Design and development of a videogame with dynamic sound based systems with Unity using Wwise.

Antoni Bellver Tirado

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

September 20, 2020

Supervised by: Vicente Cholvi Juan

To Andrea Giménez Vila

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Vicente Cholvi, for the complete freedom of subject and method regarding the project theme and development.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

This document presents the use of audio middleware software, in conjuction with Unity game engine, in the design and further implementation of different dynamic audio systems applied in a PC videogame. This sound systems include vertical and horizontal music and ambience with custom made audio assets, and how it implements in a short, highly replayable game. A provisional video showcasing the end result can be seen here: BLINDFOLDED

Contents

Chapter 1

Introduction

Contents

1.1 Work Motivation

There are dozens of different audio middleware softwares available for game developing. These serve as an intermediate layer between the game engine and the audio engineer to design and implement all audio scripts and behaviours present in the videogame. They give the framework needed to create specific, professionally finished audio functionalities to a certain project, expanding the tools most commercial game engines offer to the public. Most of these pieces of software require a per-project payment, with fees quickly escalating far off the price a single user would pay. Their business plan is designed for studios with an established budget. Nonetheless, some of them present a free license for small developers or standalone users (FMOD, Wwise), with more than enough features available for this project.

In the videogame industry, specialization is key. Learning how to use audio engines and how they interact and work with actual game engines may be very enriching for a developer.

1.2 Objectives

As both the role of main developer and audio engineer, the main idea behind this project was to develop a finished product: end up with a completely functional demo, re-playable and engaging, where all its core mechanics ans systems have been fully implemented.

Learn how to use audio middleware is the big focus on this project. Learning how to use specialized software like audio engine may potentially open a lot of professional options and opportunities.

Also, expand Unity engine knowledge. Creating the video game, being responsible of all of its different sections, will help increase proficiency in that specific engine.

1.3 Environment and Initial State

The project started with and intermediate level in Unity and no experience in any audio middleware whatsoever. To make it possible, everything will be developed from a single desktop PC exclusively, making use of both free licenses for Unity and Wwise, combined with a handful of free tutorials and information about audio engines. The learning process of audio middleware was boosted by already existing knowledge of digital audio and advanced level on some similar audio treatment systems.

Chapter 2

Planning and resources evaluation

Contents

2.1 Original planning

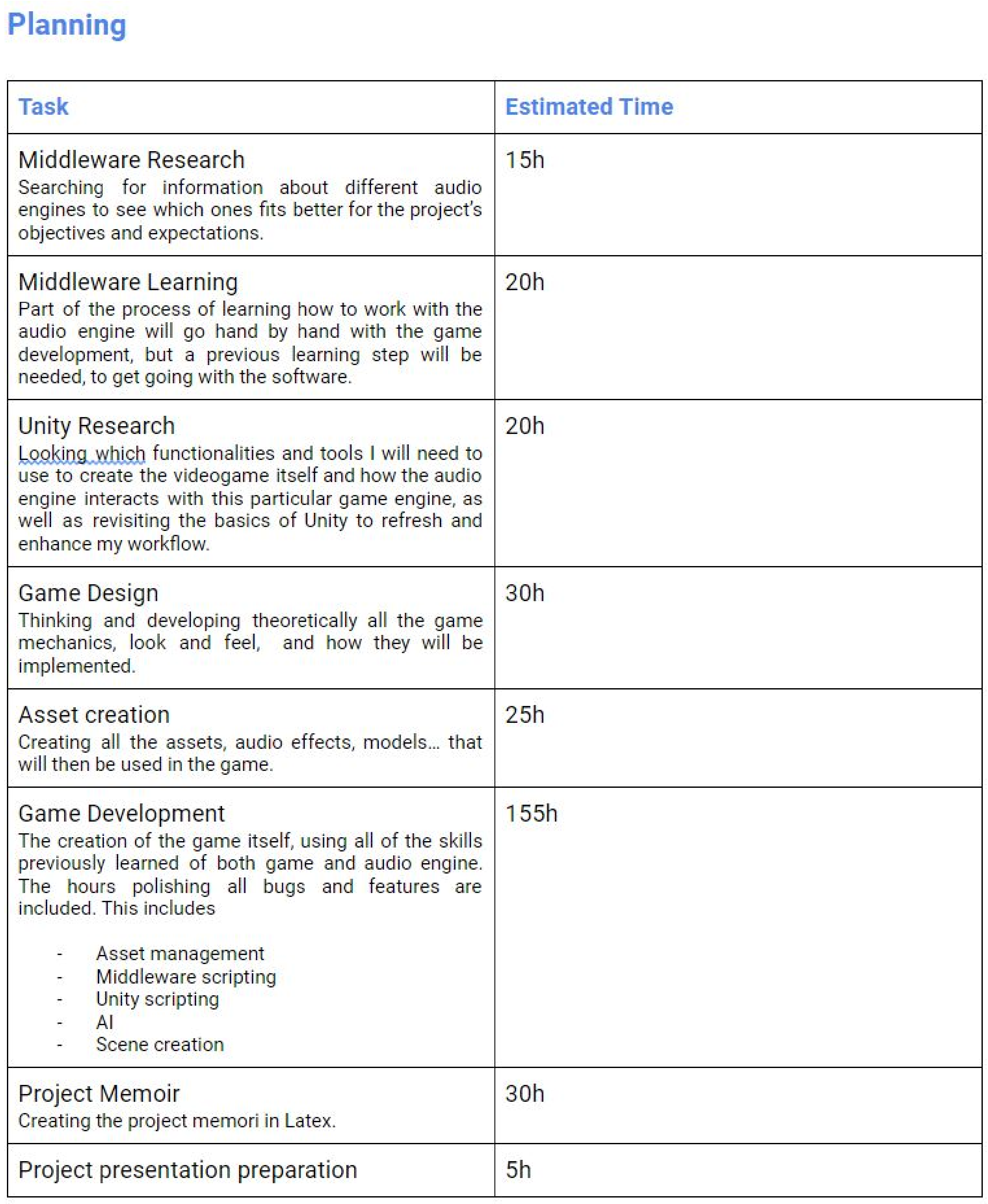
The first approach to planning the development of this project was very different from the followed planning. (see Figure 2.1)

2.2 Final planning and Resource Evaluation

As seen in the Gantt diagram (see Figure 2.2), a lot more time is put into asset creation and processing, rather than so much hours spent on pure theoretical software learning. A big part of the learning process will come with the development itself, while scripting and setting up the various systems.

To develop the project, there is both a desktop PC with more than enough computing power and a laptop at disposal, as well as studio monitors and a properly acoustic treated room.

As for the audio resources and other assets used in the game, most of them will be self made. The quality on some of them may vary or be inconsistent when comparing them to professional AAA recorded and processed content, but, as mentioned, the main



Planning

Task	Estimated Time
Middleware Research Searching for information about different audio engines to see which ones fits better for the project's objectives and expectations.	15h
Middleware Learning Part of the process of learning how to work with the audio engine will go hand by hand with the game development, but a previous learning step will be needed, to get going with the software.	20h
Unity Research Looking which functionalities and tools I will need to use to create the videogame itself and how the audio engine interacts with this particular game engine, as well as revisiting the basics of Unity to refresh and enhance my workflow.	20h
Game Design Thinking and developing theoretically all the game mechanics, look and feel, and how they will be implemented.	30h
Asset creation Creating all the assets, audio effects, models... that will then be used in the game.	25h
Game Development The creation of the game itself, using all of the skills previously learned of both game and audio engine. The hours polishing all bugs and features are included. This includes - Asset management - Middleware scripting - Unity scripting - AI - Scene creation	155h
Project Memoir Creating the project memori in Latex.	30h
Project presentation preparation	5h

Figure 2.1: Original planning created during the first conceptual steps.)

target of this project is not the production of assets. Every other asset used will be copyright free.

2.3 Why Wwise

The reason Wwise is chosen as the developing audio engine in front of other alternatives like FMOD is straightforward. In the first place, the quantity of material about Wwise (tutorials, plugins, sample projects) that can be found for free with a basic search is more then enoguh not only to learn the basics, but to build a consistent knowledge of the software. Audiokinetic, developers of Wwise have emphasized this by creating official tutorials and videos, explaining all functionalities, available fro free in their official site. This makes the learning process a lot more structured and, overall, save time that otherwise would be lost while learning "on the fly". The other reason Wwise ended up being a better candidate is, in fact, the different professional projects where it has been used as audio engine, as the audio system in most of them served as direct inspiration to conceptualize and develop the different mechanics conforming Blindfolded.

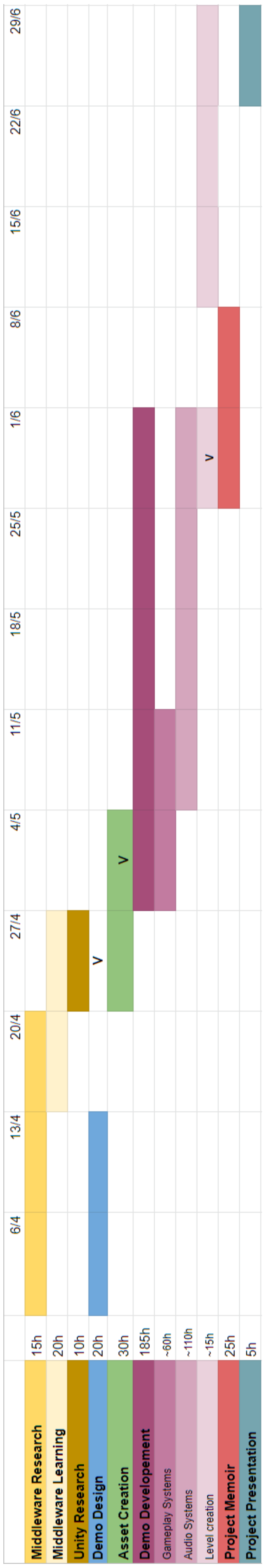


Figure 2.2: Gantt chart of this project development).

Chapter 3

System Analysis and Design

Contents

3.1 Blindfolded: Demo Concept

The playable demo to test all implemented audio systems is called Blindfolded.

Blindfolded is an oppressive adventure game with a gameplay timespan no longer than 5 minutes. In it, the player will control a blindfolded, handcuffed individual trying to escape it's captors as he runs through a small terrain parcel. With only a few visual hints in the middle of a pitch black screen, sound will be the only possible guidance to safety, as footsteps, noises, heavy breathing and music dynamically build up tension to unbearable amounts. It will make use of a top-down perspective, and it's controls will be minimal, only using computer's mouse and a reduced set of keys.

Figure 3.1:

3.2 Blindfolded: Design Goals

Blindfolded aims to create a short but rather intense gameplay experience. With its simple controls, minimal (more like non-existent) UI and small map size, the experience focuses on the overwhelming sensation of being restrained and deprived of vision, the most valuable of all five senses, in a hostile environment where auditory stimuli will become as stressful as necessary to escape successfully. The use of headphones becomes rather mandatory.

3.3 Blindfolded: Core Components

Any of them have a concrete physical appearance due to the fact that everything is obscured. The only visual clue for representing them will be the footprint they leave as they travel through the map, or a flare effect piercing through your blindfold as they approach in the case of cars.

3.3.1 Player character

A regular, anonymous person trying to escape from its captors. It could be you.You'll continuously hear your own footsteps. As you play and run away from the NPC's, you will be able to hear and perceive how your breathing gets more intense. The only visual clue you get of yourself is an stylized version of your footprints, in a white color.

3.3.2 Persecutor

These NPC's will be searching for the player. The player will be able to hear them walking if the gets close enough and, if he's been moving silently enough, he'll be able to walk around them with no trouble. If he gets spotted, they'll run towards him. The player is always faster than the persecutors, so running away from them may save you, but take into account that your breathing and stepping sound will be louder, alerting more of them in the process. It they get to you, is game over. The only visual clue representing persecutors are their footsteps, in a red color.

3.3.3 Cars

A both-ways road traverses the scenario. It depends on the procedural randomness of each level, but most of the time the player will have to cross it to scape. Cars continuously pass at outrageous speed. The sound of the engine plus the wind being pulled by the car's force will be the best clue for the player to try to dodge them as he crosses the road. Their visual representation is a flare effect from their front lenses, piercing through your blindfold as they approach. They won't see the player coming, so if he stays too much time in the middle of a road he might get hit, with an instant game over as a consequence.

3.3.4 Level

Trees and bushes will fill the area. They are harmless and just obstruct the player's movement as he blindly explores the scenario. They can be quite annoying if the player is just trying to run through the level, as he can just crash into them, getting knocked down for a couple of seconds, or getting stuck into them (in the case of bushes).

There are two-ways roads are filled with cars, that will continuously run through them. They won't see the player coming, so if he stays too much time in the middle of a road he might get hit, with an instant game over as a consequence. They are plotted in any given direction, dividing the level.

Scaping is not easy, as big part of the level is surrounded by an electrified fence. If the player walks near it, he can hear the static noise and follow it, looking for a gate or overture to overcome it. Crashing into it will stun the player for a good amount of time, and it will generate a good amount of noise that might alert persecutors that walk nearby. All these items, scattered trough the demo, create certain particle systems when the player crashes on them.

3.4 Blindfolded: Core Mechanics

3.4.1 Movement

Your footprints and footsteps will occur continuously in the direction you are moving.You can choose to walk, resulting in advancing at a slower pace, but the quieter your breathing and overall stealth will be. You can also run, being able to escape most persecutors, but increasing the danger of crashing into fences or trees.

3.4.2 Sound

High pitch strings howl as persecutors run at the player. The electrified sound of fences. Car engines rushing through the road at 120Km/h in front of you. The sound itself is the core mechanic in Blindfolded. With no vision, the player has to rely on everything he hears to locate himself in the space he is moving. A rich and deep soundscape brings the immersion and main feature of Blindfolded.

3.5 Requirement Analysis

To carry out a job, requirements where divided into three different categories: one regarding everything that has to do with Unity (game engine), another one for all the tasks needed to implement in Wwise(sound engine), and one last category for everything regarding to asset generation.

3.5.1 Functional Requirements

There is a handful amount of requirements that needed a proper implementation, both for Unity and Wwise, in order to make the demo possible. These include from scripts for player and npc's to dynamic music systems and spatial audio systems for most of the entities present. A complete list can be found in this section.

Input:	(UNITY) Player Movement Script
Output:	Player can move ingame
The player entity in the engine may need an script which allows it to move using W,A,S,D keys and rotate camera around itself, as well as choose whether to walk or run pressing the "L SHIFT" key.	

Table 3.1: Functional requirement «CRED1. Player Movement»

Input:	(UNITY) Collision Manager Script
Output:	Game knows what has the player collisions with.
The game has to properly process the different type of collisions that can happen. If the player collisions with a persecutor or car, the game ends. If the player collisions against a tree or fence, instead, it will get the proper stun penalty depending on it's current speed in the moment of the collision. Running blindly in the woods by night is not a good idea.	

Table 3.2: Functional requirement «CRED2.Collision manager»

3.5.2 Non-functional Requirements

As non-functional requirements, there is a game design document explaining most of the design decisions and concepts, that can be found with the demo. It also includes a PDF

Input:	(UNITY) Game Over Script
Output:	Game knows when it's game over, and proceeds to end the demo.
When player collisions with a persecutor or gets stomped by a car, a message that "YOU DIED" will fill the screen. When player successfully escapes the level, a "YOU ESCAPED" message fills the screen.	

Table 3.3: Functional requirement «CRED3. Game Over Manager»

Input:	(UNITY) Visual Clues
Output:	Most of the things that can happen have some kind of minimal screen representation.
From footsteps to car flashes, most of the things that can be heard can be visually perceived as abstract clues on the screen, spatially positioned near to where the sound happened.	

Table 3.4: Functional requirement «CRED3. Visual Clues»

Input:	(UNITY) Persecutors AI
Output:	State machine and behaviours for enemy NPC's.
Persecutor NPC's are able to follow pseudo-randomized patrol routes, leave them when they hear noise to chase the player, and return to them if they loose track of their target. Cars will follow roads, entering and exiting the level continuously.	

Table 3.5: Functional requirement «CRED4. AI»

Input:	(UNITY) Level Design
Output:	Combining all the assets and creating an interesting demo.
The objective is to create a interesting, highly replayable showcase demo using all of the implemented systems. Thanks to the randomized enemy patrols and gameplay mechanics (complete darkness) the player can attempt multiple runs with very different results.	

Table 3.6: Functional requirement «CRED5. Level Design»

Input:	(UNITY) Graphic and balance adjustments
Output:	Polishing the game experience.
Applying culling, as well as some graphic effects like grain or distortion to create a more gritty look and a more intense experience. This minor changes won't affect gameplay but will help to create a better cohesion with the sound aspect of the project, which is it's main focus.	

Table 3.7: Functional requirement «CRED6. Adjustments»

Input:	(WWISE) Character footsteps sound
Output:	Characters generate spatially correct placed footsteps sound.
It is important to create footsteps for both player and NPC's. These have to be different in feel and dynamic, to make clear which ones are being generated by the player and which are being generated by persecutors, as well as being placed in the game space correctly as they sound.	

Table 3.8: Functional requirement «CRED7. Footsteps audio»

Input:	(WWISE) Character breathing
Output:	Player character variable breathing intensity and feel depending the situation.
As player character gets more and more nervous (and tired), his or her breathing will increase in intensisty. This can happen because of some factors: being persecuted and enemy NPC's get near, stepping on some obstacle, sprinting for too long...	

Table 3.9: Functional requirement «CRED8. Breath audio»

Input:	(WWISE) Dynamic ambience
Output:	Game world has it's own dynamic, organic soundscape
A mix of forest sounds, static noise and synthesizers changing in pitch and dynamic create the game main soundscape. All this audio stems can vary in volume, stereo placement and signal processing depending the situation of the player.	

Table 3.10: Functional requirement «CRED9. Ambience audio.»

Input:	(WWISE) Tension and game over audio effects
Output:	Perfectly transitioned and balanced audio effects to add tension and end the game.
Every time an NPC hears the player and starts following it, an audio effect transition will occur on top on the dynamic ambience. High pitched strings and distorted sounds will push the player and urge him to run.	

Table 3.11: Functional requirement «CRED10. Effects»

Input:	(WWISE) Car sound effects
Output:	Spatial wind-breaking car effects from the vehicles travelling the road at high speed.
As player comes close to the road, a new audio system gets involved: car sounds. Being completely blind and trying to cross the street should look and feel like a completely scary situation, and this sound will help create that feeling.	

Table 3.12: Functional requirement «CRED11. Car audio»

Input:	(ASSETS) Character footsteps sound and look
Output:	Footsteps particle system
As the player moves, the most important visual clue it will get is where he or she is stepping on. Stylized footsteps will appear onscreen, synchronized with audio systems as footsteps sound.	

Table 3.13: Functional requirement «CRED12. Footstep Design»

Input:	(ASSETS) Create dynamic ambience soundscape.
Output:	Audio stems for ambience soundscape.
To create the loop stems that are used in game (and before putting them trough Wwise), a mix of sample based recordings of forest noises and some basic sound synthesis for digital strings will be.	

Table 3.14: Functional requirement «CRED13. Ambience Design»

Input:	(ASSETS) Create a believable breathing progression.
Output:	A set of audio stems that represent an increasing intensity breathing
For this purpose, and because all libraries with such materials are out of budget for this project, recording myself to create the stems is the best option. The hardware specifications for recording are specified in 1.3	

Table 3.15: Functional requirement «CRED15. Breathing Design»

Input:	(ASSETS) Create good coherent tension effects for audio detail.
Output:	A handful of audio effects used ingame for key moments.
Using some basic sound design and royalty free samples, create unique and gritty sound effects for all events happening ingame.	

Table 3.16: Functional requirement «CRED16. Effect Design»

that serves as a manual for the playable game, with its keyboard controls and a little survival guide.

3.6 System Design

(see Figure 3.2)

3.7 System Architecture

The demo does not require big computation regarding polygons or complicated calculations, so I find counter-intuitive to set system requirements to play the game. Any home machine may be able to run it with no problems.

3.8 Interface Design

As for its main menu, Blindfolded features a minimal main menu,where the game title floats in the middle of the screen, just accompanied by a big "ESCAPE" lettering. Pressing on this one and only option will start a run. Options to disable sound are not implemented, as this decision will severely impact gameplay experience (making the game impossible to play).

In game, there is no UI of any kind. All the information the player needs is transmitted via the different visual clues in the screen or, most importantly, the sound design. Stamina and health state are represented by how heavy your breathing and footsteps

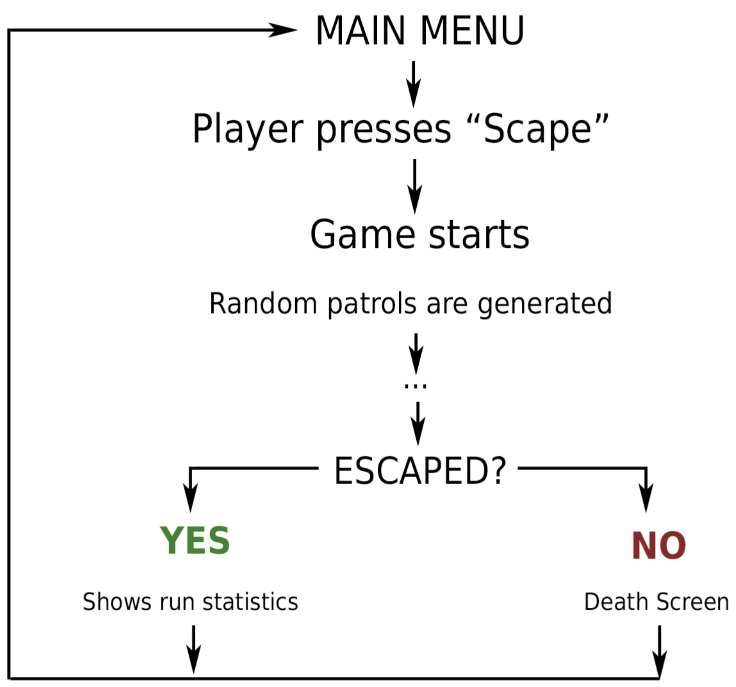


Figure 3.2: The demo game flow is very straightforward.

are, the type of terrain you are walking on can be identified by the sound of your footsteps, and so on. The only text the player will get on screen are two big announcements: rather "YOU ESCAPED" or "YOU DIED" at the end of each run.

Chapter 4

Work Development and Results

Contents

4.1 Work Development

Before even starting developing anything related with the demo or systems involved in it, it was time to either find or create the assets to use. Most of the audio assets come from royalty free libraries, but there was one specific thing very difficult to find for free with the right quality: breathing effects. This part was recorded by myself, edited and introduced to the audio engine in a homemade fashion. The result lacks professional finish, as I'm not a voice actor myself, but it serves it's purpose for the game demo. Once this was solved, it was time to start scripting and building the demo

On a first stage of developing, everything started by creating the demo structure itself: everything related to the game engine. Beneath the dark, non-graphical appearance of the game you can find a set of different low poly models placed with their collision boxes and other components attached to them. This worked as the "skeleton" of the demo, making it possible to build the level using all the prefabs generated. Using Unity navigation mesh system to create enemy and car AI,and made use of particle systems and lightning to create all the visual clues.

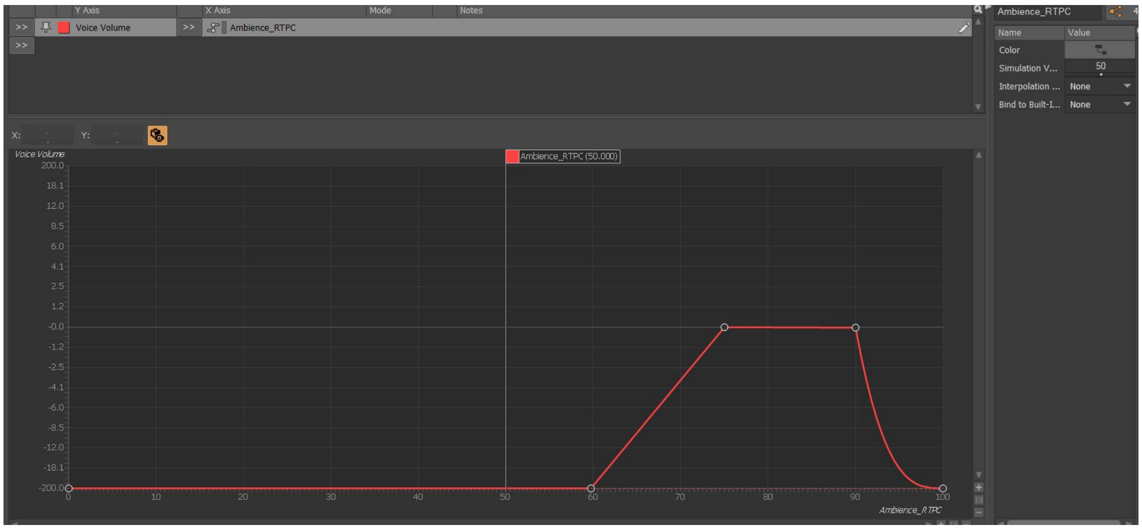
On a second stage, it was to time to code and prepared all audio systems. This was by far the most complex and time consuming part (as reflected on the Gantt diagram used above). Using Wwise functionalities, everything related with the audio engine was in-game linked. Coordinated footsteps sound, dynamic ambience and effects, etc. Unity accesses the sound bank where all sounds are stored and prepared for it's correct use.

Once everything sounded and felt right, the level was built and debugging, game balancing tasks were done. Most of the parameters defining gameplay, such as detection radius and running speed, has been set up to create the most intense, fair experience possible for the user.

4.2 Sound

4.2.1 Dynamic ambience

Dynamic ambience is composed by multiple audio layers. Each of this layers is automated in sync with the rest of them, making it possible to alternate volume, filters or any kind of effect on the fly using an RTPC ("Real Time Control Parameter"). This RTPC is first defined in the audio engine, and later accessed trough Unity in order to change its value and shape the dynamic ambience to fit the situation the player is in. The way the layers increase in volume and presence is pseudo-randomized, with little variations in its curve to create an slightly different feeling every time. (see Figure 4.1)



Ambience_RTPC

Name	Value
Color	
Simulation V...	50
Interpolation ...	None
Bind to Built-I...	None

Figure 4.1: RTPC value curve controlling level parameter on one audio track.

4.2.2 Dynamic breathing

The players character breathing system works differently to ambience generation. It also uses an RTPC to control in which point the player is. The more tired the player is (as it has been running trough the level for a long time), the heavier and more present the breathing becomes. There are four different breathing intensities, separated in four different audio stems. The transition between this stems is seamless, going from one to another as the RTPC changes according to the remaining stamina the player has.

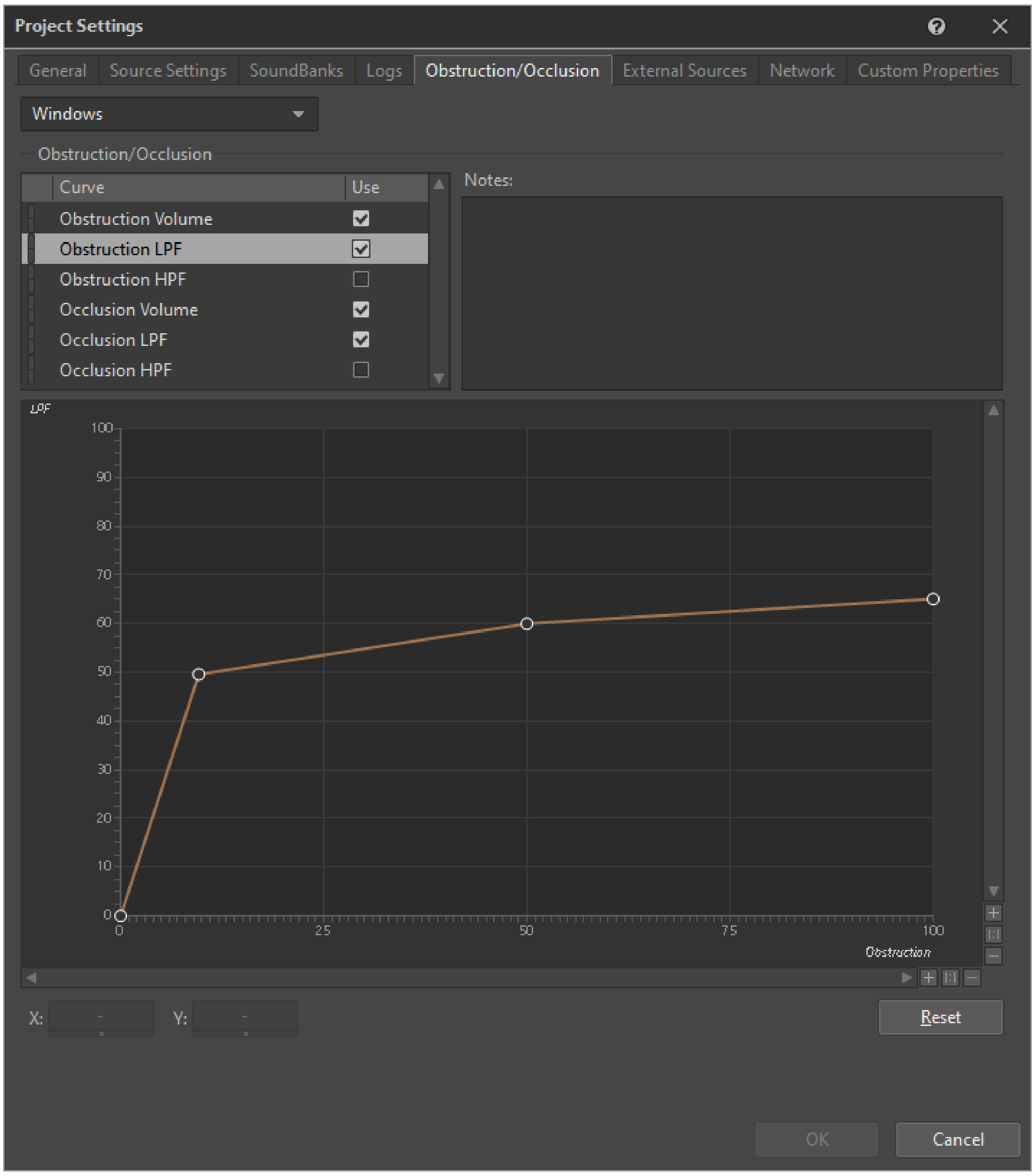
4.2.3 Footsteps

Footstep sound works in sync with the footprints FX. Every time the particle system generating footprints fires one particle, a new footstep sound is generated as well. This come inside a Wwise Random Container, where, every time a footstep sound has to be played, the audio engine will choose one of the sounds in the named container and will fire it. Using between 5 to 8 different footstep sounds, this creates a much more organic, realistic feeling. Also, the system takes into account the surface where the players in moving. The demo only includes concrete and dirt, but an unlimited amount of them could be added. An specific script will ask the audio engine to fire a different random container, one for each terrain type, each containing footstep sounds in the different materials. Regarding the enemy footsteps, they work similar to the player ones, but they get filtered as they play in the distance. Using the obstruction/occlusion system Wwise bring to the table, you can control (between others) a low pass filter. The most distant enemy footsteps will sound muffled and sturdy, sometimes almost imperceptible. When enemies get close, and their footprints can be seen, the sound becomes bright and present. This keeps the focus of the player on closer threats. This system can be brought to the next level by using other objects in the scene as "refractors" of sound, creating the possibility of emulating a much more complex audio system with realistic, real-time generated audio reverberations and echos, as well as composed occlusion effects with audio bouncing off corridor edges in a more realistic manner. For the purpose of footsteps in the demo, considering the reduced number of features present in the free license of Wwise regarding this system, occlusion was not utilized. (see Figure 4.2)

Apart from this, and by using Wwise spatial audio system, footsteps are generated at the correct position inside the stereo specter. Objects in the level server as obstructions for the purpose of spatial audio, shaping the sound considering the relative position of the player to the sound source. (see Figure 4.3)

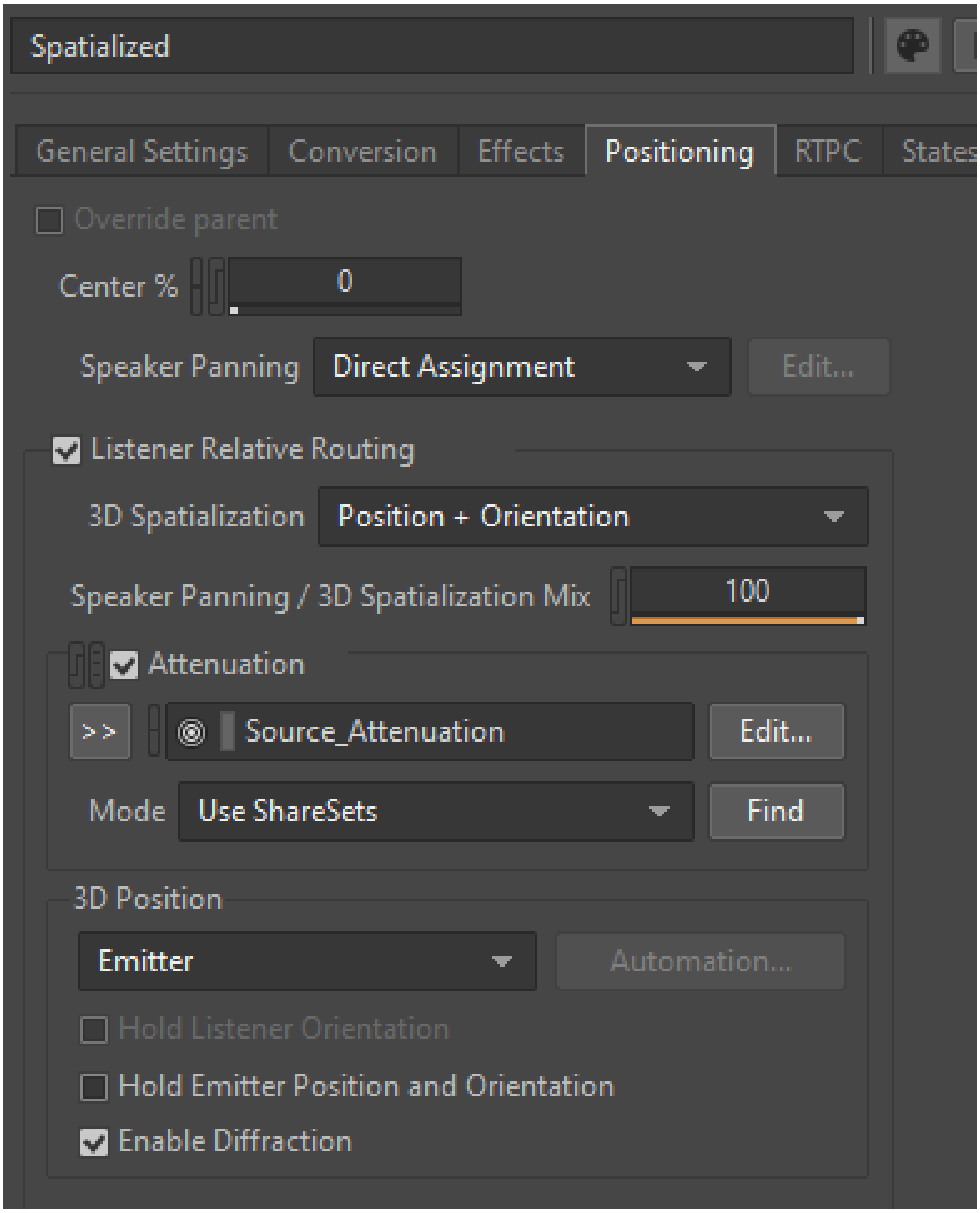
4.2.4 Car sounds and other effects

Car sounds, as the rest of sound effects, work in a simpler way. Car engine humming, as well as electric fence static noise are spatially placed for the player to identify where the sound comes from and move accordingly. The objects themselves have a sound track that will loop, maintaining a constant sound stream that will get quieter as the player walks away.



Curve	Use
Obstruction Volume	✓
Obstruction LPF	✓
Obstruction HPF	☐
Occlusion Volume	✓
Occlusion LPF	✓
Occlusion HPF	☐

Figure 4.2: Obstruction lowpass filter curve for enemy footsteps.



Override parent

Center % 0

Speaker Panning Direct Assignment

✓ Listener Relative Routing

3D Spatialization Position + Orientation

Speaker Panning / 3D Spatialization Mix 100

✓ Attenuation

Source_Attenuation

Mode Use ShareSets

3D Position

Emitter

Hold Listener Orientation

Hold Emitter Position and Orientation

✓ Enable Diffraction

Figure 4.3: Spatial audio panel in Wwise, where u can choose prefered configuration for sound 3D treatment.

4.3 Gameplay

4.3.1 Player character: Controls

Blindfolded controls follow the classic, archetypal control design philosophy behind every modern action game played on a keyboard; where left hand covers movement and functionalities linked to keys while right hand is used for mouse movement and clicks (see Figure 4.4).

4.3.2 Non playable characters: AI

To create enemy behaviour, a finite state machine has been implemented, allowing to create a set of different states and how they are linked in order to make them capable of adapting to multiple situations and creating a challenging, better experience. In any moment where the player is far away from persecutors, they stay patrolling, reaching a set of in game space points in a sequential order. This points are scattered trough the level, and are assigned manually to each patrol. As the player gets near them, the compare the noise the player character is generating in that exact moment (waling or running, intense breathing ,etc) and start chasing it if they heard something. If the player gets far enough, or if persecutors stop hearing noises for a short period of time, they'll get back on their patrolling routine. For balance purposes, persecutors are as "blind" as the player (see Figure 4.5).

4.3.3 Resources

To keep track of the current state of the player, there is a set of stats that, although are not shown to the player in any moment, define how the character is doing. When playing the game, all this information gets to the player trough the different audio systems.

- Stamina is represented by an integer that can go up to 100. Each second, a number of stamina is regained if the player is not running, in which case will decrease a certain amount. This stamina value is directly linked to the dynamic breathing RTPC, making it possible to have the player character breathe heavily and continuously as his stamina is about to deplete, and sounding completely calmed again when he has walked for a while.

- Danger is also measured with an integer. In this case, this integer starts at 0, and can go up the more enemies are near you, how close they are to the player or if they are aware of your presence and by so, persecuting you. The same way as stamina, the value of danger is linked to the RTPC controlling the dynamic ambience, making it possible to quickly set an obnoxious atmosphere of drones and string when then player is being chased by multiple enemies, and come back to just the calmed wind and some noises if he manages to scape.

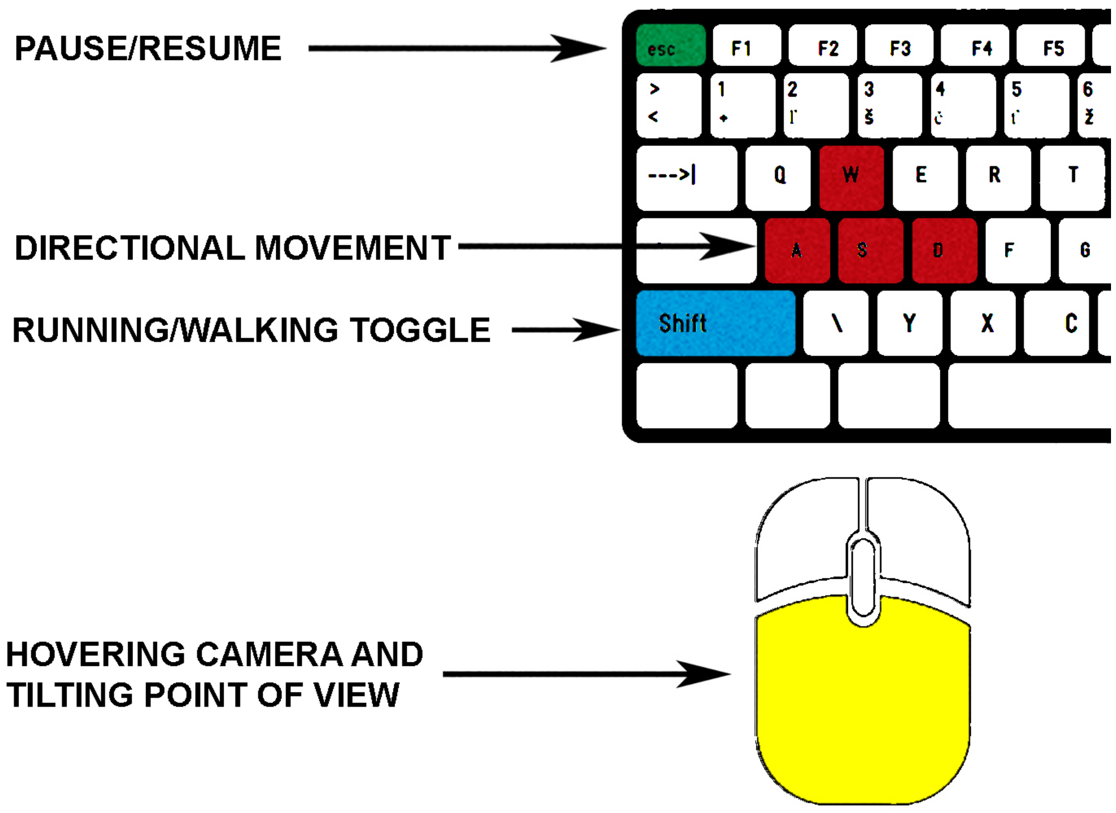


Figure 4.4: PC controls.

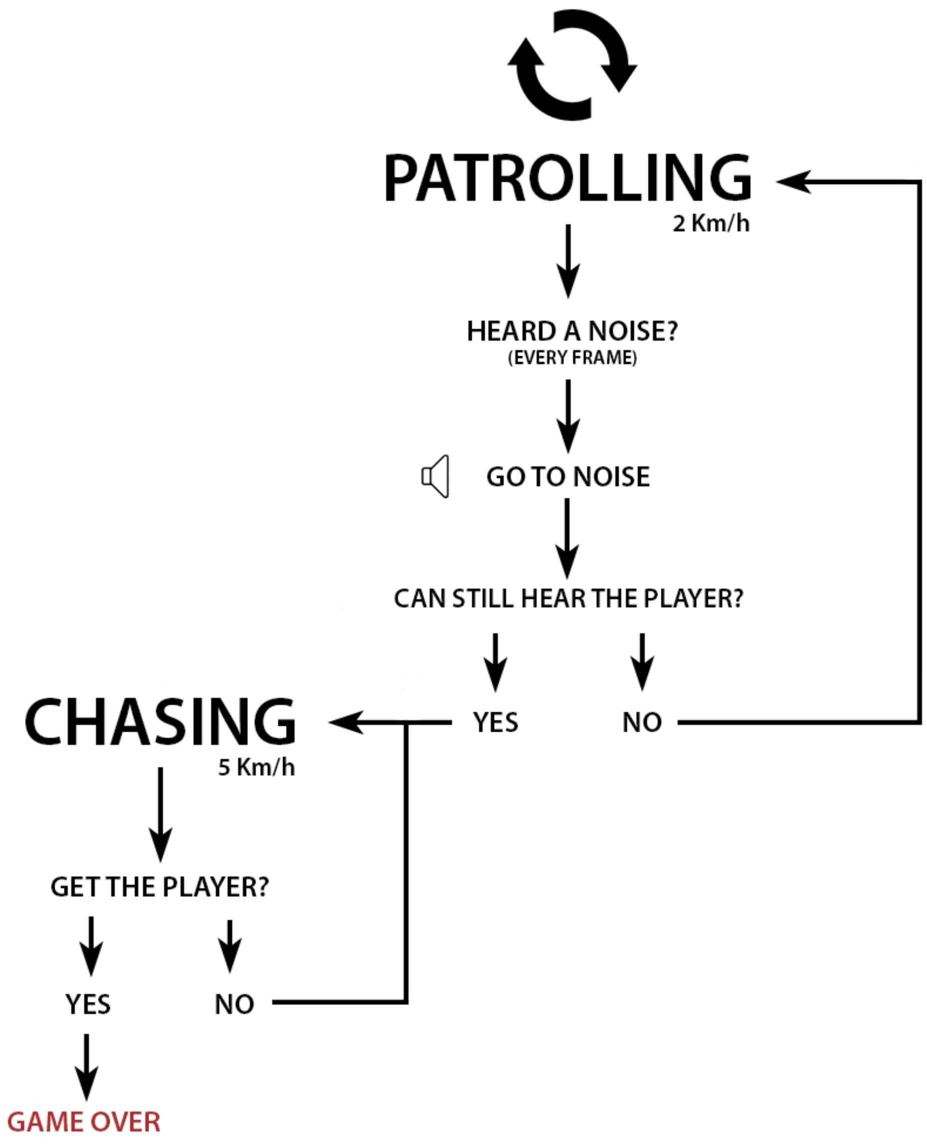


Figure 4.5: NPC behaviour flow.

- Footstep noise is represented by a collision box surrounding the player. This box gets bigger as the player is more noisy (running, breathing etc), and, it collides with the detection range of any enemy, it will start a persecution.

4.4 Assets

Most of the assets used in the demo are handmade. To create everything sound related, Ableton Live 9, a digital audio workstation, was used. This kind of software lets producers create, mix and arrange audio signals in a complete digital work space.The assets created this way were then exported and set into Wwise with already adequate volume levels, dynamic range and tonal balance. To put it simply, they were imported into the audio engine already processed and mixed to sound good, leaving only the scripting and organization part to be taken care of.

4.4.1 Environmental sound

The environmental sound in Blindfolded conforms the dynamic ambience, consisting of:

- White noise, created using an oscillator, then exported to a stem and loaded into Wwise. It fills the ambience with random frequencies, creating a chaotic feeling.
- Drone sound, playing a basic pitch shifted harmony, synthesized and then exported to audio, then loaded intro Wwise. It fills big part of the low end of the audio spectrum, creating the tension and unsettling feeling.
- Dissonant strings, coming from a sample of already recorded strings with a free license. They server as the top, most discordant and gritty part of the ambience.

(see Figure 4.6)

4.4.2 Breathing

Four different tracks of breathing were recorded and processed, then loaded into Wwise. They increase in intensity and weight, going from a pleasant, slow breath to an exhausting heavy breathing. As most of this assets are usually recorded in professional, an-echoic rooms in high end studios, the recorded signal had to go trough a gate, making sure everything below certain volume threshold was not recorded, so no background noise or extra room tone was inside the signal. The tracks were then compressed and equalized properly, so they would sit nicely in front of the ambience, as they are the nearest audio source during gameplay.

4.4.3 Visual clues and effects

All of the visual clues and effects used in the demo are built using Unity's particle system tool. Sprites used in this implementations where created in Photoshop, following

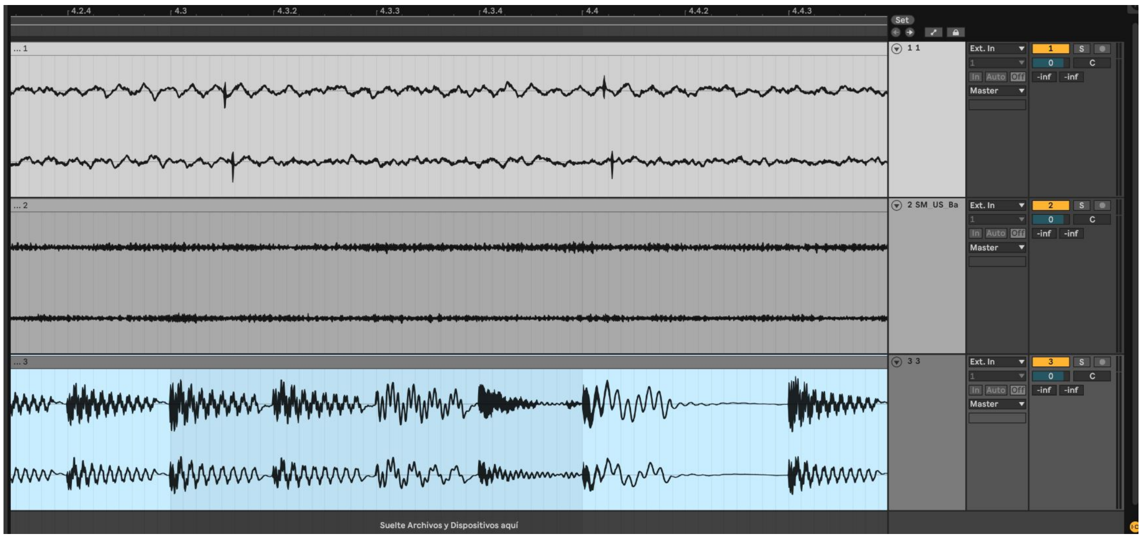


Figure 4.6: Different ambience tracks beign processed and arranged in Ableton Live 9 DAW.

references like real life items such as footprints or lightning. In the case of footsteps, for example, the particle system will generate one new footprint every time the player advances a little distance. The number of maximum footprints stays the same no matter the speed at which the player is running, making the older ones disappear. If the player stays move less, footprints already generated will start to fade away as time goes by, same happens with NPC's. In the next figures you can see some of the visual results (see Figure 4.7) (see Figure 4.8) (see Figure 4.9).

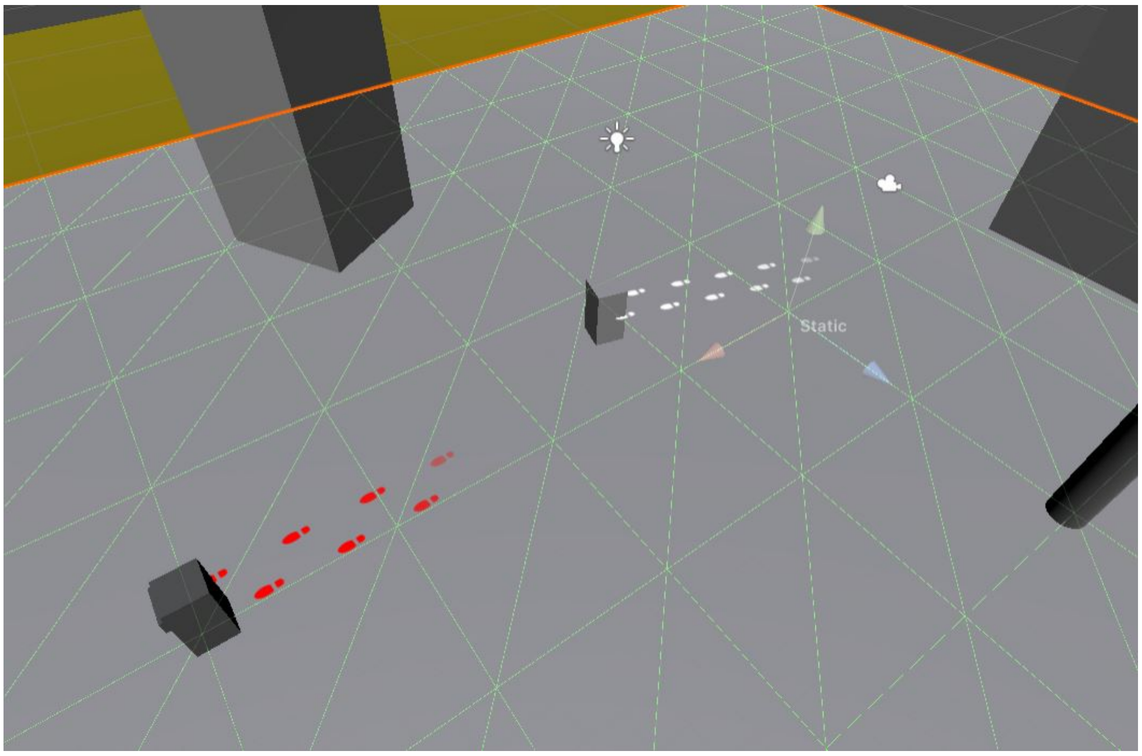


Figure 4.7: Footsteps generation "behind the curtain".

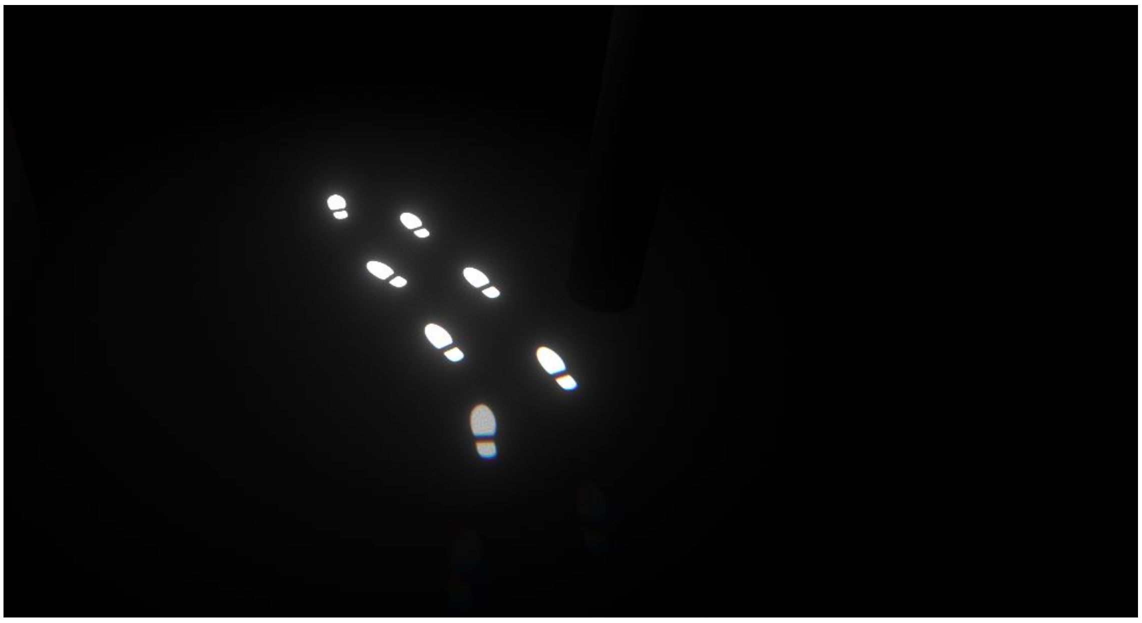


Figure 4.8: Footprint PS.

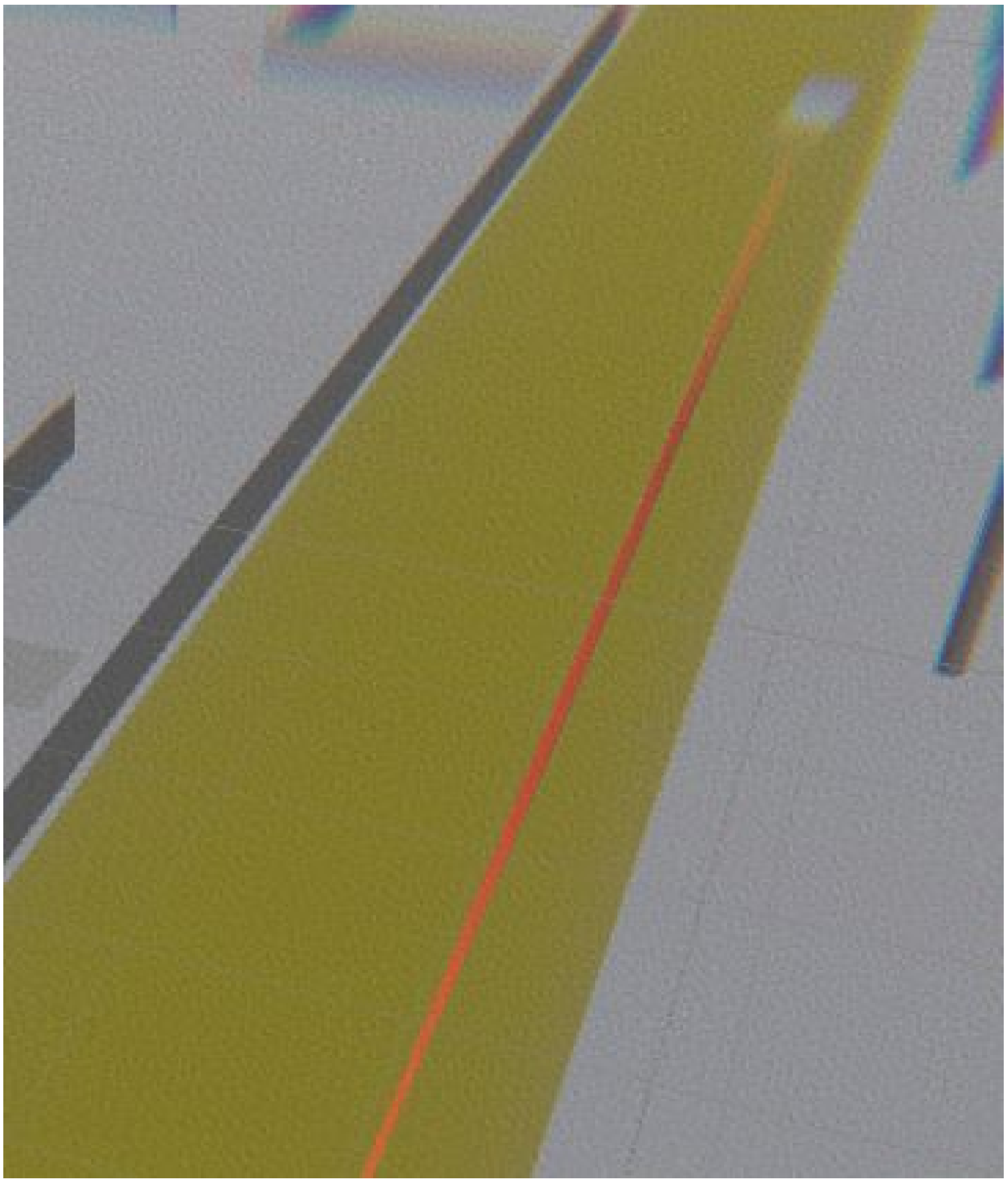


Figure 4.9: Car tracks "behind the curtain".

Chapter 5

Results

Contents

5.1 General Conclusions

As a musician and producer that moves mainly (only) on the digital world, I found this experience, more than anything, very enriching and complementary. Videogame studios search for very specialized individuals, as videogame development requires a higher level of segmented knowledge.

Learning to use Wwise and developing this demo served to set the bases in the creation of both vertical and horizontal dynamic audio systems, and combining it with self-made assets created a very centered perspective of the actual work of audio specialized developers in the industry. Even if some of the assets couldn't meet professional quality standard, as they have been recorded and processed in nothing more than a humble home studio, getting this vision on how much effort and time has to be put in sound implementation has been very revealing, as sound always comes as the "missing brother" in almost any cultural product.

5.2 Conclusions: Unity

Regarding Unity, the demo turned out simple, but effective. It was not the main goal of this project to create a very complex or long experience, but having to take care of every aspect of a little game ended up boosting my Unity knowledge.

The demo is a finished experience, highly re playable and stable, but, as specified in the section below, one of the main problems was the lack of time to create a bigger experience with all the initial mechanics implemented. Some of the extra features for Blindfolded, such as enemy variation and procedural level generation, had to be taken away in order to focus on a more polished and deep audio system.

5.3 Conclusions: Wwise

Regarding the audio engine, all the main objectives and expectations were filled. The idea of implementing both vertical and horizontal dynamic audio systems is fulfilled and these are integrated in the demo, completely functional. This included the use of containers, RTPC's and switches to establish all the internal logic and asset coordination of those systems. This also includes the use of spatial audio tools. These where not considered to be used in the first place, but turned out to enhance the footsteps system a lot. Everything introduced int his demo remains to be tested with a greater quantity of stems, more samples and, in general, a bigger game.

The process of learning from zero how audio engines work and how to work with them in conjunction with Unity has been very enriching, and gave a better, more in-depth perspective on game making.

5.4 Conclusions: Audio asset creation

Asset creation ended up being the most time consuming and complex part of it all. Recording, processing and preparing assets to be used directly into development was a long process plenty of missteps, where a lot of extra hours were spent. Far from the desired professional feeling of the results, it has been very enriching. The lack of proper studio equipment really reduced the quality of this recordings, weakening some aspects of the demo gameplay experience.

5.5 Future work

Creating a richer experience in the demo should be the main focus. This could be done by, first of all, adding procedural level generation. Implementing a system that procedurally generates NPC's, routes for them, as well as places trees, roads and other obstacles should be a great addition to it.

Some extra ideas in the first design document were not implemented due to lack of time or other systems having more priority to be done, but adding more types of NPC's with their own sound set would also help enhance the demo.

I would also like to point out that some of the assests, specially everything regarding to the main character breathing, could be re-recorded on a Foley room with better acoustic conditions, leading to a more polished breathing intensity system with smoother transitions.

Bibliography

[1] Ableton. *Making Music Creative Strategies for Electronic Music Producers.*

[2] The Creative Assembly. Total war series. Dynamic audio implemented using Wwise.

[3] Tazman Audio. Fabric. https://www.tazman-audio.co.uk/fabric.

[4] AudioKinetic. Get started using wwise. it's as easy as 1, 2, 3!

[5] AudioKinetic. Wwise official video tutorials. https://www.audiokinetic.com/learn/videos/.

[6] AudioKinetic. Wwise user guide.

[7] CAPCOM. Resident evil 3 (remake). Dynamic audio implemented using Wwise.

[8] Karen Collins. *Game sound.* The MIT Press, 2008.

[9] Nathan Galinier. Wwise tutorials. https://www.youtube.com/user/BusyMonkee.

[10] Mitch Gallagher. *Acoustic design for the home studio.* Thomson, 2006.

[11] Damian Kastbauer. *The Wwise Project Adventure.*

[12] Kojima Productions Co. Ltd. Death's stranding. Dynamic audio and adaptive music implemented using Wwise.

[13] Playdead. Inside. Dynamic audio implemented using Wwise.

[14] Kelly Surette. *Creative Miracles: A Practitioner's Guide to Adaptive Music Instruction.* Warrior Woman Publishing, 2020.

[15] Unity. Unity official manual. https://docs.unity3d.com/Manual/index.html.

[16] Wikipedia. Fmod. https://fmod.com/.

[17] Wikipedia. Wwise article. https://en.wikipedia.org/wiki/Audiokinetic$_{W}wise$.

[18] Wwise. Wwise 101. https://www.audiokinetic.com/courses/wwise101/.

[19] Wwise. Wwise sample projects. https://www.audiokinetic.com/resources/project-samples.

Appendix A

Source code

This section contains some of the code used in the project. Notice that some of the scripts have been simplified to fit in this report, and only some of them are featured, trying to give a general idea on some of the most used systems in the game. The game repository will have all code available.

Player Control Script

```
/* Player Control Script */

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{

    public float Speed = 1; //Walkng Speed
    public float speedAcceleration = 25;
    public float speedDeceleration = 100;
    private float maxSpeed;

    public float maxStamina ;
    public float currentStamina ;
    public float staminaFallRate ;
    public float staminaRegainRate ;

    public float noise = 0;

    public float ambienceRTPC = 0;
    public float breathingRTPC = 0;


    public List<Transform> enemyTransforms;
    private Transform closestPatrol;




    private void Start()
    {
        AkSoundEngine.SetState("Dead_or_alive", "Alive");
        AkSoundEngine.SetRTPCValue("Ambience_RTPC", ambienceRTPC);
        AkSoundEngine.SetRTPCValue("Breathing_RTPC", breathingRTPC);
        AkSoundEngine.PostEvent("ambience_event", gameObject);


    }

    // Update is called once per frame
    void Update()
    {

        if (Input.GetKey(KeyCode.LeftShift) && currentStamina >= maxStamina/4)
        {
            maxSpeed = 5;
            //Lose Stamina
            currentStamina = currentStamina - (Time.deltaTime / staminaFallRate);
            //Change noise detection accordingly
        }
        else
        {
            if(currentStamina <= 25)
            {
                maxSpeed = 0.5f;
            }
            else
            {
                maxSpeed = 1.5f;
            }

            if ((currentStamina < maxStamina) && !Input.GetKey(KeyCode.LeftShift))
            {
                //Regain Stamina
                currentStamina = currentStamina + (Time.deltaTime / staminaRegainRate);
            }
        }

        if (Speed >= maxSpeed)
        {
            Speed -= Time.deltaTime / speedDeceleration;
        }

        else
        {
            Speed += Time.deltaTime / speedAcceleration;
        }

        noise = Speed * 10;
        Movement();

        closestPatrol = GetClosestEnemy(enemyTransforms);
        SetAmbience(Vector3.Distance(transform.position, closestPatrol.position));
        SetBreath(currentStamina);

        AkSoundEngine.SetRTPCValue("Ambience_RTPC", ambienceRTPC);
        AkSoundEngine.SetRTPCValue("Breathing_RTPC", breathingRTPC);
    }



    Transform GetClosestEnemy(List<Transform> enemies)
    {
        Transform tMin = null;
        float minDist = Mathf.Infinity;
        Vector3 currentPos = transform.position;
        foreach (Transform t in enemies)
        {
            float dist = Vector3.Distance(t.position, currentPos);
            if (dist < minDist)
            {
                tMin = t;
                minDist = dist;
            }
        }
        return tMin;
    }

    private void SetAmbience(float distance)
    {
        if (distance < 90f)
        {
            ambienceRTPC = (100 - distance) - 10;
        }


    }

    private void SetBreath(float stamina)
    {
        breathingRTPC = 100 - currentStamina;
    }

    void Movement()
    {
        float horizontal = Input.GetAxis("Horizontal");
        float vertical = Input.GetAxis("Vertical");

        Vector3 movement = new Vector3(horizontal, 0f, vertical) * Speed * Time.deltaTime;
        transform.Translate(movement, Space.Self);

    }
}
```

Enemy Behaviour Script

```
/* Enemy Behaviour */

using System;
using UnityEngine;
using UnityEngine.AI;
using UnityEngine.SceneManagement;

public class Enemy : MonoBehaviour
{
    public NavMeshAgent agent;
    public PlayerMovement playerNoise;
    public Transform objectToChase;
    public Transform[] waypoints;
    public int currentWaypoint = 0;

    public enum EnemyStates
    {
        Patrolling,
        Chasing
    }

    public EnemyStates currentState;


    void Update()
    {
        float playerDistance = Vector3.Distance(transform.position, objectToChase.position);
        if (playerDistance > 10f)
        {
            agent.speed = 1.5f;
            currentState = EnemyStates.Patrolling;
        }



        if (currentState == EnemyStates.Patrolling)
        {
            if (agent.remainingDistance < agent.stoppingDistance)
            {
                currentWaypoint++;
                if (currentWaypoint == waypoints.Length)
                {
                    currentWaypoint = 0;
                }
            }
            agent.SetDestination(waypoints[currentWaypoint].position);
        }

        if (currentState == EnemyStates.Chasing)
        {
            agent.speed = 4f;
            agent.SetDestination(objectToChase.position);
        }

        InNoiseRange(playerDistance);
        //SetAmbience(playerDistance);

    }


    private void InNoiseRange(float distance)
    {
        if(distance < playerNoise.noise)
        {
            currentState = EnemyStates.Chasing;
        }

    }
}
```

Footseps

```
/* Footsteps */

using UnityEngine;
using UnityEngine.AI;

public class  PlayerFootsteps : MonoBehaviour
{
    public ParticleSystem system;
    public Material activeParticleMat;

    Vector3 lastEmit;

    public float delta = 1;
    public float gap = 0.5f;
    int dir = 1;
    static RingbufferFootSteps selectedSystem;

    void Start()
    {
        lastEmit = transform.position;
    }

    public void Update()
    {


        if (Vector3.Distance(lastEmit, transform.position) > delta)
        {
            var pos = transform.position + (transform.right * gap * dir);
            dir *= -1;
            ParticleSystem.EmitParams ep = new ParticleSystem.EmitParams();
            ep.position = pos;
            ep.rotation = transform.rotation.eulerAngles.y;
            system.Emit(ep, 1);
            lastEmit = transform.position;
            AkSoundEngine.PostEvent("concrete_walk", gameObject);
        }

    }

}
```

Collision Management

```
/* Collision Management */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CollisionManager : MonoBehaviour
{
    public PlayerMovement playerMovement;

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Tree")
        {
            Debug.Log("Choque con arbol");

            if (playerMovement.Speed >= 3.5)
            {
                playerMovement.Speed = 0;
                playerMovement.currentStamina = 0;
            }
        }

        //Chocar con valla

        if (collision.gameObject.tag == "Enemy")
        {
            Debug.Log("Choque con enemy. GAME OVER.");
            AkSoundEngine.SetState("Dead_or_alive", "Dead");
        }

        //Chocar con coche

        if (collision.gameObject.tag == "Car")
        {
            Debug.Log("Choque con coche. GAME OVER.");
        }
    }
}
```