Universitat de Lleida

Master's Final Thesis

ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA
INSPIRING THE FUTURE

Student: Sergio Salcedo Heredia.

Degree: Master 's Degree in Informatics Engineering

Master's Thesis Title: Development of an action-adventure game

Director/a: Francesc Sebé Feixas.

Presentation

Mes: October.

Any: 2020.

Summary

The purpose of this project has been to make a video game capable of entertaining the players while taking back to the essence of the old 'The Legend of Zelda' games.
This was a project done with Unity and implemented in `C#`. It has been implemented in order to work in computers with the chance of exporting them to android platforms.
For this project it was decided to not use the Unity tools regarding colliders in order to optimize the performance of the project and evade, as long as possible, the use of the Update method and the MonoBehaviour objects for the same reasons.
This document explains the different methodologies used, the different decisions taken and the problems faced during the whole process. It is the final deliverable of the 'Treball de Final de Master' for the Master degree in informatics in the University of Lleida.

Acknowledgments

First of all, I would like to express my most sincere gratitude towards my tutor Francesc Sebé Feixas for accepting being part of the project and guiding me towards the final presentation.

Also I would like to thank a friend that helped and cheered for me the whole project and has been my motivation to keep doing it until the end, if you read this, thanks you.

And finally, but not less important, thanks to my family and Adrian Aguilar Cervera for being there for me and supporting me.

Contents

List of Figures

List of Tables

1 Introduction

In this section we will explain the objectives of our project, the objectives of the game and also the changes in the ideas stated at the beginning.

1.1 Objectives

The objective of this project is to develop a game in 2 dimensions using Unity's framework.

The main idea behind this project is to have a game that makes the player remember the classic 'The legend of Zelda' games. For this project, the user is going to control a character that will have to clear a labyrinth with different enemies to fight against the boss of the labyrinth.

It's important to design game that poses a challenge to the player and not an easy one that can be played with no difficulty. The point of this kind of games is not about high mechanical skills but the need of thinking in different ways of opening a door, defeating an enemy, etc

The real idea for the project is a big game where Erox, our playable character, needs to clear different dungeons and take different powers (each time Erox kills a boss, he obtains its power or attack pattern). But since this idea requires a lot of time to complete, it will be reduced to the first dungeon and the first boss battle.

The game has been thought to be played in a computer with a keyboard or a controller. It's a 2D game with a 4 way movement implementing different mechanics (weapons, powers). Each of these mechanics will have a different use or objective to surpass a certain part of the game. Once you unlock weapons or powers you will be able to use them at any moment.
The whole game will be based on passing different labyrinths with your ingenuity.

1.2 Modifications over the initial idea

As stated in the previous section, the idea behind the project was so big that it couldn't be done by one person in the time available for the TFM.

One of the ideas that changed doing the development of the project was the way that our character unlocks new powers. Firstly it was stated that the character would unlock them by defeating different bosses but that's something not possible in the current project since we only have one boss. It would make no sense at all, so now the character has to unlock them by collecting different scrolls (Fire, rock, water and wind). Due to time limitations only the Fire scroll has been implemented.

Finally, some ideas have changed because of technical issues or implementation decisions. They will be explained in the Development section.

1.3 Budget

In this section we will try to simulate the cost of the development of the whole project, being fictional since it has not been a professional project.

1.3.1 Cost of the workers

Since this time only one worker made the whole work of analysis, management and implementation only 1 person has been taken into account. In Table 1 we can see the costs of this worker with an average salary per hour. For the designers, since we don't know the cost of the effort that they will take, an estimation has been done.

Kind of worker	**Hours**	**Cost per hour**	**Total**
Developer	256	8	2048
Designer	100	10	1000

Table 1: Cost of the workers

1.3.2 Licenses

Another cost to take into account is the one of the licenses of the different tools used for the project. The cost of the development tool, Unity, is free in the case of using it for personal uses an not take any benefit from it. If the benefit is lower than 200 thousand dollars, the license would cost 35 monthly dollars. If the benefits are higher than 200 thousand dollars, the license would cost 125 monthly dollars. In our case, since for now we won't take any profit, the cost of 0 dollars will be taken into account. The cost of mostly every software used for the project is 0, we will count 3/4 months of payment.

Product	**Cost**
Unity	0
Windows 10	135
Overleaf	0
Visual Studio 2019 Community version	0
GitHub - Student license	0
Total	135

Table 2: Software cost

1.3.3 General expenses

Finally, the wear of the hardware has also been taken into account even it's a subjective cost and other things like the studio, light, etc also should be taken into account.
Again, the costs had been calculated to 3/4 months.

Product	**Cost**
Computer	500
Light and electricity	400
Studio	800

Table 3: General expenses

2 State of the art

2.1 Target Audience

This game is focused on people that played the classical "The Legend of Zelda games." The idea is to find the charm of those games and capture it in our game. In order to give a deeper detail to this point, we will make a little analysis of the demographical factors of our target audience:

- Age Group: Between 20 and 50 years
- Gender: Male or female, but since the gaming industry right now is mostly males our users will be mainly males.
- Education Level: At least the mandatory education of the country, due to the age specified.
- Occupation: Mostly people already working.
- Income Level: Since the game is supposed to have a low price, there is no need to specify a certain income level.
- Location: The game is thought to be released in Europe. If it is successful, move it worldwide.
- Technological knowledge: Medium, probably daily use of mobiles or computers.
- Political barriers: None.

2.2 Monetization plan

In this section we describe the different options that we have explored in order to make benefits from our project.

2.2.1 Possible options

- Fixed Price. Putting a price to the whole game, so that in order to play, the users will need to pay. A good idea for this option is to put a free demo and if the user liked the game and wants to play, the payment pop-up will show up.
- Free with ADS. Old and annoying option. You put the game free but pop in-game advertisements so you get money from them. The worse part of this option is that the users are already tired of adds.
- Freemium. Putting the game free but adding some parts that you can only access after paying a little fee.
- Free with micro-transactions. This option makes the whole game free, but being able to pay for some little items, like cosmetics or extra lives.

2.2.2 Final decision

After analyzing the described possibilities, the 'Fixed Price' option has been taken.
This decision was taken after discarding the rest of them. Firstly, games like this one need to have a experience that puts you into the game and cannot depend on buying items so you can clear the game faster or with pretty clothes. Putting ADS on the game would destroy the game experience because you would either have to stop playing to see an advertisement or permanently have a bar showing those ADS, which is pretty annoying. Finally, making the game freemium was one good option but this kind of game relays a lot on the history you put inside, and stopping the history and forcing the user to pay in order to continue playing could annoy the users. This final example can be found in some important games like "Pokemon Sword and Shield" in which the company decided to make some DLCs and the community received them pretty bad.

After thinking about it, we decided to put the initial project as a demo in which the users can play and try the game and if they get interested in the final project it will be released after paying fixed price. In this way the

users can test the different game mechanics without spoiling the history and if they get interested they can unlock the whole game.

2.3 Project Management

In order to control the basic tasks that need to be done, a Kanban tool has been used.
Kanban is a visual system for managing a project as it moves through a process. In Figure 1 we can see an example of the Kanban we have used.

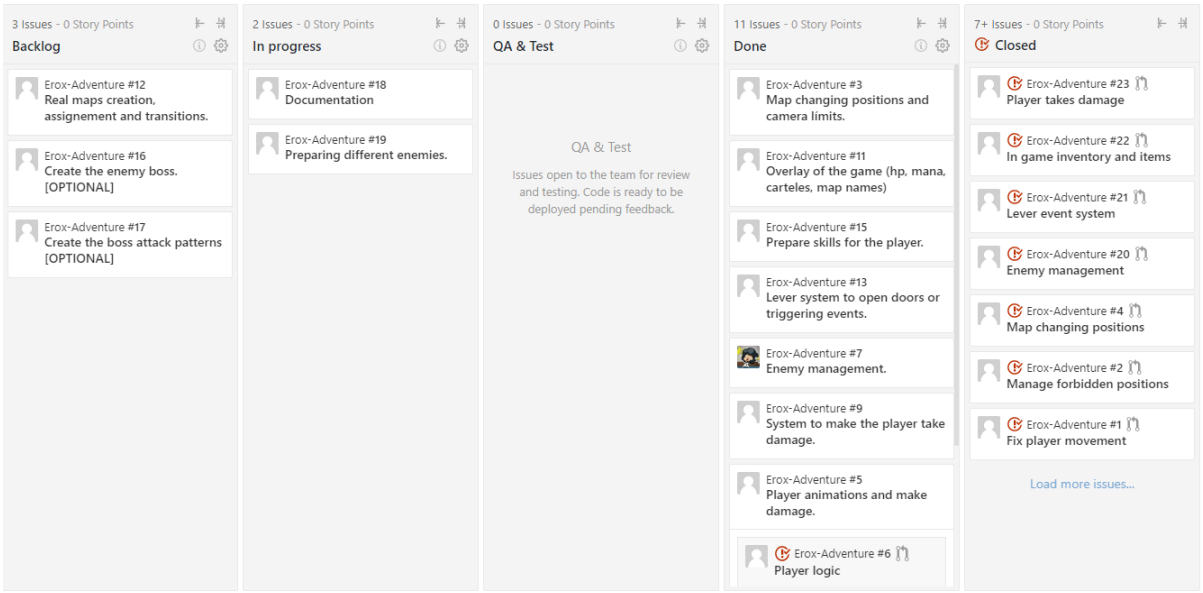

Figure 1: Kanban example

Usually Kanbans are used to visually track the tasks into which the project has been broken down and observe the general state of it. This helps to detect possible bottlenecks in early stages and solve them.
In the case of this concrete project, as there's only one person working there can't be bottlenecks but it has been decided to work with this methodology to make the game step by step and having a global idea of how it is going and how much work is left.

Obviously the initial estimation of tasks and the work done were not perfect and it has been modified through the process. The initial analysis helped to know how to start and organize everything a little bit better.

2.4 Technologies used

In this section we explain the technologies used during the project, from the management to the development.

2.4.1 Development Platform

The development platform used is Unity. That's something that has been decided from the first moment due to our previous experience with it.
Unity is a graphic engine with a simple interface and a lot of features to help the user develop the game. It allows to export the produced game to different platforms such as mobile phones, consoles, computer, etc
It uses `C#` as the main programming language even though it can also use Boo. The reason not not chose Boo is that `C#` was a better known language for the developer.
Unity has also a store that allows you to download and use different assets or widgets to help in your project. It was something to take into account.

Unity HUB has been used in order to manage the installation of Unity and Microsoft Visual Studio. Unity HUB is a tool that lets you download different unity versions and create projects that will be emulated with whichever version of Unity you have chosen. In the next figure we see the interface:

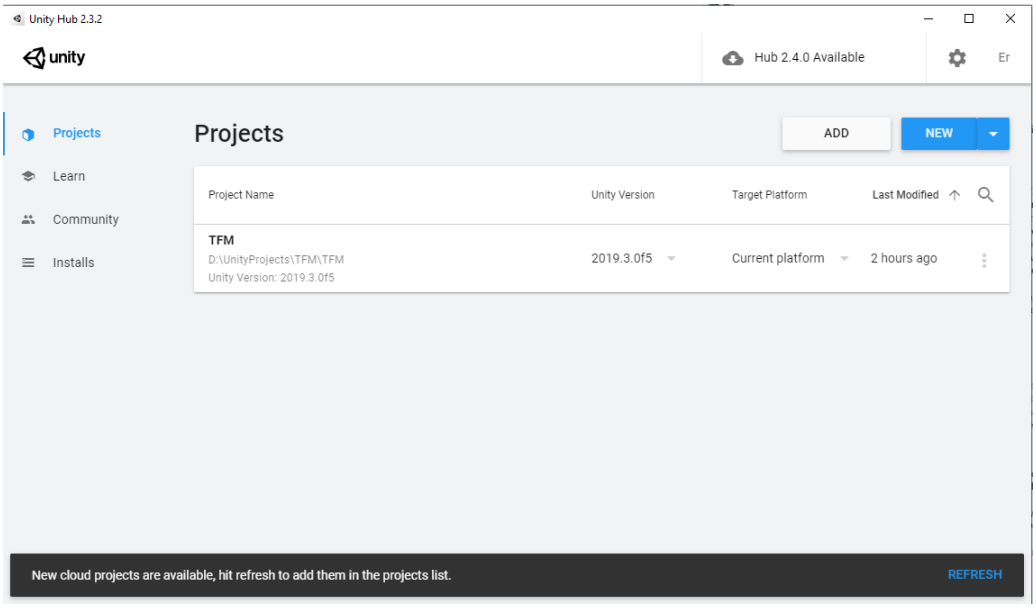

Figure 2: Unity Hub interface

2.4.2 IDE

Microsoft Visual Studio has been used as the IDE in the development. It is the one that Unity provides you as default. It has a good integration for debugging the project and it saves you from external downloads and configurations, that can be annoying. It also has a free option with the Community version.

2.4.3 Version Control

In order to have a control over our code and not being able to lose it easily we decided to use Git and use it over GitHub platform.
This decision was made from the previous bad experience with Unity Collaboratory, the tool that Unity provides. Also, the use of Git was handled with a graphic interface named "GitHub Desktop" that makes all the GIT commands for you in an interactive way. In Figure 3 we can see an example of the tool when changes are detected in our project.

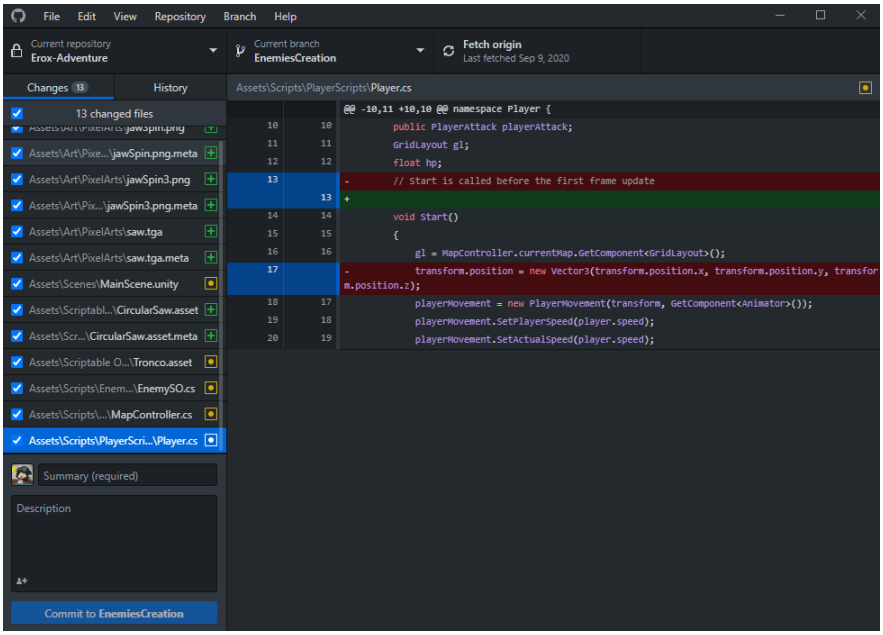

Figure 3: Github Desktop

In order to configure the project in GitHub, there are some settings that you need to make due to the different files that Unity automatically creates that can't be uploaded to the repository. GitHub helps you if you specify that you want to create an Unity project and almost set everything, but adding some more ignores needs to be done.

2.4.4 Project Management

In order to make an easily accessible Kanban we decided to use Zenhub, a Chrome/Github extension that let's the user manage projects which is build for software development teams that use Github.
The project has been split in different tasks that are represented with graphic cards in the Kanban and can be moved through the different states also created in this tool. The way to go was to make one branch for each of the task represented in the Kanban, and close the branch when the task arrives to the "Done" state.

3 Development

In this section the different parts of the game and how they were developed are explained. This part is going to be more technical and specify the different details that were handled during the process.

3.1 Event System

In order to explain how each game element was implemented, we need to explain what an event System is in Unity. A good way to see it clearly is to compare an Event System to the Observer pattern in Java.
In the Figure number 4 we can observe a GameObject with 3 different scripts attached to it. A GameObject is an Object that is placed in a scene of the game and can have different sub components, in this case, 3 scripts. It has been decided to use 3 scripts in order to separate the events depending on their goal. One script is for enemies events, one script is for the key pressed events, and lastly one for the global events of the game. Each of those classes is made like a singleton in order to always access those same events from different objects of our games.

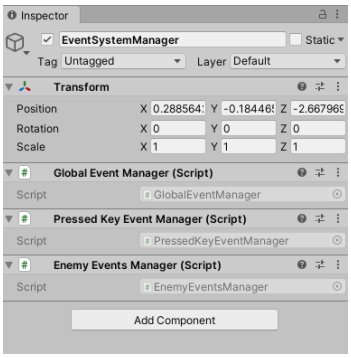

Figure 4: Event System Manager

In each of those scripts there are some Actions defined with a trigger name and a function. Different classes/Objects can subscribe to those declared Actions. When something invokes that function, every object subscribed to this function will trigger the function that has been associated with the event and

execute it's own code.
This system permits the objects to execute some code when a certain event happens and have different behaviours depending of what's needed. For example, in a Player attack, a lever can be activated and an enemy will receive damage.

It is so important that in each class where you instance an event, you also dismiss the instance when the object is destroyed. Here we show an example of how to create an event:

```
public event Action onUpKeyPress;
public void UpKeyPressed()
{
    if (onUpKeyPress != null)
    {
        onUpKeyPress();
    }
}
```

and how to subscribe to an event from an object:

```
PressedKeyEventManager.Instance.onUpKeyPress += MoveUp;
```

being MoveUp the method that will be executed when onUpKeyPress is Invoked.

3.2 Player

In this section the decisions taken for the player will be explained. First of all, the player is represented as a GameObject that has four different components: a transform, an sprite renderer, an animator and one script.
The transform component represents the GameObject itself in the game and stores its position, the rotation and the scale of the object. The sprite renderer component, as the name says, has a sprite (image) associated that will be shown when the object is put into the Scene. The animator component has an Animator Controller. This controller is the one in charge of changing the different configured animations depending on some parameters. In the Figure 6 we can see the different possible States and parameters. Each transition has a condition to be triggered. For example, a boolean parameter to go true in order to change to the next animation.
Finally the script Player.cs receives an Scriptable Object as a parameter. A ScriptableObject is a data container that you can use to save large amounts of data, independent of class instances. The idea behind using Scriptable Objects is to make it easier to add new objects of the same class, in our case new players, without having to duplicate code. This object has the information of the player, such as the health points (hp from now on), the name of the player, the attack damage... Unity let's you create new data containers with just right clicking the Menu once you have defined them as a MenuAsset.
Regarding to the Player script, it uses the Scriptable Object in order to set the values into those that will be used later. It is the one in charge of sub-

scribing to the events of taking damage when the enemies make damage and be aware if our hp is still over 0 or we lost the game. In case the hp goes to 0 or less, an event is invoked in order to show the GameOver screen.

Figure 5: Player ScriptableObject

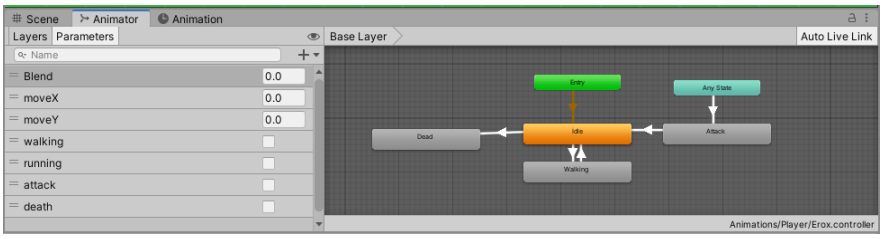

Figure 6: Player Animator Blend

Regarding the player movement and player attack, we decided to use different scripts. This decision was taken trying to optimize the game. Unity helps you a lot and has methods to let the objects work permanently, but they have a high cost. In order to evade this cost we decided to make 2 different scripts that do not extend MonoBehaviour. Making a while loop is more optimum than using the 'Update' method provided by Unity. So the Player.cs creates instances of PlayerMovement class and PlayerAttack class and calls them as a Coroutine that will be running in the background permanently. Those 2 classes are created passing the Scriptable Object information

via constructors.

3.2.1 Player Movement

The Player Movement is implemented to work with the keyboard. A Scriptable Object with the different used inputs has been created. Then we have a GameObject called KeyPressListener that contains an script that receives that scriptable object as a parameter and every time a configured key is pressed or unpressed it calls an event.
The Player Movement class is subscribed to all those events and each time that a movement key is pressed, a boolean representing the direction is put to true, or false if the key is unpressed. Each iteration in the loop a Vector3 is updated depending on the values of the previous booleans, 1 or -1 to the X or Y. In order to always have the same speed the movement is normalized multiplying those values to the speed and Time.deltatime, a fixed value from Time library.
Then the position that we are moving to is checked to see if it leads to a valid position to move and does not have and obstacle or is out of bounds.
This script has the instance of the player and changes the values of the X/Y in order to have the correct movement animation.
Since it has the information about the movement, this script is in charge of invoking the map transport or the item pick up interactions that will be explained later in this document.

3.2.2 Player Attack

The attack script has 3 methods. One is in charge of checking a boolean value to trigger the attack animation. The other method is the Attack method, which is triggered when the Key Attack Press Event is pressed and it sets the boolean to true so the exact next frame triggers the animation. It also calculates the position next to you, having into account the rotation so the new position is the one you are facing to.
It has been decided to do it this way because of the logical implementation of the game. Normally you have a Collider attached to your player that detects if you hit something and triggers a function to decide what to do. In our case, since we decided to not use Colliders or tools that overload the game it was extremely hard to manually calculate a certain area and make damage to all this area, so we calculate the Tile (in the maps section we will explain this part) in front of us, and trigger an event that we are attacking that position. Then the enemies or objects subscribed will receive damage.

Finally, the fire attack method is triggered when the fire attack key is pressed. The only thing this method does is to calculate if your mana is enough to trigger the attack and if you have the needed item to perform that attack. In that case, the player mana decreases and it instantiates the prefab of the bullet. In this case, the fire attack.

The bullet has a custom script so that each time it moves to another Tile (position) it invokes a method to make a certain damage (each type of bullet can have a different damage) in a position, so the enemies can subscribe also to this method and receive damage. Initially it was though to do it in each loop iteration, but this could overload the system so it only get triggered once at each new tile where it passes. This bullet always moves in the direction the player faced when it was invoked. At the start, this interaction was made for the exact position of the bullet. The problem is that if the position is not exactly the same, the enemies won't receive damage. So the most times, there were a difference of 0.01 at the X, so the enemy never took damage. For this, it was decided to work also with Tiles, not with exact positions.

Figure 7: Player Attacking

3.3 Enemies

In this section the different type of enemies and their different behaviours are explained. First of all, as explained for the player, the movement and attack of the enemy has been split from the main enemy script. This is done again in order to optimize the game performance and not overload with MonoBehaviour objects. For the enemies, an script 'Enemy.cs' has been created. It has 2 public parameters, an array of positions and an ScriptableObject that contains the information of the created enemy.
In this case, there are 2 important parameters to take into account: Attack Pattern and Movement Pattern. In games like this, the enemies don't have

an learning AI that improves each time, so it has been decided to implement different movement and attack patterns that, combined, will make the different types of enemies of the game.
If the player stands in the way of a moving enemy, the player will receive damage and make a little knock back behind. The same thing happens when the player hits the enemies. They get a knock back in the opposite direction where they were hit.

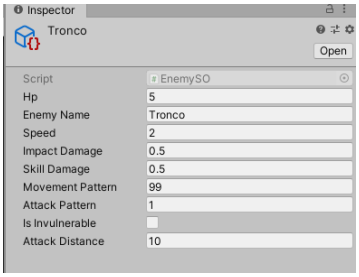

Figure 8: Enemy ScriptableObject

Right now we have 3 different enemies: 'Tronco', 'Circular Saw' and 'The Pig'. We won't enter into details of each one since their different patterns will be explained in the next sections, but one of them is an invulnerable saw that just walks between certain points and hurts the player if it touches the Saw, the other one is something like a turret, that is sleepy until you hit its range and then attacks you by throwing rocks and finally the last one is a pig that charge into the player making a lot of damage if you don't hit him before it hits you.

3.3.1 Enemy movement

When an enemy is created it has a certain ID for the movement pattern that needs to be created. Depending on this number, there are 3 different kinds of movement that can be implemented:

- Move Between points: The enemy has an array of points received through the Unity interface and it keeps moving between these points. The problem here is that if the points are not exactly the same, it never stops and with the animations the anchor point of the object(it is the

point that has the exact position of the object) was being moved, so it never looped correctly. In order to solve this, we tried to do it for tile position, but it never reached the center of the position and the loop was, again, shorter than it should. Finally the animations were fixed and the movement was stick to exact points.

- Chase the player: Simple movement pattern that follows the player wherever it goes, kicking him back each time it reaches the same position.

- Don't move: The enemy stays quiet, it can be useful to towers or static enemies that doesn't need to move.

This different types of movement were decided based on what has been seen in the game 'A link to the past'. But thanks to using that way of making the movement, it's easy to change the type of movement of an enemy, or adding new ones. No code is needed to change the movement of an enemy. We just need to modify the Scriptable Object and all the enemies that share that Scriptable object (that means they are the same kind of enemy) will change its behaviour. That is one of the reasons why Scriptable Objects have been used.

3.3.2 Enemy attack

For the attacks the same idea of the movements has been used. But in this case only 2 patterns have been implemented.

- Enemy Bullets: This pattern combines with another property of the Scriptable Object, that is the 'Attack Distance' and the 'Don't move' pattern. When the player distance between him and the enemy is closer or equals to 'Attack distance', it starts shooting bullets to the position of the player with a cooldown of 2 seconds between bullets. The bullet runs an script that, as the player bullets, invokes an event with each tile position it goes through to make the player aware of that attack.

- Charge: This attack pattern interacts with the 'Chase Player' movement pattern. Once the enemy arrives to the player, it does a huge amount of damage.

Figure 9: Enemies shooting to the player

3.4 Lever and door system

In classic games you always have a point where you have to activate a lever in order to open a door and keep exploring the dungeon. In this section it will be explained how it has been implemented in this project.

3.4.1 Levers

For the Levers, a GameObject with only the Transform, a sprite renderer and an Script components has been created. The lever consists of an ID, a list with the different sprites it can have and finally 2 booleans to see if it's active or not.
The booleans to check if it is active or horizontal are made to update the sprite of the object. Also, the levers are subscribed to the attack event of the player, and if they are in the attack position, the state of the lever is inverted and an event with the ID of the pressed lever is triggered so the doors can update their state. It happens in the same way if the lever is deactivated.

Figure 10: Player hitting a lever

3.4.2 Doors

Regarding the Doors, at the begginning of the project two ideas appeared and, in fact, both of them had been implemented but only one is being used. The first idea consisted in making the doors depending on a fixed amount of levers, for example pressing 3. The second one was to save the IDs of the needed levers to open the door, and throw an event with the ID, so the doors know when to open.
For the first idea, each time a lever was triggered, a counter in the doors was increased or decreased, depending on the event. After thinking about it, it was better to trigger certain doors triggering certain levers, since it can give a better game experience to the user. Finally each door has a list of needed IDs and another with the actual IDs, so each time a lever activates or deactivates, the activated levers changes and when the needed and the activated levers has the same IDs the door opens.
In order to open the doors, there is a event that goes to the MapController that removes or adds a tile to the map of forbidden positions. Further information about the forbidden position will be explained in the Maps sections.

Figure 11: Door

3.5 Maps

In this section the creation of maps and their components is explained. We decided to use Tilemaps in order to implement the maps. A Tilemap component is a system which stores and handles Tile Assets for creating 2D levels. A map object is one that contains a Grid as a subcomponent and it has different childs. Each one of them has 2 components: Tilemap and TilemapRenderer. Each of those childs represents a layer in the final map and are shown one upside the other in the specified order.
Basically, each of the layers is composed of a grid as big as the whole scene. Each square of the grid can be replaced with an specific Sprite that has been

imported to the project. The point of having those tiles is that they let us retrieve the global position our objects are in so we can work with this tile position instead of working with exact positions, that is pretty hard due to the accuracy of the X,Y float values. It also lets us retrieve the name and sprite of the Tile so we can work with this name. This last feature will be used and explained in the portals.

- Ground: Basic layer of the map, it represents the floor and everything else will be placed upside this layer.
- Obstacles: This layer contains the different objects that will block the user path, but just visually.
- UpsideGround: In some cases, there are some objects, like a bridge, that need to be walked by but need to be on top of the ground without losing, for example, the river behind. Once again, this is just a visual layer to represent the map.
- ForbiddenTiles: The most important layer. This layer is hidden and it has a logical use. In this layer every tile that has an sprite will make the player unable to step into it.
- Portals: Represents the positions where the user will teleport to another map if it steps into its position. They are explained with more detail in the next section.

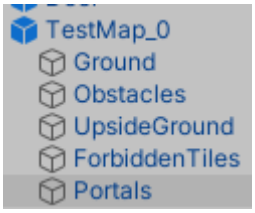

Figure 12: Map Object Prefab

After explaining how a Map GameObject is composed, we will talk about 2 scripts that manage all the work: Map.cs and MapController.cs. The Map script receives 4 parameters from the Unity Interface: the ID of the map (each prefab implemented needs to have a different ID), the position where

the main character will be loaded when stepping into this map, the child with the forbidden positions and the child with the portals. This script loops over all the tiles of both of its childs and saves into the instance of the MapController class the values for the portals and the forbidden values. In order to take this position, the map is needed, since the layout is the one that calculates the cell position of a given position. In this case, we iterate over all X and Y of the childs and save the values that has an sprite printed.
The MapController class is attached to an independent GameObject and has the objective of being aware of all the maps that needs to be charged, all the positions that have some kind of interaction, etc
This class receives 3 attributes from the Unity Interface: The player GameObject, a list with all the prefabs of the maps sorted by their ID (if the ID of a map is 1, it has to be in the index 1 of the list) and finally a list of the transforms of the items that the user will be able to pick out. This class also has 3 static lists: the invalid positions, the portals and the items positions. Those lists are used in the whole game in order to check the position each time the player moves to see if it has to pick an item, if it needs to teleport it or if it can't move in this direction.
The player is needed in order to change its position when a new map is loaded. As said before, the new map that will be loaded has the initial position of the player, so it is needed to change the position.
This script uses different events. An important thing to take into account is that this class has as an static value the instance of the map that is charged and almost every object needs this instance at some point to calculate positions or some other logical calculations. In order not to access it a lot, there is an event that triggers when the map is changed (You step into a portal, an event is triggered in MapController) that changes the map to the one with the ID associated to the event and then it instances another event to notify all the objects of the scene that the map changed and should update their references.
Finally, since it has the invalid positions when a lever is updated, this list needs to add or remove the representation of the door and also the logical invalid position of it, so there are 2 events to add or remove a certain position of the respective list.

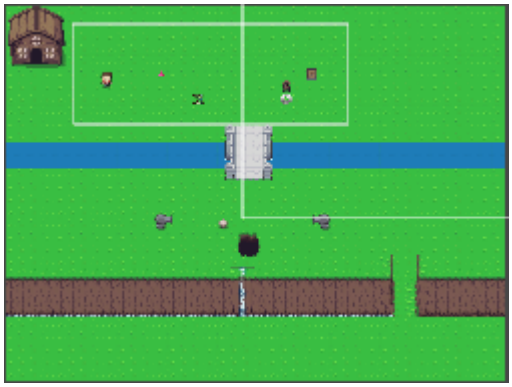

Figure 13: Map Example

3.5.1 Map portals

Regarding the map portals, as said before, there is one list that has all the portal positions saved. They are saved in a Vector2 with the associated ID for the map that you need to teleport to, the one that will be loaded. To do this, the tiles in the portal layout have an image named with the ID they need to teleport to. When the map is loaded all the positions and IDs are saved into the MapController list. Every time the user moves to a new position it checks if there is a portal in its feed, and if it is, it invokes an event to change the map.

3.6 Canvas, inventory and items

In order to properly explain the inventory and items system, we will explain how the menu is done and what the Canvas is.

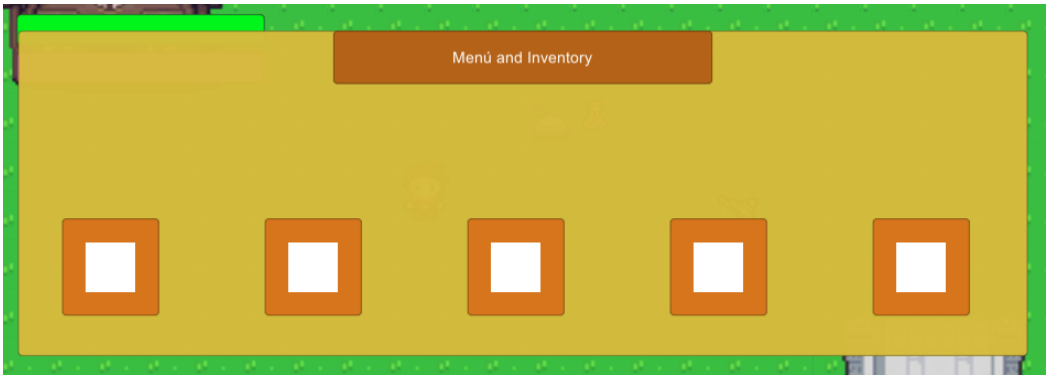

Figure 14: Menu

The Canvas is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas. As we can see in the Figure 15:

Figure 15: Canvas GameObject

Our Canvas has 4 GameObjects inside, each one representing different things. Those elements will be shown in the screen permanently if we don't disable them. The first object is the 'ManaBarHolder'. That is basically a Slider with a minimum and maximum value that represents the remaining

mana of our player. The same thing goes to the 'HpBarHolder'. This Slider represents the remaining health of our player. We also have an object named 'GamOverScreen'. This screen is a simple UI interface with a Game Over message and a button to go to the main menu that is triggered when the player's health decreases to 0.

Figure 16: GameOver Screen

We also have the 'Inventory' GameObject that shows the pause menu and the inventory at the same time. It's hidden until the user presses the 'I' key that triggers an event that makes this object visible and pauses the rest of the game.
This object is composed of 2 childs. The first one is the inventory title and has only visual purpose to show that the user is in the pause menu. In the other hand we have the ItemParents gameObject that contains a GridLayout Group of InventorySlots and has a background colour in order to give some UI to the user. Each of the InventorySlots is a GameObject that contains the script 'InventorySlot.cs' and a child with a button. As default, this button is set to white colour. But when an item is picked it changes to the item image and it's shown in the menu. The script mentioned before is composed of only 2 methods: AddItem and ClearSlot. The AddItem method adds the item object to a property of the script and also puts the image as mentioned before. The ClearSlot removes the information in the given slot but it's not use in any part of the project since it has been implemented just in case it's needed in later stages of the project.

All these items are managed by a bigger script in the Canvas object called 'UIController.cs' which has the following properties:

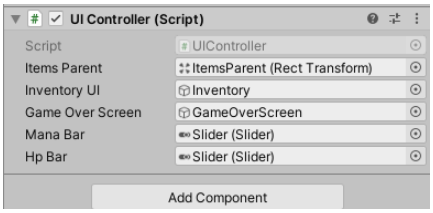

Figure 17: UIController properties

This object is subscribed to all the events that reference the mana and the hp (to increase or decrease the spinner values) so when the player mana/hp values are modified the Controller is notified and the UI shows the real values. It is also the one in charge of putting the game in pause when the 'I' key is pressed and the one that invokes the GameOver Screen when the event is triggered. It also has an event that updates all the GridLayout with the events in the Inventory.
Before keep going with the Inventory explanation, the Item implementation will be shown. In order to be able to make a lot of items with no effort, an ScriptableObject for Items has been created. This Object has the name of the object, the Sprite (art) and finally a boolean to indicate if it is consumable (potions for example) or it is not (a new power or weapon). So when we create a new item to put it in the Scene, like our bow, a new GameObject is created with the script 'PickUpItem.cs' assigned and the ScriptableObject we created passed by the Unity Interface. This script has a method that is invoked via Event each time the player moves to a tile position, it checks if its position is the same as the player and if it is, it adds the Item passed via Unity to the Inventory instance. Once the item is picked, the MapController removes the item from the list and gets notified with an Event.
Finally, we have the Inventory itself. The GameObject called 'InventoryManager' has the logical code for all the inventory in a script called 'Inventory.cs'. This script is made applying the singleton pattern so all the instances that need to access that inventory have the same items storaged and there is only 1 inventory in the whole game.
This script has 4 main methods:

- Add Method: This method gets triggered when the player walks into an item. It checks if the picked up item is consumable or not, and if it is not consumable and there is room to more space in the item list (it has a maximum of 4 values, parameter passed by the UnityInterface) it adds the item to the list and invokes an event to update the UI with the item. If it is consumable, it calls the method ConsumeItem.

- Remove Method: Simple method that removes an item from the list and triggers an event to make sure the UI also changes it up. As explained before, it's a method that is not in use yet, but is prepared for future uses.

- ConsumeItem Method: In case that the item that the user picked has the boolean 'isConsumable' to true, this method will increase the mana/hp depending on the item name and it will invoke an event to make the UI and the player aware of this change.

- ContainsItemName Method: Method created in order to check if the user has a certain item to use a certain attack. If the user has an item called 'Fire Scroll' it will be able to shoot fire.

And with all of this explained changes, we have an inventory to pick up key items with our players and have a control over our UI depending of some values.

3.7 Camera

When a new project is created in Unity, it automatically creates the Camera. The camera simulates a real camera that reproduces whatever of the scene that is in its range. For the project, we have added a new script to the Camera GameObject. This script receives 2 parameters from the Unity interface: the player GameObject and the Smoothing.
The Camera is programmed to follow the player throughout the map. In order to do this there's one parameter to make the camera move in a smooth way and give the user a better feeling when moving.
Finally, the camera has a limit on the position it can go. This is done in order not to show things that are not in the map. When the camera is initialized it takes the loaded and subscribes to the event of map changing to upload those positions in future map.
When the map is loaded it calculates the maximum positions that the camera can move to by taking into account the camera size. When the camera is already showing the limit (X or Y) of a map it stops moving in this direction

even though the player will be able to.
This has been done putting two Vector2 objects representing the max and min values, so the camera movement is limited to those values.

4 Faced problems and solutions

In this section we show a list with the different problems found and, if possible, the solution that has been implemented to solve them. Some of the problems listed don't have solution and some others have already been commented.

- Map instance: The MapController has an instance of the map that everyone can acces if it's needed or creates a local copy of the map and updates it with an event. There was a problem with the objects that needed the map but were created before the MapController had instantiated the loaded map. This caused that GameObjects that needed this instance in order to work crashed and didn't work. In order to make it work, we have used the method 'Awaken' of the MapController, so it charges up the map before the other objects runs the method 'Start'. The problem here is that the events to update the private map instance didn't trigger because they weren't started, so all the objects needed to first take the map from the static instance and then subscribe to the event to update the map every time it is changed.

- Arts problems: When the Sprites where being used there was a problem with the size of the different arts. All the objects that we had in the art file were not 1:1 scale with the tiles of the map. This caused that some objects used 4 tiles and blocked 4 tiles but some of the space was blank, so the player couldn't walk into what it seemed to be an empty space.

- HitBoxes: As explained before, when an object attacked of collided there were problems to reproduce those actions. Since the tools from Unity weren't used, we started to work with exact positions every time, but the objects position is anchored to a pivot, not to a whole object. So some times it was impossible to know that some objects needed to interact. Specially with sword attacks, that need to hit a certain area. This forced us to attack or interact by tiles position instead of areas or exact positions.

- GameObject pivots: In the previous point, the pivots were introduced. At some stage of the project, we realized that due to the arts prob-

lems, when making animations the pivot of the object changed and while walking the enemies can change the pivot from one tile or other depending on the position. Here, a solution was to modify all the arts of the affected objects (enemies and player) to have the same art size and the pivot at the same point, the feet. This modification solved the problem of the pivot.

- Camera Vision: As explained in the camera section, as the camera followed the player, it was needed to implement a system that, taking into account the camera size, calculates the maximum and minimum of its position so it doesn't show tiles outside the map.

- UI scale: When the UI interface was done (hp bar, mana bar, menu..) we do realized that when the screen size changes, some objects disappear or not rescale so the whole canvas was messed up. Since the developer has no experiencie in UI or graphic design, the solution was to fix the screen into a given size in which the UI performed properly.

- Map creation: When maps were created, a lot of failed tests were done. This caused to print tiles that won't be used in the future. This made unity calculate the max and min values of the map incorrectly taking into account the previous printed tiles even though they were not printed. This was due to Unity calculating the size of the gameobject having into account the printed tiles, but not resizing it back when the tile was deleted. Finally, after a lot of research on why this was happening we discovered that there is an option on Unity Interface to shrink the size of the map to the actual printed tiles, so this option was applied to every layer of the maps that has been created.

5 Future work

This is not a finished project. There is still a lot of work to do. In this section the different tasks that are left to do are next enumerated:

- Enemy system improvement: Right now, the enemies are placed in the scene over the map. This has its pros and cons. The good part is that when the enemy dies, it disappears from the scene and it won't be loaded again when you come back to it. The bad point of this is that the enemies are all loaded at the same time. This can produce some overloading in later stages of the project. In order to improve this,

the team should create an EnemyManager (class that is in charge of controlling the enemies) so that all the enemies are created every time the map is loaded, and delete the enemies from the manager when they die if we don't want to create them again.

- Add more enemies: There are only 3 enemies right now in the game. We should add more enemies and fix the movement of 'The Pig' since right now it has an unfair movement because it can move in all directions and is not limited to 4-way-movement as the others existing enemies/player in the game.

- History: Every good game needs a good story behind it. A new UI element should be implemented in which the story behind the game is explained. This also implies to implement a new system of NPCs to interact with you to keep the history.

- Basic UI interaction: For example, when you walk up into a new city, showing the name of this city, or being able to interact with a poster and read what's in the poster. All these things need a new system for the objects like the poster but also a new UI that can be interacted.

- Graphic design improvement: Right now, the arts being used in the project have been taken from the internet and have not been done exclusively for the game. New arts that fulfill our requirements should be created so that the problems with maps or animations are avoided. Also, we need that all the images are power of 2 in order to guarantee an optimal performance. The reason of it is that Unity process with a power of 2 resolution and if don't have it, that will cost more resources.

- Collisions and attack improvements: A lot of time should be invested on making possible the fact of colliding with certain areas or hit those areas without having to use the whole tile position and not overcharging the game with Unity tools.

- Extend the map: Right now only a little test map has been implemented. We should create a bigger map with different thematic or sections as thought at the first stage of the projects.

- Create new mechanics for the player: Right now only the fire attack has been implemented, but every time our player arrives to a new zone/temple it should unlock a new power. By now about 4 different powers have been designed and the code is ready to implement them.

These new powers will unlock taking 3 different scrolls as the first power.

6 Conclusions

First of all I want to point out that the project was not what I wanted or expected at the beginning. Regarding the project management, in the TFG previously done I learned some things that I've used this time. From the beginning I decided not to use a methodology that presses me to do things in a fast peace and pressures me to do it to a certain date, because I knew it would produce a negative effect on my productivity. The project started in a good way and with a good motivation with daily work. But this motivation started to fall off when the first problems caused from the previous decisions started to arise. It was a good idea to use kanban, because I think that if I had pressured myself with deadlines I wouldn't have concluded the project.

I tried to do something more complex than it could be in order to improve the optimization of the project without having worked much with the tool. This caused a lot of stress to me due to not being able to do the things I wanted to do, how I planned, or how I wanted so I had to change the initial plan.

Even though there have been negative parts in the project, there have also been positive ones. I've learnt quite a lot about some tools that Unity gives and help quite a lot like the Event System and the Scriptable objects. In the previous project that I had done with Unity all that we used was simple, but trying to change this was a challenge because I needed to look for an external solution for simple things. This complicated the whole project but at the same time it was more rewarding.

After this project I realized that I've received almost null formation about Unity and game programming, so I will try a new graphic engine named 'Unreal Engine', but I expect to do courses and training in order to compare how it is working by learning everything from scratch alone, and how it is after receiving an appropriate previous formation.

7 Bibliography

Kanban website
ZenhubWebsite
Scriptable Objects
Tilemaps
Canvas
Singleton pattern