Bachelor's thesis

Information and Communications Technology

2019

Markus Sukoinen

AUDIO IMPLEMENTATION METHODS IN UNITY

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2019 | 27 pages

Markus Sukoinen

AUDIO IMPLEMENTATION METHODS IN UNITY

This thesis introduces the reader to video game audio implementation. Its goal was to bridge the gap between sound designers and programmers since even cross-functional teams often lack sound designers with programming skills, leaving programmers to make decisions concerning game audio during the implementation process. The thesis compared two different implementation methods used in two individual game development projects. The first game had all implementation work completed in the game engine, while the second game incorporated the use of middleware.

The benefits of using middleware became very apparent during development. The sound designer was able to create events that contained completed functionality and these events could then be imported into the Unity game engine. Thus, the audio implementation process required significantly less programming and allowed the sound designer to be more involved.

Middleware will add additional costs to development and should be considered on a case-by-case basis. Because the licensing scheme of the investigated middleware does not line up with the commisioner's business model, competing middleware solutions should be explored for future projects.

KEYWORDS:

Game, Audio, Implementation, Unity, FMOD, Sound Design

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikan koulutus

2019 | 27 sivua

Markus Sukoinen

ÄÄNEN IMPLEMENTOINTITAVAT UNITY-PELIMOOTTORISSA

Opinnäytetyön aiheena on pelien äänien implementointi, ja sen tavoitteena on rakentaa siltaa äänisuunnittelijoiden ja ohjelmoijien väliseen kuiluun. Jopa moniosaajatiimeiltä puuttuu usein ohjelmointitaitoiset äänisuunnittelijat, jolloin ohjelmoijien on tehtävä päätöksiä pelien äänistä implementointiprosessin aikana. Opinnäytetyö vertaili kahdessa erillisessä pelinkehitysprojektissa käytettyjä implementointitapoja. Ensimmäisessä pelissä kaikki implementointityö tehtiin pelimoottorissa, kun taas toisessa apuna käytettiin väliohjelmistoa.

Väliohjelmiston tuomat hyödyt tulivat esiin opinnäytetyön aikana. Äänisuunnittelija sai väliohjelmiston avulla luotua valmiita toiminnallisuuksia sisältäviä tapahtumia, jotka voitiin tuoda suoraan Unity-pelimoottoriin. Näin äänien implementointityö vaati huomattavasti vähemmän ohjelmointia, ja äänisuunnittelija pystyi ottamaan aktiivisemman aseman tiimissä.

Lisääntyneiden kustannuksien takia tulee väliohjelmiston käyttöä harkita tapauskohtaisesti. Koska tutkitun väliohjelmiston lisensointimalli ei sovi yhteen opinnäytetyön asiakkaan liikemallin kanssa, on syytä arvioida kilpailevia ratkaisuja tulevia hankkeita varten

ASIASANAT:

Peli, ääni, implementointi, Unity, FMOD, äänisuunnittelu

CONTENTS

FIGURES

LIST OF ABBREVIATIONS

.mp3	Audio file format using lossy compression
.wav	Uncompressed audio file format
2D	Two-dimensional
3D	Three-dimensional
API	Application Programming Interface. Sofware intermediary allowing applications to talk to each other (Mulesoft, n.d.)
Audio middleware	A third party tool set sitting between the game engine and audio hardware providing common functionality (Brown, n.d.)
Beat	Unit of time used in music theory
BPM	Beats per minute. Indicates tempo of a musical track.
CPU	Computer Processing Unit
DAW	Digital Audio Workstation. An application used to record, edit, arrange and export audio.
dB	Decibel. Unit used to measure sound level, but is also used in electronics, signals and communication (Wolfe, n.d.)
EULA	End-user license agreement
GUI	Graphical User Interface
iOS	A mobile operating system created by Apple Inc.
Isometric projection	A method of presenting drawings in three dimensions
Key	A group of tones forming the foundation for a musical piece.
Mix Bus	A channel on a mixer that collects and outputs any channels sent to it.
Mixer	A mixer collects audio channels and combines them. It is usually able to control output levels on the channels.
Semitone	The smallest interval between notes used in Western music theory
SFX	Sound effects
Wet/Dry levels	The proportion between original (dry) and ”effected” (wet) signals (Indiana University, n.d.)

1 INTRODUCTION

Audio implementation in the context of a typical game development cycle is an often overlooked phase that is arguably much less straight-forward than the implementation of graphical assets. Audio implementation refers to the process of transferring audio assets into the game project in a way that results in a cohesive experience for the end user. Such assets include music, dialogue, and SFX. Audio implementation in its most basic form requires a combination of technical and artistic ability; traits that are not always both found in the same team member.

This thesis is commissioned by Plush Pop Soft Ltd, a small-sized video game studio based in Turku, Finland. Audio implementation created a gap in the development of previous projects, which prompted the company to want to expand their knowledge in the topic. As literature on audio implementation is scarce, and in regards to Unity, almost non-existant, a thesis on the subject could not only prove valuable to the commissioner but also function as a resource for students and aspiring developers. The best-known third-party resource for audio implementation is Game Audio Implementation: A Practical Guide using the Unreal Engine (2016), but as the name implies, it focuses exclusively on the Unreal Engine taking advantage of its Blueprints system. This alone makes it unhelpful for games developed in the Unity Engine.

Addressing the above mentioned need, this thesis aims to answer the following questions:

1. What happens to audio assets after they are made so that they end up in the final product?
2. Who in the team is responsible for this phase and what does it even entail?
3. Could the process be made any easier?
4. Would a middleware solution provide eventual savings for the company?

By answering these questions the thesis provides the reader with a basic understanding of how audio is implemented, specifically when working with the Unity game engine. The implementation methods are separated into two categories: implementation using stock tools in Unity and a middleware solution. The thesis will introduce the tools used in these methods and present the reader issues and potential solutions found during the development process. The thesis assumes some level of knowledge of the Unity Editor and C#.

All sound production and the following implementation was carried out specifically for this thesis.

2 AUDIO IMPLEMENTATION

2.1 Introduction to audio implementation

In order to understand the challenge audio implementation might prove to a game development team, one must first understand what audio implementation is and where it fits in the development cycle. Traditionally, audio implementation has been an additional duty of the sound designers (Berklee College of Music, n.d.) but this is not necessarily the case any longer. Audio programming is a term also often used interchangeably with audio implementation.

The unpredictable nature of an interactive medium, such as video games, poses a unique problem. In a movie, the sounds will always play in the same exact spot at the same time every time. In a video game, sounds often need to be triggered by something in the game environment, background music swells in and out and dialogue might change because of actions that the player chooses to take. This is why audio implementation is a necessary process in game development. When a developer is taking audio assets and making them fit in the game, he is implementing audio.

2.2 Audio asset flow in a standard setting

Audio assets go through many phases before they end up in the final product. Figure 1 is a process chart showing the main phases relevant from the sound design perspective.

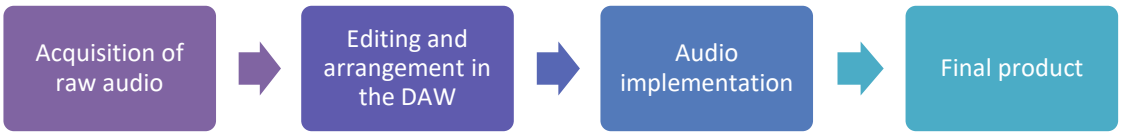

Figure 1. Audio asset flow in standard setting.

First, the sound designer has to acquire raw audio. The method of acquiring raw audio depends on the budget, size or type of game in development, for example, recording ambient sounds in a forest with a field recorder, recording dialogue with voice actors in a studio environment or simply purchasing audio files from a sound bank.

The next step is the actual production stage, where a sound designer or music producer works in the DAW editing and arranging sounds to usable audio assets. While Unity supports many audio formats and is able to compress to a suitable format and size, it is advisable to export to

an uncompressed audio format and follow a sample rate of 44.1 kHz (16bit). A higher sample rate, such as 48 kHz (24bit), will be resampled into 44.1 kHz (16bit) during compression resulting in an output that is two semitones lower than intended, possibly giving the listener an impression of out-of-tune audio. Making sure to use the correct format from the start will save developers from headaches later on (Stevens and Raybould, 2016)

In the end, the chosen format depends on the priorities of the developer. MP3 is an exception since it makes it impossible to create seamless loops and should not, therefore, be used as a format for background music or other looping audio (Strauss, n.d.). Once the audio files have been exported, they are ready to be implemented in the game.

2.3 Audio asset flow in using a middleware solution

A middleware solution provides a slightly different flow as is shown in Figure 2. While on the surface it might look like an extra step, the point is to greatly speed up the following audio implementation work in the Unity game engine and offload some of the programming work to the sound designer as well as to give the sound designer more control over how, when and where the sounds are triggered.

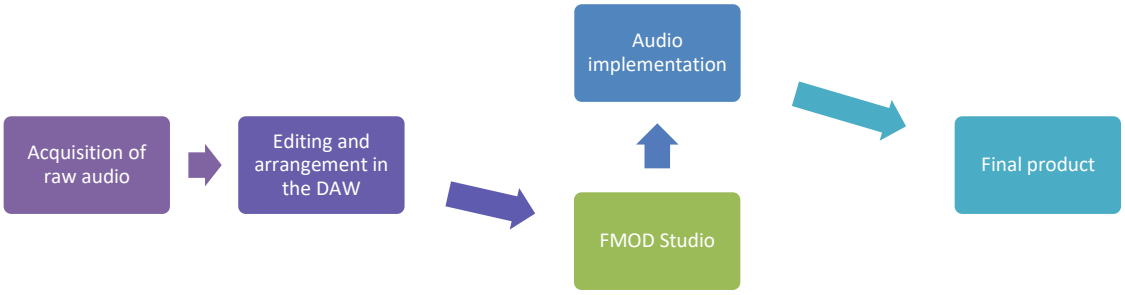

Figure 2. Audio asset flow using FMOD as a middleware solution.

Middleware is a bridge between the sound designers and the programmers. It is a codebase most often structured as two separate sections: the GUI and the API. The GUI is where a sound designer can import audio, design events and often even test the audio in real-time, thus allowing a sound designer to at least partly take on the role of an audio implementer. The API then allows a programmer to access the codebase without the need of delving into the main code. (Horowitz and Looney, 2017).

The audio middleware chosen for this thesis is FMOD. Other notable audio middleware solutions are Wwise by Audiokinetic and Fabric by Tazman-Audio.

Unlike Fabric and Wwise, which provide a custom interface and audio features and integrate into the Unity engine seamlessly, FMOD uses a desktop application completely separating the GUI from the game engine. In this application the sound designer creates audio events that can be later imported into Unity. As a result, the great bulk of the work is already carried out when the audio implementation phase reaches the Unity Editor.

3 FMOD

3.1 Overview of FMOD

FMOD is an industry standard solution for implementing audio. It offers a complete toolset for designing, organizing and optimizing adaptive audio. Often used features are, for example, volume automation, pitch shifting, audio panning and looping. While the FMOD product family includes FMOD Core and FMOD.io, the most relevant product to this thesis is FMOD Studio (Figure 3).

Figure 3. The FMOD Studio GUI.

FMOD Studio can be separated into the following technologies:

- FMOD Studio, a desktop application that uses a DAW-like GUI (Horowitz, S. and Looney, S. 2017), making it an approachable program for sound designers and music producers. Figure 3 shows the layout of the program when editing an event. Figure 4 presents the GUI of a typical mainstream DAW, Logic Pro X, for comparison.
- FMOD Studio API is the programmer interface used to load FMOD Studio banks and trigger events made by the sound designer in FMOD Studio.
- FMOD Studio Low Level API, which is a toolless programmer interface used to trigger simple sounds only. (Firelight Technologies a, n.d.)

FMOD has been used in many award-winning games such as Celeste (2018), Subnautica (2018) and Dark Souls 3 (2016). (Firelight Technologies d, n.d.)

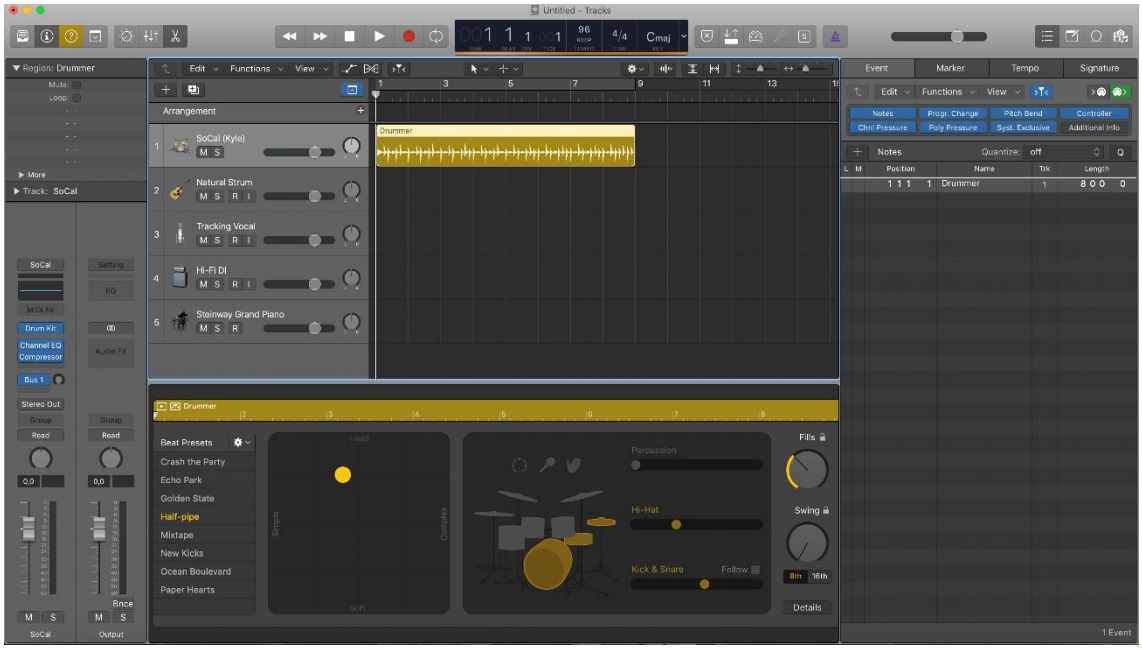

Figure 4. The GUI of Logic Pro X.

3.2 Licensing

An obvious disadvantage compared to using a custom solution or stock tools provided by the engine is the licensing costs. This is no different for FMOD. Licensing schemes are divided into three tiers:

- Indie is the most affordable scheme but it does not offer technical support. Indie is meant for studios with a development budget under $500k. This license can be used for free once every 12 months, but any additional uses within this time period will incur a fee of $2000 per game.
- Basic is the mid-tier licensing scheme. This is meant for development budgets between $500k and $1.5M. The fee is a fixed $5,000 per game. This includes support for a year.
- Premium is the most expensive licensing scheme and meant for games with a development budget over $1.5M. The fee is set to $15,000 per game and it includes support for two years.

All licenses include lifetime distribution rights and the full set of FMOD features. A license is only needed when a game or application is meant for commercial distribution and it is not necessary to purchase a license when the game or application is still in development. Personal and educational use is permitted under the terms of the EULA. (Firelight Technologies e, n.d.).

3.3 FMOD concepts

FMOD Studio uses proprietary concepts and a basic understanding of some of its core ideas is needed to follow along this thesis. The most relevant concepts will be explained in the following paragraphs .

The events are the most important concept in FMOD Studio. They are instances of sound that can be triggered and controlled by code (Firelight Technologies b, n.d.). They contain tracks, instruments, and parameters. Events are listed in the left side of the GUI, shown in Figure 5.

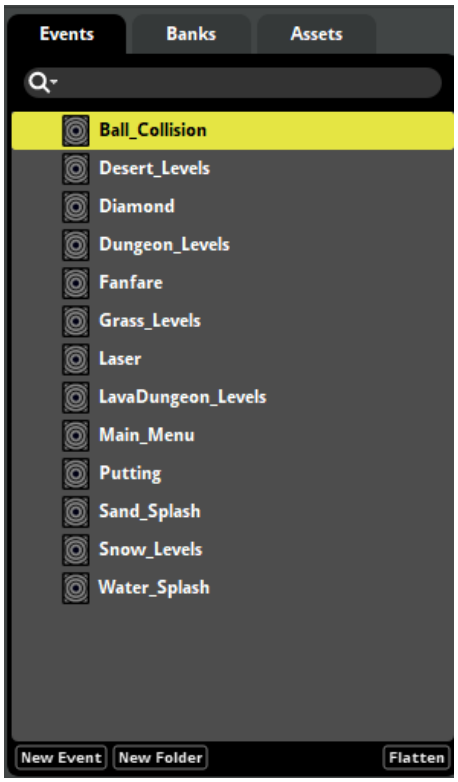

Figure 5. The Events list containing all FMOD events used in Minigolf Universe.

Tracks allow event instances to function as a kind of mixer. By selecting an event in the list, its tracks will be displayed in the editor window (Firelight Technologies b, n.d.). The tracks of an example event are displayed in Figure 6.

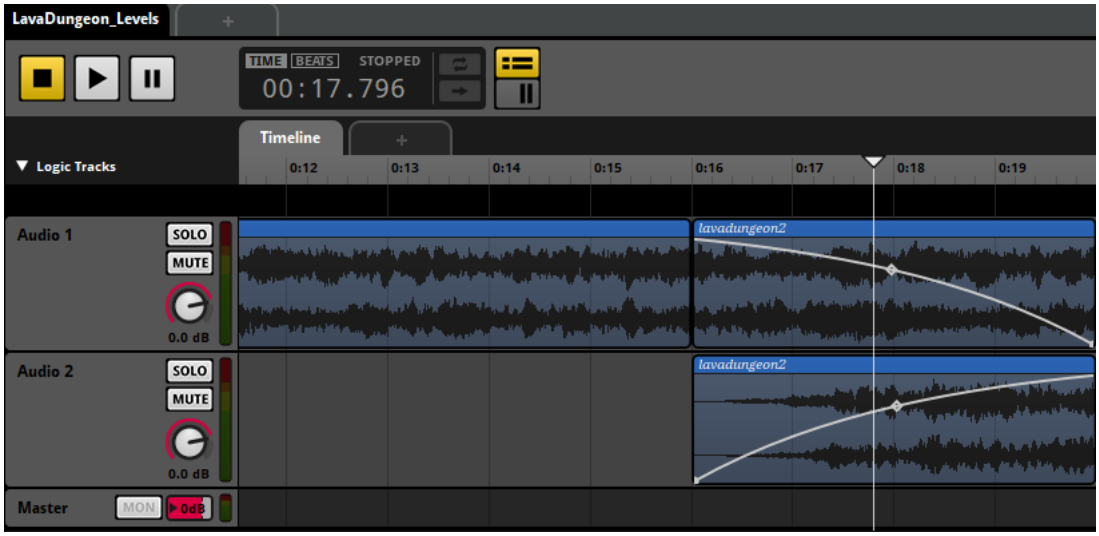

Figure 6. Audio tracks displayed in the editor windows of FMOD Studio.

Parameters can change what an event is doing after it has been triggered. Paramaters can be used to alter event properties during its run time by automating its values. (Firelight Technologies b, n.d.). Figure 6 shows an automation curve on tracks Audio 1 and Audio 2. These curves control the volumes for each track allowing a smooth transition. This kind of transition is called a cross-fade (Merrian-Webster n.d.).

4 USING UNITY TOOLS

4.1 Rapid Magic

The first audio implementation solution that will be explained in this thesis is for the game Rapid Magic, which only uses stock tools provided by the Unity game engine. Rapid Magic is a 2D side-scrolling adventure game with a pixelated art style made for mobile platforms. As such, positional audio will not be covered and most audio will come from a mono source. Rapid Magic was in development for a year and includes a full soundtrack as well as a multitude of SFX. Rapid Magic was published in 2018 for iOS devices and it was developed in Unity version 2017.1.4.

4.2 Importing audio

Importing audio into the Unity asset folder is simple. The first step is to create a folder under Assets in the Project view. This is done by right-clicking, choosing Create and then Folder. Alternatively folders can be created from the upper left corner by clicking on Assets and then following the same steps. The folder should be named Audio, and subsequent folders for SFX, level music and dialogue should be created for an organized folder structure. Naming assets and folders clearly from the start makes it easier to scale up game projects later.

Unity supports .aif, .wav, .mp3, and .ogg file formats (Unity Technologies. n.d), but all created audio assets for Rapid Magic are in .wav. for highest quality and consistency.

4.2.1 Import settings

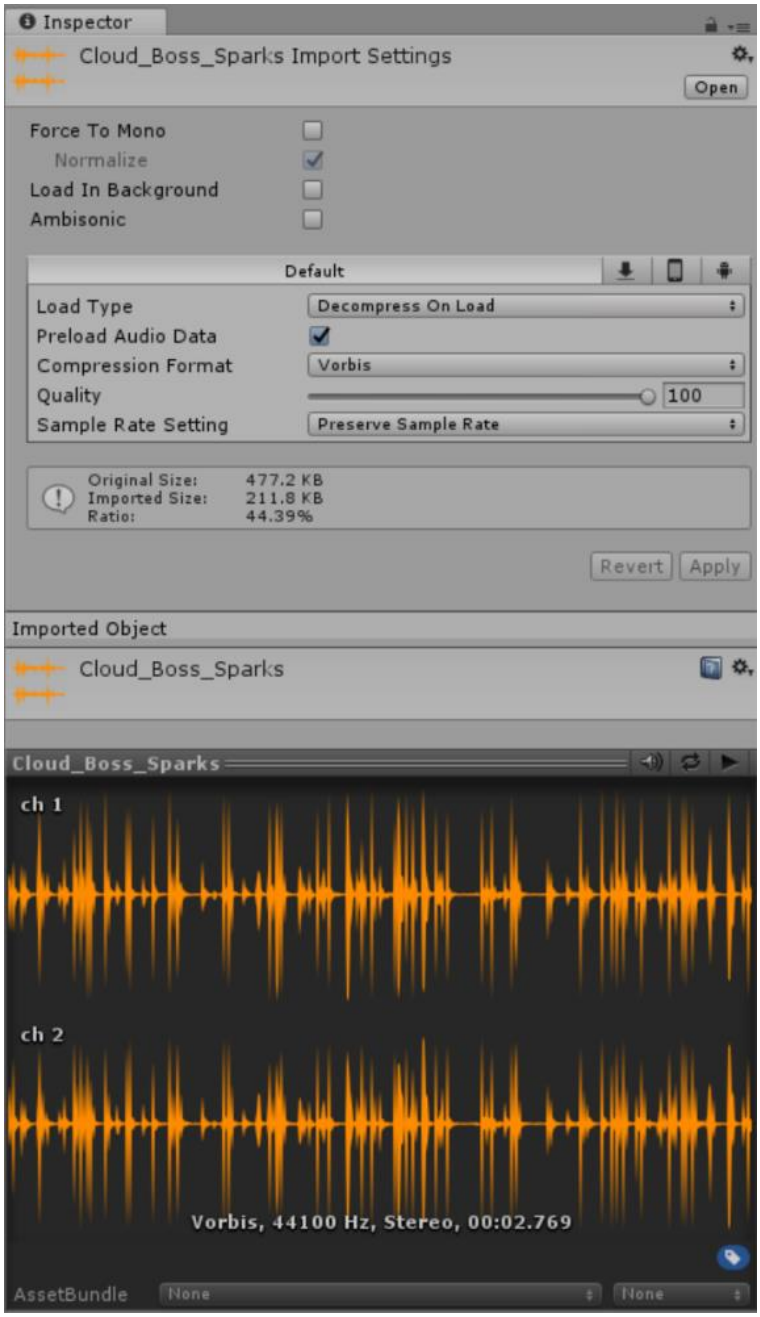

Figure 7. Import settings used for audio assets in Rapid Magic.

The Inspector shows the import settings as well as a waveform preview when selecting one or more audio assets in the project window. This view is shown in Figure 7. For Rapid Magic the decision was made to compress audio into Vorbis in Unity. Vorbis can significantly cut file sizes but this is done at the expense of audio quality (Unity Technologies, n.d.). CPU usage is relatively high (Mostovy, 2016), but it did not affect performance in this case. The Quality

setting was kept at 100 since audible fidelity loss started to occur as soon as this slider was lowered.

Preload Audio Data is checked by default. This means that the selected audio file will be loaded when the scene is loaded.

The Sample Rate Settings was also left in its default position. Generally a good practice would be to override the sample rate to 44.1 kHz, since this is the default sample rate for CDs and is supported by most soundcards (Mostovy, 2016). All audio assets were exported from the DAW in this sample rate so this omission did not end up causing any problems.

Once all audio assets have been imported correctly it is time to start implementing.

4.3 Implementing audio

All audio-related code in Rapid Magic is handled by an Audio Manager object containing an Audio Manager script component. The Audio Manager object is created when a scene is loaded.

4.3.1 Music looping

It is reasonable to expect a piece of music to have reverb. This will leave a tail of decaying audio at the end of a song or musical loop. Simply cutting the tail of the audio clip in order to loop it is not enough, since once the audio clip starts from the beginning it will be lacking reverb. This is because the audio clip starts from absolute silence. The result is a jarring transition and will clearly be heard by the end user.

To circumvent this problem, the sound designer can prepare two versions of the audio clip: one that plays only once when the section that is meant to loop starts and one that loops as long as is needed. The first clip needs no other attention except making sure to cut the clip at the exact beat it ends in. The next clip, which is meant to be able loop indefinitely, needs to have the cut off reverb tail from the first clip applied to the very beginning of the clip. By doing this the looping clip will transition seamlessly, since every time it starts from the beginning it will now include the correct amout of reverb that the end user expects to hear.

This is the way background and level music looping was implemented in Rapid Magic. Level music is a good example of such an implementation.

Each level has its own theme music. The music tracks wary in length, but they are made in a similar manner using similar instruments. Exporting the music tracks dry and without reverb from the DAW and applying reverb on the tracks in the Unity Editor instead would be a mistake in this situation. In the fantasy genre it is typical to have wide and spacious mixes. Mixing is done by the music producer in the DAW in way that leaves enough room for important instruments to be heard, and applying reverb indiscriminantly on the whole track will make it lose definition and appear muddy. This was observed during development and was

particularly offending when playing back audio on mobile devices. Therefore, music tracks should be left dry in the Unity Editor.

During a level the player might encounter various events. These events include for example a goblin bookkeeper who offers the player a wager, a rival wizard that challenges the player to a duel and a merchant character selling equipment that enhances the players stats. Each event either has its own musical theme or is categorized into “good”, “neutral” and “evil” events. The music transition is handled by if-else statements, shown in Figure 8.

```
public void PlayLevelMusic()
{
    if (muteMusic)
        return;

    if (loopTrack)
        StopCoroutine(loopTrackCoroutine);

    if (mainPlayer.volume < 1)
        StartCoroutine(FadeInMusic(1f));

    mainPlayer.loop = true;
    string sceneName = SceneManager.GetActiveScene().name.ToLower();
    if (sceneName.Contains("menu"))
        loopTrackCoroutine = LoopTrack(musicTracks[0], musicTracks[27]);
    else if (sceneName.Contains("academy"))
        loopTrackCoroutine = LoopTrack(musicTracks[3], musicTracks[4]);
    else if (sceneName.Contains("cliff"))
        loopTrackCoroutine = LoopTrack(musicTracks[5], musicTracks[6]);
    else if (sceneName.Contains("cloud"))
        loopTrackCoroutine = LoopTrack(musicTracks[7], musicTracks[8]);
    else if (sceneName.Contains("forest"))
        loopTrackCoroutine = LoopTrack(musicTracks[9], musicTracks[10]);
    else if (sceneName.Contains("village"))
        loopTrackCoroutine = LoopTrack(musicTracks[11], musicTracks[12]);
    else if (sceneName.Contains("lakeside"))
        loopTrackCoroutine = LoopTrack(musicTracks[13], musicTracks[14]);
    else if (sceneName.Contains("lighthouse"))
        loopTrackCoroutine = LoopTrack(musicTracks[15], musicTracks[16]);
    else if (sceneName.Contains("sea"))
        loopTrackCoroutine = LoopTrack(musicTracks[17], musicTracks[18]);
    else if (sceneName.Contains("desert"))
        loopTrackCoroutine = LoopTrack(musicTracks[19], musicTracks[20]);
    else if (sceneName.Contains("final"))
        loopTrackCoroutine = LoopTrack(musicTracks[21], musicTracks[22]);
    else
        loopTrackCoroutine = LoopTrack(musicTracks[3], musicTracks[4]);

    StartCoroutine(loopTrackCoroutine);

}
```

Figure 8. If-else statements handling music transitions for in-game levels.

The first level in the game, referred to as “academy” in the script, uses two elements, 3 and 4, in the Music Tracks serialized field. A collapsed view of the serialized field and its contents is displayed in Figure 9.

Element 3 corresponds to Magic_Academy_4by4_100bpm_part1, which is the clip that plays initially. Magic_Academy_4by4_100bpm_part2, which is assigned to Element 4, is the the clip that is meant to be played on loop. This technique is used on almost all music tracks in the game.

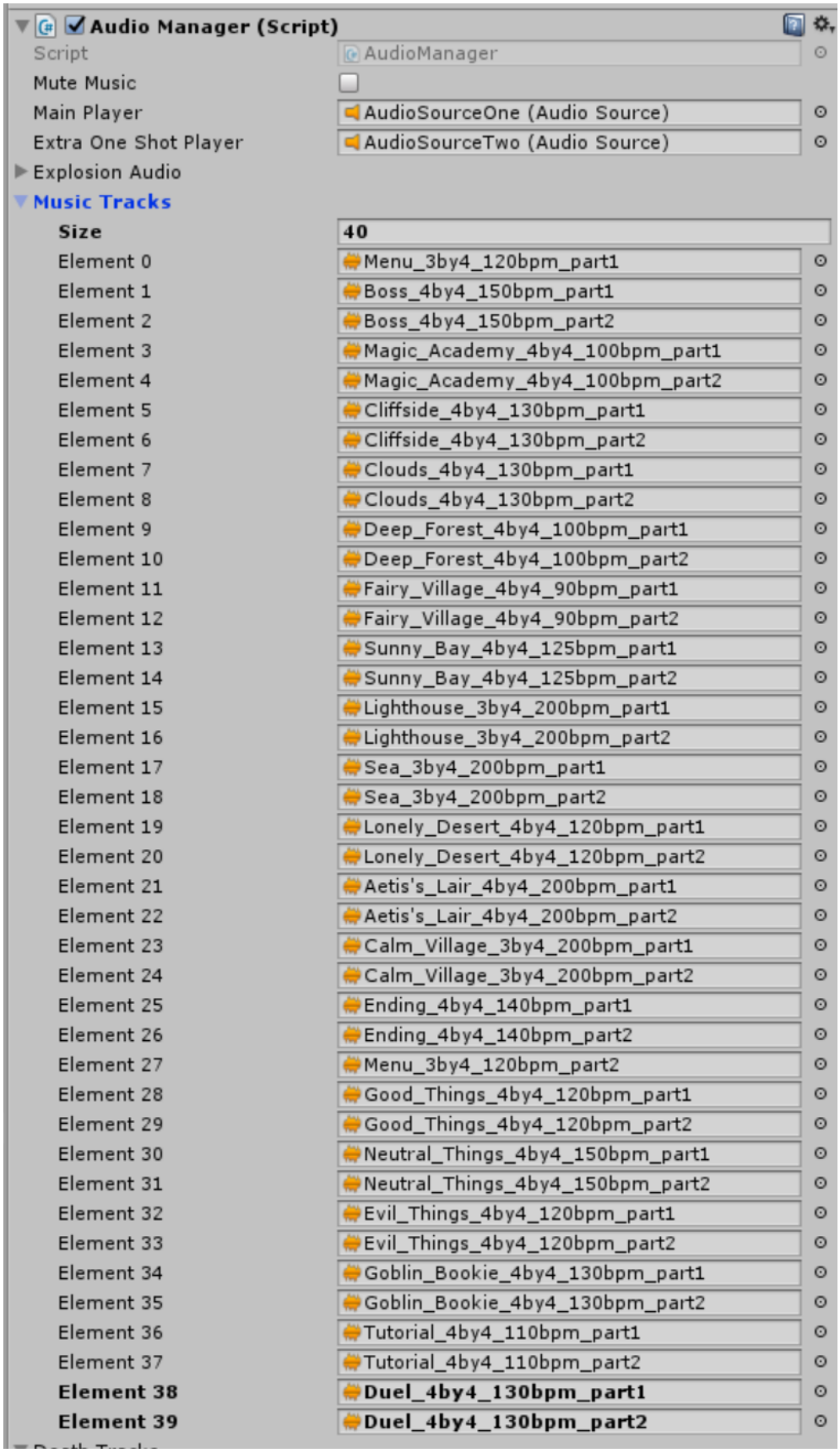

Figure 9. The collapsed view of the serialized private field Music Tracks.

4.3.2 SFX

Sound effects in Rapid Magic are used as they are and no mixing is done in the Unity Editor. All sound design, including effects, volume control and layering, was done in the sound production stage. This keeps audio assets consistent throughout the game. Also, some filters are CPU intensive (Unity Technologies, n.d.), which could become a problem on more dated mobile devices.

The SFX assets are mostly organized in serialized fields, just like the music tracks. Since almost every sound effect is played only once whenever they are triggered, no tricks to achieve seamless looping is required.

The sounds are played with the PlayOneShot method. In comparison to the Play method, PlayOneShot allows multiple sounds to be triggered at once, but they will always play all the way through. Rapid Magic can become very intensive and crowded with enemy characters, and consequently sound effects will be triggered at the same time or in succession. While the game does not rely on audio cues solely for player feedback, a missing sound effect where the player expects to hear one can throw them off.

Some actions, such as water splashes and enemies throwing items at the player, require more than one audio clip. They are often heard many times in a row, and hearing the same clip over and over again, even with varying pitches, would sound repetitive. This is remedied by using Random.Range(a, b), where “a” is the element number for the sound included in the randomisation range, and “b” is the first number excluded. This means that Random.Range(5, 8), would include elements 5, 6, and 7 from the serialized field.

5 USING FMOD

5.1 Minigolf Universe

The second audio implementation solution covered by this thesis is for the game Minigolf Universe. Minigolf Universe is a very different product gameplay-wise compared to Rapid Magic, but it is developed for the same mobile platforms by the same team. The game uses an ortographic fixed camera which gives the impression of an isometric view again placing very little importance on positional audio. The game includes six courses with nine holes. Each course, or ”world” as they are called in-game, have their own musical theme and sound effects. As of writing Minigolf Universe is not yet released to the public. The FMOD Studio version used in this project is 1.10. and the Unity version is 5.6.4.

5.2 Setting up FMOD

The latest version of FMOD Studio can be found at https://www.fmod.com/download. This is the desktop application and can be installed wherever the user chooses following the provided instructions.

The same download page also hosts downloads for the Unity Integration package. The Unity Integration package should be of the same version as the installed FMOD Studio version. After the download has finished, import the package into the project in the Unity Editor either by going to the Project window or Assets menu, clicking Import Package, then Custom Package and finally choosing the integration package from the file explorer. (Firelight Technologies c, n.d.)

Once the package has been installed into the Unity project a new FMOD tab is available on the top row of the editor GUI. Clicking on Edit Settings will bring the settings up on the inspector window, from where the FMOD Studio project path can be specified.

It is also a good idea to disable Unity’s audio engine from the project settings for optimal performance. The settings are show in Figure 10.

After removing the Audio Listener component and replacing it with the FMOD Studio Listener script, FMOD events can be played and heard in the engine.

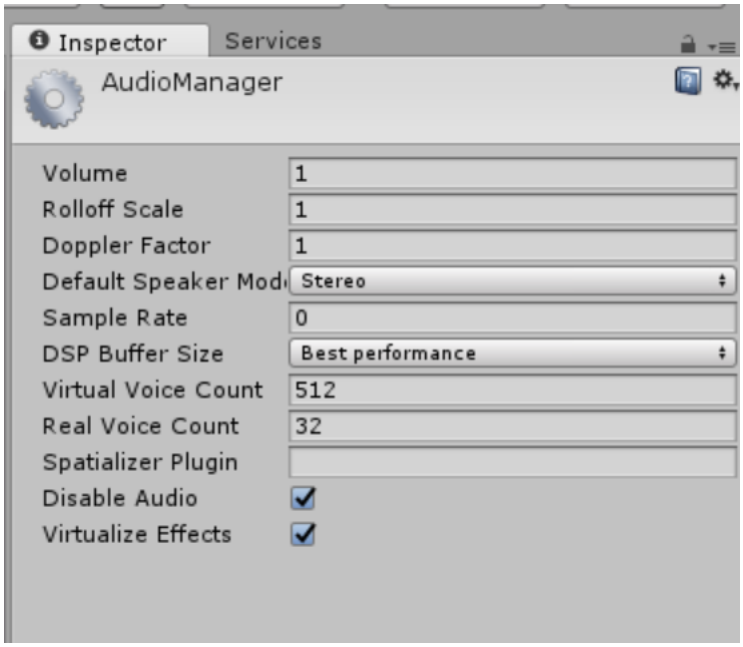

Figure 10. Project settings for audio.

5.3 Implementing audio

The sounds are handled by Music and Sound Effect manager scripts that control when FMOD events are played.

5.3.1 Music looping

Seamless looping is achieved by using an FMOD Studio event. In FMOD Studio one audio clip can be inserted twice and positioned so that the clips overlap, demonstrated in Figure 11.

It is important to be aware of the BPM and time signature of the tracks in order to line them up on beat.

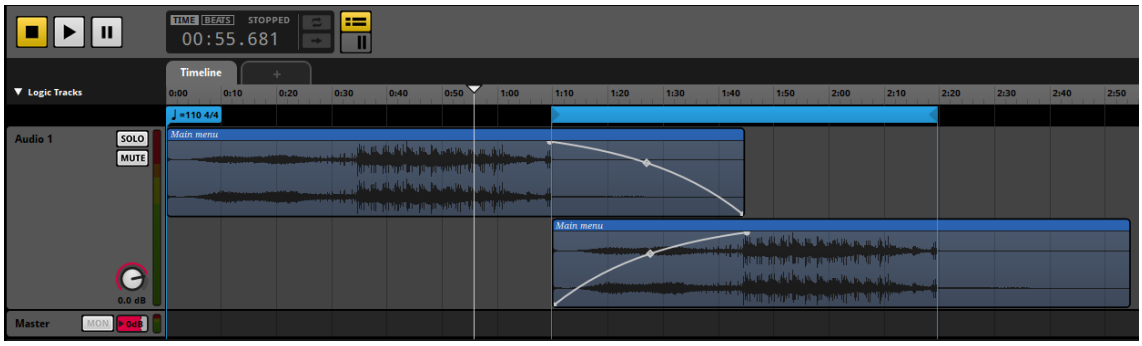

Figure 11. The main menu audio event in FMOD Studio.

Next, automation curves controlling volume levels can be added to the tracks to achieve a cross-fade. In this particular case there are no percussion instruments and only a long evolving stretched-out chord during the transition. This means that matching the beat precisely is less important and the sound designer can choose a good spot for the cross-fade based on what sounds the best.

After the event has played over the transition part once it enters a loop region. The loop region, displayed on the timeline in Figure 11, forces the marker to return to the region's beginning as it reaches the regions end. The end point is at the same spot on the second clip as the beginning point is on the first. This allows the loop to play continuously without audible seams.

In practice, this means that once the event is triggered in code, it will play the first part and then the loop region until the event is stopped. In other words, when a scene is loaded, the event is triggered, and then stopped in code before loading the next scene.

5.3.2 SFX

The ball makes a sound when it is launched and when it connects with obstacles or walls in a level. An example level is displayed in Figure 12. The sound effect is made by playing a kalimba, which is a tonal instrument originating from Africa. Normally, the way to achieve variation in a sound effect is to vary the pitch slightly by using a script. This did not work in Minigolf Universe, since all music is made in the same key, and using tonal SFX in a random pitch against background music results in dissonance. Dissonance creates tension (Timms, n.d.) and since Minigolf Universe aims for a calm atmosphere, this had to be avoided.

Figure 12. The first level of Minigolf Universe.

This meant that the sound effects for the ball needed to be chosen so that they produce a harmony together with the background music. The key used was A, and 5 notes from that key were chosen for the ball.

The sounds were then assigned to a Multi Instrument in FMOD. A Multi Instrument is a container for sounds, created when multiple audio assets are imported into one audio track. The parameters and settings for this instrument are shown in the bottom of the GUI, as displayed in Figure 13.

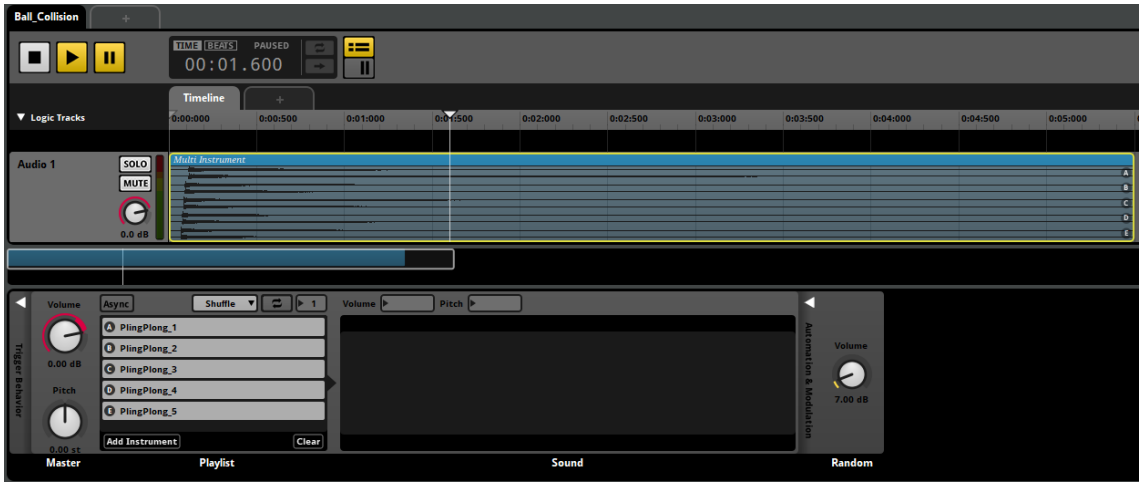

Figure 13. A Multi Instrument in FMOD Studio.

The mode is set to Shuffle, which plays the assets in a random order with a no-repeat behaviour.

When the Ball_Collision event is triggered by code in the Unity game engine the aforementioned functionality is already present without any additional programming.

6 CONCLUSION

Audio implementation is an important process that affects the end user experience greatly. Ideally a larger development team with a more flexible budget would employ full-time audio implementers, but in medium and small-sized companies, this is not possible. The thesis set out to research and compare two different ways to implement audio in the Unity game engine, and from a sound designer's perspective, one is clearly superior to the other.

In the case of the two game projects and their audio implementation processes, the one using audio middleware, Minigolf Universe, went significantly smoothly. FMOD Studio allows the sound designer to take a more active part in the development process, instead of a passive audio asset provider. This ensures that fewer creative decisions concerning audio implementation are made by a programmer who might not be familiar with audio work at all. FMOD also allows the creation of fairly complex audio events without the help of a programmer, which frees up time for other development tasks. Both games ended up with polished audio and a well-made implementation, but in Minigolf Universe, audio transitions and sound effects seemed more seamless and professional. While a learning curve is no doubt present, FMOD Studio is still very intuitive for someone who has prior experience with any mainstream DAW.

However, licensing fees for a company such as the commissioner of this thesis will make it impractical to use a solution such as FMOD Studio commercially. Mobile game development cycles are ideally fast and short. Even with the most affordable license scheme, every game would incur a fee of $2,000 after the first free game of the year. With a target development rotation of two months per game, this would result in $10,000 of licensing fees alone every year. This is right between the fees for the Basic license, which is $5000, and the $15000 Premium license. The price of convenience is high and this system does not favor smaller games.

It would seem that the best way to proceed is to attempt to balance the cost of FMOD and the longer programming hours needed to implement audio traditionally on a case-by-case basis. FMOD is at its best in larger game projects that are developed during a longer period of time. The licensing fees will have less of an impact on larger games and the benefits that FMOD provides will certainly outweigh the costs in these cases as well.

As a logical next step, it would be wise to look into other audio middleware solutions. The commissioner could find a more economical solution from competing middleware providers that would make more sense for their business model. Fabric by Tazman-Audio and Wwise by Audiokinetic are the two largest competitors and the obvious first picks for continued research.

REFERENCES

Berklee College of Music, (n.d.) *What does an audio programmer (Video Games) do?* Accessed on 24.04.2019 https://www.berklee.edu/careers/roles/audio-programmer

Brown, Y. (n.d.) *About Audio Middleware* Accessed on 25.04.2019 http://www.yannisbrown.com/about-audio-middleware/

Firelight Technologies a. (n.d.) *Firelight Technologies FMOD Studio API* Accessed on 27.04.2019 https://www.fmod.com/resources/documentation-api?version=1.10&page=welcome.html

Firelight Technologies b. (n.d.) *FMOD Studio User Manual 1.10* Accessed 22.02.2019 https://www.fmod.com/resources/documentation-studio?version=1.10&page=welcome-to-fmod-studio.html

Firelight Technologies c. (n.d.) *Unity Integration 2.0* Accessed on 28.02.2019 https://www.fmod.com/resources/documentation-unity?version=2.0&page=user-guide.html

Firelight Techologies d. (n.d.) *Featured Games* Accessed on 27.04.2019 https://fmod.com/games

Firelight Techologies e. (n.d.) *Licensing* Accessed on 27.04.2019 https://www.fmod.com/licensing#premium

Hass, J. (n.d.) *Introduction to Computer Music: Volume One* Accessed on 24.04.2019 http://www.indiana.edu/~emusic/etext/studio/chapter2_effects.shtml

Horowitz, S. and Looney, S. (2017) *Masterclass: Using Game Audio Middleware* Accessed on 23.04.2019 https://www.emusician.com/how-to/masterclass-using-game-audio-middleware

Merriam-Webster, (n.d) *Dictionary* Accessed on 28.04.2019 https://www.merriam-webster.com/dictionary/cross-fade

Mostovy, A. (2016) *Making Your Unity Game Scream and Shout and Not Killing It in the Process* Accessed on 23.04.2019 https://medium.com/@a.mstv/making-your-unity-game-scream-and-shout-and-not-killing-it-in-the-process-673a7384693c

Mulesoft. (n.d.) *What is an API? (Application Programming Interface)* Accessed on 23.04.2019 https://www.mulesoft.com/resources/api/what-is-an-api

Ryan, V. (2010) *ISOMETRIC DRAWING AND DESIGNERS* Accessed on 22.04.2019 http://www.technologystudent.com/prddes1/drawtec2.html

Stevens, R. and Raybould, D. (2016) *Game Audio Implementation: A Practical Guide using the Unreal Engine* . 1st edn. Focal Press

Strauss, R. (n.d.) *What audio format should I use for my game?* Accessed on 24.04.2019 https://indiegamemusic.com/formatguide.php

Unity Technologies. (n.d.) *Unity User Manual (2017.4)* Accessed on 24.04.2019 https://docs.unity3d.com/2017.4/Documentation/Manual/

Wolfe, J. (n.d.) *dB: What is a decibel?* Accessed on 24.04.2019 http://www.animations.physics.unsw.edu.au/jw/dB.htm

Timms, M. (n.d.) *How sound design is used to create a sense of tension and horror in video games* Accessed on 29.04.2019 https://www.academia.edu/24424213/How_sound_design_is_used_to_create_a_sense_of_tension_and_horror_in_video_games