Adaptive music in video games

G.J. Stolk

September 5, 2008
Summary

Adaptive music is more and more becoming a requirement in the modern video game industry. In most available audio solutions for games adaptive music does not fulfill an prominent role, which results in poorly featured solutions. Another issue with these solutions is that the impact of implementing them in a game engine is huge because not only the music part is taken over by it but also all other audio related tasks.

This paper presents a proposal of a system which is purely dedicated to adaptive music. It is focused to have an easy flexible implementation in a game engine with minimal impact. The system supports both state internal dynamic music as well as an advanced flexible transition system between states. For the audio designer a tool is presented which makes it easy to design adaptive music scores using this system.
Abstract

Adaptive music is defined by music which is adapting to situations it is applied to. A good example of this is the current variety of movie productions, music in movies is carefully mixed to fit the events depicted in the visual media. An area where it seems to be used less is video games, while adaptive music would suit it maybe even better than movies. The reason why this is true is the array of challenges adaptive music produces for video games.

Video games represent a user controlled and therefore a dynamic and unpredictable form of media. The unpredictable nature of games prevents it from upfront music mixing like is practiced for movies. The conflict between statically defined music and dynamic unpredictable user behavior proves to be the greatest challenge in realizing adaptive music in video games.

Coded Illusions is a video game studio willing to take on this challenge, which this paper is a result of. This paper is separated in 3 parts.

Part I contains the research done on the subject of adaptive music in games. It first creates a background about video games for those who are not already familiar with the concepts of video games. Later in Part I a closer look will be given into what adaptive music is and how it could conceptually be molded into a software solution. This is done by researching current applications of adaptive music in games, current other adaptive music solutions and more traditional music creation for games.

Part II is describing a for this project custom designed and implemented adaptive music solution. This part is limited to concepts and does not go into great technical detail. An overview of all components of the solution will be given and each individual component will be explained in more detail in the chapters that follow.

Part III contains the appendixes. The first chapter forms a reference to some of the components of the diagrams used in this paper. Later chapters are dedicated to a more in-depth technical description of the adaptive music solution which is especially useful for the implementation of the solution in a product. There is also a glossary available after this which can function as reference when certain terms need more clarification.
Contents

Summary i
Abstract ii

I Background & Research 1

1 Introduction 2

2 Video games 3
  2.1 Abstract ............................................. 3
  2.2 Point of view ...................................... 3
  2.3 Genres ............................................... 4
  2.4 Platforms .......................................... 4
  2.5 Game Development ................................. 5
  2.6 Coded Illusions .................................... 5

3 Adaptive music 6
  3.1 Abstract ............................................. 6
  3.2 Adaptive music vs. Interactive music ............ 6
  3.3 Significance of adaptive music .................... 6
  3.4 Definition .......................................... 7
  3.5 Case Studies ......................................... 8
    3.5.1 Russian Squares[3] ............................. 8
    3.5.2 No One Lives Forever[4] ..................... 9
    3.5.3 Nomos ........................................... 9

4 Digital music creation 11
  4.1 Digital music creation for games .................. 11
  4.2 Adaptive music design methods .................... 12

5 Current technology 14
  5.1 Purpose of adaptive music solutions ............... 14
  5.2 ISACT .............................................. 14
  5.3 Wwise ............................................... 16
  5.4 Kismet .............................................. 17
Part I

Background & Research
Chapter 1

Introduction

From the game industry there is a lot of interest for adaptive music solutions. Coded Illusions is a video game studio using the Unreal Engine 3 for their products. The Unreal Engine has a very complete list of features but lacks adaptive music support. The initial assignment was therefore to create an adaptive music tool for the Unreal Engine, which is a rather open ended assignment. In the creation process of this tool, called Accido, the focus of the assignment shifted more to the underlying adaptive music structures than the tool itself. It turns out to be a great challenge to come up with a good structured representation of adaptive music.

This part will start off by exploring the world of video games to provide a background for the rest of this paper. The rest of this part is dedicated to find the answers to the following questions: What is adaptive music and why is it needed? How is adaptive music currently applied in products and authoring solutions? How can adaptive music be structured? What challenges does the creation of a custom adaptive music solution pose? The answers to these questions will be found by looking at adaptive music from three different angles. In chapter 3 a theoretical definition is given about what adaptive music is or should be. This will be made more concrete by looking at three practical applications of adaptive music in video games. In chapter 4 a description will be given on how digital linear music is created and in what ways this process can be bent to make it suitable for adaptive music creation. Finally in chapter 5 existing adaptive music solutions are discussed and how these solutions handle adaptive music.
Chapter 2

Video games

This chapter is meant as a quick introduction to some core concepts of video games. The main video game genres will be discussed and what the position of music is in these genres. An overview on how a video game is produced will be given. Each section of this chapter can be skipped if the subject at hand is already familiar.

2.1 Abstract

A video game is a computer program which provides visible and/or audible feedback to the user's input with the purpose of entertainment. Typically this comes down to a user or player controlling one or multiple actors in the game world. These actors can for example be a virtual human character, an airplane or even a simple block. The virtual game world can be as complex as a realistically modeled world with seemingly living people in it or as simple as a blank square.

2.2 Point of view

This world and its actors can be visualized from different point of views. Probably one of the most common is the first person view. The idea here is about the same as first person in literature where the reader sees the world through the eyes of the main actor of the game.

Another very popular point of view is the third person view. In this case the camera is placed a small distance from the playable actor while looking at said actor. The determination to choose for a third person point of view can be motivated with a few arguments. First a third person view allows the user to see the actor itself which enables greater control over the movements of the actor because of the better visual feedback. Secondly the user has a better overview of the world because the camera is typically situated higher. The disadvantage is that the user can feel less connected to the main character than if a first person perspective was used.

The top down view is especially useful when the user is able to directly control multiple actors. The camera is in this case situated high above the world and looking downward. This allows for a great overview of the world and actors.
The views described above are mostly used in games where a 3D representation of a world is projected and rasterized on a 2D surface which is the computer or tv screen in this case. In the past the hardware was not powerful enough to render these 3D games fast enough. As alternative 2D graphics where used. Still 2D graphics is not a thing from the past because 2D graphics are a lot less complex to work with than 3D graphics. 2D graphics are currently mainly used for puzzle games like the famous Tetris game. 2D games are usually either viewed straight from above or straight from the side.

2.3 Genres

Video games can be distinguished from one another by genres. The distinction between video game genres is like music genres subjective and vague, so the following definitions are common uses of genres more than fixed rules. The four main video game genres will be discussed, which are: action, adventure, strategy and puzzle. Almost all games can be placed under one of these even though more precise genre definitions are normally used.

Currently the most common group of next-gen games are action games, they typically have most of the following properties. The game is focused on combat. The game sports either a first or third person view. The player has direct control over only one character at a time.

The adventure game is about the same as the action game with the exception that it is usually third person only and not primarily focused on combat but more on a storyline and world navigation.

In strategy games the user is usually in control of multiple actors. In Action and Adventure games the focus lies on the users human-computer interaction skills. With strategy games it is the strategic decisions of the player that makes all the difference. Strategy games have a top down view to enable the player to have a good strategic overview on the virtual world.

The puzzle genre can be very diverse. Usually it consist out of very simple actors like blocks spheres etc. and challenges the user to reach a greater goal by doing very similar simple action multiple times.

2.4 Platforms

At the moment there are a number of different platforms to release a game on. One of the main platforms used to be the PC. Even tough the personal computer was never really meant for gaming it is doing a pretty good job lately with increasingly performing graphics hardware. The current trend is however more and more moving away from the PC platform by some claimed due to piracy issues. Other platforms are Microsoft’s Xbox 360 and Sony’s Playstation 3. These platforms measure up pretty well with high end PC’s, mainly because the game can be crafted to perfectly fit the platforms hardware. The last meaningful platform is the Nintendo Wii which is not as powerful as the aforementioned two platforms and focuses on smaller more innovative games and a broader market.
2.5 Game Development

A game development team consist out of a very diverse group of people. There are game designers who design the rules, the story, the world etc. of the game. Most video games are graphically rich which means there are also a number of artist whom are doing the 3d modeling and the 2d artwork. Video games have had sound from the start so there is also one or more audio designer(s) in a game development team. The sound designer is responsible for all sound effects, ambient sounds, voice acting and the music, which is especially interesting for this paper. Then there is the programmer team whom put all the ideas and content together in the final game. And finally the production team which manages the game development team.

2.6 Coded Illusions

Coded Illusions is working and will with considerable certainty work in the future on third person action adventure games for multiple platforms (PC, Xbox 360 and Playstation 3). To accomplish this Coded Illusions is working with the Unreal Engine 3.

The Unreal Engine is a middleware solution to facilitate game development with a complete package to create games [5]. It can be used to create all kinds of graphical and audible assets in a game, and has an extensive toolset to help create said assets. It has the ability to "cook" the games created with this middleware to the Xb ox 360, Playstation 3 and PC platform.

The adaptive music solution this paper proposes will therefore mainly focus on the action adventure game for the Unreal Engine 3 middleware, but is aimed to be more flexible in the case Coded Illusions will diverge to other types of projects.

A third person action adventure game is spread out over multiple levels where each level has some main goals and, more importantly for the adaptive music system, a different part of the game world. This usually goes in pare with an unique music score for each level which fits the mood of the level.

In a level the game state is constantly changing from the more relaxing puzzle parts to intense combat areas. This is the main target for the adaptive music system. Each state needs a unique fitting musical sequence to fit the players emotion.
Chapter 3

Adaptive music

This chapter introduces adaptive music. A definition for adaptive music will be given and reinforced by three case studies. This chapter answers the questions as to why adaptive music is needed in a game, and how an adaptive music system conceptually works by definition and in examples.

3.1 Abstract

Music always had a certain emotional value. This can be seen particularly well with movie music where the music follows the events in the story. In movies the music adapts to what happens on the screen, so in a sense movies feature adaptive music. The advantage of film is that it is a linear media form. The music can be crafted completely upfront in a way it fits the movie perfectly. In the case of video games however adaptive music is much more complicated because of the interactive and because of that unpredictable nature of the medium. This is where the challenge lies and this forms the reason for the existence of this project.

3.2 Adaptive music vs. Interactive music

Adaptive music is often referred to as interactive music. The term interactive music intuitively explains better what this paper is about but strictly it is not correct.

Interactive music implies a user has interaction with the music itself, which is false in most cases. Therefore adaptive music is a more accurate term.

Just a few games feature true interactive music. A popular example is guitar hero where the players input directly affects the music. But that is not what this paper is about, in this case we want music that adapts from the player’s perspective automatically to the emotional state that the game is in.

3.3 Significance of adaptive music

Adaptive music functions as quality enhancing factor in a game. It is hard to define what makes a game good or bad. When an attempt is made to define the
Quality of a game a lot of abstract terms are used. One of these terms sticks out.

The rate on how much a game is appreciated is largely depending on the level of immersion of a game. Immersion could be described as the amount the player is emotionally involved in a game. The deepest state of immersion would be where a player forgets all about the 'real world' and all his attention is focused on the game world. There are many ways to achieve a deeper level of immersion which usually compliment each other. Examples of factors which are adding to the immersion of a game are realistic, or at least convincing, in-game events, worlds and characters.

Music can have a great emotional impact on people, and has therefore an important role in the 'immersiveness' of a game. Music that does not emphasize the emotion of the game at that time does not help with the player's immersion in the game. Music that is repeated too much will annoy the player eventually. Transitions between different pieces of music should ideally not be noticed by the player. If a player does notice it, his attention will shift too much to the music, and away from the game.

While this argument is vague and hardly measurable, it does show that adaptive music needs to be done right to be effective. There is however a more convincing argument that justifies adaptive music in games.

Where in the past games where played by a small amount of people with relatively high performance personal computers and specialized consoles, now the gaming market becomes more and more mainstream with the recent next-gen consoles. The game titles are getting closer to large movie productions and are even surpassing them in budget and revenues. These bigger titles are often referred to as AAA titles. These are productions where the game has to excel in every area, so it has to have top-notch story, narrative, graphics and gameplay but also sound and music. To get closer to the desired cinematic experience adaptive music is more and more becoming a must in video games.

3.4 Definition

To narrow the term adaptive music down we need to define it further. An article by Andrew Clark[2] provides a thorough definition:

Adaptive music is music in which a primary concern in its construction is a system for generating significantly different performance versions of a piece in response to a specified range of input parameters, where the exact timing and/or sequence and/or quantity and/or presence and/or values of input parameters are not predetermined, and where the desired output of the system is coherent and aesthetically satisfying within the musical tradition(s) selected by the composer.

The definition is about as thorough as it is dense, so it is good to lift out some more sizable concepts from it. These concepts are now still somewhat abstract but will become more concrete with the case studies in the next section.

- Significantly different performance versions: Adaptive music has by definition a different performance every time it is played. This will also be
illustrated in the upcoming case studies. In these cases the music will exclusively change performance based on the actions of the player.

- **Specified input parameters**: Two things are important here. Having input parameters indicates that the music is actually driven by external input. This is what makes the music adapt to the situation at hand. 'Specified' indicates that all ways of interaction are predetermined and therefore can be designed by an audio designer.

- **Significant degree of indeterminacy**: The word *indeterminacy* here is about how the specified inputs are used. These inputs could be used at any time, order, quantity and value. Not all these are necessarily required but are likely situations depending on the unpredictable nature of game players and audio designer's work methods.

- **Traditional musical coherency and organization**: While this input is so random in many ways the output must still be aesthetically satisfying. Ideally adaptive music should sound as good as it were linear rendered music. This can be accomplished by the design options of the specified input parameters.

### 3.5 Case Studies

To bring out the definition in perspective it is good to examine some cases where adaptive music is applied in existing products. There is not much information available about existing applications of adaptive music. Two case studies published by the Interactive Audio Special Interest Group stand out.

#### 3.5.1 Russian Squares[3]

*Russian squares* is a puzzle game where the player has to combine matching squares to form rows of the same squares thereby eliminating those rows. The game has been released in “Microsoft Plus for Windows XP”.

The game has a total of three visual themes with corresponding pieces of adaptive music. Each time the player clears a row of squares the total number of squares gets lower. The ultimate goal for each level is to eliminate all squares. This system is also represented in the music. The music is adaptively becoming more intense towards the end of a level. This is done by the use of multiple layers of music. As the game progresses layers are added and replaced by more intense ones in a synchronized matter thereby maintaining the musical coherency.

Sometimes also a sample called a *stinger* is played. For example when a line is cleared a distinct sound is integrated in the music which becomes more like a part of the music than a separate sound effect. An example on how this is done is the in-game timer. This timer starts ticking when there is almost no time left to complete a level. This ticking is synchronized with the rhythm of the music.

There are a few interesting features in this case. Repetitiveness in music can be combated by adding and removing layers of music. Gameplay elements can be integrated in the music. All this needs tight synchronization.
3.5.2 No One Lives Forever[4]

No One Lives Forever (NOLF) is a first person action adventure game developed by Monolith Productions in 2000. It is a James Bond like spy story set in the 1960s. The game has states where the player just has to walk around, and other states with intense combat.

Obviously the same musical score would not fit both states. This is why the composer defined six different states subjectively called "Silence", "Super ambient", "Ambient", "Suspense/sneak", "Action 1" and "Action 2", each with their own musical score. Any of these musical states can be called at any time in the game. The challenge here is to transition between states as soon as possible while maintaining musical coherency.

To accomplish this a transition matrix was used resulting in a need of 30 transitions. Two ways of transitions are used in this game namely using a piece of music as a bridge between states, or by just stopping one state and starting another. These transitions can take place on boundary types like "immediate", "grid", "beat", "measure" and "segment". How each transition functions is defined by a script.

Gameplay elements are also highlighted in the music by playing stinger samples. These samples would be played on the previously mentioned boundary types.

We will see later when looking at existing adaptive music solutions that these solutions are especially good for creating musical structures like described in this case.

3.5.3 Nomos

Coded Illusions is currently working on their first next-gen title called Nomos. The result of this document is not aimed for this game yet, but is planned for when the second title goes into production. Nomos already has some form of an adaptive music implementation. Why this project is still needed is explained later in chapter 5.
The current adaptive music implementation of Nomos works a lot like the adaptive music of “No One Lives Forever” in concept. The problem with Nomos is that it has multiple varying levels. Each level has its own visual theme and objectives. There are some combat, puzzle and adventure levels. Therefore every level has its own unique adaptive musical score. In combat levels states like ‘combat’ and ‘stealth’ are used. Adventure levels often have different music for every room or area. This means that not all states need to be able to transit to each other, because you can not always topologically go from one room to another.

Nomos has some extra features for its adaptive music implementation. It can also play pieces of states internally in a random order. This adds the advantage that not only the whole score but also individual states have a dynamic composition. The downside is that this adds an extra layer of complexity for state transitions.
Chapter 4

Digital music creation

Creating an adaptive music system means that we need to rip open and reorder the currently used music design. Therefore it is necessary to first analyze how music is made for games right now. This chapter will introduce traditional digital music creation for games and what is applicable to adaptive music. Two adaptive music design methods will be presented based on that. This leads to the first statement about the role of an adaptive music system in the production of adaptive music.

4.1 Digital music creation for games

The first thing to discuss when looking at digital music creation is how the fundamental sounds are generated. There are actually two major ways to do this: one is sound synthesis, the other is sampling. When sound synthesis is used the computer generates and calculates a sound wave and sends it to the audio hardware to be put out. Sampling actually saves these generated or recorded sound waves and sends that to the audio hardware. Sound synthesis can be very demanding on the cpu, while sampling requires more memory. Because the cpu time is scarce for video games and because there is in most cases an abundance of storage space available sampling is generally the way to go for video games.

Now this does not necessarily mean we cannot use synthesis, it actually is even commonly used to modulate samples. It just implies that all heavier forms of synthesis needs to be rendered to a sample to prevent the performance problem before using it in the actual video game.

It should however be noted that apart from memory constraints the use of samples have another limitation, and that is limited polyphony. Audio hardware can only play so much individual samples, or in this case referred to as voices, simultaneously. In video games music is not the only audio that is played, there are also sound effects, voice acting and ambient sounds to name a few. So the amount of available voices need to be shared between those. On platforms like the Xbox 360 this is not that much of a problem because all Xboxes have the same amount of voices which is 256[6], this is more than enough in most cases. The amount of voices on the PC platform however varies from 32 to 1024 voices depending on the audio hardware. This means that the amount of simultaneously played samples needs to be regulated.
4.2 Adaptive music design methods

In the end the audio artist will have a few layers of samples. Typically in the creative process of making the music score each of these layers represent one instrument. These samples can be arranged horizontally, which represents time and melody, and vertically, which represents the different samples that should play simultaneously, which is often referred to as harmony or orchestration. This arranging can be done in specialized auditory software. To think about music as an horizontal (melody) and vertical (harmony) structure is important because it allows for the segmentation of the music in these areas.

The vertical segmentation is pretty straightforward. These segments consist of one or multiple instruments mixed into one sample. The horizontal or time segmentation requires some extra explanation. Normally time is measured in minutes and seconds. For music this is typically not the case because time can be variable for music. Musical time is measured in measures (also known as bars) and beats. One measure for each individual piece of music consists of a predefined number of beats. These musical units can be converted to time by the use of a BPM or Beats Per Minute value. The BPM value can vary depending on the desired tempo the music should be played. Melody segments are usually made out of whole measures. This causes the horizontal segmentation to be measured mostly in measures and sometimes in some extra beats.

![Figure 4.1: A screenshot of an arranger from ableton live. Note the horizontal and vertical arrangement.](image)

In the previously mentioned "No One Lives Forever" game adaptive music was realized by mostly interchanged complete different pieces of music thereby varying the sequence of the music over the course of the game. In the rest of this paper this form of adaptive music is referred to as horizontal sequencing.

The previous mentioned "Russian Squares" case study clearly shows vertical
segmentation. Adaptive music of this game was achieved by adding and removing layers of music and thereby adjusting the orchestration of the musical piece. In the rest of this paper this form of adaptive music is referred to as *Vertical orchestration*.

**Figure 4.2:** An example of horizontal sequencing.

**Figure 4.3:** An example of vertical orchestration. In this case layer 1 is removed where layer 3 is added, where these layers could represent different (groups of) instruments.

The vertical orchestration method can be hard on the number of voices used by the music. This problem is not so much present when a horizontal design is used, because all instrument layers can be rendered to one sample in that case. This does however limit the variety of performance versions. Usually there is a gray area where both vertical and horizontal designs are represented in a hybrid form.

The ability to segment a musical composition according to these two design methods allows for dynamically reconstructing the music from these segments. When the audio designer keeps this principle in mind when composing his music the adaptiveness of the music in the game can be greatly increased. What this leaves as the main task of an adaptive music system for video games is the sequencing and orchestration of the various music segments based on a set of customizable rules defined by the audio designer.
Chapter 5

Current technology

Surely this adaptive music project is not unique. A few other game developers have made their own internal adaptive music solutions. In general there is not much publically known about the inner workings of these systems, the best that can be done here is analyzing the products made with these tools like is done in the case studies in section 3.5. There are however a few licensable tools available that feature some kind of interactive/adaptive music systems. In this chapter the ISA CT, Wise and Coded Illusions' (current) adaptive music system will be analyzed and a motivation will be given why these solutions do not fit the needs of Coded Illusions game projects. Sometimes a reference will be made to Accido, the custom made authoring tool for this project.

5.1 Purpose of adaptive music solutions

In its most basic form adaptive music can be described from a technical point of view as procedural programmed rules combined with smart synchronization. This is why traditional adaptive music productions where mostly composed by an audio artist and the rules and logic where custom coded by a programmer. While the games get bigger and require higher quality custom adaptive music programming becomes an issue. There are some obvious disadvantages of hard coding the adaptive music score. First of all it requires a lot of time for both the audio designer and programmer, resulting mostly in a less than perfect result because the audio artist knows best how the music should sound. Revisions of the musical design also become hard to accomplish, so it has to be done right the first time. This is not how audio artists usually work. In the end the goal is to get rid of programming whole adaptive music structures and standardize them instead. This is what the following solutions try to achieve.

5.2 ISA CT

One of the solutions investigated before this project was started is Creative's ISACT (Interactive Spatial Audio Composition Tool). This is not just an adaptive music solution, it basically takes over all audio related tasks from the engine and replaces it with its own sound engine. After Creative made an Unreal Engine 3 implementation for their sound engine Coded Illusions started working with
ISA CT. Soon a few problems arose, the main issue was that it did not integrate well with certain key engine features (like FaceFX and Matinee, both animation related modules). While this was a show stopper in itself there was also the issues with lacking Playstation 3 support, almost no support from Creative and no available source code.

All this led Coded Illusions to let go of ISA CT and start the project this paper is about. Still it is interesting to see what structures and methodologies ISA CT uses for its adaptive music solution.

ISA CT is in concept and structure for basic musical design quite simple but gets a lot more complex when non-standard designs are required. The complete score of a level is built with an arranger as seen in a lot of music auditory software like shown in Figure 4.1 from the last chapter. The idea is to capture all different states of the game in this one score. Later this complete score can be segmented and transitions between these segments can be defined in a simplistic way. Everything outside of this scope needs to be done through scripting.

In this case the simplicity limits flexibility. An example illustrating this is the fact that a segment can only transit to another segment at the end of the first segment or at a regular interval (measure or beat). So any form of customization is missing here.

Another part where ISA CT lacks is that it does not help with variation in states themselves. So what is left is a static segmented track with intelligent playback possibilities.

![Figure 5.1: Creative's ISA CT.](image)
5.3 WWise

AudioKinetic’s Wwise (WaveWorks Interactive Sound Engine) is as they like to call it: an integrated audio pipeline solution. It is from an implementation point of view like in some areas similar as Creative ISACT. It has an authoring tool and a sound engine that is compatible with Unreal Engine 3. The system also takes over all the audio functionality of the engine. It however has two advantages over ISACT: It does support multiple platforms (PC, Xbox 360 and Playstation 3) and the sound engine source code is available for Wwise licensees, the authoring tool source code however is not.

Overall Wwise is an enticing solution to some audio challenges. It has a lot functionality and the authoring tool looks well featured. The interactive music system seems to be very capable and flexible at first glance.

Figure 5.2: Audiokinetic’s Wwise.

It was not until half way through this project that this solution came under my attention. Up until this point the Kismet solution was used for adaptive music at Coded Illusions (more on that in the next section). At this point a certain adaptive music structure was already in use that could not be replicated in Wwise. Wwise has like ISACT the issue that state transition cannot be defined on custom timings. It only features fixed interval (beats, measures etc.) transition possibilities. This can lead to very unsatisfying transition results as even the demo tutorial[1] of Wwise demonstrates. Another disadvantage is the fact that the product relatively pricey.

Because the Wwise product was researched after the first drafts of the Accido
project where made it was good to compare my design with Wwise’s. On the abstract level both the accido concept and the Wwise solution for adaptive music are identical. The differences can mainly be found in the details.

Wwise works with states or ‘playlist containers’. Each playlist container can for example represent the state a game or game character is in, as an example the Wwise tutorials use the states ‘stealth’, ‘stress’ and ‘fight’. These playlist containers contain like the name suggest multiple samples. These samples can be played in a controlled random order.

Another interesting feature which was later also implemented in Accido is a real-time modifiable property. This property could for example be the playback volume indicating stress. The adaptive music system can than be fed from the game with a linear stress level value thereby adjusting the volume all music or certain layers of music.

Even though Wwise is a very decent solution it was in the end not a real option because of two reasons. Certain specific music structures cannot be made with Wwise but was nevertheless needed. And the interactive music module of Wwise was released when the accido project was already done half-way.

5.4 Kismet

![Figure 5.3: A screenshot of the kismet editor from the Unreal Engine website.](image)

Because the Unreal Engine does not come with any adaptive music solution whatsoever and ISACT gave more problems than solutions a quick fix for
adaptive music was needed. The most logic course of action was to create some kismet nodes to aid with music synchronization.

Kismet is the answer of Unreal Engine 3 to complex scripting languages. All game level scripting is done with kismet, a visual scripting system, so level designers do not need to learn an actual textual scripting language.

The kismet visual scripting scripting looks a lot like a flow chart comprised of event, condition, action and variable nodes. These nodes contain small portions of logic and can easily be created by a programmer. This leads to some more high level nodes that could for example place an item in a players virtual inventory, or as low level as an if statement.

So for example a level designer could place a trigger object in the world and attaches a kismet event ‘triggered’ to it. Now this event nodes output will be activated whenever this trigger is triggered by the player or otherwise. This event node output could than be connected to an action node input. This action node could have the sole task of opening a door. The trigger event output could be connected to a delay action node as well. A delay action postpones its output a predefined time. The output of the delay action could be connected to an action that closes the door again. Meanwhile an integer variable node could be set which counts the number of times the door opened.

In its very basics an adaptive music solution is all about starting audio samples in a well timed and synchronized matter. The determination about when what sample should be played is a logical system and therefore programmable. This means that a complete adaptive music system could theoretically be made with a number of nodes which can perform logical computations on variables, which are present in kismet, and proper synchronization methods, which are not present by default in kismet.

The delay node mentioned in the previous example could potentially function for synchronization, but it is not very precise and rather unwieldy in use because the delay time is measured in seconds and musical time is measured in beats and measures. The precision problem is especially noticeable when a few delays are chained in a row, the actual time passed from the begin until the end of this row is usually quite a bit higher than the sum of all the timings of the delay nodes. For most level scripting this is not much of a problem because for this task the delays are rarely chained and the timing is usually allowed to be a little off. For music however this chaining is very common for sample sequencing. The delay node does therefore not provide enough quality and ease of use for an adaptive music solution.

To create a basic solution for adaptive music two custom kismet nodes where created. The first node is called Set Beat. When this node is activated it sets three values, which are the current beats per minute, the beats per measure and the current time. These values are later used for synchronization where the difference between the later current time and the set current time is used to calculate the current measure and beat. This effectively solves the precision problem when chaining the second node created for the adaptive music solution.

The second node is the Beat Sync node. In this node a measure and beat can be given where the output of this node is synchronized to. So when said measure plus beats are reached the output is activated usually resulting in the playback of a sample.

While kismet with these additions can potentially be used for a full featured adaptive music system it is far from ideal. Adaptive music needs some complex
logical sequences. It is possible to create these sequences in kismet, but it is not particularly easy. Complex logic actually gets more complex when it is visualized in a flow chart structure, and is actually easier to do with a textual scripting language. Furthermore, kismet does not have any debugging tools, when logical errors occur they are hard to find and solve. These and more problems caused the regular attention of a programmer when these structures where created. In short the kismet solution does enable the audio designer to create an adaptive score, but is limited by the level of complexity.

Figure 5.5: Clutter caused by implementing adaptive music in kismet. This is just one state.
Chapter 6

Conclusion

This concludes the research done on the subject of adaptive music for video games. Because of the open-ended character of this assignment this research was not only needed to define how and if an adaptive music solution should be made, but also what features an adaptive music solution should and could have. There are however a few requirements for the adaptive solution, which can be mentioned now there is a background established about gaming and adaptive music.

6.1 Requirements

The solution should be focused in the first place on third person action adventure games. This means in practice that the game has a level state structure which also should be represented in the solution. Still it is an advantage if the system would be more flexible so it can be used still if Coded Illusions would choose to create a completely different title in the future.

The solution should primarily be focused on the Unreal Engine 3. This includes the feature of the Unreal Engine to compile games for different platforms. Again this is not set in stone, there are no guarantees Coded Illusions will still use the Unreal Engine 3 in a few years from now, so a more flexible implementation could be beneficial here as well.

6.2 Goals

The goals or tasks of an adaptive music system are threefold. First of all it should handle different game states with different possible musical themes and transitions between them. It should have a system which makes these musical themes automatic dynamic to prevent repetitive music. And finally it should offer the opportunity to integrate game events with audible feedback in the music.
6.3 Challenges

There are also a number of challenges to overcome. These challenges consist of technically hard to design or implement functionality and features that are not present in existing adaptive music solutions but should.

One of the things that is done in the current adaptive music solutions is the use of a complete custom sound engine. While this has to be done for solutions that also feature all kinds of sound effect functionality it is not necessary for a pure adaptive music solution. Keeping the sound engine of the game engine intact would be an advantage.

There should be some polyphony control. The exceeding of the maximal amount of voices should not be prevented but the amount of required voices on a specific musical score should be predictable and measurable.

Because the focus is mainly video games we are depending on a game loop for timing. The timing of a game loop has proven that it could be just precise enough to keep the music coherent but it must be done right.

A major flaw in most existing adaptive music solutions is that a state transition can only take place on a fixed interval. While this keeps the authoring tool interface and adaptive music structure simpler, this is not always good enough, a solution to this problem should be found.

And finally the solution to all these challenges and requirements should not produce a too complex system to use. In the end we are trying to not only reach for a flexible solution but also a solution that is easy in use.
Part II

Design & Implementation
Chapter 7

Introduction

This part presents the proposed adaptive music solution. This solution was designed with all the problems, solutions and requirements mentioned throughout Part I in mind.

It starts off with a presentation of the global architecture of how the adaptive music solution should be included in a game or authoring tool. This leads to the definition of three separate components which will be discussed separately in the next three chapters.

The first component is a data model to provide for a structured storage of an adaptive music score. This chapter does not provide a technical view on how the data is structured but more on how the adaptive music system works in a logical sense.

The second component is the playback library which is the component that needs to be implemented in an application if it requires the services of the adaptive music solution. This chapter focuses mostly on how the library should be implemented and used in an application, but it will also show in a lesser extend how the library works internally.

The final chapter will discuss the authoring tool. The interface will be presented thereby providing a more visual representation of the adaptive music data model. This opportunity will be used to explain how an adaptive music score can be designed in the data model’s structure. Descriptions with a more technical approach to the data model and library are available in the appendixes.
Chapter 8

Architecture

One of the bigger problems with the existing adaptive music solutions as described in the previous chapter is the fact that all solutions use a sound engine that completely replaces the Unreal Engine’s sound capabilities. This chapter will describe a solution to this problem and how this fits in the game and tool architecture.

8.1 Existing solution architectures

Both the previously mentioned Wwise and ISACT replace the complete sound engine of the Unreal Engine. Figure 8.1 shows how these systems work. The authoring tool creates a proprietary formatted file based on the rules defined in the tool. This file is passed on in a software library which can be included in the game but is probably also used in the authoring tool. When this is done the game or tool can start activating events through an interface on the library thereby playing samples defined by the rules set in the adaptive music setup.

The weakness here is the dependency on some black box audio engine library. This means that the audio designer can benefit from all the perks of such a system, but will also have to deal with all the limitations. Another issue is that
it takes over all sound objects resulting in sometimes incomplete meta data of the sample containers which can cause problems with modules that need this data (an example is the in 5.2 mentioned FaceFX and Matinee trouble with ISACT).

To solve this problem a library could be made that only handles the adaptive music logic and leaves the actual playback to the engine's sound system.

This could lead to an architecture shown in Figure 8.2. The library still has some kind of file input. The game can still fire events on a library. The difference is that the library does not actually play the audio samples. The library just informs the game or tool through an interface what samples should be played and what effects must be applied to them.

![Figure 8.2: Proposed architecture; The sample playback is delegated to the game engine.](image)

This system has much less impact when integrated in a game engine because it adds a module instead of replacing one. It automatically has the same limitations and advantages that the game engine has so it integrates better with all other sound functionality of the game. Because adaptive music is the main focus here we can give it much more attention, which potentially could lead to better results than the existing tools mentioned before.

### 8.2 Components

Based on this information the adaptive music project can be divided in three components. A data model describing what is contained in the input file for the playback library, which is the second component. And finally an authoring tool to create these files.

#### 8.2.1 Data model

A file specification for adaptive music should be made. This specification should describe a data model which is a structured representation of an adaptive music score.

It should be focused on flexibility more than anything. It should be able to describe any possible situation in line of the requirements and goals discussed.
in chapter 6. The details of this model will be laid out in chapter 9.

8.2.2 Playback library

There should be a software library with the sole purpose to handle all logic of the adaptive music type. The library should contain an update tick function which is called from the game engine. Predefined game events can be put through to the library which in turn orders the engine to load samples and play them. The playback library will be discussed further in chapter 10.

8.2.3 Authoring tool

There should be a tool with the creation and testing of the adaptive music data model files as purpose. Because of its flexible nature, more standard musical structures and effects are not described in the adaptive music data model. The tool should therefore function as translator between an audio designer and the adaptive music data model. This can be done by creating functions that can do common but complex or work intensive tasks with only the necessary user interaction. The focus here should be usability.

The tool should integrate the same library as will be used in the game engine. It will need a modified interface because it needs more information about the internal workings of the adaptive music system. The tool needs to be able to play back audio samples. A disadvantage here is that because there are separate sample playback systems between the tool and game the music might sound different. The tool could also be integrated in the game engine, where the engine sound system could be used for the tool as well. Otherwise the audio artist should be aware of the fact that two different sample playback engines are used.

The tool should also feature a module which is able to produce the actual data model files. In this process some data is changed and some tool specific data is removed from the model, therefore the tool needs its own file type. The easiest way to do this is by implementing the data model presented in chapter 9 and just serialize that to a file.

8.3 Pitfalls

The new architecture is not ideal in every way. There are a few points which should be taken in consideration for the design of this system. The biggest issue with the proposed architecture is the fact that it implies that the tool and the game have separate and different playback systems. Testing the music with the tool is therefore not fully representative for the behavior in the game. The same goes for the update function of the library. Both the tool and the game can have completely different time delants between each update call. This results in different precision ratings.

The library only orders an unknown playback system to start playing a sample. This implies that the library does not take the latency between ordering to play a sample and the actual playback into account. This is not a problem as long as the latency is constant, if this is not the case the quality of coherency of the music will greatly decrease.
Figure 8.3: The complete architecture of the new adaptive music system.
Chapter 9

Data model design

This chapter will give an overview of what parts the adaptive music data model is comprised of. This is done through the design of a data model which has the purpose to store and structure the adaptive music data. This chapter will only explain the basics of how the model works. A further more in-depth discussion about the data models subtleties can be found in Appendix B.

9.1 Project structure

![Figure 9.1: Project structure.]

The scope of the data model is a whole project which is in this specific case a whole game. The reason a whole game is presented in the data model instead for example just a level is that the same resources can be re-used through the whole project.

A project can have multiple levels which is called an arrangement in the data model. Only one arrangement can be active at one point in time, just like only one level can be active in a game. There are no facilities to transit between arrangements. When switching arrangement the current active arrangement will just stop and will be replaced with the new arrangement.

An arrangement is filled with a set of states. A state contains one musical theme usually corresponding to the state of the player in that specific level. This means that each state has one static BPM value. We will see later how the state internals are used to produce a dynamic score. It is possible to travel from one state to another by means of predefined state transitions.
9.2 Samples

![Diagram](image.png)

Figure 9.2: Samples.

The project manages all sound samples. By managing the samples from the project level all samples can be reused throughout all the arrangements. A project has a collection of sample libraries which is essentially a file system folder path or a group of samples. The samples all have their own object to not only indicate the absolute path to the sample but also other meta data like sample duration.

9.3 State structure

![Diagram](image.png)

Figure 9.3: State structure

The repetitiveness of the music score in a state can be greatly reduced by a system that allows automated dynamic music within a state. This can be done by means of horizontal sequencing and vertical orchestrating.

A state is structured to support both these musical design patterns. A vertical orchestrating structure can be made with channels. Multiple channels can be active and playing in parallel. Vertical orchestrating can be accomplished by activating and deactivating channels.

Channels are made out of voices which are essentially wrappers for audio samples. A voice contains only one sample and defines where on the sample the first measure starts and where the musical end of the sample is. This structure is necessary because a sample can have a lead in and lead out before and after the actual first or last note is played. By sequencing these voices over time horizontal
sequencing is achieved. The idea is therefore that only one sample should play simultaneously in a channel. This helps to control the polyphony issue, although it should be noted it is not enforced because there is usually a small overlap when transitioning from voice to voice. The number of simultaneously playing voices per channel is therefore limited to two.

9.4 State control

All we need now is to automate the voice sequencing and channel orchestration to provide for a dynamic music score. This is accomplished through the use of events. An event could best be described as a timed task, it is placed on a defined moment in time on a voice and it will do what it is supposed to do when this point on the voice is reached in playback. There are different flavors of events, in this case the most relevant events are the voice event and channel event.

The voice event will start another voice in the same channel thereby delivering the tool to creating a sequence of voices. Which other voice is started is determined randomly but can be controlled with weights. The voice event respects the lead-in time of the next voice and the lead-out time of voice containing the event. When no voice event is added to a voice it could mean that the containing channel will be deactivated because this can potentially cause the voice sequence to end.

The channel event activates another channel unless it is already active. This is done much in the same way as the voice event: a random channel will be chosen controlled by weights. This adds a new layer to the music.

Other events are an effect event which can trigger a predefined sample modulation effect or the playback of a separate sample on a given time in the voice. And lastly there is an export event which is causing a signal to be send to the game that this event is reached. The export event provides a way to let the music communicate back to the game, which can be used when the game needs to adapt to the music.
9.5 State transitions

For now techniques that only enable the creation of dynamic music have been discussed. To achieve adaptive music an audio designer has to be able to redefine events in the adaptive music model which can be called from the game. This is realized in the data model by two components, the external event and trigger.

The external event is the same as an (internal) event in the sense that it invokes an action which influences the course of the music. This is where the similarities end though. Unlike the (internal) event the external event has its action and timing separated in two components. This is done so the action can be requested by the game, and the adaptive music system can handle the timing. The method for the adaptive music system to know when it can execute a requested external event is by the use of triggers. Triggers are placed on a voice much like a normal (internal) event, they function only as markers for a point in time or a range of time. Each external event has a number of linked triggers which represent all possible timings an external event can be executed.

If this is not the case the request will be queued for a predefined time until such a situation does occur. If this situation does however not occur in the predefined time the event will be discarded.

Like the normal (internal) event there are a few different external events. To explain the more important transition external event the effect external event needs to be explained first. The effect event allows the game to request a sample modulation effect to be applied on the active voices with corresponding triggers. In practice this can be used to integrate audible feedback from gameplay events in the music.

The most important external event is the transition event. This event describes how a state transition should be carried out. This is done by linking multiple effect external events. When at a certain point all active voices have a trigger linked to the transition event at the same time all linked effect events will be executed as well based on their linked triggers. This enables the utilization of all the modulation and sample effects to cover a state transition.

There are two other external events which are the stop external event and the continues external event. The stop event stops all voices it has a trigger on when
that trigger is reached. The continues event is much like the effect external event with the exception that the modulation properties are not predefined, instead the properties can be set in real time by the game. This allows for the game to have a more analog 'tension' level represented in the music for example.

9.6 Conclusion

This data model makes it possible to describe a complete adaptive music score. This chapter has not covered all details of the model. If questions persist the more thorough data model specification with class definitions and relations can be looked up in Appendix B.
Chapter 10

Library design

The library component is arguably the most complex part of this adaptive music solution internally, but it is designed to be implemented as easy as possible in a game or tool. The library is fed with some data structured as described in chapter 8, than an external program can request the activation of certain events and the library outputs timed orders to start the playback of audio samples. While the concept sounds simple at first the implementation will prove to be the greatest challenge of the adaptive music system. It needs exact specification of input and output messages, so the external program can interface with the library accordingly.

These concepts are further explored in this chapter. First the library will be divided in more sizable components. After that a more in dept procedure will be described of how the library should be used. This leads to a more detailed specification on what the library communicates and when. Finally an explanation will be given why we need buffering and what consequences this has for the implementation.

A more technical approach of this library is available in Appendix C.

10.1 Procedure

While this library can be used in any application that makes sense in an adaptive music context there will only be referred to games in this chapter. When the library needs to be implemented a representation of the following procedure should be coded in the game.

10.1.1 input

The first step always is that an adaptive music file structured like the data model from chapter 8 should be loaded in the library. This causes the library to have a local representation of that data model. This action similar as putting a cd in a cd player. The library will use this data as long as no other data is loaded or the game is closed.

Parallel to this the game can subscribe to this library, the library in turn will relay all its future output to its subscribers. This is done by the use of two output interfaces which provide prototypes of all functions the library can
call. These two interfaces are the playback interface and the tool interface. The playback interface contains a list of functions which are exclusively aimed to handle the playback of audio samples. In other words: this drives all audible output of the library. Implementing and subscribing this interface is enough to make use of the adaptive music library for a game. For an authoring tool more feedback could be needed for playback visualization. For example it is interesting for an authoring tool to display when a voice or channel is activated or deactivated. The tool interface accommodates for these functions. The tool interface is an extension on the playback interface so by subscribing a tool interface the playback interface is automatically included as well.

At this point the library knows what to play and to whom it should output to. Right after the data model is loaded in the library the game can start gathering information about the data. It can request for a list of possible arrangements, samples and external events. The library returns with plain lists of data model elements, this way the game does not need to know much about the data model’s structure.

Now the library should be updated by the game. One way to do this is by calling the update function in the library at every iteration of the game loop, thereby giving the time the last iteration took to complete. Note that all timed logic of the library is connected to this update function, meaning that the precision of the output of the library will increase the more frequent the update function is called on the library. In practice the update function should be called about 30 times a second to maintain coherent music. There are two advantages of keeping the library run this way. First because the iteration time it passed in the update function, the rhythm in which the update function is called does not need to be constant. It should however be noticed that it is a good practice to not make it fluctuate too much. Another advantage is that this method allows to not only include the adaptive music system in the game loop but also give it its own thread, which is the second way to implement the library in the game.

When the data model is loaded, the subscribers are known and the update function is called periodically the actual playback can commence. But in order
to start play the musical score in a level we first need to define what level the game currently is presenting to the player by setting a current arrangement. Now a list of all relevant external events in that arrangement can be requested. There is no specific 'play' function in the library. Essentially to start playing a transition needs to be made between 'no state' to the desired state that should be played. This is can be defined in a transition external event. The next step would be to activate the transition event from 'no state' to the desired state. This is the point where the actual playback starts. From this point on the music progresses based on the state internal logic and any relevant external event that is activated.

10.1.2 output

When the library is running and the game is making its requests through external events the library will care about all the logic and timing. This internal work presents itself through the aforementioned playback and tool interfaces.

The playback interface handles all output, like the name suggests, that are necessary to properly playback the adaptive music. This includes the starting and stopping of samples, the appliance of sample modulation effects and the loading and unloading of sample instruction whom will be discussed later in the 'buffering' section. Overall this makes the amount of actions to be implemented in the game 4 plus the number of possible different modulation effects.

The tool interface can be used to expose all kinds of internal workings of the library. It has functions that indicate the activation and deactivation of states, channels, voices and events. The tool interface is obviously only interesting when the library is used in an authoring tool.

10.2 Buffering

There are a few input functions in the library that have not yet been discussed. These functions are related to the buffering of future events. It would be a lot easier to implement a solution that would just determine real-time every time the update function is called if the adaptive music model demands for another sample to be played. This however is not possible because of two reasons.

The first reason has everything to do with uncertain sample load time. The library has no knowledge about how and if the external audio sample player loads its samples. This is why the library uses a flexible system for sample load events by providing a function in which two values are passed. One value is the comfortable amount of time before a future use a sample a load notification should be put out. The library will try to do the load notification in this time frame, but it does not have any consequences if this time frame is not reached. The second value is the minimum time a load notification should be put out before the activation of a sample. This actually is a binding amount of time. The library can choose to discard voice events if the minimum time frame can not be made. This has a lot of correlation with the second reason why buffering is needed.

The second reason is a bit more complex because it delves deeper in the way the adaptive music system behaves. There are actually two cases which lead to this reasoning. The first case is visually presented in Figure 10.2. This
Figure 10.2: Buffering need example.

The figure shows 7 measures of a possible adaptive music score. The blue bars are the samples, they are surrounded by green voices and finally the yellow square represents a voice event which is set up to either activate the voice containing sample 2 or 3. Now if the voice containing sample 3 will be chosen there is no problem, on the 5th measure sample 3 will be started while sample 1 will play for another 3 beats in the 5th measure. However when the voice containing sample 2 is chosen by the voice event the sample activation would be two beats too late. This is the reason voice and channel events need to be planned out ahead. This is even more relevant when we look at transition events. These events employ a range of other effect events which can just as easily take place before the actual state transition. The buffer value, expressed in amount of steps planned ahead, can be set by a function on the input interface of the library. Like mentioned before the buffering is directly correlating with the sample loading. When the buffer value is too small the playback might just stop because the library cannot comply with the minimum load time restriction. It is therefore important the amount of steps planned out ahead always represent more time than the minimum load time. Note that the bigger the buffer becomes the more overhead it is going to produce each tick.
Chapter 11

Tool design

To accompany the data model and library an authoring tool has been created called Accido. Accido is a Latin verb for ‘to happen’ and ‘to occur’. This describes what this software tool does, it is an authoring tool to create music which reacts appropriate to happenings or occurrences in an unspecified context.

This chapter will first and foremost provide an overview on how the tool is made. Other than this it gives a more visual impression on how the adaptive music data model is supposed to function. This will be accomplished by the explanation of the visual representation the tool presents for the adaptive music data model.

11.1 Goals

The main goal of Accido is to obscure all unnecessary technical details of the data model structure for the audio designer. It should simplify the adaptive music creation process for the audio designer thereby reducing production time of adaptive music scores.

These are long term goals, the first step will be the creation of a graphical front end of the full adaptive music data model.

11.2 Requirements

The creation of Accido has been very open ended from the beginning. The only real requirement from Coded Illusions was the use of C# as programming language for the tool. The tool only has to work under the windows operating system so from the aspect of OS requirements C# should suffice. Other than that C# allows for rapid user-interface design and implementation. This will come in handy as a large part of the tool is the user-interface.

The whole tool is build out of 3 packages which are GUI, Data and Logic.

11.2.1 Data

The data package contains an exact representation of the data model described in chapter 9 and Appendix B, with a number of extensions.
First of all it features C#'s default XML serialization. This feature is perfect for the creation of adaptive music type files. With very little effort the data model can be converted to XML this way. The only extra thing needed here in the data model is a function to reconstruct references by id after loading the XML file. Another function of the XML serialization code is the copying and pasting of project elements. This is done by one function that serializes itself and its children to an XML string which is in turn copied to the clipboard. Another function can interpret these XML strings and place it back in the data model.

The data model also contains some functions to ensure the data model's consistency. It automatically hands out unique ids and names to project elements. All forms of simple value validation is done in this package.

The data package also contains functions for adding and deleting project elements. These actions and edit actions fire events. These events are mainly used by the GUI, this way the GUI can update in real-time.

And lastly it contains some descriptions for certain variables which can be set directly from the GUI. These descriptions are used in the propertygrid GUI component which will be shown in the layout section.

11.2.2 GUI

The GUI package contains all GUI components. All but the propertygrid are custom components based on standard .NET Framework components. All components of Accido are contained in one window. Separate windows are only opened for file browsing and some error messages.

Custom components are context menus. These menus feature functionality like adding, renaming, copying, pasting and deleting project elements. Each different type of project element has a unique context menu.

A treeview has been added to visualize the tree structure of the adaptive music data model. It is customized so treenodes can be renamed and sorted correctly.

The editview is the most custom component made for Accido. It features specialized editors for each project element type that need a custom editor. The editors are for now build out of standard .NET Framework components because these have a lot of features by default which can save a lot of development time. It is not ideal because of the limitations and poor performance of these components, but it is good enough for the first version of the tool.

11.2.3 Logic

The Logic package contains everything not closely related to the data and GUI. It has some playback related classes like a controller which communicates with the playback library and an audio sample player. Other interesting parts are handlers. Handlers consist of different classes which handle all the logic that does not belong in either the GUI or the Data package.
11.3 Layout

Accido has three panels: from left to right the treeview, the editview and the propertygrid. The treeview displays the complete project tree, and has a tab to switch to the sample libraries. The propertygrid shows variables from the currently selected project element that are relevant to the user. Both the treeview and propertygrid are rather static in layout and content. The editview however is a custom made GUI component. It can display a few different things. If a state is selected it displays all channels with in it all voices, events and samples. If an arrangement is displayed it shows an overview of all external events with buttons to fire them. This can be used for simulating the music playback in the tool.

![Figure 11.1: Accido with an example project loaded.](image)

Other parts of the interface are some menus and toolbars. These feature basic functionality like creation of project elements, deleting project elements, copying, pasting etc. This way Accido delivers all the features of the adaptive music type.

11.3.1 Designing

This subsection describes how an adaptive music score is designed by the use of the example of the HQ state in Figure 11.2.

The example above is a project with two arrangements where the HQ state has been selected as can be seen in the treeview. The HQ state is a typical example where the vertical orchestration design is applied. The idea is that we have a main channel here called Head, this channel manages the state. As can be seen in the treeview the voice contained in the head channel has a two triggers four channel events and one effect event. These elements are also shown in the editview, but most of them are positioned in the same place which is the
reason most of them are not directly visible. Each channel event starts either one of the other channels or the head channel, which is already playing at time, so the latter would effectively cancel the event. All the layers play only once, the head channel loops infinitely. This gives the HQ state the potential to have a different combination of layers every 16 measures.

To cover the looping every 16 measures an effect event has been added at the end of the head channel. In this case it will play a random sample from a selected list of samples to cover the looping process.

Triggers are place markers where an external event is allowed to activate. Triggers are here displayed as gray dots. As can be seen in the treeview the head channel's voice has two triggers, in this case one at the very beginning of the voice and one at the end.

The first trigger allows effect events which are child of the transition event representing the transition from 'no' state to the HQ state, this way the arrangement can be started with this state.

The second is for HQ to another state or to 'no' state (stop) transitions. These are linked to two types of external events: transition events to allow the state transitions and an effect event which in this case contains a fade out effect to smooth the transition.

This example is still a somewhat simple case. It has just a few voices and just one state to state transition possibility. This means that the response time of a state to state event can take up to 14 measures in the worst case scenario. A maximal response time of four measures is usually required. This can only be accomplished by manually adding triggers on every possible transition and linking those to the transition event.

![Testing function of Accido](image)

Figure 11.2: Testing functionality of Accido, left the an overview of the available external events, right a state with one voice and sample playing.

11.3.2 Testing

When an adaptive music arrangement has been created, it can be tested by clicking the pause/play button on the toolbar. This does not actually play anything it just loads the current data model in the playback library. It will start playing when a 'no state' to state transition event is called. This event can be fired from the edit view of an arrangement, which gives an overview of all possible external events and button to activate them. This will instruct the playback library to start ordering the playing of samples. This can be heard
and is visualized in the editview of a state by highlighting the playing voices and samples.

11.4 Conclusion

For now Accido will suffice as prototype tool to demonstrate the potential of this adaptive music system. It provides all features of the data model, but it could use some work on the usability aspect. A lot of this is already possible by the use of the clipboard. Features that would make the audio designers life easier could be a template system to simplify regularly applied structures like cross fades. Another nice feature could be a batch function to add multiple triggers.
Chapter 12

Conclusion

This concludes the discussion of the proposed data model. The final solution is compliant with the requirements mentioned in chapter 6 and all challenges or at least partly overcome.

The new adaptive music solution can be implemented with the use of a software library dedicated to the adaptive music logic. This library can receive high level instructions about the desired course of the music and it outputs low level instructions about in terms of samples that should be played.

The way high level instructions are translated to low level instructions is described in an adaptive music structure which is represented by a data model. This data model can cover a complete game and can therefore contain multiple music scores. These music scores are build up out of states which have internal dynamic music support to prevent repetition. Finally there is support for fully customizable state to state transitions.

To design these adaptive music structures an authoring tool has been created which should obscure the complexity of the data model, and make it easy for an audio designer to work with the adaptive music solution.

The currently the state of this new adaptive music solution is very close to completion. After which it can be tested which should reveal the true potential of this solution. In the end the audio designer should have full flexibility in creating an adaptive music score within the boundaries of the adaptive music structure without programming intervention.
Part III

Appendixes
Appendix A

Diagram clarifications

A.1 UML clarifications
Surely UML is a standard language so it should not be open for interpretation. Still I think it is good to explain exactly what is emphasized in this case for certain UML language aspects. This can function as reference as well.

A.1.1 Class
Classes in this document are represented as displayed here. For some classes there are no attributes and for others there are no operations defined. This is based on the relevance of these. Likewise the access modifier for operations and attributes are never set. All attributes in this document are automatically private and all operations are public. Private operations are not all that interesting for this document because this mainly describes how classes work internally, which can be found in the source code. Because all attributes are private they can be accessed and modified by accessors and mutators, these operations are also not included in the UML classes.

![Class1](image)

Figure A.1: Class

A.1.2 Abstract Class
Abstract classes are classes that cannot be instantiated. They are just here to provide a generic type for certain classes and to hold their common attributes and operations. They are displayed as a class with an italic class name.
AbstractClass

Figure A.2: Abstract Class

A.1.3 Generalization
These parent-child relations are commonly used in this document. Just to be clear: the arrow is pointing to the parent.

Figure A.3: Generalization

A.1.4 Association
There are two types of associations which are aggregations (white square) and compositions (black square). They both represent a reference to another class. The square is placed on the side of the class that is referencing the other class. This class holds the attribute pointing to the other class.

Figure A.4: Association: Aggregation

The difference between aggregations and compositions is that when a class is deleted with composition a of other classes, these other classes are removed as well. Whereas with a aggregation these other classes remain intact, and are usually also referenced by another class.

Figure A.5: Association: Composition

A.2 Arrangement view clarifications
In this document a state is visualized in some sort of arrangement view. This image explains some symbols used here.
Figure A.6: Explanation of the arranger diagrams.
Appendix B

Data model Specifications

This chapter describes what the adaptive music data model looks like. And how this model should be interpreted for playback.

B.1 Overview

It’s important to notice that the model has a tree structure. This gives a nice display on what level classes exist. It is best to follow the tree first from Project down to Voice. This could be considered the spine of the model, everything else
is connected to those classes. This is also how XML will be nested later when
the data model is serialized. All other relations are referenced by id in the XML
format. This model will now be discussed step by step.

B.2 Types

First we need to discuss three data structures because they are used through
the entire model.

Figure B.2: ProjectElement with all its children.

B.2.1 ProjectElement

Almost all classes in the model are derived from ProjectElement. It holds
meta data of the element like an unique ID, a user friendly name, and a parent
project element. Because the model has a tree structure, each element can have
just one (tree) parent, although it does not necessarily have to be one class type
of parent. It also contains some notes about the element. The ToolID is
an optional field which can be used by an authoring tool to link other project
elements. This is in no way used for playback so in practice it is used for tool
visualisation.

B.2.2 ProbabilityItem

A ProbabilityItem forms a connection between two project elements, it
contains a weight value which represents the chance the referenced project ele-
ment is chosen for some elsewhere defined goal. This is used to create controlled random structures throughout the model. There are actually two types of probability items. One is containing its element the other is just referencing it. When which is used depends on whether the probability item is used to reference existing project elements or if it has created its own element. Practically this means two things for a probability item which is containing an element. The first is that on deletion the containing element will also be removed. And secondly the parent of the containing element is the probability item. For a referencing probability item the opposite is true. On deletion the referenced element will not be removed (because it is contained elsewhere) and the parent of the referenced element is not the probability item, instead it is the element that is containing the probability item. This is done like this because this is also the way the XML representation is structured.

**B.2.3 TimeItem**

Time could be measured with one single number. Yet this is not that convenient for music. From music Measures (Bars) and Beats are used. `TimeItem` provides this structure. It also contains an extra value called `warmup`. This is a floating point value expressed in seconds. This warmup time is typically used when the real music in a sample starts after a short while, and the music 'warms up' to the point where the first note is played.

**B.3 Base**

This is the complete overview of the base package. The tree structure is visible again here. All other class parents are the classes with the collection of that class in it. For example the Arrangement class has Project as parent because project has a collection of arrangements.
B.3.1 Project

A project could be described as a game or in a more literal way as an audio CD. It contains a number of arrangements and sample libraries. Other than that it is just there to keep everything together. It is the root node of the tree.

B.3.2 Arrangement

An arrangement could also be a level of a game or to continue the analogy: a track on a CD. It is possible to use multiple arrangements in one level as long as they are not played simultaneously. There is no transition support between
arrangements whatsoever. If another arrangement is activated the previous arrangement will just stop right away, like when the next track is started on an audio CD.

An arrangement has a collection of external events. External events are event that can be called externally from the adaptive music system. This is the communication channel from the game to the adaptive music system. The game can request a relevant list of external events when an arrangement is loaded.

### B.3.3 State

A state is an in-game state. This could for example be a positional state of the player in a level if the audio artist wants different music for every room. Or it could be the behavioral or emotional state of the player like combat, stealth, sad, happy etc. These states can be completely different musically and still transit smoothly into each other as long as custom state to state transitions are defined.

It contains some state settings and a list of channels, the latter will be discuss later on.

- **BPM:** Beats per minute.
- **Beat:** The number of beats in a measure (the 3 of $\frac{3}{4}$).
- **Measure:** The base beat size in a measure (the 4 of $\frac{4}{4}$).

These are all timing settings meaning the rhythm of a state is constant. Another state should be made if different rhythmic settings are required. The measure setting is technically useless, but it is handy information for the audio designer.

### B.3.4 Channel

Channels have been added to group the later discussed Voices. They have two main functions, it brings more order in the state, and it helps with voice management.

A voice is not much more than a wrapper for a sample. In a channel only one voice can be played at a time (actually two for overlapping transitions between voices). This is an important thing to consider because we can not just play an infinite amount of samples simultaneously on audio hardware. A Xbox 360 can handle a total of 256 samples at a time, but for a PC no more than 32 simultaneously playing samples can be assured, and that is including all other sound effects.

Therefore this structure is important because it helps the audio artist to manage voices. A horizontal sequencing design could be made within one channel in this case. A vertical orchestration design would need a separate channel for every layer that can play simultaneously.

It contains a collection of voices and two more settings.

- **Volume:** The initial volume the samples in this channel will be played.
- **bInit:** Whether this channel should start playing when it’s parent state is activated.
B.3.5 Voice

Like said before a voice is basically a wrapper for a sample, but it also contains the controls to start a follow-up voice. It contains a list of triggers, which are important for external event timings and will be discussed later. It also contains a list of internal state events.

An event in a voice can have many functions, at the moment only two are relevant: events can start other voices in the same channel and initialize other channels. Starting other voices or channels can be done in a controlled random manner. This is how the audio designer can keep the music unique every time it is played.

A voice obviously contains a reference to a Sample, and has some other variables.

- **StartTime**: The time before the sample starts, if this timing is negative the sample will start playing even before the voice is formally activated.
- **TotalTime**: The total length of time of the voice. This allows events or triggers to be placed after the sample has stopped. This value can not be negative.
- **MinLoops**: The minimal amount of times this voice should be repeated, the value should always be smaller than or equal to the MaxLoops. While playing these loops no voice events should be triggered because that would break either the loop or the voice management (since only one voice can be played in a channel at a time).
- **MaxLoops**: The maximal amount of times this voice is allowed to repeat. When a voice arrives at its max loop the first voice event will exclude the current voice as an option. When no voice events are present the channel containing this voice will be deactivated.
- **InitProbability**: Initial probability is the chance relative to other InitProbabilities of all other voices in a channel this voice will be chosen when the parent channel is activated.

B.4 Sample Library

The sample library package is the layer between the data model and the file system. It is mostly a separate resource for the rest of the model. All the sample libraries are accessible throughout the whole project. This is why the project class it referencing all the libraries. A sample is usually referenced from voices.
B.4.1 Sample Library

The sample library is not much more than a class which holds a path to a directory with samples. This path can either be relative or absolute. A relative path is relative to where the project file is located. The sample library is actually only used for authoring tools to structure sample lists. For playback only the loose samples are relevant.

B.4.2 Sample

A sample is the data representation for an audio sample on the file system. It contains a path to the sample. The real use of the path string is really open ended since the player library will just use it as identifier when the sample needs to be started or loaded. How the path string is interpreted is completely up to the recipient of the playback library output. The sample class also contains the duration of the sample. This is done so when playing back the adaptive music score the library does not have any knowledge about the contents of a sample. The only relevant information here would be the duration of the sample. Duration is expressed in seconds.

B.5 Internal Events

Events are static points in time on voices. They can cause multiple different actions which are mostly aimed to improve the diversity of a single state. The export event is the one exception to this.

An abstract event only contains a certain timing, which defines the moment in a voice it will be triggered. An event can be excluded from certain voice looping by adding that loop number to the ExcludeLoops collection.

B.5.1 VoiceEvent

The voice event is a tool for realizing horizontal sequencing. It picks one voice from a list of voices within the same channel represented by a probability item. Typically these events can be found at the end of almost every voice when horizontal sequencing is applied. It also contains a AllowFinish variable.
When true the voice bearing the voice event will finish while the new voice is starting. This allows voice overlapping for voice transitions.

### B.5.2 ChannelEvent

The channel event is a tool for realizing vertical orchestration. It activates one random channel from a list of channels. If bStopChannel is true the channel that bears this channel event will stop either right away or when the current voice has ended depending on the bStopFinish variable. This makes it also suitable in a way for horizontal sequencing, although this would make the state structure more complex. Note here that a channel can play only once at a given time. A channel event which will start a channel that is already playing will therefore be discarded.

### B.5.3 EffectEvent

An effect event is a way to access the effect capabilities of the system from within a voice. A random list of effects are accessible through probability items. The option to create a list of effects per event is done for flexibility, but usually just one effect is needed. A certain piece of music can be accented by for example increasing the volume but also by playing a stinger sample. The bOnlyLastLoop and bOnlyFirstLoop allows the event only to be activated for respectively the
last and first time a voice is looped. This is useful for utilizing these events for covering voice transitions.

### B.5.4 ExportEvent

The export event was originally called an external event, but because that name would be ambiguous with the external event package it has been renamed to export event. It gives the ability to trigger events from the music in the game world. This is useful in situations where it is needed that the game world adapts to the music instead of the other way around. The game uses the name variable from `project element` to identify the event.

### B.6 Triggers

Triggers are points in time or ranges of time on voices where a certain predefined external events can take place. Exactly how these triggers are used is different for every type of external event, and will be explained later when the external events are discussed. The only real functionality of a trigger in itself is to mark a certain point in time or define a time range on a specific voice.

![Diagram of Trigger package and its relations](image)

**Figure B.9:** Overview of the Trigger package and how it relates.

#### B.6.1 TriggerPoint

A trigger point marks a point on a voice where the list of external events specified in the `EventLinks` collection can be activated. The `Timing` variable indicates where on the voice this point is positioned.

#### B.6.2 TriggerRange

A trigger essentially contains a trigger point with a duration. `StartTime` is where the range starts on the voice and `TotalTime` is how long the range is.
B.7 External Events

External events are the events triggered by the game. Usually the actual events are not activated until the right conditions are met. These conditions are specified by trigger points and ranges. An external event has a certain lifetime. This ensures that events are not waiting forever to be triggered. The problem with this is that some events could be taken from one state to another, where it does not have any triggers. Then it would start playing right when it possibly gets back later to a state that has triggers for that specific event. The audio designer could say in this case that he would like a certain thing to happen in his music, but if that would take longer than a certain amount of time it would not be necessary anymore. Note that this lifetime is saved as time item, so the actual lifetime can vary when going to a state with a different BPM.

Figure B.10: Overview of the ExternalEvent package and how it relates.

B.7.1 StingerEvent

Stinger events are events which will perform their predefined task when activated. This can be either using some kind of effect or stopping some voices. When a stinger event is activated from game the event task is carried out the moment some or more voice(s) come by a trigger that is linked to this stinger event. Now if bOneUse is true the event will be discarded, otherwise it will still be active and ready to wait for another trigger occurrence until its lifetime has ended. The task in a stinger event is only applied to the voices that have corresponding triggers.

B.7.2 EffectEvent

The effect event can call a few different effects which are later discussed. These effects have some duration by definition. These effects can be used to emphasize some moment in-game by for example raising the volume of a certain voice or play an extra sample. It can also be used to assist state transitions. It also
contains a Curve this curve represents linear time on the x axis and an output value on the y axis. This output value is used to modulate the predefined effect over the effect events duration.

### B.7.3 StopEvent

The stop event simply stops the voice that has a trigger for it at the time this event is alive. This is most useful for state transitions.

### B.7.4 TransitionEvent

The transition event forms the most important part of the external event package, and probably this whole adaptive music system. This is where the state transitions are set up. It contains a previous and next state. One of these states can be null which is equivalent to ‘no state’. This way the audio artist can build a ‘play’ event for his arrangement by creating a transition event from null to an existing state.

The transition event has a collection of stinger events. Since they are only referenced by a transition event they will not show up in-game as events that can be fired. Yet they feature the same functionality as normal stinger events. These stinger events can only be triggered when transition event will be or was triggered in ‘PreTime’ before the transition event or ‘PostTime’ after it.

### B.7.5 ContinualEvent

A continual event is a lot like the effect event with the exception that the effect event has its own curve and duration to perform its effect, while a continual event can continually be fed from the game with values. This event only, and always works the moments a voice has a trigger range for it. This way a game designer could input a continual stream of abstract values representing the amount of in-game action which is some value between 0 and 1. Which can then in turn for example be translated to a volume ramp. The Default Value variable is the value that will be inserted in the effect as long none are given from the game.

### B.8 Effects

The effect package has a collection of effects to be applied to the music. These events can be triggered once or continually modified by an external event or triggered internally from a voice.

### B.8.1 SampleEffect

The sample stinger effect is used to start one sample from a random list of samples. This can be used to add an extra layer of music for certain moments in-game. This list of samples is represented by a channel and uses all functionality of a channel. This channel has the sample effect as parent.
B.8.2 SoundEffect

Sound effects are used to create all kinds of analog effects. These effects can be constantly fed with a value between 0 and 1. The sound effect will then modify all its curve values according to that input value.

B.8.3 VolumeEffect

The volume effect is a sound effect which modifies the volume according the curve. How curves work will be explained in the next section. One could for example add one full sine wave as curve, which would mean that when the value of 0, 0.5 or 1 would be entered the volume would be back to its original level. And with the value of 0.25 or 0.75 it would be either very high or very low, where 'very' is the amplitude of the sine curve.

B.8.4 FilterEffect

The filter effect works about the same as the volume effect, it just has more and other parameters. The first is filter type, this could be a low pass or high pass filter with different dB values. The cutoff curve represents the value where the frequency is cut off, the slope is the value that determines how smooth this cut off is.

B.9 Curves

The curve package is in an internal system to define curves to provide smooth effect parameter changes. At the moment it is really simple and could probably be done better but that is a beyond the scope of this assignment. For now there is only a curve class with a list of 2d points and a curve type. This curve type could contain values like 'linear' which would be straight lines between points which is useful for simple ramps for fade out effects. Or it could be 'spline' for smooth lines. How these curves are exactly used is not really relevant. They are just there to provide a customizable output for a linear input.
B.10 Overview

All these pieces fall into one greater model. Note that this model does not show all relations. Especially all generalization towards the project element would not really make the model any more clear. It is added to give a complete view on the data model.
Figure B.13: Full adaptive music data model.
Appendix C

Library Specifications

This appendix will present the functions that can be called on the library and the interfaces the library uses to fire its events back to the subscriber. A few requirements will be handed on how the library handles the adaptive music model’s logic.

C.1 Library interface

<table>
<thead>
<tr>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Load(data: void*, size: int)</td>
</tr>
<tr>
<td>+Subscribe(interface: PlaybackInterface*)</td>
</tr>
<tr>
<td>+UnSubscribe(interface: PlaybackInterface*)</td>
</tr>
<tr>
<td>+Update(deltaTime: float)</td>
</tr>
<tr>
<td>+GetArrangements()</td>
</tr>
<tr>
<td>+SetArrangement(arrangement: int)</td>
</tr>
<tr>
<td>+GetEvents()</td>
</tr>
<tr>
<td>+FireEvent(event: int)</td>
</tr>
<tr>
<td>+SetParameter(event: int, value: float)</td>
</tr>
<tr>
<td>+GetSamples()</td>
</tr>
<tr>
<td>+SetBufferSize(steps: int)</td>
</tr>
<tr>
<td>+SetLoadTime(minimalTime: float, idealTime: float)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>+id: int</td>
</tr>
<tr>
<td>+name: String*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>+id: Integer</td>
</tr>
<tr>
<td>+url: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>+id: int</td>
</tr>
<tr>
<td>+name: String*</td>
</tr>
</tbody>
</table>

<<enum>>+EventType

Figure C.1: Classes involved in application to library communication.

The library has some public functions which either the game or the tool or any other program which is including the library can use to communicate with the library.

The first feature of the library is **Load**, this function loads the adaptive music data model. Its parameter is a pointer to the XML adaptive music data.

The **Subscribe** function shows the library to which **PlaybackInterfaces** it should send its events to. Multiple interfaces are possible this way by calling it multiple times with different playback interfaces. These subscriptions can be removed by using the **UnSubscribe** function.

The **Update** function should either be called by the game or tool as much as possible. The more frequent this function is called the more accurate the synchronization will be. In game, this can be tied to the game loop or if preferred,
even delegated to a separate thread. The time between the last call of Update and the current call needs to be given as delta time in seconds.

The GetArrangements function returns the available arrangements in the earlier loaded data. The returned arrangements will consist out of a name and id.

SetArrangement activates a certain arrangement by id. There can only be one active arrangement at a time, setting it again will overwrite the current arrangement. Arrangement -1 is equivalent to no arrangement. This effectively stops the playback.

GetEvents returns all available events of the activated arrangement. If no active arrangement is available no events will be returned. A returned event will consist out of not only an event name and id but also an event type. The event type is an enumeration with options for transition event, stinger event, export (internal) event and continual event.

FireEvent can fire transition and stinger events by id. These events will not be effective immediately. It has to wait until a defined moment occurs where it can become effective.

The SetParam function can supply the effect of a continual event with new values. This value is always immediately applied on the effect.

GetSamples will return all samples used in the currently activated arrangement. If no arrangements are activated all samples in the whole project will be returned. This feature provides the possibility for the engine to load all samples before actually starting the arrangement. It could also be used by an engine to import all sound files from the file system to it's own sample type.

SetBufferSize determines how many voices in a sequence will buffered. It is important to not set this value too low. The amount of buffered voices should span the largest transition pretime. Otherwise a transition can never be planned out. It should also not be too large because this gives the library more work each update.

SetLoadTime takes two arguments. The first argument is the minimal amount of time in seconds a sample should need to load. This is added as required preparation time if some event needs to be planned. Theoretically this means the lower this time is the greater the chance an event is to be fired earlier. The second argument is the ideal amount of time in seconds a sample should be loaded. If a lot of time is available this is the amount of time the library instructs its subscribers to load the sample.

Both the buffer size and load time is therefore shared among subscribers. This means it should have the values for the least performing subscriber. Also both functions only have effect when no arrangement is activated.

C.2 Playback Interface

The library communicates its subscribers through the PlaybackInterface. The playback interface provides all basic functionality to play back the adaptive music file. This interface should be registered with the library through the subscribe function. The library will call the by the interface specified functions on all its subscribers.

The LoadSample informs the subscriber that a sample will be played in a certain amount of time. This allows the subscriber to load a sample if that
PlaybackInterface
+LoadSample(sample: Sample)
+UnloadSample(sample: Sample)
+StartSample(sample: Sample)
+StopSample(sample: Sample)
+SetVolume(sample: Sample, volumesettings: VolumeSettings)
+SetFilter(sample: Sample, filtersettings: FilterSettings)
+FireEvent(event: Event)

FilterSettings
<<enum>>
+FilterType
+CutoffFrequency: Float
+Slope: Float

VolumeSettings
+Volume: Float

ToolInterface
+ActivateState(state: int)
+ActivateChannel(state: int, channel: int)
+DeactivateChannel(state: int, channel: int)
+ActivateVoice(state: int, channel: int, voice: int)
+DeactivateVoice(state: int, channel: int, voice: int)
+EnableEvent(state: int, channel: int, voice: int, event: int)
+DisableEvent(state: int, channel: int, voice: int, event: int)

Figure C.2: Classes involved in library to application communication.

would be necessary before actually having to play it.

The UnloadSample tells the subscriber that this sample will not be used for a while. This allows the subscriber to unload this sample.

The StartSample orders the subscriber to start a given sample as soon as possible.

The StopSample orders the subscriber to stop a given sample as soon as possible.

The SetVolume orders the subscriber to change the volume of a defined sample.

The SetFilter orders the subscriber to change the frequency filter on a defined sample. It features a type of filter, which could be a high or low-pass filter with different dB settings. The cutoff frequency tells the filter what frequencies to cut off, and the slope represents how smooth this should happen.

Note that all the effect settings are values between 0 and 1. This way the code in these interface functions can convert these values easily to match the sound of the effect used by the tool.

FireEvent fires the export events from the state internal events.

C.3 Tool Interface

The tool interface is a child of the playback interface. It provides some extra functions specialized for tools. It provides a set of functions to help visualizing what goes on inside the library. This information is important for the audio artist so he can debug his adaptive music score. The ids of the different components are used to communicate which specific component is handled.

The ActivateState will be called the exact moment a state transition takes place. There can only be one active state, so activating another state will deactivate the previous active state.
The `ActiveChannel` will be called the moment a channel is started. When no voices are played anymore in a channel the `DeactivateChannel` will be called.

The `ActivateVoice` and `DeactivateVoice` are used at the start and end of a voice. Also when a voice is looped the deactivate will be called right before the repeated activate voice.

`EnableEvent` and `DisableEvent` is used for both external and internal events. The voice, channel, continues, stop and export events will only be enabled. The effect event will be disabled after the stinger has completely played, the transition event will be disabled when all its related stinger events have been fully handled.
Appendix D

Data model XML example

```xml
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" id="0">
  <Name>New Project</Name>
  <Notes>Project</Notes>
  <Arrangement id="0">
    <Name>Arrangement</Name>
    <Notes>Arrangement</Notes>
    <State id="0">
      <Name>State</Name>
      <Notes>State</Notes>
      <Bpm>120</Bpm>
      <Beat>4</Beat>
      <NoteValue>4</NoteValue>
      <Channel id="0">
        <Name>Channel</Name>
        <Notes>Channel</Notes>
        <Initialize>true</Initialize>
        <Volume>1</Volume>
        <Voice id="0">
          <Name>Voice</Name>
          <Notes>Voice</Notes>
          <InitProbability>1</InitProbability>
          <StartTime>
            <Measure>0</Measure>
            <Beat>0</Beat>
            <Warmup>0</Warmup>
          </StartTime>
          <TotalTime>
            <Measure>4</Measure>
            <Beat>0</Beat>
            <Warmup>0</Warmup>
          </TotalTime>
        </Voice>
      </Channel>
    </State>
  </Arrangement>
</Project>
```
<MinimalLoops>1</MinimalLoops>
<MaximalLoops>2147483647</MaximalLoops>
<Trigger xsi:type="TriggerPoint" id="0">
   <Name>Trigger</Name>
   <Notes>TriggerPoint</Notes>
   <Timing>
      <Measure>0</Measure>
      <Beat>0</Beat>
      <Warmup>0</Warmup>
   </Timing>
   <EventLinks>0</EventLinks>
</Trigger>
<Trigger xsi:type="TriggerRange" id="1">
   <Name>Trigger(1)</Name>
   <Notes>TriggerRange</Notes>
   <Timing>
      <Measure>0</Measure>
      <Beat>0</Beat>
      <Warmup>0</Warmup>
   </Timing>
   <TotalTime>
      <Measure>0</Measure>
      <Beat>0</Beat>
      <Warmup>0</Warmup>
   </TotalTime>
</Trigger>
<Event xsi:type="VoiceEvent" id="0">
   <Name>Event</Name>
   <Notes>Voice Event</Notes>
   <TimeItem>
      <Measure>0</Measure>
      <Beat>0</Beat>
      <Warmup>0</Warmup>
   </TimeItem>
   <AllowFinish>true</AllowFinish>
   <VoiceItem id="0">
      <Name />
      <ElementId>0</ElementId>
      <Probability>1</Probability>
   </VoiceItem>
   <VoiceItem id="0">
      <Name />
      <ElementId>1</ElementId>
      <Probability>2</Probability>
   </VoiceItem>
</Event>
<Event xsi:type="ChannelEvent" id="1">
   <Name>Event (1)</Name>
   <Notes>Channel Event</Notes>
   <TimeItem>
   </TimeItem>
</Event>
<Measure>0</Measure>
<Beat>0</Beat>
<Warmup>0</Warmup>
</TimeItem>
<AllowFinish>true</AllowFinish>
<StopChannel>false</StopChannel>
<ChannelItem id="0">
  <Name />
  <ElementId>0</ElementId>
  <Probability>1</Probability>
</ChannelItem>
<ChannelItem id="0">
  <Name />
  <ElementId>1</ElementId>
  <Probability>2</Probability>
</ChannelItem>
</Event>
<Event xsi:type="EffectEvent" id="2">
  <Name>Event (2)</Name>
  <Notes>Effect Event</Notes>
  <TimeItem>
    <Measure>0</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </TimeItem>
  <ExcludeLoops>2</ExcludeLoops>
  <ExcludeLoops>4</ExcludeLoops>
  <EffectItem id="0">
    <Name />
    <ElementId>0</ElementId>
    <Probability>1</Probability>
    <Element xsi:type="SampleEffect" id="0">
      <Name>Effect</Name>
      <Notes>SampleStingerEffect</Notes>
      <Sample id="0">
        <Name />
        <ElementId>0</ElementId>
        <Probability>0</Probability>
        <Element xsi:type="Sample" id="0">
          <Name>chimes.wav</Name>
          <Notes>Sample</Notes>
          <Path>/chimes.wav</Path>
          <Duration>0.6313379</Duration>
        </Element>
      </Sample>
    </EffectItem>
  </EffectItem>
</Event>
<Event xsi:type="EffectEvent" id="0">
  <Name />
  <ElementId>0</ElementId>
</Event>
</Event>
<Probability>2</Probability>
<Element xsi:type="VolumeEffect" id="0">
  <Name>Effect (1)</Name>
  <Notes>VolumeEffect</Notes>
</Element>
</EffectItem>
<EffectItem id="0">
  <Name />
  <ElementId>0</ElementId>
  <Probability>3</Probability>
  <Element xsi:type="FilterEffect" id="0">
    <Name>Effect (2)</Name>
    <Notes>Filter Effect</Notes>
    <FilterType>LOW_PASS_6DB</FilterType>
  </Element>
</EffectItem>
</Event>
<Event xsi:type="ExportEvent" id="3">
  <Name>Event (3)</Name>
  <Notes>Export Event</Notes>
  <TimeItem>
    <Measure>0</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </TimeItem>
</Event>
</Voice>
<Voice id="1">
  <Name>Voice (1)</Name>
  <Notes>Voice</Notes>
  <InitProbability>1</InitProbability>
  <StartTime>
    <Measure>0</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </StartTime>
  <TotalTime>
    <Measure>4</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </TotalTime>
  <MinimalLoops>1</MinimalLoops>
  <MaximalLoops>2147483647</MaximalLoops>
</Voice>
</Channel>
<Channel id="1">
  <Name>Channel (1)</Name>
  <Notes>Channel</Notes>
  <Initialize>true</Initialize>
  <Volume>1</Volume>
<Voice id="0">
  <Name>Voice</Name>
  <Notes>Voice</Notes>
  <InitProbability>1</InitProbability>
  <StartTime>
    <Measure>0</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </StartTime>
  <TotalTime>
    <Measure>4</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </TotalTime>
  <MinimalLoops>1</MinimalLoops>
  <MaximalLoops>2147483647</MaximalLoops>
</Voice>
</Channel>
</State>
<ExternalEvent xsi:type="TransitionExternalEvent" id="0">
  <Name>ExternalEvent (3)</Name>
  <Notes>Transition Event</Notes>
  <Timing>
    <Measure>4</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </Timing>
  <PreTime>
    <Measure>1</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </PreTime>
  <PostTime>
    <Measure>1</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </PostTime>
  <Arrangement>0</Arrangement>
  <PreviousState>-1</PreviousState>
  <NextState>0</NextState>
  <Effect xsi:type="EffectExternalEvent" id="4">
    <Name>ExternalEvent (1)</Name>
    <Notes>Effect Event</Notes>
    <Timing>
      <Measure>4</Measure>
      <Beat>0</Beat>
      <Warmup>0</Warmup>
    </Timing>
    <OneUse>false</OneUse>
  </Effect>
</ExternalEvent>
<Duration>0</Duration>
</Effect>
<Effect xsi:type="StopExternalEvent" id="5">
  <Name>ExternalEvent (1) (1)</Name>
  <Notes>Stop Event</Notes>
  <Timing>
    <Measure>4</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </Timing>
  <OneUse>false</OneUse>
</Effect>
</ExternalEvent>
<ExternalEvent xsi:type="ContinueExternalEvent" id="1">
  <Name>ExternalEvent (1) (1)</Name>
  <Notes>Continue Event</Notes>
  <Timing>
    <Measure>4</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </Timing>
  <DefaultValue>0</DefaultValue>
</ExternalEvent>
<ExternalEvent xsi:type="EffectExternalEvent" id="2">
  <Name>ExternalEvent (2) (1)</Name>
  <Notes>Effect Event</Notes>
  <Timing>
    <Measure>4</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </Timing>
  <OneUse>false</OneUse>
  <Duration>0</Duration>
</ExternalEvent>
<ExternalEvent xsi:type="StopExternalEvent" id="3">
  <Name>ExternalEvent (3) (1)</Name>
  <Notes>Stop Event</Notes>
  <Timing>
    <Measure>4</Measure>
    <Beat>0</Beat>
    <Warmup>0</Warmup>
  </Timing>
  <OneUse>false</OneUse>
</ExternalEvent>
</ExternalEvent>
</Arrangement>
<SampleLibrary id="0">
  <Name>Accido demo library</Name>
<Notes>Folder</Notes>
<RootPath>C:/Accido demo library</RootPath>
</SampleLibrary>
</Project>
Glossary

.NET Framework[13]
Microsoft .NET Framework is a software component that is a part of Microsoft Windows operating systems. It has a large library of pre-coded solutions to common programming problems and manages the execution of programs written specifically for the framework. The .NET Framework is a key Microsoft offering and is intended to be used by most new applications created for the Windows platform.

Accessor
The accessor method, sometimes called a "getter", is most often used in object-oriented programming, in keeping with the principle of encapsulation. According to this principle, member variables of a class are made private to hide and protect them from other code, and can only be read by a public member function (the accessor method), which returns the value.

Access Modifier
A modifier which modifies from where an operation in a class can be accessed.

Arrangement
A project element in the adaptive music type which contains states. It can best be described as a music track on an audio CD. It is meant to represent a level in a game, although that is not set in stone.

Authoring tool[7]
An authoring tool is a software package which developers use to create and package content deliverable to end users.

C#[8]
C# is an object-oriented programming language developed by Microsoft as part of the .NET initiative and later approved as a standard by ECMA (ECMA-334) and ISO (ISO/IEC 23270). The C# language has a procedural, object-oriented syntax based on C++ and includes influences from aspects of several other programming languages (most notably Delphi and Java) with a particular emphasis on simplification.
Channel
A project element in the adaptive music type which contains voices. It has been added to group voices and to provide voice control by limiting the total number of simultaneously playing voices to one per channel.

Event
A project element in the adaptive music type. It can also be referred to as internal event. These events cause actions on state level. There are four different events with different functions. The voice event can start another voice, the channel event can start another channel, the effect event can trigger an effect and finally the export event can signal the game that the music has reached a certain point.

External event
A project element in the adaptive music type. External events represent actions which can be activated on trigger positions. These actions are external because these are the only events that can be called from the game. External events can cause state transitions, they can start an effect, stop a voice and they can change the values of constantly active events.

First person[9]
In video games, first person refers to a graphical perspective rendered from the viewpoint of the player character. In many cases, this may be the viewpoint from the cockpit of a vehicle. Many different genres have made use of first-person perspectives, ranging from adventure games to flight simulators. Perhaps the most notable genre to make use of this device is the first-person shooter, where the graphical perspective has an immense impact on gameplay.

GUI
Graphical User Interface (GUI) is a type of user interface which allows people to interact with a computer and computer-controlled devices. It presents graphical icons, visual indicators or special graphical elements called "widgets". Often the icons are used in conjunction with text, labels or text navigation to fully represent the information and actions available to a user. But instead of offering only text menus, or requiring typed commands, the actions are usually performed through direct manipulation of the graphical elements.

Horizontal sequencing[10]
This is the term used in this document to describe a music composition which is laid out horizontally. This means that every sample represents the music fully and the dynamic element is defined by the order these samples are placed in.
**MIDI**[11]

Musical Instrument Digital Interface (MIDI) is an industry-standard protocol that enables electronic musical instruments, computers, and other equipment to communicate, control, and synchronize with each other. MIDI does not transmit an audio signal; it simply transmits digital data "event messages" such as the pitch and intensity of musical notes to play, control signals for parameters such as volume, vibrato, and panning, cues, and clock signals to set the tempo. As an electronic protocol, it is notable for its success, both in its widespread adoption throughout the industry, and in remaining essentially unchanged in the face of technological developments since its introduction in 1983.

**Mutator**[12]

The mutator method, sometimes called a "setter", is most often used in object-oriented programming, in keeping with the principle of encapsulation. According to this principle, member variables of a class are made private to hide and protect them from other code, and can only be modified by a public member function (the mutator method), which takes the desired new value as a parameter, optionally validates it, and modifies the private member variable.

**Polyphony**

Traditionally polyphony is referring to the amount of notes a electronic instrument can sound at the same time. In this paper it usually represents the amount of samples a computer or gaming console can play at the same time.

**Project**

A project element in the adaptive music type which contains arrangements. It can best be described as an audio CD. It is the root element for the adaptive music type.

**Project Element**

A project element is an object where almost all other adaptive music type objects are derived from. It contains some meta data about the project element and defines the tree structure of the adaptive music type.

**Sample**

A sample is in this document a reference to a piece of sound data.

**State**

A project element in the adaptive music type which contains channels. It represents a piece of music in an arrangement. It is meant to represent a state in-game. This can vary from an emotional state of the player (happy, sad etc.), combat state (stealth, fighting etc.) or even the positional state of a player (which room a player is located in). States have features to transit into each other without damaging the musical coherency.
State Transition

Different states within the same arrangement can transit to each other. The adaptive music type provides features to do this as smooth as possible, the maintain the musical quality.

Third person

In video games, third person refers to a graphical perspective where it is aimed at the player character and is moving along with the character from a set distance.

Trigger

A project element in the adaptive music type. It is a place marker in a voice with links to external events. Triggers mark the places in a state where a certain external event’s actions can be activated.

Vertical orchestration[10].

This is the term used in this document to describe a music composition which is made out of layers (vertical as in stacked layers). This means that every sample represents a specific layer of the music, where the dynamics can be achieved by adding and removing layers.

Voice

A project element in the adaptive music type which contains a sample, events and triggers. It functions as wrapper for a sample. It positions the sample in time on the voice, meaning it defines a certain amount of time before and after the sample. It can contain a variety of different events which can effect the flow of the music in multiple ways. It can contain triggers which are marker points for state transitions.

Wall-to-wall music

This is a term used to describe that a usually a game has music from the beginning to the end of a level. The challenge we face here is repetitiveness in the music. This is what adaptive music tries to conquer.
Bibliography


List of Figures

3.1 Russian Squares. .............................................. 9
3.2 No one lives forever. ......................................... 9

4.1 A screenshot of an arranger from ableton live. Note the horizontal and vertical arrangement. ........................................ 12
4.2 An example of horizontal sequencing. ........................ 13
4.3 An example of vertical orchestration. In this case layer 1 is removed where layer 3 is added, where these layers could represent different (groups of) instruments. ........................................ 13

5.1 Creative’s ISACT. .................................................. 15
5.2 Audiokinetic’s Wwise. .......................................... 16
5.3 A screenshot of the kismet editor from the Unreal Engine website. ........................................ 17
5.4 Examples of how the new kismet nodes are often used. ........................................ 19
5.5 Clutter caused by implementing adaptive music in kismet. This is just one state. ........................................ 19

8.1 Architecture overview of existing adaptive music solutions. ........................................ 24
8.2 Proposed architecture; The sample playback is delegated to the game engine. ........................................ 25
8.3 The complete architecture of the new adaptive music system. ........................................ 27

9.1 Project structure. .................................................. 28
9.2 Samples. .......................................................... 29
9.3 State structure ..................................................... 29
9.4 State control ....................................................... 30
9.5 State transition .................................................... 31

10.1 Library design. ................................................... 34
10.2 Buffering need example. ...................................... 36

11.1 Accido with an example project loaded. ........................................ 39
11.2 Testing functionality of Accido, left the an overview of the available external events, right a state with one voice and sample playing. ........................................ 40

A.1 Class ............................................................. 44
A.2 Abstract Class ................................................... 45
A.3 Generalization .................................................... 45
A.4 Association: Aggregation ......................................... 45
A.5 Association: Composition ......................................... 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.6</td>
<td>Explanation of the arranger diagrams</td>
<td>46</td>
</tr>
<tr>
<td>B.1</td>
<td>Full data model tree structure</td>
<td>47</td>
</tr>
<tr>
<td>B.2</td>
<td>ProjectElement with all its children</td>
<td>48</td>
</tr>
<tr>
<td>B.3</td>
<td>ProbabilityItem</td>
<td>49</td>
</tr>
<tr>
<td>B.4</td>
<td>TimeItem</td>
<td>49</td>
</tr>
<tr>
<td>B.5</td>
<td>Overview of the Base package and how it relates</td>
<td>50</td>
</tr>
<tr>
<td>B.6</td>
<td>Visualization of two voice setups. The right example has a negative</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>start time</td>
<td></td>
</tr>
<tr>
<td>B.7</td>
<td>SampleLibrary package</td>
<td>53</td>
</tr>
<tr>
<td>B.8</td>
<td>Overview of the Event package and how it relates</td>
<td>54</td>
</tr>
<tr>
<td>B.9</td>
<td>Overview of the Trigger package and how it relates</td>
<td>55</td>
</tr>
<tr>
<td>B.10</td>
<td>Overview of the ExternalEvent package and how it relates</td>
<td>56</td>
</tr>
<tr>
<td>B.11</td>
<td>Overview of the Effect package and how it relates</td>
<td>58</td>
</tr>
<tr>
<td>B.12</td>
<td>Overview of the Curve package and how it relates</td>
<td>59</td>
</tr>
<tr>
<td>B.13</td>
<td>Full adaptive music data model</td>
<td>60</td>
</tr>
<tr>
<td>C.1</td>
<td>Classes involved in application to library communication</td>
<td>61</td>
</tr>
<tr>
<td>C.2</td>
<td>Classes involved in library to application communication</td>
<td>63</td>
</tr>
</tbody>
</table>