HELLENIC MEDITERRANEAN UNIVERSITY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

COURSE TYPE: INFORMATICS ENGINEERING T.E.

THESIS

Design and implementation of 3rd Person Action RPG Game in Unity 3D

George Beladakis – A.M. 4177

Supervisor: Ioannis Pachoulakis

Heraklion 2021

Thanking Section

At the beginning of this major project in 2020, I was at the most difficult phase in my life and in this section, I would like to thank everyone because I received a great deal of support and assistance for the completion of this project and not only.

I would like to thank and express my deepest appreciation to my supervisor, Professor Ioannis Pachoulakis, whose expertise was invaluable, for giving me the chance and providing guidance and feedback throughout this project.

I would also like to thank my parents for their unlimited support and sympathetic ear. You are always there for me. Also, I would like to thank my family and friends Vasilis Papanikolaou and Dimitris Iliadis, for supporting, giving constructive feedback, and noticing many of my mistakes.

In addition, I would like to thank my doctor, Georgia Milaki, whose dedication, and compassion are beyond limits. I really appreciate your skills and the care you have brought to the treatment, and I feel most fortunate to have you as my physician.

Abstract

This dissertation is about the development of a Third-Person Action RPG game by using the game engine Unity3D. In addition, more emphasis is placed on the development of artificial intelligence, using finite state machines that are responsible for the actions that an entity will take based on its conditions at any given time period.

In this game, I have created a 3D world, created entities like NPCs, guards and enemies that are controlled by the finite state machines. Also created animator controllers for the animations of the player and the rest of the entities. Created a simple UI and I developed a dialogue & quests system, stats and items, player's controllers, and an inventory system.

All things considered; player's purpose is to discover a fantasy/medieval world through storytelling by completing quests. At the same time, player has the ability to collect items like weapons and consumables that will come in handy when in need and danger. Another key feature is the ability to gain experience by completing quests and killing enemies, to become stronger and be able to cope with the difficulty of the quests.

// Περίληψη

Η πτυχιακή αυτή έχει ως σκοπό την ανάπτυξη ενός Third-Person Action RPG παιχνιδιού χρησιμοποιώντας την παιχνιδομηχανή Unity3D. Επιπροσθέτως έδωσα παραπάνω έμφαση στην ανάπτυξη τεχνητής νοημοσύνης χρησιμοποιώντας finite state machines, τα οποία είναι υπεύθυνα για τις ενέργειες που θα πραγματοποιήσει μια οντότητα με βάση τις συνθήκες της σε οποιαδήποτε χρονική περίοδο.

Σε αυτό το παιχνίδι, δημιούργησα έναν τρισδιάστατο κόσμο, δημιούργησα οντότητες όπως NPC, φρουρούς και εχθρούς που ελέγχονται από τα finite state machines. Επίσης δημιούργησα animator controllers για τα animations του παίκτη και των υπόλοιπων οντοτήτων. Δημιούργησα ένα απλό UI και ανέπτυξα ένα σύστημα διαλόγου και για quests, αντικείμενα και στατιστικά, controllers για τον παίκτη και ένα σύστημα για το inventory.

Με βάση τα παραπάνω, ο σκοπός του παίκτη είναι να ανακαλύψει έναν φανταστικό/μεσαιωνικό κόσμο μέσω της αφήγησης ολοκληρώνοντας quests. Ταυτόχρονα ο παίκτης έχει την ικανότητα να συλλέγει αντικείμενα όπως όπλα και αναλώσιμα που θα φανούν χρήσιμα όταν υπάρχει ανάγκη και κίνδυνος. Ένα άλλο βασικό χαρακτηριστικό είναι η δυνατότητα απόκτησης εμπειρίας ολοκληρώνοντας αποστολές και σκοτώνοντας εχθρούς, για να γίνεται δυνατότερος και να μπορεί να ανταπεξέρχεται με την δυσκολία των αποστολών.

Table of Contents

1. Introduction ... 16
    1.1 Summary of the Game ... 16
    1.2 Motivation of Making this Project ... 16
    1.3 Purpose and Objectives of the Project ... 16
2. Technologies and Concepts ... 17
    2.1 What is a Game Engine and What is Unity ... 17
    2.2 What is an Action RPG Game Genre ... 17
    2.3 What is Artificial Intelligence ... 17
        2.3.1 Artificial Intelligence in Video Games ... 17
        2.3.2 What is a Finite State Machine ... 18
    2.4 A Few Important Unity Concepts ... 18
        2.4.1 GameObject ... 18
        2.4.2 Component ... 18
        2.4.3 Prefab ... 18
        2.4.4 Scriptable Object ... 19
        2.4.5 Coroutines ... 19
3. Working in Unity and Resources ... 19
    3.1 Unity Editor's Interface ... 19
        3.1.1 Toolbar ... 20
        3.1.2 Hierarchy Window ... 20
        3.1.3 Scene View ... 20
        3.1.4 Game View ... 21
        3.1.5 Inspector Window ... 21
        3.1.6 Project Window ... 22
    3.2 About Resources and Assets ... 22
4. Introduction to the Player & The Rest of the Entities ... 23
    4.1 Main Character and Controls ... 23
    4.2 NPCs ... 24
    4.3 Enemies & Guards ... 24
    4.4 Analyzing NPC, Enemy and Guard AI ... 24
        4.4.1 Plain NPC ... 25
        4.4.2 Farmer NPC ... 25
        4.4.3 Woodcutter NPC ... 26
        4.4.4 Enemy NPC ... 26

4.4.5 Guard NPC 27
5. Game Development 28
5.1 Environment Creation 28
5.2 Lighting & Post-processing 31
5.3 Animator Controllers 35
5.4 Quest & Dialogue System 39
5.4.1 Overview 39
5.4.1 Code 40
5.5 Player 51
5.5.1 Overview 51
5.5.1 Code 51
5.6 Stats & Health 64
5.6.1 Overview 64
5.6.1 Code 64
5.7 AI 74
5.7.1 State Machine 74
5.7.1.1 Overview 74
5.7.1.2 Code 75
5.7.2 Enemy AI Controller & States 78
5.7.2.1 Overview 78
5.7.2.2 Code 78
5.7.3 Guard AI Controller & States 87
5.7.3.1 Overview 87
5.7.3.2 Code 87
5.7.4 NPC AI 94
5.7.4.1 DialogueNPC and NPCs 94
5.7.4.1.1 Overview 94
5.7.4.1.2 Code 94
5.7.4.2 NPC AI Controller & States 94
5.7.4.2.1 Overview 94
5.7.4.2.2 Code 95
5.7.4.3 Farmer AI Controller & States 99
5.7.4.3.1 Overview 99
5.7.4.3.2 Code 100

5.7.4.4 Woodcutter AI Controller & States 111
5.7.4.4.1 Overview 111
5.7.4.4.2 Code........ 112
5.8 Mover & Fighter........ 123
5.8.1 Overview........ 123
5.8.2 Code 124
5.9 Items, Weapon items & Consumables........ 128
5.9.1 Overview........ 128
5.9.2 Code 131
5.10 Projectiles 136
5.10.1 Overview........ 136
5.10.2 Code 137
5.11 Inventory 138
5.11.1 Overview........ 138
5.11.2 Code 139
5.12 UI Scripts........ 145
5.12.1 Overview........ 145
5.12.2 Code 147
5.13 Other Scripts........ 153
5.13.1 SmartRenderer.cs 153
5.13.2 WaypointPath.cs 154
5.13.3 Safe.cs 155
5.13.4 MainMenuManager.cs 157
5.13.5 SceneLoader.cs 160
5.13.6 DiscordController.cs 161
5.14 Navigation 162
6. Epilogue........ 165
6.1 Conclusion........ 165
6.2 Difficulties........ 165
6.3 Future Improvements 165
7. Bibliography 166

Table of Figures

Figure 1 – Unity Editor's workspace interface ... 19
Figure 2 – Unity Editor's toolbar ... 20
Figure 3 – Hierarchy window ... 20
Figure 4 – Example Scene view of the main scene. ... 21
Figure 5 – Example Game view of the main menu scene. ... 21
Figure 6 – Inspecting the properties of the Player GameObject. ... 22
Figure 7 – Project files as viewed in the Project window ... 22
Figure 8 – Main character "Nysa" ... 23
Figure 9 – Plain NPC's finite state machine – NPCAIController.cs ... 25
Figure 10 – Farmer's finite state machine – FarmerController.cs ... 25
Figure 11 – Woodcutter's finite state machine – WoodcutterController.cs ... 26
Figure 12 – Enemy's finite state machine – EnemyAIController.cs ... 26
Figure 13 – Woodcutter's finite state machine – GuardController.cs ... 27
Figure 14 – A Terrain GameObject and it's components. ... 28
Figure 15 – A top-down view of the Terrain. ... 29
Figure 16 – A top-down view of the final result of the Scene. ... 29
Figure 17 – Textures in Paint Texture. ... 30
Figure 18 – Bushes and grasses that were used. ... 30
Figure 19 – Screenshot of the first village ... 31
Figure 20 – Mixed Lighting in Lighting Window. ... 31
Figure 21 – Light component of Directional Light GameObject. ... 31
Figure 22 – Lightmapping Settings ... 32
Figure 23 – Baked lights & post process off ... 33
Figure 24 – Baked lights on & post process off ... 33
Figure 25 – Camera GameObject's Post-process Layer component ... 34
Figure 26 – Camera GameObject's Post-process Layer component ... 34
Figure 27 – Baked Lights & post process on ... 35
Figure 28 – Player's Animator Controller ... 36
Figure 29 – Player's "Locomotion" Blend Tree ... 36
Figure 30 – Player's Animator Controller's parameters ... 36
Figure 31 Enemy & Guard Animator Controller ... 37
Figure 32 – Plain NPC Animator Controller ... 37
Figure 33 – Farmer NPC Animator Controller's parameters ... 38

Figure 34 – Woodcutter NPC Animator Controller's parameters 38
Figure 35 – In game screenshot of the quest list........ 39
Figure 36 – In game screenshot of the Nysa being in dialogue with an NPC that gives quest. 39
Figure 37 – Quest & Dialogue System folder structure. 40
Figure 38 – Speaker.cs........ 40
Figure 39 – DialogueLine.cs........ 41
Figure 40 – Dialogue.cs........ 41
Figure 41 – DialogueManager.cs part 1........ 42
Figure 42 – DialogueManager.cs part 2........ 42
Figure 43 – DialogueManager.cs part 3........ 43
Figure 44 – DialogueManager.cs part 4........ 43
Figure 45 – Quest.cs 44
Figure 46– QuestGoal.cs 44
Figure 47 – QuestType.cs........ 45
Figure 48 – QuestEnemy.cs........ 45
Figure 49 – NPCQuestGiver.cs part 1 46
Figure 50 – NPCQuestGiver.cs part 2 46
Figure 51 – QuestManager.cs part 1........ 47
Figure 52 – QuestManager.cs part 2........ 48
Figure 53 – QuestManager.cs part 3........ 49
Figure 54 – QuestManager.cs part 4........ 50
Figure 55 – QuestManager.cs part 5........ 50
Figure 56 – PlayerController.cs part 1........ 51
Figure 57 – PlayerController.cs part 2........ 52
Figure 58 – PlayerController.cs part 3........ 52
Figure 59 – PlayerController.cs part 4........ 53
Figure 60 – PlayerController.cs part 5........ 54
Figure 61 – PlayerController.cs part 6........ 55
Figure 62 – PlayerCombat.cs part 1........ 56
Figure 63 – PlayerCombat.cs part 2........ 57
Figure 64 – PlayerCombat.cs part 3........ 57
Figure 65 – PlayerCombat.cs part 4........ 58
Figure 66 – PlayerCombat.cs part 5........ 59
Figure 67 – PlayerCombat.cs part 6........ 60

Figure 68 – PlayerMovement.cs part 1 61
Figure 69 – PlayerMovement.cs part 2 62
Figure 70 – PlayerMovement.cs part 3 62
Figure 71 – PlayerMovement.cs part 4 63
Figure 72 – IAction.cs 63
Figure 73 – ActionScheduler.cs 64
Figure 74 – CharacterClass.cs 65
Figure 75 – Stat.cs 65
Figure 76 – Progression of the Player 66
Figure 77 – Progression.cs part 1 67
Figure 78 – Progression.cs part 2 68
Figure 79 – Experience.cs 68
Figure 80 – BaseStats.cs part 1 69
Figure 81 – BaseStats.cs part 2 69
Figure 82 – BaseStats.cs part 3 70
Figure 83 – ExperienceDisplay.cs 70
Figure 84 – LevelDisplay.cs 71
Figure 85 – XPDisplay.cs 71
Figure 86 – Health.cs part 1 72
Figure 87 – Health.cs part 2 72
Figure 88 – Health.cs part 3 73
Figure 89 – Health.cs part 4 73
Figure 90 – Example states and transitions 75
Figure 91 – StateMachine.cs part 1 76
Figure 92 – StateMachine.cs part 2 77
Figure 93 – IState.cs 77
Figure 94 – EnemyAIController.cs part 1 79
Figure 95 – EnemyAIController.cs part 2 79
Figure 96 – EnemyAIController.cs part 3 80
Figure 97 – EnemyAIController.cs part 4 80
Figure 98 – EnemyAIController.cs part 5 81
Figure 99 – EnemyAttackNPC.cs 82
Figure 100 – EnemyAttackGuard.cs 83
Figure 101 – EnemyAttackPlayer.cs part 1 84

Figure 102 – EnemyAttackPlayer.cs part 2 84
Figure 103 – EnemySuspicion.cs 85
Figure 104 – EnemyGuarding.cs part 1 86
Figure 105 – EnemyGuarding.cs part 2 86
Figure 106 – GuardAIController.cs part 1 88
Figure 107 – GuardAIController.cs part 2 88
Figure 108 – GuardAIController.cs part 3 89
Figure 109 – GuardAIController.cs part 4 89
Figure 110 – GuardAttackEnemy.cs part 1 90
Figure 111 – GuardAttackEnemy.cs part 2 91
Figure 112 – GuardGuarding.cs part 1 91
Figure 113 – GuardGuarding.cs part 2 92
Figure 114 – GuardSuspicion.cs 93
Figure 115 – GuardTalk.cs 93
Figure 116 – DialogueNPC.cs 94
Figure 117 – Talk.cs 95
Figure 118 – Motion.cs part 1 96
Figure 119 – Motion.cs part 2 97
Figure 120 – Flee.cs part 1 98
Figure 121 – Flee.cs part 2 98
Figure 122 – GatherableResource.cs 100
Figure 123 – FarmerController.cs part 1 100
Figure 124 – FarmerController.cs part 2 101
Figure 125 – FarmerController.cs part 3 102
Figure 126 – FarmerController.cs part 4 103
Figure 127 – FarmerController.cs part 5 103
Figure 128 – FarmerInitialDecision.cs 104
Figure 129 – FarmerSearchResource.cs 105
Figure 130 – FarmerMoveToResource.cs part 1 106
Figure 131 – FarmerMoveToResource.cs part 2 106
Figure 132 – FarmerHarvest.cs part 1 107
Figure 133 – FarmerHarvest.cs part 2 107
Figure 134 – FarmerReturnGoods.cs 108
Figure 135 – FarmerFlee.cs part 1 109

Figure 136 – FarmerFlee.cs part 2 110
Figure 137 – FarmerTalk.cs 110
Figure 138 – WoodcutterController.cs part 1 112
Figure 139 – WoodcutterController.cs part 2 113
Figure 140 – WoodcutterController.cs part 3 114
Figure 141 – WoodcutterController.cs part 4 115
Figure 142 – WoodcutterController.cs part 5 115
Figure 143 – WoodcutterInitialDecision.cs 116
Figure 144 – WoodcutterSearchResource.cs part 1 117
Figure 145 – WoodcutterSearchResource.cs part 2 117
Figure 146 – WoodcutterMoveToResource.cs 118
Figure 147 – WoodcutterHarvest.cs part 1 119
Figure 148 – WoodcutterHarvest.cs part 2 119
Figure 149 – WoodcutterReturnWood.cs 120
Figure 150 – WoodcutterFlee.cs part 1 121
Figure 151 – WoodcutterFlee.cs part 2 121
Figure 152 – WoodcutterWalk.cs part 1 122
Figure 153 – WoodcutterWalk.cs part 2 122
Figure 154 – WoodcutterTalk.cs 123
Figure 155 – Mover.cs part 1 124
Figure 156 – Mover.cs part 2 125
Figure 157 – Fighter.cs part 1 125
Figure 158 – Fighter.cs part 2 126
Figure 159 – Fighter.cs part 3 127
Figure 160 – Fighter.cs part 4 128
Figure 161 – The “Fury” WeaponItem scriptable object. 130
Figure 162 – The “Health Potion” ConsumableItem scriptable object. 130
Figure 163 – The “Fury” sword pickup and its components. 130
Figure 164 – The “Fury” sword that is used when it’s on an agent’s hands. 131

Figure 165 – Item.cs 131
Figure 166 – ConsumableItem.cs 131
Figure 167 – WeaponItem.cs part 1 132
Figure 168 – WeaponItem.cs part 2 132
Figure 169 – WeaponItem.cs part 3 133
Figure 170 – Pickup.cs part 1 133
Figure 171 – Pickup.cs part 2 134
Figure 172 – DropItemOnDeath.cs 134
Figure 173 – ItemPickUp.cs 135
Figure 174 – DropItem.cs 135
Figure 175 – The arrow projectile GameObject and its components. 136
Figure 176 – The arrow hit effect GameObject and its components. 136
Figure 177 – DestroyAfterEffect.cs 137
Figure 178 – Projectile.cs part 1 137
Figure 179 – Projectile.cs part 2 138
Figure 180 – Projectile.cs part 3 138
Figure 181 – A screenshot of the inventory 139
Figure 182 – Inventory.cs part 1 140
Figure 183 – Inventory.cs part 2 141
Figure 184 – Inventory.cs part 3 142
Figure 185 – Inventory.cs part 4 143
Figure 186 – Inventory.cs part 5 144
Figure 187 – Inventory.cs part 6 144
Figure 188 – Inventory.cs part 7 145
Figure 189 – The HUD GameObject. 146
Figure 190 – Enemy's health bar and damage display. 146
Figure 191 – PlayerHealthDisplay.cs 147
Figure 192 – PlayerAbilityDisplay.cs 147
Figure 193 – ExperienceDisplay.cs 148
Figure 194 – LevelDisplay.cs 148
Figure 195 – XPDisplay.cs 149
Figure 196 – EscapeMenu.cs 149
Figure 197 – SetVolume.cs 150
Figure 198 – GraphicSettings.cs part 1 151

Figure 199 – GraphicSettings.cs part 2 151
Figure 200 – EnemyHealthDisplay.cs part 1 152
Figure 201 – PopupText.cs 152
Figure 202 – SmartRenderer.cs part 1 153
Figure 203 – SmartRenderer.cs part 2 153
Figure 204 – Waypoint path example. 154
Figure 205 – WaypointPath.cs part 1 154
Figure 206 – WaypointPath.cs part 2 155
Figure 207 – Safe point example. 156
Figure 208 – Safe.cs part 1 156
Figure 209 – Safe.cs part 2 157
Figure 210 – Main menu scene GameObjects. 158
Figure 211 – Camera – MainMenu Manager GameObject. 158
Figure 212 – MainMenuManager.cs part 1 159
Figure 213 – MainMenuManager.cs part 2 159
Figure 214 – SceneLoader.cs 160
Figure 215 – Discord's rich presence. 161
Figure 216 – DiscordController.cs 162
Figure 217 – Navigation, Agents tab 163
Figure 218 – Navigation, Bake tab 163
Figure 219 – NavMesh area of a part of the scene 164

1. Introduction

1.1 Summary of the Game

Nysa's Quest is a 3rd-person action RPG game developed for PC. Thanks to the elements of the RPG and action genres, the game gives the impression that the player is part of a fantasy/mediaeval world. By taking on the role of a character and the scenario, player will aim to complete quests and fight with various enemies to reach the goal.

As mentioned earlier, the game takes place in a fantasy/mediaeval world. The player takes control of a young woman named Nysa, who has lost her parents and family estate to a band of exiled knights. Nysa's passion for revenge and justice drives her on a quest to find those responsible for her family's misfortune and retrieve the stolen family sword.

1.2 Motivation of Making this Project

The motivation for creating this project stems from 3 things. First, my love of games. When I play a game similar to this project, I am always curious about the scenarios, exploration, character empowerment, gathering, and how these things actually work. Secondly, it's exciting to create something. For example, when I am working with Unity, it's always fun and exciting to create a movement script for a character and then press play to see the result. The third and final point is the will to create my own game, having an idea, writing it down and then implementing with my resources.

1.3 Purpose and Objectives of the Project

The purpose of this project is to develop a 3D Action RPG game, focusing on creating Artificial Intelligence and improving my programming skills. To achieve that purpose, I'm using the reliable Unity3D game engine and programming in C#, because it's simple, well documented and there is a wide range of tutorials available thanks to the community. The goal of the game is to make the player feel like they are a part of this world and that there is a goal to achieve. There are 4 quests to complete the game, but there are also 3 more side quests so that if the player wants to explore more or to get stronger if they want to.

2. Technologies and concepts

2.1 What is a Game Engine and What is Unity

A game engine is a software framework designed primarily for developing video game development, and in general includes related libraries and utilities. The main features that a game engine typically provides, include a rendering engine for two-dimensional (2D) or three-dimensional (3D) graphics, memory management, networking, artificial intelligence, animation, scripting, a physics engine, streaming, sound, threading, localization support, video support for cinematics, and scene graph.

Unity was created by Unity Technologies and is a cross-platform game engine that is mainly used to develop video games and simulations for computers, consoles, and mobile devices. Unity was launched in 2005 and since then it has been expanded to 27 platforms. The Unity game engine is an "all-purpose" as it supports two-dimensional (2D) and three-dimensional (3D) graphics, is also a good choice for Virtual-Reality (VR) and natively supports the C# programming language. It is important to note that the engine is not only used in the video game industry, but also in industries such as engineering, construction, architecture, film, and automotive.

2.2 What is an Action RPG Game Genre

An action RPG is a subgenre or subdivision of the RPG genre. That is, this subgenre includes RPG combat systems that combine RPG mechanics with real-time, direct, and reflexive action game combat systems. In action role-playing games, the player has real-time direct control over a character's movements, actions in combat, and stats to determine relative strength and abilities.

2.3 What is Artificial Intelligence

The natural intelligence displayed by humans or animals can also be demonstrated by machines. This term is called Artificial Intelligence (AI). The term Artificial Intelligence is also used for machines that attempt to mimic "cognitive" functions that humans perform with their minds, such as "learning" and "problem solving."

2.3.1 Artificial Intelligence in Video Games

In video games, artificial intelligence is used to achieve flexible, responsible, and intelligent behaviors, especially in non-player characters (NPCs), that resemble human intelligence. AI in video games is a distinct subfield and is different from academic AI. Today, games often use existing techniques such as decision trees and pathfinding to drive the actions of NPCs. Often AI is used in mechanisms that are not directly visible to the user, such as data mining and procedural content creation.

2.3.2 What is a Finite State Machine

A finite state machine (FSM) belongs to the field of expert systems and is represented as a graph. An FSM graph is an abstract representation of set of objects, symbols, events, actions, or properties of the phenomenon to be represented. Specifically, the graph contains nodes (states) that represent a mathematical abstraction, and edges (transitions) that represent a conditional relationship between nodes. The FSM can only be in one state at a time, and the current state can transition to another if the condition in the corresponding transition is satisfied. In short, an FSM is specified by three main components:

- **states** which store information about a task.
- **transitions** between states and are described by a condition that needs to be fulfilled for a state to change.
- **actions** that are followed in each state.

FSMs are really easy to design, implement, visualize, and debug. Also, they have proven to work well with games over the years of their existence. On the other hand, they can be extraordinarily complex on a large scale and are computationally limited to specific tasks within game AI.

2.4 A Few Important Unity Concepts

2.4.1 GameObject

A GameObject is the most important object in Unity Editor. Every object in the game is a GameObject (characters, props, scenery, etc), but they can not do much themselves. They are containers for the components that provide the actual functionality. A Transform component is always attached to a GameObject to represent its position and orientation in the scene.

2.4.2 Component

Components are the functional pieces of a GameObject that define it's behavior. For example, on a Main Camera GameObject, it's Camera component adds the functionality of the camera to the GameObject.

2.4.3 Prefab

A prefab is a copy of a GameObject that can be saved with its properties and components so that it can be used again and again in different scenes. Changes made to a prefab can be applied either manually or automatically to instances of that prefab, so that changes can be easily made throughout the project without having to repeat the same actions.

2.4.4 Scriptable Object

A ScriptableObject serves as a data container and is used to store large amounts of data, independent of class instances. ScriptableObjects are often used to reduce the memory footprint of a project by avoiding copies of values. During an editor session, ScriptableObjects are mainly used to store and hold data as an asset in a project at runtime.

2.4.5 Coroutines

A coroutine works similarly to a function, but can pause execution for a few seconds and before resuming, a condition must be met or a certain time must be waited.

3. Working in Unity and Resources

For this project I used the 2019.4.26.f1 Long Term Support (LTS) version of Unity because it provides maximum stability, 2 years of support and no API changes.

3.1 Unity Editor's Interface

Unity provides a user interface (UI) that is user-friendly and easy to customize. In the following figure, you can see the Unity workspace interface, which uses a custom layout that I used to create this project. In the next subsections, I'll go over some of the windows.

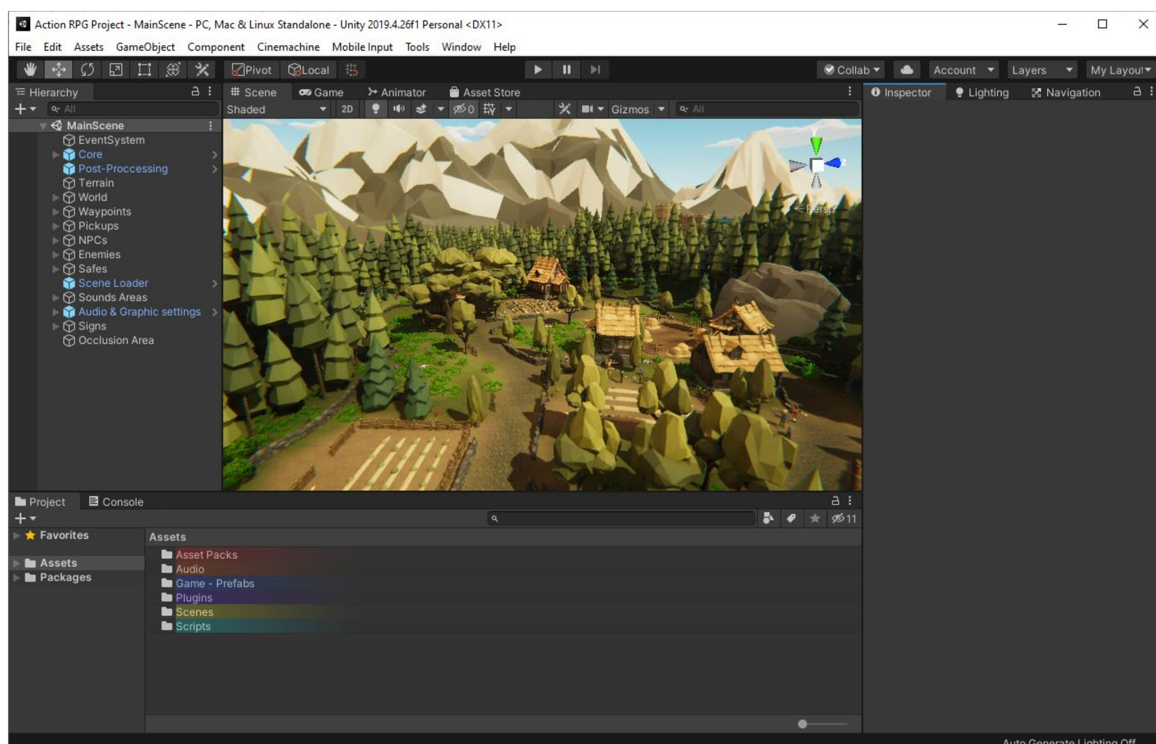


Figure 1 – Unity Editor's workspace interface

3.1.1 Toolbar

The toolbar is located at the top of Unity Editor and consists of several groups of controls. It provides quick access to the most important functions. It starts on the left and contains the basic tools for controlling the scene view and the GameObjects within it. This is followed by the Play, Pause, and Step buttons. The buttons on the right give access to Unity Collaborate, Unity Cloud Services and Unity Account, followed by a layer visibility menu and the customizable editor layout menu.

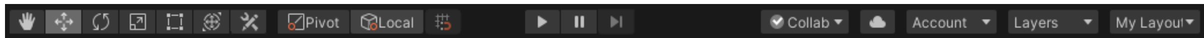


Figure 2 – Unity Editor's toolbar

3.1.2 Hierarchy Window

The Hierarchy window provides a textual representation of each game object in the scene in the hierarchy. The Hierarchy window is used to group and sort the GameObjects present in a scene. GameObjects added or removed from Scene View can also be added or removed from the Hierarchy window.

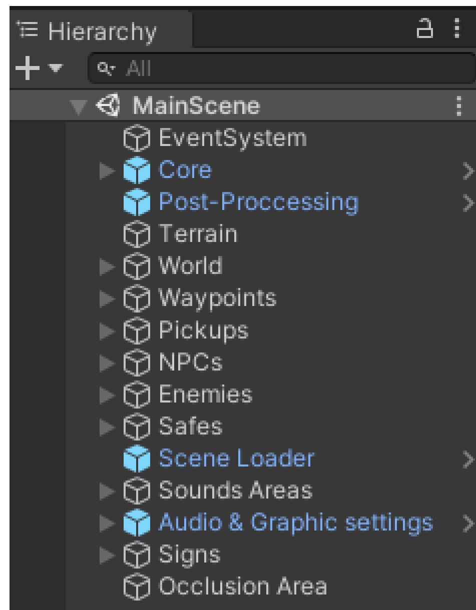


Figure 3 – Hierarchy window

3.1.3 Scene View

The Scene View allows you to visually navigate and edit the scene (add, remove, select, and position scenery, characters, cameras, lights, and all other types of game objects). The Scene View provides a 3D or 2D perspective, depending on the type of project.

Figure 4 – Example Scene view of the main scene.

3.1.4 Game View

The game view is used to simulate what the final rendering of the game will look like by the Scene Cameras. The simulation begins by clicking the Play button on the toolbar.

Figure 5 – Example Game view of the main menu scene.

3.1.5 Inspector Window

In the Inspector window, you can edit, add, or remove properties (components) of the currently selected GameObject. There are many types of GameObjects that can have different properties. The layout and contents of the Inspector window change each time a different GameObject is selected.

Figure 6 – Inspecting the properties of the Player GameObject.

3.1.6 Project Window

The project window is like a file explorer. It displays the files associated with the project and is the main method for navigating through the assets and other project files in the editor. As you can see in Figure 7, there are colored folders. This is an asset from the Unity asset store called "Rainbow Folders", which I use because it helps me find folders easily, organize them well, and increase productivity.

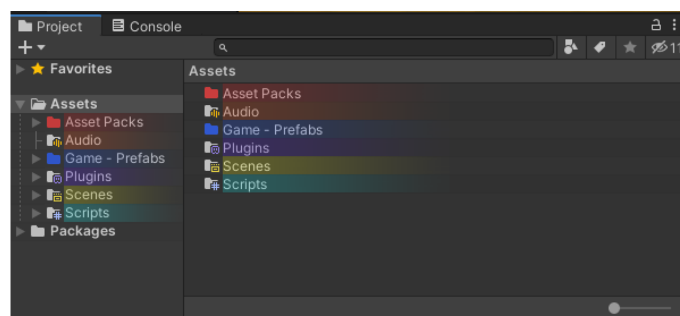


Figure 7 – Project files as viewed in the Project window

3.2 About Resources and Assets

Some assets used in this project were either freely available or purchased for legal use, or created by me using the image editor GIMP or the audio editor Audacity.

4. Introduction to the Player & The Rest of the Entities

In this chapter I go through and analyze the entities of the game. These are the player, the simple NPC, the farmer, the woodcutter, the guard, and the enemy.

4.1 Main Character and controls

Diving into "Nysa's Quest" world. player takes control of a heroine named "Nysa Fell". Her motives are what drive her forward and make the player feel like they are pursuing the same goals as "Nysa" and completing the quests to reach the goal.

Figure 8 – Main character “Nysa”

The model from Figure 8 is from Synty Studio’s Polygon Series Fantasy Kingdom. Once in the game and player has control over the character then the following actions can be performed:

- Movement: WASD or Arrow keys
- Camera rotation: Q & E keys
- Zoom in & out: Mouse scroll wheel
- Attack: Left click
- Sprint: Shift key
- Roll: Space key
- Quests: Z key
- Inventory: Tab key
- Pause Menu: ESC

By pressing the Space key, player is rolling, that is an ability that can be used every 2 seconds and when rolling player is avoiding any attacks.

4.2 NPCs

Generally, non-player characters (NPCs) are used to populate the world of a game and are usually controlled by the game's AI. NPCs can be used to advance the plot, help the player as allies or partners, and they can serve as merchants, doctors, save points, and more. In this project, however, NPCs are used to populate the world and make it more lively, to give quests, or just to have a little conversation.

Also, NPCs can do 3 things: first, they can walk around or stop to make the world feel more alive rather than static. Second, they can give quests so that the player interacts with the NPC by having a dialogue to get a quest. The NPCs that give quests can be identified by an exclamation point above them. Third and finally, some NPCs have work, their role as well is to make the world feel more alive, like the NPCs that walk around, but beyond that they add variety to the NPC population. However, these 3 types can also be combined, meaning an NPC can have a job, but also give a quest.

4.3 Enemies & Guards

Enemies are obstacles that must be overcome in order to reach the goal of a quest or reach the finish point of a level. Usually, enemies try to kill or prevent the player from reaching the goal. Also, enemies usually guard an area by standing still or walking on the path waiting for the player to attack them. Also, enemies fight guards, and they can be set to attack NPCs. Guards, on the other hand, have no conflicts with the player and only guard the town and hunt enemies.

4.4 Analyzing NPC, Enemy and Guard AI

AI behavior is easily controlled by a finite state machine. Bellow follows each AI controller's finite state machine (finite state machine is explained in sub-chapter 2.3.1) controller. Controllers:

- NPC
  - *NPCAIController.cs*
  - *FarmerController.cs*
  - *WoodcutterController.cs*
- Enemy
  - *EnemyAIController.cs*
- Guard
  - *GuardController.cs*

4.4.1 Plain NPC

Let us start with the simple NPC, and as Figure 9 shows, it is the simplest.

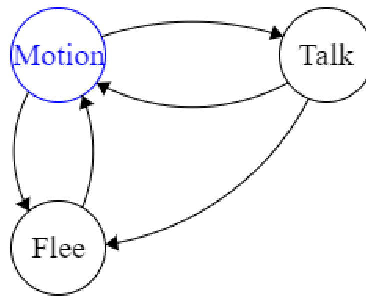


Figure 9 – Plain NPC's finite state machine – NPCAIController.cs

The initial state of a simple NPC is movement. These NPCs can either stand still or walk. If a simple NPC has a quest to give, the player can interact with him at any time to get the quest, unless he is on the run. Finally, if an enemy threatens him, he flees to a safe place.

4.4.2 Farmer NPC

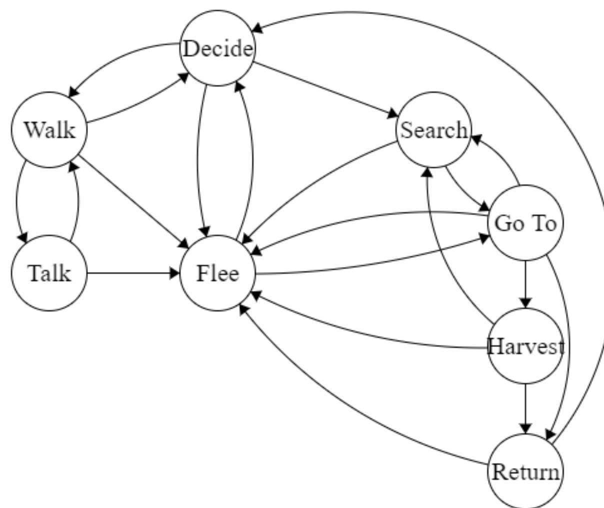


Figure 10 – Farmer's finite state machine – FarmerController.cs

The farmer's initial state is colored blue, as shown in Figure 10. It starts with the farmer deciding whether or not to go for a walk, and this depends on his work ethic. For example, if the farmer decides to look for crops in the area (work), then he finds one and walks to that crop. Once he gets there, he starts harvesting. After harvesting the first crop, the farmer searches for another crop to harvest until he has collected 4 crops. Once the 4 crops are collected, the farmer returns the goods to a return point and then decides again. If the decision was "walk", the farmer walks for some time before making another decision. However, if a threatening enemy appears, the farmer will immediately flee. If the farmer is in the process of harvesting his crops when the enemy appears, and he has collected 2 crops for example, he will drop the basket and flee

to hide in a safe place, returning to the basket after a while to pick it up and collect the remaining 2 crops. And when the farmer gives a quest, the player only talks to him when he is not working.

4.4.3 Woodcutter NPC

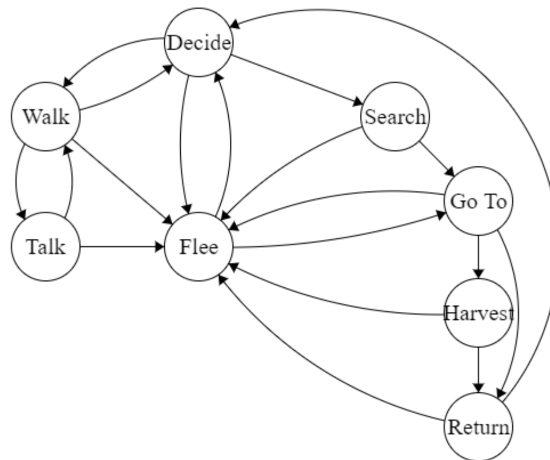


Figure 11 – Woodcutter's finite state machine – WoodcutterController.cs

Similarly, but more simply, woodcutter begins with a decision. For example, if this NPC decides to work, he finds a nearby tree, moves to that tree and begins cutting the tree there (harvesting), then returns the wood to a return point to decide again. However, if a threatening enemy attacks the NPC while he is returning the wood to the return point, the woodcutter drops the wood and flees to a point of safety. Once safe, he searches for the wood he dropped, picks it up, and returns it to the return point. Finally, when the NPC gives a quest, the player only talks to him when he is not working.

4.4.4 Enemy NPC

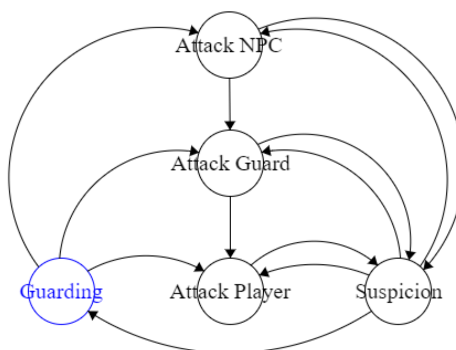


Figure 12 – Enemy's finite state machine – EnemyAIController.cs

The enemy's initial state is guarding, by either standing still or following a path. The player is the enemy's highest priority, then the guard, and lastly the NPC. For example, if the enemy attacks an NPC and a guard shows up, the enemy attacks the guard, and if the player shows up, the enemy attacks the player instead. If one of the targets disappears, the enemy

enters the suspicion state, which means that the enemy remains in the position where it last saw the target for a few seconds. If the target reappears in the meantime, the enemy attacks again, if not, it enters the guard state again

.

4.4.5 Guard NPC

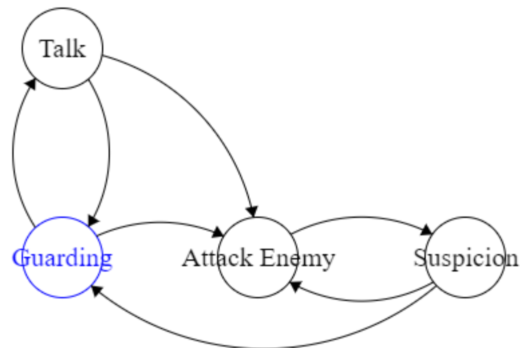


Figure 13 – Woodcutter's finite state machine – GuardController.cs

Similar to the enemy, the guard's initial state is a guarding, and if the guard has a quest, it can switch to the talk state if the player interacts with the guard. When an enemy appears, the guard chases that enemy and when the guard has taken him down, returns to the initial guard state.

5. Game Development

In this chapter I go over terrain creation, lighting and post-processing, animations, quest and dialogue system, player, stats and health, AI, locomotion and fighter, items, projectiles, inventory, UI scripts, some other scripts, and navigation.

5.1 Environment Creation

The environment was created using Unity's built-in terrain editor, which includes features for creating landscapes. Multiple terrain tiles can be created, the height and appearance of the landscape can be adjusted, and finally trees and grass can be added. At runtime, Unity optimizes the terrain rendering for more efficiency.

Also, you can create a terrain by right-clicking in the hierarchy pane, then going to 3D Objects and selecting Terrain, which will create a Terrain GameObject in the scene.

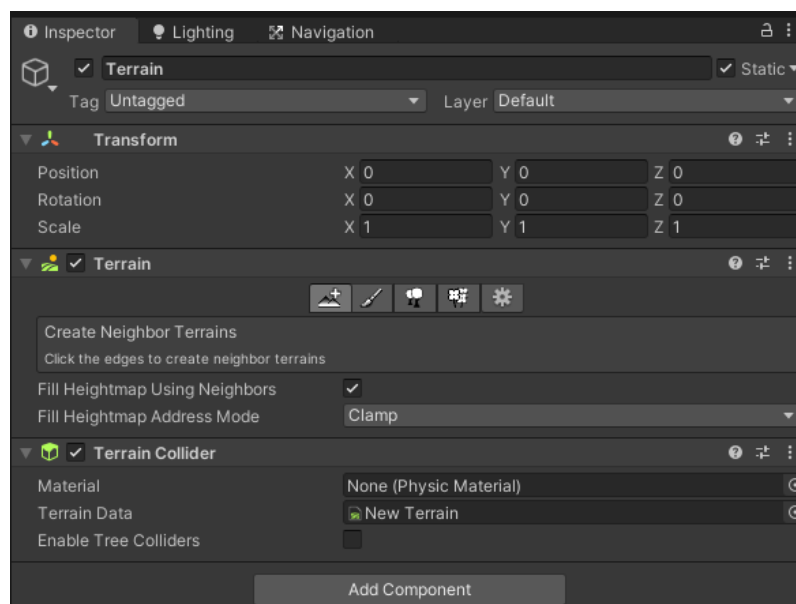


Figure 14 – A Terrain GameObject and it's components.

As Figure 14 displays, Terrain component has a toolbar that provide five options:

- Create Neighbor Terrains, used to quickly create and connect Terrains next to the selected Terrain.
- Paint Terrain, used to raise or lower the Terrain height, hides portions of the Terrain, paints surface textures, smooths height and stamps a brush shape of the current heightmap.
- Paint Trees, enables tree painting.
- Paint Details, paints grass and other details.
- Terrain Settings

The result of painting the Terrain (Figure 15) and the final result of the terrain with the rest of the details, such as buildings, trees, water, etc.) (Figure 16).

Figure 15 – A top-down view of the Terrain.

Figure 16 – A top-down view of the final result of the Scene.

To make the environment look realistic and beautiful, a number of 20 textures were used (Figure 17) and to create paths/roads quickly and easily, the Path Painter 2 painting system was used. It is important to note that Paint Details was not used and for tree placement the Tree Paint option was not used, instead trees were placed individually. Tree Paint was used to place grasses and bushes (Figure 18). A close-up of the final result (Figure 19)

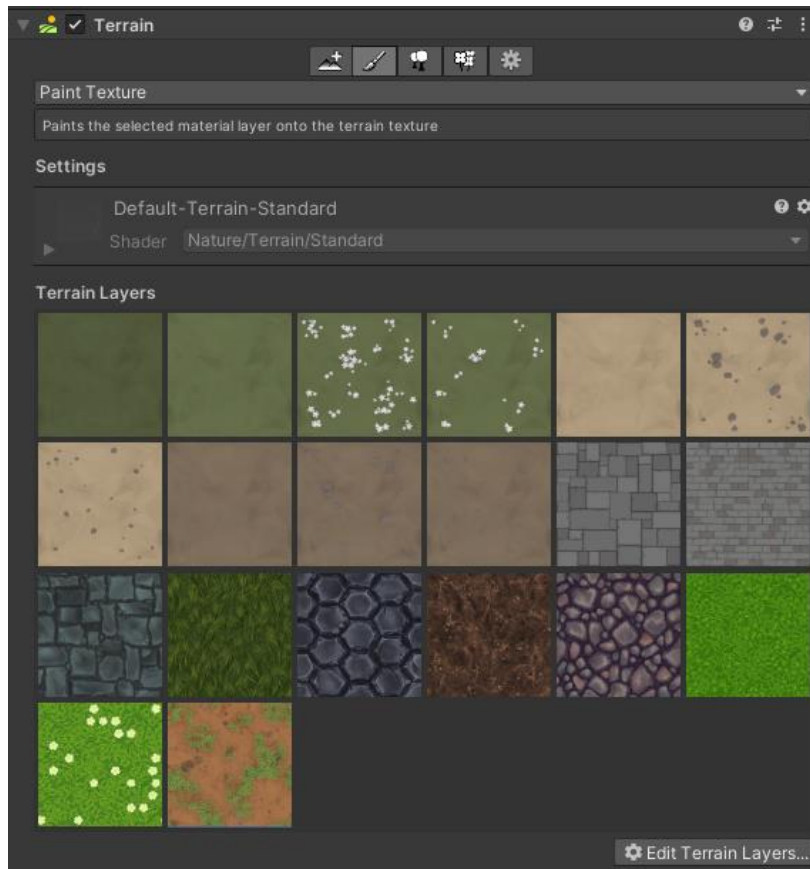


Figure 17 – Textures in Paint Texture.

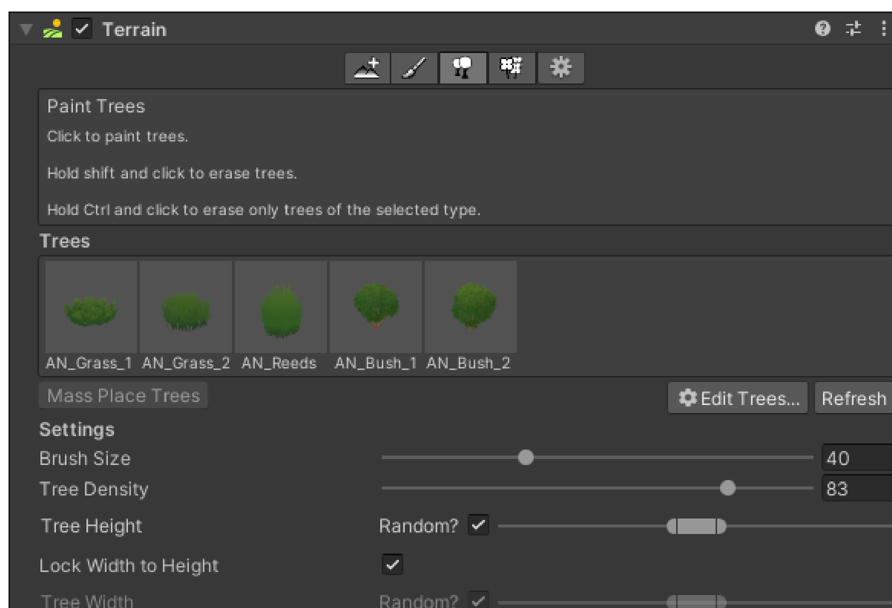


Figure 18 – Bushes and grasses that were used.

Figure 19 – Screenshot of the first village

As for Terrain Settings, it's important to enable Draw Instanced because Unity will then convert all heavy terrain data, such as height maps and splat maps, into textures on the GPU. Instead of constructing a separate mesh for each terrain patch on CPU. This reduces the workload of the terrain CPU by orders of magnitude, as a few instanced draw calls potentially replace thousands of custom mesh draws.

5.2 Lighting & Post-processing

First off starting with Lighting window by setting the lighting mode to Baked Indirect, after that in Hierarchy window, the scene's Directional Light GameObject is set to Directional type and mode set to Mixed (Figure 20 & Figure 21)

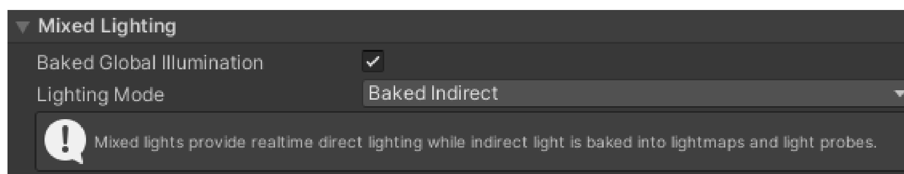


Figure 20 – Mixed Lighting in Lighting Window.

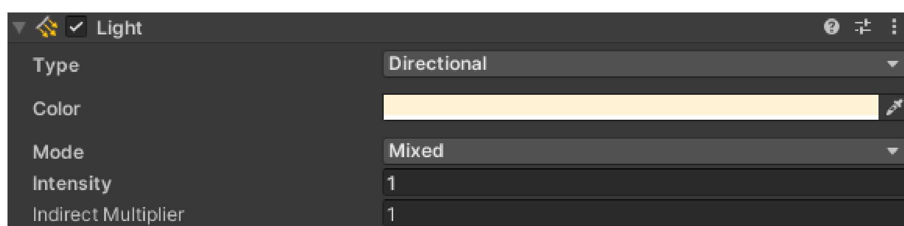


Figure 21 – Light component of Directional Light GameObject.

With Baked Indirect mode, Mixed Lights behave like Realtime Lights (Unity performs lighting calculations once per frame, it's useful for casting shadows on characters or moveable geometry), with the additional benefit of baking indirect lighting into lightmaps (pre-rendered textures that contain effects of light sources on static objects in the scene). In Directional lights the light source will behave in many ways like the sun, directional light can be thought of as a distant light source which exist infinitely far away. All objects in the scene are illuminated as if the light is always from the same direction. Lastly, Lightmapping is the process of pre-calculating the brightness of surfaces in a scene and storing the result in a texture called a lightmap, and in this project is used the Progressive Lightmapper. The Progressive Lightmapper is a fast path-tracing-based lightmapper system that provides baked lightmaps and Light Probes with progressive updates in the Editor. It requires non-overlapping UVs with small area and angle errors, and sufficient padding between the charts.

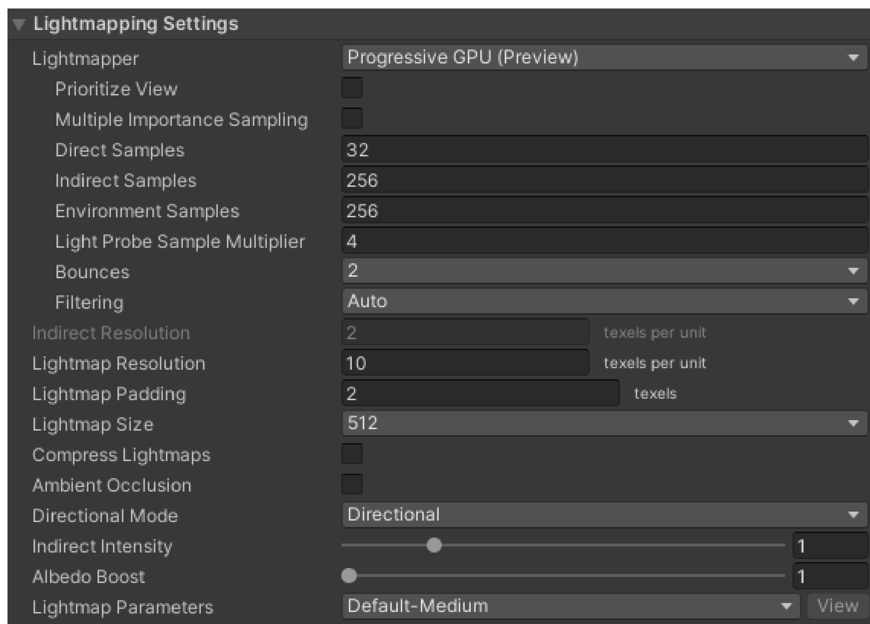


Figure 22 – Lightmapping Settings

The following figure (Figure 23) shows how the scene looks before the lightmap data is generated. The next figure (Figure 24) is the result of the generated lightmap data of the scene with the settings from Figure 22.

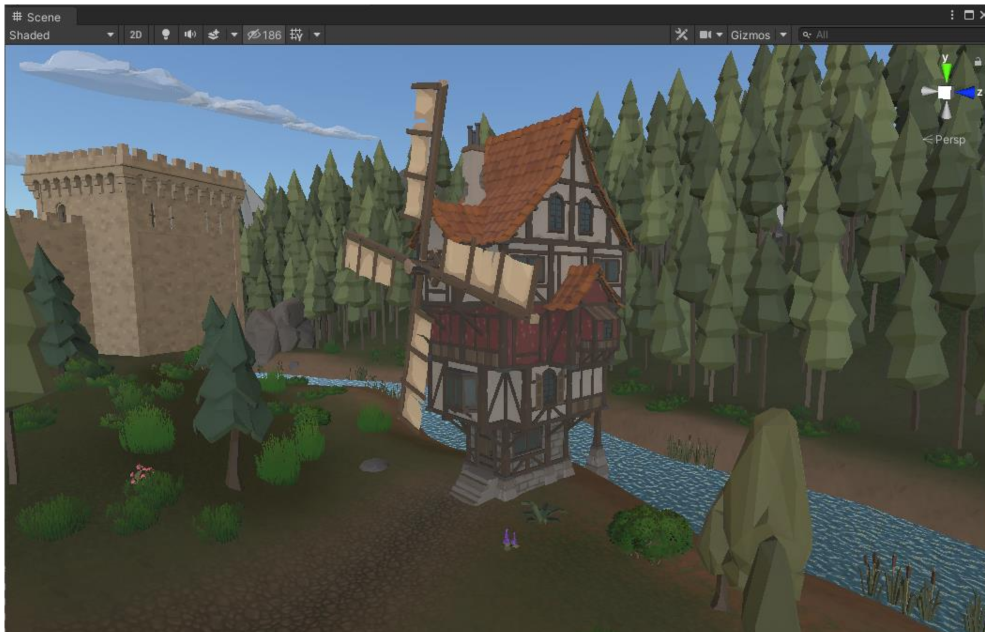


Figure 23 – Baked lights & post process off

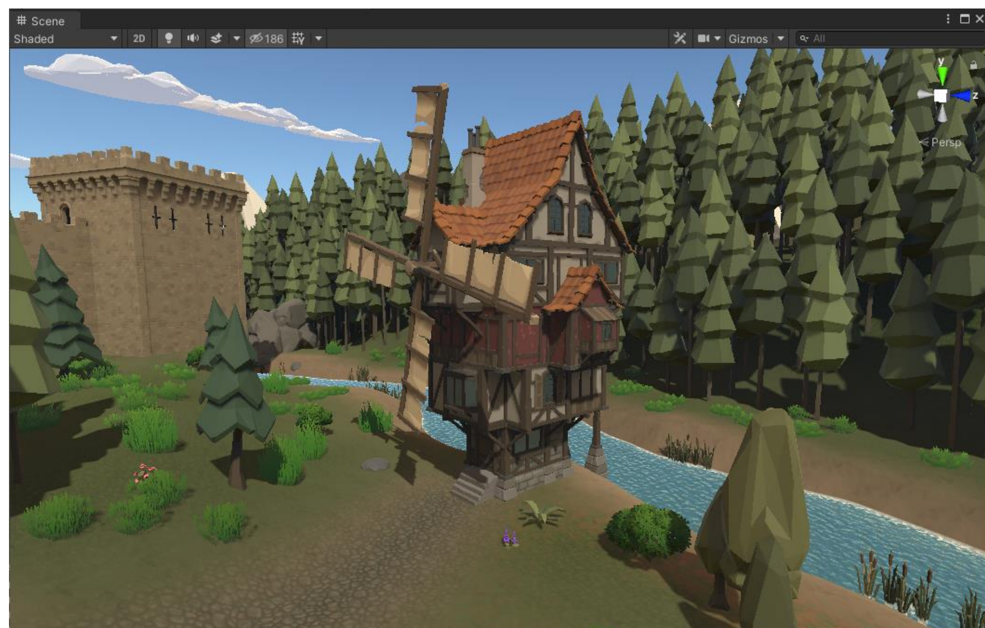


Figure 24 – Baked lights on & post process off

Now moving to the post-processing of the project, we must first understand what post-processing is. Instead of rendering 3D objects directly to the screen, the scene is first rendered to a buffer in the graphics card's memory. Pixel shaders and optionally vertex shaders are then used to apply post-processing filters to the image buffer before it is displayed on the screen.

Unity offers a number of post-processing effects and full screen effects. First, I created a new layer called "Post-Processing" and added a Post-Process Layer to the Camera GameObject and set the layer to "Post-Processing" so that post-processing would be applied to it.

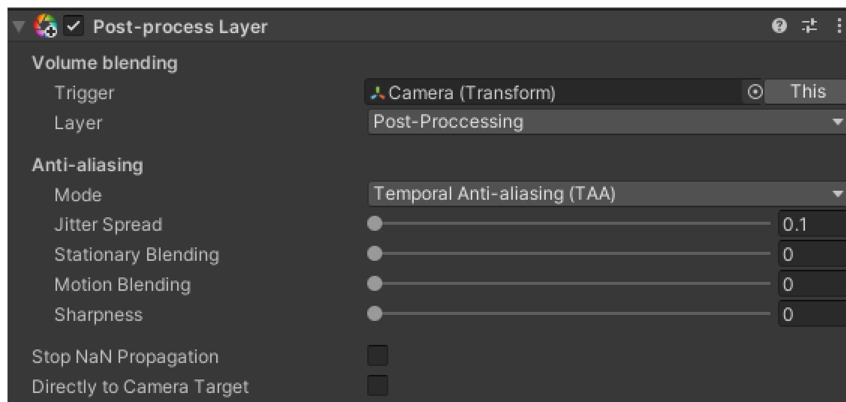


Figure 25 – Camera GameObject's Post-process Layer component

Then create an empty GameObject named Post-Processing, set the layer to "Post-Processing" and add the Post-process Volume component to add the effects.

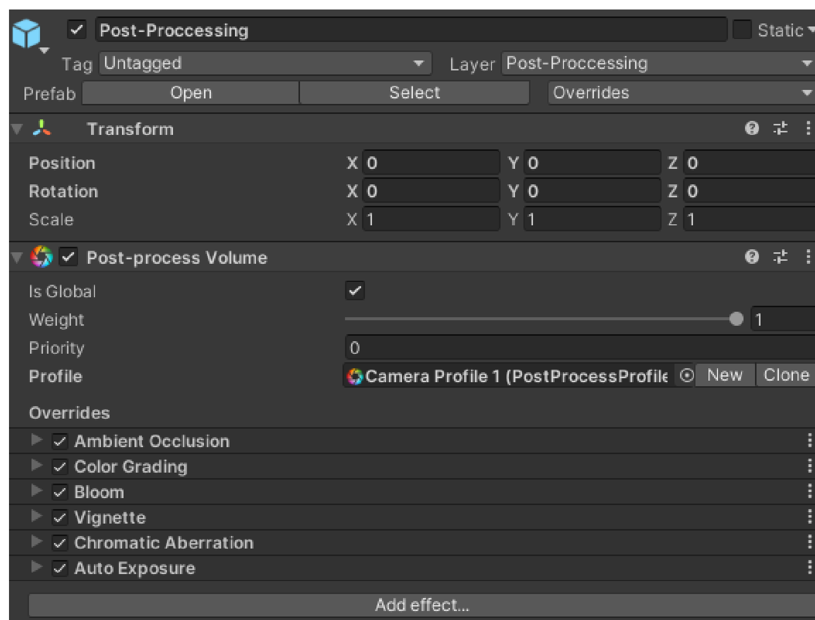


Figure 26 – Camera GameObject's Post-process Layer component

Effects that were used:

- Ambient Occlusion effect, affects the areas that are not exposed to ambient lighting and darkens them.
- Color Grading, allows to change the visual appearance by adjusting the balance of each color.
- Bloom effect. Makes bright areas in camera view.
- Vignette effect, darkens the edges of the camera view.

- Chromatic Aberration effect, spreads colors along the boundaries between dark and light areas of the camera view.
- Auto Exposure effect, dynamically adjusts the exposure according to ambient lighting.

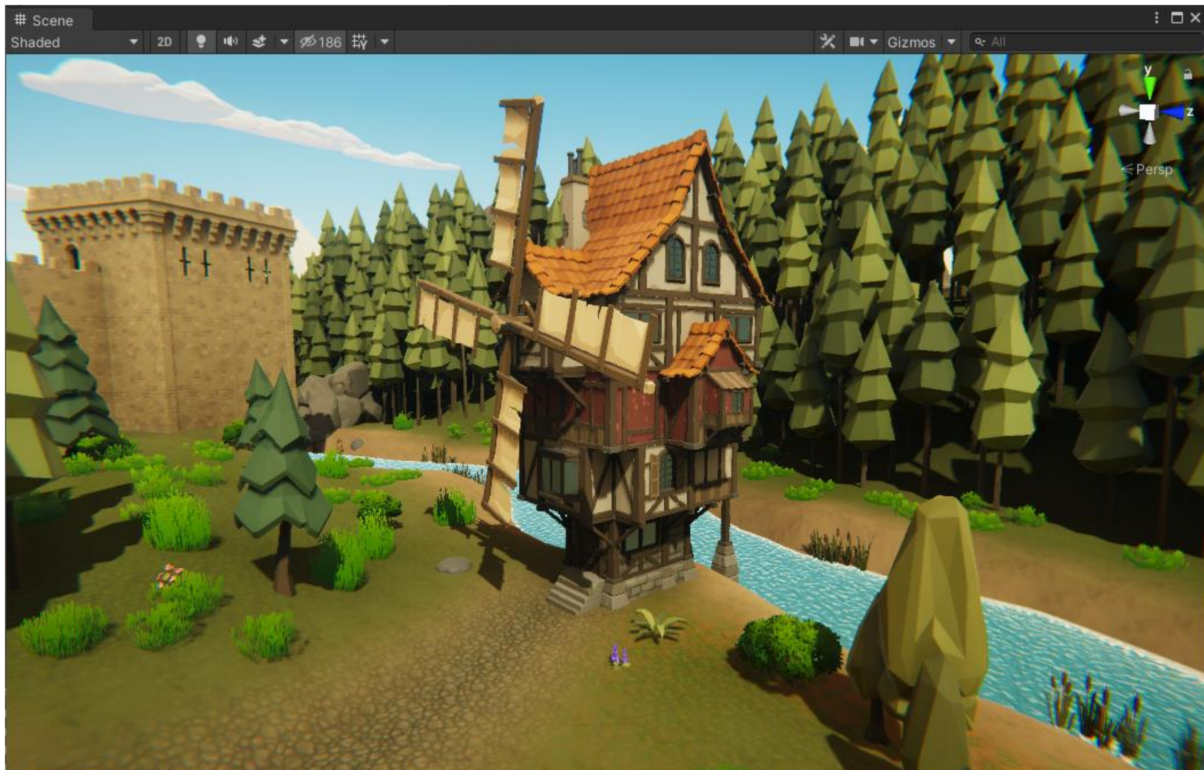


Figure 27 – Baked Lights & post process on

5.3 Animationor Controllers

Unity has an extensive and advanced animation system and is sometimes referred to as "Mecanim". It provides easy workflow and animation setup for all elements of Unity, management of complex interactions between animations with a visual programming tool, convenient preview of animation clips, retargeting of humanoid animations, and animation of different body parts with different logic.

The player's animator is simple (Figure 28) and includes Blend Tree called Locomotion (Figure 29), which manages motion animations for smooth transitions based on speed. For example, when the player is moving, the animator controller is in the Locomotion state. Based on the forwardSpeed parameter, which refers to the player's speed, the idle animation will blend with the walking animation. When attacking, triggers the attack parameter to trigger the animation. When the animation reaches its end time, stopAttack is triggered and the player goes back to the Locomotion state. The roll works just like the attack, but has its own parameters (roll & stopRoll). When the player dies, the die parameter is triggered and the player goes from any state to the death state. For example, if the player dies while running, the transition will be from the Locomotion to the death state.

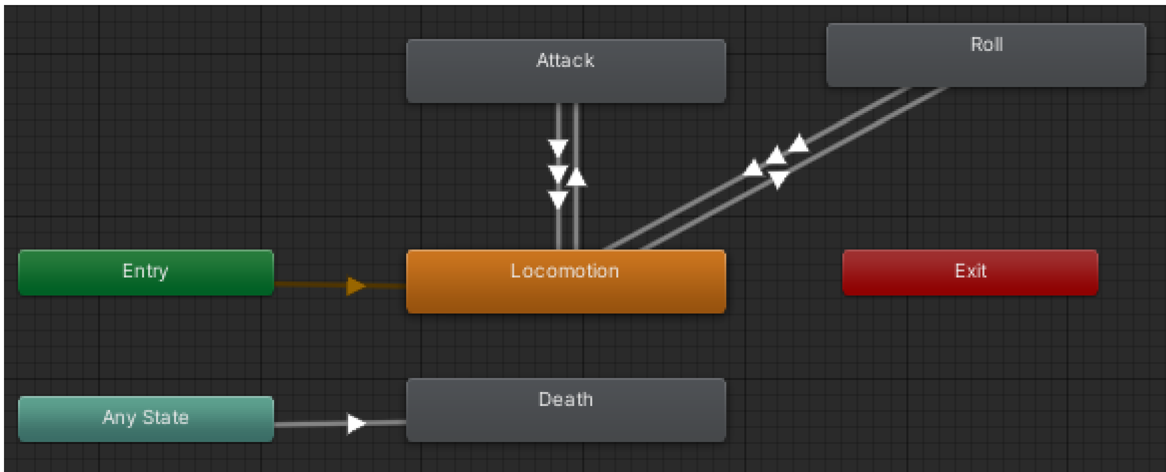


Figure 28 – Player’s Animator Controller

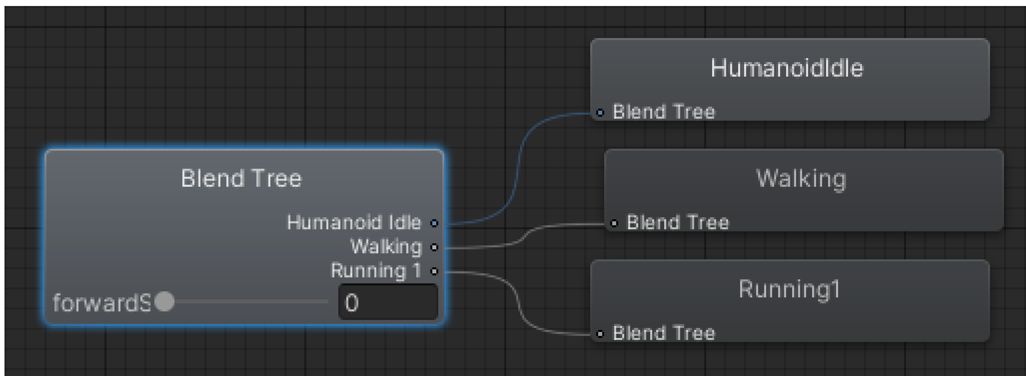


Figure 29 – Player’s “Locomotion” Blend Tree

Figure 30 – Player’s Animator Controller’s parameters

Enemies and guards have the same animator controller because they use the same animations. This animator is the same as the player's animator controller, meaning it works the same as the player's, including Locomotion Blend Tree, but does not have the Roll state.

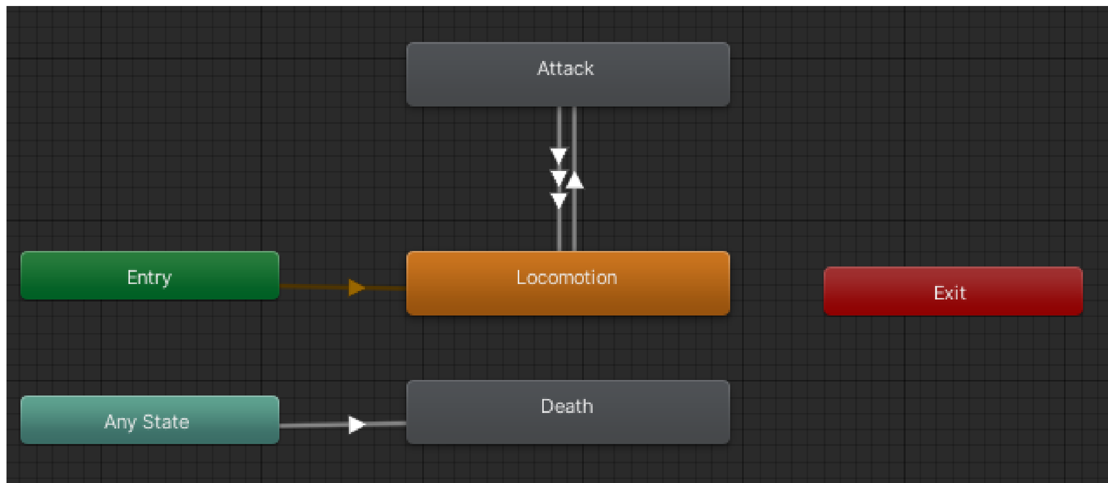


Figure 31 Enemy & Guard Animator Controller

Plain NPCs have the simplest animation controls due to their behavior. They require a locomotion blend tree and a death state.

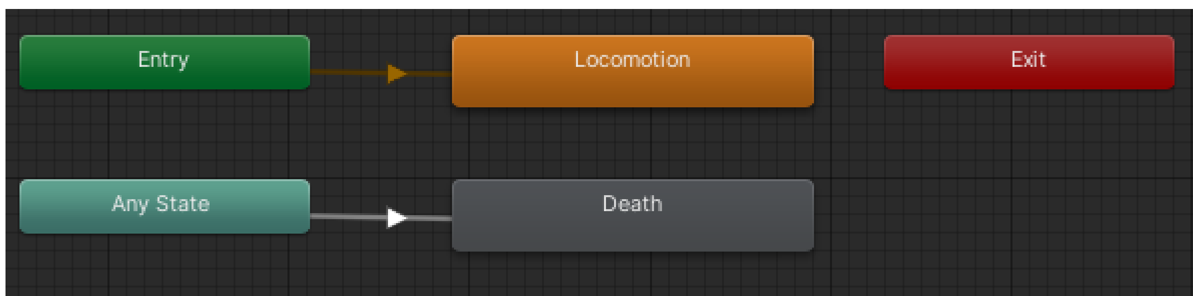


Figure 32 – Plain NPC Animator Controller

The farmer NPC's animator controller has a Collect state for the harvest animation, a death state and a Locomotion blend tree like the other animator controllers.

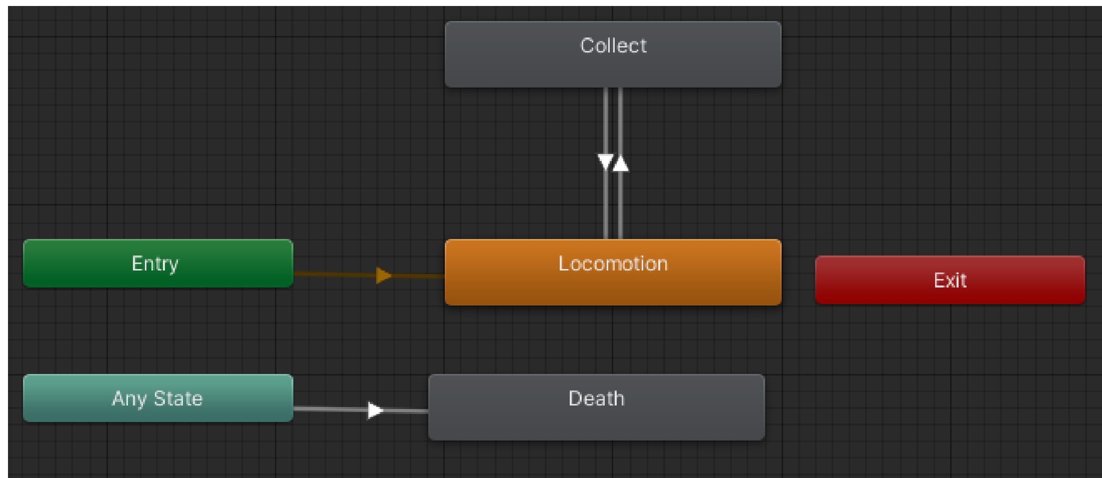


Figure 33 – Farmer NPC Animator Controller's parameters

The animator controller of the woodcutter NPC works the same as the animator controller of the farmer NPC. The only difference is that instead of the collect state, there is a chop state for the animation of chopping wood.

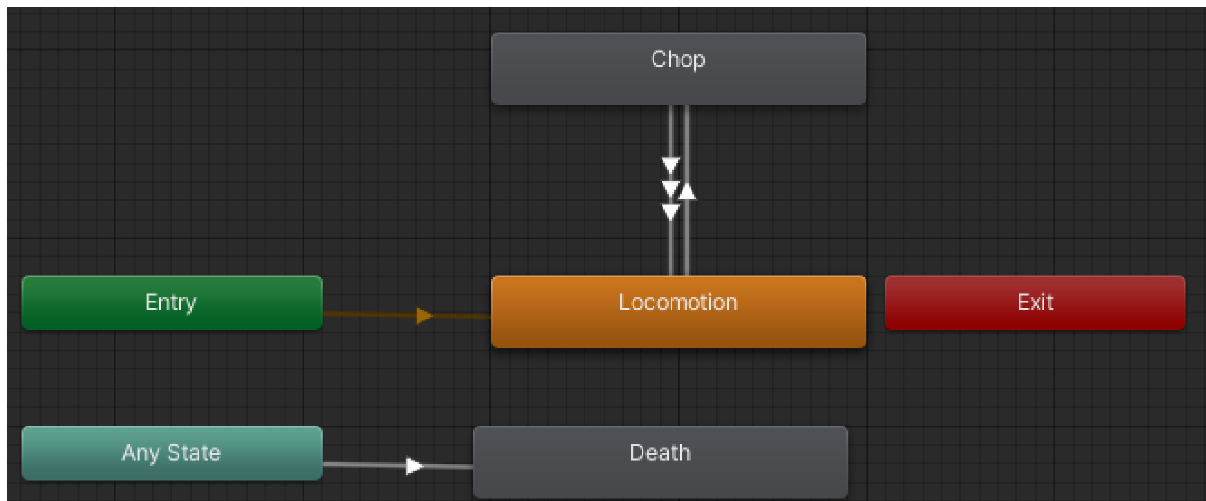


Figure 34 – Woodcutter NPC Animator Controller's parameters

Animator override controllers, which are replacing specific animations of an animator controller to create multiple variants of that controller. In this project I use them in weapons to replace some animations like the attack and the motion so that the "big" swords feel heavier when attacking and walking than the "smaller" swords (More about swords at chapter 5.9).

5.4 Quest & Dialogue System

5.4.1 Overview

The quest system is essential for Nysa's Quest, as the player must acquire quests based on the storyline in order to know where to go next, as planned. Thanks to this system, the player can have dialogues with NPCs and even acquire a quest if that NPC gives one. Besides, by pressing "Z", the player can see the accepted quests (completed and not), their information and rewards.

Figure 35 – In game screenshot of the quest list (white color for unfinished quests & green color for completed quests).

Figure 36 – In game screenshot of the Nysa being in dialogue with an NPC that gives quest.

5.4.2 Code

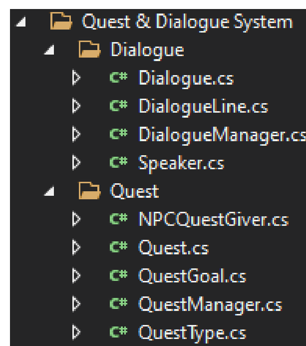


Figure 37 – Quest & Dialogue System folder structure.

First starting with *Speaker.cs*, this is a scriptable object that is going to contain information of the speaker, such as name and a sprite image.

```
using UnityEngine;

namespace ARPG.QuestDialogue
{
    [CreateAssetMenu(fileName = "New Speaker", menuName = "Dialogue/New Speaker")]
    Unity Script | 5 references
    public class Speaker : ScriptableObject
    {
        [SerializeField] private string speakerName = null;         //Defines the speaker's name
        [SerializeField] private Sprite speakerSprite = null;       //Defines the speakers sprite image

        private bool isInDialogue = false;                          //True when in dialogue, false when not in dialogue

        //Getter functions
        1 reference
        public string GetSpeakerName()
        {
            return speakerName;
        }

        1 reference
        public Sprite GetSpeakerSprite()
        {
            return speakerSprite;
        }

        17 references
        public bool GetIsInDialogue()
        {
            return isInDialogue;
        }

        //Setter function
        3 references
        public void SetIsInDialogue(bool value)
        {
            isInDialogue = value;
        }
    }
}
```

Figure 38 – Speaker.cs

Next up is the *DialogueLine.cs*, this script is used to contain a single dialogue line, containing the speaker's information and their line. This DialogueLine class is Serializable, that means its fields are going to be on the inspector and it is used on the next script *Dialogue.cs*.

```
using UnityEngine;

namespace ARPG.QuestDialogue
{
    //This DialogueLine class is a single dialogue line that its attributes are the speaker and the dialogue line

    [System.Serializable]
    5 references
    public class DialogueLine
    {
        public Speaker speaker;        //Defines the speaker, by that it means that speaker has a name and a sprite image
        [TextArea]
        public string dialogueLine;    //Defines the dialogue line of the speaker
    }
}
```

Figure 39 – DialogueLine.cs

Dialogue.cs is a scriptable object and it is used to create dialogues with multiple dialogue lines.

```
using UnityEngine;

namespace ARPG.QuestDialogue
{
    [CreateAssetMenu(fileName = "New Dialogue", menuName = "Dialogue/New Dialogue")]
    Unity Script | 11 references
    public class Dialogue : ScriptableObject
    {
        [SerializeField] private DialogueLine[] allLines;       //Defines the dialogue lines in this Dialoggue
        [SerializeField] private bool givesQuest = false;       //Defines if the dialogue gives quest or not
        [SerializeField] private Quest quest = null;            //Defines the quest

        //Getter functions
        2 references
        public DialogueLine[] GetDialogueLine()
        {
            return allLines;
        }

        4 references
        public DialogueLine GetLineByIndex(int index)
        {
            return allLines[index];
        }

        2 references
        public int GetLength()
        {
            return allLines.Length - 1;
        }

        3 references
        public Quest GetQuest()
        {
            return quest;
        }

        2 references
        public bool GetGivesQuest()
        {
            return givesQuest;
        }

        //Setter function
        0 references
        public void SetGivesQuest(bool value)
        {
            givesQuest = value;
        }
    }
}
```

Figure 40 – Dialogue.cs

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using TMPro;

namespace ARPG.QuestDialogue
{
    Unity Script | 12 references
    public class DialogueManager : MonoBehaviour
    {
        public TextMeshProUGUI speakerNameUI, dialogueUI, navButtonText;    //Assigns the UI elements of the dialogue box
        public Image speakerSprite;                                          //Assigns the image of the speaker

        private int currentIndex;                  //currentIndex refers to the index of the current DialogueLine of the Dialogue
        private Dialogue currentDialogue;          //current Dialogue refers to the whole Dialogue
        private static DialogueManager instance;   //This DialogueManager instance exists to make sure that only 1 instance exists in the scene
        private Animator animator;                 //animator refers to the Animator to use the Dialogue Box's animations
        private Coroutine typing;                  //typing is a Coroutine that makes sure to show each Dialogue Line correctly

        private GameObject npcSpeaker;

        //On Awake makes sure that only 1 instance of the DialogueManager exists, else deletes itself
        Unity Message | 0 references
        private void Awake()
        {
            if (instance == null)
            {
                instance = this;
                animator = GetComponent<Animator>();
                instance.DisableBox();
                instance.npcSpeaker = null;
            }
            else
            {
                Destroy(gameObject);
            }
        }

        //Function that gets called when having dialogue with an NPC
        4 references
        public static void StartDialogue(Dialogue givenDialogue, GameObject npcSpeaker)
        {
            StartDialogue(givenDialogue);
            instance.npcSpeaker = npcSpeaker;
        }
```

Figure 41 – DialogueManager.cs part 1

```
        //Function that Resets and Prepares the Dialogue Box and the DialogueLine based on the givenDialogue.
        4 references
        public static void StartDialogue(Dialogue givenDialogue)
        {
            instance.EnableBox();                               //Enables the dialogue UI
            instance.animator.SetBool("isOpen", true);          //Changes the animator parameter so the box can come up
            instance.currentIndex = 0;                          //Sets the current index to 0
            instance.currentDialogue = givenDialogue;           //Sets the Dialogue to the givenDialogue
            instance.speakerNameUI.text = "";                   //Resets the speaker's name
            instance.dialogueUI.text = "";                      //Reset the dialogue line text
            instance.navButtonText.text = ">";                  //Sets button text to ">" that means that there is atleast 1 more line next

            foreach (DialogueLine dl in instance.currentDialogue.GetDialogueLine())     //When the dialogue starts, finds the dialogue's speakers and sets isInDialogue to true
            {
                if (!dl.speaker.GetIsInDialogue())
                {
                    dl.speaker.SetIsInDialogue(true);
                }
            }

            instance.ReadNext();        //Calls ReadNext()
        }

        //Function that stops the dialogue
        4 references
        public static void StopDialogue()
        {
            instance.typing = null;
            instance.StartCoroutine(instance.CloseBox());
        }
```

Figure 42 – DialogueManager.cs part 2

```
//Function that goes through every dialogue line and displays it
1 reference
public void ReadNext()
{
    if (currentIndex > currentDialogue.GetLength())     //If the current index is the last DialogueLine of the Dialogue then Dialogue ends
    {
        instance.StartCoroutine(CloseBox());
        return;
    }
    speakerNameUI.text = currentDialogue.GetLineByIndex(currentIndex).speaker.GetSpeakerName();     //Sets the speaker's name

    if (typing == null)                                                                             //if typing is null then starts a coroutine of typing the current DialogueLine
    {
        typing = instance.StartCoroutine(TypeText(currentDialogue.GetLineByIndex(currentIndex).dialogueLine));
    }
    else                                                                                            //Else if it is already typing then stop typing and start typing the current DialogueLine
    {
        instance.StopCoroutine(typing);
        typing = null;
        typing = instance.StartCoroutine(TypeText(currentDialogue.GetLineByIndex(currentIndex).dialogueLine));
    }

    speakerSprite.sprite = currentDialogue.GetLineByIndex(currentIndex).speaker.GetSpeakerSprite(); //Sets the speakerSprite to the speaker of the DialogueLine
    currentIndex++;

    if (currentIndex >= currentDialogue.GetLength() + 1)        //If the curerntIndex is the last then change the text of the button to "X"
    {
        if (instance.currentDialogue.GetGivesQuest())           //And if the Dialogue gives Quest then add that Quest to the QuestManager and set Dialogue to not give again that Quest
        {
            QuestManager.AddQuest(currentDialogue.GetQuest());
            if (instance.npcSpeaker != null)
            {
                instance.npcSpeaker.GetComponent<NPCQuestGiver>().HideCanvas();
            }
        }

        navButtonText.text = "X";
    }
}
```

Figure 43 – DialogueManager.cs part 3

```
//IEnumerator that closes the dialogue box smoothly and sets the speaker's isInDialogue to false
2 references
private IEnumerator CloseBox()
{
    instance.npcSpeaker = null;
    instance.animator.SetBool("isOpen", false);
    foreach (DialogueLine dl in instance.currentDialogue.GetDialogueLine())
    {
        if (dl.speaker.GetIsInDialogue())
        {
            dl.speaker.SetIsInDialogue(false);
        }
    }
    yield return new WaitUntil(() => animator.GetCurrentAnimatorStateInfo(0).IsName("DialogueBoxClosing"));
    instance.DisableBox();
}

//Function that enables the dialogue box
1 reference
private void EnableBox()
{
    gameObject.SetActive(true);
}

//Function that disables the dialogue box
2 references
private void DisableBox()
{
    gameObject.SetActive(false);
}

//This IEnumerator is resposnible for typing 1 letter of a DialogueLine per 0.02 seconds
2 references
private IEnumerator TypeText(string text)
{
    dialogueUI.text = "";
    bool complete = false;
    int index = 0;

    while (!complete)
    {
        dialogueUI.text += text[index];
        index++;
        yield return new WaitForSeconds(0.02f);

        if (index == text.Length)
        {
            complete = true;
        }
    }

    typing = null;
}
}
}
```

Figure 44 – DialogueManager.cs part 4

Now moving to the quests by starting with *Quest.cs*, it's a serializable class with fields that are going to contain the information of a quest (title, description, experience reward, if its completed and the quest goal).

```
namespace ARPG.QuestDialogue
{
    [System.Serializable]
    11 references
    public class Quest
    {
        public string title = null;             //Defines the quest title
        public string description = null;       //Defines the quest description
        public int experienceReward = 0;        //Defines the quest experience reward
        public bool isCompleted = false;        //Defines if the quest is completed or not
        public QuestGoal questGoal;             //Defines the goal of the quest
    }
}
```

Figure 45 – Quest.cs

QuestGoal.cs is a serializable script that is used to set whether the quest is about killing enemies or collecting items and checking the progress.

```
using ARPG.AI;
using ARPG.Resources.Items;
using UnityEngine;

namespace ARPG.QuestDialogue
{
    [System.Serializable]
    1 reference
    public class QuestGoal
    {
        public QuestType questType;     //Sets the quest type
        public QuestEnemy enemyType;    //Enemy type for when the quest type is "Kill"
        public Item item;               //Item for when the quest type is "Gather"
        public int requiredAmount;      //Required amount of kills or items gathered to complete the quest
        public int currentAmount;       //Current amount of kills or gathered items

        //Function that checks if the quest goal is reached
        2 references
        public bool IsGoalReached()
        {
            //If the current amount is equals or greater than the required amount then returns true
            return (currentAmount >= requiredAmount);
        }

        //Function that checks if the killedEnemy is the enemy that
        //the quest requires so that it can increase the currentAmount
        1 reference
        public void EnemyKilled(GameObject killedEnemy)
        {
            QuestEnemy killedEnemyType = killedEnemy.GetComponent<EnemyAIController>().GetEnemyType();
            if (questType == QuestType.Kill && enemyType == killedEnemyType)
            {
                currentAmount++;
            }
        }

        //Function that checks if the pickedItem is the same item as
        //the quest requires to increase the currentAmount
        1 reference
        public void ItemGathered(Item pickedItem)
        {
            if (questType == QuestType.Gather && item == pickedItem)
            {
                currentAmount++;
            }
        }
    }
}
```

Figure 46– QuestGoal.cs

QuestType.cs is an enum used to set the quest type in *QuestGoal.cs*.

```
namespace ARPG.QuestDialogue
{
    3 references
    public enum QuestType
    {
        Kill,
        Gather
    }
}
```

Figure 47 – QuestType.cs

QuestEnemy.cs is an enum used on *QuestType.cs* to set what enemies need to be killed for a quest.

```
public enum QuestEnemy
{
    None,
    MainQuest0,
    MainQuest1,
    MainQuest2,
    MainQuest3,
    SideQuest1,
    SideQuest2,
    SideQuest3
}
```

Figure 48 – QuestEnemy.cs

NPCQuestGiver.cs is added to the NPCs and is used so the player can identify if an NPC gives quest or not, that means if an NPC gives quest (We are going to see how an NPC gives quest on chapter …) there is going to be an exclamation mark image above it, else there is not.

```
using ARPG.AI;
using UnityEngine;
using UnityEngine.UI;

namespace ARPG.QuestDialogue
{
    Unity Script | 1 reference
    public class NPCQuestGiver : MonoBehaviour
    {
        private DialogueNPC npc;
        private Dialogue dialogue;
        private Canvas canvas;
        private Image image;
        private bool show;

        Unity Message | 0 references
        void Awake()
        {
            npc = GetComponent<DialogueNPC>();
            if (npc.GetDialogue() != null)
            {
                dialogue = npc.GetDialogue();
            }
            canvas = GetComponentInChildren<Canvas>();
            canvas.enabled = false;
            image = canvas.GetComponentInChildren<Image>();
        }

        Unity Message | 0 references
        private void Start()
        {
            //If NPC has dialogue that gives quest then call ShowCanvas()
            if (npc.GetDialogue() != null && dialogue.GetGivesQuest())
            {
                ShowCanvas();
            }
            else    //If not then call HideCanvas()
            {
                HideCanvas();
            }
        }
```

Figure 49 – NPCQuestGiver.cs part 1

```
        Unity Message | 0 references
        private void Update()
        {
            if (show)   //If show is true then the image faces the camera
            {
                image.transform.rotation = Camera.main.transform.rotation;
            }
        }

        //Function that hides the canvas and the image
        2 references
        public void HideCanvas()
        {
            image.enabled = false;
            canvas.enabled = false;
            show = false;
        }

        //Function that enables the canvas and the image
        1 reference
        private void ShowCanvas()
        {
            canvas.enabled = true;
            image.enabled = true;
            show = true;
        }
    }
}
```

Figure 50 – NPCQuestGiver.cs part 2

QuestManager.cs is used to display the quests and their descriptions and show if they are completed. Also, for performance reasons all the GameObjects of the enemies are disabled, but when acquiring a quest then that quest's enemies are being enabled.

```
using ARPG.Control;
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

namespace ARPG.QuestDialogue
{
    Unity Script | 3 references
    public class QuestManager : MonoBehaviour
    {
        [Header("UI Elements")]
        //Defines the UI element that contains the rest of the UI components of the quest
        public GameObject QuestManagerBoxPanel;
        //Defines the quest list container
        public GameObject questListContainer;
        //Defines the quest's info texts
        public TextMeshProUGUI questTitleUI, questDescriptionUI, questRewardUI;
        //Defines the button prefab that the quest title will be fitted into and placed in the questListContainer
        public GameObject button;
        [Header("Ingame Elements")]
        //Defines the list of quests and their enemy parent gameObject
        public QuestEnemiesConnector[] questEnemiesConnectorList;
        [Header("Story Quests Setup")]
        public Dialogue prologue;
        public Dialogue epilogue;
        public Dialogue[] mainQuests;

        //PlayerController to know if inventory ui is open so that only 1 ui is open at the time
        private PlayerController player;
        //This QuestManager instance exists to make sure that only 1 instance exists in the scene
        private static QuestManager instance;
        //List of Quests that contains quests player has & has not completed
        private List<Quest> questsList = new List<Quest>();
        //List of buttons that's for each quest and are placed in  to questListContainer
        private List<Button> questButton = new List<Button>();
        private bool isUIEnabled = false;
        private bool completion = false;
```

Figure 51 – QuestManager.cs part 1

```
//On Awake makes sure that only 1 instance of the QuestManager exists, else deletes itself
Unity Message | 0 references
private void Awake()
{
    if (instance == null)
    {
        instance = this;
        instance.questTitleUI.text = "";
        instance.questDescriptionUI.text = "";
        instance.questRewardUI.text = "";
        instance.QuestManagerBoxPanel.SetActive(false);
        instance.player = GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerController>();
        //DialogueManager.StartDialogue(instance.prologue);
    }
    else
    {
        Destroy(gameObject);
    }
}

Unity Message | 0 references
private void Start()
{
    DialogueManager.StartDialogue(instance.prologue);
}

Unity Message | 0 references
void Update()
{
    if (Input.GetKeyDown(KeyCode.Z) && !player.OnInventory())
    {
        if (instance.QuestManagerBoxPanel.activeSelf)
        {
            instance.QuestManagerBoxPanel.SetActive(false);
            isUIEnabled = false;
        }
        else
        {
            instance.UpdateCompletedQuests();
            instance.QuestManagerBoxPanel.SetActive(true);
            isUIEnabled = true;
        }
    }
}
```

Figure 52 – QuestManager.cs part 2

```
//Used to reset the progress of the quests due to sometimes
//changes are being saved even after restarting Unity
Unity Message | 0 references
private void OnDisable()
{
    foreach (Quest quest in questsList)
    {
        quest.isCompleted = false;
        quest.questGoal.currentAmount = 0;
    }
}

//Function that adds the quest to the UI and assigns the ShowQuestInfo
1 reference
public static void AddQuest(Quest givenQuest)
{
    foreach(Quest quest in instance.questsList)
    {
        if (quest.title == givenQuest.title) return;
    }

    instance.questsList.Add(givenQuest);

    var instantiated = Instantiate(instance.button, instance.questListContainer.transform.position, Quaternion.identity);
    instantiated.GetComponent<RectTransform>().SetParent(instance.questListContainer.transform);
    instantiated.transform.GetChild(0).GetComponent<TextMeshProUGUI>().text = givenQuest.title;
    instance.questButton.Add(instantiated.GetComponent<Button>());
    int index = instance.questButton.Count - 1;
    instance.questButton[index].onClick.AddListener(() => { instance.ShowQuestInfo(index); });

    foreach(QuestEnemiesConnector item in instance.questEnemiesConnectorList)
    {
        if(item.questDialogue.GetQuest().title == givenQuest.title)
        {
            item.enemiesParentGameObject.SetActive(true);
            break;
        }
    }
}

//Function that shows the info of the clicked quest
1 reference
private void ShowQuestInfo(int index)
{
    instance.questTitleUI.text = instance.questsList[index].title;
    instance.questDescriptionUI.text = instance.questsList[index].description;
    instance.questRewardUI.text = "Rewards " + instance.questsList[index].experienceReward.ToString() + " XP";
}
```

Figure 53 – QuestManager.cs part 3

```
//Function that sets completed quests of the questListContainer to green color
1 reference
private void UpdateCompletedQuests()
{
    int index = 0;
    foreach (Quest quest in questsList)
    {
        if (quest.isCompleted)
        {
            var colors = questButton[index].colors;
            colors.normalColor = Color.green;
            questButton[index].colors = colors;
        }
        index++;
    }
}

2 references
public void CheckCompletion()
{
    if (instance.completion) return;
    for (int i = 0; i < instance.mainQuests.Length; i++)
    {
        if (!instance.mainQuests[i].GetQuest().isCompleted) return;
    }
    StartCoroutine(StartEpilogueDelayed());
}

1 reference
private IEnumerator StartEpilogueDelayed()
{
    yield return new WaitForSeconds(5f);
    instance.completion = true;
    DialogueManager.StartDialogue(instance.epilogue);
}
```

Figure 54 – QuestManager.cs part 4

```
    2 references
    public List<Quest> GetQuestsList()
    {
        return questsList;
    }

    2 references
    public bool GetIsQuestBoxOpened()
    {
        return isUIEnabled;
    }
}

[System.Serializable]
2 references
public class QuestEnemiesConnector
{
    public Dialogue questDialogue;
    public GameObject enemiesParentGameObject;
}
}
```

Figure 55 – QuestManager.cs part 5

5.5 Player

5.5.1 Overview

The player control consists of 3 scripts, *PlayerController.cs*, *PlayerMovement.cs* and *PlayerCombat.cs*. These 3 scripts are responsible for everything the player does, i.e. movement, combat, having the inventory opened, being on pause menu, dialogues, just about everything. *PlayerController.cs* and *PlayerMovement.cs* also inherit from IAction and use *ActionScheduler.cs*, which I'll go over at the end.

5.5.2 Code

Starting with *PlayerController.cs*. This is the most important script, because it checks if the player is in the menu, in dialogue, in the inventory, or checking the quests or rolling, and finally if in combat. It is important to check the above states because the player's behavior will be adjusted based on this state.

```
using UnityEngine;
using ARPG.Movement;
using ARPG.Core;
using ARPG.Resources;
using ARPG.Stats;
using ARPG.Resources.Items;
using ARPG.QuestDialogue;
using ARPG.Other;
using ARPG.Combat;
using ARPG.AI;

namespace ARPG.Control
{
    Unity Script | 15 references
    public class PlayerController : MonoBehaviour
    {
        [SerializeField] Speaker speaker;       //Reference to the player's speaker
        [SerializeField] QuestManager qm;       //Reference to the QuestManager instance
        [SerializeField] EscapeMenu escapeMenu; //Reference to EscapeMenu

        private Health health;                          //Player's health
        private Mover mover;                            //Player's mover
        private PlayerMovement playerMovement;          //Reference to playerMovement
        private PlayerCombat playerCombat;              //Reference to PlayerCombat
        private Inventory inventory;                    //Player's inventory
        private Animator animator;                      //Player's animator controller

        Unity Message | 0 references
        private void Awake()
        {
            health = GetComponent<Health>();
            mover = GetComponent<Mover>();
            playerMovement = GetComponent<PlayerMovement>();
            playerCombat = GetComponent<PlayerCombat>();
            inventory = GetComponent<Inventory>();
            animator = GetComponent<Animator>();
            speaker.SetIsInDialogue(false);
        }
```

Figure 56 – PlayerController.cs part 1

```
Unity Message | 0 references
private void Update()
{
    if (health.IsDead())    //If player is dead then do nothing
    {
        return;
    }
    else
    {
        if (onMenu()) return;
        else if (InDialogue()) return;
        else if (OnInventory()) return;
        else if (OnQuestBox()) return;
        else if (IsRolling()) return;
        else if (IsInCombat()) return;
    }
}

//If player is in combat then stops moving.
1 reference
private bool IsInCombat()
{
    if (playerCombat.GetIsAttacking())
    {
        playerMovement.setSuspend(true);
        return true;
    }
    playerMovement.setSuspend(false);
    return true;
}

//If player opens the escape menu then game pauses and player can not move.
//Returns true if menu is open.
//Returns false if menu is closed.
1 reference
private bool onMenu()
{
    if (escapeMenu.escapeMenuPanel.activeSelf)
    {
        playerCombat.setSuspend(true);
        playerMovement.setSuspend(true);
        return true;
    }
    playerMovement.setSuspend(false);
    playerCombat.setSuspend(false);
    return false;
}
```

Figure 57 – PlayerController.cs part 2

```
//If player tries to engage a dialogue with an NPC that has a dialogue then starts dialogue.
//Returns true if player is in dialogue.
//Returns false if player is not in dialogue.
1 reference
private bool InDialogue()
{
    if (speaker.GetIsInDialogue())
    {
        playerCombat.setSuspend(true);
        playerMovement.setSuspend(true);
        return true;
    }
    playerMovement.setSuspend(false);
    playerCombat.setSuspend(false);
    //Information for the impact point in world space
    RaycastHit hit;
    //Casts a ray to the hit point and returns true if ray intersects with a collider
    bool hasHit = Physics.Raycast(GetMouseRay(), out hit);

    DialogueNPC npc = hit.transform.GetComponent<DialogueNPC>();
    if (hasHit && npc != null)
    {
        if (Input.GetMouseButtonDown(0))
        {
            if (Vector3.Distance(transform.position, npc.transform.position) <= 2)
            {
                npc.wantsToTalk = true;
            }
            return false;
        }
        return false;
    }
    return false;
}
```

Figure 58 – PlayerController.cs part 3

```
//If player got the inventory opened then cancels the current
//action and can not do anything else as long as inventory is opened.
//Returns true if inventory is opened.
//Returns false if inventory is  closed.
2 references
public bool OnInventory()
{
    if (inventory.GetIsInventoryOpened() && !qm.GetIsQuestBoxOpened())
    {
        playerCombat.setSuspend(true);
        playerMovement.setSuspend(true);
        GetComponent<ActionScheduler>().CancelCurrentAction();
        return true;
    }
    playerCombat.setSuspend(false);
    playerMovement.setSuspend(false);
    return false;
}

//If player got the quest box opened then cancels the current action
//and can not do anything else as long as quest box is opened.
//Returns true if inventory is opened.
//Returns false if inventory is  closed.
2 references
public bool OnQuestBox()
{
    if (qm.GetIsQuestBoxOpened() && !inventory.GetIsInventoryOpened())
    {
        playerCombat.setSuspend(true);
        playerMovement.setSuspend(true);
        GetComponent<ActionScheduler>().CancelCurrentAction();
        return true;
    }
    playerCombat.setSuspend(false);
    playerMovement.setSuspend(false);
    return false;
}
```

Figure 59 – PlayerController.cs part 4

```
//If player right clicks then rolls.
//Returns true if player is rolling.
//Returns false if not.
1 reference
private bool IsRolling()
{
    if (playerMovement.GetIsCurrentlyRolling())
    {
        playerCombat.setSuspend(true);
        health.AddDamageImmunity();
        return true;
    }
    playerCombat.setSuspend(false);
    return false;
}

//Animation event, when animation reaches RollEnd Event
//calls the function to remove the damage immunity
0 references
void RollEnd()
{
    health.RemoveDammageImmunity();
}

1 reference
private static Ray GetMouseRay()
{
    return Camera.main.ScreenPointToRay(Input.mousePosition);   //Ray from camera
}

//Function that when player gets a kill, checks if the killed enemy is a quest goal
1 reference
public void DoKillQuestChecks(GameObject killedEnemy)
{
    foreach (Quest quest in qm.GetQuestsList())
    {
        if (!quest.isCompleted)
        {
            quest.questGoal.EnemyKilled(killedEnemy);
            if (quest.questGoal.IsGoalReached())
            {
                Debug.Log("Completed");
                GetComponent<Experience>().GainExperience(quest.experienceReward);
                quest.isCompleted = true;
                qm.CheckCompletion();
            }
        }
    }
}
```

Figure 60 – PlayerController.cs part 5

```
        //Function that when player picks up an item,
        //checks if that item is a quest goal
        1 reference
        public void DoGatherItemQuestCheck(Item pickedItem)
        {
            foreach (Quest quest in qm.GetQuestsList())
            {
                if (!quest.isCompleted)
                {
                    quest.questGoal.ItemGathered(pickedItem);
                    if (quest.questGoal.IsGoalReached())
                    {
                        Debug.Log("Completed");
                        GetComponent<Experience>().GainExperience(quest.experienceReward);
                        quest.isCompleted = true;
                        qm.CheckCompletion();
                    }
                }
            }
        }

        5 references
        public Speaker GetSpeaker()
        {
            return speaker;
        }
    }
}
```

Figure 61 – PlayerController.cs part 6

Thanks to *PlayerCombat.cs*, player is able to perform attacks, equip weapon and alert NPCs when fighting enemies.

```
using ARPG.AI;
using ARPG.Core;
using ARPG.Resources;
using ARPG.Resources.Items;
using ARPG.Stats;
using UnityEngine;
using UnityEngine.AI;

namespace ARPG.Combat
{
    Unity Script | 4 references
    public class PlayerCombat : MonoBehaviour, IAction
    {
        //Time before player can attack again
        [SerializeField] float timeBetweenAttacks = 1f;
        //Player's right hand transform
        [SerializeField] Transform rightHandTransform = null;
        //Player's left hand transform
        [SerializeField] Transform leftHandTransform = null;
        //Default weapon
        [SerializeField] WeaponItem defaultWeapon = null;
        //Audio source and clips
        [SerializeField] AudioSource combatAudioSource;
        [SerializeField] AudioClip swingClip;
        [SerializeField] AudioClip airSwingClip;
        [SerializeField] AudioClip punchClip;
        [SerializeField] AudioClip arrowReleaseClip;
        [SerializeField] AudioClip arrowHitClip;

        private bool isAttacking;
        Animator animator;
        Health health;
        float timeSinceLastAttack;
        WeaponItem currentWeapon = null;
        private bool suspend;
        RaycastHit raycastHit;

        Unity Message | 0 references
        void Start()
        {
            isAttacking = false;
            animator = GetComponent<Animator>();
            health = GetComponent<Health>();
            timeSinceLastAttack = Mathf.Infinity;
            suspend = false;
            if (currentWeapon == null)
            {
                EquipWeapon(defaultWeapon);
            }
        }
```

Figure 62 – PlayerCombat.cs part 1

```
Unity Message | 0 references
void Update()
{
    if (health.IsDead()) return;
    if (!suspend)
    {
        if (Input.GetMouseButtonDown(0) && timeSinceLastAttack > timeBetweenAttacks)
        {
            //Information for the impact point in world space
            //Casts a ray to the hit point and returns true if ray intersects with a collider
            bool hasHit = Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out raycastHit);
            DialogueNPC npc = raycastHit.transform.GetComponent<DialogueNPC>();
            if (npc == null) TriggerAttack();
        }

        if (timeSinceLastAttack > timeBetweenAttacks)//!!!!!!!!!!!!!!!!!!!
        {
            isAttacking = false;
        }
    }
    //isInCombat = false;
    timeSinceLastAttack += Time.deltaTime;
}

3 references
public void EquipWeapon(WeaponItem weaponItem)
{
    currentWeapon = weaponItem;
    Animator animator = GetComponent<Animator>();
    weaponItem.Spawn(rightHandTransform, leftHandTransform, animator);
}
```

Figure 63 – PlayerCombat.cs part 2

```
//Function that triggers annimations
1 reference
private void TriggerAttack()
{
    GetComponent<ActionScheduler>().StartAction(this);
    timeSinceLastAttack = 0;
    isAttacking = true;

    if (currentWeapon.HasProjectile())
    {
        transform.LookAt(raycastHit.point);
    }

    transform.LookAt(raycastHit.point);
    animator.ResetTrigger("stopAttack");    //Resets stop attack
    animator.SetTrigger("attack");          //Triggers attack

    GetComponent<NavMeshAgent>().speed = 0f; //!!!!!!!!!!!!!!!!!!!
}

8 references
public void Cancel()
{
    GetComponent<Animator>().ResetTrigger("attack");        //Resets attack trigger
    GetComponent<Animator>().SetTrigger("stopAttack");      //Triggers the stopAttack trigger
}

1 reference
public bool GetIsAttacking()
{
    return isAttacking;
}
1 reference
public WeaponItem GetCurrentWeapon()
{
    return currentWeapon;
}
```

Figure 64 – PlayerCombat.cs part 3

```
1 reference
void Hit()
{
    bool hit = false;
    float baseDamage = GetComponent<BaseStats>().GetStat(Stat.Damage);
    //Damage formula
    float damage = baseDamage +
        (baseDamage * currentWeapon.GetWeaponExtraPercentageDamage()) + currentWeapon.GetDamage();

    Transform temp = rightHandTransform;
    if (!currentWeapon.GetIsRightHanded())
    {
        temp = leftHandTransform;
    }

    if (currentWeapon.HasProjectile())
    {
        currentWeapon.LaunchProjectile(rightHandTransform, leftHandTransform, gameObject, damage, raycastHit);
    }
    else
    {
        Collider[] hitColliders;
        if (currentWeapon.name == "Unarmed")
        {
            hitColliders = Physics.OverlapBox(rightHandTransform.position,
                new Vector3(1f, 1f, 1f),
                rightHandTransform.localRotation,
                LayerMask.GetMask("EnemyLayer"));
            combatAudioSource.clip = punchClip;
        }
        else
        {
            temp = temp.Find("Weapon").transform;
            hitColliders = Physics.OverlapBox(temp.position + new Vector3(0f, 0.5f, 0f),
                new Vector3(0.5f, 2.5f, 0.5f),
                temp.localRotation,
                LayerMask.GetMask("EnemyLayer"));
            combatAudioSource.clip = swingClip;
        }
```

Figure 65 – PlayerCombat.cs part 4

```
            foreach (Collider hitCollider in hitColliders)
            {
                hit = true;
                if (hitCollider.gameObject.tag != "Player")
                {
                    Health enemyHealth = hitCollider.GetComponent<Health>();
                    if (enemyHealth.IsDead()) continue;
                    enemyHealth.TakeDamage(gameObject, damage);
                    AlertClosestNPC();
                }
            }

            if (hit && currentWeapon.name != "Unarmed")
            {
                combatAudioSource.clip = swingClip;
            }
            else if (!hit && currentWeapon.name != "Unarmed")
            {
                combatAudioSource.clip = airSwingClip;
            }

            combatAudioSource.volume = Random.Range(0.8f, 1);
            combatAudioSource.pitch = Random.Range(0.8f, 1.1f);
            combatAudioSource.Play();
        }
    }

    0 references
    void Shoot()            //!!! Animation event !!!
    {
        Hit();
        combatAudioSource.clip = arrowReleaseClip;
        combatAudioSource.volume = Random.Range(0.8f, 1);
        combatAudioSource.pitch = Random.Range(0.8f, 1.1f);
        combatAudioSource.Play();
    }

    10 references
    public void setSuspend(bool value)
    {
        suspend = value;
    }
```

Figure 66 – PlayerCombat.cs part 5

```
1 reference
public WeaponItem GetDefaultWeapon()
{
    return defaultWeapon;
}

//Alerts the closest NPC
1 reference
private void AlertClosestNPC()
{
    Collider[] hitColliders = null;
    hitColliders = Physics.OverlapSphere(transform.position, 10f, LayerMask.GetMask("NPCLayer"));
    foreach (Collider hitCollider in hitColliders)
    {
        if (hitCollider.gameObject.tag == "NPC" && hitCollider.GetComponent<Health>().IsDead() == false)
        {
            if (hitCollider.gameObject.GetComponent<NPCAIController>())
            {
                hitCollider.gameObject.GetComponent<NPCAIController>().TriggerNPCsAlert();
                return;
            }
            if (hitCollider.gameObject.GetComponent<WoodcutterController>())
            {
                hitCollider.gameObject.GetComponent<WoodcutterController>().TriggerNPCsAlert();
                return;
            }
            if (hitCollider.gameObject.GetComponent<FarmerController>())
            {
                hitCollider.gameObject.GetComponent<FarmerController>().TriggerNPCsAlert();
                return;
            }
        }
    }
}
}
```

Figure 67 – PlayerCombat.cs part 6

PlayerMovement.cs is the player's movement script, which allows the player to perform fluid movement. For example, when the player presses the "W" key, it starts moving at a speed of 0 and then gradually increases the speed to make the movement look more fluid. It also inherits from the IAction interface, which I will go through after *PlayerMovement.cs*.

```
using ARPG.Core;
using ARPG.Resources;
using UnityEngine;
using UnityEngine.AI;

namespace ARPG.Movement
{
    Unity Script | 4 references
    public class PlayerMovement : MonoBehaviour, IAction
    {
        //Run speed
        [SerializeField] private float runSpeed = 6f;
        //Walk speed
        [SerializeField] private float walkSpeed = 2f;
        //Turn smooth time
        [SerializeField] private float turnSmoothTime = 0.25f;
        //Sets the time needed before player can roll again
        [SerializeField] private float timeBetweenRolls = 2f;
        //AudioSource component
        [SerializeField] private AudioSource playerAudioSource;
        //AudioClip for the roll
        [SerializeField] private AudioClip rollClip;

        private NavMeshAgent navMeshAgent;
        private Animator animator;
        private float turnSmoothVelocity;
        private Transform cameraTransform;
        private Health health;
        private float timeSinceLastRoll;
        private bool isCurrentlyRolling;
        private bool suspend;

        8 references
        public void Cancel()
        {
            navMeshAgent.isStopped = true;
        }

        Unity Message | 0 references
        private void Awake()
        {
            navMeshAgent = GetComponent<NavMeshAgent>();
            animator = GetComponent<Animator>();
            cameraTransform = Camera.main.transform;
            health = GetComponent<Health>();
            timeSinceLastRoll = Mathf.Infinity;
            isCurrentlyRolling = false;
            suspend = false;
        }
```

Figure 68 – PlayerMovement.cs part 1

```
Unity Message | 0 references
private void Update()
{
    if (health.IsDead()) return;
    if (!suspend)
    {
        float horizontal = Input.GetAxisRaw("Horizontal");
        float vertical = Input.GetAxisRaw("Vertical");

        Vector3 direction = new Vector3(horizontal, 0f, vertical).normalized;
        if (Input.GetKey(KeyCode.Space) && timeSinceLastRoll > timeBetweenRolls)
        {
            Roll();
        }

        if (direction.magnitude >= 0.1f && !isCurrentlyRolling)
        {
            Movement(direction);
        }
        if (direction.magnitude < 0.1f && !isCurrentlyRolling)
        {
            navMeshAgent.speed = navMeshAgent.speed = Mathf.Clamp(navMeshAgent.speed - walkSpeed * Time.deltaTime * 4f, 0f, runSpeed);

            navMeshAgent.isStopped = false;
            float targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + cameraTransform.eulerAngles.y;
            Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) * Vector3.forward;
            navMeshAgent.destination = transform.position + (transform.forward * 1f) + moveDir.normalized * navMeshAgent.speed * Time.deltaTime;
        }
    }
    UpdateAnimator();
    timeSinceLastRoll += Time.deltaTime;
}
```

Figure 69 – PlayerMovement.cs part 2

```
//Function that is responsible for the movement based on the direction
1 reference
private void Movement(Vector3 direction)
{
    GetComponent<ActionScheduler>().StartAction(this);
    float targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + cameraTransform.eulerAngles.y;
    float angle = Mathf.SmoothDampAngle(transform.eulerAngles.y, targetAngle, ref turnSmoothVelocity, turnSmoothTime);
    transform.rotation = Quaternion.Euler(0f, angle, 0f);

    Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) * Vector3.forward;

    navMeshAgent.destination = transform.position + (transform.forward * 1f) + moveDir.normalized * navMeshAgent.speed * Time.deltaTime;


    navMeshAgent.destination = transform.position + (transform.forward * 1f) + moveDir.normalized * navMeshAgent.speed * Time.deltaTime;

    if (Input.GetKey(KeyCode.LeftShift))
    {
        navMeshAgent.speed = Mathf.Clamp(navMeshAgent.speed + runSpeed * Time.deltaTime, walkSpeed, runSpeed);
    }
    else
    {
        if (navMeshAgent.speed > walkSpeed)
        {
            navMeshAgent.speed = Mathf.Clamp(navMeshAgent.speed - walkSpeed * Time.deltaTime * 2f, walkSpeed, runSpeed);
            return;
        }
        navMeshAgent.speed = walkSpeed;
    }
    navMeshAgent.isStopped = false;
}

//Function that performs the Roll action
1 reference
private void Roll()
{
    GetComponent<ActionScheduler>().CancelCurrentAction();
    isCurrentlyRolling = true;
    timeSinceLastRoll = 0;

    //Triggers the animation triggers
    animator.ResetTrigger("stopRoll");
    animator.SetTrigger("roll");

    //Audio, plays the roll sound with a random volume and pitch
    playerAudioSource.clip = rollClip;
    playerAudioSource.volume = Random.Range(0.01f, 0.1f);
    playerAudioSource.pitch = Random.Range(0.6f, 0.7f);
    playerAudioSource.Play();

    //Moves the navMeshAgent
    navMeshAgent.destination = transform.position + (transform.forward * 8);
    navMeshAgent.speed = runSpeed * 2f;
    navMeshAgent.isStopped = false;
}
```

Figure 70 – PlayerMovement.cs part 3

```
        //Function that returns true if player is moving
        0 references
        public bool IsMoving()
        {
            Vector3 localVelocity = transform.InverseTransformDirection(navMeshAgent.velocity);
            return localVelocity.z > 0;
        }

        //Function that updates Animator's forwardSpeed with the current speed
        1 reference
        private void UpdateAnimator()
        {
            Vector3 localVelocity = transform.InverseTransformDirection(navMeshAgent.velocity);
            float speed = localVelocity.z;
            animator.SetFloat("forwardSpeed", speed);
        }

        //Animator Event
        0 references
        void RollEnd()
        {
            isCurrentlyRolling = false;
            navMeshAgent.speed = (walkSpeed + runSpeed) / 2f;
        }

        //Getters
        1 reference
        public bool GetIsCurrentlyRolling()
        {
            return isCurrentlyRolling;
        }
        1 reference
        public float GetTimeBetweenRolls()
        {
            return timeBetweenRolls;
        }

        2 references
        public float GetTimeSinceLastRoll()
        {
            return timeSinceLastRoll;
        }

        10 references
        public void setSuspend(bool value)
        {
            suspend = value;
        }
    }
}
```

Figure 71 – PlayerMovement.cs part 4

Now let us move on to IAction and the ActionScheduler. IAction.cs is an interface that contains the Cancel() function. *ActionScheduler.cs* is used every time the player or an enemy NPC leaves, for example walking, they are on a walk action, if they start attacking they are on combat action and before switching, the Cancel() function is called.

```
namespace ARPG.Core
{
    //This interface contains cancel(), it will be implemented in Fighter and Mover
    6 references
    public interface IAction
    {
        8 references
        void Cancel();
    }
}
```

Figure 72 – IAction.cs

```csharp
using UnityEngine;

namespace ARPG.Core
{
    Unity Script | 19 references
    public class ActionScheduler : MonoBehaviour
    {
        IAction currentAction;

        5 references
        public void StartAction(IAction action)     //Function that starts an action
        {
            if (currentAction == action) return;    //If the current action is the given action then skips
            if (currentAction != null)              //If the current action is not null then cancels the current action
            {
                currentAction.Cancel();
            }
            currentAction = action;                 //Sets current action to the given action
        }

        15 references
        public void CancelCurrentAction()   //Function that cancels the action
        {
            StartAction(null);
        }
    }
}
```

Figure 73 – ActionScheduler.cs

5.6 Stats & Health

5.6.1 Overview

Stats are vital for the game because they provide the sense of progress. To improve the stats, player has to either slay enemies or complete quests in order to level up. There are 4 kinds of stats:

- Health, which is the amount of health points.
- Experience Reward, which is the amount of experience rewarded after dying.
- Experience to level up, which is the amount of experience needed to level up.
- Damage, which is the amount of damage inflicted on health points.

In reality the actual stats are health and damage because the other two do not directly affect the feeling of progress.

5.6.2 Code

Starting with the Stats, they consist of 8 scripts which some of those are responsible for UI and others are for logic.

- *Basestats.cs*
- *CharacterClass.cs*
- *Experience.cs*
- *ExperienceDisplay.cs*
- *LevelDisplay.cs*
- *Stats.cs*
- *XPDisplay.cs*

Starting with the *CharacterClass.cs*, this is an enum to set the classes of the player, NPCs and enemies (there is not a class for the guards because guards and share the same class with some enemies, that class is called ExiledKnight).

```
namespace ARPG.Stats
{
    6 references
    public enum CharacterClass
    {
        Player,
        Civilian,
        Rogue,
        ExiledKnight,
        Raider,
        Mage,
    }
}
```

Figure 74 – CharacterClass.cs

Next is the *Stat.cs* which was explained earlier, it is an enum to set the stats.

```
namespace ARPG.Stats
{
    20 references
    public enum Stat
    {
        Health,
        ExperienceReward,
        ExperienceToLevelUp,
        Damage
    }
}
```

Figure 75 – Stat.cs

Now moving to the *Progression.cs* which is really important. This is a scriptable object used to set the values of a class for a number of levels. For instance, Figure 76 shows the progression scriptable object of the Player class, also we can see that player's progression uses 3 stats, Health, Experience To Level Up and Damage. And all of those 3 stats are having values till the level 10.

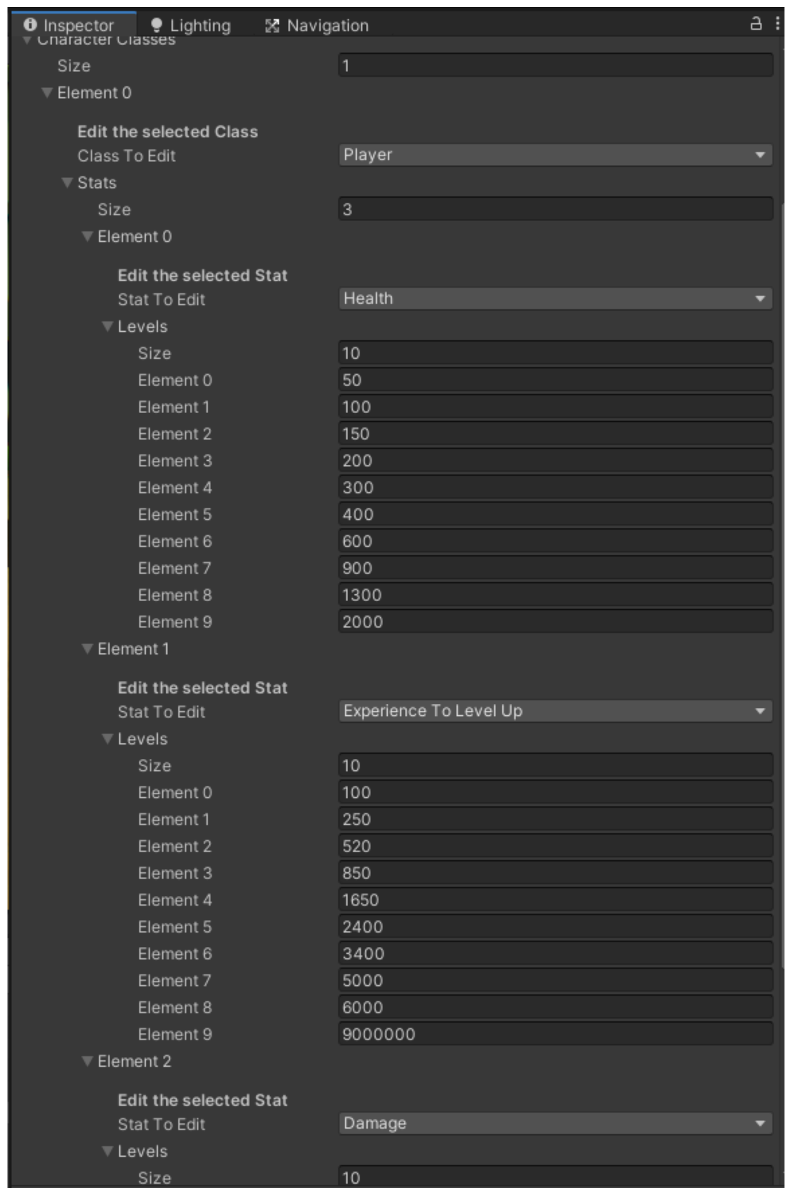


Figure 76 – Progression of the Player

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

namespace ARPG.Stats
{
    [CreateAssetMenu(fileName = "Progression", menuName = "Stats/Progression", order = 0)]
    public class Progression : ScriptableObject
    {
        [Space]
        [Header("This scriptable object provides the all the game classes for edit")]
        [Header("Progression Manager")]
        [Space]

        [SerializeField] ProgressionCharacterClass[] characterClasses = null;
        Dictionary<CharacterClass, Dictionary<Stat, float[]>> lookupTable = null;

        //This class gets serialized into the inspector so that the editor
        //can select the class and how many stats for that class to edit.
        [System.Serializable]
        class ProgressionCharacterClass
        {
            [HideInInspector]
            public string name = "Class";

            [Header("Edit the selected Class")]
            //Class to edit (Player or enemy classes)
            public CharacterClass ClassToEdit;
            //How many stats (max 4, because only 4 stats exist)
            public ProgressionStat[] stats;
        }

        //This class gets serialized into the inspector so that
        //editor can select the stat and edit the levels of that stat(or stats)
        [System.Serializable]
        class ProgressionStat
        {
            [Header("Edit the selected Stat")]
            //Stat to edit (Health, XP Reward, XP To LevelUp, Damage)
            public Stat StatToEdit;
            //Levels for the selected stat to edit
            public float[] levels;
        }
```

Figure 77 – Progression.cs part 1

```
//Function that returns the float value of the characterClass's stat at given level.
3 references
public float GetStat(Stat stat,CharacterClass characterClass, int level)
{
    BuildLookup();

    float[] levels = lookupTable[characterClass][stat];

    if(levels.Length < level)
    {
        return 0;
    }

    return levels[level - 1];
}

//Function that returns the max level of the given stat of the characterClass.
1 reference
public int GetLevels(Stat stat,CharacterClass characterClass)
{
    BuildLookup();

    float[] levels = lookupTable[characterClass][stat];
    return levels.Length;
}

//Function that builds the lookupTable Dictionary.
//For each CharacterClass in progression, assigns the levels to the StatToEdit.
2 references
private void BuildLookup()
{
    if (lookupTable != null) return;

    lookupTable = new Dictionary<CharacterClass, Dictionary<Stat, float[]>>();

    foreach (ProgressionCharacterClass progressionClass in characterClasses)
    {
        var statLookupTable = new Dictionary<Stat, float[]>();

        foreach (ProgressionStat progressionStat in progressionClass.stats)
        {
            statLookupTable[progressionStat.StatToEdit] = progressionStat.levels;
        }

        lookupTable[progressionClass.ClassToEdit] = statLookupTable;
    }
}
}
}
```

Figure 78 – Progression.cs part 2

Now moving to *Experience.cs*, this is a script used to store the experience points and contains a delegate that calls another function whenever player gains experience points.

```
using System;
using UnityEngine;

namespace ARPG.Stats
{
    Unity Script | 12 references
    public class Experience : MonoBehaviour
    {
        [SerializeField] private float experiencePoints = 0;

        public event Action onExperienceGained;

        //Function that adds experience
        3 references
        public void GainExperience(float experience)
        {
            experiencePoints += experience;
            onExperienceGained();
        }

        5 references
        public float GetExperience()
        {
            return experiencePoints;
        }
    }
}
```

Figure 79 – Experience.cs

The last of the logic scripts is *BaseStats.cs*, this script is responsible for updating the level, leveling up, getting a stat's level and for calculating the level.

```
using System;
using UnityEngine;

namespace ARPG.Stats
{
    public class BaseStats : MonoBehaviour
    {
        [Range(1,50)]
        [SerializeField] private int startingLevel = 1;
        [SerializeField] private CharacterClass characterClass;
        [SerializeField] private Progression progression = null;
        [SerializeField] private GameObject levelUpParticleEffect = null;

        private int currentLevel = 0;
        private Experience experience;

        //Currently used on Health.cs for regenerating health when leveling up.
        public event Action onLevelUp;

        private void Awake()
        {
            experience = GetComponent<Experience>();
            if (experience != null)
            {
                //Assigns onExperienceGained to UpdateLevel() when experience is gained
                experience.onExperienceGained += UpdateLevel;
            }
        }

        private void Start()
        {
            currentLevel = CalculateLevel();
        }
```

Figure 80 – BaseStats.cs part 1

```
        //Function that gets called when experience is gained and updates the level if needed,
        //also instantiates particles and regenerates health upon leveling up.
        private void UpdateLevel()
        {
            int newLevel = CalculateLevel();
            if(newLevel > currentLevel)
            {
                currentLevel = newLevel;
                if(levelUpParticleEffect != null)
                {
                    LevelUpEffect();
                }
                onLevelUp();
            }
        }

        //Function that instantiates particles
        private void LevelUpEffect()
        {
            Instantiate(levelUpParticleEffect, transform);
        }

        //Function that returns the value for the given stat's level
        public float GetStat(Stat stat)
        {
            return progression.GetStat(stat, characterClass, GetLevel());
        }

        //Function that returns the value for the given stat's given level
        public float GetStat(Stat stat, int level)
        {
            return progression.GetStat(stat, characterClass, level);
        }

        //Getter function
        public int GetLevel()
        {
            if(currentLevel < 1)
            {
                currentLevel = CalculateLevel();
            }
            return currentLevel;
        }
```

Figure 81 – BaseStats.cs part 2

```
        //Function that calculates and returns the level. (Used for player,
        //as player has the Experience component to be able to gain XP and level up. Can be used for enemies too)
        3 references
        private int CalculateLevel()
        {
            Experience experience = GetComponent<Experience>();
            if (experience == null) return startingLevel;         //If there is no experience component,returns the startingLevel.

            float currentXP = experience.GetExperience();                                      //Else, gets the current experience points
            int maxLevel = progression.GetLevels(Stat.ExperienceToLevelUp, characterClass); //gets the max level of the given characterClass
            for (int level = 1; level <= maxLevel; level++)                                    //And finds the level based on the currentXP
            {
                float xpToLevelUp = progression.GetStat(Stat.ExperienceToLevelUp, characterClass, level);
                if(xpToLevelUp > currentXP)
                {
                    return level;
                }
            }
            return maxLevel + 1;
        }
    }
}
```

Figure 82 – BaseStats.cs part 3

And before moving to the Health script, there are 3 scripts that update values for the UI as stated earlier, those scripts are *ExperienceDisplay.cs*, *LevelDisplay.cs* and *XPDisplay.cs*.

- *ExperienceDisplay.cs* is used to display the experience on screen using a slider.
- *LevelDisplay.cs* is used to display the level as text.
- *XPDisplay.cs* is used to display the experience as text.

```
using UnityEngine;
using UnityEngine.UI;

namespace ARPG.Stats
{
    Unity Script | 0 references
    public class ExperienceDisplay : MonoBehaviour
    {
        private Slider slider;
        Experience experience;
        BaseStats baseStats;

        Unity Message | 0 references
        private void Awake()
        {
            experience = GameObject.FindWithTag("Player").GetComponent<Experience>();
            baseStats = GameObject.FindWithTag("Player").GetComponent<BaseStats>();
            slider = GetComponent<Slider>();
            slider.minValue = experience.GetExperience();
            slider.maxValue = baseStats.GetStat(Stat.ExperienceToLevelUp);
            slider.value = experience.GetExperience();
            baseStats.onLevelUp += updateMinValue;
        }

        Unity Message | 0 references
        private void Update()
        {
            slider.maxValue = baseStats.GetStat(Stat.ExperienceToLevelUp);
            slider.value = experience.GetExperience();
        }

        1 reference
        private void updateMinValue()
        {
            slider.minValue = baseStats.GetStat(Stat.ExperienceToLevelUp, baseStats.GetLevel() - 1);
        }
    }
}
```

Figure 83 – ExperienceDisplay.cs

```
using TMPro;
using UnityEngine;

namespace ARPG.Stats
{
    public class LevelDisplay : MonoBehaviour
    {
        BaseStats baseStats;
        TextMeshProUGUI levelValue;

        private void Awake()
        {
            baseStats = GameObject.FindWithTag("Player").GetComponent<BaseStats>();
            levelValue = GetComponent<TextMeshProUGUI>();
        }

        private void Update()
        {
            levelValue.SetText("{0:0}", baseStats.GetLevel());
        }
    }
}
```

Figure 84 – LevelDisplay.cs

```
using TMPro;
using UnityEngine;

namespace ARPG.Stats
{
    public class XPDisplay : MonoBehaviour
    {
        Experience experience;
        BaseStats baseStats;
        TextMeshProUGUI xpValue;

        private void Awake()
        {
            experience = GameObject.FindWithTag("Player").GetComponent<Experience>();
            baseStats = GameObject.FindWithTag("Player").GetComponent<BaseStats>();
            xpValue = GetComponent<TextMeshProUGUI>();
        }

        private void Update()
        {
            xpValue.SetText("{0:0}", experience.GetExperience());
        }
    }
}
```

Figure 85 – XPDisplay.cs

Moving to the *Health.cs*, this script is responsible of storing the health points. Also, it contains a few functions that apply damage, display the damage that was dealt, add or remove damage immunity (used when player is rolling), healing and regenerating health upon leveling up.

```csharp
using UnityEngine;
using ARPG.Stats;
using ARPG.Core;
using ARPG.Control;

namespace ARPG.Resources
{
    Unity Script | 48 references
    public class Health : MonoBehaviour
    {
        [SerializeField] private float healthPoints = 100f;
        [SerializeField] private bool destroyAfterDeath = false;
        [SerializeField] private float afterDeathDestroyTime = 10f;
        [SerializeField] GameObject damagePopup = null;
        [SerializeField] AudioSource healthAudioSource;
        [SerializeField] AudioClip hurtClip;

        private bool isDead = false;
        private bool isImmuneToDamage = false;

        Unity Message | 0 references
        private void Start()
        {
            healthPoints = GetComponent<BaseStats>().GetStat(Stat.Health);
            GetComponent<BaseStats>().onLevelUp += RegenerateHealth;
        }
```

Figure 86 – Health.cs part 1

```csharp
        //Function that applies damage to agent
        3 references
        public void TakeDamage(GameObject attacker, float damage)
        {
            //If agent is not immune to damage
            if (!isImmuneToDamage)
            {
                healthAudioSource.clip = hurtClip;
                healthAudioSource.volume = Random.Range(0.8f, 1f);
                healthAudioSource.pitch = Random.Range(0.8f, 1.1f);
                healthAudioSource.Play();

                //Decreases healthPoints by damage and health won't go below 0
                healthPoints = Mathf.Max(healthPoints - damage, 0);
                if (damagePopup)
                {
                    DamagePopupInstantiation(damage);
                }

                if (healthPoints == 0)              //If health points reach 0
                {
                    Die();                          //Agent dies
                    AwardExperience(attacker);      //Awards xp to the attacker
                    //If the attacker is Player then check if the killed agent is a quest goal
                    if (attacker.tag == "Player")
                    {
                        attacker.GetComponent<PlayerController>().DoKillQuestChecks(gameObject);
                    }

                    if (GetComponent<DropItemOnDeath>())
                    {
                        if(GetComponent<DropItemOnDeath>().item != null) GetComponent<DropItemOnDeath>().Drop();
                    }
                    if(destroyAfterDeath)Destroy(gameObject, afterDeathDestroyTime);
                }
            }
        }

        //Function that used to increase the healthPoints by the given value.
        //Used when player consumes a consumableItem
        1 reference
        public void Heal(float value)
        {
            if ((healthPoints+value) > GetMaxHealthPoints())
            {
                healthPoints = GetMaxHealthPoints();
            }
            else
            {
                healthPoints = healthPoints + value;
            }
        }
```

Figure 87 – Health.cs part 2

```
//Function that enables damage immunity
1 reference
public void AddDamageImmunity()
{
    isImmuneToDamage = true;
}

//Function that removes damage immunity
1 reference
public void RemoveDammageImmunity()
{
    isImmuneToDamage = false;
}

//Function that displays the dealt damage
1 reference
private void DamagePopupInstantiation(float damage)
{
    GameObject instance = Instantiate(damagePopup, transform.position, Quaternion.identity, transform);

    instance.GetComponent<TextMesh>().text = "-" + damage.ToString();
}

//Function that makes agent dead
1 reference
private void Die()
{
    if (isDead) return;                                         //If agent is already dead then return and skip the rest
                                                                //Else
    isDead = true;                                              //Sets agent's status to dead
    GetComponent<Animator>().SetTrigger("die");                 //Uses the die trigger to use the animation
    GetComponent<ActionScheduler>().CancelCurrentAction();      //Cancels agent's current action
}

//Function that awards xp to the attacker
1 reference
private void AwardExperience(GameObject attacker)
{
    Experience experience = attacker.GetComponent<Experience>();
    if (experience == null) return;
    experience.GainExperience(GetComponent<BaseStats>().GetStat(Stat.ExperienceReward));
}

//Function that regenerates healthPoints. Used when agent levels up
1 reference
private void RegenerateHealth()
{
    healthPoints = GetComponent<BaseStats>().GetStat(Stat.Health);
}
```

Figure 88 – Health.cs part 3

```
//Function that returns true or false accordingly to agent's health status
28 references
public bool IsDead()
{
    return isDead;
}

//Function that returns the max health points based on the current level
5 references
public float GetMaxHealthPoints()
{
    return GetComponent<BaseStats>().GetStat(Stat.Health);
}

//Function that returns the percentage of healthPoints
0 references
public float GetPercentage()
{
    return 100 * (healthPoints / GetComponent<BaseStats>().GetStat(Stat.Health));
}

5 references
public float GetHealthPoints()
{
    return healthPoints;
}

//Function that returns isImmuneToDamage
1 reference
public bool GetIsImmuneToDamage()
{
    return isImmuneToDamage;
}
    }
}
```

Figure 89 – Health.cs part 4

5.7 AI

5.7.1 State Machine

5.7.1.1 Overview

As mentioned in an earlier section, the behavior of AI is controlled by a finite state machine implemented with state design pattern. The State Design Pattern is one of twenty-three design patterns documented by the gang of Four four that describe how to solve recurring design problems. These problems cover the design of flexible and reusable object-oriented software, such as objects that can be easily implemented, modified, tested, and reused.

The state pattern is set to solve two main problems:

- An object should change its behavior when its internal state changes.
- A state-specific behavior should be defined independently. That is, adding new states would not affect the behavior of existing states.

Implementing state-specific behavior directly in a class is inflexible because it locks the class into a specific behavior and makes it impossible to later add a new state or change the behavior of an existing state independently of the class. For this, the pattern describes two solutions:

- Define separate objects that encapsulate state-specific behavior for each state. That is, define an interface for performing state-specific behavior, and define classes that implement the interface for each state.
- A class delegates state-specific behavior to its current state object instead of implementing state-specific behavior directly.

Now moving on “how it works?”, there are StateMachine.cs, IState, State scripts and controller scripts.

- StateMachine.cs is the logic of the state pattern.
- IState is an interface that’s inherited by the States scripts.
    - IState has 3 methods:
        - OnEnter, called once when entered the state.
        - Tick, like Update called every frame.
        - OnExit, called once when exiting the state.
- State scripts are inheriting from IState only and not from *MonoBehavior*.
- Controller scripts have the transitions between the states of the state scripts.

And now an example:

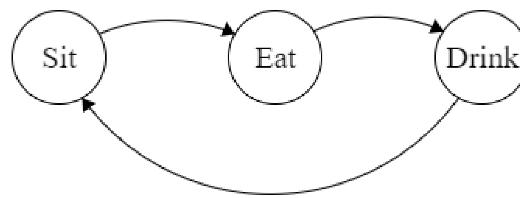


Figure 90 – Example states and transitions

Suppose the current state is the Sit state, then the Tick method is called every frame, but if the conditions for the transition from the state Sit to the state Eat are met, then before the current state is changed, the OnExit method of the state Sit is called, then the current state is changed to the state Eat and its OnEnter method is called once, and Tick is called every second.

5.7.1.2 Code

There are 2 scripts I'll go through, StateMachine.cs, which is the logic of the state pattern, and IState, which is an interface. StateMachine.cs is responsible for the functionality of the state pattern. It contains a class called Transition, which has 2 fields:

- *Funch<bool> Condition*, which is the condition that needs to be satisfied to move to the state.
- *IState To*, which is the state that the transition points to.

The next up, are StateMachine Class's fields:

- *IState _currentState*, is used to define the current state.
- *Dictionary<Type, List<Transition>> _transitions*, is used to store all the transitions between the states.
- *List<Transition> _currentTransitions*, is a list of transitions used to store the possible transitions of a state.
- *List<Transition> _anyTransitions*, is a list of transitions used to store the transitions from all states in a specific state.
- *List<Transition> EmptyTransitions,* is an empty list of transitions.

Moving to the methods:

- *GetTransition*, returns a transition that it's condition returns true, starting by the *_anytransitions* and then the *_currentTransitions* and if no condition returns true then the function returns null.
- *SetState*, sets *_currentState* to the given state and calls the *OnExit* method of the previous state if there was a previous a state and then calls the *OnEnter* method of the new state.
- *Tick,* tries to get a transition and if it does then sets state to that transition's state and calls itself again.
- *AddTransition*, adds a transition from a state to another state with the given condition.
- *AddAnyTransition*, adds transitions from all states to a specific state with the given condition.

```
using System;
using System.Collections.Generic;

namespace ARPG.AI
{
    11 references
    public class StateMachine
    {
        private IState _currentState;

        //Dictionary that stores the transitions between states
        private Dictionary<Type, List<Transition>> _transitions = new Dictionary<Type, List<Transition>>();
        //List used to store the possible transitions of a state
        private List<Transition> _currentTransitions = new List<Transition>();
        //List used to store the transitions from all states
        private List<Transition> _anyTransitions = new List<Transition>();

        private static List<Transition> EmptyTransitions = new List<Transition>(0);

        //Function used to get a transition and if it gets
        //then sets the current state to the transition's TO(Destination)
        //and calls Tick again
        5 references
        public void Tick()
        {
            var transition = GetTransition();
            if (transition != null)
                SetState(transition.To);

            _currentState?.Tick();
        }

        //Function that sets the state, calls OnExit if there was a previous state
        //And calls OnEnter of the new state that was just setted
        6 references
        public void SetState(IState state)
        {
            if (state == _currentState)
                return;

            _currentState?.OnExit();
            _currentState = state;

            //Gives the List of transitions of a type and store it to _currentTransitions
            _transitions.TryGetValue(_currentState.GetType(), out _currentTransitions);
            if (_currentTransitions == null)
                _currentTransitions = EmptyTransitions;

            _currentState.OnEnter();
        }
```

Figure 91 – StateMachine.cs part 1

```
        //Function that adds transition FROM a state TO another state,
        //
        5 references
        public void AddTransition(IState from, IState to, Func<bool> predicate)
        {
            if (_transitions.TryGetValue(from.GetType(), out var transitions) == false)
            {
                transitions = new List<Transition>();
                _transitions[from.GetType()] = transitions;
            }

            transitions.Add(new Transition(to, predicate));
        }

        //Function that adds transitions from every state to the given STATE,
        //When PREDICATE conditions are satisfied
        4 references
        public void AddAnyTransition(IState state, Func<bool> predicate)
        {
            _anyTransitions.Add(new Transition(state, predicate));
        }

        //Transition Class
        13 references
        private class Transition
        {
            3 references
            public Func<bool> Condition { get; }
            2 references
            public IState To { get; }

            2 references
            public Transition(IState to, Func<bool> condition)
            {
                To = to;
                Condition = condition;
            }
        }

        //Function that returns a transition,
        //First looking the _anyTransitions List
        //and last looks the _currentTransitions List
        1 reference
        private Transition GetTransition()
        {
            foreach (var transition in _anyTransitions)
                if (transition.Condition())
                    return transition;

            foreach (var transition in _currentTransitions)
                if (transition.Condition())
                    return transition;

            return null;
        }
    }
}
```

Figure 92 – StateMachine.cs part 2

As was stated in this chapter, IState is an interface containing 3 methods.

```
namespace ARPG.AI
{
    45 references
    public interface IState
    {
        29 references
        void Tick();
        29 references
        void OnEnter();
        29 references
        void OnExit();
    }
}
```

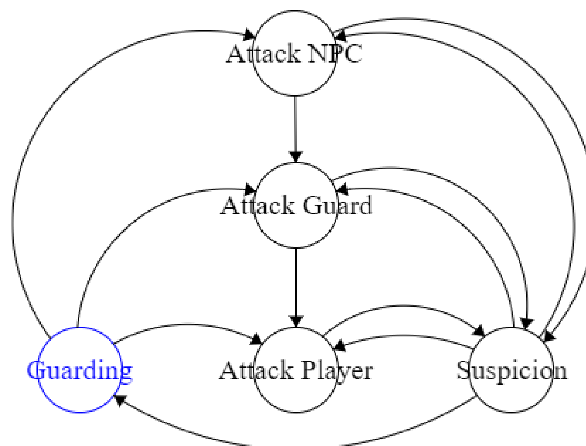
Figure 93 – IState.cs

5.7.2 Enemy AI Controller & States

5.7.2.1 Overview

Enemies are controlled by a controller that contains the transitions between the states that define the behavior. Specific those scripts are:

- EnemyAIController.cs
- State scripts:
    - *EnemyAttackNPC.cs*, defines the Attack NPC State.
    - *EnemyAttackGuard.cs*, defines the Attack Guard State.
    - *EnemyAttackPlayer.cs*, defines the Attack Player State.
    - *EnemySuspicion.cs*, defines the Suspicion State.
    - *EnemyGuarding.cs*, defines the Guarding State.



Looking back at Figure 12 – Enemy's finite state machine – EnemyAIController.cs

Enemy's behavior is based on Figure 12 and in chapter 4.4.4 that behavior was analyzed and explained.

5.7.2.2 Code

Let us start with the enemy controller. This script contains some useful methods and is responsible for setting up the transitions and logic between states. Also, Figure 95 shows the Awake method, where some variables are initialized, but most importantly, the transitions are set up exactly as in Figure 12. Figure 96 shows the logic of all the transitions shown in Figure 95, and Figure 97 shows the Update where it makes sure to Tick as long as the enemy is alive. The remaining methods are useful for the logic of the transitions and the states.

```
using UnityEngine;
using UnityEngine.AI;
using System;
using ARPG.Resources;
using ARPG.Control;
using ARPG.Combat;

namespace ARPG.AI
{
    Unity Script | 18 references
    public class EnemyAIController : MonoBehaviour
    {
        [SerializeField] QuestEnemy questEnemy;
        [Header("Behaviour & Walking")]
        [SerializeField] bool attacksNPCS = false;      //Attacks NPCs
        [SerializeField] float chaseDistance = 5f;      //Chase distance
        [SerializeField] float alertDistance = 10f;     //Distance to alert other enemies
        [SerializeField] float suspicionTime = 10f;     //Suspicion Time
        [SerializeField] WaypointPath waypointPath;     //Waypoint path
        [SerializeField] float waypointDwellTime = 3f;  //Waypoint wait time
        [Range(0, 1)]
        [SerializeField] float patrolspeedFraction = 0.35f; //Patrol speed fraction

        private Vector3 guardPosition;
        private StateMachine stateMachine;
        private Health health;

        33 references
        public GameObject closestNPC { get; set; }              //Defines the closest NPC
        13 references
        public GameObject closestGuard { get; set; }            //Defines the closest Guard
        14 references
        public float timeSinceLastSawTarget { get; set; }       //Defines the time since last saw target
        5 references
        public float timeSinceArrivedAtWaypoint { get; set; }   //Defines the time since arrived at waypoint
        6 references
        public GameObject player { get; set; }                  //Defines the player
        7 references
        public bool gotAlerted { get; set; }                    //Defines if got alerted by another enemy
        3 references
        public bool gotHitByBow { get; set; }                   //Defines if got hit by bow
```

Figure 94 – EnemyAIController.cs part 1

```
        Unity Message | 0 references
        void Awake()
        {
            timeSinceLastSawTarget = Mathf.Infinity;
            timeSinceArrivedAtWaypoint = Mathf.Infinity;
            player = GameObject.FindWithTag("Player");
            guardPosition = transform.position;
            health = GetComponent<Health>();
            gotAlerted = false;

            var fighter = GetComponent<Fighter>();
            var navMeshAgent = GetComponent<NavMeshAgent>();
            var animator = GetComponent<Animator>();

            stateMachine = new StateMachine();

            var guarding = new EnemyGuarding(this, waypointPath, waypointDwellTime, guardPosition, timeSinceArrivedAtWaypoint);
            var attackPlayer = new EnemyAttackPlayer(this);
            var attackGuard = new EnemyAttackGuard(this);
            var attackNPC = new EnemyAttackNPC(this);
            var suspicion = new EnemySuspicion(this);

            //Setting up all transitions
            At(guarding, attackGuard, CheckIfCanAttackGuard());                     //Guarding -> Attack Guard
            At(guarding, attackNPC, CheckIfCanAttackNPC());                         //Guarding -> Attack NPC
            At(attackNPC, attackGuard, CheckToAttackGuardInsteadOfNPC());           //Attack NPC -> Attack Guard
            At(attackPlayer, suspicion, SuspicionAfterPlayer());                    //Attack Player -> Suspicion
            At(attackGuard, suspicion, SuspicionAfterGuard());                      //Attack Guard -> Suspicion
            At(attackNPC, suspicion, SuspicionAfterNPC());                          //Attack NPC -> Suspicion
            At(suspicion, attackGuard, CheckToAttackGuardAfterSuspicion());         //Suspicion -> Attack Guard
            At(suspicion, attackNPC, CheckToAttackNPCAfterSuspicion());             //Suspicion -> Attack NPC
            At(suspicion, guarding, SuspicionWait());                               //Suspicion -> Guarding
            stateMachine.AddAnyTransition(attackPlayer, CheckIfCanAttackPlayer());  //All States -> Attack Player
            At(guarding, attackPlayer, () => gotAlerted);                           //Guarding -> Attack Player
            stateMachine.SetState(guarding);

            void At(IState to, IState from, Func<bool> condition) => stateMachine.AddTransition(to, from, condition);
```

Figure 95 – EnemyAIController.cs part 2

```
    //True if enemy can attack player
    Func<bool> CheckIfCanAttackPlayer() => () => (fighter.CanAttack(player) && InAttackRange(player));

    //True if enemy can attack guard
    Func<bool> CheckIfCanAttackGuard() => () => timeSinceLastSawTarget > suspicionTime && CheckForNPCsAround("Guard") && fighter.CanAttack(closestGuard) && InAttackRange(closestGuard);

    //True if enemy can attack NPC
    Func<bool> CheckIfCanAttackNPC() => () => timeSinceLastSawTarget > suspicionTime && attacksNPCS && CheckForNPCsAround("NPC") && fighter.CanAttack(closestNPC) && InAttackRange(closestNPC);

    //True if enemy can attack guard instead of NPC
    Func<bool> CheckToAttackGuardInsteadOfNPC() => () => CheckForNPCsAround("Guard") && fighter.CanAttack(closestGuard) && InAttackRange(closestGuard);

    //True if enemy can get into suspicion after losing player out of sight
    Func<bool> SuspicionAfterPlayer() => () => timeSinceLastSawTarget < suspicionTime && !InAttackRange(player) && !gotAlerted;

    //True if enemy can get into suspicion after losing guard out of sight
    Func<bool> SuspicionAfterGuard() => () => timeSinceLastSawTarget < suspicionTime && !fighter.CanAttack(closestGuard);

    //True if enemy can get into suspicion after losing NPC out of sight
    Func<bool> SuspicionAfterNPC() => () => !CheckForNPCsAround("Guard") && timeSinceLastSawTarget < suspicionTime && !fighter.CanAttack(closestNPC);

    //True if enemy can attack the guard after suspicion
    Func<bool> CheckToAttackGuardAfterSuspicion() => () => timeSinceLastSawTarget < suspicionTime && CheckForNPCsAround("Guard") && fighter.CanAttack(closestGuard) && InAttackRange(closestGuard);

    //True if enemy can attack the NPC after suspicion
    Func<bool> CheckToAttackNPCAfterSuspicion() => () => !CheckForNPCsAround("Guard") && attacksNPCS && timeSinceLastSawTarget < suspicionTime && fighter.CanAttack(closestNPC) && InAttackRange(closestNPC);

    //True if enemy waited enough time in suspicion
    Func<bool> SuspicionWait() => () => (timeSinceLastSawTarget >= suspicionTime);
}
```

Figure 96 – EnemyAIController.cs part 3

```
    Unity Message | 0 references
    void Update()
    {
        if (!health.IsDead())       //If enemy isn't dead
        {
            stateMachine.Tick();    //Call Tick
            UpdateTimers();
        }
    }

    //Function that updates timers
    1 reference
    private void UpdateTimers()
    {
        timeSinceLastSawTarget += Time.deltaTime;
        timeSinceArrivedAtWaypoint += Time.deltaTime;
    }

    //Function that checks is target is in attack range
    12 references
    public bool InAttackRange(GameObject target)
    {
        //Calculate the distance between target and enemy
        float distanceToPlayer = Vector3.Distance(target.transform.position, transform.position);
        //If distance to player is less than the chase distance returns true
        return distanceToPlayer < chaseDistance;
    }
```

Figure 97 – EnemyAIController.cs part 4

```
        //Function that checks if Guard or NPC is around
        8 references
        public bool CheckForNPCsAround(string npcType)
        {
            if (npcType == "NPC" && closestNPC != null && closestNPC.GetComponent<Health>().IsDead() == false && InAttackRange(closestNPC) && closestNPC.activeSelf == true)
            {
                return true;
            }
            else if (npcType == "Guard" && closestGuard != null && closestGuard.GetComponent<Health>().IsDead() == false && InAttackRange(closestGuard) && closestGuard.activeSelf == true)
            {
                return true;
            }

            Collider[] hitColliders = null;
            hitColliders = Physics.OverlapSphere(transform.position, chaseDistance, LayerMask.GetMask("NPCLayer"));
            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.gameObject.tag == npcType && hitCollider.GetComponent<Health>().IsDead() == false)
                {
                    if (npcType == "Guard" && hitCollider.gameObject.tag == "Guard")
                    {
                        closestGuard = hitCollider.gameObject;
                        return true;
                    }
                    if (npcType == "NPC" && hitCollider.gameObject.tag == "NPC")
                    {
                        closestNPC = hitCollider.gameObject;
                        return true;
                    }
                }
            }
            return false;
        }

        0 references
        public StateMachine GetStateMachine()
        {
            return stateMachine;
        }

        1 reference
        public float GetPatrolspeedFraction()
        {
            return patrolspeedFraction;
        }

        1 reference
        public QuestEnemy GetEnemyType()
        {
            return questEnemy;
        }
        1 reference
        public float GetAlertDistance()
        {
            return alertDistance;
        }
    }
}
```

Figure 98 – EnemyAIController.cs part 5

Now moving on to the State scripts and beginning with the *EnemyAttackNPC.cs* (Figure 99) which is the AttackNPC State. In this state *OnEnter* and *OnExit* does nothing, but *Tick* is making the enemy to attack the closest NPC whilst triggering the closest NPC' alert because of the attack.

```
using ARPG.Combat;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class EnemyAttackNPC : IState
    {
        private readonly EnemyAIController enemyAIController;

        1 reference
        public EnemyAttackNPC(EnemyAIController _enemyAIController)
        {
            enemyAIController = _enemyAIController;
        }

        //Enemy attacks the closest NPC and the closest NPC is being
        //in alert because the enemy is attacking
        29 references
        public void Tick()
        {
            AttackBehaviour(enemyAIController.closestNPC);

            if (enemyAIController.closestNPC.GetComponent<NPCAIController>())
            {
                enemyAIController.closestNPC.GetComponent<NPCAIController>().TriggerNPCsAlert();
            }
            if (enemyAIController.closestNPC.GetComponent<WoodcutterController>())
            {
                enemyAIController.closestNPC.GetComponent<WoodcutterController>().TriggerNPCsAlert();
            }
            if (enemyAIController.closestNPC.GetComponent<FarmerController>())
            {
                enemyAIController.closestNPC.GetComponent<FarmerController>().TriggerNPCsAlert();
            }
        }

        29 references
        public void OnEnter() { }

        29 references
        public void OnExit() { }

        1 reference
        private void AttackBehaviour(GameObject target)
        {
            enemyAIController.timeSinceLastSawTarget = 0;   //Resets time since last saw player
            enemyAIController.GetComponent<Fighter>().Attack(target);   //agent attacks the player
        }
    }
}
```

Figure 99 – EnemyAttackNPC.cs

Next state is Attack Guard state which is the *EnemyAttackGuard.cs* (Figure 100). *OnEnter* and *OnExit* methods does nothing, but *Tick* is triggering enemy’s attacks to the closest guard and if any NPCs are a round are being alerted because of that fight.

```
using ARPG.Combat;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class EnemyAttackGuard : IState
    {
        private readonly EnemyAIController enemyAIController;

        1 reference
        public EnemyAttackGuard(EnemyAIController _enemyAIController)
        {
            enemyAIController = _enemyAIController;
        }

        //Attacks the closest guard and checks if there are NPCs around
        //If there are NPCs around then those NPCs are being in alert because of the fight that is happening
        29 references
        public void Tick()
        {
            AttackBehaviour(enemyAIController.closestGuard);
            enemyAIController.CheckForNPCsAround("NPC");
            if (enemyAIController.closestNPC != null && enemyAIController.InAttackRange(enemyAIController.closestNPC))
            {
                if (enemyAIController.closestNPC.GetComponent<NPCAIController>())
                {
                    enemyAIController.closestNPC.GetComponent<NPCAIController>().TriggerNPCsAlert();
                    return;
                }
                if (enemyAIController.closestNPC.GetComponent<WoodcutterController>())
                {
                    enemyAIController.closestNPC.GetComponent<WoodcutterController>().TriggerNPCsAlert();
                    return;
                }
                if (enemyAIController.closestNPC.GetComponent<FarmerController>())
                {
                    enemyAIController.closestNPC.GetComponent<FarmerController>().TriggerNPCsAlert();
                    return;
                }
            }
        }

        29 references
        public void OnEnter() { }

        29 references
        public void OnExit() { }

        1 reference
        private void AttackBehaviour(GameObject target)
        {
            enemyAIController.timeSinceLastSawTarget = 0;               //Resets time since last saw player
            enemyAIController.GetComponent<Fighter>().Attack(target);   //agent attacks the player
        }
    }
}
```

Figure 100 – EnemyAttackGuard.cs

Attack Player state follows, which is the *EnemyAttackPlayer.cs* (Figure 101 & 102). *OnEnter* resets the enemy's alert and *OnExit* does nothing, but *Tick* is triggering enemy's attacks to the player, calls *AlertEnemyNPCsAround* method and checks if there are NPCs round, to be alerted because of that fight. *AlertEnemyNPCsAround* method, alerts the enemies that are close to the enemy who alerted, making them to join the fight.

```
using ARPG.Combat;
using ARPG.Resources;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class EnemyAttackPlayer : IState
    {
        private readonly EnemyAIController enemyAIController;

        1 reference
        public EnemyAttackPlayer(EnemyAIController _enemyAIController)
        {
            enemyAIController = _enemyAIController;
        }

        //Enemy attacks the player and checks if there are NPCs around
        //If there are NPCs around then those NPCs are being in alert because of the fight that is happening
        //Also if an enemy is close then that enemy is alerted to join the fight
        29 references
        public void Tick()
        {
            AttackBehaviour(enemyAIController.player);
            enemyAIController.CheckForNPCsAround("NPC");
            if (enemyAIController.closestNPC != null && enemyAIController.InAttackRange(enemyAIController.closestNPC))
            {
                if (enemyAIController.closestNPC.GetComponent<NPCAIController>())
                {
                    enemyAIController.closestNPC.GetComponent<NPCAIController>().TriggerNPCsAlert();
                    return;
                }
                if (enemyAIController.closestNPC.GetComponent<WoodcutterController>())
                {
                    enemyAIController.closestNPC.GetComponent<WoodcutterController>().TriggerNPCsAlert();
                    return;
                }
                if (enemyAIController.closestNPC.GetComponent<FarmerController>())
                {
                    enemyAIController.closestNPC.GetComponent<FarmerController>().TriggerNPCsAlert();
                    return;
                }
            }
            if (enemyAIController.InAttackRange(enemyAIController.player))
            {
                AlertEnemyNPCsAround();
                return;
            }
            enemyAIController.gotAlerted = false;
        }
```

Figure 101 – EnemyAttackPlayer.cs part 1

```
        29 references
        public void OnEnter()
        {
            enemyAIController.gotAlerted = false;
        }

        29 references
        public void OnExit() {}

        1 reference
        private void AttackBehaviour(GameObject target)
        {
            enemyAIController.timeSinceLastSawTarget = 0;              //Resets time since last saw player
            enemyAIController.GetComponent<Fighter>().Attack(target);   //agent attacks the player
        }

        //Function that alerts the enemies that are close to the enemy that is alerting
        1 reference
        private void AlertEnemyNPCsAround()
        {
            Collider[] hitColliders = null;
            hitColliders = Physics.OverlapSphere(enemyAIController.transform.position, enemyAIController.GetAlertDistance(), LayerMask.GetMask("EnemyLayer"));

            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.gameObject.tag == "EnemyNPC" && hitCollider.GetComponent<Health>().IsDead() == false)
                {
                    if (hitCollider.GetComponent<EnemyAIController>())
                    {
                        EnemyAIController enemy = hitCollider.GetComponent<EnemyAIController>();
                        if (enemy != enemyAIController)
                        {
                            enemy.GetComponent<EnemyAIController>().gotAlerted = true;
                        }
                    }
                }
            }
        }
    }
}
```

Figure 102 – EnemyAttackPlayer.cs part 2

Next state is the Suspicion state which is the *EnemySuspicion.cs* (Figure 103). *OnEnter* resets the timer since last saw target and initializes the mover script, and *OnExit* resets the *gotHitByBow*. *Tick* cancels the current action, and the enemy is just waiting to the point he last saw the player.

```
using ARPG.Core;
using ARPG.Movement;

namespace ARPG.AI
{
    2 references
    public class EnemySuspicion : IState
    {
        private readonly EnemyAIController enemyAIController;
        Mover mover;

        1 reference
        public EnemySuspicion(EnemyAIController enemyAIController)
        {
            this.enemyAIController = enemyAIController;
        }

        //Cancels the current action and the enemy does nothing
        //And as long as enemy isn't hit by a distant bow then calls the Cancel method
        //Of the mover script which stops the enemy's navMeshAgent
        29 references
        public void Tick()
        {
            enemyAIController.GetComponent<ActionScheduler>().CancelCurrentAction();
            if(!enemyAIController.gotHitByBow)
            {
                mover.Cancel();
            }
        }

        //Entering in this state, resets the timeSinceLastSawTarget
        29 references
        public void OnEnter()
        {
            enemyAIController.timeSinceLastSawTarget = 0;
            mover = enemyAIController.GetComponent<Mover>();
        }

        //Leaving this state, resets gotHitByBow
        29 references
        public void OnExit()
        {
            enemyAIController.gotHitByBow = false;
        }

    }
}
```

Figure 103 – EnemySuspicion.cs

The last state is the Guarding state which is the *EnemyGuarding.cs* (Figure 104 & 105). *OnEnter* and *OnExit* methods does nothing. *Tick* uses a waypoint system where enemy goes through a few waypoints and waits sometime before starts moving (which I will go through in the upcoming chapters) but if not, then the enemy is guarding his standing position.

```
using ARPG.Control;
using ARPG.Movement;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class EnemyGuarding : IState
    {
        private readonly EnemyAIController enemyAIController;
        private WaypointPath waypointPath;
        private float waypointDwellTime;
        private Vector3 guardPosition;
        private int currentWaypointIndex = 0;

        1 reference
        public EnemyGuarding(EnemyAIController enemyAIController, WaypointPath waypointPath, float waypointDwellTime, Vector3 guardPosition, float timeSinceArrivedAtWaypoint)
        {
            this.enemyAIController = enemyAIController;
            this.waypointPath = waypointPath;
            this.waypointDwellTime = waypointDwellTime;
            this.guardPosition = guardPosition;
        }

        //Calls PatrolBehaviour
        29 references
        public void Tick()
        {
            PatrolBehaviour();
        }

        29 references
        public void OnEnter() {}

        29 references
        public void OnExit() {}

        //Function that sets the patrol/guarding behavior of the enemy
        1 reference
        private void PatrolBehaviour()
        {
            //next positon is either going to be the guard position (standing still)
            //or the waypoints agent has to go through if he has a patrol path
            Vector3 nextPosition = guardPosition;

            if (waypointPath != null)   //If enemy has a waypoint path
            {
                if (AtIWaypoint())      //If enemy is at waypoint position
                {                       //then changes the waypoint index to the next one
                    enemyAIController.timeSinceArrivedAtWaypoint = 0;
                    NextWaypoint();
                }
                nextPosition = GetCurrentWaypoint();    //gets the position of the next waypoint
            }
```

Figure 104 – EnemyGuarding.cs part 1

```
            if (enemyAIController.timeSinceArrivedAtWaypoint > waypointDwellTime)
            {
                //moves to the next position
                enemyAIController.GetComponent<Mover>().StartMoveAction(nextPosition, enemyAIController.GetPatrolspeedFraction());
            }
        }

        //Function that tells if enemy is at waypoint
        1 reference
        private bool AtIWaypoint()
        {
            float distanceToWaypoint = Vector3.Distance(enemyAIController.transform.position, GetCurrentWaypoint());
            return distanceToWaypoint < 1f;
        }

        //Function that assigns the next waypoint
        1 reference
        private void NextWaypoint()
        {
            if (waypointPath.GetIsCyclePath() == true)
            {
                currentWaypointIndex = waypointPath.GetNextIndex(currentWaypointIndex);
            }
            else
            {
                currentWaypointIndex = waypointPath.GetNextIndexBackwards(currentWaypointIndex);
            }
        }

        //Function that returns the position of the current waypoint
        2 references
        private Vector3 GetCurrentWaypoint()
        {
            return waypointPath.GetWaypoint(currentWaypointIndex);
        }
    }
}
```

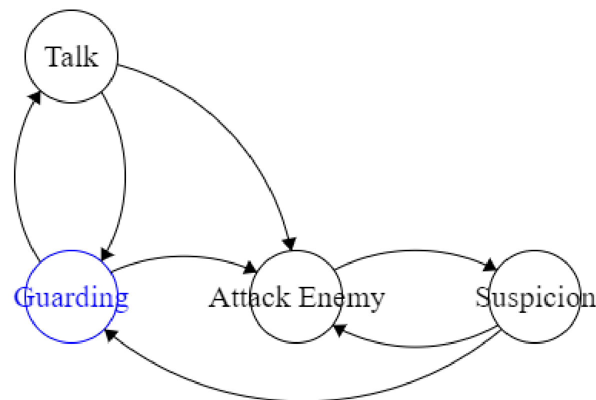
Figure 105 – EnemyGuarding.cs part 2

5.7.3 Guard AI Controller & States

5.7.3.1 Overview

Guard controller works in a similar way like the enemy's controller. Its controller contains the transitions between the states that define the behavior. Specific those scripts are:

- *GuardAIController.cs*
- State scripts:
    - *GuardAttackEnemy.cs*, defines the Attack Enemy State.
    - *GuardGuarding.cs*, defines the Guard State.
    - *GuardSuspicion.cs*, defines the Suspicion State.
    - *GuardTalk.cs*, defines the Talk State.



Looking back at Figure 13 – Guard's finite state machine – GuardAIController.cs

Guard's behavior is based on Figure 13 and in chapter 4.4.5 that behavior was analyzed and explained. Its important to note that in *GuardAIController.cs* the class inherits from the *DialogueNPC* to be able to have a dialogue, which will be analyzed on the next section.

5.7.3.2 Code

The guard controller script contains some useful methods and is responsible for setting up the transitions and logic between states. Starting in Figure 107, the variables are initialised and the transitions are set up. If the guard has a dialogue, the talk state and transitions are also added. Figures 108 and 109 contain functions used in the logic of the transitions and in the guard's states.

```csharp
using UnityEngine;
using UnityEngine.AI;
using System;
using ARPG.Resources;
using ARPG.Control;
using ARPG.Combat;

namespace ARPG.AI
{
    Unity Script | 10 references
    public class GuardController : DialogueNPC
    {
        [Header("Guarding")]
        [SerializeField] float alertDistance = 10f;         //Alert distance
        [SerializeField] float chaseDistance = 5f;          //Chase distancec
        [SerializeField] float suspicionTime = 3f;          //Suspicion time
        [SerializeField] WaypointPath waypointPath;         //Waypoint  path
        [SerializeField] float waypointDwellTime = 3f;      //Waypoint wait time
        [Range(0, 1)]
        [SerializeField] float patrolspeedFraction = 0.35f; //Patrol speed fraction

        Vector3 guardPosition;

        //Defines the player
        6 references
        public PlayerController player { get; set; }
        //Defines the closest enemy
        17 references
        public GameObject closestEnemy { get; set; }
        //Defines time since last saw target
        8 references
        public float timeSinceLastSawTarget { get; set; }
        //Defines time since arrived at waypoint
        5 references
        public float timeSinceArrivedAtWaypoint { get; set; }

        private StateMachine stateMachine;

        Health health;
```

Figure 106 – GuardAIController.cs part 1

```csharp
Unity Message | 0 references
void Awake()
{
    wantsToTalk = false;
    timeSinceLastSawTarget = Mathf.Infinity;
    timeSinceArrivedAtWaypoint = Mathf.Infinity;
    player = GameObject.FindWithTag("Player").GetComponent<PlayerController>();
    guardPosition = transform.position;
    health = GetComponent<Health>();

    var fighter = GetComponent<Fighter>();
    var navMeshAgent = GetComponent<NavMeshAgent>();
    var animator = GetComponent<Animator>();

    stateMachine = new StateMachine();

    var guarding = new GuardGuarding(this, waypointPath, waypointDwellTime, guardPosition, timeSinceArrivedAtWaypoint);
    var attackEnemy = new GuardAttackEnemy(this);
    var suspicion = new GuardSuspicion(this);
    var talk = new GuardTalk(this);

    if (GetSpeaker() != null && GetDialogue() != null)
    {
        //Guarding -> Talk
        At(guarding, talk, () => !player.GetSpeaker().GetIsInDialogue() && wantsToTalk && Vector3.Distance(player.transform.position, transform.position) < 2);
        //Talk -> Guarding
        At(talk, guarding, () => !GetSpeaker().GetIsInDialogue());
        //Talk -> Attack Enemy
        At(talk, attackEnemy, CheckIfCanAttackEnemy());
    }

    At(guarding, attackEnemy, CheckIfCanAttackEnemy());            //Guarding -> Attack Enemy
    At(attackEnemy, suspicion, SuspicionAfterEnemy());             //Attack Enemy -> Suspicion
    At(suspicion, attackEnemy, CheckToAttackEnemyAfterSuspicion()); //Suspicion -> Attack Enemy
    At(suspicion, guarding, SuspicionWait());                      //Suspicion -> Guarding

    stateMachine.SetState(guarding);        //Sets Guarding as initial state!

    void At(IState to, IState from, Func<bool> condition) => stateMachine.AddTransition(to, from, condition);

    //True if enemy attacks npcs, player is not in range, there is an npc around, enemy can attack that npc and npc is in range
    Func<bool> CheckIfCanAttackEnemy() => () => timeSinceLastSawTarget > suspicionTime && CheckForEnemiesAround() && fighter.CanAttack(closestEnemy) && InAttackRange(closestEnemy);

    //True if time since last saw target is less than suspicion time and npc is not in attack range
    Func<bool> SuspicionAfterEnemy() => () => timeSinceLastSawTarget < suspicionTime && fighter.CanAttack(closestEnemy) == false;

    //True if time since last saw target is less than suspicion time and npc is not in attack range
    Func<bool> CheckToAttackEnemyAfterSuspicion() => () => timeSinceLastSawTarget < suspicionTime && CheckForEnemiesAround() && fighter.CanAttack(closestEnemy) && InAttackRange(closestEnemy);

    //True if enemy can attack player and player is in range
    Func<bool> SuspicionWait() => () => timeSinceLastSawTarget >= suspicionTime;
}
```

Figure 107 – GuardAIController.cs part 2

```
Unity Message | 0 references
void Update()
{
    if (!health.IsDead())
    {
        stateMachine.Tick();
        UpdateTimers();
    }
}

1 reference
private void UpdateTimers()
{
    timeSinceLastSawTarget += Time.deltaTime;
    timeSinceArrivedAtWaypoint += Time.deltaTime;
}

1 reference
public float GetPatrolspeedFraction()
{
    return patrolspeedFraction;
}

1 reference
public float GetAlertDistance()
{
    return alertDistance;
}
```

Figure 108 – GuardAIController.cs part 3

```
2 references
public bool InAttackRange(GameObject target)
{
    //Calculate the distance between target and enemy
    float distanceToPlayer = Vector3.Distance(target.transform.position, transform.position);
    //If distance to player is less than the chase distance returns true
    return distanceToPlayer < chaseDistance;
}

//Function that checks for the closest enemy around
2 references
public bool CheckForEnemiesAround()
{
    if (closestEnemy != null && closestEnemy.GetComponent<Health>().IsDead() == false && closestEnemy.activeSelf == true)
    {
        return true;
    }
    Collider[] hitColliders = null;
    hitColliders = Physics.OverlapSphere(transform.position, chaseDistance, LayerMask.GetMask("EnemyLayer"));
    foreach (Collider hitCollider in hitColliders)
    {
        if (hitCollider.gameObject.tag == "EnemyNPC" && hitCollider.GetComponent<Health>().IsDead() == false)
        {
            closestEnemy = hitCollider.gameObject;
            Debug.Log("Closest npc: " + closestEnemy.name);
            return true;
        }
    }
    return false;
}
}
}
```

Figure 109 – GuardAIController.cs part 4

Moving to the guard's states and starting from the *GuardAttack.cs* script, in *Tick* method, guard is attacking the enemy and alerting the closest NPC. *OnEnter* method is alerting the closest NPC around and if the previous state was the Talk state, then the dialogue stops. *OnExit* method does nothing.

```
using ARPG.Combat;
using ARPG.QuestDialogue;
using ARPG.Resources;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class GuardAttackEnemy : IState
    {
        private readonly GuardController guardController;


        1 reference
        public GuardAttackEnemy(GuardController guardController)
        {
            this.guardController = guardController;
        }

        //Guard attacks the enemy and tries to alert the closest NPC
        29 references
        public void Tick()
        {
            if (guardController.closestEnemy.GetComponent<NPCAIController>())
            {
                guardController.closestEnemy.GetComponent<NPCAIController>().TriggerNPCsAlert();
            }
            if (guardController.closestEnemy.GetComponent<WoodcutterController>())
            {
                guardController.closestEnemy.GetComponent<WoodcutterController>().TriggerNPCsAlert();
            }
            if (guardController.closestEnemy.GetComponent<FarmerController>())
            {
                guardController.closestEnemy.GetComponent<FarmerController>().TriggerNPCsAlert();
            }

            AttackBehaviour(guardController.closestEnemy);
        }

        //On Enter if the guard was in dialogue when entered on attack enemy state,
        //Then alert the NPCs around
        29 references
        public void OnEnter()
        {
            if (guardController.GetSpeaker() != null && guardController.GetSpeaker().GetIsInDialogue())
            {
                DialogueManager.StopDialogue();
            }
            AlertNPCsAround();
        }

        29 references
        public void OnExit() { }
```

Figure 110 – GuardAttackEnemy.cs part 1

```
1 reference
private void AttackBehaviour(GameObject target)
{
    guardController.timeSinceLastSawTarget = 0;                 //Resets time since last saw player
    guardController.GetComponent<Fighter>().Attack(target);     //agent attacks the player
}

//Function that alerts the NPCs around
1 reference
private void AlertNPCsAround()
{
    Collider[] hitColliders = null;
    GameObject closestNPC;
    hitColliders = Physics.OverlapSphere(guardController.transform.position, guardController.GetAlertDistance(), LayerMask.GetMask("NPCLayer"));

    foreach (Collider hitCollider in hitColliders)
    {
        if (hitCollider.gameObject.tag == "NPC" && hitCollider.GetComponent<Health>().IsDead() == false)
        {
            closestNPC = hitCollider.gameObject;
            if (closestNPC.GetComponent<NPCAIController>())
            {
                closestNPC.GetComponent<NPCAIController>().TriggerNPCsAlert();
                continue;
            }
            if (closestNPC.GetComponent<WoodcutterController>())
            {
                closestNPC.GetComponent<WoodcutterController>().TriggerNPCsAlert();
                continue;
            }
            if (closestNPC.GetComponent<FarmerController>())
            {
                closestNPC.GetComponent<FarmerController>().TriggerNPCsAlert();
                continue;
            }
        }
    }
}
}
}
```

Figure 111 – GuardAttackEnemy.cs part 2

Next is the guarding state in *GuardGuarding.cs* script, *OnEnter* and *OnExit* methods does nothing, *Tick* is repeating the patrol behavior where if the guard has a waypoint path, then is guarding that path, else guards it’s standing position.

```
using ARPG.Control;
using ARPG.Movement;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class GuardGuarding : IState
    {
        private readonly GuardController guardController;
        private WaypointPath waypointPath;
        private float waypointDwellTime;
        private Vector3 guardPosition;
        private int currentWaypointIndex = 0;

        1 reference
        public GuardGuarding(GuardController guardController, WaypointPath waypointPath, float waypointDwellTime, Vector3 guardPosition, float timeSinceArrivedAtWaypoint)
        {
            this.guardController = guardController;
            this.waypointPath = waypointPath;
            this.waypointDwellTime = waypointDwellTime;
            this.guardPosition = guardPosition;
        }

        29 references
        public void Tick()
        {
            PatrolBehaviour();
        }

        29 references
        public void OnEnter() { }

        29 references
        public void OnExit() { }
```

Figure 112 – GuardGuarding.cs part 1

```
        1 reference
        private void PatrolBehaviour()
        {
            //next positon is either going to be the guard position (standing still)
            //or the waypoints agent has to go through if he has a patrol path
            Vector3 nextPosition = guardPosition;

            //If guard has a patrol path
            if (waypointPath != null)
            {
                if (AtIWaypoint())  //If guard is at waypoint position
                {                   //then changes the waypoint index to the next one
                    guardController.timeSinceArrivedAtWaypoint = 0;
                    NextWaypoint();
                }
                //gets the position of the next waypoint
                nextPosition = GetCurrentWaypoint();
            }

            if(guardController.timeSinceArrivedAtWaypoint > waypointDwellTime)
            {
                //moves to the next position
                guardController.GetComponent<Mover>().StartMoveAction(nextPosition, guardController.GetPatrolspeedFraction());
            }
        }

        //Function that tells if guard is at waypoint
        1 reference
        private bool AtIWaypoint()
        {
            float distanceToWaypoint = Vector3.Distance(guardController.transform.position, GetCurrentWaypoint());
            return distanceToWaypoint < 1f;
        }

        //Function that assigns the next waypoint
        1 reference
        private void NextWaypoint()
        {
            if (waypointPath.GetIsCyclePath() == true)
            {
                currentWaypointIndex = waypointPath.GetNextIndex(currentWaypointIndex);
            }
            else
            {
                currentWaypointIndex = waypointPath.GetNextIndexBackwards(currentWaypointIndex);
            }
        }

        //Function that returns the position of the current waypoint
        2 references
        private Vector3 GetCurrentWaypoint()
        {
            return waypointPath.GetWaypoint(currentWaypointIndex);
        }
    }
}
```

Figure 113 – GuardGuarding.cs part 2

Guard’s Suspicion state in *GuardSuspicion.cs* is simple as *OnExit* method does nothing, *OnEnter* resets the timer since last saw a target, resets the target, and stops the navMeshAgent. And *Tick* is only cancelling the guard’s action.

```
using ARPG.Combat;
using ARPG.Core;
using ARPG.Movement;
namespace ARPG.AI
{
    2 references
    public class GuardSuspicion : IState
    {
        private readonly GuardController guardController;

        1 reference
        public GuardSuspicion(GuardController guardController)
        {
            this.guardController = guardController;
        }

        29 references
        public void Tick()
        {
            guardController.GetComponent<ActionScheduler>().CancelCurrentAction();
        }

        //On Enter resets timeSinceLastSawTarger,
        //Resets the guard's target
        //and stops the guard's navMeshAgent
        29 references
        public void OnEnter()
        {
            guardController.timeSinceLastSawTarget = 0;
            guardController.GetComponent<Fighter>().ResetTarget();
            guardController.GetComponent<Mover>().Cancel();
        }

        29 references
        public void OnExit() { }

    }
}
```

Figure 114 – GuardSuspicion.cs

The last state is the talk state, in *GuardTalk.cs*. It's *OnEnter* method is cancelling the guard's and players current action and makes them look at each other and starts the dialogue. *OnExit* resets the *wantsToTalk* boolean and *Tick* does nothing.

```
using ARPG.Core;
using ARPG.QuestDialogue;

namespace ARPG.AI
{
    2 references
    public class GuardTalk : IState
    {
        private readonly GuardController guardController;

        1 reference
        public GuardTalk(GuardController guardController)
        {
            this.guardController = guardController;
        }

        29 references
        public void Tick() { }

        //On Enter Cancels the guard's and player's current action,
        //Both look at eachother and the dialogue begins
        29 references
        public void OnEnter()
        {
            guardController.GetComponent<ActionScheduler>().CancelCurrentAction();
            guardController.player.GetComponent<ActionScheduler>().CancelCurrentAction();
            guardController.transform.LookAt(guardController.player.transform);
            guardController.player.transform.LookAt(guardController.transform);
            DialogueManager.StartDialogue(guardController.GetDialogue(),guardController.gameObject);
        }

        //On Exit resets wantsToTalk
        29 references
        public void OnExit()
        {
            guardController.wantsToTalk = false;
        }

    }
}
```

Figure 115 – GuardTalk.cs

5.7.4 NPC AI

5.7.4.1 DialogueNPC and NPCs

5.7.4.1.1 Overview

NPCs and guards can have dialogues and player can talk with them, to achieve that I used inheritance.

- DialogueNPC (Parent Class)
    - GuardAIController.cs
    - NPCAIController
    - FarmerAIController
    - WoodcutterAIController

Instead of repeating the same piece of code for the 3 NPC types and the guard. *DialogueNPC.cs* stores information about the speaker, the dialogue and a boolean property for if the NPC wants to talk or not.

When NPCs are on Flee state, they are running towards a safe point where they hiding from the enemies for some time. This can be achieved thanks to *Safe.cs*, NPCs that are on alert can hide in a spot.

5.7.4.1.2 Code

```
using ARPG.QuestDialogue;
using UnityEngine;

namespace ARPG.AI
{
    Unity Script | 12 references
    public class DialogueNPC : MonoBehaviour
    {
        [SerializeField] private Speaker speaker;
        [SerializeField] private Dialogue dialogue;
        13 references
        public bool wantsToTalk { get; set; }

        18 references
        public Speaker GetSpeaker()
        {
            return speaker;
        }

        11 references
        public Dialogue GetDialogue()
        {
            return dialogue;
        }
    }
}
```

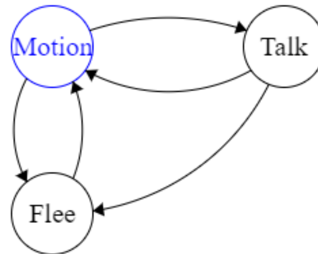
Figure 116 – DialogueNPC.cs

5.7.4.2 NPC AI Controller & States

5.7.4.2.1 Overview

Simple/Plain NPCs are controlled from NPCAIController.cs that controller sets up transitions between the states and their logic. Simple NPC's scripts are:

- *NPCAIController.cs*
- State scripts:
    - *Talk.cs*, defines the Talk State.
    - *Motion.cs*, defines the Motion State.
    - *Flee.cs*, defines the Flee State.



Looking back at Figure 9 – Simple NPC's finite state machine – NPCAIController.cs

NPC's behavior is based on Figure 9 and in chapter 4.4.1 that behavior was analyzed and explained.

5.7.4.2.2 Code

Starting from the Talk state, *OnEnter* method cancels the player's and the NPC's actions and then makes them look at each other before starting the dialogue. *OnExit* method resets the *wantsToTalk* boolean and *Tick* does nothing.

```
using ARPG.Core;
using ARPG.QuestDialogue;

namespace ARPG.AI
{
    2 references
    public class Talk : IState
    {
        private readonly NPCAIController npcAIController;
        1 reference
        public Talk(NPCAIController npcAIController)
        {
            this.npcAIController = npcAIController;
        }

        29 references
        public void Tick() { }

        //OnEnter cancels the NPC's and player's action and
        //both look at each other and the dialogue starts
        29 references
        public void OnEnter()
        {
            npcAIController.GetComponent<ActionScheduler>().CancelCurrentAction();
            npcAIController.player.GetComponent<ActionScheduler>().CancelCurrentAction();
            npcAIController.transform.LookAt(npcAIController.player.transform);
            npcAIController.player.transform.LookAt(npcAIController.transform);
            DialogueManager.StartDialogue(npcAIController.GetDialogue(),npcAIController.gameObject);
        }

        //OnExit resets wantsToTalk
        29 references
        public void OnExit()
        {
            npcAIController.wantsToTalk = false;
        }
    }
}
```

Figure 117 – Talk.cs

When in Motion state, *OnEnter* and *OnExit* methods do nothing, but *Tick* method calls *MotionBehaviour* function that is responsible for making the NPC go through the waypoints or stand still in a position.

```
using ARPG.Control;
using ARPG.Movement;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class Motion : IState
    {
        private readonly NPCAIController npcAIController;

        private WaypointPath waypointPath;
        private float waypointTolerance;
        private float waypointDwellTime;
        private float speedFraction;
        private Vector3 standPosition;

        1 reference
        public Motion(NPCAIController npcAIController, WaypointPath waypointPath, float waypointTolerance, float waypointDwellTime, float speedFraction, Vector3 standPosition)
        {
            this.npcAIController = npcAIController;
            this.waypointPath = waypointPath;
            this.waypointTolerance = waypointTolerance;
            this.waypointDwellTime = waypointDwellTime;
            this.speedFraction = speedFraction;
            this.standPosition = standPosition;
        }

        //Tick calls MotionBehabiour Function
        29 references
        public void Tick()
        {
            MotionBehaviour();
        }
        29 references
        public void OnEnter() { }

        29 references
        public void OnExit() { }
```

Figure 118 – Motion.cs part 1

```
1 reference
private void MotionBehaviour()
{
    //next positon is either going to be the guard position
    //(standing still) or the waypoints agent has to go through if he has a patrol path
    Vector3 nextPosition = standPosition;

    if (waypointPath != null)           //If agent has a patrol path
    {                                   //
        if (AtIWaypoint())              //If agent is at waypoint position
        {                               //
            NextWaypoint();             //then changes the waypoint index to the next one
        }                               //
        nextPosition = GetCurrentWaypoint();    //gets the position of the next waypoint
    }
    //moves to the next position
    npcAIController.GetComponent<Mover>().StartMoveAction(nextPosition, speedFraction);
}

//Function checks if npc is at waypoint
1 reference
private bool AtIWaypoint()
{
    float distanceToWaypoint = Vector3.Distance(npcAIController.transform.position, GetCurrentWaypoint());
    return distanceToWaypoint < waypointTolerance;
}

//Function that assigns the next waypoint
1 reference
private void NextWaypoint()
{
    if (waypointPath.GetIsCyclePath() == true)
    {
        npcAIController.currentWaypointIndex = waypointPath.GetNextIndex(npcAIController.currentWaypointIndex);
    }
    else
    {
        npcAIController.currentWaypointIndex = waypointPath.GetNextIndexBackwards(npcAIController.currentWaypointIndex);
    }
}

//Function that returns the position of the current waypoint
2 references
private Vector3 GetCurrentWaypoint()
{
    return waypointPath.GetWaypoint(npcAIController.currentWaypointIndex);
}

}
}
```

Figure 119 – Motion.cs part 2

The Flee state *OnEnter* checks if the NPC was in dialogue and if so then stops the dialogue, *OnExit* method does nothing. *Tick* is calling *FleeAndAlert* method which is responsible for moving the NPC to its safe point and alerting other NPCs around.

```
using ARPG.Movement;
using ARPG.QuestDialogue;
using ARPG.Resources;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class Flee : IState
    {
        private readonly NPCAIController npcAIController;
        1 reference
        public Flee(NPCAIController npcAIController)
        {
            this.npcAIController = npcAIController;
        }

        //Tick calls FleeAndAlert Function
        29 references
        public void Tick()
        {
            FleeAndAlert();
        }

        //OnEnter checks if the NPC is in dialogue to stop that dialogue
        29 references
        public void OnEnter()
        {
            if (npcAIController.GetSpeaker() != null && npcAIController.GetSpeaker().GetIsInDialogue())
            {
                DialogueManager.StopDialogue();
            }
        }

        29 references
        public void OnExit() { }
```

Figure 120 – Flee.cs part 1

```
        //Function that makes the NPC flee to the safe point and alerts other NPCs around
        1 reference
        private void FleeAndAlert()
        {
            npcAIController.GetComponent<Mover>().MoveTo(npcAIController.GetSafePoint().position, 1f);

            Collider[] hitColliders = null;
            GameObject closestNPC;
            hitColliders = Physics.OverlapSphere(npcAIController.transform.position, npcAIController.GetAlertDistance(), LayerMask.GetMask("NPCLayer"));

            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.gameObject.tag == "NPC" && hitCollider.GetComponent<Health>().IsDead() == false)
                {
                    closestNPC = hitCollider.gameObject;
                    if (closestNPC.GetComponent<NPCAIController>())
                    {
                        closestNPC.GetComponent<NPCAIController>().TriggerNPCsAlert();
                    }
                    if (closestNPC.GetComponent<WoodcutterController>())
                    {
                        closestNPC.GetComponent<WoodcutterController>().TriggerNPCsAlert();
                    }
                    if (closestNPC.GetComponent<FarmerController>())
                    {
                        closestNPC.GetComponent<FarmerController>().TriggerNPCsAlert();
                    }
                }
            }
        }
    }
}
```

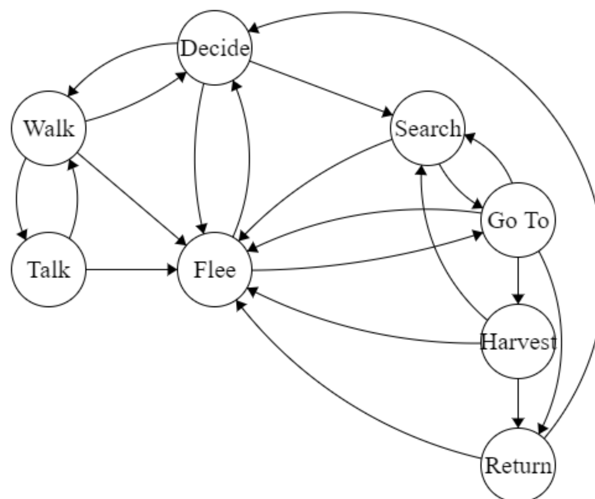
Figure 121 – Flee.cs part 2

5.7.4.3 Farmer AI Controller & States

5.7.4.3.1 Overview

Farmer's controller is the *FarmerController.cs* script, that controller sets up transitions between the states and their logic. But also contains a few useful functions like, *InitWalkingCoroutine* which starts the *Walking* coroutine, some animation events for calculations and sounds, and setters and getters, all these functions can be seen in Figures 123-127. Simple NPC's scripts are:

- *FarmerController.cs*
- State scripts:
    - *FarmerInitialDecision.cs*, defines the Decide State.
    - *FarmerSearchResource.cs*, defines the Search State.
    - *FarmerMoveToResource.cs*, defines the Go To State.
    - *FarmerHarvest.cs*, defines the Harvest State.
    - *FarmerReturnGood.cs*, defines the Return State.
    - *FarmerFlee.cs*, defines the Flee State.
    - *FarmerWalk.cs*, defines the Walk State.
    - *FarmerTalk.cs*, defines the Talk State.



Looking back at Figure 10 – Farmer's finite state machine – FarmerController.cs

NPC's behavior is based on Figure 9 and in chapter 4.4.2 that behavior was analyzed and explained.

The resource that farmers harvest is a *GatherableResource* from the *GatherableResource.cs* script. Woodcutters also harvest *GatherableResource* and in order to sort those resources, farmers got their resources on a layer called "FarmResource" and woodcutters on "WoodResource".

5.7.4.3.1 Code

First starting with the GatherableResource.cs and it is added on every resource GameObject for the farmers to harvest.

```
using UnityEngine;

namespace ARPG.AI.ResourceGathering
{
    Unity Script | 12 references
    public class GatherableResource : MonoBehaviour
    {
        //Hits to harvest
        [SerializeField] public int hits = 5;
        //If it is being harvested by another NPC already
        9 references
        public bool isOccupied { get; set; }

        Unity Message | 0 references
        private void Awake()
        {
            isOccupied = false;
        }
    }
}
```

Figure 122 – GatherableResource.cs

Moving to the FarmerController.cs, in this script is where all the states, the transitions and the logic behind them are being setted up.

```
using ARPG.AI.ResourceGathering;
using ARPG.Control;
using ARPG.Movement;
using ARPG.Resources;
using System;
using System.Collections;
using UnityEngine;
using UnityEngine.AI;
using Random = UnityEngine.Random;

namespace ARPG.AI
{
    Unity Script | 44 references
    public class FarmerController : DialogueNPC
    {
        [Header("Working")]
        [Range(0, 100)]
        [SerializeField] int workEthic = 60;                    //Farmer's work ethic
        [SerializeField] GameObject basket;                     //Basket game object
        [SerializeField] int harvestLimit = 4;                  //Harvest limit
        [SerializeField] float resourceSearchDistance = 20f;    //Resource search distance
        [SerializeField] Transform goodsReturnPoint;            //Return point of the goods
        [Header("Walking")]
        [SerializeField] WaypointPath waypointPath;             //Waypoint path for the farmer when walking
        [Range(0, 1)]
        [SerializeField] float speedFraction = 0.2f;            //Speed fraction of the farmer
        [SerializeField] float alertDistance = 10f;             //Distance that farmer is going to alert other npc's
        [SerializeField] Transform safePoint;                   //Point where farmer is going to when he is being chased
        [Header("Animation")]
        [SerializeField] AnimatorOverrideController overrideController; //Animator override controller
        [Header("Audio")]
        [SerializeField] AudioSource audioSource;
        [SerializeField] AudioClip harvestClip;

        private StateMachine stateMachine;
        private Health health;
        private Mover mover;
        private Vector3 standPosition;
        private FarmerHarvest harvest;

        6 references
        public PlayerController player { get; set; }
        4 references
        public bool doneWalking { get; set; }           //Defines whether farmer is done from walking or not
        4 references
        public bool returningFromWalk { get; set; }     //Defines if farmer is returning to the goodsReturnPoint or not
        7 references
        public bool isOnAlert { get; set; }             //Defines if farmer is on alert (being chased by enemies) or not
        8 references
        public int currentWaypointIndex { get; set; }   //Defines the waypoint index farmer is currently at
        18 references
        public GatherableResource resourceTarget { get; set; }      //Defines the gatherable resource that farmer is aiming to harvest
        3 references
        public int basketGoods { get; set; }                        //Defines the number of total harvested goods in the basket
        7 references
        public int totalHarvested { get; set; }                     //Defines the number of the total harvested goods
        9 references
        public bool carryingBasket { get; set; }                    //Defines whether farmer is carrying the basket or not
        11 references
        public GameObject droppedBasket { get; set; }               //Defines the dropped basket instance (if there is one)
```

Figure 123 – FarmerController.cs part 1

```
Unity Message | 0 references
private void Awake()
{
    wantsToTalk = false;
    doneWalking = false;
    returningFromWalk = false;
    isOnAlert = false;
    carryingBasket = false;
    droppedBasket = null;
    currentWaypointIndex = 0;
    standPosition = transform.position;
    player = GameObject.FindWithTag("Player").GetComponent<PlayerController>();
    health = GetComponent<Health>();
    mover = GetComponent<Mover>();

    var navMeshAgent = GetComponent<NavMeshAgent>();
    var animator = GetComponent<Animator>();

    stateMachine = new StateMachine();

    var initDecision = new FarmerInitialDecision(this);
    var searchResource = new FarmerSearchResource(this);
    var gotoResource = new FarmerMoveToResource(this, animator, overrideController);
    harvest = new FarmerHarvest(this, animator);
    var returnGoods = new FarmerReturnGoods(this, animator, overrideController);
    var walking = new FarmerWalk(this, waypointPath, speedFraction, standPosition);
    var flee = new FarmerFlee(this, animator);
    var talk = new FarmerTalk(this);

    //Setting up all transitions
    if (GetSpeaker() != null && GetDialogue() != null)
    {
        //Walking -> Talk
        At(walking, talk, () => !player.GetSpeaker().GetIsInDialogue() && wantsToTalk
            && Vector3.Distance(player.transform.position, transform.position) < 2);
        //Talk -> Walking
        At(talk, walking, () => !GetSpeaker().GetIsInDialogue());
    }

    At(initDecision, searchResource, () => initDecision.WorkDecision());    //Decision -> Search
    At(initDecision, walking, () => !initDecision.WorkDecision());          //Decision -> Walk
    At(walking, initDecision, () => doneWalking == true);                   //Walk -> Decision
    At(searchResource, gotoResource, GotResourceTarget());                  //Search -> Go To Resource
    At(gotoResource, returnGoods, ReturnRetrievedBasket());                 //Go To Resource -> Return Goods
    At(gotoResource, harvest, ArrivedAtResourceToHarvest());                //Go To Resource -> Harvest Goods
    At(gotoResource, searchResource, RetrievedBasketButNotEnoughGoods());   //Go To Resource -> Search Resource
    At(harvest, searchResource, NeedsMoreGoods());                          //Harvest -> Search Resource
    At(harvest, returnGoods, HarvestedEnoughGoods());                       //Harvest -> Return Goods
    At(returnGoods, initDecision, () => returnGoods.returned == true);      //Return Goods -> Decision
    At(flee, gotoResource, BasketDroppedDown());                            //Flee -> Go To Resource
    At(flee, initDecision, () => !isOnAlert);                               //Flee -> Decision
    stateMachine.AddAnyTransition(flee, () => isOnAlert);                   //All States -> Flee

    stateMachine.SetState(initDecision);
```

Figure 124 – FarmerController.cs part 2

```
    void At(IState from, IState to, Func<bool> condition) => stateMachine.AddTransition(from, to, condition);

    //True if basket is not dropped down
    Func<bool> BasketDroppedDown() => () => droppedBasket != null;

    //True if there is a resource target and no basket dropped down
    Func<bool> GotResourceTarget() => () => resourceTarget != null && droppedBasket == null;

    //True if there is no basket dropped and basket is carried, there is no resource target and there are enough goods in basket
    Func<bool> ReturnRetrievedBasket() => () => droppedBasket == null && carryingBasket == true && resourceTarget == null && basketGoods == harvestLimit;

    //True if there is no dropped basket, there is a resource target and farmer is at the harvest location
    Func<bool> ArrivedAtResourceToHarvest() => () => droppedBasket == null && resourceTarget != null && gotoResource.CheckIfArrivedAtResourceToHarvest();

    //True if there is no dropped basket and basket is being carried and there is no resource target
    Func<bool> RetrievedBasketButNotEnoughGoods() => () => droppedBasket == null && carryingBasket == true && resourceTarget == null;

    //True if just harvested , animation has finished and there are still goods needed
    Func<bool> NeedsMoreGoods() => () => harvest.hitCounter == resourceTarget.hits && totalHarvested < harvestLimit && animator.GetCurrentAnimatorStateInfo(0).IsName("Collect");

    //True if just harvested, got enough goods and animation just finished
    Func<bool> HarvestedEnoughGoods() => () => harvest.hitCounter == resourceTarget.hits && totalHarvested == harvestLimit && animator.GetCurrentAnimatorStateInfo(0).IsName("Collect");
}

Unity Message | 0 references
void Update()
{
    if (!health.IsDead())
    {
        stateMachine.Tick();
    }
}

//starts the Walking coroutine
1 reference
public void InitWalkingCoroutine()
{
    StartCoroutine(Walking());
}

2 references
public IEnumerator Walking()
{
    yield return new WaitForSeconds(20f);       //Walk time is 20 seconds
    if (GetSpeaker() == null || !GetSpeaker().GetIsInDialogue())    //has no speaker or is not in dialogue
    {
        returningFromWalk = true;                           //returningFromWalk is true so agent won't go to the next waypoint
        mover.MoveTo(goodsReturnPoint.position, .3f);       //Go to goodsReturnPoint
        //Wait till farmer goes to goodsReturnPoint
        yield return new WaitUntil(() => Vector3.Distance(transform.position, goodsReturnPoint.transform.position) <= 2);
        currentWaypointIndex = 0;                           //resets the currentWaypointIndex
        doneWalking = true;                                 //doneWalking is true so farmer can init a new decision
    }
}
```

Figure 125 – FarmerController.cs part 3

```
//Stops the Walking coroutine
1 reference
public void StopWalkingCR()
{
    StopCoroutine(Walking());
}

//Animation event
//Calculates the hits on a resource
//and adds to basket
0 references
void Harvest()
{
    harvest.hitCounter++;
    if(harvest.hitCounter == resourceTarget.hits)
    {
        totalHarvested++;
        basketGoods = totalHarvested;
        carryingBasket = true;
    }
}

//Animation event
//Plays harvest sounds
0 references
void HarvestSound()
{
    audioSource.clip = harvestClip;
    audioSource.volume = Random.Range(0.1f, 0.2f);
    audioSource.pitch = Random.Range(0.6f, 0.7f);
    audioSource.Play();
}

//alerts the NPC
8 references
public void TriggerNPCsAlert()
{
    isOnAlert = true;
}

1 reference
public int GetWorkEthic()
{
    return workEthic;
}

1 reference
public float GetResourceSearchDistance()
{
    return resourceSearchDistance;
}
```

Figure 126 – FarmerController.cs part 4

```
2 references
public Transform GetGoodsReturnPoint()
{
    return goodsReturnPoint;
}

8 references
public GameObject GetBasket()
{
    return basket;
}
1 reference
public Transform GetSafePoint()
{
    return safePoint;
}
1 reference
public float GetAlertDistance()
{
    return alertDistance;
}

//Function used to display resource search distance on the scene view
//for debugging reasons
Unity Message | 0 references
private void OnDrawGizmosSelected()
{
    Gizmos.color = Color.white;
    Gizmos.DrawWireSphere(transform.position, resourceSearchDistance);
}
}
}
```

Figure 127 – FarmerController.cs part 5

The first state is the Decision state from *FarmerInitialDecision.cs* and is *Tick*, *OnEnter* and *OnExit* methods do nothing. There is a method called *WorkDecision* that based on the work ethic of the farmer it is getting a random number between 0 and 100, if work ethic is greater than that random number then the farmer decides to work, if not then decides to walk.

```
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class FarmerInitialDecision : IState
    {
        private readonly FarmerController farmerController;

        1 reference
        public FarmerInitialDecision(FarmerController farmerController)
        {
            this.farmerController = farmerController;
        }

        //Function that picks a random number based on the work ethic
        //to decide to work or to walk
        2 references
        public bool WorkDecision()
        {
            int number = Random.Range(0, 101);
            return number < farmerController.GetWorkEthic();
        }

        29 references
        public void Tick() { }

        29 references
        public void OnEnter() { }

        29 references
        public void OnExit() { }
    }
}
```

Figure 128 – FarmerInitialDecision.cs

Search state in *FarmerSearchResource.cs* and it is responsible for finding a resource. *OnEnter* and *OnExit* methods do nothing, but *Tick* is calling *FindResourceNear* that locates a resource on “FarmResource” layer, and checks if that resource is occupied by another farmer, if it is then searches for another resource, else it doesn’t.

```
using ARPG.AI.ResourceGathering;
using System.Collections.Generic;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class FarmerSearchResource : IState
    {
        private readonly FarmerController farmerController;

        1 reference
        public FarmerSearchResource(FarmerController farmerController)
        {
            this.farmerController = farmerController;
        }

        29 references
        public void Tick()
        {
            //Trying to find a resource
            farmerController.resourceTarget = FindResourceNear();
            if (!farmerController.resourceTarget.isOccupied)
            {   //If that resource is not occupied then occupy it
                farmerController.resourceTarget.isOccupied = true;
            }
        }
        29 references
        public void OnEnter() { }
        29 references
        public void OnExit() { }
        //Function that locates a resource from around
        1 reference
        private GatherableResource FindResourceNear()
        {
            Collider[] hitColliders = null;
            List<GatherableResource> resources = new List<GatherableResource>();
            hitColliders = Physics.OverlapSphere(farmerController.transform.position,
                farmerController.GetResourceSearchDistance(),
                LayerMask.GetMask("FarmResource"));

            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.GetComponent<GatherableResource>().isOccupied == false)
                {
                    resources.Add(hitCollider.GetComponent<GatherableResource>());
                }
            }

            if (resources == null)
            {
                Debug.LogError("Not Enough Farm Resources, Add More!");
                return null;
            }
            return resources[Random.Range(0, resources.Count)];
        }
    }
}
```

Figure 129 – FarmerSearchResource.cs

Go To state from *FarmerMoveToResource.cs*, is moving the farmer close to the resource. *OnEnter* is checking if the farmer has dropped the basket (the farmer has dropped the basket of goods if was chased by an enemy) to move to the basket's position, else if has a resource then goes to that resource, enables the animator and changes the animations using am override controller. *OnExit* assigns the main animation controller again, and *Tick* method is doing nothing. There is a method used on the transition's logic, called *CheckIfArrivedAtResourceToHarvest* that checks if the farmer is close to the resource.

```
using ARPG.Movement;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class FarmerMoveToResource : IState
    {
        private readonly FarmerController farmerController;
        private Animator animator;
        AnimatorOverrideController overrideController;
        RuntimeAnimatorController MainRuntimeController;
        0 references
        public bool arrived { get; set; }
        1 reference
        public FarmerMoveToResource(FarmerController farmerController, Animator animator, AnimatorOverrideController overrideController)
        {
            this.farmerController = farmerController;
            this.animator = animator;
            this.overrideController = overrideController;
            this.MainRuntimeController = animator.runtimeAnimatorController;
        }

        29 references
        public void Tick() { }

        //OnEnter
        29 references
        public void OnEnter()
        {
            if (farmerController.droppedBasket == true) //If the basket is dropped, farmer moves to the dropped basket
            {
                farmerController.GetComponent<Mover>().MoveTo(farmerController.droppedBasket.transform.position, .2f);
                farmerController.GetBasket().SetActive(false);
            }
            else if (farmerController.resourceTarget != null)//If has a resource target then moves to the resource target
            {
                farmerController.GetComponent<Mover>().MoveTo(farmerController.resourceTarget.transform.position, .2f);
                farmerController.GetBasket().SetActive(true);
                animator.enabled = true;
                //Switches animator to the override controller
                animator.runtimeAnimatorController = overrideController;
            }
        }
```

Figure 130 – FarmerMoveToResource.cs part 1

```
        //OnExit switches animator to to the main animator controller
        29 references
        public void OnExit()
        {
            animator.runtimeAnimatorController = MainRuntimeController;
        }

        //This function checks if farmer has arrived at the resource so he can start harvesting
        1 reference
        public bool CheckIfArrivedAtResourceToHarvest()
        {
            if (farmerController.resourceTarget != null)
            {
                return Vector3.Distance(farmerController.transform.position, farmerController.resourceTarget.transform.position) <= 2;
            }
            return false;
        }
    }
}
```

Figure 131 – FarmerMoveToResource.cs part 2

Harvest state from *FarmerHarvest.cs* is making the farmer to chop the tree every two seconds till reaches a specific number of hits. *OnEnter* resets the hit counter, stops the navMesh, makes the farmer look at the resource and enables the animator, *OnExit* resets timers, the hit counter and the resource target, disables the animator and un-occupies the resource target. *Tick* method is making the farmer trigger the chopping animation every 2 seconds.

```csharp
using UnityEngine;
using UnityEngine.AI;

namespace ARPG.AI
{
    3 references
    public class FarmerHarvest : IState
    {
        private readonly FarmerController farmerController;
        private Animator animator;
        float timeSinceLastChop = Mathf.Infinity;
        float timeBetweenChops = 2f;
        6 references
        public int hitCounter { get; set; }

        1 reference
        public FarmerHarvest(FarmerController farmerController, Animator animator)
        {
            this.farmerController = farmerController;
            this.animator = animator;
        }

        //Tick, using the timeBetweenChops to harvest every 2 seconds
        29 references
        public void Tick()
        {
            if (timeSinceLastChop > timeBetweenChops)
            {
                TriggerCollect();
                timeSinceLastChop = 0;
            }
            timeSinceLastChop += Time.deltaTime;
        }

        //Function that triggers the collect animation
        1 reference
        private void TriggerCollect()
        {
            animator.ResetTrigger("stopCollect");
            animator.SetTrigger("collect");
        }
```

Figure 132 – FarmerHarvest.cs part 1

```csharp
        //OnEnter resets the hitCounter to 0
        //stops the navMesh
        //farmer looks at the resource
        //enables animator
        29 references
        public void OnEnter()
        {
            hitCounter = 0;
            farmerController.GetComponent<NavMeshAgent>().isStopped = true;
            farmerController.transform.LookAt(farmerController.resourceTarget.transform);
            farmerController.GetBasket().SetActive(true);
            animator.enabled = true;
        }

        //OnExit
        29 references
        public void OnExit()
        {
            farmerController.resourceTarget.isOccupied = false; //Un-occupies the resource target
            farmerController.resourceTarget = null;             //Resource target is null
            animator.enabled = false;                           //Disabling animator
            animator.gameObject.SetActive(false);               //Disabling animator gameObject
            animator.gameObject.SetActive(true);                //Enabling animator gameObject
            hitCounter = 0;                                     //Resets hitCounter to 0
            timeSinceLastChop = Mathf.Infinity;                 //Resets timeSinceLastChop
        }
    }
}
```

Figure 133 – FarmerHarvest.cs part 2

Return state from *FarmerReturnGoods.cs* is responsible for moving the farmer to the point that returns the harvested goods. *OnEnter* method is moving the farmer to the goods return point and changes the animator to the override controller. *OnExit* is assigning the animator controller again and *Tick* method is checking if the farmer is at the return point of the goods.

```
using ARPG.Movement;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class FarmerReturnGoods : IState
    {
        private readonly FarmerController farmerController;
        private Animator animator;
        AnimatorOverrideController overrideController;
        RuntimeAnimatorController MainRuntimeController;

        4 references
        public bool returned { get; set; }

        1 reference
        public FarmerReturnGoods(FarmerController farmerController, Animator animator, AnimatorOverrideController overrideController)
        {
            this.farmerController = farmerController;
            this.animator = animator;
            this.overrideController = overrideController;
            this.MainRuntimeController = animator.runtimeAnimatorController;
        }

        29 references
        public void Tick()
        {
            //if woodcutter is at woodReturnPoint then wood is returned
            if (Vector3.Distance(farmerController.transform.position, farmerController.GetGoodsReturnPoint().transform.position) <= 1)
            {
                returned = true;
            }
        }
        29 references
        public void OnEnter()
        {
            returned = false;                                   //Resets returned
            farmerController.carryingBasket = true;             //carryingWood is set to true
            animator.enabled = true;                            //animator enabled
            animator.runtimeAnimatorController = overrideController;    //Using the overrideController
            //Moves to the goods return point
            farmerController.GetComponent<Mover>().MoveTo(farmerController.GetGoodsReturnPoint().position, .2f);
            farmerController.GetBasket().SetActive(true);
        }
        29 references
        public void OnExit()
        {
            farmerController.carryingBasket = false;    //is no longer carrying a basket with goods
            farmerController.totalHarvested = 0;        //resets the total harvested goods
            returned = false;       //Resets returned
            //animator's runtimeController is being resetted to the main controller
            animator.runtimeAnimatorController = MainRuntimeController;
            farmerController.GetBasket().SetActive(false);//basket on hand visible
        }
    }
}
```

Figure 134 – FarmerReturnGoods.cs

Flee state from the *FarmerFlee.cs* script is responsible for making the farmer to drop the basket of goods if having one and moving to the safe point while trying to alert other NPCs. *OnEnter* is checking if farmer is in dialogue to stop it, reenables the animator, resets the resource target and if the farmer carried a basket with collected goods, then drops the basket down. *OnExit* method does nothing, and *Tick* is calling *FleeAndAlert* method where moves the farmer to the safe point and notifies any NPCs that are near.

```
using ARPG.Movement;
using ARPG.QuestDialogue;
using ARPG.Resources;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class FarmerFlee : IState
    {
        private readonly FarmerController farmerController;
        private Animator animator;

        1 reference
        public FarmerFlee(FarmerController farmerController,Animator animator)
        {
            this.farmerController = farmerController;
            this.animator = animator;
        }

        //Tick calls FleeAndAlert function
        29 references
        public void Tick()
        {
            FleeAndAlert();
        }

        //OnEnter if the farmer is in dialogue then stos the dialogue
        //reenables the animator, resets the resource target
        //If the farmer carried a basket with collected goods
        //then drops the basket
        29 references
        public void OnEnter()
        {
            if (farmerController.GetSpeaker() != null && farmerController.GetSpeaker().GetIsInDialogue())
            {
                DialogueManager.StopDialogue();
            }
            farmerController.resourceTarget = null;
            animator.enabled = false;
            animator.enabled = true;

            if (farmerController.carryingBasket)
            {
                farmerController.droppedBasket = farmerController.GetComponent<DropItem>().Drop();
                farmerController.carryingBasket = false;
                farmerController.totalHarvested = 0;
            }

            farmerController.GetBasket().SetActive(false);  //Tool is not on hand
        }
```

Figure 135 – FarmerFlee.cs part 1

```
        29 references
        public void OnExit() { }

        //Function that moves the farmer to the safe point and alert other NPCs
        1 reference
        private void FleeAndAlert()
        {
            farmerController.GetComponent<Mover>().MoveTo(farmerController.GetSafePoint().position, 1f);

            Collider[] hitColliders = null;
            GameObject closestNPC;
            //Finds npc's in the NPC Layer and triggers their alert
            hitColliders = Physics.OverlapSphere(farmerController.transform.position, farmerController.GetAlertDistance(), LayerMask.GetMask("NPCLayer"));

            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.gameObject.tag == "NPC" && hitCollider.GetComponent<Health>().IsDead() == false)
                {
                    closestNPC = hitCollider.gameObject;
                    if (closestNPC.GetComponent<NPCAIController>())
                    {
                        closestNPC.GetComponent<NPCAIController>().TriggerNPCsAlert();
                    }
                    else if (closestNPC.GetComponent<WoodcutterController>())
                    {
                        closestNPC.GetComponent<WoodcutterController>().TriggerNPCsAlert();
                    }
                    else if (closestNPC.GetComponent<FarmerController>())
                    {
                        closestNPC.GetComponent<FarmerController>().TriggerNPCsAlert();
                    }
                }
            }
        }
    }
}
```

Figure 136 – FarmerFlee.cs part 2

Lastly the Talk state from the *FarmerTalk.cs* is setting up the farmer and the player before starting the dialogue. *OnEnter* stops the farmer's walking coroutine, cancels the player's and farmer's action, makes them face each other and starts the dialogue. *OnExit* resets the *wantsToTalk* boolean and *Tick* does nothing.

```
using ARPG.Core;
using ARPG.QuestDialogue;

namespace ARPG.AI
{
    2 references
    public class FarmerTalk : IState
    {
        private readonly FarmerController farmerController;

        1 reference
        public FarmerTalk(FarmerController farmerController)
        {
            this.farmerController = farmerController;
        }

        29 references
        public void Tick() { }

        //OnEnter calls StopWalkingCR which stops the walking coroutine
        //cancels the farmer's and the player's actions and
        //makes them look at each other before starting the dialogue
        29 references
        public void OnEnter()
        {
            farmerController.StopWalkingCR();
            farmerController.GetComponent<ActionScheduler>().CancelCurrentAction();
            farmerController.player.GetComponent<ActionScheduler>().CancelCurrentAction();
            farmerController.transform.LookAt(farmerController.player.transform);
            farmerController.player.transform.LookAt(farmerController.transform);
            DialogueManager.StartDialogue(farmerController.GetDialogue(),farmerController.gameObject);
        }

        //OnExit Resets the wantsToTalk
        29 references
        public void OnExit()
        {
            farmerController.wantsToTalk = false;
        }
    }
}
```

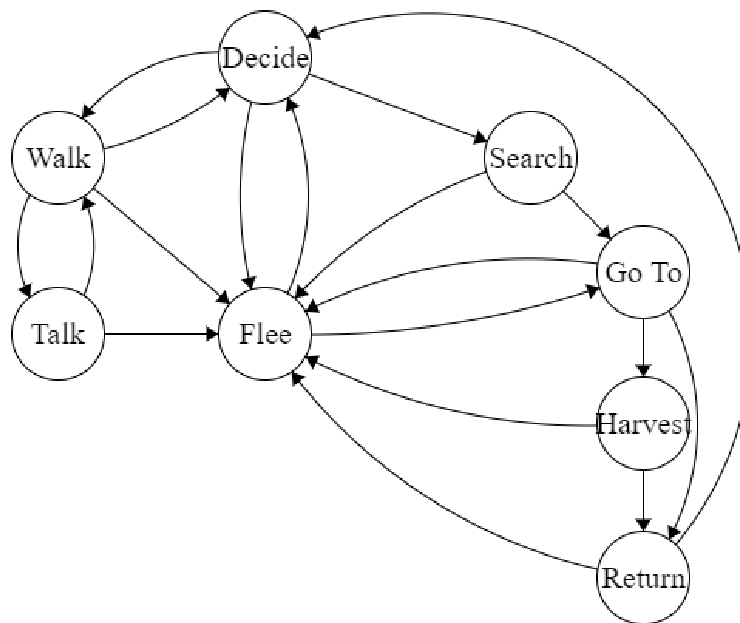
Figure 137 – FarmerTalk.cs

5.7.4.4 Woodcutter AI Controller & States

5.7.4.4.1 Overview

Woodcutter is controlled by the *WoodcutterController.cs* script, that controller sets up transitions between the states and their logic. But also contains a few useful functions like, *InitWalkingCoroutine* which starts the *Walking* coroutine, some animation events for calculations and sounds, and setters and getters, all these functions can be seen in Figures 123-127. Simple NPC's scripts are:

- *WoodcutterController.cs*
- State scripts:
    - *WoodcutterInitialDecision.cs*, defines the Decide State.
    - *WoodcutterSearchResource.cs*, defines the Search State.
    - *WoodcutterMoveToResource.cs*, defines the Go To State.
    - *WoodcutterHarvest.cs*, defines the Harvest State.
    - *WoodcutterReturnWood.cs*, defines the Return State.
    - *WoodcutterFlee.cs*, defines the Flee State.
    - *WoodcutterWalk.cs*, defines the Walk State.
    - *WoodcutterTalk.cs*, defines the Talk State.



Looking back at Figure 11 – Woodcutter's finite state machine – WoodcutterController.cs

NPC's behavior is based on Figure 11 and in chapter 4.4.3 that behavior was analyzed and explained.

The resource that woodcutter harvests is a *GatherableResource* from the *GatherableResource.cs* script attached on the tree GameObjects and with setted Layer on "WoodResource".

5.7.4.4.2 Code

Starting with the WoodcutterController.cs, in this script is where all the states, the transitions and the logic behind them are being setted up.

```
using System.Collections;
using UnityEngine;
using UnityEngine.AI;
using System;
using ARPG.Resources;
using ARPG.Control;
using ARPG.AI.ResourceGathering;
using Random = UnityEngine.Random;

namespace ARPG.AI
{
    Unity Script | 46 references
    public class WoodcutterController : DialogueNPC
    {
        [Header("Work")]
        [Range(0,100)]
        [SerializeField] int workEthic = 60;        //Woodcutters work ethic
        [SerializeField] GameObject toolOnHand;     //Defines the tool on hand (its an axe)
        [SerializeField] GameObject toolCarried;    //Defines the tool when carried on the belt
        [SerializeField] GameObject woodCarried;    //Defines the wood carried after chopping the tree
        [SerializeField] float resourceSearchDistance = 20f;    //Resource search distance
        [SerializeField] Transform woodReturnPoint; //Return point of the wood
        [Header("Walking")]
        [SerializeField] WaypointPath waypointPath;     //Waypoint path
        [SerializeField] float waypointTolerance = 1f;  //Is how close will pass through the waypoint
        [Range(0, 1)]
        [SerializeField] float speedFraction = 0.2f;    //speed fraction
        [SerializeField] float alertDistance = 10f;     //alerting distance
        [SerializeField] Transform safePoint;           //Safe point location
        [Header("Animation")]
        [SerializeField] AnimatorOverrideController overrideController; //animator override controller
        [Header("Audio")]
        [SerializeField] AudioSource audioSource;       //AudioSource component
        [SerializeField] AudioClip chopClip;            //AudioClip for the chop

        private StateMachine stateMachine;
        private Health health;
        private Vector3 standPosition;
        private WoodcutterHarvest harvest;

        4 references
        public bool doneWalking { get; set; }       //Defines if done with walking
        6 references
        public PlayerController player { get; set; }    //Defines the player
        7 references
        public bool isOnAlert { get; set; }     //Defines if is on alert
        9 references
        public int currentWaypointIndex { get; set; }   //Defines the current waypoint
        13 references
        public GatherableResource resourceTarget { get; set; }  //Defines the resource target
        8 references
        public bool carryingWood { get; set; }  //Defines if carries wood (when returning)
        10 references
        public GameObject droppedWood { get; set; } //Defines if dropped wood (when fleeing)
```

Figure 138 – WoodcutterController.cs part 1

```
Unity Message | 0 references
void Awake()
{
    wantsToTalk = false;
    doneWalking = false;
    isOnAlert = false;
    carryingWood = false;
    droppedWood = null;
    currentWaypointIndex = 0;
    player = GameObject.FindWithTag("Player").GetComponent<PlayerController>();
    standPosition = transform.position;
    health = GetComponent<Health>();

    var navMeshAgent = GetComponent<NavMeshAgent>();
    var animator = GetComponent<Animator>();

    stateMachine = new StateMachine();

    var initDecision = new WoodcutterInitialDecision(this);
    var searchResource = new WoodcutterSearchResource(this);
    var gotoResource = new WoodcutterMoveToResource(this);
    harvest = new WoodcutterHarvest(this, animator);
    var returnWood = new WoodcutterReturnWood(this, animator, overrideController);
    var walking = new WoodcutterWalk(this, waypointPath, waypointTolerance, speedFraction, standPosition);
    var flee = new WoodcutterFlee(this);
    var talk = new WoodcutterTalk(this);

    //Setting up all transitions
    if (GetSpeaker() != null && GetDialogue() != null)
    {
        //Walk -> Talk
        At(walking, talk, () => !player.GetSpeaker().GetIsInDialogue() &&
            wantsToTalk && Vector3.Distance(player.transform.position, transform.position) < 2);
        //Talk -> Walk
        At(talk, walking, () => !GetSpeaker().GetIsInDialogue());
    }

    At(initDecision, searchResource, () => initDecision.WorkDecision());    //Decision -> Search
    At(initDecision, walking, () => !initDecision.WorkDecision());          //Decision -> Walk
    At(walking, initDecision, ()=> doneWalking == true);                    //Walk -> Decision
    At(flee, gotoResource, NoWoodDroppedDown());                            //Flee -> Go To Resource
    At(searchResource, gotoResource, GotResourceTarget());                  //Search -> Go To Resource
    At(gotoResource, returnWood, ReturnRetrievedWood());                    //Go To Resource -> Return Wood
    At(gotoResource, harvest, ArrivedAtResourceToHarvest());                //Go To Resource -> Harvest Wood
    At(harvest, returnWood, GatheredWoood());                               //Harvest Wood -> Return Wood
    At(returnWood, initDecision, DeliverWood());                            //Return Wood -> Decision
    At(flee, initDecision, IsNonOnAlert());                                 //Flee -> Search
    stateMachine.AddAnyTransition(flee, () => isOnAlert);                   //All Stats -> Flee

    stateMachine.SetState(initDecision);

    void At(IState from, IState to, Func<bool> condition) => stateMachine.AddTransition(from, to, condition);
```

Figure 139 – WoodcutterController.cs part 2

```csharp
        //True if there is no wood dropped by the woodcutter
        Func<bool> NoWoodDroppedDown() => () => droppedWood != null;

        //True if woodcutter has resource target and no wood dropped down
        Func<bool> GotResourceTarget() => () => resourceTarget != null && droppedWood == null;

        //True if woodcutter pickedup his dropped wood
        Func<bool> ReturnRetrievedWood() => () => droppedWood == null && carryingWood == true && resourceTarget == null;

        //True if woodcutter is at position to harvest wood
        Func<bool> ArrivedAtResourceToHarvest() => () => droppedWood == null && carryingWood == false && gotoResource.CheckIfArrivedAtResourceToHarvest();

        //True if woodcutter has hit the tree enough times and the harvest animation is over
        Func<bool> GatheredWoood() => () => harvest.hitCounter == resourceTarget.hits;

        //True if the wood is returned
        Func<bool> DeliverWood() => () => returnWood.returned == true;

        //True if woodcutter is not on alert
        Func<bool> IsNonOnAlert() => () => !isOnAlert;
    }

    //Starts the Walking Coroutine
    1 reference
    public void InitWalkingCoroutine()
    {
        StartCoroutine(Walking());
    }

    //This function is an animation event called by the animator.
    //More specifically Hit is called on the harvest animation
    0 references
    void ChopHit()
    {
        harvest.hitCounter++;
    }

    //Animation event
    //Plays chopping sounds
    0 references
    void ChopSound()
    {
        audioSource.clip = chopClip;
        audioSource.volume = Random.Range(0.5f, 0.6f);
        audioSource.pitch = Random.Range(0.6f, 0.7f);
        audioSource.Play();
    }

    //Walks for 20 seconds
    1 reference
    public IEnumerator Walking()
    {
        yield return new WaitForSeconds(20f);
        doneWalking = true;
    }
```

Figure 140 – WoodcutterController.cs part 3

```
Unity Message | 0 references
void Update()
{
    if (!health.IsDead())
    {
        stateMachine.Tick();
    }
}

//alerts the NPC
9 references
public void TriggerNPCsAlert()
{
    isOnAlert = true;
}

1 reference
public Transform GetSafePoint()
{
    return safePoint;
}

1 reference
public float GetAlertDistance()
{
    return alertDistance;
}

7 references
public GameObject GetToolOnHand()
{
    return toolOnHand;
}

7 references
public GameObject GetToolCarried()
{
    return toolCarried;
}

7 references
public GameObject GetWoodCarried()
{
    return woodCarried;
}

1 reference
public float GetResourceSearchDistance()
{
    return resourceSearchDistance;
}

2 references
public Transform GetWoodReturnPoint()
{
    return woodReturnPoint;
}
```

Figure 141 – WoodcutterController.cs part 4

```
    1 reference
    public int GetWorkEthic()
    {
        return workEthic;
    }

    //Function used to display resource search distance on the scene view
    //for debugging reasons
    Unity Message | 0 references
    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.white;
        Gizmos.DrawWireSphere(transform.position, resourceSearchDistance);
    }
}
}
```

Figure 142 – WoodcutterController.cs part 5

Identical to the farmer, starting with the Decision state from the *WoodcutterInitialDecision.cs* script which is responsible for returning a true or false based on the work ethic of the woodcutter. Specifically, *OnEnter* makes the tool (axe) appear on the belt and makes sure the tool or wood isn't on hand. *OnExit* methods does nothing. There is a method called *WorkDecision* that based on the work ethic of the farmer it is getting a random number between 0 and 100, if work ethic is greater than that random number then the farmer decides to work, if not then decides to walk.

```
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class WoodcutterInitialDecision : IState
    {
        private readonly WoodcutterController woodcutterController;

        1 reference
        public WoodcutterInitialDecision(WoodcutterController woodcutterController)
        {
            this.woodcutterController = woodcutterController;
        }

        //Function that picks a random number based on the work ethic
        //to decide to work or to walk
        2 references
        public bool WorkDecision()
        {
            int number = Random.Range(0, 101);
            return number < woodcutterController.GetWorkEthic();
        }

        29 references
        public void Tick() { }

        29 references
        public void OnEnter()
        {
            woodcutterController.GetToolOnHand().SetActive(false); //Tool is not on hand
            woodcutterController.GetToolCarried().SetActive(true); //Tool is being carried
            woodcutterController.GetWoodCarried().SetActive(false);//Wood is being carried
        }

        29 references
        public void OnExit() { }
    }
}
```

Figure 143 – WoodcutterInitialDecision.cs

Next up is the Search state from *WoodcutterSearchResource.cs*, this state finds a resource. *OnEnter* resets the *carryingWood* boolean, makes sure that the tool is on the belt and nothing on hand. *OnExit* method does nothing. *Tick* is calling *FindResourceNear* which locates a resource in the resource search distance, and makes sure that resource is un-occupied, unless searches again.

```
using System.Collections.Generic;
using UnityEngine;
using ARPG.AI.ResourceGathering;


namespace ARPG.AI
{
    2 references
    public class WoodcutterSearchResource : IState
    {
        private readonly WoodcutterController woodcutterController;

        1 reference
        public WoodcutterSearchResource(WoodcutterController woodcutterController)
        {
            this.woodcutterController = woodcutterController;
        }

        29 references
        public void Tick()
        {   //Trying to find a resource
            woodcutterController.resourceTarget = FindResourceNear();
            //If that resource is not occupied then occupy it
            if (!woodcutterController.resourceTarget.isOccupied)
            {
                woodcutterController.resourceTarget.isOccupied = true;
            }
        }

        29 references
        public void OnEnter()
        {
            //When searching for resource, woodcutter does not carry wood
            woodcutterController.carryingWood = false;
            woodcutterController.GetToolOnHand().SetActive(false); //Tool is not on hand
            woodcutterController.GetToolCarried().SetActive(true); //Tool is being carried
            woodcutterController.GetWoodCarried().SetActive(false);//Wood is not carried
        }
```

Figure 144 – WoodcutterSearchResource.cs part 1

```
        29 references
        public void OnExit() { }

        //This function returns a GatherableResource. Searching for a resource in the WoodResource Layer,
        //adds all those resources in a list and then picks a random resource and returns it.
        1 reference
        private GatherableResource FindResourceNear()
        {
            Collider[] hitColliders = null;
            List<GatherableResource> resources = new List<GatherableResource>();
            hitColliders = Physics.OverlapSphere(woodcutterController.transform.position,
                woodcutterController.GetResourceSearchDistance(), LayerMask.GetMask("WoodResource"));

            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.GetComponent<GatherableResource>().isOccupied == false)
                {
                    resources.Add(hitCollider.GetComponent<GatherableResource>());
                }
            }
            if (resources == null)
            {
                Debug.LogError("Not Enough Resource Trees, Add More!");
                return null;
            }
            return resources[Random.Range(0, resources.Count)];
        }
    }
}
```

Figure 145 – WoodcutterSearchResource.cs part 2

Go To state from *WoodcutterMoveToResource.cs* is responsible for moving the woodcutter to the resource. *OnEnter* checks if the woodcutter has dropped wood (if was fleeing when carrying

wood then there is wood dropped down), if so then moves to that wood, goes to the resource, and makes sure the tool is on the belt while there is no tool on hand or wood. *OnExit* and *Tick* do nothing. *CheckIfArrivedAtResourceToHarvest* is used by the transitions logic to check if woodcutter is close to the resource before starting chopping/harvesting the tree.

```
using ARPG.Movement;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class WoodcutterMoveToResource : IState
    {
        private readonly WoodcutterController woodcutterController;
        0 references
        public bool arrived { get; set; }
        1 reference
        public WoodcutterMoveToResource(WoodcutterController woodcutterController)
        {
            this.woodcutterController = woodcutterController;
        }

        29 references
        public void Tick() { }

        //This function checks if woodcutter has arrived at the resource so he can start harvesting
        1 reference
        public bool CheckIfArrivedAtResourceToHarvest()
        {
            if (woodcutterController.resourceTarget != null)
            {
                return Vector3.Distance(woodcutterController.transform.position,
                    woodcutterController.resourceTarget.transform.position) <= 2.5f;
            }
            return false;
        }

        29 references
        public void OnEnter()
        {
            if (woodcutterController.droppedWood == true)//If wood is dropped, woodcutter moves to the dropped wood
            {
                woodcutterController.GetComponent<Mover>().MoveTo(woodcutterController.droppedWood.transform.position, .2f);
            }
            else if (woodcutterController.resourceTarget != null)//Else if has a resource target then moves to the resource target
            {
                woodcutterController.GetComponent<Mover>().MoveTo(woodcutterController.resourceTarget.transform.position, .2f);
            }
            woodcutterController.GetToolOnHand().SetActive(false);  //Tool is not on hand
            woodcutterController.GetToolCarried().SetActive(true);  //Tool is being carried
            woodcutterController.GetWoodCarried().SetActive(false); //Wood is not being carried
        }

        29 references
        public void OnExit() { }
    }
}
```

Figure 146 – WoodcutterMoveToResource.cs

Harvest state from *WoodcutterHarvest.cs* is responsible for triggering the woodcutter's chopping animation every 2 seconds and count the hits till it reaches the resource's hit limit. *OnEnter* makes sure the tool is on hand and not on belt, stops the navMesh, enables the animator and resets the hit counter. *OnExit* un-occupies the resource target, and resets the resource target, the animator, the hit counter and the chop timer. *Tick* method makes sure the chop animations are being triggered every 2 seconds.

```
using UnityEngine;
using UnityEngine.AI;

namespace ARPG.AI
{
    3 references
    public class WoodcutterHarvest : IState
    {
        private readonly WoodcutterController woodcutterController;
        float timeSinceLastChop = Mathf.Infinity;
        float timeBetweenChops = 2f;
        4 references
        public int hitCounter { get; set; }
        private Animator animator;

        1 reference
        public WoodcutterHarvest(WoodcutterController woodcutterController, Animator animator)
        {
            this.woodcutterController = woodcutterController;
            this.animator = animator;
        }

        29 references
        public void Tick()
        {
            if (timeSinceLastChop > timeBetweenChops)   //after timeBetweenChops
            {
                TriggerChop();              //Triggers chop animation
                timeSinceLastChop = 0;      //Resets timeSinceLastChop
            }
            timeSinceLastChop += Time.deltaTime;
        }

        //Function that triggers the chop animation
        1 reference
        private void TriggerChop()
        {
            animator.ResetTrigger("stopChop");
            animator.SetTrigger("chop");
        }
```

Figure 147 – WoodcutterHarvest.cs part 1

```
        29 references
        public void OnEnter()
        {
            hitCounter = 0;                                              //Resets hitCounter to 0
            woodcutterController.GetToolOnHand().SetActive(true);       //Tool is on hand
            woodcutterController.GetToolCarried().SetActive(false);     //Tool is not being carried
            woodcutterController.GetWoodCarried().SetActive(false);     //Wood is not being carried
            woodcutterController.GetComponent<NavMeshAgent>().isStopped = true; //navmesh stops
            //woodcutter looks at the resource target
            woodcutterController.transform.LookAt(woodcutterController.resourceTarget.transform);
            animator.enabled = true;
        }

        29 references
        public void OnExit()
        {
            woodcutterController.resourceTarget.isOccupied = false; //Un-occupies the resource target
            woodcutterController.resourceTarget = null;             //Resource target is null
            animator.enabled = false;                               //Disabling animator
            animator.gameObject.SetActive(false);                   //Disabling animator gameObject
            animator.gameObject.SetActive(true);                    //Enabling animator gameObject
            hitCounter = 0;                                         //Resets hitCounter to 0
            timeSinceLastChop = Mathf.Infinity;                     //Resets timeSinceLastChop
        }
    }
}
```

Figure 148 – WoodcutterHarvest.cs part 2

Return state from *WoodcutterReturnWood.cs*, moves the woodcutter to the return point of wood, while holding the wood and it is in *OnEnter* method, as well removing the tool from the hand, enabling it on the belt and enabling the wood on hands (so it seems the woodcutter is carrying the wood back). *OnExit* resets the animator controller to the main animator. *Tick* checks if woodcutter is at return point.

```
using ARPG.Movement;
using UnityEngine;


namespace ARPG.AI
{
    2 references
    public class WoodcutterReturnWood : IState
    {
        private readonly WoodcutterController woodcutterController;
        private Animator animator;
        AnimatorOverrideController overrideController;
        RuntimeAnimatorController MainRuntimeController;

        4 references
        public bool returned { get; set; }

        1 reference
        public WoodcutterReturnWood(WoodcutterController woodcutterController, Animator animator, AnimatorOverrideController overrideController)
        {
            this.woodcutterController = woodcutterController;
            this.animator = animator;
            this.overrideController = overrideController;
            this.MainRuntimeController = animator.runtimeAnimatorController;
        }

        29 references
        public void Tick()
        {   //if woodcutter is at woodReturnPoint then wood is returned
            if (Vector3.Distance(woodcutterController.transform.position, woodcutterController.GetWoodReturnPoint().transform.position) <= 1)
            {
                returned = true;
            }
        }

        29 references
        public void OnEnter()
        {
            returned = false;                                   //Resets returned
            woodcutterController.carryingWood = true;           //carryingWood is set to true
            animator.enabled = true;                            //animator enabled
            animator.runtimeAnimatorController = overrideController;    //Using the overrideController
            //Moves to the woodReturnPoint()
            woodcutterController.GetComponent<Mover>().MoveTo(woodcutterController.GetWoodReturnPoint().position, 0.166f);
            woodcutterController.GetToolOnHand().SetActive(false);      //Tool is not on hand
            woodcutterController.GetToolCarried().SetActive(true);      //Tool is being carried
            woodcutterController.GetWoodCarried().SetActive(true);      //Wood is being carried
        }

        29 references
        public void OnExit()
        {
            returned = false;                                           //Resets returned
            animator.runtimeAnimatorController = MainRuntimeController; //animator's runtimeController is being resetted to the main controller
        }
    }
}
```

Figure 149 – WoodcutterReturnWood.cs

Flee state from *WoodcutterFlee.cs* is responsible for making the woodcutter run to a safe point while alerting other NPCs. *OnEnter* checks if the woodcutter is in dialogue, if so then stops the dialogue, then checks if carried wood, if yes then drops it down and makes sure the tool is on belt and carries no wood on hands. *OnExit* does nothing and *Tick* method calls *FleeAndAlert* which makes the NPC go to the safe point while alerting any other close NPCs.

```
using ARPG.Movement;
using ARPG.QuestDialogue;
using ARPG.Resources;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class WoodcutterFlee : IState
    {
        private readonly WoodcutterController woodcutterController;
        1 reference
        public WoodcutterFlee(WoodcutterController woodcutterController)
        {
            this.woodcutterController = woodcutterController;
        }

        29 references
        public void Tick()
        {
            FleeAndAlert();
        }

        29 references
        public void OnEnter()
        {   //If woodcutter is in dialogue then stop the dialogue
            if (woodcutterController.GetSpeaker() != null && woodcutterController.GetSpeaker().GetIsInDialogue())
            {
                DialogueManager.StopDialogue();
            }
            if (woodcutterController.carryingWood)      //If woodcutter carries wood then drops it
            {
                woodcutterController.droppedWood = woodcutterController.GetComponent<DropItem>().Drop();
                woodcutterController.carryingWood = false;
            }

            woodcutterController.GetToolOnHand().SetActive(false);  //Tool is not on hand
            woodcutterController.GetToolCarried().SetActive(true);  //Tool is being carried
            woodcutterController.GetWoodCarried().SetActive(false); //Wood is not being carried
        }
```

Figure 150 – WoodcutterFlee.cs part 1

```
        29 references
        public void OnExit() { }

        //Function that is making the woodcutter to flee to a safe and notify other npc's
        1 reference
        private void FleeAndAlert()
        {   //Moves to the safe point
            woodcutterController.GetComponent<Mover>().MoveTo(woodcutterController.GetSafePoint().position, 1f);

            Collider[] hitColliders = null;
            GameObject closestNPC;
            //Finds npc's in the NPC Layer and triggers their alert
            hitColliders = Physics.OverlapSphere(woodcutterController.transform.position,
                woodcutterController.GetAlertDistance(), LayerMask.GetMask("NPCLayer"));

            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.gameObject.tag == "NPC" && hitCollider.GetComponent<Health>().IsDead() == false)
                {
                    closestNPC = hitCollider.gameObject;
                    if (closestNPC.GetComponent<NPCAIController>())
                    {
                        closestNPC.GetComponent<NPCAIController>().TriggerNPCsAlert();
                    }
                    if (closestNPC.GetComponent<WoodcutterController>())
                    {
                        closestNPC.GetComponent<WoodcutterController>().TriggerNPCsAlert();
                    }
                }
            }
        }
    }
}
```

Figure 151 – WoodcutterFlee.cs part 2

Walk state from *WoodcutterWalk.cs* is like all the Walk states, *OnEnter* makes sure the tool is on belt and starts the walking coroutine. *OnExit* resets the *doneWalking* boolean which means

that the woodcutter is no longer walking. *Tick* calls *MotionBehaviour* which is responsible for making the NPC to follow the waypoint path but if it doesn't have one then stands still.

```
using ARPG.Control;
using ARPG.Movement;
using UnityEngine;

namespace ARPG.AI
{
    2 references
    public class WoodcutterWalk : IState
    {
        private readonly WoodcutterController woodcutterController;
        private WaypointPath waypointPath;
        private float waypointTolerance;
        private float speedFraction;
        private Vector3 standPosition;

        1 reference
        public WoodcutterWalk(WoodcutterController woodcutterController, WaypointPath waypointPath, float waypointTolerance, float speedFraction, Vector3 standPosition)
        {
            this.woodcutterController = woodcutterController;
            this.waypointPath = waypointPath;
            this.waypointTolerance = waypointTolerance;
            this.speedFraction = speedFraction;
            this.standPosition = standPosition;
        }

        29 references
        public void Tick()
        {
            MotionBehaviour();
        }

        29 references
        public void OnEnter()
        {
            woodcutterController.GetToolOnHand().SetActive(false);  //Tool is not on hand
            woodcutterController.GetToolCarried().SetActive(true);  //Tool is being carried
            woodcutterController.GetWoodCarried().SetActive(false); //Wood is not being carried
            woodcutterController.InitWalkingCoroutine();
        }

        29 references
        public void OnExit()
        {
            woodcutterController.doneWalking = false;
        }
```

Figure 152 – WoodcutterWalk.cs part 1

```
        1 reference
        private void MotionBehaviour()
        {   //next positon is either going to be the guard position (standing still)
            //or the waypoints agent has to go through if he has a patrol path
            Vector3 nextPosition = standPosition;

            if (waypointPath != null)                   //If agent has a patrol path
            {                                           //
                if (AtIWaypoint())                      //If agent is at waypoint position
                {                                       //
                    NextWaypoint();                     //then changes the waypoint index to the next one
                }                                       //
                nextPosition = GetCurrentWaypoint();    //gets the position of the next waypoint
            }
            //moves to the next position
            woodcutterController.GetComponent<Mover>().StartMoveAction(nextPosition, speedFraction);
        }

        1 reference
        private bool AtIWaypoint()
        {
            float distanceToWaypoint = Vector3.Distance(woodcutterController.transform.position, GetCurrentWaypoint());
            return distanceToWaypoint < waypointTolerance;
        }

        1 reference
        private void NextWaypoint()
        {
            if (waypointPath.GetIsCyclePath() == true)
            {
                woodcutterController.currentWaypointIndex = waypointPath.GetNextIndex(woodcutterController.currentWaypointIndex);
            }
            else
            {
                woodcutterController.currentWaypointIndex = waypointPath.GetNextIndexBackwards(woodcutterController.currentWaypointIndex);
            }
        }

        2 references
        private Vector3 GetCurrentWaypoint()
        {
            return waypointPath.GetWaypoint(woodcutterController.currentWaypointIndex);
        }
    }
}
```

Figure 153 – WoodcutterWalk.cs part 2

Lastly the Talk state from *WoodcutterTalk.cs*. *OnEnter* cancels the woodcutter's and player's action and makes them look at each other and starts the dialogue. *OnExit* resets the *wantsToTalk* boolean and *Tick* does nothing.

```
using ARPG.Core;
using ARPG.QuestDialogue;

namespace ARPG.AI
{
    public class WoodcutterTalk : IState
    {
        private readonly WoodcutterController woodcutterController;

        public WoodcutterTalk(WoodcutterController woodcutterController)
        {
            this.woodcutterController = woodcutterController;
        }

        public void Tick() { }

        //OnEnter cancels the woodcutter's and player's action
        //Makes them look at each other and starts the dialogue
        public void OnEnter()
        {
            woodcutterController.GetComponent<ActionScheduler>().CancelCurrentAction();
            woodcutterController.player.GetComponent<ActionScheduler>().CancelCurrentAction();
            woodcutterController.transform.LookAt(woodcutterController.player.transform);
            woodcutterController.player.transform.LookAt(woodcutterController.transform);
            DialogueManager.StartDialogue(woodcutterController.GetDialogue(), woodcutterController.gameObject);
        }

        //OnExit resets the wantsToTalk
        public void OnExit()
        {
            woodcutterController.wantsToTalk = false;
        }
    }
}
```

Figure 154 – WoodcutterTalk.cs

5.8 Mover & Fighter

5.8.1 Overview

As it was seen on previous chapters, NPCs (friendly and enemies) are using a few methods, like MoveTo and Attack, which are responsible for moving the NPC to a given Transform and to set target for combat, those methods are in Mover.cs and Fighter.cs. Moreover, Mover.cs contains useful methods that are related to an NPCs movement and the update of speed for motion animations, while Fighter.cs is responsible for combat stuff, like attacking, attacking animations and more.

Specifically, some important methods in Mover.cs are:

- *MoveTo*, which moves the NPC to a transform
- *UpdateAnimator*, is updating the parameter "forwardSpeed" of the animator of the agent with the agent's speed to have to correct motion animation based on the Locomotion blend tree.

Some important methods in Fighter.cs are:

- *Attack*, which sets the given target as target

- *CanAttack*, which checks if the agent can attack the given target.

Every important method has comments for what it is responsible for, for Mover.cs are figures 155 & 156 and for Fighter.cs are figures 157-160.

5.8.2 Code

```
using UnityEngine;
using UnityEngine.AI;
using ARPG.Core;
using ARPG.Resources;
using Random = UnityEngine.Random;

namespace ARPG.Movement
{
    Unity Script | 23 references
    public class Mover : MonoBehaviour, IAction
    {
        [SerializeField] private float maxSpeed = 6f;
        [SerializeField] private AudioSource movementAudioSource;
        [SerializeField] private AudioClip runningClip;
        [SerializeField] private AudioClip walkingClip;
        [SerializeField] private AudioClip[] steps;

        private NavMeshAgent navMeshAgent;
        private Health health;

        Unity Message | 0 references
        private void Awake()
        {
            navMeshAgent = GetComponent<NavMeshAgent>();
            health = GetComponent<Health>();
        }

        Unity Message | 0 references
        void Update()
        {   //If agent is not dead keeps navmeshagent enabled
            navMeshAgent.enabled = !health.IsDead();
            UpdateAnimator();
        }

        //Function that moves to destination
        5 references
        public void StartMoveAction(Vector3 destination, float speedFraction)
        {
            GetComponent<ActionScheduler>().StartAction(this);  //Starts move action
            MoveTo(destination, speedFraction);                 //Moves agent to destination
        }

        //Function that moves Agent to the destination
        12 references
        public void MoveTo(Vector3 destination, float speedFraction)
        {
            navMeshAgent.destination = destination;
            navMeshAgent.speed = maxSpeed * Mathf.Clamp01(speedFraction);
            navMeshAgent.isStopped = false;
        }
```

Figure 155 – Mover.cs part 1

```
        //Function that stops agent's navMeshAgent  (Implemented Function from IAction)
        8 references
        public void Cancel()
        {
            navMeshAgent.isStopped = true;
        }

        //UpdateAnimator is getting the agent's velocity, then converts it to local velocity
        //And then sets the animator's "forwardSpeed" as the speed
        1 reference
        private void UpdateAnimator()
        {
            Vector3 velocity = navMeshAgent.velocity;
            Vector3 localVelocity = transform.InverseTransformDirection(velocity);
            float speed = localVelocity.z;
            GetComponent<Animator>().SetFloat("forwardSpeed", speed);
        }

        //Animation event that once called plays the step sounds
        0 references
        void Step()
        {
            movementAudioSource.clip = steps[Random.Range(0, steps.Length)];
            movementAudioSource.volume = Random.Range(0.4f, 0.6f);
            movementAudioSource.pitch = Random.Range(0.8f, 1.1f);
            movementAudioSource.Play();
        }
    }
}
```

Figure 156 – Mover.cs part 2

```
using UnityEngine;
using ARPG.Movement;
using ARPG.Core;
using ARPG.Resources;
using ARPG.Stats;
using ARPG.Resources.Items;
using Random = UnityEngine.Random;

namespace ARPG.Combat
{
    Unity Script | 10 references
    public class Fighter : MonoBehaviour, IAction
    {
        [SerializeField] float timeBetweenAttacks = 1f;     //Time before agent can attack again
        [SerializeField] Transform rightHandTransform = null;   //Right hand transform
        [SerializeField] Transform leftHandTransform = null;    //Left hand transform
        [SerializeField] WeaponItem defaultWeapon = null;       //the default weapon
        [SerializeField] AudioSource combatAudioSource;         //Audio source
        [SerializeField] AudioClip swingClip;                   //swing audio clip
        [SerializeField] AudioClip punchClip;                   //punch audio clip
        [SerializeField] AudioClip projectileReleaseClip;       //projectile release audio clip

        Health target;
        float timeSinceLastAttack = Mathf.Infinity;
        WeaponItem currentWeapon = null;

        Unity Message | 0 references
        private void Start()
        {
            if (currentWeapon == null)
            {
                EquipWeapon(defaultWeapon);
            }
        }

        Unity Message | 0 references
        private void Update()
        {
            timeSinceLastAttack += Time.deltaTime;      //Increasing the time since agent's last attack

            if (target == null) return;                 //If agent has not target then skip the rest
            if (target.IsDead()) return;                //If agent's target is dead then skip the rest

            if (!GetIsInRange())                        //If target is not in weapon's attack range
            {
                GetComponent<Mover>().MoveTo(target.transform.position, 1f); //Moves agent to target
            }
            else //Else if target is in weapon's attack range
            {
                GetComponent<Mover>().Cancel(); //Cancels agent's move
                //Makes agent to look at the target and through
                //the attack animation the Hit() applies damage to the target
                AttackBehaviour();
            }
        }
```

Figure 157 – Fighter.cs part 1

```
//Function that equips the given weaponItem
1 reference
public void EquipWeapon(WeaponItem weaponItem)
{
    currentWeapon = weaponItem;
    Animator animator = GetComponent<Animator>();
    weaponItem.Spawn(rightHandTransform, leftHandTransform, animator);
}

//Function that returns target's Health component
0 references
public Health GetTarget()
{
    return target;
}

//Function that makes the agent look at it's target and !!!attack!!!
1 reference
private void AttackBehaviour()
{   //Makes agent to look at target
    transform.LookAt(target.transform.position);
    //If the time since agent last attacked is more than the time
    //betweeen attacks then triggers the agents
    //attack animation and resets the timer
    if (timeSinceLastAttack > timeBetweenAttacks)
    {
        TriggerAttack();          //This will trigger the Hit() event
        timeSinceLastAttack = 0;  //Resets attack timer

    }
}

//Function that triggers annimations
1 reference
private void TriggerAttack()
{
    GetComponent<Animator>().ResetTrigger("stopAttack"); //Resets stop attack
    GetComponent<Animator>().SetTrigger("attack");       //Triggers attack
}
```

Figure 158 – Fighter.cs part 2

```
//Animation event & this function will trigger when the agent lands a hit on animation !!!
1 reference
void Hit()
{
    if (target == null)  //If agent's target is null
    {
        return;         //Then skip
    }

    float baseDamage = GetComponent<BaseStats>().GetStat(Stat.Damage);
    //Damage is calculated by adding the base damage, weapon extra damage and weapon damage
    float damage = baseDamage + (baseDamage * currentWeapon.GetWeaponExtraPercentageDamage()) + currentWeapon.GetDamage();

    if (currentWeapon.HasProjectile())
    {
        currentWeapon.LaunchProjectile(rightHandTransform, leftHandTransform, target, gameObject, damage);
    }
    else
    {
        if (currentWeapon.name == "Unarmed")
        {
            combatAudioSource.clip = punchClip;
        }
        else
        {
            combatAudioSource.clip = swingClip;
        }

        combatAudioSource.volume = Random.Range(0.8f, 1);
        combatAudioSource.pitch = Random.Range(0.8f, 1.1f);
        combatAudioSource.Play();
        target.TakeDamage(gameObject, damage);//Agent's target takes the weapon damage
    }
}

//Animation event calling Hit and playing the projectile release sound
0 references
void Shoot()
{
    Hit();
    combatAudioSource.clip = projectileReleaseClip;
    combatAudioSource.volume = Random.Range(0.8f, 1);
    combatAudioSource.pitch = Random.Range(0.8f, 1.1f);
    combatAudioSource.Play();
}

//Function that returns true or false accordingly if agent is in weapon's range to attack with it's target
1 reference
private bool GetIsInRange()
{
    return Vector3.Distance(transform.position, target.transform.position) < currentWeapon.GetRange();
}
```

Figure 159 – Fighter.cs part 3

```
//Function that returns true if combat target is alive and false if it
11 references
public bool CanAttack(GameObject combatTarget)
{   //If combat target is dead then return false
    if (combatTarget == null || combatTarget.activeSelf == false)
    {
        return false;
    }
    Health targetToTest = combatTarget.GetComponent<Health>();
    //If combat target exists and it's not dead return true, else false
    return targetToTest != null && !targetToTest.IsDead();
}

//Function that sets agent's attack target as the given combatTarget
4 references
public void Attack(GameObject combatTarget)
{
    GetComponent<ActionScheduler>().StartAction(this);  //Starts attack action
    target = combatTarget.GetComponent<Health>();       //Sets target as the given combat target
}

//Function that cancels attack
8 references
public void Cancel()
{
    StopAttack();       //Stops the attack animation
    target = null;      //Sets agent's target to nothing
}

//Function that stops the attack animation
1 reference
private void StopAttack()
{
    GetComponent<Animator>().ResetTrigger("attack"); //Resets attack trigger
    GetComponent<Animator>().SetTrigger("stopAttack"); //Triggers the stopAttack trigger
}

1 reference
public void ResetTarget()
{
    target = null;
}
}
}
```

Figure 160 – Fighter.cs part 4

5.9 Items, Weapon items & Consumables

5.9.1 Overview

Items are divided into 2 categories, weapon items and consumable items. Also, items (weapon items and consumable items) are scriptable objects that can be easily created and manipulated. That is, the player can obtain a new weapon by picking it up. This can be achieved through the *Pickup.cs* and the scriptable object weapon item. For each sword, there is a GameObject that contains the Pickup.cs and the Weapon Item scriptable object and is placed in the world for the player to find and pick up. After picking it up, the player equips the weapon (and adds it to the inventory, but that's chapter 5.11) and it's ready to use. Exactly the same is true for consumable items (i.e. for each consumable item there are GameObjects to pick up).

Weapons can be found around the scene or can be dropped for the player to pick them up. These are the weapons:

- Unarmed, deals 2 damage and 0 extra damage percentage.
- Arming Sword, deals 25 damage and 30% extra damage percentage.
- Epilogue Sword, deals 200 damage and 90% extra damage percentage.
- Fury Sword, deals 40 damage and 50% extra damage percentage.

- Heavy Fall Sword, deals 45 damage and 40% extra damage percentage.
- Roar Sword, deals 25 damage and 20% extra damage percentage.
- Short Sword 1, deals 20 damage and 20% extra damage percentage.
- Short Sword 2, deals 10 damage and 20% extra damage percentage.
- The Lost Crusader Sword, deals 250 damage and 70% extra damage percentage.
- Bow, deals 15 damage and 20% extra damage percentage.
- Magic, deals 40 damage and 50% extra damage percentage.

Each weapon has damage and damage percentage, as player, guards and enemies have a base damage based on their level. When player attacks an enemy, the damage output gets calculated by the following formula:

$$base\ damage + ((base\ damage * weapon's\ extra\ damage\ percentage) + weapon\ damage)$$

Consumable items are used to restore health and are placed around the scene for the player to pick them up and use them when in need. These are the consumable items:

- Apple, restores 50 health points.
- Honey, restores 100 health points.
- Lesser Health Potion, restores 150 health points.
- Health Potion, restores 200 health points.

Scripts of this chapter:

- *Item.cs*, which is a scriptable object containing information.
    - *WeaponItem.cs*, is a scriptable object inheriting from Item and having more specific information, like damage amount.
    - *ConsumableItem.cs*, is a scriptable object inheriting from Item and having more specific information like, health restoration amount.
- *Pickup.cs*, is attached to pickup GameObjects and upon colliding with the player, considering the pickup gives a WeaponItem or a ConsumableItem. Also, a dialogue can be added for when player picks it up, the dialogue starts.
- *DropItem.cs*, is used by the Farmer and the Woodcutter on the Flee state to drop the basket and the wood.
- *ItemPickUp.cs*, is used by the farmer and the Woodcutter to pick-up the dropped wood and basket of goods, after they dropped it because of Flee state.
- *DropItemOnDeath.cs*, drops the pickup GameObject that was given upon death, it is used for the storyline for the player to get specific weapon.

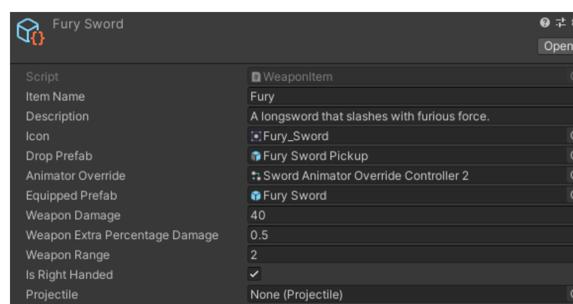


Figure 161 – The “Fury” WeaponItem scriptable object.

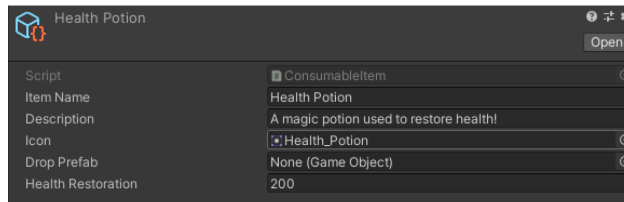


Figure 162 – The “Health Potion” ConsumableItem scriptable object.

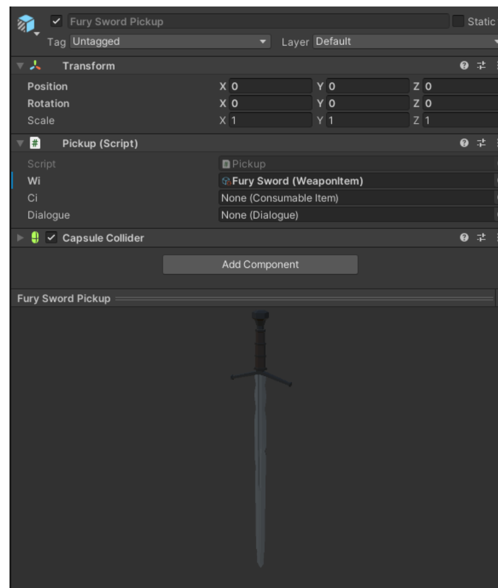


Figure 163 – The “Fury” sword pickup and its components.

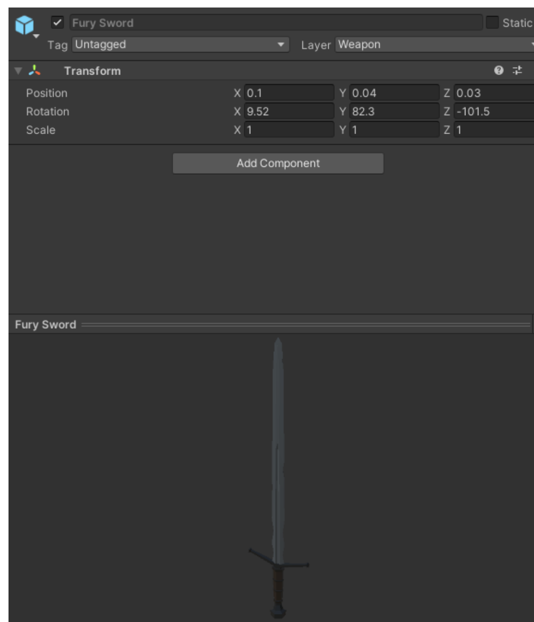


Figure 164 – The “Fury” sword that is used when it’s on an agent’s hands.

5.9.2 Code

```
using UnityEngine;

namespace ARPG.Resources.Items
{
    Unity Script | 8 references
    public class Item : ScriptableObject
    {
        //Item's name
        [SerializeField] public string itemName = "";
        //Item's description
        [SerializeField] public string description = "";
        //Item's icon
        [SerializeField] public Sprite icon = null;
        //!!! Items pick-up GameObject !!!
        [SerializeField] public GameObject dropPrefab = null;
    }
}
```

Figure 165 – Item.cs

```
using UnityEngine;

namespace ARPG.Resources.Items
{
    [CreateAssetMenu(fileName = "Consumable Item", menuName = "Item/Make A New Consumable Item", order = 1)]
    Unity Script | 9 references
    public class ConsumableItem : Item
    {
        //Health restoration amount
        [SerializeField] float healthRestoration;

        2 references
        public float GetHealthRestoration()
        {
            return healthRestoration;
        }
    }
}
```

Figure 166 – ConsumableItem.cs

```
using ARPG.Combat;
using UnityEngine;

namespace ARPG.Resources.Items
{
    [CreateAssetMenu(fileName = "Weapon Item", menuName = "Item/Make A New Weapon Item", order = 0)]
    Unity Script | 17 references
    public class WeaponItem : Item
    {
        //Animator override controller, containing animations to override the current ones
        [SerializeField] AnimatorOverrideController animatorOverride = null;
        //!!! The game object used to be instantiated when equiping a weapon !!!
        [SerializeField] GameObject equippedPrefab = null;
        //Weapon damage
        [SerializeField] float weaponDamage = 5f;
        //Weapon extra damage
        [SerializeField] float weaponExtraPercentageDamage = 0.2f;
        //Weapon range
        [SerializeField] float weaponRange = 2f;
        //Is it right handed?
        [SerializeField] bool isRightHanded = true;
        //Does it have projectile (used for bow, with arrow projectile)
        [SerializeField] Projectile projectile = null;
        //Is it equiped?
        [HideInInspector] public bool isEquiped = false;

        const string weaponName = "Weapon";
        2 references
        public void Spawn(Transform rightHand, Transform leftHand, Animator animator)
        {   //Destroys old weapon, if there is one
            DestroyOldWeapon(rightHand, leftHand);
            //If there is an equipped prefab (for example a sword for the sword weapon)
            if (equippedPrefab != null)
            {
                //set's the transform that the weapon is going to be at
                Transform handTransform = GetTransform(rightHand, leftHand);
                //Instantiates the  equipped prefab at the correct hand transform
                GameObject weapon = Instantiate(equippedPrefab, handTransform);
                //Set's the weapon gameobject name to the weapon's name
                weapon.name = weaponName;
                weapon.tag = "Weapon";
                weapon.transform.GetChild(0).tag = "Weapon";
                weapon.SetActive(true);
            }
```

Figure 167 – WeaponItem.cs part 1

```
        var overrideController = animator.runtimeAnimatorController as AnimatorOverrideController;

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

    //Function that destroys old equipped weapon
    private void DestroyOldWeapon(Transform rightHand, Transform leftHand)
    {   //Gets the transform of the weapon on right hand
        Transform oldWeapon = rightHand.Find(weaponName);
        if (oldWeapon == null)  //If it doesnt exist on the right hand
        {   //Then it should be on left hand and gets its transform
            oldWeapon = leftHand.Find(weaponName);
        }
        if (oldWeapon == null) return; //If it is still null then return

        oldWeapon.name = "DESTROYING";  //Renames
        Destroy(oldWeapon.gameObject);  //Destroys the old weapon
    }

    //Function that returns the transform that the weapon is going to be at
    private Transform GetTransform(Transform rightHand, Transform leftHand)
    {
        Transform handTransform;
        if (isRightHanded)              //If the weapon is right handed
        {
            handTransform = rightHand;  //Returns the right hand's transform
        }
        else                            //Else the weapon is left handed
        {
            handTransform = leftHand;   //Returns the left hand's transform
        }

        return handTransform;
    }
```

Figure 168 – WeaponItem.cs part 2

```
    //Function that launches projectile (used by NPCs)
    public void LaunchProjectile(Transform rightHand, Transform leftHand, Health target, GameObject attacker, float damage)
    {
        Projectile projectileInstance = Instantiate(projectile, GetTransform(rightHand, leftHand).position, Quaternion.identity);
        projectileInstance.SetTarget(target, attacker, damage);
    }

    //Function that launches projectile (used by Player)
    public void LaunchProjectile(Transform rightHand, Transform leftHand, GameObject attacker, float damage, RaycastHit raycastHit)
    {
        Projectile projectileInstance = Instantiate(projectile, GetTransform(rightHand, leftHand).position, Quaternion.identity);
        projectileInstance.SetTarget(attacker, damage, raycastHit);
    }

    public bool HasProjectile()
    {
        return projectile != null;
    }

    public float GetDamage()
    {
        return weaponDamage;
    }

    public float GetWeaponExtraPercentageDamage()
    {
        return weaponExtraPercentageDamage;
    }

    public float GetRange()
    {
        return weaponRange;
    }

    public bool GetIsRightHanded()
    {
        return isRightHanded;
    }
}
}
```

Figure 169 – WeaponItem.cs part 3

```
using ARPG.QuestDialogue;
using ARPG.Resources;
using ARPG.Resources.Items;
using System.Collections;
using UnityEngine;

namespace ARPG.Combat
{
    Unity Script | 1 reference
    public class Pickup : MonoBehaviour
    {
        [SerializeField] WeaponItem wi = null;
        [SerializeField] ConsumableItem ci = null;
        [SerializeField] Dialogue dialogue = null;

        Unity Message | 0 references
        private void OnTriggerEnter(Collider other)
        {
            if (other.gameObject.tag == "Player")
            {
                Inventory playerInventory = other.GetComponent<Inventory>();
                if (!playerInventory.IsInventoryFull())          //If inventory isn't full then pick up the item
                {
                    if (wi != null)                              //If pickup is WeaponItem then add that to the inventory
                    {
                        playerInventory.addToInventory(wi);
                    }
                    if (ci != null)                              //If pickup is ConsumableItem then add that to the inventory
                    {
                        playerInventory.addToInventory(ci);
                    }
                    if (dialogue != null)                        //If pickup has a dialogue then start coroutine
                    {
                        StartCoroutine(DialogueAfterPickup());
                        return;
                    }
                    Destroy(gameObject);
                }
            }
        }
```

Figure 170 – Pickup.cs part 1

```
        //Coroutine that starts the dialogue only if player is not near enemies
        1 reference
        private IEnumerator DialogueAfterPickup()
        {
            transform.GetChild(0).gameObject.SetActive(false);
            yield return new WaitUntil(()=> !CheckForEnemiesAround());  //Waits until there are no enemies close to the player
            DialogueManager.StartDialogue(dialogue);                    //Starts the dialogue
            Destroy(gameObject);
        }

        //Function that returns true if there are enemies close to player
        1 reference
        public bool CheckForEnemiesAround()
        {
            Collider[] hitColliders = null;
            hitColliders = Physics.OverlapSphere(GameObject.FindGameObjectWithTag("Player").transform.position, 10f, LayerMask.GetMask("EnemyLayer"));
            foreach (Collider hitCollider in hitColliders)
            {
                if (hitCollider.gameObject.tag == "EnemyNPC" && hitCollider.GetComponent<Health>().IsDead() == false)
                {
                    return true;
                }
            }
            return false;
        }

        //Function that removes the dialogue. Used when player drops an item from the inventory
        //so that when it gets picked up again no dialogue appears.
        1 reference
        public void RemoveDialogue()
        {
            dialogue = null;
        }
    }
}
```

Figure 171 – Pickup.cs part 2

```
using UnityEngine;

namespace ARPG.Resources
{
    //DropItemOnDeath is used for the storyline, in order for
    //an NPC to drop a specific sword pick-up gameobject for the player after dying.
    //But it can be used on every npc to drop specific items upon dying.
    Unity Script | 3 references
    public class DropItemOnDeath : MonoBehaviour
    {
        [SerializeField] public GameObject item;

        1 reference
        public void Drop()
        {
            Instantiate(item, transform.position, Quaternion.identity);
        }
    }
}
```

Figure 172 – DropItemOnDeath.cs

```
using ARPG.AI;
using UnityEngine;

namespace ARPG.Resources
{
    //ItemPickUp is used by the NPC's (farmer & woodcutter) to pick up the
    //wood and basket of goods after they dropped it because of Flee state.
    Unity Script | 0 references
    public class ItemPickUp : MonoBehaviour
    {
        Unity Message | 0 references
        private void OnTriggerEnter(Collider other)
        {
            //If(NPC collides and its a woodcutter npc and its not on alert
            //and the dropped wood of the npc is this gameobject
            //then picks up the wood again and destroys this gameobject
            if (other.tag == "NPC" && other.GetComponent<WoodcutterController>() &&
                other.GetComponent<WoodcutterController>().isOnAlert == false &&
                other.GetComponent<WoodcutterController>().droppedWood == gameObject)
            {
                WoodcutterController temp = other.GetComponent<WoodcutterController>();
                temp.droppedWood = null;
                temp.carryingWood = true;
                Destroy(gameObject);
            }
            //If(NPC collides and its a farmer npc and its not on alert
            //and the dropped basket of goods of the npc is this gameobject
            //then picks up the basket again and destroys this gameobject
            else if (other.tag == "NPC" && other.GetComponent<FarmerController>() &&
                other.GetComponent<FarmerController>().isOnAlert == false &&
                other.GetComponent<FarmerController>().droppedBasket == gameObject)
            {
                FarmerController temp = other.GetComponent<FarmerController>();
                temp.droppedBasket = null;
                temp.carryingBasket = true;
                temp.totalHarvested = temp.basketGoods;
                temp.GetBasket().SetActive(true);
                Destroy(gameObject);
            }
        }
    }
}
```

Figure 173 – ItemPickUp.cs

```
using UnityEngine;

namespace ARPG.Resources
{
    //DropItem is used on NPCs (farmer and woodcutter) to drop thep wood and basket
    //when entering on Flee state using the Drop() method.
    Unity Script | 2 references
    public class DropItem : MonoBehaviour
    {
        [SerializeField] GameObject item;

        2 references
        public GameObject Drop()
        {
            GameObject instance = Instantiate(item, transform.position, Quaternion.identity);
            instance.SetActive(true);
            return instance;
        }
    }
}
```

Figure 174 – DropItem.cs

5.10 Projectiles

5.10.1 Overview

Other that the sword weapons there is another weapon for the player to use, that weapon is the bow. Bows are shooting arrow projectiles, so when player or the enemy shoots with the bow it needs to instantiate an arrow GameObject from the weapon to the position where player clicked or for the location of a target if it’s the enemy who is using the bow, this can be achieved thanks to the *Projectile.cs*.

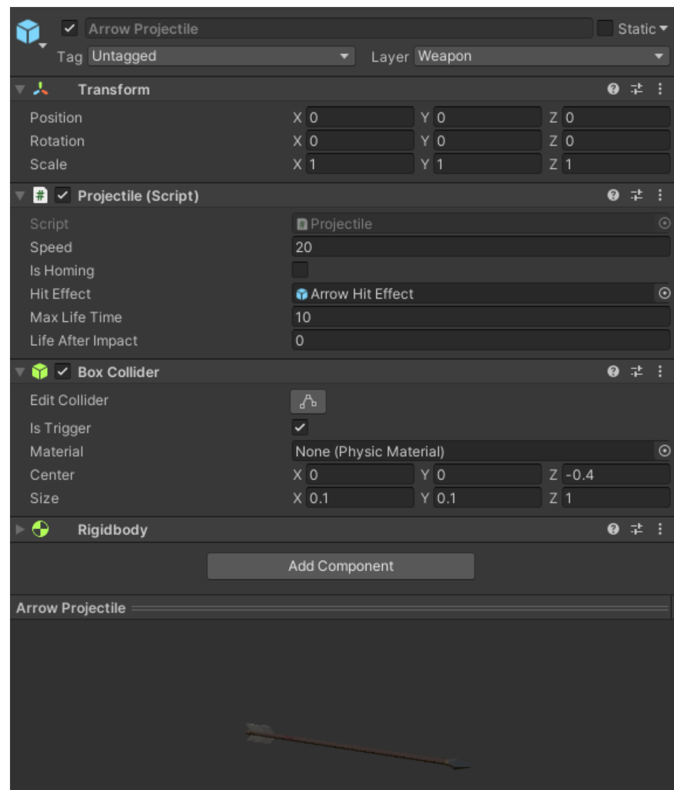


Figure 175 – The arrow projectile GameObject and its components.

As seen in figure 180, the GameObject has the Projectile.cs attached, as well as a Rigidbody to control the projectile GameObject's position through physics and a Box Collider so it can collide with objects.

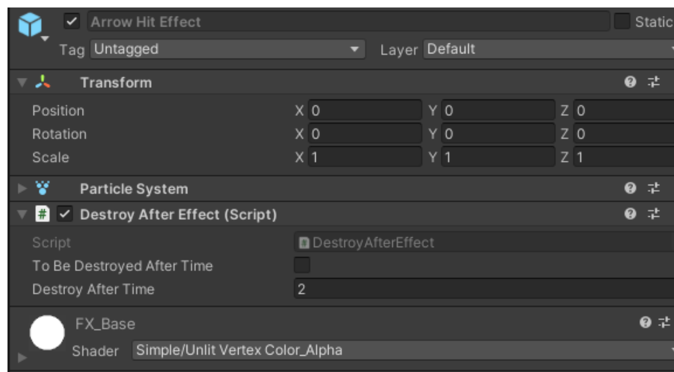


Figure 176 – The arrow hit effect GameObject and its components.

Lastly in figure 181, when the projectile GameObject collides with an object that has a Particle System attached that is instantiated when the projectile collides with an object. That GameObject has the *DestroyAfterEffect.cs* which is either destroys the GameObject when particle system stops or after a given time.

5.10.2 Code

```
using UnityEngine;

namespace ARPG.Core
{
    public class DestroyAfterEffect : MonoBehaviour
    {
        [SerializeField] bool toBeDestroyedAfterTime = false;
        [SerializeField] float destroyAfterTime = 2f;

        void Update()
        {
            //If the particle system stops then destroy the gameobject or destroys after time
            if (!GetComponent<ParticleSystem>().IsAlive() || toBeDestroyedAfterTime)
            {
                if (toBeDestroyedAfterTime)
                {
                    Destroy(gameObject, destroyAfterTime);
                }
                else
                {
                    Destroy(gameObject);
                }
            }
        }
    }
}
```

Figure 177 – DestroyAfterEffect.cs

```
using ARPG.AI;
using ARPG.Core;
using ARPG.Resources;
using UnityEngine;

namespace ARPG.Combat
{
    public class Projectile : MonoBehaviour
    {
        [SerializeField] float speed = 1;
        [SerializeField] bool isHoming = true;
        [SerializeField] GameObject hitEffect = null;
        [SerializeField] float maxLifeTime = 10;
        [SerializeField] float lifeAfterImpact = 2;

        Health target = null;
        GameObject attacker = null;
        float damage = 0;
        RaycastHit raycastHit;

        private void Start()
        {
            Physics.IgnoreCollision(GetComponent<Collider>(), attacker.GetComponent<Collider>());   //Ignores collision between the projectile and the attacker
            if (attacker.tag == "Player")   //If player is the attacker then sets the rotation and rigidbody's velocity at the raycastHit point
            {
                GetComponent<Rigidbody>().velocity = (raycastHit.point - transform.position).normalized * speed;
                transform.rotation = Quaternion.LookRotation(GetComponent<Rigidbody>().velocity);
                return;
            }
            transform.LookAt(GetAimLocation());     //Else if attacker is not player, projectile looks towards the target
        }

        private void Update()
        {
            if (attacker.tag != "Player")                   //If player is not the attacker
            {
                if (target == null) return;                 //And target is null, return
                if (isHoming && !target.IsDead())           //If projectile is homing and the target is not dead then the projectile follows the target
                {
                    transform.LookAt(GetAimLocation());     //projectile looks at the target
                }
            }
            transform.Translate(Vector3.forward * speed * Time.deltaTime);      //Projectile moves forward
        }
```

Figure 178 – Projectile.cs part 1

```
//This function sets the projectile's target,attacker and damage. (Used by Enemies and guards)
1 reference
public void SetTarget(Health target, GameObject attacker, float damage)
{
    this.target = target;
    this.attacker = attacker;
    this.damage = damage;
    Destroy(gameObject, maxLifeTime);   //Destroys the gameobject when it reaches it's max life time
}

//This function sets the projectile's target,attacker,damage and raycastHit. (Used by the Player)
1 reference
public void SetTarget(GameObject attacker, float damage,RaycastHit raycastHit)
{
    this.attacker = attacker;
    this.damage = damage;
    this.raycastHit = raycastHit;
    Destroy(gameObject, maxLifeTime);   //Destroys the gameobject when it reaches it's max life time
}

//Function that returns the aim location of the target
2 references
private Vector3 GetAimLocation()
{
    CapsuleCollider targetCapsule = target.GetComponent<CapsuleCollider>();
    if (targetCapsule == null)
    {
        return target.transform.position;
    }
    return target.transform.position + Vector3.up * targetCapsule.height / 2;
}
```

Figure 179 – Projectile.cs part 2

```
        //This function is called when the projectile collides with a target(other)
        Unity Message | 0 references
        private void OnTriggerEnter(Collider other)
        {
            if (other.tag == "Weapon") return;          //If collided with the weapon return

            speed = 0;                                  //Stops the projectile
            Destroy(gameObject, lifeAfterImpact);       //Destroys the gameobject after 0.2 seconds(lifeAfterImpact)

            if (hitEffect != null)                      //if there is a hit effect provided then instantiate the hit effect at the right location
            {
                Instantiate(hitEffect, transform.position, transform.rotation);
            }

            Health temp = other.GetComponent<Health>();         //Tries to get the Health component of other

            if (!temp) return;                                  //If it doesn't have Health component return
            if (temp.IsDead()) return;                          //And if target is dead then return

            if (attacker.tag == "Player")                               //If attacker is the Player
            {
                if (other.GetComponent<DialogueNPC>()) return;          //Make sure projectile didn't collide with a NPC
                if (other.GetComponent<GuardController>()) return;      //Or Guards
                other.GetComponent<ActionScheduler>().CancelCurrentAction();    //Else it collided with enemies
                other.GetComponent<EnemyAIController>().gotAlerted = true;      //And enemy gets alerted
                other.GetComponent<EnemyAIController>().gotHitByBow = true;      //And enemy gets alerted
            }
            temp.TakeDamage(attacker, damage);      //Apply damage from the attacker
        }
    }
}
```

Figure 180 – Projectile.cs part 3

5.11 Inventory

5.11.1 Overview

The player needs a place to store the useful items he needs in times of need. This is where the inventory system plays an important role. For example, in this project, the player needs a place to store the weapons and consumables, items that he needs when picking up or later. The inventory does not have an infinite amount of space, which means that the player will have to discard some items to make room for other items. The inventory consists of a grid

of 5 x 5 slots, so it has 25 slots for storing items. The inventory can be opened and closed by pressing TAB. Clicking on a sword item will equip that sword, and clicking on a consumable item will consume that item. There are also three other functions: First, the slots that contain items have a red button in the top right, if you click on it, the item will be placed in front of the player. The UNEQUIP button in the top left of the inventory will un-equip the currently equipped weapon and the player will be able to fight with their fists. Finally, hovering the mouse over the boxes that contain items will bring up a tooltip window with the useful information about that item.

Figure 181 – A screenshot of the inventory

5.11.2 Code

The Inventory system is in the Inventory.cs script, where all the methods are

```
using System.Collections.Generic;
using UnityEngine;
using ARPG.Combat;
using UnityEngine.UI;
using ARPG.Resources.Items;
using System.Linq;
using UnityEngine.EventSystems;
using TMPro;
using ARPG.Control;

namespace ARPG.Resources
{
    Unity Script | 5 references
    public class Inventory : MonoBehaviour
    {
        //Variable that contains the parent Inventory UI
        [SerializeField] GameObject inventoryUIContainer = null;
        //Variable that contains the parent GameObject of the tooltip
        [SerializeField] public GameObject tooltipParent;

        //The inventory instance
        private static Inventory instance;
        //This list contains all the items
        private List<Item> inventoryItems = new List<Item>();
        //This array is used to that we know which button is occupied or not, temporary ?
        private bool[] isOccupied = new bool[25];
        //This array is used to add the discard item function
        private Button[] discardItemButton = new Button[25];
        //This array is used to add the equip item function
        private Button[] equipButton = new Button[25];
        //Defomes the un-equip button
        private Button unEquipButton;

        private bool isUIEnabled = false;
        private PlayerController player;
        private Fighter fighter;
        private PlayerCombat playerCombat;
        private WeaponItem equipedWeapon = null;

        Unity Message | 0 references
        private void Awake()
        {
            //Checks if the instance is null, if it is then sets as instance this instance
            //and initializes the player, the fighter and the combat.
            if(instance == null)
            {
                instance = this;
                instance.player = GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerController>();
                instance.fighter = player.GetComponent<Fighter>();
                instance.playerCombat = player.GetComponent<PlayerCombat>();
            }
            else    //Else destroys the gameobject (trying to make sure there is only one inventory instance)
            {
                Destroy(gameObject);
            }
        }
```

Figure 182 – Inventory.cs part 1

```
Unity Message | 0 references
private void Start()
{
    InitButtonFunctionsCreation();
}

// Update is called once per frame
Unity Message | 0 references
void Update()
{
    if (Input.GetKeyDown(KeyCode.Tab) && !player.OnQuestBox() &&
        !player.GetComponent<PlayerController>().GetSpeaker().GetIsInDialogue())
    {
        if (instance.inventoryUIContainer.activeSelf)
        {
            inventoryUIContainer.SetActive(false);
            isUIEnabled = false;
        }
        else
        {
            inventoryUIContainer.SetActive(true);
            CheckOccupied();
            isUIEnabled = true;
        }
    }
}

//Function that returns if the inventory is full or not
1 reference
public bool IsInventoryFull()
{
    foreach(bool item in isOccupied)
    {
        if (item) continue;
        else return false;
    }
    return true;
}
```

Figure 183 – Inventory.cs part 2

```
//Function that initializes equipButtons and DiscardItemButtons
1 reference
private void InitButtonFunctionsCreation()
{
    Transform slotsContainer = inventoryUIContainer.transform.GetChild(0).transform;
    unEquipButton = inventoryUIContainer.transform.GetChild(1).gameObject.GetComponent<Button>();
    unEquipButton.onClick.AddListener(() => { UnEquip(); });
    int i = 0;
    foreach (Transform child in slotsContainer)
    {
        int index = i;
        equipButton[i] = child.GetChild(0).gameObject.GetComponent<Button>();
        equipButton[i].onClick.AddListener(() => { Use(index); });
        equipButton[i].gameObject.AddComponent<Hover>();
        equipButton[i].GetComponent<Hover>().index = i;
        equipButton[i].GetComponent<Hover>().tooltipParent = tooltipParent;

        discardItemButton[i] = child.GetChild(0).transform.GetChild(1).transform.gameObject.GetComponent<Button>();
        //https://answers.unity.com/questions/1288510/buttononclickaddlistener-how-to-pass-parameter-or.html
        discardItemButton[i].onClick.AddListener(() => { DiscardItem(index); });
        i++;
    }
}

//Function that adds the picked item to the inventory
2 references
public void addToInventory(Item pickedItem)
{
    player.GetComponent<PlayerController>().DoGatherItemQuestCheck(pickedItem);
    inventoryItems.Add(pickedItem);
    Transform slotsContainer = inventoryUIContainer.transform.GetChild(0).transform;    //Gets the slots container

    int i = 0;
    foreach (Transform child in slotsContainer) //Will look every slot
    {
        if (isOccupied[i] == false) //For a position that is not occupied to add the item
        {
            isOccupied[i] = true;   //Occupies that position
            i++;
            //Enables the image
            child.GetChild(0).GetChild(0).gameObject.GetComponent<Image>().enabled = true;
            //Sets the image to be the weapons icon
            child.GetChild(0).GetChild(0).gameObject.GetComponent<Image>().sprite = pickedItem.icon;
            //Makes the discard item button ineractable
            child.GetChild(0).GetChild(1).gameObject.GetComponent<Button>().interactable = true;
            return;
        }
        i++;
    }
}
```

Figure 184 – Inventory.cs part 3

```
//Function that uses the clicked item
1 reference
private void Use(int i)
{
    if (isOccupied[i] && inventoryItems[i].GetType() == typeof(WeaponItem))
    {
        //fighter.EquipWeapon((WeaponItem)inventoryItems[i]);
        //equipedWeapon = fighter.GetCurrentWeapon();
        playerCombat.EquipWeapon((WeaponItem)inventoryItems[i]);
        equipedWeapon = playerCombat.GetCurrentWeapon();

    }
    else if (isOccupied[i] && inventoryItems[i].GetType() == typeof(ConsumableItem))
    {
        foreach(ConsumableItem ci in inventoryItems.OfType<ConsumableItem>()){
            if(ci == inventoryItems[i])
            {
                player.GetComponent<Health>().Heal(ci.GetHealthRestoration());
                inventoryItems.RemoveAt(i);
                RefreshInventory();
                break;
            }
        }
    }
}

//Function that is part of the inventory UI refresh
1 reference
private void addToInventoryUI(Sprite icon)
{
    Transform slotsContainer = inventoryUIContainer.transform.GetChild(0).transform; //Gets the slots container

    int i = 0;
    foreach (Transform child in slotsContainer) //Will look every slot
    {
        if (isOccupied[i] == false) //For a position that is not occupied to add the item
        {
            isOccupied[i] = true;  //Occupies that position
            i++;
            //Enables the image
            child.GetChild(0).GetChild(0).gameObject.GetComponent<Image>().enabled = true;
            //Sets the image to be the weapons icon
            child.GetChild(0).GetChild(0).gameObject.GetComponent<Image>().sprite = icon;
            //Makes the discard item button ineractable
            child.GetChild(0).GetChild(1).gameObject.GetComponent<Button>().interactable = true;
            return;
        }
        i++;
    }
}
```

Figure 185 – Inventory.cs part 4

```
//Function that refreshes inventory UI
2 references
private void RefreshInventory()
{
    Transform slotsContainer = inventoryUIContainer.transform.GetChild(0).transform;
    int i = 0;
    foreach (Transform child in slotsContainer)
    {
        if (isOccupied[i] == true)
        {
            child.GetChild(0).GetChild(0).gameObject.GetComponent<Image>().enabled = false;
            child.GetChild(0).GetChild(1).gameObject.GetComponent<Button>().interactable = false;
        }
        isOccupied[i] = false;
        i++;
    }
    CheckOccupied();
    for (i = 0; i < inventoryItems.Count; i++)
    {
        addToInventoryUI(inventoryItems[i].icon);
    }
}

//Function that discards item/weapon from inventory
1 reference
private void DiscardItem(int i)
{
    if (equipedWeapon == inventoryItems[i])
    {
        UnEquip();
    }
    if (inventoryItems[i].GetType() == typeof(WeaponItem) || inventoryItems[i].GetType() == typeof(ConsumableItem))
    {   //Instantiate the discarded item(Pickup)
        GameObject temp = Instantiate(inventoryItems[i].dropPrefab, transform.position + (transform.forward * 2), Quaternion.identity);
        temp.GetComponent<Pickup>().RemoveDialogue();   //Removes the dialogue from the instantiated Pickup, if there is one
    }
    Debug.Log("Discard item " + i);
    isOccupied[i] = false;                                              //Sets the items position as not occupied
    Transform slotsContainer = inventoryUIContainer.transform.GetChild(0).GetChild(i);      //Gets the items slot
    slotsContainer.GetChild(0).GetChild(0).gameObject.GetComponent<Image>().sprite = null;    //Removes that slot's icon
    slotsContainer.GetChild(0).GetChild(0).gameObject.GetComponent<Image>().enabled = false; //Disables the slot's image
    slotsContainer.GetChild(0).GetChild(1).gameObject.GetComponent<Button>().interactable = false; //makes it's discard button no longer interactable
    inventoryItems.RemoveAt(i); //Removes item
    RefreshInventory();         //Refreshes inventory
}

//Function that un-equips the equiped weapon
2 references
private void UnEquip()
{
    playerCombat.EquipWeapon(playerCombat.GetDefaultWeapon());
}
```

Figure 186 – Inventory.cs part 5

```
//Function that un-equips the equiped weapon
2 references
private void UnEquip()
{
    playerCombat.EquipWeapon(playerCombat.GetDefaultWeapon());
}

//Function that checks if index is occupied
2 references
private void CheckOccupied()
{
    for (int i = 0; i < isOccupied.Length; i++)
    {
        if (isOccupied[i] == true && inventoryUIContainer.transform.GetChild(0).transform.GetChild(0).transform.gameObject.activeSelf == false)
        {
            isOccupied[i] = false;
        }
    }
}

2 references
public bool GetIsInventoryOpened()
{
    return isUIEnabled;
}
```

Figure 187 – Inventory.cs part 6

```
    //Hover class is responsible for showing an items information when mouse is hovering
    Unity Script | 3 references
    class Hover : MonoBehaviour, IPointerEnterHandler, IPointerExitHandler
    {
        public int index;
        public GameObject tooltipParent;
        16 references
        public void OnPointerEnter(PointerEventData eventData)
        {
            if (instance.isOccupied[index])
            {
                string tooltipText = null;
                if (instance.inventoryItems[index].GetType() == typeof(WeaponItem))
                {
                    WeaponItem temp = null;
                    foreach (WeaponItem wi in instance.inventoryItems.OfType<WeaponItem>())
                    {
                        if (wi == instance.inventoryItems[index])
                        {
                            temp = wi;
                            break;
                        }
                    }
                    tooltipText = "<color=orange><b>" + instance.inventoryItems[index].itemName +
                        "</b></color>\n" + "<size=32>" + instance.inventoryItems[index].description +
                        "\n<color=#800000ff>Damage: " + temp.GetDamage() + "\n<color=#900C3F>Percentage Damage: " +
                        temp.GetWeaponExtraPercentageDamage() + "</color> </size>";
                }
                else if (instance.inventoryItems[index].GetType() == typeof(ConsumableItem))
                {
                    ConsumableItem temp = null;
                    foreach (ConsumableItem ci in instance.inventoryItems.OfType<ConsumableItem>())
                    {
                        if (ci == instance.inventoryItems[index])
                        {
                            temp = ci;
                            break;
                        }
                    }
                    tooltipText = "<color=orange><b>" + instance.inventoryItems[index].itemName +
                        "</b></color>\n" + "<size=32>" + instance.inventoryItems[index].description +
                        "\nRestores " + temp.GetHealthRestoration() + " health!</size>";
                }
                tooltipParent.SetActive(true);
                tooltipParent.GetComponentInChildren<TextMeshProUGUI>().text = tooltipText;
            }
        }

        15 references
        public void OnPointerExit(PointerEventData eventData)
        {
            tooltipParent.SetActive(false);
            tooltipParent.GetComponentInChildren<TextMeshProUGUI>().text = "";
        }
    }
  }
}
```

Figure 188 – Inventory.cs part 7

5.12 UI Scripts

5.12.1 Overview

At any moment player has to know useful information like the current health, the enemy's health, the experience and the level. I created an empty GameObject where I added other GameObjects that contain some useful information and not only, for example the Inventory, the Quest window and the Escape menu.

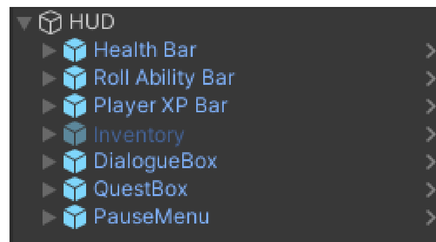


Figure 189 – The HUD GameObject.

- Health Bar GameObject, uses the *PlayerHealthDisplay.cs* which displays the health using a slider and a TMP (TextMeshPro) component.
- Roll Ability Bar, uses *PlayerAbilityDisplay.cs* which displays when the ability is ready to be used again using a slider.
- Player XP Bar:
    - Uses *ExperienceDisplay.cs* to display the experience using a slider.
    - Uses *LevelDisplay.cs* displays the level using a TMP component.
    - Uses *XPDisplay.cs* to display the experience using a TMP component.
- Inventory, displays the inventory as seen in a previous chapter.
- DialogueBox, which displays the dialogues as seen in a previous chapter
- QuestBox, which displays the quests window as seen in a previous chapter
- PauseMenu, uses *EscapeMenu.cs* do display a menu with settings and the option to go to the main menu.

To control the audio and the graphic settings I created a separate GameObject that:

- Uses *SetVolume.cs* to control the sounds and music volume using 2 different mixers for each one and sliders.
- Uses *GraphicSettings.cs* to select a graphic setting using a dropdown component.

Another important thing is that player needs to know about the enemy's health and the damage that is being dealt by every attack. These 2 features are provided by *EnemyHealthDisplay.cs* and *PopupText.cs* using UI elements.

Figure 190 – Enemy's health bar and damage display.

5.12.2 Code

```
using UnityEngine;
using UnityEngine.UI;

namespace ARPG.Resources
{
    Unity Script | 0 references
    public class PlayerHealthDisplay : MonoBehaviour
    {
        private Slider slider;
        private Image fillImage;
        private Health health;
        private Color color;
        private TextMeshProUGUI textValue;

        //On Awake it initializes the fields
        Unity Message | 0 references
        private void Awake()
        {
            health = GameObject.FindWithTag("Player").GetComponent<Health>();
            slider = GetComponent<Slider>();
            slider.maxValue = health.GetMaxHealthPoints();
            slider.value = health.GetHealthPoints();
            fillImage = transform.GetChild(0).GetComponent<Image>();
            color = fillImage.color;
            textValue = GetComponentInChildren<TextMeshProUGUI>();
        }

        Unity Message | 0 references
        private void Update()
        {
            if (health.GetIsImmuneToDamage())   //If is immune to damage
            {
                fillImage.color = Color.yellow; //makes the bar color yellow
            }
            else
            {
                fillImage.color = color;        //Else changescolor to default(red)
            }
            slider.maxValue = health.GetMaxHealthPoints();//The max health value
            slider.value = health.GetHealthPoints();//value is set to the current health points
            textValue.SetText("{0:0}",health.GetHealthPoints());//sets the text
        }
    }
}
```

Figure 191 – PlayerHealthDisplay.cs

```
using ARPG.Movement;
using UnityEngine;
using UnityEngine.UI;

Unity Script | 0 references
public class PlayerAbilityDisplay : MonoBehaviour
{
    private Slider slider;
    private PlayerMovement playerController;

    //On Awake initializes the fields
    //Using the PlayerMovement that has all the information about the ability (2 seconds between rolls)
    Unity Message | 0 references
    private void Awake()
    {
        playerController = GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerMovement>();
        slider = GetComponent<Slider>();
        slider.maxValue = playerController.GetTimeBetweenRolls();
        slider.value = playerController.GetTimeSinceLastRoll();
    }

    Unity Message | 0 references
    private void Update()
    {   //Updates the value
        slider.value = playerController.GetTimeSinceLastRoll();
    }
}
```

Figure 192 – PlayerAbilityDisplay.cs

```csharp
using UnityEngine;
using UnityEngine.UI;

namespace ARPG.Stats
{
    Unity Script | 0 references
    public class ExperienceDisplay : MonoBehaviour
    {
        private Slider slider;
        Experience experience;
        BaseStats baseStats;

        Unity Message | 0 references
        private void Awake()
        {   //Gets the experience and basestats from the Player
            //abd initializes the fields
            experience = GameObject.FindWithTag("Player").GetComponent<Experience>();
            baseStats = GameObject.FindWithTag("Player").GetComponent<BaseStats>();
            slider = GetComponent<Slider>();
            slider.minValue = experience.GetExperience();
            slider.maxValue = baseStats.GetStat(Stat.ExperienceToLevelUp);
            slider.value = experience.GetExperience();
            baseStats.onLevelUp += updateMinValue;
        }

        Unity Message | 0 references
        private void Update()
        {   //Updates the max value with how much experience is needed to level up
            slider.maxValue = baseStats.GetStat(Stat.ExperienceToLevelUp);
            //Updates the valie with the current experience
            slider.value = experience.GetExperience();
        }

        //Function called whenever player level ups, and is responsible of updating the slider's
        //minimum value in order to dispay the experience bar correctly after leveling up
        1 reference
        private void updateMinValue()
        {
            slider.minValue = baseStats.GetStat(Stat.ExperienceToLevelUp, baseStats.GetLevel() - 1);
        }
    }
}
```

Figure 193 – ExperienceDisplay.cs

```csharp
using TMPro;
using UnityEngine;

namespace ARPG.Stats
{
    Unity Script | 0 references
    public class LevelDisplay : MonoBehaviour
    {
        BaseStats baseStats;
        TextMeshProUGUI levelValue;

        Unity Message | 0 references
        private void Awake()
        {
            //Gets the basestats from the player
            baseStats = GameObject.FindWithTag("Player").GetComponent<BaseStats>();
            levelValue = GetComponent<TextMeshProUGUI>();
        }

        Unity Message | 0 references
        private void Update()
        {   //Updates the text with the current level
            levelValue.SetText("{0:0}", baseStats.GetLevel());
        }
    }
}
```

Figure 194 – LevelDisplay.cs

```
using TMPro;
using UnityEngine;

namespace ARPG.Stats
{
    Unity Script | 0 references
    public class XPDisplay : MonoBehaviour
    {
        Experience experience;
        TextMeshProUGUI xpValue;

        Unity Message | 0 references
        private void Awake()
        {   //Gets experience from player
            experience = GameObject.FindWithTag("Player").GetComponent<Experience>();
            xpValue = GetComponent<TextMeshProUGUI>();
        }

        Unity Message | 0 references
        private void Update()
        {   //updates the xp value by using the experience points
            xpValue.SetText("{0:0}", experience.GetExperience());
        }
    }
}
```

Figure 195 – XPDisplay.cs

```
using UnityEngine;

namespace ARPG.Other
{
    Unity Script | 2 references
    public class EscapeMenu : MonoBehaviour
    {
        public GameObject escapeMenuPanel;
        private static EscapeMenu instance;
        AudioSource[] audioSources;

        Unity Message | 0 references
        void Awake()
        {
            if (instance == null)   //Makes sure only one instance of EscameMenuExists and initializes the fields
            {
                instance = this;
                instance.escapeMenuPanel.SetActive(false);
                instance.audioSources = GameObject.FindObjectsOfType(typeof(AudioSource)) as AudioSource[];
            }
            else
            {
                Destroy(gameObject);
            }
        }

        Unity Message | 0 references
        void Update()
        {
            if (Input.GetKeyDown(KeyCode.Escape))                       //When pressing ESC
            {
                if (instance.escapeMenuPanel.activeSelf)                //If panel is active
                {                                                       //Then
                    instance.escapeMenuPanel.SetActive(false);          //Close the panel
                    Time.timeScale = 1;                                 //Unpause the game
                    foreach (AudioSource audioSource in audioSources)   //Unpause all the AudioSources
                    {
                        audioSource.UnPause();
                    }

                }
                else                                                    //Else if panel is not active
                {                                                       //Then
                    instance.escapeMenuPanel.SetActive(true);           //Open the panel
                    Time.timeScale = 0;                                 //Pause the game
                    foreach (AudioSource audioSource in audioSources)   //Pause all the AudioSources
                    {
                        audioSource.Pause();
                    }
                }
            }
        }
    }
}
```

Figure 196 – EscapeMenu.cs

```csharp
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.UI;

namespace ARPG.Audio
{
    Unity Script | 0 references
    public class SetVolume : MonoBehaviour
    {
        public AudioMixer mixer;
        public Slider slider;

        Unity Message | 0 references
        private void Awake()
        {
            if (mixer.name == "MusicMixer") //If the mixer is the MusicMixer
            {   //checks if the volume has been saved already, if not then returns
                //else gets the saved value
                if (PlayerPrefs.GetInt("changedMusicVolume") == 0) return;
                slider.value = PlayerPrefs.GetFloat("musicVolume_slider");
                mixer.GetFloat("volume", out float value);
                return;
            }

            if (mixer.name == "SoundsMixer") //If the mixer is the SoundsMixer
            {   //checks if the volume has been saved already, if not then returns
                //else gets the saved value
                if (PlayerPrefs.GetInt("changedSoundsVolume") == 0) return;
                slider.value = PlayerPrefs.GetFloat("soundsVolume_slider");
                mixer.GetFloat("volume", out float value);
            }
        }

        Unity Message | 0 references
        private void Start()
        {
            SetLevel(slider.value);
        }
        1 reference
        public void SetLevel(float sliderValue)//sets volume with the given sliderValue
        {
            mixer.SetFloat("volume", Mathf.Log10(sliderValue) * 20);
        }
        0 references
        public void SaveSoundsVolume() //Saves the sound volume
        {
            PlayerPrefs.SetFloat("soundsVolume_slider", slider.value);
            PlayerPrefs.SetInt("changedSoundsVolume", 1);
            PlayerPrefs.Save();
        }
        0 references
        public void SaveMusicVolume() //Saves the music volume
        {
            PlayerPrefs.SetFloat("musicVolume_slider", slider.value);
            PlayerPrefs.SetInt("changedMusicVolume", 1);
            PlayerPrefs.Save();
        }
    }
}
```

Figure 197 – SetVolume.cs

```csharp
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;

namespace ARPG.Other
{
    Unity Script | 0 references
    public class GraphicSettings : MonoBehaviour
    {
        public TMP_Dropdown dropdown;
        string[] names;

        Unity Message | 0 references
        private void Awake()
        {
            names = QualitySettings.names;                  //Gets all the quality setting names
            dropdown.options.Clear();                       //Clears the dropdown options
            for (int i = 0; i < names.Length; i++)          //Assigns the quality setting names to the dropdown
            {
                TMP_Dropdown.OptionData temp = new TMP_Dropdown.OptionData();
                temp.text = names[i];
                dropdown.options.Add(temp);

            }
            dropdown.value = QualitySettings.GetQualityLevel();             //dropdown value equals the the current quality setting
            if (PlayerPrefs.GetInt("changedGraphics") == 0) return;         //If setting was changed manually by user then
            dropdown.value = PlayerPrefs.GetInt("graphicSettingsIndex");    //gets the index of the quality setting that was selected
            ChangeGraphics(dropdown);                                       //changes the graphic settings
        }

        Unity Message | 0 references
        private void Start()
        {
            dropdown.onValueChanged.AddListener(delegate { ChangeGraphics(dropdown); });
        }
```

Figure 198 – GraphicSettings.cs part 1

```csharp
        public void ChangeGraphics(TMP_Dropdown dropdown)
        {
            QualitySettings.SetQualityLevel(dropdown.value);
            PlayerPrefs.SetInt("graphicSettingsIndex", dropdown.value);
            PlayerPrefs.SetInt("changedGraphics", 1);
            PlayerPrefs.Save();
        }

        //Function that gets called when scene is loaded
        0 references
        private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
        {
            names = QualitySettings.names;                  //Gets all the quality setting names
            dropdown.options.Clear();                       //Clears the dropdown options
            for (int i = 0; i < names.Length; i++)          //Assigns the quality setting names to the dropdown
            {
                TMP_Dropdown.OptionData temp = new TMP_Dropdown.OptionData();
                temp.text = names[i];
                dropdown.options.Add(temp);

            }
            dropdown.value = QualitySettings.GetQualityLevel();             //dropdown value equals the the current quality setting
            if (PlayerPrefs.GetInt("changedGraphics") == 0) return;         //If setting was changed manually by user then
            dropdown.value = PlayerPrefs.GetInt("graphicSettingsIndex");    //gets the index of the quality setting that was selected
            ChangeGraphics(dropdown);                                       //changes the graphic settings
        }
    }
}
```

Figure 199 – GraphicSettings.cs part 2

```
using UnityEngine;
using UnityEngine.UI;

namespace ARPG.Resources
{
    Unity Script | 0 references
    public class EnemyHealthDisplay : MonoBehaviour
    {
        public GameObject enemyHealthBarPrefab;     //Defines the enemy's health bar gameobject
        public float showDistance = 10f;            //Visible in that distance
        private Health health;                  //Health component
        private Slider slider;                  //healthbar slider component
        private GameObject player;              //defines the player
        private Canvas canvas;                  //canvas that contains the health bar

        Unity Message | 0 references
        private void Awake()    //Initializes the fields
        {
            health = GetComponent<Health>();
            slider = enemyHealthBarPrefab.GetComponent<Slider>();
            slider.maxValue = health.GetMaxHealthPoints();
            slider.value = health.GetHealthPoints();
            player = GameObject.FindWithTag("Player");
            canvas = enemyHealthBarPrefab.transform.GetComponentInParent<Canvas>();
        }

        Unity Message | 0 references
        private void Update()
        {
            if (health.IsDead())            //If enemy is dead then disables the canvas
            {
                canvas.enabled = false;
            }
            if (!health.IsDead())           //else makes the health bar visible if player is in the distance
            {
                if (Vector3.Distance(player.transform.position, health.gameObject.transform.position) < showDistance)
                {
                    canvas.enabled = true;
                    slider.value = health.GetHealthPoints();
                    enemyHealthBarPrefab.transform.rotation = Camera.main.transform.rotation;
                }
                else   //when player is not in that distance disables the canvas
                {
                    canvas.enabled = false;
                }
            }
        }
    }
}
```

Figure 200 – EnemyHealthDisplay.cs part 1

```
using UnityEngine;

namespace ARPG.Other
{
    Unity Script | 0 references
    public class PopupText : MonoBehaviour
    {
        public float lifeTime = 1f;                         //text life time
        public Vector3 offset = new Vector3(0, 2f, 0);      //offset
        public bool randomize = true;                       //to be in random position ?
        public Color color = Color.white;                   //color

        private float randomXPosition = 1f;

        Unity Message | 0 references
        void Start()
        {
            Destroy(gameObject, lifeTime);                  //Will destroy the gameobject after lifetime expires
            transform.localPosition += offset;              //changes the text's vertical position by +2
            GetComponent<TextMesh>().color = color;

            if (randomize)
            {
                transform.localPosition += new Vector3(Random.Range(-randomXPosition, randomXPosition), 0, 0);
            }
        }

        Unity Message | 0 references
        private void Update()   //Update, makes the text to look the camera
        {
            transform.rotation = Camera.main.transform.rotation;
        }
    }
}
```

Figure 201 – PopupText.cs

5.13 Other Scripts

5.13.1 SmartRenderer.cs

The NPCs throughout the scene "run" and do what they are supposed to do, but that means they are enabled even when the camera is not rendering them, and that leads to increased CPU consumption, e.g. the CPU time to process a frame was 8.3 miliseconds and the render time was 121 FPS (Frames Per Second). However, using SmartRenderer.cs attached to each NPC, guard and enemy ensures that they are not activated and rendered unless they are within a certain distance of the player. The result is that CPU took 5.2 miliseconds and 192 FPS to process one frame, which is an improvement.

```
using ARPG.AI;
using UnityEngine;

namespace ARPG.Core
{
    Unity Script | 0 references
    public class SmartRender : MonoBehaviour
    {
        private GameObject player;
        private bool isActivated;

        Unity Message | 0 references
        private void Awake()    //Initializes the fields
        {
            player = GameObject.FindGameObjectWithTag("Player");
            isActivated = true;
        }

        Unity Message | 0 references
        private void Update()
        {   //if activated and the distance between the npc and thhe player is greater than 50
            //then deactivate the npc
            if (isActivated && Vector3.Distance(gameObject.transform.position, player.transform.position) > 50f)
            {
                gameObject.transform.GetChild(0).gameObject.SetActive(false);
                Manage(false);
                Debug.Log("deactivate");
                isActivated = false;

            }
            //if not activated and the distance between the npc and thhe player is less than 50
            //then activate the npc
            else if (!isActivated && Vector3.Distance(gameObject.transform.position, player.transform.position) <= 50f)
            {
                gameObject.transform.GetChild(0).gameObject.SetActive(true);
                Manage(true);
                Debug.Log("activate");
                isActivated = true;
            }
        }
```

Figure 202 – SmartRenderer.cs part 1

```
        //Function that checks what type of NPC is the gameobject and activates
        //or deactivates it's controller based on the given parameter
        2 references
        private void Manage(bool state)
        {
            if (gameObject.tag == "NPC")
            {
                GetComponent<DialogueNPC>().enabled = state;
                return;
            }
            else if (gameObject.tag == "Guard")
            {
                GetComponent<GuardController>().enabled = state;
                return;
            }
            else if (gameObject.tag == "EnemyNPC")
            {
                GetComponent<EnemyAIController>().enabled = state;
                return;
            }
        }
    }
}
```

Figure 203 – SmartRenderer.cs part 2

5.13.2 WaypointPath.cs

Basic NPCs, farmers, woodcutters, enemies, and guards use a waypoint path system to move around the map. More specifically, it is an empty GameObject that has the WaypointPath.cs script attached to it and has empty GameObject children, where these child GameObjects are the waypoints. For example, in Figure 204, the GameObject containing the WaypointPath.cs script has 19 waypoints. It can be a cycle path and a non-cycle path, which means that either the last waypoint can be connected to the first or not.

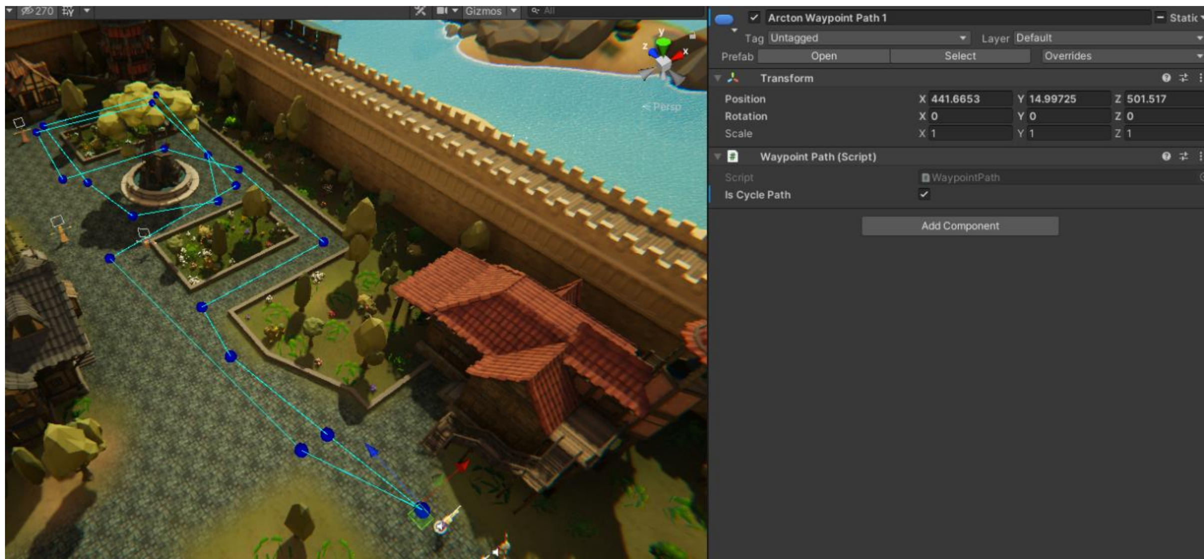


Figure 204 – Waypoint path example.

```
using UnityEngine;

namespace ARPG.Control
{
    Unity Script | 15 references
    public class WaypointPath : MonoBehaviour
    {
        [SerializeField] bool isCyclePath = false;

        bool goBackwards = false;

        Unity Message | 0 references
        private void OnDrawGizmos()
        {
            const float waypointGizmoRadius = 0.5f;
            //For all waypoints, draw a sphere gizmo and connect them with lines
            for (int i = 0; i < transform.childCount; i++)
            {
                Gizmos.color = Color.blue;
                //Draws sphere to indicate the waypoint location, for debug
                Gizmos.DrawSphere(GetWaypoint(i), waypointGizmoRadius);

                if(GetNextIndex(i) == 0 && isCyclePath == false)
                {
                    break;
                }
                else
                {
                    Gizmos.color = Color.cyan;
                    //Draws line to each waypoint, for debug
                    Gizmos.DrawLine(GetWaypoint(i), GetWaypoint(GetNextIndex(i)));
                }
            }
        }

        5 references
        public bool GetIsCyclePath()
        {
            return isCyclePath;
        }
```

Figure 205 – WaypointPath.cs part 1

```
        //Function that returns the next index
        7 references
        public int GetNextIndex(int i)
        {
            if (i + 1 == transform.childCount)
            {
                return 0;
            }
            return i + 1;
        }

        //Function returns the next index when going backwards
        5 references
        public int GetNextIndexBackwards(int i)
        {

            if (i+1 == transform.childCount)
            {
                goBackwards = true;
                return i - 1;
            }
            if(goBackwards == true && i == 0)
            {
                goBackwards = false;
                return i + 1;
            }
            if (goBackwards == true)
            {
                return i - 1;
            }
            return i + 1;
        }

        //Function that returns a waypoint's position
        8 references
        public Vector3 GetWaypoint(int i)
        {
            return transform.GetChild(i).position;
        }
    }
}
```

Figure 206 – WaypointPath.cs part 2

5.13.3 Safe.cs

When all NPCs (Plain, Farmers, Woodcutters) are on alert because of a fight or because an enemy is trying to attack them, they enter the Flee state. When fleeing, the NPCs stop what they were doing and hide in a safe place. This safe place is an empty GameObject to which the *Safe.cs* script is attached, and a Box Collider (so that the script can identify the NPCs in collisions). Example in Figure 207.

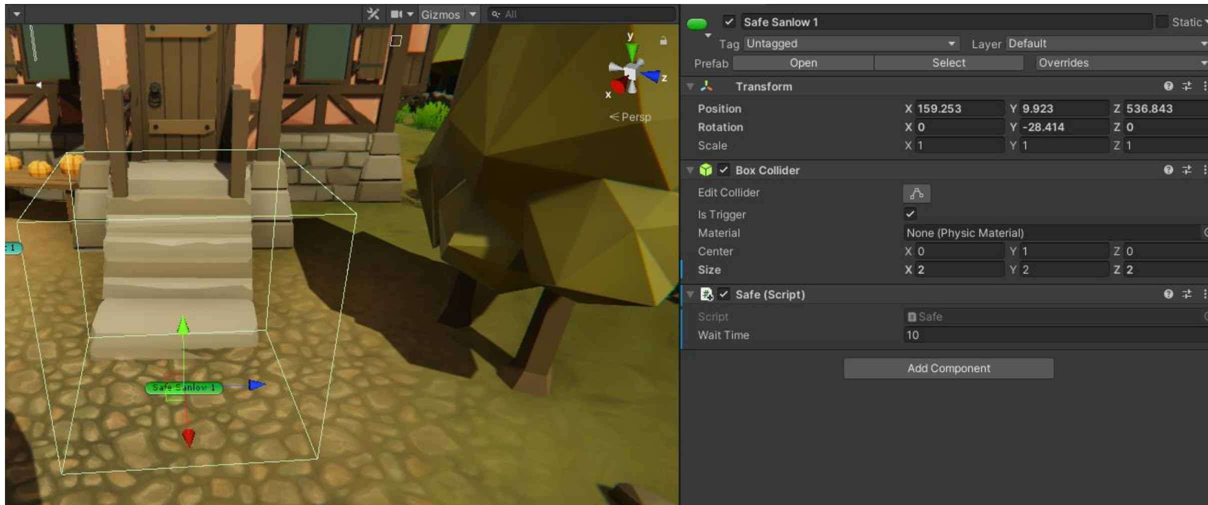


Figure 207 – Safe point example.

```
using ARPG.AI;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace ARPG.Control
{
    Unity Script | 0 references
    public class Safe : MonoBehaviour
    {   //Wait time in the safe point
        [SerializeField] float waitTime = 10f;

        List<NPCAIController> npcs;//list of plain NPCs
        List<WoodcutterController> woodcutters;//List of woodcutters
        List<FarmerController> farmers;//List farmers

        Unity Message | 0 references
        private void Start()    //initializes the lists
        {
            npcs = new List<NPCAIController>();
            woodcutters = new List<WoodcutterController>();
            farmers = new List<FarmerController>();
        }
        //When collides
        Unity Message | 0 references
        private void OnTriggerEnter(Collider other)
        {   //If the colider has NPC tag and its a plain npc
            if (other.tag == "NPC" && other.GetComponent<NPCAIController>())
            {
                NPCAIController temp = other.GetComponent<NPCAIController>();
                if (temp.isOnAlert) //Checks if that NPC is on alert
                {
                    npcs.Add(temp);//If so, adds the NPC to the npcs list
                    foreach (NPCAIController npc in npcs)//and for every npc in npcs list
                    {  //disables the npc's gameobject
                        npc.gameObject.SetActive(false);
                        StartCoroutine(WaitInSafe(npc));//starts WaitInSafe coroutine
                    }
                }
            }
            //If the colider has NPC tag and its a woodcutter
            if (other.tag == "NPC" && other.GetComponent<WoodcutterController>())
            {
                WoodcutterController temp = other.GetComponent<WoodcutterController>();
                if (temp.isOnAlert)//Checks if that NPC is on alert
                {
                    woodcutters.Add(temp);//If so, adds the NPC to the woodcutters list
                    foreach (WoodcutterController woodcutter in woodcutters)//and for every woodcutter in woodcutters list
                    {  //disables the woodcutter's gameobject
                        woodcutter.gameObject.SetActive(false);
                        StartCoroutine(WaitInSafe(woodcutter));//starts WaitInSafe coroutine
                    }
                }
            }
```

Figure 208 – Safe.cs part 1

```
                //If the colider has NPC tag and its a farmer
                if (other.tag == "NPC" && other.GetComponent<FarmerController>())
                {
                    FarmerController temp = other.GetComponent<FarmerController>();
                    if (temp.isOnAlert)//Checks if that NPC is on alert
                    {
                        farmers.Add(temp);//If so, adds the NPC to the farmers list
                        foreach (FarmerController farmer in farmers)//and for every farmer in farmers list
                        {   //disables the farmer's gameobject
                            farmer.gameObject.SetActive(false);
                            StartCoroutine(WaitInSafe(farmer));//starts WaitInSafe coroutine
                        }
                    }
                }
            }
            //WaitInSafe for the plain NPCs
            1 reference
            IEnumerator WaitInSafe(NPCAIController npc)
            {   //waits for waitTime seconds
                yield return (new WaitForSeconds(waitTime));
                npc.isOnAlert=false;//resets the alert
                npc.gameObject.SetActive(true);//activates the gameobject
                npc.currentWaypointIndex = 0;//resets the waypoint index
                npcs.Remove(npc);//removes the plain npc from the list
            }
            //WaitInSafe for the plain woodcutters
            1 reference
            IEnumerator WaitInSafe(WoodcutterController woodcutter)
            {   //waits for waitTime seconds
                yield return (new WaitForSeconds(waitTime));
                woodcutter.isOnAlert = false;//resets the alert
                woodcutter.gameObject.SetActive(true);//activates the gameobject
                woodcutter.currentWaypointIndex = 0;//resets the waypoint index
                woodcutters.Remove(woodcutter);//removes the woodcutter from the list
            }
            //WaitInSafe for the farmers
            1 reference
            IEnumerator WaitInSafe(FarmerController farmer)
            {   //waits for waitTime seconds
                yield return (new WaitForSeconds(waitTime));
                farmer.isOnAlert = false;//resets the alert
                farmer.gameObject.SetActive(true);//activates the gameobject
                farmer.currentWaypointIndex = 0;//resets the waypoint index
                farmers.Remove(farmer);//removes the farmer from the list
            }
        }
}
```

Figure 209 – Safe.cs part 2

5.13.4 MainMenuManager.cs

When player first starts the game, the main menu scene is loaded and shows the game logo with a fade out effect while the camera moving downwards. When fade out is over then the main menu buttons are coming into the camera view and player has the options:

- Start game, which starts the game.
- How to play, shows a panel with instructions.
- About, shows a panel with the quick view on the storyline.
- Settings, shows a panel with graphics and audio settings.
- Exit game, closes the game.

Figure 210 – Main menu scene GameObjects.

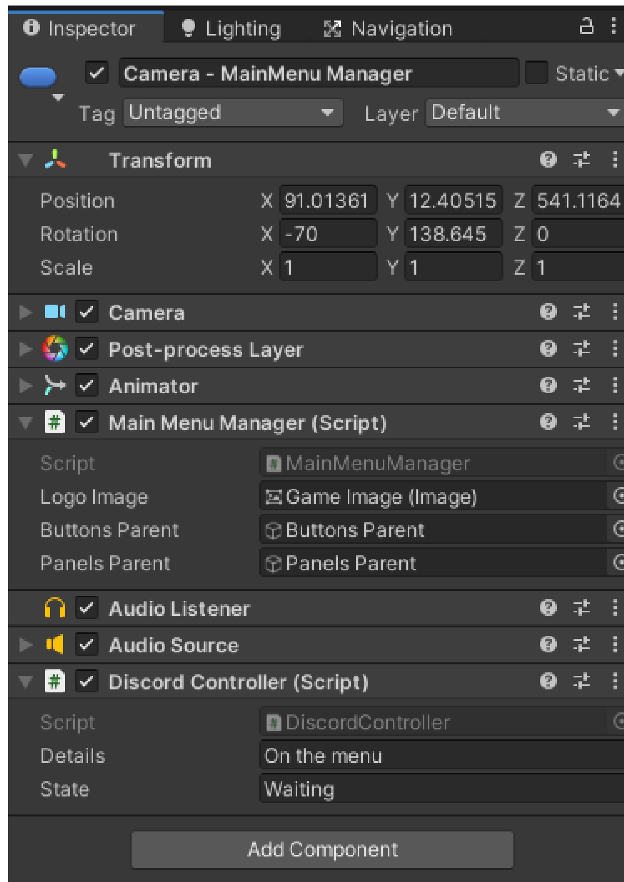


Figure 211 – Camera – MainMenu Manager GameObject.

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

namespace ARPG.Other
{
    Unity Script | 0 references
    public class MainMenuManager : MonoBehaviour
    {
        [SerializeField] Image logoImage;
        [SerializeField] GameObject buttonsParent;
        [SerializeField] GameObject panelsParent;

        Unity Message | 0 references
        private void Start()
        {   //On start unpauses the game and starts the ImageDadeOut coroutine
            Time.timeScale = 1;
            StartCoroutine(ImageFadeOut());
        }

        1 reference
        private IEnumerator ImageFadeOut()  //fades out the logoImage
        {
            Color temp = logoImage.color;

            while (temp.a >= 0)
            {
                temp = logoImage.color;
                temp.a = temp.a - (0.2f * Time.deltaTime);
                logoImage.color = temp;
                yield return null; //Don't freeze Unity
            }

            ButtonsEnter();     //When image is faded out the butons animation is triggered
        }

        //Triggers the animation that makes the buttons to move upwards to the camera view
        1 reference
        private void ButtonsEnter()
        {
            buttonsParent.GetComponent<Animator>().SetTrigger("ButtonsEnter");
        }
```

Figure 212 – MainMenuManager.cs part 1

```
        //Enables/Disables the how to play panel
        0 references
        public void HowToPlayPanelManager(bool value)
        {
            ClosePanels();
            panelsParent.transform.GetChild(0).gameObject.SetActive(value);
        }
        //Enables/Disables the settings panel
        0 references
        public void SettingsPanelManager(bool value)
        {
            ClosePanels();
            panelsParent.transform.GetChild(1).gameObject.SetActive(value);
        }
        //Enables/Disables the about panel
        0 references
        public void AboutPanelManager(bool value)
        {
            ClosePanels();
            panelsParent.transform.GetChild(2).gameObject.SetActive(value);
        }

        //closes the game and deletes the saved settings
        0 references
        public void ExitGame()
        {
            PlayerPrefs.DeleteAll();
            Application.Quit();
        }

        //closes all the panels
        3 references
        private void ClosePanels()
        {
            for (int i = 0; i < panelsParent.transform.childCount; i++)
            {
                panelsParent.transform.GetChild(i).gameObject.SetActive(false);
            }
        }
    }
}
```

Figure 213 – MainMenuManager.cs part 2

5.13.5 SceneLoader.cs

When player clicks the start game from the main menu or returning from the game scene to the main menu scene, the SceneLoader.cs is used. It is responsible for loading the scene asynchronously in the background and showing random tips about the game.

```
using System.Collections;
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

namespace ARPG.SceneManagement
{
    Unity Script | 0 references
    public class SceneLoader : MonoBehaviour
    {
        public GameObject loadingScreen;
        public Slider slider;
        public TextMeshProUGUI progressText;
        public TextMeshProUGUI tipText;
        public bool showTip = true;
        //Defines the tips of the loading screen
        private string[] tips = { "NPC's with a exclamation mark are quest givers.",
            "Press SPACE to roll.", "Did you read the instructions?",
            "When rolling you avoid any damage" };
        //loads the given scene index
        0 references
        public void LoadScene(int sceneIndex)
        {
            StartCoroutine(LoadAsynchronously(sceneIndex));
        }
        //Loads asynchronously the scene index
        1 reference
        private IEnumerator LoadAsynchronously(int sceneIndex)
        {
            loadingScreen.SetActive(true);  //Enables the loading screen gameobject
            Color color = loadingScreen.GetComponent<Image>().color;
            color.a = 1f;
            loadingScreen.GetComponent<Image>().color = color;
            if (showTip)//If true, shows a random tip
            {
                tipText.gameObject.SetActive(true);
                tipText.text = tips[Random.Range(0, tips.Length)];
            }
            yield return null;
            //used to measure the loading of the scene
            AsyncOperation operation = SceneManager.LoadSceneAsync(sceneIndex);

            while (!operation.isDone)//while the loading isnt done
            {   //shows the loading progress using a slider and a text component
                float progress = Mathf.Clamp01(operation.progress / 0.9f);
                slider.value = progress;
                progressText.SetText("{0:0}" + "%", progress * 100f);
                yield return null;
            }
            //Unloads the correct scene once loaded is done
            if (sceneIndex == 1)
                SceneManager.UnloadSceneAsync(0);
            else
                SceneManager.UnloadSceneAsync(0);
        }
    }
}
```

Figure 214 – SceneLoader.cs

5.13.6 DiscordController.cs

First what is Discord? It is a popular online communication platform that lets you communicate with your friends directly via text, voice or video and join servers where small and large communities interact together. I added the Discord's rich presence integration so the player can show to others on Discord, what game is currently playing and what is doing.

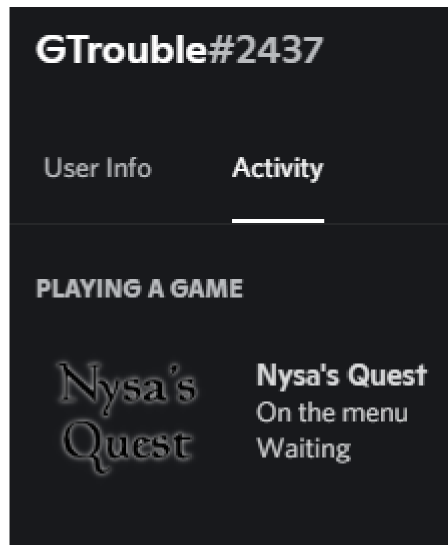


Figure 215 – Discord's rich presence.

After following the Discord's developers portal, I downloaded the SDK for Unity (Csharp) and installed it in the Plugins folder of the Project. Created an application on the Discord Developer portal and provided a game name and a logo and lastly, I created the *DiscordController.cs*.

```
using UnityEngine;

namespace ARPG.Other
{
    Unity Script | 0 references
    public class DiscordController : MonoBehaviour
    {
        [SerializeField] string details = "";
        [SerializeField] string state = "";

        public Discord.Discord discord;

        Unity Message | 0 references
        void OnDisable()
        {   //destroys the instance
            discord.Dispose();
        }

        Unity Message | 0 references
        void Start()
        {
            //https://discord.com/developers/docs/game-sdk/sdk-starter-guide#code-primer-unity-csharp
            discord = new Discord.Discord(                 , (System.UInt64)Discord.CreateFlags.Default);
            //Fetches an instance of the manager for interfacing with activities in the SDK
            var activityManager = discord.GetActivityManager();
            var activity = new Discord.Activity
            {
                Details = details,
                State = state,
                Assets =
            {
                LargeImage = "game_logo_1024_c",
                LargeText = "Nysa's Quest"
            }
            };
            activityManager.UpdateActivity(activity, (result) =>
            {
                if (result == Discord.Result.Ok)
                    Debug.Log("Discord status set!");
                else
                    Debug.LogError("Discord status failed!");
            });
        }

        Unity Message | 0 references
        void Update()
        {   //Checks any new infos from Discord
            discord.RunCallbacks();
        }
    }
}
```

Figure 216 – DiscordController.cs

The red mark in figure 216 is used to hide the application’s client id.

5.14 Navigation

Unity provides a navigation system that allows the creation of intelligent characters that can move around the game world, using navigation meshes that were created automatically from the scene’s geometry and dynamic obstacles allow you to alter the navigation of the characters at runtime. The following pieces are part of the navigation system:

- NavMesh, is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another. The data structure is built, or baked, automatically from the level’s geometry.
- NavMesh Agent component helps to define characters that avoid each other when moving in the scene.

- Off-Mesh Link component allows to create navigation shortcuts between two locations.
- NavMesh Obstacle is a component that allows you to describe moving obstacles for the agents to avoid when moving in the scene.

In this project, for player and NPC navigation, I first marked the terrain and some other surfaces as Navigation Static to include them in the NavMesh baking process. I also added the NavMesh obstacle component to trees, NPCs, and other buildings. On the Agents tab of the navigation window, I left the settings unchanged because they fit my characters (player and NPCs). On the Bake tab, I adjusted the settings as I did with the Humanoid agent and baked the navigation.

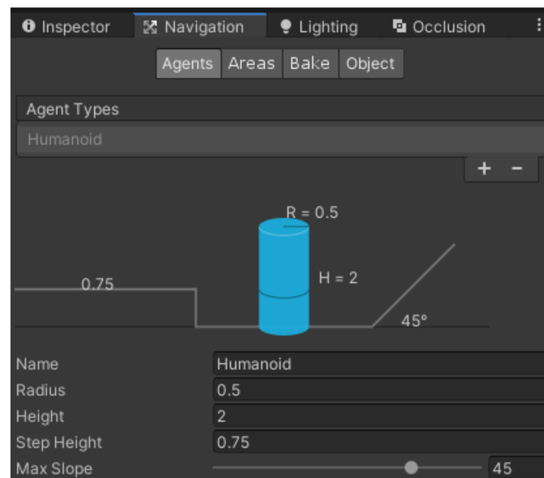


Figure 215 – Navigation, Agents tab

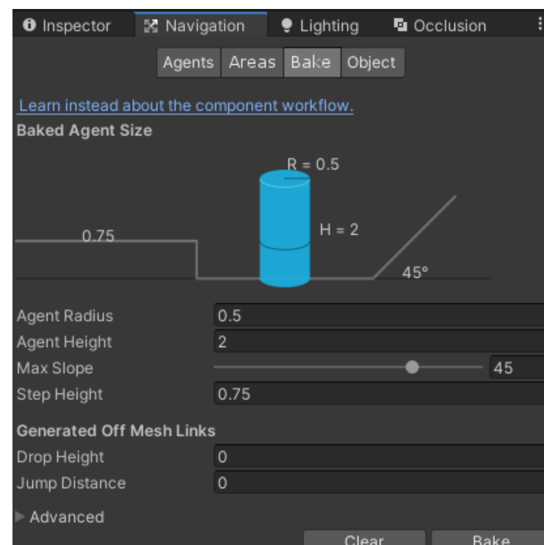


Figure 218 – Navigation, Bake tab

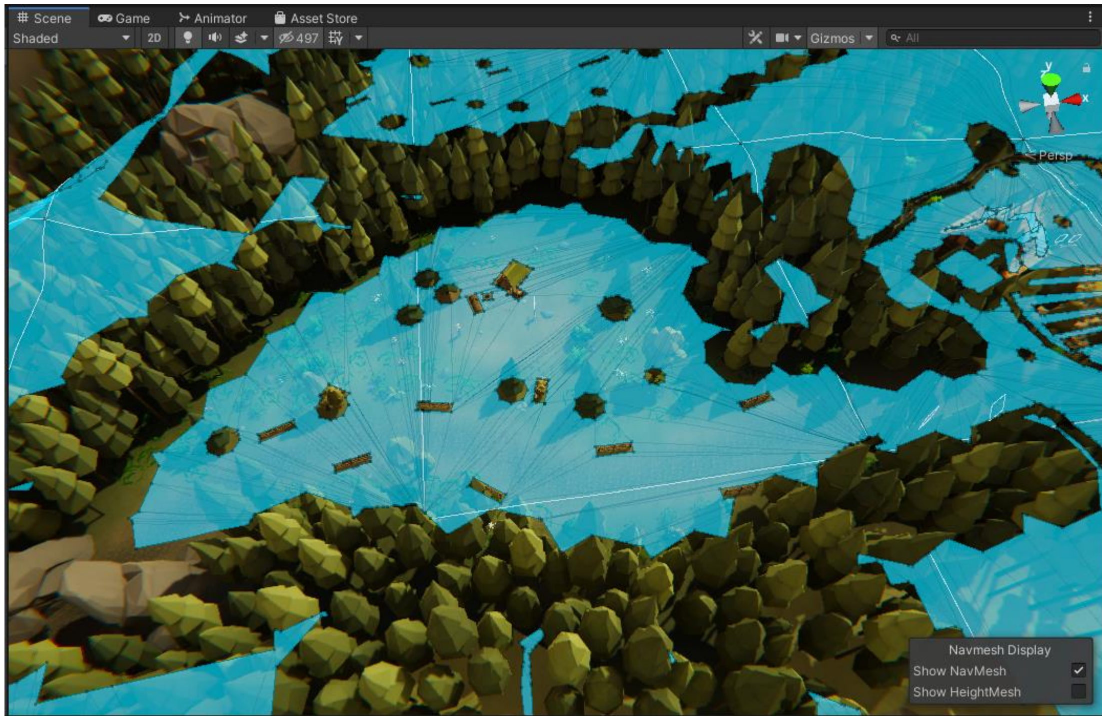


Figure 219 – NavMesh area of a part of the scene

6. Epilogue

6.1 Conclusion

In this action RPG project, Nysa's Quest, I designed and created a small 3D world populated with many friendly entities such as simple NPCs, farmers, woodcutters, and many aggressive enemies. For all entities, both friendly and aggressive, I created finite-state machine controllers to control their actions. The result is a living world where friendly entities can go from doing their chores or going for a walk, to fleeing from aggressive entities or from a fight that is happening. The animations are simple and the transitions between them are as smooth as I could make them with my current assets. Another great addition was the dialogue and quest system, which gives the player the ability to receive and follow quests from the NPCs or just talk to some of them. The stats and items make the player feel like they are progressing and getting stronger. Another point is the UI of the game. It is simply designed, but contains everything the player needs to know. Lastly, the game's storyline is a small one, consisting of 4 quests and 3 additional side quests that do not clash with the storyline and give the player a quick but fun experience.

6.2 Difficulties

There were a few difficulties that got in the way while developing Nysa's Quest, but after study and research I was always able to solve them. When creating the inventory system, I found it difficult to implement the tooltip feature, and I spent some time looking in Unity's scripting documentation and on the Unity forums. Next, it was a matter of creating the AI. Initially I created a simple controller that controlled both the NPCs and the enemies, but when I started creating the finite state machines for the NPCs, the guard, and the enemy, while it was easy to create the states, I found it difficult to create the logic of the transitions, meaning I had to test, tweak, and iterate some of the transitions several times, which took a lot of time to get to the current state of the AI behavior.

6.3 Future Improvements

To improve the game, I think adding new maps/areas to explore and new quests will make the game significantly longer, more intriguing, and more fun. Next, improvements can be made to the game's audio, where I think it still lacks. More and better music tracks, sound effects, and ambient sounds can help control emotions and set the tone of a situation or story. Another thing that needs to be improved is the game's combat, as the current combat is simple and for an action RPG game it would be nice to have more attacks and complex attack combos. Adding equipment and new types of items would also be a must, because the player will spend a lot of time collecting better equipment to have a nice cosmetic look and items to help them in the fights with the enemies. There can be more improvements of course, but these 4 improvements I mentioned are the ones that are most needed at the current state of the game.

7. Bibliography

1. https://en.wikipedia.org/wiki/Unity_(game_engine)
2. https://www.freecodecamp.org/news/unity-game-engine-guide-how-to-get-started-with-the-most-popular-game-engine-out-there/
3. https://en.wikipedia.org/wiki/Game_engine
4. https://en.wikipedia.org/wiki/Action_role-playing_game
5. https://en.wikipedia.org/wiki/Artificial_intelligence
6. https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games
7. http://gameaibook.org/book.pdf
8. https://docs.unity3d.com/ScriptReference/GameObject.html
9. https://docs.unity3d.com/Manual/Components.html
10. https://docs.unity3d.com/Manual/Prefabs.html
11. https://docs.unity3d.com/Manual/class-ScriptableObject.html
12. https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity3.html
13. https://docs.unity3d.com/Manual/UsingTheEditor.html
14. https://docs.unity3d.com/Manual/Toolbar.html
15. https://docs.unity3d.com/Manual/Hierarchy.html
16. https://docs.unity3d.com/Manual/UsingTheSceneView.html
17. https://docs.unity3d.com/Manual/GameView.html
18. https://docs.unity3d.com/Manual/UsingTheInspector.html
19. https://docs.unity3d.com/Manual/ProjectView.html
20. https://gamedevbeginner.com/coroutines-in-unity-when-and-how-to-use-them/
21. https://docs.unity3d.com/Manual/LightingInUnity.html
22. https://docs.unity3d.com/Manual/AnimatorControllers.html
23. https://docs.unity3d.com/Manual/AnimatorOverrideController.html
24. https://www.youtube.com/watch?v=mhEiJ_-jyTs
25. https://www.youtube.com/watch?v=V75hgcsCGOM
26. https://docs.unity3d.com/Manual/Navigation.html