



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

Thesis

Title	(English) Rise of the Village Hero: A Roleplaying Game: Dialogue System, Inventory System, Quest System, Shop System, User Interface, Item Creation and Combat (Greek) Rise of the Village Hero: Ένα παιχνίδι ρόλων: Σύστημα Διαλόγων, Σύστημα Αποθέματος, Σύστημα Αποστολών, Σύστημα Αγοραπωλησιών, Διεπαφή Χρήστη, Δημιουργία Αντικειμένων και Λειτουργία Μαχών
Student's Full name	IOANNIS PETRIS
Father's name	DIMITRIOS
Registration Number	Π18125
Supervisor	Themis Panayiotopoulos, 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.

The copying, storage, and distribution of this work, in whole or in part, for commercial purposes is prohibited. Reprinting, storage, and distribution for non-commercial, educational, or research purposes is permitted, provided that the source is cited, and this message is retained. The views and conclusions contained in this document express solely the author's perspective and do not represent the official positions of the University of Piraeus. As the author of this work, I declare that this work is not a product of plagiarism and does not contain material from unreferenced sources.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Themis Panayiotopoulos, for his invaluable guidance, feedback, and support throughout my research. His extensive knowledge and experience were instrumental in the completion of this Thesis Project.

I am also extremely thankful to my classmate and co-researcher, George Samaras, for his help completing this project and his support over the years.

Last but not least I would like to thank my friends and family for their support, understanding and belief in me and my dreams.

Abstract

This Thesis Project is a 3D Video Game built using the Unity 3D Engine, an free tool used for the creation and publication of all kinds of video games. The type of game I chose to create is a Role Playing Game with movement and controls inspired from games like Diablo, Baldur's Gate and Dragon Age. The player in this game must venture through the virtual world created and save a small village from the threat of the goblin army that has been amassed by the Goblin King. This threat is communicated to the player from the first moment so he can be immersed in the game world. By using the systems I created, such as the Dialogue, Inventory and Quest systems he has all the tools to defeat the goblins and become a legend.

Key words: Unity 3D engine, Video Game, Quests, Dialogue, Combat, Inventory

Περίληψη

Αυτή η διπλωματική εργασία είναι ένα τρισδιάστατο βιντεοπαιχνίδι που κατασκευάστηκε με τη χρήση της μηχανής Unity 3D Engine, ένα δωρεάν εργαλείο που χρησιμοποιείται για τη δημιουργία και δημοσίευση όλων των ειδών βιντεοπαιχνιδιών. Το είδος του παιχνιδιού που επέλεξα να δημιουργήσω είναι ένα Role Playing Game με κίνηση και έλεγχο εμπνευσμένο από παιχνίδια όπως το Diablo, το Baldur's Gate και το Dragon Age. Ο παίκτης σε αυτό το παιχνίδι πρέπει να περιπλανηθεί στον εικονικό κόσμο που έχει δημιουργηθεί και να σώσει ένα μικρό χωριό από την απειλή του στρατού των καλικάντζαρων που έχει συγκεντρώσει ο βασιλιάς των καλικάντζαρων. Η απειλή αυτή γνωστοποιείται στον παίκτη από την πρώτη στιγμή, ώστε να μπορέσει να βυθιστεί στον κόσμο του παιχνιδιού. Χρησιμοποιώντας τα συστήματα που δημιούργησα, όπως τα συστήματα διαλόγου, απογραφής και αναζήτησης, έχει όλα τα εργαλεία για να νικήσει τους καλικάντζαρους και να γίνει θρύλος.

Λέξεις-κλειδιά: Unity 3D engine, Video Game, Quests, Dialogue, Combat, Inventory

1 Contents

Copyright ©.....	i
Acknowledgements.....	ii
Abstract	iii
Περίληψη	iii
Table of Figures	vii
Introduction.....	1
Running the project.....	2
1 Core Scripts.....	3
1.1 Action Scheduler and IAction.....	4
1.2 Condition and IPredicateEvaluator	5
1.3 Persistent Object Spawner	5
2 Dialogue System.....	6
2.1 Dialogue.cs	6
2.2 Dialogue Logic and NPC Dialogue Logic	7
2.3 Dialogue Node.....	8
2.4 Dialogue Trigger	9
3 Inventory System	10
3.1 Stats Equipment and Equipable Items	10
3.2 Random Dropper.....	10
3.3 Purse and Currency Item.....	11
3.4 Collector	11
3.5 Item Creation.....	12
4 Shop System	13
4.1 Shop.....	14
4.2 Shop Item.....	15
4.3 Shopper.....	16
5 Quest System	17
5.1 Quest	18
5.2 Quest Completion.....	18
5.3 Quest Giver	18
5.4 Quest Status.....	19
5.5 Quest List.....	21

5.6	Death Counter	22
5.7	Achievement Counter	22
6	User Interface	23
6.1	Main Menu	23
6.1.1	Game Related Windows	23
6.1.2	Settings Window	24
6.1.3	Information Window	25
6.2	In Game Menu.....	26
6.2.1	Inventory.....	26
6.2.2	Dialogue	27
6.2.3	Quest.....	27
6.2.4	Shop.....	28
6.2.5	Stats	28
6.3	Heads Up Display.....	29
6.3.1	Mana, Life and Level Display.....	29
6.3.2	Pause Menu	30
7	Combat.....	31
7.1	Fighter	32
7.2	Aggro Group.....	33
7.3	Combat Target.....	33
7.4	Projectile	34
7.5	Weapon Config.....	34
7.6	Weapon Pickup	36
7.7	Wave System	36
8	Game Walkthrough	38
8.1	Main Menu	38
8.1.1	Main Menu Screen	38
8.1.2	Settings Screen	38
8.1.3	Information Screen	39
8.1.4	New Game Screen	39
8.1.5	Load Game Screen	40
8.1.6	Continue and Quit Buttons.....	41
8.2	Playthrough and UI.....	41
8.2.1	Player's Graphical Interface.....	41

8.2.2	Pause Menu	42
8.2.3	First Steps	43
8.2.4	Equipment and Inventory Interface	44
8.2.5	Quest Interface	46
8.2.6	Leveling Up	47
8.2.7	Shop Interface	49
8.2.8	Portals	50
8.2.9	Spells and abilities	51
8.3	Rest of the Walkthrough and side Quests.....	53
8.3.1	Priest's Quest	53
8.3.2	Mayor's Quest	55
8.3.3	Goblin Camp	55
8.3.4	Fighting Goblin Camp.....	58
8.3.5	The Goblin Cave.....	60
9	Conclusions and Future Plans	61
10	References and Citations	62

Table of Figures

Figure 0-1. Project's Decompressed files	2
Figure 1-1 Core Prefab	3
Figure 1-2 Action Scheduler Script	4
Figure 1-3 Persistent Object	5
Figure 2-1 Dialogue Editor	6
Figure 2-2 Dialogue Components Example	7
Figure 2-3 Dialogue Trigger Script	8
Figure 2-4 Part of Dialogue Scrip	9
Figure 3-1 Item Scriptable Object	11
Figure 3-2 Weapon Scriptable Object	12
Figure 4-1 Shop Code Part One	13
Figure 4-2 Shop Code Part Two	15
Figure 4-3 Shop Component	16
Figure 5-1 Quest Class	17
Figure 5-2 Quest Components Example	18
Figure 5-3 Quest Object Example	20
Figure 6-1 Main Menu UI	23
Figure 6-2 New Game UI	23
Figure 6-3 Settings UI	24
Figure 6-4 Information UI	25
Figure 6-5 Inventory UI	26
Figure 6-6 Dialogue UI	27
Figure 6-7 Quest UI	27
Figure 6-8 Shop UI	28
Figure 6-9 Stats UI	29
Figure 6-10 Heads Up Display	29
Figure 6-11 Pause UI	30
Figure 7-1 Fighter Script	31
Figure 7-2 Weapon Config	35
Figure 7-3 Wave Serialized Fields	37
Figure 8-1. Main Menu Screen	38
Figure 8-2. Settings Screen	39

Figure 8-3. Information Screen	39
Figure 8-4. New Game Screen	40
Figure 8-5. Load Game Screen	40
Figure 8-6. First Steps in Game	42
Figure 8-7. Pause Interface	42
Figure 8-8. Pickups	44
Figure 8-9. Equipment and Inventory Interfaces	44
Figure 8-10 Equipping Items	45
Figure 8-11 First Quest: Save the villager	45
Figure 8-12. Dialogue Interface 1	46
Figure 8-13. Dialogue Interface 2	47
Figure 8-14. Leveling Up	47
Figure 8-15. Attributes Interface	48
Figure 8-16. Defeating Enemies in Evercoast Village	49
Figure 8-17 Shop Interface	49
Figure 8-18. Quests Window.	50
Figure 8-19. Village's wizard.	51
Figure 8-20. Retrieve the wizard's staff	52
Figure 8-21. Equipping Spells	52
Figure 8-22. Casting a spell	53
Figure 8-23. Village's Priest	53
Figure 8-24. Village's Graveyard	54
Figure 8-25. Village's Mayor	55
Figure 8-26 Goblin Camp Cutscene	56
Figure 8-27. Mysterious Ruins	56
Figure 8-28. Secret Door	57
Figure 8-29. Underground Ruins.	57
Figure 8-30. The Dark Blade	58
Figure 8-31. Secret Path	58
Figure 8-32. Fighting the Goblin Camp	59
Figure 8-33. Ruins Shopkeeper	59
Figure 8-34. Goblin Cave	60
Figure 8-35. Goblin War chief	60

Introduction

This Thesis Project is a 3D video game built with the Unity3D Engine. It is a Role Playing Game where the player plays an adventurer saving a village from the attacks of a vicious goblin horde.

The main influence for this project were two classes I had on my last year of university called "Virtual Reality" and "Video Game Technologies". These classes were taught by Mr. Themis Panayiotopoulos and were used as the base for this Thesis.

There I learnt for the first time how to create a small game with the Unity engine and I decided that is what I want to do. So I decided my Thesis project to be a video game built with these tools.

The game mimics other Role Playing Games like Diablo, Baldur's Gate and more. It takes inspiration from them but with a little twist. This is mainly because I did not have the resources to create levels and mechanics like these games, even though I would love to. So with the limited resources I had I created a low poly and simplified version of those games, while keeping the main mechanics of an Rpg game.

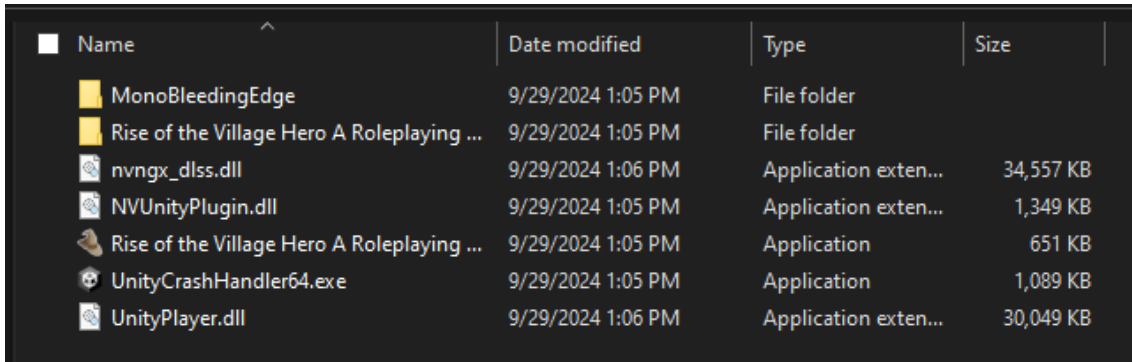
The game consists of five scenes all with unique visuals and style while keeping a similar tone. There are villages, goblin camps, hidden cursed temples and many more locations to explore. The character must venture through all these locations and save the region from the goblin threat, while using all the tools on his disposal. From items on the ground to items and spells bought or learned from the local wizard, to the experience gained through combat. There are several quests for the player to complete and view the story of the game and numerous Non player characters to to commune with and immerse yourself inside the world of the game.

I created the game for this Thesis with my co student George Samaras. We both wanted to create a video game for our projects so we decided to co create one game together and use this as the base for our research.

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 will 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	
nvngx_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 0-1. 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.

1 Core Scripts

These are the essential scripts of the game. They are used on every scene on every moment the game runs. They are mostly related with the game's camera and the low-level control of the player's actions and the implementation of the condition system used by many other frameworks in the game.

Most of these scripts are located as components on the Core Object prefab which is loaded on each scene, and it is never destroyed or discarded while the game is running.

Table 1-1

Script Name	Brief Explanation
Action Scheduler	Class for action scheduling
Camera Controller	Controls the Camera
Camera Facing	Controls the rotation of objects to face the camera
Camera Layer Toggle	Toggles on and off different layers
Follow Camera	Script for the main camera
Condition	Condition logic
Destroy after effect	Destroy an object as an event
IAction	Interface for action aborting
IPredicateEvaluator	Interface to evaluate conditions
Persistent Object Spawner	A spawner to spawn an object that does not unload

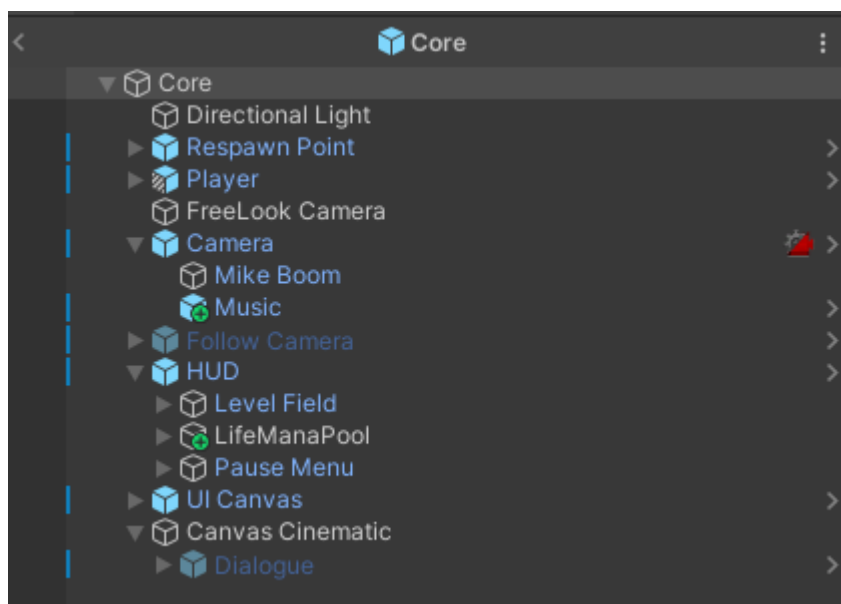


Figure 1-1 Core Prefab

1.1 Action Scheduler and IAction

This class and interface are used throughout the game for controlling the players actions and schedule them. When a new action is being called the action scheduler is first stopping the previous one and then starts the new one. Some basic actions that are being controlled by this are move, attack, shop, talk to someone and pickup items.

Overall, the ActionScheduler class can 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.

```
11 references
7   public class ActionScheduler : MonoBehaviour
8   {
9       4 references
10      IAction currentAction;
11
12      7 references
13      public void StartAction(IAction action)
14      {
15          if (currentAction == action) return;
16          if (currentAction != null)
17          {
18              currentAction.Cancel();
19          }
20          currentAction = action;
21      }
22
23      4 references
24      public void CancelCurrentAction()
25      {
26          StartAction(null);
27      }
28  }
```

Figure 1-2 Action Scheduler Script

1.2 Condition and IPredicateEvaluator

This class and this interface implement game wide condition evaluation to be used by almost everything. Check if an item can be equipped or not, or if a quest condition has been met or even select the right dialogue based on the player status in the world.

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 conduct the evaluation. The method returns a nullable Boolean 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.

Overall, this piece of code is intended to check the collection of evaluators implementing a predicate under a condition whether it must 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.

1.3 Persistent Object Spawner

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. It consists of three game objects, the fader, the saving system and the event system. The first two are responsible for the saving of the game so they must be active all the time and not unload in between scenes. They also must be unique and every scene must contain only one set of these items.

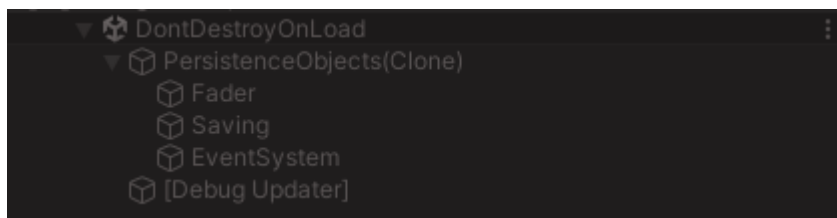


Figure 1-3 Persistent Object

2 Dialogue System

An rpg game without a dialog system is not an rpg game, so a custom Dialogue Editor was created. A custom editor in unity is a really powerful tool, but they are somewhat difficult to make. The editor in this game is aa rather simple one. Dialogue consists of nodes connected to one another. Each node is one dialogue line. When you have ten or fifteen nodes in a row you have a proper dialogue.

Table 2-1

Script Name	Brief Explanation
Dialogue	The core script of dialogue module containing the logic and the implementation of the dialogue scriptable object
Dialogue Logic	Contains the dialogue logic for the main player
Dialogue Node	This class is used to create node objects
Dialogue Trigger	An event to do something when something happens in a dialogue
NPC Dialogue Logic	Contains the dialogue logic for the non player characters

2.1 Dialogue.cs

This code is used to create a scriptable object to contain in it all the dialogue information. This class is designed to manage dialogue nodes in a Unity game, allowing for the creation, deletion, and organization of dialogue sequences.

Inside the editor for each Dialogue object you can create new dialogue nodes for the player, new nodes for the non player character. And then for each dialogue node you can add text, define who is talking , add triggers to activate evens when the dialogue is played.

The class contains a list of Dialogue Node objects, which represent individual dialogue entries, and a dictionary to quickly look up nodes by their names. The Awake method initializes the dialogue by calling Populate Dialogue, which fills the dictionary with node. The Populate Dialogue methods both clear the dictionary and repopulate it by iterating through all nodes. This ensures that the dictionary accurately reflects the current state of the node list. The GetAllNodes method returns all nodes, while GetRootNode returns the first node in the list, which is assumed to be the root. Overall, this class provides a framework for managing dialogue in a Unity game.

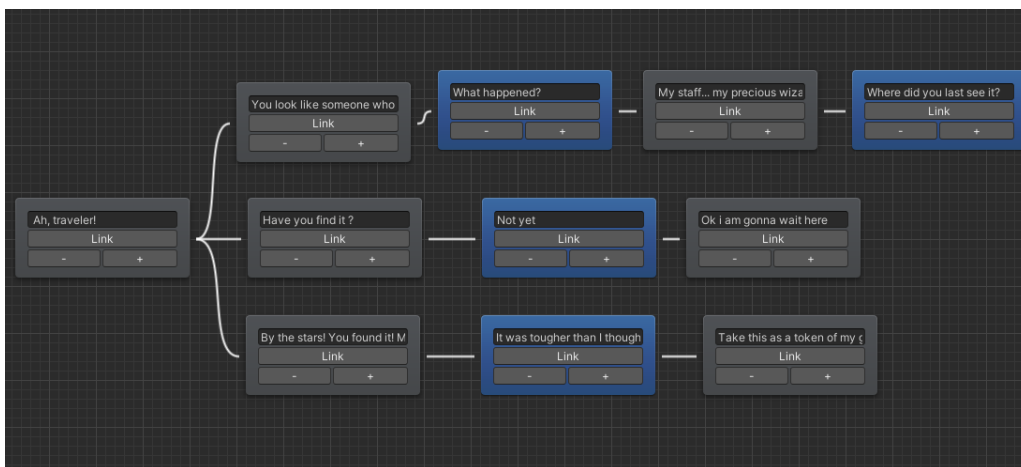


Figure 2-1 Dialogue Editor

2.2 Dialogue Logic and NPC Dialogue Logic

These Scripts manage the flow of the dialogue between the player and the non player characters. The first one because it enables a new action for the player implements the IAction interface.

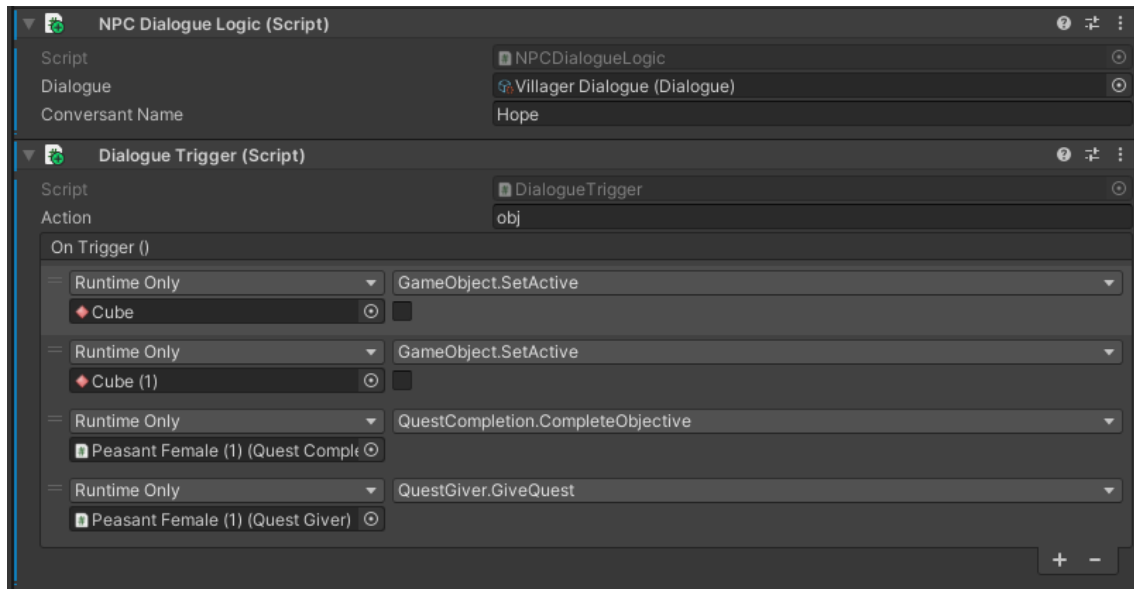


Figure 2-2 Dialogue Components Example

The player one has several private fields to track the current dialogue, the current dialogue node, the target dialogue, and the state of the dialogue (e.g., whether the player is choosing a response). It also holds references to the current and target NPCs involved in the dialogue and a serialized field for the player's name.

Most of the methods inside the class control the dialogue flow, the StartDialogue method initializes a new dialogue with a given NPC, setting the current dialogue and node to the root node of the new dialogue. There is also a method to quit the dialogue and reset all relevant fields and triggering any exit actions associated with the current node. The onConversationUpdated event triggers on a dialogue node change and can be called by many other systems, to do many things in response to a dialogue.

The IsActive and IsChoosing methods will tell whether a dialogue node is active and whether the user is selecting a dialogue response respectively. The GetText method provides the user with the current dialogue node Active text or an empty string in case no active node exist. The GetCurrentConversantName method provides the name of the current conversant which in case of response choosing is the player's name and otherwise is the NPC's name.

The `SelectChoice` method allows the player to select a dialogue choice, setting the current node to the chosen node, triggering any enter actions, and advancing to the next node. The `GetChoices` method returns a filtered list of player response nodes based on certain conditions. The `Next` method advances the dialogue to the next node, either allowing the player to choose a response or selecting a random NPC response if no player responses are available. If no further nodes are available, the dialogue ends.

In this paragraph, the HasNext method looks for further nodes to go forward with. The FilterOnCondition method is used to filter a list of dialogue nodes, employing the GetEvaluators method to access components that evaluate to the desired dialogue nodes considering some criteria. The TriggerEnterAction' and TriggerExit Action' methods activate any responsible actions on entering and leaving the current node respectively using the TriggerAction method to perform the actions in that action block.

The StartDialogueAction' helper method starts the new dialogue action with a certain NPC and removes any previously associated conversant and uses the ActionScheduler' component to initiate the new sequence of actions. The Cancel' method cancels the action that is currently in process and all movements that were associated to it which ended the conversation.

The Update' method keeps monitoring the distance between the player and his target NPC, moving the player in the direction of the NPC when such distance is excessive. Only when the player approaches a certain distance will movement be terminated and dialogue initiated with the target NPC.

The NPC Dialogue Logic only has three functions. One to get the name of the conversant, one to change the cursor and one to check if the conversant is alive or not.

```
2 references
public class DialogueTrigger : MonoBehaviour
{
    1 reference
    [SerializeField] string action;
    1 reference
    [SerializeField] UnityEvent onTrigger;

    1 reference
    public void Trigger(string actionToTrigger)
    {
        if (actionToTrigger == action)
        {
            onTrigger.Invoke();
        }
    }
}
```

Figure 2-3 Dialogue Trigger Script

2.3 Dialogue Node

The DialogueNode class, defined in DialogueNode.cs, represents individual nodes in a dialogue tree. Each node contains several serialized fields, such as isPlayerSpeaking, text, children, rect, onEnterAction, onExitAction, and condition. These fields store information about whether the player is speaking, the text of the dialogue, the children's nodes, the position and size of the node in the editor, and actions to be triggered when entering or exiting the node. The `condition` field is of type `Condition`, which encapsulates logic to determine whether the node should be active based on certain criteria.

The CheckCondition method contained in DialogueNode.cs is used to check the current status of a condition defined in the node. It accepts an IEnumerable<IPredicateEvaluator>, which is a collection of objects that will be able to evaluate predicates. The method invokes the Check method of the condition object with the evaluators as passed parameters. It also returns a boolean value indicating whether the condition has been satisfied or not.

It also includes the FilterOnCondition method implemented in DialogueLogic.cs that narrows down a given set of DialogueNode objects in regard to the conditions of the nodes. An IEnumerable<DialogueNode> is provided as the input and every node taken through. For every node, the CheckCondition method is invoked using the result of GetEvaluators(), which returns a list of IPredicateEvaluator objects. If the condition of a node is satisfied, the node is yielded as part of the output. It operates in such a way that it eliminates nodes whose conditions have not been satisfied hence only valid nodes are left for further processing in the dialogue system.

2.4 Dialogue Trigger

The last script associated with the dialogue framework is the dialogue ttrigger, a script whose function is to invoke the triggers given by the dialogue scriptable object. In the game most of these triggers are either to give the player a que after a correct dialogue option or complete an objective after a dialogue.

```

namespace Thesis.Dialogue
{
    [CreateAssetMenu(fileName = "New Dialogue", menuName = "Dialogue", order = 0)]
    10 references
    public class Dialogue : ScriptableObject, ISerializationCallbackReceiver
    {
        [SerializeField]
        5 references
        List<DialogueNode> nodes = new List<DialogueNode>();

        1 reference
        [SerializeField] Vector2 newNodeOffset = new Vector2(250, 0);

        8 references
        Dictionary<string, DialogueNode> nodeLookup = new Dictionary<string, DialogueNode>();

        0 references
        private void Awake()
        {
            PopulateDialogue();
        }
        #if UNITY_EDITOR
            OnValidate();
        #endif
    }

    3 references
    private void OnValidate()
    {
        if (nodeLookup != null) nodeLookup.Clear();
        foreach (DialogueNode node in GetAllNodes())
        {
            if (node != null)
            {
                nodeLookup[node.name] = node;
            }
        }
    }
}

```

Figure 2-4 Part of Dialogue Scrip

3 Inventory System

Another basic component of an RPG game is an Inventory system for the player to store all the items he has gathered along the way and equip them so he can use them. In our game the Inventory is a readymade asset from GameDev TV, a team that creates paid tutorials and courses. The asset used in this game contains not only the Inventory but also an action bar and a script to create custom scriptable objects for weapons and wearable items.

For the needs of the game, we built on top of this asset implementing some major upgrades. The simplest one is the change on the UI components of this asset, we changed the skin, the dimensions and the whole look and feel. The big changes will be presented below.

Table 3-1

Script Name	Brief Explanation
Stats Equipment	An extension of the Equipment class contained in the asset
Stats Equipable Item	An extension of the Stats Equipable Item class contained in the asset
Random Dropper	Code to be used when dropping items down
Purse	A class containing the logic behind the money of the player
Collector	The script giving the player the actions to collect staff
Currency Item	The currency class for our world's money
Item Category	The categories of items inside the game

3.1 Stats Equipment and Equipable Items

The scriptable object script that the inventory asset contained for the creation of items did not support the functionality of items giving the player extra stats. So, this version just extends the code we got from the asset and adds this functionality with the use of `IModifierProvider`, an interface for letting items give the player or even non player characters stat bonuses. It supports both additive bonuses and percentage bonuses.

3.2 Random Dropper

This contains an implementation for dropping items at a random location within a specified distance in unity games. It takes all items to be dropped and scatters them around the player but within the specified radius.

3.3 Purse and Currency Item

Using an interface provided with the asset we created a Purse class sorting inside it all the logic for the money of the player. It also extends the `ISaveable` interface so the balance of the player can be saved and loaded.

3.4 Collector

The Collector class implements the `IAction` interface because it gives the player the action to pick up items from the ground.

The `Cancel` method is called to end any collection action that is in progress. It clears the `currentPickup` variable which corresponds to the object to be collected at the moment, and therefore there is no object being collected, and invokes the `Cancel` method of `Mover` component to halt any movements.

In turn, the `StartCollectAction` method accepts a `ClickablePickup` object and begins the process of collecting such an object. The `ClickablePickup` object is that which should be fetched from the game. The method calls the `StartAction` method, thanks to the `ActionScheduler` component, to carry out the collection action and changes the `currentPickup` to the passed item.

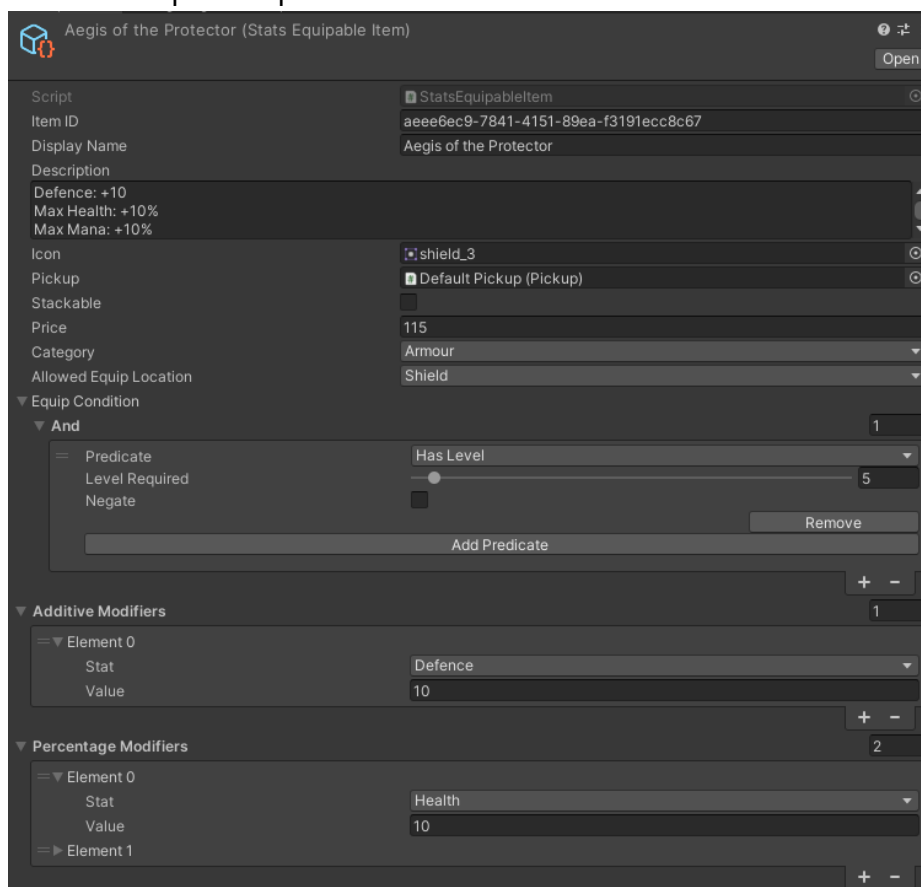


Figure 3-1 Item Scriptable Object

3.5 Item Creation

By using scriptable objects for both the weapons and the wearables the item creation is as simple as it gets. There are two steps for this creation. First you create the scriptable object through the unity menu and then you just populate all the serialized fields with the attributes you want the item to have. Each field has a self-explanatory name to aid the population of the items stats and information.

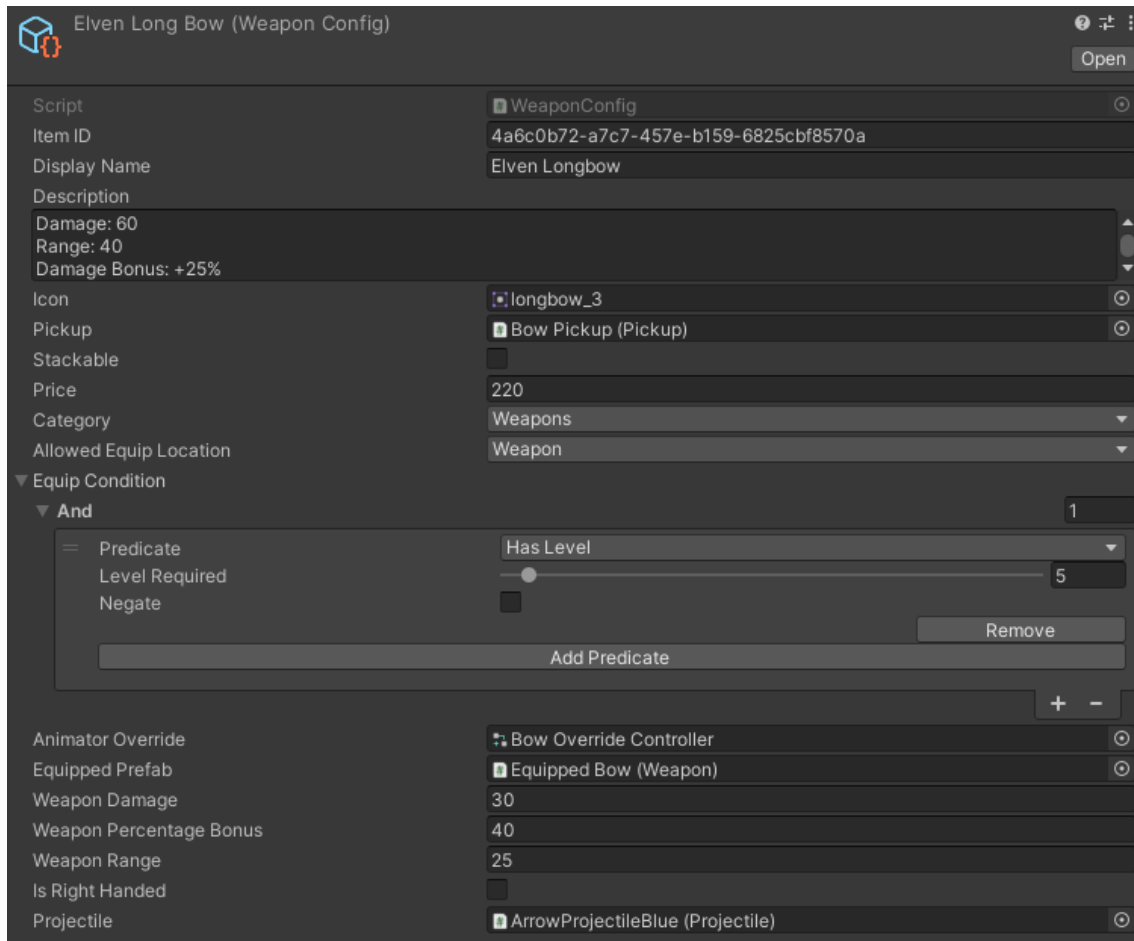


Figure 3-2 Weapon Scriptable Object

4 Shop System

To build upon the inventory framework, there is a shop framework so the player can buy items and sell or equip them. There are shops with customizable stock and prices where the player can buy and sell items.

Table 4-1

Script Name	Brief Explanation
Shop	An extension of the Equipment class contained in the asset
Shopper	An extension of the Stats Equipable Item class contained in the asset
Shop Item	Code to be used when dropping items down

```

1 reference
public void ConfirmTransaction()
{
    Inventory shopperInventory = currentShopper.GetComponent<Inventory>();
    Purse shopperPurse = currentShopper.GetComponent<Purse>();
    if (shopperInventory == null || shopperPurse == null) return;

    // Transfer to or from the inventory
    foreach (ShopItem shopItem in GetAllItems())
    {
        InventoryItem item = shopItem.GetInventoryItem();
        int quantity = shopItem.GetQuantityInTransaction();
        float price = shopItem.GetPrice();
        for (int i = 0; i < quantity; i++)
        {
            if (isBuyingMode)
            {
                BuyItem(shopperInventory, shopperPurse, item, price);
            }
            else
            {
                SellItem(shopperInventory, shopperPurse, item, price);
            }
        }
    }
    // Removal from transaction
    // Debting or Crediting of funds
    if (onChange != null)
    {
        onChange();
    }
}

2 references
public float TransactionTotal()
{
    float total = 0;
    foreach (ShopItem item in GetAllItems())
    {
        total += item.GetPrice() * item.GetQuantityInTransaction();
    }
    return total;
}

```

Figure 4-1 Shop Code Part One

4.1 Shop

The code contains the logic around Shops implements `IRaycastable` as well as `ISaveable` interfaces. In this perspective, the shop acts as a center of in-game activities to procure and sell items. Several serialized fields of this shop include, but are not limited to, `shopName`, `sellingPercentage`, `maximumBarterDiscount`, and collections of `StockItemConfig` objects. The `StockItemConfig` class is a nested serializable class, containing data for all of the stock items in the shop such as the item, the quantity in stock, a percentage to discount on buying, and a level to unlock the item.

The `Shop` class has instance variables in the form of two dictionaries: One that maps to transactions and one that maps to sold items including the current shopper and the shop mode e.g. buying mode or selling mode. There is also an event indicated by `onChange` which is fired any time the state of the shop changes. The `SetShopper` method can be used to assign the current shopper while the `SelectFilter` and `SelectMode` methods enable the shop to search for a particular item in a certain category or switch modes respectively.

The `GetFilteredItems` and `GetAllItems` methods provide enumerations of items of type `ShopItem`, representing the items which are currently present available in the shop with respect to the chosen filter and mode. The `AddToTransaction` method appends items into the active transaction's list, while also making sure that number of such items does not surpass their availability. The `ConfirmTransaction` method concludes the transaction, providing the necessary items to the shopper and modifying the amount of inventory and purse they have.

The `Shop` class also has some auxiliary functions of `GetCursorType`, `HandleRaycast` allowing to use the shop through raycasting. The methods `HasSufficientFunds`, `IsTransactionEmpty` and `HasInventorySpace` assess the aforementioned conditions and determine if the transaction is possible or not. The `GetShopName` method know what the name of the shop is.

Internal methods of this type include `CountItemsInInventory`, `SellItem`, `BuyItem`, `FindFirstItemSlot`, `GetShopperLevel`, `GetAvailabilities`, and `GetPrices` — these methods are the part of a shop's internal logic – counting items, buying, selling, finding item slots, pricing items. A `GetBarterDiscount` method would invoke discount policies taking into account shopper's stats.

The `CaptureState`, `RestoreState` are ones that are responsible for saving and loading the contents of the saveable object, which is an interface implemented by the shop, which is its likeness and an instance of the `ISaveable` interface. These methods serialize and deserialize the `stockSold` dictionary which entities the total items sold.

In general, the `Shop` class acts as an efficient module in the game for managing the shop, transactions of items, filtering items, keeping the shop states and communicating with other game modules through interfaces and events.

```

10 references
public class Shop : MonoBehaviour, IRaycastable, ISaveable
{
    1 reference
    [SerializeField] string shopName;
    [Range(0, 100)]
    1 reference
    [SerializeField] float sellingPercentage = 80f;
    1 reference
    [SerializeField] float maximumBarterDiscount = 80;

    // Stock Config
    // Item:
    // InventoryItem
    // Initial Stock
    // buyingDiscount
    [SerializeField]
    1 reference
    StockItemConfig[] stockConfig;

    [System.Serializable]
    5 references
    class StockItemConfig
    {
        15 references
        public InventoryItem item;
        2 references
        public int initialStock;
        [Range(0, 100)]
        3 references
        public float buyingDiscountPercentage;
        1 reference
        public int levelToUnlock = 0;

        2 references
        public StockItemConfig(InventoryItem item)
        {
            this.item = item;
            this.buyingDiscountPercentage = 0;
            this.initialStock = 0;
        }
    }
}

```

Figure 4-2 Shop Code Part Two

4.2 Shop Item

Shop Item is considered an item that can be bought from a shop present in a game. This class contains different properties and behavior of the shop item such as availability, cost of the item, and quantity that is involved in any transaction.

This ShopItem class contains four private fields: item, availability, price and quantityInTransaction. The item field is of type InventoryItem, which probably is the item in question within that inventory system that is in the game. The availability field is of type integer which is used to show how many of the items are in stock available for purchase. The price field is a float which captures how much the item will cost. And then quantityInTransaction field is integer number that is used to indicate how many heads/units of that item are in the current transaction and are expected to be sold.

The class constructor sets these fields to the values provided in the parameters. Which means that once a ShopItem object is created, it will have information on what the item is, its availability, price and quantity of the transacted item.

The ShopItem class contains a number of public methods that can be used in order to get access to the private fields. Within the GetIcon method, item's icon is retrieved from the InventoryItem object by calling GetIcon method. Availability of the item is returned by GetAvailability method. Upon calling the GetDisplayName method of the InventoryItem object, the GetName method gives out the display name of the item. Price of the item is also returned by the GetPrice method. The GetInventoryItem method gives out the InventoryItem object itself, and the GetQuantityInTransaction method gives out the quantity of the particular item that is in the transaction.

4.3 Shopper

Shopper is the component that enables the player to use the shops providing him with the shop action through the IAction interface.

It controls the players movement to reach the shop stop and then open the UI and activate the shop.

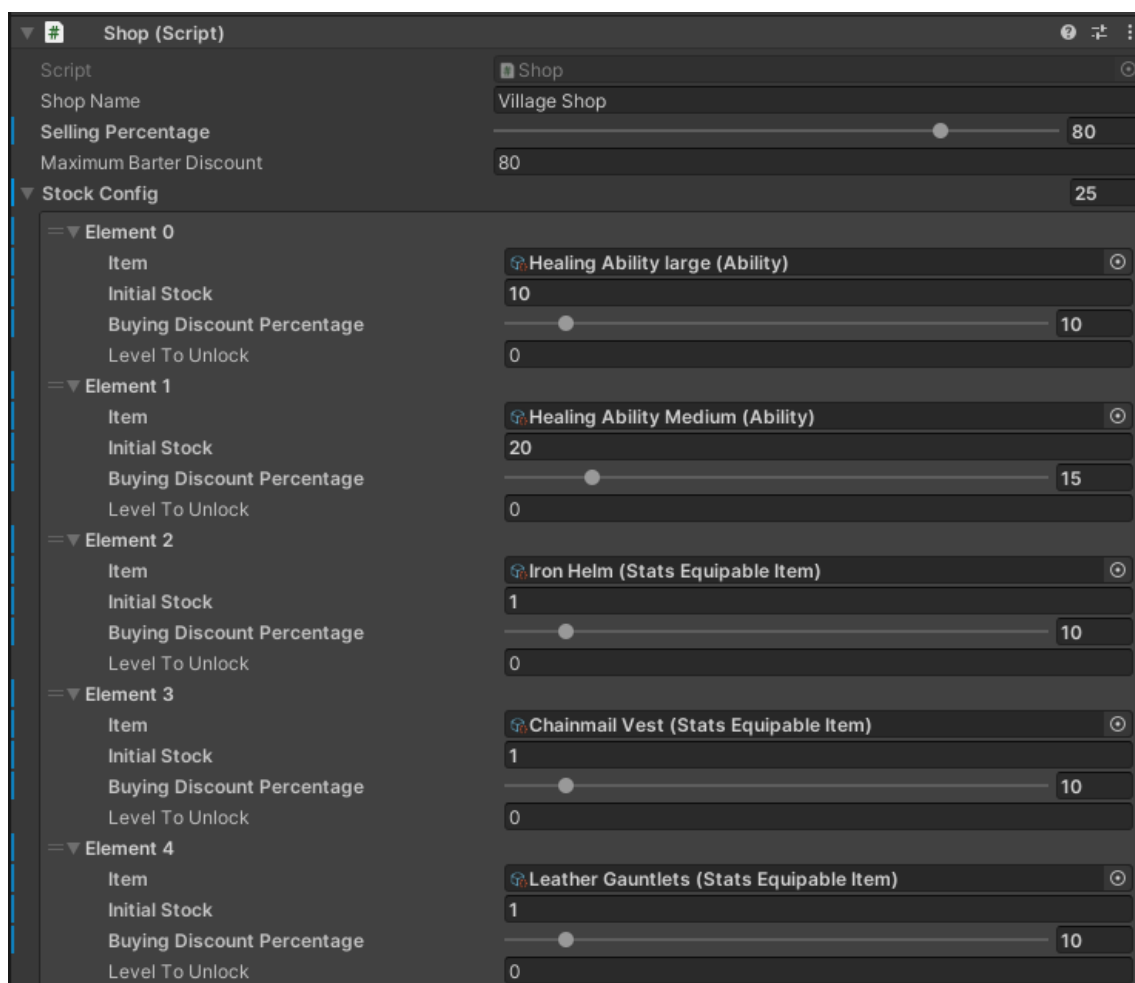


Figure 4-3 Shop Component

5 Quest System

In this game the player can accept and complete a variety of quests all with different objectives and rewards. With this custom framework you can create quest scriptable objects with custom objectives with their on requirements for the player to complete and feel like he is progressing he story of the game.

Table 5-1

Script Name	Brief Explanation
Quest	The core script of the quest module containing the logic an the implementation of the quest scriptable object
Quest Completion	Code to complete the quest objectives
Quest Giver	Code tot give active quests to the player
Quest Status	A script that controls he status of the quest (if it isa active etc.)
Player Quest List	A script that is keeping rack of every quest the player has
Death Counter	A script to be added to enemies o keep track of killing them
Achievement Counter	Code for the objectives that require a certain number of conditions to be met (for example kill 10 goblins)

```
[CreateAssetMenu(fileName = "Quest", menuName = "Thesis/Quest", order = 0)]
31 references
public class Quest : ScriptableObject
{
    3 references
    [SerializeField] List<Objective> objectives = new List<Objective>();
    1 reference
    [SerializeField] List<Reward> rewards = new List<Reward>();

    [System.Serializable]
    3 references
    public class Reward
    {
        [Min(1)]
        6 references
        public int number;
        4 references
        public InventoryItem item;
    }

    [System.Serializable]
    5 references
    public class Objective
    {
        8 references
        public string reference;
        5 references
        public string description;
        1 reference
        public bool usesCondition = false;
        1 reference
        public Condition completionCondition;
        1 reference
        public bool hasCounter;
        1 reference
        public string token;
        1 reference
        public int required;
        2 references
        public string GetDescription()
        {
            if (!hasCounter) return description;
            AchievementCounter counter = GameObject.FindWithTag("Player").GetComponent<AchievementCounter>();
            if (counter)
            {
                return $"{description} ({counter.GetCounterValue(token)}/{required}";
            }
            else return description;
        }
    }
}
```

Figure 5-1 Quest Class

5.1 Quest

The quest class creates a serialized object that contains all data that a proper quest could need.

The Quest class contains two serialized lists which are objectives and rewards. These lists are serializable for proper editing inside the unity editor.

The rewards list can be populated with rewards for the player to get after they complete the quest. Those can range from experience for the player to level up , to really powerful weapons and armor.

The Objective list contains a number of fields denoting the properties of an objective in a quest. These fields include reference, description, usesCondition, completionCondition, hasCounter, token, and required field. The GetDescription member function in the Objective returns description including details for the particular objective. In a situation where the objective is aimed at utilizing a counter, the current counter is accessed from the player's AchievementCount component and edifies the objective description.

The Quest class contains a number of methods to perform actions with its data. The GetTitle method is the one used to get the title of the quest while the GetObjectiveCount method gives an overall figure of the number of objectives contained in the quest. The GetObjectives method is used to get a corresponding enumerable collection of objectives and the HasObjective method verifies the existence of an objective reference within the quest.

5.2 Quest Completion

Is a small script to be added to objects in game so an event can trigger it and complete an objective from a particular quest selected from the unity inspector.

5.3 Quest Giver

Same as Quest Completion with the only difference being that instead of completing objectives it gives the selected quest to the player.

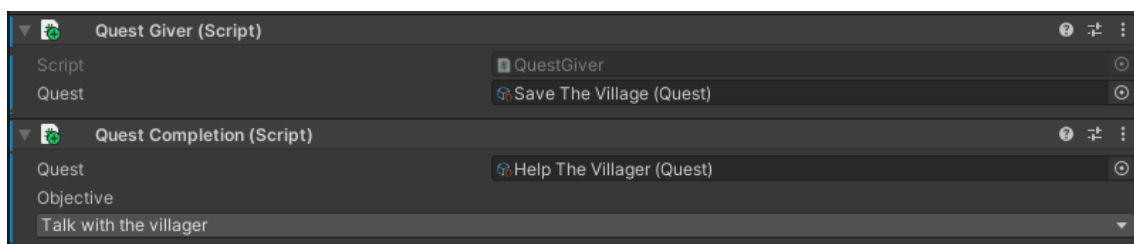


Figure 5-2 Quest Components Example

5.4 Quest Status

The quest status class is responsible for quest tracking in the game. It tracks the objectives of the quests and which of them are completed or not.

QuestStatus class has two constructors. The first constructor assigns the quest field to a corresponding Quest object passed as a parameter. The next one is for the objectType parameter, which will be Cast to QuestStatusRecord. This constructor uses a quest by name Quest.GetByName to get a quest and a questStatusRecord which it uses to get the completed objectives and uses to initialize the completedObjectives List.

The GetQuest method gets the corresponding QuestStatus's Quest object. The GetCompletedCount method returns how many of the objectives are marked fulfilled. The IsObjectiveComplete method effectively states that one objective is complete when it is stated that a completed objective is present in a completedObjectives list.

Through the CompleteObjective method an objective is marked complete and is placed in the completedObjectives list but only if the objective is contained within the quest. The IsComplete method circles if all the quest's objectives have been met for every one of the space objectives. It goes through all the objectives of the quest and if there is an incomplete objective it returns false. It will display 'done' in the console and return true if all the objectives have been completed.

The CaptureState method is responsible for instantiation of QuestStatusRecord object setting records' questName and completedObjectives and returning it. This particular method helps impose a situation whereby the QuestStatus object is preserved making it possible to come up with it after some time.

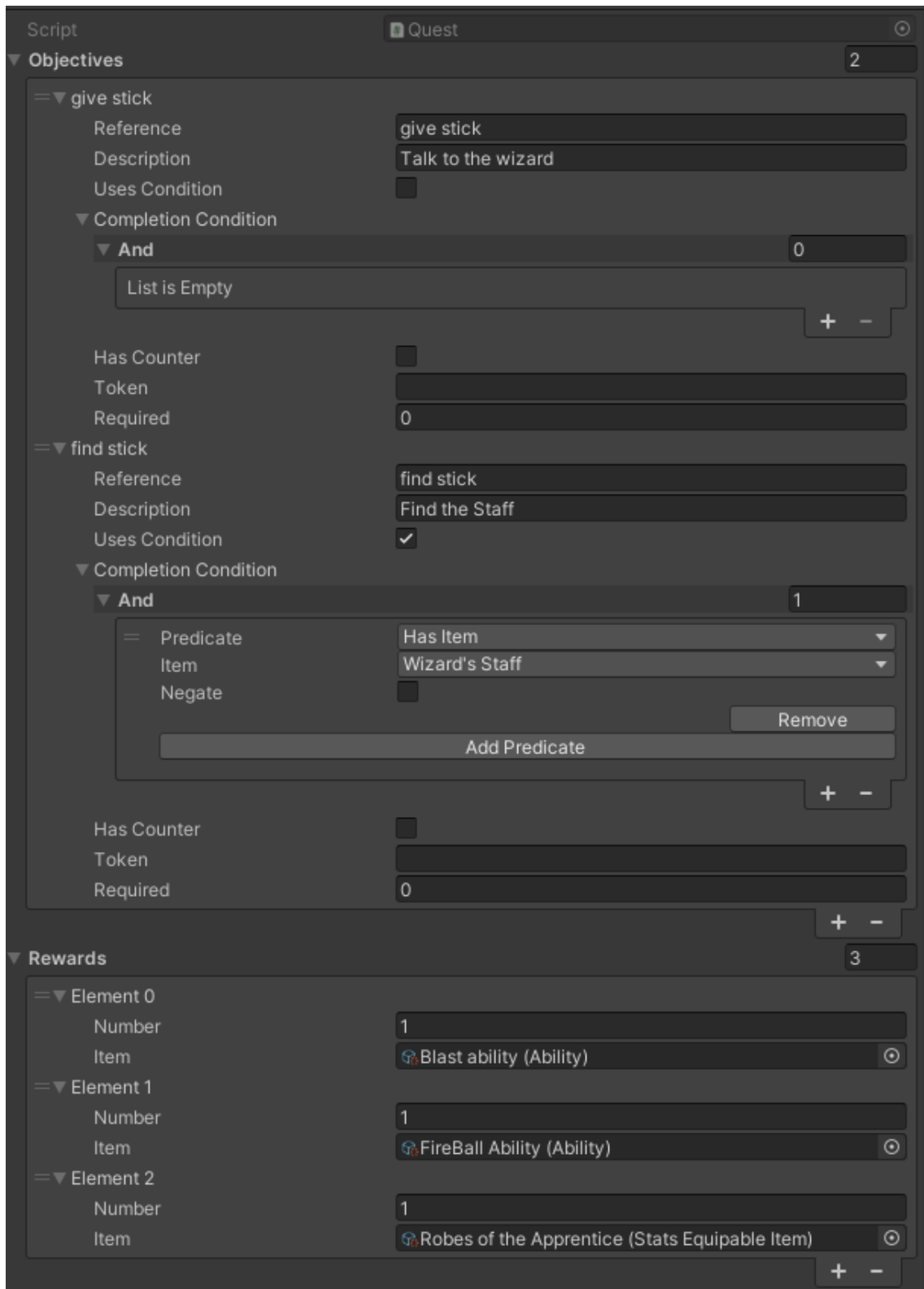


Figure 5-3 Quest Object Example

5.5 Quest List

The quest list class manages the different quests that a player has in a game. The class also includes the interfaces `ISaveable`, which allows saving and restoring the state of the object, and `IPredicateEvaluator`, which allows evaluating the conditions. The `PlayerQuestList` class has a collection of `QuestStatus` that are the statuses of the quests of a particular player. There is also an event `onUpdate`, that gets fired when the quest list is changed.

The `CompleteObjectivesByPredicates` method explains the process of going through all the `QuestStatus` in the list and determining whether that particular quest has been completed or not. If it is still active, it gets the quest, and the goals associated with it and checks them one by one to determine whether they have been attained and more importantly, whether they employ a criterion. If the objective uses a criterion, then the method `CompletionCondition` uses its `Check` method to find out if the objective has been met or not. In the event that the requirement is satisfied, instead of regular marking, a more advanced `CompleteObjective` method is invoked.

The `AddQuest` method adds a new quest to the list whenever it is not already included. Added additional on the added quest which triggers the `onUpdate` event which adds an object `QuestStatus` that is new to the Quest. The '`CompleteObjective`' method marks a certain objective of a quest as accomplished and determines whether the whole quest is accomplished. When such a quest is completed successfully, it uses the `GiveReward` method to reward the player. The '`HasQuest`' method determines whether this type of quest has already been placed in the persistent storage.

The '`GetQuestStatus`' method returns information about the status of certain quests captured in the quest status list. Under the `GiveReward` method, the player is rewarded after completing a quest by giving them certain items that have been stored in the players inventory or by dropping the items if several items cannot be stored in inventory. The `CaptureState` and '`RestoreState`' methods are responsible for the quest list state saving and quest list state restoring, respectively. The '`Evaluate`' method assesses several quest related predicates such as whether the player possesses any given quest, whether this quest was already finished or whether a given target was achieved.

The `QuestStatus` class encapsulates the status of the quest, the quest itself and also a collection of the quest objectives fulfilled. Its methods include retrieving the quest, checking if a certain objective has been completed, completing an objective, checking if the whole quest is completed, and saving the quest status state. The nested `QuestStatusRecord` class is intended for the serialization and deserialization of quest status.

All in all, the '`PlayerQuestList`' class resolves this issue and provides in-game quest management features including, but not limited to, providing progress monitoring, condition checks and rewards issuance, as well as loading and saving the state of the quest list. The '`quest status`' class refers to the information of each quest being referred to making the quest easier to control and update which is made possible with the changing progression of the player within the game.

5.6 Death Counter

Death counter is the script for the component to be added on enemies for kill tracking. It is used on quests that have objectives requiring the player to kill multiple enemies, for example killing ten goblins. This Component just keeps track of how many kills he player has of the indicated enemy type.

5.7 Achievement Counter

This class is used to implement and keep track of the different achievement counters within the game. There is a private dictionary, counts, where different achievement tokens and associated count values are held; the key is a token in string form and the value is the count in integer form. It implements the ISaveable and IPredicateEvaluator interfaces the first one to be able to save and restore the tokens and the second one to be able to use custom predicates for the tokens.

The class also declares an onCountChanged event which gets raised whenever the count is changed. This enables the rest of the game to react towards the changes which take place in the achievement counts.

The AddToCount method adds an amount to the value associated with the specified token. If the token is not found in the dictionary where it is supposed to be, if the onlyIfExists parameter is true, the method will just return 0 without making any changes. Line 76,87 If this is the first record, it makes an entry of this token along with the specified amount and fire the onCountChanged event. In case if the token already exists, it increases the current count by the given amount and fires the event.

The RegisterCounter method allows to add a new counter under the specified token. In case of such a token not being found in the dictionary or the overwrite parameter is true, the method initializes the token's count to 0 and raises an onCountChanged event. Next, it will provide the current count for the specified token

The GetCounterValue method gives the value how many tokens of the given type one has. If the token does not exist in the dictionary, then it returns zero.

The CaptureState method is part of ISaveable interface and is responsible for getting the current values of achievement counters. Returns the counts dictionary, which can be stored. The RestoreState also part of ISaveable and is used for re-allocating the state of achievement counters that was saved previously. It obtains the saved state in the form of an object, converts the object to a dictionary and assigns this object to the counts.

6 User Interface

6.1 Main Menu

The main menu is the first scene the player sees when opening the game. From here he can either start a new game or continue some old save of his.



Figure 6-1 Main Menu UI

6.1.1 Game Related Windows

In the main menu there are three buttons, the Continue, the New Game and the Load Game. All these three are ways to start the game. The first one creates a new save file the second one uses the latest save file and the third one lets the player choose the save file he wants.



Figure 6-2 New Game UI

6.1.2 Settings Window

Here the user can change some graphical settings by choosing between four quality presets and lower the music of the game.

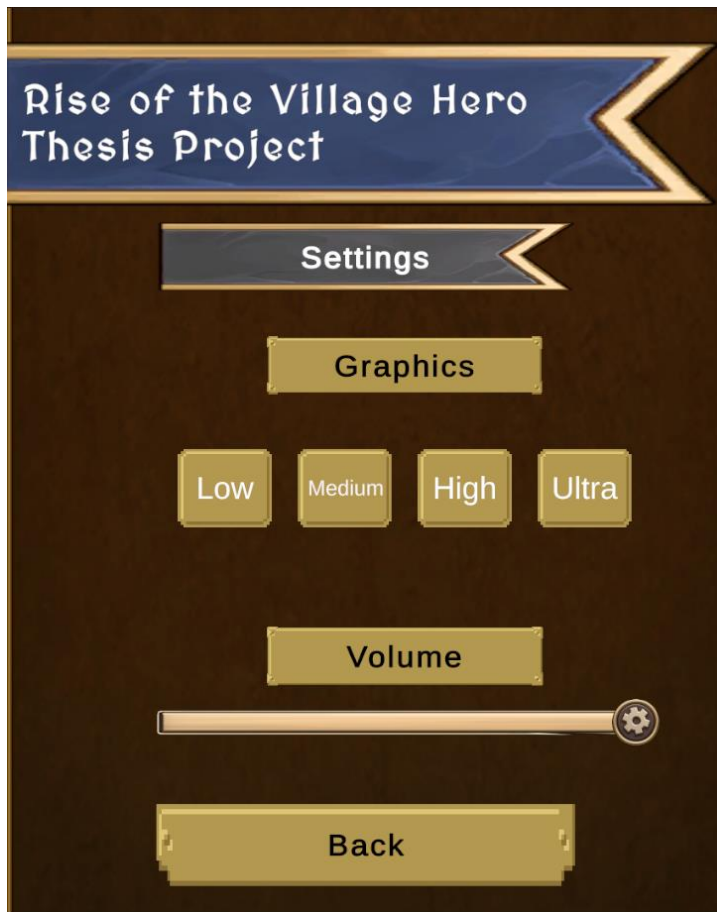


Figure 6-3 Settings UI

6.1.3 Information Window

On this panel the user can view the controls for the game and read a brief description

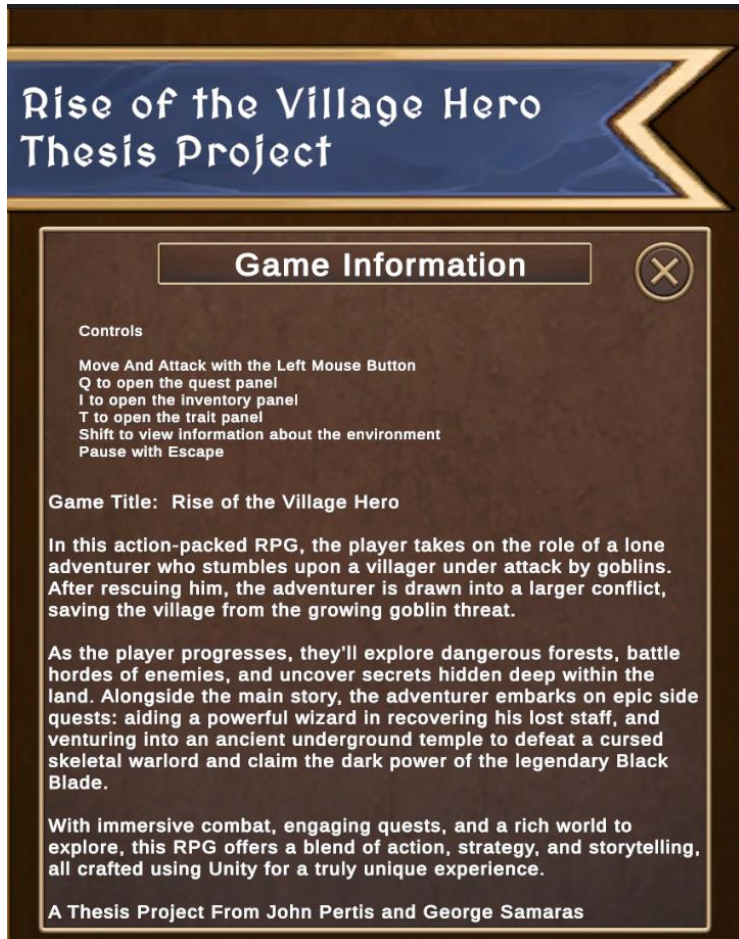


Figure 6-4 Information UI

6.2 In Game Menu

This menu is the main game menu containing almost all user interface elements of the game. It is the same on all scenes.

6.2.1 Inventory

This menu opens by pressing the “I” button and consists of two menus. The inventory tab and the equipment tab. The first one stores all the items and treasures the player has accumulated while playing the game. The other one has item slots that the player can populate with items to be equipped to the character giving him bonuses, like higher attack or more health.

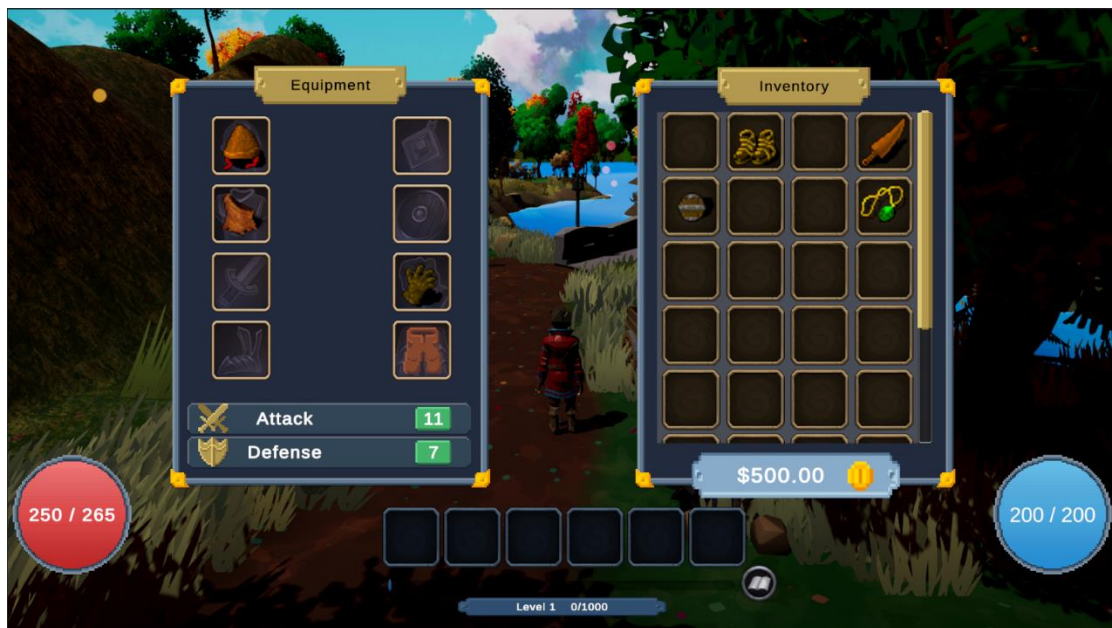


Figure 6-5 Inventory UI

6.2.2 Dialogue

This menu shows up only when the player has started a dialogue with someone and it is simply a text field showing the conversant name and the text of the dialogue. There are two buttons, a next button and an exit button. The first one skips to the next dialogue and the latter one exits the dialogue.

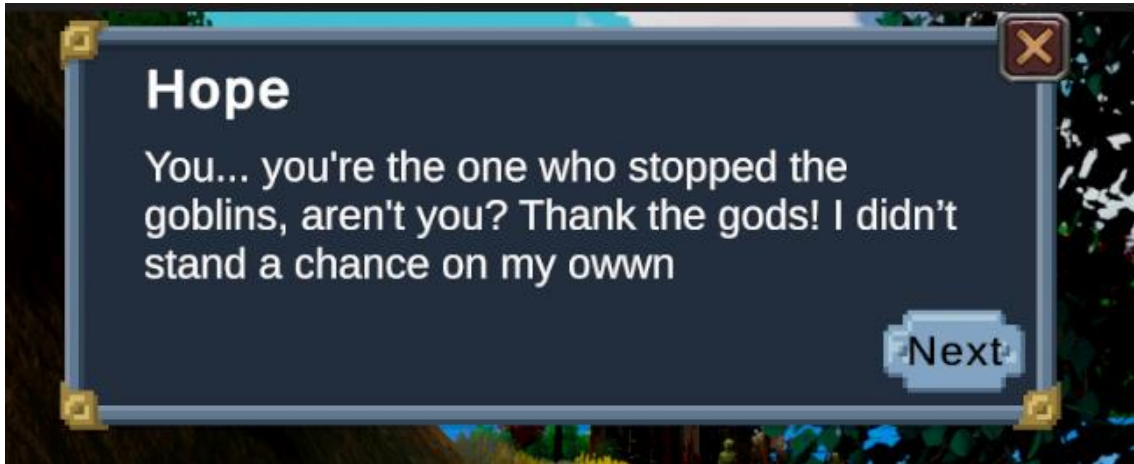


Figure 6-6 Dialogue UI

6.2.3 Quest

This menu contains all the data related to quests. It toggles on and off by pressing the "Q" button and by doing so the player can see all quests he has collected. Every quest record contains the title, the description, the objectives and if they are completed or not and lastly the rewards. That way the player can see anytime what his progress is.

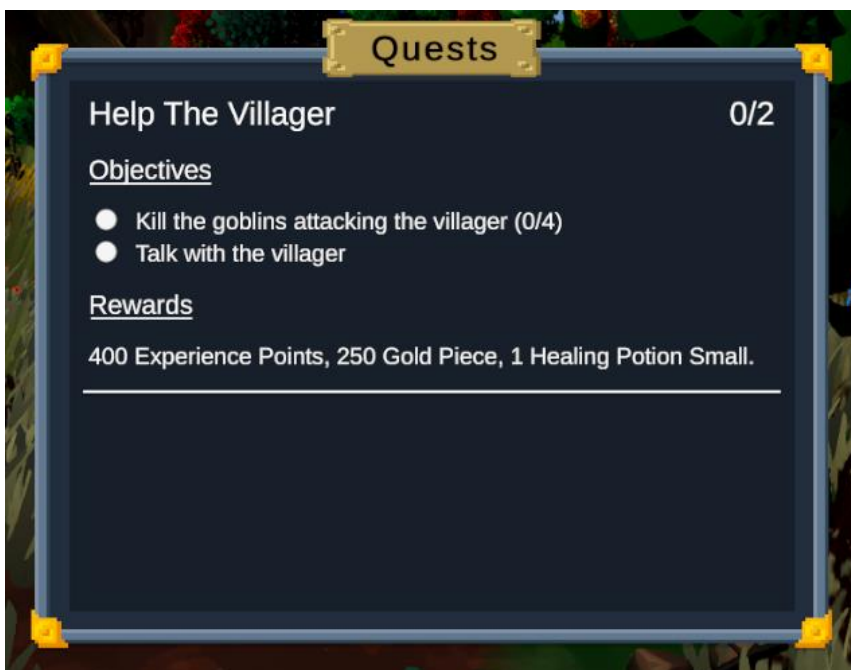


Figure 6-7 Quest UI

6.2.4 Shop

By talking to a shopkeeper, the Shop menu opens up. Here the player can either shop the items the seller has in shop but can also sell any items he has in his inventory. There are buttons to filter the items through some premade categories, such as weapons, armors and potions. By pressing the sell button, the menu changes and instead of displaying the items of the seller it now displays the players items so he can sell them.



Figure 6-8 Shop UI

6.2.5 Stats

This menu contains the user interface for the traits the player has. By pressing the “T” button a menu opens displaying all the traits the player has and the points he has put on each. By pressing the “plus” button can upgrade his skills with the trait points he accumulates throughout leveling up.

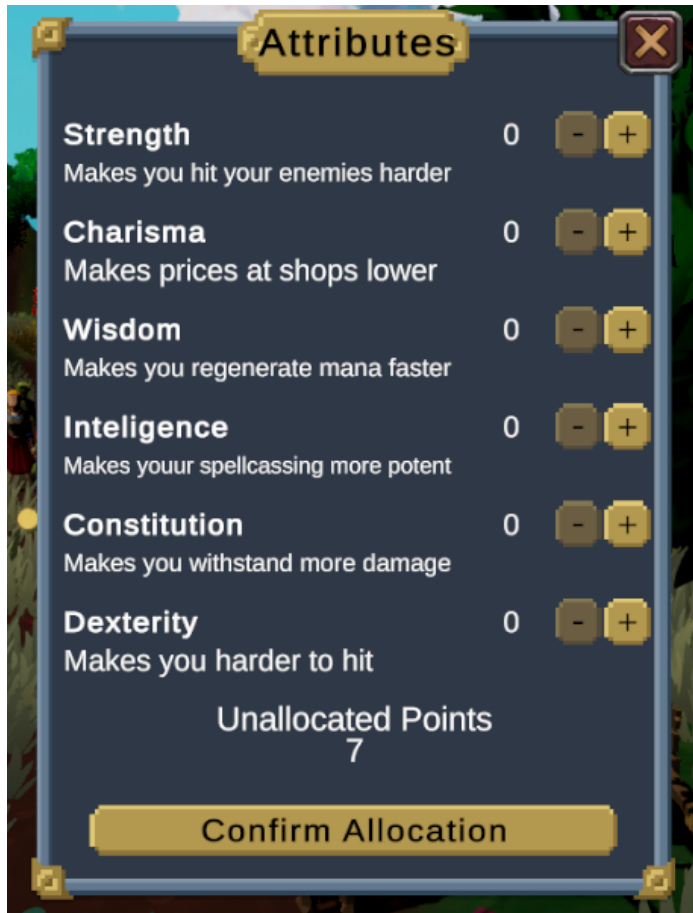


Figure 6-9 Stats UI

6.3 Heads Up Display

The Heads Up Display is always active when the game running and provides the player with all the important information about the game.

6.3.1 Mana, Life and Level Display

The main elements of the Heads Up Display are the mana and health bubbles and the level bar. The first two are bubbles that depending on how filled they are the player knows how much life or mana he has at any giving point.



Figure 6-10 Heads Up Display

The level bar has two components, the level bar that fills when you get experience and the level label that shows the current level of the player.

6.3.2 Pause Menu

This menu plays a significant role in the flow of the game and the player experience. The player can pause the game by pressing the “Escape” button displaying this way the pause menu. While this is open the player loses all control of the character and the game freezes in place. The menu consists of three buttons, a resume button that resumes the game, a save button that saves the game and a save and exit button that exits the game and loads up the Main Menu scene.

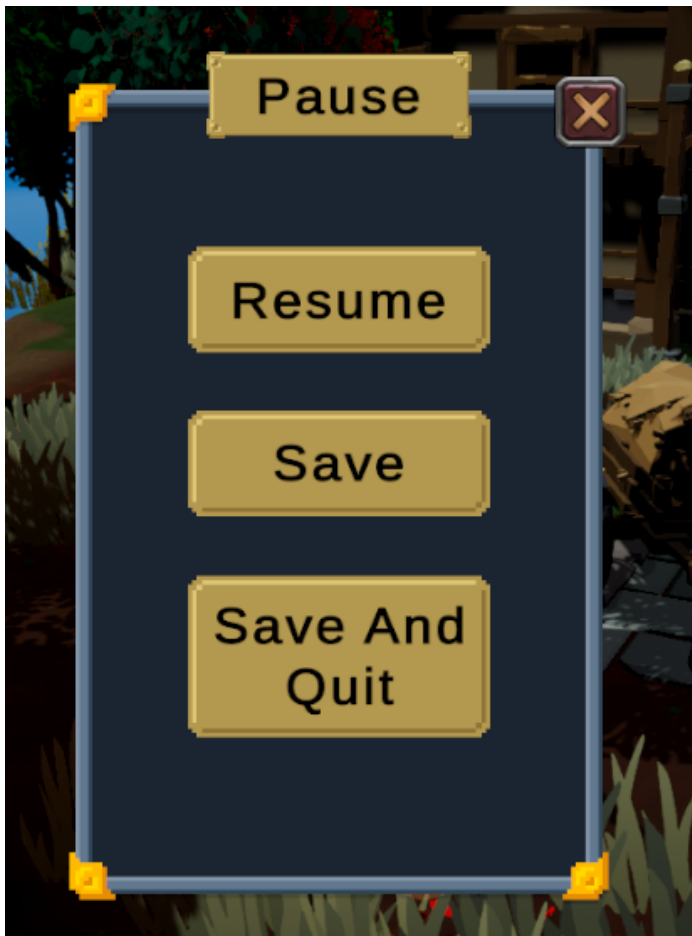


Figure 6-11 Pause UI

7 Combat

In the game the player must fight and defeat a goblin horde. There are numerous kinds of goblins and weapons but only one way to defeat them, run towards them and start attacking. In this section the whole concept of attacking enemies will be covered.

The script responsible for this are the following:

Table 7-1

Script Name	Brief Explanation
Fighter	The core script of the combat module responsible for handling the attacking process
Combat Target	Designate that the character can be attacked
Aggro Group	Makes a group of enemies aggressive towards the player
Projectile	Script added to a weapon so it can support shooting projectiles like arrows
Weapon Config	A scriptable object for the creation of weapons
Weapon Pickup	Script that implements the ability to pickup weapons from the floor
Wave System	A script that contains the logic for the final battle

```

11 references
public class Fighter : MonoBehaviour, IAction
{
    1 reference
    [SerializeField] float timeBetweenAttacks = 1f;
    3 references
    [SerializeField] Transform rightHandTransform = null;
    3 references
    [SerializeField] Transform leftHandTransform = null;
    3 references
    [SerializeField] WeaponConfig defaultWeapon = null;
    1 reference
    [SerializeField] float autoAttackRange = 4f;
    14 references
    Health target;
    4 references
    Equipment equipment;
    3 references
    float timeSinceLastAttack = Mathf.Infinity;
    5 references
    WeaponConfig currentWeaponConfig;
    5 references
    LazyValue<Weapon> currentWeapon;

    0 references
    private void Awake()
    {
        currentWeaponConfig = defaultWeapon;
        currentWeapon = new LazyValue<Weapon>(SetupDefaultWeapon);
        equipment = GetComponent<Equipment>();
        if (equipment)
        {
            equipment.equipmentUpdated += UpdateWeapon;
        }
    }
}

```

Figure 7-1 Fighter Script

7.1 Fighter

Firstly, like every one of the characters actions, the fighter class which controls the attacking action, implements the IAction Interface. . This class is responsible for handling combat mechanics, including attacking, equipping weapons, and finding new targets within range. Several serialized fields are defined, such as `timeBetweenAttacks`, `rightHandTransform`, `leftHandTransform`, `defaultWeapon`, and `autoAttackRange`. These fields can be adjusted in the Unity Editor to control various aspects of the fighter's behavior, such as the time between attacks, the transforms for weapon attachment, the default weapon configuration, and the range for auto-attacking.

Unity's `Awake` method initializes the weapon of the character with the default weapon so every character always has a weapon equipped. The equipment component is also retrieved, and if it exists, an event handler is added to update the weapon whenever the equipment is updated . The `Update` method which is called once per frame handles the logic for attacking a target. Also checks if the current target is null or dead and if the target is dead, it attempts to find a new target within range. If a target is found and is within range, the fighter stops moving and initiates the attack behavior. Otherwise, it moves towards the target.

The `EquipWeapon` method allows the fighter to equip a new weapon, updating the weapon model and attack behavior based on the new weapon's configuration. This method is important for switching between different weapons during gameplay. Additionally, the class ensures that the fighter cannot attack too quickly by enforcing a delay between attacks using the `timeBetweenAttacks` field.

The `Attack` and `Cancel` methods manage the fighter's attack sequence. `Attack` initiates the process, while `Cancel` stops the current action if the fighter disengages or switches targets.

These were the core methods of the script containing the basic logic of attacking. There are more methods being called by them such as the `FindNewTargetInRange` and the `FindAllTargetsInRange` which are used to detect enemies near the player that the player can automatically attack when his first target is dead.

Overall, this class is crucial for managing combat interactions in the game, providing the fighter with the ability to switch weapons, target enemies, and perform attacks while handling the cooldown between actions.

7.2 Aggro Group

The AggroGroup class is a Unity MonoBehaviour that manages a group of Fighter objects, allowing them to be activated or deactivated as a group. This class is particularly useful in scenarios where you want to control the combat readiness of multiple enemies simultaneously, such as when entering or exiting a combat zone.

The class has two serialized fields taking an array of enemies and a Boolean that checks if the group should be initialized as aggro from the start or not. When a group is not aggravated they do not attack the player and the player cannot attack them.

This class is mainly used with other events calling it. Most often it is being called from dialogue events to activate an enemy group so the character can fight them.

7.3 Combat Target

The CombatTarget class is a Unity MonoBehaviour that implements the IRaycastable interface. This class is designed to be attached to game objects that can be targeted in combat. Everything with this component is considered an enemy and thus can be attacked.

Using the IRaycastable interface whenever the cursor is above an object with this component the cursor changes to indicate that you can attack it. Clicking on it begins the attack sequence with the help of the fighter class.

The GetCursorType method returns a Cursor Type. Combat value, indicating that the cursor should change to a combat-specific icon when hovering over this target. This helps provide visual feedback to the player, indicating that the object can be attacked.

Overall, the CombatTarget class integrates with the player's combat system, allowing game objects to be targeted and attacked, while providing visual cues and handling input interactions.

7.4 Projectile

The Projectile class is a Unity MonoBehaviour script designed to handle the behavior of projectile objects in a game. This class manages the projectile's movement, targeting, and interactions upon collision. It includes several serialized fields that can be configured in the Unity Editor, such as speed, homing capability, hit effects, maximum lifetime, objects to destroy on hit, and the time the projectile remains after impact. The projectiles are mainly used as an ammunition for the bows in the game.

The class defines private fields to store the target (Health component), target point (Vector3), damage value (float), and the instigator (GameObject). The Start method is called when the projectile is initialized, and it orients the projectile to face its target or target point using the transformLookAt method.

The SetTarget methods set the target of the projectile. Telling the object where it should move. Then with each call of the update method the projectile is moved little by little towards the target.

The GetAimLocation method calculates the aim location for the projectile. If a target is specified, it returns the position of the target, adjusted to the center of its CapsuleCollider if one is present. If no target is specified, it returns the target point.

The OnTriggerEnter method handles collisions with other objects. When the projectile collides with another collider, it checks if the collider has a Health component and whether it is the intended target. If the conditions are met, the projectile deals damage to the target using the TakeDamage method of the Health component. Then the projectile is destroyed.

Overall, the Projectile class provides a comprehensive system for managing projectile behavior, including movement, targeting, collision handling, and visual effects, making it a versatile component for various projectile-based mechanics in a game.

7.5 Weapon Config

The Weapon Config is the scriptable object representing the weapons in the game. It extends the Equipable Item a scriptable object provided with the asset used for the inventory. It provides numerous different weapon customizations to be customized through the inspector of the unity editor.

The class contains several serialized fields that define the properties of the weapon, such as animatorOverride, equippedPrefab, weaponDamage, weaponPercentageBonus, weaponRange, isRightHanded, and projectile. These fields can be configured in the Unity Editor to customize the weapon's behavior and appearance.

The Spawn method is responsible for instantiating the weapon and attaching it to the specified hand of the character. It first calls DestroyOldWeapon to remove any previously equipped weapon. The method also handles the animator override, ensuring that the correct animations are used for the equipped weapon.

It also implements the IModifierProvider Interface which handles adding the stat increases that each weapon provides to the base stats of the character.

```

namespace Thesis.Combat
{
    [CreateAssetMenu(fileName = "Weapon", menuName = "Weapons/Make New Weapon", order = 0)]
    6 references
    public class WeaponConfig : EquipableItem, IModifierProvider
    {
        2 references
        [SerializeField] AnimatorOverrideController animatorOverride = null;
        2 references
        [SerializeField] Weapon equippedPrefab = null;
        2 references
        [SerializeField] float weaponDamage = 5f;
        2 references
        [SerializeField] float weaponPercentageBonus = 0;
        1 reference
        [SerializeField] float weaponRange = 2f;
        1 reference
        [SerializeField] bool isRightHanded = true;
        2 references
        [SerializeField] Projectile projectile = null;

        3 references
        const string weaponName = "Weapon";

        1 reference
        public Weapon Spawn(Transform rightHand, Transform leftHand, Animator animator)
        {
            DestroyOldWeapon(rightHand, leftHand);

            Weapon weapon = null;
            if (equippedPrefab != null)
            {
                Transform handTransform = GetTransform(rightHand, leftHand);

                weapon = Instantiate(equippedPrefab, handTransform);
                weapon.gameObject.name = weaponName;
            }

            var overrideController = animator.runtimeAnimatorController as AnimatorOverrideController;
            if (animatorOverride != null)
            {
                animator.runtimeAnimatorController = animatorOverride;
            }
            else if (overrideController != null)
            {
                animator.runtimeAnimatorController = overrideController.runtimeAnimatorController;
            }

            return weapon;
        }
    }
}

```

Figure 7-2 Weapon Config

7.6 Weapon Pickup

The Weapon Pickup class is used to allow items to be picked up by the player. It implements the `IRaycastable` interface to be able to make these items detectable by the user. When the user hovers the cursor above such item the cursor changes to signal the ability to pickup the item.

It includes serialized fields for configuring the weapon to be picked up (`WeaponConfig`), the respawn time after the weapon is picked up (float), and the amount of health to restore to the player upon pickup (float). These fields can be set in the Unity Editor.

The `OnTriggerEnter` method is called when another collider enters the trigger collider attached to the `WeaponPickup` object. If the collider belongs to a game object tagged as "Player", the `Pickup` method is called giving the item to the player.

7.7 Wave System

The wave system code contains the logic for the waves of enemies spanning on the final battle. It provides serializable fields for the kinds of enemies to spawn, the number of enemies and the pickups to spawn.

It also contains the cutscenes to be played in between each wave. The `Start` method calculates the total number of enemies across all waves by summing `wave1Length`, `wave2Length`, and `wave3Length`. It also sets boolean flags to track the progress of the waves.

The `Update` method is responsible for checking the state of the game and triggering events based on the number of enemies left. If the total number of enemies left equals the sum of `wave2Length` and `wave3Length` and the second wave hasn't started yet, it spawns the first set of pickups and plays a cutscene. Similarly, if the total number of enemies left equals `wave3Length` and the third wave hasn't started yet, it spawns the second set of pickups and plays another cutscene. When all enemies are defeated, it plays the ending cutscene.

The `FirstWave`, `SecondWave`, and `FinalWave` methods are called by cutscenes to spawn enemies for each respective wave. The `SpawnItems` method activates all `GameObjects` in the provided array, making them appear in the game. The `EnemyKilled` method decreases the total number of enemies left and prints the updated count, which helps in tracking the game's progress.

```
public class WaveSystem : MonoBehaviour
{
    1 reference
    [SerializeField] GameObject[] pickups;
    1 reference
    [SerializeField] GameObject[] pickups2;
    1 reference
    [SerializeField] GameObject boss;

    1 reference
    [SerializeField] List<WeightedValue> enemies1;
    1 reference
    [SerializeField] List<WeightedValue> enemies2;
    1 reference
    [SerializeField] List<WeightedValue> enemies3;

    1 reference
    [SerializeField] PlayableDirector playableDirectorEnd;
    1 reference
    [SerializeField] PlayableDirector playableDirectorEmpty;
    1 reference
    [SerializeField] PlayableDirector playableDirectorBoss;

    2 references
    [SerializeField] int wave1Length;
    3 references
    [SerializeField] int wave2Length;
    4 references
    [SerializeField] int wave3Length;

    8 references
    [SerializeField] int totalLeft = 0;
    3 references | 3 references | 3 references
    bool wave2Started, wave3Started, end;
    4 references
```

Figure 7-3 Wave Serialized Fields

8 Game Walkthrough

8.1 Main Menu

8.1.1 Main Menu Screen

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



Figure 8-1. 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.

8.1.2 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 8-2. Settings Screen

8.1.3 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 8-3. Information Screen

8.1.4 New Game Screen

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



Figure 8-4. 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.

8.1.5 Load Game Screen

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



Figure 8-5. Load Game Screen

8.1.6 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.

8.2 Playthrough and UI

8.2.1 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 8-6. First Steps in Game

8.2.2 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 8-7. 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.

8.2.3 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.

8.2.4 Equipment and Inventory Interface



Figure 8-8. 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.

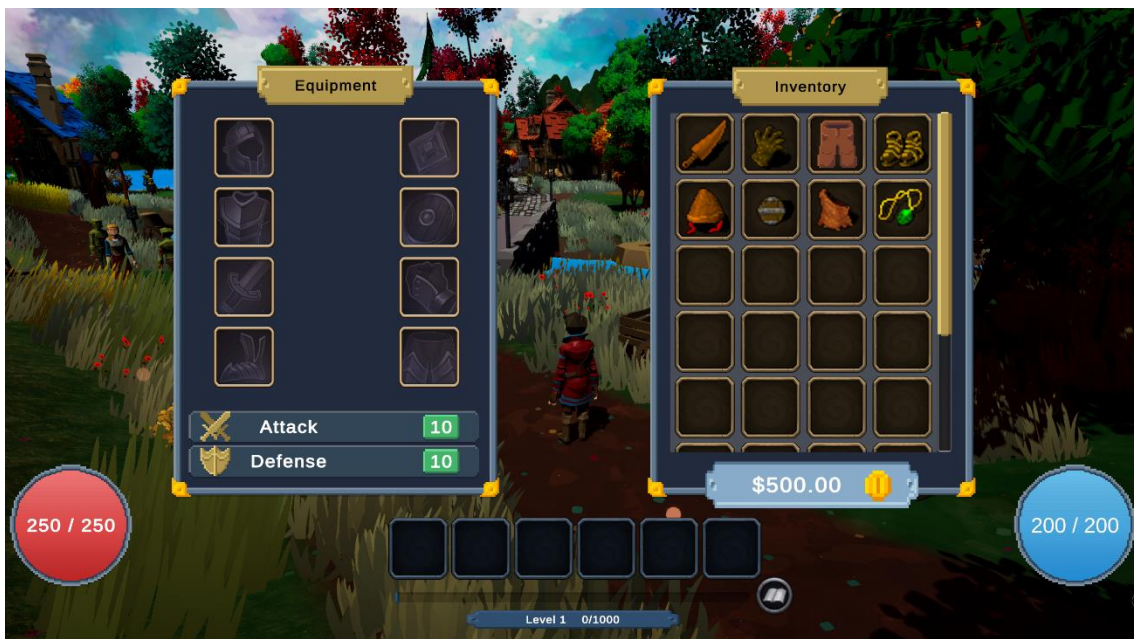


Figure 8-9. Equipment and Inventory Interfaces

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.

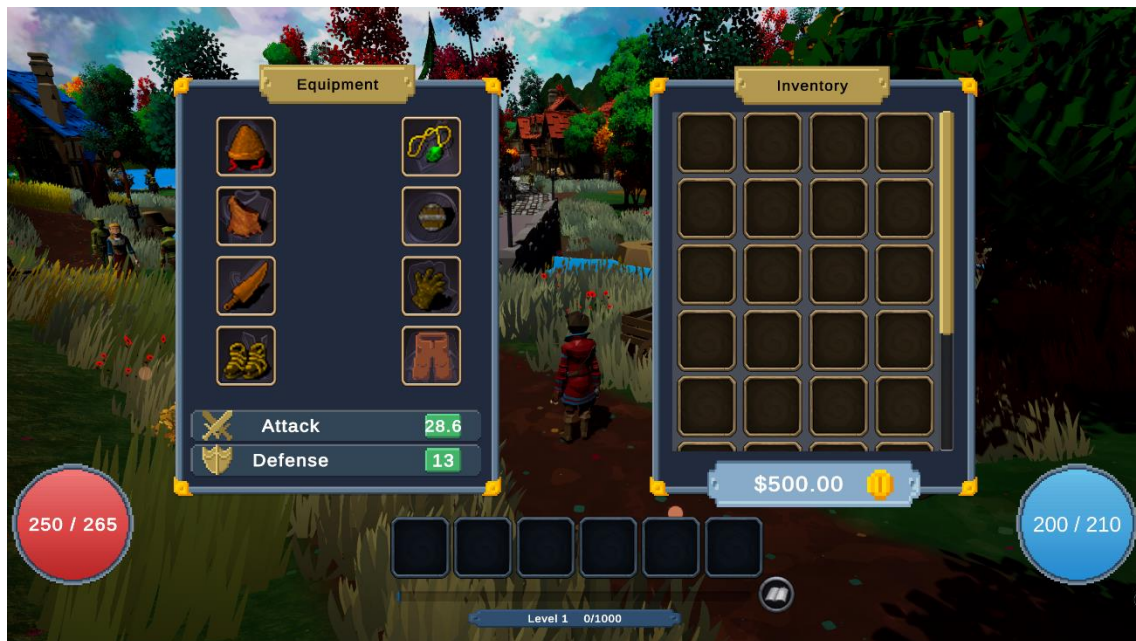


Figure 8-10 Equipping Items

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



Figure 8-11 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.

8.2.5 Quest Interface

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



Figure 8-12. 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 8-13. 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.

8.2.6 Leveling Up



Figure 8-14. 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 8-15. 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 8-16. Defeating Enemies in Evercoast Village

8.2.7 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 8-17 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 8-18. 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.

8.2.8 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.



8.2.9 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 8-19. 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 8-20. 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.

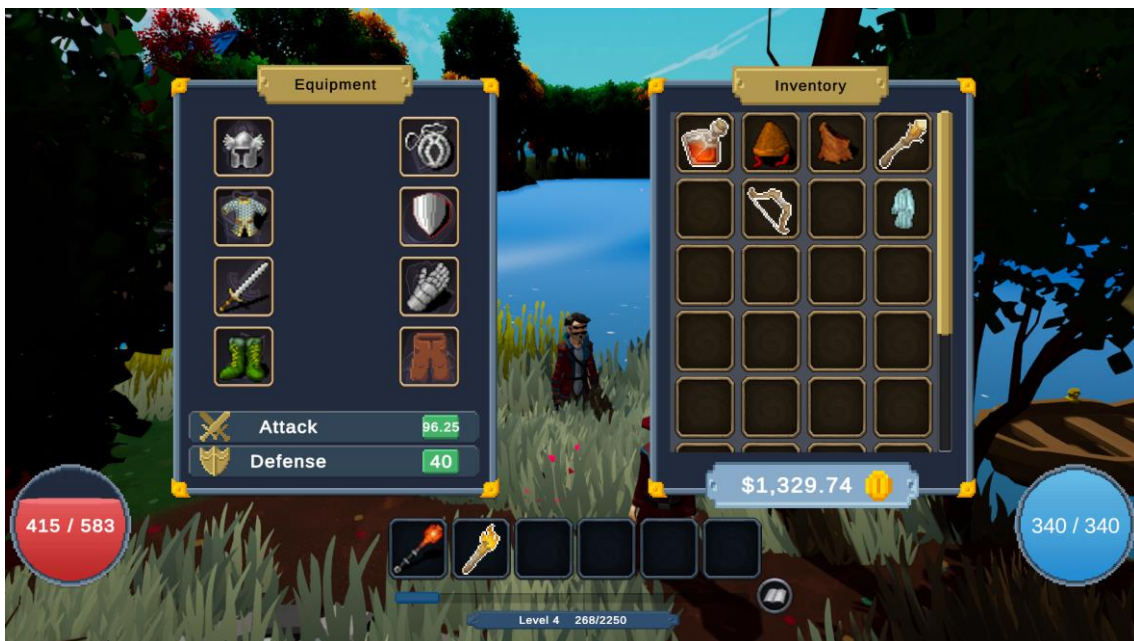


Figure 8-21. Equipping Spells

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 8-22. 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.

8.3 Rest of the Walkthrough and side Quests

8.3.1 Priest's Quest

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



Figure 8-23. 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 8-24. Village's Graveyard

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

8.3.2 Mayor's Quest

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



Figure 8-25. 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.

8.3.3 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 8-26 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.

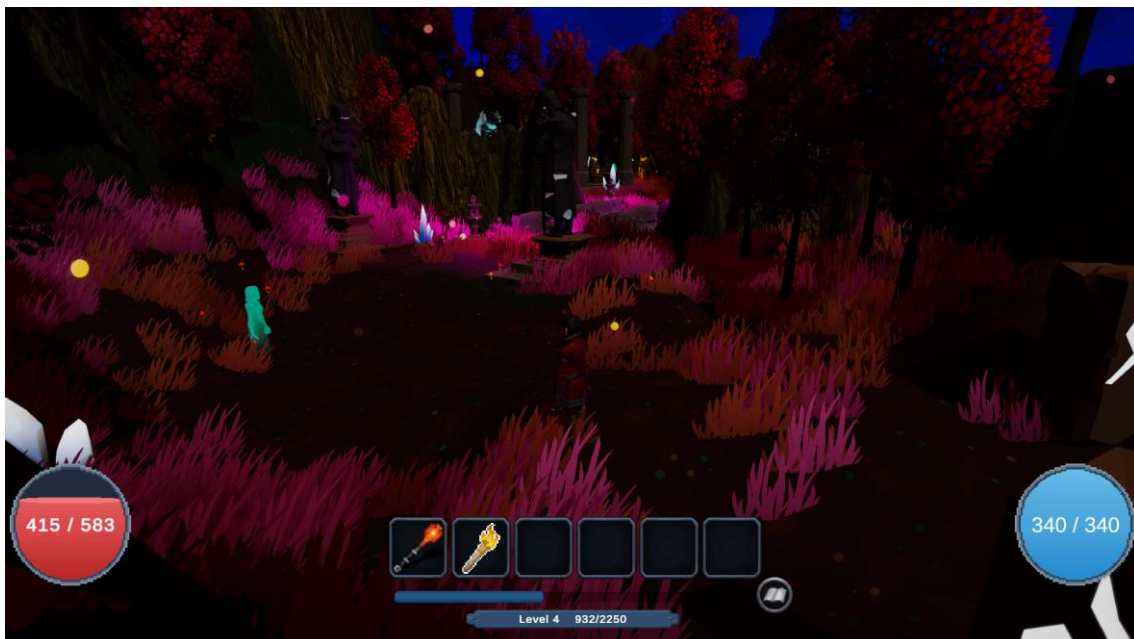


Figure 8-27. Mysterious Ruins

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 8-28. 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 8-29. 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.

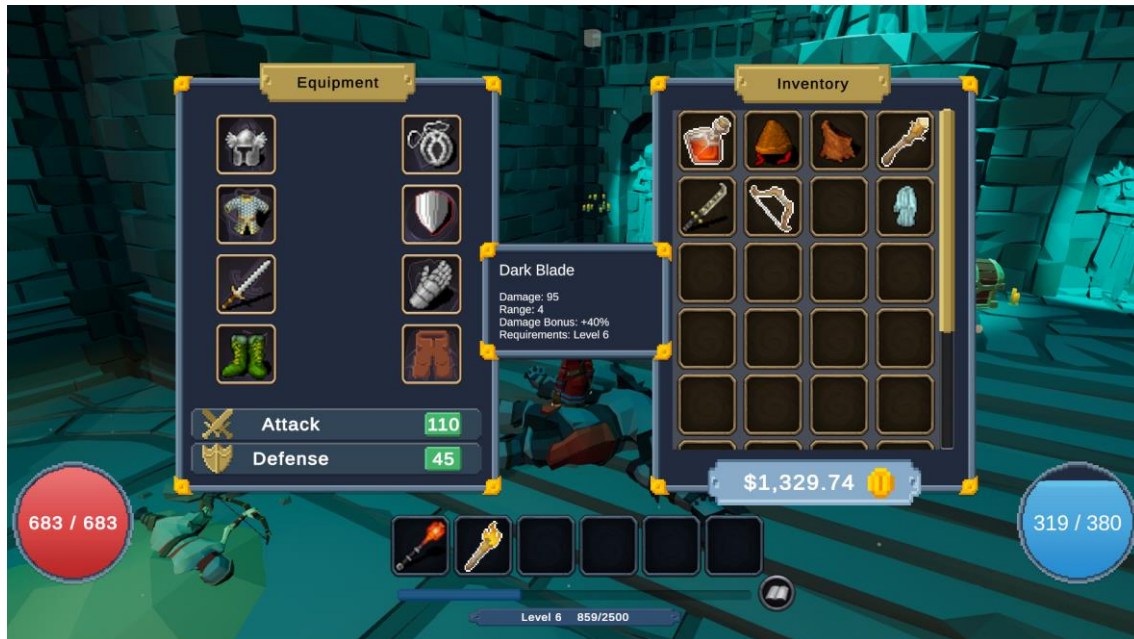


Figure 8-30. The Dark Blade

8.3.4 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 8-31. Secret Path



Figure 8-32. 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 8-33. Ruins Shopkeeper

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

8.3.5 The Goblin Cave

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



Figure 8-34. Goblin Cave

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



Figure 8-35. 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.

9 Conclusions and Future Plans

This thesis project has been a rewarding journey, blending creativity and technical knowledge to build a fully functioning 3D role-playing game using the Unity3D engine. Throughout the development process, I encountered numerous challenges, from managing limited resources to ensuring the game maintained its core RPG elements despite a simplified design. By working through these obstacles, I gained a deeper understanding of game development pipelines, especially in areas like scene creation, character interaction, and implementing mechanics such as combat, exploration, and quest systems.

The final product successfully incorporates key RPG mechanics, such as experience points, spellcasting, and item usage, while featuring unique environments and engaging quests. However, the game's low-poly style and limited level complexity reflect the constraints faced during development, particularly in terms of time and resources. While this simplicity added a distinct aesthetic to the project, it also serves as an area for future enhancement.

Looking forward, there are several directions this project could take:

Expanded Content: Adding more scenes, quests, and characters would enrich the game's world. Introducing new enemies, such as bosses or unique creatures, could increase the challenge and diversity of the gameplay.

Improved Graphics: As resources and skills develop, the game's visuals could be enhanced, moving towards a more detailed or realistic style without losing the charm of the original concept. Textures, lighting, and animations could also be improved for a more immersive experience.

Refined Mechanics: Incorporating more complex RPG mechanics, like skill trees, crafting systems, or more advanced combat, would enhance the player experience. This could be really useful in extending playtime and adding replay value.

Multiplayer Features: Expanding the game into a multiplayer experience would open up new possibilities for collaborative or competitive play, allowing players to team up or battle against each other in the world I created.

10 References and Citations

George Samaras Thesis Project: Rise of the Village Hero: A Roleplaying Game: Character Attributes, 3D animation, Virtual World Implementation, Sound Camera and Characters' Movement.

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 Animation Doc: <https://docs.unity3d.com/Manual/Animation>

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>