

HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

SCHOOL OF PHILOSOPHY

DEPARTMENT OF MUSIC STUDIES

GRADUATE PROGRAM

MUSIC TECHNOLOGY AND CONTEMPORARY PRACTICES

SPECIALIZATION:

MUSIC CREATION FOR NEW MEDIA

GRADUATE THESIS

Real-Time Adaptive Orchestration in Video Games

Ioannis S. Kaneris

Supervisor: **Areti Andreopoulou,** PhD, Assistant Professor

Music Supervisor: **Orestis Karamanlis,** PhD, Composer

ATHENS

NOVEMBER 2020

GRADUATE THESIS

Real-Time Adaptive Orchestration in Video Games

Ioannis S. Kaneris

A.M.: 18304

Evaluation committee:	**Areti Andreopoulou,** PhD, Assistant Professor
	Christina Anagnostopoulou, PhD, Associate Professor
	Anastasia Georgaki, PhD, Professor

Author's Note:

This essay was submitted in partial fulfillment of the requirements for the Master's degree of the Department of Music Studies of National and Kapodistrian University of Athens on September 2020. The author, Ioannis S. Kaneris, affirms that the content of the present thesis is the result of personal effort, and when third-party work has been used, the appropriate references have been made, following the rules of academic ethics.

The views presented on this thesis are those of the author, and not the supervising professor.

Abstract

This thesis discusses the history of music in video games, and how their non-linear nature calls for an adaptive orchestration approach, as well as how a linear medium like music can be reshaped into fitting the needs of video games. Furthermore, a basic analysis of the state of the art middleware software employed for this alteration is performed, as well as an application of all the research conducted in this paper, by designing the audio and composing the music for a fully functional video game level. The research on history and aesthetics was mainly bibliographical, while the study of aforementioned software was a combination of bibliographical, user's manuals, and hands-on research. The composition of the game's music and audio was based on personal aesthetic criteria heavily influenced by the research conducted beforehand for the sake of this paper. Adaptive orchestration in modern video games has raised the bar to the point of it constituting a new standard for scores and making middleware software an industry standard as well. As technology advances, these tools will continue to improve allowing, therefore, an even more tailored and spot-on adaptation of the music according to the players unique play style which makes video games even more immersive.

Keywords: video game music, adaptive orchestration, middleware software, FMOD, Wwise, ADX2

Table of Contents

Abstract 3
I. Short History of Video Game Platforms and Music 7
1.1 Video Game Consoles 7
1.2 Computers and Video Games 9
1.3 Conclusions 11
II. Adaptive Music in Video Games 12
2.1 The Issue of Linearity in Music 12
2.2 Game and Music – The Importance of Adaptability 12
2.3 Types of Video Game Music 14
2.4 Conclusions 14
III. State of the Art – Composing Real-Time Adaptive Music for Video Games 15
3.1 Pre-Composed Music 15
3.2 Generative Music 17
3.3 Hybrid Music 18
3.4 Conclusions 19
IV. Real-Time Adaptive Orchestration Software (Middleware) 20
4.1 FMOD 20
4.2 Audiokinetic Wwise 25
4.3 ADX2 31
4.4 Conclusions 34
V. Composing Adaptive Music for a Video Game – Application 35
5.1 Software 35
5.2 Sound Design 36
5.2.1 Ambience 37
5.2.2 Events 37
5.2.3 Character Sounds 38
5.3 Music Score 39
5.3.1 Aesthetics 39
5.3.2 FMOD 40
5.3.3 Parameters 40
5.3.4 Vertical Re-Sequencing 40
5.3.5 Horizontal Re-Sequencing 41
Conclusions 43
References 45
Manuals 46

Video games are one of the most popular forms of entertainment of our days, with enough genres and niches to appeal to practically anyone's taste. One can find a game to roam calmly and peacefully through a green meadow as a flower (*Flower* 2009), or fight an army of hell-spawn demons on Mars (*Doom 3* 2004). Between these two extremes, every single color of the spectrum can be found: games focusing on psychological drama (*Heavy Rain* 2010), fantasy games set in the medieval era (*The Witcher 3* 2015), sport-based games (*FIFA* 2019) or simpler classic games (*Super Mario Bros.* 1985)—to name but a few more game genres.

However, it is not the vast variety of genres that makes video games so appealing. What makes them special is their ability to interact with the player. Unlike movies or books, video games can take one's choices into account and present a different version of a story accordingly. Moreover, in order to progress through a game, the player must learn how to play it and earn the rest the story, which no other media can do.

Since video games respond to our decisions and actions in real-time, the element of variability is constant. Different players will have different approaches on the challenges that are set by the game. For example, some players may complete a task within a matter of minutes, whereas others may need several hours. In addition, in a lot of games, especially in modern ones, the story changes according to one's choices and actions. By taking into account both the differences in the game style and story choices of each player, it is highly unlikely that even two people would get an identical experience playing the same game. The game does not simply offer challenges and the ability to make choices for its outcome, but it is tailored to each individual.

Modern games in particular have come a very long way in making themselves as much detailed and dynamic as possible. From very carefully designed levels and artificial intelligence for the NPCs (Non Playable Characters) and environmental events, to masterfully created graphics creating beautiful imagery that keeps the player captivated and in awe while experiencing the game. The thrilling and emotional arcs of the characters the player meets and gets to help or harm also allow for further customization of the universe on the users' whim. Some games even go on to make use of psychoanalytic algorithms to predict a player's future choices and actions based on his past ones, so as to make the game more challenging or emotional for the specific individual (*The Evil Within* 2014).[1]

[1] The game world can transform during scripted events and as a result of player actions, by altering locations and creating new paths, or teleporting the player to new areas (McWhertor, 2014).

All these characteristics lead to the most iconic part of experiencing a video game, which is called *immersion:* a state in which the player feels as if they are really living the events presented to them. A lot of the game's elements contribute to creating this experience for the player, but the most significant component in successfully immersing a player is considered to be music. The ability of music to describe emotional changes and states, validate or contradict events and expectations, and most importantly, do all of these while playing quietly in the background, blending effortlessly into the events that unfold before us, and speaking to our subconscious, makes it an indispensable part of video games.

Naturally, if video games interact with our choices and change accordingly, music should do the same. Instead, for many years, music in video games had been linear, a simple playback, instead of adapting to the events in action, though. The reason for this compromise of the past was unavoidable due to the limitations of the technology. These days, however, the aforementioned limitations have been lifted, and we are now able to create music that can evolve and adapt to the flow of choices presented to the modern video game player. The methods and tools that have been and are being used, as well as an application of these methods are the topic of this thesis.

More specifically, the starting point of this endeavor is the history of video game consoles and gaming with home computers, including detailed references to the process of the evolution of audio. The next subject analyzed is that of the nature of adaptive music in video games, and why it is so paramount to successfully create music which evolves in real-time according to in-game stimuli. After the importance of adaptive music in video games has been established, the next chapter will present the various techniques that are currently being employed in order to create adaptive audio, and debate upon the pros and cons of each individual technique.

Moving on to more practical subjects, a way of implementing this theory is offered through the discussion of some of the most common middleware's capabilities and functions. The basic architecture and utilities of Firelight Technology's FMOD, Audiokinetic's Wwise, and CRI Middleware's ADX2 are presented, with the intent to give the reader a basic understanding of what these middleware have to offer to today's video game music composer.

Finally, an application of all the theory of this thesis is presented in detail through the implementation of audio on a fully functional video game level, with every aspect of the game's sound design and music having been composed and programmed to react to real-time adaptive parameters by myself.

I. Short History of Video Game Platforms and Music

A short history of video games and their relation to music will be presented in this chapter, with the intention of leading to a better understanding of the technological limitations of the past, as well as the reasons why an adaptive music score was not introduced simultaneously with them.

1.1 Video Game Consoles

Video games made their commercial appearance in the 1970s, with the first video game accompanied by sound being *Computer Space* (1971). The audio of the game consisted of sounds designed to emulate a space battle. The first video game to have made a commercial success, however, was *Pong* (1972). *Pong* made a substantial impact on the market, leading to a major boost in the number of companies that made video games (Collins, 2008). While *Pong*, as well as *Computer space,* had merely beeps as sounds, which are far from what one would call music, the importance of adding sound to a video game for the first time should be noted, nonetheless.

In 1979, the first step towards creating actual music for video games was made: the Mattels Intellivision System, whose sound generator chip was capable of producing three-part harmonies. In the following years, there were many advancements in audio for video games made by various companies such as Atari[2], Nintendo[3], Sega[4] and NEC[5] (Marks, 2009). All these systems, however, were limited to what is known as an *8-bit* sound: a sound characterized by its unnaturalness and harshness.

The famous 8-bit sound, also known as chiptune, refers to the audio synthesized through an actual computer chip, rather than musical instruments. This procedure was extremely complicated, and greatly limited the potential for musical creation, since it was more of a programming challenge than a compositional one. Moreover, since all of the game's audio had to come from the same chips that the console used to produce all of the other elements of the game (graphics, controls, etc.), the memory space available for music and sound effects was very limited, repeatedly leading to the gradual removal of music parts for the sake of functionality (Collins, 2008).

[2] Atari *Pokey* in 5200 platform (1982) was a dedicated chip able to produce sound through a four-piece band which controlled the pitch, volume and distortion of each individual channel (Marks, 2009).
[3] Nintendo's NES platform (1985) used 5 monophonic channels (Collins, 2008).
[4] Sega (1986) used monophonic generators capable of generating sound within a 4 octave range (Marks, 2009).
[5] NEC turbo Grafx (1989) supported 6 voices with stereo output (Marks, 2009).

Some iconic video games famous for their 8-bit soundtracks are *Super Mario Bros*. for the NES (1985), *The Legend of Zelda* for the FDS and NES (1986), *Castlevania II: Simon's Quest* for the FDS (1987), and *DuckTales* for the NES (1989). The 8-bit sound, despite its harshness, has made a significant impact on the gaming community, which resulted in many gaming companies creating soundtracks in the style of 8-bit music to this very day. Without the technological limitations of the past, chiptune music can be used easily on its own, or in combination with actual musical instruments and synthesizers, as it happens, for example, in the video game *Terraria* (2011).

It is also worth mentioning that until 1981 all video game music was composed of a specific loop which would repeat itself over and over again. The first game to ever introduce music that was not looped was *Frogger* (1981), which featured eleven distinct gameplay songs. In order to achieve this, the songs had to last 30 seconds, since that was the time a player would have either died or arrived at a safe house (Fritsch, 2013).

The leap from the 8-bit sound to a more natural and musical sound came with the introduction of the MIDI (Musical Instrument Digital Interface) protocol in the mid 1980's (Young, 2012). The significance of this protocol is to be attributed to many different factors. First of all, it is a universal protocol, and therefore can be used by a synthesizer, as well as a video game console, with no changes. Composers no longer had to code—or hire someone to code—their music into a video game console; they just had to compose it in the same way they would with a synthesizer, and then export it. Furthermore, the MIDI protocol supports 128 different sounds and a 16 voice polyphony, which makes it a perfect tool for composing actual music. Its sound may still be far from fooling the listener that what they hear is actual musical instruments, but the improvement is undeniable. Some of the most famous video games that made use of the MIDI protocol for their soundtrack are *Wolfenstein* for PC-DOS (1991), *Doom* for MS-DOS (1993), *Duke Nukem* 3D for PS-DOS (1995) and *Final Fantasy VII* for PlayStation (1997) and MS Windows (1998).

From the 1990s onwards, the advancements in technology allowed the quality of video game music to reach that of commercial music. Sega was the first company to feature a console capable of CD-quality (44.100Hz, 16-bit) soundtracks with the Sega Genesis add-on Sega CD (Wolf, 2007). This technological breakthrough green-lit the production of pre-recorded soundtracks of actual human performances, creating, thus, an unprecedented level of musicality in video game scores.[6]

[6] The first video game to achieve this is *Sonic*, with the release of *Sonic CD* in 1993 (Young, 2012).

The impact of the use of CDs was so great that, from this point on, almost all video game consoles were built around the potential of this technology. The most important consoles of the 90s built around the CD technology are Sega's Saturn (1995) and Sony's PlayStation (1995). Another major console of this era was Nintendo's Nintendo 64 (1996), which relied on the cartridge technology instead of CDs. All of these consoles came with improved sound chips, which permitted the use of both MIDI internal synthesizers and pre-recorded music files: an important improvement due to the high storage cost of pre-recorded music tracks (Young, 2012). Nintendo 64 was able to partially solve this problem by making use of the *.mp3* format.[7] From this point on, video game music rose up to the level of commercial music, with soundtracks such as *Sonic CD* for SEGA CD (1993), *MediEvil* for PlayStation (1998), *Metal Gear Solid* for Playstation (1998) praised for the quality of their music, and their companies selling the game soundtracks as stand-alone commercial CDs.

At the dawn of the 21st century, with the exception of Sega Dreamcast introduced in 1998, 128-bit consoles, like Sony's PlayStation 2 (2000), Nintendo's GameCube (2001) and Microsoft's Xbox (2001) became the new state of the art. Due to the improvements to the sound chips, storage space and processors of these consoles, video game companies oriented themselves to the concept of adaptive music. After all, these consoles were able to support sound better than the quality of CDs (Wolf, 2007). The next generation of consoles—Sony's PlayStation 3 (2006), Nintendo's Wii (2005) and Microsoft's Xbox 360 (2006)—featured even greater audio capacities, which included audio processors with DSP effects and increased audio channels (Young, 2012). It is in this generation of consoles that the broader public familiarized themselves with dynamic or adaptive video game soundtracks, with games such as *Spore* (2008). The current generation of consoles consists of Sony's PlayStation 4 (2013) Microsoft's Xbox One (2013), and Nintendo's Switch (2017). This generation brings an even greater quality of sound, and primarily, far greater processors capable of supporting 3D-audio, going hand in hand with the consoles' ability of Virtual Reality video games.

1.2 Computers and Video Games

Throughout all these years, the home computer has been following the technological advancements side by side with the game consoles of each era, and has been a way for people to play video games since 1977, when three separate computers were released to the market: the Commodore PET, the TRS-80 and the Apple II. All three made use of a monitor, a

[7] The .mp3 (MPEG Audio Layer 3) is a digital audio codec that compresses sound files using a *lossy* compression method, which allows the minimization of storage required at the cost of audio quality. For more information on the MP3 format, see: *MP3: The Meaning of a Format* by Jonathan Sterne (2012), Duke University Press.

keyboard, a motherboard and a cassette storage unit (Chandler, 2005). Even though each system had its own characteristics and functions, the Apple II was a bit more advanced, as it featured multiple expansion slots allowing future upgrades, and, most importantly, came with a colorful display (Wolf, 2007).

All of these systems featured video games ranging from text-based games and emulations of public arcade games to brand-new original video games, created specifically for the new systems (Fritsch, 2013). Text-based games such as *Time Traveler* (1980), *Eliminator* (1981) and *Dungeons of Daggorath* (1982) were the only type of available video games for the TRS-80, due to the limitations of its graphics, which supported only ASCII characters. The PET and the Apple II also featured arcade emulations, such as *Space Invaders* and *Pac-Man* (1980), and other graphic-based video games, such as *The Wizard's Castle* (1980), *Time Trek* (1979) for Commodore's PET, *Buck Rogers: Planet of Zoom* (1982), and *Choplifter* (1982) for the Apple II.

With the development of the floppy disk drive, the explosion of video games, in both legal and illegal forms, was at its start. All of the above systems were able to be upgraded to feature a floppy disk drive, making, therefore, floppy disks the new standard. This led to a new era of sharing software amongst computer users which gave a boost to the piracy of video games and other programs (Wolf, 2007).

Many other systems were introduced in the following years, nevertheless, the ones that created the trends in personal computers as we know them were IBM's Personal Computer—or PC—in 1981, and Apple's Macintosh in 1984. IBM's PC made use of the DOS operating system, which constituted a platform for many video games, while Apple's Macintosh was the first computer ever to have featured a mouse (Hey & Papay, 2014). The introduction of the mouse was a breakthrough for video games as it gave users the ability to directly point and click objects on screen. Some of the games introduced for the Apple Macintosh were *Balance of Power* (1985), *Dark Castle* (1986) and *Crystal Quest* (1987).

Another landmark for home computer video gaming was crafted in 1985, with the introduction of the Commodore Amiga, whose soundchip, Paula, was able to use DSP in the form of sampling for the very first time in home computer history. This enabled the computer to play back short samples of pre-recorded sound waves (Fritsch, 2013). From this point on, the concept of *computer system* started to lose its original meaning. Nowadays, anyone can create their own home computer from scratch, or choose their own operating system, motherboards, and so on. Every year, the specs of each individual part rise to new heights, achieving better graphics, audio and more complex games through more advanced processors.

1.3 Conclusions

The main reason video games were not introduced along with an adaptive score complementing their non-linear nature was mainly due to the technological limitations of the past. Nowadays, however, these limitations have been lifted for the most part, and video game developers are using adaptive scores for their titles.

II. Adaptive Music in Video Games

With the alleviation of past technological limitations, video games began to make use of adaptive music scores more often than not. The issues that a music composer had to face now were not technological in nature, but aesthetic and practical ones. In this chapter, we are going to present the most usual and significant issues, as well as their possible solutions.

2.1 The Issue of Linearity in Music

For the vast majority of their existence, video games had to rely on the model set by composers of film music in order to build a music experience for the player. Video games and films are, however, fundamentally different, since films are a linear form of art, whereas video games are characterized by their non-linear and interactive nature (Liebe, 2013). By taking the aforementioned nature of video games into consideration, we are left with one option that truly solves the problem: the music of a video game must be interactive, therefore adapt to the events and actions taking place.

Creating such music is not an easy task, though. Music, as a form of art, is also linear; it has a beginning, a middle and an ending. A musical piece cannot be cut and pasted according to one's wishes, because in doing so, the coherency and flow of music is at risk (Berndt & Hartmann, 2007). One of the first attempts to solve this problem was the use of looped music: music composed in such a way that its end is able to transition into its beginning without the listener noticing. This looping can repeat itself practically forever, allowing the events of the level to unravel, no matter how long it takes the player to complete the task. In games like *Space Invaders* (1978), the music changes its tempo in response to the progress of the player, going faster until the level is completed. In other games, such as *Plants vs. Zombies* (2009), there is a variation of the looping mindset, which consists of a looping song introducing new elements, such as instruments and variations, while the game progresses. Using this technique, the game keeps the player captivated and gives a sense of progress (Abreu, 2018).

Such music, however, cannot be called adaptive, because it is unable to change quickly or flexibly enough to properly highlight the vast and complex thematic and emotional changes that a game may present. True adaptive music must be able to respond real-time in accordance with the events that take place in the game; it must possess a significant amount of flexibility without damaging its ability to evoke emotion and highlight changes (Young, 2012).

2.2 Game and Music – The Importance of Adaptability

When speaking of adaptive music in video games, we have to think carefully about the reasons why the music needs to be able to change in moment's notice. Only by understanding

the motives of adaptive music, one can effectively compose music that corresponds to the needs of a video game. A primary function of the music in a video game is that of narrating a story, reinforcing, and bringing forward the deeper meaning of what we are experiencing (Berndt & Hartmann, 2007). The music of a specific scene may support and bring forward the emotional situation projected to us, or it may counter it, bringing, for an example, a sinister overtone to a seemingly friendly interaction, making, therefore, the player concerned about what lies ahead.

Another way in which music can narrate the events of a video game is by making use of *leitmotifs*. The leitmotif—a specific musical motif tied to a certain character—is a concept that is closely tied to the name of Richard Wagner (1813-1883), who made it popular by using it extensively in his operatic works. The distinctive characteristic of this motif can be harmonic, melodic, or rhythmic in nature, but it may also combine more than one distinctive elements. Throughout different musical parts, if the main unique characteristic of the motif reappears, regardless of the potential transformations that have happened to the rest of its elements, the reference to the specific person tied to the leitmotif will be apparent (Bribitzer-Stull, 2017). By making use of a leitmotif database for the characters in a game, we can achieve continuity in music that changes constantly to the players actions, even in situations when doing so would otherwise appear challenging, to say the least—like in computer-generated music. Even though the leitmotif is by no means an innovation in music creation, it is an invaluable tool in the world of adaptive video game music.

The significance of music able to adapt real-time in a video game is not limited to the vital function of narration; video games provide challenges and choices for the players, too. Said challenges may appear with or without warning, changing the *mode* of a game or what is referred to as a *state*. While experiencing a game, the player will transition between peaceful and hostile situations—a change that could come at any moment. Such an event constitutes a different game state that changes the way the controls work by trading the possible peaceful dialogue interaction options with different attacks and use of weapons (Young, 2012). Apart from the way the player interacts with the environment, another difference in these modes is the psychological state of the player. The player will transition from a mental state of safety to a fight-for-your-life mentality; a change that must be complimented by the music, which empowers and inspires the player for victory, or even tries to intimidate them, in order to highlight the gravity of the situation.

Even within a singular state, there are constant changes that take place and must be taken into account for the evolution of music. The progress that a player has made throughout the level, the interaction with characters and objects, and the validity of the players' actions are

just some of the variables that can change in a peaceful state. On the other hand, while in combat mode, the music may have to change depending on the number and/or the strength of enemies left, the victory progress, the health points of a character, a special move, or a vast number of other variables. The different strategies and actions that a player may tap into can stack up very quickly, and the music must be able to properly highlight these changes so as to make the state of immersion possible (Young, 2012). Consequently, a quick and accurate adaptation of the music is paramount.

Apart from the music swayed by the players, it could also work the other way round. Music can also encourage the player to take certain actions. For example, a very intense and suspicious soundtrack is more likely to convince a player to take the aggressive option over the peaceful one in a possible dialogue (Peerdeman, 2010). Apart from that, it may also respond to the player's decision in accordance with its morality, validity or possible consequences (Berndt & Hartmann, 2007). Such musical choices could easily derive from data the game has collected from the player's prior behavior, subsequently creating a bidirectional dynamic correlation between said player and the music of the game.

2.3 Types of Video Game Music

In a video game, therefore, we can employ three main types of music complimenting each other with its own merits and uses, and producing a complete video game soundtrack; these music types are usually used together in a video game (Liebe, 2013). These three types are:

- *Linear music*, which cannot be affected in any way by the player's actions and choices. This type of music is commonly used in loading screens, cut-scenes, and menu screens. Linear music may change according to milestones in the game's progress, or with different levels.
- *Reactive or Adaptive music*, which is affected directly by the player's actions—the main subject of this thesis.
- *Proactive music*, which suggests certain actions and choices to the player.

2.4 Conclusions

Using adaptive scores in video games creates a more intense experience for the player by enhancing the state of immersion, and even though music, as a media, is traditionally linear, there are actions that can be taken to overcome this linearity. In the following chapter, we will examine the most commonly used manners in which we can create an adaptive music score.

III. State of the Art – Composing Real-Time Adaptive Music for Video Games

The main approaches employed as far as composing adaptive music is concerned can be categorized as such:

- *Pre-composed Music* (pre-composed material that does not stay static and linear, but can be re-arranged in real-time).
- *Generative Music* (music that composes itself real-time).
- *Hybrid Music* (a combination of pre-composed and generative music).

Each category will be analyzed separately in the following sections, with comments on the strengths and weaknesses of each.

3.1 Pre-Composed Music

Pre-composed music is the most familiar and popular method when it comes to composing adaptive music for video games. This comes as no surprise, since pre-composed music is high-quality and high-effort music that can be composed more or less with the traditional methods composers over the years are used to. The backside of using pre-composed music comes from the rather limited flexibility it offers (Berndt & Hartmann, 2007).That does not mean, however, that pre-composed music has to be linear and unchanging. There are a number of techniques that can be used to make a pre-composed musical piece adaptable to a variety of situations.

First of all, the expression of the performance can be manipulated. By keeping the structure of the composition unchanged, we can still interfere with the expressive character style. We can therefore change the tempo, the articulations, the reverberation, the pitch, the timbre, and many more performance parameters of the track, depending on a number of possible *triggers* that can be tied to each individual, or group, of parameters. This technique is the least commonly used, due to its simplistic and not very flexible nature. It is also the very same technique used in the game Space Invaders, in which the game's 4-note music sequence would get faster as the player progressed through the level (Fritsch, 2013). For that time, this technique was nothing less than revolutionary, but by today's standards, it is primitive to say the least.

In order to use expressive performance manipulation and have a decent quality of sound, a midi track connected to a sample library must be used for the piece. Recorded music files, such as .wav and .mp3, are not feasible source material, because, due to the nature of the

formats, it is impossible to interfere with the aforementioned performance parameters—at least in a way which retains the audio quality to a decent level. The quality of sample libraries, however, has improved phenomenally in the latest years, resulting in commercial level music also used in film scoring (Berndt & Hartmann, 2007).

Another approach to the matter in hand is that of *sequential arrangement* or *horizontal re-sequencing*. This technique has actually been used in the past, in the form of a musical game even Mozart himself made use of called *dice game.*[8] Since the tracks are pre-composed and will not be modified in terms of internal performance, we have the luxury of making use of actual recorded instruments, which is a huge advantage in the quality aspect. The adaptability of this method is a result of a carefully composed list of tracks able to succeed one another at certain points. In other words, the tracks must be compatible in terms of harmony, melody and rhythmic meter aspects for the flow of music to remain incessant. The triggers for such a succession could be a change in the game state (peace, battle), the arrival in a new area, the use of a special move-set, and so on. For the change to be successful, one must set marker points in the musical tracks, so that there are no abrupt fluctuations to disturb the flow of music; the tracks can be tempo-synced for a more musically coherent result, or un-synced for a more abrupt effect. One could even use a transitional score to make the transition from one track to another even more subtle, if they so please (Lassi, 2018). The iMuse engine is a great example of a great application of sequential arrangement, since due to its arrangement of music, it can make use of jump marks at any position of the musical piece without the use of crossfades that is common in such arrangements (Strank, 2013). Another way of making such transitions is by tapping into audio middleware like FMOD*,* Wwise, or ADX2.

In addition to horizontal re-sequencing, there is the technique of *vertical remixing* (also called *re-orchestration* or *layering*), in which the musical track is divided into layers called *stems*. By combining different stems or layers on top of a base layer, we can create a vast amount of musical combinations and variations on the same track, avoiding, thus, repetition. Musical instruments may come and go, blend in seamlessly and subtly, this way underlining the events of the game, creating the appropriate mood, and contributing to the immersion of the player. Connecting certain game elements to certain musical instruments can be very helpful for achieving immersion; for example, the appearance of a certain character could be

[8] In dice games the composer would write music for a specific number of bars; those fragments of music were carefully composed in order to be able to connect with each other in a musical manner. After their composition the "players" would roll dice in order to decide which of the bars would succeed the previous one, and so on (Hedges, 2015)

tied to a specific musical instrument, or a battle can be underlined by the introduction of a strong rhythmic element. Of course, for such a concept to function, all of the game's music material must be composed in one tonality and tempo, so that they may come and go at any given point. Making sudden and unexpected changes, therefore, is out of the question. Another possible disadvantage in making use of this technique is the timing of the additions or subtractions; if, for example, an important theme is played by a guitar, but the player moves away from the parameter causing it, the resulting interruption of the melody can come across as unnatural and impact the immersion (Lassi, 2018).

As a result, one may wonder which technique is better. The answer is that both approaches have their strengths and are used together in most games. Vertical remixing can be emotionally ambiguous, since the sonic result is largely based on the triggers of the game engine which can change in real-time the harmony, rhythmic elements, and orchestration—directly swaying, therefore, the player's emotions according to the situation. On the other hand, horizontal re-sequencing, with its ability to transition between different musical sections at any time, gives us the capacity for sudden and unexpected changes. For these reasons, horizontal re-orchestration is typically used for transitions amongst events of significance like boss battles and changes in game state, when the player must feel that a noteworthy change has occurred. Vertical re-orchestration is utilized in more delicate and slow changes in the game, when the player should not be made directly aware of the event taking place, although they must still be able to feel that something is progressing, such as an emotional state or the way to victory (Young, 2012).

3.2 Generative Music

Generative music (sometimes also called *procedural music*) is a score that is composed and produced in real-time, by a set of algorithms. Based on a number of templates given a priori, these algorithms produce on-the-spot music with the parameters of the game as an input (Collins, 2009). The great advantage of this type of adaptive music is its unparalleled potential of flexibility; the music can change instantly in response to the events taking place without concerns for continuity, since everything is being composed in real-time. This type of adaptive music is the least used in video games, mainly due to aesthetic and recourse management concerns (Young, 2012).

There are a number of techniques a composer may utilize in order to compose a generative music concept. One of them is the use of different algorithms for different emotional states of the game. The changes on the musical mood can be triggered by assigned parameters

which will send the signal to the engine to choose a different algorithm for the music production. An obvious problem that arises in this situation is that of musical coherency, since the transition between the two algorithms will need a transitional score, which, in turn, takes away the instantaneous character we want. The potential of this approach remains great nonetheless, and it should be taken into account (Berndt & Hartmann, 2007).

Another approach is that of a single algorithm that may change the harmonies, the melodies, and practically every aspect of a musical soundtrack to a great degree. Such an algorithm is that of the game *Spore*, programmed in the PureData environment. In *Spore* (2008), the music changes constantly in subtle ways according to the player's decisions when it comes to creating creatures, cities, civilizations or every other step of the game's evolution progress. Rhythmic elements, melodies and ambient features evolve, or come and go as the game progresses, avoiding, thus, the problem of monotony, while simultaneously underlining the player's choices. Spore, in particular, even gives players the ability to choose some of these musical elements, giving them the ability to directly alter the algorithm to their liking (Young, 2012).

Although the potential of generative music is immense, some fundamental drawbacks have to be taken into account. The main problem with generative music systems is their inability to properly comprehend emotional and meta-structural features that lie outside the basic algorithmic aspect of music. It is not enough to have the right harmony or contrapuntal features in order to express certain emotions, like happiness; there is a greater emotional intelligence that resides in great musical pieces, which generative music systems are not quite able to fully mimic yet (Berndt & Hartmann, 2007). Another disadvantage of these systems is their great computational demand increasing the recourse consumption of the video game and discouraging game developing companies from using them (Collins, 2009).

3.3 Hybrid Music

By the term hybrid music we refer to a combination of pre-composed and generative music. Conjoining the two composing styles for real-time adaptive music in video games gives us the ability to keep the best characteristics of each style, while disregarding the cons that come with them; we get to keep the high musical quality of pre-composed music, and the great versatility of generative music, without the limited adaptiveness of the former and weak musical coherency of the latter (Berndt & Hartmann, 2007).

In order to achieve a music piece that can evolve in real-time in accordance with the events of the game, and retain musical coherency, we must first contemplate on some basic

points. First of all, the adaptations that will take place should disturb the structure as little as possible for the transitions to feel natural and connect to their destinations in a musically valid manner. Moreover, the adaptations should occur at certain boundary points, in order to avoid a motif, or theme changing before we have even heard its original form. Harmonic and melodic changes should take place in points such as cadences, where new beginnings are musically probable, while metric changes should be used only at structural boundary points, or when leading to a target (Berndt & Hartmann, 2007).

The way hybrid music is implemented into video games is usually via the use of middleware software such as FMOD, Wwise, and ADX2. Even though a detailed analysis on such software will follow in the next chapter, I would like to highlight a few of their basic characteristics here. Especially orientated for the needs of real-time adaptations, middleware are essentially DAWs for composing music for video games. While some functions like vertical orchestration could easily be programmed in a DAW, others, such as horizontal orchestration, are problematic. Furthermore, with the use of middleware, we can have a main *base* track playing, while the actions of the player trigger the middleware to choose new audio files from pre-arranged categories, creating, as such, an automatized horizontal orchestration.

3.4 Conclusions

The more technology advances, the more generative music becomes a promising method of creating an endlessly adaptive score. Until then, however, using pre-composed or hybrid music for an adaptive score remains the dominant choice.

IV. Real-Time Adaptive Orchestration Software (Middleware)

The intent of this chapter is to present the basic elements and functions of some of the most commonly used middleware and fully-functioning engines used for the creation of real-time adaptive music—and not only—for video games.

Middleware programs are software composers utilize to create adaptive music without having to go through the ordeal of coding or scripting, and make the whole process more accessible, while lacking the vast limitations of DAWs when dealing with adaptive video game music. The musical material a composer has composed can be imported into a middleware program, where it can have its parts organized. Once imported and organized, the middleware program gives the option for triggers to be set, which will activate as soon as the corresponding data is set from the game engine middleware work with—hence the name middleware (Lassi, 2018)—resulting in playback of said musical parts. Following the same principles, middleware programs are also used for the implementation of sound design audio. Middleware are usually composed of a software part similar to a DAW, in terms of interaction with the user, and an API—an Application Program Interface—which is used for the more technical aspect of integrating the music to the game engine.

4.1 FMOD

Developed by Firelight Technologies in 1995 under the name Firelight MOD player, FMOD is one of the first audio engines to be introduced to the market. It was not until 2013, however, with the version FMOD 4.0 that an editor tool was included and FMOD became FMOD Studio version 1.0, signaling the era of its use as a middleware (Somber, 2016). Nowadays, the FMOD software consists of two distinctive technologies that work in unison: the *FMOD Studio* (version 2.1) and the *FMOD API* (version 2.01). Two individual parts comprise the FMOD API: the *FMOD Studio API* and the *FMOD Core API*. When referring to the collective of FMOD Studio and the FMOD Studio API, the term used is that of *FMOD Studio Suite*, while the term for the FMOD Studio API and the FMOD Core API is FMOD API (FMOD, 2020a). Some iconic video games that make use of FMOD are: *Batman Arkham Asylum* (2009), *Bioshock 2* (2010), *Dark Souls 3* (2016), and *Deus Ex: Mankind Divided* (2016).

FMOD Studio. FMOD Studio is the part of the FMOD Studio suite in which the user can manually create music. The main concepts of the FMOD Studio are *events, tracks, instruments, parameters and cues, effects, routing and mixing,* and *banks.*

Events. The first step in forming a project with FMOD Studio is creating an *event*. An event is a sound content unit that can be triggered and manipulated through the game code. Every situation that calls for a sound produced in a game should have a matching event. Events contain and are composed of tracks, instruments, and parameters. The different events that will be created throughout the project can be organized in the *events browser* (see figure 1).

Each event can have multiple instances of itself playing at the same time, allowing for a single event to be used for multiple in-game events. An example of this would be a gun being fired multiple times; in this occasion, each time a new shot is fired in-game, a new instance of the gunshot event would be created and played. Moreover, in situations where randomization or any other type of variation is needed, an event allows for each single instance of itself to have unique parameter values (FMOD, n.d. a).

Tracks. Tracks inside an event can be distinguished to *audio tracks* and *return tracks*. Audio tracks function as group buses, creating a submix from the outputs of instruments of the audio track and other group tracks routed to it, whereas return tracks function according to their name and are the receiving end for sends on other tracks. Finally, all the tracks of the event are routed into the events master track, and then to the *project mixer.* The main function of the tracks is the creation of a small mix, conferring each event instance its own characteristics (FMOD, 2020a).

Effects. FMOD Studio offers a large variety of effects that can be inserted at any point in a project's signal chain. The parameters of these effects can be adjusted in real-time as the game evolves and the in-game circumstances change. The way these effects work is no different from that of a standard DAW, featuring pre-set settings, as well as the option for users to save their own pre-sets (see figure 1) (FMOD, n.d. a).

Instruments. Instruments are audio-file-based triggerable elements that route audio content in the tracks of an event. The way the instruments are utilized is by being assigned on a specific *parameter sheet* which is able to trigger it. Each instrument has a range of parameter values that represent a trigger region for the particular instrument, and when the parameter sheet's playback position begins to overlap the instrument's trigger region, then the instrument is triggered, and will remain in this state until the playback position ceases to overlap the trigger region.

Instruments throughout the project can be grouped into *playlists*, which may also contain other project events (FMOD, n.d. a).

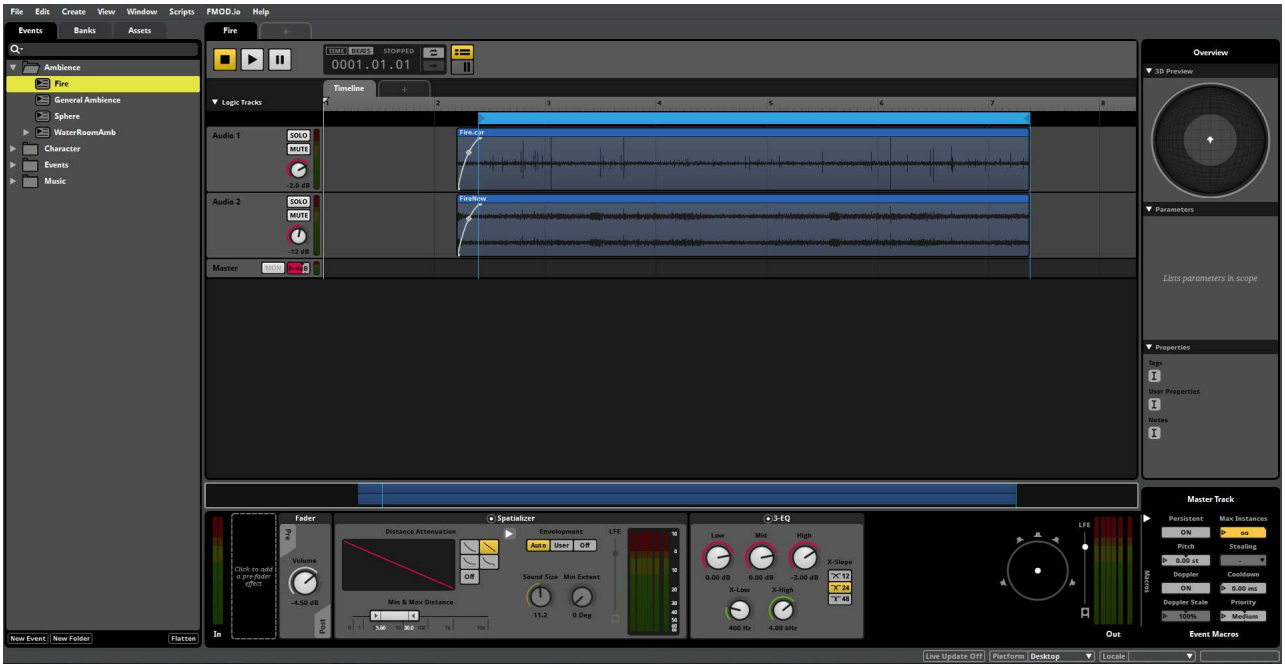


Figure 1. FMOD Events and Effects. The main interface of FMOD Studio, with the event browser on the left, the audio tracks in the center, and the available effects and their respective parameters on the bottom.

Parameters and Cues. Parameters and cues are the two main methods to change an event real-time. The more common of these two are the parameters, which can be used to trigger instruments, or automate the values of event properties. The most fundamental parameter is the *Timeline,* which is nothing more than a linear increasing value based on time. The rest of the parameters are known as *game parameters*, because they are representative of the game state.

To further change parameters, the user may use the FMOD Studio API to change the game's code, or can set *modulators*, or even construct events that reference the event containing the parameter (see figure 2). Cues, on the other hand, tell the timeline *playback position* of an event instance to move past the next *sustain point* (FMOD, n.d. a).[9]

[9] Sustain points are a type of logic marker, that change how the timeline playback position moves (FMOD 2020 a).

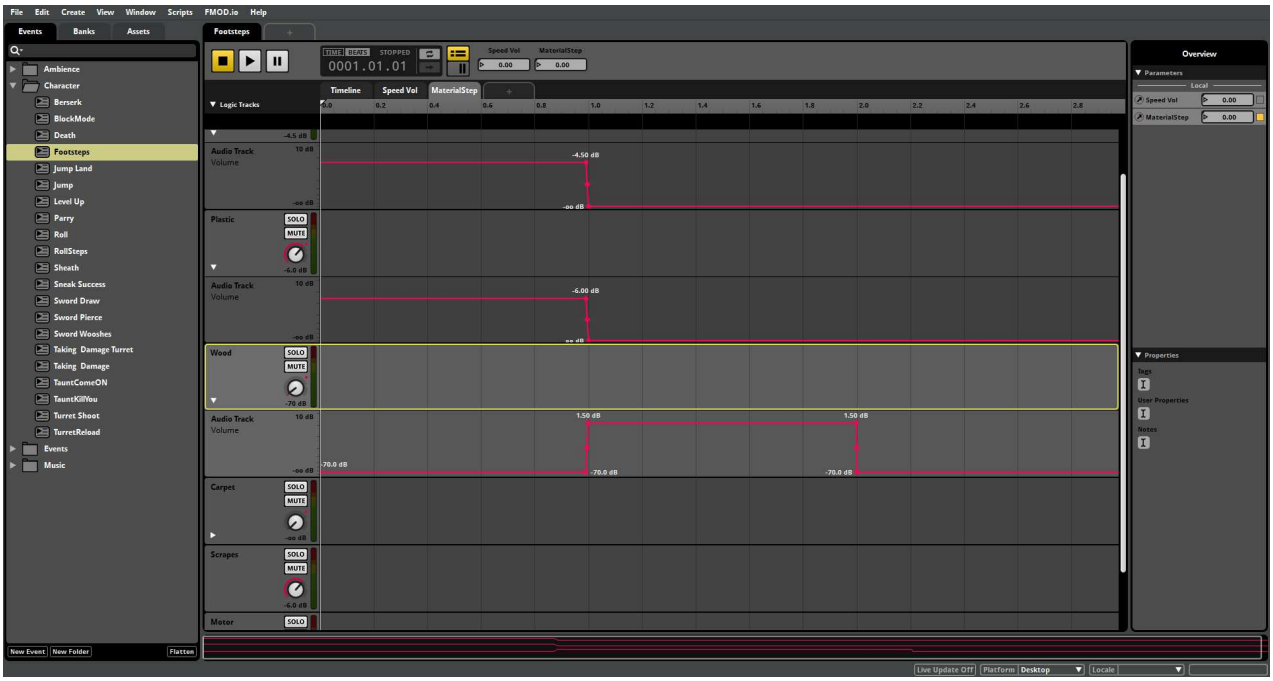


Figure 2. FMOD Parameters. In this picture, we can see some of the game parameters in FMOD, and how they can be set to interact with certain values.

Routing and Mixing. The routing in FMOD Studio is very similar to that of DAWs, as it consists of buses that transfer the sound, allowing, thus, the user to customize their routing to add effects and returns, or to group them into *group buses* (see figure 3). The user can then mix them in a fashion identical to mixing a standard DAW project (FMOD, n.d. a).

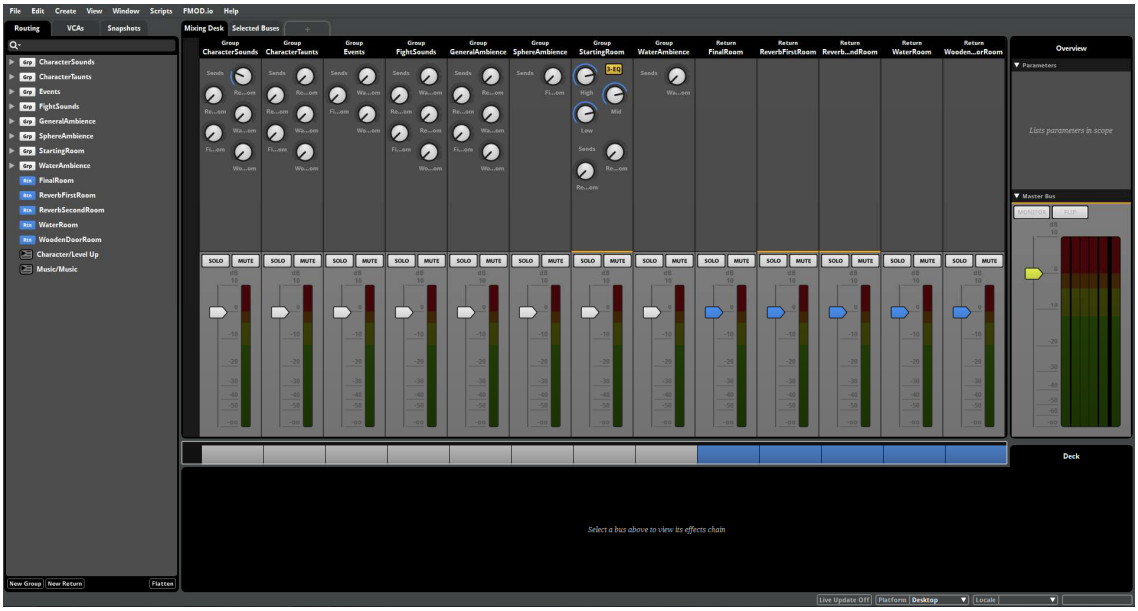


Figure 3. FMOD Routing and Mixing Panel of FMOD, along with the automation parameters (top of the first 8 tracks).

Banks. Banks are the collections of the events of a FMOD Studio project which have been formatted and compressed in way suitable to be used in a game. The purpose of a bank is to allow a dynamic choice of the content loaded into the game, this way optimizing the use of memory from the game. For an event to be used in a game, it has to be assigned to at least one bank first.

Before the game can make use of any banks, it must first load at least one *master bank*. A master bank is a bank which contains data related to the entire FMOD Studio project. Since a master bank will be loaded into memory at all times, it is advisable to keep it as small as possible (FMOD, n.d. a).

FMOD API. FMOD's API can be programmed in C, C++, C#, or Java Script programming languages (FMOD, n.d. b).

Studio API. The FMOD Studio API acts as a means to load the bank files created in FMOD Studio, and play them into the FMOD Engine. Then, the FMOD Engine can be integrated into the game and allows the use of adaptive audio (FMOD, n.d. a). The FMOD Studio API is meant to be used by programmers so as to interact with the data driven projects that have been created through the FMOD Studio at run time, and is built on top of the Core API, providing, therefore, further functionality than the one the Core API offers (FMOD, n.d. b).

Core API. The FMOD Core API is a standalone programmer API, with no requirements for any sound designer tools to interface with. It is tasked with the coverage of the basics of sound, such as *channels, channel groups, sounds, sound groups, DSP, recording,* and *3D sound and occlusion*. All of its features are implemented in code by a programmer (FMOD, n.d. b).

Using FMOD for Sound Design. In *Dark Souls 3,* FMOD is used mainly for the sound design of the video game, since the lack of level music in most of the game is one of its main characteristics as a title. Using the information presented above, we will go through a mock process of creating the sound design for this particular video game.

For starters, an audio event will be created for each separate in-game sound event, and then will be tied to an in-game trigger. This way, each time the character is hit, for example, the corresponding taking-damage audio event will be triggered. Multiple audio tracks will comprise this audio, since the event of taking damage should trigger an audio cue composed of a groan and an impact sound.

These sounds, however, should also change according to the environment the player is in—for example, a cave should have way more reverb than an open field—and this is where parameters come in use. We could create a collection of different reverb effects to match the

different areas in the game by using the effects that FMOD provides us with, and then program a parameter which will trigger the corresponding reverb effect to activate in response to the in-game environment. This way, we do not have to create a different audio event for each separate environment, which leads to a great resource management.

4.2 Audiokinetic Wwise

The next middleware program we are going to look into is Wwise by Audiokinetic. Wwise consists of the Wwise middleware, the main interface for the design of adaptive audio, and the Wwise SDK, utilized for the integration of the sound to the sound engine. It is worth mentioning that Wwise can be configured in order to be linked with a DAW such as Reaper or Nuendo, this way enabling users to create a large portion of the sound in an environment they are more familiar with (Audiokinetic Wwise, 2019a). A few AAA game titles that make use of Wwise are: *Far Cry 5* (2018), *Hitman 2* (2018), *Doom Eternal* (2020), *Code Vein* (2019) and *Ghost of Tsushima* (2020).

Wwise. Wwise has three main modules named *Hierarchies*: the *Actor-Mixer*, the *Interactive Music*, and the *Master-Mixer*. It is within these parent folders that a vast variety of sub-folders are organized (see figure 4). Like with any big project, it is very important that one carefully organizes their elements, in order to avoid a large chaotic project which will unnecessarily complicate the composition process (Audiokinetic Wwise, 2019b).

Hierarchy Sections. The Actor-Mixer Hierarchy is the section in which all of the sound assets of the game are included. Essentially, it is here that all the sound and motion resources are grouped and organized, and where the user can define the properties and behaviors of said sound objects. Within this hierarchy, there can be different types of objects, each one with its own properties, for the multipart nature of a game's audio to be accommodated.

The Interactive Music Hierarchy is utilized for the interactive part of a game's music, allowing for ways to assemble interactive music objects into a game score. Two of the basic structures offered by this hierarchy are the *vertical project structure* and the *horizontal project structure*. The former enables the re-sequence of the game's music by shuffling the tracks contained in music segments—identical to the vertical re-sequencing technique examined earlier—while the latter is used to create variations in the game score by switching the segments played at any given time—identical to the horizontal re-sequencing technique described above. The usual approach is a combined use of these two structures. Inside this hierarchy, music files can be stored within various containers according to their characteristics and possible uses. The

aforementioned containers can be *Music Switch Containers*, *Music Playlist Containers*, or *Music Segments*.

Finally, the Master-Mixer Hierarchy sits on top of the other two hierarchies, and is a separate hierarchical structure functioning in a similar fashion to that of a DAW. It is from this hierarchy that the final mix for the game will be created. The Master-Mixer hierarchy is divided into two distinct sections: one for sound and music, and one for motion. By making use of the buses and groupings this hierarchy provides, an abundance of alterations to the sound are possible, such as volume and modulation automations, or changes in the spectrum of the music according to the in-game events through the use of filters (Audiokinetic Wwise, 2019b).

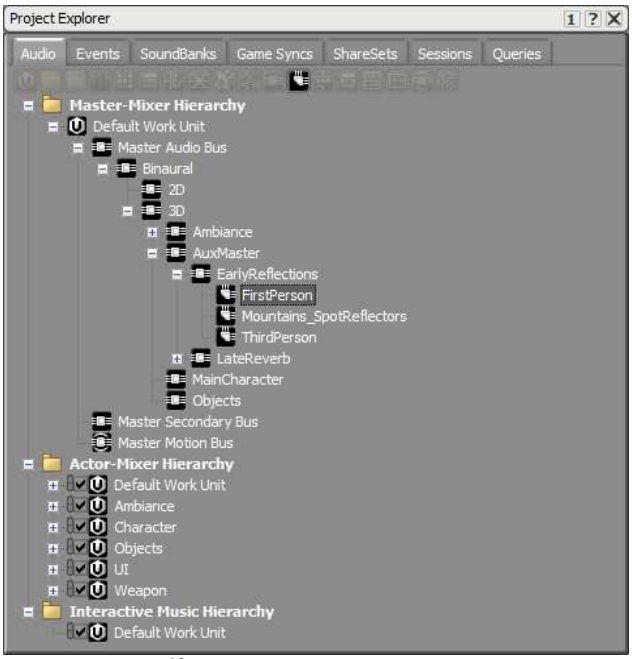


Figure 4. Wwise Hierarchy Sections.[10] In this picture, we can see the manner in which the Hierarchy sections are organized within Wwise.

Containers. A container is essentially a group of objects or containers, which in turn may contain other groups of objects or containers. (see figure 5). Depending on their nature, containers can be divided into the following categories:

Sequence Container: The objects inside a sequence container are played one after another in a certain order, like a playlist. An example in which such a container can be useful could be a car's start, when we must have a specific order of sounds: starting with the key insertion, the sound of the ignition and finally, the revving of the engine

[10] Figure used from: https://blog.audiokinetic.com/fr/simulating-dynamic-and-geometry-informed-early-reflections-with-wwise-reflect-plug-in-in-unreal

and idling sound. These sounds must be played in a specific order to make sense (Audiokinetic Wwise, 2019c).

Random Container: As the name suggests, objects inside a random container are played in a random order. There containers are especially useful for sounds that benefit from randomization, such as footsteps, or fire sounds. By using a small number of sounds, one can create a considerable amount of variations, this way creating the illusion of a grander amount of audio samples (Audiokinetic Wwise, 2019d).

Switch Container: The objects or containers that are inside a switch container are mapped to different matching switches inside the game. According to the switch that is active in the game, an object—or multiple objects—are played. A great example of this would be footstep sounds, which must change according to the terrain the character is stepping on. In other words, while the player is stepping on sand, the sand switch is triggered, and the container with the sand footsteps is activated. This container would be a random container due to the nature of footsteps. When the player is walking on a wooden floor, the corresponding container would be switched on, and so on (Audiokinetic Wwise, 2019e).

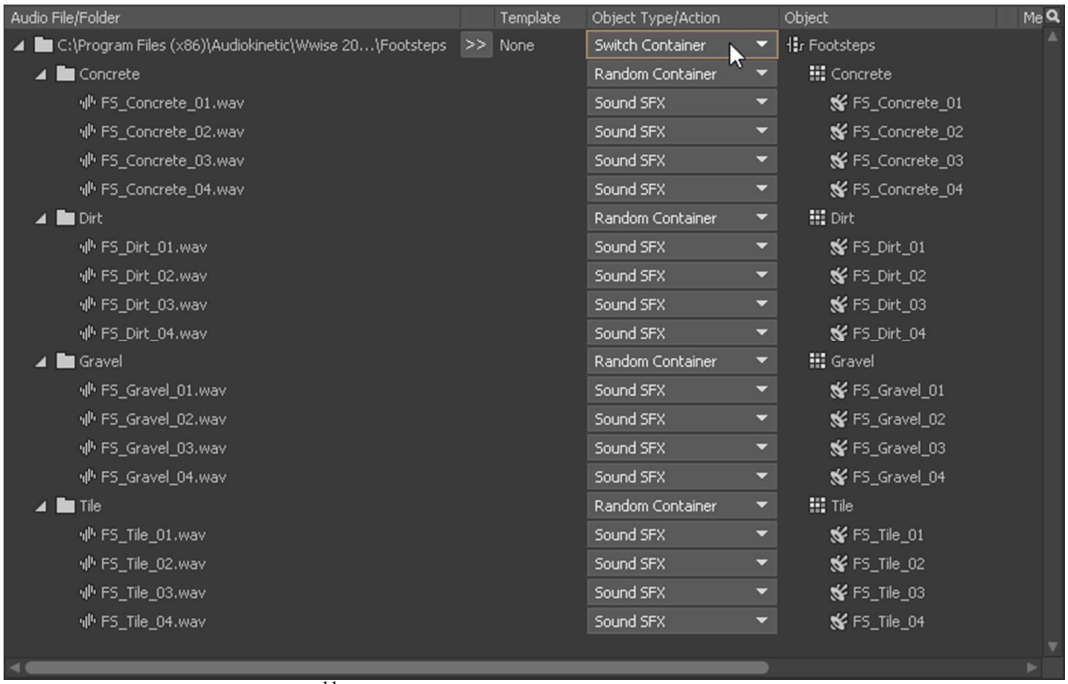


Figure 5. Wwise Container Types.[11] In this picture, we can see some examples of Wwise containers, as well as how we can change the container type.

[11] Figure used from: https://www.audiokinetic.com/fr/courses/wwise101/?source=wwise101&id=creating_switch_container

Blend Container: A blend container is able to play multiple objects simultaneously, blended according to RTPC's (Real Time Game Parameters). Different properties of the objects can be controlled in order to create the desired effect, and crossfades can also be employed between objects. This type of containers is often used for ambiences in games, such as the *Witcher 3* (2016), in which the weather sounds are mapped in blend containers. The RTPC has values for sunny, or windy and stormy weather which correspond to certain containers. This way, the ambience of the weather can change real-time without any lag (Lassi, 2018).

Events, and other tools for adaptive music within Wwise. The role of *Events* within Wwise is that of driving the audio into the game. Events interact with the various object groups and sound objects within the projects hierarchy, according to the user's specifications. These actions are usually tied to a specific function inside Wwise, which can be triggered from the game's engine. For instance, if the player throws a rock to a pile of bottles, the game engine in turn launches an Event that corresponds to the sound that should be played inside Wwise. The actions the user specifies will be the criteria upon which Wwise will decide to play, pause, stop, or alter the volume, pitch and filters of the objects.

There are two different types of Events: *Action Events*, and *Dialogue Events*. In Action Events, Wwise will use one or more actions, like play and stop, to drive the music, sound, and motion in game. In Dialogue Events, though, a decision type tree with *States* and *Switches* is utilized in order to dynamically determine which object is played (Audiokinetic Wwise, 2019b). Other tools utilized by Wwise to make an adaptive approach on the game audio possible are the following:

- *Game objects* are a rudimentary concept in Wwise, because for an Event to be triggered in the sound engine, an association with a game object is required. These game objects can be characters, weapons, ambient objects, or any other object or element in a game capable of producing a sound. The user may register game objects, and in turn, Wwise will store information which will be used to determine how each sound will be played back in-game (Audiokinetic Wwise, 2019b).
- *Game Syncs* are the means for the user to streamline and organize the changes and possible choices that are part of the game. Game syncs are usually applied after the initial game design has come to an end, and may be of the following kinds:
    - *States* are essentially pre-sets of values corresponding to the active state of the game, much like a mixer snapshot. They can be employed for changes that occur

through physical and environmental conditions in-game, and are a powerful tool for streamlining the way a user designs audio and motion. By making use of States, different properties are created for sounds, music, and motion objects, without any extra burden added to the memory and disc usage (Audiokinetic Wwise, 2019b). A State could be whether the player is underwater or not. When underwater, the volume of the sounds outside of the water should be lower. The high frequencies of the sounds should also be lowered using a low pass filter, in order to give the impression that we are, indeed, underwater. When the player goes underwater, the game engine switches the active State to "underwater", and when Wwise receives this change, it applies the volume and filtering settings (Audiokinetic Wwise, 2019b).

- *Switches* are representations of different alternatives for particular objects in-game. When a change is made, the appropriate sound, music, or motion object will be selected from the ones assigned to the particular switch. The Switch will then stay active until the aforementioned change stays active; when another change occurs, another Switch will be activated, and so on. An example of their use could be a change of the material on which the player walks; as long as the player walks on a specific material, such as wood, the corresponding Switch will stay on, and the footstep sounds played will be matching. When the material changes to sand, for instance, the Switch is turned off and another one corresponding to the new material is turned on, leading, thus, to the choice of footstep sounds from ones assigned to the sand Switch (Audiokinetic Wwise, 2019b).
- *RTPCs* enable the user to edit specific object properties in real-time according to the changes of in-game parameters. Practically, this allows custom automations in parameters such as *volume*, *pitch*, *filters*, and any other kind of sound processing, in order to match the events of the game (see figure 6). An example of this could be a racing game in which, depending on the speed and RPM of the car, the engine sounds should change. A way of achieving this change would be to map the pitch and volume level parameters of the car's engine to the speed and RPM values of the in-game car (Audiokinetic Wwise, 2019b).
- *Triggers* are used in response to spontaneous events in-game by adding embellishments on top of the musical score called *stingers*. Stingers are short musical audio cues that are tempo sensitive, and may bring intensity or information to the player in a very effective manner due to that characteristic (Paul, 2013).

Instead of being triggered instantly, like a regular SFX (Special Effect) would, musical stingers can be set to play on the next beat or bar, or any other musical division. This way, the desired effect is achieved without the disruption of the musical flow (Audiokinetic Wwise, 2019b).

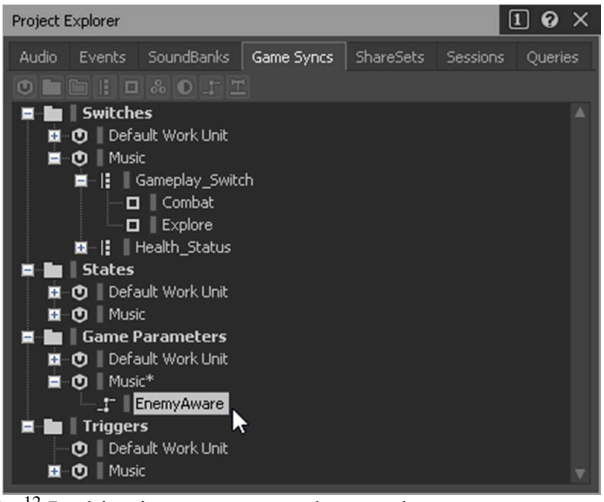


Figure 6. Wwise RTCPs.[12] In this picture, we can observe the way a game parameter is set in Wwise in order to trigger a change in the game's music.

SDK. The Wwise SDK plays the role of integrating the sound into the Game Engine, which is connected to the game and enables the use of adaptive audio. Wwise SDK makes use of the programming language C++, therefore a programmer is essential to the project (Audiokinetic Wwise, 2019g).

Using Wwise to Create a Music System. *Code Vein* (2019) is a successful video game title, with its music contributing a lot to its success. The music changes dynamically through the game, in response to the threat level the player experiences. This threat level can change according to the state of the game—peace or fight—the number of the enemies, and the player's health. As we did with *Dark Souls 3*, we will try to replicate the process of programming the audio of the game, using the information presented earlier.

We will begin by creating a scene under the Interactive Music Hierarchy, and then creating three different music playlist containers inside that scene. Each playlist container will include a single music segment corresponding to one of the states: no threat for the no threat playlist container, medium threat for the medium threat playlist, and high threat for the high threat playlist container. Inside these segments, we will place our audio files: an ambient sound

[12] Figure used from: https://www.audiokinetic.com/fr/courses/wwise201/?source=wwise201&id=smoothing_transition_decisions_configuring_an_rtpc_values_to_control_switch

for the no threat, silence for the medium threat, and a fight theme for the high threat. In each case, we should also program the audio to loop infinitely.

Our next step should be to program these different states within Wwise. We will do this by creating a state group under the Wwise's states, and three new states under the state group. Each of these states should then be associated with the corresponding music playlist containers, and will be chosen by Wwise according to the in-game threat level. Following this, we can calibrate the fade-ins and fade-outs through the scene's music switch container.

We now have a music system that will play and fade among the ambient audio of this game when the threat level is zero, nothing when the threat level is medium, and the fight theme when the threat level is high. However, we have still not programmed the heavy damage audio effect, or the stinger for the end of a high threat battle. In order to create the heavy damage audio effect, we will have to link a RTCP to a low pass filter, which will activate when the player's life drops under a certain value. The filter will then be programmed to deactivate after some seconds, since in the video game, this effect acts more as a warning to the player that their health is low, rather than a permanent effect of low health points. Finally, a musical trigger will be added to play the stinger of the high threat level end, in order to end the fight theme in a musical and satisfying manner.

4.3 ADX2

ADX2 is developed by the Japanese company CRI Middleware, which was founded in 1983. In combination with the movie encoding and playback solution *Sofdec2*, they constitute the *CRIWARE SDK*, which is available for *Unreal Engine*, *Unity* and native mode for main game platforms. The ADX2 consists of an authoring tool titled *AtomCraft*, and a run-time component named *CRI Atom library*.

The authoring tool is a means for the import of audio assets, their transformation into dynamic events that can be triggered and controlled, and finally, their organization into sound banks that will be exported to the game. The run-time component is the middleware part of the engine that provides specific features for sound effects, interactive music, and dialogue (see figure 10). The run-time component also offers means for the communication of the audio engine and the game, in order to achieve dynamic changes to the audio according to the in-game events (CRI Middleware, 2014). ADX2 has been used in numerous video game titles, mostly by Japanese companies such as Bandai and Namco, which produced games like *Dragon Ball Xenoverse* (2015), and *Street fighter V* (2016).

AtomCraft. AtomCraft's projects are divided into *Work Units*: non-run-time objects used to organize the sound by type or function, making, thus, the projects organized and easy

to work with. These Work Units may contain one or more *CueSheets*: an equivalent of a sound bank. These Cuesheets can in turn contain one or more *Cues*: audio events that will be triggered from the game.

A Cue is comprised of several *Tracks*, with each of them having a timeline on which *Waveform Regions* can be arranged. Cues may also be organized vertically in blocks with transition rules to make them more ideal for interactive music. Depending on the type of Cue, a different behavior will be presented, therefore determining the way the Tracks will be selected and played; some of the behaviors could be polyphonic, while others may be random or sequential, and so on. Every Track may also have its own parameters and *Actions,* which can apply looping at the Track level, alter the parameters, or call other Cues (CRI Middleware, 2014).

Cues can be found within the *Materials Tree*, inside the Cue Folder of the specific CueSheet. The Material tree aids the user by allowing them to simply drag and drop audio in order to import it into a project. Upon registering an audio file as a *Material,* the user can then specify the looping and encoding parameters, as well as using it as a Waveform Region on a Track.

After the user has finalized the cues for sound effects, music, and dialogue, the next step is exporting the cue sheets to the game. AtomCraft generates these three distinct files upon exportation:

- The *ACF* (Atom Configuration File) file holds the project's general information and which cue sheets are parts of it. Only one ACF file is exported per project.
- The *ACB* (Atom Cue sheet Binary) files include a specific cue sheet's parameters and all sample data intended to be played in memory by that cue sheet. Any audio data expected to be streamed will be kept in the matching AWB file.
- The *AWB* (Atom Wave Bank) files contain the encoded data for the streams that the cue sheet references.

In order to achieve an interactive relation between the game and the music/audio part, programmers and sound designers have to work together to devise sounds that react to the game context. In ADX2 there is a plethora of ways this can be achieved, with three of them being*:*

- *Selectors* are variables that can take a number of predefined values: the *selector labels*. When the Cues of type *Switch* are activated during run-time, they check the value of the selector and select to play only Tracks related to the specific value (i.e. selector label). An ideal application for a selector would be to change the audio of

footsteps according to the material the player traverses through (CRI Middleware, 2014).

- *AISACs* are the RTPC part of the ADX2. Exactly like the principle of RTPCs, one or more audio parameters of a Cue may be related to an AISAC which controls their values. AISACs can also control Cues which are not local to them, and they may also control other AISACs (CRI Middleware, 2014).
- *Blocks* are often utilized for interactive music, without that meaning they cannot be used for sound effects. They are a good way to jump between different sections of a Cue's timeline and permit the design of more complex sound schemes. The rules by which the block transitions are bound and their timing can be set in the authoring tool, although it can also be overridden at run-time (CRI Middleware, 2014).

Another way of controlling groups of sound via game related information is by making use of *categories* (see figure 7). Categories are like a mix snapshot for a pre-determined group of parameters of selected sounds tied to them. Categories make it easy to update the parameters of multiple sounds at the same time, which makes them ideal for managing sounds organized according to their type, such as dialogue or item sound (CRI Middleware, 2014).

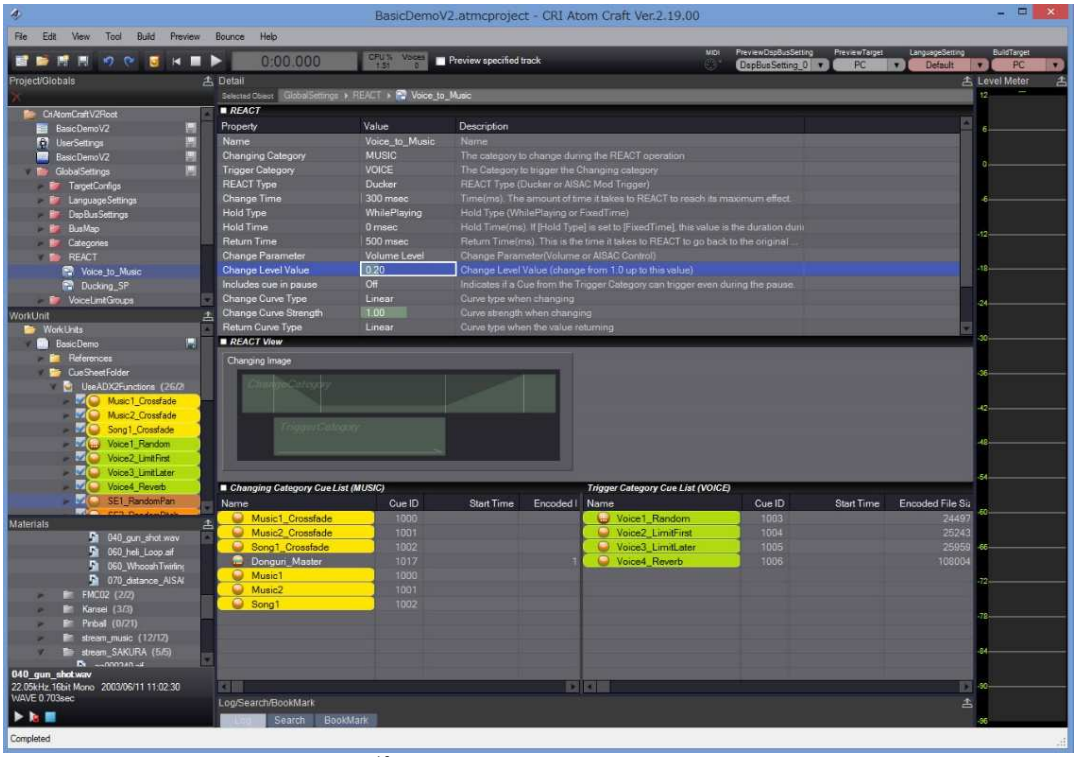


Figure 7. ADX2 Categories.[13] In this picture, we can see the manner in which we can organize the audio material within the categories of ADX2.

[13] Figure used from: https://www.criware.com/en/products/adx2.html

CRI Atom Library. The CRI Atom Library is the part of the ADX2 responsible for the loading of the CueSheet files generated in Atom Craft into the Game Engine. The CRI Atom library, like any API, makes use of a programming language (in this case C/C++) and as a result, a programmer is required to run it. However, many of the actions that are only possible through programming in other middleware—for example, combo sequential Cues, or automatic ducking according to sound categories—can also be produced by sound designers in ADX2 due the nature of its architecture (CRI Middleware, 2015).

4.4 Conclusions

Each middleware program has a lot to offer to its users, and has its own characteristics which will tend to each composer's needs and preferences. FMOD's interface, for example, is closer to that of a DAWs an audio engineer or a composer may be more familiar with, whereas Wwise has a layout closer to that of a game engine's, as a result making it more appealing to a game designer. For this reason, it's not unheard of for a video game to use more than one of these software; creating a video game is a grand task, and many people are involved in the process. This could easily lead to the sound design team using one middleware for the sound design of the video game, and the music composition team to use another one for the music. The possibilities are numerous and ever growing, and with the features of these software increasing every day, it is getting harder and harder to imagine a modern video game without them.

V. Composing Adaptive Music for a Video Game – Application

Taking into account all of the above research, we are now going to create a complete adaptive music score and audio design, and implement them via the use of middleware software to a fully-functional video game engine level.

5.1 Software

The selected software for this thesis are the Unreal Engine 4 game engine (see figures 8 and 9), the FMOD middleware and the Reaper DAW for sound design and composing music. The reason for using Unreal Engine 4 over other game engines is its abundance of features, as well as its popularity among AAA video game titles. FMOD, on the other hand, is picked over other middleware software due to its interface, since being closer to a DAW is a great way to ease into the world of middleware. Finally, the choice of Reaper for composing and audio design is based on its compatibility with the aforementioned software.

The video game level is a pre-designed level called *Close Combat: Swordsman* by *ZzGERTzZ*, and was purchased from the Unreal Engine Marketplace.[14] This package includes a fully-functional combat system, an animated character set with a sword, a move system, a leveling system, as well as a structured level. This package does not involve any sound elements, neither in the form of sound effects, nor music. For the purposes of this thesis, I designed the sound effects and music of this level, in an effort to apply the research presented in the previous chapters. Both the sound effects and the music for this level were created in Reaper and then imported into FMOD, via which they were routed into Unreal Engine.

The goal for the sound effect design was to achieve a real-time response from the numerous elements that may affect them—for example, the effect that a different floor material would have on footstep sounds, or an attack hitting a target—and to evolve according to the changing environment. The music I used was to accomplish an enhancement of the immersive experience of the player, by evolving as the character progresses through the level, defeats enemies, levels up, or changes status between an in-fight mode or out-of-fight mode, and in other similar fashions.

[14] For purchase details see: https://www.unrealengine.com/marketplace/en-US/product/close-combat-swordsman

5.2 Sound Design

The sounds created for this project were organized in the following categories: ambience, events, and character. These categories were also used in the FMOD's interface, in order to keep the project organized and transparent.

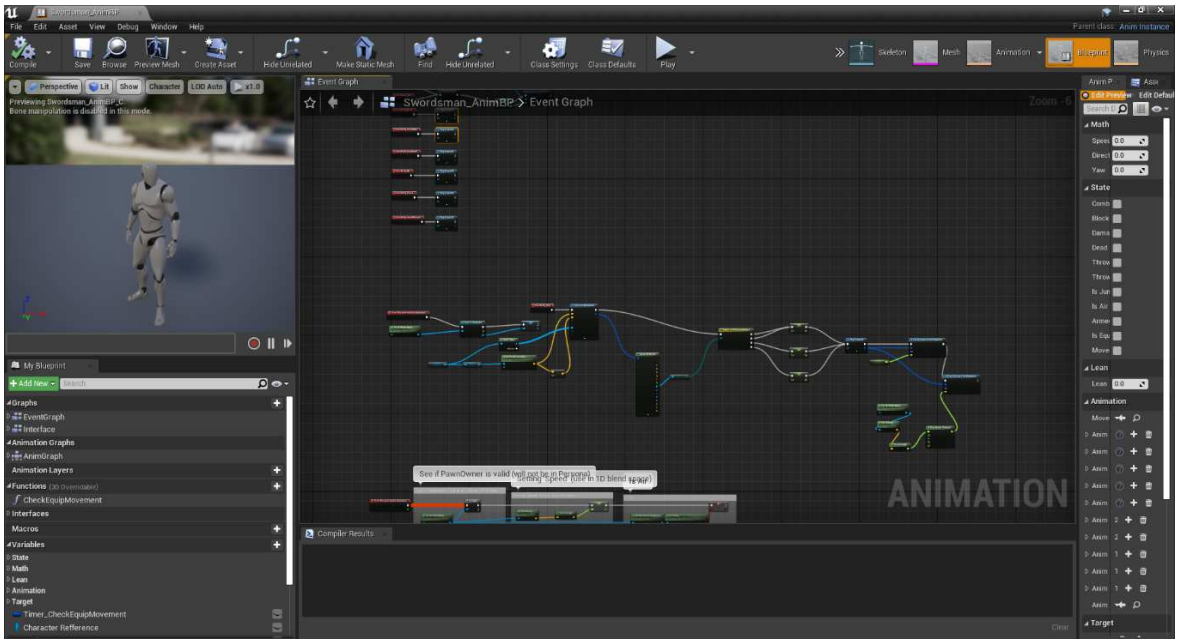


Figure 8. Unreal Engine's graphical programming environment. In order to trigger any data into being sent to FMOD, we first have to perform the necessary programming within Unreal Engine.

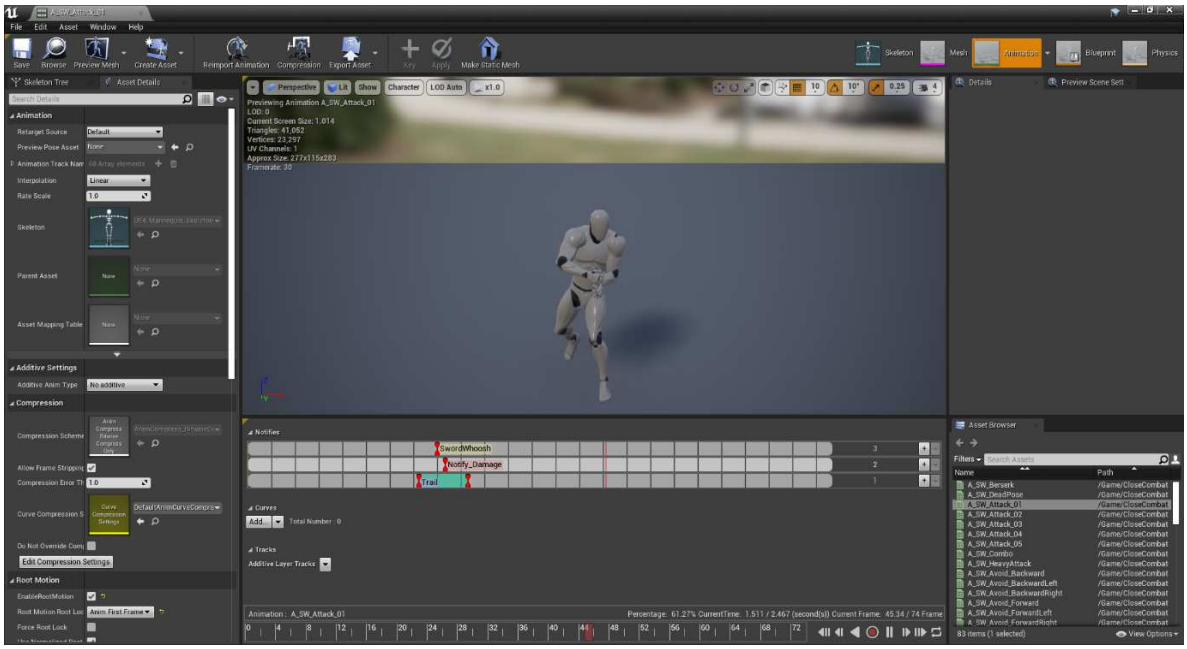


Figure 9. Unreal Engine Notify Markers. If the change in audio/music is triggered by certain movements of the in-game character, we have to place markers within the animation blueprints, this way securing the correct timing of the audio change.

5.2.1 Ambience

Four ambient sounds were used in this project: a general ambience played throughout the level, a special water ambience designed for the pool room, a fire sound for the flames in the first room, and a sound designed for the green orb at the end of the level. These four ambient sounds can be categorized into two separate categories: the *2-D events* and the *3-D events*.

The general ambience is a 2-D event; in other words, no matter where the character is located, the sound does not change in any way, this way giving the player a sense of consistency throughout the game level. The remaining sounds are 3-D events. In FMOD, this type of events offer the designer the ability to set a certain point in the level from which the sound is played by placing the 3-D event there as an object, this way allowing values to determine how the sound will fade out depending on the character's distance. This type of events is ideal for sound emitting sources, such as flames and water.

Another great tool for creating a convincing experience for the players is *mix snapshots*. FMOD gives each user the ability to create different mixes, which can then be recalled through Unreal Engine. This way, we can create different mixes depending on the room or space the character is in. Using mix snapshots enables sound designers, for example, to create a sound for a flame which fades in volume and goes through a high-cut filter when the character moves to another room, thus mimicking the acoustics that apply in the real world.

5.2.2 Events

The sounds included in this category were meant to be played only once, from start to finish, with no variations. These sounds were tied to certain events through the level, such as the opening of wooden gates, and although they are linear when it comes to their reproduction, their triggering point still has to be programmed. In order to do this, we have to create a *Play 2-D Event* inside the *Main Event Graph* of the level, and route it to the corresponding *Game Event*. This way, wherever in-game events producing sound occur, the event itself triggers the audio. It should be noted that the timing of the animations and the sounds should be carefully linked to avoid a poor result. The events in this example project are: the rise and activation of the turret, the death of the turret, the lowering of the bridge following it, the wooden gates that open in the third room (see figure 10), and the reach of max level of the player.

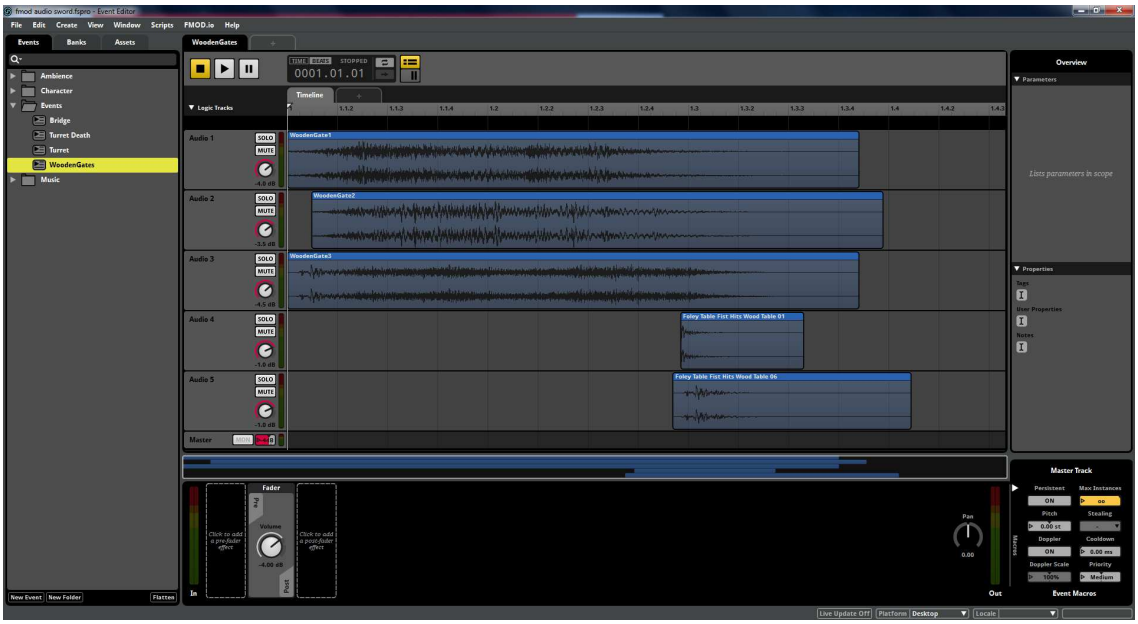


Figure 10. In this picture, we can see the construction of the linear audio event WoodenGates, which is triggered when the wooden gates of the third room open within the game. Notice the different timing for different audio tracks, creating a natural sounding succession to the sounds heard.

5.2.3 Character Sounds

This category contains all the sounds produced as a result of the actions of the game characters. These sounds include: footsteps, jumping, rolling sounds, hit sounds, sword sounds, taunts, turret shootings, jumping, blocking, berserk mode, death, parrying, activation of super mode, and taking damage (see figure 11). Depending on the room the character is in, these sounds are played through the corresponding room's reverb channel.

These sounds were comprised of multiple alternative audio files in order to avoid a repetitive and fake result when triggered multiple times. Moreover, *random volume* and *pitch modulators* were used inside FMOD to give an even more realistic feel to these sounds. In order to tie the sounds to the corresponding events in this category, 2-D events were programmed corresponding to markers in the *animation blueprints*. This means that whenever the character performs an *animation*, such as walking, a 2-D audio event is triggered on the exact moment when the *marker* was placed.

A special reference to the sounds of walking and rolling should be made, because they have some additional *modifiers*. Both of these sounds were programmed to change in accordance with the material the character is walking or rolling on; therefore, when the character is on the default stone surface, all of these markers trigger sounds tied to the stone

surface, Nevertheless, when the floor *material* is wood, then the markers trigger sounds tied to wooden surfaces. The same principle was applied on carpet surface as well.

An additional modifier was added to the footsteps of the characters, this time affecting their volume (intensity). Depending on the speed that the character walks or runs at, the volume of the footsteps changes, since when a person runs fast, their steps tend to be more intense than when they walk slowly.

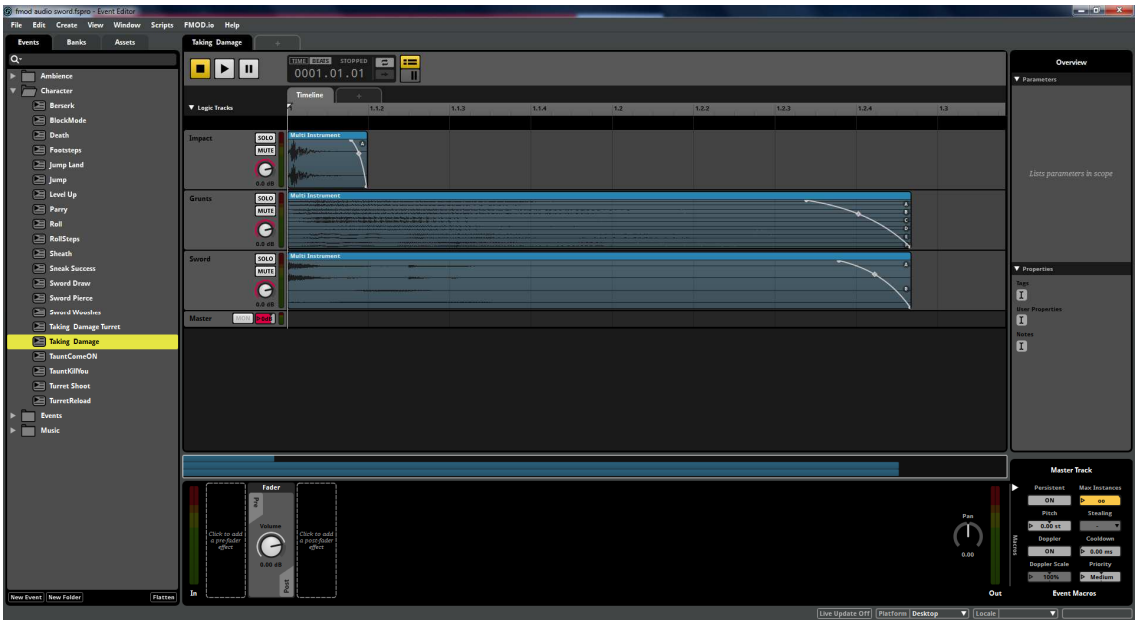


Figure 11. Taking Damage Event. In this picture, we can see two multiple instrument tracks (bottom two tracks), which are set to play the containing audio files in shuffle. This way, whenever the player takes damage, a unique audio cue is heard, therefore avoiding repetition of identical sounds.

5.3 Music Score

The music of this level was composed and programmed through the use of stems, in the manner of the aforementioned pre-composed music. The reason behind this choice was primarily the high quality which is typical of this type of adaptive music, and secondly, the popularity of this practice in the modern video game industry. All music for this level was composed in Reaper using VST instruments, and was then exported in the form of individual stems for each instrument. This standard practice enabled the organization of the stems inside FMOD, allowing for a convenient way of forming the vertical and horizontal reshaping that take place.

5.3.1 Aesthetics

The character of the music is quite a mysterious synth-like style, which was created to highlight the game's dark, stealthy, and murky atmosphere. The peace mode music focuses mainly on creating suspense by having an ambient character without noticeable rhythmic

instruments like drums, while the fight mode music has a strong rhythm in order to highlight the action taking place and motivate the player. The tempo of all music is 110 BPM, and the time signature 4/4.

5.3.2 FMOD

The music stems were imported into FMOD and then organized either vertically or horizontally according to their use. Vertically organized stems related to the game state were routed through pre-fader returns to allow the automation of multiple tracks at once. Horizontally organized stems were placed throughout the timeline, inside loop transition and regions. Each transition region was programmed to transition to the next one once the appropriate parameters are met, with each one quantized to a quarter note in order to maintain the rhythm of the music. The only transition not quantized was that of the final fight, for reasons we will discuss later on.

Apart from the music stems, a number of ambience stems, swells and drum fills were also imported within FMOD in the form of either multiple instruments, or *scatterer instruments*;[15] these stems appear in a relatively random fashion in terms of succession and timing, this way giving the music a more living feel, since some of its elements change constantly. As a result, every time the player goes through the same process in the game, the music is a little bit different.

5.3.3 Parameters

In order to trigger changes in either the vertical or the horizontal sequencing, several parameters were chosen and programmed in Unreal Engine and FMOD. These parameters were the following: the game mode (peace or fight state), the player's level, the turret (specifically active or inactive), and the final battle state.

Each parameter required a singular approach in the manner in which the information was extracted from the game's code and imported into FMOD. Out of these parameters, the ones tied to vertical re-sequencing were the game mode, the turret, and the super mode. The player's level, and the final battle state parameters lead to horizontal re-sequencing.

5.3.4 Vertical Re-Sequencing

The first parameter we are going to look into is the game mode. This parameter could have also been programmed to lead to horizontal re-sequencing, but doing this would mean

[15] A scatterer instrument in FMOD is a special type of instrument which randomly plays one or more of its containing audio files according to the user's specifications. This type of instrument is very useful for creating a more organic and natural feel, since it is quite difficult to have the same results in two different playbacks.

that each time the game state changed from fight to peace and vice versa, the music would begin from the top; this approach gives a rather boxed result, which, in my opinion, interferes with the immersion of the player. Due to this reason, a more organic result is produced when both tracks are played simultaneously, and given the state of the parameter, the tracks of the corresponding mode are muted. This way, each time the game state changes, the music will not start from the same point, but will rather start from a seemingly random point—quantized to a quarter note in FMOD so as not to ruin the rhythm of the music—giving the player, therefore, a more immersive experience.

Another parameter which triggers a vertical re-sequencing is the activation of the turret (see figure 12). Upon activation, one of two stems, each containing an electric-like sounding melody, is added to the already existing music. Which one of the two melodies will be played depends on the game state; if the player is not noticed by the turret and the state of the game remains peaceful, then the stem routed through the peace state is played; when the player engages the turret and the game state changes to fight mode, the corresponding turret stem will activate. In case the player is noticed right away, the peace stem will not activate in this playthrough.

The final vertical re-sequencing parameter is that of the super mode. The super mode can only be activated once the player reaches max level. Upon activation, a choir stem is added on top of the existing music. The same principle applies to the final battle fight music, but this time, an extra layer of epic drums is added as well.

5.3.5 Horizontal Re-Sequencing

The player's level is the parameter which causes the most changes in the game music: it triggers changes in both the game modes by activating FMOD's transition regions, and leads to new timeline locations where the music changes horizontally, allowing new harmonies and rhythmic changes.

Whenever a player proceeds to the next level, the music progresses in order to reward their actions. In peace mode level 1, the music is rather static, with the bell melody stem having only a single note every two bars. However, when the player defeats the first opponent and reaches level 2, the bell melody has two notes each time. In the corresponding fight music levels, level 1 music has a less aggressive rhythmic pattern, while level 2 music increases tension by using a more intense one.

In level 3, the player reaches the end of the game, and the music acknowledges this fact by changing in a more drastic fashion. In peace mode, the music abandons its static character

by making use of a harmonic progression, which gives the player the required stimulation to feel rewarded for their actions. In fight mode, a new melody appears, this time with more dramatic characteristics that increase the tension leading to the game's finale.

Finally, in level 4 peace music, an extra layer of emotionally charged synth strings is added to the level 3 peace music, which prepares the player for the final battle. In level 4, there is no fight music composed, since the only fight the player has in this level is the final battle, which has its own music.

The final fight parameter is the only parameter with a non-quantized transition. This choice was made due to the importance of the transition, as well as the manner in which it happens: the player drops down a hole—a rather abrupt transition game-wise—which, in turn, requires an abrupt transition music-wise. Once the player in engaged in the final fight, they hear music which, while retaining the character and the sounds played earlier in the game, has completely new musical themes, melodies and rhythm (see figure 12).

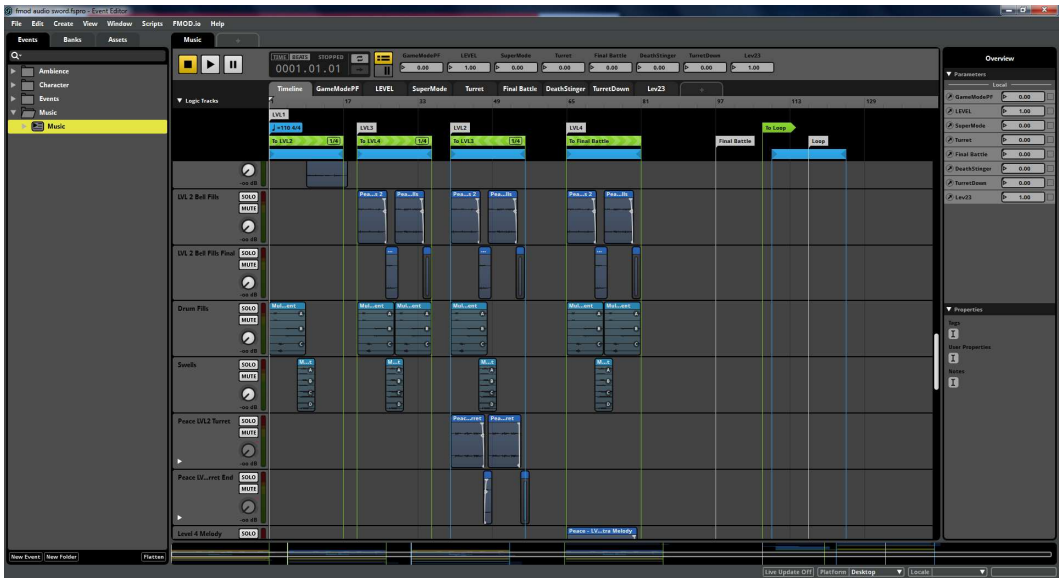


Figure 12. FMOD Music Event. In this picture, we can see the timeline of the music event within FMOD. The music event contains the numerous stems, loop areas, and the destination areas in which the timeline jumps to when the corresponding conditions are met.

Conclusions

The video game industry is one of the biggest industries of this era, reaching people all around the globe and of all preferences. The quality of video games constantly increases in every aspect, from storytelling to controls, and from graphics to music, and it is clear that as technology progresses, the video game world will follow. Music has ascended from a pleasant extra feature to an essential part of what makes a video game what it is, with dedicated teams working towards the task of creating a unique and adaptive music score; a new standard, that is.

The modern player has come to expect a musically respectable score reflecting their actions and tailored to their specific game style and story choices. Even small changes can subconsciously satisfy the player and enhance their immersion experience, thus creating a quality video game. Such scores cannot be created by programmers, as they once were, and actual music composers must take action.

Through the use of middleware software, a composer can nowadays create adaptive music without the need for programming knowledge. A composer's creation, whichever adaptive technique it may be created with, can be intergraded into any video game engine through these software. It is due to the abundance of features these software possess that they have become a standard for the industry.

In this thesis, we discussed an overview of the history of adaptive music in video games through the years, from its early experimental stage to its dominating role today—an evolution fueled by the advances in technology, gaming consoles and computers alike. This transformation in the way music interacts with the in-game events shaped new techniques for composing and arranging music based on already existing knowledge, but also innovations created for this purpose alone.

As expected, new software programs in the form of Middleware software were created to assist these endeavors. In this thesis, the basic structure and features of some of the most used middleware software in the market were presented and used in mock examples, in order to give the reader a better understanding of the way audio is implemented in today's video games.

Finally, an application of the research presented in this thesis was performed through the audio design, music composition, and implementation of them in a functional video game level. All creative decisions were explained, giving the reader an insight into the process of

creating the audio for a video game—both on a theoretical and a practical level—and therefore presenting a complete approach on the matter.

References

Abreu, V. (2018). *Recreating tracker music sequencers in modern videogames: an integrated model approach for adaptive music*. University of Porto.

Berndt, A., & Hartmann, K. (2007). Strategies for Narrative and Adaptive Game Scoring. *Audio Mostly 2007 Conference Proceedings 27-28/9/2007,* pp. 141-147.

Bribitzer-Stull, M. (2015). *Understanding the Leitmotif*. Cambridge University Press.

Brown, A., & Kerr, T. (2009). Adaptive Music Techniques. *The Australasian Computer Music Conference 2009*, pp. 26-31.

Chandler, A. (2015). *Inventing the Electronic Century.* Harvard University Press.

Collins, K. (2008). *Game Sound*. The MIT Press.

Collins, K. (2009). An introduction to procedural music in video games. *Contemporary Music Review 28*(1), pp. 5-15.

Farnell, A. (2007). "Synthetic game audio with Puredata", *Audio Mostly 2007 Conference Proceedings 27-28/9/2007,* pp. 76-83

Fritsch, M. (2013). History of Video Game Music. *Music and Game, Springer VS 2013*, pp. 10-40.

Gillespie, S., & Bown, O. (2017). *Solving adaptive game music transitions from a composer centred perspective.*

Hedges, S. (2015). Dice Music in the Eighteenth Century. *Music & Letters 59*(2), Oxford University Press 2015, pp. 180-187.

Hey, T., & Papay, G. (2014). *The Computing Universe.* Cambridge University Press.

Lassi, K. (2018). *Producing Adaptive Music for Non-Linear Media.* Tampere University of Applied Sciences.

Liebe, M. (2013). Interactivity and Music in Computer Games. *Music and Game, Springer VS 2013*, pp. 41-62.

Marks, A. (2009). *The Complete Guide to Game Audio*. Focal Press.

McWhertor, M. (2014). *The Evil Within mixes survival horror with psychological terror, then steeps it in gore.* Polygon. within-review-xbox-one-PS4

Paul, L. (2013). Droppin' Science: Video Game Audio Breakdown. *Music and Game, Springer VS 2013*, pp. 63-80.

Peederman, P. (2010). *Sound and Music in Games.* Vrije University of Amsterdam.

Pichlmair, M., & Kayali, F. (2007). Levels of Sound: On the Principles of Interactivity in Music Video Games. *DiGRA 2007*, pp. 424-430.

Somberg, G. (2016). *Game Audio Programming.* CRC Press.

Strank, W. (2013). The Legacy of iMuse: Interactive Video Game Music in the 1990s. *Music and Game, Springer VS 2013*, pp. 81-91.

Summers, T. (2011). Playing the Tune: Video Game Music, Gamers, and Genre, *ACT 2011/02*

Wallis, I., & Ingalls, T., & Campana, E. (2008). Computer-generating emotional music: the design of an affective music algorithm. *The 11th Int. Conference on Digital Audio Effects (DAFx-08).*

Wolf, M. (2012). *Before the Crash*. Wayne State University Press.

Wolf, M. (2007). *The Video Game Explosion*. Greenwood Press.

Young, D. (2012). *Adaptive Game Music: The Evolution and Future of Dynamic Music Systems in Video Games*. The Honors Tutorial College.

Zhang, J., & Fu, X. (2015). *The Influence of Background Music of Video Games on Immersion,* Journal of Psychology & Psychotherapy 2015

Manuals

Audiokinetic Wwise (2019). *Wwise Documentation a*. Retrieved November 14, 2020, from https://www.audiokinetic.com/library/edge/?source=Help&id=wwise_help

Audiokinetic Wwise (2019). *Wwise Documentation b*. Retrieved November 14, 2020, from https://www.audiokinetic.com/library/edge/?source=WwiseFundamentalApproach&id=wwise_fundamentals

Audiokinetic Wwise (2019). *Wwise Documentation c*. Retrieved November 14, 2020, from https://www.audiokinetic.com/learn/videos/hWO9dkD4mzo/

Audiokinetic Wwise (2019). *Wwise Documentation d*. Retrieved November 14, 2020, from https://www.audiokinetic.com/learn/videos/Vk5A93kSDKE/

Audiokinetic Wwise (2019). *Wwise Documentation e*. Retrieved November 14, 2020, from https://www.audiokinetic.com/learn/videos/829S9ipncOA/

Audiokinetic Wwise (2019, November). *Wwise Documentation f*. Retrieved November 14, 2020, from https://www.audiokinetic.com/learn/videos/GV05BtSlpSw/

Audiokinetic Wwise (2019, November). *Wwise Documentation g*. Retrieved November 14, 2020, from https://www.audiokinetic.com/library/edge/?source=SDK&id=index.html

CRIWARE (2016). *Audio Director Quick Start 2016*. https://www.criware.com/en/products/adx2.html

CRIWARE (2015). *Audio Programmer Quick Start 2015*. https://www.criware.com/en/products/adx2.html

CRIWARE (2014). *Sound Designer Quick Start 2014*. https://www.criware.com/en/products/adx2.html

FMOD (n.d.). *FMOD Studio Documentation a*. Retrieved November 14, 2020, from https://www.fmod.com/resources/documentation-studio?version=2.1&page=fmod-studio-concepts.html

FMOD (n.d.). *FMOD API Documentation*. Retrieved November 14, 2020, from https://www.fmod.com/resources/documentation-api?version=2.1&page=welcome.html

Composed Video Game Score: https://www.youtube.com/watch?v=ou56QzlNI0o