Goldsmiths University of London

Final Project Report

Music Computing

Candidate: 33368191

Listen

Author:
Nathan de Castro

Supervisor:
Sylvia Xueni Pan

May 15, 2017

Acknowledgments

First and foremost I would like to thank my project supervisor, Dr. Xueni "Sylvia" Pan, for the constant support and practical advice she has given me during the whole project. I would also like to thank Goldsmiths University of London for giving me the opportunity to pursue this project. Studying in such a unique and amazing University has been delightful. Lastly, I would like to thank my family for both the financial and moral support they have given me during every single year of my degree.

Abstract

Project report on the implementation of the game "Listen"; a virtual reality navigation video game based on echolocation, where the player is left in the dark inside a big house hunted by a mysterious ghost-like entity. They must navigate undetected to locate the light-switch. The report covers everything from the concept, to prototype, to the implementation of the game itself.

Contents

List of Figures

Chapter 1

Introduction

In most cases, when we talk about the future of gaming, we see photo-realistic graphics running smoothly on a single chip the size of a watch. Now this sort of thing could very well be possible in the coming 10 to 15 years; but what about sound? Sound has only very rarely been at the core of a video game's targeted gameplay, and thus we have not seen many innovative ways to use it.
But recently, as virtual reality became more accessible to the public, thanks to relatively affordable prices and the reduced size of equipment, game creators have started to look at new ways to make video games attractive. Today's computer graphics standards are higher than ever and demand immense computational power to render. Unfortunately, the rendering of virtual reality games requires even more power to provide a comfortable gaming experience. We are talking about doubling the number of screens and moving the defaults for frame rates from a comfortable 45-60Hz up to 75-90. However, sound is drastically less demanding and is, therefore, becoming an attractive field to work on for developers. As both a programmer and amateur audio engineer, I am well equipped to start working on my very own virtual reality game focusing on sound as a medium to navigate. It will also be a great opportunity utilize the making of this game as a way to better understand and use sophisticated audio analysis algorithms such as FFT (Fast Fourier Transform).

I will first go through the necessary notions one needs to grasp on what sound is and how it is perceived before introducing the concept of 'echolocation'.

I will then explain how echolocation can be used as a way to navigate in space as well as how this unique navigation method can be at the core of a great video game concept. Furthermore, one of my slightly "concealed" intents is to help sighted people relate to visually impaired individuals through this novel mechanic.

By comparing and referring to other games with similar initiatives, I tried to remain up to date with what was happening around me in the tech world in order to, with a few twists, try and contribute in my own small way to the future of virtual reality gaming.

1.0.1 My Aims:

- Implementing a game concept never seen before
- Implementing complex real-time audio analysis tools using only the Unity's API in C-Sharp
- Create a compelling virtual reality experience inside a credible 3D environment
- Create a compelling artificial intelligence
- Explore the idea of using sound as a medium to navigate in space
- Make players curious, scared and looking for more
- Learn as much as I can in every field of game development

Chapter 2

Background / Literature

2.1 Echolocation

Please note that for the purpose of simplicity and clarity, I will be considering sound in "every day" conditions, meaning under ambient temperature air and at sea levels of pressure.

2.1.1 What is sound?

Before we can begin talking about anything at all, I believe it is important to go through a few relevant definitions. Let us start with what constitutes the core aspect of my game: Sound. In short, *"Sound is a pressure wave [...] created by a vibrating object."* [1]
In other words, when an object vibrates, the air surrounding it is being compressed and relaxed forming a "sound wave." The rate at which these compressions occur constitute the frequency and are equals to the related objectâĂŹs vibrating rate. If the frequency is between $\sim 20Hz$and $\sim 20kHz$, than most humans will be able to "hear" it (although statistically, the average revolves more around $\sim 15Hz$to $\sim 18kHz$) [2].

Our ear contains many complex organs responsible for the recognition of pitch, amplitude, and phase. We will focus on two in particular:

- The "ear drum"
- And the "cochlea"

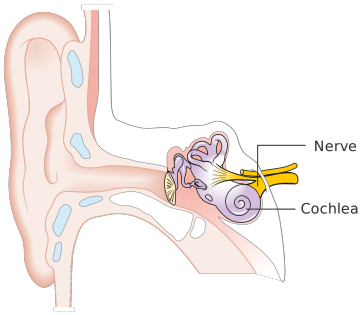


Figure 2.1: Representation of the inner ear system [4]

The eardrum acts very much like a microphone. It is a membrane receptive to the sound waves coming from outside the ear. It is "built" to be pushed and pull with the same strength and frequency as the sounds coming in. It then transfers that information to the following organs to be filtered and analyzed before becoming relevant. Although the eardrum is usually the one receiving all the fame, much like the lead singer in a band, the cochlea is probably the most complex and fascinating organ at the back of the stage for its incredible ability to detect pitch.

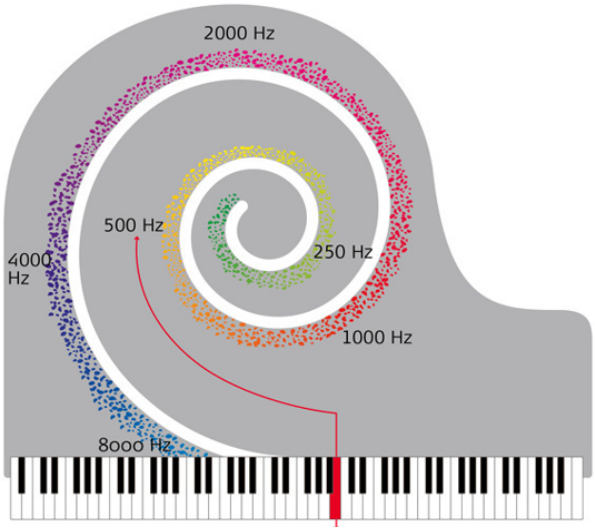


Figure 2.2: Representation of the cochlea [3]

As shown in the image above, the cochlea contains a plethora of sensors (sensitive hairs) distributed around a snail-like shape with each one responsible for the detection of a small range of frequency. These sensors will vibrate when exposed to their respective frequency sending away a signal to the brain. This is why when we press two keys next to each other on a piano, the sound produced feels unpleasant. This is because two sensors sitting next to each other are being stimulated. This results in conflicting information for the brain to translate which is in term treated as a feeling of discomfort. This is part of a larger concept introduced by Masahiro Mori called the

uncanny valley where a similar feeling arises when we see, hear or touch something that is close to being familiar but not entirely and ends up confusing our brain into thinking it must be something wrong for us and we should probably avoid it [1].

2.1.2 How do we locate them?

We now know how to perceive pitch and amplitude. These two senses are already very powerful tools for us to recognize what it is that is making the sound (Frequency spectrum) as well as how close it is from us (Amplitude) but how do we locate them?

Thankfully, as humans we are given a *pair* of ears. Just like we have two eyes to discern depth we have two ears to perceive orientation. The way it works is quite clever. Our brains can measure the slightest differences in phase between one ear and the other.
The slight shift in phase allows the brain to calculate the amount of time the sound took to reach one ear before the other. This combined with the change in amplitude between the two allows for a better indication of the angle and distance. The brain makes these calculations continuously at microsecond rates: Updating our spacial awareness continuously [5].

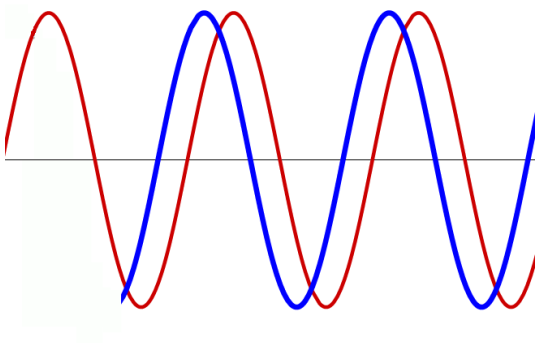


Figure 2.3: Red for left ear, blue for right ear

For example, in this case, it is safe to assume the sound came from the left since the sound reached the left ear before the right one at a slightly lower amplitude.

2.1.3 What is echolocation?

Echolocation allows the navigation in space using reflected sound. Bats are one of the few animals famous for their ability to use echolocation as their primary way to navigate [7]. The bat produces an ultrasonic sound and waits for the sound to come back to their ears to calculate how far objects are (Because they go out and hunt at night, they are dependent on their echolocation skill to locate preys and avoid obstacles while flying).

[1] It is important to note that the uncanny valley may be generationally and culturally dependent where "younger generations, more used to CGI, robots, and such", as well as people used to other cultures, "may be less likely to be affected by this hypothesized issue." [6].

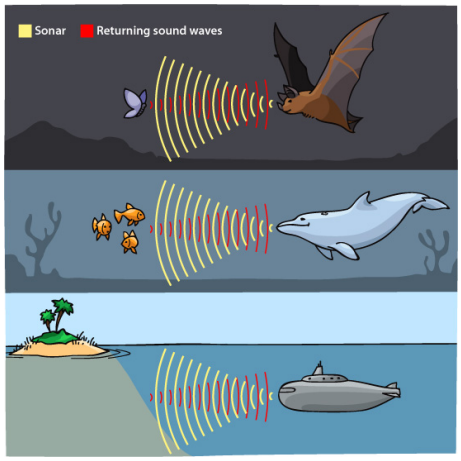


Figure 2.4: Representation of the a bat's echolocation method [7]

2.1.4 Can we, humans, use echolocation?

We, humans, use this principle on many of our everyday devices like radars, speed cameras [2] however only a few humans on the planet manage or managed to use echolocation in the same way bats or dolphins do. In most cases, these people were either blind or had some sort of disabilities that forced them to use other methods than sight to "see".
One of the most famous examples of this is the case of an incredible individual called 'Ben Underwood' [8] who makes "clicking" sounds with his mouth and waits for the reflected sound to come back to his ears to determine the location and size of objects around him. This exceptional navigation method allows him to walk, run and even ride his bike without the use of any external help (e.g. Walking stick, guide dogs).

2.1.5 How can we implement this into a game?

People capable of echolocation have had time to let their brains compensate for their lack of vision [3].However, I cannot expect all players to possess such an ability. One way to replicate this uncommon skill is to try and convert sound waves to visible waves. Or in other words, make sound waves visible.

A visually appealing way I found to do this is to replace what would be the propagated sound waves by a ripple of light making the area around the player visible for a short period of time. Some objects (e.g. A fridge) would constantly be making a sound, revealing what is around it. Yet I decided not include this feature in the end product since I thought it would make more sense to have the player rely on his ears to locate these objects.

[2] By using the Doppler effect principle
[3] Thanks to the incredible power of brain plasticity

2.2 Is it relevant?

2.2.1 Has it been done before?

Using sound to navigate in a video game is responsible for an almost inexistent part of the market today but, like discussed in the introduction, it is slowly emerging as graphical enhancements in games are slowing down. A great example of echolocation in a game would be "Lurking" [9]. In fact, the team behind this game is already working on a sequel called stifled [10].

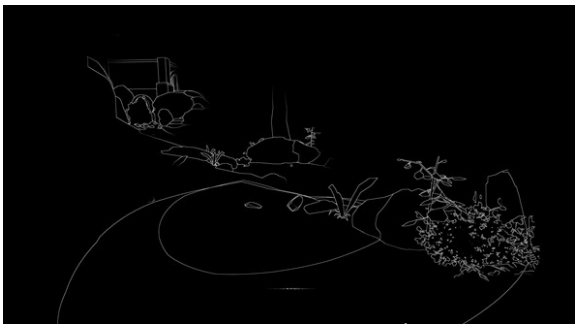


Figure 2.5: Screen shot of the game Stifled [10]

Both of these games focus on the fact that you have to walk or throw objects to reveal what is around you and allow you to move around but the key element of this game is to give the same ability to the enemy. Effectively making it blind when you remain silent.

"Perception" [11], meant to be released this year (2017), is yet another game that uses similar navigation techniques where the player is hunted by a 'presence, ' and you must use your echolocation ability scarcely to avoid detection. Although this gameplay seem to share much with mine at first, very little information has yet been released, and it is impossible to say whether or not it will be available in virtual reality or what the goal of the game will actually be. Moreover, the sounds will all be artificially created and not taken from a microphone. Nevertheless, it is a game worth stalking if I want to remain relevant and also a great source of inspiration. A wise person once said "It's about standing on the shoulders of giants".

Figure 2.6: Screen shot of the game Perception [11]

In a slightly different approach, two companies took the idea and brought it to mobile devices. The first, called "Blind Side Game" [13], does not even have any visuals and relies on a linear progression where the player can only walk forward or backward and is left with only his ears to understand what is happening around him.
The second being "Dark Echo" [12] probably qualifies as the closest equivalent to my game yet put into a 2D environment.

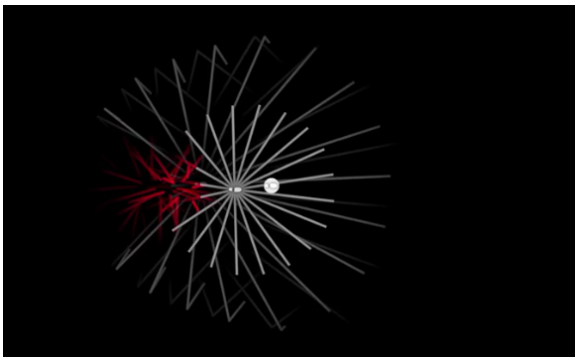


Figure 2.7: Screen shot of the game Dark Echo [12]

The point of the game is to move around in complete darkness and use the sounds your feet are making when walking to know where to go without being detected by shapeless entities.

So has it been done before? Yes. Is there room for improvement and novel ideas? Absolutely. What is left for me to do is to remain up to date with what is happening around me while staying focused on building my own compelling experience. It is clear that this topic's popularity is on the rise and I must be vigilant if I want to reside ahead of the race.

2.2.2 How can I contribute?

Working on an unpopular topic such as echolocation in-game makes me feel both very excited and terribly anxious. For it lets me bring my own new rules into the giant pool of video games, it also diminishes the number of items I can inspire form.

We have seen that only a handful of games have echolocation in mind, but none of them have yet to implement virtual reality into them. With the video game industry constantly coming up with new titles, originality and creativity are scarce. "Echolocation" provides me with an original theme to work on and virtual reality gives me the tech to take full advantage of it.

Unlike casual gaming methods, virtual reality allows you to place the player in a vulnerable position where simply looking away is no longer an option. There is a reason why "horror" is one of the most popular VR genres. Furthermore, in my case, the combination of virtual reality and echolocation blended with some spooky element is about as scary as a game can get.

2.2.3 Why virtual reality?

In order to accurately replicate such an uncommon way to navigate, you need the most immersive tech you can get today, and nothing compares to virtual reality in the realm of video game immersion. I decided to use the Vive from HTC for its ability to allow free movement within a designated area which in my opinion adds a whole new degree of immersion. To allow the player to physically strafe to avoid the ghost instead of pressing a button is essential in a game where you must move without making a sound. It also gives the player the freedom to crouch, sit and more importantly jump in a game where the microphone is both your best friend your worst enemy. Furthermore, being able to move your head to listen more closely to a sound brings a great sense of immersion that you simply cannot replicate on a 2D screen.

In a non-virtual reality game, you do not expect to be able to interact with objects around you nor do you often look up or down to make sense of your surrounding which in my opinion is great if your focus is gameplay but is less impressive if your focus is immersion. In a virtual reality game, as soon as you give the player two controllers, the first thing the player wants to do is to test the limits of their "new world".
"Can I grab this lamp?"
"Can I jump?"
"How does this object look from up-close?"
These are just a few examples of what an average player will ask themselves as soon as he is given an HMD (Head Mounted Displays) but none have asked themselves if the sounds they made while playing would affect the gameplay, well guess what, it does now. Time to tiptoe.

2.3 The limitation of virtual reality

2.3.1 Practicality

Virtual reality comes in many forms. From phone dependent devices like Google Cardboard [17] or Samsung Gear VR [18] to more advanced virtual reality headsets like the Oculus Rift [19] and the HTC vive [20].

The first bunch, solely relying on the phone's capacities to run applications. Although they run in very low resolution and graphics, they happen to be quite convenient to use thanks to their compact form and their availability (Who does not have a smartphone in their pockets?). Furthermore, they are light and comfortable and require no external help from another machine. Therefore a compromise had to be made by manufacturers and the gaming industry between computational power and practicality.

If you want to experience the true power of virtual reality, you will have to move your gaze towards the second group of devices. Unfortunately, with our current technologies, constructors are still manufacturing big, bulky devices. Cables are in the way and prevent the user from walking too far from the main unit which also happens to prevent the user from easily relocating all the equipment. Although companies try as hard as they can to make the setting up process as painless as possible; it is still very cumbersome to transport, and very long to setup since every piece of equipment needs to be powered. The room has to contain power plugs in almost all corners, unless the idea of perilous extension cords laying around everywhere sounds good to you.

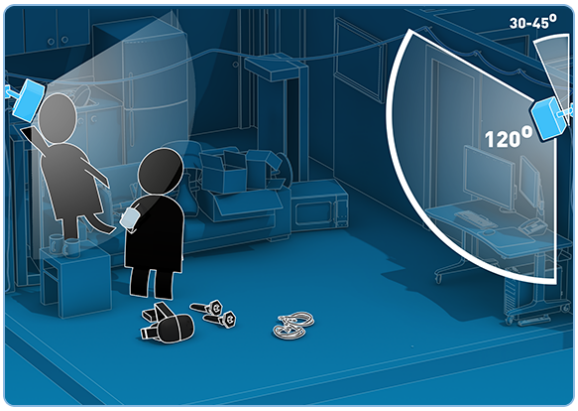


Figure 2.8: HTC Vive room setup example [21]

Wireless HMDs and portable units are starting to appear, but we are still a long way from the light, comfortable high definition wireless experience Christopher Lloyd promised us.

Figure 2.9: The future of VR? [22]

In the meantime, we are hindered by the immense graphical power virtual reality requires.

2.3.2 Efficiency

Since it is very difficult to render a realistic scene at high frame rates and resolution, we (VR game creators) must rely on other visual aspects to make our games look nice and sharp. One technique is to forcefully change the whole atmosphere to a less realistic one and go for a more cartoon-like look thanks to the use of shaders. Shaders can be used to allow the creator to come up with a beautiful looking game while avoiding the need for high-quality textures. Clever color adjustments and lighting effects can completely fool us. Games like "The Witness" [23] look absolutely fantastic

but remain lightweight and efficient. Having a cartoon-like shader can help to lighten the GPU load in many ways and having a black outline around objects in the game can drastically reduce the amount of anti-aliasing required.

Figure 2.10: Screen shot of the game: "The Witness" [23]

Another counter-intuitive way to enhance the overall visual quality of a game without overloading the GPU is by focusing on non-graphical aspects and rather direct our efforts into making actions in the game look sharp and fluid. If you are building a game with magical spells, make them pretty and exciting to use. If you are building a racing game, focus on making the car feel nice to control. Most of the time, if the gameplay is good, the overall visual atmosphere will also be enhanced.

Some really simple post processing effects like motion blur, blooming or screen shaking are amazing cheap ways to make a game look better without asking too much to the GPU.

2.4 Implementing virtual reality

My game will be compatible for both the Oculus and the HTC Vive thanks to Unity's beta compatibility with SteamVR. Once the user has run the steam VR room setup, all that is left to do is to drag a prefab onto the scene, scale it to the appropriate size and there you have it, full motion tracking. In reality, it took a bit more tweaking and updating deprecated code but at least that is what they advertise. Therefore, after a few clicks and only a few extra lines of code, I managed to see my game in virtual reality for the first time. I immediately noticed two things:

- First: It is much scarier than I expected it to be right out of the box
- Second: Reading text is incredibly uncomfortable

I thus decided to opt for audible instructions instead. As I discussed earlier, virtual reality can be relatively impractical and ask for the player to remove and put on their HMD form time to time. I could not have a single audio file played at the beginning telling you what to do and how to do it so instead I opted for a triggering system that waits for certain events to occur such as "player next to light switch" before playing the relevant instruction for this task. I also added a button on the controller the player can press to repeat the last instruction.

I wanted the experience to be as immersive as possible, so I decided to allow the player to interact with the objects in their immediate surroundings. For this, I had to build a script that

checks for a collision between the controller and a nonstatic object and applies different forces to it to allow for a "true to life" feel.

The script checks for a trigger collision between the controllers and an object and then waits for the player to press the trigger button after which the object becomes a child of the controller and a force vector is applied to the controller and the object that pulls towards the controller while the player holds the trigger.

Before opting for this final solution, I tried many other interaction methods such as making the object non-kinematic or making it no longer affected by gravity but both of these generated problems that rendered them unrealistic.
The first one allowed the object to go through objects and walls (effectively making it possible to exit the map) and the second allowed the player to lift objects that were drastically too massive to be manipulated with one hand.

Chapter 3

Work plan

3.1 Thought process

3.1.1 Original idea

The first virtual reality headset I had the chance to try out for myself was the Oculus DK2 during a tech conference. The only application I could try however was a roller coaster simulator where *shockingly* you sit inside a roller coaster and go for a ride. After the first minute, rather than feeling impressed and amazed at out the future of virtual reality will be, I removed the headset both feeling sick and disappointed at how blurry, 'laggy' and unrealistic the experience was. Luckily, a few years later, I got to try the HTC Vive. After a few minutes with it, my reaction was drastically different (Now we are talking). The latency was forgettable, and the resolution was sufficiently high that the graphics allowed me to immerse myself properly.

However, something struck me a couple of days after: Is it possible to replicate, the same feeling of "being somewhere else" I got from this experience, and bring it to visually impaired people? Is it feasible to make a sound experience compelling enough that a blind person could feel like they were somewhere else? I decided to look into binaural microphones and had the opportunity to record a few minutes of an environment using them so I could later listen to the recording somewhere else in the hopes of feeling "teleported". This experience was a success, and I am still amazed to this day as to how something so small and cheap as a binaural microphone could replicate the sonic atmosphere of a place so vividly. A well-known experience using the same technique, and one I cannot recommend enough, is called "Virtual barber shop" and can easily be found online [14].

The only problem with this technique is that it cannot be used in real time and requires the listener to always look in the same direction.
Thanks to the current motion tracking technologies and sophisticated 3D sound algorithms that come with the HTC Vive, I now have the tools I need to make a truly immersive experience.

3.1.2 Steps to final idea

I can finally draw my attention to my final project: Creating a fully immersive virtual reality game focusing on sound. At first, I wanted to make a game exclusively for visually impaired people. I wanted a blind person to be able to feel the same rush I had felt when I first put the HTC Vive.

The only way to achieve this was to have incredibly accurate sounds. I wanted more than just a relocation experience. As I was gathering ideas, I fell upon an interesting way to use virtual reality as an interactive storytelling game where the player has control over time and must understand why something specific happens in the scene by collecting information from dialogs around him [15]. My initial plan was thus to recreate the atmosphere of a bar containing many interactive dialogs you would have had to pay attention to in order to understand why the situation ends up in a certain way (In the way of a detective). However looking for visually impaired people to test my game and give feedback quickly became too complicated and involved many steps of paperwork I could not be dealing with in a reasonable time frame so I had to come up with a new idea: A game fun to play for both sighted and visually impaired people.
I thought of a game revolving around an intelligent use of the "play area" coming with the HTC Vive where the player had to keep track of its virtual location in the game as well as it's physical location in the real world to solve many different types of puzzles but as I was trying out some things with the Vive, I quickly found out that the inability of simply looking away to escape what you see is impossible making virtual reality the perfect candidate for a claustrophobic experience.
After a few days, I finally came up with my final design. A virtual reality horror game where the player and the ghost are both dependent on echolocation to achieve their goals. The player will constantly be balancing his echolocation abilities to be able to see just enough to navigate but use it scarcely enough not to attract the ghost. Of course, this meant abandoning the ability for visually impaired individuals to play the game but instead took on a bit of a teaching role for sighted people to relate.

Chapter 4

Game design

4.1 The plot?

The plot of the game is fairly straightforward yet original. You are left in the dark inside a house haunted by a mysterious ghost-like entity. Your goal is to navigate undetected and locate an ever moving light switch. As a video game enthusiast and a fan of the horror genre, I noticed that when playing a horror game set in the dark, the one thing you are really wishing for rather than trying to find 10 pages of a book or where the demonic baby's screams are coming from, is to figure out a way to turn the lights back on. Therefore when my supervisor offered "finding the light switch" to be the game's objective, I immediately approved and got on with it. I can finally give players a goal that resembles the one they would actually have if they were to be put in the same conditions.

The ghost-like entity does not appear to have a name. No-one knows who or what it is. However, if you happen to look carefully, you may find some clues as to who it may be.

4.2 Aesthetics

Blinded in the dark, only being able to see a limited area for a few seconds at a time is particularly fitting for the horror genre. It resembles a concept I particularly enjoy that has the player stuck with a broken camera but a conveniently functioning flash. Because most of the game will be set in the dark using uncommon lighting, I initially did not want the game to contain too many colors or complex textures. However, like will be explained in the evaluation section, the aesthetics changed quite drastically. The game had a rather abstract look with very simple textures and borders and was using shaders to both increase GPU optimization and giving a special feel to the game. I then changed to a adopt a more realistic look.

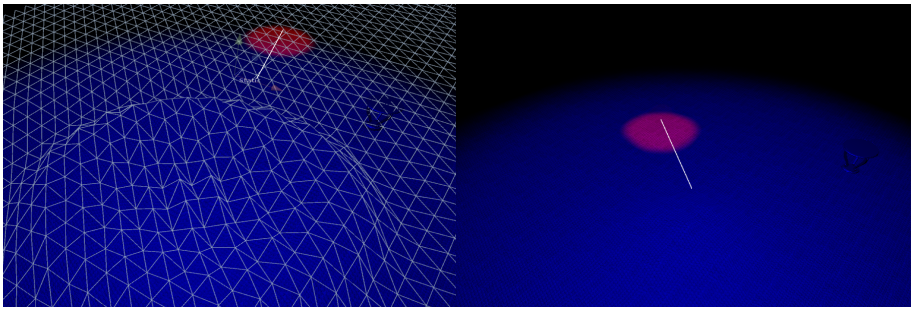
(a) Pre-evaluation version of the textures

(b) Post-evaluation version of the textures

Figure 4.1: Change in aesthetics between the pre and post-evaluation versions of the game

4.3 Sounds

Because the microphone plays a major role in this game, and because the information gathered from it is used in almost all other scripts, I decided to make dedicated singleton class for it. This means only one instance of this class will exist through the entirety game. This class is very special, I gave it a very special name: "Audio Manager". My music background allows me to create all of the sound effects and soundtracks for the game, leaving me in total control over the way sounds will be perceived in the game.



(a) Mesh distortion visible

(b) In game result

Figure 4.2: Screen shot of my game "Listen"

The entity, when moving, makes a subtle hovering sound to allow the player to locates its approximate location in-game. Some small noises will be produced randomly around the house as well to add a bit of depth to the feeling of oppression and claustrophobia as you play. I decided to opt for a music-less experience during the gameplay sections in order to allow the players to fully immersive themselves.

Chapter 5

Game implementation

5.1 Audio

As sound plays such a big role in this game; many aspects of it rely on data from the microphone. Complex audio analysis can quickly become CPU intensive if poorly written. This is why I decided to have a single class controlling all audio aspects of the game so I would have all the audio analysis calculations be done in one place and only once per loop. The AudioManager class is organized in a way that divides it into two sections. The first one is responsible for converting audio buffers into relevant data and the second extricates this data, scales, and filters it in meaningful values for other classes to use. Because I want these analyses to be as precise as they can be in real time while keeping the game playable I deliberately chose to not use the built-in "Update" and "FixedUpdate" but rather have all the relevant methods running on a different thread where I could have them run more frequently and consistently regardless of frame rates while also giving the player a way to adjust their parameters to ensure a smooth experience regardless of their computers (Given they are VR ready machines of course). You can do this in Unity by invoking the following method once at the start of a program:

$InvokeRepeating("BufferPopulator", 0, 1.0f/audioUpdateRate)$;

Of which the first parameter must be the name of the method to be called on another thread, the second parameter being when to start it and the last parameter being the frequency at which this method should call the specified function (The variable *audioUpdateRate* allows the user to control this frequency in Hertz rather than seconds). "BufferPopulator" contains:

- $micSource.clip.GetData(sampData, 0)$; (1)
- $AudioListener.GetSpectrumData(specData, 0, FFTWindow.Hamming)$; (2)

The first item is responsible for populating a pre-allocated buffer meant to contain sample amplitudes. In this case, the buffer contains 64 values, it is the lowest power of 2 this function will take. Because I am using this data to calculate the average amplitude of the microphone in real time, I want the lowest value possible so as to reduce latency to a minimum. Unfortunately, the actual latency will depend greatly on the quality of the computer's sound card. Since this is

a virtual reality game supposedly running on a VR ready computer, however, it is safe to assume that this dependency will never be a problem.

Unfortunately, in order to understand what the second function does, we must first understand what an FFT is.

The Fast Fourier Transform algorithm:

The Fast Fourier Transform algorithm or (FFT) is an algorithm used in many scientific applications and in our case is designed to extract pitch information from an audio clip in real-time as efficiently as possible. It is commonly described as a way to transform audio data from time domain to frequency domain. The algorithm goes as followed:

$$c_k = \frac{1}{N} \sum_{j=0}^{j=N-1} x_j \omega^{jk}. \tag{5.1}$$

While the algorithm may look like witchcraft to most, audio engineers included, it allows us to extract the energy of each frequency in a very fast and efficient way so long as one rule is respected: Because of the way the neperien logarithm works, the buffer must always be a power of two [1]. This means that you may only "cut" the audio buffer into 2^n bits [2]. We call these "bits" bins. However, the sample rate may contain more or fewer frequencies (Usually 44100 or 48000). This means that each "bin" actually covers a range of frequencies. In my case, I chose to have a spectrum array size of 1024 values. Unity's default sample rate is 44100. So if I divide my sample rate (44100) by the number of bins (1024), I will be able to have the spectrum sample precision (In this case ≈ 43). This represents the size of the interval between each bin. I realize this can be confusing so let me provide an example. Let us envisage that I wanted to check the loudness of a particular frequency, let us say 100 Hertz, I would divide this desired frequency by the number of bins. The result, In this case, 2.32, would give me the values to look for in my buffer array. However, you will notice that the result is a float value. Thus in order to get hold of the loudness level at a 100Hz, I will have to select the array index (so an integer value) below 2.32, above 2.32 or in my case both. I could also use a bigger buffer size at the cost of latency.

Back to our function:

$AudioListener.GetSpectrumData(specData, 0, FFTWindow.Hamming)$;

Which contains the following objects:

- AudioListener (1)
- GetSpectrumData (2)
- specData (3)
- "0" (4)
- FFTWindow.Hamming (5)

[1] Having a power of 2 allows us to exploit a symmetry in the regular DFT (Discrete Fourier Transform) analysis that cancels a component that would otherwise take a long a time to process

[2] In Unity, "n" must be a value of 8 or above. This is why the minimum size of the buffer is 64

Let us go through them,

1 The AudioListener variables holds the current AudioListener present inside the Unity scene

2 This method requires 3 parameters and is responsible for the FFT analysis

3 This variable has to be a preallocated array variable that will tell the number of bins Unity should split the sample rate into. Which means that in my case, this variable is a float array containing 1024 values

4 This simply tells the GetSpectrumData method which audio channel it should grab the data from

5 This particular variable tells the method which spectrum analysis windowing type to use (out of possible 7). It is used to reduce leakage of signals across bins.

The last item can be confusing, but it basically applies an envelope (In this case a Hamming envelope) on each bin to try and prevent the analysis from taking into account identical frequencies. It makes the result more accurate and usable while reducing some load on the CPU. Granted it makes the analysis go through another calculus but, in most cases, applying this kind of envelope requires little to no effort on behalf of the CPU while drastically improving results. The hamming envelope I am currently using is following:
$0.54 - (0.46 * cos(n/N))$

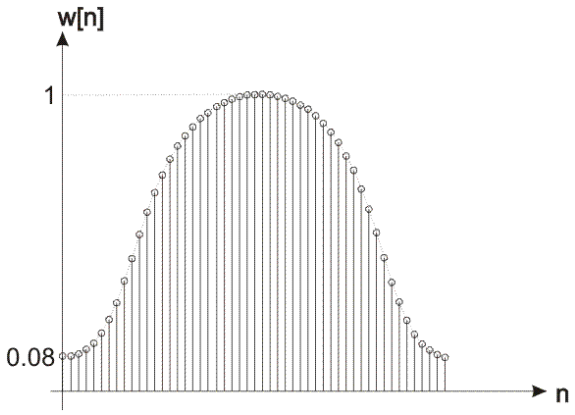


Figure 5.1: Example of the Hamming envelope I am using. 'n' represents the frequencies and 'W[n]' the "loudness"

If, however, I wished to ignore leakage in favor of a small performance boost, I could have used a rectangular envelope which effectively adds and removes nothing (multiplies the output data by 1) but satisfies the method's number of parameters.

Now that we have all the data we needed from the microphone input, we can get to work.

5.1.1 Amplitude analysis

Amplitude analysis is rather simple. Thanks to Unity's "GetData" method, we are able to get the volume of each sample in the provided buffer. To calculate the average volume of the microphone, I must sum the volume of each sample over the number of samples in my array. Unity's sample rate being 44100 samples per seconds, I tried multiple buffers size to get the optimal precision to latency ratio.

Let $\mathcal{B}$ represent the buffer of size N and $\mathcal{A}$ the average volume
for all samples S in $\mathcal{B}$ **do**
  $\mathcal{A} = \mathcal{A} + S_{energy}$
end for
return $\frac{\mathcal{A}}{N}$

A bigger buffer means a longer time period which in terms means a higher latency. Because all of the calculations have to be done for use in real-time, I cannot afford to have a higher latency than 200_{ms} which is why I chose a buffer size of 1024 values. I believed it was the best precision to latency ratio. A buffer size of 64 values (or 2^8, the lowest value Unity will take) did not reduce the latency by a noticeable amount but was a bit too reactive for my taste. I did not want every single sound to be picked up, it felt too chaotic. Buffer sizes above 1024 introduced yet another problem. Because I am averaging the values of each sample, fast, discreet sounds would not be picked up. Different buffer sizes mean different reaction times. Let us say you were in a quiet room and you suddenly decided to snap your fingers. Now let us say we chose a buffers size of 8. we would have the following the following array:

$$[0, 0.25, 0.5, 1, 0.5, 0.25, 0, 0] \tag{5.2}$$

The average volume would be:

$$\frac{(0 + 0.25 + 0.5 + 1 + 0.5 + 0.25 + 0 + 0)}{8} \approx 0.3125 \tag{5.3}$$

The effect would definitely be noticeable. However with a buffer size of 64, for the same clap, the average volume would be ≈ 0.04 which would render it imperceptible.
Now, these are model values and are here to help understand the problems that arise when it comes to real-time data analysis and buffer size to latency/accuracy ratios.

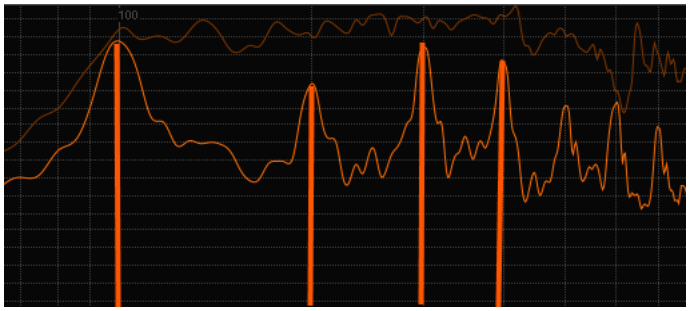
A similar drawback comes up with real-time pitch detection.

5.1.2 Pitch-analysis

Pitch analysis is much trickier to get a hold of than amplitude analysis. But like we have seen in the FFT section above, we managed to be able to get hold of the overall energy of a given frequency or small range of frequency. Now unfortunately for me, as humans, we are far from making perfect tones when we speak. Each individual has a distinct timbre and way to express pitch so asking the player to produce a certain frequency is obviously out of the question. I thus decided to separate

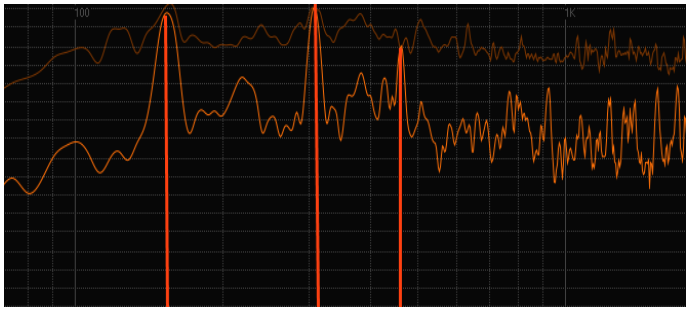
the pitch detection process into two methods instead. A hard coded version and one the player can manual calibrate at startup.

For the first, I decided to gather some data from students at Goldsmiths University. With a microphone attached to an FFT spectrum analyzer, I asked each of them to formulate what they perceived to be a low, medium and a high pitch sound.
For the purpose of clarity I will call these three pitch separations: frequency bands
The first observation, one that I expected but greatly underestimated, was the huge gap between people's interpretations (Especially between genders). Due to the logarithmic nature of sound perception, the gaps for each of the three frequency bands were also exponential. I observed, over a sample size of twenty students, an average gap of 60Hz for the low pitch test, 200Hz for the medium pitch test and up to 500Hz for the high pitch one.



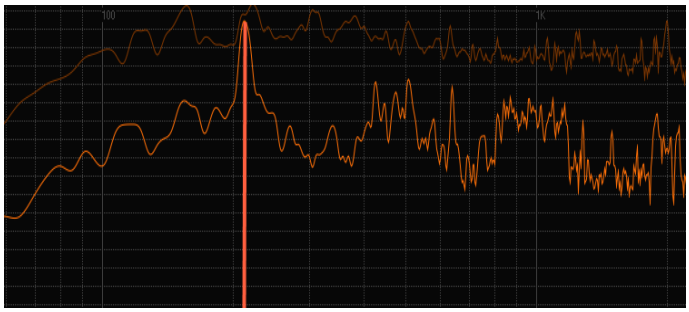
(a) Example of one of the observed frequency spectrum of a volunteer producing a low pitch sound

We can notice peaks at 100Hz, 200Hz, 300Hz, onwards (Harmonics)



(b) Example of one of the observed frequency spectrum of a volunteer producing a medium pitch sound

Producing a pitch created a much more complex frequency spectrum, but we can still easily pick up peaks at 160Hz, 310Hz, and 460Hz



(c) Example of one of the observed frequency spectrum of a volunteer producing a high pitch sound

The frequency spectrum here is even harder to filter out, but we can still notice one outstanding peak around 210Hz

These observations rose three crucial issues:

1 The range of the frequency bands I was comparing had to be significantly increased

2 Some of the harmonics overlap the carrier frequencies of higher pitches

3 The now updated medium pitch range slightly overlapped the two other frequency bands

It became very apparent that hard-coding these values was far from ideal. If I wanted the game to work and react in the same way for everyone, I had to come with another solution.

The first solution I came up with was to have three configurable sliders, one for each of the three frequency bands, and adjust them for each player before playing the game. The value each slider determined the first value to look for in the spectrum array and check for a certain number of values following that first value. The number of values to add to the average on top of the first one increases as select a higher frequency band.

This happened to be a great solution during the building process of my game but became very unfriendly and awkward to use during actual gameplay.

The third and final solution I came up with was to ask for the player in-game to produce what they perceived as low, medium and high pitch and asked them to press a button while making the sound. Once the button is pressed, it calibrates the game's frequency bands to these new values. This fitted well in the tutorial phase present at the beginning of the game.

5.1.3 Scaling and filtering data

The second section of this Audio Manager class is responsible for the scaling and filtering of the analyzed audio data. It contains all the necessary methods to be called by other scripts to affect the game:

GetRMS()

This method converts the average sample amplitude into the "Root-mean-square" value or RMS. The RMS is often used in digital audio processing for converting the average amplitude data into more readable and user-friendly values. The conversion is quite simple and goes as followed:

$$RMS = \sqrt{\frac{Sum\ of\ all\ the\ samples}{Sample\ size}} * 100 \tag{5.4}$$

GetVolume()

This method converts the RMS value to a value in decibel. It is not used in the game but happened to be useful for debugging purposes. Once again the conversion is very simple and goes as followed:

While $dB \leq 0$

$$dB = 20 * \log \frac{RMS}{0.1} \tag{5.5}$$

GetAveragePitch()

This method is in charge of comparing the pitch values the player is producing with the three frequency bands. It works by storing the calibrated vectors corresponding to each frequency band and comparing them with the currently provided vector. I do this by dividing the two and check if the result is close to 1 (How close I want it to be is determined by the confidence value) in which case a certain boolean would be set to true.

For example, if I wanted to check if the player was making a "low pitch sound" I would make the following comparison:

$$ComparingVector[low, mid, high] = \frac{CurrentPitch[low, mid, high]}{CalibratedLow[low, mid, high]} \tag{5.6}$$

---

if $ComparingVector \approx 1$ **then**
  $CurrentPitch = Low$
end if

What I particularly like about this functions is the ability to change the confidence level of the control vector to whichever threshold I desire. The combination of manual calibration and adjustable confidence level allows me to quickly set the game up to be as easy or hard as I want it to be.

GetSpotAngle()

This method checks the RMS level and set the angle of the spotlight accordingly. It does with a smooth transition by adding and subtracting some number (essentially the animation speed) to different maximum values rather than setting them directly.

The amount that is added increases as the RMS goes up to allow quick, loud noise to still have an instantaneous visual feedback.

GetSplashForce()

This method is responsible for setting the force of the generated ripple. Essentially how hard do you splash the floor. It does that simply by taking the RMS and multiplying it to the desired scalar value.

GetLightColor()

This one is slightly more tricky and has the role of making color transitions as smooth as possible. The color of the light can be modified by two different values. It can either depend on the volume or on the pitch. Although both of these methods use linear interpolation, one is slightly more simple to calculate than the other. Let us say you have selected to use volume to control the how red the color becomes. All I have to do is set how much red I want depending on the RMS value and set the blue color to be the inverse of this value as followed:

$$red = \left| \frac{RMS - minimum\ color\ range}{maximum\ color\ range} \right| \quad \& \quad blue = |1 - red| \tag{5.7}$$

$$Light\ color = (red, 0, blue, 0.8) \tag{5.8}$$

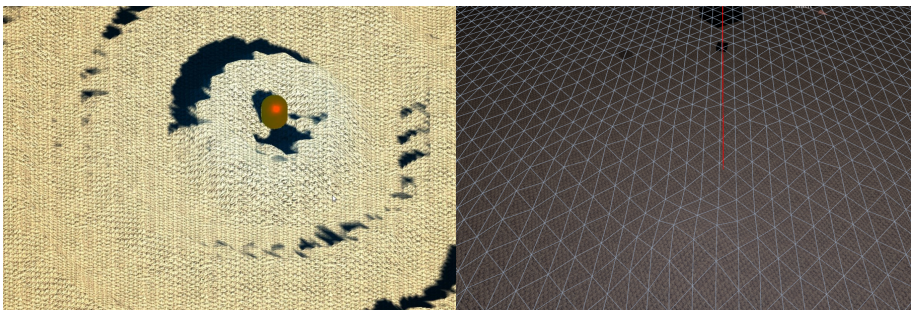
And there you have it, low volume = light blue, high volume = bright red.

Now if you choose to have it based on pitch, it is pretty much the same story only the RMS value is replaced by the average high pitch value divided by a scalar.

5.2 Visual effects

5.2.1 The Ripple Effect

Unity 3D comes with a lot of standard assets and primitive shapes out of the box. All of these have been optimized for GPU performance and thus only contain what most users will need. To make a plane act like water. I need to be able to distort the plane. For optimization purposes, the built in Unity plane renderer only has a couple of vertices which does not allow for much movement apart from basic folding. Now the more vertices you add to the plane, the more distortable it will become but the more computationally intensive it will be. I decided to have a 64x64 plane containing exactly 16641 vertices for my game so as not to ask too much to the GPU while keeping a somewhat realistic movement.



(a) Ripple effect from above under light (b) Ripple effect in game conditions

Figure 5.2: In-game ripple effect examples

Now that we have a plane with enough vertices to simulate a wave we need to animate them. Thankfully, Unity's API contains neat functions to allow the manipulation of individual vertices. The main algorithm behind the ripple effect consists of a nested for loop responsible for the rise of a leading central vertex to its maximum height and the neighboring vertices (behind and after the main one) to a gradually lower height. A second for-loop (inspired by an open source project made by Ben Britten [24] to simulate water) then moves the "main vertex" by using the same array process but shifting the result by a certain number of bits. To keep this effect under control, I had to implement a few modifiable variables. The following are listed below:

- Splash force (1)
- Dampener (2)
- Maximum wave force (3)
- Maximum wave height (4)
- Bit shift (5)
- Update timer (6)

Let me explain how each of them is affecting the ripple:

1 Essentially how hard do you pluck the plane. values between 50 and 1000 make more sense even though you could go as high as 60k.

2 Controls how quickly the wave should fade out or in other words how far it should go before dying (Drastically reduces GPU load). Values below 0.5 will make the effect last only a few milliseconds whereas value above 0.9999 can make the effect last up to 10 seconds or more.

3 Prevents the user from splashing the plane too hard and create uncontrollable ripples.

4 Sets the threshold at which the wave cannot rise more. A value above 3 units can make the wave turn more into a tsunami.

5 Controls how spread the effect should be. A value of 1 here is ideal. A value of 2 can make for a smaller effect, but a value of 5 or more will make the effect hard to notice.

6 Because the ripple is both generated and controlled by the **continuous** microphone output, I had to monitor the rate at which the ripple will be updated to reduce the GPU load. A value of 5 will generate a ripple every 5 frames. The lower the value, the cheaper for the GPU but, the higher the latency.

Your voice and the sounds you produce now feel like they have an impact in the virtual world. But the game is set in the dark. What is the point of adding a feature you cannot see?

5.2.2 The player's light

The ripple is an effect that both looks great, feels great and tries to replicate the effects of echolocation by converting sound into a physically visible sound wave but before it becomes actually useful in-game, I had to add some kind of lighting in order to make the effect complete.
My first attempt at this was to add a spotlight at the center of the player and adjust the range of the spotlight according to the strength of the ripples you generate. But something felt wrong when I noticed that it allowed you see every single thing around you, including the ceiling, whereas the ripple only affected the floor. I also noticed that it was very hard to sync the range between the ripple effect and the distance at which the light allowed you to see. If the player screamed for example, because the light had to be located at the center of the player, the floor would become very bright in order to increase the range of the light. I quickly realized that I had to come up with another type of light. One that could allow me to accurately follow the range of the ripple without necessarily affecting the brightness at the center of the player and luckily Unity has such a light (Queue holy music...): the spotlight.
So I instead opted for a spotlight placed above the player facing downwards. Like other types of lights in Unity, the spotlight has three main configurable parameters:

- The light's color (1)
- The light's angle (2)
- The light's intensity (3)

1 This first item can be both controlled by pitch **or** amplitude. The higher the pitch or the amplitude, the redder the color will get

2 I effectively use this item to control the range at which the player can see. It starts at the center of the player and grows following the ripple. This gives me great control over how far the player can see

3 This last item is very much dependent on the color used and the height at which the light is placed so I am keeping it simple and have it at around 80% constantly

I now have a ripple effect physically changing the aspect of the game according to the sounds you produce. So I wondered, what else could I introduce into the game to make it more interactive and fun using sound?

5.2.3 Dynamic object rendering

I wanted to implement pitch detection into the game in order to control the color of the light, but I kept thinking that there had to be more I could use it for. One of the properties of sounds is that it can be absorbed and reflected in different ways depending on their frequency. I thought I could explore this interesting characteristic in my game by making certain objects only show up when the player produces a certain pitch. I slightly underestimated the adversity of this implementation when checking the raw inputted frequency, but this issue was solved by the implementation of the "GetComplexPitch" function.

I have two different scripts on each object I want to be rendered depending on the pitch, one check if a particular frequency band matches the one I want the object to react upon and the other is in charge of making it fade in and out of the scene. Essentially, the first script checks that the energy of the desired frequency band is high enough to trigger the fade in animation and if not triggers the fade out animation.

---

if $Low\ frequency\ energy \geq Threshold$ **then**
  $Fade\ in$ = `True`
  $Fade\ out$ = `False`
else
  $Fade\ out$ = `True`
  $Fade\ in$ = `False`
end if

At the start, I was simply activating or deactivating the mesh renderer of the object but of course, having object pop in and out of the scene looked a bit unpolished. This is where the second script comes in: Although modified to suit my needs, has mainly been written by Hayden Scott-Baron from starfruitgames [25]. His code basically gradually changes the transparency component of a game object by directly affecting the alpha value of its color. This means that I had to convert each texture in my game to make them able to become transparent. I then added two fade in and fade out functions callable outside of the scope of the class and added a few lines of code to allow me to tweak certain aspects of the effect such as fading speed, color and the ability to change the transparency state halfway through an ongoing animation.

5.3 The Ghost

The ghost's artificial intelligence combines three elements that work together to create life-like behaviors.

1 Ray-casting

2 State Machine

3 Path Finding

Let us see what each of them are responsible for:

5.3.1 Ray-casting

According to Roth, Scott D, "Ray casting is the use of rayâĂŞsurface intersection tests to solve a variety of problems in computer graphics and computational geometry [26]". It is an extremely simple principle that allowed us to build many complex things from rasterization to volume visualization. In short, it consists of casting a virtual ray connecting two things in a virtual space. An application for it would be casting a ray from the barrel of a gun to determine where to apply an impact texture on a wall. In my case, however, I am using ray-casting to solve two problems in one:

1 Collision detection

2 Distance detection

To deal with the first problem, I continuously cast a ray between the player's head and the ghost, I can check whether the ray is being intercepted by an object and effectively cancel the way from this point on. Essentially gifting the ghost with the sense of vision, allowing him to stay aware of the player's presence. This allows the player to hide behind furniture to avoid detection.

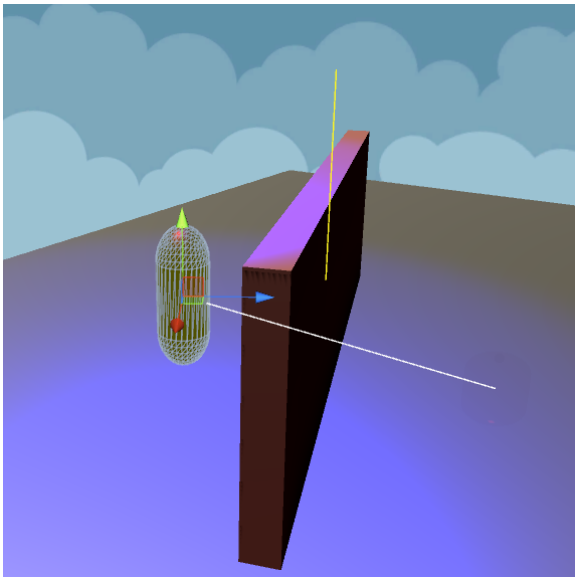


Figure 5.3: Example of an intercepted ray

To deal with the second problem, I configured the ray to be roughly equal to the size of the player's vision. This parameter varied a lot during testing, going from insanely difficult to ridiculously easy. In the end, 0.8 times the player's vision ended up being the best value for it.

5.3.2 State machine

Any agent fitted with artificial intelligence must have some sort of state machine built inside it. This is pretty much how any intelligent species works only more complex. A state machine is quite simply a wide set of states an agent can be in. They are usually triggered by events such as "Sees the player" or "Sees an obstacle" and can result in consequent behaviors like "Follow player", "Attack player" or "Search for player". My agent is no different and complies with similar rules.

The ghost has 3 movement states:

- Patrol
- Chase player
- Remain idle

And 3 Action states:

- Look for player
- Harm the player
- Kill player

When the ghost spawns, his default state is set to "Patrol" which makes him go around the house with no particular goal in mind, unaware of your presence. The ghost selects a destination at random, navigates to it and sometimes remains idle for a while before heading for the next random destination.
If however, the ghost happens to see you, it will go in a chasing state rushing towards you for as long as you are visible (making enough noise).
If during the chase, the player manages to remain silent while moving to a different location, then the ghost will remember the last position the player was seen at and rush towards it in the hopes of finding you there (This is the "look for player" state). Now, quite unsurprisingly, if the player does not manage to escape the ghost in time and the ghost gets too close to the player, The ghost will decide to either harm the player and respawn, leaving the player a second chance at finding the switch or straight up kill the player.

5.3.3 Navmesh agent

All of these states revolve around being able to move around. The ghost manages to do that thanks to the use of A* path-finding combined with navmeshes and navmesh obstacles. By setting the ghost to be a navmesh agent and placing it over a navmesh map, I am able to give the ghost directions, destinations, speed and angular momentum. The way A* works is quite complex and does not fit with the topic of this report, but in short, it is an algorithm that takes multiple parameters into account to find the fastest, most efficient way to head to a goal while avoiding obstacles. A* needs to have a certain amount of "nodes" to go through. These nodes are provided by the navmesh map which separates the floor into smaller sections and stores their center into a list. In Unity, they look like this:

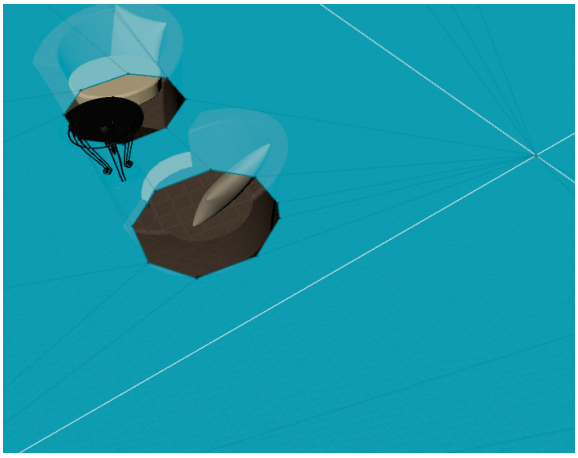


Figure 5.4: Example of a navmesh map

To make efficient use of this method, I decided to opt for a "waypoint" system. I scattered some waypoints (essential "dots" in space with a defined position) around the house. Each waypoint is set to be a possible destination for the ghost's patrol state. This allows me to control where the ghost is allowed to go and make sure it does not go somewhere non-threatening for the player. It

allows me to dynamically add waypoints to its waypoint list making the ghost essentially able to learn from the player's ways. For example, when the ghost goes into "chasing" state and misses the player, a waypoint is added at the last location the player was seen and will be revisited later on. This method, combined with randomizing the light switch's location makes each game feel uniquely challenging.

Chapter 6

Evaluation

Now that I had a fully playable game, I had to put it to the test. Because my game is meant to be played using the HTC Vive, I had to come up with an appropriate time and place to set up the playtest space. Being one of the committee members at the Goldsmiths tech society "Hacksmiths", I had a few opportunities to both have access to a lot of potential testers and the necessary hardware and space during one of the many Hackathons (Anvil Hack III) we run every year.

6.1 The procedure

The play testing procedure is divided in goes as followed:

1 Consent form

2 Game session

3 Survey

First, as the participants came in the room, I explained to them the purpose of my study and what they were about to experience. After answering any question they may have had, I asked them to fill in the carefully crafted consent form making sure of two things:
That they are formerly aware of what they are about to experience as well as protecting me from any potential legal consent problems.

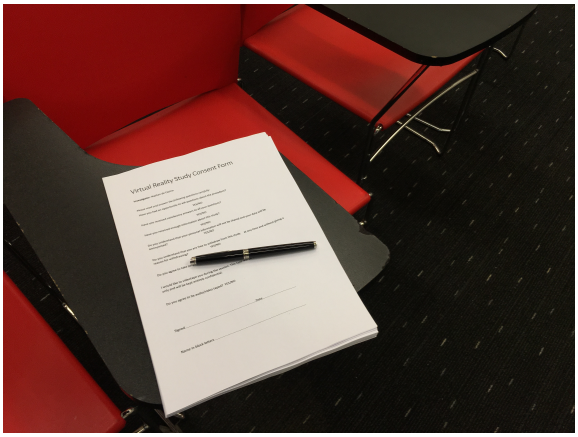


Figure 6.1: Consent form before play testing

The consent form now filled up, the moved to the virtual reality "play area" where I would help them put on the headset and give them the controllers.

Figure 6.2: Play area

My game contains a tutorial phase before the player is sent into the actual house which allows the player to either listen to the recorded audio tutorials or listen to me in order to understand how the game works and have a bit of fun before jumping in. This allowed people that are new to virtual reality to make sense of how to look and move around as well as understand how their voice affected the game. After a minute or two inside this safe environment, they were asked to proceed to the next area by flicking a switch at the other end of the space. Although once inside the next area they have a whole minute without any actual thread, the thought of a presence while being in the dark is enough to set the mood instantly.

I wrote a script that records every session and stores some information on how the player performed. Data like how long did it take them to find the switch, die, complete the tutorial, etc.

This information combined with the survey gives me great insight on what went well and what went wrong for every game session and tweaked or fix some of my code.

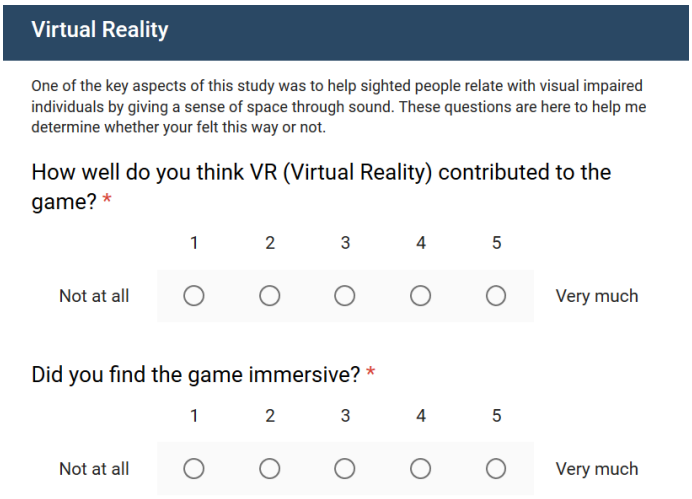


Figure 6.3: Screen shot of the survey

I ran two evaluation sessions at two different stages of my game. I, unsurprisingly, called them Beta and Release versions. The point of running evaluations at immature stages is that it allows me to actually use this feedback to improve the gaming experience. I am able to amend or adjust some aspects of the game immediately and confirm with participants that the changes I have made were the one they were looking for whereas once released, these "adjustments" become "bugs" and are much harder to forgive. We observe this kind of procedure in the gaming world all the time. Players are able to participate (usually for free or otherwise in the promises of getting extra items at the game's release) in some pre-launch beta testing phase where players become avid critics and report problems in the game but remain considerate with lower expectations.

6.2 Results and actions taken

Now that we have seen the way my game has been evaluated we can start to talk about the results and how to I got around fixing what needed fixing and tweaking what needed tweaking.
Immediately, the first thing I noticed as soon as I told the participant they could grab objects, was that I had to change the input I used to do so on the controllers from the "grip" buttons to the "trigger" button. This is typically the kind of issues one can find when going from a purely theoretical approach (coding) to a practical one (actually playing the game). While I was writing my code it felt natural to assign a grabbing gesture to the closest equivalent on the controllers, it being the "grip" buttons (Please refer to input n°1 bellow). However, once you start grabbing the controllers to test the game, your brain is not entirely fooled, and all the gaming experience most candidates had from their past told them that such a button does not exist and that the nearest equivalent had to be the trigger button (Please refer to input n°2 bellow). Of course, as soon as I told them about

the grip button, everyone quickly got used to it, but I did not see the point in the majority of players to change their habits for just one game and thus decided to go on and switch the input method.

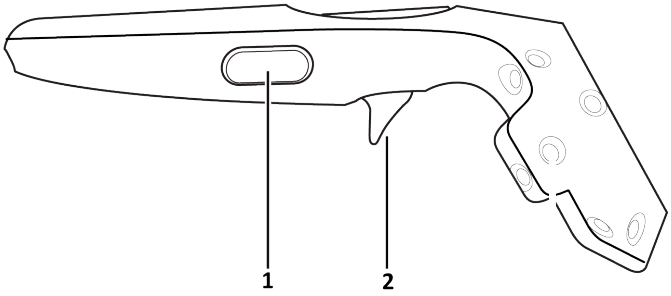


Figure 6.4: HTC vive controller

The second, less expected discovery, was the color of the light feeling too dull. Only being able to distinguish blue and red seemed to have "bored" players. Not only that but they also got confused between the light they were producing and the one from the Ghost. Although I can concede the latter as being a mistake, the first comment was a design decision I took because at the time I thought that blue for low and red for high felt natural (Probably due to its similarity with the light spectrum and wavelength frequencies). I eventually opted for a clearer "dark white" light approach with some tints of yellow as the pitch goes up. Another quick adjustment was to reduce the fading speed of the light to allow players to see the area for longer before making more noise.

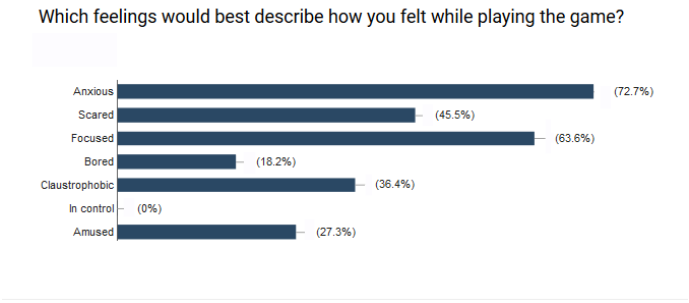


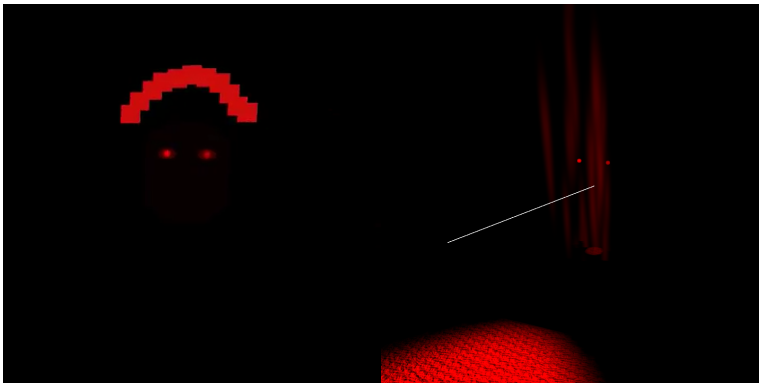
Figure 6.5: One of results of the first evaluation batch

During the first evaluation session, One of the complaints I received after spending a few minutes inside my game was the motion sickness that hit some participants. Many people have spent many hours trying to come up with more natural ways to navigate in virtual reality, and so did I. One thing is for sure, it is remarkably difficult. A few potential techniques rose from the mountain of clunky methods but at the end of the day, having a button to move where towards your gaze while

"fake walking" turned out to be the best effectiveness to complexity ratio (granted no one is around you to see you play...).

One of the key elements to make a virtual reality immersive is to build a credible virtual environment. When I first started thinking about what kind of aesthetics I wanted to have, I rapidly agreed to go for a rough simple texturing for I figured it was fitting with the idea of incarnating a blind person. As you are now aware, the concept of mimicking a blind person faded away, but my vision for the game's artistic taste remained. This asynchronous aspect was picked up during testing, and I quickly realized I had to update my textures for more realistic ones.

Another aspect of the game that went through many alterations was the appearance of the ghost. The original design was extremely minimal to fit with the what the game's initial raw looks. The ghost was made of two floating eyes and a crown-like "hat" meant only to appear when chasing the player. The crown was simply there to notify the player that they have been seen. The results from the evaluations were quite clear. The ghost had to have a more menacing aspect and could use for some of those "fancy" special effects. I removed the body, removed the outline of the eyes and the crown. Instead, we now have a smoke particle effect, two small marble shaped eyes and a big spotlight pointing towards you when it sees the player.



(a) Pre-evaluation version of the ghost(b) Post-evaluation version of the ghost

Figure 6.6: Different versions of the ghost's appearance

Yet, another problem I noticed was even though players have a tutorial phase to get used to the game before heading to the haunted house, the player would still get killed extremely fast straight from the beginning of the game. I thus decided to change two things. First is that the small room you spawn in when the game starts is now unapproachable by the ghost leaving the player the time to adjust and move to the main room before being hunted. The second is the implementation of a threaded function that will make the ghost appear and disappear from the scene every 30 seconds (unless the player is currently being chased). This Allows for 30 seconds of peace at the start of the game (Which turns out to be the time more than 90% of players took to exit the small room) and also creates a more threatening feeling in game since the player can see what the ghost looks

like and what to expect before leaving moments of "where is it?" from time to time. This simple change drastically improved players enjoyment on the second evaluation.

This is just a few of the many improvements that had to be made following the evaluations' results. Running these tests was a particularly insightful part of the building process and a definite game changer (pun intended).

Chapter 7

Conclusion

7.0.1 General review

My main objective was to create an immersive virtual reality experience where the player could explore an innovative way to navigate using nothing but the sound they make. I came across many complications along the way that made me take radical changes to the game. However, in the end, I believe my primary goals have been reached:

Implemented a game concept never seen before:

Indeed although we have seen that a few games have taken sound as their core gaming mechanics, none of them have used complex microphone input data (other than volume). During the process of building this game, I quickly came to realize why. It is an incredibly "clunky" feature to implement, but I took on the challenge, and I fought well enough to make an enjoyable experience.

Implemented complex real-time audio analysis tools using only the Unity's API in C-Sharp:

One of the first research I have done at the beginning of this project was how to grab real-time microphone input directly on Unity without the help of external libraries, and the results were clear: "Do not do it.". But people were not saying it was impossible, they simply said it was going to be very hard because Unity's audio API was ancient, surprisingly unhelpful and the documentation was all deprecated (which it still is to this day). But all that was needed to make the magic happen was one function (GetSpectrumData) and even thought it had not been updated since three patch generations it was still very much there. Working with almost nothing and zero documentation (not counting fellow sound engineers in forums) forced me to work twice as hard and come up with low-level solutions that significantly improved my knowledge of FFTs and other general audio analysis.

Created a compelling virtual reality experience inside a credible 3D environment:

Contrary to audio, the virtual reality tools that were made for developers were both up to date and built to make our life simple (thanks to Steam and their SteamVR plug-in). Setting the scene to be able to be played in VR was therefore very easy. I had to write a few scripts to allow movement,

the ability to pick up objects and of course a few tweaks here and there for the experience to feel smooth but overall, it was relatively painless.
When it comes to the house design, however, I am no architect. And although I have built the house itself, most of the furniture have been downloaded and imported onto the scene using 3DS MAX. It being a tool I had never used before, I had to quickly learn how to merge meshes together to improve in game performance, remove some components from items and add UV lightmaps onto them to make them all compatible with Unity. In the end, I believe I a coped relatively fine with my unartistic side and managed to come up with a believable result where the player can immerse themselves into.

Created a compelling artificial intelligence:

AI was one of the aspects of the game where I felt I would be somewhat comfortable dealing with thanks to the experience I gathered building artificially intelligent agents during the year prior to this project. Except this time, I decided to use Unity's built-in navmesh API rather than making my own A* pathfinding algorithm from scratch again. Although the pathfinding part was thus made relatively simple by Unity, the way the ghost behaved outside of simply finding the next destination to go to was all built on top of nothing and taught me a lot about ray casting and state machines. I tried very hard to create a ghost smart enough to be feared but dull enough to render the game playable.

Explored the idea of using sound as a medium to navigate in space:

Now this being the core attribute of my game, it also was by far the hardest aspect to implement correctly. Adding pitch detection was an extremely laborious and frustrating feature to carry out. Many many lines of code have been written and deleted and many hours of spectrum analysis have been put into making a tool that works. I am aware that most of this work could have been avoided by simply downloading a pre-made tool on the asset store but where is the fun in that? Also because this was such an important element of the game, I needed to grasp every aspect of it if I wanted to tweak and adjust components to suit my vision. Of course, in the end, all that work was not for nothing, and although Unity made it extremely hard for me reduce input latency and put "real" in real-time, I still managed to create an experience where the player must explore a novel way to see.

Made players curious, scared and wanting for more:

Every testing session I have witnessed and all the feedback I have received are clear: The game is fun to play, scary (borderline claustrophobic) and the concept is great. If anything, I am proud to say I have a made a game worth playing. As soon as I told the participants what they were about to experience, they were immediately on board with a smile on their faces and came out of it saying "this is cool!".

Learned a great deal in game design, aesthetics and audio analysis:

Implementing all of these features taught me more than I would have hoped so. I have acquired knowledge in nearly every field of game design.

7.0.2 Reflections on less positive notes

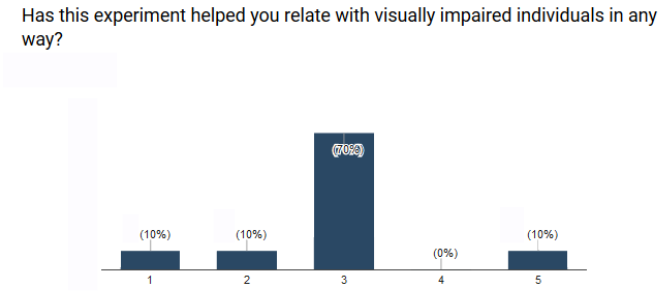


Figure 7.1: Feedback on the relating aspect of the game

Although the results obtained from my evaluations mostly very positive regarding gameplay, the aspect of empathy was definitely not omnipresent. In my opinion, this unfortunate outcome could be explained by:

- Poor description of my intentions pre-evaluation
- The lack of obvious reference to the concept in game
- The format of the experiment may be too short

For me, although the educational aspect of this game very much interested me, the will to create a **fun** game outweighed the will to create an **educational** one. Furthermore, I believe a game with that emphasis **gameplay** before education max contains less potential knowledge but will compensate by the number of people it has an impact on.

7.0.3 The challenges I have yet to face

Although this-this project must comply with the deadline, my work is not done yet. I still have a few elements I did not have time to implement and I think would be substantial before officially call it a day. Let us go through some them:

Movement:

I am keen to try and find more immersive ways to navigate in virtual reality. The touch pad input method did make a few participants feel dizzy. I have been looking into implementing one method in particular that consists of making the player simulate a walking motion with their hands. Although this may seem silly at first the act of physically moving a member to move around and physically stop it to stop help the brain cope immensely. This is why combining the touch pad method with fake walking remains very effective only this time the player would have no choice but simulate.
Another movement related adjustment I may add is an artificial sound in the game when the player is moving to prevent the player from moving around too much when close to the ghost. At the beginning of this project, I thought I could have microphone inputs that would be precise enough

to pick up walking in real life. Unfortunately, Unity does not allow this level of precision without having multiple receivers. Latency would also be an issue here.

Story:

Although I only received one complaint concerning the story of the game, I very much agree with it. The game's story is not shown at any point during the game. We have an objective and obstacles that prevent the player from reach it, but I believe the game would gain a lot of points in the immersion department if the players could incarnate a more interesting fictional character. Maybe through finding audio clips around the house.

More effective audio recognition:

Although I took on the challenge of using Unity's audio API only, I know I can improve the game's audio recognition immensely simply by adding more tools to my tool belt. Using a third party software to accommodate all the sounds or use one of other libraries would allow me to reduce latency and increase precision. I have done the best I could given the tools I had, but they are not the only tools out there.

More interactions:

"Throw an object to attract the ghost at a specific location." This is one of the things I simply did not have time to implement correctly. The code is there, but I had some trouble pinpointing the position of the fallen object because of the odd combination of a floor for collision and one to render the ripples. This is definitely something I will be adding very soon.
I thought of having different light switches trigger different gameplay changing events such as different ghosts, different shaders or simply ask the player to turn multiple switches on before turning on the light.

7.0.4 Final words

Overall, I am extremely satisfied with the project I ended up with. Not only was it thrilling to build such a unique project knowing from the get go that I will be facing many obstacles on my own due to the lack of documentation on the subject. I am also delighted to look back and witness the amount of knowledge I have acquired over these couple of months and cannot wait to now use it to improve the current project as well as create bigger ones in the future.

Appendix A

The Scripts

A.1 AI

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class AI : MonoBehaviour {
    //GAMEOBJECT RELATED VARIABLE
    private GameObject target; //In this case the player
    protected bool played_alert = false; //Makes sure the alert sound
        isn't looping
    public GameObject left_eye; //The exclamation point
    public GameObject right_eye;
    static public Vector3 last_player_T; //Remembers the player's
        location at the moment of death for the death cam.
    public Light flashlight;
    public bool VR = true;
    public GameObject head;

    //MOVEMENT RELATED VRIABLES
    public float patrol_speed = 2; //speed at which the agents will
        travel from waypoint to waypoint
    public float chase_speed = 5; //speed at which the agents chase
        the player
    public float proximity = 2; //distance at which the agent will
        kill the player
    static public bool enemies_converge = false; //IF true, all agents
        will converge towards the player
    private float delta = 0;
    public GameObject waypoints; //All waypoints
    protected Transform[] wayPos; //Array to store each waypoint's
        transform
```

```
private int dest_point = 0; //Init the destination
private bool reset_seen = false;
public bool reset_mode = true;

//RAYCAST RELATED VARIABLE
private float ray_magnitude = 0; //Essentially  the detection
    distance (length of the ray)
public float ghost_awarness = 1.0f;
private bool sees_the_player = false; //true if the agent sees the
    player
protected UnityEngine.AI.NavMeshAgent agent; //Sets the ghosts as
    navmeshagents allowing the use of nevmesh related functions

//Audio
private AudioSource hover;
private AudioSource alert;
private AudioSource death;

// Use this for initialization
void Start () {
    if (VR)
        target = head;
    else
    if (target == null) target = GameObject.FindGameObjectWithTag
        ("head");

    agent = GetComponent<UnityEngine.AI.NavMeshAgent> ();
    wayPos = new
        Transform[waypoints.GetComponentsInChildren<Transform>
        ().Length];
    for (int i = 1; i <
        waypoints.GetComponentsInChildren<Transform> ().Length;
        i++) //i is set to 1 because GetComponentsInChildren also
        takes the parent once. We don't want that
    {
        wayPos[i] = waypoints.GetComponentsInChildren<Transform>
            ()[i];
    }

    AudioSource[] samples = GetComponents<AudioSource> (); //Neat
        way to have multiple samples with a single audio source!
    alert = samples[0];
    hover = samples[1];
    death = samples[2];
}
```

```
void FixedUpdate () {
    ray_magnitude = AM.i.GetSpotAngle () * ghost_awarness;
    if (GS.i.lose_state == false && GS.i.win_state == false) //IF
       you die, these won't be called anymore.
    {
        state_manager (); //Manages the AI's different states
        ray_casting (); //Draws the ray
    }
    sound_effects (); //Manages all the sound effects
    image_effects (); //Manages all the visual effects (Next CGI
       going on here)
}

void ray_casting () {
    RaycastHit hit; //Basically the impact point for the ray
    Ray ray = new Ray (transform.position,
       (target.transform.position - transform.position).normalized
       * (ray_magnitude)); //Casts the ray
    Debug.DrawRay (transform.position, (target.transform.position
       - transform.position).normalized * (ray_magnitude));
       //Casts a visible ray for us to debug
    if (Physics.Raycast (ray, out hit, ray_magnitude))
       Debug.DrawRay (hit.transform.position, Vector3.up * 5,
       Color.yellow); //Casts a ray on collission with object

    if (Physics.Raycast (ray, out hit, ray_magnitude) &&
       (hit.collider.gameObject.tag == "Player" ||
       hit.collider.gameObject.tag == "head")) //checks for
       collision with he player
    { //Checks if the ray has collided
        sees_the_player = true;
        enemies_converge = true;
        Debug.DrawRay (transform.position,
           (target.transform.position -
           transform.position).normalized * (ray_magnitude),
           Color.red); //change color if hit
    } else {
        played_alert = false;
        enemies_converge = false;
        sees_the_player = false;
    }

}

void state_manager () {
```

```
    if (sees_the_player || enemies_converge) //if any agent sees a
        live player, all will converge towards it's location
    {
        chase_player_state ();
    } else {
        agent.speed = patrol_speed;
        if (agent.remainingDistance < 2f) //Goes to the next point
            slightly before reaching the destination to give a more
            natural movement
        {
            patrol_state ();
        }
    }
}

void chase_player_state () {
    transform.LookAt (target.transform.position);
    delta = Vector3.Distance (transform.position,
        target.transform.position);
    if (delta > proximity) {
        reset_seen = false;
        agent.speed = patrol_speed * 4;
        agent.destination = head.transform.position;

    } else {
        if (!reset_seen) {
            wayPos[1] = target.transform; //Sets the first
                waypoint to be the last player's location (Adding
                does not seem to work...)
            agent.destination = wayPos[1].transform.position;
            reset_seen = true;
        }
    }

    if (Vector3.Distance (transform.position,
        target.transform.position) < 5) GS.i.lose_state = true;

    if (Vector3.Distance (transform.position,
        target.transform.position) < 5) //If agent is to close:
        play death sound, destroy the player, sets the state to
        "lost"
    {
        bool harm_only = false;
        if (harm_only) { death.Play (); transform.position.Set (8,
            5.6f, 9); return; }
```

```
        last_player_T = target.transform.position;

        death.Play ();
        if (Camera.main.GetComponent<AudioListener> ())
            Camera.main.GetComponent<AudioListener> ().enabled =
                false;
        else
            GameObject.FindObjectOfType<AudioListener> ().enabled
                = false;
        GS.i.death_cam.SetActive (true);
        GS.i.death_cam.GetComponent<AudioListener> ().enabled =
            true;
        GS.i.death_cam.transform.LookAt (AI.last_player_T);
        GS.i.death_cam.transform.Translate (Vector3.right *
            Time.deltaTime);
        target.SetActive (false);
        agent.speed = patrol_speed - 2;

        //Reset mode:
        if (reset_mode) {
            print ("Reseting...");
            transform.position = new Vector3 (8, 5, 9);
            PlayerLight.instance.ResetPosition ();
            target.SetActive (true);
        } else {
            print ("Player died.");
            GS.i.lose_state = true;
        }

    }
}

void patrol_state () {
    // stops if no points have been set up
    if (wayPos.Length == 0) return;

    dest_point = Random.Range (1, wayPos.Length); //Selects a
        random point and travels to it taking the best rout and
        avoiding obstacles
    agent.destination = wayPos[dest_point].position; //Sets the
        new destination
}

void sound_effects () //This is where all the samples are played.
{
    if (sees_the_player) {
```

```
            if (!played_alert) {
                if (!alert.isPlaying) {
                    played_alert = true;
                    alert.Play ();
                }
            }
        }
        if (!hover.isPlaying) {
            hover.Play ();
        }
    }

    void image_effects () {
        if (sees_the_player) {
            flashlight.gameObject.SetActive (true);
            left_eye.SetActive (true);
            right_eye.SetActive (true);

        } else {
            flashlight.gameObject.SetActive (false);
            left_eye.SetActive (false);
            right_eye.SetActive (false);
        }
    }
}
```

A.2 Player

CurrentPlayer

```
ïż£using UnityEngine;

//Another singleton that stores all player relevant information.
public class CurrentPlayer : MonoBehaviour {
    public static string player_name = "unknown";

    public void PlayerName (string name) {
        player_name = name;
    }
}
```

PlayerLight

```
ïżŁusing UnityEngine;

public class PlayerLight : MonoBehaviour {
    public static PlayerLight instance;

    //LIGHT
    public Light spotlight;

    // Use this for initialization
    void Start () {
        instance = this;
        if (spotlight == null) spotlight =
            GetComponentInChildren<Light> ();
    }

    // Update is called once per frame
    void Update () {
        spotlight.spotAngle = AM.i.GetSpotAngle ();
        spotlight.color = AM.i.GetLightColor ();
    }

    public void ResetPosition () {
        transform.position.Set (-9, 3.18f, -25);
    }
}
```

A.3 Effects

BigEnabler

```
using UnityEngine;

// A simple rotating script for the walls of the player area
public class BigEnabler : MonoBehaviour {

    public float volume_threshold = 20.0f;
    public float pitch_threshold = 0.2f;
    public float fade_speed = 0.5f;
    public bool pitch_based = true;

    private bool Activated = false;
    private FadeInAndOut enabler;

    void Start () {
        enabler = GetComponent<FadeInAndOut> ();
    }
    // Update is called once per frame
    void Update () {

        if (!pitch_based) {
            if (AM.i.GetRMS () > volume_threshold && Activated ==
                false) {
                Activated = true;
                enabler.FadeIn (fade_speed);
            } else if (AM.i.GetRMS () < volume_threshold && Activated
                == true) {
                Activated = false;
                enabler.FadeOut (fade_speed);
            }
        }

        if (pitch_based) {
            if (AM.i.GetComplexPitch ()[0] > pitch_threshold &&
                Activated == false) {
                Activated = true;
                enabler.FadeIn (fade_speed);
            } else if (AM.i.GetComplexPitch ()[0] < pitch_threshold &&
                Activated == true) {
                Activated = false;
                enabler.FadeOut (fade_speed);
            }
        }
```

```
        }
    }
```

MediumEnabler

```
using UnityEngine;

// A simple rotating script for the walls of the player area
public class MediumEnabler : MonoBehaviour {

    public float volume_threshold = 10.0f;
    public float pitch_threshold = 0.2f;
    public float fade_speed = 0.5f;
    public bool pitch_based = true;

    private bool Activated = false;
    private FadeInAndOut enabler;

    void Start () {
        enabler = GetComponent<FadeInAndOut> ();
    }
    // Update is called once per frame
    void Update () {

        if (!pitch_based) {
            if (AM.i.GetRMS () > volume_threshold && Activated ==
                false) {
                Activated = true;
                enabler.FadeIn (fade_speed);
            } else if (AM.i.GetRMS () < volume_threshold && Activated
                == true) {
                Activated = false;
                enabler.FadeOut (fade_speed);
            }
        }

        if (pitch_based) {
            if (AM.i.GetComplexPitch ()[1] > pitch_threshold &&
                Activated == false) {
                Activated = true;
                enabler.FadeIn (fade_speed);
            } else if (AM.i.GetComplexPitch ()[1] < pitch_threshold &&
                Activated == true) {
                Activated = false;
                enabler.FadeOut (fade_speed);
            }
        }
    }
}
```

SmallEnabler

```
using UnityEngine;

// A simple rotating script for the walls of the player area
public class SmallEnabler : MonoBehaviour {

    public float volume_threshold = 3.0f;
    public float pitch_threshold = 0.2f;
    public float fade_speed = 0.5f;
    public bool pitch_based = true;

    private bool Activated = false;
    private FadeInAndOut enabler;

    void Start () {
        enabler = GetComponent<FadeInAndOut> ();
    }
    // Update is called once per frame
    void Update () {

        if (!pitch_based) {
            if (AM.i.GetRMS () > volume_threshold && Activated ==
                false) {
                Activated = true;
                enabler.FadeIn (fade_speed);
            } else if (AM.i.GetRMS () < volume_threshold && Activated
                == true) {
                Activated = false;
                enabler.FadeOut (fade_speed);
            }
        }

        if (pitch_based) {
            if (AM.i.GetComplexPitch ()[2] > pitch_threshold &&
                Activated == false) {
                Activated = true;
                enabler.FadeIn (fade_speed);
            } else if (AM.i.GetComplexPitch ()[2] < pitch_threshold &&
                Activated == true) {
                Activated = false;
                enabler.FadeOut (fade_speed);
            }
        }
    }
}
```

Rotator

```
ïż£using UnityEngine;

// A simple rotating script for the walls of the player area
public class Rotator : MonoBehaviour
{
    public float speed = 4;

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(Vector3.up * (Time.deltaTime * speed));
    }
}
```

SoundScaler

```
ïżfusing UnityEngine;

public class SoundScaler : MonoBehaviour {

    // Update is called once per frame
    void Update () {
        this.gameObject.transform.lossyScale.Set (1 +
           AM.i.GetComplexPitch ().x, 1 + AM.i.GetComplexPitch ().y, 1
           + AM.i.GetComplexPitch ().z);
    }
}
```

A.4 Tools

SwitchStart

```
ï»¿using UnityEngine;
using UnityEngine.SceneManagement;

public class SwitchStart : MonoBehaviour {

    public GameObject switch_on;
    public GameObject switch_off;
    public WandController wand;
    private bool controller_near_switch = false;
    private bool player_near_switch = false;

    static public Light glow;

    void Start () {
        glow = GetComponentInChildren<Light> ();
    }

    // Update is called once per frame
    void Update () {
        controller_near_switch = Vector3.Distance
            (wand.transform.position, transform.position) < 1;
        player_near_switch = Vector3.Distance
            (PlayerLight.instance.transform.position,
            transform.position) < 7;

        //Turns on the light when the button is pressed
        if ((wand.trigger_button_pressed && controller_near_switch) ||
            (Input.GetKeyDown (KeyCode.Return) && player_near_switch)) {
             SceneManager.LoadScene ("Game");
             switch_off.SetActive (!switch_off.activeSelf);
             switch_on.SetActive (!switch_off.activeSelf);
        }
    }
}
```

SwitchState

```
ïżŁusing UnityEngine;

public class SwitchState : MonoBehaviour {

    public Light main_light;
    public GameObject switch_on;
    public GameObject switch_off;
    public WandController wand;

    static public bool switch_status = false;

    private bool controller_near_switch = false;
    private bool player_near_switch = false;

    public bool start_with_light = true;

    // Use this for initialization
    void Start () {

        if (start_with_light)
            main_light.enabled = true;
        else
            main_light.enabled = false;
    }

    // Update is called once per frame
    void Update () {
        controller_near_switch = Vector3.Distance
           (wand.transform.position, transform.position) < 1;
        player_near_switch = Vector3.Distance
           (PlayerLight.instance.transform.position,
           transform.position) < 7;

        //Turns on the light when the button is pressed
        if ((Input.GetKeyDown (KeyCode.Return) && player_near_switch)
           || (wand.trigger_button_pressed && controller_near_switch))
           {

            main_light.enabled = switch_off.activeSelf;
            switch_status = switch_off.activeSelf;

            switch_off.SetActive (!switch_off.activeSelf);
            switch_on.SetActive (!switch_off.activeSelf);
        }
```

```
        if (Input.GetKeyDown (KeyCode.L) || wand.sys_button_pressed) {
            main_light.enabled = !main_light.enabled;
        }
        if (GS.i.win_state) { main_light.enabled = true;
            GS.i.win_state = false; }
    }
}
```

AudioManager

```
ïż£using UnityEngine;

//Singleton class for managing audio
public class AM : MonoBehaviour {
    public static AM i;

    private AudioSource mic_source;

    //Globals
    public bool pitch_based = true; //Wether or not to use pitch
        rather then amplitude for calculations
    public float color_range_min = 3; //Sets how low can the light
        color go
    public float color_range_max = 40;
    public float spot_fading_speed = 0.5f; //How fast should the light
        angle fade back to the center
    public float splash_force_scaler = 100; //How powerful can a
        splash be
    public float audio_update_rate = 40.0f; //How fast do you want to
        calculate the audio (Drastically affects FPS on low end
        computers)
    public float pitch_precision = 0.1f; //How precise do you want the
        check to be
    public float current_pitch = 0;
    public float refValue = 0.01f;

    private float[] spec_data; //Will contain the spectrum information
        from the microphone
    private float[] samp_data; //Will contain sample amplitude from
        the microphone

    private int spec_size = 1024; //Size of the spectrum buffer
    private int samp_size = 64; //Size of the sample buffer (this can
        be low since we are just averaging the volume)
    private int buffer_size = 24000; // Half of the actual buffer size
        (from 0 until the Nyquist rate)
    private float rms = 0; //Will store the RMS values
    private float vol = 0; //Will store the volume in dB
    private float spot_angle = 1; //The spot angle at start
    private float average_pitch_low = 0;
    private float average_pitch_mid = 0;
    private float average_pitch_high = 0;
    private float red_value = 0;
```

```
[Range (0, 64)]
public int low_range = 13;
public bool set_low = false;
private Vector3 calibrated_low;

[Range (0, 512)]
public int mid_range = 44;
public bool set_mid = false;
private Vector3 calibrated_mid;

[Range (0, 1024)]
public int high_range = 51;
public bool set_high = false;
private Vector3 calibrated_high;

void Awake () {
    i = this;
    buffer_size = AudioSettings.outputSampleRate / 2;
    spec_data = new float[spec_size];
    samp_data = new float[samp_size];

    //Sets up the audiosource to use the Microphone
    mic_source = GetComponent<AudioSource> ();
    mic_source.clip = Microphone.Start (null, true, 1,
        buffer_size); //Select the Microphone as input
    mic_source.pitch = 0.95f; //slightly slower to avoid stuttering
    mic_source.loop = true; //loops the second of audio
    mic_source.Play ();
    set_low = false;
    set_mid = false;
    set_high = false;

    InvokeRepeating ("BufferPopulator", 0, 1.0f /
        audio_update_rate); //Begins another thread

    InvokeRepeating ("ComplexAmplitudeAnalysis", 0, 1.0f /
        audio_update_rate); //Begins another thread
    //InvokeRepeating ("Ampy", 0, 1.0f / audio_update_rate);
        //Begins another thread

    InvokeRepeating ("ComplexPitchAnalysis", 0, 1.0f /
        audio_update_rate); //Begins another thread
    //InvokeRepeating ("Pitchy", 0, 1.0f / audio_update_rate);
        //Begins another thread
}
```

```
void BufferPopulator () {
    mic_source.clip.GetData (samp_data, 0);
    mic_source.GetSpectrumData (spec_data, 0, FFTWindow.Hanning);
}

//Anaylsis function:
void ComplexAmplitudeAnalysis () {
    float vol_sum = 0;
    for (int i = 0; i < samp_size; i++) {
        vol_sum += Mathf.Abs (samp_data[i] * samp_data[i]);
           //Calculates the square sum of all the data
    }
    rms = Mathf.Sqrt (vol_sum / samp_size) * 100; //RMS value
    vol = 0;
    if (!(vol < 0))
        vol = 20 * Mathf.Log10 (rms / 0.1f); //Amplitude in dB
    else vol = 0;
}

void ComplexPitchAnalysis () {
    //Low - Around 84Hz
    float Low = (spec_data[low_range] + spec_data[low_range + 1] +
       spec_data[low_range + 2] + spec_data[low_range + 3]) / 4;
    if (Low > 0.00001) average_pitch_low = Low * 100;
    else average_pitch_low = 0;

    //Mid - Around 512Hz
    float Mid = (spec_data[mid_range] + spec_data[mid_range + 1] +
       spec_data[mid_range + 2] + spec_data[mid_range + 3] +
       spec_data[mid_range + 4]) / 5;
    if (Mid > 0.00001) average_pitch_mid = Mid * 100;
    else average_pitch_mid = 0;

    //High - Around 1025Hz
    float High = (spec_data[high_range] + spec_data[high_range +
       1] + spec_data[high_range + 2] +
        spec_data[high_range + 3] + spec_data[high_range + 4] +
           spec_data[high_range + 5] + spec_data[high_range + 6] +
           spec_data[high_range + 7]) / 8;
    if (High > 0.00001) average_pitch_high = High * 100;
    else average_pitch_high = 0;

    //Manually calibrate values on the spot:
    if (set_low) { calibrated_low = GetComplexPitch (); set_low =
       false; print ("Low calibrated to: " + calibrated_low); }
    if (set_mid) { calibrated_mid = GetComplexPitch (); set_mid =
```

```
        false; print ("Mid calibrated to: " + calibrated_mid); }
    if (set_high) { calibrated_high = GetComplexPitch (); set_high
        = false; print ("High calibrated to: " + calibrated_high); }
}

//Getter functions:
public float GetRMS () {
    return rms;
}
public float GetVolume () {
    return vol;
}

public string GetAveragePitch () {
    string output = "N/A";
    float safe_zero = 0.0000000001f; //Makes sure I never divide
        by zero without affecting the output

    //Divides each value from the from the calibrated buffers with
        the current buffers to see whether a certain pitch is being
        produced.
    //This part makes the caluclations
    double delta_low_x = GetComplexPitch ().x / (safe_zero +
        calibrated_low.x);
    double delta_low_y = GetComplexPitch ().y / (safe_zero +
        calibrated_low.y);
    double delta_low_z = GetComplexPitch ().z / (safe_zero +
        calibrated_low.z);

    double delta_mid_x = GetComplexPitch ().x / (safe_zero +
        calibrated_low.x);
    double delta_mid_y = GetComplexPitch ().y / (safe_zero +
        calibrated_low.y);
    double delta_mid_z = GetComplexPitch ().z / (safe_zero +
        calibrated_low.z);

    double delta_high_x = GetComplexPitch ().x / (safe_zero +
        calibrated_low.x);
    double delta_high_y = GetComplexPitch ().y / (safe_zero +
        calibrated_low.y);
    double delta_high_z = GetComplexPitch ().z / (safe_zero +
        calibrated_low.z);

    // And compares and concludes
    if (delta_low_x <= 1 + pitch_precision && delta_low_x >= 1 -
        pitch_precision &&
```

```
            delta_low_y <= 1 + pitch_precision && delta_low_y >= 1 -
                pitch_precision &&
            delta_low_z <= 1 + pitch_precision && delta_low_z >= 1 -
                pitch_precision)
            output = "LOW";

        if (delta_mid_x <= 1 + pitch_precision && delta_mid_x >= 1 -
            pitch_precision &&
            delta_mid_y <= 1 + pitch_precision && delta_mid_y >= 1 -
                pitch_precision &&
            delta_mid_z <= 1 + pitch_precision && delta_mid_z >= 1 -
                pitch_precision)
            output = "MEDIUM";

        if (delta_high_x <= 1 + pitch_precision && delta_high_x >= 1 -
            pitch_precision &&
            delta_high_y <= 1 + pitch_precision && delta_high_y >= 1 -
                pitch_precision &&
            delta_high_z <= 1 + pitch_precision && delta_high_z >= 1 -
                pitch_precision)
            output = "HIGH";

        return output;
    }

    public Vector3 GetComplexPitch () {
        return new Vector3 (average_pitch_low, average_pitch_mid,
            average_pitch_high);
    }

    public float GetSpotAngle () {
        if (GetVolume () > 8f && spot_angle < 30) spot_angle += 0.3f;
        if (GetVolume () > 20f && spot_angle < 80) spot_angle += 1;
        if (GetVolume () > 40f && spot_angle < 140) spot_angle += 3;

        if (spot_angle >= 1) spot_angle -= spot_fading_speed;

        return spot_angle;
    }

    public int GetSplashForce () {
        return (int) (rms * splash_force_scaler);
    }

    public Color GetLightColor () {
        //Be aware! Some mad linear interpolation is happening here:
```

```
    if (pitch_based) {
        if (average_pitch_mid > 0.5f && red_value <= 0.5)
            red_value += 0.1f;
        else if (red_value >= 0) red_value -= 0.01f;
        if (average_pitch_high > 0.9f && red_value <= 0.9f)
            red_value += 0.1f;
        else if (red_value >= 0) red_value -= 0.01f;

        float blue_value = Mathf.Abs (1 - red_value);
        Mathf.Clamp (red_value, 0.0f, 0.8f);
        Mathf.Clamp (blue_value, 0.0f, 0.8f);

        //return new Color (red_value, 0, blue_value, 0.8f);
        return new Color (0.6f, 0.6f, 0.7f, 0.5f);
    }
    if (!pitch_based) {
        float r = Mathf.Abs ((rms - color_range_min) /
            color_range_max);
        float b = Mathf.Abs (1 - r);
        return new Color (r, 0, b, 0.8f);
    }
    return new Color (0.6f, 0.6f, 0.7f, 0.5f);
}

void Pitchy () {
    // get sound spectrum
    float maxV = 0;
    var maxN = 0;
    for (int i = 0; i < spec_size; i++) { // find max
        if (!(spec_data[i] > maxV) || !(spec_data[i] > 0.02f))
            continue;

        maxV = spec_data[i];
        maxN = i; // maxN is the index of maxa
        float freqN = maxN; // pass the index to a float variable
        if (maxN > 0 && maxN < spec_size - 1) { // interpolate
            index using neighbours
            var dL = spec_data[maxN - 1] / spec_data[maxN];
            var dR = spec_data[maxN + 1] / spec_data[maxN];
            freqN += 0.5f * (dR * dR - dL * dL);
        }
        current_pitch = freqN * (buffer_size) / spec_size; //
            convert index to frequency
    }
}
```

```
    void Ampy () {
        float sum = 0;

        for (int i = 0; i < samp_size; i++) {
            sum += samp_data[i] * samp_data[i]; // sum squared samples
        }

        rms = Mathf.Sqrt (sum / samp_size); // rms = square root of
            average
        vol = 20 * Mathf.Log10 (rms / refValue); // calculate dB
        if (vol < -160) vol = -160; // clamp it to -160dB min
    }
}
```

FadeInAndOut

```
ïżż/*
        FadeObjectInOut.cs
        Hayden Scott-Baron (Dock) - http://starfruitgames.com
        6 Dec 2012

        This allows you to easily fade an object and its children.
        If an object is already partially faded it will continue from
            there.
        If you choose a different speed, it will use the new speed.

        NOTE: Requires materials with a shader that allows
            transparency through color.
*/

using System.Collections;
using UnityEngine;

public class FadeInAndOut : MonoBehaviour {

    // publically editable speed
    public float fadeDelay = 0.0f;
    public float fadeTime = 0.5f;
    public bool fadeInOnStart = false;
    public bool fadeOutOnStart = false;
    private bool logInitialFadeSequence = false;

    // store colours
    private Color[] colors;

    // allow automatic fading on the start of the scene
    IEnumerator Start () {
        //yield return null;
        yield return new WaitForSeconds (fadeDelay);

        if (fadeInOnStart) {
            logInitialFadeSequence = true;
            FadeIn ();
        }

        if (fadeOutOnStart) {
            FadeOut (fadeTime);
        }
    }
```

```
// check the alpha value of most opaque object
float MaxAlpha () {
    float maxAlpha = 0.0f;
    Renderer[] rendererObjects = GetComponentsInChildren<Renderer>
        ();
    foreach (Renderer item in rendererObjects) {
        maxAlpha = Mathf.Max (maxAlpha, item.material.color.a);
    }
    return maxAlpha;
}

// fade sequence
IEnumerator FadeSequence (float fadingOutTime) {
    // log fading direction, then precalculate fading speed as a
        multiplier
    bool fadingOut = (fadingOutTime < 0.0f);
    float fadingOutSpeed = 1.0f / fadingOutTime;

    // grab all child objects
    Renderer[] rendererObjects = GetComponentsInChildren<Renderer>
        ();
    if (colors == null) {
        //create a cache of colors if necessary
        colors = new Color[rendererObjects.Length];

        // store the original colours for all child objects
        for (int i = 0; i < rendererObjects.Length; i++) {
            colors[i] = rendererObjects[i].material.color;
        }
    }

    // make all objects visible
    for (int i = 0; i < rendererObjects.Length; i++) {
        rendererObjects[i].enabled = true;
    }

    // get current max alpha
    float alphaValue = MaxAlpha ();

    // This is a special case for objects that are set to fade in
        on start.
    // it will treat them as alpha 0, despite them not being so.
    if (logInitialFadeSequence && !fadingOut) {
        alphaValue = 0.0f;
        logInitialFadeSequence = false;
    }
```

```
        // iterate to change alpha value
        while ((alphaValue >= 0.0f && fadingOut) || (alphaValue <=
            1.0f && !fadingOut)) {
            alphaValue += Time.deltaTime * fadingOutSpeed;

            for (int i = 0; i < rendererObjects.Length; i++) {
                Color newColor = (colors != null ? colors[i] :
                    rendererObjects[i].material.color);
                newColor.a = Mathf.Min (newColor.a, alphaValue);
                newColor.a = Mathf.Clamp (newColor.a, 0.0f, 1.0f);
                rendererObjects[i].material.SetColor ("_Color",
                    newColor);
            }

            yield return null;
        }

        // turn objects off after fading out
        if (fadingOut) {
            for (int i = 0; i < rendererObjects.Length; i++) {
                rendererObjects[i].enabled = false;
            }
        }
    }

    public void FadeIn () {
        FadeIn (fadeTime);
    }

    public void FadeOut () {
        FadeOut (fadeTime);
    }

    public void FadeIn (float newFadeTime) {
        StopAllCoroutines ();
        StartCoroutine ("FadeSequence", newFadeTime);
    }

    public void FadeOut (float newFadeTime) {
        StopAllCoroutines ();
        StartCoroutine ("FadeSequence", -newFadeTime);
    }
}
```

FPS_Camera

```
ïżŁusing UnityEngine;

public class FPS_Camera : MonoBehaviour
{
    Vector2 mouseLook;
    Vector2 smoothV;
    public float sensitivity = 5.0f;
    public float smoothing = 2.0f;
    GameObject character;

    void Start()
    {
        character = this.transform.parent.gameObject;
    }
    void Update()
    {
        var md = new Vector2(Input.GetAxisRaw("Mouse X"),
            Input.GetAxisRaw("Mouse Y"));

        md = Vector2.Scale(md, new Vector2(sensitivity * smoothing,
            sensitivity * smoothing));
        smoothV.x = Mathf.Lerp(smoothV.x, md.x, 1f / smoothing);
        smoothV.y = Mathf.Lerp(smoothV.y, md.y, 1f / smoothing);
        mouseLook += smoothV;

        transform.localRotation = Quaternion.AngleAxis(-mouseLook.y,
            Vector3.right);
        character.transform.localRotation =
            Quaternion.AngleAxis(mouseLook.x, character.transform.up);
    }
}
```

FPS_Control

```
ïż£using UnityEngine;

public class FPS_Control : MonoBehaviour
{
    public float speed = 0.5f;
    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;
    }
    void Update()
    {
        float translation = Input.GetAxis("Vertical") * speed;
        float straffe = Input.GetAxis("Horizontal") * speed;
        translation *= Time.deltaTime;
        straffe *= Time.deltaTime;

        transform.Translate(straffe, 0, translation);

        if (Input.GetKeyDown("escape"))
            Cursor.lockState = CursorLockMode.None;
    }
}
```

GhostSound

```
using UnityEngine;

public class GhostSound : MonoBehaviour {

    public Light light_self;
    public Vector3 vector_down;

    public float echo_max_range = 60f;
    public float echo_speed = 2.0f;
    public float light_current_angle = 0.0f;
    public float echo_reset = 0.0f;
    public float echo_fading_speed = 0.2f;

    private bool ripple_forward = true;
    static public bool collided = false;

    void Update () {
        vector_down = this.transform.TransformDirection
           (Vector3.down);
        if (collided) echo ();
    }

    void echo () {
        if (ripple_forward) {
            if (!(light_current_angle >= echo_max_range)) {
                light_current_angle += echo_speed;
            } else {
                ripple_forward = false;
            }
        }
        if (!ripple_forward) {
            if (!(light_current_angle <= echo_reset)) {
                light_current_angle -= echo_speed;
            } else {
                collided = false;
                ripple_forward = true;
            }
        }
        light_self.enabled = true;
        light_self.spotAngle = light_current_angle;
    }

    public Vector3 GetVectorDown () {
        return this.vector_down;
```

```
    }

    void OnCollisionEnter (Collision collision) {
        foreach (ContactPoint contact in collision.contacts) {
            Debug.DrawRay (contact.point, contact.normal,
                Color.white);
        }
        collided = true;
    }

    void OnCollisionExit (Collision collision) {
        collided = false;
    }
}
```

LightScript

```
ïż£using UnityEngine;

public class LightScript : MonoBehaviour
{
    public float echo_max_range = 60f;
    public float echo_speed = 1.0f;
    public float echo_current_range = 1.0f;
    public float echo_current_intensity = 4.0f;
    public float echo_fading_speed = 0.1f;
    private bool ripple_forward = true;

    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetMouseButton(0))
        {
            if (ripple_forward)
            {
                if (!(echo_current_range >= echo_max_range))
                {
                    echo_current_range += echo_speed;
                }
                else
                {
                    ripple_forward = false;
                }
            }
            if (!ripple_forward)
            {
                if (!(echo_current_range <= 1.0f))
                {
                    if (!(echo_current_intensity <= 0))
                       echo_current_intensity -= 0.1f;
                    echo_current_range -= echo_speed;
                }
                else
                {
                    echo_current_intensity = 1.0f;
```

```
                    ripple_forward = true;
                    Destroy(this.gameObject);
                }
            }
            GetComponent<Light>().enabled = true;
            GetComponent<Light>().range = echo_current_range;
            GetComponent<Light>().intensity = echo_current_intensity;
        }
    }
}
```

RandSwitch

```
ïż£using UnityEngine;

public class RandSwitch : MonoBehaviour {

    GameObject[] switches;

    // Use this for initialization
    void Start () {
        switches = GameObject.FindGameObjectsWithTag ("Switch");
            //Finds all the switch
        foreach (GameObject s in switches) { //Sets them all to
            inactive
             s.SetActive (false);
        }

        int rand = new System.Random ().Next (0, switches.Length); //
            creates a number between 0 and the number of switches
            present
        print ("Selected switch " + rand + " over " +
            switches.Length); //Prints out the selected one
        switches[rand].gameObject.SetActive (true); //Sets it to actif
    }
}
```

RippleEffect

```
// An small part of this script has been written by other contributors
   on Stack Overflow. Notably Ben Britten.
using UnityEngine;

public class RippleEffect : MonoBehaviour {

    public int cols; //Number of the columns we want our plane to have
       (The bigger this is, the more computer intensive the
       calculations will be)
    public int rows; //Number of the rows we want our plane to have
    public float dampner; //How fast we want the wave to fade out
    public float threshold = 5.0f;
    public int spread_bit_shift;
    public Light VR_spotlight;
    public Light player_spotlight;
    static public int update_rate = 5;
    public bool VR = false;

    //Because we want  a whole plane to distort, we need the plane
       have more then just 4 vertices to simulate the wave
    //This is why we need to create our own plane with many many
       vertices
    private int[] buffer1; //Is the first buffer of vertices
    private int[] buffer2; //Is the second buffer of vertices
    private int[] vertex_indexes; //Stores the indexes of the vertices
    private int splash_force_mic = 0; //Variable that will update the
       splash_force according to the mic
    private Mesh mesh; //Will store the created mesh
    private Vector3[] vertices; //private Vector3[] normals ;
    private Vector3[] current_vertices;
    private bool swap_buffer = true; //Switches between the first and
       the second buffer

    void Start () {
        MeshFilter mesh_filter = GetComponent<MeshFilter> ();
           //Attaches the mesh filter in he object to the mesh filter
           variable
        mesh = mesh_filter.mesh; //Sets mesh as the mesh of the mesh
           filter
        vertices = mesh.vertices; //Sets the vertices variable to the
           vertices of the the created mesh
        current_vertices = new Vector3[vertices.Length]; // initiates
           a array of vector3 vertices
        buffer1 = new int[vertices.Length]; //Initializes both buffers
```

```
        with the number of vertices in the mesh
    buffer2 = new int[vertices.Length];
    Bounds bounds = mesh.bounds;

    float xStep = (bounds.max.x - bounds.min.x) / cols; //Sets the
        number of tiles to animate on the X axis according to the
        number of columns we selected and the size of the mesh
    float zStep = (bounds.max.z - bounds.min.z) / rows; //Sets the
        number of tiles to animate on the Y axis according to the
        number of columns we selected and the size of the mesh

    vertex_indexes = new int[vertices.Length]; // Create an int
        array and allocates it's size to the number of vertices

    /* ***************** Initializes and Populates then buffers
        (These two for loops have been inspired by the work of Ben
        Britten) ********************  */
    int vertices_index = 0;
    for (vertices_index = 0; vertices_index < vertices.Length;
        vertices_index++) {
        vertex_indexes[vertices_index] = -1;
        buffer1[vertices_index] = 0;
        buffer2[vertices_index] = 0;
    }

    for (vertices_index = 0; vertices_index < vertices.Length;
        vertices_index++) {
        float column = ((vertices[vertices_index].x -
            bounds.min.x) / xStep);
        float row = ((vertices[vertices_index].z - bounds.min.z) /
            zStep);
        float position = (row * (cols + 1)) + column + 0.5f;
        vertex_indexes[(int) position] = vertices_index;
    }
}

void SplashAtPoint (int x, int y) //Function that initiates the
    splash at a given location
{
    // initiates the values of each vertices according to their
        distance from the center such as the further you are from
        it the weak the force is.
    int position = ((y * (cols + 1)) + x);
    buffer1[position] = splash_force_mic;
    buffer1[position - 1] = splash_force_mic;
    buffer1[position + 1] = splash_force_mic;
```

```
    buffer1[position + (cols + 1)] = splash_force_mic;
    buffer1[position + (cols + 1) + 1] = splash_force_mic;
    buffer1[position + (cols + 1) - 1] = splash_force_mic;
    buffer1[position - (cols + 1)] = splash_force_mic;
    buffer1[position - (cols + 1) + 1] = splash_force_mic;
    buffer1[position - (cols + 1) - 1] = splash_force_mic;
}
// Update is called once per frame
void Update () {

    if (5 < update_rate)
        update_rate = (int) (1 / AM.i.GetVolume ()) * 20;
    else update_rate = 5;
    if ((tick (update_rate) && AM.i.GetVolume () >= threshold) ||
       Input.GetKeyDown (KeyCode.Space)) {
        splash_force_mic = AM.i.GetSplashForce ();
        SplashDetect (); //Checks wether a splash has been
           initiated
    }
    int[] current_buffer;
    if (swap_buffer) {
        // process the ripples for this frame
        ripple_process (buffer1, buffer2);
        current_buffer = buffer2;
    } else {
        ripple_process (buffer2, buffer1);
        current_buffer = buffer1;
    }
    swap_buffer = !swap_buffer;
    /*
       ************************************************************************
        */

    // THE CODE THAT RAISES THE WAVE
    int current_index; //Will store the current index of a vertex
    int i = 0;
    for (i = 0; i < current_buffer.Length; i++) //For all the
       vertices in the current buffer
    {
        current_index = vertex_indexes[i];
        current_vertices[current_index] = vertices[current_index];
        current_vertices[current_index].y += (current_buffer[i] *
           1.0f / 100); //This is what raises the current "tile"'s
           Y value by the splash force times the maximum height of
           the wave}
    }
```

```
    GetComponent<MeshFilter> ().mesh = mesh; //Updates it to unity
    mesh.vertices = current_vertices; //Updates the mesh! :)
}
void SplashDetect () {

    /* Ray cast function to set the splash position */
    RaycastHit spotlight_hit; //Will store all the info on the
       object the ray will collide
    Vector3 spotlight_down; //Create a raycast from the player's
       position downwards
    Vector3 spotlight_pos;

    if (VR) {
        spotlight_down = VR_spotlight.transform.forward; //Create
           a raycast from the player's position downwards
        spotlight_pos = VR_spotlight.transform.position;
    } else {
        spotlight_down =
           player_spotlight.transform.TransformDirection
           (Vector3.forward);
        spotlight_pos = player_spotlight.transform.position;
    }
    Debug.DrawRay (spotlight_pos, spotlight_down * 5, Color.red);

    if (Physics.Raycast (spotlight_pos, spotlight_down, out
       spotlight_hit)) // Draws a ray downwards from the player
       and stores the impact location
    {
        if (spotlight_hit.collider !=
           GameObject.FindGameObjectWithTag
           ("floor").GetComponent<Collider> ()) return; //Makes
           sure the ray is only considered if on a ripple plane

        // First we must init the tiles
        Bounds bounds = mesh.bounds;
        float xStep = (bounds.max.x - bounds.min.x) / cols;
        float zStep = (bounds.max.z - bounds.min.z) / rows;

        float xCoord = (bounds.max.x - bounds.min.x) -
           ((bounds.max.x - bounds.min.x) *
           spotlight_hit.textureCoord.x); //Sets the X coordinates
           for the splash
        float zCoord = (bounds.max.z - bounds.min.z) -
           ((bounds.max.z - bounds.min.z) *
           (spotlight_hit.textureCoord.y)); //Sets the Y
```

```
                coordinates for the splash
            float column = (xCoord / xStep);
            float row = (zCoord / zStep);
            SplashAtPoint ((int) column, (int) row); //Generates the
                splash!
        }
    }

    //THE CODE THAT MAKES THE WAVE MOVE AND FADE OUT
    void ripple_process (int[] source, int[] dest) {
        int x = 0;
        int y = 0;
        int position = 0;
        for (y = 1; y < rows - 1; y++) {
            for (x = 1; x < cols; x++) {
                position = (y * (cols + 1)) + x;
                dest[position] = (((source[position - 1] +
                    source[position + 1] + source[position - (cols +
                    1)] + source[position + (cols + 1)]) >>
                    spread_bit_shift) - dest[position]); //Makes the
                    wave move forward
                dest[position] = (int) (dest[position] * dampner);
                    //Makes the wave fade out
            }
        }
    }

    //Custom Timer function to control how fast you want to update the
        splash while using th mic (A value of 5 means every 5 frames)
    int timer = 0;
    public bool tick (int update_rate) {
        timer++;
        if (timer >= update_rate) {
            timer = 0;
            return true;
        }
        return false;
    }
}
```

Timer

```
ïż£using UnityEngine;

public class Timer : MonoBehaviour {
    public int tempo = 0;

    int timer = 0;
    public bool tick () {
        timer++;
        if (timer >= tempo) {
            timer = 0;
            return true;
        }
        return false;
    }
}
```

A.5 UI

GameState

```
ïżŁusing UnityEngine;
using UnityEngine.SceneManagement;

//This is a big singleton class containing all the game settings
   information.
//Most variables are straightfoward

public class GS : MonoBehaviour {
    static public GS i; //Singleton

    private GameObject player;

    public bool stay_on_scene = false;
    public GameObject death_cam;
    public bool debugging = true;
    public GameObject ghost;

    //I don't want these public variables to appear in the inspector
    [HideInInspector]
    public bool win_state = false;
    [HideInInspector]
    public bool lose_state = false;

    private GameObject[] pickups;

    void Awake () {
        i = this;
        InvokeRepeating ("SpawnGhost", 30, 30);
        //if (stay_on_scene)
            UnityEditor.SceneView.FocusWindowIfItsOpen (typeof
            (UnityEditor.SceneView));
        player = GameObject.FindGameObjectWithTag ("Player");
        //death_cam = GameObject.FindGameObjectWithTag ("Death Cam");
    }

    void Update () {
        if (player)
            if (!player.gameObject.activeSelf || lose_state) {
                SceneManager.LoadScene ("Game");
                lose_state = false;
            }
```

```
        if (SwitchState.switch_status && SceneManager.GetActiveScene
            ().name == "Game") {
            SceneManager.LoadScene ("Tutorial");
            win_state = true;
        }

        if (Input.GetKeyDown ("r")) {
            SceneManager.LoadScene (SceneManager.GetActiveScene
                ().name);
        }
    }

    void SpawnGhost () {
        ghost.SetActive (!ghost.activeSelf);
    }
}
```

Info

```
ïżżusing UnityEngine;
using UnityEngine.UI;

public class Info : MonoBehaviour {
    private Text info_text;

    void Start () {
        info_text = GetComponent<Text> ();
        info_text.text = "";
    }

    void Update () {
        info_text.text = "RMS: " + AM.i.GetRMS () + " dB: " +
            AM.i.GetVolume ();
    }
}
```

Instructions

```
ïż£using UnityEngine;
using UnityEngine.UI;

public class Instructions : MonoBehaviour {
    private Text info_text;
    public GameObject player_mode;

    void Start () {
        info_text = GetComponent<Text> ();
        info_text.text = "";
    }

    void Update () {
        if (player_mode.activeSelf) {

                info_text.text = "Don't worry, you are safe here =)" +
                    "\n" + "Frist, try to find the switch and turn on
                    the light." + "\n";

                info_text.text = "That's it! you found it!" + "\n" +
                    "Now press switch in on by moving the controller
                    towards the switch and press the touchpad";

                info_text.text = "The the dark, you are only safe when
                    silent, but you must make noise to see!" + "\n" +
                    "Head to the green switch to start the game." +
                    "\n" + "Good luck!";
        }
    }
}
```

LoadStart

```
ïżżusing UnityEngine.SceneManagement;
using UnityEngine;

public class LoadStart : MonoBehaviour
{
    public void Load_Start()
    {
        SceneManager.LoadScene("Game");
    }
}
```

LoadTutorial

```
ïżżusing UnityEngine.SceneManagement;
using UnityEngine;

public class LoadTuto : MonoBehaviour
{
    public void Load_Tuto()
    {
        SceneManager.LoadScene("Tutorial");
    }
}
```

SessionRecorder

```
using System.IO;
using UnityEngine;

public class SessionRecorder : MonoBehaviour {
    private string state = "stopped";
    private string player_name = "Player";

    void Start () {
        if (CurrentPlayer.player_name != null) player_name =
            CurrentPlayer.player_name;
    }

    void FixedUpdate () {
        if (GS.i.win_state) { state = "won"; File.AppendAllText
            (Application.dataPath + "/Data/SessionData.txt", session
            ()); }
        if (GS.i.lose_state) { state = "lost"; File.AppendAllText
            (Application.dataPath + "/Data/SessionData.txt", session
            ()); }
    }

    void OnApplicationQuit () {
        if (!GS.i.lose_state && !GS.i.win_state) File.AppendAllText
            (Application.dataPath + "/Data/SessionData.txt", session
            ());
    }

    string session () {
        string output = "";
        string session_date = System.DateTime.Now.ToString () + "  |
            ";
        string session_duration = player_name + " " + state + " after:
            " + (int) Time.time + " seconds" +
            System.Environment.NewLine;

        output = session_date + session_duration;
        print ("Game session saved in Data folder...");
        return output;
    }
}
```

Visibility

```
ïżŁusing UnityEngine;
using UnityEngine.UI;

public class Visibility : MonoBehaviour {
    private Text info_text;

    void Start () {
        info_text = GetComponent<Text> ();
        info_text.text = "";
    }

    void Update () {
        info_text.text = "\n" + "Average: " + AM.i.GetAveragePitch ()
            + "/n" + "Complex: " + AM.i.GetComplexPitch ();
    }
}
```

A.6 VR

InteractableItem

```
ïż£using UnityEngine;

public class InteractableItem : MonoBehaviour {
    private Rigidbody rb;

    private bool currentlyInteracting;

    private float velocityFactor = 20000f;
    private Vector3 posDelta;

    private float rotationFactor = 400f;
    private Quaternion rotationDelta;
    private float angle;
    private Vector3 axis;

    private WandController attachedWand;

    private Transform interactionPoint;

        // Use this for initialization
        void Start () {
        rb = GetComponent<Rigidbody>();
        interactionPoint = new GameObject().transform;
        velocityFactor /= rb.mass;
        rotationFactor /= rb.mass;
        }

        // Update is called once per frame
        void Update() {
            if (attachedWand && currentlyInteracting) {
            posDelta = attachedWand.transform.position -
                interactionPoint.position;
            this.rb.velocity = posDelta * velocityFactor *
                Time.fixedDeltaTime;

            rotationDelta = attachedWand.transform.rotation *
                Quaternion.Inverse(interactionPoint.rotation);
            rotationDelta.ToAngleAxis(out angle, out axis);

            if (angle > 180) {
                angle -= 360;
            }
```

```
            this.rb.angularVelocity = (Time.fixedDeltaTime * angle *
                axis) * rotationFactor;
        }
        }

    public void BeginInteraction(WandController wand) {
        attachedWand = wand;
        interactionPoint.position = wand.transform.position;
        interactionPoint.rotation = wand.transform.rotation;
        interactionPoint.SetParent(transform, true);

        currentlyInteracting = true;
    }

    public void EndInteraction(WandController wand) {
        if (wand == attachedWand) {
            attachedWand = null;
            currentlyInteracting = false;
        }
    }

    public bool IsInteracting() {
        return currentlyInteracting;
    }
}
```

LightFollow

```
ïż£using UnityEngine;

public class lightfollow : MonoBehaviour {
        public GameObject player_head;


        // Update is called once per frame
        void Update () {
          Vector3 pos = player_head.transform.position;
          pos.y += 6;
          transform.position = pos;
        }
}
```

PickedUpObject

```
ïżżusing System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PickedUpObject : MonoBehaviour {
    static public GameObject pickup = null;

    // Use this for initialization
    void Start () {

    }

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

    }

    void OnTriggerEnter (Collider coll) {
        if (!coll.gameObject.isStatic &&
            coll.gameObject.GetComponent<Rigidbody> () != null) {
            pickup = coll.gameObject; print ("Picked up!"); }
    }

    void OnTriggerExit (Collider coll) {
        pickup = null;
    }
}
```

Tutorial

```
ïż£using UnityEngine;

public class Tutorial : MonoBehaviour {
    private AudioSource source;

    public WandController wand;
    public AudioClip intro;
    public AudioClip instructions;
    public AudioClip goal_1;
    public AudioClip goal_2;
    public AudioClip goal_3;
    public AudioClip end;

    private int tutorial_index = 0;
    private bool reset = true;

    // private bool intro_played = false;
    // private bool instructions_played = false;
    // private bool goal_1_played = false;
    // private bool goal_2_played = false;
    // private bool goal_3_played = false;
    // private bool end_played = false;

    // Use this for initialization
    void Start () {
        source = GetComponent<AudioSource> ();
        source.clip = intro;
        source.volume = 0.7f;
    }

    // Update is called once per frame
    void Update () {
        if (wand.sys_button_pressed || Input.GetKeyDown (KeyCode.T)) {
           tutorial_index++; tutorial_index = tutorial_index % 7;
           source.Stop (); }

        switch (tutorial_index) {
            case 1:
                if (!source.isPlaying) {
                    reset = true;
                    source.clip = intro;
                    source.PlayOneShot (intro);
                    print ("Playing clip: " + source.clip.name);
                }
```

```
                break;
            case 2:
                if (!source.isPlaying) {
                    source.clip = instructions;
                    source.PlayOneShot (instructions);
                    print ("Playing clip: " + source.clip.name);
                }
                break;
            case 3:
                if (!source.isPlaying) {
                    source.clip = goal_1;
                    source.PlayOneShot (goal_1);
                    print ("Playing clip: " + source.clip.name);
                }
                break;
            case 4:
                if (!source.isPlaying) {
                    source.clip = goal_2;
                    source.PlayOneShot (goal_2);
                    print ("Playing clip: " + source.clip.name);
                }
                break;
            case 5:
                if (!source.isPlaying) {
                    source.clip = goal_3;
                    source.PlayOneShot (goal_3);
                    SwitchStart.glow.enabled = true;
                    print ("Playing clip: " + source.clip.name);
                }
                break;
            case 6:
                if (!source.isPlaying) {
                    source.clip = end;
                    source.PlayOneShot (end);
                    SwitchStart.glow.enabled = true;
                    print ("Playing clip: " + source.clip.name);
                }
                break;
            default:
                if (reset) { print ("Beginning of clips"); reset =
                    false; }
                break;
        }
        SwitchStart.glow.enabled = false;
    }
}
```

VRMove

```
ïżŁusing UnityEngine;

public class VRMove : MonoBehaviour {

    public WandController wand;
    public GameObject head;

    private Vector3 forward;
    private Vector3 backward;
    private Vector3 left;
    private Vector3 right;

    private float max_speed = 0.06f;
    public bool simple = true;
    private float speed = 0;

    void Start () {
        forward = new Vector3 (0, 0, 0);
        backward = new Vector3 (0, 0, 0);
        left = new Vector3 (0, 0, 0);
        right = new Vector3 (0, 0, 0);
    }
    // Update is called once per frame
    void Update () {
        if (simple) {
            if (wand.pad_button_pressed) {
                if (speed <= max_speed) speed += 0.001f;
            } else {
                if (speed >= 0) speed -= 0.0005f;
            }
            Vector3 translate_vector = new Vector3
               (head.transform.forward.x, 0, head.transform.forward.z);
            transform.Translate (translate_vector * speed,
               Space.World);

        } else {
            if (wand.pad_button_pressed) {
                if (wand.pad_up > 0) {
                    forward.x = head.transform.forward.x * wand.pad_up
                       / 10;
                    forward.y = head.transform.forward.z * wand.pad_up
                       / 10;
                    transform.Translate (forward, Space.World);
                }
```

```
                if (wand.pad_down > 0) {
                    backward.x = -head.transform.forward.x *
                        wand.pad_down / 10;
                    backward.y = -head.transform.forward.z *
                        wand.pad_down / 10;
                    transform.Translate (backward, Space.World);
                }
                if (wand.pad_left > 0) {
                    left.x = head.transform.forward.x * wand.pad_left
                        / 10;
                    transform.Translate (left, Space.World);
                }
                if (wand.pad_right > 0) {
                    right.x = -head.transform.forward.x *
                        wand.pad_right / 10;
                    transform.Translate (right, Space.World);
                }
            }
        }
    }
}
```

WandButtons

```
ïż£using UnityEngine;

public class WandButtons : MonoBehaviour {

    private SteamVR_TrackedController controller_event;

    // Use this for initialization
    void Start () {
        controller_event = GetComponent<SteamVR_TrackedController> ();
    }

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

        if (controller_event.gripped) {
            SteamVR_Controller.Input ((int)
               controller_event.controllerIndex).TriggerHapticPulse
               (100);
        }
    }
}
```

WandController

```
ïżŁusing UnityEngine;

public class WandController : MonoBehaviour {
    private Valve.VR.EVRButtonId grip_button =
        Valve.VR.EVRButtonId.k_EButton_Grip;
    private Valve.VR.EVRButtonId trigger_button =
        Valve.VR.EVRButtonId.k_EButton_SteamVR_Trigger;
    private Valve.VR.EVRButtonId pad_button =
        Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad;
    private Valve.VR.EVRButtonId sys_button =
        Valve.VR.EVRButtonId.k_EButton_ApplicationMenu;

    private SteamVR_Controller.Device device { get { return
        SteamVR_Controller.Input ((int) tracked_object.index); } }
    private SteamVR_TrackedObject tracked_object;
    private SteamVR_TrackedObject controller;

    private GameObject pickup;
    public bool pad_button_pressed = false;
    public bool sys_button_pressed = false;
    public bool trigger_button_pressed = false;

    public float pad_up = 0;
    public float pad_down = 0;
    public float pad_left = 0;
    public float pad_right = 0;

    // Use this for initialization
    void Start () {
        tracked_object = GetComponent<SteamVR_TrackedObject> ();
    }

    // Update is called once per frame
    void Update () {
        if (device == null) {
            Debug.Log ("Controller not initialized");
            return;
        }

        pickup = PickedUpObject.pickup;

        pad_button_pressed = device.GetPress (pad_button);
        sys_button_pressed = device.GetPressDown (sys_button);
        trigger_button_pressed = device.GetPressDown (trigger_button);
```

```
        if (device.GetPress (grip_button) && pickup != null) {
            pickup.transform.parent = this.transform;
            pickup.GetComponent<Rigidbody> ().isKinematic = true;
        } else if (pickup != null && !device.GetPress (grip_button)) {
            pickup.GetComponent<Rigidbody> ().isKinematic = false;
            pickup.transform.parent = null;
        } else {
            return;
        }

        if (device.GetAxis ().x != 0 || device.GetAxis ().y != 0) {
            if (device.GetAxis ().y > 0) pad_up = device.GetAxis ().y;
            else pad_up = 0;
            if (device.GetAxis ().y < 0) pad_down = device.GetAxis
               ().y;
            else pad_down = 0;
            if (device.GetAxis ().x > 0) pad_right = device.GetAxis
               ().x;
            else pad_left = 0;
            if (device.GetAxis ().x < 0) pad_left = device.GetAxis
               ().x;
            else pad_right = 0;
        }
    }
}
```

Appendix B

Documents

B.1 Gitlab: Unity project

Gitlab: *http://ndeca001@gitlab.doc.gold.ac.uk/ndeca001/Listen.git*

B.2 Github: Release version

Github: *https://github.com/Yukisando/Listen—Builds*

B.3 Gitlab: Report

Gitlab: *http://ndeca001@gitlab.doc.gold.ac.uk/ndeca001/Listen-Project-Report.git*

B.4 The Consent Form

Virtual Reality Study Consent Form

Investigator: Nathan de Castro

Please read and answer the following questions carefully:

Have you had an opportunity to ask questions about the procedure?

YES/NO

Have you received satisfactory answers to all your questions?

YES/NO

Have you received enough information about this study?

YES/NO

Do you understand that your personal information will not be shared and your data will be anonymised? YES/NO

Do you understand that you are free to withdraw from this study at any time and without giving a reason for withdrawing? YES/NO

Do you agree to take part in this study? YES/NO

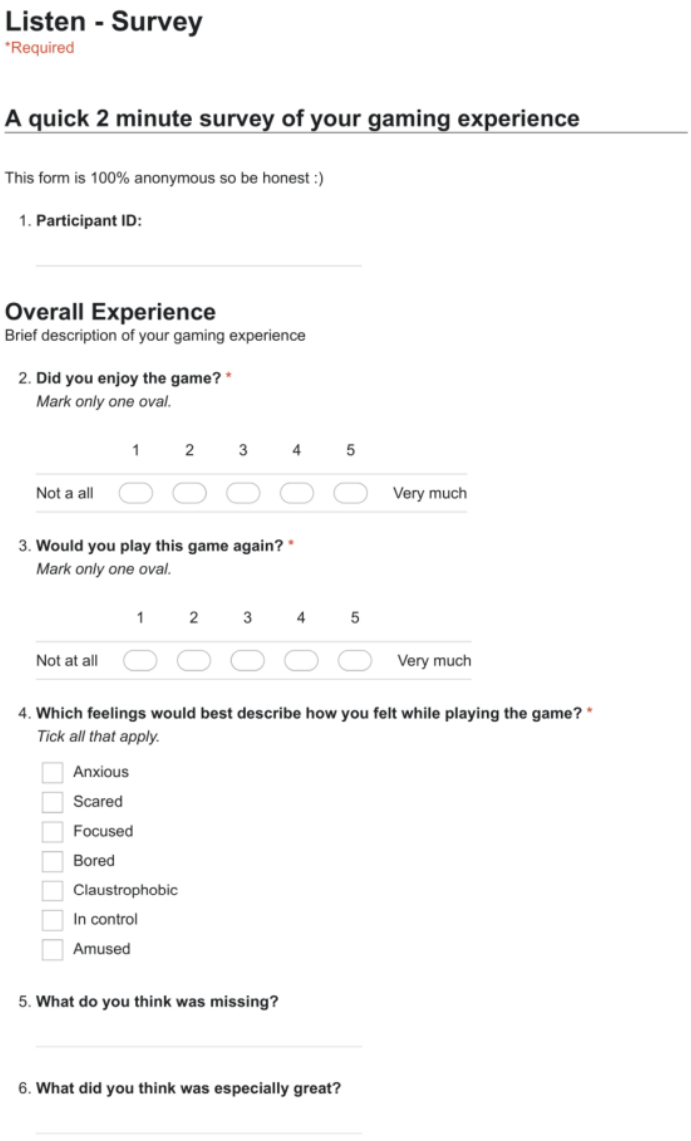
I would like to videotape you during the session. This tape will be used for data analysis purposes only and will be kept entirely confidential.

Do you agree to be audio/video taped? YES/NO

Signed……...……Date……

Name in block letters ……...……

B.5 The Survey



Listen - Survey

*Required

A quick 2 minute survey of your gaming experience

This form is 100% anonymous so be honest :)

1. **Participant ID:**

Overall Experience

Brief description of your gaming experience

2. **Did you enjoy the game?** *
   Mark only one oval.

	1	2	3	4	5	
Not a all	◯	◯	◯	◯	◯	Very much

3. **Would you play this game again?** *
   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very much

4. **Which feelings would best describe how you felt while playing the game?** *
   Tick all that apply.

- ☐ Anxious
- ☐ Scared
- ☐ Focused
- ☐ Bored
- ☐ Claustrophobic
- ☐ In control
- ☐ Amused

5. **What do you think was missing?**

6. **What did you think was especially great?**

Figure B.1: Survey page n°1 of 2

7. **Have you ever played a game with similar incentives?** *
Mark only one oval.

- ◯ Yes
- ◯ No

8. **If yes, which one ?**

Virtual Reality

One of the key aspects of this study was to help sighted people relate with visually impaired individuals by giving a sense of space through sound. These questions are here to help me determine whether your felt this way or not.

9. **How well do you think VR (Virtual Reality) contributed to the game?** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very much

10. **Did you find the game immersive?** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very much

11. **Has this experiment helped you relate with visually impaired individuals in any way?** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very much

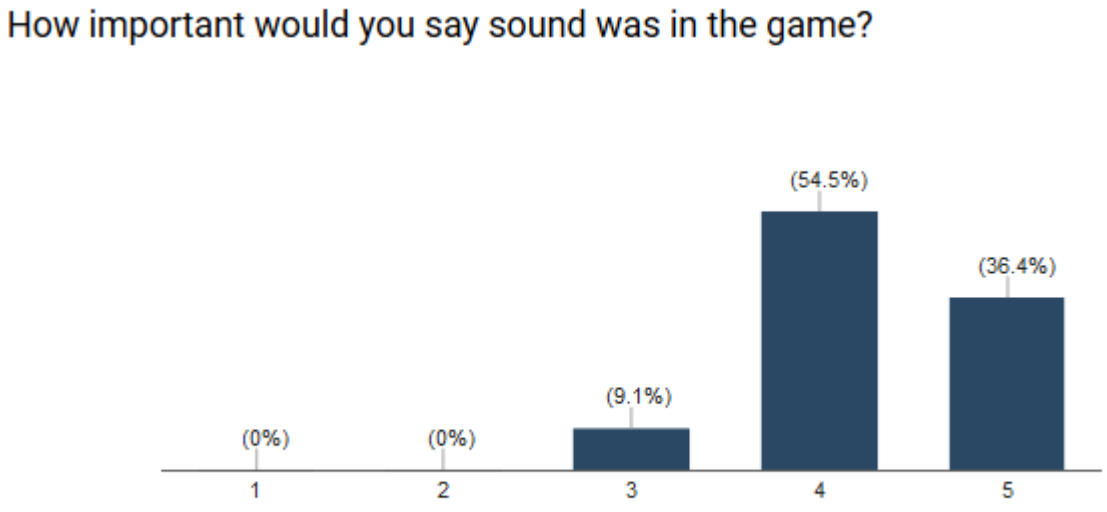
12. **How important would you say sound was in the game?** *
Mark only one oval.

	1	2	3	4	5	
Useless	◯	◯	◯	◯	◯	Very Important

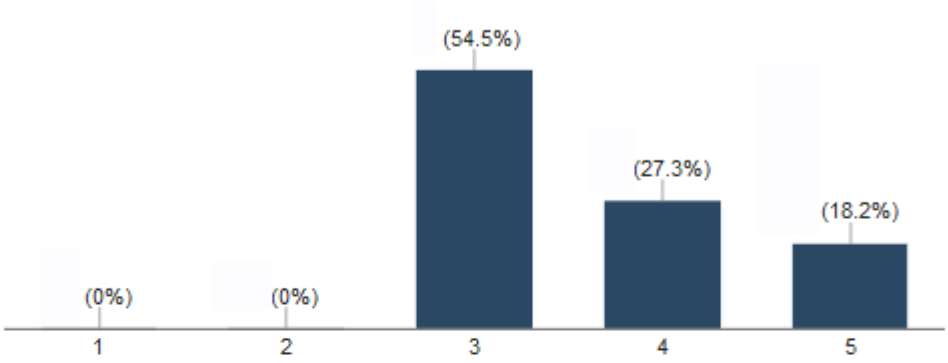
13. **Overall feedback?**

Figure B.2: Survey page n°2 of 2

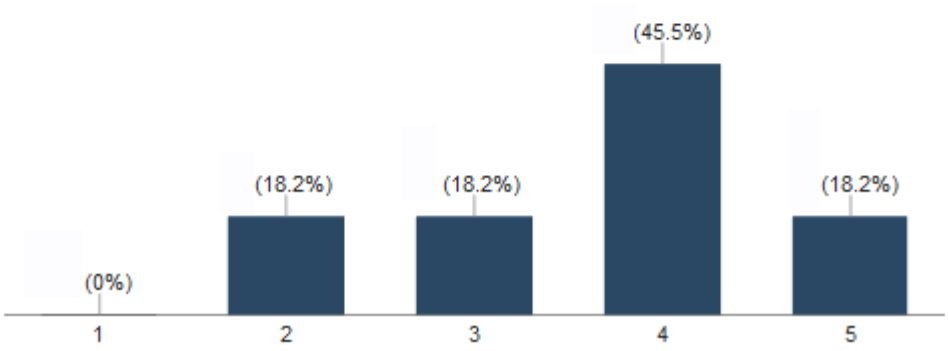
B.6 Non-textual answers



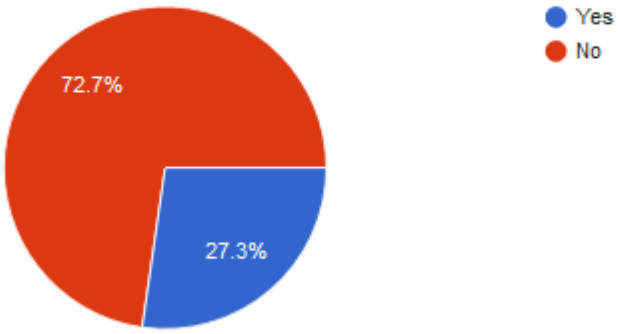
Did you enjoy the game?



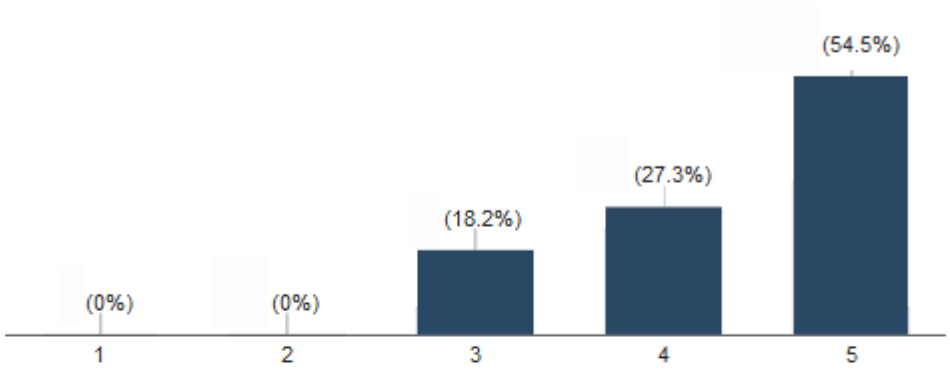
Would you play this game again?



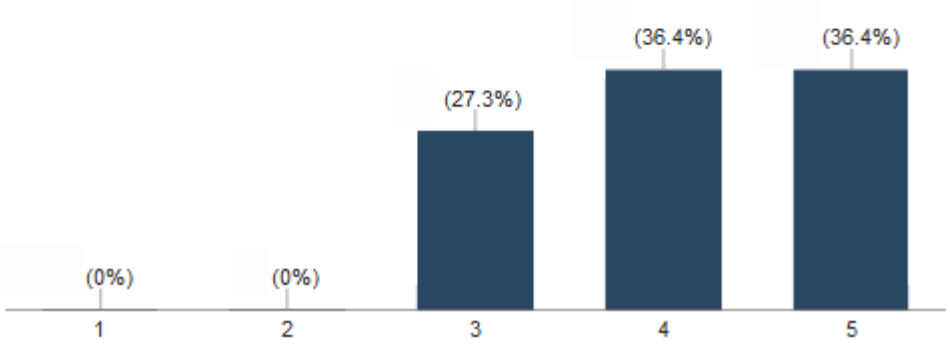
Have you ever played a game with similar incentives?



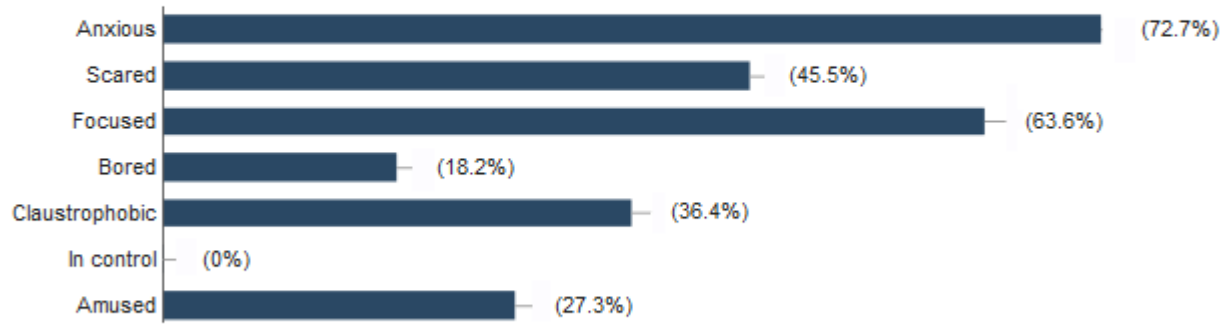
How well do you think VR (Virtual Reality) contributed to the game?



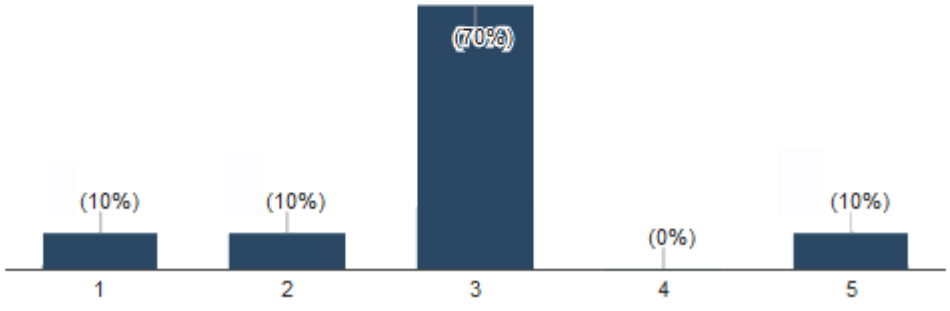
Did you find the game immersive?



Which feelings would best describe how you felt while playing the game?



Has this experiment helped you relate with visually impaired individuals in any way?



Bibliography

[1] Gerald Penn âĂŞ Natural Language computing "SoundASR"

[2] Lerder, Francis C. "The Human Ear." Encarta Encyclopedia Deluxe 2004

[3] MEDEL - The Benefits of Complete Cochlear Coverage

[4] Cancer Research UK - Cancer of the ear canal or middle ear and the inner ear

[5] Article from "Knowing Neurons" *[link: knowingneurons.com]*

[6] Newitz, Annalee: "Is the uncanny valley a myth?"

[7] ASU - School of life sciences - What is echolocation

[8] Ben Underwood's official website *[link: benunderwood.com]*

[9] Horror game by Justin xiong *[link: lurking-game.com]*

[10] Sequel from "Lurking" *[link: stifledgame.com]*

[11] Horror game by "The Deep End Games" *[link: store.steampowered.com/app/426310/]*

[12] "Dark Echo", horror game by RAC Seven *[link: darkechogame.com]*

[13] Horror game by Aaron Rasmussen and Michael T. Astolfi *[link: blindsidegame.com]*

[14] Virtual barber shop experience *[link: youtube.com/watch?v=IUDTlvagjJA]*

[15] A virtual reality narrative game *[link: quanero.com]*

[16] 3D model website *[link: turbosquid.com/Search/3D-Models/free]*

[17] VR headset by Google *[link: vr.google.com/cardboard/]*

[18] VR headset by Samsung *[link: samsung.com/global/galaxy/gear-virtual reality/]*

[19] VR headset by Oculus *[link: oculus.com/rift/]*

[20] VR headset by HTC *[link: vive.com]*

[21] Steam VR support website *[link: support.steampowered.com]*

[22] Screen shot from “Return to the future” by Bob Gale and Zemeckis

[23] cross-platform 3D puzzle game by Thekla created and directed by Jonathan Blow

[24] ‘Fun with Unity’ project by Ben Britten *[link: BenBritten.com]*

[25] “FadeObjectInOut.cs” by Hayden Scott-Baron (Dock) *[link: starfruitgames.com]*

[26] Roth, Scott D. 1982, "Ray Casting for Modeling Solids", Computer Graphics and Image Processing.