. Having said that, the negative arguments that were implicated during the questionnaire were mostly encountered by the features which were not related to the core functionality of the system. Thus, those specific points could be easily improved in the future to resolve the stated issues. In fact, that is the main idea of having an extensible and a modular architecture for the proposed approach in the first place.

All in all, the evaluation phase in general shows that the engine—both conceptually and practically—achieves a promising potential to be used in external applications in a desired manner. It is an encouraging result and an important step in the academia as well as for the interactive entertainment industry to proceed the idea further.

Chapter 7

Conclusion

An adaptive music engine, *barelyMusician*, has been proposed in this research paper to the field of computer science and music technology. The proposed engine was implemented providing full capabilities of the approach and demonstrated accordingly for the evaluation and justification of the research done. In brief, the resulting work shows promising potential as a comprehensive solution to the stated problem with regards to both theoretical and practical means.

7.1 Contributions

Main contributions to the field include a novel approach for combining real-time music generation and transformation methodologies together into a single *bundle* that offers a practically justifiable architecture to be used in any type of interactive entertainment systems in academia as well as in the industry. Moreover, the proposed music transformation method is somewhat unique in the field by allowing soundwaves level manipulation such as timbre modulation during the process on the fly.

The project as a whole furthermore introduces an important step for practical use of such methodologies. More precisely, the previous research done in the field was more or less limited in terms of the concrete works provided, resulting a gap in the field for the actual usage of the stated techniques and technologies in later applications. That

being said, this research project offers a powerful framework with a broad range of features which would lead the way of substantial applications to be developed in the area of research in the future.

7.2 Future Work

While the core engine is functionally complete, the resulting work should be considered only as the initial step. Having said, further development is crucial to elaborate and extend the research done in the future.

One major future work planned to be done is adding more presets—generators, instruments and audio effects—to the engine. Particularly, more sophisticated generators with various capabilities would dramatically enhance the ease of use, so that, the users would not necessarily have to rely on their own musical backgrounds and programming skills while using the engine. Additionally, further abstractions are planned to be included, such as *ensemble presets*, in order to introduce greater diversity. For instance, instead of managing the generators and instruments individually, the user will be able to select a pre-loaded ensemble, say *a Rock ensemble* or *a Jazz ensemble*, in that case to fulfill their needs accordingly.

7.2.1 Community Support

Having considered the capabilities mentioned above, there is an enchanting way to speed up this process by creating a community of users and developers of the engine. In other words, it is planned to announce a new community for the engine in which the users can upload and share their own presets, so that other users can reach even more materials to work with. The idea also opens a brand new viewpoint for further improvements for the engine in a later stage.

7.2.2 Licensing & Release

Finally, as expected, *barelyMusician* is scheduled to be released officially in the near future. Before the release, several optimizations and relevant documentation should be established for the engine. Additionally, while the intention is to keep the project and all the relevant materials open-source and free to use, due to the potential licensing issues caused by the pipeline architecture (see Section 3.1), certain considerations might be undertaken accordingly. Overall, it surely is an overwhelming yet fascinating opportunity to reach out a wider community of users with this approach.

Appendix A

Survey Questions

This informal survey was prepared to deliver an insight of what *barelyMusician* is capable of, offering features and expressions as a full experience in terms of practical means. Participants are expected to answer the questions provided below personally with respect to their honest impressions about the outcome they are exposed to. Resulting findings may only be used as anonymous feedback for the project in order to evaluate and enhance the functionality of the practical aspects of the stated approach.

The questions below are open to the participant in a way that it is possible to interpret them any way s/he desires.

1. How affective was the interaction with regards to your experience? Did you experience an immediate change in the mood of the playing song when you interact with it accordingly? If so, were those changes accurate in terms of the selection of that particular mood?

2. How did you find the songs you have listened to? Were the choice of instruments appropriate to you as the listener? Did you notice any particular piece felt unnatural during the playback?

3. What would be your reaction if you were to hear such compositions as the background music in a movie or a game? Would that please or annoy you? What level of distraction would it lead to in terms of the immersion?

4. How was the overall experience? Please feel free to include any additional thoughts or comments you would like to add below.

Appendix B

Disk Information

The attached disk contains the full source code of the project implementation including all the assets and binary files.

In addition, the demonstrative applications are provided as executables (in *.exe* format for Windows) along with an illustrative video footage per each.

Lastly, digital copies of the abstract, the poster, the final presentation and the report itself are attached in case of need.

Bibliography

[1] A. Robertson, "How 'the last of us' sound design outshines gameplay," July 2013 (accessed on 31 August, 2014). http://www.forbes.com/sites/andyrobertson/2013/07/04/the-last-of-us-in-depth-review-sound-design/.

[2] P. Vorderer and J. Bryant, *Playing Video Games: Motives, Responses, and Consequences*. LEA's communication series, Lawrence Erlbaum Associates, 2006.

[3] D. Valjalo, "Game music: The next generation," August 2013 (accessed 31 August, 2014). http://www.gamesindustry.biz/articles/2013-08-12-game-music-the-next-generation.

[4] Audiokinetic, "Wwise: Waveworks interactive sound engine," 2014 (accessed on 31 August, 2014). https://www.audiokinetic.com/products/208-wwise/.

[5] Firelight Technologies, "Fmod studio," 2014 (accessed on 31 August, 2014). http://www.fmod.org/fmod-studio/.

[6] K. Larson, J. Hedges, and C. Mayer, "An adaptive, generative music system for games," 2010 (accessed on 31 August, 2014). http://www.gdcvault.com/play/1012465/An-Adaptive-Generative-Music-System.

[7] M. Hoeberechts, R. J. Demopoulos, and M. Katchabaw, "A flexible music composition engine," *Audio Mostly*, 2007.

[8] W. A. Sethares, *Tuning, timbre, spectrum, scale*, vol. 2. Springer, 2005.

[9] F. Lerdahl, R. Jackendoff, and R. Jackendoff, *A Generative Theory of Tonal Music*. MIT Press series on cognitive theory and mental representation, MIT Press, 1983.

[10] D. Tymoczko, "What makes music sound good?*," 2010 (accessed 31 August, 2014).

[11] D. Tymoczko, *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. Oxford Studies in Music Theory, Oxford University Press, USA, 2011.

[12] J. A. Maurer IV, "A brief history of algorithmic composition," 1999 (accessed on 31 August, 2014). `https://ccrma.stanford.edu/~blackrse/algorithm.html`.

[13] M. Simoni and R. Dannenberg, *Algorithmic Composition: A Guide to Composing Music with Nyquist*. University of Michigan Press, 2013.

[14] C. Wolff, *Johann Sebastian Bach: The Learned Musician*. W.W. Norton & Company, 2000.

[15] G. Nierhaus, *Algorithmic Composition: Paradigms of Automated Music Generation*. Mathematics and Statistics, Springer, 2009.

[16] A. Harper, *Infinite Music: Imagining the Next Millennium of Human Music-Making*. John Hunt Publishing Limited, 2011.

[17] I. Xenakis, *Formalized Music: Thought and Mathematics in Composition*. Harmonologia series, Pendragon Press, 1992.

[18] M. Puckette, "Puredata - pd community site," 2014 (accessed on 31 August, 2014). `http://puredata.info/`.

[19] J. McCartney, "Supercollider," 2014 (accessed on 31 August, 2014). `http://supercollider.sourceforge.net/`.

[20] M. Z. Land and P. N. McConnell, "Method and apparatus for dynamically composing music and sound effects using a computer entertainment system," May 24 1994. US Patent 5,315,057.

[21] Bush, Gershin, Klein, Boyd, and Shah, "The importance of audio in gaming: Investing in next generation sound," 2007 (accessed on 31 August, 2014). `http://www.gdcvault.com/play/667/The-Importance-of-Audio-In`.

[22] A. Berndt, R. Dachselt, and R. Groh, "A survey of variation techniques for repetitive games music," in *Proceedings of the 7th Audio Mostly Conference: A Conference on Interaction with Sound*, pp. 61–67, ACM, 2012.

[23] Wolfram Research, "Wolframtones: An experiment in a new kind of music," 2002 (accessed on 31 August, 2014). `http://tones.wolfram.com/`.

[24] S. Wolfram, "Music, mathematica, and the computational universe," June 2011 (accessed on 31 August, 2014). `http://blog.stephenwolfram.com/2011/06/music-mathematica-and-the-computational-universe/`.

[25] M. Hoeberechts and J. Shantz, "Realtime emotional adaptation in automated composition," *Audio Mostly*, pp. 1–8, 2009.

[26] K. Demopolous, "Musido: A framework for musical data organization to support automatic music composition," 2007.

[27] P. Casella and A. Paiva, "Magenta: An architecture for real time automatic composition of background music," in *Intelligent Virtual Agents* (A. Antonio, R. Aylett, and D. Ballin, eds.), vol. 2190 of *Lecture Notes in Computer Science*, pp. 224–232, Springer Berlin Heidelberg, 2001.

[28] K. Jolly, "Usage of pure data in spore and darkspore," in *Pure Data Convention Weimar*, 2011.

[29] E. Berlekamp, J. Conway, and R. Guy, *Winning Ways for Your Mathematical Plays*. No. v. 2, Taylor & Francis, 2003.

[30] S. R. Livingstone and A. R. Brown, "Dynamic response: Real-time adaptation for music emotion," in *Second Australasian conference on Interactive entertainment*, (Sydney), pp. 105–111, Creativity & Cognition Studios Press, 2005.

[31] S. R. Livingstone, R. Muhlberger, A. R. Brown, and W. F. Thompson, "Changing musical emotion: A computational rule system for modifying score and performance," *Computer Music Journal*, vol. 34, no. 1, pp. 41–64, 2010.

[32] A. Friberg, "pdm: An expressive sequencer with real-time control of the kth music-performance rules," *Comput. Music J.*, vol. 30, pp. 37–48, Mar. 2006.

[33] M. Eladhari, R. Nieuwdorp, and M. Fridenfalk, "The soundtrack of your mind: Mind music - adaptive audio for game characters," in *Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, ACE '06, (New York, NY, USA), ACM, 2006.

[34] F. K. Timothey Adam, Michael Haungs, "Procedurally generated, adaptive music for rapid game development," in *FDG 2014 Workshop Proceedings*, Foundation of Digital Games, 2014.

[35] M. Hoeberechts, R. Demopoulos, and M. Katchabaw, "Flexible music composition engine," Nov. 15 2011. US Patent 8,058,544.

[36] S. Bokesoy, "Presenting cosmosf as a case study of audio application design in openframeworks," in *International Computer Music Conference Proceedings*, vol. 2012, 2012.

[37] M. Edwards, "Algorithmic composition: computational thinking in music," *Communications of the ACM*, vol. 54, no. 7, pp. 58–67, 2011.

[38] "Openal soft," 2014 (accessed on 31 August, 2014). `http://openal-soft.org/`.

[39] R. Bencina, "Portaudio - an open-source cross-platform audio api," 2014 (accessed on 31 August, 2014). `http://www.portaudio.com/docs.html`.

[40] M. Grierson, "Maximilian," 2011 (accessed on 31 August, 2014). `http://maximilian.strangeloop.co.uk/`.

[41] Z. Lieberman, T. Watson, and A. Castro, "openframeworks," 2014 (accessed on 31 August, 2014). `http://openframeworks.cc/documentation/`.

[42] Unity Technologies, "Unity - game engine, tools and multiplatform," 2014 (accessed on 31 August, 2014). `http://unity3d.com/unity`.

[43] A. Gungormusler, "Analysis of interactive sound synthesis and audio plugin development." Undergraduate Thesis, Department of Computer Engineering, Bogazici University, 2013.

[44] R. Bencina, "Real-time audio programming 101: time waits for nothing," 2011 (accessed on 31 August, 2014). http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing.

[45] A. Veltsistas, "C# synth project," 2014 (accessed on 31 August, 2014). http://csharpsynthproject.codeplex.com/.

[46] R. Boulanger and V. Lazzarini, *The Audio Programming Book*. MIT Press, 2011.

[47] B. H. Suits, "Formula for frequency table," 1998 (accessed on 31 August, 2014). http://www.phy.mtu.edu/~suits/NoteFreqCalcs.html.

[48] A. P. Oliveira and A. Cardoso, "Towards affective-psychophysiological foundations for music production," in *Affective Computing and Intelligent Interaction*, pp. 511–522, Springer, 2007.

[49] Unity Technologies, "Unity - manual: Custom editors, unity documentation," 2014 (accessed on 31 August, 2014). http://docs.unity3d.com/Manual/editor-CustomEditors.html.

[50] P. S. Langston, "Six techniques for algorithmic music composition," in *Proceedings of the ICMC*, (Ohio), pp. 164–167, The Ohio State University, Columbus, Ohio, 1989.

[51] D. Carlton, "I analyzed the chords of 1300 popular songs for patterns. this is what i found.," June 6 2012 (accessed on 31 August, 2014).

[52] O. Bown, "Experiments in modular design for the creative composition of live algorithms," *Computer Music Journal*, vol. 35, pp. 73–85, 2011.

[53] D. Temperley, "Scalar shift in popular music," *Society for Music Theory*, vol. 17, no. 4, 2011.

[54] C. Comair, R. Johnston, L. Schwedler, and J. Phillipsen, "Method and apparatus for interactive real time music composition," Nov. 23 2004. US Patent 6,822,153.