Final Report

School Nightmares

02823 Introduction to Computer Game Prototyping E19

17 December 2019

Nicoló Girardini s191681
Michala Lindblad s133992

DTU

Contents

1 Introduction

1.1 Motivation and First Ideas Development

The idea for this project came out naturally, thinking about a lot of games that allow to change worlds between levels or groups of levels, but almost never in a level itself: one that has this mechanic is *Super Mario Bros*[1], with a slight change in the same world (from outside to real underground or from the surface into a lake).

This basic idea got immediately validated by asking our friends. The first elaboration was having a story that allowed to switch between worlds and thus the idea of being a school kid having his friends kidnapped by monsters and finding out about this mysterious **underworld**. On top of this the underworld had to be something strange, very different from the "normal" world in which the kid comes from, so we thought about introducing spells: the kid gains this ability when he travels in the underworld and he is able to keep it even when he comes back to the **school world**. Still there was no reason to switch world again and come back to the school world the second time in the same level during the story: it is at this point that we thought about **items** to collect in the school world that have to be used to progress in the underworld, which is actually the main setting for the game.The school world has also a good characterization, even if the areas are very small: each one represent a school area and the kid has to face his fears in school (teachers, bullies and the librarian). Facing these scholastic fears and the fact that the kid is facing monsters in the underworld gave birth to the game's name: **School Nightmares**.

1.2 Related Games and Genres

The game inspiration came without thinking too much of already existing games. Having an underground world and jumping into a hole may look similar to what happens at the start of the game *Binding of Isaac*[2], still the inspiration came before thinking of it. Also *Super Mario Bros* can be considered an inspiration as there is some world change thanks to the pipes: more in particular Mario goes into a pipe to get down to the underground and then comes back to finish the level, much like in our game the boy goes back to the school world to gather some items to finish the level in the underworld.

As an example the idea of portals leading to other worlds is not new and it is pretty common in fantasy genres, and also using magic, so it can be said that for sure this game is **fantasy**. Still, even with this name it cannot be pushed to horror fantasy, because of the lack of elements that could scare the player. It is for sure a **puzzle** game for the player has to figure out how to pass by all the obstacles. Overall the game is a **puzzle fantasy indie** game, with **isometric** perspective.

The view of the game is inspired by *Transistor*[3], with the isometric perspective, as this will be more interesting to design and challenging to implement compared to the classic 2D perspective. The design of the tiles for the underworld is mainly based on the different caves from the old Pokemon games, such as *Pokemon Red/Blue/Yellow*[4] and *Pokemon Gold/Silver/Crystal*[5], but with a colour scheme close to the underground in *Super Mario Bros*. The character in our game does have some features from the design of the Pokemon Trainer in the Pokemon games, but the player has to move more than four directions, so Unity's own sprite[6] is used as a reference to the design, animation and movement of the player. The idea of the monsters design come from the game *Heart of Darkness*[7], but the monsters is less scary in our game. The game is about a boy who needs to fight these monsters in order to rescuing his dog, which reminds of our story.

2 Game Description

2.1 The Actual Game Story

The story of the game is not complicated and it is told at the start of the game by some storyboards. You impersonate a young kid at school. During the interval the sky gets dark and some strange holes appear in the ground, where monsters pour out of them. These monsters then kidnap your friends and you decide firmly that you have to save them. That's the reason you decide to jump into the holes and face the unknown. During the travel magic appears and you will have to use portals to get back to the school world and progress in the next level in the underworld. You will face different types of monsters in the underworld and all the typical fears a kid can have during school, like teachers, bullies and the librarian, but it's worth if it will save your friends.

2.2 Game Mechanics and Elements

The game starts with the player knowing nothing about the underworld and thus there are some pop-up messages in the first level that allow the player to catch up on what he can really do. The player can run around in the underworld, in "trails", marked by rocks present in it. To reach and save his friends, the player will have to avoid **monsters** in the underworld and even other obstacles.

To pass through the obstacles the player needs either to use **spells** he learns or **items**, gathered by going back to the school world. For example in the first level there is a block of ice that has to be melted with a **fireball** and some high-grown weed that has to be cut with **scissors**, found in a classroom (school world).

To achieve the items, an important mechanic is that the player has to use **portals** in order to move from the underworld the the schoolworld and vice-versa. The portals are unique for each world for each level and have to be found by the player. By stepping on a portal the player will be teleported to the other world.

The other important mechanic is **saving friends**. Each friend is locked into an underworld area, protected either by monsters or obstacles to avoid/destroy. After going through these obstacles, a friend is saved just by the player walking up close to him. The friend will thank the kid and disappear.

The final mechanic is to decide when the level is over and the player can pass to the **following level**. The level can be considered finished only when the player finds the **final gate** (different from the portals) that leads to the following area of the underground, mainly the following level. This gate is activated only when he has rescued all his friends in the current level. The following paragraphs will describe the spells available, the items the player can use and the enemies he will encounter. The implementations will be discussed in Section 3.

2.2.1 Spells

All the spells are learned by gathering **magic scrolls** (which are another kind of collectible item) in the underworld. They can still be used in the school world, but they won't have any effect (apart from Greedy Illusion) on the environment or school enemies and will just disappear at their contact.
Spells are used to fight enemies or to destroy obstacles and they are the following:

- **Fireball.** It's a spell effective against ice, because it can meld it.
- **Greedy illusion**. This spell will stun the "greedy" enemies (namely the bully) for 3 seconds when picking up the illusional coin which is left on the ground by the spell, so the player is able to pass through the enemy without getting caught.
- **Frostbolt.** This spell is found in the third level and it's effective against the monsters. When the spell hits a normal monster, it will become "frozen": it will slow down his patrol and it'll become made of ice, making the fireball effective against him.

2.2.2 Items

The **items** can only be gathered in the school world (apart from the magic scroll). Each level has only one item, that will help the kid progress his search into the underworld. To safeguard the items, there will be school-themed enemies that will either react to the player's actions or act right away. Each of these items has just one purpose in the level and when activated (with the key "Q") they will interact with the environment if they are close to the object that they are supposed to interact with (the grass, the cauldron, and a magical button).
Following this, each item will now be explained in detail.

- **Scissors**. The scissors are present in the first level. They are gathered in the classroom and when the player collect them, they will trigger the teacher to follow the player. The purpose of this item is to cut the weed that is precluding the player to find the final portal of the level.
- **Ladle.** The ladle is in the second level in the canteen. The player needs to collect the items in order to drink from the cauldron in the underworld and pass through it.

- **Dictionaries.** The stack of dictionaries is present in the third level in the library. The player needs to get these books in order to translate the text on the magical button that will open the path for the player to pass over the lava.

2.2.3 Enemies

The enemies pervade the game and protect items, magic scrolls and the prisoners. They can be divided mainly in two categories, described below.

- **Underworld Enemies**. These enemies are just found within the underworld.
    - **a.** **Green Oozes**. These are the easiest monsters in the game. They have fixed spots to move to, after a fixed amount of time.
- **School Enemies**. These enemies are found in the school world sections of the game and they mainly reach to the player's actions.
    - **a.** **Teacher:** The teacher can sometimes be an enemy to a student and in this game it really shows. She is just standing still in the classroom, until the point the player retrieves the scissors. That's when she triggers and starts following the kid, being mad because of him taking and running away with the scissors.
    - **b.** **Bully:** Being bullied or to bully is typical during childhood. This bully is found in the canteen and he runs around like crazy trying to find his next prey.
    - **c.** **Librarian:** The librarian has gone crazy as well because of all the noise, and she will run around her desk, trying to protect the magic dictionaries.

If the player gets hit by one of the enemies, the game will restart from the current level.

2.3 Technical Challenges

The technical challenges have mainly been implementing a working spell system, the portals system and the enemies AI. **The spell system** has been a challenge, because it is a core mechanic of the game and it has many components in it: spell storage, spells that the player has learned and can use, learning a spell and using a spell, and considering their cooldowns.

The portal system presented challenges because of the way Unity handles the various scenes when they load: it was difficult to pass data as well, without using other supporting files, so we decided that the smoothest way was just to have both worlds in the same scene, putting them at a "safe distance", so that one world cannot be seen while being in the other. **The enemies AI** has been a challenge mainly for the enemies that follow the player. The algorithm found needs the tuning of a lot of parameters and thus it required a long process of trial and error. Each of these technical challenges are discussed in detail in section 3.

Another very technical issue at first has been how to reference the various game objects and scripts. There are also a lot of different methods to solve the same problem and thus in our project it happens that problems with common solutions were solved in different ways. Also the choice of having the game with an isometric perspective instead of a classical 2D perspective following the platformer stereotype, was challenging. One of the reasons has been producing the tilemap and use the coordinate system in quite a different way. We decided later that the maps/rooms for the schoolworld had to be just one big image, where collider and school objects were added, instead of using a tilemap to make each room. This was mainly because during each level, the player is visiting a new area in the school. The tiles would be different for each level for the school compared to the underworld, where we used the same tiles. So it was not necessary to make new tiles for the school.

3 Game Implementation

In this section it will be illustrated the implementation of the mechanics that compose the game. The implementation has been done through Unity, using the language C# for the scripts. Because of the huge amount of scripts present in the project, just the main ones are presented.

3.1 World Switching and Portals Logic

The first thing to say is that portals teleport the player to the opposite world thanks to a **trigger box collider**: when the kid hits it he is teleported in the other world.

At first the two worlds (in one level) were thought of as two different scenes that would be connected by the portals, that would've loaded the scene when the kid steps on them. We found some issues with Unity and loading scenes already instantiated and it was not really convenient to pass a lot of data to the scene loader method.

Our workaround is using **two scripts, one for each portal** and to place both worlds in the **same scene**, with a "safe distance" between them, so that the player cannot see one world while he is in the other. What this allows is that the portals can now just **translate the position of the player** (using its **transform**) and we don't have to worry about any data or variable change happening in one world that has to affect the other world as well.

Still how the mechanic works now is that the player can move immediately and there is no actual freeze of the game. Following this, we had to set the new position of the player not exactly on the portal, but a bit further away to avoid the player having issues being teleported back in a loop. In Figure 1 we can see the first level scene for how it is in the editor.

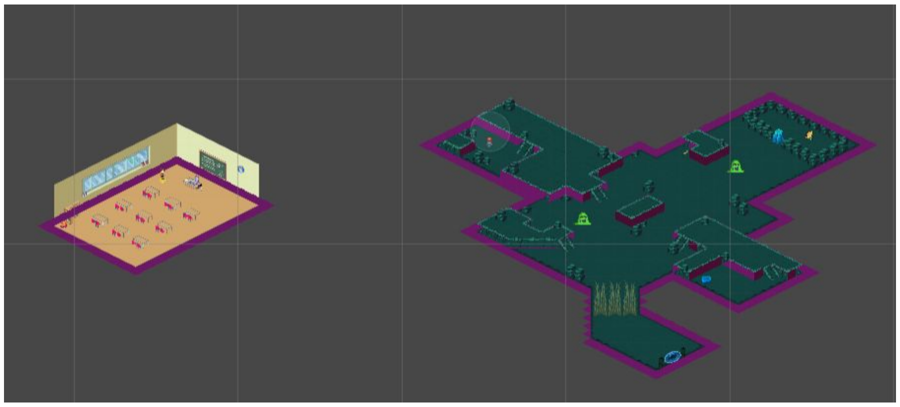


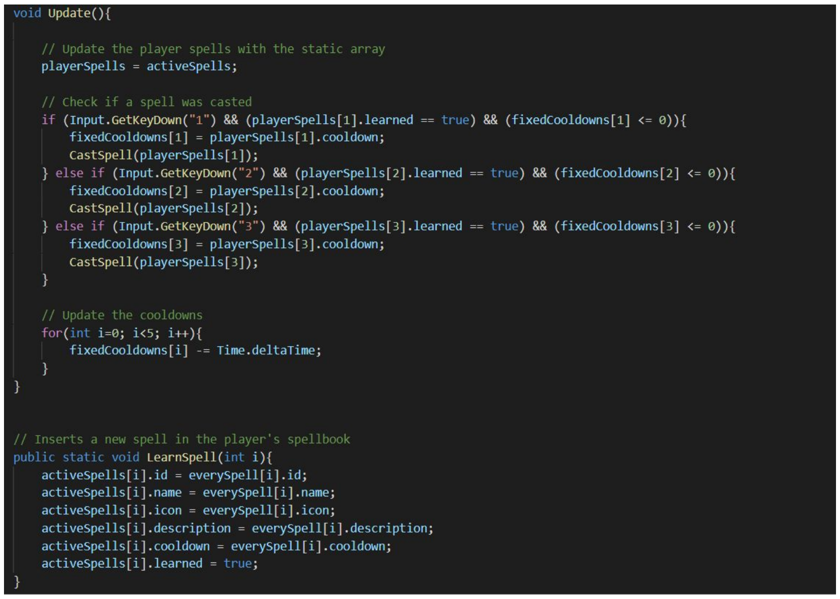
Figure 1

3.2 Spell System

The spell system is the core of the game: it is not complicated but has a lot of elements in it. The first one is the **Spell class**, which defines what a spell is. Figure 3 shows how this class is represented. It just contains the class attributes. Every attribute has a self-explaining name: still it is better to specify that **learned** is a control variable to set to true when the players learns the spell. These are set in the **SpellController** object, through the inspector. The latter has an attached script that manages the spell system and contains the arrays of spells needed:

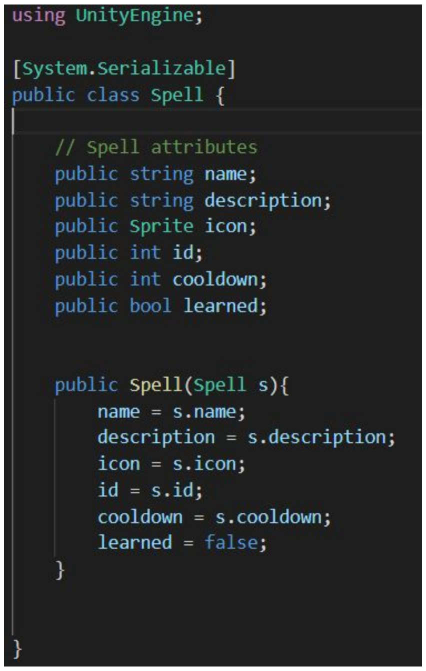
- **allSpells**: This is the array that contains all the spells the player can learn.
- **playerSpells**: These are the spells that the player actually knows.
- **activeSpells and everySpell**: These are static arrays needed to learn and update *playerSpells*.

To better understand the use of the last two arrays there's the need of explaining how learning a spell works. The solution could've been referencing the items in a different way, but having static arrays seemed less computationally heavy.



```
void Update(){

    // Update the player spells with the static array
    playerSpells = activeSpells;

    // Check if a spell was casted
    if (Input.GetKeyDown("1") && (playerSpells[1].learned == true) && (fixedCooldowns[1] <= 0)){
        fixedCooldowns[1] = playerSpells[1].cooldown;
        CastSpell(playerSpells[1]);
    } else if (Input.GetKeyDown("2") && (playerSpells[2].learned == true) && (fixedCooldowns[2] <= 0)){
        fixedCooldowns[2] = playerSpells[2].cooldown;
        CastSpell(playerSpells[2]);
    } else if (Input.GetKeyDown("3") && (playerSpells[3].learned == true) && (fixedCooldowns[3] <= 0)){
        fixedCooldowns[3] = playerSpells[3].cooldown;
        CastSpell(playerSpells[3]);
    }

    // Update the cooldowns
    for(int i=0; i<5; i++){
        fixedCooldowns[i] -= Time.deltaTime;
    }
}


// Inserts a new spell in the player's spellbook
public static void LearnSpell(int i){
    activeSpells[i].id = everySpell[i].id;
    activeSpells[i].name = everySpell[i].name;
    activeSpells[i].icon = everySpell[i].icon;
    activeSpells[i].description = everySpell[i].description;
    activeSpells[i].cooldown = everySpell[i].cooldown;
    activeSpells[i].learned = true;
}
```

Figure 2



```
using UnityEngine;

[System.Serializable]
public class Spell {

    // Spell attributes
    public string name;
    public string description;
    public Sprite icon;
    public int id;
    public int cooldown;
    public bool learned;


    public Spell(Spell s){
        name = s.name;
        description = s.description;
        icon = s.icon;
        id = s.id;
        cooldown = s.cooldown;
        learned = false;
    }


}
```

Figure 3

The player learns a spell when he gathers a **scroll**. In the script controller the function ***SpellController.LearnSpell(int i)*** will be called: it can be seen in Figure 2. This has an integer as a parameter, which is the spell id. At this point all the information of that spell is present in the array *everySpell*, position i: such information will be copied in the other static array, *activeSpells*. At this point to transfer the information to the *playerSpells* array, which is the one that is used to cast spells. In the ***Update()*** function of the spell controller, the *activeSpells* array is copied into *playerSpells*.

As seen in Figure 2, the *Update()* function is very important in the spell logic. Not only it updates the spells available to the player, but it also checks if he is trying to cast a spell and if the spell can be cast. When the player presses a key that corresponds to one of the spells, the first part of the check is if the player has actually learned the spell. The second part involves instead a new array, ***fixedCooldowns***. It is a float array and it contains the time left before the player can cast that spell (the indexing is compliant with the spells arrays): if the time is 0 or negative the spell can be cast, and if it is positive the player has to wait. To decrease the cooldown time, a small for cycle is used to lower all the spells cooldowns over time. When the player knows the spell and that spell is not on cooldown, the ***CastSpell(Spell s)*** method is called and the *fixedCooldowns* array is fixed to that spell's cooldown at the correct index. The method is actually just a switch-case construct: it checks the id of the spell passed and then calls the function proper to the called spell.

The actual spells objects are implemented using **prefabs**. Each time the *CastSpell* method calls a function, this function instantiates the prefab corresponding to that spell. Since the prefab instantiation allows to set the initial position, a **firePoint** object is placed on the boy figure: the reference to this firePoint transform is stored as a class variable of *SpellController* and it's used as the instantiated object's position. Each prefab has a script attached to it and it controls the behaviours of the new game object. The spells are described in the following subsections.

3.2.1 Fireball

The script attached to the fireball has to control how the fireball moves and its effect on the environment. It is called **FireballBulletControl**. The ***Start()*** function checks the direction the boy is currently facing, using a static string variable of the **BoyAnimation** class: this one gives the direction (N, NE, E, S, etc.) that the spell has to follow, while the speed is just a class variable. The fireball prefab has a **trigger collider**: when any object enters the collider, its tag is checked. If the tag is "Ice" then the object is destroyed, because the fireball in this game melds the ice. The fireball bullet object is destroyed as well.

3.2.2 Greedy Illusion

The script needed for this spell is attached to its prefab (**GreedyIllusionCoin**) and it is called **GreedyIllusionCoinController.** It leaves a coin on the ground, in the position the **firePoint** is. When this coin collides with the **bully**, it will call a function implemented in the **BullyController**, called ***Stunned()*** that will keep stunned, thus still, the bully for 3 seconds.

3.2.3 Frostbolt

The frostbolt has the same prefab instantiation as the fireball: the script **FrostboltBulletControl** will take the direction the frostbolt has to travel to from the **BoyAnimation**. When it hits something that is not a green monster nothing happens. If it hits, however, the monster will turn blue and it will slow its speed, like frozen. It will change "type" as well, becoming an ice enemy, so that it is now vulnerable to be killed with a fireball.

3.3 Enemies AI

There are basically two different enemies AIs. The common part is that, when the player collides with them (as usual we used a **collider** set to **trigger**), he has to restart the level. This is done simply by loading again the scene, because it is intended that he loses his progress. Still, the choice affects the performance and the loading times are not optimal, also because of one of the algorithms used for the enemies.

Most enemies, the various **monsters**, the **bully** and the **librarian,** have a **random patrolling logic**. This means that they have **waypoints** to where they can move: these points are just empty objects, placed by hand around the level (these are the green diamonds in Figure 4) in fixed positions. Each monster cannot reach any of these waypoints, but just a fixed subset. The way this is done is thanks to the ***Update()*** function, depicted in Figure 5.

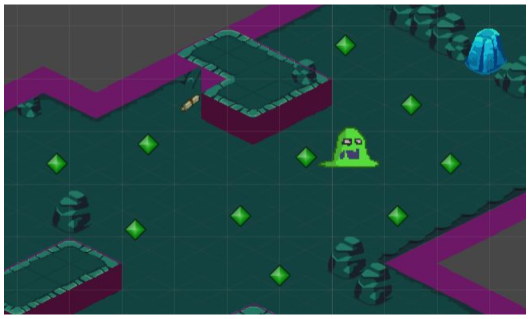
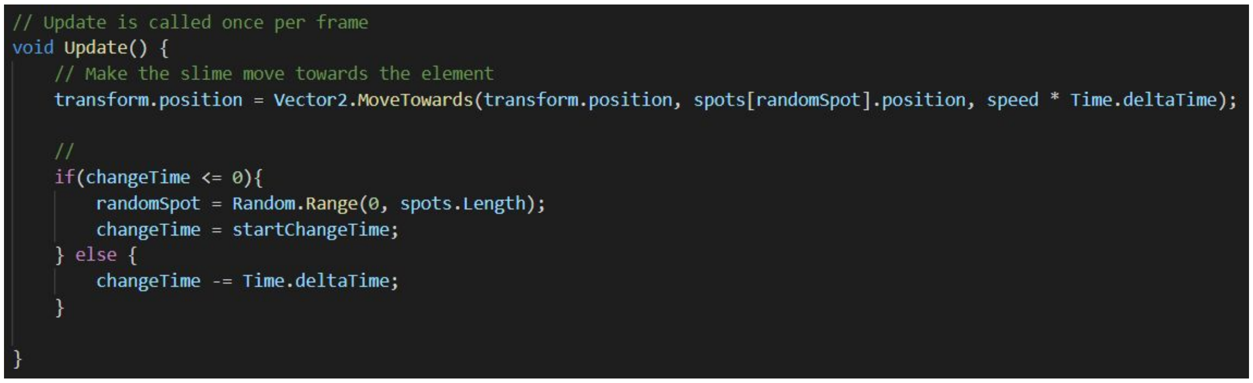


Figure 4

The subset of waypoints is stored in the **spots** variable. To select one random point between these, there is an integer variable, ***randomSpot***, set to a random value valid as an index for the spots array. The monster move in that spot's direction, thanks to the ***MoveTowards()*** function. To change direction there is a timer: **changeTime** is the variable counting down the time that has to pass and **startChangeTime** is the fixed amount of seconds that have to pass before the monster can change direction. In this way the monster can be still for a couple of seconds if the time it takes to move from the previous position to the new one is very short.



```
// Update is called once per frame
void Update() {
    // Make the slime move towards the element
    transform.position = Vector2.MoveTowards(transform.position, spots[randomSpot].position, speed * Time.deltaTime);

    //
    if(changeTime <= 0){
        randomSpot = Random.Range(0, spots.Length);
        changeTime = startChangeTime;
    } else {
        changeTime -= Time.deltaTime;
    }

}
```

Figure 5

It may happen that in the path of the monster there is an obstacle: this case is handled by adding another collider, not trigger this time, that allows to handle this border case with the function ***OnCollisionEnter2D()***. The latter checks whether the monster collides with an object that it's not the player, it will select another waypoint to move towards and reset the time.

For what concerns the "more clever" enemies, that follow the player, an external tool has been adopted: **Astar**[8]. This is a pathfinding project that works with Unity2D. It works by generating a **graph** of the selected area, given the node size. It will create a **grid** on the selected area, with each cell having a size equal to the given node size. With this grid it will operate on a **mask**, set by using the **layer** property of Unity's game objects. This mask will allow the algorithm to identify the correct obstacles that the selected enemy will have to avoid and these points will therefore be excluded from the path computation.
The object that has to move is interpreted as the small yellow circle that is seen in Figure 6, which depicts the grid of the classroom in the first level. The blue area represents the nodes the enemy can move to, while the other areas (colored) are the obstacles it has to avoid.

The last element needed for the pathfinding is a **destination**. This destination can be any game object's transform and it has obviously been set to the **boy**'s transform. With a destination, the algorithm will compute the shortest path to the destination, avoiding the objects. All this has been set up into the **Astar** game object, thanks to the **Astar components**. Less important parameters are present in the enemy game object as Astar components, such as the frequency with which the path has to be computed again. The optimal setting of the parameters to have a good behaviour for the enemies has not been easy and it has been a trial-and-error process. Because of the small size of the assets we use, the graph's node size has been set to a very small value: this slows a lot the computation of the graph and the obstacles (it takes around 12 seconds) and that is the main reason behind the slow loading times in the first level.

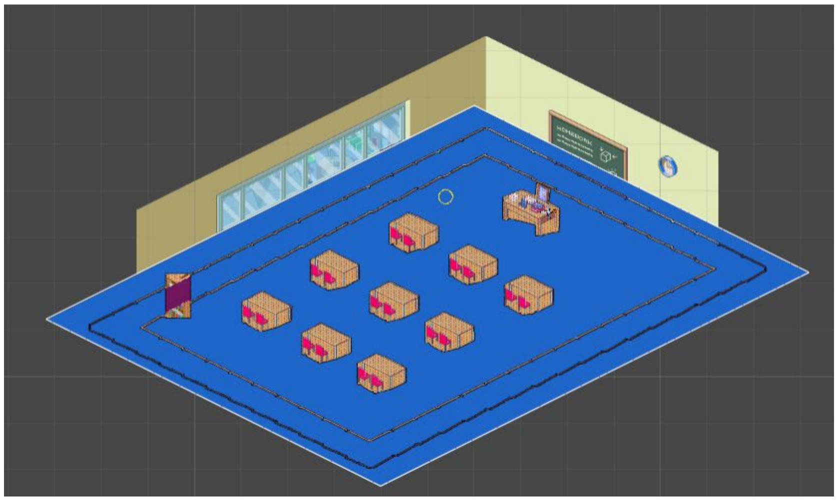


Figure 6

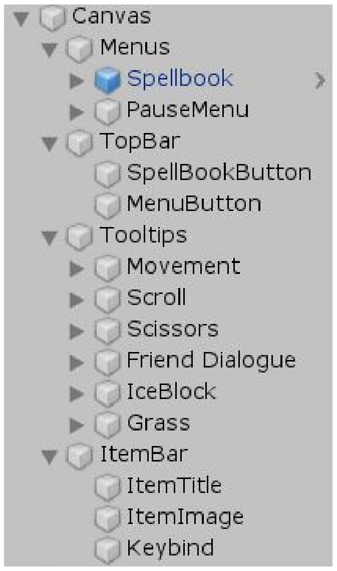
3.4 UI Implementation

We tried to keep the UI of the game as minimal as possible, to let the player have a free view on the space he is moving into. Apart from the elements shown in Figure 7, there is a **UIManager**. It manages some logic of the more dynamic elements of the UI, which are the spellbook and the menu. It keeps track whether these two elements are visible or not to the player. Each button that shows/hides one of these two elements has then to interact with the UIManager. It also checks if the player is pressing one of the buttons which is given a specific function regarding the UI, namely "P" for the spellbook and "escape" for the menu (it closes the spellbook as well).

In Figure 7, the **hierarchy** of the UI elements can be seen and is explained below:

- **Menus**: This contains the two menus there are in the game. One is the **Spellbook** (see following section), while the other is the **Pause Menu** of the game.
- **TopBar**: This contains the two buttons at the top of the screen: **SpellBookButton** allows to show/hide the spellbook, while the **MenuButton** shows/hides the pause menu.
- **Tooltips**: This object contains all the tooltips pop-ups that are present in the first level. It will contain as well the dialogues with the characters, as it is the case for **Friend Dialogue**.
- **ItemBar**: As the name suggests, this is the section of the UI that shows the player if he has an item and which item it is. The **ItemTitle** is just the header of the section, **Keybind** is an image of the key "Q" of a keyboard, to ease the player when he has to remember how to use the item. **ItemImage** is the actual image of the item: since there is only one item per level, this is fixed for every level and activated when the player finds the item itself.

In Figure 8, we can instead see the perspective of the player while he is in the Pause Menu. The menu is in the center, while the other elements are highlighted and labelled.



```
Canvas
  Menus
    Spellbook
    PauseMenu
  TopBar
    SpellBookButton
    MenuButton
  Tooltips
    Movement
    Scroll
    Scissors
    Friend Dialogue
    IceBlock
    Grass
  ItemBar
    ItemTitle
    ItemImage
    Keybind
```

Figure 7

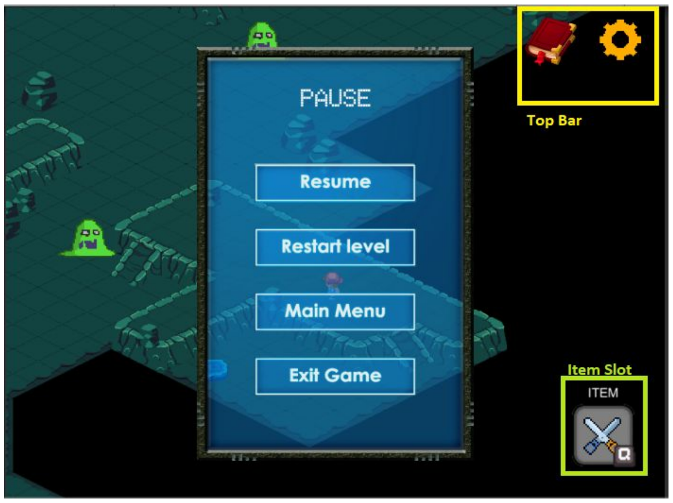


Figure 8

3.4.1 SpellBook

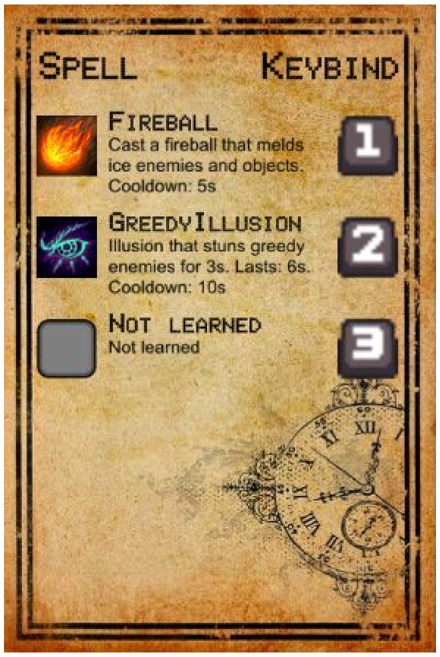

Spell
Keybind
Fireball
Cast a fireball that melds ice enemies and objects.
Cooldown: 5s
1
GreedyIllusion
Illusion that stuns greedy enemies for 3s. Lasts: 6s.
Cooldown: 10s
2
Not Learned
Not learned
3

Figure 9

The spellbook is the most unique element in the UI, because of its evolving structure. It is managed by the script **BookController**. It just contains the spells that the player has learned so far. This is possible using the ***OnEnable()*** function. At first the spellbook game object is not active. When the key "P" or the top bar button is pressed, the engine will call the *OnEnable()* function. This allowed to use a reference to the static array present in the **SpellController** and check what spells were learned and which were unknown to the player. Then the spell attributes (**name**, **description**, **icon**) are substituted into the placeholders already present. This actually happens even if the spell is not known yet: the text will be *"Not Learned"* and the icon will be a default placeholder. In Figure 9 we can see an example of the SpellBook while just the fireball is known.

4 Analysis

Three iteration has been done. The group had tested some mechanics and improve these by implemented them while developing the game and before having the first iteration. It was to make sure that the game was playable to the test-user. Each implementation was intended to implement new features and improve existing features. In order to get feedback, fellow classmates and friends was testing the game.

4.1 First Iteration

In the first iteration, the following is implemented before testing the game for the user:

- The first level - the underworld and the schoolworld (classroom)
- The playable boy - movement and animation
- The book spell and the fireball spell
- Portals between the worlds
- Collectable items: spell scrolls and scissor
- Enemies: the teacher at the school - chasing
- Ice block blocking the way to get to the friend
- Tall grass - need to cut

The first test of the game was in the class, where everyone could go around and try each others games. We were able to get three participants to test our game and giving feedback while playing the game. All test-user did not notice when they picked up the magic scroll and did not understand what the magic scroll was for. They had some troubles finding out that you could open the spellbook, because there was no text telling them what key to press.

One test-user suggested to either light up or do other things to indicate that you can pick up the different items (the scissor in the schoolworld and the spell scroll in the underworld) and also showing that the spellbook has updated due to the new spell scroll. Another test-user mentioned that the direction of the fireball was not good, as you could only shoot in one direction even if you are facing or walking in another direction.

Because the test-users did not notice when the player learn a new spell, some suggested that the different spells learned could be shown at the bottom of the screen instead of opening the book everytime you need to check your spells.

4.2 Second Iteration

For the second iteration, some changes have been made due to the feedback from the first iteration. The following features have been implemented:

- The start screen
- The storyboard - telling the story with images and text before getting into the game.
- The dialogues in the game to explain what to do through the levels
- Fireball shooting in all directions
- More enemies: Monsters in the underworld - avoid them.
- Ingame menu
- Visible item holder at the bottom corner of the game - when you pick up the scissor
- Arrow pointing at collectable items
- Game freezes: happens when the dialogue pops up after collecting the scissors and will unfreeze the game when exit (to make the player aware of the teachers behavior)
- Added audio - background music

After implemented all the new features, the game was tested by four of our friends, who are daily playing video games. The overall feedback was very positive and they all enjoy the game. There was some feedback regarding improvement for the game.

The volume of the background music was very low even if they had turned up the volume on max. A friend suggested that there should be some info or text showing, when you get hit by the monster or the teacher. As it was now, they thought the game freeze when they died. The same friend would like to use a key to close down the dialogue instead of using the mouse.

Another friend was asking, if it was possible to show the cooldown in someway for the fireball, so the player knows when the spell is ready to use again.

4.3 Final Iteration

For the final iteration, the following features have been implemented regarding the feedback from the second iteration and further development of the game:

- When hitting an enemy, a dialogue is shown and when closing down, the game restarts.
- Volume of the music increased - and looping when the music end.
- The end screen - when the player exits the last level and win.
- Level 2 has been added
    - Schoolworld is now the canteen
    - New enemy: the bully - running fast
    - New collectable items: ladle
    - New spellscroll: Greedy Illusion - stuns the enemy for 3 seconds
    - New friend to save from the underworld
    - The cauldron has been added to the underworld as an obstacle
- Level 3 has been added
    - Schoolworld is now the library
    - New enemy: the librarian - chasing
    - New collectable items: books of dictionaries
    - New spellscroll: Frostbolt - freeze the enemy for 10 seconds
    - A magic button with unknown language text has been added - to make a path through the lava
    - The last friend added to save from the underworld
    - Lava floor - block the way for the player - if the player touch the lava, the game restarts

Not all requested improvements from the second iteration feedback was done due to limited time and will be considered in future development of the game.

5 Conclusion

Our initial idea was to create an isometric 2D game, where the player has to travel through different worlds in order to save his friends from the monsters. The initial focus was on the controls of switching world, the player, enemies and the spell system.

The first idea was to make a scene for each world the player are traveling to, but after facing many issues with that, the decision was made to make both worlds in the same level to make it easier for collecting items through the worlds. While testing the game, we realise that people didn't understand the story of the game when they just started in the underworld. Storyboards was then included to tell the story about the boy that ended up in this strange world.

In the end, with all the iteration and the feedback, we managed to make a full playable game from start to end with three levels as we had imagined from the beginning.

With the posterior knowledge we have now, there are things that we would’ve done differently. One of them was drawing the underworld on paper first to figure out where to place stuff. This could have saved some time, as it would have been easier to decide the placement of the different objects before mapping it out in Unity. Another thing to be done differently was better exploit and more consistency of the Unity referencing system, to have an even better structured hierarchy and more optimization. For the last thing was to make larger use of the prefabs functionality.

Going forward, we are planning to implement more features to the game. One of them is adding visible cooldown for the spells, as one test-user suggested, because it can be hard to figure out when the cooldown is down to zero again without having any visible countdown on the screen. Another future implemented feature would be adding shortcuts for closing down the dialogue. A test-user find it a bit annoying to use the mouse to close down the menu all the time, where if they could just press a key to close it down, that will help moving forward with the game faster.

The group had already ideas for different abilities for the school enemies, but because the time was running out, the different abilities were not implemented. One for example was that the librarian will throw books at the player, as soon as the player collected the dictionaries. The same goes to the bully in the canteen, where he should throwing food at the player, when he got the ladle. The problems with these implementations are mainly due to the referencing and activation of game objects that are not active (for example to have a box saying that the kid was hit) and to the fact that they always have to follow the player’s position to be effective. Even if facing and throwing objects in the direction of it can be done, given the proper time, this would not be difficult, because the player could easily avoid the projectiles. Some kind of prediction would be the way in this case.

Another future implementation that the group had in mind was to add sound effects, when the player is shooting the spells, collection items and hitting obstacles.

The Astar pathfinding can be implemented in a more custom way using its libraries in the scripts. This could allow an easier tuning of the parameters and a more custom use of Astar elements, which should lead to a better result for the pathfinding for the teacher.

6 Main Contributions

Being only two, we worked on everything, but the main contributions are distributed in this way:

- **Michala Lindblad s133992**: art style(making the start screen, storyboards and other graphics to the game), animations, level and tilemaps design, audio, world switching logic.

- **Nicolò Alessandro Girardini s191681**: spell system, enemies AI, items scripts, UI.

References

[1] Super Mario Bros - https://www.imdb.com/title/tt0177266/plotsummary

[2] The Binding of Isaac - https://store.steampowered.com/app/113200/The_Binding_of_Isaac/

[3] Transistor - https://store.steampowered.com/app/237930/Transistor/

[4] Pokemon Red/Blue/Yellow -
https://www.ign.com/wikis/pokemon-red-blue-yellow-version/What_is_Pokemon

[5] Pokemon Gold/Silver/Crystal -
https://www.ign.com/wikis/pokemon-gold-silver-crystal-version/Game_Basics

[6] Isometric 2D Environments with Tilemap -
https://blogs.unity3d.com/2019/03/18/isometric-2d-environments-with-tilemap/

[7] Heart of Darkness - https://www.ign.com/articles/1998/09/09/heart-of-darkness-2

[8]Astar Pathfinding - https://arongranberg.com/astar/