Entropy : A First Person RPG

made with Unity

Final Year Project
B.Sc.(Hons) in Software Development

BY
Alex Cherry

May 9, 2020

Advised by Dr. Dominic Carr
Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

CONTENTS

Keywords : Game Engine, Programming Language, Role Playing Game (RPG), Non-Playable Character (NPC), Beta, First Person, Mechanics.

Abstract

Game development today has exceeded human expectations from over a decade ago, which has lead to crazy and ambitious ideas being implemented into games. Developing a game can create an imagination, and motivate Developers to create games better than previously developed games. Due to games constantly exceeding each other, games have advanced to a level in which there are many duplicate variants of them. Creating a unique game is difficult, and often holds back indie developers to create games that could potentially be ground breaking in the market. Entropy strives to create new ideas and mechanics, and this was achieved successfully in some aspects of the game. The game is mostly mechanic based, leading to smooth and fluid game-play for the players enjoyment. Knowing what mechanics could create this experience was an important step in the development process, and good mechanics were applied to the game in the final stages.

List of Figures

Chapter 1

Introduction

Entropy is an **RPG** style game (Role Playing Game - a game in which players take on the roles of imaginary characters who engage in adventures, usually in a fantasized world) that is created using Unity and C# with Visual Studio. The application was designed over a 7 month period, meaning it is still in **Beta** version. This means that it is in pre-release stage. The games core mechanics have been developed fully which is the true purpose of the development of the game. Using Unity has given me great learning experiences and has definitely helped to understand the hardships of developing a game in a short period of time.

Figure 1.1: An image for Unity's Main Logo.

However, using Unity to develop a game can be both simple and complicated. Inside of Unity's engine, there aren't many ways of developing animations and custom graphics, so using an outside application such as Blender is necessary. This has led me to figuring out ways of developing these inside Unity by using C# to manipulate objects in certain fashions.

1.1 Requirements

Operating System : Windows, Mac or Linux will work. Others may vary.
Unity Version : Unity 2018.4.9f1. Project can be imported to later version.
Code Editor : Visual Studio 2019+, 2015/2017 may work.

1.2 Game Overview

Entropy is a **Role Playing Game** (referenced as RPG from here on out) that is designed to be a **Beta** version of a full sized RPG. This means that it is in pre-release stage, as developing a full sized version would take years of development, and there was only a 7 month period of development assigned to this particular project, as described in the Introduction. It is designed to set out the basics and fundamental designs and mechanics of a game, leading to further development in future. It is important to implement mechanics first as it is hard to imagine a game without a proper layout.

1.3 Purpose of Development

The reason of developing Entropy with Unity and C# is to hone my skills as a Game Developer, which is the field I wish to pursue. C# is a very common programming language and is a high-level programming language used in various games, making it the ideal choice for honing development skills. It is also designed so that I can understand the hardships that game developers must go through while designing a game instead of criticizing existing games and the ways they could have done things better. Due to playing a lot of games myself, it has given me a better understanding as to why things are done a certain way.

1.4 Objectives of Development

The main objective behind developing Entropy is to understand the hardships and time limitations it takes to develop a game. As we see from popular game companies and hearing stories of how employees are mistreated, games like Call of Duty are released on a yearly basis. Employees are overworked due to a tight deadline and can cause stress and work to be rushed or incomplete.

The aim of Entropy is to deliver proper functionality within a certain time frame, and not resort to half functioning features or buggy glitches in the game. Any, if not all features, are to function correctly and efficiently at all times when new systems are implemented into the program. When an extra feature is added, it is System Tested, meaning every other functionality is tested alongside the added feature. As time consuming this may be, it ensures the correct addition of it into the system as a whole.

1.5 Structural Overview

This document will go on to explain the Methodologies used to develop Entropy, the Technical Components behind the development, the Systems Design as a whole and why it was designed this way, the Evaluation of the System as a whole and the Conclusions that the project has allowed me to evaluate.

1.5.1 Chapter 2 : Methodology

The Methodologies are important as it gives the users and readers an insight as to why it was developed the way it was. Showing insight as to why I, the developer, have decided to use these methods in order to develop Entropy is vitally important for people to understand any hardships and hurdles that may have been encountered during the development stages. All of the methods used in developing Entropy are discussed in this section.

1.5.2 Chapter 3 : Technical Review

Reviewing the technical aspects injected into the system is important to know the infrastructure and how every part links in with each other. Having more components can sometimes be bad due to complications, which is why Entropy has limited the number of external components it uses. This section goes on to discuss the main components used in Entropy's design.

1.5.3 Chapter 4 : System Design

The design of the system in question can be linked to the technical review section as the parts that are in the system are strongly related to why it was designed this way. The way the system was designed will be explained in this section.

1.5.4 Chapter 5 : System Evaluation

Evaluation of a system is important as it helps users and readers to understand why certain items were added into the system and how they affect and improve its design. The evaluation of Entropy is shown in this section.

1.5.5 Chapter 6 : Conclusion

Coming to a conclusion is the whole reason for going so in-depth in the document. If there are no learning experiences acquired during the development stages, a document should not be written in-depth to begin with. Knowing what should and shouldn't have been done is one of these learning experiences.

1.6 GitHub Repository and Documentation

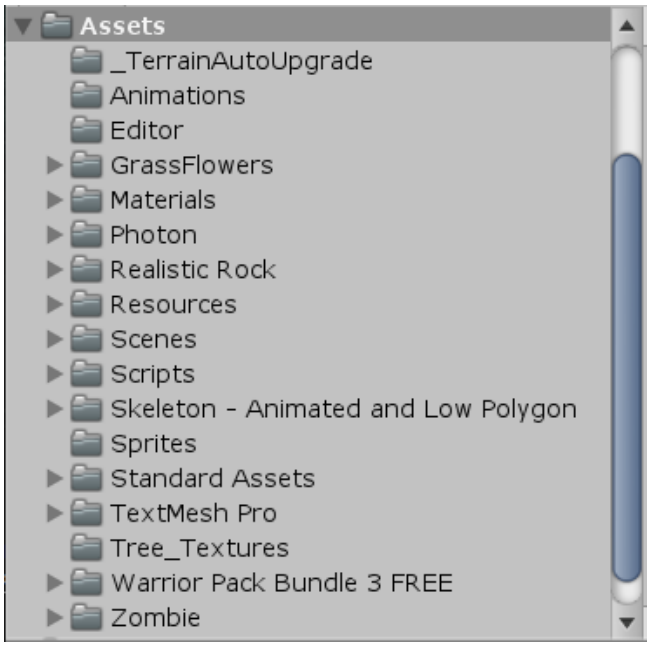
All documentation for development of the game is located at **https://github.com/moecherry99/AlexCherryFinalYearProject**. This repository contains all of the major files required for downloading, cloning, running or editing the application.

1.6.1 Contents of GitHub Repository

The Repository contains 3 folders and 5 files in the main page, only one of which contains the actual contents of the project in the FinalYearProjectEntropy folder. This is where all of the Unity files and logs are, basically where everything in the project was done coding wise. The main directory of the application contains documentation.

Inside of the main folder holds mostly Unity log files and Photon files. The main folder to note here is the Assets folder. This contains all of the scripts and extra resources used in the application. Below is a screenshot from the Unity editor to show what is happening inside here :

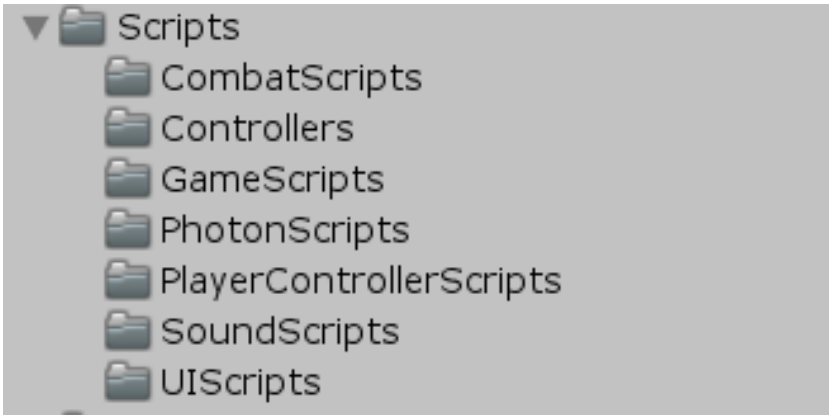
Figure 1.2: File Structure of the Project



Most of these files are for resources, such as texture packs, animations, sprites (both character and enemies) and Photon packages.

The **Scripts** folder is very important as this is where 90% of the development occurred. Below is a screenshot of the folder and its organised contents :

Figure 1.3: Scripts Folder Structure



This folder was carefully organised, as not having an organised project makes it very difficult to memorise and navigate through certain directories. The key to developing a game is efficiency, so knowing where everything is located while developing the project is the best use of your time.

This document will go on to explain every detail about Entropy and its design stages. All of the information in this document may not necessarily reflect all of the work that has been done, as showing every detail of every single script inside of the application would make the document too lengthy. For this reason, please visit the GitHub repository referenced at the beginning and the end of this document if you wish to view all work that has been done.

Chapter 2

Methodology

Having experience with gaming in general through my life has given me insight on how to develop a game. This was my main source of Methodology in developing Entropy. Knowing what issues a game can have, knowing what makes a game unique and what makes a game successful. It is also important to experience what it is like to develop a game instead of criticizing current game developers in what they should be doing or could do in future to their game to improve its experience. Personal experience helps to understand their issues which can lead to an open mind, which to me is one of the most important factors in designing a game.

2.1 Researching Methods

One of the main forms of research for the application was Unity Docs [1]. This site has most of the Unity Documentation on it, and has useful Questions and Answers in Forums. Any functions used inside of the applications code were researched on this site first, and if there were complications or various ways to use these code samples from Unity Docs, Stack Overflow was used [2].

Stack Overflow, not to be confused with the actual definition, is a great site due to the way it operates, in which people will post questions about how their code may be improved or why something is not working. If there are any issues on anything, it has most likely been solved at this point in time (from May 2020) as many applications have already been developed at this stage, and most functions have been used and adapted to suit various systems.

2.2 Types of Methods

Knowing how to apply researching methods is difficult, especially if it's your first time developing a certain type of application. Through my college course I have only developed two types of games, both of which were Mobile games.

These are not quite as in-depth as to what a PC game or a full scale RPG will entail. But there are still some common knowledge ways of applying research, most of which are very feasible and proved extremely useful in the development of Entropy.

2.2.1 Agile Development

Ensuring flexibility and adaptability while developing a game, or any application for that matter, is extremely important for many reasons.

Figure 2.1: Agile Development Image.

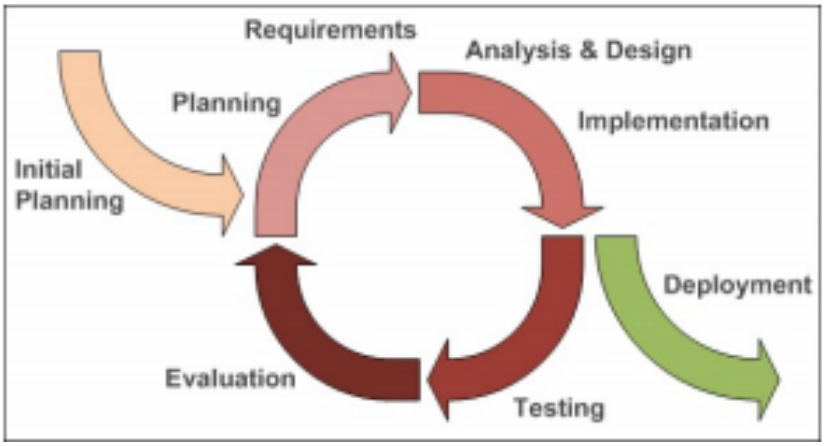


Image source : agile-development-tools.com

Agile methods are a highly evolutionary method of developing software applications [3]. This links in with incremental research methods, as every part of the application is designed in a short term, approximately 1-2 weeks for this application. Some parts may have only taken a couple of days, and other parts could have been a full month. Constantly expanding upon ideas makes Agile Development Methods extremely useful, as it allows for mass amounts of flexibility and adaptability as mentioned before.

2.2.2 Incremental Research Methods

This form of research method was important as visualizing a game from scratch is very difficult to do. While developing a game, having everything set in stone at the beginning is just unreasonable. When certain mechanics are applied to the game, other mechanics will be built on top of those and more ideas will flow when this starts to be visualized properly. This is incremental research, as it is work that is built upon previous developed work. This was one of the main forms of methodology used within the project.

2.2.3 Meetings with Supervisor

I had several meetings with my supervisor **Dr. Dominic Carr** over the course of the college year. After March however, we had to use Microsoft Teams as our main method of communication due to certain restrictions involving COVID-19. This did not impact our communication however.

The structure of our meetings initially involved planning and preparation. I had discussed ideas with Dominic about certain types of games. Originally, the game was going to be a First Person Shooter (FPS - a type of video game whose gameplay involves shooting enemies and other targets and in which a player views the action as though through the eyes of the character they are controlling), but it seemed very plain and simple in my eyes. FPS games are in by no means simple, but the idea of Entropy is to create more imaginative ideas while creating a game. This is why I felt an RPG style game would give me more options into new mechanics and such.

Throughout the first semester up until Christmas, I had just been preparing ideas with Dominic and asking for advice on where to go about certain functions within the application. Not much of the development had started before Christmas as it was all planning. I had structured a Word Document on my computer as a template, and had a notebook for meeting purposes. Eventually, every aspect of the game was transferred to the GitHub repository in the ReadMe section.

In the second semester of the college year the proper development of the game had started. This is where the meetings with Dominic had really started to become in-depth. We would meet once a week or two weeks at this point, and we would email through Outlook as I would discuss ideas and how to go about what I had done next. Dominic gave excellent feedback, and this improved many aspects of the game. Before the meeting structure changed, I was able to show Dominic my game through my computer for a visual representation of all the hard communication we had.

Towards the end of the second semester, the college was forced to close and meetings had to be scheduled on Microsoft Teams. This is when I had to upload frequent videos on the game to the website, which was particularly useful as all of that communication is now documented online and there is proof that valid communication had occurred. I was given excellent feedback every week from this point on as well, and this lead to a successful ending to the projects development.

2.3 Testing

Testing the application was done in 3 ways :

Regression Testing : This type of testing means to confirm that a recent code change has not affected any of the existing features in the application at hand. This form of testing was particularly useful due to the nature of the application. It required excessive testing every time something was added. In particular, adding the health bars in for the player was a very difficult task, as it had to combine effectively with the combat system that was already implemented into the game. However, this type of testing was used for most, if not all, of the functions inside of the application.

Integration Testing : This type of testing means to test individual units together as a group to see if they work in synchronization. One example of this type of testing done within the project is the combat system. Firstly, the enemy layers had to be added so that enemies could be registered. Secondly, when the weapon was added to the player, it needed to connect with this enemy layer in order for it to do damage. Then, damage had to be applied to the enemy or the player if the enemy was in range. All of these had to be tested together, as you can not test damaging an enemy if the enemy doesn't exist, or if you have no way of damaging the enemy either.

System Testing : This type of testing means to test the system as a whole. This had to be done at the end, as it involves testing the system after it is completed as one large unit. It is mainly for evaluation of the system, and seeing if it meets the standards of the developers and/or the customers. However, I could use this method of testing without developing the main menu scene as that particular scene had no effect on the game itself. This means I could start system testing a slight bit earlier that intended, as the menus did not take up much development time.

Again due to certain restrictions, bug testers were limited. Due to living with my brother, he happily accepted to test the game for me for any bugs several times. Being the developer of the game, it is easy to miss these bugs as I would play the game as it is intended, but using bug testers to play the game means they have no idea what will actually happen in the game. As he did not know the code, this was a "Black Box Testing" method. This is important for Bug Testers as they will be the people that would actually play a game if it was released on to a website such as Steam [4].

Chapter 3

Technology Review

There were many technologies used inside of the application, many of which were located inside Unity itself. Researching different game engines (a program where the application would be developed) such as Unreal Engine and Frostbite are some of the many examples of the large array of game engines that I compared Unity to. Knowing which programming language to use such as Python, C#, C++ or Java was all an obstacle in itself. Using certain code editors such as Visual Studio versions is important as well. There was also issues of which external applications to implement into the application, as the project itself had to be quite complex and robust. Applying different technologies helps with the robustness of an application.

The importance of research in itself is extremely important. An application can not be developed to the best of ones ability if they don't research what software to use, where to use it and why they should use it over something else. There are many complications such as your machine not having enough memory to run the software efficiently, your internet speed may be too slow for certain online aspects of the software or it might have a price tag to it.

3.1 Languages

Organising which programming language to use was a difficult task. There are many good programming languages used for games development such as Python and C#. There are popular games such as Minecraft that are developed in Java, but Java itself is difficult to incorporate into a game for various reasons. It was important to summarize all of the possible programming languages to use as this is the key component to designing a game. Game engines are important as well, but linking in the programming language to the game engine is a key aspect too.

3.1.1 Java

Java is currently the most popular object oriented programming language in the world [5]. It is used or can be used in most pieces of software today, such as audio devices, video devices, smart devices and so on. Its diversity knows no bounds with our current technology.

Using Java 8 or 9 is a very viable option when developing a game, but due to some research I discovered that it is more so frequent in mobile applications development with Apache NetBeans [6]. Designing a mobile application is certainly profit worthy in today's market, however it is not worth developing something that you don't enjoy. Mobile applications can seem very bare or simple, and designing an application that you want people to experience and use for a long time is very important to me as a developer. An application with a long life or even live service (an application that is constantly updated for months, even years) will give much more incentive towards developing a robust system.

Java certainly seemed like a strong programming language to use due to its diversity and flexibility, but due to several years of experience with this language over my 4 years in college, it did not feel right to use this as the main language in this application. This is because I wanted to learn something new and exciting.

3.1.2 C++

C++ is a very old yet strong programming language, beginning in 1972. It was one of the first programming languages used to date, being an extension of the C language. It can compile any C code, meaning it can be used with older applications [7].

C++ in games development is very common as it has been used in some of the most popular games to date. World of Warcraft and Counter Strike : Global Offensive were both developed in C++. These two games are extremely popular and have very big differences in them, showing the diversity of the language, with World of Warcraft having massive graphical quality and being a Massively Multiplayer Online Role Playing Game, and Counter Strike : Global Offensive being a strategy based First Person Shooter with much lower graphical quality. It has also been used with phones development such as older Motorola phone models [7].

Researching C++ in games development gave me a lot of insight as to why it is a very good candidate for using as my main programming language in an application. C++ however is not widely used amongst popular game engines that are free to use, giving it some downsides as to why it wasn't used.

3.1.3 Python

Python is popular in game development, however it is used for several applications such as websites and devices. [8]. Python is also a very simple yet in-depth programming language to use, so it helps developers to focus more on the actual game itself such as animations, characters and the over all aesthetics of the game.

I did not want to choose a programming language that would do all of the work for me. The purpose of my course and of this application was to learn a programming language that I enjoyed and any external applications. I have used Python in past modules throughout my 4 years in GMIT, and it was quite enjoyable to use. However, using it for a game didn't seem right, as I wanted to experience all of the major hardships for game developers myself as I would love to be in the seat of a proper game developer. Playing games is one of my favourite hobbies and to understand what is happening is very important to me.

Python was not an option with any of the game engines I researched either. Most of the ones that were free did not support Python use, and any others that were free did not have many features to them to make a game unique and different. This was not a problem with either Python or the game engines themselves, it was combining the two that was the issue as I have no objection to any of the items I have researched personally.

3.1.4 C#

C# is a very complex and versatile programming language designed in the year 2000 and is an extension of the C programming language, much like C++. It is a high-level object oriented programming language, meaning it is capable of very detailed functionality. It is not entirely difficult to use and resembles Java in a lot of senses. Due to my experience with Java in the past, this means I had some idea of how to utilise the language, but it was still a very new experience for me to embark upon.

C# was one of the main games development programming languages that I had researched, and it was a very good choice due to the fact there are vast amounts of documentation online. It is a very popular programming language used with game engines, in particular Unity. It is excellent at catching errors when paired with Unity and is a great learning experience as your mistakes are corrected while developing [9].

In short, the language was an excellent choice for developing Entropy with as it was new to me, diverse, complex and had great teaching capabilities. Due to it being a somewhat new programming language, it is still undergoing development in some areas and is making great improvements day by day. I thoroughly enjoyed using it for my project.

3.2 Game Engines

Researching game engines came from personal experience. Due to being an avid game player, I have heard of popular game engines in the past. This made recovering them from memory a bit easier, and knowing the popular and free game engines was an important aspect of choosing one.

The game engines each had to be tested or researched in advance, as there are many issues with picking a game engine too soon. Each of these are discussed in the different game engine sections listed below.

3.2.1 Unreal Engine

Unreal Engine was designed by Epic Games, a very popular company today. It was created in 1998, meaning a lot of bugs have been ironed out since its creation, and it was used to modify one of the first First Person Shooter games [10]. Code could be replaced for any item in a game if the games files were loaded up using Unreal Engine. This changed the gaming industry massively, as "gamers" like myself know that modifying a game can lead to extraordinary results. It creates an amazing and fun atmosphere for people who love to experiment with the games capabilities.

Unreal Engine is a very user friendly and free game engine that supports both C# and C++ for its games development. This made it a very strong platform to consider using in this application, and it has very strong graphical capabilities as well. Comparing it to Unity, which requires assets to be downloaded and external programs such as Blender to be used for animations in 3D models, it seemed like the better choice.

However, Unreal Engine requires a high-powered device in order for efficient development. This made the early days of development very difficult on the machine that I used, and it just didn't seem worthwhile waiting fifteen minutes just to load up my application when there was barely any content inside of it. I knew that if I had continued using this game engine for Entropy, it would have greatly impaired my development stages. It had to be removed as an option.

3.2.2 Frostbite

Frostbite is a relatively new game engine, with its first game being released being back in 2008 (Battlefield : Bad Company). It is an excellent game engine that deals with large, outdoor environments that are of extremely large scale [11]. It was designed specifically for the Battlefield franchise that was produced by DICE.

Much like the Unreal Engine as discussed before, it requires large amounts of processing power purely due to the fact it deals with massive landscapes. A

good graphics card is needed to even be able to process this, let alone a large amount of RAM and a good processor. The game did not have to be large either, meaning the use of this game engine seemed redundant as its key aspects are the fact it can deal with large games. A smaller, yet efficient game engine was necessary.

To begin with, I did not rule out Frostbite instantly. I could have used the game engine with my application as I did not necessarily need to make a large application inside the engine, demanding high processing power. However, at this point I had already begun using Unreal Engine and was a bit wary of my time limitations. I needed to find a game engine fast to avoid being left behind and struggling, so I did not want to test this theory of making a smaller game inside a large game engine due to the issues that had already arisen with Unreal Engines performance on my machine.

3.2.3 Unity

This is the game engine that I decided to go with for the application as mentioned in the introduction. It was an easy choice, as it came with three programming languages : **UnityScript**, **Java** or **C#**. Due to my initial research on Java, I had a much easier time deciding on using C# with Unity. UnityScript is very similar to Java, but it did not seem wise to use due to these similarities. It also had the potential to not catch code errors, meaning you could type something and Unity would not pick up on the errors you created [9].

The version of Unity I decided to go with in the end was Unity 2018.4.9f1. This was a very good Unity interface that I had downloaded previously for another module in my course in the previous semester. While working on another project in my second semester of the final year, I had been using Unity v2019.3.0b5. The interface for this version of Unity seemed very clunky and unorganized, so I opted to stay with the current version of Unity that I already had installed instead of importing the full project to a new version.

The Game Engine itself is free, just like the other options listed here, but Unity has a very user friendly Asset Store that has a large amount of free assets at the developers disposal. Due to this being a college project, free was obviously the best option. There were some limitations however to certain assets such as enemy models and character models that were not free. Some of the ones that were free did not suit the games atmosphere, so it ended up not being as optimal as a project where spending money was a possibility.

3.3 Code Editors - Visual Studio

This section is purely devoted to all of the versions of Visual Studio that could have been used for editing Entropy as an application. It had to link in with the

Game Engine I was using as well, which in this case was Unity.

This section did not need much thorough research due to the fact I have used Visual Studio in the past and felt it was quite comfortable to use. Speaking from experience from some of my friends and acquaintances, they however did not have a good time using Visual Studio. This did not affect my decision to use it.

The version of Visual Studio I decided to use was **Visual Studio 2019 IDE**. It is an integrated, complete solution with development tools, cloud services and extensions that enables you and your team to create great applications and games for desktops, the web, Windows Store, Android and iOS [12]. I used the full version instead of Visual Studio Code which is a lighter IDE and does not incorporate as many tools as the full version.

3.4 External Applications

There were many ideas of adding externals into the project such as for the Multiplayer, the Quest Systems implemented into the game or even the animations applied to certain models.

3.4.1 Multiplayer

Figuring out a way to incorporate the Multiplayer into the game proved difficult, and a lot of research was put into this. Eventually, I went with the **Photon** plug-in [13], as the standard Unity Multiplayer development platform UNet has since depreciated. Unet has depreciated as it did not meet the basic needs of games developers, so another matchmaking system (A system that allows several clients connect to the same server) is being created. However, Photon has developed a way to replace this in the meantime, as have many other platforms.

The main reasons I picked Photon are as follows :

Free : A very self explanatory argument for using any application. If there is no price tag, a college student such as myself will gladly take up the opportunity to use free software.

Server Based Matchmaking : The way Photon works in terms of matchmaking is that it is Server Based. This means that a dedicated server is set up and players will connect with their clients to the server. This makes it easier to connect as the server gets initialised on the first client that loads up the application. Peer to Peer can get quite complicated as it involves players clients connecting to other players clients. This involves firewall setup which can take the simplicity out of using an external program in your application.

Accessibility : Photon was very easy to set up with the project. I followed a series of videos online that helped set it up in a few hours. Anyone can set up Photon with relative ease, and due to it being a new add on for Unity projects, it is still in development and will remain of the top used Multiplayer match-making systems used.

3.4.2 Animations

Unfortunately after researching game engines and deciding to pick Unity, there is very limited access as to what animations and textures that are available at a developers disposal. This was not a big issue as there are external applications that can be used.

Blender is an example of an external application that can be used for making 3D models and rendering. Blender is the worlds foremost free and open-source 3D graphics application [14]. It has many uses, and it pairs extremely well with Unity 3D.

Applications like Blender also have very little learning time, as they do not require coding skills or courses to master them. Of course there is a learning curve, but it is very slight. It is also cross platform, meaning you can use it wherever you want meaning it has great accessibility and can be used amongst a wide variety of operating systems.

Using Blender with Entropy was a very viable option, but there were not many animations inside of the game to develop. It felt as though not having many objects to animate would have been an unnecessary use of time due to the fact it was a whole new application that would have to be learned. If there were more objects inside of the game it would be an extremely useful tool to be at my disposal.

3.4.3 Databases

During the development of a game, I realised that a database for Quests or even for Enemies in the game would have been a useful addition. Due to the nature of the game and the fact it is a Beta (pre-release), it did not seem to have a large need for databases. My main database that I would have used would have been **MySQL** [15]. Due to using this in several other college modules over the years, this was an easy decision as I have known the capabilities of such a database.

However as mentioned before, using a database seemed redundant due to the fact only one quest has been implemented into the game and a few enemy types. Setting up a database for a small amount of items could have been an excellent addition to the game, as adding further quests and enemies with appropriate stats to the game would become much easier. Having only a few months of

development time required more effort to be put into other items, so making sure the game was functional and fluid was of the utmost importance.

3.5 Review of Researched Technologies

After careful consideration of all of the possible technologies that could have been used such as the Languages, Game Engines and Code Editors discussed in this section, I decided to go for **Unity**, **C#** and **Visual Studio 2019 IDE** as my chosen software to develop Entropy. All of the reasons have been mentioned before in the previous sections, but to put it all in perspective there will be an additional section below.

3.5.1 Chosen Software

1. Unity : This Game Engine is adaptive, flexible and has high performance for lower end computers. It is perfect for Indie game developer companies, Indie meaning that it is used by smaller companies instead of AAA developers, due to the fact big game developer companies can afford better equipment for developing their applications. Unity is the perfect choice of the three game engines listed here.

2. C# : This Programming Language was a tough choice compared to Unity as the main options to pair with Unity were Java and UnityScript. Java however had a lot less documentation online and would have proved difficult to use with the application and UnityScript had drawbacks due to the fact it would not catch errors during the development stages. C# is also relatively new to me as a developer and this was an excellent choice to use as it was quite easy to learn due to large amounts of documentation online.

3. Visual Studio IDE : This was an easy choice as not many Code Editors are free and easy to use like Visual Studio. Visual Studios full IDE can seem a bit daunting to people due to the fact the interface is packed with content such as the solution explorer and the debugger, but these items can be minimized and are not always necessary. Especially if it is paired with Unity, it is very easy to maintain an organised coding environment due to the compatibility both pieces of software serve for each other.

Chapter 4

System Design

Designing a suitable system that worked efficiently for Entropy was step number one in the development stage. How this was done efficiently will be described in this chapter. Most of the work that is done in the system itself was researched before hand and has been explained in the Technical Review chapter of this document.

4.1 Structure

Structuring a project is very important while designing it. This should be mapped out before beginning the project, but with Unity and games development in general due to the flow of things and coming up with new ideas while developing the game, it is not set in stone how it should be laid out.

Figure 4.1: Entropy's Hierarchy Structure

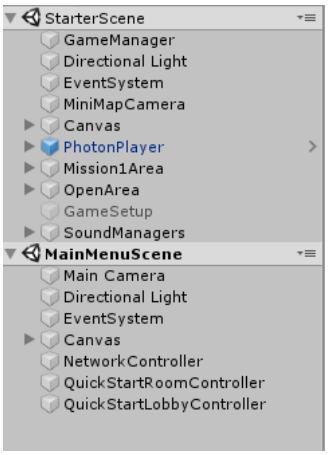


Fig.4.1 clearly shows that the Hierarchy is structured into two scenes : **StarterScene** and **MainMenuScene**. These are the main game scenes and menu scenes respectively. Upon opening the application, the MainMenuScene is loaded first as it is set in the Build Settings inside of Unity's Engine. All of the required game objects are contained inside of these two scenes. The files and scripts structure are shown in the **Introduction** section.

4.2 Early Development Stages - Setting

When beginning the development of Entropy, there were a few items to be discussed. Some of this planning was done with my supervisor, some of it on my own. Creating a setting for a game is very important as it gives the players an immersive experience while playing your game. There are many factors combined that will create this setting.

4.2.1 Terrain

The terrain of a game is the surroundings such as the **Rocks**, **Trees** and **Landscape**. Designing a suitable terrain was not overly difficult as the Unity Asset Store had a lot of excellent packages that were free to use. Instead of designing my own rocks and trees, I had taken assets and used them amongst the landscape. In regards to the landscape itself, a simple cube was taken and stretched to make a large plane. It was then covered with a material which can be created in Unity with a colour scheme. This material was eventually replaced with a grassy theme so it wasn't just a plain full green colour, giving it a realistic feel.

To make the landscape diverse, I had placed rocks and trees scattered around the map. I decided to make a sort of maze, where the player would travel around while defeating enemies. The Open World area of the game which is described later on, where the game begins, let the player freely travel around it while killing enemies. The Mission Area was given the maze like design to it as the player should only go one way to complete this said mission.

Inside of the mission area is also a small **Town** that the player will be teleported to upon entering it. This was developed by using various shapes that are built into the Unity Engine and changing their sizes and dimensions to fit it properly. I created some fences in this area as well so that the player could not escape or go "out of bounds", meaning to travel outside the limits of where the player is permitted to be. Checks like this were necessary as unusual effects would happen if the player were to travel somewhere they weren't supposed to be.

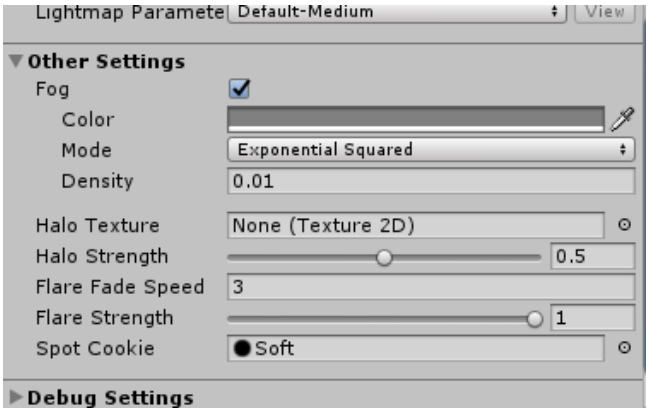
4.2.2 Atmosphere

The atmosphere of the game ties in with the Terrain. These aspects both go hand in hand as you can't have an atmosphere without a place to travel around.

An atmosphere is related more to the less physical items in the game, in this case being **Fog** and **Lighting**.

1. Fog : Adding fog into a game can create a nice feeling of "emptiness" and "fear". This was particularly useful for Entropy's theme as it was designed to be a semi-horror game. To add fog, you must use Unity's built in **Rendering** tools. This can be done by going into Window, then Rendering, then Lighting Settings.

Figure 4.2: Unity's Fog Settings



You can change all of these settings to your liking. I decided to go with a slightly dense fog and less render distance, meaning the player can't see very far. This means that anything they encounter will be a surprise. The density variable in **Fig.4.2** is very sensitive, meaning it must be handled with between 0.01 and 1.00. The higher the number, the more dense the fog. 1.00 fog density will just give the screen a fully grey effect, meaning you can't see anything, so anywhere between 0.01 and 0.2 is a safe number to go with. There are other settings to tamper with this and can be tested thoroughly before applying. I recommend taking off the Fog while developing a game as it is very hard to see the game in the "Scene View" if it's turned on.

2. Lighting : Unity has excellent built in lighting tools to avail of. While creating a Game Object inside of the project Hierarchy, you can select the "Light" option. There are many different options to make a light here such as point lights and directional lights. Entropy uses mostly point lights as they are excellent for lamps throughout the game and give it an eerie feel. There is a colour wheel to make some beautiful looking colours for any type of game. I chose a wide variety such as green, yellow, blue, red, orange and purple. These are used as "markers" through the game and help with the **Mini-Map** functionality which is discussed later on in the Mechanics section of the document.

4.2.3 Game Controls

Unity has built in game controls that can be viewed in the "Preferences" window which can be viewed under the "Edit" sub menu. These are basic tools used when a player controller script is applied to a certain Game Object. To create a Game Object, simply right-click in the hierarchy and create anything, such as a cube. When the player controller script is applied to this object, we can work on the controls. To view any scripts discussed in this document, they are all referenced in the GitHub repository referenced at the beginning and at the end of this document.

Movement : A basic movement system is implemented into the game for the player. The W, A, S and D keys are used to move the player according to the direction on the keyboard. Arrow keys can also be used, as implemented by Unity's basic input tools. The player can jump using the space key, and gravity is applied to the jumping mechanic for a realistic feel.

Looking : A **Mouse Lock** system is implemented into the game as well. Once the game is loaded up, the mouse will lock to the middle of the screen. This gives the camera free access to the players mouse control, therefore making the character look wherever the player moves the mouse. This is quite a smooth system and works very well. This code is implemented in a singular script and must be attached to the **Main Camera** in your scene. This can be done by dragging the script on to the Main Camera, much like dragging the player controller script on to the Player Game Object.

Figure 4.3: Mouse Lock Script

```
void Start()
{
  // get camera component
  cam = GetComponent<Camera>();

  // Locks Cursor to middle of screen
  Cursor.lockState = CursorLockMode.Locked;
}
```

Whenever the mouse must be unlocked, for example in the pause menu when the player presses the escape key, the .Locked variable must be changed to .None. Controlling the mouse this way gives the game a smooth experience.

To view any additional controls, please refer to **Fig.4.7** later on in this section.

4.2.4 Menus and User Interface

Designing menus and a suitable user interface in Unity is a relatively simple task compared to any other section listed here. It does not require much coding, and once one menus coding aspect has been completed it can be copy and pasted on to any other menu.

1. Menus : The main items to note about developing menus are **Navigation** and **Scene Management**. There are many ways to navigate between menus, but the main method I opted to use was disabling game objects by setting them as active or inactive upon a button click. A simple code snippet is demonstrated on how to do this :

Figure 4.4: Menu Navigation Code

```
public void ToMainMenu()
{
    tipMenu.SetActive(false);
    mainMenu.SetActive(true);
}
```

This code represents disabling the help menu to activate the main menu where all of the main navigation buttons are. We can see clearly in the screenshot that the **SetActive** variable returns false for one menu and will return true for the other menu, replacing what will be shown on the main Canvas on the screen. If the user were to click the button to return to the main menu in this case, clicking it would activate the **ToMainMenu()** function.

The same for scene navigation can be seen in the screenshot below. It is important to navigate between scenes as it reduces overall processing power instead of creating one large scene which contains every bit of content inside of a game.

Figure 4.5: Scene Navigation Code

```
public void MainGame()
{
    // Load main game
    SceneManager.LoadScene("StarterScene");
}
```

This code represents swapping between scenes using the **SceneManager.LoadScene**

function. This is a built in Unity function which should be imported on the top of the script as **using UnityEngine.SceneManagement;**. Any scripts containing scene management should contain this import.

2. User Interface : The user interface of a game relates to the structure and positioning of all of the items on the screen. There are two main aspects to this in the game : The **Main Menu** screen and the in-game **HUD** (Heads Up Display, meaning what the player can see on the screen at all times).

Figure 4.6: Main Menu Interface

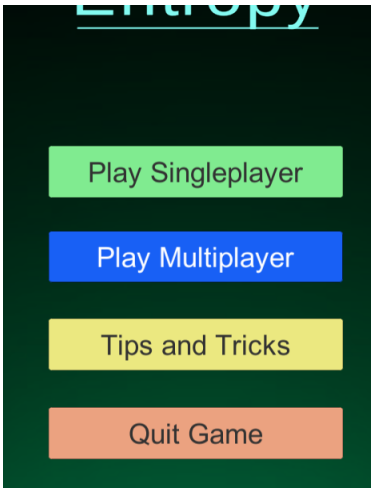


Fig.4.6 shows the main section of the main menu screen inside of Entropy. It holds four buttons : **Play Singleplayer**, **Play Multiplayer**, **Tips and Tricks** and **Quit Game**. Their functionality is explained below.

1. Play Singleplayer : This menu will load up the game in Singleplayer mode. The player will spawn as normal and can play the game.

2. Play Multiplayer : This menu will load the game with the Photon scripts and Game Objects. This will allow the player to be duplicated upon loading in and both players should be able to move around simultaneously. This function unfortunately could not be tested effectively as mentioned before.

3. Tips and Tricks : This menu will load up an informative menu, explaining the game mechanics to the players. It is a simple menu containing text objects in a clear yet concise menu.

4. Quit Game : Very self explanatory. This will quit the game if clicked.

This menu is not cluttered in any way and gives the player a nice interface to work with upon opening the game. Each of the functionalities listed in **Fig.4.4** and **Fig.4.5** are applied to all of these buttons.

The **Heads Up Display** can be seen in **Fig.4.7** below. It shows all of the items clearly on the screen and will be explained below the figure.

Figure 4.7: Entropy's Heads Up Display

1. Top Right Corner : This is the Mini-Map, which can be found in the Mini-Map section. It shows players their location and is done by using a separate camera above the players head that follows them around. All of this is explained in the section below.

2. Top Middle : This text holds the players current objective in the quest they are embarking on. This is described in detail in the Quest System section.

3. Bottom Right Corner : This text is to give the player tips on what the controls are inside of the game. This menu can also be seen in the middle of the screen, labeled Controls.

4. Bottom Left Corner : This area contains several items.

4.1. The topmost item is the **Skill Ready Text**. This shows the player when their weapon skill is ready. This can be seen in the Skill System section.

4.2. The second item is the **Potion Count Text**. This shows the players potion count, described in the Potion and Chest System section.

4.3. The third item is the players current **Health**, shown by a health bar. This is described in the Level and Stat System section.

4.4. The final item in this area is the **Experience** value that the player holds.

This is explained also in the Level and Stat System section.

The middle of **Fig.4.7** also shows two in-game menus which show the **Controls** and the **Stats** of the player. These will be discussed in the Main Development Stages of the document.

4.2.5 Character and Enemy Design

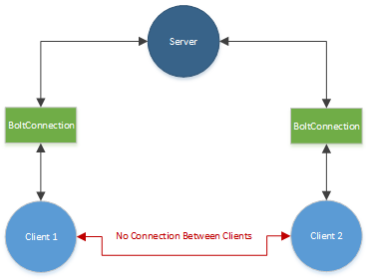
To create a "spooky" atmosphere to Entropy, proper enemies were necessary to create this feeling. Two types of enemies were used from the Unity Asset Store : **Skeletons** and **Zombies**. A proper character design for the player was not necessary due to the fact they could not see themselves through the first person camera. However, if multiplayer were to be implemented with full functionality in the final design of the game, character design would be necessary to use. These are easy to implement into the project as they come with a series of **Animations** and **Colliders**.

The reasons for not designing characters with Entropy were due to the fact that this takes up a lot of time and requires great artistic skills. Some of the main goals of this project was to demonstrate coding abilities and time efficiency. This is one of the major downsides to developing a game alone in Unity as it is very difficult to learn all of the aspects alone. This is why the Unity Asset Store is available to people, so that they can take their mind off of the tasks they personally find difficult and can get assets for free with no licensing required.

4.2.6 Photon Setup with Multiplayer

In order for Photon to work, a server must be set up first. When loading up the application in Unity, scripts from the **PhotonScripts** folder in the GitHub repository will all connect to the servers in synchronization. The connection will be established with your hosting server (in this case the Europe servers), and the player will connect after establishing and allowing that connection to go through.

Figure 4.8: Client-Server Model with Photon



In order to implement Photon, I watched a series of YouTube videos from an excellent channel hosted by InfoGamer. Scripts were added to the project with the videos guidance. They work together to connect to the server and initiate a new "Photon Player" as it is called. This creates another player that can be controlled by the second client that wants to connect to the game via the Multiplayer option in the main menu as described above.

The functionality for the Multiplayer was implemented in the early stages of development in a different scene than the main scene where the game was created. This was because it was very easy to implement into a new scene so in the late stages of development it was added for further testing.

4.3 Main Development Stages - Mechanics

This section is where the real development of Entropy truly begins. The main purpose of the application is to lay down all of the mechanics so that future enemies and quests can be added to the game with ease. Developing the mechanics demonstrates problem solving which was one of my favourite things to do while working with Entropy.

4.3.1 Combat System

The enemies around the map will attack the player when they are in close proximity. They deal damage over time, and all have different values depending on the types of enemies. There are two main types :

1. Zombies

2. Skeletons

Skeletons have 4 different types to them : Small, Medium, Large and Boss. These increase in size to differentiate them from eachother. They also have different health values and damage values for difficulty variance.

Zombies have 2 different types to them : Small and Medium. These work the same as the Skeleton enemies. They are slightly tougher in comparison to the Skeletons however, as they are considered "mini-bosses".

All of these files can be found at the Controllers directory of the GitHub repository, showing the variations between enemy stats.

The player can attack the enemies using the '**Q**' key or the '**R**' key which will unleash a special attack, explained in the **Skill System** section. The player will do a set amount of damage to each enemy, but as their health varies as mentioned above it will take more hits to defeat them.

Figure 4.9: Player Attack Function

```
// attack delay
if (Time.time >= nextAttackTime)
{

    // if press Q, do attack animation and damage if in range of enemy
    if (Input.GetKeyDown(KeyCode.Q))
    {
        // sets sword position relative to our main camera, as to follow rotation
        swordSwing.transform.rotation = Quaternion.Euler(40, Camera.main.transform.eulerAngles.y, 0);

        // Delay for 0.1 seconds so we can reset the sword position before attacking again
        Invoke("Delay1", 0.1f);

        //swordSwing.transform.rotation = Quaternion.Euler(40, 0, 0);
        Attack();

        // Delays attack accordingly (trial and error)
        nextAttackTime = Time.time + 0.4f / attackRate;

        SoundManagerScript.PlaySound("Swing");
    }

}
```

A lot is happening in **Fig.4.9**. Firstly, the main functionality is the **Input.GetKeyDown** function. This can be used in Unity for any type of key you wish to add, meaning pressing any key will activate the function inside the code block you have created. It is particularly useful for "developer tools" as these were used during the development process for testing phases such as healing the player or increasing their level automatically.

The smaller functions inside **Fig.4.9** refer to all of the extras needed in order to attack such as the sword animation which rotates with the camera, a delay function which will play every 0.1 seconds which prevents the player from attacking too fast, the actual attack function which will damage the enemy and the sound effect used when swinging the sword. You may choose to put whatever you want in this function as long as it is logical to what you are doing. The function for the Skill that the player has is the exact same as this, except it alters the players health to increase it and will deal more damage to the enemy.

If the player happens to die to an enemy or enemies, all of the remaining enemies will have their health restored to full health. They can also teleport back to the mission area if they happen to die by pressing '**Enter**' only if the player has progressed that far in the game, otherwise they will respawn as normal in the regular open world area. This is done with a series of "**boolean**" value checks.

Figure 4.10: Boolean Check

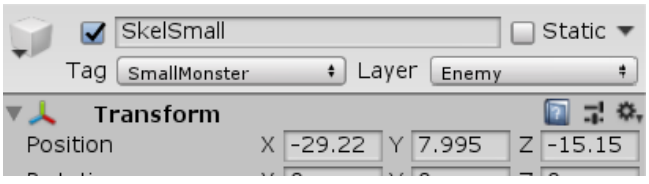
```
void Update()
{
    if (active == true)
    {
        if(Input.GetKeyDown(KeyCode.KeypadEnter))
        {
            text.GetComponent<UnityEngine.UI.Text>().text = "Current Ob
            areaText.GetComponent<UnityEngine.UI.Text>().text = "Area :
            Move2();
            active = false;
        }

    }
}
```

In **Fig.4.10** we can see that if the **active** variable is true, we can press '**Enter**' to teleport back to the area we were in. It is then set to false, meaning there are error checks and the player can not teleport when they feel like it.

This creates a certain difficulty to the bosses in the game, and prevents the player from just respawning and killing their enemy. It creates an emphasis on "**grinding**", which means to kill enemies repeatedly in this case, and gain more experience to level up. More details on leveling up is described in the **Level and Stat System** section.

The important part to note about this section is how to register the enemies as "Enemy's". Inside of Unity's engine there are items called "**Layers**". Adding layers into a game gives you endless possibilities as you can expand to amazing functionality and diversity, allowing certain items to be interacted with other items.

Figure 4.11: Layers in Unity



It is important to label any enemies as this specific layer due to how the attack functionality works. Creating a layer can be done by clicking on Layers and going to "Add Layer". This should be done for every type of object you have in your game including Player, Ground and Scenery.

Figure 4.12: Attack Register on Enemies

```
void Attack()
{

    // animator.SetTrigger("Attack");

    // array for our enemies in enemy tag layer
    Collider[] hitEnemies = Physics.OverlapSphere(attackPoint.position, attackRange, enemyLayers);

    // if several enemies are in range we can hit all at once
    foreach(Collider enemy in hitEnemies)
    {
        Debug.Log("Hit " + enemy.name);
        enemy.GetComponent<EnemyStats>().TakeDamage((attackDamage / 2) + PlayerHealthScript.damage);
    }


}
```

The **Attack()** function in **Fig.4.12** shows clearly that the enemies are listed in an **Array** labelled **Collider[]**. Simply put, this function creates a circle around the weapon wielded by the player and if an enemy in the enemy layer section interacts with this circle, they will be damaged and the Attack() function is called. There is a simple **Debug.Log** function showing that this actually occurs in the Unity editor for testing purposes. This is a very useful function to place in various areas of your game as it shows that the function is actually happening and isn't returning null.

4.3.2 Skill System

The weapon in the game has a skill that can be activated with the keyboard shortcut '**R**'. This has a cooldown of 3 seconds when used, and will drain the health of an enemy and deal more damage to it, giving the player 25 health every time it is used. This is done by simply adding the health to the players stat and updating the health bar slider variable. The cooldown works in the way that every time it is activated, the key can't be pressed again until the timer of 3 seconds bypasses. There is a text object in the games UI that symbolizes this as well, so it will notify the player when the '**Drain Skill**' is ready to be used.

Figure 4.13: Player Skill Function

```
if (Input.GetKeyDown(KeyCode.R))
{
      currentHealth += 25;
      healthBar.SetHealth(currentHealth);
      nextAttackTime = Time.time + 12f / attackRatePower;
      cdTimer.GetComponent<UnityEngine.UI.Text>().text = "Skill : Not Ready";
      if (currentHealth >= maxHealth)
      {
          currentHealth = maxHealth;
          nextAttackTime = Time.time + 12f / attackRatePower;
      }
}
```

We can see in **Fig.4.13** that you can only press the **R** key when the time has elapsed for the cooldown. Once this occurs, the **PowerAttack()** function plays in another script, which will absorb the health and deal extra damage. The **currentHealth** variable raises by 25, showing this health increase for the player.

4.3.3 Health System

In order for the Health system to work, a slider must be created for the player. After following a series of videos by **Brackeys Tutorials**, it was very simple to create this health bar with a slider attached. A basic image must be placed over the previous image to symbolize the "Green" and "Red" parts of the health bar, in which green symbolizes remaining health. Using the Slider functionality you can increase or decrease the size of the green image to overlap the red image and give it the effect that the players health is increasing or decreasing.

Figure 4.14: Player Health Bar

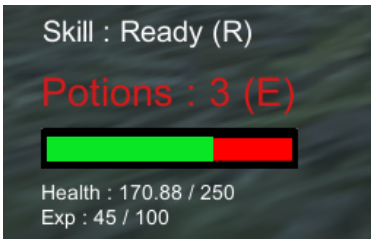


Fig.4.14 shows the health remaining and the health taken, and we can see that the figures to in fact match the health bar. In Unity's editor you can create Sliders as game objects and adjust minimum and maximum variables for them. This can create very diverse health bars for players or even for enemies.

4.3.4 Level and Stat System

A leveling system is designed in the game. The player has a certain experience value, and every time they defeat an enemy they will gain experience. Depending on the type of monster such as Skeletons, Zombies and their sizes, the player will gain less or more experience. There are 10 levels designed for the game, and the player has 3 stats to accompany this level :

1. Health : A base value of 250 health is in the game, and this will increment by 30 every time the player gains a level. This is useful for survivability. If the players health reaches zero, they will die and respawn. Respawning teleports the player to the previous area. There is also a health regeneration function in the game to increase the health every second.

Figure 4.15: Player Health Regeneration Function

```
// health regeneration
if (currentHealth <= maxHealth)
{
    currentHealth += regeneration * Time.deltaTime;
    healthBar.SetHealth(currentHealth);
    if (currentHealth >= maxHealth)
    {
        currentHealth = maxHealth;

    }

}
```

2. Damage : The player has a base damage value, which will let the player deal a certain amount of damage to enemies. The higher the players damage, the more damage they deal with attacks. The Drain Attack is also affected by this variable, and is enhanced even more than the basic attacks if the player decides to level up. This Attack is described below in the "Skill System" section of this file.

3. Defense : The player has a base defense value, which will reduce the damage an enemy does to them. This works by dividing the damage variable of the enemy by the defense stat of the player. This gives the player a great incentive to level up before taking on harder enemies.

Here is a quick code snippet on what actually happens when the player levels up :

Figure 4.16: Level Up Function

```
public void LevelUp()
{
    // get variables to increase stats
    PlayerHealthScript.maxHealth += 30;
    PlayerHealthScript.currentHealth += 30;
    PlayerHealthScript.damage += 8;
    PlayerHealthScript.defense += 2;

    // set max health properly
    HealthBarScr.maxHp += PlayerHealthScript.maxHealth;
}
```

It is relatively easy to understand what is happening here. Each of the variables in the **PlayerHealthScript** get increased accordingly.

It is necessary for the player to level up to complete the game due to the way the defense variable works. As it is divided by the enemies damage, it creates great advantages for the player, but if the player is too low of a level they will die too quickly in certain sections.

Figure 4.17: Enemy Damage Function

```
public void TakeDamage(int damage)
{
    currentHealth -= damage / defense;
    Debug.Log("Damage Dealt = " + damage);
```

Fig.4.17 shows how the damage works for enemies. This works the exact same for the player too, and as explained before will give the player incentive to keep working to level up.

4.3.5 Potion and Chest System

Potions can be used by the player to increase their current health. In order for potions to be used, the player must have some in their inventory first. Potions are obtained by either killing monsters, leveling up, completing Quests or by hitting chests. Chests are scattered around the two areas and can be attacked to give the player 5 potions. Players can use potions by pressing the '**E**' key.

Figure 4.18: Use Potion Function 1

```
if (Input.GetKeyDown(KeyCode.E))
{
    // if more than 1 potion, call UsePotion
    if (potionCount > 0)
    {
        UsePotion();
        SoundManagerScript.PlaySound("Potion");
    }

    // if 0 potions or max health, call DontUsePotion and do nothing
    if (potionCount == 0)
    {
        DontUsePotion();
    }

}
```

In **Fig.4.18** we can see that when the **potionCount** variable is above 0, a potion can be used. If it is 0, a function returning null is called and there will be no effect. It will also play the sound effect for the potion noise. The **UsePotion()** function is called, which is shown below. This works the same

way as using the Skill or the regular Attack for the player in which the '**E**' key is used to use the potion, however there is no cooldown on it.

Figure 4.19: Use Potion Function 2

```
// call this function if potion count is over 1
    public void UsePotion()
    {
        if (potionCount >= 1)
        {
            currentHealth += 20 + (PlayerExperience.level * 2);
            healthBar.SetHealth(currentHealth);
            potionCount--;
        }

        if (potionCount <= 0)
        {
            potionCount = 0;
        }

        // set health
        if (currentHealth >= maxHealth)
        {
            currentHealth = maxHealth;
        }
    }
```

The health of the player is restored by 20 plus the level of the player multiplied by 2. For example : The players level is 4, if they use a potion at this level they will heal 20 + (4 x 2) health, equaling 28 health. This is so that potions will not lose their effectiveness the higher the level of the player. This is an important game balance issue that was addressed as bug testers had no incentive to keep leveling due to this. Calling this function will deduct 1 potion from the potion count.

4.3.6 Mini-Map System

A basic mini map function is designed for the player. It is done by creating a separate camera, which will hover over the player and change direction as well depending on where the player is facing. This is handy for pinpointing enemies, and due to the way the lighting system works in the game, it can become even more accurate for the player. The code snippet for this is featured below :

Figure 4.20: Mini-Map Code

```
void LateUpdate()
{
    Vector3 newPosition = player.position;
    newPosition.y = transform.position.y;
    transform.position = newPosition;

    // for camera to rotate with player
    transform.rotation = Quaternion.Euler(90f, player.eulerAngles.y, 0f);
}
```

The **LateUpdate()** function is used as it must be called after the Update() function has been called. The reason this must be done is because the player moves and the main camera updates, but we can't simultaneously update another camera to match it perfectly. It is about 1 frame behind, meaning it has time to adjust to the sudden movements the player might do. It must be updated after the player moves as it is following those movements. This helps reduce the lag in the game as well, and allows for better frame rate.

Figure 4.21: Mini-Map Screenshot

It shows clearly how the lighting works for the players advantage. It is very clear and creates a nicer looking User Interface as well.

4.3.7 Quest System

In the game there is a single quest that the player can activate. The objective of the game is to "Find NPC(Non-Playable-Character) Toland". Once the player has found the NPC, they can proceed with the quest. They are teleported into the mission area, and are given the task to eliminate all of the enemies and save the other NPC. Once the quest is finished, the player can return to Toland and receive experience and potions. This is to aid the player in future quests that may be implemented into the game.

The quest system also makes use of the User Interface elements frequently. Every time an action is done in the game involving the NPCs, the current objective must be updated for the player so that they know what they are doing or what they need to do next.

Figure 4.22: Update Objective Function

```
void Update()
{
    if (active == true)
    {
        if(Input.GetKeyDown(KeyCode.KeypadEnter))
        {
            text.GetComponent<UnityEngine.UI.Text>().text = "Current Ob
            areaText.GetComponent<UnityEngine.UI.Text>().text = "Area :
            Move2();
            active = false;
        }

    }
}
```

This code says if the player hits the '**Enter**' key on their keyboard, they will move with the Move2() function which will transport the player. The main thing to note here though is the **text.GetComponentUnityEngine.UI.Text().text** element is being changed in this if statement. This changes two text elements : The objective text and the area text. This is a perfect example of how the UI elements are manipulated by key presses for the quest system. This works the exact same way as the boolean check functionality for teleporting the player after death, as the original function for teleporting to the main area also requires the same input. These are both regulated very carefully with several boolean values so the player does not have the ability to teleport whenever they wish.

4.4 Late Development Stages - Refinements

When all of the main mechanics were finished inside of the game, some extra fixes had to be applied to refine them as would be done with any application, game or not.

4.4.1 Multiplayer Adjustments

To apply **Photon** to the game it had to be transferred from the "Multiplayer Scene" to the "Main Scene". In order to do this, all of the instructions in the videos referenced before were followed again and everything that was done was transferred over.

Upon entering the game, the player must click the "Play Multiplayer" button in order for the Photon servers to initialise. This would enable a game object called

"**GameSetup**" which wouldn't originally happen in the Singleplayer section. This essentially duplicated the player and each could be controlled individually.

4.4.2 File Organisation

This was done during the whole development of the game, but a necessary clean up is always necessary. Organising files into proper directories and cleaning un-used code should always be ensured before uploading a project during its final stages. All of these file structures are seen in Figures **1.2** and **1.3** in the Introduction section.

4.4.3 Bug Fixing

Fixing bugs mainly includes removing un-used code, developer tools as mentioned before and finishing touches to any strange mishaps that happen in a game. The main items that had to be fixed at the end of the game were the **Game Balance** issues, which refers to the difficulty of the game. The game was too difficult at first glance, and it required rigorous testing to fix this up. This was done in every development stage of the game as certain functionality such as the Combat System required enemies to at least be tolerable and ensure that they could be defeated, or did not defeat the player in a single hit.

Chapter 5

System Evaluation

In order to evaluate the design behind Entropy, we must view how well the whole system works together. Are all of the functions inside of the project necessary? Or do they fit together in harmony? This chapter will discuss the application as a **Full System** in order to find out if the project can be evaluated as a robust piece of software.

5.1 Did Entropy Meet Expectations

This section is devoted to comparing and countering the objectives set out in the **Introduction** section of this document. Did Entropy meet the expectations that I had for it when I started to develop it? Yes and no. There are always issues with applications and there is no shame in admitting that, but there is no issue in praising the good parts about the application either.

5.2 Stability and Performance

This section relates to the decision to use **Unity** as the main Game Engine in developing Entropy. Due to the fact Unity is used with more Indie game developers (Indie referring to smaller game companies rather than AAA game companies such as EA and Activision), it is designed to be more catered towards smaller games as smaller groups of people of about 4-8 people would generally work on games created in Unity. Bigger game companies use bigger game engines that require higher performance computers. Due to my computer being middle of the range I can safely say that Entropy did not cause any issues in terms of stability as it did not experience any "crashes" or speed and performance issues.

5.3 Robustness

Robustness refers to how all of the systems worked together to prevent unexpected outcomes from occurring. Was there enough error handling in the application so that nothing seemed out of the ordinary?

I believe the answer to this question would be yes, the software is Robust. It was tested thoroughly throughout the development stages and it shows as there are very few bugs in the game (Bugs refer to mishaps that may occur in the game and are unintentional). The game remained stable during development due to the fact Unity has an excellent game engine for handling errors and performance issues, and it is very noticeable when your computer starts to experience performance issues with it. These were very easy fixes due to the fact they were found as soon as they occurred and nothing was left as an issue throughout the whole development stage in terms of performance and behaviour.

5.4 Limitations

This section is designed to show what Entropy is capable of, and what it is not capable of as of now. These points will be countered in the Issues and Future Developments section in this chapter.

5.4.1 Strengths

Entropy has a few strengths :

1. Stable : Relating to the **Stability and Performance** section mentioned above

2. Error-Handling : All of the errors encountered during the development stages were countered with boolean check values and collider prevention systems to enforce rules for the game such as out of bounds bugs or unintentional flaws to occur.

3. Complex Yet Simple : Even with vast amounts of scripts in the game, all of the code is very easy to find within the projects directories. This is evident in the **GitHub Repository**.

4. User Interface : The User Interface of Entropy was very simple and has an excellent, easy to use interface. It explains everything inside of the game (and even outside) on how to play the game and what the controls are. These are important to place inside of a game because not everyone will read any outside documentation related to the game.

5.4.2 Weaknesses

Entropy does not have as much external software as I had hoped it to have. This includes the **Databases** and **Animations** sections referenced in the Introduction part of this document. This is not necessarily a weakness of the application, but if the game were to be developed further then it would become a serious issue as the code would become very heavy and not reusable at all, causing big issues in performance. Databases help to maintain this performance greatly.

5.5 Issues and Future Developments

This section is combined due to the fact any of the main issues in the application upon release are also the items that will be added into Entropy in future. This section is written in a more personal manner due to the fact it comes straight from the developers point of view.

5.5.1 Current Issues

1. Executable : For some reason the .exe does not take a liking to the Mouse Look system. It works fine when actually editing in Unity but the mouse stays on the middle of the screen. Another issue is the camera view in the .exe. Attacking still works and kills enemies, but the sword won't swing the way it is intended to, so the animation looks off.

2. Multiplayer : The Multiplayer was not working correctly as I had no access to another machine to test the functionality due to COVID-19 restrictions. When duplicating the player, I could not control the other player with a standalone build, which was recommended to use in any tutorial videos I followed and websites I viewed if another machine could not be used. The functionality of Multiplayer was evident, as the player was duplicated and upon using the command in scripts called "**PhotonView.IsMine**", it was working properly as I had no access to the other players controls. If I removed this component however, I did. So this shows that I had made the Multiplayer properly, but I had no idea where to use the other standalone build to even access the controls as it was not appearing upon developing on a single machine. If another machine had been used and connected through the client, it may have been possible to access this "Unknown Player" as I called it.

3. Health Bars : This issue is not major, but the health bars on the enemies all seem to be connected to the same variable. If one enemy dies, they all die. This is strange as I linked this to the enemies health variable, and without the health bars they all die separately, but putting a functioning bar on them will kill all of them if one is killed. It is not a major issue as the mechanics of the game still work, it just means people have a harder time visualising when an enemy will actually die. They must pay extra attention to how big the enemy

is in order to calculate the level of difficulty it holds compared to the previous enemies that have been encountered in the game. In order for the game to work properly I had to remove this feature entirely, however it is still evident in the code.

5.5.2 Future Developments

The addition of many items would make Entropy a very versatile and strong application, many of which have been mentioned before.

1. Proper Multiplayer and Teamwork : A good multiplayer system where several players can join together into one big mission has been one of the beginning items that I have wanted to be in Entropy. Players would be teleported to a mission area and a bigger boss would be available for them to kill. Additional skills to enforce proper multiplayer mechanics such as healing other players, dealing extra damage, taking "agro" which refers to who the enemies will attack specifically, and ranged attacks are all items I wish to be part of a great multiplayer scene in future.

2. Databases : Two separate databases for enemy stats and quests would be ideal as a future development for Entropy. This will give limitless bounds as to how many enemies and quests you can add into the game, Creating databases also gives a great sense of organisational skills and might give people a good impression towards the developers of the databases.

3. Extra Maps : This item is a pretty obvious one due to the fact Entropy is designed to be a "live-service game", meaning that it will be developed constantly over a period of a few months or years. This gives players more and more content and will keep them interested in the game for years to come.

4. Bug Fixes : If Entropy were to be released properly in its current state, just like any game there will be bugs found. Bug testing is very important in a game but from viewing AAA games we see that because millions of people play these games, bugs appear anywhere and everywhere. There is always a possibility that bugs will arise while developing a game.

Chapter 6

Conclusion

In linking with the **System Evaluation** section, I have learned many things regarding development of a project like Entropy . It is important to note these while developing the project and it is very easy to tell if you have a sense of achievement while doing so. Every time I created a new function that worked first try or even after several tries, it held a great sense of achievement. Fixing bugs that were outlasting gives that sense of achievement as well.

Entropy did in fact meet all of its expectations. It delivered a very stable and economical application and maintained complex mechanics. The use of Unity Engine combined with C# was overall an amazing combination of software and programming language to use with Entropy. The flexibility and diversity of both of these components allowed for capabilities that I initially didn't think were possible, but due to the large amount of people that use Unity and C# it was amazing to find the vast amounts of documentation for these online which helped the development stages massively.

6.1 Learning Outcomes

Important items to note that I have learned include :

1. Developing a suitable HUD and UI/Menu system - The HUD is the interface of the characters screen. It must be user friendly, and making one that accommodated that was important. All of the menus had to be designed nicely too.

2. Organizational skills - My files were messy during the first month of development in the project. I quickly adapted to this and had made the file structure extremely organized.

3. Timing - It was important to space out the development as taking on too

many tasks at once would become difficult.

4. Depth - Making a complex application just for the sake of it is not always the main goal. Anything and everything in this application is necessary, and there are no extra items in that would have increased development time but had little to no impact on the application itself. Everything was efficient.

6.2 Improvements

How did I improve my coding skills during developing Entropy?

I started to realise over time that coding is more-so about problem solving. It took me quite a while to realise this aspect of it. There were a lot of mathematical equations involved in developing Entropy such as health, damage and defense values. These were all very important aspects of the Combat and Health systems.

Developing time management skills as mentioned before were a very important part of the development of Entropy. Most students will procrastinate until the very end of when a project is due and it becomes impossible to concentrate when the time comes to develop the application due to the fact time is no longer available and is of the essence. I admit while developing Entropy I did not start right away as intended and began around Christmas of 2019, finishing up in May 2020. This however did not impede the progress of the project and I was able to manage my time in the remaining 4 months I had left.

Around the beginning of March of 2020, a pandemic occurred due to a virus outbreak. This caused a worldwide "Lockdown" and made some serious changes to lifestyle occur. This required some adjusting, but it did actually help with developing Entropy in a sense. I realised that people work in hard environments all over the world every day of the week. Having to develop the project in isolation has given me a great insight into what other people might be enduring out there in the world, and I feel as if I understand it all a little bit better. It was not an ideal environment to work in, but adjusting to something that you are not comfortable with is very important when being released out into the working world, such as finding a new job. I believe now that I will be able to cope much better regarding the situation that happened and is currently ongoing in the writing of this document.

6.3 Conclusion

All in all the project was incredibly fun to develop and experiment with. I felt that when I was developing the game, new ideas were constantly flowing into my head. I had made a plan as well but as you develop a game, certain parts seem to come together and new ideas can be formed after visualising clearly what has been done in front of you. Especially since this is the first game I have developed, knowing what order to do things in and what should have been added in the development plans originally are much clearer to me now, and I have learned some very valuable lessons from doing this project. I now understand the hardships that are overcome by games developers, and being an avid game player who has criticized the work of other game developer companies I have learned that adding simple ideas into games can actually lead to very daunting tasks.

I would like to thank my supervisor Dr. Dominic Carr for all the great feedback and assistance throughout the development of the project. A lot of ideas were discussed in detail together and I feel that Entropy would not be the game it is now without our combined teamwork.

6.4 Documentation

All documentation for the project can be found at
https://github.com/moecherry99/AlexCherryFinalYearProject as mentioned at the start of this document.

6.4.1 Downloading and Running the Application

1. Clone the repository at
https://github.com/moecherry99/AlexCherryFinalYearProject into a new folder on your Desktop.

2. Move to the Executables folder and select the .exe file.

3. If you wish to edit, open in Unity v2018.4.9f1, any version of Visual Studio will work for editing code. You must select the full folder and open it in order for it to work.

Bibliography

[1] Unity. Unity documents. `https://docs.unity3d.com/Manual/index.html`, 2020-05-04.

[2] Stack Overflow. `https://stackoverflow.com/`, 2020.

[3] ABM Moniruzzaman and Dr Syed Akhter Hossain. Comparative study on agile software development methodologies. *arXiv preprint arXiv:1307.3356*, 2013.

[4] Steam. `https://store.steampowered.com/`, 2020.

[5] Wallace Jackson. *Beginning Java 8 Games Development*. Apress, 2014.

[6] NetBeans. Netbeans. `https://netbeans.org/`, 2020.

[7] Ralph Barbagallo. *Wireless game development in C/C++ with brew*. Wordware Publishing, Inc., 2003.

[8] Will McGugan. *Beginning game development with Python and Pygame: from novice to professional*. Apress, 2007.

[9] Terry Norton. *Learning C# by developing games with unity 3D*. Packt Publishing Ltd, 2013.

[10] Joanna Lee. *Learning unreal engine game development*. Packt Publishing Ltd, 2016.

[11] Johan Andersson. Terrain rendering in frostbite using procedural shader splatting. In *ACM SIGGRAPH 2007 courses*, pages 38–58. 2007.

[12] Microsoft. Visual studio ide. `https://www.comparex-group.com/web/microsites/microsoft/products/development/ms-visual-studio/ms-visual-studio-2019.htm`, 2020.

[13] Photon. Photon engine. `https://www.photonengine.com/`, 2020.

[14] Chris Totten. *Game Character Creation with Blender and Unity*. John Wiley & Sons, 2012.

[15] MySQL. Sql databases. `https://www.mysql.com//`, 2020.