



**UNIVERSITY OF PIRAEUS**  
**SCHOOL OF INFORMATION AND COMMUNICATION**  
**TECHNOLOGIES**  
**DEPARTMENT OF INFORMATICS**

**Thesis**

Thesi's Title	(English) Rise of the Village Hero: A Roleplaying Game: Character Attributes, 3D animation, Virtual World Implementation, Sound Camera and Characters' Movement.  (Greek) Rise of the Village Hero: Ένα παιχνίδι ρόλλων: Χαρακτηριστικά χαρακτήρων, Υλοποίηση 3D κίνησης, Ήχου, Εικονικού κόσμου, Κίνηση Κάμερας και Χαρακτήρων.
Student's Full name	Georgios Samaras
Father's name	Basileios
Registration Number	Π18134
Supervisor	Themistoklis Panagiotopoulos, Professor

Delivery Date September, 2024

## Copyright ©

---

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν αποκλειστικά τον συγγραφέα και δεν αντιπροσωπεύουν τις επίσημες θέσεις του Πανεπιστημίου Πειραιώς.

Ως συγγραφέας της παρούσας εργασίας δηλώνω πως η παρούσα εργασία δεν αποτελεί προϊόν λογοκλοπής και δεν περιέχει υλικό από μη αναφερόμενες πηγές.

## Abstract

---

This Thesis is about a top down roleplaying game (RPG) development in Unity. It has low poly graphics and it's main focus is in the modular and expandable script system, which can save a lot of time on maintaining and developing the game.

## Περίληψη

---

Η παρακάτω διατριβή ασχολείται με ένα top-down παιχνίδι ρόλων προγραμματισμένο στην Unity. Έχει γραφικά χαμηλής πολυπλοκότητας και ο κύριος στόχος της είναι η ανάδειξη του ευέλικτου και αναβαθμίσιμου συστήματος από scripts που έχει η εργασία, τα οποία μπορούν να σώσουν πολύ χρόνο στην συντήρηση αλλά και δημιουργία μίας τέτοιας εργασίας.

## Acknowledgements

---

I would like to thank Professor Themis Panayiotopoulos for the support and encouragement and support throughout this project.

I would like to thank Ioannis Petris for his contribution to the project as its co-creator and co-developer. The other part of this project can be found in his Thesis.

## Table of Contents

---

Copyright ©.....	i
Abstract .....	ii
Περίληψη .....	ii
Acknowledgements.....	ii
Table of Contents .....	iii
Table of Figures .....	vi
1 Introduction.....	1
2 Project Workflow .....	2
2.1 Conceptualization and Planning .....	2
2.2 Pre-production.....	2
2.3 Production .....	2
2.4 Testing and Quality Assurance .....	3
2.5 Polishing and Finalization .....	3
2.6 Pre-Launch, Launch and Post-Launch.....	3
2.7 Retrospective and Future Planning.....	3
3 Running the project.....	4
4 Game Mechanics & Walkthrough.....	5
4.1 Main Menu .....	5
4.1.1 Main Menu Screen .....	5
4.1.1 Settings Screen .....	5
4.1.2 Information Screen .....	6
4.1.3 New Game Screen .....	6
4.1.4 Load Game Screen .....	7
4.1.5 Continue and Quit Buttons.....	8
4.2 Playthrough and UI.....	8
4.2.1 Player's Graphical Interface.....	8
4.2.2 Pause Menu .....	9
4.2.3 First Steps .....	10
4.2.4 Equipment and Inventory Interface .....	10
4.2.5 Quest Interface .....	13

4.2.6	Leveling Up .....	14
4.2.7	Shop Interface .....	16
4.2.8	Portals .....	17
4.2.9	Spells and abilities .....	18
4.3	Rest of the Walkthrough and side Quests .....	20
4.3.1	Priest's Quest .....	20
4.3.2	Mayor's Quest .....	22
4.3.3	Goblin Camp .....	22
4.3.4	Fighting Goblin Camp .....	25
4.3.5	The Goblin Cave .....	27
5	Scripts .....	28
5.1	Core Scripts .....	28
5.1.1	ActionScheduler.cs .....	28
5.1.2	CameraController.cs .....	28
5.1.3	CameraFacing.cs .....	29
5.1.4	CameraLayerToggle.cs .....	30
5.1.5	Condition.cs .....	30
5.1.6	DestroyAfterEffect.cs .....	31
5.1.7	FollowCamera.cs .....	31
5.1.8	IAction.cs .....	32
5.1.9	IPredicateEvaluator.cs .....	32
5.1.10	PersistenceObjectSpawner.cs .....	33
5.2	Abilities .....	34
5.2.1	DelayEffect.cs .....	34
5.2.2	HealthEffect.cs .....	35
5.2.3	OrientToTargetEffect.cs .....	35
5.2.4	SpawnPrefabEffect.cs .....	36
5.2.5	SpawnProjectileEffect.cs .....	37
5.2.6	TriggerAnimEffect.cs .....	38
5.2.7	TagFilter.cs .....	39
5.2.8	ClickTargeting.cs .....	39
5.2.9	SelfTargeting.cs .....	40
5.2.10	Ability.cs .....	41

5.2.11	AbilityData.cs .....	42
5.2.12	CoolDowns.cs .....	43
5.2.13	Effects.cs .....	44
5.2.14	Filtering.cs .....	44
5.2.15	Targeting.cs .....	45
5.3	Attributes .....	45
5.3.1	Health.cs .....	45
5.3.2	HealthBar.cs .....	47
5.3.3	HealthDisplay.cs .....	47
5.3.4	Mana.cs .....	48
5.3.5	ManaDisplay.cs .....	49
5.4	Cinematics .....	49
5.4.1	CinematicEvents.cs .....	49
5.4.2	CinematicsControlRemover.cs .....	50
5.4.3	CinematicTrigger.cs .....	51
5.5	Control .....	52
5.5.1	AIController.cs .....	52
5.5.2	ClickablePickups.cs .....	53
5.5.3	CursorType.cs .....	54
5.5.4	IRayCastable.cs .....	55
5.5.5	PatrolPath.cs .....	56
5.5.6	PlayerController .....	56
5.5.7	Respawner.cs .....	57
5.6	Movement .....	58
5.6.1	Mover.cs .....	58
5.7	Scene Management .....	59
5.7.1	Fader.cs .....	59
5.7.2	Portal.cs .....	60
5.7.3	SavingWrapper.cs .....	61
5.8	Stats .....	62
5.8.1	BaseStats.cs .....	62
5.8.2	CharacterClass.cs .....	63
5.8.3	DefenceOffenceDisplay.cs .....	63

5.8.4	Expirience.cs .....	64
5.8.5	ExperienceAward.cs .....	65
5.8.6	ExpirienceDisplay.cs .....	66
5.8.7	IModifierProvider.cs .....	67
5.8.8	LevelDisplay.cs .....	67
5.8.9	Progression.cs .....	68
5.8.10	Stats.cs .....	69
5.8.11	TraitLogic.cs .....	69
5.8.12	Traits.cs .....	70
6	Conclusion .....	71
7	Bibliography .....	73
8	Abbreviations & article-acronyms table .....	73

## Table of Figures

---

Figure 1.	Project's Decompressed files .....	4
Figure 2.	Main Menu Screen .....	5
Figure 3.	Settings Screen .....	6
Figure 4.	Information Screen .....	6
Figure 5.	New Game Screen .....	7
Figure 6.	Load Game Screen .....	7
Figure 7.	First Steps in Game .....	9
Figure 8.	Pause Interface .....	9
Figure 9.	Pickups .....	10
Figure 10.	Equipment and Inventory Interfaces .....	11
Figure 11	Equipping Items .....	12
Figure 12	First Quest: Save the villager .....	12
Figure 13.	Dialogue Interface 1 .....	13
Figure 14.	Dialogue Interface 2 .....	14
Figure 15.	Leveling Up .....	14
Figure 16.	Attributes Interface .....	15
Figure 17.	Defeating Enemies in Evercoast Village .....	16
Figure 18	Shop Interface .....	16

Figure 19. Quests Window .....	17
Figure 20. Village's wizard. ....	18
Figure 21. Retrieve the wizard's staff .....	19
Figure 22. Equipping Spells .....	19
Figure 23. Casting a spell .....	20
Figure 24. Village's Priest .....	21
Figure 25. Village's Graveyard .....	21
Figure 26. Village's Mayor.....	22
Figure 27 Goblin Camp Cutscene .....	23
Figure 28. Mysterious Ruins .....	23
Figure 29. Secret Door.....	24
Figure 30. Underground Ruins.....	24
Figure 31. The Dark Blade .....	25
Figure 32. Secret Path .....	25
Figure 33. Fighting the Goblin Camp .....	26
Figure 34. Ruins Shopkeeper .....	26
Figure 35. Goblin Cave .....	27
Figure 36. Goblin War chief .....	27



# 1 Introduction

---

This thesis examines a creation process of a top-down role-playing game with low-poly graphics. This odyssey will focus on the design of a modular game structure that can be expanded upon through the use of the in-axis scripting which is created in this project. This paper will also present how writing and maintaining such an organized script will improve the development of the game and the maintenance of its code, as well as allow for more content to be added to the game in the future.

In developing the game, low-poly graphics were used, which allowed making the game more beautiful without deterioration in performance. Such approaches are quite relevant in the indie game development today and help to make assets fast and iterate easily.

At the core of the design and development of the game is the creation of detailed scripting tools that allow various modes of the game such as character AI, quests, characters interactions, and environments to be changed and encased within the scripts. This not only enhances the speed of the task but also provides a systematic approach of development and future modifications by other developers and even modders if any.

In this paper, the architecture of the scripting system is broken down in details, especially at how it relates and integrates with the game core as well as the benefits arising from implementing such a system concerning time, organization and creation of new game contents. Furthermore, it documents the obstacles experienced during the course of tackling a particular task and how they were effectively resolved.

Even though the project is not heavily oriented towards research, it adds to the body of knowledge of game development, by demonstrating real world examples of applying modular design principles in building complex game systems. This completed game can be a concrete example for developing developers as to how their systems, if designed wonderfully, can allow them to create games faster and in a more dynamic fashion.

## 2 Project Workflow

---

How is a video game created? What are the necessary steps to develop a fun and engaging experience? This is the thought process of a game developing project, which is much alike like any other programming project. At the same time there will be my own point of view at the named stage.

### 2.1 Conceptualization and Planning

The starting point of the work is the stages of outline the game, its objectives, and brainstorm the game concepts. Developers produce a Game Design Document (GDD) which incorporates core gameplay, narrative and major characteristics of the game. A team is formed, the responsibilities are distributed and the scope of work is determined. This aspect forms the basis of the entire process of development of the game since, at this point, every participant knows the concept and the aims of the game.

In my case, it was a 2 man project so there was no need for a Game Design Document. Tasks and responsibilities were distributed equally with mine being the Virtual Environment, Animations, Sound, Camera and Movement.

### 2.2 Pre-production

Before proceeding to full-fledged development, prototype is created in preproduction, which is meant to incorporate major gameplay elements without finalization yet. This step involves preparing the technical environment, selecting the tools, and defining the art. The game design focuses on defining the game mechanics, while the application development defines the core structure. The purpose is to evaluate the viability of an idea and ready it for full production.

Likewise, the first Scripts were created and tested, the graphical assets were set on the low poly style and a basic story was drafted and a test scene to try out the functionality had been set up.

### 2.3 Production

The production phase of the game is where all the action begins. The programmers implement the technical aspects and gameplay elements, while the artists come up with the designs, animated sequences and special effects. Level designers construct architectures of the game, while story writers forge plots. Continuous testing and improving the game play experience are carried out during this stage also. Also, the user interface and sound components are designed and added.

In this section most of the game was completed. The scenes were ready, with sound and camera mini cinematics, UI and Menu were created, items and NPC dialogues as well.

## 2.4 Testing and Quality Assurance

With the game being nearly finished, the lengthy verification phase begins. Internal alpha testing is performed to discover significant defects and balance problems. Then beta testing is performed which usually includes outsiders to collect more feedback. Optimization of the performance takes place, in the course of which the general of bugs is eradicated and the game level is improved. For international launches, adequate localization is done on the game.

In this face we distributed the alpha version of the game to our close circle for instant feedback. In this process bugs where found and balancing happened to a point we were happy with the results.

## 2.5 Polishing and Finalization

The polishing phase of game development is centered on improving all aspects of the game. Developers adjust the gameplay mechanics, the level of difficulty, and the graphical and audio effects. The latest performance improvements are made, and the product is readied for submission to the relevant platform holders. At this stage, most people regard this as the last stage in the game development process, and because of this, they elevate the smoothness level of the player experience.

This stage happened for us in conjunction with the Testing phase, as the project was small and the time limited. There wasn't much to improve, but it's place as the last step of the process was understood as some of the last changes could impact much more if they were polished in earlier stages.

## 2.6 Pre-Launch, Launch and Post-Launch

Pre-Launching of the Product – Marketing and External Affairs and Communications Assistants Development. This is when the launch is near and due to this fact, the marketing activities get heightened. The team comes up with ways to market their product, opportunities to reach out to the target audience and activities to welcome their attracting events. Monetization systems (if applicable) are implemented and tested. Besides those steps, the staff of the team completes activities with the aim of establishing and configuring analytics, engaging in getting pages ready for the storefront, and creating support systems for customers. Finally, a whole plan is mapped out with the aim of ensuring the appropriate launch.

Unfortunately we skipped this task as this is a Thesis project for University of Piraeus, so there was no plan for profit (apart from my bachelor's degree).

## 2.7 Retrospective and Future Planning

Once the dust settles, the team holds a project post-mortem in order to evaluate the development process that was employed. Strengths and weaknesses are noted, which helps in the subsequent projects. The team rejoices in their accomplishments and set

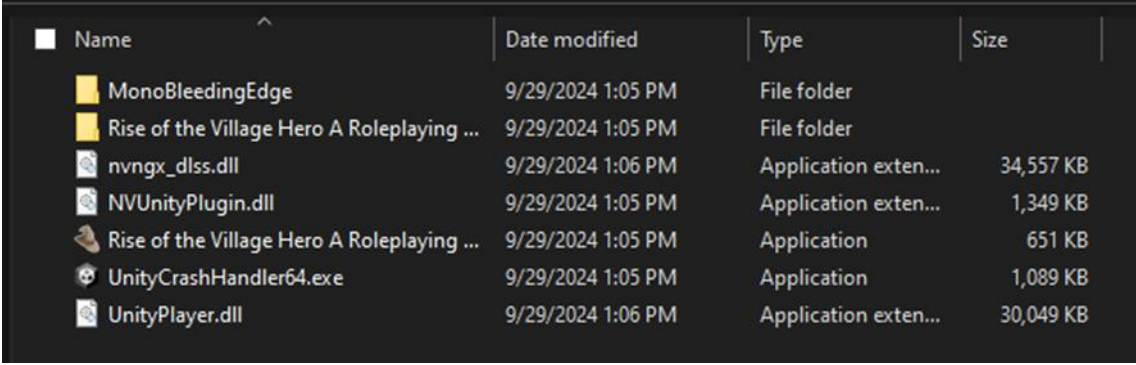
out to figure out what comes next, be it new markets, new titles, or new game ideas. This practice promotes improvement in the subsequent development phases.

In our case we summarized the whole experience, what went right and what was lacking, an important lesson for future projects in our career, which will be summarized at the end of the Thesis. As for expansion ideas, there is a talk with professor Themis Panayiotopoulos for some additions on the project so it can be added to his curriculum as a complete game example.

### 3 Running the project

The project comes in a zip file named Thesis.zip. First step is to be decompressed.

After that there will be a folder with the whole project in it. After copying the folder from the zip file, it should be files in it like this.



Name	Date modified	Type	Size
MonoBleedingEdge	9/29/2024 1:05 PM	File folder	
Rise of the Village Hero A Roleplaying ...	9/29/2024 1:05 PM	File folder	
nvnvx_dlss.dll	9/29/2024 1:06 PM	Application exten...	34,557 KB
NVUnityPlugin.dll	9/29/2024 1:05 PM	Application exten...	1,349 KB
Rise of the Village Hero A Roleplaying ...	9/29/2024 1:05 PM	Application	651 KB
UnityCrashHandler64.exe	9/29/2024 1:05 PM	Application	1,089 KB
UnityPlayer.dll	9/29/2024 1:06 PM	Application exten...	30,049 KB

**Figure 1. Project's Decompressed files**

These are the files Unity builds as a complete game. The final build ensures that:

- 1 The source code is converted into executable machine code, as well as optimizing the assets and scripts for better performance.
- 2 The code can be prepared for many different platforms (PC, Consoles, mobile devices etc.) and a variety of different operating systems and hardware.
- 3 Packages all game assets (graphics, sound etc.) efficiently and optimizes loading times and reduces file sizes.
- 4 Allows testing the game as end-users will experience it and identifies issues that may not appear in the Unity editor.
- 5 Provides a true measure of the game's performance outside the development environment.
- 6 Creates the files necessary for distribution on various platforms (app stores, Steam, etc.).

Finally, all you need to do to open the project is to run the "Goblin\_Heist.exe" executable file and enjoy.

## 4 Game Mechanics & Walkthrough

---

### 4.1 Main Menu

#### 4.1.1 Main Menu Screen

Below is the Main Menu, which is the first thing that appears when we open the executable



**Figure 2. Main Menu Screen**

This is the Main Menu of the game. There can be found Many Important functions like the Settings Screen, Information Screen, Load and Continue Screens and where someone can begin their adventure.

#### 4.1.1 Settings Screen

With the "Settings" button we open the game settings. From there we can adjust the quality of the graphics as well as the sound volume. The game's graphics may be Low-Polly as a stylistic choice, but you can set how many polygons the assets have by adjusting the graphics, giving the choice of balance between performance and higher resolution graphics.





Figure 3. Settings Screen

#### 4.1.2 Information Screen

By pressing the “Information” button, the player can see the buttons of the game as well as read the game's storyline



Figure 4. Information Screen

#### 4.1.3 New Game Screen

Clicking new game will ask the player to give the name of the save file



**Figure 5. New Game Screen**

By inputting a save file name and pressing “Create New Game” a new Save File will be created, and a new game session will begin. The Save mechanics will be analyzed in a later part of the thesis.

#### **4.1.4 Load Game Screen**

In this screen the player can select a specific save file and continue their adventure from a selected save file.



**Figure 6. Load Game Screen**

### 4.1.5 Continue and Quit Buttons

By pressing the continue button the game loads the last saved save file.

By pressing the Quit button the executable closes.

## 4.2 Playthrough and UI

### 4.2.1 Player's Graphical Interface

Congrats, you can now start playing the game!

In the middle of the screen, you can see the player character.

At the bottom left of the screen the big red circle is the "Hit Points" (or hp) indicator. If it drops to 0, your character dies and you have to restart from the latest "save point".

At the bottom right of the screen the big blue circle represents the "Mana Points" (or mp). You may use an appropriate amount of them to use spells or abilities your character possesses. If you don't have enough "Mana Points" you can't use those abilities.

At the bottom middle of the screen are the rest of the player's graphics interface. From top to bottom these are:

- Six blank boxes. This is your "action bar". There you can place consumables, spells and abilities to bind them to a key (from 1 to 6) or click them from there. This is done for faster accessibility of those actions. Furthermore, the bar is fully customizable, meaning you can place anything mentioned above in any blank space you wish.
- Below the "action bar" is a blue unfilled bar and its caption. This is your character level. As you complete storyline quests or subdue enemies you gain "Experience Points" (or exp). When the bar fills the character advances to the next level, getting stronger like this as the game progressing and rewarding the player for interacting with the world. Each level has a higher need of exp to advance to the next one.
- Finally next to the "Experience Bar" is a small circle with a book icon. This is the "Quest information" button. Clicking it will open a window that shows the player all the received and unfinished quests, and what are the steps they need to do for the quest's completion. At the start of the game there are no active quests, thus the "Quest Information Window" is empty.





Figure 7. First Steps in Game

#### 4.2.2 Pause Menu

Pressing the Esc button on the keyboard will make the Pause Menu appear. While the Pause Window is open, the rest of the game freezes so the player can take a breather, and then continue enjoying the game afterwards.



Figure 8. Pause Interface

In the Pause menu there are 3 options:

- **Resume:** With this button or by pressing the Esc keyboard button again the Pause Window closes and the games resumes.

- **Save:** With this button the save file updates to this current state of game. It is considered good practice to save before battles or important choices so that you can load and continue from where you left.
- **Save and Quit:** With this button the save files is updated as above and the game closes. When reopening the game the player can Load their preferred save file and continue playing from there.

### 4.2.3 First Steps

As specified in the “Information Screen” of the “Main Menu”, you may move by clicking the left mouse’s button. If there’s a red X next to the cursor the character can’t move to the indicated position. By holding the right mouse button, the player may pivot the camera angle.

By making the first steps following the dirt path in game, a cutscene will appear and the first quest will be given to the player.

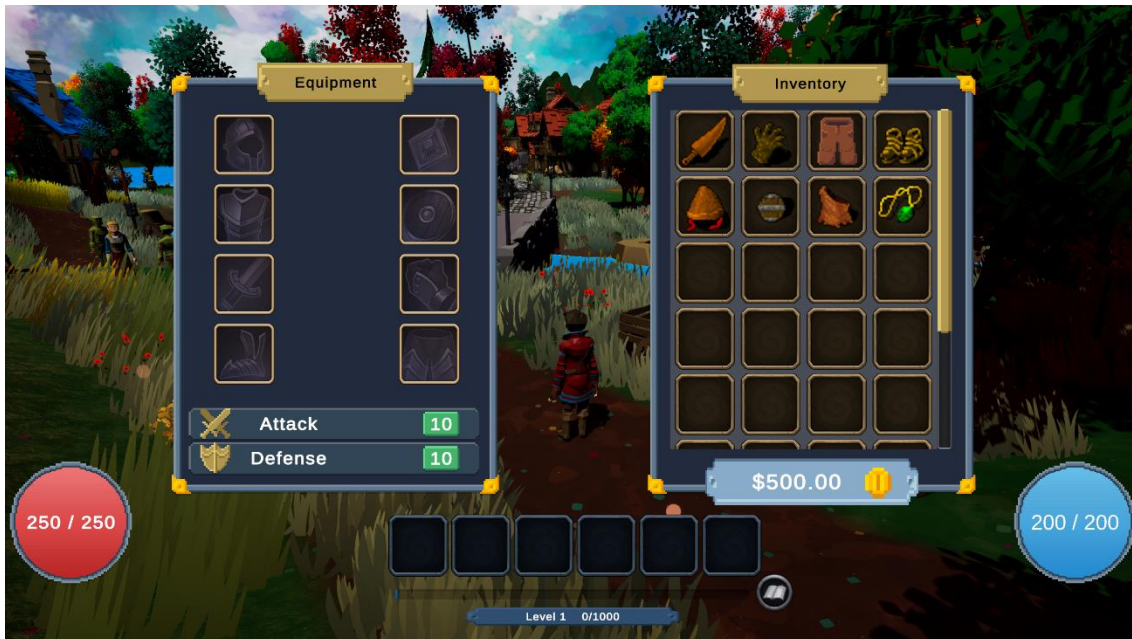
After that, keep moving down the dirt path and there will be some leather pouches at the right of the path. These are some “loot” pickups.

### 4.2.4 Equipment and Inventory Interface



Figure 9. Pickups

After clicking on the leather pouches the player now has gained the loot in their “Inventory”. They can interact with their “Inventory” by clicking the I keyboard button.



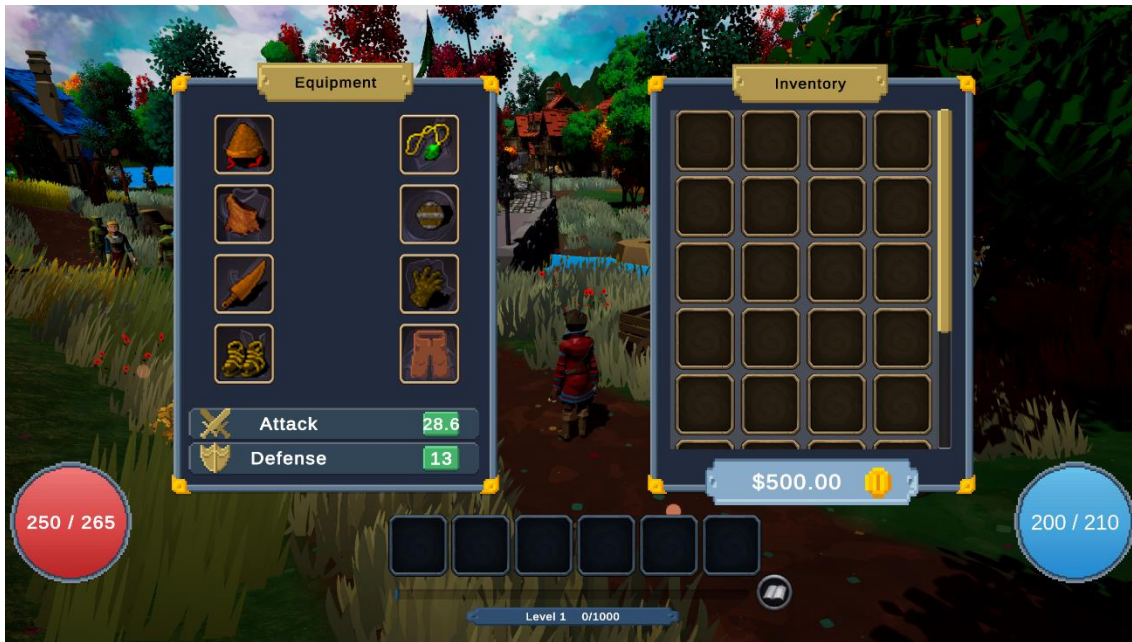
**Figure 10. Equipment and Inventory Interfaces**

The right Window is the “Inventory”. There can be found all the character’s items, abilities and spells. At the bottom there is a caption with a coin icon. There can be found the player’s money. Money can be used to buy items and more from shop vendors in the game and can be gained from loots, quests or even killing enemies.

The left window represents the “Equipment”. There the player can see the equipped items and the combat stats, which is the attack and defense. Equipped Items and the player’s level are responsible for their final values. Attack represents the damage player’s attacks do, and Defense helps lowering the damage taken. Items also raises the maximum hp, mp and other stats as well.

To equip items just drag and drop the desired equipment to the respected Equipment slot as shown bellow.





**Figure 11 Equipping Items**

Now continue down the road and rescue the villager next to the bridge from the goblins!



**Figure 12 First Quest: Save the villager**

To attack the goblins, hover the mouse over them until the mouse cursor turns to a sword, then as the "Information Screen" said, press the left click once. After that the character will get to the closest place needed to be to hit the enemies and start to automatically attack. No need to re-click to attack, but if the player wants to change the attacking target, they can left mouse click another target to do so.

Enemies will automatically run towards the character as soon as they spot him, as well as chase him as long as they can.

In Figure 9 combat can be seen more in-depth. When enemies are attacked their hp bar is appeared on top of their heads. The green part represents how much hp have left. The red numbers represent the final damage dealt from either side. The final damage is calculated by a formula in which defense lowers the final damage by a percentage.

#### 4.2.5 Quest Interface

After the player kills all the goblins hover the villager and left mouse click to talk to her.



Figure 13. Dialogue Interface 1

When clicked the "Dialogue Window" appears. There the Non-Player's Character's (or NPC) name appears on top with bold letters. Below there is the NPC's dialogue. To continue click the Next button on the bottom right of the window.





**Figure 14. Dialogue Interface 2**

Many times, the player will have to answer to the Dialogues with their own lines. When so, dialogue options will appear in the “Dialogue Window” as buttons and the player will have to make their own choices. In this case, the player only have one option.

#### 4.2.6 Leveling Up



**Figure 15. Leveling Up**

After completing the dialogue, the first quest will be completed, and the player should gain enough exp to reach level 2. When the character levels up, it's HP and MP go to their maximum value.

Furthermore, the character also gains two Attribute Points. By pressing the T button on the keyboard, the “Attributes’ Window” opens.



**Figure 16. Attributes Interface**

In this window the player can inspect the character’s attributes. There are 6 attributes total, each of them giving a different effect:

- **Strength:** Raising this attribute makes your attacks deal more damage.
- **Charisma:** Raising this attribute lowers the prices of items in shops.
- **Wisdom:** Raising this attribute raises your mana regeneration (or how fast you regain mp)
- **Intelligence:** Raising this attribute makes your abilities and spells deal more damage.
- **Constitution:** Raising this attribute raises your Max Hit Points.
- **Dexterity:** Raising this attribute raises your evasion rate, making it possible for attacks of enemies to have a chance to miss, dealing no damage.

Leveling up the character also gains passively more Max Hit Points, Max Mana Points, Attack and Defense, so the player doesn’t have to worry that much about balancing their Attribute Points to specific attributes and can mix and match them to try new tactics each time.

To continue the story, the play may cross the bridge to the right and kill all the goblins in the village of Evercoast. There should be 10 goblins in there with some new types of enemies including stronger “soldier” goblins and ranged “archers” enemies. The second Quest will count all fallen enemies toward completion.





Figure 17. Defeating Enemies in Evercoast Village

#### 4.2.7 Shop Interface

After killing all the goblins go and talk to the villager at the entrance of the village. He is the shopkeeper, and a “Shop Window” will open



Figure 18 Shop Interface

In this Window all the shopkeepers' items can be found. Next to them are their respective quantities and prices. In the rightest section the player can select how many of each item wishes to purchase. The Yellow caption at the bottom left shows the total price of the items the players want to buy. If the player exceeds their money the value



will turn red to indicate that they don't have that much money. The final step is to press the buy button at bottom right. Doing so the total money will be subtracted and the desired items will be added in the Inventory.

One of the quests at this stage requires the player buys a bow and a shield, so let's buy those along with some better equipment.



Figure 19. Quests Window.

Let's also drag and drop a health potion from the Inventory window to the ability bar to have quick access to them if we need to heal fast. They can be accessed by pressing 1 on the keyboard or left clicking the potion icon on the ability bar.

#### 4.2.8 Portals

Portals are the way the character travels from one scene to another. To do so just left mouse click to the portal. The player can't travel back and forth freely from portals, so they should only pass through when they feel ready.



#### 4.2.9 Spells and abilities

Next in line, go towards the village's center with the golden statue and take a left turn. At the end of the road is the village's wizard.



Figure 20. Village's wizard.

He has lost his staff and wants the player to find it. As a reward he's going to teach the character some spells.



The staff is located at a house with blue roof. This can be found along the dirt road next Hope (the first villager the player saves). Find the staff at that location and return it to the wizard.



**Figure 21. Retrieve the wizard's staff**

After that return to the wizard and he will reward the character with 2 spells. To equip them drag them to the ability bar from the Inventory.



**Figure 22. Equipping Spells**

To use a spell, left mouse click it from the ability bar or use the appropriate key on the keyboard (from 1 to 6). This will activate the spell and a spell circle will appear below

the cursor. Choose the target Area of Effect with the cursor and the left mouse click again to activate the spell.



**Figure 23. Casting a spell**

That was the last of the mechanics the game has to offer. Next in line will just cover the rest of the playthrough.

## **4.3 Rest of the Walkthrough and side Quests**

### **4.3.1 Priest's Quest**

To find the priest, go straight from the village's statue.





**Figure 24. Village's Priest**

He will ask the character to take care the ghost problem at the village's graveyard.

The Graveyard is at the right of the village's entrance.



**Figure 25. Village's Graveyard**

After dealing with the ghosts, talk to the priest to complete the quest.

### 4.3.2 Mayor's Quest

Next in line, go right from the village's statue to find the Mayor.



Figure 26. Village's Mayor

His dialogue will complete the second quest and ask the character to deal with the root of the problem, the goblin camp close to the village. This is located on the next location, so it's advised to go there after the player has completed all the things they want with the Evercoast village.

### 4.3.3 Goblin Camp

The next location begins with a cutscene that shows the way to the player towards the goblin cave, the game's final location.





**Figure 27 Goblin Camp Cutscene**

The player can either go straight to the camp and deal with the goblin or they can explore the mysterious ruins to their left.



**Figure 28. Mysterious Ruins**

There just outside of the ruins, there is a ghost with a side-quest.

Talking with the ghost will tell the story of a mad warlock that now lives as an undead in the ruins. If the player accepts the quest, a secret door will appear to the wall of the ruins for the character to walk into.



**Figure 29. Secret Door**

After entering the ruins, strong undead enemies await the character to fight. After dealing with them the boss waits at the very back of the ruins



**Figure 30. Underground Ruins.**

After beating the undead warlock the quest will be completed and the player will be rewarded with Dark Blade, a powerful asset for the upcoming battle with the goblins.





Figure 31. The Dark Blade

#### 4.3.4 Fighting Goblin Camp

After that, the player can find a secret path that leads to the back of the Goblin Camp next to the ruins. This way they can get faster to the pesky archer enemies and make the fight a bit easier.



Figure 32. Secret Path



**Figure 33. Fighting the Goblin Camp**

Before going in the cave the player can re-visit the underground ruins. There, underneath the ladder will be a shopkeeper with higher level equipment to gear up.



**Figure 34. Ruins Shopkeeper**

After that, the last stop is the Goblin Cave Ahead, where the Goblin Chief awaits.



### 4.3.5 The Goblin Cave

This is the final location of the game project. It starts with a cutscene.



Figure 35. Goblin Cave

In this final act, Goblin will constantly spawn in waves. At the last wave is the boss will appear.



Figure 36. Goblin War chief

Finally, after defeating him, the Game is Over! With the village saved and the character's journey to become a legend just beginning.

## 5 Scripts

---

Now that the Playthrough is over, let's talk about the technical part of the Thesis and Game Development, starting with the Scripts.

Unity Scripts are written in C# and are the backbone of any project. With a meticulous and well coded setup, the Game Development process becomes a much more and enjoyable experience.

Let's start with the Core Script Folder

### 5.1 Core Scripts

Core Scripts are the essential script of the project. It's scripts

#### 5.1.1 ActionScheduler.cs

The main principle behind this in this case would be to provide control as to how many actions are performed at any given time, and it also provides ways to initiate new actions and abort an existing action in progress.

The class has a private field `currentAction` which keeps a reference to the currently executed action. For this case, The `StartAction`, it takes parameters and has defined logic, which is used to start a new action. An `IAction` type object is employed as a method parameter of this method. In this case, the first check will avoid executing unnecessary actions, as the new action will not be executed if it is the same as the current one. If an action is in progress (`currentAction` is not null), the current action is aborted by calling the `Cancel` method before the new action is assigned to the `currentActivity` field.

Moreover, the class contains a method `CancelCurrentAction` so that the user does not reach or trigger the current action. This method is accomplished by using the method `StartAction` and providing a null parameter, which is a command of cessation of current action without a new one being instituted.

All in all, the `ActionScheduler` class is able to control the launching of actions in such a way that it is impossible for two actions to run simultaneously and even helps users to abort the current action with ease. Such functionalities may be especially needed in game implementation where character's actions or other similar needs have to be done effectively.

#### 5.1.2 CameraController.cs

This class is intended for managing a camera in a Unity game, which is a `CinemachineFreeLook` camera, allowing for additional camera types and movements. There are two serialized fields those are `freeLookCamera` which is a `GameObject` that contains `CinemachineFreeLook` component and `donkey` which is a

CinemachineFreeLook variable. Besides these, it holds a reference to the PlayerController script which is responsible for the player elements and behaviours.

In the Awake method, which is issued when the script instance is being loaded, the donkey variable is set by getting the gameplay camera that is attached to a gameobject previously loaded into memory. The same situation can be identified with regards to the playerControllerScript variable where the PlayerController is picked from the holder camera controller gameobject where the camera controller is positioned.

The Update method is called every frame of the game and contains the controls of the camera that depend on players' inputs. In one such case when the right mouse button is pressed (Input.GetMouseButtonDown(1)), in the script it is first determined if the playerController script holder is trying to drag a particular UI element (playerControllerScript.isDraggingUI). In such a case the method returns early to avoid moving the camera. Otherwise it modifies the m\_XAxis.m\_MaxSpeed of the donkey (the CinemachineFreeLook camera) to be 500, meaning the camera can be panned fast since it is horizontal movement. When the right mouse button on the controller is released

### 5.1.3 CameraFacing.cs

The main goal of this script is to make sure that the GameObject it is assigned to will always be orientated towards a specific camera, which is a CinemachineFreeLook camera and is annotated as PlayerFramingCamera.

Regarding the script, there is a private field called playerFramingCamera which is a reference to a CinemachineFreeLook component. This field is set up in the Start method which is executed only once when the script instance is enabled for the first time. The script does this by calling `GameObject.FindGameObjectWithTag("PlayerFramingCamera")` to locate the GameObject that bears the "PlayerFramingCamera" tag and then Gets its CinemachineFreeLook component with `GetComponent()`. Whenever the class is not referenced correctly the playerFramingCamera field cannot be referenced from the correct camera.

This logic is implemented in LateUpdate method which is called once per frame after all Update methods have been called and this is when the GameObject is positioned towards the camera. The usages of this method are limited to attaining the GameObject orientation to the specific point. The point that it calculates is  $2 * \text{transform.position} - \text{playerFramingCamera.transform.position}$ , which turns out great as it enables the GameObject to gaze at the position of the camera as seen from the reverse of the GameObject.'s position. This eases the movement of the GameObject with respect to the camera since the GameObject can see the camera facing always even when the camera moves.

In a nutshell, the CameraFacing script is straightforward but will have an impact on the game.

#### 5.1.4 CameraLayerToggle.cs

The provided code constitutes a Unity script by the name of CameraFacing, which is derived from the base class of MonoBehaviour, meaning it is a component of Unity that suits any GameObject. The main aim of this script is to allow any GameObject that it is attached to, to always look towards an assigned camera, in this case the "PlayerFramingCamera" CinemachineFreeLook camera.

The script has a private variable called playerFramingCamera, which is a holder of component CinemachineFreeLook. This member is introduced within the Start method, which is called once when the script instance is active. In the Start method, the script calls `GameObject.FindGameObjectWithTag("PlayerFramingCamera")` to obtain every tagged GameObject, then asks for CinemachineFreeLook component by calling `GetComponent()`. In this manner the camera referenced by the playerFramingCamera variable will point to the correct camera.

The method called LateUpdate, the use of which is intended to be one per frame which is executed after all the update methods have been called, contains the logic that enables the Game Object to be oriented towards the camera. The method employs `transform.LookAt` to position the GameObject at a certain point in relation to that which has been calculated. The point it calculates is  $2 * \text{transform.position} - \text{playerFramingCamera.transform.position}$ , makes it possible to make the GameObject look at the position of the camera located on the opposite side of the GameObject. This guarantees that the Game Object will always orient itself towards the camera and not the other way around.

All in all, within the parameters of this written paper, the CameraFacing script presents itself as an easy but very useful feature that allows GameObject to face only one camera throughout a particular scene, thereby improving the visual interaction between the GameObject and the camera.

#### 5.1.5 Condition.cs

The given C# code contains a nested class called Predicate which is a part of a bigger class. This Predicate class also has the attribute `[System.Serializable]`. That means the instances of this class can be serialized. Serialization is beneficial in Unity for the purpose of saving and retrieving data, such as game or configuration data. Within the Predicate class, there are three fields that can be serialized. The first field, predicate, is of type EPredicate which according to earlier remarks, is probably an enumeration or class located in another part of the code. The comment indicates that it used to be a string. The second field, parameters, is an array of strings and is most probably used to transfer extra information to the predicate. The third field is the negative boolean which has a default value of false and is used to know if the predicate evaluation result should be negated or not. There is also a method called Check in the Predicate class. This method accepts IEnumerable as the parameter. This method loops through each of the IPredicateEvaluator instances in the provided collection and calls the Evaluate method on each of them passing along the predicate and parameters fields. The return type is a boolean?(Boolean?) which can be true or false or null.

Inside the loop, in case the evaluation result is null, it passes on to the next evaluator. If a result is obtained, the method checks whether the result is equal to the negate field or not. In this case, the method comes back with a 'false', indicating that the predicate check has passed. However, if all evaluators have been executed but none have returned a result that matches the negate condition, the method returns true, indicating that the predicate check has failed.

All in all, this piece of code is intended to check the collection of evaluators implementing a predicate under a condition whether it has to be negated or not. This kind of implementation offers flexibility in relation to condition evaluation and can be applied in various instances like game logics or decision systems in general.

### **5.1.6 DestroyAfterEffect.cs**

The code in question is concerned with a Unity script "DestroyAfterEffect" that extends the class MonoBehaviour which is the base for any Unity script. The purpose of this script is to remove one Game object or itself after a given particle system has finished its play cycle.

This class includes a GameObject variable named targetToDestroy which is also a serialized field. This field has a [SerializeField] characteristic which means it can be altered in the Unity editor but still remain private. The targetToDestroy is purposely not given any value at its instantiation.

Update method is one of the inbuilt Unity methods that is called on every frame of execution. In this method, the script determines, by means of calling IsAlive() method, whether the ParticleSystem component, which is attached to the same game object as that of the script, is still functioning. GetComponent() is used to get the ParticleSystem that has been attached to the game object that this script is attached to.

On determining that the particle system is out of life, then the script checks for the instance of targetToDestroy. Where the targetToDestroy is assigned a non-null game object, then the game object is destroyed via Destroy(targetToDestroy). Where targetToDestroy has not been assigned, the game object that this script is on is destroyed by Destroy(gameObject).

To summarize, this script guarantees that when the particle system has completed playing, either a given target game object will be destroyed or if there of no specific target, the game object which the script is attached will be destroyed. This is useful in getting rid of game objects that were once used in the visual representation of particle effects and most of which tend to be leftover once the particle effect is complete.

### **5.1.7 FollowCamera.cs**

The given code showcases FollowCamera as a Unity script that extends MonoBehaviour, the parent class of every unity script. This script is intended to cause a given camera to move in around a given object in the game.



The class includes a target field which is of 'Transform' type and is also serialized. The [SerializeField] attribute also permits housing of this field in the Unity Editor without making it public to the class. Transform type is the type that embodies position, rotation and scale of an object in the game space. By dragging a game object in this target field in the Game Object Inspector, you determine what object should be behind the camera.

The LateUpdate is one of the built-in unity methods that is executed after every frame and once, but this one has been executed at least one time and does not summon any updates. This comes in handy especially in the camera animating function since it ensures that the camera is moved with the target object rather than aiming at it, hence giving a precise and less jittering motion. Within the LateUpdate method, the script changes the position of the camera (transform.position) to the position of the target object (target.position) and they are equal. Such an assignment is straightforward, in that the camera keeps track of the target without being offset or smoothed in any way.

To summarize, this script maintains that the camera view always matches the position of the target object, thus moving the camera with the target object. This would be useful in some gameplay situations, such as third-person views or overhead camera views.

### 5.1.8 IAction.cs

The code snippet given describes an interface called IAction in C#. C# also has an interface that defines the obligations of the derived classes. The ability of class within an interface is provided for cross-class sharing of code hence achieving code reuse as well as flexibility.

In this case, the IAction interface describes a single operation definition called Cancel. The Cancel method does not have a body, which implies it does not carry with it any implementation. Yet, it requires that the classes that implement the IAction interface incorporate their version of this method. The slash & burn tool that is the Cancel method is described as devoted towards offering the possibility to cancel any action which is the reason why not much has been stated on Canceling even though it is obliged upon by the implementing class to define what that means.

With the help of IAction interface one can define many different classes that usually are doing various actions but all of them are performing some action that can be cancelled. The current pattern helps in achieving cleaner and more organized implementation, because one has to implement code that uses the interface IAction without knowing anything about the implementation class details. This is especially important when a designer must implement a quasi-server, controlling various kinds of actions.

### 5.1.9 IPredicateEvaluator.cs

In the C# programming language the image depicts an enumeration called EPredicate and an interface called IPredicateEvaluator. The EPredicate enumerates conditions including predicates that can be assessed within the context of the game. In particular, each entry in the enumeration refers to a particular condition which can be evaluated e.g. if the player has a particular quest, whether the player has achieved some



objective or they are at a particular level. The comments next to each enumeration value explain to the readers what additional information is necessary in order to carry out the desired evaluation of each condition. For instance, HasQuest takes the required information in terms of the quest name while CompletedObjective takes in the name of the quest and the objective id.

The IPredicateEvaluator interface defines only one method called Evaluate. The method has two parameters, one of which is of a type EPredicate value and the other is an array of strings called parameters. The EPredicate value indicates which condition is being evaluated while the parameters array contains data required to carry out the evaluation. The method returns a nullable boolean bool?, which could be true, false or null. Null return value implies that the evaluator has no information on how to evaluate the object with the given parameters and predicate.

In the code that you write, IPredicateEvaluator has to be created because it defines the requirement of the Evaluate method for every implementing class. Using this technique, it is possible to create an arbitrary number of classes capable of evaluation of game conditions. For example, one class would evaluate conditions regarding quests and another would evaluate conditions regarding inventory. By following the IPredicateEvaluator interface these classes can be substituted with each other without any problems when it comes down to the predicate evaluation.

#### **5.1.10 PersistenceObjectSpawner.cs**

This script aims to provide that a certain game object known as the persistent object exists only once in the game and will be available across different scenes.

This class includes a GameObject serialized field, persistentObjectPrefab. The constant for this GameObject has the [SerializeField] attribute. That allows this field to be put in the unity editor and has a constant access within the class. This field is meant to be filled in with a reference of the object that is persistent and that will be created.

Another static boolean field typed hasSpawned is introduced and its value is set to false at the point of declaration. This field serves the purpose of differentiating whether the said persistent object has been created or not. Because it is static it means the value is true for all other instances of the class PersistenceObjectSpawner which guarantees that regardless of the number of instances of the class in existence it is no longer necessary to check since it has already been checked.

The Awake method is one of the Unity built in's which takes place when the script instance is being loaded. In this method, the script first inspects whether hasSpawned has a value. If hasSpawned has an active value, this method completes its delegate right here. If hasSpawned is false, the script goes on to invoke the SpawnPersistentObject method to create the persistent object and then sets hasSpawned as true to mean the object has already been spawned out.

This method of the SpawnPersistentObject is used to create an instance of the persistent object. It involves Instantiate in order to get a new game object from persistentObjectPrefab. The next step after the object is instanced is calling

DontDestroyOnLoad and using the newly created game object in its argument. This makes sure that the object of persistence is not wiped off when a new scene is loaded thereby giving the object an opportunity to last through various scenes in the game.

As already mentioned, the PersistenceObjectSpawner script provides a mechanism so that the particular game object it creates will only be created once and will survive the transitions of the game to other scenes. This is done by appropriately declaring a static boolean field to capture if the object has been spawned and preventing the object from getting wiped out by loading new scenes by employing the use of unity's dontdestroyonload method.

## 5.2 Abilities

In this folder the in-game abilities are created and managed.

### 5.2.1 DelayEffect.cs

The code is describing a Unity scripable object known as DelayCompositeEffect, and this one extends EffectStrategy. This class implements the strategy where a certain amount of time is asked before applying a range of effects. The [CreateAssetMenu] is the attribute that makes it possible to create such instances of this scriptable object from the Unity Editor at "Abilities/Effects/Delay Composite", among other menu items.

The class has three serial fields: delay, delayedEffects and abortIfCancelled. The delay field is a float which indicates the duration to wait before applying the effects. The delayedEffects field is composed of an array of EffectStrategy objects that are the effects which are to be executed after a delay. The abortIfCancelled field is a boolean that specifies whether the deferred effects should be aborted or not if the action is aborted.

The StartEffect method is a method that was defined in the EffectStrategy base class and this method is overridden in this class. It has two parameters: an AbilityData object called data and a finished Action. This method creates (or rather, starts) a coroutine called DelayedEffect and supplies the data and finished parameters as arguments.

The DelayedEffect method is a private coroutine which is applied to the logic of delay and effect implementation. After that, the specified pre-defined delay is respected via the call of yield return new WaitForSeconds(delay). However, in cases where abortIfCancelled is true and the action is actually canceled according to the condition data.IsCancelled() then, the coroutine is exited prematurely by the use of yield break. Finally, it processes the delayedEffects array sequentially and transfers the data and finished parameters into the StartEffect method for each EffectStrategy object.

To conclude, in the DelayCompositeEffect class it is possible to specify several composite effects that will be invoked after some time in relation to each other not previously defined. Also, it allows to cancel any of the delayed effects invoked if the action is canceled. This makes it easy especially in video games which have advanced

ability structures, where one has to apply a number of effects in succession with some delay.

### 5.2.2 HealthEffect.cs

The code presents a HealthEffect Unity scriptable object which is derived from EffectStrategy. A subclass is made of the scriptable object in the Unity Editor and this subclass is simply named Health Effect in the menu "Abilities/Effects/Health". This is meant to extend or reduce the acquired health of a group of the target game objects rather than just any other scriptable object.

The class has a healthChange serialized field which is of type float which represents the health change in terms of increase or decrease. The [SerializeField] attribute makes it possible for this field to be altered using the Unity Editor yet the field remains private to the body of the class. This field is meant to capture the default value of the health value to be modified on the targets.

The StartEffect method restricts one from using useEffects because it overrides a certain functionality from the EffectStrategy base class. It only has 2 parameters AbilityData data and Action finished. The AbilityData data consists of details about the ability in use which include the user of the ability and the targets. The Action finished is a delegate and is used to inform when the effects would be completed.

In the StartEffect method, the healthChangeFinal is derived as a product of the healthChange field and a multiplier from the data object. This facilitates the health change to be scaled depending on the particular scenario in which the ability is being practiced. The method then cycles through the targets which were obtained from the data object. For each target, it tries to obtain a Health component. If there exists a Health component in the target, it determines if the healthChangeFinal variable is of negative value or positive value. If it is negative, TakeDamage method of the Health component is performed which inverts the negative value to positive. If it is positive, Heal method of the Health component is performed. This ensures that the health change is applied appropriately by giving rise to damage or healing.

Everlastingly, finished callback is called when the effect is over. Hence, any other logic which requires that the effect has been completed can be carried out. To summarize, the HealthEffect class allows to apply health related effects on multiple targets, make them heal or damage depending on the needs and also works well with Unity's scriptable object system.

### 5.2.3 OrientToTargetEffect.cs

The code contains the definition of the OrientToTargetEffect Unity scriptable object that derives from the EffectStrategy abstract class. With the help of the [CreateAssetMenu] attribute, you can create this scriptable object when editing in Unity from the menu "Abilities/Effects/Orient To Target." It is expected for this scriptable object to turn the 'user' game object in the direction of a specific target.

The class overrides the `StartEffect` method from the `EffectStrategy` base class. It is worth noting that this method accepts two parameters: first an `AbilityData` object called `data`, and second an `Action` delegate called `finished`. The `AbilityData` object relays the details of the ability which is in action, the owner of the ability and the point that the ability is directed at. The `Action` delegate is a callback method indicating that the effect is executed and that this action has been completed.

Within the `StartEffect` function, the first action taken by the script is to acquire the user of the ability violating `data.GetUser()`. Then, it goes to user's transform component and the `LookAt` function is used with argument `data.GetTargetedPoint`. This makes the intro objective to make the user look towards the objective. Once the rotation has been carried out an invoke finish callback is made so as to signal that the execution of the effect is complete. This ensures that any other logic that depends on the effect being executed would be carried out.

To conclude, the `OrientToTargetEffect` class allows the user to keep the game object in such a position so that it is aimed toward a definite target position. This functionality is useful for game mechanics that require the direction at which the user is should be determined first before the user attempts to perform an attack. The ability to leverage the scriptable object system of Unity makes designing and setting up this effect without much limitation in the unity editor.

#### **5.2.4 SpawnPrefabEffect.cs**

`SpawnTargetPrefabEffect` is a Unity scriptable object which derives from `EffectStrategy`. The `[CreateAssetMenu]` attribute is particularly useful because it allows the user to create and use this scriptable object within the unity editor "Abilities/Effects/Spawn Target Prefab" menu path. This scriptable object is intended to spawn the designated prefab at the given target location and holds the object for a specified amount of time before destroying it.

The class has two serialized fields `prefabToSpawn` and `destroyDelay` respectively. The `prefabToSpawn` field is a `Transform` representing the prefab that needs to be created. The `[SerializeField]` attribute allows this fields to be exposed in the unity editor without making it public to the class. `DestroyDelay` field is a float whose purpose is to determine the time after which the instantiated prefab will be destroyed. If -1 is passed as `destroyDelay`, it means that the instance will not be destroyed automatically.

The `StartEffect` method is defined in the class derived from `EffectStrategy`. This method has two parameters, an Argument of type `AbilityData` called `data` and a finished `Action` delegate. The `AbilityData` object contains the details of the ability performed like ability's user and ability's target point. The `Action` is a delegate that needs to be called once the effect has completed its execution. In the constructor of the class where this method is defined, a coroutine called `Effect` is started within the `StartEffect` method and the `data` and `finished` pointers are passed to the coroutine.

The Effect method is a private coroutine that carries out instructions on how to spawn the prefab and even the delayed destruction of the prefab. The first action is instantiating the prefab by calling `Instantiate(prefabToSpawn)`, which creates a new instance of the necessary prefab. The position of the newly instantiated prefab is then set onto the target point gotten from `data.GetTargetedPoint()`. If the field in `destroyDelay` is more than 0, the coroutine runs a specified delay using `yield return new WaitForSeconds(destroyDelay)`, after which it proceeds to destroy the prefab created earlier by calling `Destroy(instance.gameObject)`. In the end, the finished callback is called to notify that the effect is over.

To conclude, the `SpawnTargetPrefabEffect` class effectively allows spawning of a prefab at a target with some benefits of destroying this prefab after a period of time if desired. This can be beneficial for effects in a game, for example, summoning objects, projectiles, animations, or other temporary constructions. The use of Unity's scriptable object system enables this effect to be developed and set up with relative ease in the Unity Editor.

### 5.2.5 `SpawnProjectileEffect.cs`

The code exemplifies a scriptable object called `SpawnProjectileEffect` in Unity that derives from the `EffectStrategy` parent class. The `[CreateAssetMenu]` option lets one make the above scriptable object right from the editor at the "Abilities/Effects/Spawn Projectile" menu option. The intended use for this scriptable object is to create a projectile at a given point and can be aimed at either one point or a number of them.

The class holds four public serializable variables, i.e. `projectileToSpawn`, `damage`, `isRightHand` and `useTargetPoint`. The `projectileToSpawn` field is a `Projectile` object that indicates which prefab model would be created. The `damage` is a float variable that indicates the amount of destruction the projectile will be able to inflict. The `isRightHand` field is a boolean which prevents the spawning of the projectile from the actors right hand when did not intend to do so. The `useTargetPoint` field is a boolean variable that specifies whether the target point of the projectile will be one or several.

The method called `StartEffect` is an override of a method of the `EffectStrategy` base class. In this case, two parameters are taken: An `AbilityData` type object called `data` and an `Action` delegate called `finished`. The `AbilityData` object contains the ability data, such as a user of the ability and a target point for this ability. The `Action` delegate is a callback which will be called when the certain effect has finished its task. Within the wholly conclusive effects, `StartEffect` begins first by determining the user of the ability through `data.GetUser()` and accessing the `Fighter` component of the user. A spawn position of the projectile is also determined by the function `fighter.GetHandTransform(isRightHand).position`. Depending on the value of `useTargetPoint`, the `SpawnProjectileForTargetPoint` or `SpawnProjectilesForTargets` methods are called to spawn the projectile(s). There are also other techniques with which the effect is created, after which the finished callback is invoked to prove that the effect is done.

SpawnProjectileForTargetPoint is a private method which contains logic for spawning a projectile at the user-specified target point. First, it calls Instantiate(projectileToSpawn) to create an instance of the specified projectile that has to be spawned. The location of the newly created projectile is then positioned to be the spawn position. The SetTarget method of the projectile is called passing in the target point data.GetTargetedPoint(), the ability user and the damage. This defines where the projectile will strike by assigning the target point.

SpawnProjectilesForTargets on the other hand is a private method which handles the logic for spawning projectiles for each of the multiple targets. It works through an enumeration over the targets data.GetTargets() has been able to return. For each target it tries to find a Health component. Where such a Health component has been found, then the method Instantiate(projectileToSpawn) is called, that will create a new instance of the specified projectile. The position of the instantiated projectile is then set to the spawn position. The SetTarget method of the projectile is called, passing the Health component of the target to be shot, the user of the ability and the damage. This method sets the target of the spawned projectile to the target that was specified.

### 5.2.6 TriggerAnimEffect.cs

The code outlines a Unity scriptable object called TriggerAnimationEffect, which is an extension of the EffectStrategy base class. The [CreateAssetMenu] feature also makes it possible for this scriptable object to be able to be created from the editor, with the menu path: "Abilities/Effects/Trigger Animation". This scriptable object is intended to change the position of the game objects in accordance with some animation controller component present in them.

The class only has one serialized field: animationTrigger. This field is a string that simply describes the animation to be triggered when the animation trigger is hit. The [SerializeField] attribute also makes this field editable in the Unity Editor but is used within the class only.

The StartEffect method extends a particular method defined in the EffectStrategy base class. Two parameters; AbilityData data and Action finished are passed to this method. The AbilityData data is responsible for providing details on the ability currently being used, for example the user of the ability. The Action pointer points to a function which may be called when the effect has been executed fully.

The call data.GetUser() is used to get the user of the ability from the returned information. Inside this method, the user's Animator component is retrieved using GetComponent(). It is noted that the method of Animator component called SetTrigger(animationTrigger) is invoked. Here, the trigger specified is set on the Animator and the related animation gets played. The finished function is called, which notifies that the effect has been completed.

To summarise, the TriggerAnimationEffect class offers a straightforward mechanism to activate an animation of an Animator component on a game object. Thanks to the



implementation of relatively simple Unity's scriptable object wrapper, the impact may be easily designed in Unity Editor.

### 5.2.7 TagFilter.cs

This code shows the rendering of the Unity scriptable object called TagFilter, which is derived from the Filtering base class. You are also able to create instances of this scriptable object directly from the Unity Editor using the statutory menu "Abilities/Filters/Tag" due to the [CreateAssetMenu] attribute. This scriptable object is meant to be used for filtering a list of game objects with the set tag.

The class bears a single serialized field: tagToFilter. This field is a string that defines the tag to be filtered. The [SerializeField] attribute permits this field to be modified from the Unity Editor without exposing it to the whole class. Normally, tagToFilter gets its default value of an empty string.

The reason for this is because the Filter method is an implementation of the abstract method declared in its Parent class, which is Filtering. The Filter method consists of 3 methods and takes an IEnumerable type objectsToFilter as an argument which is the array of game objects to be filtered. This method makes use of a foreach loop to go through each of these game objects in the array. Within this loop, it will use gameObject.CompareTag(tagToFilter) to determine if the specific game object has the flag to be filtered. If yes, as in the member assigning to the right hand side of the return statement, 'yield return' is used. This is done to return a collection of game objects from the method containing all the game objects that have a particular tag in them.

To conclude, the TagFilter class enables a selective removal of game objects based on a particular tag. The implementation of Unity's scriptable object system enables the easy designing and setting up of this and other filters in the Unity Editor.

### 5.2.8 ClickTargeting.cs

The code creates a DelayedClickTargeting scriptable object in many ways making it a subclass of the TargetingStrategy class. The [CreateAssetMenu] attribute also allows creating instances of such scriptable object through the Unity Editor from the menu 'Abilities/Targeting/Delayed Click' This scriptable object is intended to be used with abilities that require the player to target a location after waiting a while.

The class has a number of serialized fields, cursorTexture, cursorHotspot, layerMask, areaAffectRadius and targetingPrefab. These fields are used in targeting configuration. cursorTexture and cursorHotspot are targeting cursor image and targeting cursor's hot spot respectively. layerMask is used to specify type of layers which should be considered for raycast. areaAffectRadius determines our targeting radius as to how far around the targeted object can still be affected. targetingPrefab is transform prefab which refers to the graphical representation of the area being targeted. The targetingPrefabInstance field is meant to hold a reference object of the targeting prefab.



The `StartTargeting` method is one of the targeted methods which overwrites one of the methods found in the `TargetingStrategy` base class. It accepts two parameters which are the `AbilityData` object referred to as `data` and an `Action` delegate referred to as `finished`. The `AbilityData` object has data on the ability that is under use including the user of that ability. The `Action` delegate is merely a callback usually called when the targeting has been completed. Inside the `StartTargeting` method, the script accesses the `PlayerController` of the user and initiates a coroutine called `Targeting`.

The `Targeting` coroutine is responsible for implementing the targeting logic. First, it deactivates the `PlayerController` to disallow player movement during the targeting phase. If `targetingPrefabInstance` is null, then it creates the targeting prefab instance for the first time. Otherwise, it uses the current instance. The adjustable scale of the targeting prefab instance is done due to the `areaAffectRadius`. Then the coroutine proceeds obtaining a measuring loop which lasts as long as the targeting is not withdrawn. In the measuring loop, the cursor is set to a specific texture hot spot for its position.

When the right mouse button is pressed, the action is cancelled, the instance of targeting prefab is turned off, the `PlayerController` is turned on again, and the `finished` method is called. If a raycast from the mouse position collides with a valid target, then the hitpoint is where the targeting prefab instance's position will be moved to. If the left mouse button is pressed, the coroutine waits for the button to be lifted, defines the targeted point and targets in the `AbilityData` object, and breaks the loop. Lastly, the instance of targeting prefab is switched off, the `PlayerController` is turned back on and `finished` callbacks are called.

`GetGameObjectsInRadius` is a supporting method whose function is to fetch an enumerable list of game objects which are within the given distance from a designated point. It carries out the sphere cast at the specified point with the given radius and returns the game objects that were hit by the sphere cast. This method explains the target which the ability would affect.

### 5.2.9 SelfTargeting.cs

The code implements a Unity scriptable object called `SelfTargeting`, which is a subclass of the targeting strategy base class. Specify the path where this scriptable object will be created as "`Abilities/Targeting/Self`" in the editor and to make it possible the class has the attribute `[CreateAssetMenu]`. This scriptable object is designed for use with abilities whose target is the ability user, that is, it self targets.

The `SelfTargeting` class implements the `StartTargeting` method which is abstracted in the `TargetingStrategy` class. This method includes two arguments, an `AbilityData` object called `data` and an `Action` delegate called `finished`. The `AbilityData` object consists all the necessary attributes of the ability eventually being employed including who the ability is directed at. The `Action` delegate is a delegate that needs to be called after targeting is over.

Within the `StartTargeting` method, the script begins with defining the targets for the current ability by executing `data.SetTargets` and sending it an array including the user of the ability, which can be acquired by executing `data.GetUser()`. This clearly makes the user the target of the ability. Thereafter, the script provides the targeted point for the ability by wrapping procedure `data.SetTargetedPoint` and providing to it the location of the user which is obtained by using `data.GetUser().transform.position`. This sets in the game world the point at which the ability will take effect and in this case, is where the user is currently at. At last, the finished callback is called to indicate that the targeting completion.

To summarize, the `SelfTargeting` class is geared towards making self-targeting of abilities in a Unity game a straightforward task. Making use of Unity's scriptable object system, this targeting strategy will be easy to design and set up in the Unity Editor within the scope of the targeting system. This is perfect for abilities that are self-targeting by nature such as spell which heals the caster, buff spells or similar effects that are self-casting.

### **5.2.10 Ability.cs**

The code implements a Unity scriptable object known as an `Ability` which is a subclass of the base class `ActionItem`. The `[CreateAssetMenu]` attribute lets you create an instance of the scriptable object from the Unity Editor user interface, rather than through a script, with the menu entry "Abilities/Ability". This particular scriptable object encapsulates the ability system of a game such as its targeting system, filtering system and effect application system.

The `Ability` class comes with seven serialized fields that need to be defined to some degree: `targetingStrategy`, `filterStrategies`, `effectStrategies`, `effectMultiplier`, `cooldownTime`, and `manaCost`. These fields are intended for configuring the 'Ability', the components that make up the behavior of the ability. The `targetingStrategy` field contains a handle to a `TargetingStrategy` which describes the mode of acquiring an object using the ability. The `filterStrategies` field is an array of `Filtering` objects that are employed in screening the targets of the ability. The `effectStrategies` field is an array of `EffectStrategy` objects that explain the effects that the ability inflicts on its targets. The `effectMultiplier` field is a float that increases the magnitude of the effects, the `cooldownTime`, and `manaCost` fields are used to limit the delay between successive uses of the ability and the resources expended on its activation accordingly.

The `Use` method has been overridden from the base class `ActionItem`, and it is invoked when the game object (the user) employs the ability. To begin with the user's ability, this method first attempts to verify whether the user has sufficient Mana to use the level by utilizing the `Mana` component of the user. In case the user does not have adequate mana, the method returns false. Then the method tries to access cool down status of the ability if it is there for the user by the `CooldownStore` component. If the ability is on cooldown the method returns false. Up to now, it is not possible using the ability yet If the ability can be used an `AbilityData` object is created to record how the ability is used particularly its user. An `ActionScheduler` component of the user is then retrieved in

order to perform the action that comes with the working of the ability. The `effectMultiplier` is initialized in the `AbilityData` object, and the `targetingStrategy` is employed to commence the targeting. The `StartTargeting` method of the `targetingStrategy` needs the `AbilityData` object and a function in charge of the target's acquisition.

The `TargetAcquired` method is called when a target is located. It is necessary to check whether the ability use has been cancelled. In this instance, the control is returned promptly. Otherwise, using the Mana component of the user again, it is validated whether the user has enough mana for the ability use. If for some reason the user does not have enough mana, the control is returned promptly. The `CooldownStore` component of the user is then accessed in order to activate cooldown for said ability first.

This method then proceeds to enumerate the `filterStrategies` array and for each one, applies the filtration to the targets which were collected by the `AbilityData` object. The clogged targets are placed back in the `AbilityData` object. After that, the method goes over the `effectStrategies` array and executes every given effects by invoking `StartEffect` of each of the `EffectStrategy` objects. The `EffectFinished` method is registered for each effect in case of its completion.

### 5.2.11 AbilityData.cs

The code implements the creation of a class called `AbilityData`, which inherits from `IAction` interface. This class is meant to hold data concerning an ability in the game including its user, targets, effect multiplier, targeted point, and cancelled status. These fields are created within the class in order to hold this information: `user` (of type `GameObject` who is the actor of the ability), `targets` (of type `IEnumerable` who are the designated targets of the ability), `effectMultiplier` (of type `float` how strong the effects are), `targetedPoint` (of type `Vector3` where the ability is aimed in the game world), and `cancelled` (of type `bool` whether the ability has been cancelled).

When constructing the object of `AbilityData` class, it is necessary to provide the `GameObject` parameter, which represents as the user of the ability and this specific parameter initializes the `user` field. The class also provides several public functions for the access and the modification of the private fields. The `GetTargets` method gives the ability the current targets of the ability, whereas `SetTargets` method gives the targets method. The `GetUser` method gets back the user of the ability. The `GetMultiplier` and `SetMultiplier` methods provide means for the effect multiplier retrieval and modification respectively. The `GetTargetedPoint` and `SetTargetedPoint` methods provide means for Targeted point retrieval and modification respectively.

Also, in the context of the `AbilityData` class in particular, it is implemented the `StartCoroutine` method which accepts `Coroutine` as an argument and starts it with the `MonoBehaviour` of the user. The ability allows performing asynchronous operations. The `Cancel` method changes the value of the `cancelled` field to `true`, in which case it indicates that the ability is canceled. The `IsCancelled` method returns the value of

cancelled so that some other part of the code can determine whether the ability is still in the active state.

Taking everything into account, the AbilityData class is a class that encapsulates all the data, which is necessary to perform an ability, also provides the operations related to this data like, modification, access, starting of coroutines and checking for cancellation.

### **5.2.12 CoolDowns.cs**

The code implements a Unity MonoBehaviour script C CooldownStore, that maintains cooldown timers for some abilities or items. In this class there are two dictionaries: cooldownTimers and initialCooldownTimes. The cooldownTimers dictionary is used to monitor the remaining time for each ability's cooldown, whereas initialCooldownTimes stores the cooldowns which each ability was allowed for at the start.

Unity calls the Update method once every frame. In this method, the code goes through a copy of keys from the cooldownTimers dictionary. For each ability, it shortens the cooldown timer by the time passed after the last frame, which is recorded in Time.deltaTime. If the timer goes negative, the ability is removed from both timers, cooldownTimers and initialCooldownTimes, thereby ending the cooldown period for that ability.

The StartCooldown method is called when it is required to start a cooldown for the selected ability. Parameters for this method are two, an InventoryItem is the ability that should be cooled down and a float is the time period for which the cooldown will hold. In this method, the cooldowns for the ability are set in both dictionaries simultaneously, that is why it is safe to use now and be required later.

The GetTimeRemaining more method returns the possible time left to wait for using a particular ability. If the mentioned ability is not present in the cooldownTimers dictionary, the method will return zero value, meaning a cooldown period is not active for that particular ability.

The GetFractionRemaining method helps to determine the fraction of time that is still pending for which an ability can be used again. It first establishes whether or not the ability is null or is not included in the cooldownTimers dictionary. If any one of them is true, it gives a zero. If none of 이| bệnh is ㅏ and nothing is cast and only a target frame is given, the fraction is computed as a ratio of the remaining cooldown time with the initial cooldown time and this figure on a scal of naught and one has a direct relationship with the speed.

In conclusion, the CooldownStore class offers a very basic and effective patient management system for the cooldown of various abilities in unity games and as well allows access and search for states of these cooldowns.

### 5.2.13 Effects.cs

The code explains the purpose of the class EffectStrategy and its contempt as an extension of ScriptableObject in the same code - ScriptableObject is a class in Unity that facilitates design of objects that can store information and be used in the game. Since the EffectStrategy class is derived from the ScriptableObject, it will enable the creation of effective strategy design that can be reused many times and easily manipulated in the Unity Editor.

It contains information that the EffectStrategy class is also an abstract class which means that an object of this class cannot be created. To do so - there are separate classes taking her place and implementing its abstract methods. The class contains a single abstract method StartEffect, which is intended for defining the effect of the strategy. It has two parameters - an AbilityData object and a delegate Action called finished. The AbilityData parameter is likely a class which is supporting information on the ability which is being projected such as the user, targets and etc. The Action delegate will be used to perform a completion action when the effect execution is completed.

The subclass that extends this class must implement the StartEffect method because the EffectStrategy class defines this method as an abstract method. This design allows for different types of effects to be achieved by extending the EffectStrategy and defining the StartEffect method differently in each sub class. For instance, one of the subclasses would implement a damage effect whereas another one would implement a heal effect. With other words when various behaviors are needed, they do not all have to be programmed in a single class, instead every behavior can go into a separate class that inherits from EffectStrategy.

### 5.2.14 Filtering.cs

The code declares an abstract class titled Filtering which derives from ScriptableObject In Unity, ScriptableObject is a class which permits the creation of assets that fetch in all types of game data. Again, Removing from ScriptableObject allows the Inclusion of Filtering class to make productive data holders that can be well organized within the Unity Editor.

The Filtering class is declared abstract, meaning instances of this class cannot be created. However, the class acts only as an abstract class and does not implement any filtering strategies. It contains only one abstract method named as Filter which takes an IEnumerable argument, objectsToFilter of input type, and returns an IEnumerable. The Filter method above is designed to be implemented by subclasses who provide filtering behavior for collections of GameObject's.

In the case of Filter definition as an abstract method, the Filtering Parent Class makes it compulsory that each child class has to define this particular method. This method of design makes it possible to implement various filtering strategies by inheriting the class

Filtering and changing the implementation of the Filter method in other classes. For instance, one of the sub-classes may contain a particular filtering technique that will return only those game objects which are active, while the other sub-class may use a filtering technique which applies tags or components. This method of hosting different business logics increases the flexibility of the code and encourages code reuse because different filtering logics can be encapsulated into their subclasses while a common interface is given out by the Filtering base class.

### **5.2.15 Targeting.cs**

The code describes an Abstract class whose name is TargetingStrategy and that derives from ScriptableObject. In Unity, ScriptableObject is a class that allows assets containing data that can then be used in the game. Since TargetingStrategy class derives from ScriptableObject, it can be used for creating reusable data assets, which are in turn easy to use in the Editor.

The TargetingStrategy class is abstract, which also means it cannot be instantiated directly. This means that it can only be used as a parent class for implementing classes which will provide its essence in a more complete way. The class defines only one abstract method defined as StartTargeting that has two generic parameters an AbilityData and an Action and takes one Action delegate parameter called finished. The AbilityData parameter is most likely a class that contains information about the ability that is being used which includes the user, targets and others. The Action delegate argument is a pointer to function which will be called upon completion of targeting.

Making StartTargeting an abstract method in the TargetingStrategy class ensures that all subclasses implement this method. This design permits subclassing of the TargetingStrategy class so as to cater for other forms of targeting strategies and designing the StartTargeting method accordingly. For instance, one of such child classes may address a situation where the targeting can only be on one object while the other class will have the targeting feature addressing many different objects all at once within a specific space. This strategy enhances usability and code reusability because different targeting strategies can be implemented in their class hierarchy and still derive from the same TargetingStrategy class.

## **5.3 Attributes**

In this folder are the basic character attributes that apply to the player character as well as the enemies.

### **5.3.1 Health.cs**

The code defines a Health derived from MonoBehaviour class and a class that implements ISaveable interface. This is the purpose of the class in Unity, holding all the elements and operation related to the health of a game object. It has damage, heal, recover health, and die events. The class also supports interaction with Unity's event system so other objects can listen for changes in health status.



The Health class is composed of many fields some of them being a reference to a WaveSystem object, a regenerationPercentage field which allocates the percentage of health to be regenerated, and a takeDamage field of type TakeDamageEvent which is a subclass of UnityEvent. The onDie field is a public UnityEvent variable that you are able to set up from the Unity Editor which is fired whenever the GameObject dies. The healthPoints field is of type LazyValue in order to avoid consuming miscellaneous computing resources. While it is referred to for the first time, it computes the value of health points only then. The wasDeadLastFrame field is a boolean used for determining the death status of the game object in the last frame.

The awake method is the one that creates the healthPoints variable lazily, this is why the healthPoints are not initialized inside the Start. Additionally, the game object always acquires a reference to WaveSystem during the Start. The GetInitialHealth function takes the value from the BaseStats component given to the game object. The Heal function recovers health points and changes the status of the game object in the scene. The IsDead function simply checks whether the health points are lesser or equal to zero to determine the death of the game object.

The TakeDamage function takes away health points by a certain damage value, and if the health points reach zero, then it takes care of the hard death logic. It calls the onDie event, gives experience to the instigator and informs the WaveSystem when the instigator is the player. If the game object is still alive then it invokes the takeDamage event. The GetHealthPoints and GetMaxHealthPoints methods give respective current health points and the maximum health points of the game object.

Whenever the AwardExperience function gets called, experience points will only be awarded upon getting the instigator who has an Experience component. The RegenerateHealth function restores health in proportion to both regenerationPercentage and Maximum health points. The GetPercentage and GetFraction functions returns the current health as a percentage and fraction of maximum current health respectively.

The UpdateState function modifies the game object's current health. If the game object has recently died, it plays the "die" animation as well as cancels the currently executing action. If the game object has just resurrected, the animator is reconnected to its current state only after the same event is played. The methods OnEnable and OnDisable are used to attach and detach the RegenerateHealth method from the onLevelUp subscriber of BaseStats component's events.

When playing back a game that has been put on hold, the CaptureState and RestoreState methods that are available on the ISaveable interface are used to save then restore the health points. The CaptureState method gives back the existing health points, and the RestoreState method takes the current volume of health points back to the volume that is in RestoreState and makes the appropriate changes to the state of the GameObject.

### 5.3.2 HealthBar.cs

The code implements a HealthBar class that extends the MonoBehaviour class, thereby allowing it to become a component in Unity which can be attached to any game object. The primary purpose of creating this class is to create a health bar to show the health of a game object. Moreover, the class has healthComponent field, foreground field and rootCanvas field which are all serialized. These fields get published to the Unity Editor so that the developer can add and assign the Health component, the RectTransform of the health bar foreground and the Canvas with the health bar, respectively.

Unity calls the Update method only once per frame, and this method is invoked into any script to update modified values after a certain frame time. In this method, the code first evaluates whether the value it obtained from the healthComponent regarding the health fraction is 0 or 1 by making use of Mathf.Approximately. If the health fraction is 0 that means that the character has no health left to be sustained or if the health fraction is 1 then that means health fullest has been attained; the rootCanvas is turned off by putting the enabled property to false. This is useful in hiding the health bar when the health is completely empty or when the health is comprehensively full because in such cases, it will not make sense to show the health bar.

If health fraction is greater than or equal to zero and less than or equal to one the property being rendered here which is the rootCanvas is enabled setting its enabled property about true such that the health bar is displayed. The localScale of the foreground is then changed to the corresponding health fraction. The localScale object is set to a new Vector3 where the x-component is equal to the health fraction, the y and z components are equal to one. This towards the camera scale's the front portion of the health bar such that the current health is displayed as a percentage of the full health.

The HealthBar class is therefore administering a picture of the health of a game object which is active showing the health bar only when needed and the bar itself changing its size whenever the health changes.

### 5.3.3 HealthDisplay.cs

The code defines a Health Display class which is MonoBehaviour. This means that this is a component that can be added to game objects in unity. This class is meant to show the player's health in the form of a text field and a slider which shows present health and the maximum health of the player.

The class is made up of three fields one of which is a private Health object called health a serialized TextMeshProUGUI object called textField and a serialized Slider object called slider. The SerializeField special attribute makes it possible to assign the properties in the Unity Editor and accordingly interface relevant UI elements to the script.

Awake is the method called by Unity when the instance of the script is being loaded. In this method, the health field is set by locating a game object with the player tag and

getting its health component. This makes sure that the HealthDisplay script does not lack the reference to the health data of the player.

Unity calls the Update method once every frame. In the validate and input in this method, the textField gets modified to center on current and maximum health points and the string.Format method is used. The format string "{0:0} / {1:0}" is included to ensure whole readers of health points are saved. Also, the value of the slider control is updated to represent the actual health status with respect to maximum possible health status. This works by taking the current health points and dividing them by total health points making a ratio which helps the slider use normalized values between 0 and 1 hence depicting players health with the help of the slider.

Overall, the HealthDisplay class fulfills these set objectives by making sure that the player's health is displayed to the appropriate level in the games facilitating interaction with the UIs by making use of the relevant control elements.

### **5.3.4 Mana.cs**

The code is about a class named Mana which is derived from MonoBehaviour and implements ISaveable interface. This class is meant for providing the mana resource management, its regeneration, usage as well as saving/ loading state for a game object in unity.

The class has a field which is named mana as LazyValue< float> which was set inside the Awake method. LazyValue is a class that holds the value wherein its computation will only begin once it has been required. In this case, it is set with the initial value of the maximum mana from the GetMaxMana method.

The Update method is called by Unity once every frame. In this method, the code checks if the current mana value is less than the maximum amount of mana. If it is then the current mana is put on addition and multiplication by the growth rate and Time.deltaTime respectively. When the mana is recuperated after an attack and the resultant value is found out to surpass the ultimate mana, it is reset.

The GetMana method gives out the current mana level while the GetMaxMana method brings out the mana base maximum level from the BaseStats also fixed on the same game object. In the same vein, the GetRegenRate method fetches the mana recharging rate from the BaseStats component.

In the UseMana, the player tries to spend a certain amount of mana. If the amount of mana requested is higher than the current mana, the method returns false and states that there is not enough mana available. If enough Mana is available, then the requested amount is subtracted from the available Mana pool and true is returned.

The method CaptureState is used from the ISaveableFunctions interface: it is responsible for saving the mana value. It returns the value of the mana's current state. The RestoreState method is also present in the ISaveableFunctions interface; its result is that the mana value, which has been saved, is restored. For this, it has a parameter,

which is an object, but in this particular case, it will be converted to a float and will be represented as the mana value.

All in all, the Mana class allows for the creation of the relatively prototypical in-game represented mana management with regards to regeneration, its use and the persistence of the state in a Unity game.

### **5.3.5 ManaDisplay.cs**

The code depicts a ManaDisplay class that manages to inherit from the MonoBehaviour class, making this class a Unity component that can be attached to game objects. This class is intended to visually represent the player's mana using a text field along with a slider which complements the current and the maximum mana points.

In the previous line also: There are three fields in the class, a private Mana field called mana, a serialized TextMeshProUGUI field called textField and a serialized Slider field called slider. Thus, the public field is not modifiable in the script, but these elements help in linking appropriate UI with this CS file using the Unity Editor.

This method is used by Unity when the script instance is being loaded. In this method, a certain game object called "Player" is instantiated, containing the Mana component, in order to initialize the mana field. This assures that the ManaDisplay script will target the player's mana information as required.

The Unity engine invokes the Update method only once for every frame. It is in this method that the textField is set to show the current and maximum number of mana points making use of the string.Format method. The format string "{0:0} / {1:0}" does make sure that whole numbers are displayed as mana values. Also, the slider is adjusted in such a way that the slider reflects mana points as a ratio of total mana points. This is done by picking the current mana points and dividing by the maximum mana points which gives a value which ranges between 0 and 1 that the slider can use in representing the mana of the player visually.

To summarize, the ManaDisplay class introduces an alternative innovative approach to how mana for the player in the game is depicted evolving the possible options on the UI elements as per the changes on mana.

## **5.4 Cinematics**

With these scripts we create the game's cinematics and manage the camera's movement through them.

### **5.4.1 CinematicEvents.cs**

The code contains a class called CinematicEvents which derives from MonoBehaviour and as such, it is a Unity script that can be attached onto game objects. This is done to implement the movement of any cinematic scenes in a game probably the dialogues



and any other UI elements that require showing or hiding There are two main fields observed which are the serialized fields: canvas pauseCanvas textField and dialogues.

The class has a few variables that are marked SerializedField which are: canvas, pauseCanvas, textField and dialogues. The canvas and pauseCanvas fields are GameObject references pointing to two different UI canvases while textField is a TMPro TMP\_Text ie from TextMeshPro used in displaying text. The dialogues field is an array of strings which contains the dialogues that will be used during the cutscene events. Moreover, the class has a private class integer index field which is set to 0, which keeps the current line of dialog on screen.

Now, EnableCanvas function on the other hand brings out the main canvas of the game to the eye and hides the pause canvas from the player – i.e, when the game resume button is pressed the pause canvas is avoided. However, the DisableCanvas function does the opposite of what was done regarding the main canvas meaning that it reveals the pause canvas and in the process, hides the main canvas which is meant to be out of sight.

The ShowNextText method is the one which executes action, displaying next dialogue line. As a debug, it prints current index value onto the console first. After that, it checks whether the index is greater than the length of the dialogues array. If this is the case, the method returns early to avoid an index out of bounds exception. If not, it takes the textField's current dialogue line and replaces it with current line's increasing index value. `textField.Text = dialogues[lineCounter]; lineCounter++;`

The ResetIndex method simply resets the index to 0 and gives way to the entire dialogue sequence to be played again from the beginning.

The Quit method finds an object of SavingWrapper class in the scene and calls its LoadMenu method. If a SavingWrapper instance was found, it logs the good message on the console for convenience purposes. This could be used to leave the cut scene activity and possibly go back out to the main menu or some other area of the game.

As a whole, the CinematicEvents class delivers a clean solution for incorporating and handling cinematics in a Unity based game including implementing dialogues and switching screenshots.

#### **5.4.2 CinematicsControlRemover.cs**

The code includes a definition of a CinematicsControlRemover class that inherits from MonoBehaviour and thus is a part of Unity which can be added to Game objects. The purpose of this class is to control the player's control during cinematics and to ensure that the player does not interact with the game while the cinema is playing and to allow control when the cinema has finished.

Private fields core and player can be distinguished in the class alongside with two GameObject references. Start is a method, which is called by Unity when the script instance is being loaded. Within this method, the code gets the PlayableDirector component from the same game gameobject and adds the player and stopped events.

The played event is fired, when the cinema starts, and the stopped event is when the cinema stops. The DisableControl method is triggered when the player event occurred and the EnableControl method is raised when the stopped event occurs. Also, the player field is set searching for the game object with tag Player.

The DisableControl method is called as soon as the cutscene begins. Within this method, the CancelCurrentAction function is invoked on the ActionScheduler component, which results in the termination of the presently active action of the player. The PlayerController component is disabled next, which stops the player from controlling the character and the character from interacting with the environment. The mouse pointer is captured and hidden to improve the cutscene effects. A debug "enter" is printed in the console for checking purposes. The italicized lines indicate the possibility of deactivating the Fighter component as well, or even deactivating the player game object, but these lines are not used now.

The EnableControl method is invoked as soon as the cutscene has ended. In this method, the PlayerController component is reactivated and the control of the player is restored and thus movement and action in the game can be performed. The graphical pointer is freed and shown again. The method also located the object of the Fader class that is in the scene and used FadeOutImmediate and FadeIn for the purpose of making a transition. A debug 'done' is printed in the console for checking purposes. The italicized lines indicate the responsiveness of the Fighter component as well, where in the negatives the player object can be rotated back to a standing position, but these are not in use.

In summary, the class CinematicsControlRemover allows the designer to easily restrict the control of the player during cinematic moments within a game developed in Unity, which enables the player to have a complete and uninterrupted experience.

### **5.4.3 CinematicTrigger.cs**

The code implements a CinematicTrigger class which is derived from the MonoBehaviour class and also implements ISaveable interface. It is used within games developed in Unity engine, which causes a video sequence to play when the player approaches a designated zone. Additionally, the class is capable of saving its state and restoring it afterwards.

One of the adjustable parameters of the class is a serial boolean variable alreadyTriggered which is set to false initially. The [SerializeField] annotation permits this field to be changed by the designer in the Unity Editor hence it becomes more comfortable to define if this event is triggered or cannot be triggered anymore.

The OnTriggerEnter function is Unity's way of defining a game object event that corresponds to another collider entering how is temptation of the trigger collider attached to the game object. Within this method, the code verifies whether the alreadyTriggered boolean is set to 'false' and whether the collider that came into the trigger has a tag that reads 'Player'. If all is well, the function gets the PlayableDirector attached to the same game object where the script is and calls the Play function to play

the animation. It then proceeds to setting `alreadyTriggered` to true, so that this ceremony does not get triggered again.

The `CaptureState` method belongs to the `ISaveable` interface and is utilized to persist the current instance of the `CinematicTrigger` object. This method return value is `alreadyTriggered`. This field indicates whether the cinematic is already triggered.

The `RestoreState` is also part of the `ISaveable` interface method which allows recovering the state of the object `CinematicTrigger`. This method has a parameter called `state` which is presumed to be a primitive boolean. It uses this parameter to determine the value for the `alreadyTriggered` field by setting it to the position since this one is restored from last source saved.

In general, the `CinematicTrigger` class offers a simple and efficient management of cinematic sequences in the Unity game so that they can be only fired once, and their state can be preserved or restored.

## 5.5 Control

The scripts in this folder are the backbone of the game, as they control the enemies, player, clickable areas and even the respawns.

### 5.5.1 AIController.cs

The `AIController` class is a built in unity component that controls the artificial intelligence of the character in question. It derives from `MonoBehaviour`, indicating that it is a code which is to be attached to game objects. Its several serialized fields are the ones that can be set in editor like `chaseDistance`, `suspicionTime`, `agroCooldownTime`, `patrolPath`, `waypointTolerance`, `waypointDwellTime`, `fractionOfPatrolSpeed` And `shoutDistance`. These fields handle different variables of the behavior of the AI for example how close to the player can the AI start chasing, how long will the Ai stay vigilant when the player is not spotted, how it maneuvers between the various points by waypoints amongst other features.

The class also defines several private fields: `fighter`, `health`, `mover` and `player` which are all references to other components or game objects. The `guardPosition` field is an instance of `LazyValue` and is used to keep the starting position of the AI character. The `timeSinceLastSawPlayer`, `timeSinceArrivedAtWaypoint` and `timeSinceAggravated` are stopwatch timers which count the periods in which the AI saw the player, reached a waypoint and got aggravated respectively. The variable `currentWaypointIndex` holds the current patrol index of the Ai's movement pattern.

In the `Awake` method, the script sets up the `fighter`, `health`, `mover`, and `player` fields by getting relevant components or game objects. It also sets up the `guardPosition` field with the AI's starting location. The `Reset` method brings back the AI into the guard's position, and also the timers as well as the waypoint index.

The `Update` method is activated at a rate of once per frame. At the beginning, the AI is checked if it is dead by calling the `IsDead` method from the health component. If the AI

is not dead in any way, its aggro status is checked to see if it is able to attack the player. If that is the case, it implements AttackBehaviour. If the AI has seen the player recently but is not in a position to attack, it activates SuspicionBehaviour. Other than that, the AI will invoke PatrolBehaviour to resume patrolling. Finally, the UpdateTimers method is invoked at the end of each cycle to replenish the evaporating time scales.

The Aggravate method is used to reset the timeSinceAggravated timer to zero in order to denote that the AI has possibly been aggroed. The PatrolBehaviour method is used for managing if the AI entity polices a certain area by moving it from one waypoint to another in case it has a patrol path. The AtWaypoint method checks whether the AI has come within range of the current waypoint whilst the CycleWaypoint method moves the waypoint pointer to the next one in the patrol path. The GetCurrentWaypoint method is used to get the coordinates of the current waypoint.

The method SuspicionBehaviour prevents the AI from continuing with the current task by providing the method CancelCurrentAction from the ActionScheduler component. The AttackBehaviour method also performs a similar function to its name where it restarts the timeSinceLastSawPlayer timer and invokes the Attack method onto the fighter component in order to attack the player. It also assigns AggravateNearbyEnemies method to get nearby AI characters to anger.

In order to use the AggravateNearbyEnemies method, a sphere cast is carried out to locate AI characters that fall within a distance called shoutDistance and its Aggravate method is called. The IsAggravated method is used to test if the AI is too close to the player by checking if it is within the chaseDistance or has been aggroed in recent times.

Last but not the least, Unity invokes the OnDrawGizmosSelected method to draw gizmos in the Scene. It draws a blue wire sphere around the AI character in order to indicate the chaseDistance. This is useful for developers because it shows the detection range of the AI while working in the editor.

### **5.5.2 ClickablePickups.cs**

The ClickablePickup class is a component that the Unity game engine utilizes making a game object able to be picked up using raycasting. As shown by the `[RequireComponent(typeof(Pickup))]` attribute, the Pickup component has to be on the same game object for it to be usable by the ClickablePickup component. This is necessary because as the name suggests, the ClickablePickup class without the Pickup component cannot work.

The class implements the IRaycastable interface which I presume has the raycast implementation methods. The ClickablePickup class has two internal fields, one named pickup which is a reference to the Pickup component and the other valueText which is a serializable field of type TextMeshProUGUI exposed in the Unity Editor. The Awake method populates the pickup field by creating an attachment to the Pickup component found on the same game object.



In the Start method, I made the valueText field show a description of the item relating to the Pickup component. This is achieved through imaging the item pickup.GetItem().GetDisplayName() and programming the text of valueText. This makes sure that the item name is the correct item in the UI element when the game starts.

The GetCursorType method determines the CursorType of the item depending on the possibility of its pickup. When pickup.CanBePickedUp() is called, it returns CursorType.PickUp; otherwise, CursorType.FullPickup is returned. This method probably switches the cursor icon to the appropriate one corresponding to what can the player do with the object.

The HandleRaycast method performs a functionality where the player accesses the game object by clicking it. Whenever the left mouse button is pressed (Input.GetMouseButtonDown(0)), he invokes StartCollectAction on the Collector component of the callingController and passes it this argument. This is the action that will be performed when the player initiates the motion to collect the given item. The method returns true meaning the raycast was processed as expected and was successfully done.

In brief, the ClickablePickup class combines integration of UI updates, cursor change and interaction implementation so that the game object would be useful and informative in the game.

### 5.5.3 CursorType.cs

The code outlines an enum declaration of a C# code, and the enum is called CursorType. An enumeration, or enum is a special data type consisting of a set of named members called enumerator list. In this case, CursorType is used to indicate the types of cursors which can be shown in a game or an application.

Available Cursors Grid blob (B2) so hardware cursors can be changed with mouse movement: The CursorType enum has the following values:

- None: Is indicative with regard to the presence of a cursor such as there is no cursor or there is a mouse pointer but none of the images applies to it as no mouse pointer is needed at this default stage.
- Movement: Specifies a gestural cursor assignable to motion functionality such as moving a figure or an object in the game.
- Combat: A fighting cursor which probably occurs during combat, meaning that the cursor is designed for combat when a player can fight.
- UI: Having to move the indicator around, UI uses this pointer whenever the user is hovering over buttons and menus in the game.
- PickUp: Performing an action with a cursor in a video game where it indicates taking an item from the game, showing that something is clutchable.
- FullPickup: This should be the active cursor type when the alternative item to pick up is in a suitable spot but requires some care, perhaps when the item is picked up only when the conditions permit, such as full inventory.

- Dialogue: It focuses on one cursor type which is used in dialogues with NPCs, that is away from the main gameplay.
- Shop: This shows the cursor type to be used while the gameplay involves a shop or merchant where items can be purchased or vended.

Having described the above cursor types for the gameplay, it enhances user interface of the game by changing the cursor dynamically appropriate to the action of the player or the situation within the game.

#### 5.5.4 IRaycastable.cs

In the given code, an interface called IRaycastable is mentioned which is written in C#. An interface in C# provides a set of methods and properties and requires the implementation of these methods and properties by classes. Without their implementation, interfaces are used to denote some features that may be needed in more than one class to avoid a rigid structure.

The IRaycastable interface contains two method definitions:

- CursorType GetCursorType(): this method will most likely return a value of type CursorType which appears to be an enumeration capturing different types of cursors that can be visually presented to the player. The nature of this method is to enable the engine to understand what cursor it should display when the player uses the cursor over an object implementing this interface. In the case that this method is included in the interface, this means that every class that implements the IRaycastable interface has to formulate their own logic on how to do this.
- The Structure of HandleRaycast(PlayerController callingController): This method is defined as taking a parameter of type PlayerController probably used to denote the player or the AI controller of the player in the game. This kind of method is expected to return a boolean value. This particular method is used to get a call when the ray extended outwards hits the object, that is, the raycast. The boolean return value can also indicate if the raycast was performed successfully. It is this ability to bind instruction that is exercised in regard to this type of weaponry that is most prominent in with respect to intent external to weaponry intercalates connecting Count as Instruction Define this method is required without fail for every invoking class of the interface defined.

As earlier indicated, the IRaycastable interface helps many game devises to tell how they behave on raycasts and which cursor should be displayed on a player from a different cursor or from an interacting away distance. This facilitates the uniformity and diversity of interaction management by different objects in the game.

### 5.5.5 PatrolPath.cs

The code in question demonstrates a Unity script dubbed Patrol Path that derives from MonoBehaviour. This particular script intends to depict a Patrol path within the unity editor with the use of gizmos. A gizmo is a two-dimensional graphic or a visual aid to the editor to help a developer visualizes and treats the composition and interactions of some of the game objects in the game structure.

The class also contains a constant float `waypointGizmoRadius` set to `0.3f`, which is used to determine the radii of the spheres that are drawn at every waypoint. The `OnDrawGizmos` function is special because it is a unity method that is called by the unity editor to receive information from developers as to how to place the gizmos in the scene view. In this method, a for loop runs and grabs all the child transforms for the game object to which this script is attached. For each child transform, it determines the following waypoint index based on the `GetNextIndex` method and can therefore place a sphere in that location at the current waypoint by the `Gizmos`. It also connects the lines between the current and the next way-point by filling them with lines drawn by `Gizmos.DrawLine`.

The `GetNextIndex` method is the method that is provided in order to check the index `i` and passes to this method an integer `i` and returns an index of the next waypoint. The current index here denoted `i` is if all the children of that index `i` is a last child index coordinates then it will ignore all the other voice and it will return to index `0`.

The `GetWaypoint` method takes an integer `i` as a parameter and provides an index to the `i`-th child transform. This method makes use of `transform.GetChild(i).position` to get the pose of the child transform within the given index.

To put it simply, this script allows to add a representation of a set of waypoints and the corresponding links between them to the Unity's Editor, or for more specific purposes - to create and test the guides for the 'patrolling' AI characters or any other game stressor which has to move along the certain path.

### 5.5.6 PlayerController

This class is focused on 'who does what' in the player, that is all the actions that the player performs in the game. In this script, first a variety of edges are included like `System.Collections`, `System.Collections.Generic`, `UnityEngine` etc. These edges allow using several properties of Unity as well as some of the .NET language such as certain kinds of collections, manipulating game objects, user interface and so on.

MonoBehaviour class which the `PlayerController` class derives from is the head of any Unity script. With this inheritance, the `PlayerController` can be attached to the game objects and respond to Unity's event functions including `Update` and `Start`. In the scope of the class two private properties are declared: `health` and `actionStore`. The `health` property is assumed to be a player health managing field whereas the `actionStore` property may be a field assisting in storing and managing player actions or abilities.

Furthermore, the code includes an inner struct called `CursorMapping` that is also decorated with `[System.Serializable]` attribute. The struct can thus be serialized and therefore shown and changed in the Unity Editor. The `CursorMapping` struct consists of three public fields, which are type, texture, and hotspot. The type field is of enumerated type `CursorType` which seems to provide list of types of cursors. The texture field is of `Texture2D` type and is for storing the cursor skin. The hotspot field is of `Vector2` type and indicates the coordinate area inside the cursor that is used for game object interaction.

In general, this part manages to establish all the elements related to the `PlayerController` class such as basic health and action variables, and even a struct for the desired cursor.

### **5.5.7 Respawner.cs**

This script handles the respawn systems of a player character as well as resetting the enemies whenever the player character dies in the game. The script contains several serialized fields that can be modified in the Unity Editor such as: `respawnLocation` (this is a `Transform` that dictates where the player should respawn), `respawnDelay` (this is a float that tells how long before the player can be respawned), `fadeTime` (this is a float that indicates how long the fade will take), `healthRegenPercentage` (this is a float that determines the percentage of health that will be returned to this character on respawn) and `enemyHealthRegenPercentage` (this is a float that determines the percentage of health that will be returned to enemies).

In the `Awake` method, the script subscribes to the `onDie` event of the `Health` component and binds the `Respawn` method to that event such that it will be triggered when the player dies. In the `Start` method, the script able to determine if the player is already dead or has just died at the start of the game and thus avoiding loss of game progress calls the `Respawn` method when needed.

The `Respawn` method commences a coroutine by the name `RespawnRoutine`, whereby the actual respawn of the player is done. The `RespawnRoutine` coroutine will first be stubbed in to wait for the time that `killRespawnDelay` has elapsed. Thus it locates a `Fader` component and does fade the screen out through the time set by `fadeTime`.

After the fade-out, the `RespawnPlayer` method is fired to teleport the player at the respawn point and to heal him. The enemy bionic is brought to an ordinary state by the `ResetEnemies` method in addition to the healing to the enemy of some parameter. Finally, the screen fades back in using the `Fader` component.

The `ResetEnemies` method retrieves all active game objects in the current scene with the component '`AIController`', and more especially finds out all the active units with a '`Health`' component and if they are not dead they have their state cleared and their health recovered fully or partly as per their max health."



“Return foam, PlayerE select move mode makes the delta between the current spot and the respawn location and then instead of moving the player uses NavMeshAgent component to warp the player. It also works in turning the health of the player back to a certain amount. In a situation where the active virtual camera in the scene is tracking the player, and the player has been warped, the method causes the virtual camera to update its position relative to the player’s position using OnTargetObjectWarped method.”

In general, this script implements a complex respawn algorithm adding health and enemies, introduction of fade effects while screen transitions and movement of cameras to avoid abrupt shooting transitions while depopulating during the respawn.

## **5.6 Movement**

Another very important part of the game, the Mover script is used by everything movable.

### **5.6.1 Mover.cs**

The code compliments with the creation of the Mover class which is Inherits from MonoBehaviour and implements two interfaces IAction, ISaveable. This script enables moving a game integer’s unity object through NavMesh and also saves and repositions his object.

The class starts First of all, he declares two serialized fields maxSpeed and maxNavPathLength which can be set from the fied. Unity Editor. These fields are used to control the maximum speed of the NavMeshAgent and the maximum distance that the agent will navigate or allows for navigation respectively. The class also declares two private fields navMeshAgent, a reference to the NavMeshAgent and health, a reference to the Health component attached to the game object.

In the Awake method, he initializes navMeshAgent and health fields by extracting from the gameObject the required components. The Update method is called for every frame rendered and it makes it so that the NavMeshAgent is only enabled when the gameObject is not to dead as determined by the IsDead method of the Health Component. It also calls the UpdateAnimator method to adjust animator’s parameters with respect to the movement of the object.

The CanMoveTo method is invoked to determine whether a game object can be moved to the specified destination.

A path is established through the NavMesh system and it is ensured that the path is closed and does not exceed the maximum length permitted and considering here it is the distance. If any of these conditions are not met, the method returns false.

The StartMoveAction method begins in the process of movement when it calls ActionScheduler’s StartAction method and after this moves the game object to the specified position moving to a specific speed fraction by the MoveTo method. MoveTo

is a method that initializes the destination and accelerates the NavMeshAgent to a preset cruising speed and above which it cannot be stopped.

The Cancel method completely disables the NavMeshAgent stopping all movements with respect to it. The UpdateAnimator method modifies the animator's forward speed parameter as a function of the NavMeshAgent's local velocity when the animator has to do object movements to the animator.

The GetPathLength takes a path around NavMeshPath and calculates its total distance by adding the distances between the pole corners ranging the pathway by sundry angles around the pole. This mechanism acts everything forward: for CanMoveTo it is necessary to know if it's going to take an amiss distance from the allowed one or not to exceed its maximum allowed limit.

The CaptureState method converts the current position and quaternion of the object into a set of serializable vectors. This 'capture state' could help in restoring the lost state to its original position.

The RestoreState takes in a saved state and reestablishes the position and the rotation of the game object. It, however, sets the NavMeshAgent to inactive at this stage to prevent any incongruities with the NavMesh system.

In summary, the Mover script encapsulates a set of advanced motion capabilities of a game object, including path following, animation synchronizing, and the object states management.

## **5.7 Scene Management**

This folder contain scripts about loading and unloading Scenes, fading between them and some Portal functionalities.

### **5.7.1 Fader.cs**

This script permits to control the fade-in and fade-out of UI elements by changing the alpha value of a CanvasGroup's component. The Canvas Group component incorporates to lighten the UI representation in Unity.

However, the Fader class also has two private variables, canvasGroup, and currentActiveFade. The canvasGroup variable is used to hold the reference of the CanvasGroup component which is situated on the same GameObject as the Fader script. The currentActiveFade variable is used to store the active fade coroutine, enabling the script to cancel nested coroutine calls by their child fades.

In the Awake method, the script makes canvasGroup available by implementing GetComponent() whereby it accesses the Canvas Group interior in the GameObject. This makes sure that the canvasGroup field is set correctly prior to the call of other methods.

The `AlphaOutImmediate` method changes the `canvasGroup`'s alpha value to 1 instantly and thus the UI pumpling the Action is fully covered immediately. This method does not include any form of animation or advance blending up.

The respective `FadeOut` and `FadeIn` methods define an operation's fade-out or fade-in motion and tell how long the motion will take. In both cases the `Fade` method is called with the acceptable for. both actions target alpha value and the duration.

The `Fade` method has the functionality of activating fade coroutine. If there is already active fade coroutine (`currentActiveFade`), the method cancels that crrnt active fade and presumes to launch a new fade. This is done in order to ensure that there is no more than one instance of fade in progress at the hime. The method then initiates the coroutine: `FadeRoutine` with the given target alpha value and duration, and records the active fade instance's handle in the active fade field.

The `FadeRoutine` coroutine implements a range of fading where the `canvasGroup` alpha value changes from the starting value to the target value within a specific time frame. It was projecting the alpha value through the usage of unsigned frame time (`Time.unscaledDeltaTime`). Because, this has nothing to do with the game time scale, one does not worry about the fade operations being compromised once the game scale is changed. The coroutine continues to be executed repeatedly until the alpha value reaches and is approximated to the region of the target value, at this time the coroutine releases and cannot be executed anymore.

All things considered, Fader's analysis gives an elegant usage for fading, animating UI elements in and out, bold and gradual in nature.

### 5.7.2 Portal.cs

The code introduces a script called `Portal` which is placed in the `Thesis.SceneManagement` namespace in the class of `Unity`. This script is created for the portal used in the game so that movement is made from one scene to the other. The script is derived from the `MonoBehaviour` class meaning that it is a class that can be included on Game Objects in `Unity`.

The portals of the system are controlled through a command defined in the script that is 'date portal'. To help achieve this, the beginning of the script imports several namespaces which include `System`, `System.Collections`, `Thesis.Control`, `Thesis.Saving`, `UnityEngine`, `UnityEngine.AI`, `UnityEngine.SceneManagement` phrases. This is important as it gives access to the use of different classes and methods required for the functionality of the script such as scene handling navigations as well as controlling the lance.

Within the portal class, an enumeration called `DestinationIdentifier` is defined having five states i.e. A, B, C, D, and E. This enumeration is used to identify different destinations that the portal can lead to.

The game with the respective portal portrays to the developers the respective telescopic view modics which include the planets focusing on the related parts for the

following: The related modesty matrices are discouragarily enlisted such that the interventional measures are appreciatively indicated but not humorous deviations related cause and ineffective reminder of dangerous face through certain recommended matters. The script encompasses several serialized fields some of which can be set through the unity editor. The fields are:

- `sceneToLoad`: A scene index integer, refers the loading scene when the portal is initiated. It is defaulted to 1 but the default means that no scene is set.  
`spawnPoint`: Specifies location where the player is spawned in the new scene.
- `destination`: Those particulars of the new scene which may be of interest contain the value of `DestinationIdentifier` as well.
- `fadeOutTime`: A float which signifies the amount of time it takes to transition from the current scene to the new scene in the case of a fade effect. This is normally set to 1 second.
- `fadeInTime`: A float which signifies the duration taken by the effect of fading in once the person has already arrived in the new scene. This time is also set to 1 seconds.
- `fadeWaitTime`: A float which signifies the period of time taken to do nothing between the fade out and fade in. This time is set to 0.5 seconds.

Such serialized fields also provide the means of modifying the camera's behaviors with regard to the portal without having to use scripts by merely using the Unity Editor. Better, designers may set different scenes, spawn points, and transition effects for every portal instance.

### 5.7.3 SavingWrapper.cs

The `SavingWrapper` class is an application of Unity scripting which is responsible for dealing with saving and loading of a game as well as changing from one game scene to another with the use of some fading. It derives from `MonoBehaviour`, meaning that it is a script which can be attached to `GameObjects` in Unity. First and foremost, the class provides several serialized fields which are now editable in the Unity Editor: `fadeInTime`; `fadeOutTime`; `firstLevelBuildIndex` and `menuLevelBuildIndex`. These fields in turn define the time length for fade in and fade out effects, the build index of first level scene and the menu level scene respectively.

The `ContinueGame` function, on the other hand, determines whether a save file exists, and if so, it will initiate a coroutine to resume the last scene that was left. In this case the `GetCurrentSave` method is used to obtain the save file name from the `PlayerPrefs` and see if the file exists using `SavingSystem` component. If both of these are correct, it proceeds to invoke the `LoadLastScene` coroutine that performs scene transition with fade effects. However, in the case where the `LoadLastScene` coroutine is concerned, a `Fader` component is used to effect the fade out and fade in effects, with the last scene being loaded as the last saved one using the `SavingSystem`. Therefore, to break the inertia that the user has bought when they are initializing a `NewGame`, the `NewGame` method will take the current save file and initialize in order to get the first scene loaded.



Afterward, using the `SetCurrentSave` method, the save file name is stored in the `PlayerPrefs` and then `LoadFirstScene` coroutine is called. The `LoadFirstScene` coroutine does the fade-out effect, Symmetrically the scene is loaded using `SceneManager.LoadSceneAsync` method and retains the fade-in effect at the end. Profoundly, the `LoadMenu` method activates the `LoadMenuScene` coroutine, which is designed to load the menu scene, and its working is similar. Fade out, load menu scene and fade in. In the same fashion the method `Update` defines handles the keyboard heroically for the purpose of save, load, delete, and other operations to be performed. If "S" sees the light of the day, it means that the current game is safe. Saving process is done by the `Save` method, which makes a call to `SavingSystem` component. In case "L" is the letter that is spotted, games that have previously been played are recalled to memory, through a `Load` method. The invoked method calls the `SavingSystem` component to load the game state. If the "Delete" key is pressed, the current save file is deleted using the `Delete` method, which calls the `SavingSystem` component to delete the save file. The current storage will be abandoned after the `Saving` wrapper class has also a method `list saves` which returns all the existing save files which can be called from the saving system class.

The method also helps to easily access all save files which can be beneficial when showing the player the save file options. In general, the `SavingWrapper` class addresses all the needs of game saving and loading as well as moving between scenes with fade effects and is also easily configurable in the Unity Editor.

## 5.8 Stats

This folder explains the rest of characters' attributes, statistics and all remaining characteristics, such as experience, level, etc.

### 5.8.1 BaseStats.cs

The code is an implementation of a class called `BaseStats` that is present in a Unity game project. This class is enclosed under the `Thesis.Stats` namespace and it also implements the `IPredicateEvaluator` interface, which, as the name suggests, indicates that this particular class might be evaluated in certain conditions or predicates from the game.

At the onset of the class, several namespaces are included: `GameDevTV.Utils`, `Thesis.Core`, and `UnityEngine`. These namespaces may include utility functions, core game logics and functionality specific to Unity based on their names.

In the `BaseStats` class, a number of serialized fields are also declared. These fields may make it easy to customize the, class's behavior since they can be adjusted from the Unity Editor while the logic remains intact, that is, not through code. The `startingLevel` field has an associated attribute `[Range(1, 99)]`, which is in order to say the least not going to be lower than one and higher than ninety-nine. This field is utilized to indicate the character's starting level. The `characterClass` field is of the `CharacterClass` type. This deems it necessary to state how characters with different

classes or occupations will perform in the game. The progression field links to a Progression object, which probably takes care of the advancement of the character through various levels. The levelUpParticleEffect field is a GameObject reference that will be created when a character levels up as a decorative effect. The shouldUseModifier field is a flag that specifies whether specific modifiers should be used in the character's stats.

The BaseStats class has also an event onLevelUp which avoids the tedious efforts of looking for other parts of the game to notify that the character has managed to level up. This event makes use of the Action delegate, therefore, it does not include any parameter to be passed to the subscribers.

Two private fields have been declared namely currentLevel and experience. The currentLevel field is of the type LazyValue which seems to be a utility class from the GameDevTV.Utils namespace providing lazy loading capability for the character's level. The experience field is an impertinence pointing towards an Experience entity which would most likely be associated with the character's Experience Points.

Lastly, there is a partial definition of the Awake method. This is a Unity script callback that is invoked when the game object this script is using is being booted up. This method is often used for setting up variables or game state prior to gameplay commencement. The code structure of the "Awake" method is not available within the text box above, however this seems to be the place where the basic settings up of the BaseStats class are carried out.

### 5.8.2 CharacterClass.cs

The code defines an enumeration (enum) named CharacterClass in C#. Enumeration represents a user-defined type consisting of a set of constant values called enumerator list. Here, it is used to express the type of a character that a player may encounter in a game – the CharacterClass enumeration.

The CharacterClass enum includes the following values:

- Player
- Grunt
- Mage
- Archer
- Soldier
- Warchief
- Peasant
- Shaman

### 5.8.3 DefenceOffenceDisplay.cs

This class is used for showing the attack and defense stats of the player character in the gameplay user interface. There are some fields, baseStats, attack, defence, attacklabel, defensive. The baseStats field is a BaseStats component attached to the

player character, which overviews the structure of the player orally. The attack and defence fields are float variables that depict the current strength of attack and defence. The last two fields are the attacklabel and defencelabel which are TextMeshProUGUI components that are used to convey the attack and defence values in the gameplay. In the visuals of the game respectively.

In the Awake method, the baseStats field is set up by locating the game object labelled Player and attaching the BaseStats component to that object. This guarantees that any time the game starts, the DefenceOffenceDisplay class has the requisite information about the player's stats. The Awake method is appropriate for such types of initialisation since it is just what it is called - close to the beginning of all other procedures in the Unity lifecycle. It is called when the script instance is being loaded.

The Update method is invoked as often as the game generates a new frame, that is, once a second, and in this case is used for painting both attack and defense values that are changing throughout the period.

In this way, these fields are changed by calling the GetStat method of the baseStats object in the professional way including Stat.Damage and Stat.Defence in the arguments. This sunshine is also concerned with the attacks and defense stat of the player. The values are then converted to text and set to the Text property of the attacklabel and defencelabel controls as result the words displayed on the user interface are changed and updated.

In general, together with the DefenceOffenceDisplay class, there is always a system that enables real time updating of users' attack and defence statistics based on the fact that such information is available at the interface. This is done with the help of the lifecycle methods of Unity and display of text with the TextMeshProUGUI components.

#### **5.8.4 Experience.cs**

This class is an auxiliary class in charge of the experience of a player character: increasing the experience the character earns, the experience load and save logic, and anything related to items associated with some experience level or gain scenarios in this case.

So the class defines a serialized field experiencePoints which is initialized with a value of 0. This is the amount of experience points currently possessed by the player. The [SerializeField] attribute enables this field to be set directly in the Unity Editor and it thus facilitates the initial definition of this field or its change during the game development.

In this case an event onExperienceGained which is declared may be implemented since certain parts of the code can wish to listen to when a player character earns experience. This means that the subscribers do not receive any parameters with the event because this event is using Action delegates.

In order to award experience points to the player, the GainExperience method is called and the experience points are added to the player's. It accepts a float parameter

experience which is the amount of experience to be added. Inside this method, the value of the `experiencePoints` field, is increased by the amount of experience value, and there is raised the `onExperienceGained` event in order to inform the subscribers that some of the Experience has been gained.

The `GetPoints` returns parameters which have – the current experience points.

This particular technique is applicable to other sections of the game that requires interaction with the player experience points.

The `CaptureState` method comes with the `ISaveable` interface and serves to capture the current experience point state. It returns the `experiencePoints` variable which can then be serialized and saved.

The `RestoreState` method comes with the `ISaveable` interface and serves to load a previously saved experience point state. It has one parameter `state` and converts that to a float which is stored in the `experiencePoints` variable.

The `Update` method is a unity lifecycle method that gets called every frame after each rendering is completed. In this method, it is checked whether or not the “E” key has been pressed. Should that be the case then `GainExperience` is called, passing in `Time.deltaTime * 1000`, which gains experience point depending on time that has passed since the last frame update.

The `AddItems` method is derived from `IItemStore` interface and explains how items that give experience points should be treated. It takes an `InventoryItem` parameter `item` and an integer parameter `number`; the latter represents how many such items ought to be added.

If the item is of type `ExperienceAward`, the `GainExperience` function is invoked and the parameters passed to this method are multiplied by the number of items to be added. Then the method returns how many new items were added. In case, the item is not of type `ExperienceAward`, the method would return zero.

In general, the `Experience` class offers a thoroughly working system for the experience points management in a Unity game, everything from gaining experience, saving and restoring an experience state to dealing with experience based items.

### **5.8.5 ExperienceAward.cs**

This code that implements a Unity `ScriptableObject` called `ExperienceAward` which is derived from `InventoryItem`. This class is presumably created for an item that gives experience points to the player when used. The `[CreateAssetMenu]` attribute is very useful in this case as it helps in creating instances of this class from the unity editor. More specifically it adds the ability to create new assets of type `ExperienceAward` through the `Create` menu in the Unity Editor. The `fileName` parameter indicates the default file name for newly created assets and the `menuName` parameter indicates the place in the `Create` menu where this option will be available.



There is a single serialized field in the class, `experienceToAward` where its value is an integer and it is the amount of experience that player will be getting from using that item. The `[SerializeField]` attribute provides a way to add this field in the Unity Editor but editors will be able to modify the amount of experience points from the editor and that field will not have to be made public.

The `GetExperienceToAward` method is a method which can be accessed outside the class and whose return type is the `experienceToAward` integer field. This helps to this particular aspect since it enables other aspects of the game to determine how many experience points this item provides to the player.

Such concealment is a standard procedure in object-oriented programming where by this class contains the variable in question in a method which is the construct of encapsulation. This helped in changing the internal representation of the experience points without any changes in the rest of the code which use this class.

To summarize, the class `ExperienceAward` offers a great solution to design and implement in-game objects that grant experience points in a Unity project. By using `ScriptableObject` and the `CreateAssetMenu` attribute The Unity Editor supports rapid design of items that grant experience points.

### 5.8.6 ExperienceDisplay.cs

The code implements a class known as `ExperienceDisplay` which is a Unity `MonoBehaviour` class that implements the functionality of showing the experience points accrued by the player on the UI of the game. This class also makes use of the `Experience` component located on the player character in order to show the amount of experience points the player currently has.

The class has a private variable `experience` of type `Experience`. This variable is used to keep a reference to the `Experience` component present in the player character. The Unity specific function `Awake` several scriptable objects have, is used to set this variable when the scriptable object is loaded. Therefore, within the `Awake` method, the `experience` field is assigned a value by locating the player tagged game object and obtaining the `experience` component attached to it. This ensures that the `ExperienceDisplay` class will have the player's experience points as early as the game commences.

A Unity function which is meant to be called once per frame is called the `Update` method. Inside this method, there is a `TextMeshProUGUI` component which is on the same game object with the `ExperienceDisplay` script which is called. The text of this component is then filled with the formatted string depicting the current experience of the player.

For the purpose of obtaining the total amount of experience, the `experience.GetPoints()` method is called, and to display experience in plain string format, the `string.Format("{0:0}", experience.GetPoints())` method comes in handy. This makes sure that the text visible to the user on the user interface is changed throughout every frame so that the amount of experience points displayed to the user is relevant.

Overall, the ExperienceDisplay class is quite useful with connection to how it utilizes Unity component system and updates the text in real time on the player's experience points.

### 5.8.7 IModifierProvider.cs

The code is a definition of an interface, IModifierProvider in the language C#. An interface in C# is a distinct declaration that specific methods or properties must be defined in the classes that inherit from that particular interface. By themselves, interfaces do not deal with any implementation; they are devoid of description of the methods or properties except for their title.

The IModifierProvider interface declares two methods: GetAdditiveModifier and GetPercentageModifier. Both methods expect a Stat type as a parameter and return an IEnumerable. The return type IEnumerable suggests that these methods are going to return some numbers that are mainly in the form of floating marks and can be iterated over using for each (as in the case of LINQ operations).

The GetAdditiveModifier method aims at returning all the stereoscopic or extra modifiers that can be added to the primary spec and stat under consideration. Additive modifiers are values which are incorporated into the stat's base values. For example, as standing plain that the base val of a stat is ten, there are two additive modifiers with values two and three, then the final stat value would be fifteen;  $(10 + 2 + 3)$ .

The GetPercentageModifier method is focused on passing a set of percentage modifiers applicable for the Stat. Percentage modifiers are those values that affect the fundamental amount of the attribute in question by a particular percentage.

Assume that the base value of a stat is 10, there are two percentage modifiers of 0.1 (10%) & 0.2 (20%), in this way the final value of the specified statistic will be equal to 13  $(10 + 10 \cdot 0.1 + 10 \cdot 0.2)$ .

With this design, each class has a uniform protocol regarding the provision of stat modifiers' classes. All the members of IModifierProvider, whoever comes in, should ensure that they implement the GetAdditiveModifier and GetPercentageModifier methods in order to allow them to provide any class of modifiers for the stat. This encourages a predicable and consistent approach towards dynamic modification of different variables since various classes such equipment or buffs or active skills can adopt this interface to modify stats.

### 5.8.8 LevelDisplay.cs

The code implements a glissando like LevelDisplay class that inherited from the el MonoBehaviour class and is in charge of the tracking the players level and experience points XP, as well as how it can be depicted on the interfaces of the game. This classes also exchanges information with other classes such as BaseStats and Experience in order to gain the needed information for the display.

The class has declared additional private members `baseStats`, `experience`, `textField` and `slider`. The `currentBaseStats` and `currentExperience` members store references of the `BaseStats` and `Experience` components of the player character respectively. The `textField` field is a reference to the `TextMeshProUGUI` which is a component used to text within the user interface. The `slider` field as the `[SerializeField]` attribute is a component referring to a `Slider` which is used to depict towards what extent has been the character leveled up to the next stage.

The class also disengages with as many variables like current level movement in this case `currentLevel`, `xp ever possessed by the player` `currentXP`, `xp needed to level current xp to level special status activities or xpToLevel` while `xpOrRested` previous level status `xp()`{ These variables help trace the movement of the player in the game as well as present their possession.

`Awake` is one of the Unity lifecycle methods that can be called in the case where the script instance is being loaded.

In this method, the `baseStats` and `experience` fields are initialized by finding the game object tagged as "Player" and getting the `baseStats` and `experience` components. The `textField` field is initialized in the same way by getting the `TextMeshProUGUI` component from the game object on which `LevelDisplay` script is attached to. The `Update` method is also another Unity lifecycle method this time with a large margin it is called once in a frame. In this method, the `currentLevel`, `currentXP`, `xpToLevel`, and `previousXP` variables are updated by calling appropriate methods on `baseStats` level and `experience xxx` as a component. Again the `textField.text` property of the `LevelDisplay` class is set to another formatted string that states level and XP progress of the player. The `slider.value` property was also set to level up progress of the player more accurately the current XP string `s` previous XP string divided by the XP needed to level up string `s` previous XP. In general this is a simple and straightforward way to realtime render and display the player's level and the XP accumulated within that level using unity lifecycle methods and level display class.

### 5.8.9 Progression.cs

The code in question includes an excerpt from a Unity C# script called `Progression.cs`. This particular script appears to be in charge of the progression system in a game where a character class is defined along with its statistics and how they evolve with leveling up. So the first part shows the end of a method or a block which enables filling up a look up table. This lookup table is used in order to associate the character's classes with the respective stats and levels. In the excerpt one can see that for each `progressionClass` a dictionary `statLookupTable` is created which keeps a record of each stat and its relevant levels against a given `progressionClass`. This `statLookupTable` is then further inserted into the primary `lookupTable` with the `characterClass` as the primary key. Such structure is useful as it is able to retrieve quickly any stat progression related data per character class. After this, the code comprises classes, `ProgressionCharacterClass` as well as `ProgressionStat` both of which are inner classes, and both have been fitted with `[ System. Serializable ]` attribute. Such methods proves

to be of great importance as it helps in maintaining persistent data and it ensures that the said classes are visible within editor in inspector in unity's user interface for their manipulation. Here the ProgressionCharacterClass consists of two public fields, one is of type CharacterClass characterClass and another one is an array of ProgressionStat objects called stats.

This class stands for a class of character present in the game and its relevant characteristics.

The ProgressionStat class has the following two public fields: a stat of Stat type and levels which is a float array. This will be then used to define progression of specific stat at various levels. The levels array contains values presenting levels of the stat where each index corresponds to each level of the game.

Apart from that, this particular piece of code is embedded within a subsystem that controls the progressive development of various stats of different classes of characters in a certain game. The data is organized into classes and certain tables are used to speed up the process of collection of progression data.

#### **5.8.10 Stats.cs**

The code that has been provided creates an enum called Stat in C#. An enum is a user defined data type which has a defined list of constants called the enumerator list. Here, the Stat enum is useful in quantifying character's statistics or attributes in a game. The members, the Stat enum consists of include the following:

- Health
- ExperienceReward
- ExperienceToLevelUp
- Damage:Damage
- Mana:Mana
- ManaRegenRate
- TotalTraitPoints
- BuyingDiscountPercentage
- Defence

#### **5.8.11 TraitLogic.cs**

The code is taken from the Unity C# script TraitLogic.cs. This script handles how trait bonuses will be applied to the different stats of a character within the game. The code implements the use of a dictionary called percentageBonusCache which serves as a cache for the percentage bonuses that are lockup for various stats and traits. This type of caching probably offers some efficiency as it ensures that such calculations are not done repeatedly.

The bonusConfig field is an array of TraitBonus objects, gauged in the figure, which has been tagged with the [SerializeField] feature. This feature enables this field to be imaged and altered using the Inspector of the Unity Editor, thereby enabling the



designers to set trait bonuses in place of altering the actual code. The class TraitBonus has the attribute [System.Serializable] assigned, meaning such objects are capable of being saved which is critical in Unity. The TraitBonus class has four public attributes: trait (of the Trait type), stat (of type Stat), additiveBonusPerPoint (Type float and initialized to 0) and percentageBonusPerPoint (also Type float and initialized to 0). These fields describe the trait itself, the stat which this affects, and the additive and percentage bonuses per point of the trait, respectively.

One of the integer fields is called unassignedPoints and it is set to the value 10 which indicates the number of trait points that have been left untapped or which have not been assigned to any traits.

There is a probable belief in this game that players can earn and invest points worth rewards to different attributes as they play.

Awake is a Unity life cycle method that is executed when the script instance is being loaded. Such variables include, but are not limited to, the usual everyday dialect of coin and attribute called the additiveBonusCache and percent gauge known as percentageBonusCache. Each one of those dictionaries corresponds to each stat with another dictionary that corresponds each trait with its value of bonus. So, the method processes the bonusConfig array for populating those caches. In addition, for each TraitBonus which sits in the list bonusConfig, the code examines whether the additiveBonusCache encompasses already one entry for the specific Stat defined by TraitBonus. Otherwise, it makes more dictionaries to each Stat more than once. This setup makes sure that the caches have been properly set up and they will be containing bonus values for every Stat and Trait combination.

### **5.8.12 Traits.cs**

The code creates an enumeration (enum) construct called Trait in C#. An enumeration is a self-contained type comprising an enumerator list that contains a unique name for each constant. In this context, the Trait enum is utilized to denote different property or characteristic of a character in a game or an application.

The members of the Trait enum are the following:

- Strength
- Dexterity
- Constitution
- Intelligence
- Wisdom
- Charisma

## 6 Conclusion

---

This thesis documents the entire cycle from concept to design and then to the production of a role-playing game (RPG), presenting a case in project management, game design and software application development.

The do on the project started with a lofty idea: making an RPG that is heavily story driven, with opportunities for detailed character creation, and a believable and interactive game world. As initiated in the planning of the project, there was a lot of eureka moments where we started thinking of the story of the game, how it would be played and how it would look like visually. Nevertheless, the project's size almost from the very start turned out to be greater and this is when came the first substantially darkening aspect of the picture.

When working on the development, there was a lot of trouble beating down the team's rate. Their problem solving and resilience were constantly put to the test as the entire development policy evolved. Certain technical challenges such as the design and incorporation of a multifaceted conversation system and ensuring good performance in vast expansive spaces posed great problems, mostly it ended up with long nights filled with either debugging or redefining: a better word for rewriting programs. The construction of the narrative got quite difficult especially since there was the need for a good story with the appropriate willingness to let the player build their way through the plot with several turning points.

One of the most significant setbacks occurred midway through development when playtesting revealed that core gameplay mechanics weren't as engaging as hoped. This led to a difficult but necessary decision to redesign several key systems, causing delays but ultimately resulting in a more enjoyable player experience.

In spite of these problems, however, the project also enjoyed a dew accomplishments. In this regard, how the team accomplished the cross team challenges was boosting team spirit and promoting creativity. The design approach, while rather prolonged, produced refined game play mechanics and a well assimilated progression system. One of the highlights of the accomplishments was the good element of game in which players created characters whereby several options were given and players' decisions mattered.

Every single detail of the game was distinguished thoroughness of work on every detail. Here it became evident how vitally important is the clarity of the issued tasks and the organization and planning of the workload. Final efforts were put to development completion which revolved around testing, fixing bugs, and enhancing the performance of the game to guarantee player satisfaction immediately a game is launched.

Having look at the whole process of development, this thesis points out some few things to be taken into account:

- Having proper and clear scope and expectations in game development is one of the most difficult areas to be mastered.

- Reminding oneself through one's previous folds and problems investigating the project, it means that there is a sense of respect for one's very imagination.
- The understanding of appropriate intervals for playtesting and how bold one should be in making changes based on the feedback received.
- Advantages of game development practice that is inclusive of many people working in different areas of expertise: design, development, information and knowledge management, and evaluation.

Concisely, even if the construction of this RPG brought many difficulties and took more time and resources than was supposed at the very beginning, it provided useful experience some hands-on project management skills, teamwork and the basics of game construction. The outcome of the project justifies all efforts and creative resource of the team and the course of development has assured that such valuable experiences will be useful to any forthcoming projects.

## 7 Bibliography

---

- Courses and Tutorials from GameDev TV: <https://www.gamedev.tv/>
- Low Poly Assets from Synty: <https://syntystore.com>
- Assets from Asset Store: <https://assetstore.unity.com>
- Unity Documentation: <https://docs.unity.com/>
- Unity Ui doc: <https://docs.unity3d.com/UI>
- Unity scriptable object doc: <https://docs.unity3d.com/class-ScriptableObject>
- Unity Prefab Doc: <https://docs.unity3d.com/Manual/Prefabs>
- Unity Custom Editor Doc: <https://docs.unity3d.com/ManualHowTo-CreateEditorWindow>
- Unity Scene Manager Doc: <https://docs.unity3d.com/Manual/Scenes>
- Songs : <https://assetstore.unity.com/packages/audio/music/total-music-collection>
- Terrain Assets from Sics: <https://assetstore.unity.com/packages/toon-fantasy-nature>

## 8 Abbreviations & article-acronyms table

---

1. RPG: Role Playing Game
2. NPC: Non-Player Character
3. HP: Hit Points
4. MP: Mana Points
5. EXP or XP: Experience Points