THE FINAL SPELL

3D RPG VIDEO GAME **DEMO DESIGN** AND **DEVELOPMENT** SUPPORTED BY **AGILE PRACTICES** FINAL BACHELOR'S DEGREE PROJECT

BY DAVID GÓMEZ RUIZ

Supervised by: Diego José Díaz García

Acknowledgements

There are no big projects without hardworking people, and I would like to thank Cristian Cantos for not only being that type of person, but also following my ambitious ideas since the beginning. As Louisa May Alcott once said, "It takes two flints to make a fire".

I would also like to thank Diego Díaz for his support during the whole process of this project, especially for making all the meetings enjoyable - something not easy considering how serious Cristian and I are - and appreciating all the work we have been doing these four long months.

Last and not least, thanks to my family for accepting each and every decision I have made, and making the path warmer and prettier since the beginning.

Abstract

This document describes the process of developing the demo of a 3D RPG video game called “The Final Spell” for the Bachelor’s Thesis in the Bachelor's Degree in Video Game Design and Development, including the design, implementation, art, music and teamwork required for the creation of said demo.

Specifically, this document emphasizes the development from a programming view, explaining the implementation of the different mechanics of an RPG, such as the main character movement and spells, the interactions with NPCs - dialogues, quests… -, the interactions with the environment and a save system.

The game has been developed using Unity Engine and coded in C#.

Keywords

Game development, game programming, game art, game design, agile practices.

Index

1. Introduction
    1.1. Motivation
    1.2. Goals
    1.3. Initial state and tools
    1.4. Related subjects
2. Game design
    2.1. Story and setting
    2.2. Exploration and quests
    2.3. Items and inventory
    2.4. Character actions and abilities
    2.5. Experience
    2.6. Combat
    2.7. Interface and graphics
3. Code
    3.1. Clean code
        3.1.1. Scriptable objects
        3.1.2. Custom inspector structure
    3.2. Main character
        3.2.1. MainCharacter component
        3.2.2. MainCharacterMovement component
        3.2.3. MainCharacterAnimations component
        3.2.4. MainCharacterCamera component
        3.2.5. MainCharacterGatherer component
        3.2.6. MainCharacterNoise component
        3.2.7. MainCharacterSpells component
    3.3. NPCs
        3.3.1. NPC component
        3.3.2. NPCAnimations component
        3.3.3. NPCDialogues component
        3.3.4. NPCQuests component
        3.3.5. NPCWaypoints component
    3.4. Enemies
        3.4.1. EnemyInfo
        3.4.2. Enemy component
        3.4.3. EnemySpawn component
    3.5. Data persistence
    3.6. Other systems
        3.6.1. Pathfinding
        3.6.2. Achievements
        3.6.3. Items

4. Art
    4.1. Characters
    4.2. Enemies
    4.3. Buildings
    4.4. Environment
    4.5. User Interface
    4.6. Shaders
5. Music
6. Team organization
    6.1. Agile methodology
    6.2. Playtestings
    6.3. About us
7. Results
8. Conclusions
9. Bibliography
10. GitHub repository

1. Introduction

This document shows the process of the creation and development of The Final Spell, a 3D RPG low-poly third person game focused on magic elements.

1.1. Motivation

Unfortunately, the game design and development degree lacks student teams with a fully working game finished. In our opinion, after finishing the degree, every student should have several games completely done by themselves (alone or within a team), with their own ideas and focusing on the part that the student is interested in: design, art, coding, music and sound creation, marketing, etc. To this end, both authors of this document have decided to show what's the potential of doing a game with all the parts required (narrative, art, programming and music) within a hardworking and resourceful team. Cristian and David, the people that have been working on this project, have more than a year of experience in the game development area, obtained with extracurricular game projects that have given both students a powerful vision of the game creation process, and the knowledge of how to handle all the development parts and how to work within a timeline.

1.2. Goals

The main objective of this work is to show the process of designing a demo of a videogame and the implementation of said design, which have been done using the Unity 3D engine. The development of the game has been made by two students, David Gómez who focused on the technical part and Cristian Cantos who was in charge of the artistic part. They were organized using agile methodologies to facilitate collaboration. The game is a fantasy RPG that features the typical mechanics of other RPGs on the market (inventory system, enemies, missions and skills), but with the innovations that distinguish an indie development team.

1.3. Initial state and tools

Before starting the project, the design of the game was already available, done during 2020 using the tool WikidPad[1]. The art and code was done from scratch starting the first day of the final project subject.

For the development of the game, Unity 3D has been used together with Visual Studio 2019 to code and Blender for the models. Discord has been the main communication tool, and the platform GitHub has given the students a way to share the code and art.

1.4. Related subjects

Due to its complexity, several subjects have been key for the development of the project:

- VJ1204 - Artistic Expression
- VJ1208 - Programming II
- VJ1216 - 3D Design
- VJ1222 - Video Game Conceptual Design
- VJ1223 - Video Game Art
- VJ1224 - Software Engineering
- VJ1226 - Character Design and Animation
- VJ1227 - Game Engines
- VJ1231 - Artificial Intelligence

2. Game design

The Final Spell is a 3D role-playing game focused on magic elements. It has most of the characteristics of the other RPGs in the game industry[2].

The first design of the game was done using WikidPad, a tool to create a wiki-like bible. Once the development of the game started, the information of the bible was divided and placed in different channels from the Discord tool, where the team could organize, check and revise the whole story and design of the game.

2.1. Story and setting

Since the Great Fire happened, La Acacia has been in balance separating virtue from danger, prosperity from ruin, the human race from the dark beings, and the North from the South.
A constant war, started several millennia ago, confronts the most privileged with the dark. Both sides have the same objective: to unify the continent towards their side.
The North is protected by the Summit of the Wizards, chosen for their skills and potential and made the most powerful in the upper half of La Acacia.
The South, however, is invaded by the so-called dark beasts, who have controlled every corner of their side, taming - with dark magic - all living beings in their way.
A few years ago, a spy wizards team managed to penetrate deep into the hot, dangerous and heavily populated lands of the South, discovering something that could defeat the North and end the war.

In The Final Spell, the player manages a sorceress that has just arrived to the Valley of Immortality, the place where every sorcerer and sorceress is taught and prepared to face the real world, helping with the war against the South.

The story revolves around the discovery that the spy wizards team made, something that the player has to find and understand taking information from different NPCs along the game (beyond the demo). This team, on one of the trips to investigate the South to analyze weak points, found a small town where dark high ranks were experimenting with the creation of dark sorcerers and sorceresses, something never done before. In that trip, the team captivated one of the experiments, a baby girl half sorceress half dark beast. This baby girl would evolve into a secondary character, which faces the player at the end of the demo, and that is a deuteragonist[3] whose relation with the main character will determine the future of the world.

2.2. Exploration and quests

The demo is divided into four areas, separated by the roads that join through bridges the Valley of Immortality with the rest of the world.

The area between the north and east roads is formed by the library, the maze and the cage. The library is the only building in the demo that can be accessed. It has two floors, although only the first one can be seen by the player. The cage has a spawn of wild pigs, and is connected to the first and second main quest.

Between the east and south roads the Command Center is built, with the forbidden nook placed behind the building. The forbidden nook is connected with the secondary quest.

The rest room and the plains - connected with the third main quest - are located between the south and west roads.

The area between the west and north roads is formed by the rooms and the secondary character in charge of the pets achievement.

Finally, the fourth roads are connected with the training area, where the instructor and the deuteragonist character are located. The fourth and last main quest of the demo happens there.

Quests are divided into main and secondary, and the demo has the following ones:

First main quest	Kill 1 wild pig (cage). Given by the instructor (training area)
Second main quest	Kill 5 wild pig (cage) Given by the instructor (training area)
Third main quest	Kill 8 butterflies (plains) Given by the instructor (training area)
Fourth main quest	Fight and win another apprentice (training area) Given by the instructor (training area)
Secondary quest	Kill all slimes (forbidden nook) Given by a kid (entrance forbidden nook)

Apart from the quests, the player can complete funny achievements that are focused on exploring the world. Completing an achievement unlocks books from the library, and those books contain lore related to the achievement completed. There are two achievements available on the demo:

Pet the dog	Pet 10 times any dog and talk with the secondary character with dogs. Can be completed in front of the rooms.
Complete the maze	Finish the maze and talk with the sorceress. Can be completed behind the library.

2.3. Items and inventory

There are different items around the world that the player can get by buying, gathering or killing enemies. Depending on it’s utility, there are three main types of items: resources, consumables and saleables. For the demo, those are the items inside each category:

Resources	Geranium
Consumables	Health potion
Saleables	Dark essence
	Butterfly wing
	Tusk
	Horn
	Slime liquid

Resources and consumables can be bought from the merchant, who can buy from you any saleable item the player has in the inventory.

Each enemy has different item pools that can spawn when being killed: wild pigs can spawn items from the wild pig pool and the dark enemy pool, butterflies from the butterfly and dark enemy pools, and slimes from the slime and dark enemy pools. Each pool has unique items:

Dark enemy pool	Dark essence
Wild pig pool	Tusk Horn
Butterfly pool	Butterfly wing
Slime pool	Slime liquid

2.4. Character actions and abilities

Hablar con NPCs, acariciar animales
Diferentes habilidades (Quick Water Drop, Floral Flame, Lethal Orb). Explicar que es una de las claves del proyecto: habilidades difíciles a priori, pero cuanto más se usan más fáciles son de hacer (práctica). Las habilidades tienen fases y requieren de recursos, y el jugador debe ir descubriéndolas con prueba y error

2.5. Experience

As in most RPGs, the main objective of the player is to save the world. To do that, instead of having stats that increase throughout the game, the player must focus on his real experience: learning and mastering the different abilities that require precision, organization and resources. A player whose abilities are above average will complete the game faster. On the other hand, a player that requires more resources to learn and has less experience with RPGs will need more time to master the game and fully complete it. This way, the game doesn't have a fixed length, it depends on the player and there aren't time limited events or long, tedious tasks that can't be done fast.

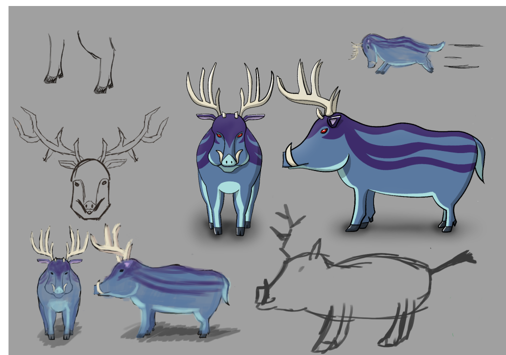
2.6. Combat

Quests and story progression depend on the player defeating different enemies or NPCs. The combat is based on real time fighting where the player has to use complex spells to deal damage.

There are three enemies available in the demo: wild pigs, butterflies and slimes.

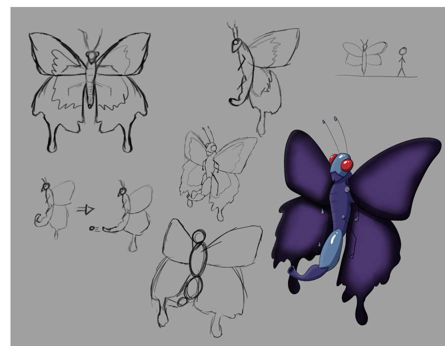
Wild pigs are the first enemies the player has to face. They need to be close to the player to hit them (melee attack), and use their head to deal a small amount of damage.

They've been designed to be easy to defeat but funny to fight.



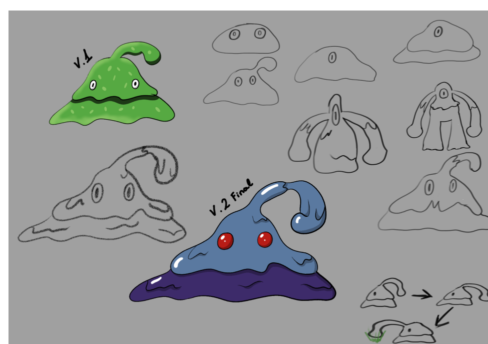
Butterflies are medium-range enemies that throw balls to the player dealing a high amount of damage. Their attacks are strong, but they have a low amount of health.

The balls are projectiles that target and follow an objective until they reach it.



Slimes are the stronger enemies in the game. Fighting them is optional - part of a secondary quest.

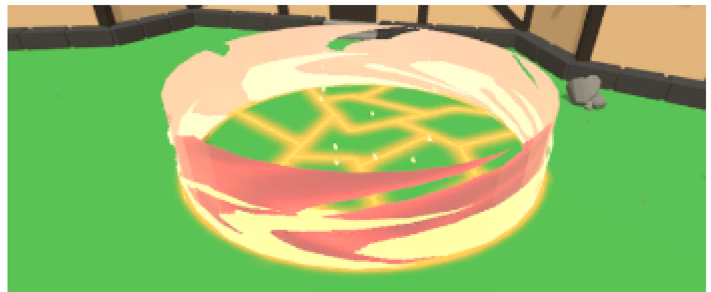
They are big and deal a big amount of damage, but close range.



The enemies can be fought using the three available spells in the demo: Quick Water Drop, Floral Flame and Lethal Orb.

The Quick Water Drop is performed in two steps: right click + left click. The direction of the projectile is the same as the direction the character is facing.
It requires a water source close (river, lake or training area water).

The Floral Flame has x steps: it starts like the Quick Water Drop (right click), then the player must crouch (ctrl), and use left click three times.
It needs flora around the burning radius (grass or flowers), which burns once used.



The Lethal Orb is done in two steps: Q and space pressed at the same time for 1 second and then the E button pressed.
To explode the orb on the second step, the player needs one geranium in the inventory.

2.7. Interface and graphics

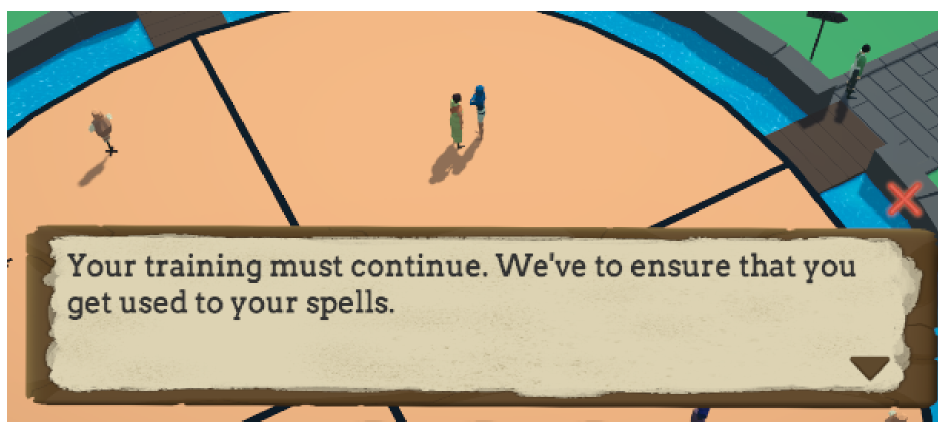
User Interface helps the player in The Final Spell to know how the situation of his character is and learn how to perform the different spells.

There's a health bar that shows the current character's life located on the upper left corner of the screen. A bar of buttons is placed on the middle down part of the interface, with the map button, the inventory button and the spells book button.

The map button shows a map from La Acacia, the world where the game takes place.
On the other hand, the inventory button displays a window with the items the player has taken (loot, shop…) and the amount. Consumables (health potions) can be used by clicking them.
And the last button, the spells book, has different pages explaining how to perform the three available spells on The Final Spell demo.

In-game UI

Dialogues are the main way the player can learn about the objectives and lore of the world. Quests and achievements are triggered with dialogues.



In-game dialogue

The last important piece of interface on The Final Spell is the shop, where the player can buy items at a fixed price from the merchant and sell enemies loot to earn gold.

In-game shop

The art of the game is low-poly with plain colors, used in many indie games for it's ease of use and performance, while satisfying industry quality standards. It will be fully explained in the art section.

3. Code

The Final Spell is programmed in an efficient and understable way. Instead of focusing on the numbers of lines, the code is focused on simple game loops carefully designed to fit perfectly in the game. As Dijkstra once said,

> "My point today is that, if we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent": the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger."[4]

Every line of code has been written having in mind the clean code principles that will be explained below. Every system is divided into small systems that can work together, making it easy to modify, delete or add characteristics to an existing system.

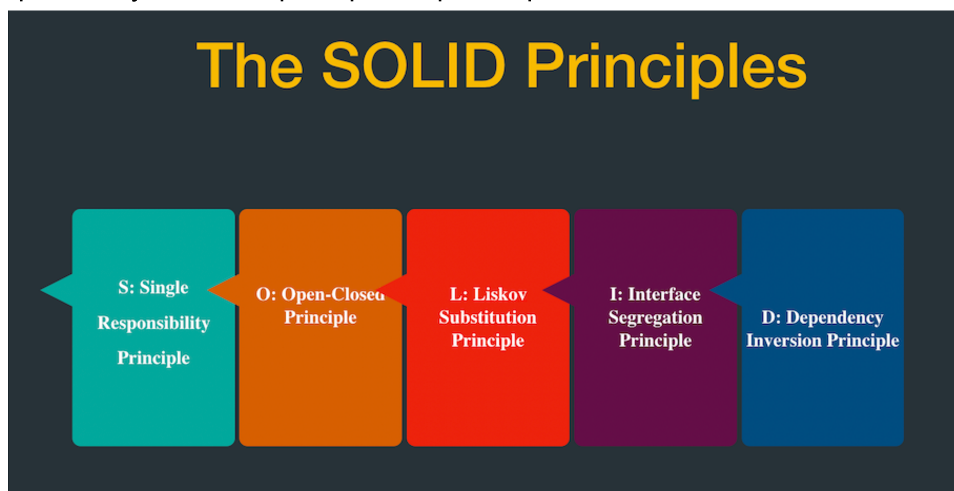
The game has been coded in C#, using the Visual Studio 2019 IDE, and designed with simple diagrams that explain the main functionality of every system, some of those diagrams will be explained in this section.

There have been several programmers that served the team as references and whose tutorials, tips and knowledge have been widely used on this project, like Freya Holmér[5], Sebastian Lague[6] and Liam Sorta[7].

3.1. Clean code

The Final Spell is coded following the SOLID principles[8]:

- Single responsibility principle: every class should have only one use
- Open-closed principles: entities should be open for extension but closed for modification
- Liskov substitution principle: objects in a program should be replaceable with instances of their subtypes without altering the quality of the program
- Interface segregation principle: many client-specific interfaces are better than one general-purpose interface
- Dependency inversion principle: depend upon abstractions, not concretions



DevX Blog - The Solid Principles by AbdulMujeeb Aliu[9]

There are also other techniques used so the code is more readable and makes it easier to implement and update objects in the game.

3.1.1. Scriptable objects

Those objects can store data and other GameObjects can access that data without having to instantiate the object inside a scene. It's specially useful when using lists of objects: we can code a "base" object and create different objects with different properties from that base. It's widely used on this project: enemies, items, achievements, spells…
The use of scriptable objects helps when adding new objects: without coding, we can add and modify new objects to a list. This way, we don't need to use databases, making it faster to save and load objects.

Scriptable objects[10] are C# classes that inherit from ScriptableObject, and those classes can be instantiated in the project folder, storing data and with the possibility of having functions non derived from the MonoBehaviour class (like OnMouseDown or Update).

One example is the ConsumeItem action, used on the MainCharacterSpells system (explained below).

```
[CreateAssetMenu(fileName = "ConsumeItem", menuName = "Spells/Actions/ConsumeItem", order = 0)]
public class ConsumeItem : IAction
{
    [SerializeField] private Item item = null;
    [SerializeField] private int amount = 0;

    public override void DoAction()
    {
        GameObject.FindGameObjectWithTag("Player").GetComponent<MainCharacterInventory>().RemoveItem(item, amount);
    }
}
```

The "CreateAssetMenu" attribute lists the object inside the create menu, just as any other object that can be added to a folder (C# script, Material…).
In this case, IAction inherits from ScriptableObject.
The "item" and "amount" fields, which are private but serialized so it can be accessed on the editor, stores data that will persist between game sessions, as that data is part of the scriptable object saved on the folder.
The "DoAction" method runs a small piece of code to update the inventory from the player. It doesn't use anything related to MonoBehaviour (like the transform or gameObject attribute).

3.1.2. Custom inspector structure

Every GameObject follows a structure that makes it easier to modify and debug a GameObject without coding:

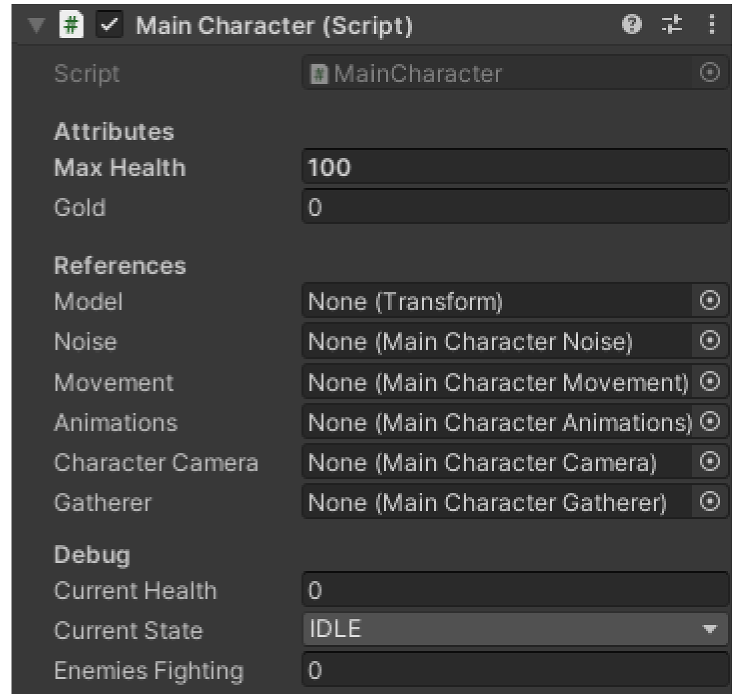
- Attributes header where customizable variables are displayed
- References header where dependencies are stored (loaded on game start)
- Debug header where not editable variables are displayed

With that structure in mind, anyone could test the GameObjects without using the debug console or modifying the classes.

3.2. Main Character

In The Final Spell, the main character is the tool that lets the player interact with the world: fight, travel, buy, talk… It is also the most complex system, and that's why it is divided into several components, each of them focusing on one task, following the single responsibility principle (SOLID): movement, animations, camera, spells, noise and gatherer. All of them (except the spells) are connected using a parent component, called MainCharacter.

3.2.1. MainCharacter component



This component is the one in charge of connecting all the small pieces that form the main character (noise, movement, animations, camera and the gatherer). It serves as the main way to send information between one component and the other. It also handles the player's health and the model.

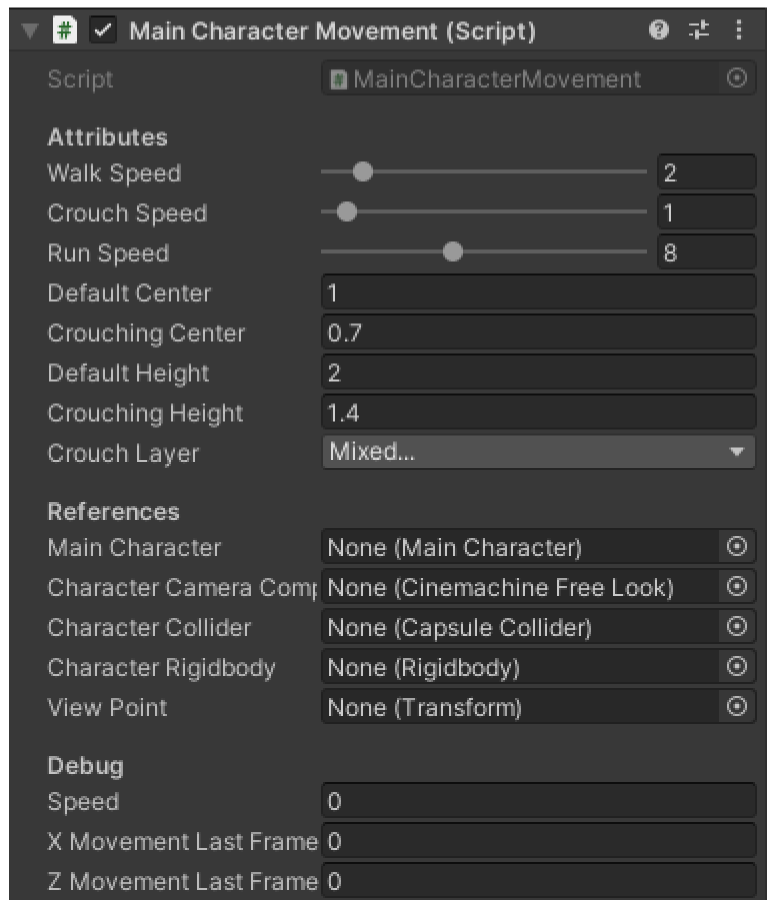
If any of the components referenced is not found at the start of the game, the character will disable itself and send a warning message, keeping the other game systems working.

Following the guidelines presented before, the inspector is divided into:

- Attributes:
    - Max health: specifies the maximum and starting health the main character has.
    - Gold: amount of gold the player has.
- References:
    - Model: stores a direct reference to the transform where the main character model and animator is.
    - Noise: MainCharacterNoise component.
    - Movement: MainCharacterMovement component.
    - Animations: MainCharacterAnimations component.
    - Character Camera: MainCharacterCamera component.
    - Gatherer: MainCharacterGatherer component.
- Debug:
    - Current Health: starts with maximum health value, and gets updated when receiving damage.
    - Current State: starts on idle, gets updated with player movement/spells.
    - Enemies Fighting: stores the amount of enemies the player is fighting. This way we can know when the player is in or out of a fight.

3.2.2. MainCharacterMovement component

The way the player moves in The Final Spell works like in most of the RPG games[11]: using WASD or the keys, the player can move around the world freely. He can also crouch to go through spaces with small doors or run.



In order to keep the movement smooth and logical with the camera position, there have been several handicaps that needed some tweaks: the first one is the local axis of the player, which changes when rotating the camera. That means that when the player walks in one direction, if the camera is rotated, that direction should remain the same for the player, which means updating the local axis to align the new direction. To this end, the view point position was created, which is updated with the camera rotation and serves as a guide to know how to align the axis.

The other handicap was dealing with the smooth movement and physics: moving an object in the Update function (the number of times it runs depends on the performance of the computer) glitches the main character model, letting the player go through other objects with colliders and giving wrong visuals (like the model vibrating). On the other hand, moving an object in the FixedUpdate function (it happens every 0.02 seconds independently of the computer performance) could lead to missing player inputs: if the player presses a button between FixedUpdates, that button input will be lost. The fix for this problem has been to run the player input in Update and store it (xMovementLastFrame and zMovementLastFrame), and move the player on the FixedUpdate (using the stored information from the Update)[12].

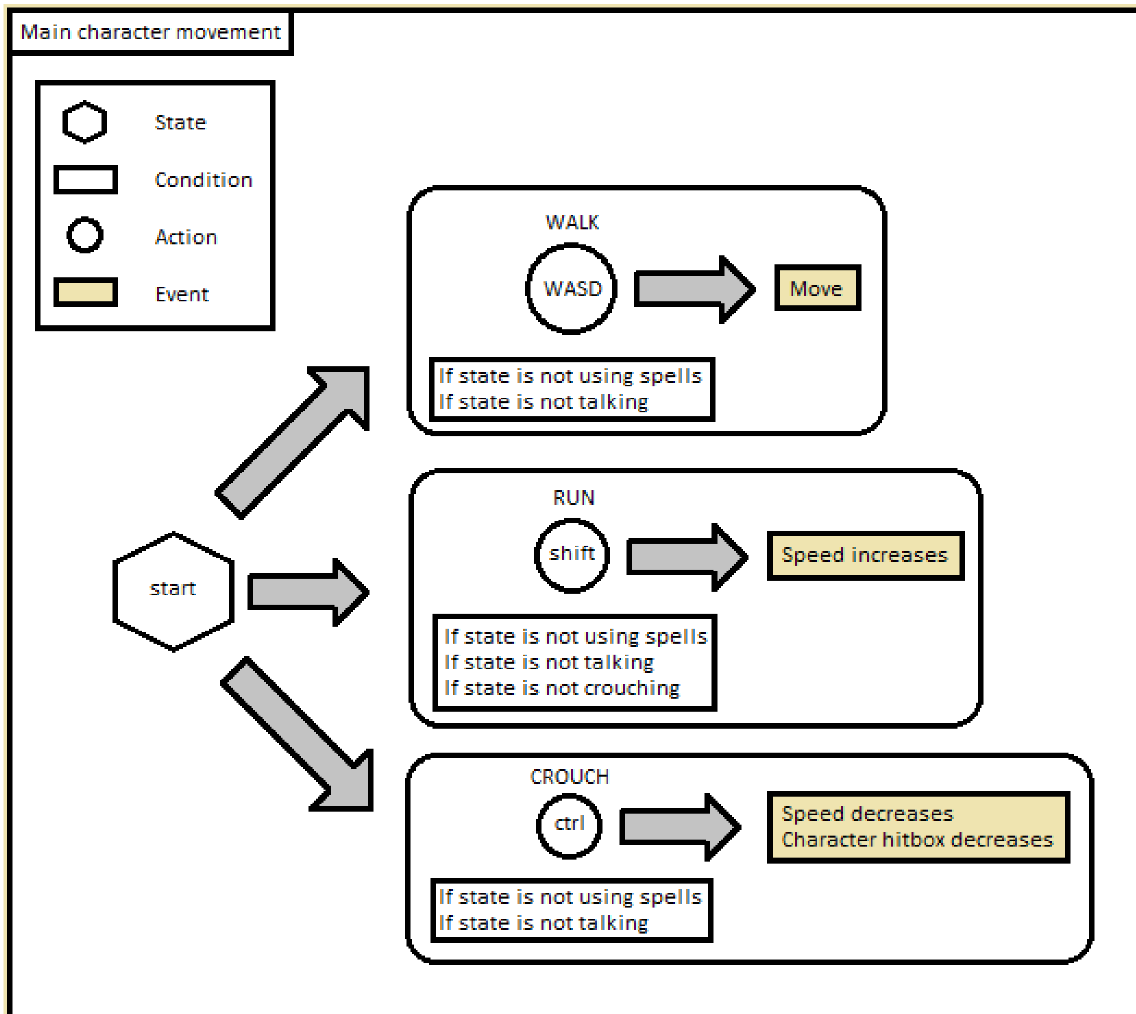
The inspector is divided into:

- Attributes:
    - Walk Speed: speed the player has when walking.
    - Crouch Speed: speed the player has when crouching.
    - Run Speed: speed the player has when running.
    - Default Center: the default center position of the character collider.
    - Crouching Center: the center position of the character collider when crouching.
    - Default Height: the default height of the character collider.
    - Crouching Height: the height of the character collider when crouching.
    - Crouch Layer: the layers that the character can't go through, which are checked when trying to stop crouching. If the main character has an object

above itself with one of those layers when the player stops crouching, the main character will keep crouching.

- References:
    - Main Character: MainCharacter component.
    - Character Camera Component: Cinemachine Free Look component that handles the main character position.
    - Character Collider: Capsule Collider component that delimits the size of the model.
    - Character Rigidbody: Rigidbody component that handles the physics of the main character.
    - View Point: transform with the position where the main character should look at.
- Debug:
    - Speed: current character speed.
    - X Movement Last Frame and Z Movement Last Frame: movement calculated with the player input that happened on Update and will be used at FixedUpdate.

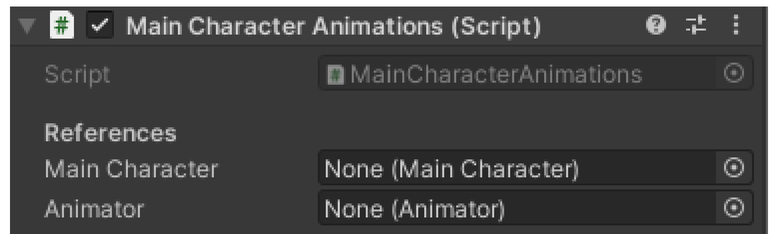
The way the movement has been implemented follows the next diagram:



3.2.3. MainCharacterAnimations component

All the main character animations are handled in this component: movement, spells and others.

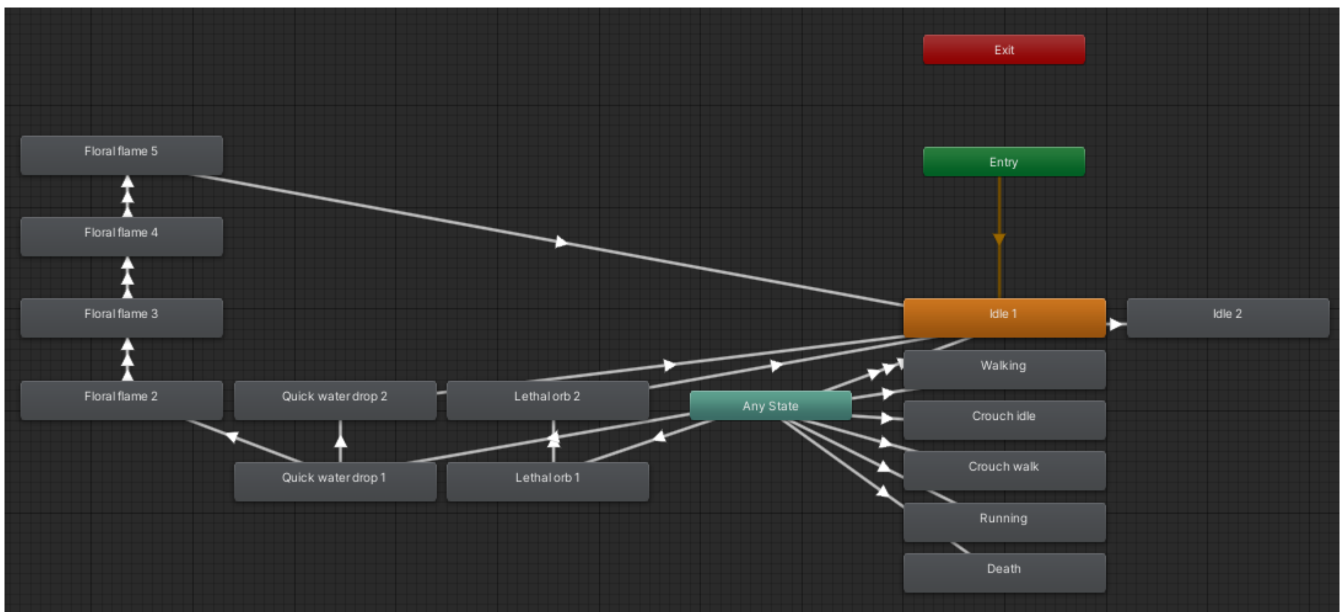
This component is also the only one that can interact with the animator.



The inspector only needs a couple of references to make the component work correctly:

- References:
    - Main Character: MainCharacter component.
    - Animator: Animator component.

The animator works mainly with triggers and ids for the spells:

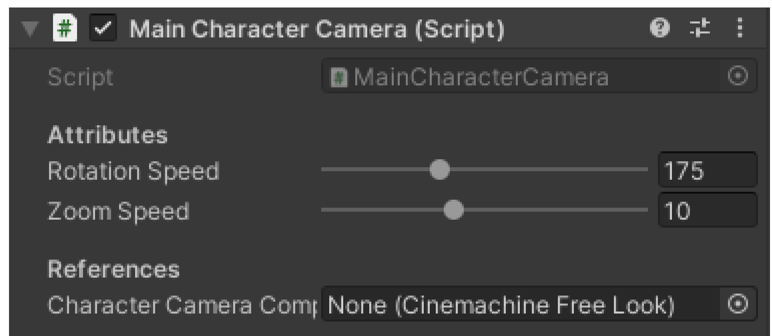


All spells end on Idle 1, which is the default idle.
When the player is on idle for more than 5 seconds, the animator runs the Idle 2 animation.

Having the SOLID principles in mind, adding a new animation (like talking or dancing) would need just a trigger (and its parameter) and a connection with Any State. On the other hand, removing one animation would be even simpler: just delete the trigger parameter and the animation itself.

3.2.4. MainCharacterCamera component

The player can explore the world using a Cinemachine Free Look Camera[13], which focuses the player but can be rotated or zoomed.



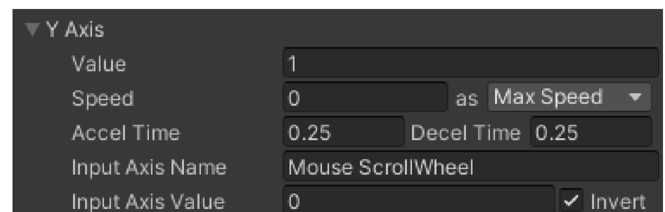
The MainCharacterCamera component is heavily connected with the Cinemachine Free Look Camera. It handles the rotation and zoom.

The inspector has the following information:

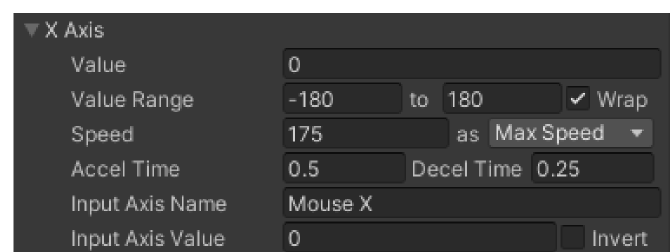
- Attributes:
    - Rotation Speed: speed at which the camera rotates when the player moves the mouse in the X axis.
    - Zoom Speed: speed at which the camera zooms in/out when the player uses the zoom button (mouse scroll wheel on computer).
- References:
    - Character Camera Component: the Cinemachine Free Look camera component where the rotation/zoom speed are modified.

On the other hand, the Cinemachine Free Look Camera component has been modified to give a player a sense of smoothness when interacting with the camera.

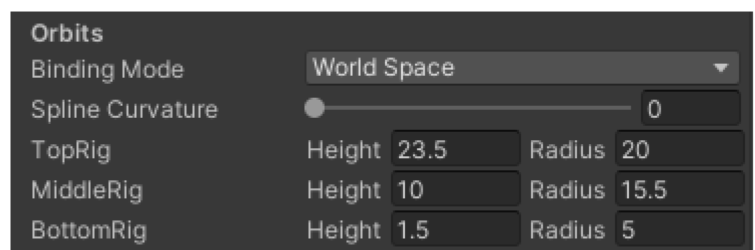
The zoom (Y axis) is inverted to follow the common player controls. It has an acceleration and deceleration time to make the camera zoom smoother. The speed is controlled by the MainCharacterCamera component.



The X axis, which moves the camera around the player, has a bigger acceleration time so the player can make smaller rotations. The speed is also controlled by the MainCharacterCamera, which updates this component depending on the speed of the player mouse.

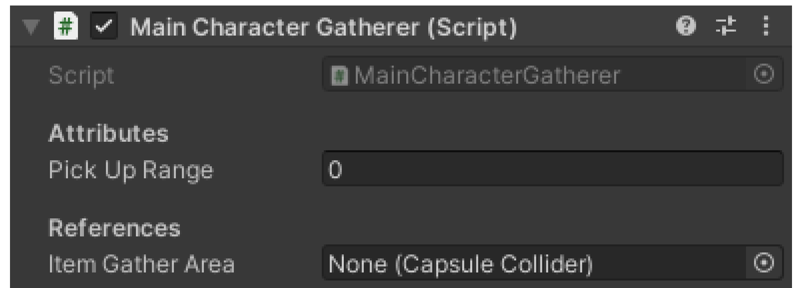


The last important feature of the Cinemachine Free Look Camera are the orbits, which have been defined focusing the center of the main character model. When the camera is closer, the radius and the height is smaller.



3.2.5. MainCharacterGatherer

When enemies spawn items, those items stay in the world until they get into the gatherer area of the main character. This area can be modified by external scripts, making it bigger or smaller. For example, when buying a special item that improves the area.

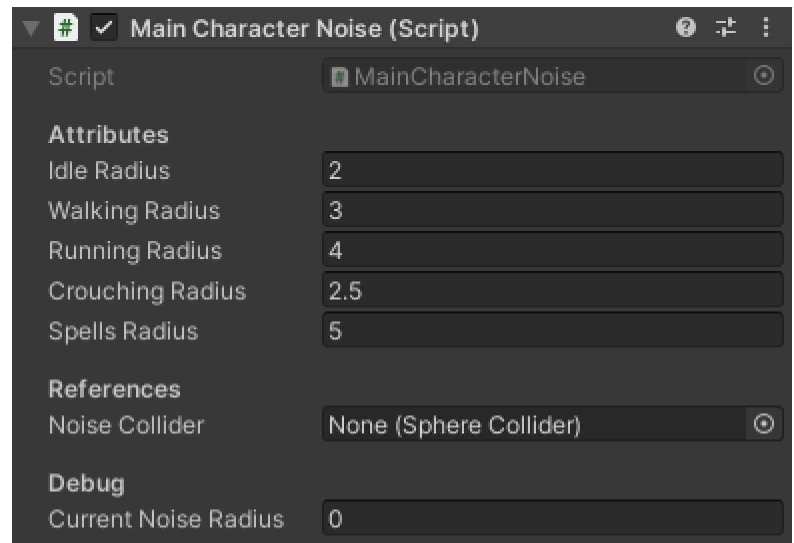


The inspector, as the other scripts, is divided into:

- Attributes:
    - Pick Up Range: range of the Item Gather Area.
- References:
    - Item Gather Area: Capsule Collider that defines the area where the player can automatically pick up items placed on the world.

3.2.6. MainCharacterNoise component

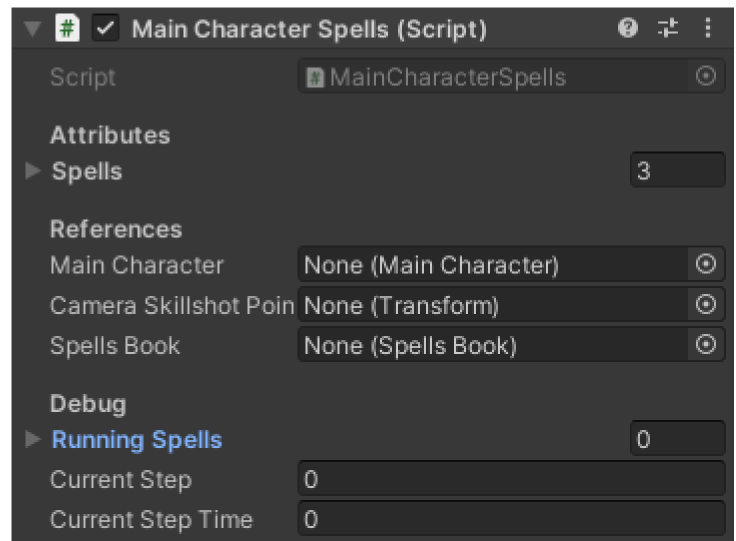
The way the enemies know when to start chasing the player is when the enemies enters the Noise Collider, which radius depends on the current state of the player: when the player is doing something noisy like running or using spells, the trigger area (the collider) is bigger; and the other way around: when the player is doing something quiet, like standing still or crouched down, the trigger area is smaller.



As any other component, the inspector is divided into:

- Attributes:
    - Idle Radius: noise radius when the player is idle.
    - Walking Radius: noise radius when the player is walking.
    - Running Radius: noise radius when the player is running.
    - Crouching Radius: noise radius when the player is walking while crouched down.
    - Spells Radius: noise radius when the player is using a spell.
- References:
    - Noise Collider: stores a reference to the collider that handles the triggers.
- Debug:
    - Current Noise Radius: radius that the Noise Collider has.

3.2.7. MainCharacterSpells



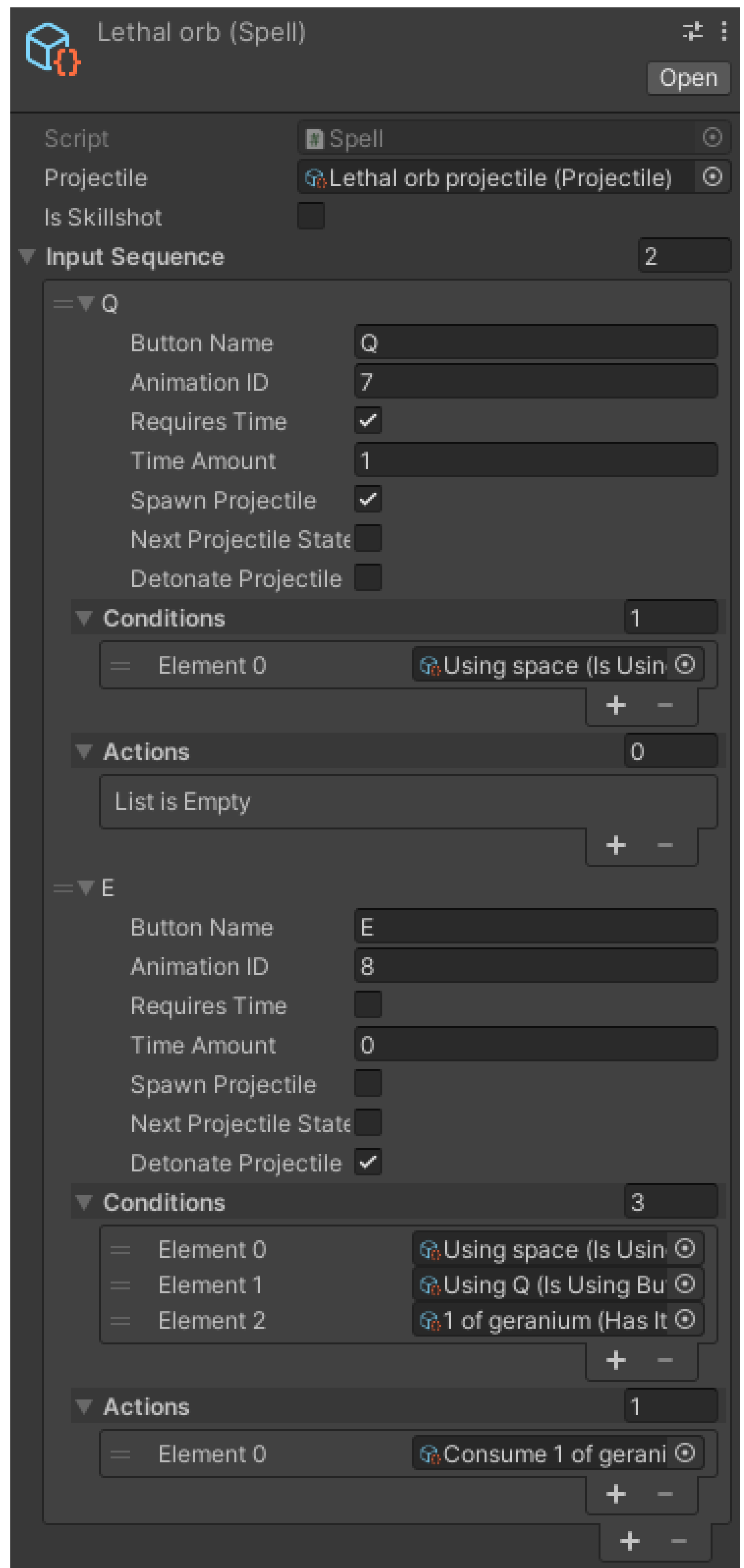
The system in charge of the execution of the spells is probably the most complex component of the main character - and the game itself -. This component relies on the Spells scriptable object, which will be explained later on this section. Every time the player uses the keyboard or the mouse, this system checks if there's a spell that uses one of the buttons or keys pressed. Once a spell is started, it is stored and everytime the player uses the next input required by the spell to continue, the next spell animation is run and the step is increased, until the spell reaches the last step. When the player stops using the keyboard/mouse, when the spell ends or when the input used by the player doesn't match any spell, the system is restarted and able to accept new spells.

One interesting feature about this system is that it is implemented in a way that several spells with the same beginning can be performed at the same time, creating a branch-like system where the player isn't stuck to only one possibility until the steps followed are only part of one spell. For example, the player could start positioning to perform either Floral Flame or Quick Water Drop - both start with the right click pressed and same animation -, and with the both spells started, the player can choose one of those spells. This could lead to interesting player versus player fights, where the opponent wouldn't know which spell is being performed until the end.

In order to see the full implementation, this script can be found on the GitHub repository (section 10), inside the Game Engine folder > Assets > Code > Main character > Spells > Scripts > MainCharacterSpells.cs[14].

The inspector is divided into:

- Attributes:
    - Spells. The list of Spells scriptable objects that the player can perform.
- References:
    - Main Character: the MainCharacter component.
    - Camera Skillshot Point: the point where the camera has to look at when doing a spell with skillshot.
    - Spells Book: the UI with spells information.
- Debug:
    - Running Spells: current possible spells being performed by the player.
    - Current Step: current step of the spells.
    - Current Step Time: time that has passed in the current step. Used for spells that require specific time intervals to continue performing the next step.



On the other hand, the Spell scriptable object has the guidelines to perform a specific spell.
The first field of the spell component indicates which projectile to spawn.
The second field tells the spells system if this spell is a skillshot - like the Quick Water Drop -, so the player has to face where the camera is looking at.
The next field is a list of PlayerInputs, ordered by execution time. First input is the start of the spell, last input is the end.

The key of the spells is the PlayerInput, a serialized class that have the following attributes:

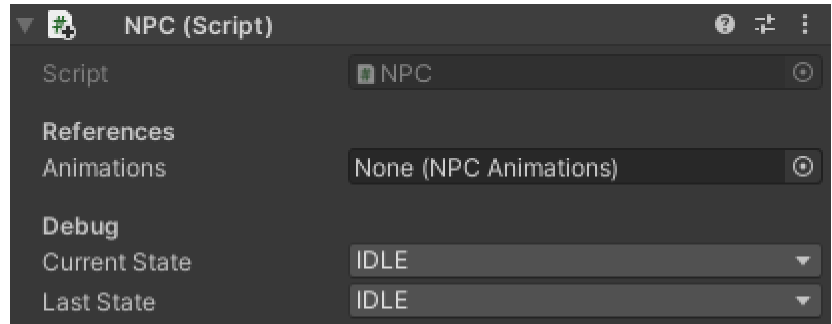
- Button Name. The name of the button to use in order to perform that step of the spell. Must have the same name as in the Input Manager from Project Settings (Unity Engine)[15].
- Animation ID. Serves as a way to identify which spell animation to run on the MainCharacter animator.
- Requires Time. If the step needs to be performed for a specific time amount.
- Time Amount. In case that the step requires time.
- Spawn Projectile. Indicates at which step the projectile has to be spawned. True means that the projectile is spawned at that step.
- Next Projectile State. If the projectile is spawned and it needs to transition to a new step, this field will be true.
- Detonate Projectile. Indicates at which step the projectile explodes.
- Conditions list. The list of conditions that the player has to meet in order to perform that step of the spell.
- Actions list. Actions performed once the step finished.

3.3. NPCs

The Final Spell world is filled with different NPCs. Every NPC has different objectives: some carry the quests that will let the player understand the story, some will help the player in one way or another, some will just be there filling the town with their presence. The main types are quests givers, merchants and sorcerers. The three of them can have quests, dialogues and routes.

3.3.1. NPC component

This component has the same function as the Main Character component: it serves as the main handler of the NPC. But, in this case, it works a bit differently: as quests, dialogues and routes are optional, they can work autonomously, while the NPC component handles the animation and the state of the NPC.

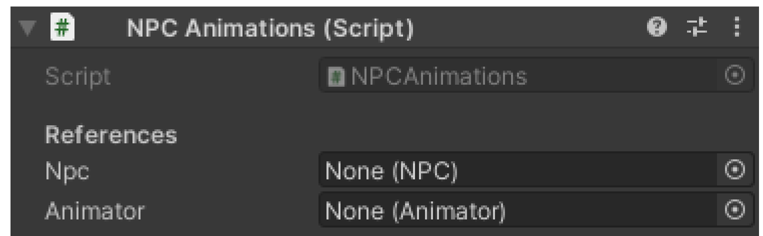


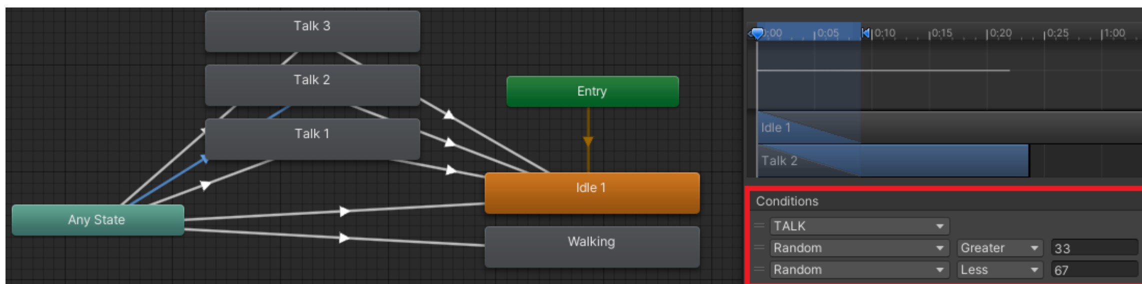
The inspector is organized as this:

- References:
    - Animations: NPC Animations component.
- Debug:
    - Current State: current state of the NPC.
    - Last State: the last state the NPC was. Used for resuming previous states. An example: when the player talks (current state) with a NPC that is walking (last state), it will continue walking once the dialogue is finished.

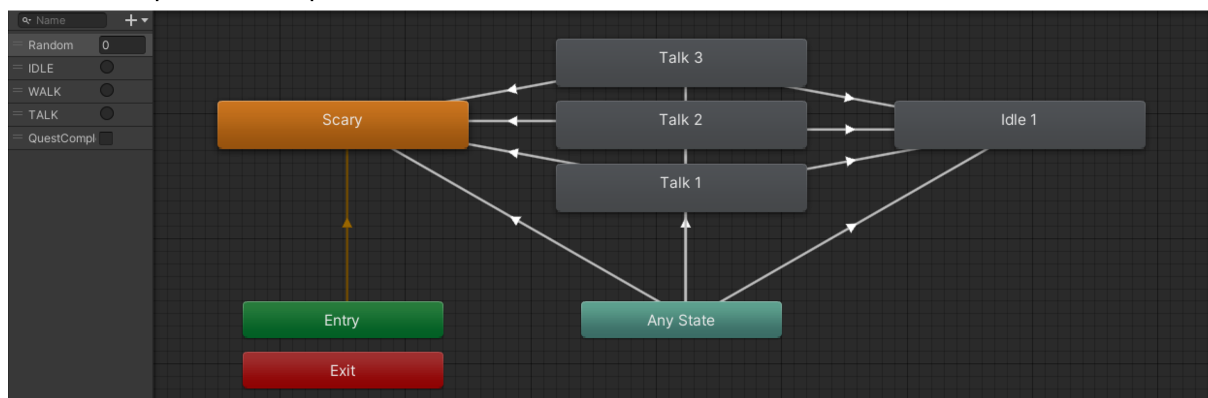
3.3.2. NPCAnimations component

In order to handle the animations, NPCs follow a similar system used for the Main Character. Once the NPC component updates the state, it updates the current animation depending on the state. One interesting feature that differentiates from the Main Character is that the NPC animations can be randomized. When transitioning to a new state, a random value is assigned between 0 and 100. For example, when transitioning to “talk” animation, there’s a 33% change it will run one of the three different animations.





As any other system on The Final Spell, the animator keeps the SOLID principles, making it possible to expand the animations easily, or substitute the already existing ones. An example is the way the kid in charge of the first secondary quest works, which has the walking animation removed and an alternative to idle: the “scary” state, which is enabled before the quest is completed.



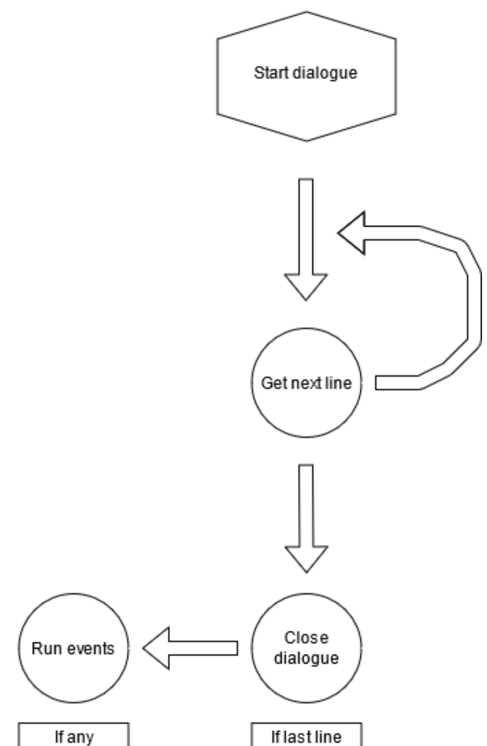
When the player interacts with the kid, the talk animations will work as expected for any other NPC. Once those talk animations end, the next animation will be either scary or idle, depending on the “QuestCompleted” boolean, which is updated by an event when the secondary quest is completed.

3.3.3. NPCDialogues component

Dialogues are the main tool used for teaching the player and explaining quests and lore about the world.

NPCs have different dialogues and dialogue types. The four main types are:
- Quest dialogue that explains the player how to complete a quest and what will be the reward.
- Achievement dialogue that unlocks a book in the library.
- Story dialogue that explains important and relevant facts about the world lore.
- Recurrent dialogue that appears several times when story/quests/achievements dialogues are not available.

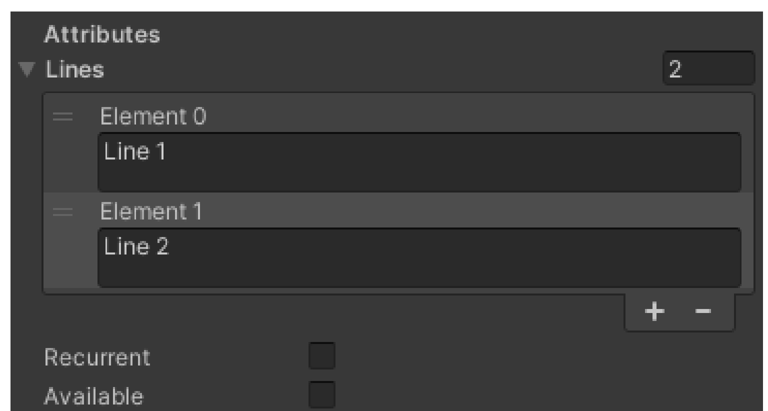
The way the dialogues are implemented is simple:
- When the player starts interacting with an NPC, if it has dialogues, it will take a random dialogue (ordered by priority - story, achievement and quest dialogues will go first).
- The dialogue system will take the next line, which in this case is the first line, and will display it on screen.
- If the player presses the "next" button, it will get the next line.
- If the current line is the last line, pressing the "next" button will close the dialogue.
- After closing the dialogue, all events required will be performed: unlock books in the library, enable an NPC interaction, etc.



This system is built upon the SOLID principles, letting us use it for any dialogue-related event, handling not only simple dialogues but also quests and other important interactions between the player and the NPCs.

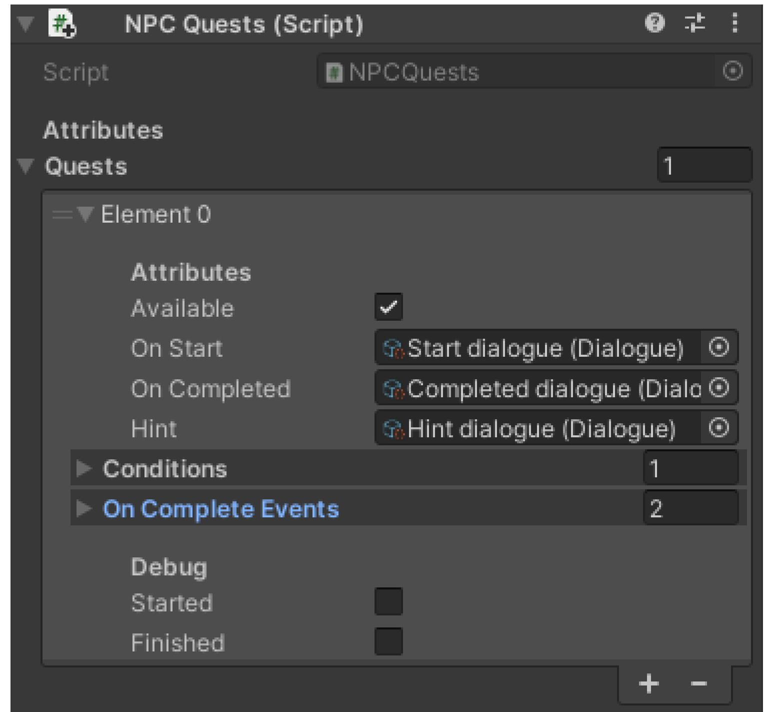
The component NPCDialogues is added to the NPCs with dialogues. It stores the dialogues for each NPC, the current listener and the interaction between the NPC and the listener.

On the other hand, dialogues are scriptable objects that follow the standard guidelines for the inspector:
- Attributes:
    - Lines: array of dialogue texts ordered.
    - Recurrent: if that dialogue can run more than once.
    - Available: if that dialogue has been unlocked.



3.3.4. NPCQuests component

The story is presented with quests. There are mainly two types: the story quests, which will let the player progress in the world; and the secondary quests, which reward items and are a fun way to let the player interact with the world out of the main story.

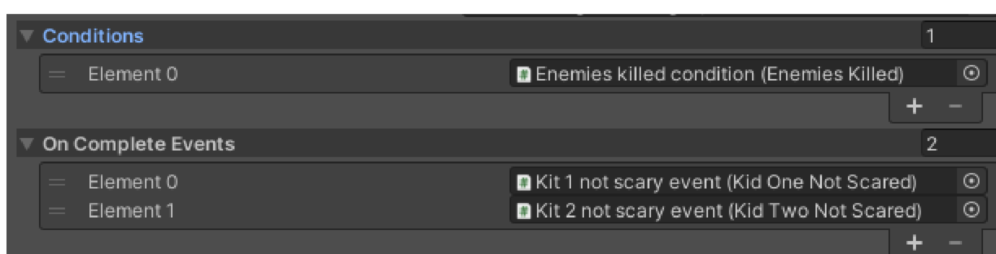


Following the guidelines, the NPC Quests component has an attribute area, which in this case only handles the list of quests. This list is filled with objects of type Quest. A NPC can have different quests, and the way the player can complete them is the way they're ordered on the quests list of this component.

The Quest component, on the other hand, is presented as this:

- Attributes:
    - Available: if the quest can be done
    - On Start: dialogue that will be displayed the first time the character knows about the quest
    - On Completed: dialogue that will be displayed when the quest is completed.
    - Hint: dialogue that will be displayed whenever the player talks with the NPC after starting the quest and before completing it. Will help the player explaining again what needs to be done.
    - Conditions: a list of IQuestCondition objects. The quest will be completed once all the conditions are satisfied.
    - On Complete Events: a list of IEvents that will happen when the quest is completed.
- Debug:
    - Started: if the quest has started.
    - Finished: if the quest has finished.

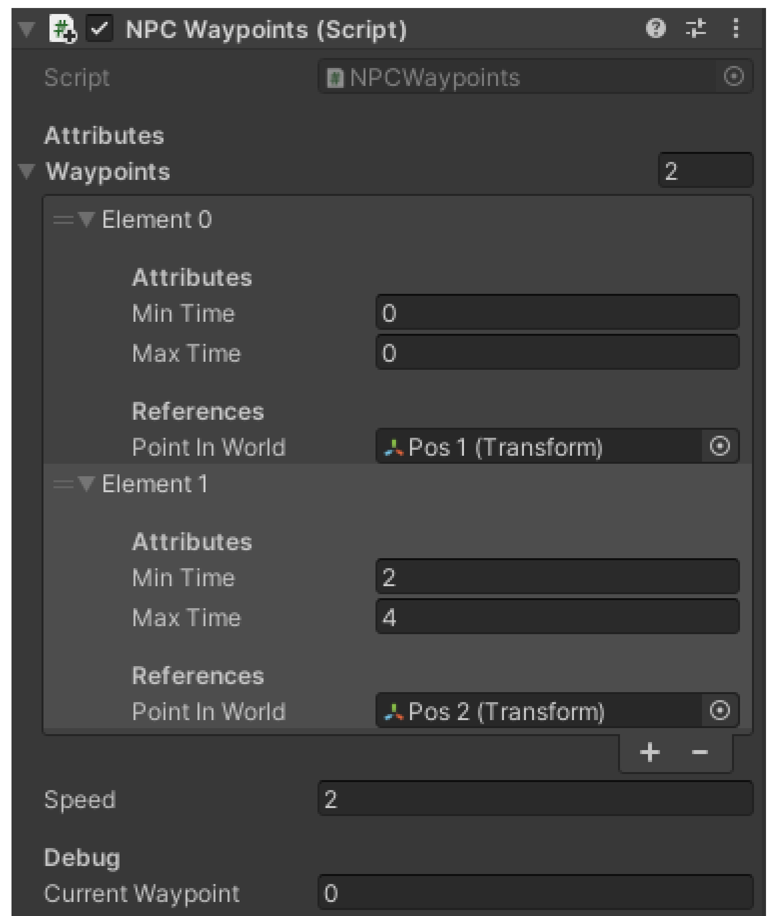
The IQuestCondition and IEvent objects are crucial to the good development of the quests. Here's an example of the first secondary quest, where the player has to kill all the enemies in a spawner (hence the "Enemies killed condition", that checks if the selected spawner is empty). Once completed, the kids will no longer be scared, changing their animations.



3.3.5. NPCWaypoints component

Some friendly NPCs have routes to make the world of The Final Spell feel alive. Those routes are formed by a list of waypoints.

Every NPC with a route has the NPCWaypoints component, that handles the movement and animations of the NPC.



The inspector is divided into:

- Attributes:
    - Waypoints: an array of waypoints. The NPC will travel from the current position to the first waypoint, then to the next one, and so on.
    - Speed: how fast the NPC walks/runs between the waypoints.
- Debug:
    - Current Waypoint: the next waypoint the NPC is walking to.

And the waypoints are divided as follows:

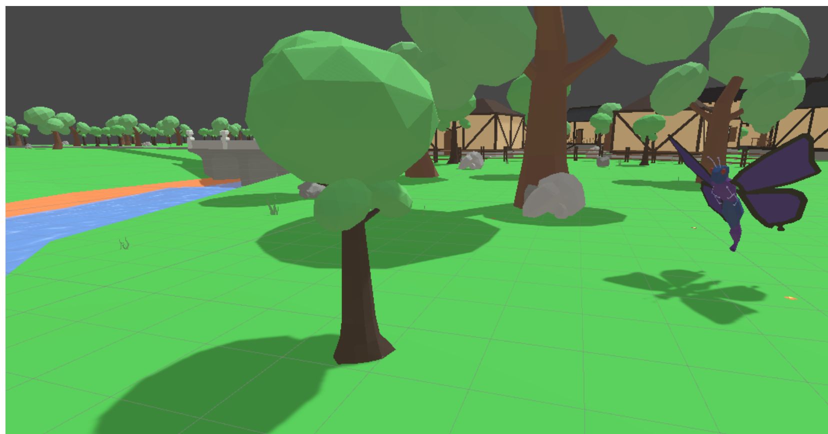
- Attributes:
    - Min Time: minimum time the NPC will stay on that waypoint on idle.
    - Max Time: maximum time the NPC can stay on idle in that waypoint. 0 means that the NPC won't stop on that waypoint.
- References:
    - Point In World: Transform with the position of that waypoint.

One important thing to consider is the way the points are saved. Using empty GameObjects as points will let us place and move the waypoints with more precision and visual feedback while using a small amount of resources, making it more easy to update and expand the routes. Those points could be defined with world elements like floors, doors or even buildings, or moving elements like other NPCs, following the Liskov substitution principle and the dependency inversion principle (SOLID).

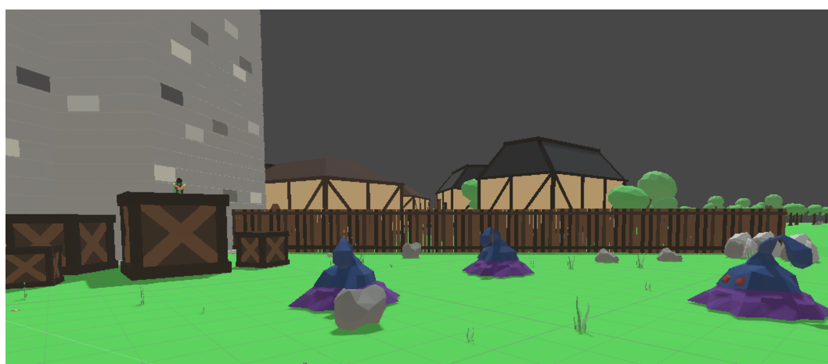
3.4. Enemies

Most quests are focused on killing enemies. There are three enemies implemented in The Final Spell, each of them with their own characteristics: wild pig, butterfly and slime.

Wild pigs on the cage



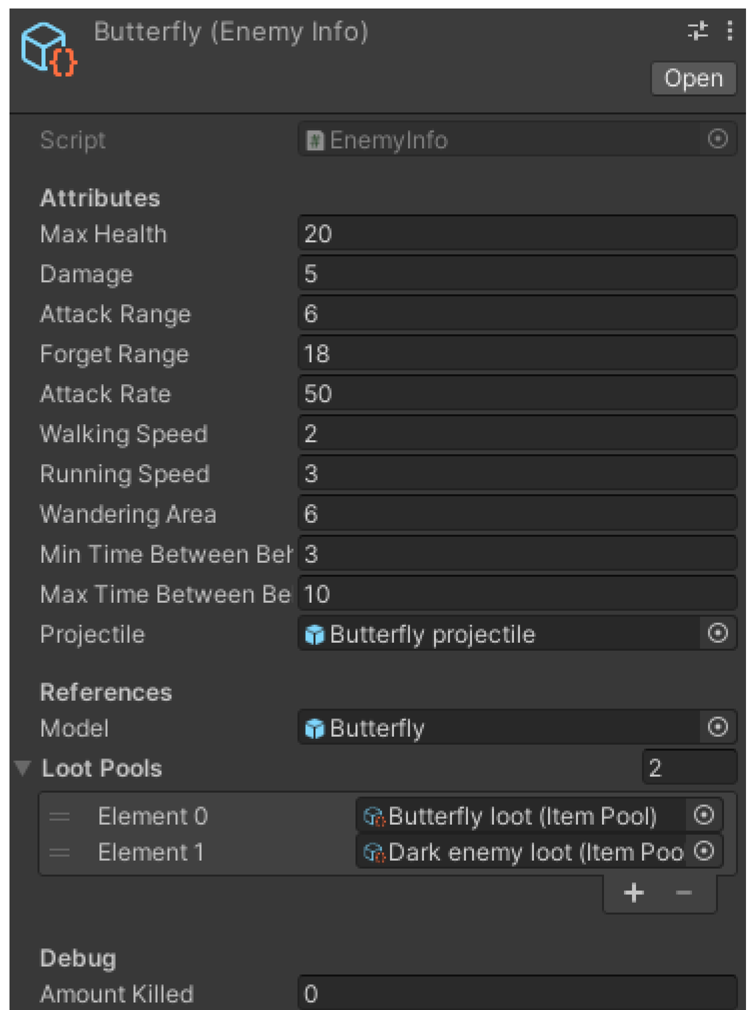
Butterfly on the plains



Slimes on the forbidden nook

3.4.1. EnemyInfo

Following the clean code practices, which have been key in the development of The Final Spell, enemy data is stored in scriptable objects, making it easy to update existing enemies and to add new ones.



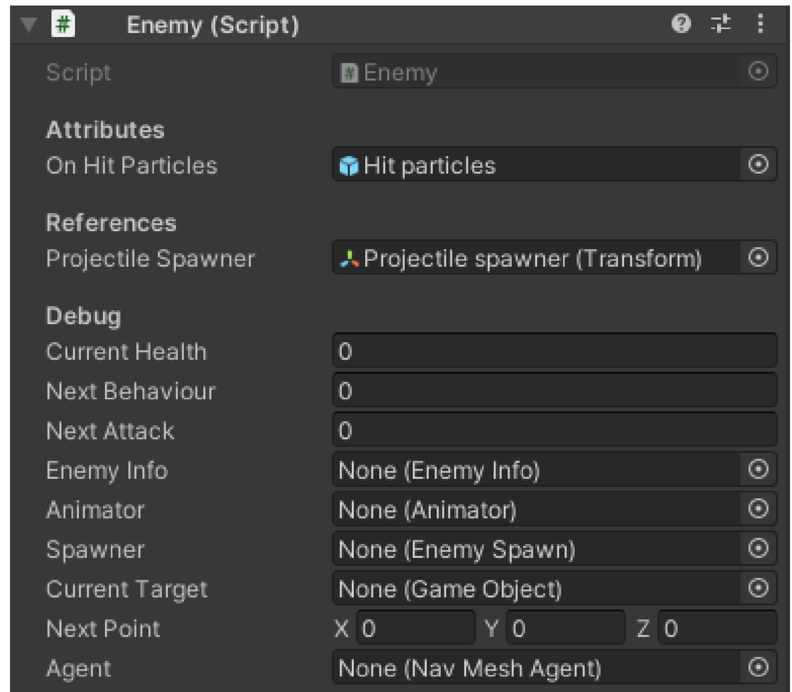
That enemy data is directly used by the Enemy Component and the EnemySpawn component, which have direct references to this object in order to make it accessible and easy for those scripts to use the information stored in the scriptable objects.

The scriptable object displayed on the right is the one in charge of the butterfly data, storing its stats, models used and items pools.

The EnemyInfo component is divided as follows:

- Attributes:
    - Max Health: maximum health the enemy can have.
    - Damage: amount of damage dealt on every attack of the enemy.
    - Attack Range: range the enemy has to attack.
    - Forget Range: range at which the enemy loses track of its objective.
    - Attack Rate: number of attacks per minute.
    - Walking Speed: default speed at which the enemy moves when patrolling.
    - Running Speed: speed at which the enemy moves when chasing an objective.
    - Wandering Area: area in which the enemy patrols.
    - Min Time Between Behaviours and Max Time Between Behaviours: used for choosing the time for the next behaviour.
    - Projectile: model of the projectile the enemy launches when attacking. Null for melee attack.
- References:
    - Model: model of the enemy.
    - Loot Pools: pools of the items the enemy spawns when killed.
- Debug:
    - Amount Killed: number of enemies of this type killed. Used for quests and story events.

3.4.2. Enemy component



The enemy component manages the behaviours of the enemy and triggers possible attacks to an objective.

One important thing to consider is the use of a Selector when the walking behaviour is triggered: there's a possibility that the enemy can't find a good spot to walk to, as there's a cap on the amount of tries for finding that spot, to avoid infinite loops. To avoid that, the Selector runs a check, if it finds a spot to walk to, the walking behaviour is triggered. Else, the idle behaviour is triggered.

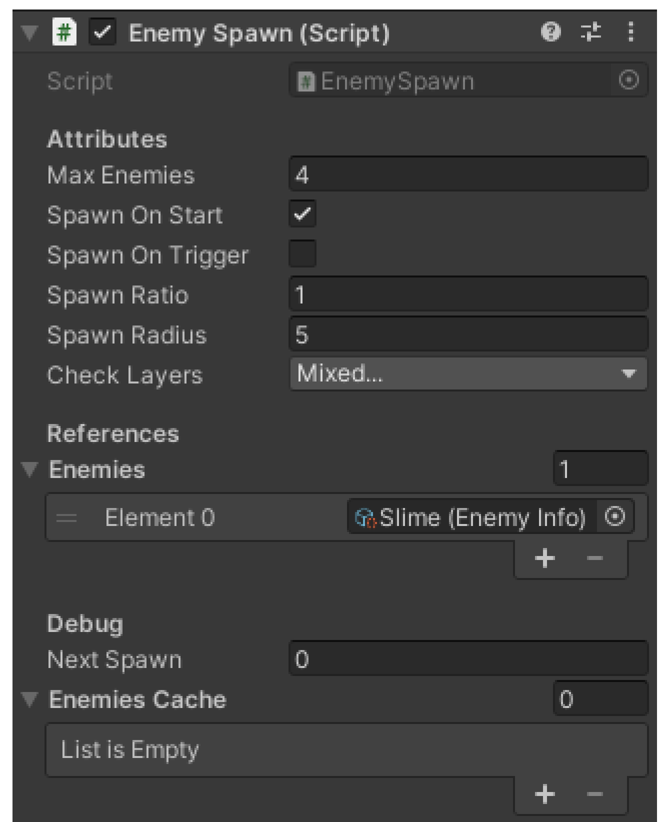
The inspector has the next variables:

- Attributes:
    - On Hit Particles: particles that spawn when the enemy gets hit.
- References:
    - Projectile Spawner: point at which the projectiles appear. Can be null if the enemy has melee attacks.
- Debug:
    - Current Health: the health the enemy has.
    - Next Behaviour: time for the next behaviour to trigger.
    - Next Attack: time for the next attack to trigger (if on fight).
    - Enemy Info: EnemyInfo scriptable object attached to the enemy from which all the data is loaded.
    - Animator: enemy Animator component.
    - Spawner: EnemySpawn component from the spawner.
    - Current Target: current objective of the enemy.
    - Next Point: next random generated point for the enemy patrolling state.
    - Agent: NavMeshAgent component in the pathfinding.

While the EnemyInfo handles all the basic information of the enemy, as it name says, the EnemyComponent is focused on the functioning, that's why the debug area is more important than the attributes and references area, which only store the information that can't be stored on the scriptable object, due to the objects only being accessible on runtime.

3.4.3. EnemySpawn component

This component is in charge of spawning the enemies. It needs to be added to a GameObject, which positions the spawner in the world. When an enemy spawns, it chooses a position between the GameObject position and the radius of spawn. If the spawner component can't find a position for the enemy to spawn, the next spawn counter resets to 60 seconds.



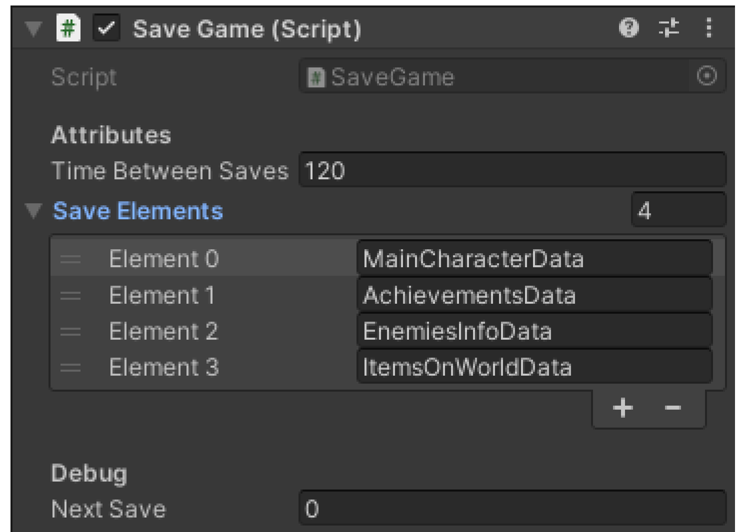
As the rest of the components, the inspector follows the clean code structure:

- Attributes:
    - Max Enemies: maximum number of enemies there can be spawned at once.
    - Spawn On Start: if the enemies only spawn when the spawner is initialized. Used for quests.
    - Spawn On Trigger: if the enemies are only spawned by an external trigger.
    - Spawn Ratio: number of enemies spawned every minute.
    - Spawn Radius: area in which the enemies can appear.
    - Check Layers: layers the spawner avoids when selecting a position for an enemy spawn.
- References:
    - Enemies: list of enemies that the spawner can spawn (EnemyInfo components).
- Debug:
    - Next Spawn: time until the next enemy will be spawned.
    - Enemies Cache: enemies spawned by the spawner and alive. Used for keeping the max enemies spawned. When the player kills an enemy from this spawner, it updates this list and starts the next enemy spawning.

One interesting feature of the EnemySpawn component is that it can spawn any type of enemies. This could be used for specific areas where different enemies coexist, enriching the environment and making the world more interesting.

3.5. Data persistence

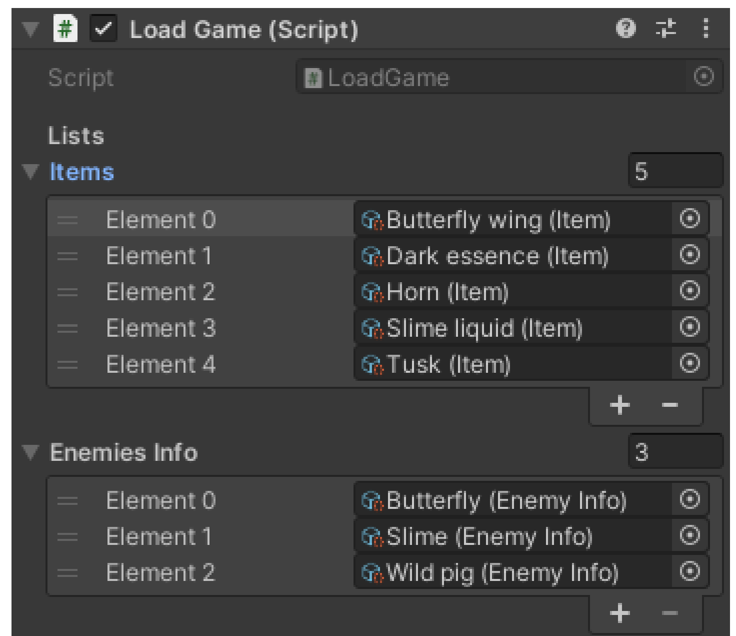
The Final Spell has a save system implemented to keep data between play sessions. This save system is divided into data saving and data loading.



Data saving occurs every specified amount of seconds if the player is not fighting (checked using Main Character component). When the conditions are met, the Save Game component takes all elements that need to be saved (main character, achievements, enemies…), creates the script components with those names (that inherit from SaveElement and can be serialized), and saves every element into a list that will be encrypted to be loaded in another session

The Save Game component inspector follows the standard guidelines of this project:

- Attributes:
    - Time Between Saves: seconds between every save.
    - Save Elements: array of strings with the names of the components that need to be instantiated and serialized.
- Debug:
    - Next Save: current time needed until the next save.



On the other hand, data loading happens only when the player selects a slot in the main menu. The Load Game component takes the file with the session's name, decrypts the data converting it into a SaveElement list, and runs the Load method for every element of the list. The component is filled with useful data used when loading the elements: items models, enemies information...

One example is the ItemsOnWorldData element, which would be part of the decrypted list, the Load method would take every element inside the ItemsOnWorld list (a list of SaveElements of type ItemOnWorld) and call again the Load method, which spawns the item in the saved position, using the Items list in the Load Game component to get the model and characteristics.

The way data is encrypted and stored into a file inside the player's computer is easy yet powerful:

```
string savedGamesPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + "\\The Final Spell\\Saves\\";
if (Menu.GameName == "Null") Menu.GameName = string.Format("Game ({0})", DateTime.Now).Replace("/", "-").Replace(":", ".");

// Create path if doesn't exist
if (!Directory.Exists(savedGamesPath)) Directory.CreateDirectory(savedGamesPath);

// Create game data object
GameData gameData = new GameData();

// Save elements
foreach (string element in saveElements)
{
    gameData.SaveElements.Add(((SaveElement)Activator.CreateInstance(Type.GetType(element))).Save());
}

// Create file
BinaryFormatter bf = new BinaryFormatter();
FileStream file = File.Create(string.Format("{0}{1}.save", savedGamesPath, Menu.GameName));
bf.Serialize(file, gameData);
file.Close();

nextSave = timeBetweenSaves;
StartCoroutine(saveTimer());
```

First of all, the game saves path is generated using the player's documents folder and a custom path for the game (The Final Spell/Saves). If the path doesn't exist, that means that this is the first time the game is getting saved on this computer (probably the start of the first game, or the player could have removed the folder while playing). In this case, the directory is initialized.

The second step is creating an instance of the GameData element, a serializable script that stores a list of SaveElements. Once the GameData script is initialized, the SaveElements list is filled with SaveElement data:

- Using the element's name, an instance of the element is created.
- The Save method is called to the instance created, which returns a SaveElement element with data stored (position, current state, name…).
- The SaveElement element is added to the SaveElements list of the GameData instance.

The third step consists in creating the file, using a binary formatter. A FileStream instance is created, initializing an empty file inside the game saves path and using the name of the session (with .save extension). With that file, the binary formatter Serialize method is called, which adds the GameData element serialized to the specified file.

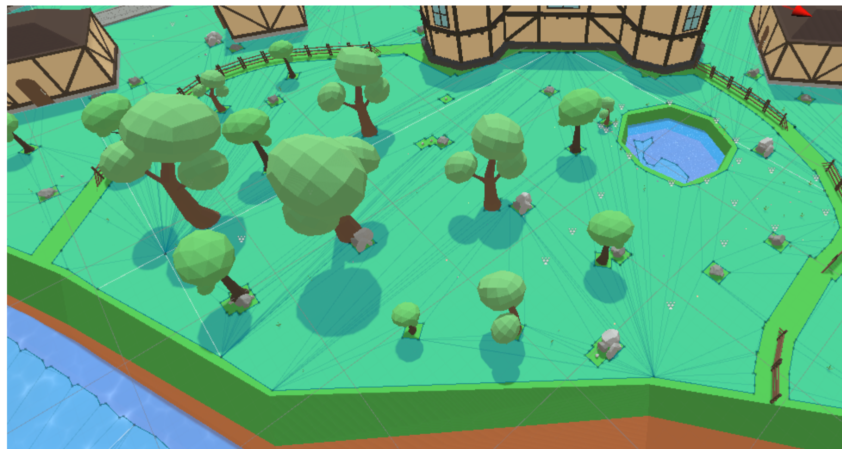
Finally, the timer resets and a new coroutine is started, which will save the game data again after the specified time.

3.6. Other systems

There are several small systems in the game that supplement the bigger ones, making the game longer and more interesting. Every system has been added having in mind the Clean Code principles, which means that adding or removing a whole system is doable.

3.6.1. Pathfinding

For walking and chasing the objectives, the enemies use Unity's built-in pathfinding system. There are three main areas where the enemies spawn and where they must stay: the plains, the cage and the forbidden nook. The navigation mesh is implemented in a way that the enemies can't leave those three areas, but enemies can be spawned out of these areas for specific quests.



Plains navigation mesh, limited by the fences

When an enemy has to move to a specific point (target's position or a random generated position), the navigation agent component from the enemy is started with the speed required (walking or chasing speed, depending on the situation) and with the position selected as the destination.

One problem with the navigation limits is that enemies can be chasing forever their target if it goes through an exit. For example, when the player leaves the cage (where wild pigs spawn) and stays at the door, the wild pigs won't be far enough to reset their behaviour or close enough to attack, so they'd be stuck chasing the player. To fix this problem, the exits are covered by a collider that only the enemies can't overpass. While chasing, if they get too close to those colliders, they reset their behaviour, fixing the problem.

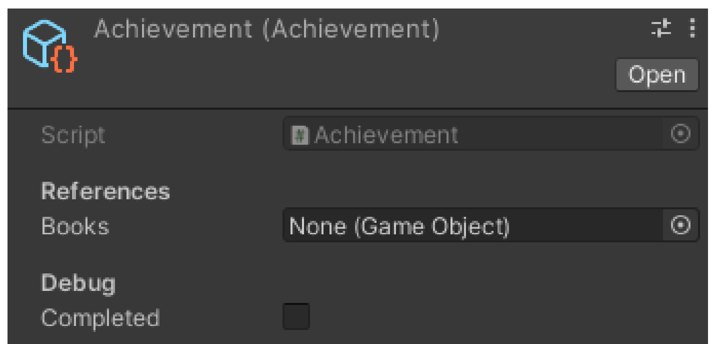
3.6.2. Achievements

One of the small features that has been implemented in The Final Spell and is part of every other RPG game in the industry is an achievement system. Doing non-world changing but funny actions (for example: pet several times a friendly NPC), gives the player a small reward, consisting of a book with lore about the game that can be read in the library.

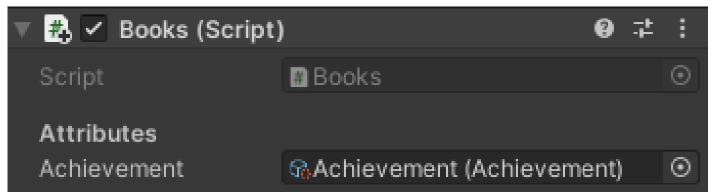
Achievements follow the SOLID principles, making it easy to add new achievements. The triggers are located in different scripts, in charge of the counters (example: friendly NPC that have been petted) or other activators; but two main scripts shape the base of the achievements system: the Achievement (ScriptableObject) and the Books script.

Achievement ScriptableObject inspector:

- References:
    - Books: GameObject that will be enabled when the achievement is completed.
- Debug:
    - Completed: if the achievement has been completed.



On the other hand, the Books script initiates the reference to the achievement selected, and once the achievement is completed (and the GameObject with the Books script enabled) the player can click the books to read a small piece of story of The FInal Spell.



3.6.3. Items

The way items work is divided into three steps:

- An item is a ScriptableObject that can be spawned in the world using the Model reference.
- Every item is part of an item pool. An item pool is a list of items with the same theme that can be used in different systems of the game to select what types of items the system requires. For example, a Butterfly enemy will have two item pools: the Butterfly loot (with butterfly wings) and the dark enemy loot (with dark essence).
- Once the item is spawned, the MainCharacterGatherer component will be in charge of picking up the item when the player is within an area.

Items spawned after killing several butterflies

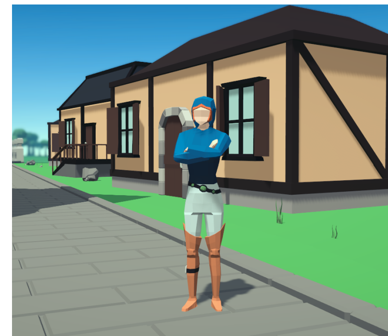
4. Art

The Final Spell has used two styles of art: a 3D low-poly style for the world and characters, and a 2D style with lineart for the user interface. The art design and modeling has been done by Cristian Cantos using Blender and Clip Studio Paint.

4.1. Characters

Characters have been modeled using the same base, so animations are easily recycled along the project for the different NPCs. There are three main types of models: default, kid and old human bodies. The project has more than 20 different bodies implemented.

The main character model also has several animations related to the spells casting, and some other specific animations for character movement, like crouching or running.
Unity's animator controller, explained at the section NPCAnimations component (3.3.2) has been used for management of the different animations.



4.2. Enemies

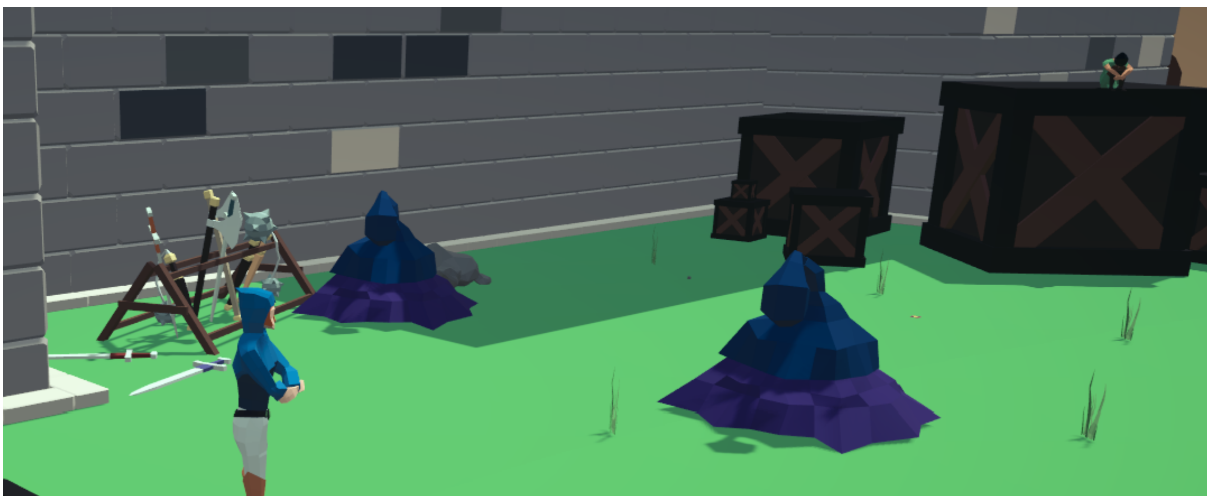
There are three enemies modeled for the demo: wild pig, butterfly and slime. All of them have the same types of animations, but adapted to the different body sizes and shapes: idle, wandering, running, attacking and dying. They're modeled following the same style for the color palette: purple and blue colors that represent the darkness they have inside.

4.3. Buildings

Several buildings have been designed and implemented on the demo world area, including four big buildings and two small houses.

4.4. Environment

To complete the world, different small elements have been modeled and placed around the world, includying flora (grass, trees, flowers…) and other visual elements like signs, boxes, weapons stands...

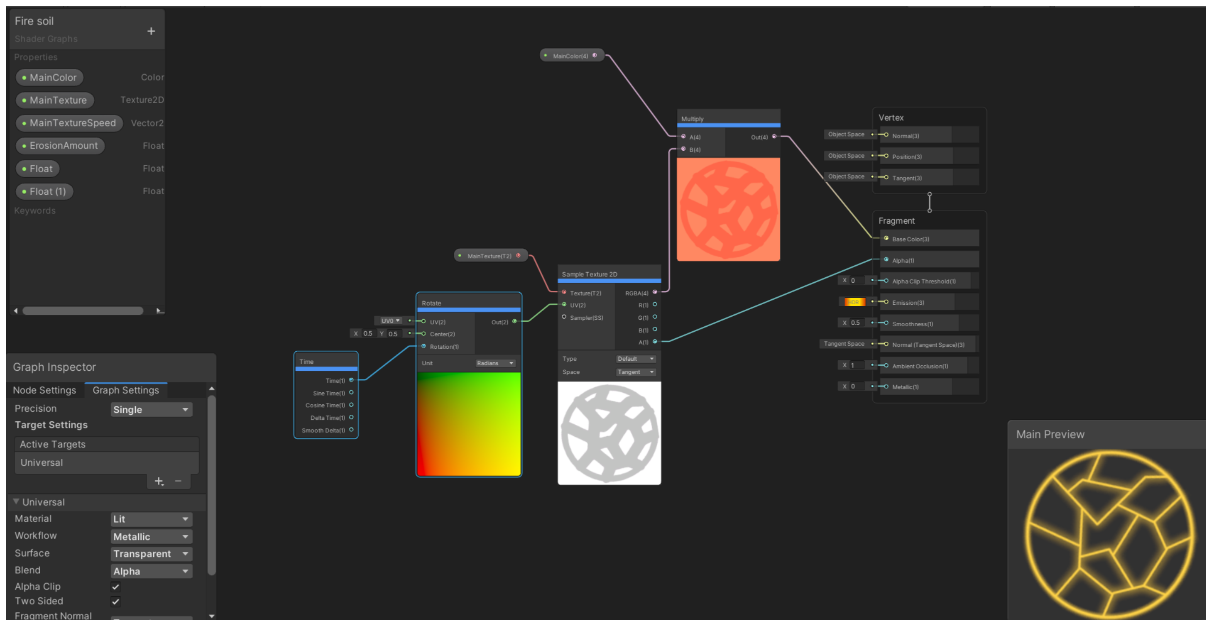


4.5. User Interface

UI elements have a medieval-like style to fit correctly in the world. The main colors used have been yellowish white and woodish brown.
There are several UI windows implemented: shop, dialogues, spells book, map, inventory, health bar, buttons bar and the items.

4.6. Shaders

One last important tool used for the creation of The Final Spells is the shaders Unity's system. It has been key for the development of the different spells, that relies on the shaders. It has also been used for other small elements in the game, like the blocking portals placed on the bridges.



5. Music

To achieve a fully RPG experience with The Final Spell, medieval-like music is played during the game.

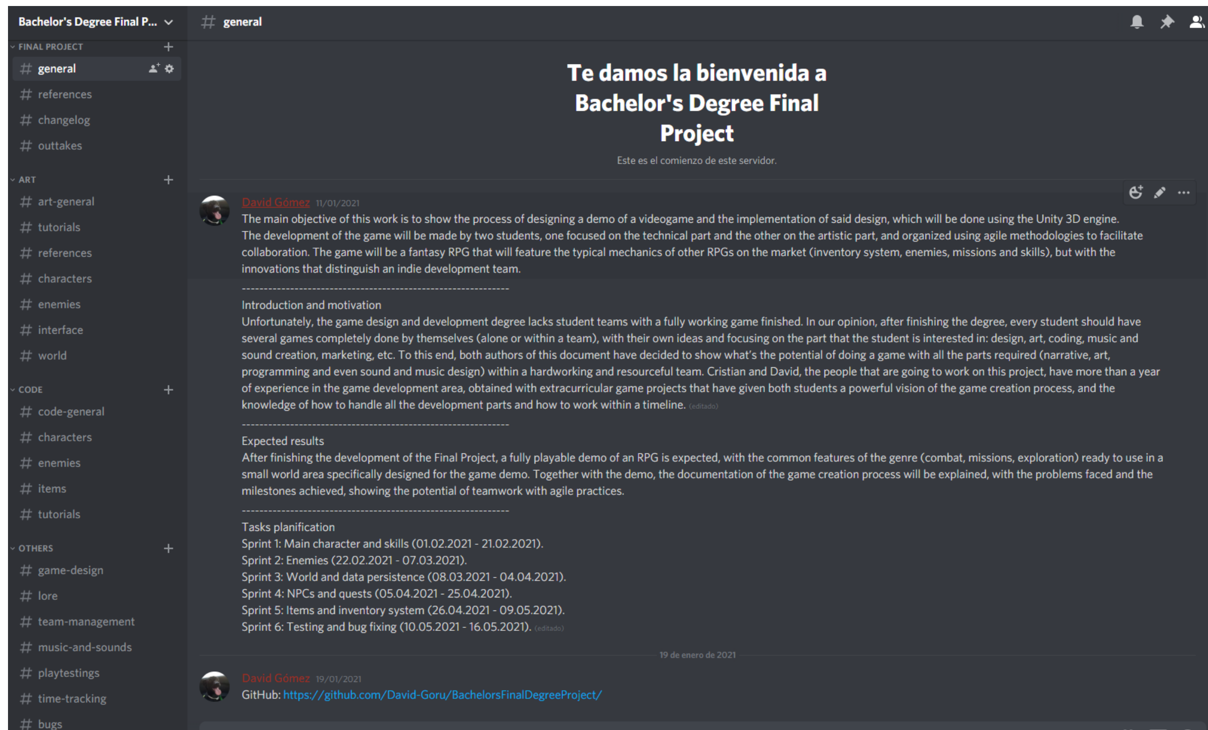
With the help of the musician Manel Espinosa, two songs have been produced: one default theme that loops throughout the game, which is relaxing and suitable for the medieval age, and another one that plays when the player is fighting. That last song, named “The Great Fire”, motivates the player to fight with an intense and epic tone.

The music is based on other medieval RPG games like Valheim and World Of Warcraft, from which the Azeroth ambiental music has heavily influenced the music on The Final Spell.

Music can be downloaded from the GitHub repository (section 10).

6. Team organization

The Final Spell has been developed with the collaboration of another student focused in art: Cristian Cantos. It's development has lasted four months, and Discord has been the main tool used for communication and design.



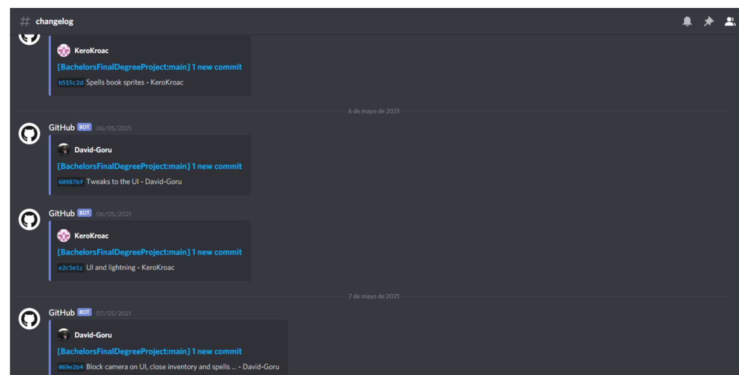
Discord channel for the development of this project

The Discord channel used for the project is divided into four sections:

- Final Project: the main section with text channels for references, outtakes and the changelog.
- Art: used for everything related to the project's art: tutorials, art references, characters, enemies, interface and world art.
- Code: stores design information for code-related work, like the characters, enemies and items implementation.
- Others: this section is focused in the game organization and design, with the design, lore, team management and time tracking channels. There's also a channel for the music, and one last channel for the bugs (art and code errors).

GitHub has been used as the tool for sharing the project within the team.

Every time a change was pushed to the git repository, the GitHub bot sent a notification to the changelog channel on Discord, specifying the



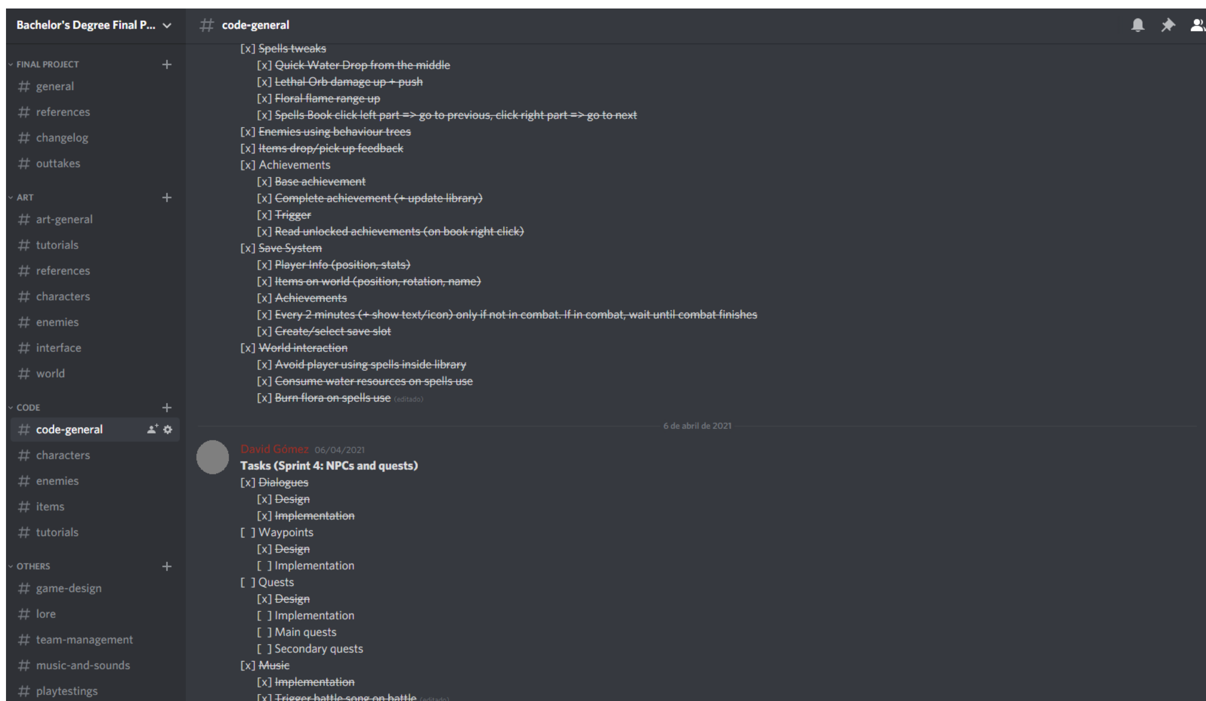
person who did the commit and the changes done.

6.1 Agile methodology

The project development has been divided into sprints, following the agile methodology:

- Sprint 1: Main character and skills (01.02.2021 - 21.02.2021).
- Sprint 2: Enemies (22.02.2021 - 07.03.2021).
- Sprint 3: World and data persistence (08.03.2021 - 04.04.2021).
- Sprint 4: NPCs and quests (05.04.2021 - 25.04.2021).
- Sprint 5: Items and inventory system (26.04.2021 - 09.05.2021).
- Sprint 6: Testing and bug fixing (10.05.2021 - 16.05.2021).

Every new sprint, a meeting has been done where the two students shared their ideas, their objectives and the possible problems. After the meeting, the Discord channels art-general and code-general were used to share the tasks for every student in a custom and simple Kanban-like environment, using the formatting tools that Discord offers.



Example of sprint 3 and 4 tasks

6.2. Playtestings

Although the game has been constantly tested by the two students developing the project, different people have also helped with two playtestings: the first one was done early on the development, when only the main character and enemies were implemented, and the second one was done once the project was finished. Together with the game build, the players who tested the game had a survey to send their ideas and problems.

The objective for the first playtesting was to get feedback about the interaction with enemies, the character spells and the movement.

The people who tested the game were asked with four questions, and these were their answers:

What problems did you have?
- Quick water drop gets stuck in second animation / Spells stop working after a while
- Camera is hard to control / Camera's sensibility is too high
- Floral flame weird final state when finishing the spell(editado)

What things would you change?
- Controls having buttons icons
- Feedback when dropping/picking up loot
- More floral flame range
- Avoid using the shift for spells
- Butterfly shots going straight forward instead of having a target
- Spells book going to next page when clicking half right area/previous page when clicking half left area
- Change the font used on the spells book to avoid 'cl' becoming 'd'.
- Jump action
- More controls for controlling the camera

What didn't you like?
- Quick water drop needs feedback displaying where's going to fall
- Close-range spells don't have much power, so cost/efficacy relation is unbalanced
- Camera rotation is confusing
- Models have wrong shades due to the lights

What did you like?
- Visual aspect of the game
- Models
- Spells' VFX
- Animations
- Fluent movement + camera
- Every spell has a unique way of use
- Picking up items
- Spells book use
- Mix between existing animals (purple pigs with deer horns)

With those answers, the team made a meeting to decide which suggestions to accept and implement. We fixed all the errors, balanced the spells and implemented some suggestions like avoiding the shift for the spells - which was changed with the use of the space -, changing the fonts of the spells, and changing the current page of the spell when clicking the left or right part of the UI.

The objective for the second playtesting was to review the implementation of the main character's interaction with the world: exploration, NPCs, achievements, quests and buildings.

For this playtesting, this was the result of the questions asked:

What problems did you have? Should something work in another way?
- When opening the inventory then pressing right click, the character gets stuck in attack position
- Some colliders make it difficult to go through specific areas
- The character runs without animations when pressing fast the shift button
- Can't click "close game" button in the final screen
- Camera controls are chaotic
- Music appears to stop because of the silence in the loop
- Character can go through the castle stairs

What things would you change?
- Change box colliders to capsule colliders to improve the movement
- Camera not going through objects
- Signs should be brighter
- The player could decrease or raise the volume of the music
- Know the spells from the beginning
- Add a mini map
- Add a dodge mechanic
- Be able to pass dialogues with enter or space button

What didn't you like?
- Controls are confusing
- Can't redirect the water attack
- Not enough places where the character can crouch
- The change of the music
- The basic fight system could be more flexible
- Don't know how many pages are in the spells book
- Characters don't repeat what to do in the quests

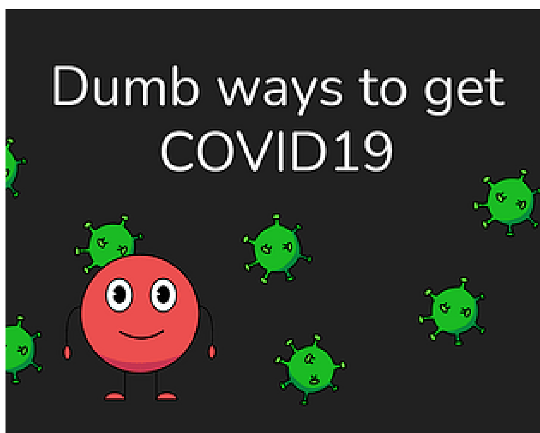
What did you like?
- Visual aspect
- Can change the camera angle
- Characters and enemies animations
- Missions
- Models
- Music

- Merchant interface
- Can pet dogs
- Hidden achievements
- Water shader
- Music fade in/out when starting a fight
- Main character crossed arms animation when being idle for too long

Due to being this too close for the end of the development of the demo, the big mechanics suggestions (like dodge mechanic or changing the way spells are displayed) can't be accepted, although those ideas can be interesting if the development of The Final Spell goes beyond the demo. On the other hand, the team fixed all the bugs reported.

6.3. About us

The two students that have developed this project have been working together for a couple of years, including several gamejams, degree projects like Dumb Ways To Get COVID-19[16] and Vivaio[17], a game developed between 2019 and 2020 during our free time.



Dumb Ways To Get Covid-19

Vivaio

7. Results

After four months of development, and although the idea was ambitious, everything initially designed for the game demo has been implemented: game world, player character, enemies, NPCs, dialogues, quests, items, inventory, shop, achievements, music...

A fully playable demo can be found on the Github repository (that can be accessed on the chapter 10). The demo has four main story quests, one secondary quest, two achievements, three enemies, the merchant, several NPCs, four big buildings with the interior of the library modeled, and a small but complete space for exploration with plains, a maze, a cage, the forbidden nook and the training area.

The sprints designed for the agile methodology have been completed on time with every task planned done.

8. Conclusions

Thanks to the agile methodologies, the previous design of the game and the experience of the team, the project has been completed without setbacks. It has been an interesting project where the two students have had enough freedom to design and implement the most attractive features of an RPG game.

The role of the supervisor, Diego Díaz, has been close to a real project manager role, where fortnightly meetings with the team (the two students) have helped direct the game to what is now, within a calm, professional and pleasant environment. The meetings have helped to show the process of the game, check for possible bottlenecks and plan the following sprints.

In conclusion, the steadiness of the work flow of both students supported by agile methodologies and the correct design of the game has evolved into a finished 3D RPG game demo, with several interesting and fully implemented features.

9. Bibliography

1. WikidPad (https://github.com/WikidPad/WikidPad)
2. Characteristics of RPGs in the game industry (https://en.wikipedia.org/wiki/Role-playing_video_game#Characteristics)
3. Deuteragonist (https://en.wikipedia.org/wiki/Deuteragonist)
4. Dijkstra phrase (http://plasmasturm.org/log/linesspent/)
5. Freya Holmér (https://twitter.com/FreyaHolmer)
6. Sebastian Lague (https://twitter.com/sebastianlague)
7. Liam Sorta (https://twitter.com/LiamSorta)
8. SOLID principles (https://en.wikipedia.org/wiki/SOLID)
9. DevX Blog - The Solid Principles by AbdulMujeeb Aliu (https://devx.blog/solid-ood/)
10. Scriptable objects (https://docs.unity3d.com/Manual/class-ScriptableObject.html)
11. RPG games (https://www.gamesradar.com/best-rpg-games/)
12. FixedUpdate vs Update (https://stackoverflow.com/questions/56954073/)
13. Cinemachine Free Loock Camera (https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineFreeLook.html)
14. MainCharacterSpell script (https://github.com/David-Goru/BachelorsFinalDegreeProject/blob/main/Game%20Engine/Assets/Code/Main%20character/Spells/Scripts/MainCharacterSpells.cs)
15. Input Manager (https://docs.unity3d.com/Manual/class-InputManager.html)
16. Dumb Ways To Get Covid-19 (https://wedogames-studio.itch.io/dumb-ways-to-get-covid-19)
17. Vivaio (https://vivaio.itch.io/game)

10. GitHub repository

A playable demo, a gameplay, the music and the source code of the project can be found at https://github.com/David-Goru/BachelorsFinalDegreeProject.