Tampereen ammattikorkeakoulu

Creating Two Audio Features with Wwise

Red Stage Entertainment

Jaakko Liukkala

BACHELOR'S THESIS
November 2019

Business Information Systems
Game Development

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

LIUKKALA, JAAKKO:
Creating Two Audio Features with Wwise
Red Stage Entertainment

Opinnäytetyö 38 sivua, joista liitteitä 1 sivu
Marraskuu 2019

Opinnäytetyön tavoitteena oli esitellä lyhyesti äänisuunnittelun historiaa, tutkia Wwise-äänimoottorin perusominaisuuksia ja selvittää, miten äänimoottorin avulla pystytään toteuttamaan kaksi ääniominaisuutta toimeksiantaja Red Stage Entertainmentille: interaktiivisen musiikin hyödyntäminen pelissä ja pelaajan antaman syötteen sovittaminen musiikin iskuille. Tarkoituksena oli toteuttaa nämä ominaisuudet uuteen Unity-pelimoottorin projektiin ja kuvailla niiden luomisprosessia. Tarkoitus oli myös reflektoida toimeksiantajan saamaa hyötyä ominaisuuksien kehittämisestä.

Opinnäytetyön teoriaosuutta varten etsittiin aiheisiin liittyviä verkkolähteitä ja dokumentaatiota sekä analysoitiin niiden tekstiä. Käytännön osuudessa hyödynnettiin Wwise-äänimoottoria kahden ominaisuuden luomiseen. Ensimmäisessä ominaisuudessa tutkittiin Wwisen tarjoamia mahdollisuuksia interaktiivisen musiikin käyttämiseen, missä pelin musiikin osat kykenevät muuttumaan pelaajan vuorovaikutuksen ansiosta. Toisessa ominaisuudessa pyrittiin saamaan pelaajan syöttämien näppäimen painalluksien aiheuttamat vaikutukset tapahtumaan musiikin iskuilla riippumatta siitä, painaako pelaaja näppäintä liian aikaisin tai liian myöhään. Tätä tapahtumaa voidaan kutsua pelaajan syötteen kvantisoinniksi.

Opinnäytetyössä toteutettiin kaksi ääniominaisuutta Unity-projektiin ja syvennettiin Wwise-pelimoottorin perustoimintojen ymmärrystä. Toimeksiantaja oli tyytyväinen tuloksiin ja halusi hyödyntää interaktiivisen musiikin toteuttamistapoja peliprojektissaan tulevaisuudessa. Pelaajan syötteen kvantisointiominaisuus yhdistettiin toimeksiantajan projektiin jo opinnäytetyön tekemisen aikana.

Interaktiivisen musiikin toteutustapojen tutkiminen ja kehittäminen opinnäytetyötä varten meni suunnitelman mukaisesti. Kvantisointiominaisuus oli hankalampi toteuttaa ja vaati apua sekä jatkokehitystä toimeksiantajan yrityksessä työskentelevältä ohjelmoijalta.

Asiasanat: Wwise, Unity, interaktiivinen musiikki, syöte, kvantisointi

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Game Development

LIUKKALA, JAAKKO:
Creating Two Audio Features with Wwise
Red Stage Entertainment

Bachelor's thesis 38 pages, appendices 1 page
November 2019

The objective of the thesis was to briefly introduce the history of sound design, examine the basic functions of the sound engine Wwise, and to find out how the sound engine can be used to create two audio features for the client of the thesis, Red Stage Entertainment. The features required utilizing interactive music in a game and fitting player input onto the beats of a music track. The purpose was to create these features inside of Wwise and a new Unity game engine project, and to describe the creation process. Reflecting upon the benefits that the features provide for the client was also a goal.

Suitable web sources and documentation were researched and analyzed for the theory portion of the thesis. For the practical side, the Wwise sound engine was utilized in the creation of the two features. In the first feature, the possibilities that Wwise offers for interactive music were studied. Interactive music is the interaction of a player affecting the whole or parts of a music track. The second feature focused on getting the reaction of the player input to happen on the beats of a music track, regardless of whether the player gives the input too early or too late. This event can be referred to as player input quantization.

The results of the thesis were the creation of the two features into a Unity project, and a better understanding of the basic functions of Wwise. The client of the thesis was satisfied with the results and wanted to utilize the different ways of creating interactive music in their game project in the future. The feature for player input quantization was already integrated into the project of the client during the making of the thesis.

The research for the methods of utilizing interactive music and the creation of them in a project went according to plan. The quantization feature was more difficult to accomplish and required help and further development from a programmer who works at the client company.

Key words: Wwise, Unity, interactive music, input, quantization

CONTENTS

GLOSSARY

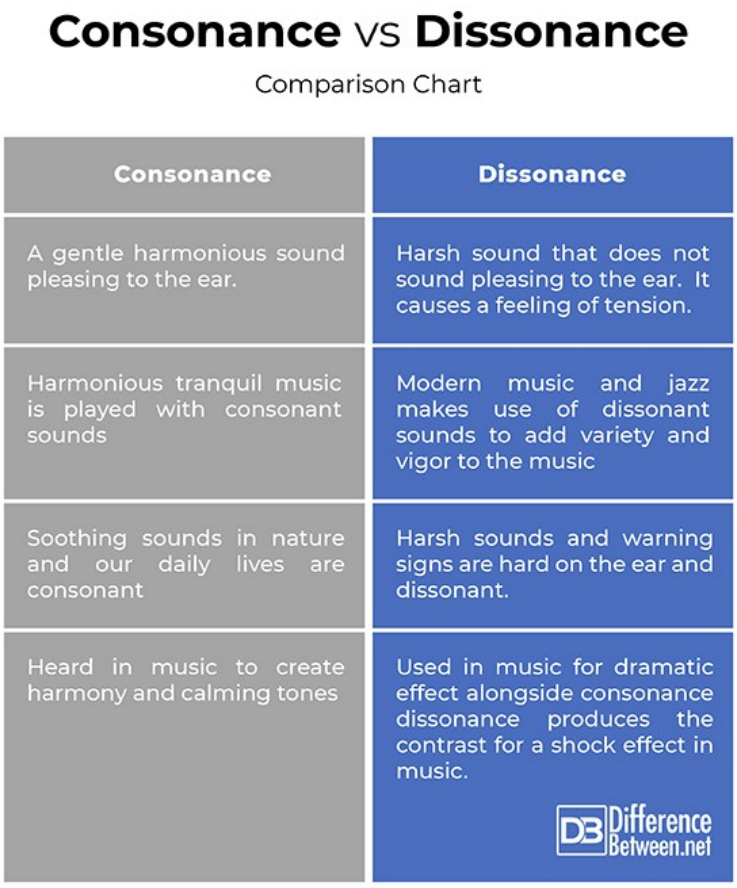
Child object	Objects that are housed under a parent object.
Collider	In Unity, allows collision on a game object and can also trigger methods by another collider entering or exiting it.
Cross-platform	E.g. programs that offer functionality for multiple platforms, such as PC, Mac, PS4 and Xbox One.
Float	In scripting, using float variables allows the use of decimal values.
Game engine	E.g. Unity, offers a framework to develop games in. (Baker 2016).
Input lag	Lag between entering an input and a receiver reacting to the input (Shafer 2019).
Middleware program	A program that acts as a connecting element between two programs and offers its capabilities to the other program (Red Hat N.d.).
Script class	Most scripting in Unity is done by creating and utilizing classes which feature variables and methods inside of them.
Script component	Can be attached to game objects, variables can be added to them via scripting and controlled from the game objects.

1 INTRODUCTION

A sound designer creates the *soundscape* for media. Soundscapes usually consists of music, *ambience*, *sound effects* and dialogue. The work requires the finding, recording, editing and mixing of all these elements. (Berklee College of Music N.d.; Metcalfe 2013.) Sound design for video games differs from movies and music, as games are a combination of audiovisual elements and player interaction (Esposito 2005). Another separating aspect is the way the sound is implemented, often demanding programming skills (Berklee College of Music N.d.).

Sounds in the earliest games were meant to make the experience of a player feel more enjoyable. Sound effects were also essential in giving feedback to the player. When the ball in the game Pong hits one of the paddles, an audio cue is given in the form of a beep. This signals to the player that they should pay attention to the next pass. (Gamedesigning.org 2018.) The game reacting to the interaction of the player is important in game sound design. As the player jumps in Super Mario Bros, a sound effect is played. This is the ultimate example of a sound signaling interaction. (WIRED 2017, 4:29-4:46.) Sound in games also helps the player immerse in the game world along with the visuals and gameplay. If the player is hearing sounds that they think they should hear in the game, depending on its setting and style, it can be easier to get into the game without getting distracted. (Campbell 2016.)

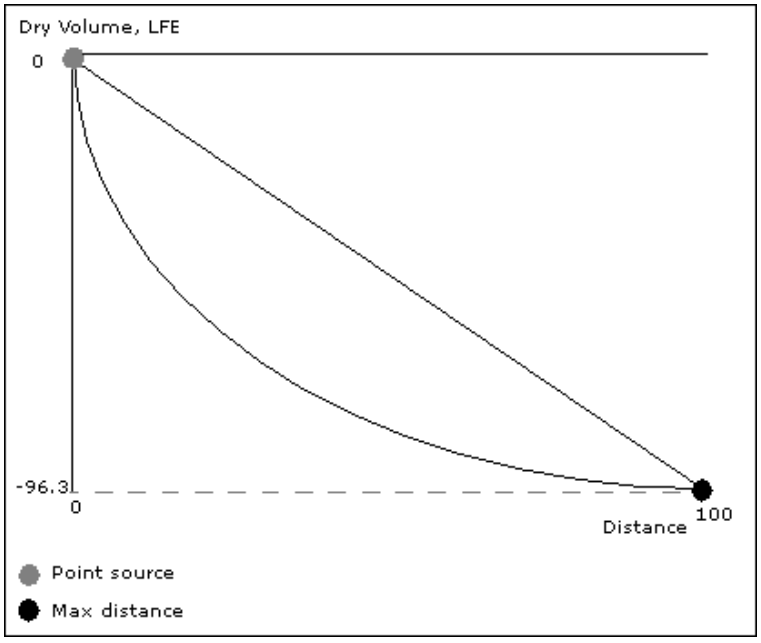
A game's music and sound effects can relay the sorts of emotions the player is intended to be feeling through how they sound. The emotions evoked can be positive, such as the feeling of succession through *consonance*, or negative, such as the feeling of failure through *dissonance* or anything in between (WIRED 2017, 3:57-4:16). Picture 1 presents a simple comparison of consonance and dissonance.



PICTURE 1. Consonance versus dissonance chart (Wither N.d.)

The space that was made available for storing music and sound effects in games has increased from a few kilobytes to multiple gigabytes over the years. For music, using simple *waveform* sounds and only a few *instrument tracks* was necessary to keep memory usage to a minimum, while now it is common to have many lengthy and complex orchestral compositions, recorded with real instruments, comprise the soundtrack of a game. Sound effects have changed from bleeps to actual sounds heard in real life recorded with microphones. Game sound designers have been able to utilize techniques largely used in film, such as creating *foley* sounds. (ACMI N.d.) Foley sound effects are used to recreate commonplace sounds in a studio or otherwise quiet environment to ensure pristine audio quality (ACMI N.d.; Stinson N.d.). This often requires unconventional ways of achieving the desired sound, for example imitating the footsteps of a fox by using a glove with paper clips attached to each finger (Hogan 2014).

Games evolving constantly introduced new audio problems that required solving. One of those was *positioning*. Sounds needed to only trigger and be audible when the player was near an object that was producing sound, e.g. an enemy. (WIRED 2017, 8:26-8:36.) This aided with saving resources that would be required by all the sounds playing over and over. With the rise of 3D games, 3D positioning of sounds became necessary. Sound effects were to be heard from the correct position relative to the position of the player, making the player a *listener* and all the objects producing sound, *audio sources*. For example, if the player is standing to the left of a sound-emitting object, the sound effect needs to be heard from the right side of the headphones or speaker system. (Cullen N.d.) *Looping* sounds, like a waterfall, need to have their volume increase as the player gets close, and decrease as the player walks away from their vicinity, otherwise known as *distance-based attenuation* (Audiokinetic Applying... N.d.). Picture 2 shows how this works. All these advancements made more realistic and inventive sound design possible in games.



PICTURE 2. Distance-based attenuation (Audiokinetic Applying... N.d.)

Nowadays, *interactive music* and real-time effects on sounds offer many possibilities for reacting to player interaction and giving feedback. Music in the game can switch to another track depending on e.g. the progress or the state of the player character in the game. A music track can also be split into many individual instrument tracks and have different game events *triggering* the

addition or removal of these instruments while the music is playing. (Audiokinetic Understanding Interactive... N.d.; Sweet 2016.) Sound effects can have effects put on them in real-time, such as a *low-pass filter* or *reverb* (Audiokinetic Using Real-time... N.d.). A low-pass filter cuts off higher audio frequencies giving a sound a deeper quality. The amount of the effect can be adjusted. (Audiokinetic Glossary N.d.) Reverb is an effect “that simulates the acoustics of a particular room or space” (Audiokinetic Glossary N.d.).

Audio *middleware programs*, such as FMOD and Wwise, were created by audio design professionals to help streamline the game sound design process. These *audio engines* give sound designers tools and features to create dynamic and varied audio in games. They nowadays *integrate* into many *game engines*, such as Unity and Unreal Engine, with the integration also providing custom *scripting* and *components* for the user. (Audiokinetic About N.d.; FMOD N.d.; Audiokinetic Using the... N.d.)

The aim of this thesis is to utilize one of these audio engines, Wwise, to develop two audio features that fit the needs of the client, Red Stage Entertainment. The first feature is the utilization of interactive music, with the other one being player *input quantization*. Interactive music is music that responds to player actions. Music is created for the first feature, and a few interactive music methods are applied to the song with Wwise. For the second feature, a script that fits the input of a player onto the *beats* of a music track is created. Both features use the Wwise integration into Unity.

Describing the process of development and reflecting on what is accomplished is important. The reflection portion also includes the analysis of how useful the developed features are for Red Stage Entertainment. Seeing as the client only uses Unity for developing their games at this moment, how Wwise integrates into other game engines, such as Unreal Engine, is left out of the thesis. The programming side is only referenced as much as required. In order to help clarify the terms and features that Wwise has, the next section of the thesis is about the basics of Wwise.

2 WWISE BASICS

2.1 Overview

Audiokinetic, a company comprised of music, film and gaming industry experts, released the first commercial version of Wwise in 2006. They wanted sound designers to be able to give players more immersive experiences in games in terms of audio. Wwise offers *cross-platform* solutions for sound and many tools for creating interactive and varied audio. (Audiokinetic About. N.d.) Today, the middleware audio engine can be downloaded for free from the Audiokinetic website.

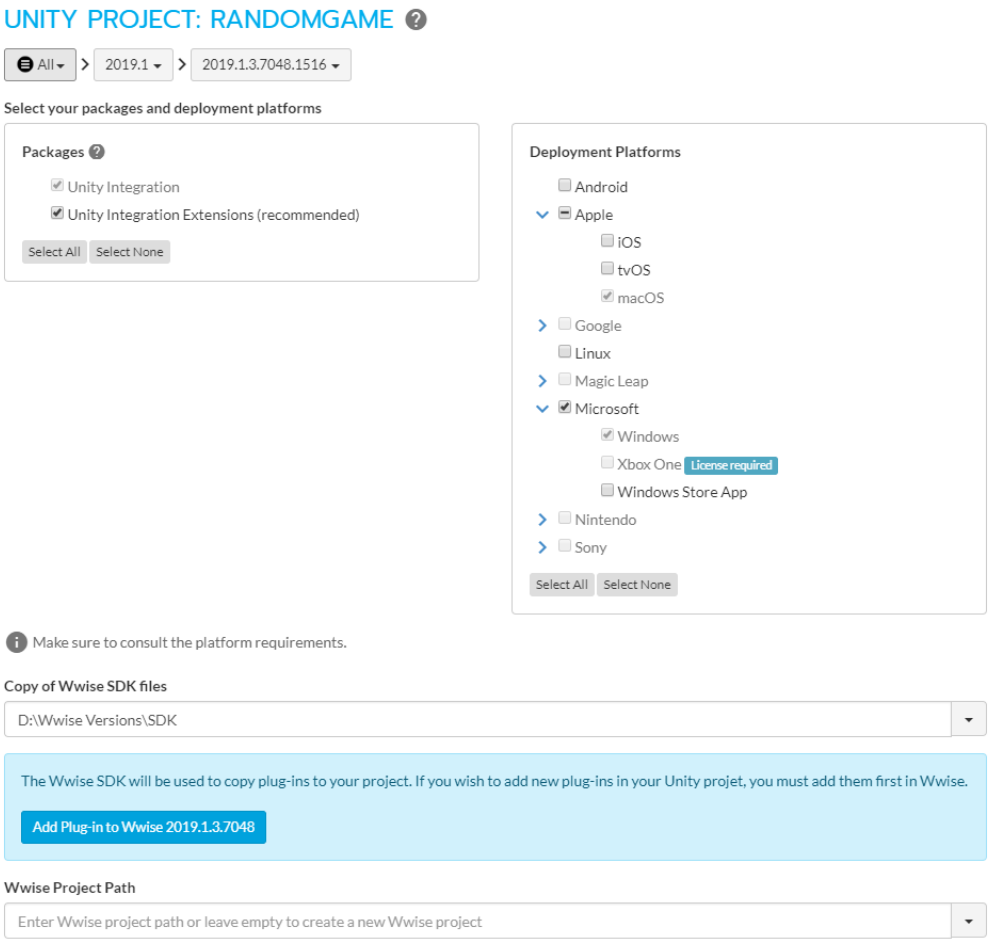
This thesis only explains the basic functions of Wwise that are essential for the creation of the two features intended for Red Stage Entertainment: a solution for interactive music and player input quantization. The essentials consist of the installation of the audio engine, the *importing* of audio assets, the creation and utilization of *events*, *sound banks* and *game syncs* as well as the introduction to the integration of Wwise into Unity. The functionality for interactive music that Wwise offers will be introduced and explored during the creation of the first feature.

2.2 Installation

The *Wwise Launcher* program can be downloaded from the Audiokinetic website. After installing and starting the program up, the newest version of Wwise can be downloaded from the Wwise tab in the Launcher. A pre-existing Unity project can have Wwise integrated into it by going into the Unity tab in the Wwise Launcher and selecting the *Integrate Wwise into Project...* option on the intended Unity project.

Before going ahead with the integration, a version of the Wwise Unity integration and SDK files that correspond to the installed Wwise version need to be installed. The Wwise Launcher will give advice on how to do this. SDK is an acronym for

software development kit, and is the way Wwise provides their tools, scripting and documentation for the integration (Sandoval 2018). Once the installation is done, the platforms planned for the project are to be declared. The folder path for a Unity installation needs to also be set. After everything is done, the *Integrate* button can be pressed to commence the integration. The integration window is shown in picture 3.

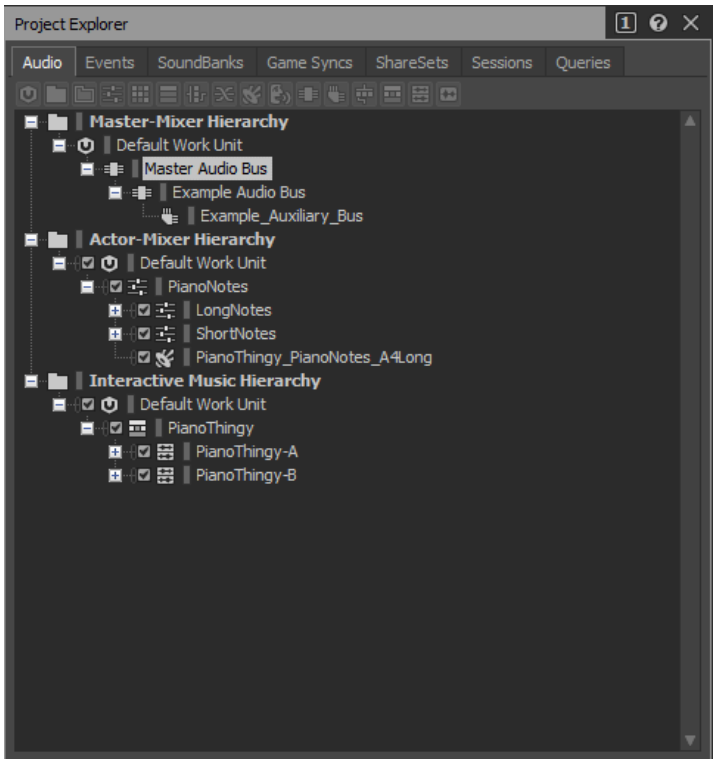


PICTURE 3. Adding Wwise integration for a Unity project in the Wwise Launcher

2.3 Assets

The *Project Explorer* window is where the audio files brought into the Wwise project can be found. The window features multiple tabs which all house different assets, with the imported audio files being shown in the Audio tab. The Audio tab is split into three sections: The *Master-Mixer Hierarchy*, the *Actor-Mixer Hierarchy* and the *Interactive Music Hierarchy* (picture 4).

These hierarchies, and most elements in Wwise, feature a *Default Work Unit*. Work Units are used for splitting the workload so that multiple team members can work on the same Wwise project simultaneously. They have the relevant information of a specific element, such as the Actor-Mixer Hierarchy or *Events*, inside of them. Default Work Units are created for the project by default. They need to exist for a user to be able to create more objects for an element in the project. (Audiokinetic What are Work Units? N.d.)



PICTURE 4. The Audio tab view of the Project Explorer window

The Master-Mixer Hierarchy is for audio and auxiliary buses. A *Master Audio Bus* is created in a Wwise project by default, and all the audio is routed through it. This means that by e.g. changing the volume of a Master Audio Bus, the changes apply to all sounds if they are routed through that audio bus. (Audiokinetic The Master Audio… N.d.) Sound effects, dialogue, music and ambience are often assigned to separate audio buses so that each type of audio can be controlled individually (Audiokinetic Structuring a Bus… N.d.). These new audio buses are *child objects* of the Master Audio Bus, meaning all the audio in the project can still be globally changed via the master bus. Auxiliary buses can have effects put on them and are used for what Wwise refers to as *environments*, more commonly known as *reverb zones*. (Audiokinetic The Master Audio… N.d.) They are zones

where effects are applied, most often a reverb effect, to create the necessary *echo* and *delay* for sounds in that specific environment. For example, a large empty cave might require huge reverb with plenty of echo, delay and perhaps other effects. (Audiokinetic Understanding Sends N.d.)

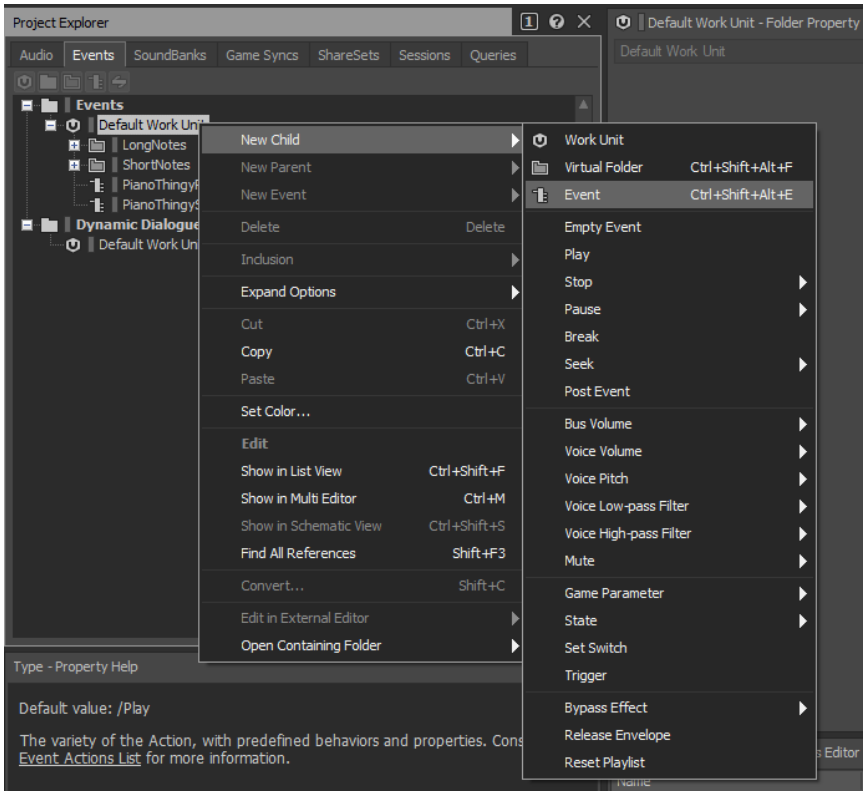
Sound effects and dialogue are most commonly imported into the Actor-Mixer Hierarchy while the Interactive Music Hierarchy houses the music and ambience tracks of the project. Audio file importing happens by right-clicking on the Default Work Unit or a folder-type object inside it in either the Actor-Mixer Hierarchy or the Interactive Music Hierarchy and selecting *Import Media Files*. New objects can also be created by dragging files into either hierarchy's Default Work Unit (Audiokinetic Importing Media Files N.d.).

Depending on which hierarchy the audio files are imported into, different types of Wwise objects are created (Audiokinetic Creating Wwise Objects… N.d.). *Sound SFX* and *Sound Voice* objects relate to the Actor-Mixer Hierarchy, and are usually meant for sound effects and dialogue, respectively. Sound SFX objects can be music and ambience as well. Importing audio files into the Interactive Music Hierarchy creates Music objects, either in the form of a *Music Segment* object, a *Music Track* object or a *Music Clip* object. Music Segments can have multiple Music Tracks inside of them, and Music Tracks are composed of a single, or multiple Music Clips which are a representation of the imported audio file. (Audiokinetic Glossary N.d.)

2.4 Events

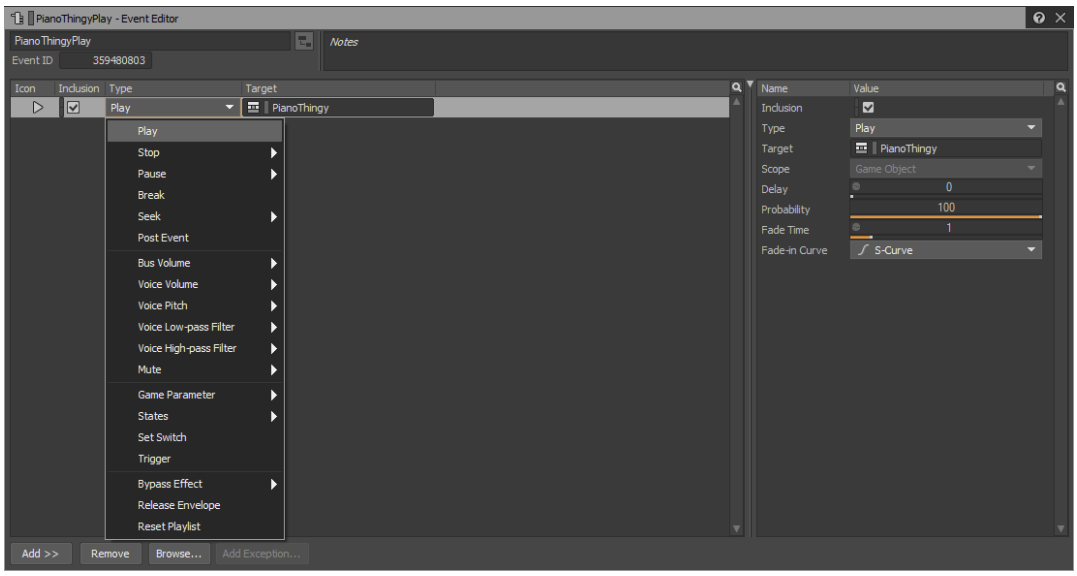
Events are objects that perform an *action* or multiple actions for the Sound SFX, Sound Voice and Music objects assigned to them. The *type* of action can simply be playing or stopping an audio clip, but it can also be something more complex, such as setting the volume of a clip to a different level or changing the amount of an effect that is being put on an audio clip. Some event types can affect audio in a game *globally*, e.g. pausing or stopping all sounds. Events reference the audio objects assigned to them and are posted in a game engine to perform the action or actions that they are set to do. (Audiokinetic Understanding Events. N.d.)

Events are accessible through the Events tab in the Project Explorer window. A new event can be created in multiple ways (picture 5). The desired Sound or Music object must be assigned to the event for it to perform the action on that specific object. This can be done from the *Event Editor* window.



PICTURE 5. Adding a new empty event into the Wwise project

According to the Audiokinetic Event Actions List (N.d.), the options for what an event action type has are different for nearly all actions. The most common *properties* are *target*, *scope*, *delay* and *fade time*. For example, the Play action type features properties for the target of the event, the scope of the event action, the initial delay after *posting* the event before the audio begins playing, the probability of the event action happening and the duration and *curve shape* of a *fade-in* that helps with smoothly transitioning into the audio clip (picture 6). The scope of the action can be global or within one *game object*.

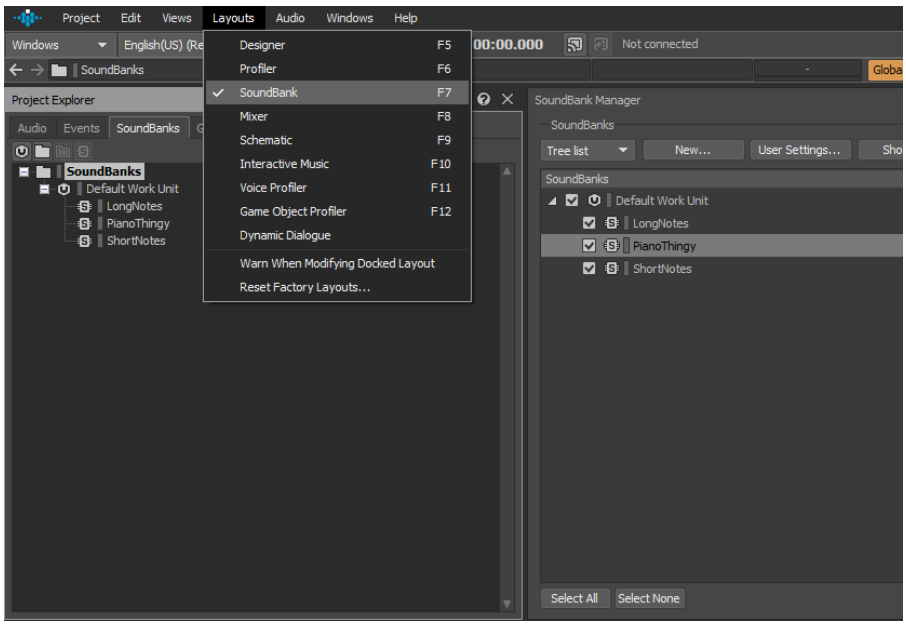


PICTURE 6. Properties related to the Play event action shown on the right

2.5 Sound banks

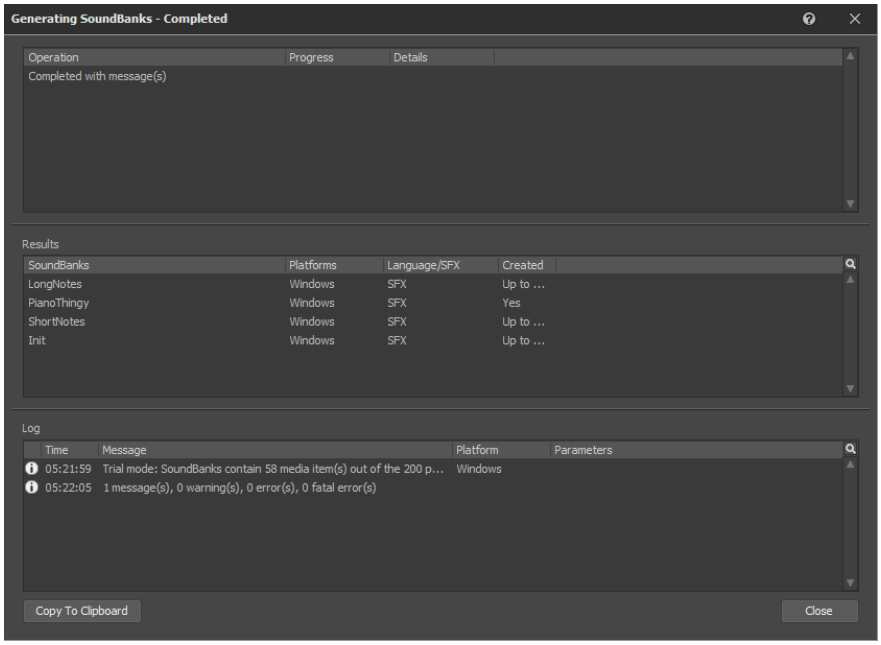
The Audiokinetic Glossary (N.d.) which explains terms related to Wwise and audio in general defines a sound bank in this way: “A group of Event data, Sound, Music, and Motion structure data, and/or media files that will be loaded into the game's platform memory at a particular point in a game.” Sound banks are used to store the data of audio files and events that are created in the project. The audio of a project can be divided into many sound banks that are *loaded* and *unloaded* into the game when needed. There is also an *initialization bank* that has the basic information of a project in it (Audiokinetic Understanding SoundBanks N.d.). *Generating* these sound banks inside of Wwise is the means of getting the data onto the side of the game engine.

New sound banks can be created in a similar way to the creation of events. The *SoundBank Manager* window showcases the sound banks in the project. It can be viewed by making sure the *SoundBank layout* is presented. As shown in picture 7, this is done by pressing the *Layouts* button in the top-left corner options and selecting SoundBank.



PICTURE 7. How to change layouts in Wwise

When a new sound bank has been created, events can be added into it by e.g. dragging events from the Events tab in the Project Explorer window into the sound bank. Generating sound banks is done from the SoundBank Manager window. Having a sound bank checkmarked means it will be a part of the generation process. The *Generate Selected* button at the top of the window can be pressed to start the generation. This opens a separate window that informs the user of the progress of the sound bank generation (picture 8). To simply generate all the sound banks of a project, *Generate All* can be selected.

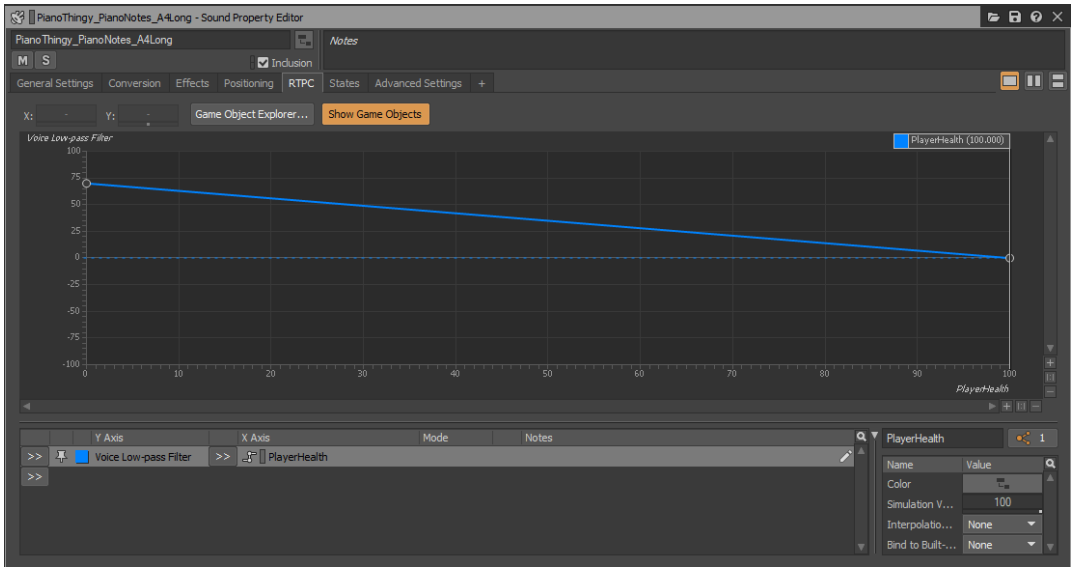


PICTURE 8. Wwise sound bank generation window

2.6 Game syncs

Game syncs are elements that are affected by what is happening in a game, and then influence the audio of the game in a set way (Audiokinetic Glossary N.d.). The game syncs that are necessary to explain for the thesis are *switches*, *states* and *game parameters*. Switches change a sound to another while states change the properties of one sound. Games often feature multiple *surface materials* for the footsteps of a character, such as grass and dirt. Usually, there are multiple footstep sounds for each material, and switches are used for switching between these collections of sounds. (Audiokinetic Glossary N.d.) States are used to e.g. add effects to a sound or change the volume of it (Audiokinetic Glossary N.d.). The change from one state or switch to another is done via programming.

Game parameters can cause real-time effects on sounds via *real-time parameter controls*, abbreviated as *RTPCs* (Audiokinetic Glossary N.d.). One way to utilize them is to link a game parameter to the health of the player in a game. With an RTPC, a low-pass filter effect can intensify as the player's health decreases (picture 9). The game parameter can then be assigned to respond to changes in player health through code. This makes for dynamic feedback to the player in terms of audio.



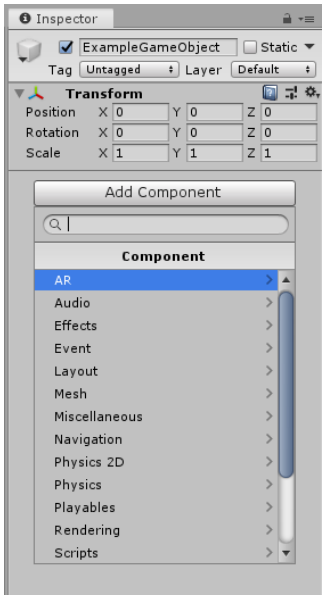
PICTURE 9. An RTPC with a low-pass filter and a game parameter

2.7 Unity integration

If a Unity project has Wwise integrated into it, Wwise-based scripting is made available for use in the project. This makes control of sound banks, events, game syncs, etc. possible on the side of Unity via the *script components* of Audiokinetic and the ability to create custom code with Wwise *script classes*. There are a few components that are required to be used in the Unity project for the functions of the Wwise integration to work.

The *WwiseGlobal* object initializes Wwise functionality in the Unity project by referencing the *AkWwiseInitializationSettings* asset and is automatically added to every *scene* in Unity by default. (Audiokinetic Using the Wwise Unity Integration N.d.) The Unity documentation on Scenes (N.d.) states that Scenes can be thought of as the levels in a game that is being developed. They are where the work is done, and where objects, characters and environments are placed. The Scene window displays a visual representation of the scene for the user.

With a new game object selected and the *Inspector* window open, the components of an object are visible. A desired component can be added to a game object via the *Add Component* button. The ability to search for specific components helps with finding the correct one (picture 10).



PICTURE 10. The Inspector window in the Unity Editor

An *audio listener* component is required to make positioning in a Unity project possible. For the Wwise integration, the default audio listener of Unity, located in the Main Camera object, is replaced with the *AkAudioListener* component. (Audiokinetic Using the Wwise Unity Integration N.d.)

Many of the other Wwise-specific components are for utilizing objects created in Wwise on the side of Unity, such as loading sound banks, playing sounds and controlling game syncs (Audiokinetic Using the Wwise Unity Integration N.d.). For a Wwise event to play, the sound bank it is assigned to must be loaded in the Unity scene. This is done by adding the *AkBank* component to a game object. There are multiple options for when the bank should be loaded and unloaded, e.g. upon the start of a scene, entering a *collider* or the destruction of the game object the component is assigned to. The intended sound bank can be assigned by pressing the box next to the *Name* option.

A sound bank, much like all the Wwise elements brought over to Unity through the integration, can also simply be dragged into the Inspector window of a game object from the *Wwise Picker* (Audiokinetic Using the Wwise Unity Integration N.d.). This adds an AkBank component to the game object. The Wwise Picker window conveniently shows all Wwise objects as a list.

Events can be played in multiple ways. The components *AkEvent* or *AkAmbient* can be attached to game objects, and then set to play a specific event. Like the AkBank component, how the event is triggered can be stated. An event can also be dragged from the Wwise Picker into a game object.

On the scripting side, the *AkSoundEngine* class can be used for more general code. The class contains many of the different *functions* available for use in the Wwise integration. (Audiokinetic Using the Wwise Unity Integration N.d.) An example of using the AkSoundEngine class to load a bank and then play an event in the game would look like the code in picture 11.

```
0 references
void Start()
{
    AkSoundEngine.LoadBank("ExampleBank", AkSoundEngine.AK_DEFAULT_POOL_ID, out uint bankID);
    AkSoundEngine.PostEvent("ExampleEvent", gameObject);
}
```

PICTURE 11. Utilizing the AkSoundEngine class

For the *LoadBank* function, the name of the desired sound bank needs to be inserted as the first *parameter* in *string* form. String variables require the text to be wrapped around quotation marks. The second parameter refers to what memory pool the sound bank should be in. The last parameter is required for the bank load request to go through. (Wwise SDK LoadBank N.d.) The *PostEvent* function requires the name of the desired event as a string and the game object the script is attached to as a component, to be the parameters. By writing *gameObject*, Unity knows to reference the correct game object.

There are separate custom classes known as *Wwise Types* for sound banks, events, RTPCs, states and switches that can be *declared* and used as variables (Audiokinetic Using the Wwise Unity Integration N.d.). The example code in picture 12 demonstrates the use of Wwise Types to declare sound bank and event variables, and then load the sound bank and play the event.

```
public AK.Wwise.Bank ExampleBank;
public AK.Wwise.Event ExampleEvent;

0 references
void Start()
{
    ExampleBank.Load();
    ExampleEvent.Post(gameObject);
}
```

PICTURE 12. Utilizing Wwise Types

Public variables show up as options on a component in the Inspector window of the Unity Editor. They can also be referenced in any other script freely. After declaring the variables at the start of a script, they can be used in a desired function. The Start function in the example uses the functions of the *Bank* and *Event* classes to load a sound bank and post an event from the game object, whose component is this script. Finally, the user needs to set the references to the intended sound bank and event on the component from the Unity Editor.

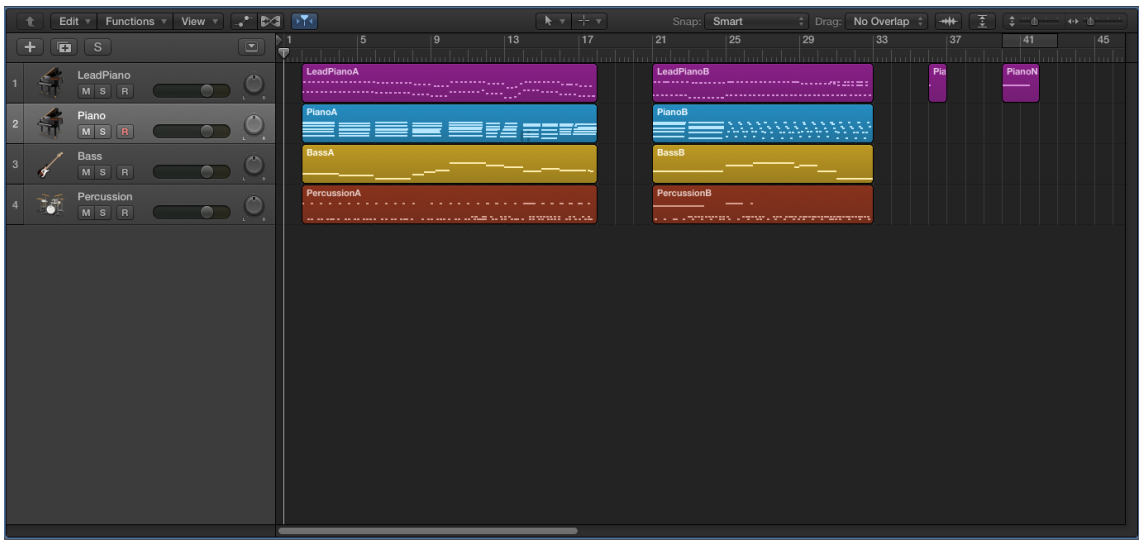
3 INTERACTIVE MUSIC

3.1 Starting out

Interactive music in Wwise can be divided into two structures: a *horizontal* and a *vertical* project structure. In a horizontal structure, the order of the *segments* of a music track can be changed by e.g. introducing sequences of randomness for certain segments. A structure like this requires multiple segments of music to make it varied. With a vertical project structure, the music is made varied by having instrument tracks added to and removed from the music score. This can be used to e.g. build a music track up with layers of instruments as a player progresses through a level in a game. Another way to utilize the structure is by using *sub-tracks*, which are different versions of the same instrument track that can be e.g. switched to after a player interaction. A vertical structure requires all the instrument tracks of a music segment, and their possible other versions, to be brought into Wwise separately. (Audiokinetic Interactive Music Project... N.d.)

The structures can be combined by e.g. making multiple music segments that all have different versions of each instrument track that they contain (Audiokinetic Interactive Music Project... N.d.). This is a very multiphased way of creating interactive music, but it offers many possibilities for varying up the music both randomly and via player interaction.

The first feature starts off with the creation of a music track in a digital audio workstation program, *DAW*, called Logic Pro X. DAWs allow for music production via software by offering solutions for recording, *mixing* and *mastering* audio (Case 2014). The track features two sections that both have four instrument tracks (picture 13). The individual instrument tracks are *exported* out of Logic Pro X as audio files, adding up to eight files in total for the music.



PICTURE 13. The *multitrack* view in Logic Pro X showing the instrument tracks of the music created for the feature

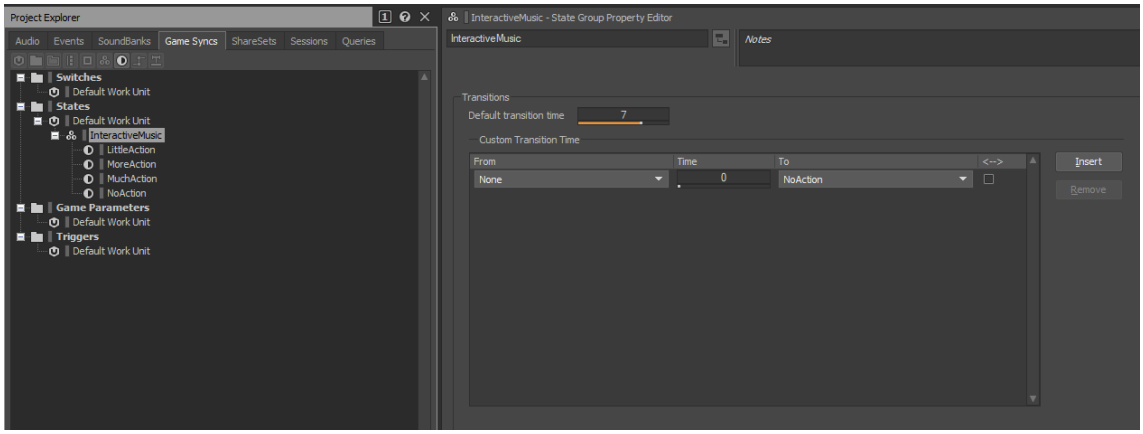
In Wwise's Interactive Music Hierarchy, a *Music Playlist Container* object called *PianoThingy* is created, and two empty Music Segment objects are put inside of it as child objects. The Music Segment objects represent the two sections of the track, *PianoThingy-A* and *PianoThingy-B*. Four audio files are imported into both Music Segments as Music Track objects, representing the different instrument tracks of the sections. By selecting the *PianoThingy* parent object, the editor window for the object opens. On the *General Settings* tab, there is a *Time Settings* section where the *tempo* and *time signature* of the song can be set. The song made for the feature has a tempo of 126 *beats per minute* and is in 4/4 *time*. Setting the time settings on the parent object will pass the settings down to the child objects. Selecting the PianoThingy-A object that has the four Music Track objects inside it and pressing *F10* to switch to the *Interactive Music* layout opens a window that shows a multitrack view of all the audio files. This *Music Segment Editor* window is shown in picture 14.

PICTURE 14. The Music Segment Editor window in Wwise showing the first section of the music created for the feature

The window offers options for the individual tracks, such as *muting* them or changing their volume level. The green and red vertical lines with arrows at the top indicate the start and end positions for a Music Segment, also known as *cues*. The positions can be changed by dragging a cue arrow to the desired position. By making sure *Bars/Beats* is selected in the *Snap to* option, the cues and visual representations of the audio files will snap to the bars and beats of the music.

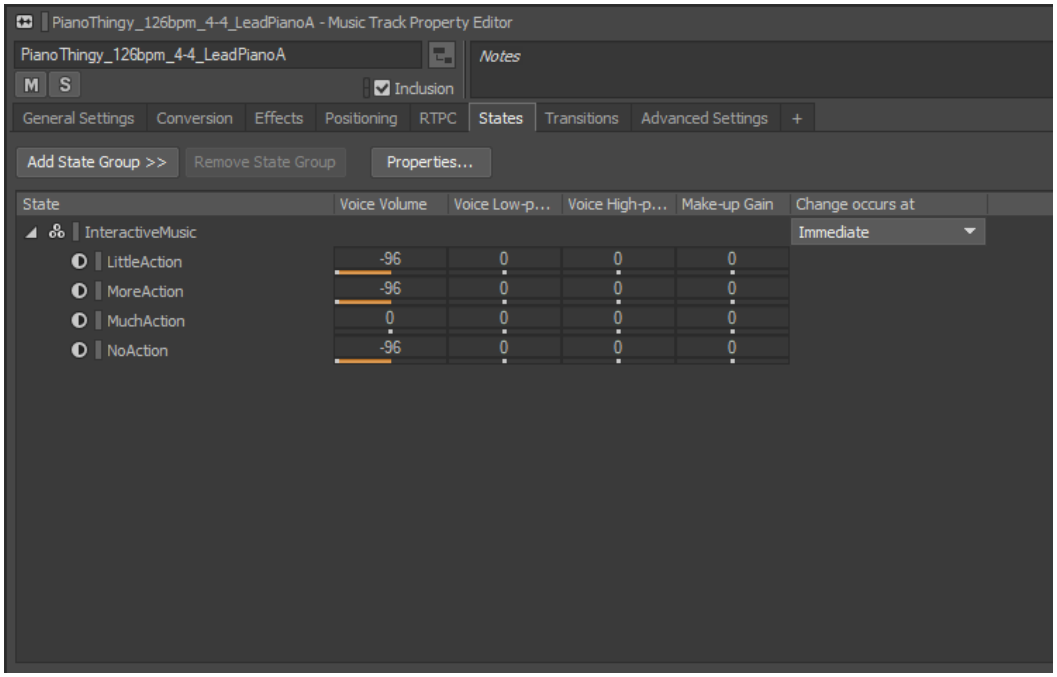
3.2 Controlling Music Tracks

A way to control which Music Tracks are audible while the music is playing is to use states to control the volume level of the tracks. Creating a *State Group* is done from the *Game Syncs* tab in the Project Explorer window. States are added into the State Group as child objects. The *State Group Property Editor* window will open if a state group is selected. Here it is possible to change *transition times* for switching to other states either on a global level or between two specific ones (picture 15). If long transition times are used as a default, a custom transition can be created for switching between states *None* and *NoAction*. The state in Unity is always None until it is set to another one. Having the switch between None and NoAction be zero seconds and setting the state to NoAction upon starting *Play Mode* in Unity, will ensure that the NoAction state will immediately be active when the game starts.



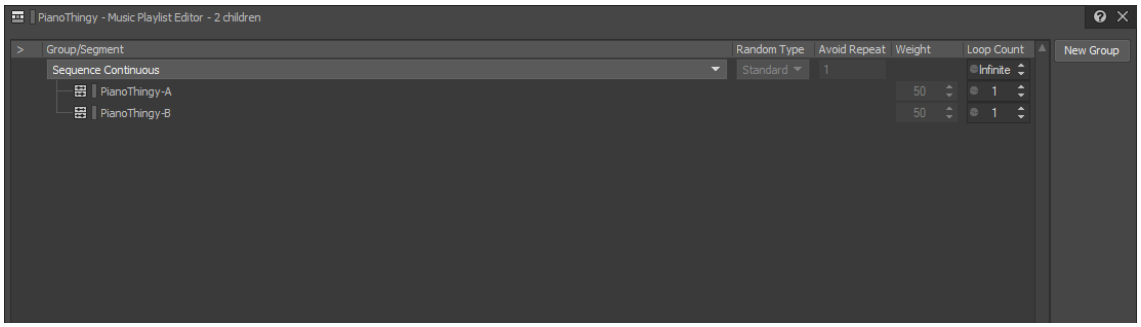
PICTURE 15. Showing the *InteractiveMusic* state group and its transitions

The different states need to be added to the Music Tracks of the PianoThingy-A and PianoThingy-B Music Segments. Selecting one of the Music Tracks opens a *Music Track Property Editor*. On the *States* tab, the new state group can be added which adds a list of the possible states to the tab. Next to the states are options for how the properties of the track change when a certain state is active. The *Piano* Music Track has no states as it is always audible, but the other tracks can have *Voice Volume* set to the value -96 for the states that should not feature that specific instrument. For example, if the *LeadPiano* track comes in last, all other states except the last one where all the instruments can be heard should be set to -96 (picture 16). In this case, the last state would be the *MuchAction* state.



PICTURE 16. The state setup for the LeadPiano Music Track

Once all the states have been set for the Music Tracks in both Music Segments, the segments can be dragged into the empty space under *Sequence Continuous* in the *Music Playlist Editor* window (picture 17). This makes it possible for the music to play from the Music Playlist Container object when it is put into an event. The loop count for the *sequence* should also be set to *Infinite*, so that the music will loop and play indefinitely unless otherwise told. Playlists can be used to create structures for songs, and then break them by introducing e.g. random loop counts for individual segments or segments that are sometimes skipped. Now the PianoThingy Music Playlist Container is ready and can be put into a Play action event and a sound bank.



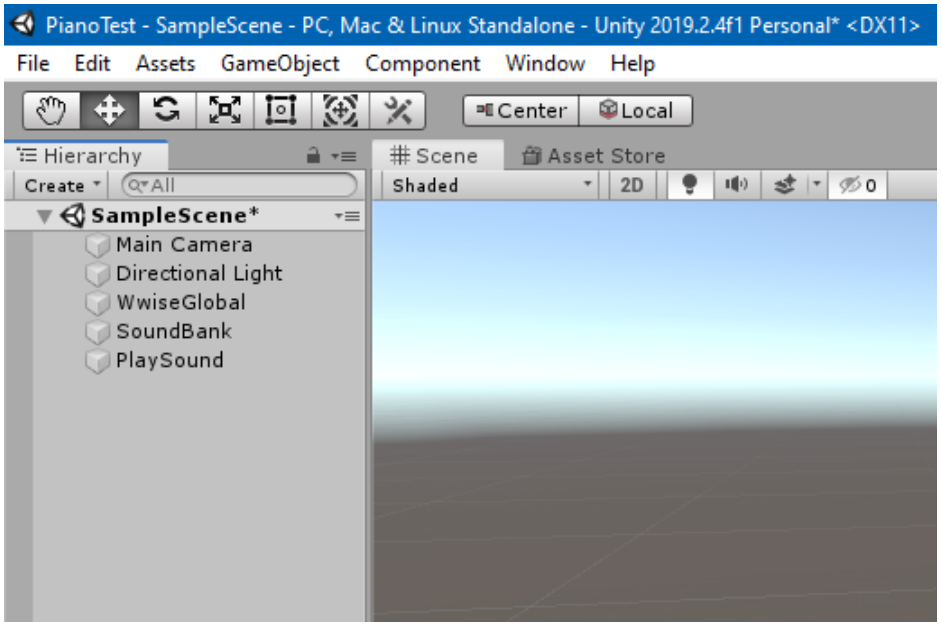
PICTURE 17. Setting the playlist for a Music Playlist Container object

As a summary, the ways of making interactive music presented in this thesis are controlling music tracks individually through states, which requires the use of a vertical project structure, as well as building varied playlists out of Music Segments, which needs a horizontal project structure. A combination of these options is what Red Stage Entertainment are looking for in their project. Being able to affect both the segments of a song as well as the instrument tracks the segments are built out of provides plenty of variety for the music.

3.3 Making it work in Unity

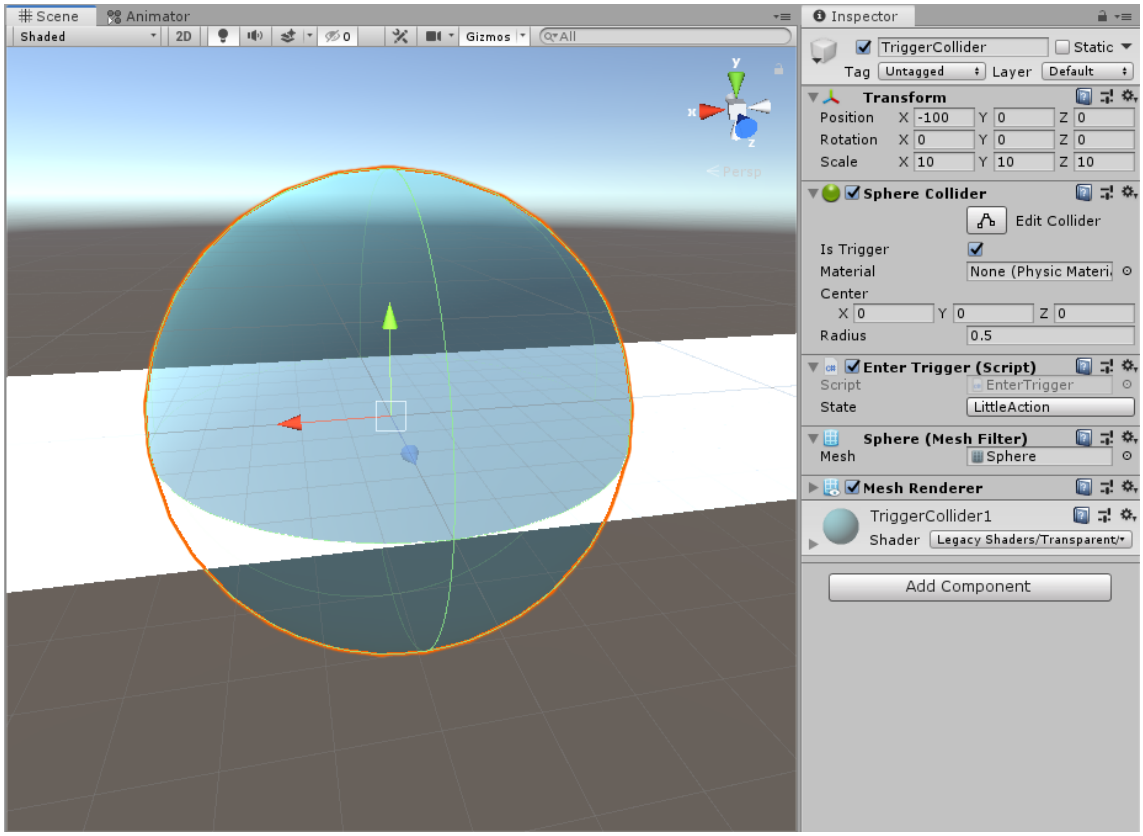
The adding of instruments with state changes can be triggered by a player. For this feature, a player character walks across a long pathway, triggering the addition of instruments along the way by hitting colliders. In order to achieve this, a player character and *trigger colliders* need to be created in the Unity project.

After the Music Playlist Container has been put into an event and a sound bank, and the sound bank has been generated, the state functionalities for the music should work on the side of Unity. Assuming the Wwise integration has automatically replaced the basic Unity Audio Listener for the Main Camera object with AkAudioListener and added the WwiseGlobal game object inside of the default scene, the next phase is to create new game objects and add AkBank and AkEvent components to them (picture 18). After that, the correct sound bank and event references need to be set for components. The event reference should be the music created for the feature.



PICTURE 18. Unity scene hierarchy after Wwise integration and the creation of *SoundBank* and *PlaySound* game objects

A player character is created in the form of a *capsule* object. The ground that the character stands under can be a *plane* object. A script that allows the player to move either forward or back is attached to the character as a component. The plane is made into a long pathway. The player should move forward on the pathway and hit colliders that trigger the state changes. The colliders can be *sphere* objects that contain a *Sphere Collider* component (picture 19). There is a checkbox for a Boolean *IsTrigger* variable in this component that should be checked. This sets the variable to be *true*, meaning the collider of the sphere object now works as a trigger for when other objects *enter* it, *exit* it or *stay* inside it.



PICTURE 19. A sphere object with a Sphere Collider that is set to IsTrigger containing a custom *EnterTrigger* script component

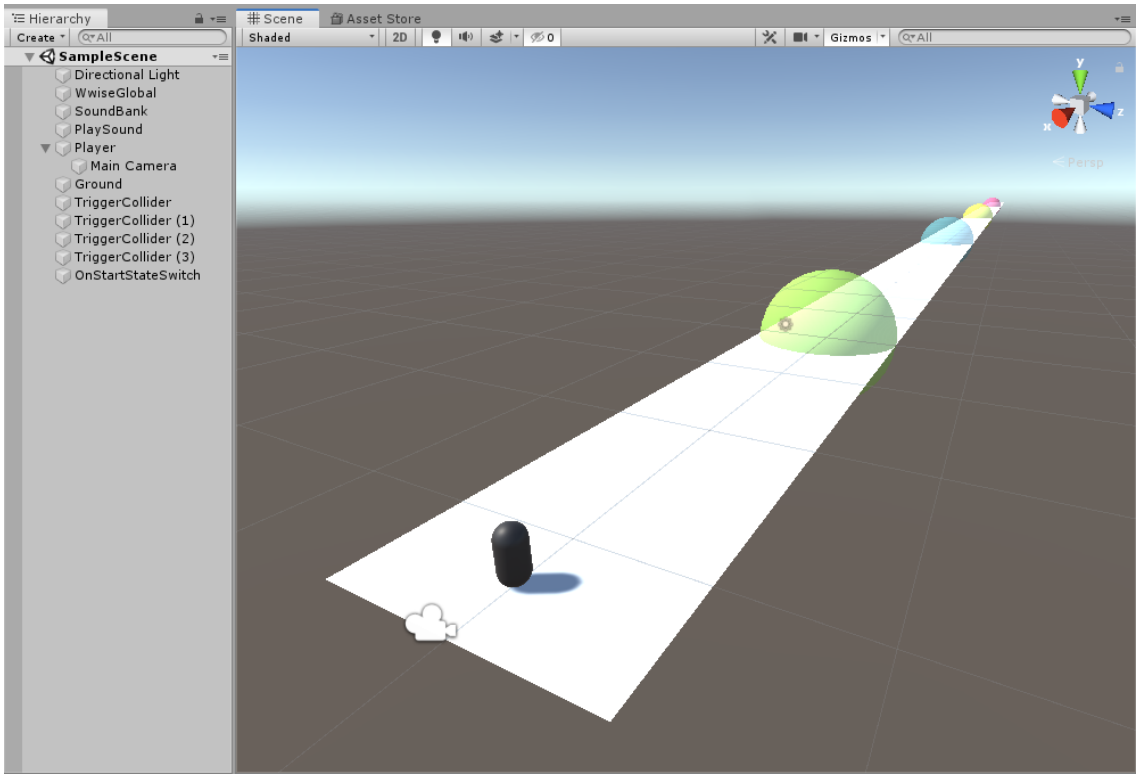
A new script called EnterTrigger is created and added on to the sphere object as a component. This is where a *Wwise state variable* is declared and made visible on the component in the Unity Editor. A function called *OnTriggerEnter* is created where the state is set to be the same as the state selected on the component via a *SetValue* function. Picture 20 shows the lines of code in the EnterTrigger script.

```
0 references
public class EnterTrigger : MonoBehaviour
{
    [SerializeField] private AK.Wwise.State State;

    0 references
    private void OnTriggerEnter(Collider other)
    {
        State.SetValue();
    }
}
```

PICTURE 20. The code in the EnterTrigger script

Now the sphere objects can be multiplied and placed along the plane object (picture 21). When the state references have been set correctly on the components, the feature is almost ready for use. A script much like EnterTrigger called *OnStart* can be created to change the state or switch to NoAction in a Start function. By starting Play Mode from the Unity Editor, the player character can be moved forward and back triggering different states. The music should also change from one Music Segment to another and then loop again.



PICTURE 21. The Scene and Hierarchy views of the interactive music feature

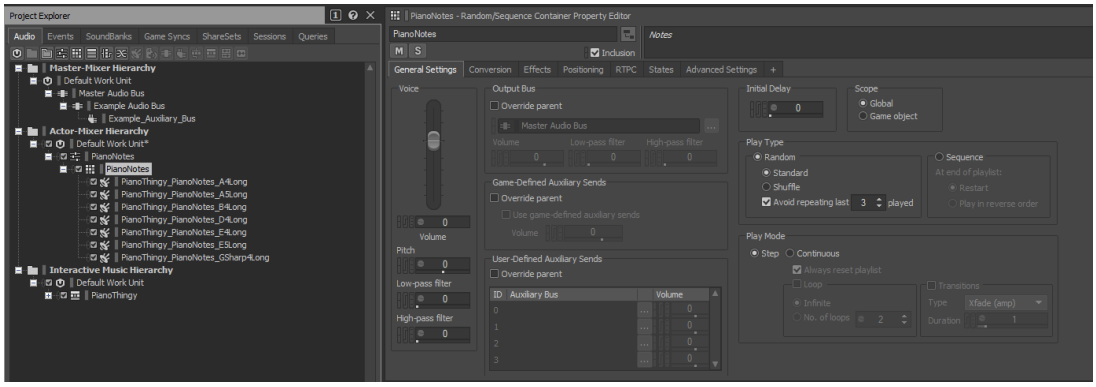
Experiencing the result of the feature gives ideas and inspiration for furthering these implementations. The actions of the player affecting the music is an interesting way to help with e.g. feedback and atmosphere. More ambitious and efficient versions of these interactive music methods can be applied into the game Red Stage Entertainment is developing.

4 INPUT QUANTIZATION

4.1 Getting started

What is desired from the player input quantization feature is the ability to have player inputs happen on the beats of a music track. This means that inputs that are given too early or too late will either be delayed until the next beat or made to happen instantly after a beat, respectively. The reason for the latter is to give the player a bit more time to react to the upcoming beat. That is why a short window for getting an input through is presented right at the start of a beat. The way to keep track of beats on the side of Unity is done by using *Wwise callbacks*. Callbacks can be used to e.g. gain information about the tempo and time signature of a song as well as when a *measure* or beat occurs in a song. One of the suggestions for the feature from the client is to have the quantization happen to *sub-beats* as well. Sub-beats are beats that are split into shorter segments, for example *1/8th* of a beat.

Firstly, individual notes on the piano are exported out of Logic Pro X as the sounds that the player will input. The notes are brought into Wwise and put into one *Random Container* object as shown in picture 22. Random Containers play the audio files put into them one at a time, picking which file is played at random. This object is put into an event and a sound bank, and the sound bank is generated.



PICTURE 22. The *PianoNotes* Random Container object

4.2 Creating the script

For the previous feature, the music event was posted with an AkEvent component. Since it is easier to control callbacks via code, the posting of the music event should be moved to a new *InputQuantization* script. An event variable can be declared in a script with the *AK.Wwise.Event* class. An event for the piano notes that are the input of the player should be declared as well. A function for the callbacks requires parameters, of which the *AkCallbackType* variable is the most important one for telling Unity which callbacks should be considered. In this case, finding out when a beat occurs, and the music starts are the desired callbacks (picture 23).

```
void CallbackFunction(object in_cookie, AkCallbackType in_type, object in_info)
{
    if (in_type == AkCallbackType.AK_MusicPlayStarted)
    {
        StartTimer = true;
    }

    if (in_type == AkCallbackType.AK_MusicSyncBeat)
    {
        ResetTimer();
    }
}
```

PICTURE 23. Using Wwise callbacks to give Unity information about when the music starts and when each beat occurs

Initially, as a test, player input is only allowed on the beat. The accuracy is not ideal at all as barely any of the inputs register because of *input lag* and lag between the two programs, Wwise and Unity. A timer that counts down the length of a beat in seconds is created via code. The duration of a beat is then divided by the possible sub-beat amount (picture 24). The timer is reset and brought back up via the beat callback. The first attempt at input quantization is declaring a *Boolean* variable that when set to true, posts the PianoNotes event at the next possible beat callback. The value is set to true when a player input happens.

Boolean variables offer two options, *true* and *false* values. They are most often used to control when statements, e.g. *if statements*, should be executed. (Processing N.d.) If statements are only executed if the value of the parameter attached to them is true. For the feature, there are now a couple of restrictions for when the player can input an action in terms of timing. Input lag and other

forms of lag are only addressed by having a *LatencyBalance float* value that is manually set.

```
public void CountSecondsInBeat()
{
    SecondsInBeat = (60f / BPM) / BeatDivision;
}
```

PICTURE 24. Counting the length of a beat in seconds divided by the intended sub-beat amount

This way of doing it produces some results, but it is not consistent at all. Sometimes sound doesn't play even when it is completely on the beat with the actual music because of input and Wwise to Unity lag determining that the beginning or end of the beat is somewhere else than it should be. Using a simple float value to try to fix this is not going to work.

At this point of creating the feature, one of the programmers working at Red Stage Entertainment cleans up the script by getting rid of unneeded variables and lines of code. They make the code work in a more consistent manner by using more efficient scripting and doing better calculations on accepting player inputs at an earlier or later time than right on the beat (picture 25). In order to make everything work properly, they also get rid of any code related to dividing the beats into sub-beats, which requires that to be investigated again. The script is integrated into the game project that Red Stage Entertainment is working on.

```
private void TryToConsumeInputNow()
{
    if ( HasInputQueued && TimeSinceLastInputConsumed > HalfTargetInterval )
    {
        // If <very close to last beat> OR <next beat will happen closer to this frame than the next>
        if ( IsVeryCloseToLastTarget || IsCloserToNextTargetThanNextFrame )
        {
            OnBeatMatch.SendInput();
            HasInputQueued = false;
            LastInputConsumedTime = Time.unscaledTime;
        }
    }
}
```

PICTURE 25. The function that sends the input on the beat or very close to it

After a few days, the programmer makes the InputQuantization script work with the music system that has been created for the project of the client. This means that information regarding the tempo and duration of a beat is retrieved from Wwise via callbacks. The retrieval is done in a separate script and the InputQuantization script references that script for all the relevant information about a song. The most important thing to add to the feature again is allowing more inputs during a beat which means dividing the beat into smaller segments through code. Picture 26 displays how this is achieved by simply dividing the correct beat duration time received from Wwise by the desired *beat division*. The script is functional, and the beat can be divided into anything between one and 16 sub-beats.

```
/// <summary>
/// Beat interval measured in seconds, calculated from BPM.
/// If SubBeatDivision > 1, then interval will be equal to a single sub-beat segment duration.
/// </summary>
private float TargetInterval
{
    get
    {
        return MusicHandler.BeatDuration / (float)BeatDivision;
    }
}
```

PICTURE 26. A more refined version of the code in picture 24 done by a programmer

4.3 Summary

Creating a feature such as this requires adequate programming knowledge and familiarity with the Wwise to Unity integration. To get the feature to the point where it needed to be for the game project of Red Stage Entertainment and the thesis, help from a programmer was necessary. The feature is specific and might not have many uses outside of quantizing input of some kind. It is a feature that Red Stage Entertainment feels would fit for their game.

Tackling something more challenging helped with learning basic and more advanced Wwise theory and techniques. Wwise callbacks were a new thing to learn for the creation of the feature. Calculating the quantization of input was new and interesting as well.

5 CONCLUSION

Wwise is a powerful tool that allows audio designers to achieve varied and dynamic audio for games. It works on multiple platforms, integrates into the most common game engines and offers a free trial version which includes most capabilities of the sound engine. After about a year of getting to know Wwise, there is still a lot to learn about not only the features in the program itself but also in the game engine integration.

The objectives of the thesis were all met to a degree, some better than others. The introduction featured some untrustworthy sources but did its job in introducing the reader to video game audio design and the features that were developed for the thesis. Thorough source searching was not done due to time constraints. The section about Wwise basics introduced most of the basic elements of Wwise.

The interactive music feature section of the thesis managed to successfully convey a few ways in which interactive music can be created. While the actual player input quantization feature created for Red Stage Entertainment turned out to work well after further development by a programmer, the documentation of it in the thesis was not very detailed in the end. Both features were received well by the client and have further usage and iteration planned.

In terms of further research into Wwise and the features that can be created with it, the topic of this thesis is mainly tied to the client, Red Stage Entertainment. Wwise will most likely come up in future theses but what is done with the program and what the reasons behind doing it are, will be different. So, the topic of this thesis specifically warrants no further research but e.g. Wwise as a sound engine and more detailed interactive music testing are interesting topics.

REFERENCES

ACMI. N.d. The evolution of audio in videogames. Web article. Article posted on the Australian Centre for the Moving Image (ACMI) web page. Read 2.10.2019. https://www.acmi.net.au/ideas/read/evolution-audio-videogames/

Audiokinetic About. N.d. Audiokinetic About page. Web page. Read 1.10.2019. https://www.audiokinetic.com/about/

Audiokinetic Applying Distance-Based Attenuation. N.d. Applying Distance-Based Attenuation. Web documentation. Wwise documentation about basic attenuation for audio. Read 1.10.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=applying_distance_based_attenuation

Audiokinetic Creating Wwise Objects on Import. N.d. Creating Wwise Objects on Import. Web documentation. Wwise documentation about Wwise objects. Read 16.11.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=creating_wwise_objects_on_import

Audiokinetic Event Actions List. N.d. Event Actions List. Web documentation. Wwise documentation showcasing all the different event actions and properties. Read 14.11.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=event_actions_list

Audiokinetic Glossary. N.d. Audiokinetic Glossary. Web page. A web page explaining terms related to Wwise and audio. Read 30.8.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=glossary

Audiokinetic Importing Media Files. N.d. Importing Media Files. Web documentation. Wwise documentation about media file importing. Read 16.11.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=importing_media_files

Audiokinetic Interactive Music Project Structure. N.d. Interactive Music Project Structure. Web documentation. Wwise documentation about interactive music structures. Read 21.11.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=interactive_music_project_structure

Audiokinetic Structuring a Bus Hierarchy - Example. N.d. Structuring a Bus Hierarchy - Example. Web documentation. Wwise documentation about the structuring of an audio bus hierarchy. Read 15.11.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=structuring_bus_hierarchy_example

Audiokinetic The Master Audio Bus Hierarchy. N.d. The Master Audio Bus Hierarchy. Web documentation. Wwise documentation about the Master Audio Bus hierarchy. Read 15.11.2019.

https://www.audiokinetic.com/library/edge/?source=Help&id=the_master_audio_bus_hierarchy

Audiokinetic Understanding Events. N.d. Understanding Events. Web documentation. Wwise documentation about the basics of Wwise events. Read 14.11.2019. https://www.audiokinetic.com/library/edge/?source=WwiseFundamentalApproach&id=understanding_events_understanding_events

Audiokinetic Understanding Interactive Music. N.d. Understanding Interactive Music. Web documentation. Wwise documentation about the basics of interactive music. Read 1.10.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=understanding_interactive_music_understanding_interactive_music

Audiokinetic Understanding Sends. N.d. Understanding Sends. Web documentation. Wwise documentation about auxiliary sends. Read 15.11.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=understanding_sends

Audiokinetic Understanding SoundBanks. N.d. Understanding SoundBanks. Web documentation. Wwise documentation about the basics of Wwise sound banks. Read 14.11.2019. https://www.audiokinetic.com/library/edge/?source=WwiseFundamentalApproach&id=understanding_soundbanks_understanding_soundbanks

Audiokinetic Using Real-time Effects. N.d. Using Real-time Effects. Web documentation. Wwise documentation about the basics of real-time effects on sounds. Read 4.11.2019. https://www.audiokinetic.com/library/2017.2.10_6745/?source=WwiseProjectAdventure&id=using_real_time_effects

Audiokinetic Using the Wwise Unity Integration. N.d. Using the Wwise Unity Integration. Web documentation. Wwise documentation about the Wwise to Unity integration. Read 1.10.2019. https://www.audiokinetic.com/library/edge/?source=Unity&id=pg_appscripts.html

Audiokinetic What are Work Units? N.d. What are Work Units? Web documentation. Wwise documentation about Wwise Work Units. Read 16.11.2019. https://www.audiokinetic.com/library/edge/?source=Help&id=what_are_work_units

Baker, M. 2016. How Do Game Engines Work? Web article. Published 2.11.2016. An article on the Interesting Engineering website. Read 2.12.2019. https://interestingengineering.com/how-game-engines-work

Berklee College of Music. N.d. Sound Designer (Games and Tech) Career Explanation. Web page. Read 27.10.2019. https://www.berklee.edu/careers/roles/sound-designer-games

Campbell, B. 2016. History of Video Game Music. Web page. A part of a lesson for a course. Read 27.10.2019. http://bdcampbell.net/music/

Case, A. 2014. Digital Audio Workstation. Web page. Published 31.1.2014. From the Grove Music Online website. Read 23.11.2019. https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-1002256346

Cullen, M. N.d. Basics of Sound Design for Video Games. PDF document. Slides made for a presentation during a lesson by an experienced sound designer. Read 1.10.2019. https://frost.ics.uci.edu/ics62/BasicsofSoundDesignforVideoGames-MichaelCullen.pdf

Esposito, N. 2005. A Short and Simple Definition of What a Videogame Is. Conference paper. PDF available. Read 2.10.2019. https://www.researchgate.net/publication/221217421_A_Short_and_Simple_Definition_of_What_a_Videogame_Is

FMOD. N.d. FMOD Studio: The adaptive audio solution for games. Web page. Introduction to FMOD Studio on the official website of FMOD. Read 1.10.2019. https://www.fmod.com/studio

Gamedesigning.org. 2018. Video Game Sound Design 101. Web blog article. Published 31.12.2018. Read 1.10.2019. https://www.gamedesigning.org/learn/video-game-sound/

Hogan, B. 2014. How the Sound and Music Came Together. Web blog post. Published 15.12.2014. Blog post detailing sound design for the game Never Alone. Read 27.10.2019. http://neveralonegame.com/sound-music-came-together/

Metcalfe, N. 2013. Sound Design For Visual Media & Radio. Web article. From Sound On Sound online article. Read 1.10.2019. https://www.soundonsound.com/techniques/sound-design-visual-media-radio

Processing. N.d. True/False. Web page. A web page on Processing.org explaining Boolean variables. Read 30.11.2019. https://processing.org/examples/truefalse.html

Red Hat. N.d. What is middleware? Web page. Read 2.12.2019. https://www.redhat.com/en/topics/middleware/what-is-middleware

Sandoval, K. 2018. What is the Difference Between an API and an SDK? Web blog post. Published 16.11.2018. Posted on the Nordic APIs website. Read 14.11.2019. https://nordicapis.com/what-is-the-difference-between-an-api-and-an-sdk/

Shafer, R. 2019. What Is Input Lag And How Important Is It For Gaming? Web article. Published 10.6.2019. Read 2.12.2019. https://www.displayninja.com/what-is-input-lag/

Stinson, J. N.d. Real-time Sound Effects: The Foley Way. Web article. Article posted on the Videomaker website. Read 27.10.2019. https://www.videomaker.com/article/7220-real-time-sound-effects-the-foley-way

Sweet, M. 2016. Top 6 Adaptive Music Techniques in Games – Pros and Cons. Web article. Published 13.6.2016. An article on the Designing Music Now website. Read 1.10.2019. https://www.designingmusicnow.com/2016/06/13/advantages-disadvantages-common-interactive-music-techniques-used-video-games/

Unity Documentation Scenes. N.d. Unity Documentation Scenes. Web documentation. Unity documentation about Scenes. Read 21.11.2019. https://docs.unity3d.com/Manual/CreatingScenes.html

WIRED. 2017. Classic Video Game Sounds Explained By Experts (1972-1998) | Part 1 | WIRED. Youtube video. Published 12.6.2017. Cited parts 3:57-4:16, 4:29-4:46 and 8:26-8:36 of the video. Viewed 1.10.2019. https://www.youtube.com/watch?v=jlLPbLdHAJ0

Wither, C. N.d. Difference Between Consonance and Dissonance. Web article. An article on the Difference Between website. Read 14.11.2019. http://www.differencebetween.net/science/difference-between-consonance-and-dissonance/

Wwise SDK LoadBank. N.d. Wwise SDK LoadBank. Web documentation. Wwise documentation about the function LoadBank of the AkSoundEngine class. Read 21.11.2019. https://www.audiokinetic.com/library/edge/?source=SDK&id=namespace_a_k_1_1_sound_engine_ada82aef371a6375339196976e80a38f8.html

APPENDICES

Appendix 1. Link to Youtube video: the interactive music feature

https://www.youtube.com/watch?v=X33w1eSWs6s