UNIVERSIDADE BEIRA INTERIOR

Development Of A First-Person Horror Game With High-Fidelity Graphics

(Versão final pós-defesa)

João Pedro Costa Tinoco

Relatório de projeto para obtenção do Grau de Mestre em
Design e Desenvolvimento de Jogos Digitais
(2º ciclo de estudos)

Orientador: Prof. Doutor Frutuoso Gomes Mendes da Silva

Covilhã, Agosto 2022

Dedicação de Integridade

Eu, João Pedro Costa Tinoco, que abaixo assino, estudante com o número de inscrição M10665 de Design e Desenvolvimento de Jogos Digitais da Faculdade de Artes e Letras, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior.**

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 31 / 08 / 22

João Pedro Costa Tinoco

Acknowledgements

The conclusion of this project, as well as most of my academic life, would not be possible without the help of all the people who were present.

Work developed at ”Instituto de Telecomunicações” under supervision of the Prof. Frutuoso Silva, whom I thank for mentoring and supporting me once more.

I want to thank my family, especially my mother and my stepfather who I am proud to call my father. Without them none of this would be possible, they were always making sure that I lacked nothing, always available to support me and take care of me. I also want to thank my grandparents for all the love they've given me since I was born. A big thanks to my sister and her boyfriend for all the advice they gave me to complete this stage.

I want to thank all my friends, especially the closest ones in my life, João Barbas, José Oliveira, and Carolina Dores for giving me unforgettable moments since our childhood, for all their advice given, and for being incredible friends.

I want to thank my great friends/roommates, Rafael Silva, Sérgio Gonçalves, and Pedro Silva, for all the hours, minutes and seconds spent together playing card games, watching films and TV, partying, ”coffee conversations”, as well as for the advice and help that I've been given throughout the years, hoping that we remain great friends once we all go to our separate ways.

I want to thank my godmother Inês Ramos, for all the friendship and advice throughout my academic life. I also want to thank my godsons, Ricardo Matias, Daniel Oliveira, Gabriel Maria, Manuel Casacão, and Ângelo Morgado, for all the friendship and delightful memories in 2021/2022.

Last but not least, I want to thank my good friends João Garcia, who without him this project was not possible to develop, Inês Inácio, André Martins, Eliseu Batista, Rita Martins, Luis Rei, Joana Almeida, and António Duarte for all the friendship and countless moments spent together to unwind from work.

Resumo

Victims of Dead Stories é um jogo de terror psicológico para um jogador na perspectiva da primeira pessoa, com gráficos de alta fidelidade, onde o jogador explora os corredores de uma casa enquanto experiencia ocorrências sobrenaturais, e progride na narrativa enquanto percorre a casa. A principal inspiração para *Victims of Dead Stories* é *P.T. Silent Hills*, desenvolvido pela Kojima Productions, onde o jogador está constantemente a experienciar loops dos corredores.

O principal objectivo deste projecto é conceber e desenvolver um jogo com gráficos realistas, alcançando a aparência de um jogo AAA, mantendo ao mesmo tempo um bom desempenho e oferecendo opções de optimização ao jogador. Este projecto é também uma prova do que dois estudantes de áreas diferentes podem alcançar, do que os criadores independentes podem realizar com amor e dedicação a um jogo.

Quanto à metodologia, este documento consiste em detalhar o design de várias áreas do jogo - a narrativa, o *level design*, as mecânicas do jogo, entre outras - assim como em detalhar o desenvolvimento dessas áreas de design, utilizando o motor de jogo *Unity* e a linguagem de programação C#, para implementar e exportar o jogo para a plataforma PC.

Palavras-chave

Jogo de Terror, Unity Engine, Primeira-Pessoa, Gráficos Realistas, C#

Abstract

Victims of Dead Stories is a psychological single-player horror game in the first-person perspective, with high-fidelity graphics, where the player explores the hallways of a house while experiencing supernatural occurrences, and progressing in the narrative while looping around the house. The major inspiration for Victims of Dead Stories is P.T. Silent Hills, developed by Kojima Productions, while the player is constantly looping around the house.

The main objective of this project is to design and develop a game with realistic graphics, reaching an appearance of a AAA game, while maintaining good performance and offering optimization options to the player. This project is also proof of what two students from different areas can achieve, of what indie developers can accomplish with love and dedication to a game.

As for the methodology, this document consists of detailing the design of various areas of the game - the narrative, level design, game mechanics, among others - as well as detailing the development of those design areas, by using the Unity Engine and the C# language, to implement and export the game for the PC platform.

Keywords

Horror Game, Unity Engine, First-Person, Realistic Graphics, C#

Contents

List of Figures

List of Tables

Code Sample List

Acronyms List

API Application Programming Interface

FXAA Fast Approximate Anti-Aliasing

FOV Field-Of-View

HDRP High Definition Render Pipeline

IDE Integrated Development Environment

MSAA Multi-Sample Anti-Aliasing

OOP Object-Oriented-Programming

POV Point-Of-View

SFX Sound Effects

SMAA Subpixel Morphological Anti-Aliasing

SSAO Screen Space Ambient Occlusion

UBI University of Beira Interior

UI User Interface

VoDS *Victims of Dead Stories*

Chapter 1

Introduction

This project was developed during the second year of the Master's degree in Game Design and Development at University of Beira Interior (UBI) and consists on the realization of a horror game, called *Victims of Dead Stories* (VoDS).

1.1 Scope of the Project

When it comes to designing and developing a video game many people, especially indie developers, find it almost impossible to build a project on a large scale or to achieve the game of their dreams. This might be since, in order to create a solid and stable game with high complexity, it would require a large team and different departments within the team to focus on various aspects of the game. If that's the case, why won't indie developers just join an established game developer studio and work for them when their dream job is to develop games? Saying it is easy but the requirements to work on prestigious companies are demanding, especially when said companies produce AAA games[7] thus requiring employees with years of experience to work for them. For that sole reason, indie developers have no choice but to design and develop their game alone or with a small team with practically no salary since they would only earn any profit when the product releases after many years in development.

The five men team behind the hit game *Valheim* [8], an open-world survival game played from a third-person perspective developed by Iron Gate Studio [9], is a good example of what indie developers can achieve despite the years that took to develop the game. Before *Valheim*, one of the studio's co-founders, Richard Svensson, worked at a Swedish game company. In 2017, Svensson began his work on *Valheim*. Later in 2018, he left to work on the game full-time while gathering a few people to work with him. In February 2021, *Valheim* was released into early access and received positive reviews from various critics. A month later, the game sold five million copies and was one of the most-played games on Steam [10]. While big game studios may take 2 to 3 years to launch a AAA game, Iron Gate Studios is a good example of what indie developers can reach, despite the years of hard work, dedication, and love that is laid on the design and development of a single game.

Inspired by the will of these indie developers and to prove to the industry of video games that indie games are not to be underrated, this project is aimed to design and develop a 3D first-person horror game with realistic graphics, where a player finds himself inside a house confined with only hallways and a room, exploring its surroundings. This document should also overview the approaches and difficulties during the development

of the game, regarding the skill limits of the developer and the maintenance of good performance and optimization of the game while offering high-fidelity graphics.

1.2 Motivation

Games, just like movies or TV series, are part of our everyday life and offer a lot of entertainment, mostly among young people and adults [11]. Having a huge love for games since childhood and an aspiration involving the digital games industry (namely game designer, game developer and/or creative director) this project provides an increase of knowledge and experience in game design and development. Thus, this project aims to design and develop a first person game addressing the horror genre.

1.3 Objectives and Methodology

As described in sections 1.1 and 1.2, the goal of this project is to design and develop a 3D horror game, VoDS, in the first-person perspective, focusing on the narrative and the gameplay that is offered to the player, comprised only with sections of a house (hallways and a living room) in which the player will be constantly looping around. The game's narrative revolves around the main character that suffers from a mental disorder know as schizophrenia and how this problem affects him.

However, the main purpose of the project is to improve and develop new skills in creating games that include gameplay mechanisms such as first-person view, interaction, event systems depending on the narrative, high-fidelity graphics, among others. The game should have at least ten levels of gameplay, meaning one loop represents a level, where the player will have to explore in order to progress in the narrative.

In order to make this project possible, the game engine Unity [12] should be used, since it is very powerful to develop any kind of game, has an enormous amount of support, excellent documentation and it has free access. To achieve and maintain high-fidelity graphics, the HDRP [13] within Unity should be used. The 2D and 3D modeling and animation software Blender [14] should also be used, which is a great tool to create game assets that increase the detail of the game environment. However, the area of 3D modeling will be handled by the co-developer of this project, João Garcia.

Thus, this project was developed by a team of two students. Both students worked on the design areas of the game, however one student was responsible for the programming and development section while the other student was responsible for the design and modeling of the game assets. This report describes the part of development related with the programming issues, as well the design section of the game.

1.4 Document Organization

In order to reflect the work that has been done, this document is structured as follows:

1. The first chapter – **Introduction** – presents the project and its framework, as well as its motivation, objectives and methodology, and the respective organization of the document.
2. The second chapter – **Related Work** – presents some games developed by game developers/studios, which served as inspiration for the way in which some of their elements were implemented in this project.
3. The third chapter – **Technologies and Tools Used** – presents the technologies and tools used that allowed the development of this project.
4. The fourth chapter – **Design of Victims of Dead Stories** – describes the design of the game, detailing its functionalities, mechanics, level design and others.
5. The fifth chapter – **Development of Victims of Dead Stories** – describes the development of the project's functionality, its operations and approaches.
6. The sixth chapter – **Tests, Optimization and Performance** – describes the tests performed and the feedback received by some testers, the optimization and performance of the game.
7. The seventh chapter – **Conclusions and Future Work** – describes the conclusions coming from the design and development of this project, what was left undone, what can be improved and its possible future work.

Chapter 2

Related Work

This chapter presents some works (in this case, films and games) that were developed by game developers and film studios. These works served as inspiration, as some elements were taken from them that deemed fit for the realization of this project.

2.1 Video Games

2.1.1 P.T. (Silent Hills)

P.T.[15] (short for ”Playable Teaser”), is an interactive teaser, a first-person psychological horror game in the survival horror genre, developed by Kojima Productions[16] and published by Konami[17]. The game was directed and designed by Hideo Kojima, in collaboration with film director Guillermo Del Toro. The game was intended to be a teaser for Silent Hills[18], an installment in the Silent Hill franchise[19], but it was canceled by the publisher.

The main and sole gameplay of P.T. is exploring an L-shaped hallway that appears to be in a loop, meaning that at some point the player will find himself at the beginning of the hallway, just as he started the game. The game also offers the player the ability to zoom the camera in order to advance or complete some puzzles/loops of the game.

Figure 2.1: Image showing the beginning of the looping hallway[1].

P.T. is one of the main inspirations to develop VoDS simply because it possesses a linear, clear, and effortless gameplay throughout the loops, making it also a strong and viable way to progress through the narrative and expose it to the player while making it easier for an indie developer to create stories without the need for cinematics or cutscenes. Therefore, the idea of having this looping gameplay is a perfect choice to implement in VoDS to tell a horror story through the environment. P.T.'s realistic graphics is also the intended goal for this project to provide an authentic horror feel for the players. The zoom functionality will be a great and simple addition to implement in the game as well as some elements of horror present in P.T. will also serve as inspiration for the game.

2.1.2 Visage

Visage[20] is a first-person psychological horror game, developed and published by Sad-Square Studios[21]. According to the developer, Visage is conceptualized to mainly utilize "the uncanny" to create a sense of dread and fear, was inspired by P.T. and after its cancellation it motivated the studio to pursue the creation of Visage.

In Visage, while the player explores a mysterious ever-changing house in a slow-paced, atmospheric world that combines both uncannily comforting and horrifyingly realistic environments, the player can pick up some objects and examine them, by rotating and zooming in and out the objects, either to find clues or keys to advance in the narrative or to just simply visualize objects with greater detail.

Figure 2.2: Image exemplifying the examine system[2].

This type of game mechanic in *Victims of Dead Stories* would be perfect for the player to lose himself and sink in the narrative offered by the developer as it offers yet a new way to progress in the game and also awakes the explorer within the player. With the examine system, the developer can create innumerable objects for the player to interact with and inject them with the game's lore.

2.1.3 Phasmophobia

Phasmophobia[22] is a four player online co-op investigative horror game in the first-person perspective, developed by the indie game studio Kinetic Games[23] using the Unity game engine.

Phasmophobia allows the player to experience thrilling moments of gameplay as he interacts with ghosts, spirits and other paranormal phenomena. The player can join up to three more players and monitor the location with CCTV cameras and motion sensors from the safety of the van or head inside and experience ghostly activity that gets increasingly hostile as time goes on.

Figure 2.3: Image of the game's graphics[3].

Being a game made with Unity, Phasmophobia serves as an inspiration for *Victims of Dead Stories* in the way to achieve a great threshold of performance and optimization while having realistic graphics with the Unity engine while using the HDRP.

2.2 Films

2.2.1 Hereditary

Hereditary[4] is a horror film from A24[24], written and directed by Ari Aster[25].

Annie is a wealthy model maker with a family history of mental instability. After her elderly mother passes away, Annie, her disturbed daughter Charlie, and, eventually, her teenaged son Peter begin to experience strange visions and compulsions, much to the distress of her skeptical husband, Steve.

Figure 2.4: Hereditary by A24[4].

This film helped in understanding the behaviors of a family with a history of mental instability and how some moments, choices and thoughts affected their lives, thus some horror and disturbing elements from this film served as an inspiration for the game.

Chapter 3

Technologies and Tools Used

This chapter provides a brief overview of the technologies and tools used during the development and elaboration of this project.

3.1 Unity - Game Engine

The game engine used to develop VoDS was Unity. This technology is incredibly powerful to developing any kind of game, spanning multiple popular platforms such as PC, Mac, Linux, PS4, PS5, Xbox One, Xbox Series X, Android and iOS, among others. With great documentation support and usage available for free, anyone can develop from a simple game to a game of high complexity and detail, be it 2D or 3D.

3.1.1 High Definition Render Pipeline (HDRP)

The HDRP is a high-fidelity Scriptable Render Pipeline built by Unity to target modern (Compute Shader compatible) platforms.

HDRP utilizes Physically-Based Lighting techniques, linear lighting, HDR lighting, and a configurable hybrid Tile/Cluster deferred/Forward lighting architecture and provides the tools necessary to create games, technical demos, animations, and more to a high graphical standard.

3.1.2 The C# language

Programming is indispensable for making any game, regardless of the game engine used, because without it would be impossible to create and develop a game.

All programming in Unity is processed through the C# programming language. This language, developed by Microsoft running based on the .NET Framework, is used to develop web applications, desktop applications, mobile applications, games among others. Just like the Java language, C# is an Object-Oriented-Programming (OOP) in which it works based on classes, objects and methods.

3.1.3 Visual Studio Community 2022

Upon using Unity for the first time, the engine recommends using as Integrated Development Environment (IDE) the Visual Studio Community 2022, developed by Microsoft so that all the programming can be interpreted. Developers can choose either another IDE of their choice or stick with the recommended one, but for simplicity and already being accustomed to it from previous works, the latter was chosen.

3.1.4 Unity's Packages

The Unity Packages [26] that were used in this project are described in the next sections.

3.1.4.1 Input System

The Input System [27] package implements a system to use any kind of Input Device to control our Unity content. It's intended to be a more powerful, flexible, and configurable replacement for Unity's classic Input Manager (the *UnityEngine.Input* class).

With the use of this package it was possible to configure generic and brand gamepads (PlayStation and Xbox controllers) and the mouse & keyboard to work simultaneously, offering the player more flexibility upon playing the game.

3.1.4.2 Animation Rigging

The Animation Rigging [28] package allows us to create and organize sets of constraints to address our rigging needs. For example, it's possible to create deform rigs with procedural secondary animation to control character armor, props, and accessories as well to create world interaction rigs (sets of IK and Aim constraints) for interactive adjustments, targeting, and animation compression correction.

The use of this package allowed us to create a rig on the paranormal entity haunting the player, to create eerie animations, and play them at certain moments of the gameplay.

3.1.4.3 Platforms

The Platforms [29] package provides platform and build Application Programming Interface (API) foundation used by other platform packages such as the Entities package.

By using the Entities package to create, load and unload sub-scenes the Platforms package is mandatory to be used within the project to build the game to play it because the default methods to build the game doesn't support sub-scenes.

3.1.4.4 Post-Processing

Post-processing [30] is a generic term for a full-screen image processing effect that occurs after the camera draws the scene but before the scene is rendered on the screen. This package comes with a collection of effects and image filters to simulate physical camera and film properties.

The usage of this package allowed our game to drastically improve its visuals with little setup time, by achieving a horror cinematic feel.

3.1.4.5 ProGrids

ProGrids [31] is an advanced grid and object snapping tool for the scene view. This package is a dynamically placed grid, available on all three axes and at any position in the scene. ProGrids snaps objects in world space, ensuring consistent placement of objects while building our levels.

The benefit of ProGrids provided a uniform arrangement upon building the sections of the house with no hassle.

3.1.4.6 Unity UI

Unity UI [32] is a User Interface (UI) toolkit for developing user interfaces for games and applications. It is a GameObject-based UI system that uses Components and the Game View to arrange, position, and style user interfaces.

Without this package, it's practically impossible to create a UI for the player to navigate through the game's menus and receive visual feedback. As no game exists without UI, and for its purpose, this package was used to create a fluid and simple UI for the Main Menu and Pause Menu in the game.

3.1.4.7 TextMeshPro

TextMeshPro [33] is the ultimate text solution and replacement for Unity's UI Text and the legacy Text Mesh. Powerful and easy to use, TextMeshPro uses Advanced Text Rendering techniques along with a set of custom shaders, delivering substantial visual quality improvements while giving users incredible flexibility when it comes to text styling and texturing. TextMeshPro provides Improved Control over text formatting and layout with features like character, word, line and paragraph spacing, kerning, justified text, Links, over 30 Rich Text Tags available, support for Multi Font & Sprites, Custom Styles and more.

VoDS's UI contains multiple information for the player. Therefore this package was indispensable to freely customize the text and the way they behaved in certain moments of interaction with the player and for the implementation of subtitles to provide readable information for players with hearing disabilities or difficult comprehension of the dialogue within the game.

3.1.5 Unity's Libraries

The Unity Libraries that were used in this project are described in the next sections.

3.1.5.1 *UnityEngine*

The *UnityEngine* [34] is Unity's mother library, where it contains all the essential functionality for a game to work, covering artificial intelligence, animations, audio, networks and more.

Upon creating a default C# script this library is already implemented and by utilizing *UnityEngine* we are actually using the C# version of Unity, as without it it's not possible to perform the operations that the game engine provides.

3.1.5.2 *UnityEngine.UI*

UnityEngine.UI [35] is a UI toolkit for developing user interfaces for games and applications. It is a GameObject-based UI system that uses Components and the Game View to arrange, position, and style user interfaces.

This library contains various classes that were capable of controlling the behavior of the implemented UI like buttons, sliders, toggles, and others to update the relative information in the game.

3.1.5.3 *UnityEngine.InputSystem*

To take full advantage of the Input System package mentioned in 3.1.4.1, the use of this library was indispensable to register and interpret callbacks from the supported devices that are connected throughout the game, in order to perform the desired actions established during the design and development phase.

3.1.5.4 *UnityEngine.InputSystem.Utilities*

This library is an extension of the main library mentioned in 3.1.5.3 for the sole purpose of having extended support with the use of its classes and structs.

3.1.5.5 *UnityEngine.Rendering*

The use of *UnityEngine.Rendering* [36] allowed us to use the Volume class, responsible to gain access to the basic properties of the Post-Processing used in the game.

3.1.5.6 *UnityEngine.Rendering.HighDefinition*

Even though the library mentioned in 3.1.5.5 allowed us to access the basic functionalities of the Post-Processing, in order to modify the desired image filters and effects, through scripting, used in the game, the *UnityEngine.Rendering.HighDefinition* [37] was necessary to alter values of various proprieties when desired, such as Bloom, Motion Blur, Anti-Aliasing, among others.

3.1.5.7 *UnityEngine.EventSystem*

The *UnityEngine.EventSystem* [38] is a way of sending events to objects in the application based on input, be it keyboard & mouse, gamepads, or custom input. The Event System consists of a few components that work together to send events. As such, the use of this library was crucial to manage the behavior of certain interactions within the UI.

3.1.5.8 *UnityEngine.SceneManagement*

The Unity engine allows the creation of scenes [39], meaning in each different scene the developer can create different levels or areas instead of having the entire game product in one scene only, which has a dramatic effect on the game's performance.

The use of scenes was indispensable and the most obvious way to organize the game flow when transitioning from the main menu to the gameplay and vice versa, and for that, the main use of this library was to control at which point the scenes would be loaded and unloaded.

3.1.5.9 *UnityEngine.Audio*

A game would be incomplete without some kind of audio, be it background music or sound effects. Unity's audio system is flexible and powerful. It can import most standard audio file formats and has sophisticated features for playing sounds in 3D space, optionally with effects like echo and filtering applied.

Just like the previous libraries, the audio library [40] allowed us to completely change the way the game's audios behave, such as when to play and to pause, and change their volume within the game settings that are offered to the player.

3.1.6 Unity Asset Store

The Unity Asset Store [41] contains a vast library of free and commercial assets that Unity Technologies as well as members of the community create. A wide variety of assets are available, including Textures, Models, animations, entire project examples, tutorials, and Editor extensions.

Although most of the game models and textures present in the game were design and modeled by João Garcia, some other models and textures from the Unity Asset Store were used in order to save time and resources upon the development of VoDS.

Chapter 4

Overall Design of Victims of Dead Stories

Designing a game is one of, if not, the most important parts when developing a game because game design is where all the ideas and viewpoints are discussed and brainstormed in order to achieve a concise, balanced, and enjoyable game experience for most players. This chapter mainly focuses on the final aspects of the game. VoDS is available to download and play in the following link:

`https://khallington.itch.io/victims-of-dead-stories`.

The design of this project followed the four basic elements, Mechanics, Story, Aesthetics, and Technologies, called the *Elemental Tetrad*, formulated by the author Jesse Schell in *The Art of Game Design: A Book of Lenses* (p. 41-43) [42], as a guide in order to initiate the production of the project. The *Elemental Tetrad* is illustrated in figure 4.1 (Schell 2008 p. 42 figure 4.3).

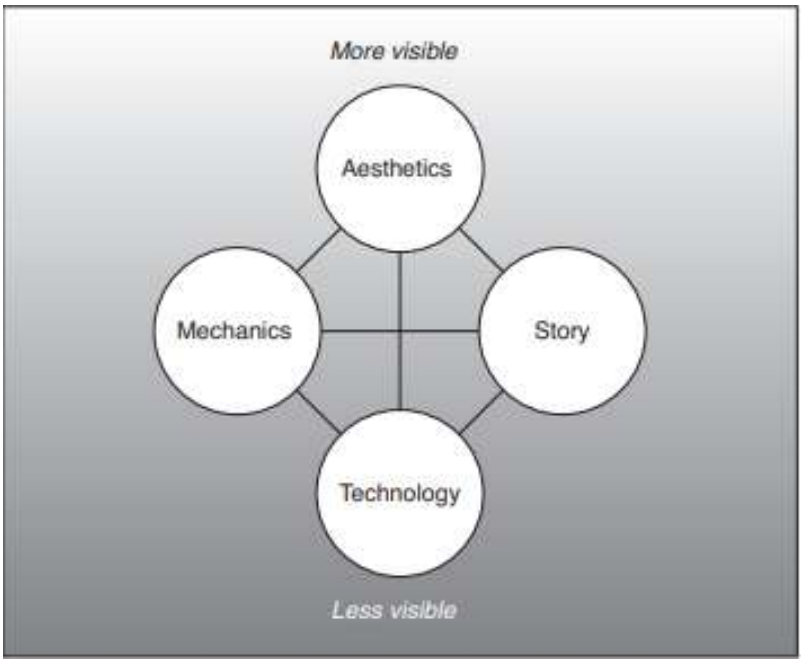

Figure 4.1: The Elemental Tetrad and its correlations between the elements.

Being a project developed by a team of two students coming from distinct areas of study, both students designed the overall feel, experience, and looks of the game, while each one was in charge of a specific area of their expertise to develop the game. Nonetheless, this chapter describes the design and choices that best fit the game's creative vision.

4.1 High-Concept

The purpose of high-concept is to easily pitch ideas, upon the production of any project, with a succinctly stated premise. Therefore, this project's main premise is the following:

VoDS is a psychological single-player horror game in the first-person perspective where the player is invited to explore parts of a haunted house while experiencing supernatural occurrences, and progressing in the narrative in looping events around the house.

The game has no fighting mechanics, thus meaning the player has no means of defense upon facing a ghostly apparition. The mechanics are extremely simple and linear; the player can only walk and interact with a few objects and zoom in to observe the environment in greater detail. The player cannot die and there are no saving features, meaning all progress is lost upon quitting the game.

The game's graphic style is of high-fidelity, which can be relatively compared with games that have the same, or higher, degree of graphic realism, such as the ones referred to in 2.1, and other games.

4.2 Narrative Overview

In consonance with the author, the Story element is the sequence of events that unfolds in the game. It may be linear and pre-scripted, where the story has a predetermined series of events that are presented in a specific order, or it may be branching and emergent, where the story consists of a graph of nodes connected by arcs that represent user choices, meaning that every possible path through the graph represents a story that can be told to the user (Schell, 2008, p. 41).

Schell also states that to unfold the narrative along with the game, we have to choose mechanics that would both nourish that story and let that story emerge. This process is simplified due to the fact of VoDS is a horror game. In many games, it's possible to observe the relationship between the Story element and the Mechanics element. For example, in *Visage* (2.1.2) the story is told in chapters and the mechanics are fairly simple and it simulates the realism of human behaviors like opening drawers and doors, storing objects in an inventory, and others. The developers most certainly chose these mechanics because this way they fit within the narrative, meaning it's in conformity with the story and helps to strengthen it.

Our intention in VoDS's narrative is to tell it in an anthological way with chapters, where each chapter presents a different story and a different set of characters, with the possibility of the mechanics varying between the chapters following the story.

The narrative developed for this project is the first chapter called *The Never-Ending Thoughts*, in a linear form. Vaguely, this chapter is about a man suffering from schizophrenia who ultimately murdered his pregnant wife due to his condition getting worse. The way this narrative's told is through the main characters thoughts, in the first-person perspective, throughout the various loops, across the scripted events the player stumbles upon as well as the occasional analysis of objects and conditions of the environment.

Although the narrative for the first chapter was developed, due to time and schedule limitations, the story is somewhat left unfinished but this is not entirely disadvantageous for the game, meaning that the majority of what the player finds and is depicted in the game is open to interpretation, leading them to develop and discuss theories about the nature of the events that occur in the game, keeping them interested in the game while more content can be produced and completed.

4.3 Graphics Style

According to Schell, the Aesthetics element is how a game looks, sounds, smells, tastes and feels (Schell, 2008, p. 42). This element is an incredibly important aspect of game design since it has the most direct relationship to a player's experience. If a game has appealing graphics and colors, it might attract more players and increase their immersion in it, rather than a game with an inconsistent or unpleasant graphics style.

We opted for a high-fidelity graphics style for VoDS because in horror games it helps to keep them more grounded, threatening and frightful in correlation with the narrative, as seen in figure 4.2.

Figure 4.2: Representation of the high-fidelity graphics achieved for the game.

4.4 The Main Character

Forward in *The Art of Game Design: A Book of Lenses*, Jesse Schell talks about the main character called the avatar (Schell, 2008, p. 312). This special character is the one the player controls in a game. During the gameplay, there are times when the relationship between the player and the character is distinctly apart, but there are other times where the player's mental state is completely immersed in and projected into the avatar, to the point where the player may feel emotion when the avatar is injured or threatened.

Schell also states that designers often debate about which is more immersive, the first-person perspective or the third-person perspective. The author elaborates that one argument is that greater projection can be achieved by providing a first-person perspective on a scene with no visible avatar to the player. However, the power of empathy between the player and the avatar is strong when the player is controlling a visible character, to the point where the player can often wince in imagined pain or suffering upon seeing the avatar suffer a blow or a sigh in relief upon seeing the avatar escape physical harm.

Schell concludes that the avatar is a player's gateway into the world of a game and to ensure the avatar brings out as much of the player's identity as possible, it's important to bear in mind these questions:

- Is my avatar an ideal form likely to appeal to my players?
- Does my avatar have iconic qualities that let a player project themselves into the character?

As for the design choice in VoDS, it was chosen for the avatar to not be visible to provide more immersion to the player, in relation to the story and its environments, from the first-person perspective. Does this avatar fit within the questions Schell asks to consider when creating the main character? No, although we believed that if the avatar was visible, even in the first-person perspective, it could distract the player due to its movements (i.e. walking, picking items) during the gameplay.

4.5 First-Person Camera

Video games with the first-person perspective attempt to show the world from the character's Point-Of-View (POV). The player observing the environment from the avatar's perspective provides gameplay advantages.

One of the advantages of a POV camera is that it's not required to create complex animations for the character nor it's needed to implement a manual or automated camera-control scheme, present in games with third-person perspective, because the player controls the camera at will.

Another advantage is that the first-person perspective can also make it easier for the player to interact within the environment. Since the character may not be shown on the screen, the player can precisely aim at objects to interact with them whereas, in a game with a third-person perspective, this level of precision is impossible, therefore interactive objects must be marked with visual feedback so the player knows that it can be interacted with. An example of a first-person game with shooting mechanics is illustrated in figure 4.3.

Lastly, the final major benefit is the level of immersion offered to the player, previously discussed in the section 4.4. Since the avatar is not shown to the player, this means that he can project himself on the game as it feels that they are the main character in the game, rather than controlling one.

Figure 4.3: Representation of a first-person perspective game where the player can easily aim a gun and feel immersed[5].

Unfortunately, the first-person camera also contains some disadvantages. Any game with this kind of perspective will instantly seclude some players to play it. In some people, the first-person perspective can cause headaches and motion sickness, therefore some people may refuse to play these types of games..

Another disadvantage is that, even though this perspective increases the player's immersion, it also limits how cinematic the game can be. If in a game the developers intend to show cutscenes in the first-person perspective, this can be harsh when the player no longer has control. However, if the game then suddenly changes to a different perspective to better represent cutscenes, this might also break the player's immersion in the game.

Ultimately, for VoDS, the first-person perspective was chosen as the way the player sees and interacts with the world. Since the game contains interactive mechanics (further explored in the next section 4.6 and is a horror game, it helps the player to analyze and observe specific objects within the environment and the ambiance itself as well as immerse the player within the game in moments of threat and danger.

Additionally, this choice also helped in regards to animations, where it was not needed to create complex animations, thus saving time and resources during the design and development of this project.

4.6 Game Mechanics

In the Elemental Tetrad, the Mechanics are the procedures and rules of a game as they describe the goal of a game, how players can and cannot try to achieve it and what happens when they try. By comparing games to a more linear form of entertainment like films, TV Series, books, and etc., it is notable that they do not involve any type of mechanics whatsoever, and for that it's mechanics that make a game truly a game to experience. When choosing a set of crucial mechanics to a gameplay, it is needed to choose a technology that can support them, aesthetics that emphasize them clearly to players, and a story that allows the mechanics to make sense to the players (Schell, 2008, p. 41).

In VoDS, the game mechanics that were designed are:

- Character Movement;
- Camera Movement;
- Camera Zoom;
- Interact with objects;
- Examine objects;
- The Loop System.

4.6.1 Character Movement

The first and one of the two most important **basic game mechanics** present in all games is the character movement as it is indispensable in a video game for a player to be able to control the avatar's movements at will. In general, when discussing during the design phases, the game mechanics are only related to the specific and special rules that make a game unique or standout, and so, the basic game mechanics are as if they're already implemented in said game. Even though it's obvious that in a game a player must control the character's movement, it's still important to briefly review it regarding this project.

In VoDS, the player is able to move the avatar, within the game's 3D environment, forwards, backwards, and side to side.

4.6.2 Camera Movement

The second and last of the two most important **basic game mechanics** is the ability to control the movement of the game's camera. Before designing the camera's behavior it's important to choose if the player is able to control the camera or not, or if the player can sometimes control the camera.

Since in VoDS the camera has a first-person perspective, this means that the camera will be positioned at the level of the eyes of the avatar, therefore it is needed for the player to fully control the camera's behavior at will in order to observe, process and analyze the environment that is offered to the player. As so, the player is able to move the camera by tilting it, rotating up and down at a 90º degrees angle, and by panning it, rotating side to side at a 360º degrees angle. These movements are represented in figure 4.4.

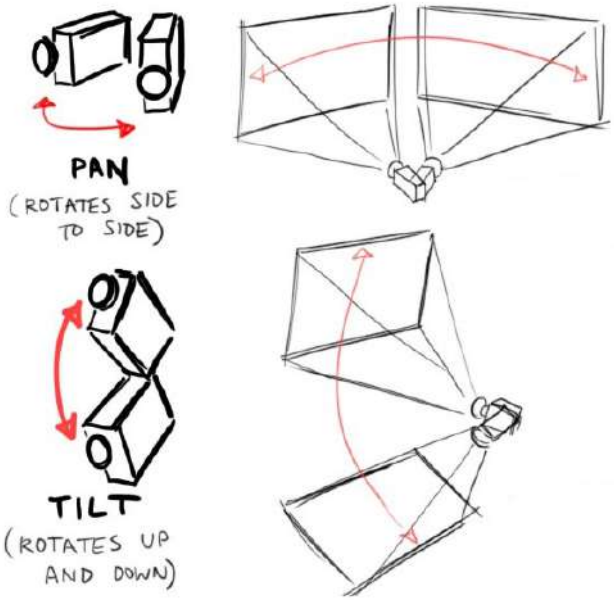

Figure 4.4: Simple representation of camera movements [6].

4.6.3 Camera Zoom

The camera zoom mechanic is an ability that, throughout the gameplay, the player can execute in order to change the camera's Field-Of-View (FOV) so that it can perform a zooming effect. This mechanic is designed with the intention of the player to solve some puzzles along the game to advance the loops.

However, due to time and schedule limitations, this mechanic didn't achieve its full purpose but it was chosen to remain in the game because we believe it may offer players a way to visualize objects and observe the environment with greater detail. More information about this mechanic's full purpose are discussed in the future work section (7.2). The camera zoom mechanic is represented in the figures 4.5 and 4.6.

Figure 4.5: Camera with no zooming effect.

Figure 4.6: Camera with the zooming effect.

4.6.4 Interact with objects

Along with the gameplay, the player will find some objects to interact with, because this mechanic is a good way to trigger some events to progress to the next loops, or to simply just let the player interact with other objects with subliminal messages in relation to the narrative.

When the player stumbles upon an interactive object, within a certain range, visual feedback will be shown on the screen to prompt the player to interact with it, as seen in figure 4.7.

Figure 4.7: Interaction prompt shown to the player.

4.6.5 Examine objects

In VoDS, the capability to interact with objects is perfect for the player to lose himself and sink in the narrative since the developer can create innumerable objects for the player to interact with and inject them with the game's lore, therefore interacting with objects with such information is a good mechanic to increase immersion on the player.

Represented in the figure 4.8, upon examining an object, the player can control the object's behavior, by adjusting its rotation on its axis and scale to simulate a zooming effect on the object. When the player is in the Examine State, he cannot control the avatar's movement to simplify the development and programming of this mechanic and to avoid conflicts with both the avatar and object's movement.

Figure 4.8: The Examine State.

4.6.6 The Loop System

The Loop System is the main mechanic of this project, which centers on looping the level design with each loop containing subtle differences in the environment and gameplay events.

When playing the game for the first time, the player explores the house. Upon proceeding to the next loop, the player will notice that he's back to where he started, confusing and or startling him, by potentially intriguing him more to discover the story and explore the environment.

The figure 4.9 exemplifies the concept of the loop system to apply to the level design. The player starts the loop on the **Starting Point**, proceeded by exploring the level in the **Exploration Point**, to the moment where the player reaches the **Ending Point**, triggering a new loop in the Starting Point.

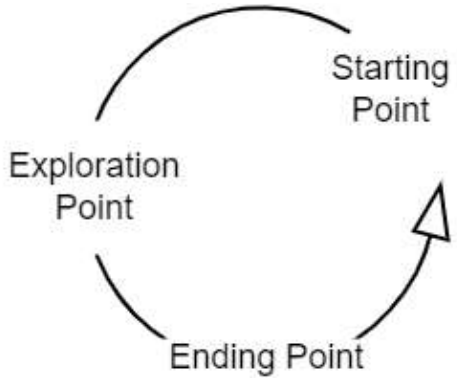

Figure 4.9: Exemplification of the loop system for the level design.

4.7 Controls

This section refers to the overall controls of the game, concerning the game mechanics, and how the player can trigger them.

In video games there are various devices a player can use to play a game, such as a gamepad (more common in console games), keyboard & mouse (for PC games), a steering wheel (for racing games), and other types of devices.

For this project, the gameplay was designed to be controlled with a gamepad or a keyboard and mouse. Playing VoDS with a gamepad is recommended for a more comfortable experience, although there are players who prefer a keyboard & mouse, and for that, it was designed a control scheme for both devices.

The main controls available to the player are:

- **Movement** - To move the avatar within the environment;
- **Camera / Zoom** - To move the camera / zoom the camera;
- **Interact** - To interact with objects;
- **Pause Menu** - To access the pause menu;

For the Examine State, there controls are:

- **Movement** - To move the object;
- **Zoom** - To zoom in or out on the object;
- **Back** - To exit the examine state;

4.7.1 Gamepad Control Scheme

The player is able to connect a wired or wireless gamepad. The supported gamepads can be a generic gamepad, an Xbox Controller or a DualShock4.

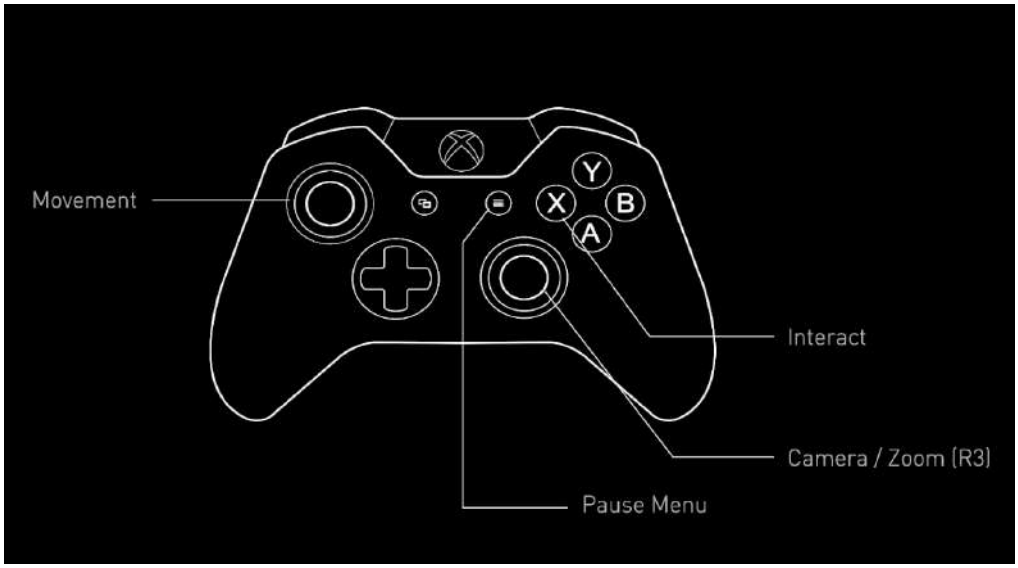

Figure 4.10: Representation of the gamepad control scheme in the Controls Screen.

For example, on an Xbox Controller, pressing the X button performs the interact mechanic and pressing the pause button will pause the game. The avatar movements are controlled with the Left Stick and the camera movements are controlled with the Right Stick. The zooming effect is performed when the Right Stick is pressed and held.

There are specific controls when the player enters the Examine State, however that control scheme is shown as the player plays the game (figure 4.8), therefore there is no necessity to illustrate it in the previous figure 4.10.

4.7.2 Keyboard & Mouse Control Scheme

Opposite to the gamepad, the player can play with a keyboard & mouse.

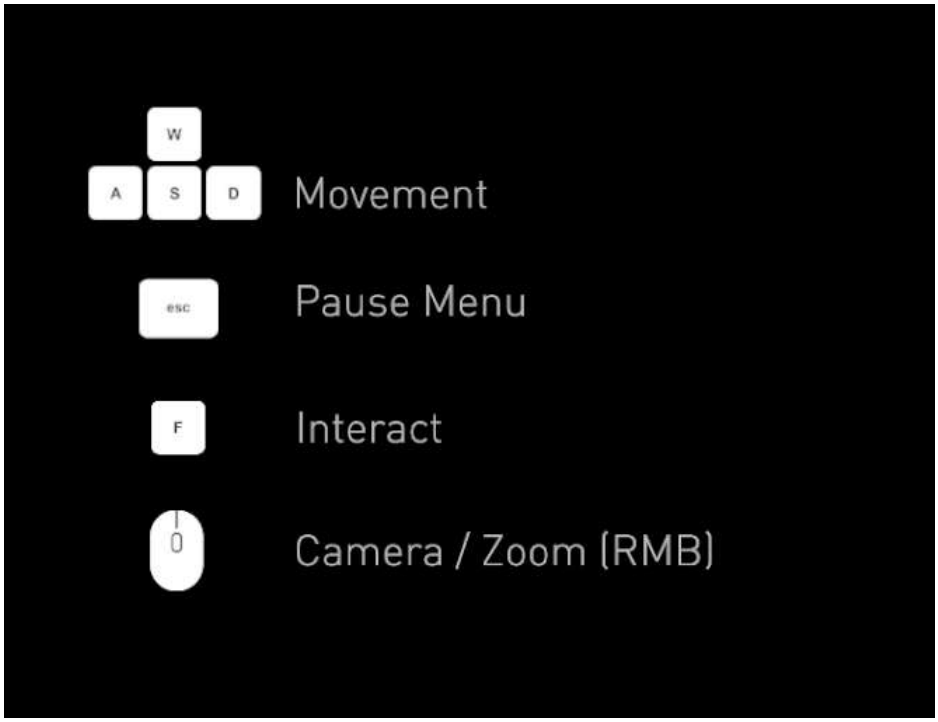

Figure 4.11: Representation of the keyboard & mouse control scheme in the Controls Screen.

In the keyboard, the W, A, S, and D keys trigger the avatar movements, the ESC key enables the pause menu, and the F key performs the interact mechanic. To control the camera's view, a simple movement of the mouse will change the camera's behavior, and by pressing the Right Mouse Button the zoom mechanic is triggered, as seen in figure4.11.

4.8 User Interface and Menus

As described in the subsections 4.6.4 and 4.6.5, the player will be able to interact with some objects throughout the gameplay experience. For that, it is necessary to provide visual feedback for the player to know which controls trigger the mechanic.

The visual feedback is a user interface, which is an excellent way to give information to the player that something may happen if the user prompts the given controls. As stated in section 4.7, the player can use an Xbox Controller, a DualShock4, or a Keyboard & Mouse to play the game. Each device has different icons to represent its controls, therefore, the user interface will be updated according to the current device connected to the game's system. The user interface for the interaction mechanic is illustrated in figure 4.7.

In VoDS, there are several menu screens the player can navigate through:

- **Title Screen**;
- **Main Menu Screen**;
- **Loading Screen**;
- **Pause Menu Screen**;

4.8.1 Title Screen

The Title Screen is the first experience the player will have upon launching the game.

Firstly, two sentences are shown, containing some information for the player to read, followed by the name of the developers, as seen in figures 4.12 and 4.13.

Figure 4.12: The first sentence to be shown to the player.

Figure 4.13: The second sentence to be shown to the player.

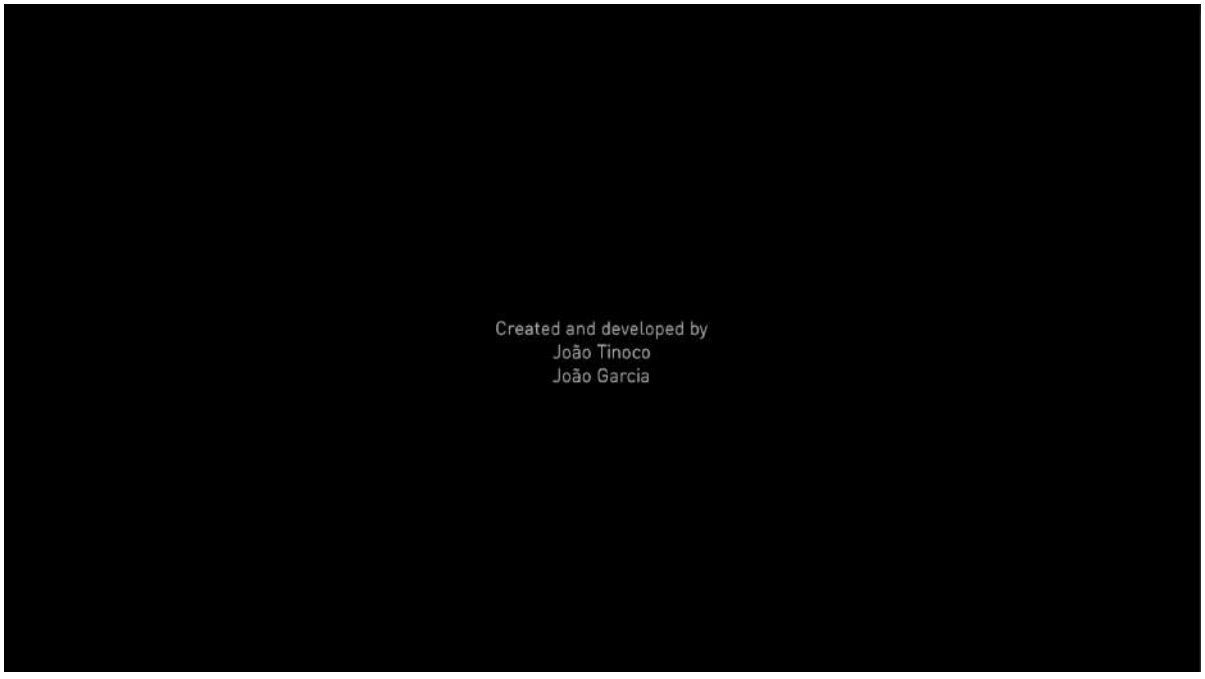

Figure 4.14: Credits to the developers.

Lastly, represented in figure 4.15, the game's title is shown.

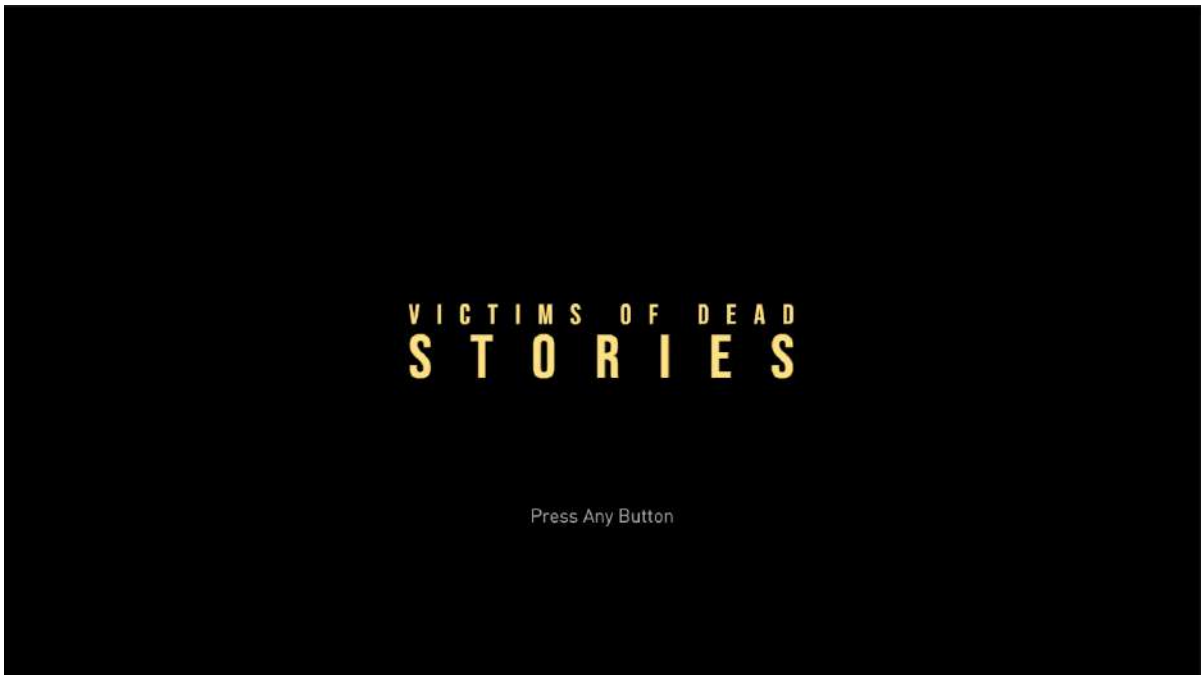

Figure 4.15: Logo of VoDS.

When the player presses any button, the game's title fades out, showing the Main Menu Screen.

4.8.2 Main Menu Screen

The Main Menu Screen contains three buttons, Chapters, Options and Quit, represented in figure 4.16. The Quit button lets the player exit the game.

Figure 4.16: The Main Menu Screen.

4.8.2.1 Chapters Menu Screen

To play the game, the player must enter the Chapters Menu Screen and select the first chapter, since it's the only one available, as shown in figure 4.17.

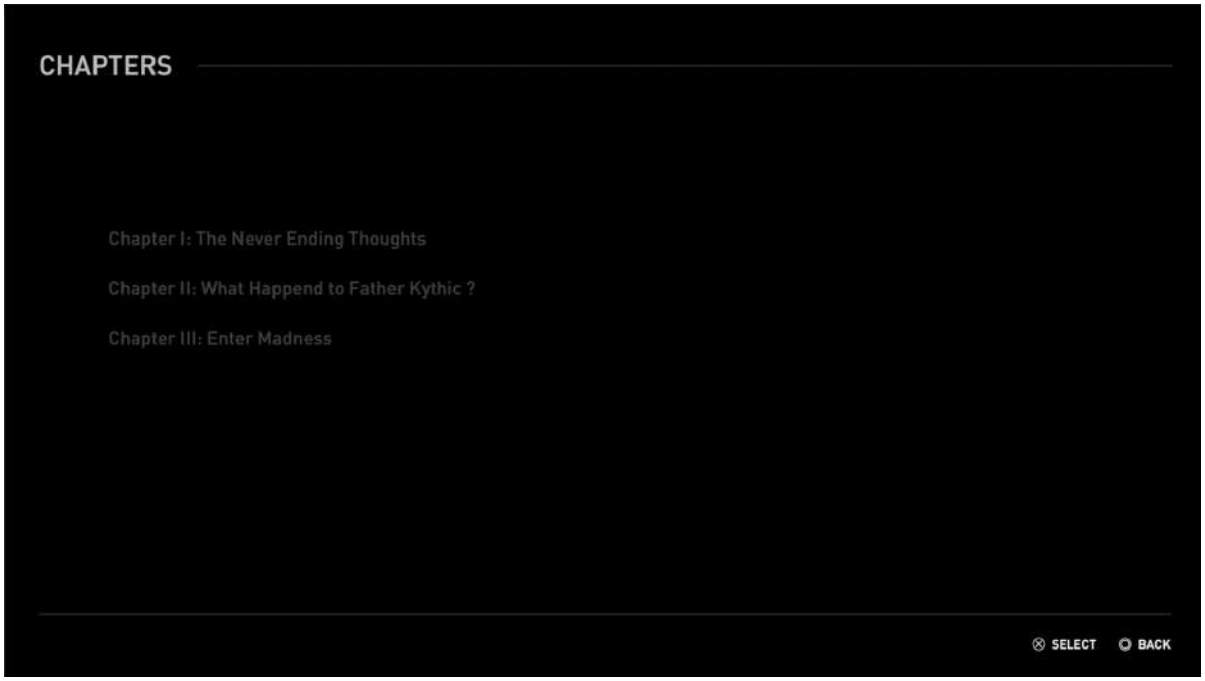

Figure 4.17: The Chapters Menu Screen.

4.8.2.2 Options Menu Screen

In the Options Menu Screen, as seen in the figure 4.18, the player is able to access the following properties:

- Display;
- Graphic Settings;
- Audio;
- Controls.

The Controls screen's only purpose is to provide information about the controls to the player, despite being located in the Options Menu Screen.

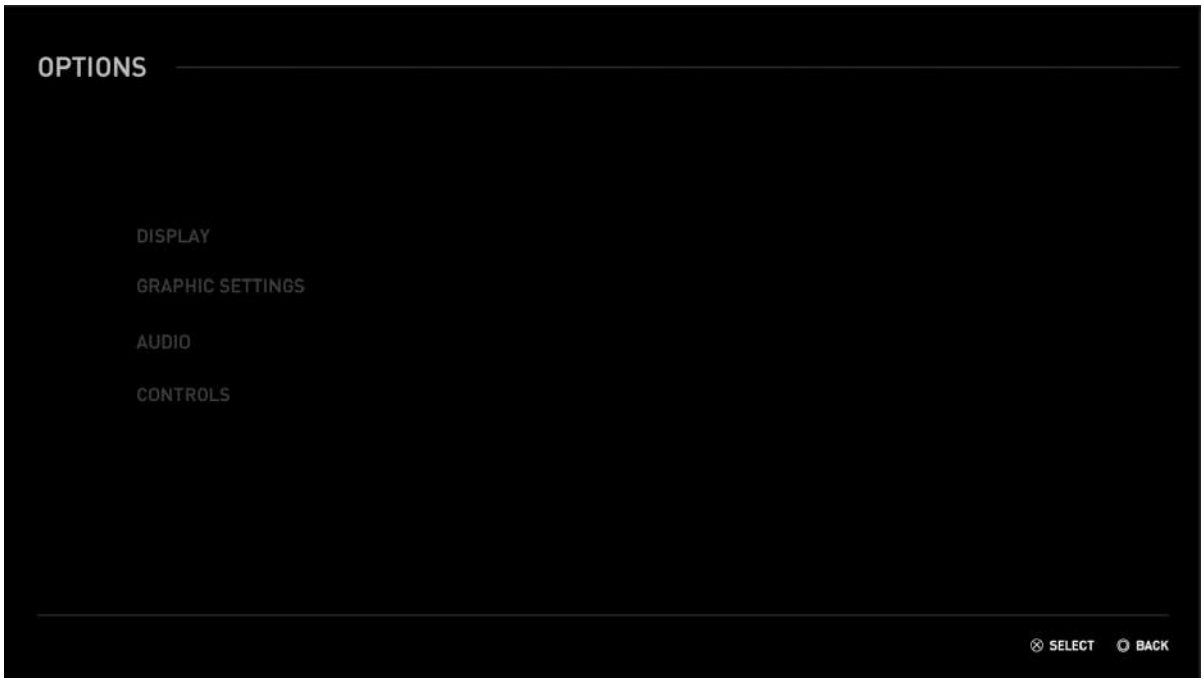

Figure 4.18: The Options Menu Screen.

In the **Display** settings (figure 4.19), the player can change the following options to manipulate the display values:

- **Resolution** – Adjusts the game resolution;
- **Display Mode** – Changes between Fullscreen or Windowed mode;
- **Brightness** – Adjusts the overall brightness of the in-game image;
- **Vertical Sync** – Toggles vertical sync on or off.;
- **Subtitles** – Toggles subtitles on or off;
- **Motion Blur** – Enabling this option applies a cosmetic effect where rapidly moving objects and scenery get blurred, giving the action a more cinematic feel;
- **Bloom** – Enabling this option applies a cosmetic effect where intense light sources produce fringes of light that bleed beyond edges of geometry in front of light sources;
- **Film Grain** – Enabling this option applies a cosmetic effect that simulates the random optical texture of photographic film, usually caused by small particles being present on the physical film, for a more cinematic feel;
- **Chromatic Aberration** – Enabling this option applies a cosmetic effect in which light rays passes through a lens focus at different points, depending on their wavelength. As a result, objects become distorted and blurry.

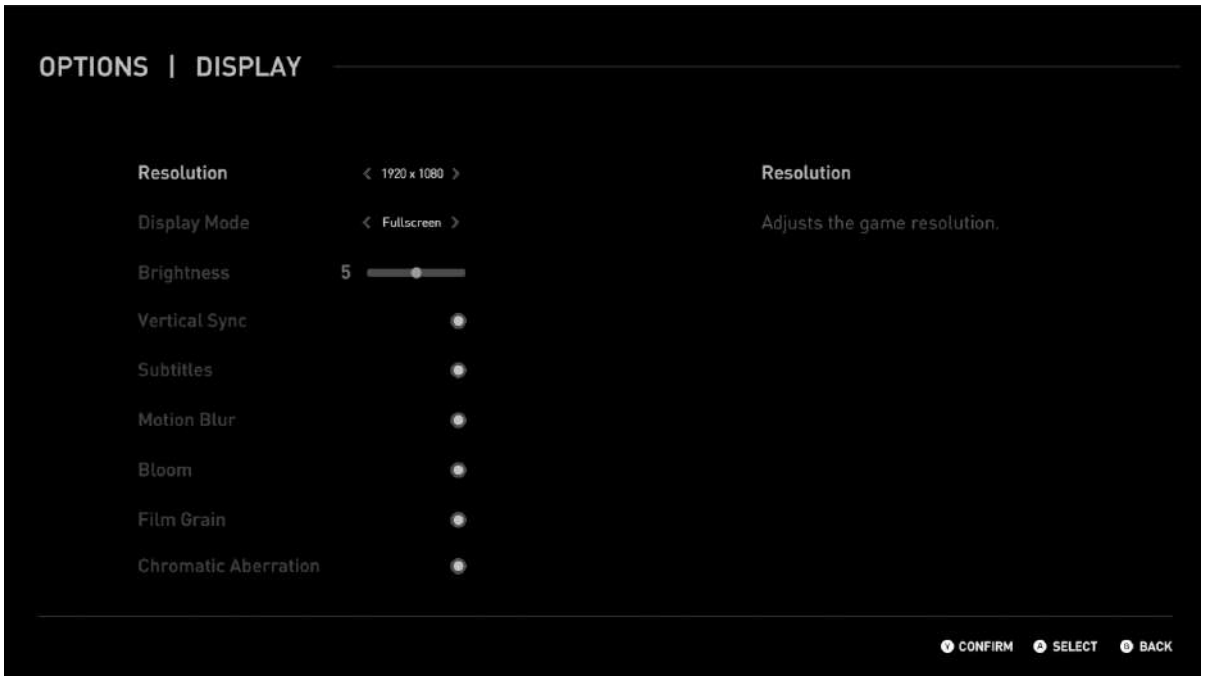

Figure 4.19: The Display settings.

Next, in the **Graphic Settings** screen (figure 4.20), the player can change the following options to manipulate the graphical values of the game:

- **Preset** – This allows for a quick modification of all settings to the selected suggestion;
- **Texture Quality** – Changes the resolution of textures used throughout the environment. A higher setting improves detail and clarity of textures but may impact performance;
- **Shadow Quality** – Changes the amount of detail of environmental shadows rendered by the engine. A higher setting may impact performance;
- **Ambient Occlusion** – The lighting and brightness of objects is rendered taking into consideration the placement of other nearby objects. Turning this option off improves performance;
- **Anti-Aliasing** – This option softens jagged edges on objects and may improve the visual experience;
- **Soft Shadows** – Soft Shadows produces shadows with a soft edge that enhances realism. Turn this option off to opt for Hard Shadows to improve performance at the cost of quality.

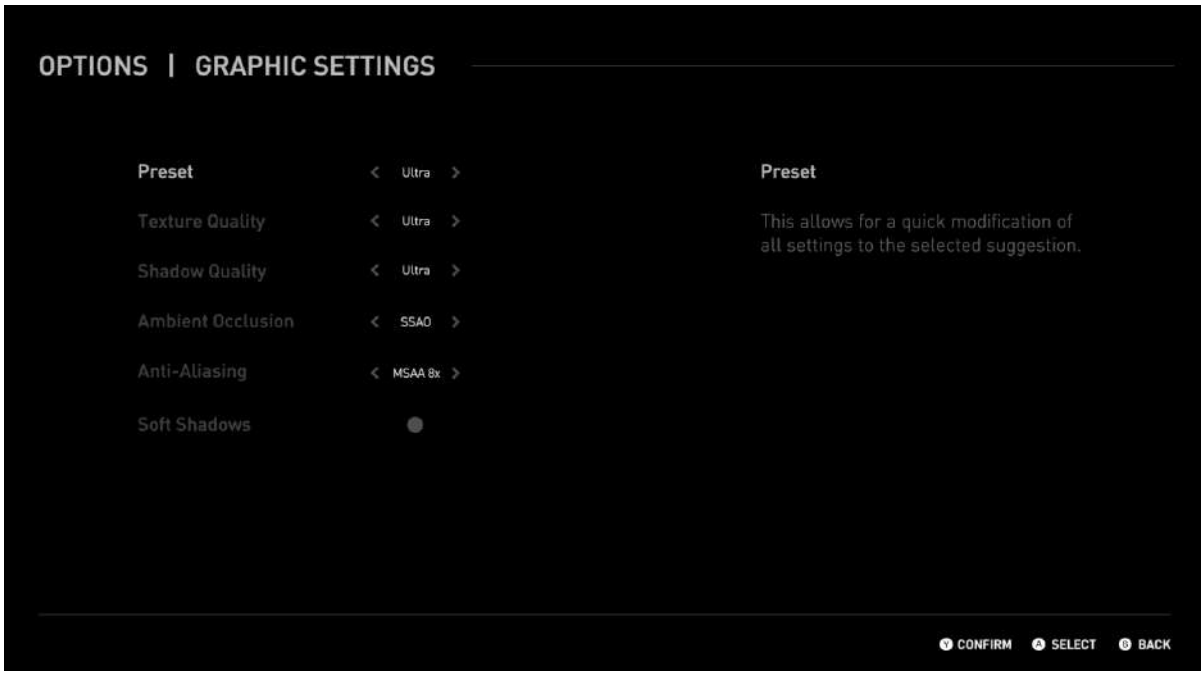

Figure 4.20: The Graphic Settings screen.

In the **Audio** settings (figure 4.21), the player can change the following options to manipulate the sound values of the game:

- **Music Volume** – Adjusts the volume of the game music;
- **SFX Volume** – Adjusts the volume of the game's sound effects;
- **Dialogue Volume** – Adjusts the volume of the game's dialogue;

Figure 4.21: The Audio settings.

Last but not least, in the **Controls** screen (figure 4.22), the player can view the controls for different devices:

- DualShock4;
- Xbox Controller;
- Keyboard & Mouse.

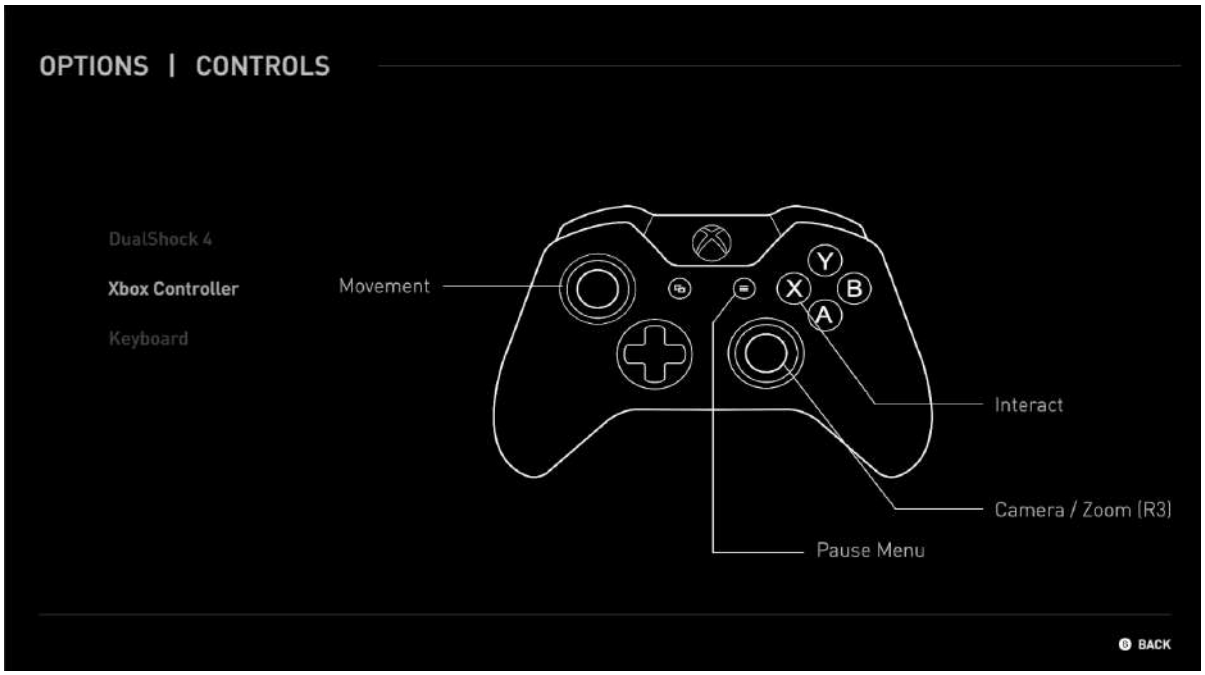

Figure 4.22: The Controls screen.

4.8.3 Loading Screen

The loading screen's sole purpose is to present visual feedback to let the players know that something is happening within the game, this being the loading of the game. In many games, the loading screens also contain tips or tricks (for gameplay purposes) for the players to read while the game loads, but in the case of VoDS, no such tips or tricks are needed since the game is extremely simple to play. Therefore, the loading screen contains only visual feedback, in a form of pulsating circles, to let the player know the game is still loading, as seen in figure 4.23.

Figure 4.23: The Loading Screen.

4.8.4 Pause Menu Screen

The Pause Menu Screen can only be accessible, although at any time, once the user starts to play the game, as seen in figure 4.24. The Pause Menu Screen contains the same settings as the Options Menu Screen (4.8.2.2), excluding the Graphic Settings screen, which can only be accessible on the Main Menu Screen.

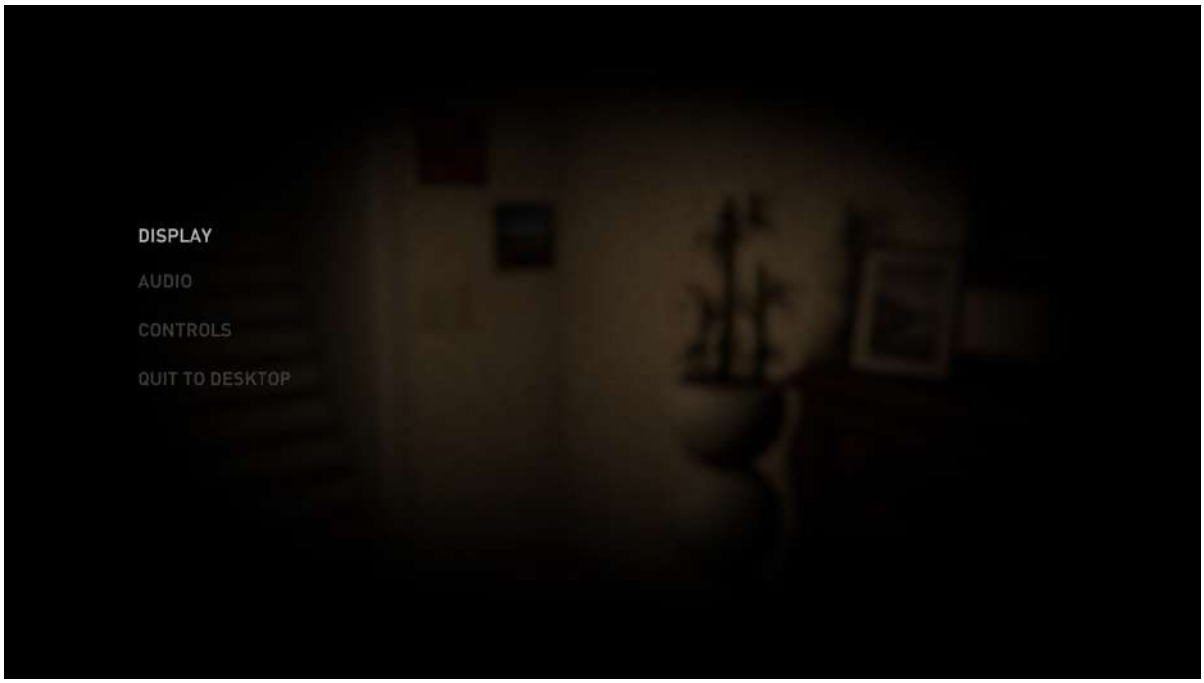

Figure 4.24: The Pause Menu Screen.

4.9 Level Design

After the main story, all the mechanics, and the overall aesthetics have been designed, the next step is to build the level design of a game. According to Schell, the level design aims to arrange the architecture, props, and challenges in a game in ways that are fun and interesting (Schell, 2008, p. 343). That said, the overall level design process is to create interactive events within the game environment to challenge the player and keep them engaged. How? By bringing together all the elements of the game to shape the player experience, such as the mechanics, the story, and the aesthetics elements, incorporating them with the technology element which refers to the medium that allows the developers to give life to the game.

Since the main mechanic in VoDS centers around the various loops the player experiences in the haunted house, the level design cannot be too big, otherwise, the user might feel lost and confused about where to go. Therefore, the level design must consist of a small portion of a house (hallways and a living room) that the player is able to explore. Taking inspiration from *P.T.*, the level design of VoDS is comprised of an L-shaped hallway, with the addition of a small living room and a staircase leading to a door that progresses to the next loop. The figure 4.25 represents the initial concept of level design for the game. After a full breakdown of the concept and experimenting various scenarios, the achieved level design of the game is represented in figure 4.26.

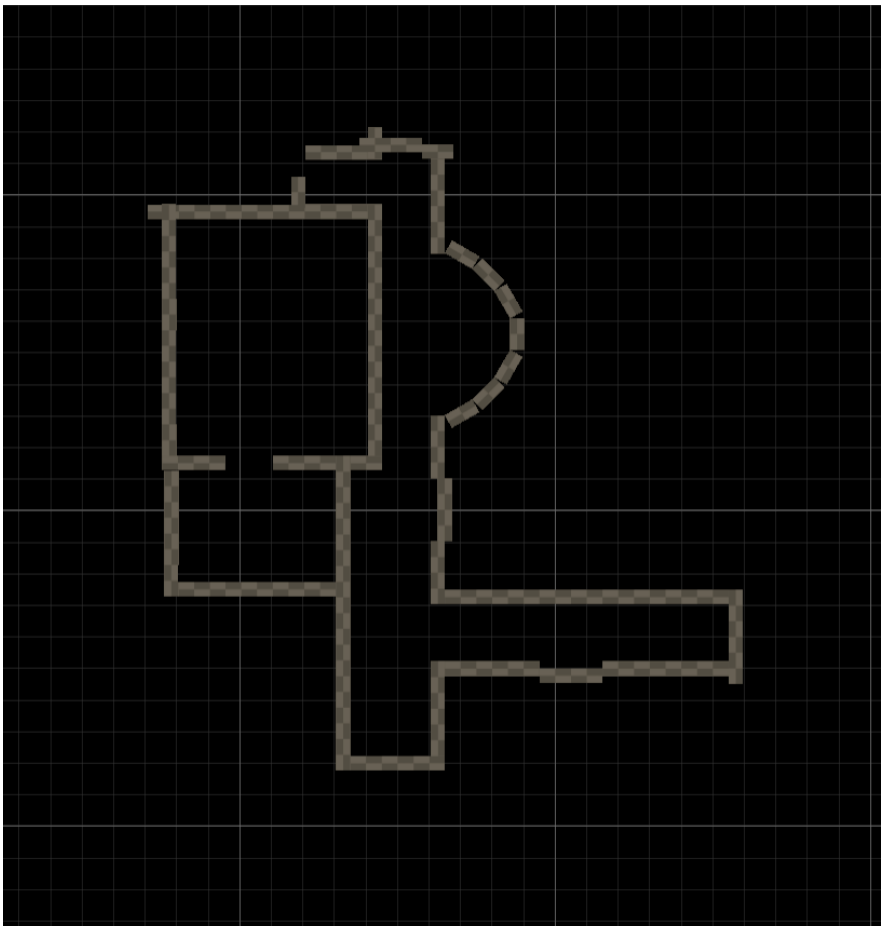

Figure 4.25: Initial concept of the level design.

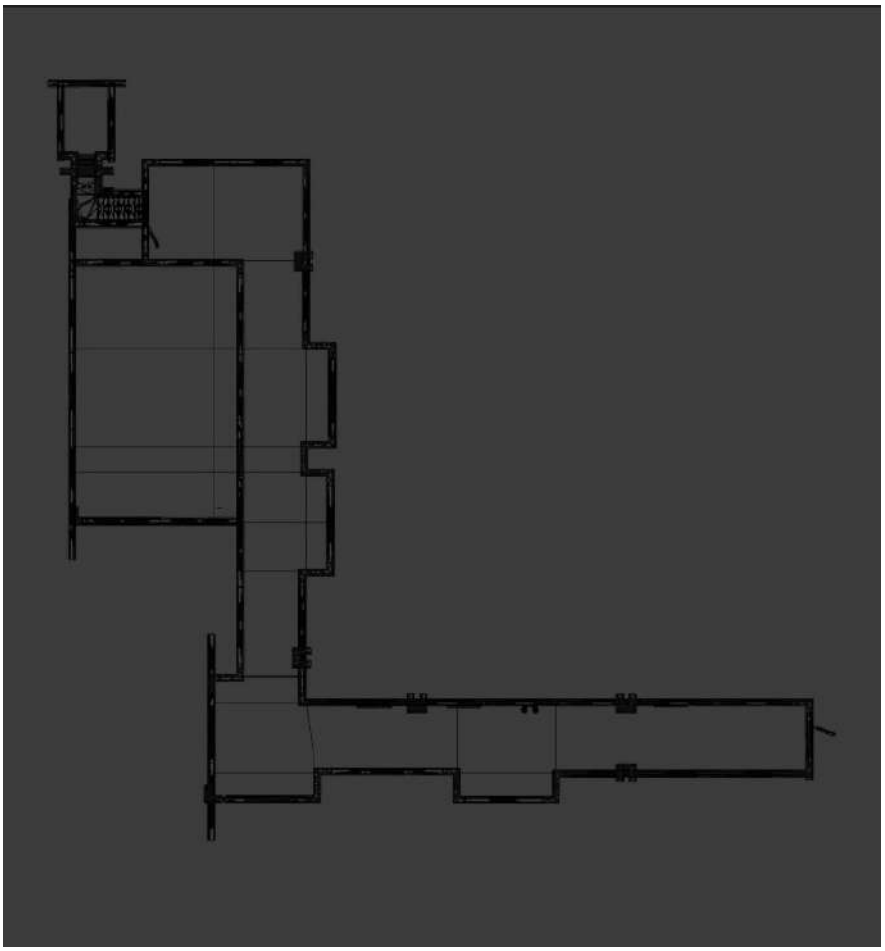

Figure 4.26: The achieved level design for the game.

The game contains a total of eleven loops, each one having different events and ways for the player to progress, as well as some changes to the environment. On some loops, the player can freely progress to the next loops, but on others, the player must find a way to progress. To control this behavior, before the staircase there's a door that has two meanings:

- **Door is open** – If the door is open, the player can advance to the next loop;
- **Door is closed** – If the door is closed, the player must find a way to unlock the door.

On the very first loop, the staircase door is open so that the player knows whenever that door is closed, he must find a way to unlock it. Describing them briefly, the eleven loops consist of:

- **Loop One** – Only the hallways are allowed for exploration – A radio is broadcasting some local news. The staircase door is open;
- **Loop Two** – Only the hallways are allowed for exploration – The staircase door is closed. When the player passes by the living room door, it starts to bang for ten seconds. After going back to the beginning of the hallway, the staircase door opens;

- **Loop Three** – Only the hallways are allowed for exploration – When the player passes by the living room door, it bangs loudly three times, followed by a sound of a woman crying. An eerie soundtrack is played. The staircase door is open;

- **Loop Four** – Only the hallways are allowed for exploration – In the middle of the hallway there is a ghostly apparition, slightly lit up by the hallway lights, emitting a crying sound. Upon reaching a certain point, the lights are turned off. When the player gets close to the apparition, the lights turn on and the ghost disappears. The staircase door is open;

- **Loop Five** – The entirety of the loop is allowed for exploration – At the beginning of this loop, some strange sounds of a woman gasping can be heard behind the player. When the player reaches by the staircase door, it slowly closes. Next, passing by the living room door, it opens on itself. To unlock the staircase door, the player must listen to a cassette tape located in the living room;

- **Loop Six** – The entirety of the loop is allowed for exploration – Upon reaching this loop, a different ominous and eerie soundtrack is played, lasting for a few minutes. There are changes in the environment lights. The living room door is open, but the staircase door is closed. A flashing flashlight is located in the living room and the player can notice it, by passing by the living room door. When he picks it up, the door closes. When the player tries to open the door an event happens. The TV starts to broadcast a video. Once the video ends, both the living room door and the staircase door open. With the use of the flashlight the player can illuminate his path for the upcoming loops;

- **Loop Seven** – Only the hallways are allowed for exploration – There are strong significant changes in the environment. Sounds of a baby crying and gore are heard coming from the living room. The radio is broadcasting ominous static sounds. The staircase door is open;

- **Loop Eight** – Only the hallways are allowed for exploration. – The environment is the same as Loop Seven. At the beginning of this loop, some strange sounds of a woman gasping can be heard behind the player. Passing by the radio, the player hears a voice coming from it talking to him. The staircase door is open;

- **Loop Nine** – Only the hallways are allowed for exploration – The environment is the same as Loop Seven. There is an ominous presence looking at the player just before the staircase door. Upon reaching further, the paranormal presence disappears, followed by eerie sounds. The staircase door is open. Upon reaching the top of the stairs, a window falls from above.

- **Loop Ten** – The entirety of the loop is allowed for exploration – The environment is the same as Loop Seven. A radio is broadcasting some local news but with strange sounds overlapping it. The staircase door is closed. To open the door, the player

must find the ghost, located at the beginning of the loop. A different ominous soundtrack is played during this loop.

- **Loop Eleven** – The entirety of the loop is allowed for exploration – The environment is the same as Loop One. Once the player reaches the final loop, a phone starts to ring. The staircase door is closed. Passing by the living room door, it opens. When the player picks up the phone, a voice speaks briefly. The staircase door opens. When the player exits the loop, the end credits begin to play.

4.10 Soundtrack, Sound Effects, and Voice Overs

The soundtracks and sound effects help to give life to all video games. Just like in films and TV Series, the soundtracks and sound effects help to emphasize the scenarios and feelings when playing a game, especially in the horror genre where the sounds are a top priority.

In VoDS, there are a variety of soundtracks, sound effects, and voiceovers. To give a more realistic feel, the sound effects and voiceovers are 3D sounds. This means the sound volume varies according to the distance between the player and the source of the sounds. The soundtracks are, however, 2D sounds since there is no necessity to vary its volume throughout the gameplay.

The following sounds present in VoDS are:

- **Walking Cycle** - This sound is played whenever the player moves the avatar;
- **Door Locked** - This sound is played whenever the player tries to open a locked door;
- **Lamp Noises** - This sound is played whenever the player passes by some wall lamps;
- **Old Ventilator** - This sound is played whenever the player passes by an old ventilator turned on;
- **Loop Door** - This sound is played whenever the player opens the door that leads to the next loop;
- **Banging Door** - This sound is played in Loop Two;
- **Rapid Door Banging** - This sound is played in Loop Three;
- **Woman Crying** - This sound is played in loops three and four;
- **Woman Laughing** - This sound is played in loops six and ten;
- **Woman Gasping** - This sound is played in loops five and eight;
- **Opening Doors & Closing Doors** - This sound is played in various events throughout the loops;
- **Radio Static Noise** - This sound is played in loop seven;
- **Flashlight On & Off** - This sound is played when the player picks the flashlight and the game turns off the flashlight, respectively;
- **Lamp On & Off Noises** - This sound is played in various events throughout the loops;
- **Baby Crying** - This sound is played in loop seven;

- **Gore** - This sound is played in loop seven;
- **Glass Breaking** - This sound is played in loop nine;
- **UI** - This sound is played when the player navigates in the main menu or pause menu;
- **Radio, Cassette, and Phone Dialogue** - These sounds are played in loops one, eight, and ten, loop five, and loop eleven, respectively;
- **Soundtracks** - Soundtracks are played throughout the loops;

All the sounds mentioned above are royalty-free.

Chapter 5

Development of Victims of Dead Stories

5.1 The High Definition Render Pipeline

As mentioned before in 3.1.1, the HDRP allows the developers to access cutting-edge real-time 3D rendering technology designed to deliver high-fidelity graphics. Since the graphics style of VoDS and the main purpose of this project is to develop a game with high-fidelity graphics, it is indispensable to achieve the best graphics possible and for that, the HDRP that Unity offers helped greatly to guide us to our main objective.

5.1.1 Installation

After creating a 3D blank project in Unity, the next step was to set up the pipeline within the project. For that, it was needed to head to the package manager and install the HDRP.

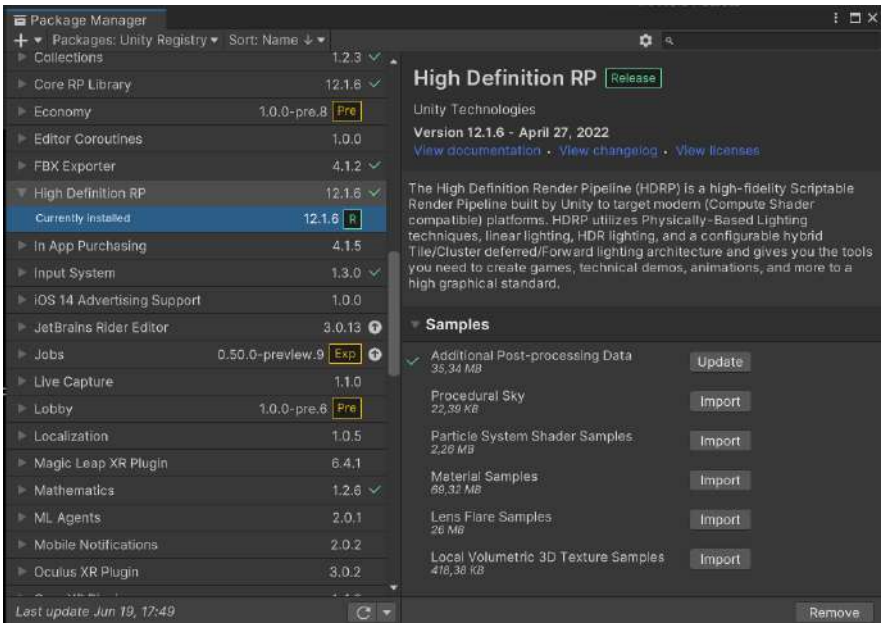

Figure 5.1: The HDRP in Unity's package manager.

This pipeline also offers some samples available to import, as seen in figure 5.1. The "Additional Post-processing Data" sample was installed because it offers various post-processing filters and elements to improve the overall visuals of the game.

5.2 Integrating the Input System

In VoDS, the player can play using two types of devices, the gamepad or the keyboard & mouse (4.7). Unity has a package capable to handle what type of controls are meant to be triggered, called Input Manager. The Input Manager uses the following types of controls:

- Key, which refers to any key on a physical keyboard, such as W, Shift, or the space bar.
- Button, which refers to any button on a physical controller (for example, gamepads), such as the X button on the remote control.

After some investigation and testing, the Input Manager was deemed as not an ideal package to configure the designed controls for this project, simply because it was not intuitive to establish and manage the controls for the two devices, along with its illogical behavior.

Discarding the Input Manager, the Input System was regarded as the ideal and perfect way to configure and set up all the controls with different actions maps for different control schemes. To use the Input System, the package needed to be installed through the package manager, as seen in figure 5.2.

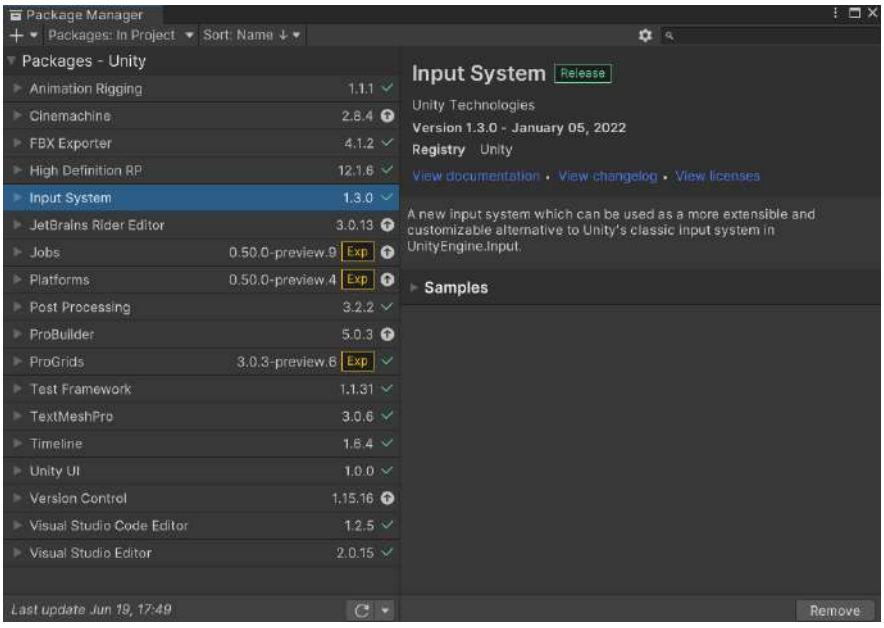

Figure 5.2: The Input System in Unity's package manager.

5.2.1 Configuring the Controls

To begin the mapping of the controls, an Input Action asset was created entitled *InputMaster*. In this asset, illustrated in figure 5.3, three action maps were created to configure controls for three different scenarios:

- **The *MainMenu_UI*** – To configure the controls capable to be recognizable inside the Main Menu Screen;

- **The *Player_Controller*** – To configure which controls trigger the avatar and camera's movement as well as some game mechanics;
- **The *Examine_System*** – To configure the controls when the player is in the Examine State.

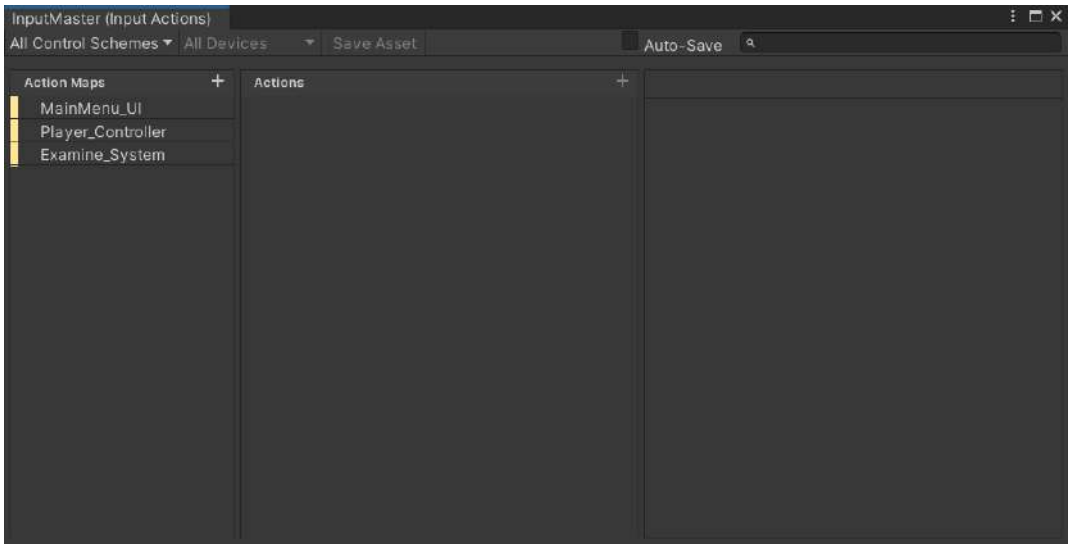

Figure 5.3: The three action maps created.

With the three action maps created, the organization and flow management of the controls was already subpar to that of the Input Manager. With the action maps created, it was possible to create and configure the actions for each map, following the previously designed controls. For example, in the *Player_Controller* map, the actions are presented in figure 5.4.

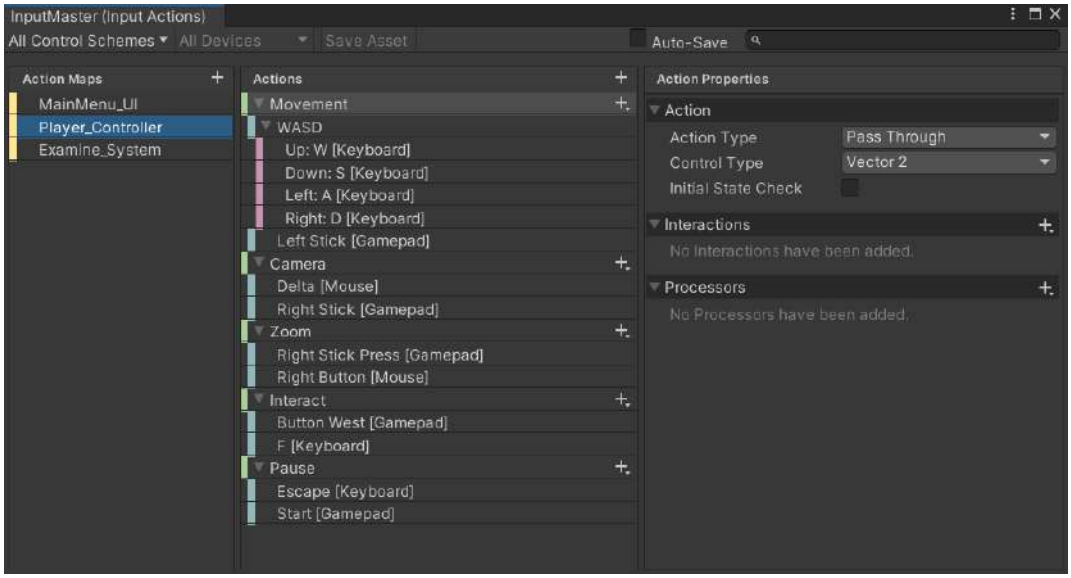

Figure 5.4: Actions for the *Player_Controller*.

In this action map, there are two types of actions:

- The **Pass Through** – This means that the action will not explicitly start and will never cancel. Instead, for every value change on any bound control, the action will perform;
- The **Button** – This means that the action will start when a button is pressed and perform when the press threshold is reached. Also, if a button is already pressed when the action is enabled, the button has to be released first.

The Pass Through action type is the most indicated type to handle the avatar and camera's movement, since those actions are performed under Vector 2, meaning that their (x, y) values range between -1 and 1 to determine the precision of their position and movement speed.

As for the Button action type, this is the preferred type when a mechanic must be triggered **only when** a button is **pressed**, like the Pause action or the Interact action, or when a mechanic must be triggered **only when** a button is **pressed and held**, like the Zoom action. The Button action type accepts two states, `true` when a button is triggered and `false` when a button is not triggered.

As soon as the actions for the actions maps were configured, a C# *Player_Behavior* was created to handle the behavior of these actions. The code sample in 5.1 is an exemplification of the methods to manage the behaviors of the actions.

```
public static Vector2 Movement_values;
public static bool isInteractable;
controls = new InputMaster();

//Movement callback
controls.Player_Controller.Movement.performed += context => WASD(context);

//Interactable Callback
controls.Player_Controller.Interact.performed += context => Interactable(
    context);
controls.Player_Controller.Interact.canceled += context =>
    Interactable_Canceled(context);
...

public void WASD(InputAction.CallbackContext Context)
{
    Movement_values = Context.ReadValue<Vector2>();
}

public void Interactable(InputAction.CallbackContext Context)
{
    isInteractable = true;
}

public void Interactable_Canceled(InputAction.CallbackContext Context)
{
```

```
    isInteractable = false;
}
...
```

Code Sample 5.1: Implementing callbacks for each action.

Since the Movement action is of type Pass Through under a Vector 2, a variable of type *Vector2* was created to read and store the (x, y) values, as seen in the function *WASD(...)*.

As for the Interactable action, of type Button, a variable of type *bool* was created to state the behavior of the button, as seen in functions *Interactable(...)* and *Interactable_Canceled(...)*. This type of logic behavior is the same for the actions in the *MainMenu_UI* and the *Examine_System* action maps.

5.3 The Foundations of the Game

5.3.1 HDRP Assets

Firstly, before creating any scenes in Unity, it was important to create HDRP assets. These assets control the global rendering settings of the project and create an instance of the render pipeline. They offer a large variety of graphics settings for the developer to change as necessary. For this project, three assets were created with different quality levels: low, medium, and high quality.

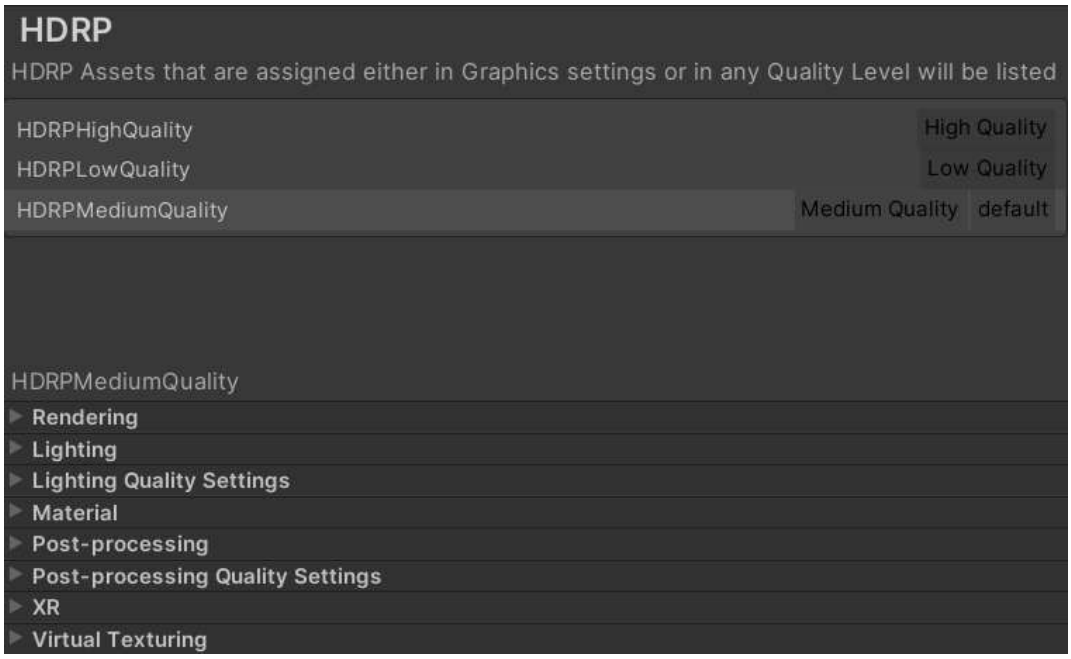

Figure 5.5: The created HDRP assets.

As seen in figure 5.5, the *HDRPMediumQuality* is set as default. This means that while developing the game and after its build and release, the graphics quality will be of medium quality, which is a balanced graphics quality. Having different quality level assets, enabled us to further test the performance and optimization of the game while maintaining the

purpose of achieving high-fidelity graphics, and implementing some options for the player to tweak under his machine's specifications, that are further discussed in section 5.4.4.

5.3.2 Scene Handling

In VoDS there are three main subjects:

- The Main Menu Screen – Where the player can start the game, as well as access some game settings;
- The Loading Screen – Where the game loads the first level;
- The Gameplay – Where all the game events and mechanics happen.

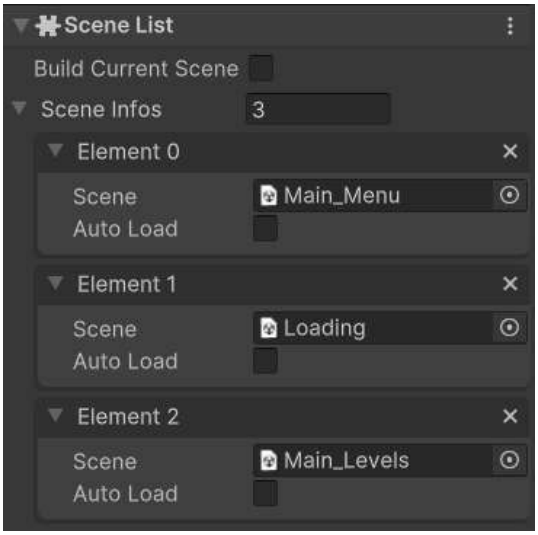

Figure 5.6: List of scenes.

In Unity, a scene is where developers work with content, where it can contain the entire game or just part of it, in case it is too complex. Since those three main subjects are the main core of the game, it is logical to create three scenes, as seen in figure (5.6. When playing the game, each scene is loaded in order, this means that upon launching the game, it goes to the Main Menu Screen, followed by the Loading Screen when the player starts the game, followed by the Gameplay after the game finishes the loading process.

The method to load a scene when needed is possible by instructing the game to do it via script. For example, in a script called *Menu_Check*, where the main menu behavior is managed, a function called *StartGame()* is responsible to load the "Loading" scene to begin the loading of the game.

```
...
SceneManager.LoadSceneAsync("Loading", LoadSceneMode.Single);
...
```

Code Sample 5.2: Loading a scene.

5.3.3 The Character and the Camera

Since VoDS is a game with a first-person perspective, the game camera will be leveled at the eyes of the character. As stated in section 4.4, the avatar was chosen to not be visible to the player for more immersion, but a reference to the avatar was still needed to implement its related mechanics as well as to check collision with the environment.

The avatar will not be affected by complex physics and other forces in the game and for that, it contains a ”Character Controller” component, as seen in figure 5.7. This component allowed us to easily do movement constrained by collisions and it only moves when prompted by calling the *Move()* function.

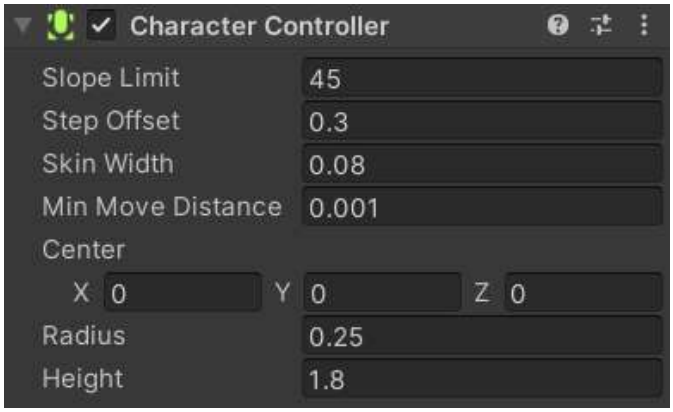

Figure 5.7: Character Controller Component.

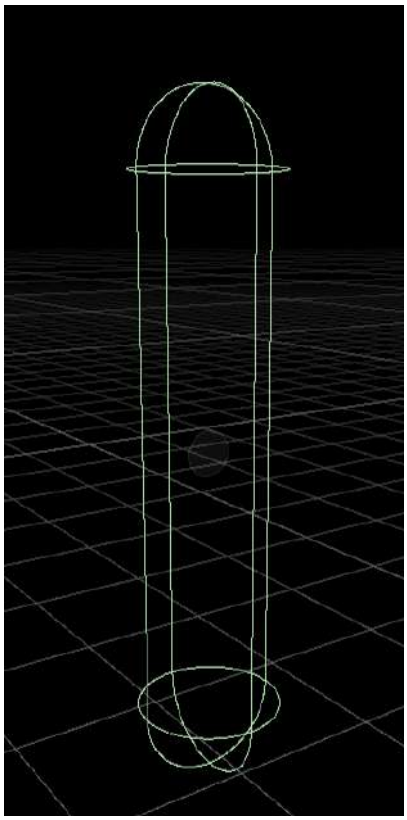

Figure 5.8: The avatar’s collider in a cylinder shape.

Considering that the avatar's collider is essentially its body (figure 5.8), the camera was placed in the upper part of the collider, thus simulating a first-person perspective (figure 5.9).

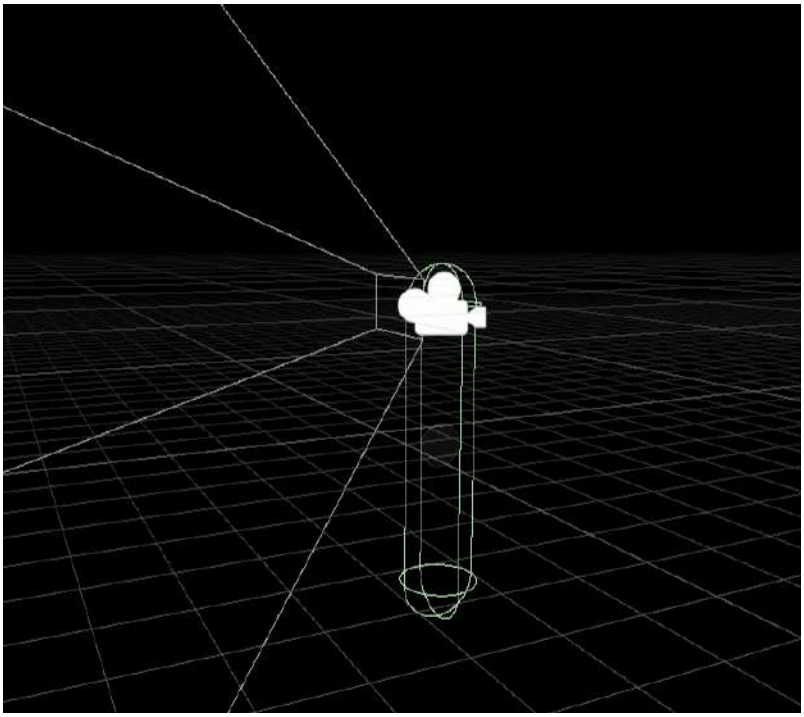

Figure 5.9: The camera relatively placed at the eye level of the avatar.

After creating the avatar and placing the camera in the right position, the implementation of the player's movement was possible but first, the camera had to be attached to the avatar, meaning that no matter where the player moves the avatar, the camera is sure to follow. This simple problem was solved by making the camera object as a child of the avatar, as seen in figure 5.10.

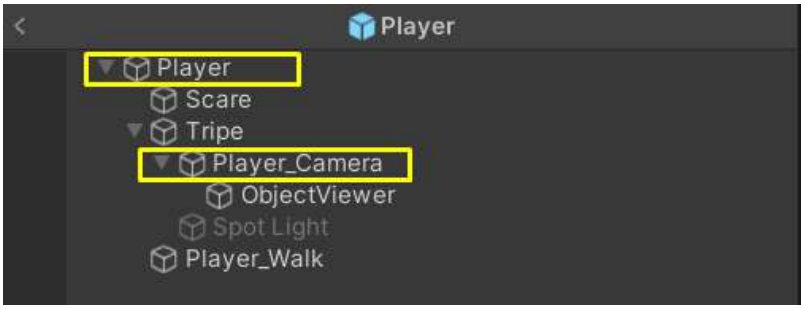

Figure 5.10: The "Player_Camera" object being a child of the "Player" object.

After guaranteeing that the camera would follow the player's movement, defining that mechanic was simple. The purpose of the script called *Player_Movement* is to store the (x, y) values and apply them on the *Move()* function from the "Character Controller" component.

```
...
public float gravity = -9.81f;
public CharacterController playerController;
public float speed = 12f;
Vector3 velocity;

void PlayerMovement()
{
    float x = Player_Behaviour.Movement_values.x;
    float z = Player_Behaviour.Movement_values.y;

    Vector3 move = transform.right * x + transform.forward * z;

    playerController.Move(move * speed * Time.deltaTime);

    velocity.y += gravity * Time.deltaTime;

    playerController.Move(velocity * Time.deltaTime);
    ...
}
...
```

Code Sample 5.3: Implementing the movement for the player.

In the function called *PlayerMovement()* (code sample 5.3), two variables of type *float* were declared to store the (x, y) values. Those variables were then used on a variable *move* of type *Vector3* to apply those values in a 3D environment. Having established the behavior of those variables, the *Move()* function was called, taking one argument, where the *move* variable was multiplied by the speed of the player and the interval in seconds from the last frame to the current one (*Time.deltaTime*).

With this piece of code the avatar movement was possible, however, the player would float in the air which was not desirable. To fix this issue, gravity was needed to ensure that the player was grounded at all times. To achieve this solution, the avatar's y position equals to its value plus the value of the gravity multiplied by *Time.deltaTime*. The result of this value was then applied to the *Move()* function, taking one argument, where that value was also multiplied by *Time.deltaTime*.

The way the camera movement is managed is similar to how the player movement works.

```
...
public float mouseSense = 100f;
public Transform playerBody;
float RotationX = 0f;

void FirstPersonCamera()
{
    float mouseX = Player_Behaviour.Camera_values.x * mouseSense * Time.
```

```
        deltaTime;
    float mouseY = Player_Behaviour.Camera_values.y * mouseSense * Time.
        deltaTime;

    RotationX -= mouseY;
    RotationX = Mathf.Clamp(RotationX, -90f, 90f);

    transform.localRotation = Quaternion.Euler(RotationX, 0f, 0f);
    playerBody.Rotate(Vector3.up * mouseX);
}
...
```

Code Sample 5.4: Implementing the movement for the camera.

In a script called *Player_Camera*, the function *FirstPersonCamera()* was created, as seen in code sample 5.4. In it, two *float* variables were declared to store the (x, y) values multiplied by the *mouseSense* variable and *Time.deltaTime*. It was then limited the amount of tilting possible from the camera by locking the rotating at a 90º degrees angle with a *Mathf.Clamp()*, followed by applying the rotation with *transform.localRotation* of the camera. As it was described in 4.5, the camera pans at a 360º degrees angle, therefore the camera will need to rotate the avatar's body to position it according to the camera's movements. That effect was achieved by applying a rotation to the avatar, this being the *playerBody*, with the camera's x value.

The camera zoom is a mechanic that simulates a zooming effect, therefore it was wise to implement this mechanic inside the camera's script because we can get access to all camera values.

To develop the camera zoom, a function *Zoom()* (code sample 5.5) was created to execute the mechanic. Inside that function, a variable *angle* of type *float* was created to store an exact value of the camera's angle. This allowed the camera to maintain its angle when performing the zooming effect. The *playerCam.fieldOfView* is the property that enabled us to change the current FOV value of 60 to a target one, this being the zoom value of 40.

```
...
void Zooms(float target)
{
    float angle = Mathf.Abs((defaultFOV / 2) - defaultFOV);

    playerCam.fieldOfView = Mathf.MoveTowards(playerCam.fieldOfView, target,
        angle / zoomDuration * Time.deltaTime);
}
...
```

Code Sample 5.5: Implementing the zoom mechanic.

As soon as the related camera mechanics were implemented, the next step was to adjust some of the properties in its component. The Projection and the Rendering are the two main properties that were tweaked to obtain the stipulated camera results and looks.

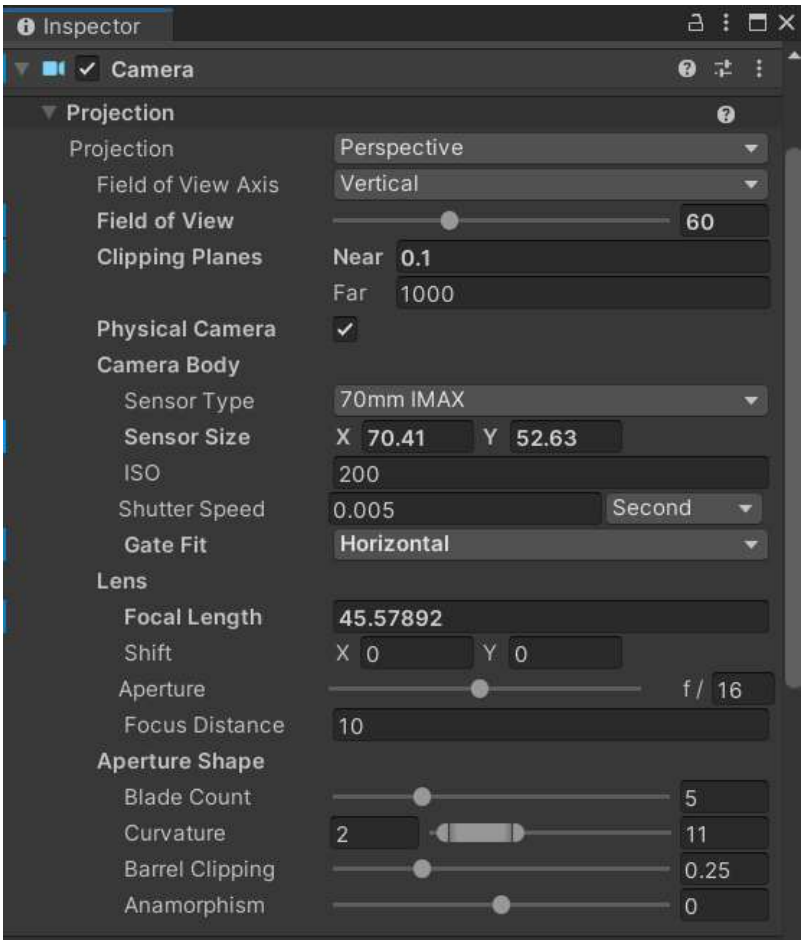

Figure 5.11: The Projection property in the Camera component.

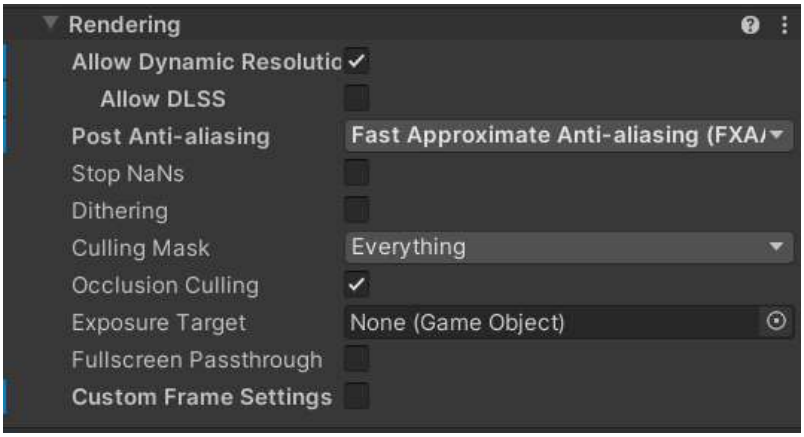

Figure 5.12: The Rendering property in the Camera component.

In the Projection property (figure 5.11), there were three settings that needed to be regulated, such as:

- **Field of View** – This setting represents the camera's view angle, measured in degrees along the axis specified in the FOV Axis. The default FOV of the game has the value of 60.

- **Clipping Planes** – This setting represents the distances from the camera to start and stop rendering. The Near value represents the closest point relative to the camera where the drawing will occur while the Far value represents the furthest point relative to the camera where the drawing will occur. Upon testing the level design by controlling the avatar, these values needed to be tweaked to achieve the estimated design results, as such the Near value was changed from its default value of 0.3 to 0.1 in order to render objects more close to the camera;
- **Sensor Type** – It specifies the real-world camera format for the camera to simulate. Among various options from a list that Unity provides, the 70mm IMAX camera was chosen;

In the Rendering property (figure 5.12), there three settings that needed to be enabled, such as:

- **Allow Dynamic Resolution** – The HDRP asset provides a dynamic resolution option that reduces the workload on the GPU and maintains a stable target frame rate by lowering the resolution of the render targets that the main rendering passes use. Since the graphics in VoDS are of high-fidelity, this option is excellent to improve performance.
- **Post Anti-Aliasing** – Anti-aliasing is the smoothing of jagged edges in digital images by averaging the colors of the pixels at a boundary. Unity offers various qualities of Anti-Aliasing, but the lower quality was chosen while developing the game to improve the workflow.
- **Occlusion Culling** – The occlusion culling means that objects that are hidden behind other objects are not rendered, for example, if they are behind walls. Enabling this option was perfect to greatly improve performance while playing the game.

More details about performance and optimization of the Rendering properties of the camera are discussed in chapter 6.

5.4 The Main Menu and User Interfaces

The Main Menu Screen is where the player can start the game through the Chapters Menu Screen and access the Options Menu Screen to change some display, graphics, or audio settings at will. But first, the main menu had to be built so that the player could navigate through it, either with a gamepad or a keyboard & mouse.

For this project, the Unity UI package, which is installed by default, was used to create the entire UI of the game. This package contains various components that were used to make an essential and coherent UI throughout the game, such as:

- **Interaction Components**

    - Button;

- Slider;

- Toggle.

- **Visual Components**

  - Text;

  - Image.

The Interaction Components are responsible to handle the interactions, in the Main Menu Screen, from the player to the machine, such as a keyboard & mouse or a gamepad. These components are not visible on their own and must be combined with one or more Visual Components to work correctly. In order to integrate such components, another component was indispensable to use to visualize such behaviors, the Canvas.

With the Unity UI package also comes the Canvas component which corresponds to the area that all UI elements should be inside. This component has a Render Mode setting which can be used to make it render in three modes:

- The **Screen Space - Overlay** – This render mode places UI elements on the screen rendered on top of the scene. If the screen is resized or changes resolution, the Canvas will automatically change the size to match this.

- The **Screen Space - Camera** – It is similar to the Overlay mode but in this render mode, the Canvas is placed at a given distance in front of a specified Camera. The UI elements are rendered by this camera, which means that the Camera settings affect the appearance of the UI.

- The **World Space** – In this render mode, the Canvas will behave like any other object in the scene, making it a somewhat 3D UI.

Out of the three render modes, the Screen Space - Overlay was the one that fitted the most with the game's planned UI and given how the player is able to change the game's resolution, making the UI scalable and adaptive. But before creating the Main Menu Screen, it was important to know its flow and how each menu would connect with each other, as seen in figure 5.13.

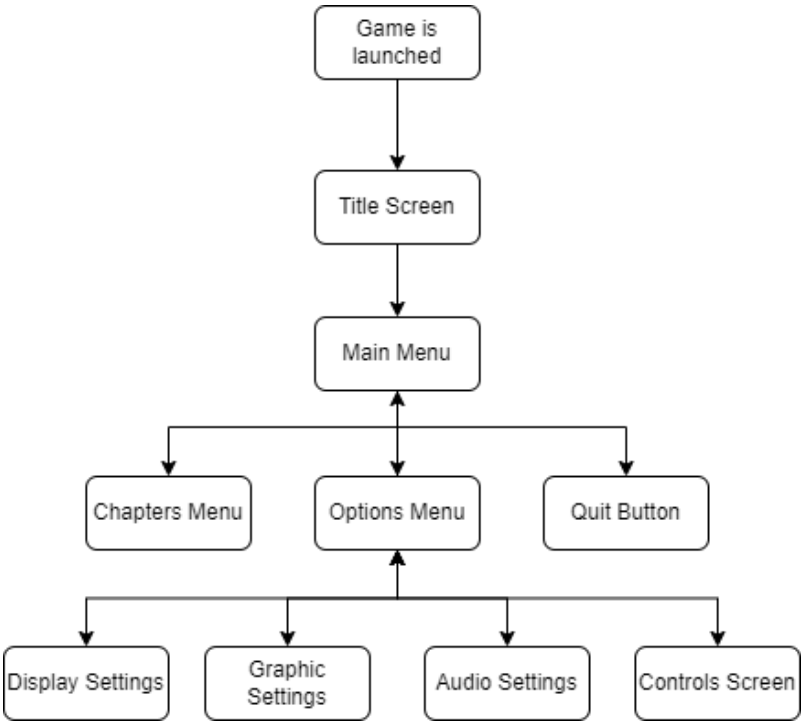

Figure 5.13: The flow of the Main Menu Screen.

5.4.1 Title Screen

As for the Title Screen behavior, it consists on showing some messages to the player, followed by displaying the game's title. As shown in figure 4.15, to enter the Main Menu the player must press any button on the gamepad or the keyboard. In the *MainMenu_UI* action map, a *Submit* action was created to be used on both control schemes to detect if the player pressed any key that is recognizable by the Input System, as seen in figure 5.14.

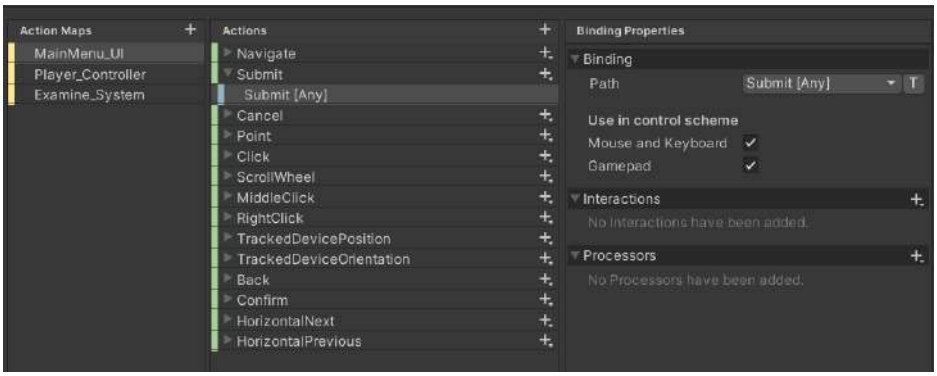

Figure 5.14: The *Submit* action.

To exit the Title Screen and proceed to the Main Menu, a script called *Title_Behavior* was created. With a *if* statement, it is possible to verify if the player presses any button on the expected devices. If that statement is true, the game will then disable the Title Screen and enables the Main Menu.

But in order to detect if the player can indeed press **ANY** button of any device, a new component must be placed within the Main Menu Screen scene to recognize such actions. The Event System is a way of sending events to objects in the application based on input, be it keyboard, mouse, touch, or custom input. The primary roles of the Event System are as follows:

- Manage which UI element is considered selected or hovered;
- Manage which Input Module is in use;
- Manage Raycasting (if required);
- Updating all Input Modules as required.

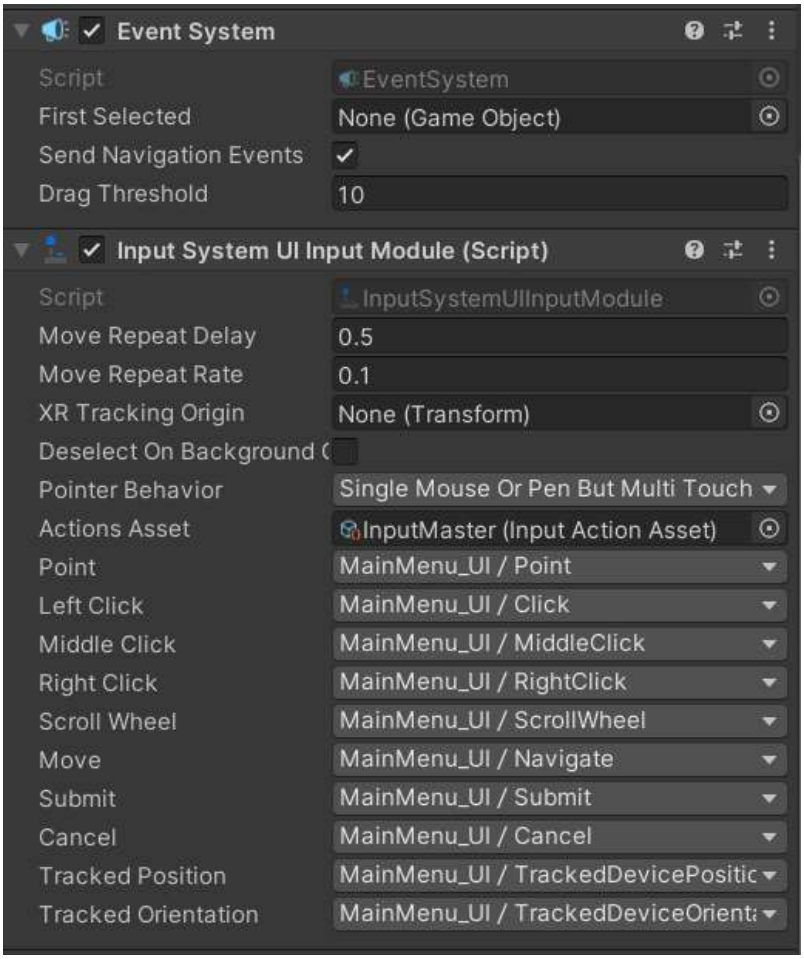

Figure 5.15: Event System Component.

As seen in figure 5.15, the input action asset previously created is added to the *Actions Asset* property in the *Input System UI Input Module*. This module is then responsible to

detect which action for the selected action map is triggered (the action map in the Main Menu Screen being the *MainMenu_UI*), meaning that this way the process for the game to register which controls the player presses is completed.

During the showcase of the Title Screen, all the controls available to the player, be it on a gamepad or a keyboard & mouse, are locked to avoid undesired actions or problems. To achieve this effect, the Event System is disabled, by disabling the *Send Navigation Events* option, as well as the mouse cursor. In the *Title_Behavior*, a reference to the respective Event System was created to define its behaviors.

```
...
EventSystem.current.sendNavigationEvents = false;
Cursor.lockState = CursorLockMode.Locked;
Cursor.visible = false;

//Showcase the title screen (messages and game's title)

EventSystem.current.sendNavigationEvents = true;
Cursor.lockState = CursorLockMode.None;
Cursor.visible = true;
...
```

Code Sample 5.6: Manipulation of the Event System behaviors.

As soon as the showcase Title Screens ends, inside the same script in 5.6, the Event System is then enabled, along with the mouse cursor.

According to the figure 5.13, whenever the game is launched, it proceeds to show the Titles Screen, followed by the Main Menu. From the Main Menu, the player can enter the Chapters Menu Screen and the Options Menu Screen and is offered the option to quit the game. To allow the player to do such actions, three buttons, accompanied by three texts, were implemented in the Main Menu.

Figure 5.16: The buttons in the Main Menu.

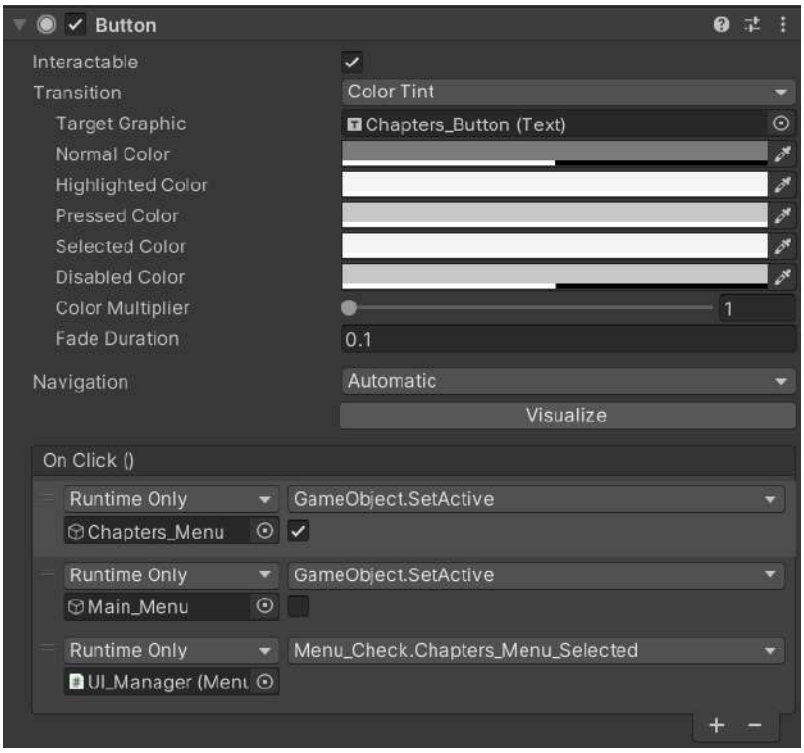

Figure 5.17: The Button component.

When two or more buttons are created, Unity automatically configures the way their navigation functions. As depicted in the figure 5.16, the yellow arrows represent the navigation that is possible between the three buttons, which is vertical. Upon entering the Main Menu, the player expects to be able to change which buttons to press.

By controlling a mouse, the player can easily choose the buttons, but not with a gamepad or a keyboard. To solve this problem, in the *Title_Behavior* script, after the player is prompted to press any key to open the Main Menu, it is told to the Event System to mark the Chapter button as the currently selected button, for the player to freely navigate through the buttons, as show in code sample 5.7.

```
...
EventSystem.current.SetSelectedGameObject(chaptersButton);
...
```

Code Sample 5.7: Defining which button is selected automatically by the Event System.

However, if the player chooses a button to press, he will expect visual feedback indicating that a button is pressable. As seen in figure 5.17, the button component has Transition elements to change the color of the buttons **text** in certain moments. To provide said visual feedback to the player, a script called *ButtonHover* was created, as shown in code sample 5.8.

```
...
 public Text text;

public void OnSelect(BaseEventData eventData)
{
    text.color = new Color(245, 245, 245);
    text.CrossFadeAlpha(0.4980f, 0f, true);
}

public void OnDeselect(BaseEventData eventData)
{
    text.color = new Color(0.4823f, 0.4823f, 0.4823f);
    text.CrossFadeAlpha(0.2509f, 0f, true);
}

public void OnPointerEnter (PointerEventData eventData)
{
    text.color = new Color(245, 245, 245);
    text.CrossFadeAlpha(0.4980f, 0f, true);
}

public void OnPointerExit(PointerEventData eventData)
{
    text.color = new Color(0.4823f, 0.4823f, 0.4823f);
    text.CrossFadeAlpha(0.2509f, 0f, true);
}
...
```

Code Sample 5.8: Functions to change text color when a button is selected.

The visual feedback is simply a color change to the text. That so, a reference to the text is needed to apply color changes. In the *ButtonHover* script, four interfaces were implemented to access its functions:

- The **ISelectHandler** and **IDeselectHandler** Interfaces:

  OnSelect() and OnDeselect() – To change the text color whenever a button is selected or deselected by the player with a gamepad or the keyboard.

- The **IPointerEnterHandler** and **IPointerExitHandler** Interfaces:

  OnPointerEnter() and OnPointerExit() – To change the text color whenever a button is selected or deselected by the player with the mouse cursor.

This button behavior is expected throughout all the selectable and navigable buttons in the game. After establishing the buttons' behaviors and their navigation, the desired outcome of the buttons when pressed was possible to develop.

5.4.2 Chapters Menu

Represented in figure 5.17, is the *OnClick()* function that determines what happens when the player presses a button, this being the Chapters button. When the player presses the Chapters button, the Main Menu is disabled and the Chapters menu is enabled. Along with these two behaviors, Unity then invokes a method called *Chapters_Menu_Selected()* in the *UI_Manager* script.

```
...
 public void Chapters_Menu_Selected()
{
    if (chaptersMenu.activeSelf)
    {
        EventSystem.current.SetSelectedGameObject(null);
        EventSystem.current.SetSelectedGameObject(chp1Button);
    }
}
...
```

Code Sample 5.9: The *Chapters_Menu_Selected()*.

This method performs the same effect present in 5.7, to mark the very first button of the Chapters Menu as selected, so the player can navigate freely with the keyboard or gamepad.

Now that the player can enter the Chapters Menu, it is expected that he could go to the previous, the Main Menu. **This type of behavior is expected throughout the entire Main Menu Scene as well as the Pause Menu**.

Another example of this is when the player is in the Graphic Settings and after changing some settings, he decides to play the game. Analyzing the figure 5.13, if the Graphic Settings is enabled, exiting it will disable it and enable the Options Menu Screen, and exiting this menu will disable it and enable the Main Menu so that the player can enter the Chapters Menu and play the game.

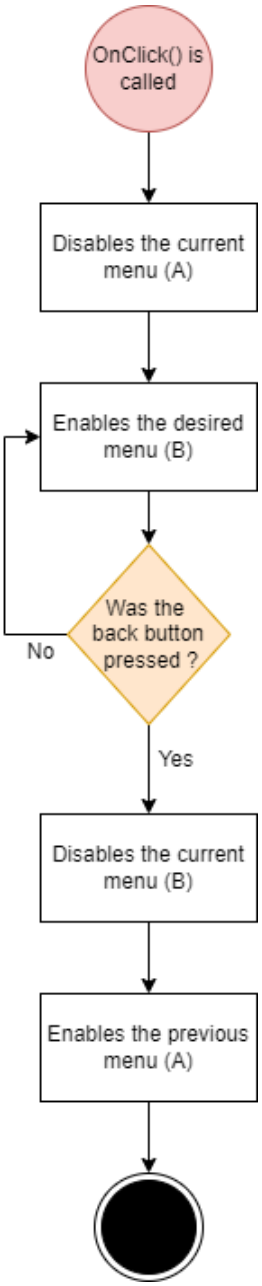

Figure 5.18: The menu navigation logic.

Following the logic in figure 5.18, to go back to a previous menu it was necessary to store which menu the player is currently in and which menu the player came from. To tackle this issue, a script called *MainMenu_ConBack* was created. In it, there are references to the current active menu and the previous menu.

```
...
public GameObject activeMenu;
public GameObject previousMenu;
...

if(Player_Behaviour.isBack)
    {
        activeMenu.SetActive(false);
        previousMenu.SetActive(true);

        Menu_Check.menu_Check.Verify_Menu(previousMenu, activeMenu);
    }
...
```

Code Sample 5.10: Going to the previous menu.

As so, whenever the player presses the back button, the game closes the active menu and enables the previous menu from which he came. Having the menu navigation problem solved, there was still an issue that was notable upon returning to a previous menu, the currently selected button should be the one the player used to enter a certain menu, before leaving it, by the Event System component, which was not.

As is expected, for example, when the player exits the Graphic Settings to the Options Menu, the Graphic Settings button should be selected by the Event System and not any other buttons. To solve this problem, the method *Verify_Menu()* is invoked in code sample 5.10. That method was created in the *Menu_Check* script.

Basically, this method also verifies which menu the player is currently in and which previous menu he came from. After said verification, this method informs the Event System to select the button that allowed the player to enter the currently active menu on the previous menu that will now be the currently active menu.

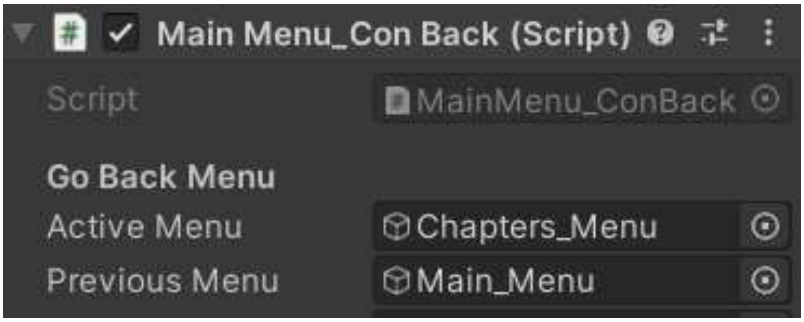

Figure 5.19: The component to allow the player to go to previous menus.

For the menu navigation logic to fully work on all the menus, the script *MainMenu_ConBack* is applied as a component to all the menus in the game, as seen as an example in figure 5.19, where the component is applied to the Chapters Menu with the respective menu references.

5.4.2.1 Loading Screen

After the player starts the game in the Chapters Menu, the game loads the first level in the Loading Screen. As it was described in the section 5.3.2, the Loading Screen is an independent scene. Within this scene, a Loading Manager was created to ensure the loading process begins and to start the game once the loading is finished.

The Loading Manager contains a script called *LoadingManager*. This script consists of unloading the Loading Screen scene and loading the gameplay scene on a list of type *AsyncOperation*, as seen in code sample 5.11.

```
...
List<AsyncOperation> scenesList = new List<AsyncOperation>();

public void LoadGame()
{
    scenesList.Add(SceneManager.UnloadSceneAsync("Loading"));
    scenesList.Add(SceneManager.LoadSceneAsync("Main_Levels", LoadSceneMode.
        Additive));

    StartCoroutine(LoadProcess());
}

IEnumerator LoadProcess()
{
    for (int i = 0; i < scenesLoading.Count; i++)
    {
        while (!scenesLoading[i].isDone)
            yield return null;

    }
}
...
```

Code Sample 5.11: Loading and Unloading Scenes.

The loading process is done with a coroutine. A coroutine allows spreading tasks across several frames. In Unity, a coroutine is a method that can pause or wait for an execution to finish and return control to Unity. After adding the respective scenes to the list, the coroutine starts. In it, a *for* cycle is used to iterate through the list to start the loading process. To verify if the loading is finished, a *while* condition is used for this purpose with the *isDone* variable from the *List<AsyncOperation>()*.

Once the loading process finishes, the first level is ready to be played.

5.4.3 Display Settings Menu

The Display menu contains nine display settings available for the player to tweak to his liking, as seen in figure 4.19.

Whenever the player navigates through each display setting, brief information is shown regarding the respective button the player has currently selected, as seen in figure 4.19. For example, if the Motion Blur setting is selected, a tooltip appears to let the player know what that setting does and how it affects the game. **This tooltip behavior is present in all the graphics settings and audio settings as well**. A script called *Show_Tooltip* was created for this effect.

```
...
if (EventSystem.current.currentSelectedGameObject == this.gameObject)
    Info_Tooltip.SetActive(true);

else
    Info_Tooltip.SetActive(false);
...
```

Code Sample 5.12: Tooltip Behavior.

In a simple *if/else* statement, as seen in code sample 5.12, it is verified on the Event System component, if the respective button is selected. If that case is true, the tooltip is shown on screen. The figure 5.20 shows the created script attached as a component.

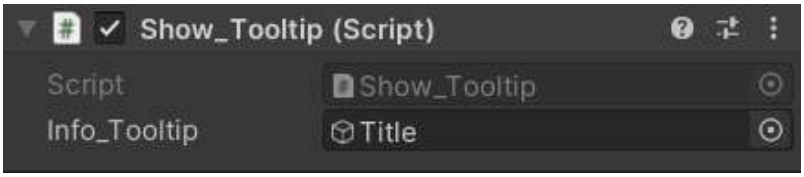

Figure 5.20: The Tooltip component.

5.4.3.1 Resolution and Display Mode

For these two settings, it was designed to implement a horizontal selector to allow the player to change the values horizontally. For this, a script called *Horizontal_Selector* was created.

```
...
public Text textValue;
public string[] selectorValues;

public void Next_Value()
{
    //If we do not exceed the array length
    if (i + 1 < selectorValues.Length)
    {
        i++;
        textValue.text = selectorValues[i];
```

```
    }

    Player_Behaviour.isNext = false;
}
...
private void LateUpdate()
{
    if (Player_Behaviour.isNext)
        Next_Value();
}
...
```

Code Sample 5.13: Horizontal Selector.

In the script, an array of *strings* can store various values. Alongside the array there is a reference to a Text element, to display the string values on the screen. In the method called *Next_Value()*, the array is iterated through an if statement to show the currently selected value, meaning that whenever the player presses to move forward on the array it will show the correspondent value on screen, as seen in code sample 5.13. The same logic is applied, however, reversed, whenever the player presses to move backward on the array on the *Previous_Value()* method.

In the figures 5.21 and 5.22, the components and its array values are implemented.

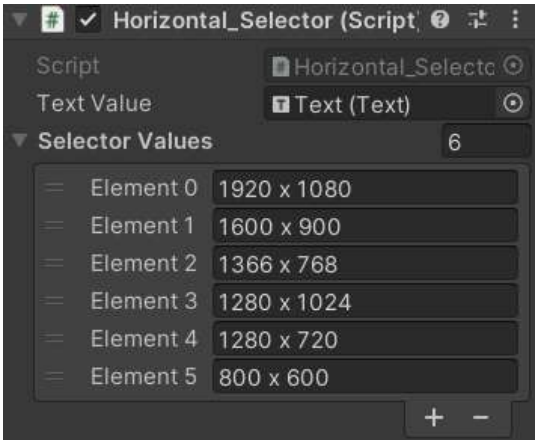

Figure 5.21: The Horizontal Selector component for the resolutions.

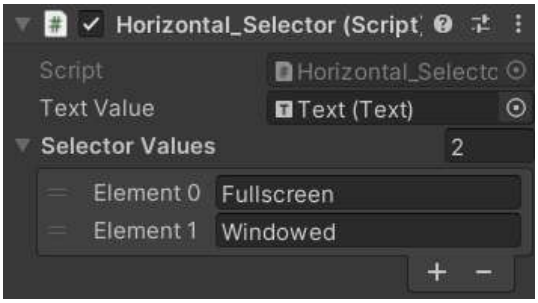

Figure 5.22: The Horizontal Selector component for the display modes.

In order to apply the different resolutions and display modes, a script called *SavePlay-*

erPrefs was created to apply all the display and graphic settings of the game. In the script, various methods were created to change the quality of each setting.

The code sample 5.14 exemplifies how the resolution and display mode can me changed.

```
...
//resolution
Screen.SetResolution(1920, 1080, true);

...

//display mode
Screen.fullScreenMode = FullScreenMode.ExclusiveFullScreen;
...
```

Code Sample 5.14: Change resolution and display mode.

5.4.3.2 Brightness

To tweak the brightness of the game, a Slider element was implemented that contains values from 0 to 10, where 0 represents the lowest value of brightness and 10 represents the highest value. To apply these value changes, a script called *Slider_Values* was created. In it, a method called *Brightness_Value()* was created to be invoked whenever the value in the slider is changed. In that method changes to the overall exposure of the screen are done, by matching its values where *exposure.postExposure.value* equals the current slider value. The figure 5.23 shows the slider invoking the method whenever the value changes.

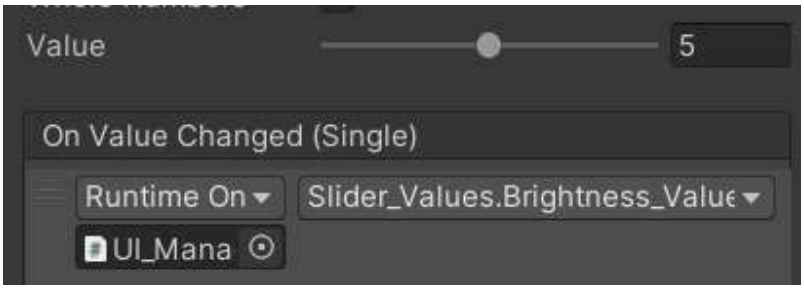

Figure 5.23: The brightness slider invoking the method on value changed.

5.4.3.3 Vertical Sync and Subtitles

These settings contain the Toggle element that accepts two states, on or off. When the toggle is on, the setting is enabled and disabled when the toggle is off.

To effectively enable or disable the vertical sync option, a method called *Save_VSync()* was created in the *SavePlayerPrefs* to be invoked on value changed, where *QualitySettings.vSyncCount=1* if the toggle is on or *QualitySettings.vSyncCount=0* if the toggle is off. The figure 5.24 shows the toggle invoking the method whenever the value changes.

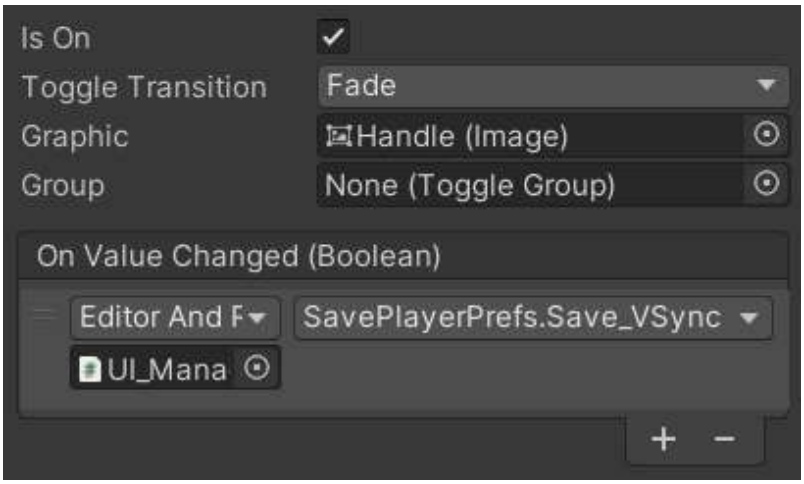

Figure 5.24: A Toggle element, Vertical Sync, invoking the method on value changed.

As for the subtitles, their behavior is the same as any other Toggle element. To enable or disable the subtitles, a method called *Save_Subtitles()* was created in the *SavePlayerPrefs* to be invoked on value changed, where a reference to the Text element of the subtitles is enabled or disabled, per the toggle value.

5.4.3.4 Post-Processing Effects

The last four settings available in the Display menu are settings related to the Post-Processing. As it was described in section 3.1.4.4 and detailed in section 4.8.2.2, these Post-Processing elements enabled us to enhance the visuals of the game by giving it a more realistic feel.

To access these elements and their properties, a Volume component was necessary to implement in the Main Menu Screen, to apply the needed modifications, and in the gameplay, to also apply the needed modifications and to visualize the effects working in unison while playing the game.

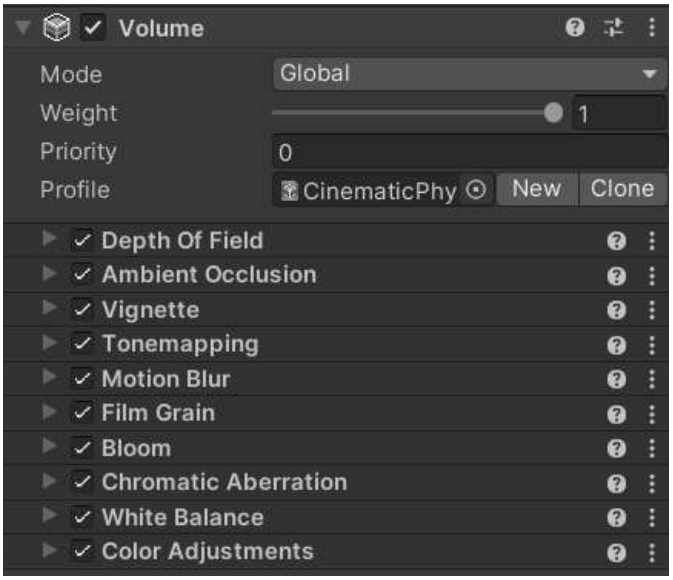

Figure 5.25: The Volume component.

In figure 5.25, the Volume component contains several Post-Processing elements that were needed to change the overall visuals of the game, however, only four of them are available for the player to enable or disable, them being the Motion Blur, Bloom, Film Grain, and Chromatic Aberration.

In order to disable and enable these elements, in the *SavePlayerPrefs*, four methods, one for each element, were created to change their status.

```
...
public Volume volume;

...

volume.profile.TryGet<MotionBlur>(out motionBlur);
volume.profile.TryGet<Bloom>(out bloom);
volume.profile.TryGet<FilmGrain>(out filmGrain);
volume.profile.TryGet<ChromaticAberration>(out chromaticAberration);

...

motionBlur.active = true;
bloom.active = false;
filmGrain.active = true;
chromaticAberration.active = false;
...
```

Code Sample 5.15: Change the status of the four elements.

A reference for the Volume component was necessary to access its properties with *volume.profile.TryGet<>*, as seen in code sample 5.15. After getting access to the respective elements, it was then possible to alter the status of the elements with one of the two states, *true* to enable it and *false* to disable it. Since these four settings are a Toggle element, their behavior is the same as of the Vertical Sync and the Subtitles. In the following figures, 5.26 and 5.27, it is possible to compare when the four Post-Processing elements are disabled and enabled, respectively.

Figure 5.26: The Post-Processing elements disabled.

Figure 5.27: The Post-Processing elements enabled.

5.4.4 Graphics Settings Menu

Since VoDS is a game with high-fidelity graphics, providing options for the player to tweak some graphics settings is ideal. This is due to a large variety of PC components and the possibility of their combinations between the CPU, GPU, and RAM, among others, because every machine behaves differently and there is no guarantee that every PC is capable to play the game with a high-quality level while maintaining a good performance.

The Graphics Settings menu contains six options for the player to manage. The first five options consists of a horizontal selector, as seen in figure 4.20, identical to the Resolution and Display Mode settings, discussed in 5.4.3. The sixth and last graphic option is a Toggle element, identical to some of the options present in the Display settings menu.

The very first option in the Graphics Settings is the Preset setting. This setting allows the player to quickly modify all settings to the selected suggestion, ranging from Ultra Quality to Low Quality, as seen in figure 5.28. This means that whenever the Preset quality changes, the Texture and Shadow Quality, the Ambient Occlusion, and the Anti-Aliasing settings all change automatically according to the selected Preset value.

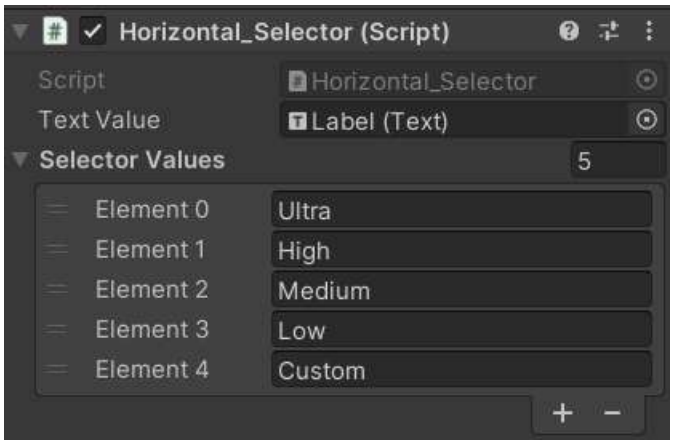

Figure 5.28: The Horizontal Selector component on the Preset setting.

No matter the Preset value set, the other four graphics settings remain locked. However, the Preset setting has another quality entitled Custom. When this quality is set, the other graphics settings are unlocked, so that the player can freely choose their quality levels. Once either quality levels are chosen, in the *SavePlayerPrefs* script, a method for each setting was created to apply the options to the selected levels.

5.4.4.1 Textue Quality

To alter the Texture Quality of the game, Unity contains an option in the Project Settings that contains four different quality levels, representing Ultra, High, Medium, and Low quality respectively:

- **Full Res** - The base texture level is show with full resolution;
- **Half Res** - The base texture level is show with half resolution;
- **Quarter Res** - The base texture level is show with a quarter of the resolution;
- **Eighth Res** - The base texture level is show with an eighth of the resolution;

To change this option in the script, its value has to change according to the selected quality level. For example, if the player chooses the Ultra Texture Quality, the *QualitySettings.masterTextureLimit* must equal to 0. These values range from 0 to 3, representing a decreasing quality whenever the value increases. In the figures 5.29 and 5.30, it is possible to compare the Low Texture Quality with the Ultra Texture Quality, respectively.

Figure 5.29: Low Texture Quality.

Figure 5.30: Ultra Texture Quality.

5.4.4.2 Shadow Quality

To change the Shadow Quality of the game, Unity contains an option in the Project Settings that contains four different quality levels, representing Ultra, High, Medium, and Low quality respectively:

- **Very High** - The shadow resolution is set to very high;
- **High** - The shadow resolution is set to high;
- **Medium** - The shadow resolution is set to medium;
- **Low** - The shadow resolution is set to low;

To indeed modify this option in the script, its value has to change according to the selected quality level. For example, if the player chooses the Ultra Shadow Quality, the *QualitySettings.shadowResolution = ShadowResolution.VeryHigh*. In the figures 5.31 and 5.32, it is possible to compare the Low Shadow Quality with the Ultra Shadow Quality, respectively.

Figure 5.31: Shadows with Low Quality.

Figure 5.32: Shadows with Ultra Quality.

5.4.4.3 Ambient Occlusion

To change the Ambient Occlusion of the game, the Volume component is necessary since it is a Post-Processing element. When the Ambient Occlusion setting is enabled, the lighting and brightness of objects are rendered taking into consideration the placement of other nearby objects. The Ambient Occlusion offered in HDRP is the Screen Space Ambient Occlusion (SSAO). The SSAO is a technique for approximating ambient occlusion efficiently in real-time.

To indeed modify this option in the script, the setting has to be enabled or disabled, as such where *ambientOcclusion.active* can assume two states: *true* to be enabled or *false* to disable it. The figures 5.33 and 5.34 shows when the SSAO is enabled and disabled, respectively.

Figure 5.33: SSAO is enabled.

Figure 5.34: SSAO is disabled.

5.4.4.4 Anti-Aliasing

The Anti-Aliasing method is a setting that softens jagged edges on objects to improve the visual experience. In the HDRP, Unity offers several Anti-Aliasing options. The ones used on this project were:

- **Fast Approximate Anti-Aliasing (FXAA)** – This option smooths edges on a per-pixel level. This is the least resource-intensive Anti-Aliasing technique in HDRP and slightly blurs the final image;
- **Subpixel Morphological Anti-Aliasing (SMAA)** – This option finds patterns in the borders of an image and blends the pixels on these borders according to the pattern it finds. For that, it has much sharper results than FXAA and is well suited for flat, cartoon-like, or clean art styles. It has three quality levels: High, Medium, and Low;
- **Multi-Sample Anti-Aliasing (MSAA)** – This option samples multiple locations within every pixel and combines these samples to produce the final pixel. This is better at solving aliasing issues than the other techniques, but it's much more resource-intensive. This option contains three sample qualities: MSAA 8x being the highest quality, MSAA 4x, and MSAA 2x being the lowest quality;
- **Off/None** – Disables the Anti-Aliasing options.

To change the quality of this option in the script, its value has to change according to the selected quality level. For example, if the player chooses the MSAA 4x, the *QualitySettings.antiAliasing* must equal the sampling quality chosen, which is 4. In another example, if the player chooses the FXAA, that option is configured as *HDAdditionalCameraData.AntialiasingMode.FastApproximateAntialiasing;*. The figures 5.35, 5.36, 5.37, and 5.38 shows the various Anti-Aliasing options in action.

Figure 5.35: No Anti-Aliasing.

Figure 5.36: With FXAA.

Figure 5.37: With SMAA on High quality.

Figure 5.38: With MSAA 8x.

5.4.4.5 Soft Shadows

The Soft Shadows is an option that produces shadows with a soft edge that enhances realism. When this option is disabled, the game renders Hard Shadows instead.

To perform the modification for this option in the script, the setting has to be enabled or disabled, as such where *QualitySettings.shadows* can assume two options: *ShadowQuality.HardOnly* to render Hard Shadows only or *ShadowQuality.All* to render both Hard and Soft Shadows for greater realism.

This setting is a Toggle element, therefore its behavior is identical to other Toggle elements present in the Display settings menu.

In the figures 5.39 and 5.40, it is possible to compare the Hard Shadows and the Soft Shadows, respectively.

Figure 5.39: Shadows with Hard Shadows.

Figure 5.40: Shadows with Soft Shadows.

5.4.5 Audio Settings Menu

In VoDS, several audio sources are present throughout the game, both in the Main Menu Screen and in the entire gameplay. As per the design of the game, the player can define the volume of the game. There are three categories of sounds in the project:

- **Music** – This category comprises various soundtracks/music;
- **Sound Effects (SFX)** – This category comprises all the sound effects;
- **Dialogue** – This category comprises voice over audios.

In Unity, the Audio Source component is responsible to play the respective audio clips in the game. This component also comes with a property called *Output*, as seen in figure 5.41, where an Audio Mixer can be placed.

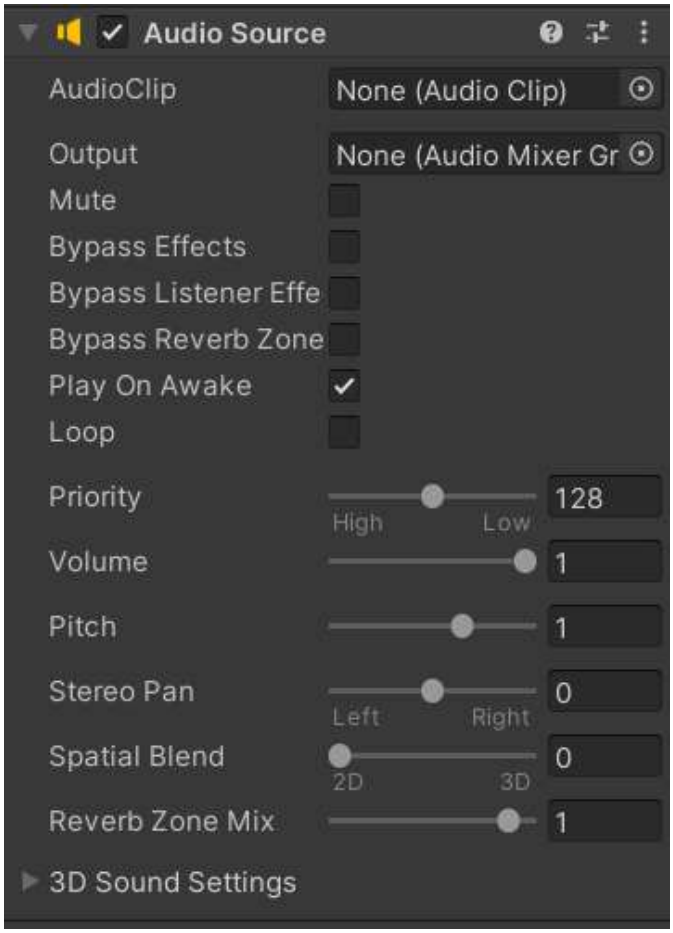

Figure 5.41: The Audio Source Component.

An Audio Mixer is essentially a mix of audio, a signal chain which gives the possibility to apply volume attenuation on a certain audio mixer group. Since there are three categories of sounds in this project, three audio mixer groups were created to represent those sounds, as seen in figure 5.42.

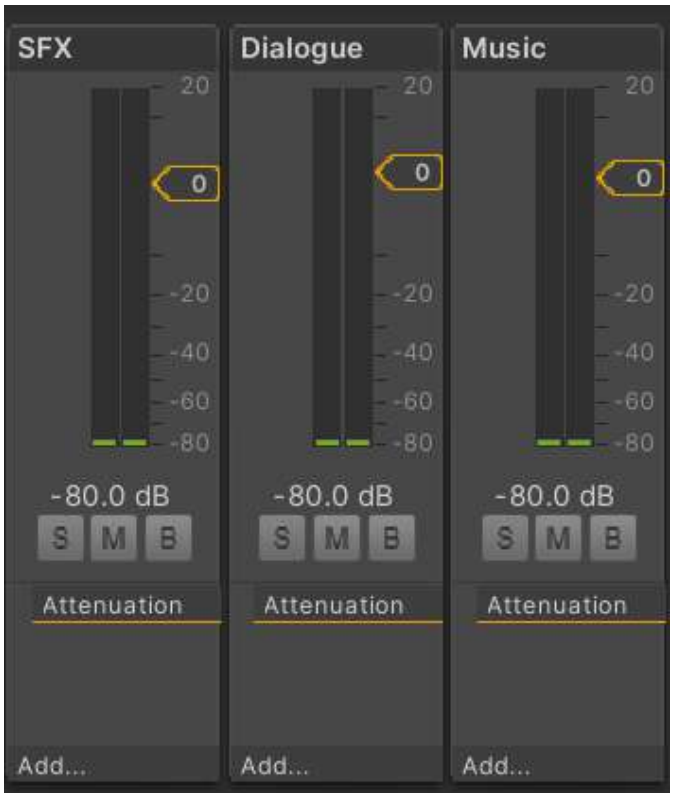

Figure 5.42: The Audio Mixer with three audio mixer groups.

By creating these audio mixer groups, it was possible to implement the option that changes the volumes of each audio mixer group. In the Audio settings menu, three sliders were implemented. In the already created script, *Slider_Values*, for each slider, was created a method to manage its values, thus changing the volume of each audio mixer group.

```
...
public AudioMixer gameVolumes;
public Text musicVolume;
...
public void Music_Volume(float volume)
{
    gameVolumes.SetFloat("Music", Mathf.Log10(volume)*20);
    int value = (int)(volume * 10);
    musicVolume.text = value.ToString();
}
...
```

Code Sample 5.16: Change the Music audio mixer group.

The code sample 5.16, shows the method to change the volume of the Music audio mixer group. It also contains a reference to the Audio Mixer and the Text element of the Slider element, so the game can show on the screen the current volume value. The variable *gamesVolumes* contains a reference to the Music audio mixer group so that, whenever the value changes on the slider, the Music volume changes as well. Afterward, the volume value is transformed to a *string* with the *ToString()* method, so that the value on the screen

is updated accordingly. To change the volumes on the SFX and the Dialogue audio mixer groups, the process is identical to the Music audio mixer group, where only the reference in the *gamesVolume* needed to be changed to the specific group.

5.5 Overall UI Icons

Throughout the Main Menu Screen and the gameplay itself, there are several UI icons that give information to the player about the actions he can perform. For example, as seen in figure 4.19, in the lower right corner, there are three UI icons that show the player the actions to the respective button controls on an Xbox Controller. These actions represent the following:

- **Confirm** – When this action is triggered, it saves the changes done to all settings throughout the Main Menu Screen;

- **Select** – The purpose of this action is to enter the menus and to enable or disable the Toggle elements;

- **Back** – This action allows the player to go to the previous menu.

The player can play the game either with a keyboard & mouse or a gamepad. Both devices can be connected at the same time, meaning that the player can easily change the type of device to play the game. Therefore, it is logical to update the UI icons to show the player the respective device controls. For example, if the player has these two devices connected to the game and navigates the menus with the gamepad, the UI icons show the gamepad controls, but if the player starts controlling a keyboard & mouse, the UI icons shows said device controls instead.

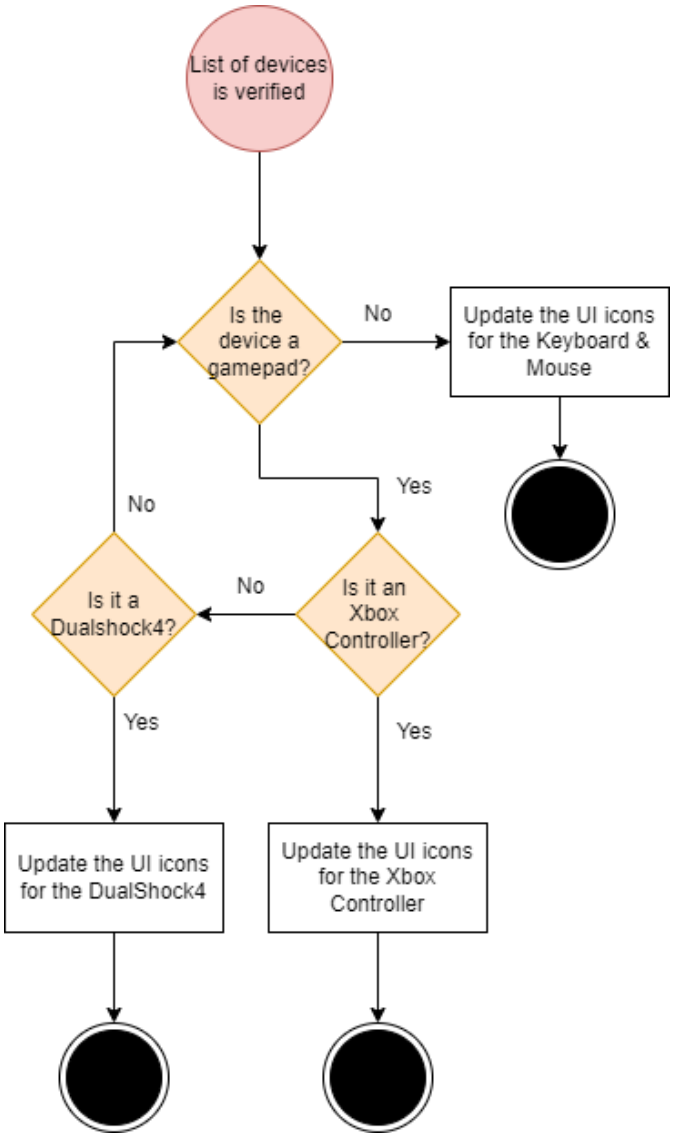

Figure 5.43: The UI icons update logic.

The figure 5.43 illustrates the *UI icons* update logic. To implement this logic, a script called *UI_Button_Changer* was created, as seen in code sample 5.17.

```
...
public PlayerInput PL;
ReadOnlyArray<InputDevice> inputDevices;

...

inputDevices = PL.devices;

if (PL.currentControlScheme == "Gamepad")
{
    foreach (InputDevice device in inputDevices)
    {
        if (device.name.Contains("Xbox"))
        {
```

```
            //Shows the Xbox Controller UI icons
        }

        else if (device.name.Contains("Xbox"))
        {
            //Shows the Dualshock UI icons
        }
    }
}

else if(PL.currentControlScheme == "Keyboard & Mouse")
{
    //Shows the keyboard & mouse UI icons
}
...
```

Code Sample 5.17: Change the UI according to the device currently being used.

With the Input System library it's possible to obtain the current devices that are connected to the machine, therefore a list of devices was created, the *inputDevices*. With it, it's possible to know which control scheme is currently being used. If the control scheme is the Gamepad, the script then verifies if the respective gamepad is either an Xbox Controller or a Dualshock4 and updates the UI icons accordingly. However, if the control scheme is the Keyboard & Mouse, the script updates the UI icons to that of a keyboard & mouse. Since this logic needs to be constantly verified, this script is always being compiled in every frame of the game.

5.6 The Gameplay

Once the implementations of the Main Menu Screen, the camera and character, and the setup of the HDRP assets and its settings concluded, the main gameplay of VoDS was possible to develop.

5.6.1 The Base Loop

Following the attained level design of the game illustrated in figure 4.26, the base loop was built on the *Main_Levels* scene. **The base loop is where all the other loops implemented in the game derive from.**

5.6.1.1 Structure and Furniture

Firstly, the structure of the level was assembled, meaning that the walls, roof, and floor were placed to form the skeleton of the base loop, as seen in the figures 5.43 and 5.44.

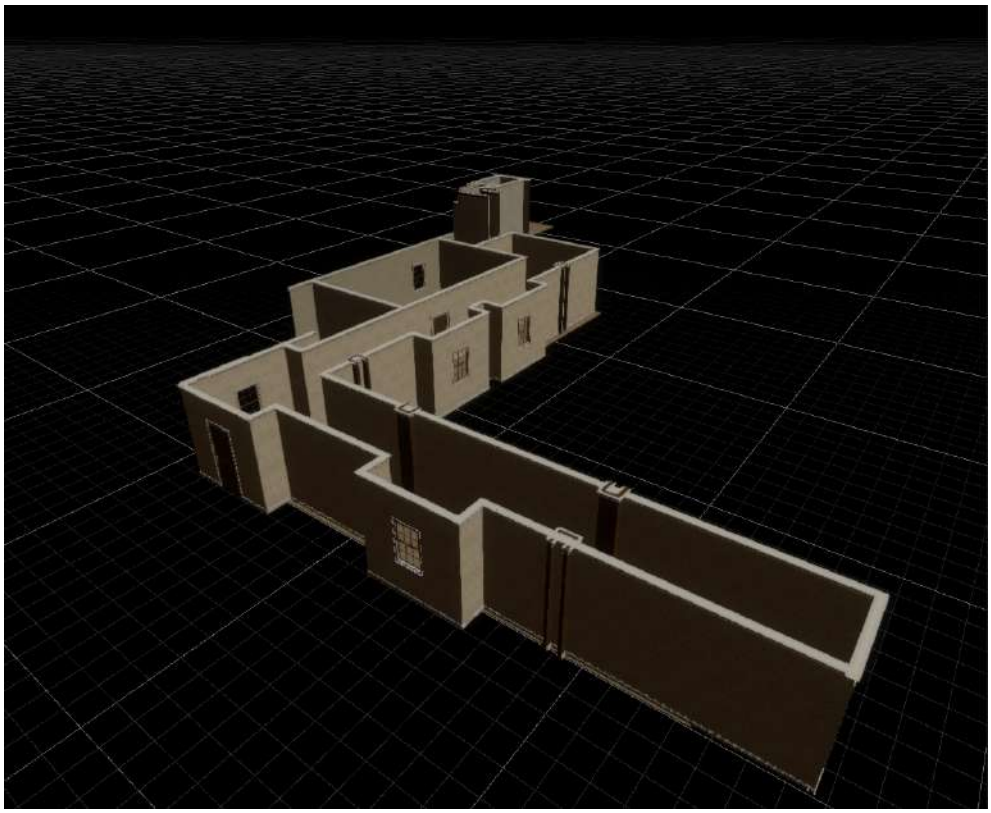

Figure 5.44: Outside view of the structure.

Figure 5.45: Inside view of the structure.

As soon as the structure was built, it was possible to populate the environment with the 3D objects, modeled and textured by João Garcia, as well as some free 3D objects from Sketchfab[43], that were deemed necessary for the level's environment. These placement of these objects are illustrated in the figures 5.46 and 5.47. The figure 5.48 shows a top-down view of the structure and furniture of the level blending in.

Figure 5.46: View of the level's environment.

Figure 5.47: Another view of the level's environment.

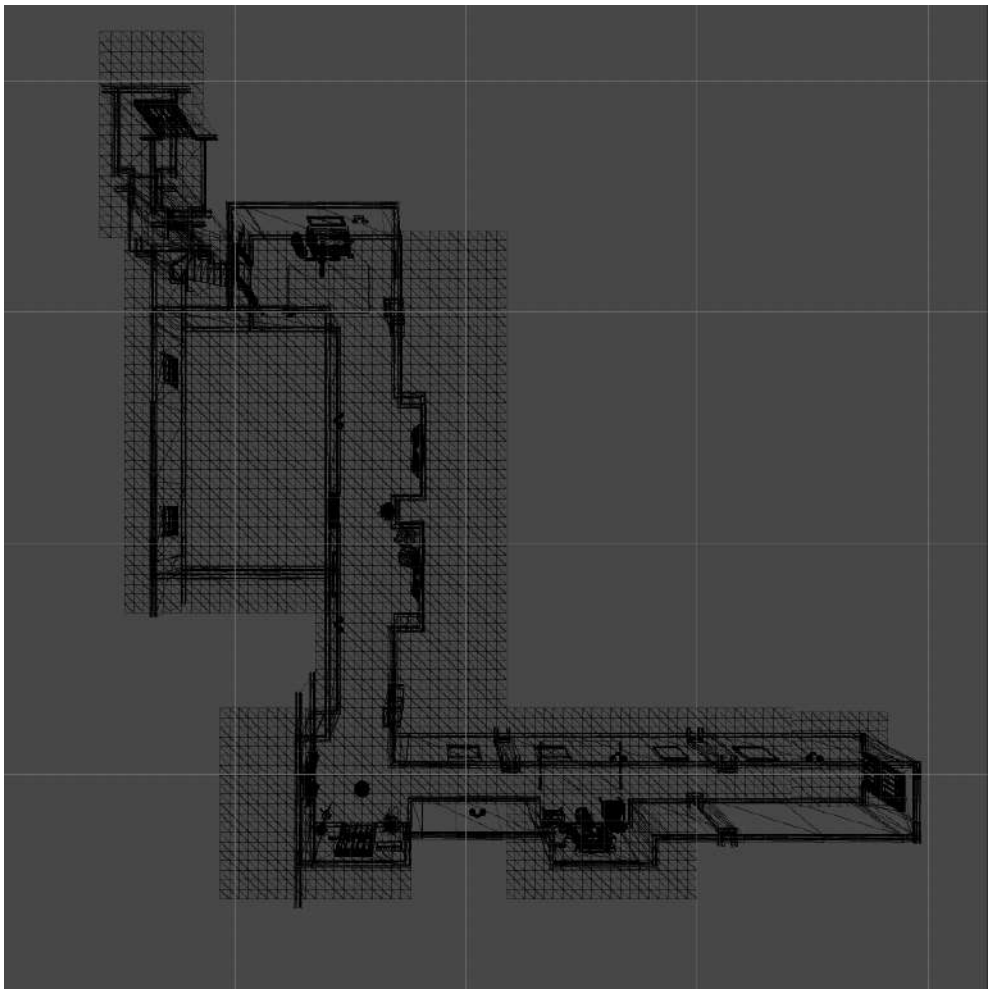

Figure 5.48: Top-down view of the level's environment.

5.6.1.2 Environmental Lighting

In many games, lighting is one of the most important things, especially in the horror genre. Generally, lighting has a drastic effect on the feel of a game, as it determines how a player interacts with and views the world while playing the game. In horror games, lighting is one of, if not the most important aspect of the game, for the sole purpose of making games scarier for the player with dramatic lighting and shadows.

In Unity, the HDRP asset offers lighting methods that allowed us to enhance the project's visual appeal with ease, having four types of Light components:

- **Directional Light** – It's a light that's located infinitely far away and emits light in one direction only;
- **Point Light** – It's a light that's located at a point in the scene and emits light in all directions equally;
- **Spot Light** – It's a light that's located at a point in the scene and emits light in a cone shape;

- **Area Light** – It's a light that's defined by a rectangle or disc in the scene, and emits light in all directions uniformly across its surface area but only from one side of the rectangle or disc.

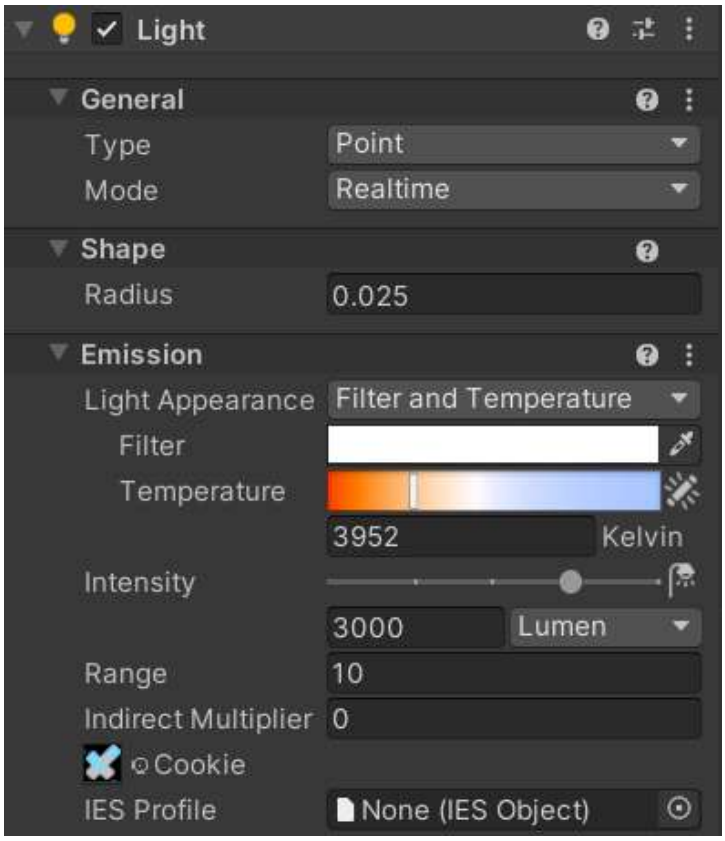

Figure 5.49: The Light component.

The Light component also contains a mode that defines how the light behaves, as seen in figure 5.49. There are three modes:

- **Realtime** – Where Unity calculates and updates the lighting of Realtime Lights every frame during runtime.
- **Baked** – Where Unity pre-calculates the illumination from Baked Lights before runtime, and does not include them in any runtime lighting calculations;
- **Mixed** – Where Unity performs some calculations for Mixed Lights in advance, and some calculations at runtime.

For VoDS, all four types of light were used and the Realtime mode was chosen, to light up and modify the visuals and feels of the environment. As for the Directional Light, its Intesity property value was set to 0 to avoid the exterior lighting to affect the interior of the house. The Spot Light was only used for the flashlight that the player picks up later in the game. With it, a flashlight lighting feel was able to be implemented. The figures 5.50 and 5.51, illustrate the Spot Light component in the scene and the final look of the flashlight lighting, respectively.

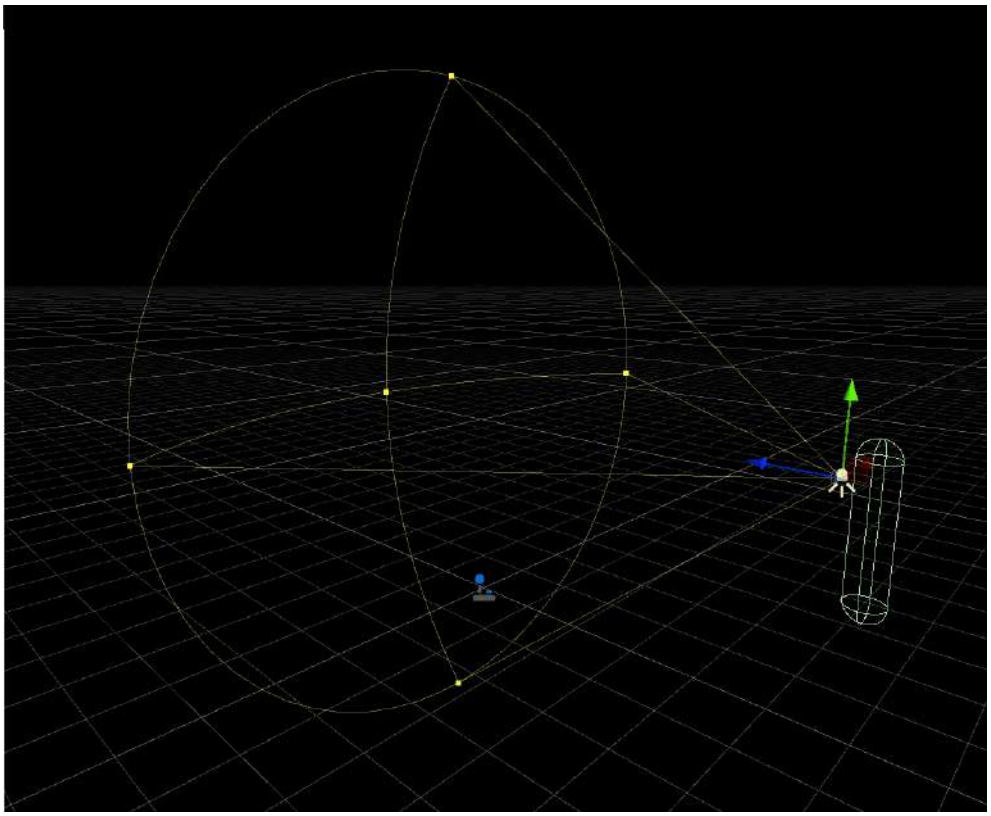

Figure 5.50: The Spot Light component in the scene.

Figure 5.51: The achieved light for the flashlight.

As for the Point Light, they were used to illuminate the overall area of the environment. They were placed to simulate the light emitted from the wall, ceiling, floor, and desk lamps. The figures 5.52 and 5.53, illustrate the Point Light component in the scene and an example of the final look of a wall lamp light, respectively.

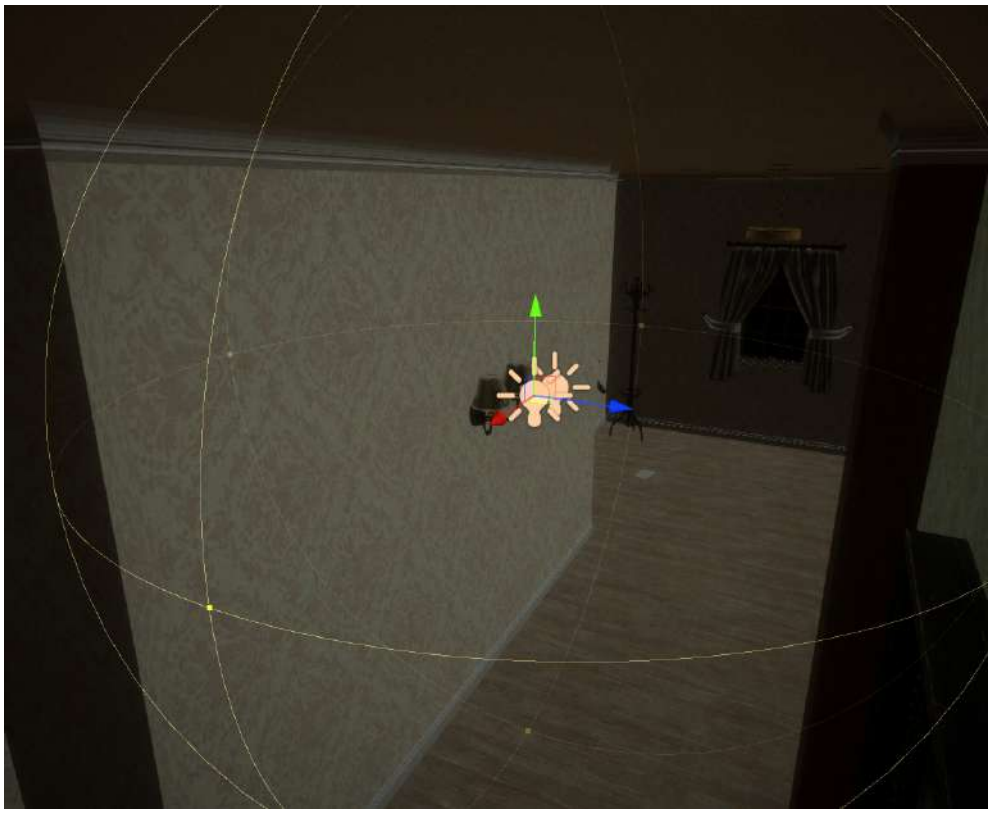

Figure 5.52: The Point Light component in the scene.

Figure 5.53: The achieved light for a wall lamp.

Finally, the Area Light was simply used to simulate the light being emitted from a TV

later in the game. The figures 5.54 and 5.55, illustrate the Area Light component in the scene and the achieved simulation of light being emitted from a TV, respectively.

Figure 5.54: The Area Light component in the scene.

Figure 5.55: The achieved light for the TV.

Below are two more figures to illustrate the environment of the game lit up by the implemented lights, as seen in figures 5.56, 5.57, and 5.58.

Figure 5.56: View of the environment with lights.

Figure 5.57: Another view of the environment with lights.

Figure 5.58: Top-down view of the environment with lights.

5.6.2 Loop Transitions

The most difficult part developed in the project was the process to implement the transition to the next loop. How was this problem solved? Since there are eleven loops for the player to explore and transition to, the initial thoughts were to connect each loop where the end of a loop was connected to the beginning of the other loop. This method was tested with just three of the loops, but it was then discarded because it generated various performance issues, such as low FPS, and high workloads on both CPU and GPU, due to the game practically rendering the base loop three times with all the lights, shadows, events, and audio sources implemented.

To tackle this issue once and for all, it was then thought about the Portal mechanic seen in the Portal games[44] to solve the loop transition problem. The Portal mechanic consists of creating two inter-spatial portals between flat planes, allowing **instant travel and a visual and physical connection between any two different locations** in 3D space, as seen in figure 5.59, where the player enters a Portal A and exits on a Portal B.

Figure 5.59: Simple portal method representation.

The portal method allowed to only load one loop at a time and only load the next loop right before transitioning to it, thus improving the performance of the game. The figure 5.60 represents the loop transition logic to fully comprehend it before its implementation.

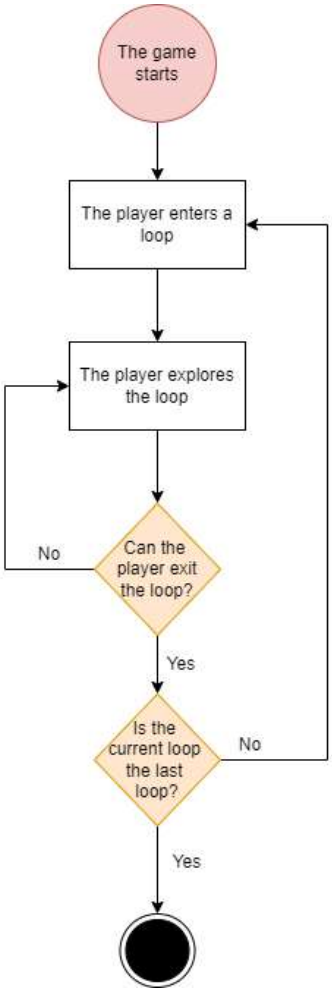

Figure 5.60: Loop Logic.

Following the loop logic to implement the loop transition, three scripts were created:

- ***Portal_Camera*** – For the player to view the loop while transitioning, a new Camera component was placed in the scene to render the upcoming loop. This camera is then disabled once the player finishes the transition process. This script was created to configure the camera movements concerning the player camera to view the next loop;
- ***Portal_Texture_Setup*** – For the player to view the loop through the camera, a render texture was created within the end of the base loop so that whenever the player is transitioning, the player can view the rendering of the next loop. This script is responsible for setting up the texture as soon as the gameplay is loaded in the Loading Screen scene;
- ***Teleport*** – To teleport the player, two colliders were placed in the scene. One was placed within the end of the base loop and the other one was placed at the beginning of the next loop so that when the player enters the first collider, he can be transported to the second collider.

A Loop Manager was created so that the three scripts could blend in together, to create the loop transition. The figure 5.61 represents the Loop Manager with its components, the new Camera component, the portals (colliders) and the render texture.

Figure 5.61: The Loop Manager.

5.6.3 Implementing the Game Mechanics

5.6.3.1 Interact

In Unity, the camera's view contains an infinite line in world space that starts at the origin of the camera and goes to a destination within the camera's view. This line is called Ray. The Ray always corresponds to a point in the view, so the Camera class provides certain functions to perform a *raycast* out into the scene. A *raycast* sends an imaginary "laser beam" along with the Ray from its origin until it hits a collider in the scene. Information is then returned about the object and the point that was hit in a RaycastHit object.

Unity also offers the possibility to tag game objects with specific tags. Since VoDS has a first-person perspective, the player can explore the environment with the camera, while a *raycast* is being performed according to the player's camera movements. Therefore, to identify which game objects were interactive, a tag named "Interactable" was applied to those objects.

With this tag, it was possible to detect with the *raycast* if the object the player is looking at has the ”Interactable” tag, in order to perform the Interact mechanic. To implement this mechanic, a script called *Camera_Interact* was created and attached to the player camera. The figure 5.62 represents an object with the ”Interactable” tag.

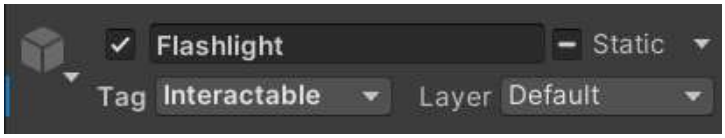

Figure 5.62: The ”Interactable” tag.

```
...
void CheckObjectOnCam()
{
    //Maximum distance that the ray can detect the objects
    float rayDistance = 2.5f;
    GameObject hitObject;

    //Ray from the center of the viewport.
    Ray ray = playerCamera.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0f));
    RaycastHit rayHit;

    //Check if we hit something.
    if (Physics.Raycast(ray, out rayHit, rayDistance))
    {
        hitObject = rayHit.collider.gameObject;

        //Checks if the object is interactable
        if (hitObject.tag == "Interactable")
        {
            if(Player_Behaviour.isInteractable)
               //Perform interact events
        }

        if (hitObject.tag == "Examinable")
        {
            if(Player_Behaviour.isInteractable)
               //Perform examine system
        }
    }
    ...
}
...
```

Code Sample 5.18: Raycast on objects in the scene.

In the code sample 5.18, a *raycast* is performed to detect objects in the scene. If an object is hit with the *raycast*, it is then stored information about said object. Afterward, on an *if* statement, it's verified if the hit object contains the respective tag so that the UI

related to the Interact control can be shown on the screen. It is then verified if the player pressed the respective Interact control so that the Interact mechanic can be performed.

5.6.3.2 Examine System

The perform the Examine System, it was necessary to integrate it within the Interact mechanic. However, once the *raycast* hits an object, it is then verified if said object contains the ”Examinable” tag so that the player can enter the Examine State. The figure 5.63 represents an object with the ”Examinable” tag.

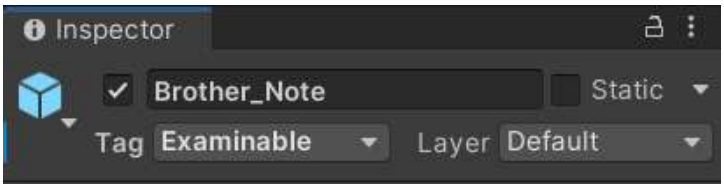

Figure 5.63: The ”Examinable” tag.

Inside the same script, the three method were created to perform the Examine System:

- ***examineObject()*** – This method contains the main core of the Examine System mechanic. This method is where all the object's information is manipulated by allowing the player to rotate and scale the object with mathematical functions. When the player interacts with a specific object, it is then translated close to the camera, so that the player can see the object more clearly.
- ***pickItem()*** – This method is invoked along with the *examineObject()* method, to enable the Examine State's UI.
- ***dropItem()*** – This method is invoked when the player exits the Examine State, by placing the picked object in its original position in the environment.

The figure 5.64 represents the *Camera_Interact* attached to the player camera as a component.

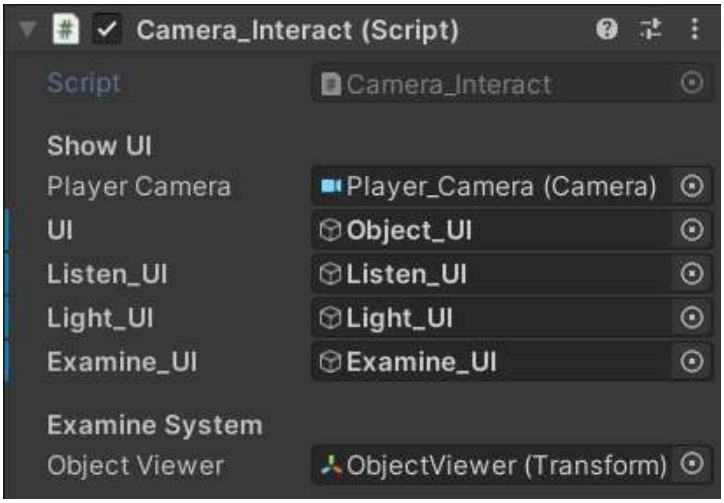

Figure 5.64: The *Camera_Interact* component.

5.6.4 Loop Events

With all the designed mechanics appropriately implemented, the last thing to develop was the loop events. To implement the designed loops, briefly detailed in section 4.9, various checkpoints were placed throughout the environment of those loops. These checkpoints are simply colliders so that whenever the player goes through one, an event is triggered. Some of the events are handled by Coroutines, briefly mentioned in section 5.4.2.1 so that the events can be performed at given moments during the gameplay.

```
...
private void OnTriggerEnter(Collider other)
{
    if (other.tag != "Player")
        return;

    //Perform Events
}
...
```

Code Sample 5.19: Checkpoint to trigger an event.

In the code sample 5.19, when the *OnTriggerEnter()* method is invoked, it verifies if the player went through the collider with an *if* statement. If that result is true, the desired event is performed. If not, the event will not be triggered.

5.6.5 The Pause Menu

The Pause Menu is identical to the Options Menu Screen, excluding the Graphics Settings. Therefore, to implement the Pause Menu, the developed Options Menu Screen for the Main Menu was duplicated and placed in the *Main_Levels* scene. To activate the Pause Menu, a script called *Pause_Behavior* was created.

```
...
void PauseGame()
{
    if (Player_Behaviour.isPaused)
    {
        Time.timeScale = 0f;

        //Pause all audio
        AudioListener.pause = true;

        ...
    }
}

void UnPauseGame()
{
    if (Player_Behaviour.isPaused)
```

```
    {
        Time.timeScale = 1f;

        //Pause all audio
        AudioListener.pause = false;

        ...
    }
}
...
```

Code Sample 5.20: Pause Menu.

In the code sample 5.20, it is verified if the player pressed the related pause control to enable the Pause Menu in the *PauseGame()* method. The *Time.timeScale = 0f* is used to pause all animations and events happening in the game, while the *AudioListener.pause = true* serves to pause all the audios being played.

To close the Pause Menu, the *UnPauseGame()* method is invoked and verifies if the player pressed the pause control to disable the Pause Menu. Opposite values, to that in the *PauseGame()* method, are applied in the *UnPauseGame()* method.

Chapter 6

Tests, Performance and Optimization

6.1 Tests and Feedback

At a point when this project was partially implemented, it was possible to test the early stages of the game with some players. However, before performing the playtests, a survey was required to create in order to obtain the players' opinions about the gameplay experience.

The survey was created based on questions formulated by the author Nicola Whitton in *An investigation into the potential of collaborative computer game-based learning in Higher Education* (p. 153) [45]. These questions are classified as the way to evaluate the engagement effectiveness of games. Each question falls into specific factors, and the survey consisted to evaluate the game in all factors, such as:

- **Challenge:**

  Motivation;

  Clarity;

  Achievability.

- **Control;**
- **Immersion;**
- **Interest;**
- **Purpose.**

The total number of questions that the author provides is 42, however, after a brief examination, only 20 questions, along side 6 new questions created, were deemed as important to evaluate the game. All the questions the players answered to are located in appendix A. To evaluate the questions, the Likert Scale [46] was used because in general, this type of scale is commonly used to measure attitudes, knowledge, perceptions, values, and behavioral changes.

Figure 6.1: The Likert Scale applied for the survey.

The figure 6.1, represents the Likert Scale on the survey that ranges from *Strongly Disagree* (quantitative value of 1) to *Strongly Agree* (quantitative value of 5). The figure 6.2, illustrates an example of a graph about the result of a question asked on the survey regarding the immersion of the game.

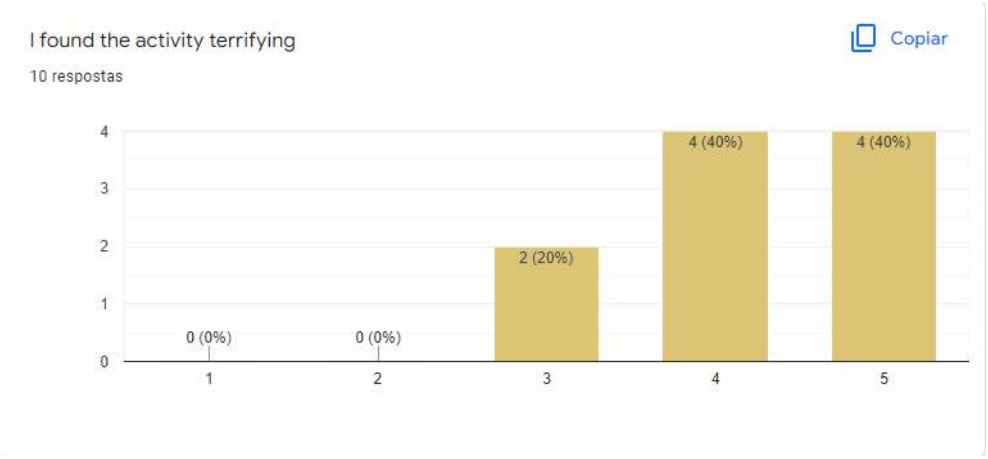

Figure 6.2: Graph containing the results from the question "I found the activity terrifying".

Since the game was still in its early stages of development, VoDS was tested on one of the developers' personal computer. Of all the eleven loops, **only the first six were available to be tested**, although the full scope of this project was tested later on. Being a single-player game, only one player at a time was able to test the game and no help was given to the players unless they asked. A total of ten players were able to test the game, where six of them consisted of students from the first year of the Master's Degree in Game Design and Development at UBI, during the 2021/2022 academic year. The students were only able to test the first six loops. The remaining four playtesters are friends of the developers and fortunately, they were able to **test the full game.**

Curiously, after analyzing the answers from the survey, it was possible to obtain a statistical result of what the testers thought about the game. The survey was answered by 10 testers, therefore, the maximum number of points each question could have was 50 ($5 * 10 = 50$). Since there are 26 questions, the total maximum number of points possible are 1300 ($50 * 26 = 1300$). By summing all the points obtained from each question, the total number of points gathered was 1123. By dividing the total number of points gathered with the total maximum number of points, it was estimated that 87.1% of the answers were very positive towards the game ($1123/1300 = 0.871 * 100 = 87.1$).

At the end of the survey, the testers were able to provide additional feedback by writing brief responses. A total of five responses were sent. Three responses advised increasing the hitbox colliders of interactive objects because, at the time, it was hard to aim and detect the objects due to their small colliders. One response was aimed to slightly improve the lighting whenever the Examine State was enabled because some objects would create shadows on themselves. The last response aimed to improve the overall volume of the 3D sound ambiance. All the problems stated in each feedback were solved as soon as possible.

Out of pure curiosity, to the remaining four playtesters was applied a smartwatch capable to measure their heart rates while playing the game. To register the beats per minute, three checkpoints were performed by the developers during the tests – upon the launch of the game (**Game Launch**), along with gameplay moments of the early loops that were not deemed as scary by the developers (**Soft Gameplay**), and along with gameplay with critical loop shifts and scary events (**Critical Gameplay**).

	Player 1	Player 2	Player 3	Player 4	Average
Game Launch	68 bpm	73 bpm	65 bmp	71 bmp	69 bpm
Soft Gameplay	84 bpm	86 bpm	83 bmp	90 bmp	86 bpm
Critical Gameplay	105 bpm	113 bpm	108 bmp	106 bmp	108 bpm

Table 6.1: Heart rate of four players on three stages of gameplay.

By observing the table 6.1, it is safe to say that VoDS was successfully developed to be a horror game that contains terrifying and scary events to increase the immersion of the players.

6.2 Performance and Optimization

During the development stage of VoDS, more precisely during the implementation of the loops (5.6.2), various performance issues emerged. The consequences of those issues were low FPS, high CPU and GPU workload during runtime. To discover what caused these issues, it was needed to backtrack the main gameplay development, by using a tool that Unity provides, the Profiler, to obtain performance information about the game. Generally, the performance and optimization of a game is nothing but tweaking the graphic settings of the engine to achieve a stable frame rate without the possible decline in graphic quality. The figure 6.3, illustrates the Profiler tool being used to identify the performance issues.

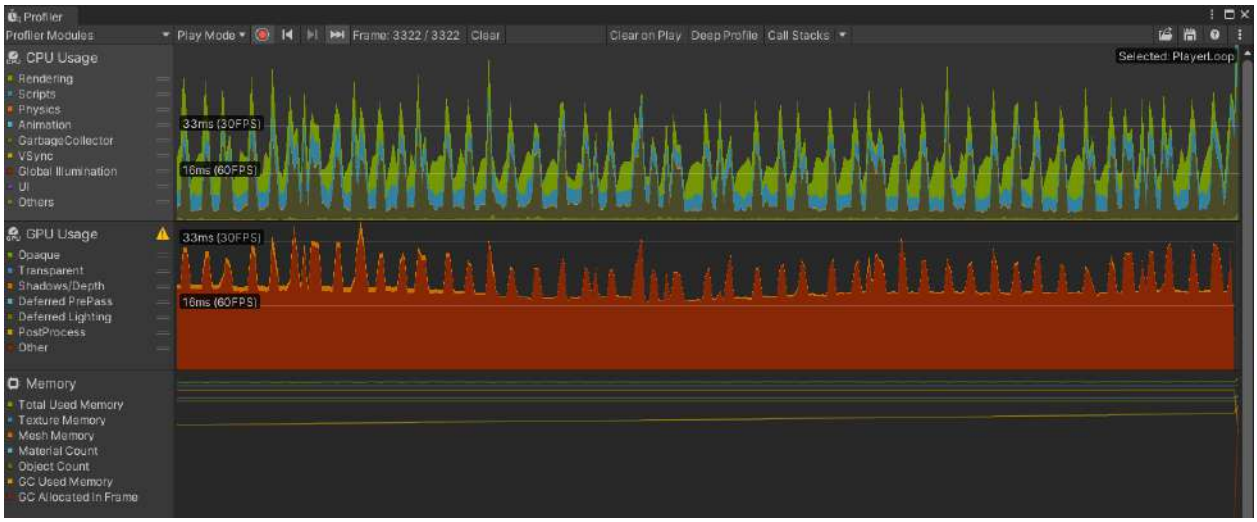

Figure 6.3: The Profiler tool.

After analyzing the Profiler, the causes responsible for the performance issues were:

- **High vertices and triangles count**;
- **High shadow count**;
- **High resolution of textures**;
- **Certain lighting properties**;
- **VSync option**.

To provide a certain degree of realism to the game, various 3D models contained a high number of vertices and triangles. This makes sure that the objects can be seen with greater detail when close to them, however, this is highly demanding in terms of processing power and mostly useless in almost every game. Various 3D models suffered a process in Blender called *Decimate*, by João Garcia, to decrease the number of triangles and vertices as much as possible.

With the use of the *Decimate* process, small performance was gained, but that issue still persisted. Thanks to Unity, the Camera component contains a property called *Occlusion Culling* as it was discussed in section 5.3.3. With *Occlusion Culling* enabled, the camera only renders objects that are visible from the camera'sPOV while also ignoring objects that are hidden behind other objects. The figure 6.4, illustrates the *Occlusion Culling* performing in the game, giving an example of what happens when the player looks down.

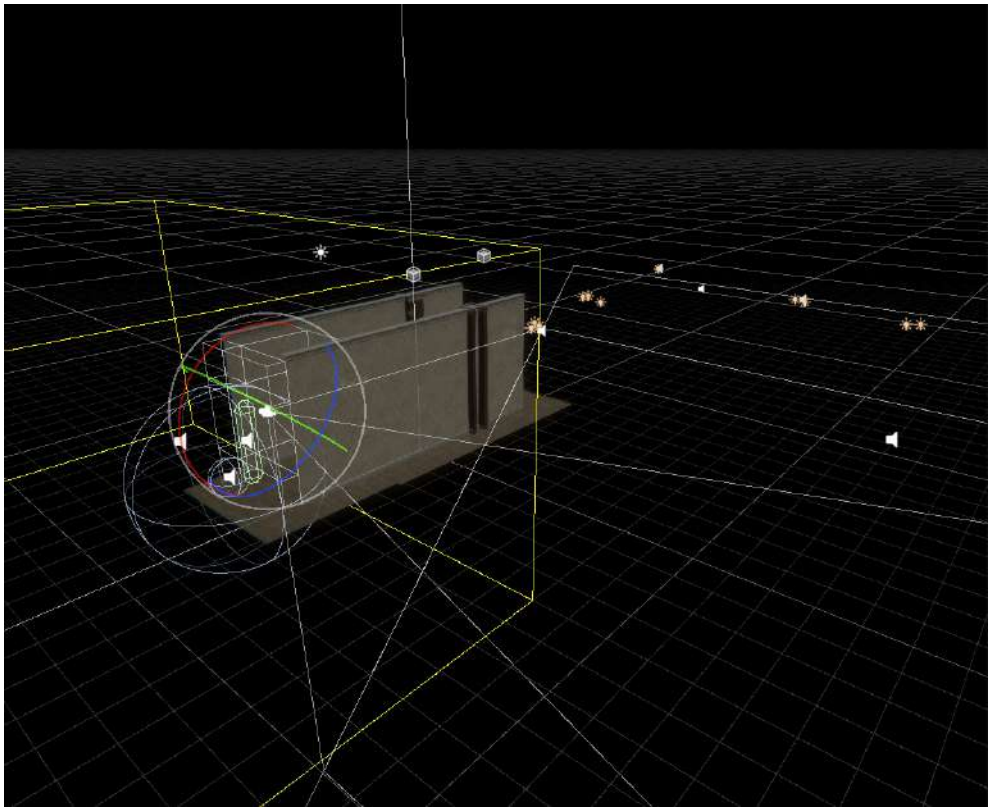

Figure 6.4: The *Occlusion Culling*.

The VSync option is a setting that throttles the frames being drawn to match the number of times a monitor refreshes itself every second. For example, with a 60Hz monitor (refreshing 60 times a second), VSync will adjust the framerate for the game being played to max out at 60 frames per second. However, on some devices, this option can bring performance issues, and deactivating it can greatly boost the FPS gain. The figure below (6.5) illustrates a graph comparing the VSync option when on and off.

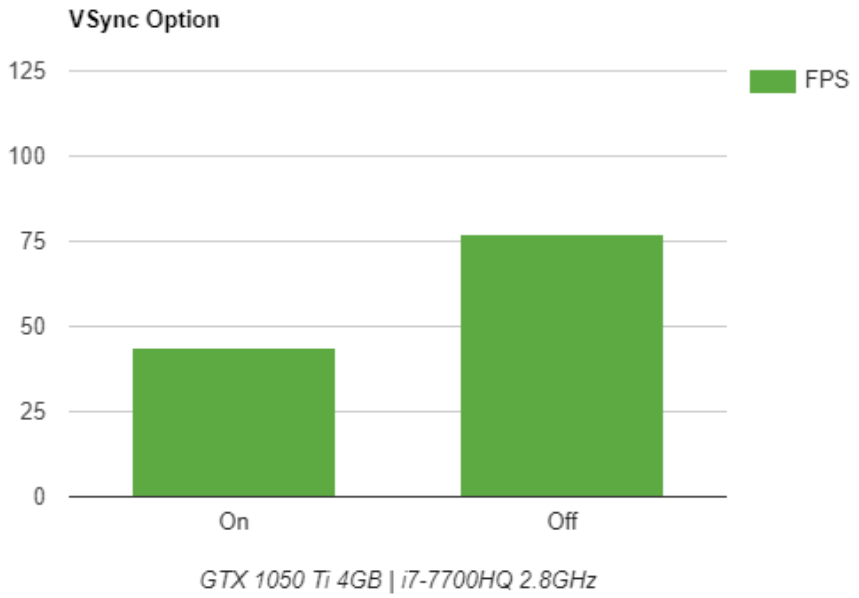

Figure 6.5: Graph to compare when VSync is on and off.

During gameplay, in certain areas of the environment, the game reached low FPS due to a high number of shadows being rendered. The HDRP asset contains properties that allowed us to tweak the number of shadows being rendered on the game on different levels of shadow quality. The figure 6.6, illustrates the HDRP asset properties with changed values to achieve better performance while maintaining high graphics quality.

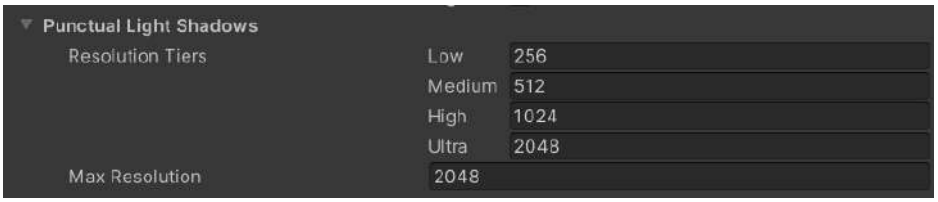

Figure 6.6: The shadow properties in the HDRP asset.

The HDRP asset contains various lighting properties and some of them are enabled by default. Some of those enabled properties were too demanding, therefore they were immediately disabled to increase the performance. Those properties were the following:

- **Volumetrics** – When enabled, a volumetric lighting system will render volumetric fog;

- **Contact Shadows** – The goal of using Contact Shadows is to capture small details that regular shadow mapping algorithms fail to capture, through lights;
- **Screen Space Global Illumination** – This property uses the depth and color buffer of the screen to calculate diffuse light bounces;
- **Subsurface Scattering** – This property handles light that penetrates and moves within the area under a surface. It makes organic materials, like skin, look smooth and natural rather than rough and plastic-like.

Last but not least, during gameplay, it's possible to adjust the texture quality. However, even in low quality, some textures were still rendered with high detail due to their 4K resolution. This issue was simply solved by reducing some of those textures to 1K resolution.

Chapter 7

Conclusion

7.1 Contributions and Achievements

Designing and developing a game is not easy. It's impossible to please all players, everyone has their tastes and opinions. Some play just for the fun and the gameplay, some play just for the narrative, and some try to find a balance between a fun game and a great story, but these types of games are rare and often the most valuable.

The design and development of VoDS began in mid-October and lasted till the first week of June. Working on this project throughout these months made me realize how tough it is to develop a small portion of a demo game, let alone the AAA games that large game companies produce in a span of 3 to 4 years. Having the first experience to develop a game of this caliber made me appreciate the amount of sweat, hard work, and love that developers pour into the games they create, especially the indie developers.

Working on this project vastly increased the knowledge and the usability of the Unity Engine and the C# language, in which various of its packages were used to complement the proposed design of the game. It also allowed me to gain knowledge on first-person perspective mechanics, creating interaction with objects and examining them, developing scripted events, and achieving a game with high-fidelity graphics while maintaining good performance and optimization.

This game was developed alongside another student, João Garcia, who focused on the modeling and texturing of 3D objects that were used to populate the environment. Working with João Garcia improved my experience and knowledge of working and integrating with a team, by respecting his opinions, feedback, design choices, and the overall contribution towards the creation of VoDS. Nonetheless, the main premise of the creation of this game is to inspire other game developers, more importantly, indie developers, that despite the time spent on the creation of a game, it is possible to achieve the game of one's dreams be it with highly realistic graphics or a simple cartoon-style game. VoDS is living proof to not underestimate the capabilities of two indie developers, let alone the skills and experience of what a full team of hard-working indie developers can achieve.

7.2 Future Work

Although the main objective of VoDS for this project was accomplished, there's still a good amount of potential work that can be made in the future, such as implementing new features/mechanics and improving the overall feel of the game.

As it was stated in section 4.2, the way the narrative of VoDS was meant to be told was in an anthological way with various chapters, where each one contains a different story with different characters, alongside the possibility of implementing different mechanics for each chapter. However, the first chapter was somewhat left unfinished, therefore, to initiate the first course of action as future work is to complete the overall narrative of the chapter, extend the size of the level from simple hallways to a full big house, and develop more loops so that the player can experience the full narrative with even more terrifying moments during the gameplay.

As soon as the first chapter is fully completed, the design of the second chapter can begin. It is also possible to bring more talented people to the team with different areas of expertise to develop the game.

Despite the Unity Engine offering tons of support and various pipelines to develop many games, it's still an engine targeted to develop mostly mobile games and although the use of the HDRP helped to create VoDS, it's still not as effective it's competitive, the Unreal Engine[47]. One of the proposed future works for this game is to rebuild it from scratch on the Unreal Engine 5 as it offers tons of properties of high-fidelity graphics and other default components that in Unity, a developer would have to create from the ground up, which can be time-consuming.

Bibliography

[1] Silent Hill Wiki Fandom. P.T. | Silent Hill Wiki | Fandom, 2022. [Online] https://silenthill.fandom.com/wiki/P.T.?file=The_Never-ending_hallway.jpg. Last checked on June 2022. xiii, 5

[2] WCCFTECH Nathan Birch. Visage Interview, 2022. [Online] https://wccftech.com/visage-interview-pt-influence-next-gen-ports-future-plans/. Last checked on June 2022. xiii, 6

[3] Kinetic Games. Kinetic Games | Gallery 1, 2022. [Online] https://kineticgames.co.uk/reasources/img/gallery_1.png. Last checked on June 2022. xiii, 7

[4] A24. Hereditary | A24, 2022. [Online] https://a24films.com/films/hereditary. Last checked on June 2022. xiii, 8

[5] Wikipedia. S.T.A.L.K.E.R.: Shadow of Chernobyl, 2022. [Online] https://en.wikipedia.org/wiki/S.T.A.L.K.E.R.:_Shadow_of_Chernobyl. Last checked on June 2022. xiii, 19

[6] Giancarlo Volpe. Giancarlo Volpe - Camera movement terminology, 2022. [Online] https://giancarlovolpe.tumblr.com/post/32195933498/camera-movement-terminology-is-not-nearly-as-fun/amp. Last checked on June 2022. xiii, 21

[7] Wikipedia. AAA (video game industry), 2022. [Online] https://en.wikipedia.org/wiki/AAA_(video_game_industry). Last checked on June 2022. 1

[8] Iron Gate Studio. VALHEIM, 2022. [Online] https://www.valheimgame.com. Last checked on June 2022. 1

[9] Iron Gate Studio. Iron Gate Studio, 2022. [Online] https://www.irongatestudio.se/. Last checked on June 2022. 1

[10] Steam. Welcome to Steam, 2022. [Online] https://store.steampowered.com/. Last checked on June 2022. 1

[11] Daniel M. & Garry C. Video Games As Culture: Considering the Role and Importance of Video Games in Contemporary Society (1st ed.), 2018. [Online] https://doi.org/10.4324/9781315622743. Last checked on June 2022. 2

[12] Unity. Unity Real-Time Development Platform, 2022. [Online] https://unity.com/. Last checked on June 2022. 2

[13] Unity. Unity HDRP, 2022. [Online] https://unity.com/srp/High-Definition-Render-Pipeline. Last checked on June 2022. 2

[14] Blender. About Blender, 2022. [Online] https://www.blender.org/about/. Last checked on June 2022. 2

[15] Silent Hill Wiki Fandom. P.T. | Silent Hill Wiki | Fandom, 2022. [Online] https://silenthill.fandom.com/wiki/P.T.. Last checked on June 2022. 5

[16] Kojima Productions. Company | Kojima Productions, 2022. [Online] https://www.kojimaproductions.jp/en/company. Last checked on June 2022. 5

[17] Konami. About KONAMI, 2022. [Online] https://www.konami.com/corporate/en/. Last checked on June 2022. 5

[18] Silent Hill Wiki Fandom. Silent Hills | Silent Hill Wiki | Fandom, 2022. [Online] https://silenthill.fandom.com/wiki/Silent_Hills. Last checked on June 2022. 5

[19] Silent Hill Wiki Fandom. Silent Hill (franchise) | Silent Hill Wiki | Fandom, 2022. [Online] https://silenthill.fandom.com/wiki/Silent_Hill_(franchise). Last checked on June 2022. 5

[20] Visage Wikia Fandom. Visage Wikia | Fandom, 2022. [Online] https://visage.fandom.com/wiki/Visage. Last checked on June 2022. 6

[21] SadSquare Studio. SadSquare Studio, 2022. [Online] https://sadsquarestudio.com/. Last checked on June 2022. 6

[22] Phasmophobia Wiki Fandom. Phasmophobia, 2022. [Online] https://phasmophobia.fandom.com/wiki/Phasmophobia. Last checked on June 2022. 7

[23] Kinetic Games. Kitenic Games, 2022. [Online] https://kineticgames.co.uk/. Last checked on June 2022. 7

[24] A24. A24, 2022. [Online] https://a24films.com/. Last checked on June 2022. 8

[25] IMDb. Ari Aster - IMDb, 2022. [Online] https://www.imdb.com/name/nm4170048/?ref_=nv_sr_srsg_0. Last checked on June 2022. 8

[26] Unity. Packages and Feature Sets, 2022. [Online] https://docs.unity3d.com/Manual/PackagesList.html. Last checked on June 2022. 10

[27] Unity. Input System, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/index.html. Last checked on August 2022. 10

[28] Unity. Animation Rigging, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.2/manual/index.html. Last checked on August 2022. 10

[29] Unity. Platforms, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.platforms@0.9/manual/index.html#technical-details. Last checked on August 2022. 10

[30] Unity. Post-Processing, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.postprocessing@2.0/manual/index.html. Last checked on August 2022. 10

[31] Unity. ProGrids, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.progrids@3.0/manual/index.html. Last checked on August 2022. 10

[32] Unity. Unity UI, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html. Last checked on August 2022. 11

[33] Unity. TextMesh Pro, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/index.html. Last checked on August 2022. 11

[34] Unity. Unity - Scripting API:, 2022. [Online] https://docs.unity3d.com/ScriptReference/. Last checked on August 2022. 12

[35] Unity. Unity - Manual: Unity UI, 2022. [Online] https://docs.unity3d.com/Manual/com.unity.ugui.html. Last checked on August 2022. 12

[36] Unity. Namespace UnityEngine.Rendering, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.render-pipelines.core@5.9/api/UnityEngine.Rendering.html. Last checked on August 2022. 12

[37] Unity. Namespace UnityEngine.Rendering.HighDefinition, 2022. [Online] https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.0/api/UnityEngine.Rendering.HighDefinition.html. Last checked on August 2022. 12

[38] Unity. Unity - Scripting API: EventSystem, 2022. [Online] https://docs.unity3d.com/2018.2/Documentation/ScriptReference/EventSystems.EventSystem.html. Last checked on August 2022. 13

[39] Unity. Unity - Scripting API: SceneManagement, 2022. [Online] https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html. Last checked on August 2022. 13

[40] Unity. Unity - Scripting API: AudioMixer, 2022. [Online] https://docs.unity3d.com/ScriptReference/Audio.AudioMixer.html. Last checked on August 2022. 13

[41] Unity. Unity Asset Store, 2022. [Online] https://assetstore.unity.com/. Last checked on August 2022. 13

[42] Morgan Kauffman Publishers Jesse Schell. The Art of Game Design: A Book of Lenses , 2008. [Online] https://www.taylorfrancis.com/books/mono/10.1201/9780080919171/art-game-design-jesse-schell-jesse-schell. Last checked on June 2022. 15

[43] Sketchfab. Newsfeed - Sketchfab, 2022. [Online] https://sketchfab.com/feed. Last checked on June 2022. 87

[44] Half-Life Wiki Fandom. Portal | Half-Life Wiki | Fandom, 2022. [Online] https://half-life.fandom.com/wiki/Portal. Last checked on June 2022. 95

[45] Nicola Whitton. An investigation into the potential of collaborative computer game-based learning in Higher Education, 2007. [Online] https://www.napier.ac.uk/research-and-innovation/research-search/outputs/an-investigation-into-the-potential-of-collaborative-computer-game-based-learning-in. Last checked on June 2022. 103

[46] Mount Wachusett Community College. RESOURCES FOR SURVEY PLANNING, DESIGN AND IMPLEMENTATION, 2022. [Online] https://mwcc.edu/wp-content/uploads/2020/09/Likert-Scale-Response-Options_MWCC.pdf. Last checked on June 2022. 103

[47] Epic Games. The most powerful real-time 3D creation tool | Unreal Engine 5, 2022. [Online] https://www.unrealengine.com/en-US/?sessionInvalidated=true. Last checked on June 2022. 110

Appendix A

Victims of Dead Stories - Gameplay Feedback Survey

The following 26 questions were asked on the survey that the playtesters answered to after testing the game:

1. I wanted to complete the activity;
2. I wanted to explore all the options available to me;
3. I did care how the activity ended;
4. I did find it easy to get started;
5. The instructions were clear
6. I felt that I could achieve the goal of the activity;
7. I found the activity frustrating;
8. From the start I felt that I could successfully complete the activity;
9. It wasn't clear what I could and couldn't do;
10. The types of task were too limited;
11. I found the activity terrifying;
12. I felt absorbed in the activity;
13. I felt that time passed quickly;
14. I felt emotion during the activity;
15. I felt insecure/anxiety during the activity;
16. I felt excited during the activity, despite being scared or not;
17. The activity was aesthetically pleasing;
18. I wanted to know more about the narrative;
19. I found the activity boring;
20. I was not interested in exploring all of the environment;
21. I enjoyed the activity;

22. I'm eager to wait for the full game to release so I can play and finish the game;
23. I'll recommend my friends to play the game once it's released;
24. The activity was pointless;
25. The feedback I was given was useful;
26. The activity was worthwhile;